The GNU C Library:
System & Network Applications

For GNU C Libraries version 2.3.x

by Sandra Loosemore
with Richard M. Stallman, Roland McGrath,
Andrew Oram, and Ulrich Drepper

This manual documents the GNU C Libraries version 2.3.x.
ISBN 1-882114-24-8, First Printing, March 2004.

Published by:

GNU Press
a division of the
Free Software Foundation
51 Franklin St, Fifth Floor
Boston, MA 02110-1301 USA

Website: www.gnupress.org
General: press@gnu.org
Orders: sales@gnu.org
Tel: 617-542-5942
Fax: 617-542-2652

Cover art by Etienne Suvasa. Cover design by Jonathan Richard. Printed in USA.

Short Contents

Table of Contents

The GNU C Library: System and Network Applications

1 Introduction

The C language provides no built-in facilities for performing such common operations as input/output, memory management, string manipulation and the like. Instead, these facilities are defined in a standard *library*, which you compile and link with your programs.

The GNU C Library, described in this document, defines all of the library functions that are specified by the ISO C standard, as well as additional features specific to POSIX and other derivatives of the Unix operating system, and extensions specific to the GNU system.

The purpose of this manual is to tell you how to use the facilities of the GNU library. We have mentioned which features belong to which standards to help you identify things that are potentially nonportable. But the emphasis in this manual is not on strict portability.

1.1 Getting Started

This manual is written with the assumption that you are at least somewhat familiar with the C programming language and basic programming concepts. Specifically, familiarity with ISO standard C (see Section 1.2.1 [ISO C], page 2), rather than "traditional" pre-ISO C dialects, is assumed.

The GNU C Library includes several *header files*, each of which provides definitions and declarations for a group of related facilities; this information is used by the C compiler when processing your program. For example, the header file '`stdio.h`' declares facilities for performing input and output, and the header file '`string.h`' declares string-processing utilities. The organization of this manual generally follows the same division as the header files.

If you are reading this manual for the first time, you should read all of the introductory material and skim the remaining chapters. There are a *lot* of functions in the GNU C Library and it is not realistic to expect that you will be able to remember exactly *how* to use each and every one of them. It is more important to become generally familiar with the kinds of facilities that the library provides, so that when you are writing your programs you can recognize *when* to make use of library functions, and *where* in this manual you can find more specific information about them.

1.2 Standards and Portability

This section discusses the various standards and other sources that the GNU C Library is based upon. These sources include the ISO C and POSIX standards, and the System V and Berkeley Unix implementations.

The primary focus of this manual is to tell you how to make effective use of the GNU library facilities. But if you are concerned about making your programs compatible with these standards, or portable to operating systems other than GNU,

this can affect how you use the library. This section gives you an overview of these standards, so that you will know what they are when they are mentioned in other parts of the manual.

See Appendix B [Summary of Library Facilities], page 475, for an alphabetical list of the functions and other symbols provided by the library. This list also states which standards each function or symbol comes from.

1.2.1 ISO C

The GNU C Library is compatible with the C standard adopted by the American National Standards Institute (ANSI)as *American National Standard X3.159-1989—"ANSI C"* and later by the International Standardization Organization (ISO) as ISO/IEC *9899:1990, "Programming languages—C"*. In this manual, we refer to the standard as ISO C since this is the more general standard with respect to ratification. The header files and library facilities that make up the GNU library are a superset of those specified by the ISO C standard.

If you are concerned about strict adherence to the ISO C standard, you should use the '`-ansi`' option when you compile your programs with the GNU C Compiler. This tells the compiler to define *only* ISO standard features from the library header files, unless you explicitly ask for additional features. See Section 1.3.4 [Feature-Test Macros], page 8, for information on how to do this.

Being able to restrict the library to include only ISO C features is important because ISO C puts limitations on what names can be defined by the library implementation, and the GNU extensions don't fit these limitations. See Section 1.3.3 [Reserved Names], page 6, for more information about these restrictions.

This manual does not attempt to give you complete details on the differences between ISO C and older dialects. It gives advice on how to write programs to work portably under multiple C dialects, but does not aim for completeness.

1.2.2 POSIX (The Portable Operating System Interface)

The GNU library is also compatible with the ISO POSIX family of standards, known more formally as the *Portable Operating System Interface for Computer Environments* (ISO/IEC 9945). They were also published as ANSI/IEEE Std 1003. POSIX is derived mostly from various versions of the Unix operating system.

The library facilities specified by the POSIX standards are a superset of those required by ISO C; POSIX specifies additional features for ISO C functions, as well as specifying new additional functions. In general, the additional requirements and functionality defined by the POSIX standards are aimed at providing lower-level support for a particular kind of operating system environment, rather than general programming language support that can run in many diverse operating system environments.

The GNU C Library implements all of the functions specified in ISO/IEC *9945-1:1996, the* POSIX *System Application Program Interface*, commonly referred to as POSIX.1. The primary extensions to the ISO C facilities specified by this stan-

dard include file-system interface primitives (see Chapter 3 [File-System Interface], page 71), device-specific terminal control functions (see Chapter 6 [Low-Level Terminal Interface], page 179) and process control functions (see Chapter 7 [Processes], page 209).

Some facilities from ISO/IEC *9945-2:1993, the* POSIX *Shell and Utilities standard* (POSIX.2) are also implemented in the GNU library. These include utilities for dealing with regular expressions and other pattern-matching facilities.[1]

1.2.3 Berkeley Unix

The GNU C Library defines facilities from some versions of Unix that are not formally standardized, specifically from the 4.2 BSD, 4.3 BSD and 4.4 BSD Unix systems (also known as *Berkeley Unix*) and from SunOS (a popular 4.2 BSD derivative that includes some Unix System V functionality). These systems support most of the ISO C and POSIX facilities, and 4.4 BSD and newer releases of SunOS in fact support them all.

The BSD facilities include symbolic links (see Section 3.5 [Symbolic Links], page 87), the `select` function (see Section 2.8 [Waiting for Input or Output], page 37), the BSD signal functions (see Section 17.10 [BSD Signal-Handling], page 426) and sockets (see Chapter 5 [Sockets], page 125).

1.2.4 SVID (The System V Interface Description)

The *System V Interface Description* (SVID) is a document describing the AT&T Unix System V operating system. It is to some extent a superset of the POSIX standard.

The GNU C Library defines most of the facilities required by the SVID that are not also required by the ISO C or POSIX standards, for compatibility with System V Unix and other Unix systems (such as SunOS) that include these facilities. However, many of the more obscure and less generally useful facilities required by the SVID are not included. (In fact, Unix System V itself does not provide them all.)

The supported facilities from System V include the methods for inter-process communication and shared memory, the `hsearch` and `drand48` families of functions, `fmtmsg` and several of the mathematical functions.

1.2.5 XPG (The X/Open Portability Guide)

The *X/Open Portability Guide*[2] is a more general standard than POSIX. X/Open owns the Unix copyright and the XPG specifies the requirements for systems that are intended to be Unix systems.

---

[1] See Sandra Loosemore et al., "Pattern-Matching" in *GNU C Library: Application Fundamentals* (Boston: GNU Press, 2004), available online at `http://www.gnu.org/manual/manual.html`.

[2] X/Open Company, *X/Open Portability Guide,* Issue 4 (Reading, UK: X/Open Company, Ltd., 1992).

The GNU C Library complies with the *X/Open Portability Guide*, Issue 4.2, with all extensions common to XSI (X/Open System Interface) compliant systems and also all X/Open Unix extensions.

The additions on top of POSIX are mainly derived from functionality available in System V and BSD systems, though some of the really bad mistakes in System V systems were corrected. Since fulfilling the XPG standard with the Unix extensions is a precondition for getting the Unix brand, chances are good that the functionality is available on commercial systems.

1.3 Using the Library

This section describes some of the practical issues involved in using the GNU C Library.

1.3.1 Header Files

Libraries for use by C programs really consist of two parts: *header files* that define types and macros and declare variables and functions, and the actual library or *archive* that contains the definitions of the variables and functions.

(Recall that in C, a *declaration* merely provides information that a function or variable exists and gives its type. For a function declaration, information about the types of its arguments might be provided as well. The purpose of declarations is to allow the compiler to correctly process references to the declared variables and functions. A *definition*, on the other hand, actually allocates storage for a variable or says what a function does.)

In order to use the facilities in the GNU C Library, you should be sure that your program source files include the appropriate header files. This is so that the compiler has declarations of these facilities available and can correctly process references to them. Once your program has been compiled, the linker resolves these references to the actual definitions provided in the archive file.

Header files are included into a program source file by the '`#include`' preprocessor directive. The C language supports two forms of this directive; the first,

```
#include "header"
```

is typically used to include a header file *header* that you write yourself; this would contain definitions and declarations describing the interfaces between the different parts of your particular application. By contrast,

```
#include <file.h>
```

is typically used to include a header file '`file.h`' that contains definitions and declarations for a standard library. This file would normally be installed in a standard place by your system administrator. You should use this second form for the C library header files.

Typically, '`#include`' directives are placed at the top of the C source file, before any other code.[3] If you begin your source files with some comments explaining what the code in the file does (a good idea), put the '`#include`' directives immediately afterward, following the feature-test macro definition (see Section 1.3.4 [Feature-Test Macros], page 8).

The GNU C Library provides several header files, each of which contains the type and macro definitions and variable and function declarations for a group of related facilities. This means that your programs may need to include several header files, depending on exactly which facilities you are using.

Some library header files include other library header files automatically. However, as a matter of programming style, you should not rely on this; it is better to explicitly include all the header files required for the library facilities you are using. The GNU C Library header files have been written in such a way that it doesn't matter if a header file is accidentally included more than once; including a header file a second time has no effect. Likewise, if your program needs to include multiple header files, the order in which they are included doesn't matter.

Compatibility Note: Inclusion of standard header files in any order and any number of times works in any ISO C implementation. However, this has traditionally not been the case in many older C implementations.

Strictly speaking, you don't *have to* include a header file to use a function it declares; you could declare the function explicitly yourself, according to the specifications in this manual. But it is usually better to include the header file because it may define types and macros that are not otherwise available and because it may define more efficient macro replacements for some functions. It is also a sure way to have the correct declaration.

1.3.2 Macro Definitions of Functions

If we describe something as a function in this manual, it may have a macro definition as well. This normally has no effect on how your program runs—the macro definition does the same thing as the function would. In particular, macro equivalents for library functions evaluate arguments exactly once, in the same way that a function call would. The main reason for these macro definitions is that sometimes they can produce an in-line expansion that is considerably faster than an actual function call.

Taking the address of a library function works even if it is also defined as a macro. This is because, in this context, the name of the function isn't followed by the left parenthesis that is syntactically necessary to recognize a macro call.

You might occasionally want to avoid using the macro definition of a function—perhaps to make your program easier to debug. There are two ways you can do this:

---

[3] For more information about the use of header files and '`#include`' directives, see Richard M. Stallman and the GCC Developer Community, "Header Files" in *The GNU C Preprocesser Manual* (2003), `http://gcc.gnu.org/onlinedocs/gcc-3.3.2/cpp/`.

1. You can avoid a macro definition in a specific use by enclosing the name of the function in parentheses. This works because the name of the function does not appear in a syntactic context where it is recognizable as a macro call.
2. You can suppress any macro definition for a whole source file by using the '`#undef`' preprocessor directive, unless otherwise stated explicitly in the description of that facility.

For example, suppose the header file '`stdlib.h`' declares a function named `abs` with:

```
extern int abs (int);
```

and also provides a macro definition for `abs`. Then, in:

```
#include <stdlib.h>
int f (int *i) { return abs (++*i); }
```

the reference to `abs` might refer to either a macro or a function. On the other hand, in each of the following examples, the reference is to a function and not a macro:

```
#include <stdlib.h>
int g (int *i) { return (abs) (++*i); }

#undef abs
int h (int *i) { return abs (++*i); }
```

Since macro definitions that double for a function behave in exactly the same way as the actual function version, there is usually no need for any of these methods. In fact, removing macro definitions usually just makes your program slower.

1.3.3 Reserved Names

The names of all library types, macros, variables and functions that come from the ISO C standard are reserved unconditionally; your program *may not* redefine these names. All other library names are reserved if your program explicitly includes the header file that defines or declares them. There are several reasons for these restrictions:

- Other people reading your code could get very confused if, for example, you were using a function named `exit` to do something completely different from what the standard `exit` function does. Preventing this situation helps to make your programs easier to understand and contributes to modularity and maintainability.
- It avoids the possibility of a user accidentally redefining a library function that is called by other library functions. If redefinition were allowed, those other functions would not work properly.
- It allows the compiler to do whatever special optimizations it pleases on calls to these functions, without the possibility that they may have been redefined by the user. Some library facilities, such as those for dealing with variadic arguments (see Section A.2 [Variadic Functions], page 456) and nonlocal exits (see Chapter 16 [Nonlocal Exits], page 367), actually require a considerable

amount of cooperation on the part of the C compiler, and with respect to the implementation, it might be easier for the compiler to treat these as built-in parts of the language.

In addition to the names documented in this manual, reserved names include all external identifiers (global functions and variables) that begin with an underscore ('_') and all identifiers regardless of use that begin with either two underscores or an underscore followed by a capital letter. This is so that the library and header files can define functions, variables, and macros for internal purposes without risk of conflict with names in user programs.

Some additional classes of identifier names are reserved for future extensions to the C language or the POSIX.1 environment. While using these names for your own purposes right now might not cause a problem, there is the possibility of conflict with future versions of the C or POSIX standards, so you should avoid using them:

- Names beginning with a capital '`E`' followed by a digit or uppercase letter may be used for additional error-code names.[4]
- Names that begin with either '`is`' or '`to`' followed by a lowercase letter may be used for additional character-testing and conversion functions.[5]
- Names that begin with '`LC_`' followed by an uppercase letter may be used for additional macros specifying locale attributes.[6]
- Names of all existing mathematics functions suffixed with '`f`' or '`l`' are reserved for corresponding functions that operate on `float` and `long double` arguments, respectively.[7]
- Names that begin with '`SIG`' followed by an uppercase letter are reserved for additional signal names (see Section 17.2 [Standard Signals], page 379).
- Names that begin with '`SIG_`' followed by an uppercase letter are reserved for additional signal actions (see Section 17.3.1 [Basic Signal-Handling], page 389).
- Names beginning with '`str`', '`mem`', or '`wcs`' followed by a lowercase letter are reserved for additional string and array functions.[8]
- Names that end with '`_t`' are reserved for additional type names.

In addition, some individual header files reserve names beyond those that they actually define. You only need to worry about these restrictions if your program includes that particular header file.

- The header file '`dirent.h`' reserves names prefixed with '`d_`'.
- The header file '`fcntl.h`' reserves names prefixed with '`l_`', '`F_`', '`O_`', and '`S_`'.
- The header file '`grp.h`' reserves names prefixed with '`gr_`'.

[4] Loosemore et al., "Error-Reporting" (see chap. 1, n.1).
[5] Ibid., "Character Handling".
[6] Ibid., "Locales and Internationalization".
[7] Ibid., "Mathematics".
[8] Ibid., "String and Array Utilities".

- The header file ‘limits.h’ reserves names suffixed with ‘_MAX’.
- The header file ‘pwd.h’ reserves names prefixed with ‘pw_’.
- The header file ‘signal.h’ reserves names prefixed with ‘sa_’ and ‘SA_’.
- The header file ‘sys/stat.h’ reserves names prefixed with ‘st_’ and ‘S_’.
- The header file ‘sys/times.h’ reserves names prefixed with ‘tms_’.
- The header file ‘termios.h’ reserves names prefixed with ‘c_’, ‘V’, ‘I’, ‘O’, and ‘TC’; and names prefixed with ‘B’ followed by a digit.

1.3.4 Feature-Test Macros

The exact set of features available when you compile a source file is controlled by which *feature-test macros* you define.

If you compile your programs using ‘gcc -ansi’, you get only the ISO C library features, unless you explicitly request additional features by defining one or more of the feature macros.[9]

You should define these macros by using ‘#define’ preprocessor directives at the top of your source code files. These directives *must* come before any #include of a system header file. It is best to make them the very first thing in the file, preceded only by comments. You could also use the ‘-D’ option to GCC, but it is better if you make the source files indicate their own meaning in a self-contained way.

This system exists to allow the library to conform to multiple standards. Although the different standards are often described as supersets of each other, they are usually incompatible because larger standards require functions with names that smaller ones reserve to the user program. This is not mere pedantry—it has been a problem in practice. For instance, some non-GNU programs define functions named getline that have nothing to do with this library’s getline. They would not be compilable if all features were enabled indiscriminately.

This should not be used to verify that a program conforms to a limited standard. It is insufficient for this purpose, as it will not protect you from including header files outside the standard, or relying on semantics undefined within the standard.

_POSIX_SOURCE Macro

If you define this macro, then the functionality from the POSIX.1 standard (IEEE Standard 1003.1) is available, as well as all of the ISO C facilities.

The state of _POSIX_SOURCE is irrelevant if you define the macro _POSIX_C_SOURCE to a positive integer.

[9] See Richard M. Stallman and the GCC Developer Community, “Invoking GCC” in *Using GCC: The GNU Compiler Collection Reference Manual* (Boston, MA: GNU Press, October 2003), http://gcc.gnu.org/onlinedocs/gcc-3.3.2/gcc/, for more information about GCC options.

_POSIX_C_SOURCE Macro

Define this macro to a positive integer to control which POSIX functionality is made available. The greater the value of this macro, the more functionality is made available.

If you define this macro to a value greater than or equal to `1`, then the functionality from the 1990 edition of the POSIX.1 standard (IEEE Standard 1003.1-1990) is made available.

If you define this macro to a value greater than or equal to `2`, then the functionality from the 1992 edition of the POSIX.2 standard (IEEE Standard 1003.2-1992) is made available.

If you define this macro to a value greater than or equal to `199309L`, then the functionality from the 1993 edition of the POSIX.1b standard (IEEE Standard 1003.1b-1993) is made available.

Greater values for `_POSIX_C_SOURCE` will enable future extensions. The POSIX standards process will define these values as necessary, and the GNU C Library should support them some time after they become standardized. The 1996 edition of POSIX.1 (ISO/IEC 9945-1: 1996) states that if you define `_POSIX_C_SOURCE` to a value greater than or equal to `199506L`, then the functionality from the 1996 edition is made available.

_BSD_SOURCE Macro

If you define this macro, functionality derived from 4.3 BSD Unix is included as well as the ISO C, POSIX.1, and POSIX.2 material.

Some of the features derived from 4.3 BSD Unix conflict with the corresponding features specified by the POSIX.1 standard. If this macro is defined, the 4.3 BSD definitions take precedence over the POSIX definitions.

Due to the nature of some of the conflicts between 4.3 BSD and POSIX.1, you need to use a special BSD *compatibility library* when linking programs compiled for BSD compatibility. This is because some functions must be defined in two different ways, one in the normal C library, and one in the compatibility library. If your program defines `_BSD_SOURCE`, you must give the option '`-lbsd-compat`' to the compiler or linker when linking the program, to tell it to find functions in this special compatibility library before looking for them in the normal C library.

_SVID_SOURCE Macro

If you define this macro, functionality derived from SVID is included as well as the ISO C, POSIX.1, POSIX.2 and X/Open material.

_XOPEN_SOURCE Macro
_XOPEN_SOURCE_EXTENDED Macro

If you define this macro, functionality described in the *X/Open Portability Guide*[10] is included. This is a superset of the POSIX.1 and POSIX.2 functional-

[10] X/Open Company, *X/Open Portability Guide,* Issue 4, Version 2 (Reading, UK: X/Open Company, Ltd., 1994).

ity and in fact _POSIX_SOURCE and _POSIX_C_SOURCE are automatically defined.

As the unification of all Unices, functionality only available in BSD and SVID is also included.

If the macro _XOPEN_SOURCE_EXTENDED is also defined, even more functionality is available. The extra functions will make all functions available that are necessary for the X/Open Unix brand.

If the macro _XOPEN_SOURCE has the value 500, this includes all functionality described so far plus some new definitions from the Single Unix Specification, version 2.

_LARGEFILE_SOURCE Macro

If this macro is defined, some extra functions are available that rectify a few shortcomings in all previous standards. Specifically, the functions fseeko and ftello are available. Without these functions, the difference between the ISO C interface (fseek, ftell) and the low-level POSIX interface (lseek) would lead to problems.

This macro was introduced as part of the Large File Support extension (LFS).

_LARGEFILE64_SOURCE Macro

If you define this macro, an additional set of functions is made available that enables 32-bit systems to use files of sizes beyond the usual limit of 2GB. This interface is not available if the system does not support files that large. On systems where the natural file size limit is greater than 2GB (i.e., on 64-bit systems), the new functions are identical to the replaced functions.

The new functionality is made available by a new set of types and functions that replace the existing ones. The names of these new objects contain 64 to indicate the intention, e.g., off_t vs. off64_t and fseeko vs. fseeko64.

This macro was introduced as part of the Large File Support extension (LFS). It is a transition interface for the period when 64-bit offsets are not generally used (see _FILE_OFFSET_BITS).

_FILE_OFFSET_BITS Macro

This macro determines which file-system interface will be used, one replacing the other. Whereas _LARGEFILE64_SOURCE makes the 64-bit interface available as an additional interface, _FILE_OFFSET_BITS allows the 64-bit interface to replace the old interface.

If _FILE_OFFSET_BITS is undefined, or if it is defined to the value 32, nothing changes. The 32-bit interface is used and types like off_t have a size of 32 bits on 32-bit systems.

If the macro is defined to the value 64, the large file interface replaces the old interface. The functions are not made available under different names (as they are with _LARGEFILE64_SOURCE); instead, the old function names now reference the new functions, e.g., a call to fseeko now indeed calls fseeko64.

This macro should only be selected if the system provides mechanisms for handling large files. On 64-bit systems this macro has no effect since the `*64` functions are identical to the normal functions.

This macro was introduced as part of the Large File Support extension (LFS).

_ISOC99_SOURCE Macro

Until the revised ISO C standard is widely adopted the new features are not automatically enabled. The GNU libc nevertheless has a complete implementation of the new standard. To enable the new features the macro `_ISOC99_SOURCE` should be defined.

_GNU_SOURCE Macro

If you define this macro, everything is included: ISO C89, ISO C99, POSIX.1, POSIX.2, BSD, SVID, X/Open, LFS, and GNU extensions. In the cases where POSIX.1 conflicts with BSD, the POSIX definitions take precedence.

If you want to get the full effect of `_GNU_SOURCE` but make the BSD definitions take precedence over the POSIX definitions, use this sequence of definitions:

```
#define _GNU_SOURCE
#define _BSD_SOURCE
#define _SVID_SOURCE
```

If you do this, you must link your program with the BSD compatibility library by passing the '`-lbsd-compat`' option to the compiler or linker. If you forget, you may get very strange errors at run time.

_REENTRANT Macro
_THREAD_SAFE Macro

If you define one of these macros, reentrant versions of several functions get declared. Some of the functions are specified in POSIX.1c, but many others are only available on a few other systems or are unique to GNU libc. The problem is the delay in the standardization of the thread safe C library interface.

Unlike on some other systems, no special version of the C library must be used for linking. There is only one version—but while compiling this, it must have been specified to compile as thread safe.

We recommend you use `_GNU_SOURCE` in new programs. If you don't specify the '`-ansi`' option to GCC and do not define any of these macros explicitly, the effect is the same as defining `_POSIX_C_SOURCE` to 2 and `_POSIX_SOURCE`, `_SVID_SOURCE` and `_BSD_SOURCE` to 1.

When you define a feature-test macro to request a larger class of features, it is harmless to define, in addition, a feature-test macro for a subset of those features. For example, if you define `_POSIX_C_SOURCE`, then defining `_POSIX_SOURCE` as well has no effect. Likewise, if you define `_GNU_SOURCE`, defining either `_POSIX_SOURCE`, `_POSIX_C_SOURCE`, or `_SVID_SOURCE` as well has no effect.

Note, however, that the features of `_BSD_SOURCE` are not a subset of any of the other feature-test macros supported. This is because it defines BSD features that take precedence over the POSIX features that are requested by the other macros. For this reason, defining `_BSD_SOURCE` in addition to the other feature-test macros does have an effect—it causes the BSD features to take priority over the conflicting POSIX features.

1.4 Road Map to the Manual

Here is an overview of the contents of the remaining chapters of this manual.

The following chapters are found in the first volume, Sandra Loosemore et al., *GNU C Library: Application Fundamentals* (Boston: GNU Press, 2004), available online at `http://www.gnu.org/manual/manual.html`.

- "Error Reporting" describes how errors detected by the library are reported.
- "Virtual Memory Allocation and Paging" describes the GNU library's facilities for managing and using virtual and real memory, including dynamic allocation of virtual memory. If you do not know in advance how much memory your program needs, you can allocate it dynamically instead, and manipulate it via pointers.
- "Character Handling" contains information about character-classification functions (such as `isspace`) and functions for performing case conversion.
- "String and Array Utilities" has descriptions of functions for manipulating strings (null-terminated character arrays) and general byte arrays, including operations such as copying and comparison.
- "Character-Set Handling" contains information about manipulating characters and strings using character sets larger than will fit in the usual `char` data type.
- "Locales and Internationalization" describes how selecting a particular country or language affects the behavior of the library. For example, the locale affects collation sequences for strings and how monetary values are formatted.
- "Mathematics" contains information about the math library functions. These include things like random-number generators and remainder functions on integers as well as the usual trigonometric and exponential functions on floating-point numbers.
- "Arithmetic Functions" describes functions for simple arithmetic, analysis of floating-point values, and reading numbers from strings.
- "Date and Time" describes functions for measuring both calendar time and CPU time, as well as functions for setting alarms and timers.
- "Message Translation" describes how to write programs that are capable of delivering messages in whatever language the user selects without filling the source code with sets of translations.
- "Searching and Sorting" contains information about functions for searching and sorting arrays. You can use these functions on any kind of array by providing an appropriate comparison function.

- "Pattern Matching" presents functions for matching regular expressions and shell file-name patterns, and for expanding words as the shell does.
- "The Basic Program/System Interface" tells how your programs can access their command-line arguments and environment variables.
- "Input/Output Overview" gives an overall look at the input and output facilities in the library, and contains information about basic concepts such as file names.
- "Debugging Support" describes functions provided by the library to make the debugging process easier, whether or not a dedicated debugger program is being used.
- "Input/Output on Streams" describes I/O operations involving streams (or `FILE *` objects). These are the normal C library functions from '`stdio.h`'.
- "Summary of Library Facilities" gives a summary of all the functions, variables, and macros in the library, with complete data types and function prototypes, and says what standard or system each is derived from. This section is also found in the second volume, for convenient reference.

The following chapters are found in the second volume, Sandra Loosemore et al., *GNU C Library: System & Network Applications* (Boston: GNU Press, 2004), available online at `http://www.gnu.org/manual/manual.html`.

- "Low-Level Input/Output" contains information about I/O operations on file descriptors. File descriptors are a lower-level mechanism specific to the Unix family of operating systems.
- "File-System Interface" has descriptions of operations on entire files, such as functions for deleting and renaming them and for creating new directories. This chapter also contains information about how you can access the attributes of a file, such as its owner and file-protection modes.
- "Pipes and FIFOs" contains information about simple interprocess-communication mechanisms. Pipes allow communication between two related processes (such as between a parent and child), while FIFOs allow communication between processes sharing a common file-system on the same machine.
- "Sockets" describes a more complicated interprocess-communication mechanism that allows processes running on different machines to communicate over a network. This chapter also contains information about Internet host-addressing and how to use the system network databases.
- "Low-Level Terminal Interface" describes how you can change the attributes of a terminal device. If you want to disable echo of characters typed by the user, for example, read this chapter.
- "Processes" contains information about how to start new processes and run programs.
- "Job Control" describes functions for manipulating process groups and the controlling terminal. This material is probably only of interest if you are writing a shell or other program that handles job control specially.

- "System Databases and Name-Service Switch" describes the services that are available for looking up names in the system databases, how to determine which service is used for which database, and how these services are implemented so that contributors can design their own services.
- "Users and Groups" tells you how to access the system user- and group-databases.
- "System Management" describes functions for controlling and getting information about the hardware and software configuration your program is executing under.
- "System-Configuration Parameters" tells you how you can get information about various operating system limits. Most of these parameters are provided for compatibility with POSIX.
- "DES Encryption and Password Handling" discusses the legal and technical issues related to password encryption and security, as well as the functions necessary to implement effective encryption.
- "Resource Usage and Limitation" tells you how to monitor the memory and other resource usage totals of processes, and how to regulate this usage. It also covers prioritization and scheduling.
- "Syslog" describes facilities for issuing and logging messages of system administration interest.
- "Nonlocal Exits" contains descriptions of the `setjmp` and `longjmp` functions. These functions provide a facility for `goto`-like jumps that can jump from one function to another.
- "Signal Handling" tells you all about signals—what they are, how to establish a handler that is called when a particular kind of signal is delivered, and how to prevent signals from arriving during critical sections of your program.
- "POSIX Threads" describes the pthreads (POSIX threads) library. This library provides support functions for multithreaded programs: thread primitives, synchronization objects, etc. It also implements POSIX 1003.1b semaphores.
- "C Language Facilities in the Library" contains information about library support for standard parts of the C language, including things like the `sizeof` operator and the symbolic constant `NULL`, how to write functions accepting variable numbers of arguments, and constants describing the ranges and other properties of the numerical types. There is also a simple debugging mechanism that allows you to put assertions in your code and have diagnostic messages printed if the tests fail.
- "Installing the GNU C Library" provides a detailed reference for installing, compiling and configuring the GNU C Library. Configuration and optimization command-line options are covered here.
- "Library Maintenance" explains how to port and enhance the GNU C Library and how to report any bugs you might find.

If you already know the name of the facility you are interested in, you can look it up in Appendix B [Summary of Library Facilities], page 475. This gives you a

summary of its syntax and a pointer to where you can find a more detailed description. This appendix is particularly useful if you just want to verify the order and type of arguments to a function, for example. It also tells you what standard or system each function, variable, or macro is derived from.

2 Low-Level Input/Output

This chapter describes functions for performing low-level input/output operations on file descriptors. These functions include the primitives for the higher-level I/O functions,[1] as well as functions for performing low-level control operations for which there are no equivalents on streams.

Stream-level I/O is more flexible and usually more convenient; therefore, programmers generally use the descriptor-level functions only when necessary. These are some of the usual reasons:

- For reading binary files in large chunks
- For reading an entire file into core before parsing it
- To perform operations other than data transfer, which can only be done with a descriptor; you can use `fileno` to get the descriptor corresponding to a stream.
- To pass descriptors to a child process; the child can create its own stream to use a descriptor that it inherits, but it cannot inherit a stream directly.

2.1 Opening and Closing Files

This section describes the primitives for opening and closing files using file descriptors. The `open` and `creat` functions are declared in the header file '`fcntl.h`', while `close` is declared in '`unistd.h`'.

`int` **open** (`const char *`*filename*`, int` *flags*`[, mode_t` *mode*`]`) Function

The `open` function creates and returns a new file-descriptor for the file named by *filename*. Initially, the file position indicator for the file is at the beginning of the file. The argument *mode* is used only when a file is created, but it doesn't hurt to supply the argument in any case.

The *flags* argument controls how the file is to be opened. This is a bit mask; you create the value by the bit-wise OR of the appropriate parameters, using the '`|`' operator in C (for the parameters available, see Section 2.14 [File Status Flags], page 59).

The normal return value from `open` is a nonnegative integer file descriptor. In the case of an error, a value of -1 is returned instead. In addition to the usual file-name errors, the following `errno` error conditions are defined for this function:[2]

`EACCES` The file exists but is not readable or writable as requested by the *flags* argument; the file does not exist and the directory is unwritable, so it cannot be created.

[1] See Sandra Loosemore et al., "Input/Output on Streams" (see chap. 1, n. 1).

[2] Ibid., "File-Name Errors".

`EEXIST` Both `O_CREAT` and `O_EXCL` are set, and the named file already exists.

`EINTR` The `open` operation was interrupted by a signal (see Section 17.5 [Primitives Interrupted by Signals], page 408).

`EISDIR` The *flags* argument specified write access, and the file is a directory.

`EMFILE` The process has too many files open. The maximum number of file descriptors is controlled by the `RLIMIT_NOFILE` resource limit (see Section 14.2 [Limiting Resource Usage], page 338).

`ENFILE` The entire system, or perhaps the file system that contains the directory, cannot support any additional open files at the moment. This problem cannot happen on the GNU system.

`ENOENT` The named file does not exist, and `O_CREAT` is not specified.

`ENOSPC` The directory or file system that would contain the new file cannot be extended, because there is no disk space left.

`ENXIO` `O_NONBLOCK` and `O_WRONLY` are both set in the *flags* argument, the file named by *filename* is a FIFO (see Chapter 4 [Pipes and FIFOs], page 119), and no process has the file open for reading.

`EROFS` The file resides on a read-only file system and any of `O_WRONLY`, `O_RDWR` or `O_TRUNC` are set in the *flags* argument; or `O_CREAT` is set and the file does not already exist.

If on a 32-bit machine the sources are translated with `_FILE_OFFSET_BITS == 64`, the function `open` returns a file descriptor opened in the large file mode that enables the file-handling functions to use files up to 2^63 bytes in size and offset from -2^63 to 2^63. This happens transparently for the user, since all of the low-level file-handling functions are equally replaced.

This function is a cancellation point in multithreaded programs. This is a problem if the thread allocates some resources (like memory, file descriptors, semaphores, etc.) at the time `open` is called. If the thread gets canceled, these resources stay allocated until the program ends. To avoid this, calls to `open` should be protected using cancellation handlers.

The `open` function is the underlying primitive for the `fopen` and `freopen` functions, which create streams.

`int` **open64** (`const char *`*filename*`, int` *flags*`[, mode_t` *mode*`]`) Function

This function is similar to `open`. It returns a file descriptor that can be used to access the file named by *filename*. The only difference is that on 32-bit systems, the file is opened in the large file mode, so file length and file offsets can exceed 31 bits.

When the sources are translated with `_FILE_OFFSET_BITS == 64`, this function is actually available under the name `open`—the new, extended API using 64-bit file sizes and offsets transparently replaces the old API.

`int` **creat** (`const char *`*filename*`, mode_t` *mode*) Obsolete function

This function is obsolete. The call:

```
creat (filename, mode)
```

is equivalent to:

```
open (filename, O_WRONLY | O_CREAT | O_TRUNC, mode)
```

If on a 32-bit machine the sources are translated with `_FILE_OFFSET_BITS == 64`, the function `creat` returns a file descriptor opened in the large file mode that enables the file-handling functions to use files up to 2^63 in size and offset from -2^63 to 2^63. This happens transparently for the user, since all of the low-level file-handling functions are equally replaced.

`int` **creat64** (`const char *`*filename*`, mode_t` *mode*) Obsolete function

This function is similar to `creat`. It returns a file descriptor that can be used to access the file named by *filename*. The only difference is that on 32-bit systems the file is opened in the large file mode, so file length and file offsets can exceed 31 bits.

To use this file descriptor, you must not use the normal operations but instead the counterparts named `*64`, such as `read64`.

When the sources are translated with `_FILE_OFFSET_BITS == 64`, this function is actually available under the name `open`—the new, extended API using 64-bit file sizes and offsets transparently replaces the old API.

`int` **close** (`int` *filedes*) Function

The function `close` closes the file descriptor *filedes*. Closing a file has the following consequences:

- The file descriptor is deallocated.
- Any record locks owned by the process on the file are unlocked.
- When all file descriptors associated with a pipe or FIFO have been closed, any unread data is discarded.

This function is a cancellation point in multithreaded programs. This is a problem if the thread allocates some resources (like memory, file descriptors, semaphores, etc.) at the time `close` is called. If the thread gets canceled, these resources stay allocated until the program ends. To avoid this, calls to `close` should be protected using cancellation handlers.

The normal return value from `close` is 0; a value of -1 is returned in case of failure. The following `errno` error conditions are defined for this function:

`EBADF` The *filedes* argument is not a valid file-descriptor.

`EINTR` The `close` call was interrupted by a signal (see Section 17.5 [Primitives Interrupted by Signals], page 408). Here is an example of how to handle `EINTR` properly:

```
TEMP_FAILURE_RETRY (close (desc));
```

`ENOSPC`
`EIO`
`EDQUOT` When the file is accessed by NFS, these errors from `write` can sometimes go undetected until `close` (see Section 2.2 [Input and Output Primitives], page 20 for details on their meaning).

There is *no* separate `close64` function. This is not necessary, since this function does not determine nor depend on the mode of the file. The kernel that performs the `close` operation knows which mode the descriptor is used for and can handle this situation.

To close a stream, call `fclose` instead of trying to close its underlying file-descriptor with `close`.[3] This flushes any buffered output and updates the stream object to indicate that it is closed.

2.2 Input and Output Primitives

This section describes the functions for performing primitive input and output operations on file descriptors: `read`, `write` and `lseek`. These functions are declared in the header file '`unistd.h`'.

ssize_t Data Type

This data type is used to represent the sizes of blocks that can be read or written in a single operation. It is similar to `size_t`, but must be a signed type.

`ssize_t` **read** (`int` *filedes*, `void *`*buffer*, `size_t` *size*) Function

The `read` function reads up to *size* bytes from the file with descriptor *filedes*, storing the results in the *buffer*. This is not necessarily a character string, and no terminating-null character is added.

The return value is the number of bytes actually read. This might be less than *size*; for example, if there aren't that many bytes left in the file or if there aren't that many bytes immediately available. The exact behavior depends on what kind of file it is. Reading less than *size* bytes is not an error.

A value of 0 indicates end of file (except if the value of the *size* argument is also 0). This is not considered an error. If you keep calling `read` while at end of file, it will keep returning 0 and doing nothing else.

If `read` returns at least one character, there is no way you can tell whether end of file was reached. But if you did reach the end, the next read will return 0.

[3] Ibid., "Closing Streams".

In case of an error, `read` returns −1. The following `errno` error conditions are defined for this function:

`EAGAIN` Normally, when no input is immediately available, `read` waits for some input. But if the `O_NONBLOCK` flag is set for the file (see Section 2.14 [File Status Flags], page 59), `read` returns immediately without reading any data, and reports this error.

Compatibility Note: Most versions of BSD Unix use a different error code for this, `EWOULDBLOCK`. In the GNU library, `EWOULDBLOCK` is an alias for `EAGAIN`, so it doesn't matter which name you use.

On some systems, reading a large amount of data from a character-special file can also fail with `EAGAIN` if the kernel cannot find enough physical memory to lock down the user's pages. This is limited to devices that transfer with direct memory access into the user's memory, which means it does not include terminals, since they always use separate buffers inside the kernel. This problem never happens in the GNU system.

Any condition that could result in `EAGAIN` can instead result in a successful `read` that returns fewer bytes than requested. Calling `read` again immediately would result in `EAGAIN`.

`EBADF` The *filedes* argument is not a valid file-descriptor, or is not open for reading.

`EINTR` `read` was interrupted by a signal while it was waiting for input (see Section 17.5 [Primitives Interrupted by Signals], page 408). A signal will not necessarily cause `read` to return `EINTR`; it may instead result in a successful `read` that returns fewer bytes than requested.

`EIO` For many devices and for disk files, this error code indicates a hardware error.

`EIO` also occurs when a background process tries to read from the controlling terminal, and the normal action of stopping the process by sending it a `SIGTTIN` signal isn't working. This might happen if the signal is being blocked or ignored, or because the process group is orphaned. (See Chapter 8 [Job Control], page 221 for more information about job control, and Chapter 17 [Signal Handling], page 377 for information about signals.)

There is no function named `read64`. This is not necessary, since this function does not directly modify or handle the possibly wide file offset. Since the kernel handles this state internally, the `read` function can be used for all cases.

This function is a cancellation point in multithreaded programs. This is a problem if the thread allocates some resources (like memory, file descriptors, semaphores, etc.) at the time `read` is called. If the thread gets canceled, these

resources stay allocated until the program ends. To avoid this, calls to `read` should be protected using cancellation handlers.

The `read` function is the underlying primitive for all of the functions that read from streams, such as `fgetc`.

`ssize_t` **pread** (`int` *filedes*, `void *`*buffer*, `size_t` *size*, `off_t` *offset*) Function

The `pread` function is similar to the `read` function. The first three arguments are identical, and the return values and error codes also correspond.

The difference is the fourth argument and its handling. The data block is not read from the current position of the file descriptor `filedes`. Instead the data is read from the file starting at position *offset*. The position of the file descriptor itself is not affected by the operation. The value is the same as before the call.

When the source file is compiled with `_FILE_OFFSET_BITS == 64`, the `pread` function is in fact `pread64`, and the type `off_t` has 64 bits, which makes it possible to handle files up to 2^63 bytes in length.

The return value of `pread` describes the number of bytes read. In the case of an error, it returns -1 like `read` does. The error codes are also the same, with these additions:

`EINVAL` The value given for *offset* is negative and therefore illegal.

`ESPIPE` The file descriptor *filedes* is associate with a pipe or a FIFO and this device does not allow positioning of the file pointer.

The function is an extension defined in the Unix Single Specification, version 2.

`ssize_t` **pread64** (`int` *filedes*, `void *`*buffer*, `size_t` *size*, `off64_t` *offset*) Function

This function is similar to the `pread` function. The difference is that the *offset* parameter is of type `off64_t` instead of `off_t`, which makes it possible on 32-bit machines to address files larger than 2^31 bytes and up to 2^63 bytes. The file descriptor `filedes` must be opened using `open64` since otherwise the large offsets possible with `off64_t` will lead to errors with a descriptor in small file mode.

When the source file is compiled with `_FILE_OFFSET_BITS == 64` on a 32-bit machine, this function is actually available under the name `pread` and so transparently replaces the 32-bit interface.

`ssize_t` **write** (`int` *filedes*, `const void *`*buffer*, `size_t` *size*) Function

The `write` function writes up to *size* bytes from *buffer* to the file with descriptor *filedes*. The data in *buffer* is not necessarily a character string, and a null character is output like any other character.

The return value is the number of bytes actually written. This may be *size*, but can always be smaller. Your program should always call `write` in a loop, iterating until all the data is written.

Once `write` returns, the data is enqueued to be written and can be read back right away, but it is not necessarily written out to permanent storage immediately. You can use `fsync` when you need to be sure your data has been permanently stored before continuing. It is more efficient for the system to batch up consecutive writes and do them all at once when convenient. Normally they will always be written to disk within a minute or less. Modern systems provide another function, `fdatasync`, which guarantees integrity only for the file data and is therefore faster. You can use the `O_FSYNC` open mode to make `write` always store the data to disk before returning (see Section 2.14.3 [I/O Operating Modes], page 62).

In the case of an error, `write` returns -1. The following `errno` error conditions are defined for this function:

`EAGAIN` Normally, `write` blocks until the write operation is complete. But if the `O_NONBLOCK` flag is set for the file (see Section 2.11 [Control Operations on Files], page 54), it returns immediately without writing any data and reports this error. An example of a situation that might cause the process to block on output is writing to a terminal device that supports flow control, where output has been suspended by receipt of a STOP character.

Compatibility Note: Most versions of BSD Unix use a different error code for this: `EWOULDBLOCK`. In the GNU library, `EWOULDBLOCK` is an alias for `EAGAIN`, so it doesn't matter which name you use.

On some systems, writing a large amount of data from a character-special file can also fail with `EAGAIN` if the kernel cannot find enough physical memory to lock down the user's pages. This is limited to devices that transfer with direct memory access into the user's memory, which means it does not include terminals, since they always use separate buffers inside the kernel. This problem does not arise in the GNU system.

`EBADF` The *filedes* argument is not a valid file-descriptor, or is not open for writing.

`EFBIG` The size of the file would become larger than the implementation can support.

`EINTR` The `write` operation was interrupted by a signal while it was blocked waiting for completion. A signal will not necessarily cause `write` to return `EINTR`; it may instead result in a successful `write` that writes fewer bytes than requested (see Section 17.5 [Primitives Interrupted by Signals], page 408).

`EIO` For many devices and for disk files, this error code indicates a hardware error.

`ENOSPC` The device containing the file is full.

`EPIPE` This error is returned when you try to write to a pipe or FIFO that isn't open for reading by any process. When this happens, a `SIGPIPE` signal is also sent to the process (see Chapter 17 [Signal Handling], page 377).

Unless you have arranged to prevent `EINTR` failures, you should check `errno` after each failing call to `write`, and if the error was `EINTR`, you should simply repeat the call (see Section 17.5 [Primitives Interrupted by Signals], page 408). The easy way to do this is with the macro `TEMP_FAILURE_RETRY`, as follows:

```
nbytes = TEMP_FAILURE_RETRY (write (desc, buffer, count));
```

There is no function named `write64`. This is not necessary, since this function does not directly modify or handle the possibly wide file offset. Since the kernel handles this state internally the `write` function can be used for all cases.

This function is a cancellation point in multithreaded programs. This is a problem if the thread allocates some resources (like memory, file descriptors, semaphores or whatever) at the time `write` is called. If the thread gets canceled, these resources stay allocated until the program ends. To avoid this, calls to `write` should be protected using cancellation handlers.

The `write` function is the underlying primitive for all of the functions that write to streams, such as `fputc`.

`ssize_t` **pwrite** (`int` *filedes*, `const void *`*buffer*, `size_t` *size*, `off_t` *offset*) Function

The `pwrite` function is similar to the `write` function. The first three arguments are identical, and the return values and error codes also correspond.

The difference is the fourth argument and its handling. The data block is not written to the current position of the file descriptor `filedes`. Instead the data is written to the file starting at position *offset*. The position of the file descriptor itself is not affected by the operation. The value is the same as before the call.

When the source file is compiled with `_FILE_OFFSET_BITS == 64`, the `pwrite` function is in fact `pwrite64` and the type `off_t` has 64 bits, which makes it possible to handle files up to 2^63 bytes in length.

The return value of `pwrite` describes the number of written bytes. In the case of an error, it returns -1 like `write` does. The error codes are also the same, with these additions:

`EINVAL` The value given for *offset* is negative and therefore illegal.

`ESPIPE` The file descriptor *filedes* is associated with a pipe or a FIFO, and this device does not allow positioning of the file pointer.

The function is an extension defined in the Unix Single Specification, version 2.

`ssize_t` **pwrite64** (`int` *filedes*, `const void *`*buffer*, `size_t` *size*, `off64_t` *offset*) Function

This function is similar to the `pwrite` function. The difference is that the *offset* parameter is of type `off64_t` instead of `off_t`, which makes it possible on 32-bit machines to address files larger than 2^31 bytes and up to 2^63 bytes. The file descriptor `filedes` must be opened using `open64`, since otherwise the large offsets possible with `off64_t` will lead to errors with a descriptor in small file mode.

When the source file is compiled using `_FILE_OFFSET_BITS == 64` on a 32-bit machine, this function is actually available under the name `pwrite` and so transparently replaces the 32-bit interface.

2.3 Setting the File Position of a Descriptor

Just as you can set the file position of a stream with `fseek`, you can set the file position of a descriptor with `lseek`. This specifies the position in the file for the next `read` or `write` operation.[4]

To read the current file-position value from a descriptor, use `lseek (`*desc*`, 0, SEEK_CUR)`.

`off_t` **lseek** (`int` *filedes*, `off_t` *offset*, `int` *whence*) Function

The `lseek` function is used to change the file position of the file with descriptor *filedes*.

The *whence* argument specifies how the *offset* should be interpreted, in the same way as for the `fseek` function, and it must be one of the symbolic constants `SEEK_SET`, `SEEK_CUR` or `SEEK_END`.

`SEEK_SET`
: This specifies that *whence* is a count of characters from the beginning of the file.

`SEEK_CUR`
: This specifies that *whence* is a count of characters from the current file position. This count may be positive or negative.

`SEEK_END`
: This specifies that *whence* is a count of characters from the end of the file. A negative count specifies a position within the current extent of the file; a positive count specifies a position past the current end. If you set the position past the current end, and actually write data, you will extend the file with zeros up to that position.

The return value from `lseek` is normally the resulting file position, measured in bytes from the beginning of the file. You can use this feature together with `SEEK_CUR` to read the current file position.

---

[4] Ibid., “File Positioning”.

If you want to append to the file, setting the file position to the current end of file with `SEEK_END` is not sufficient. Another process may write more data after you seek but before you write, extending the file so the position you write onto clobbers their data. Instead, use the `O_APPEND` operating mode (see Section 2.14.3 [I/O Operating Modes], page 62).

You can set the file position past the current end of the file. This does not by itself make the file longer; `lseek` never changes the file. But subsequent output at that position will extend the file. Characters between the previous end of file and the new position are filled with zeros. Extending the file in this way can create a *hole*: the blocks of zeros are not actually allocated on disk, so the file takes up less space than it appears to—it is then called a *sparse file*.

If the file position cannot be changed, or the operation is in some way invalid, `lseek` returns a value of −1. The following `errno` error conditions are defined for this function:

`EBADF` The *filedes* is not a valid file-descriptor.

`EINVAL` The *whence* argument value is not valid, or the resulting file offset is not valid. A file offset is invalid.

`ESPIPE` The *filedes* corresponds to an object that cannot be positioned, such as a pipe, FIFO or terminal device. POSIX.1 specifies this error only for pipes and FIFOs, but in the GNU system, you always get `ESPIPE` if the object is not seekable.

When the source file is compiled with `_FILE_OFFSET_BITS == 64`, the `lseek` function is in fact `lseek64`, and the type `off_t` has 64 bits, which makes it possible to handle files up to 2^63 bytes in length.

This function is a cancellation point in multithreaded programs. This is a problem if the thread allocates some resources (like memory, file descriptors, semaphores, etc.) at the time `lseek` is called. If the thread gets canceled, these resources stay allocated until the program ends. To avoid this, calls to `lseek` should be protected using cancellation handlers.

The `lseek` function is the underlying primitive for the `fseek`, `fseeko`, `ftell`, `ftello` and `rewind` functions, which operate on streams instead of file descriptors.

`off64_t` **lseek64** (`int` *filedes*, `off64_t` *offset*, `int` *whence*) Function

This function is similar to the `lseek` function. The difference is that the *offset* parameter is of type `off64_t` instead of `off_t`, which makes it possible on 32-bit machines to address files larger than 2^31 bytes and up to 2^63 bytes. The file descriptor `filedes` must be opened using `open64`, since otherwise the large offsets possible with `off64_t` will lead to errors with a descriptor in small file mode.

When the source file is compiled with `_FILE_OFFSET_BITS == 64` on a 32-bit machine, this function is actually available under the name `lseek` and so transparently replaces the 32-bit interface.

You can have multiple descriptors for the same file if you open the file more than once, or if you duplicate a descriptor with `dup`. Descriptors that come from separate calls to `open` have independent file positions; using `lseek` on one descriptor has no effect on the other. For example,

```
{
  int d1, d2;
  char buf[4];
  d1 = open ("foo", O_RDONLY);
  d2 = open ("foo", O_RDONLY);
  lseek (d1, 1024, SEEK_SET);
  read (d2, buf, 4);
}
```

will read the first four characters of the file '`foo`'. (The error-checking code necessary for a real program has been omitted here for brevity.)

By contrast, descriptors made by duplication share a common file position with the original descriptor that was duplicated. Anything that alters the file position of one of the duplicates, including reading or writing data, affects all of them alike. Thus, for example:

```
{
  int d1, d2, d3;
  char buf1[4], buf2[4];
  d1 = open ("foo", O_RDONLY);
  d2 = dup (d1);
  d3 = dup (d2);
  lseek (d3, 1024, SEEK_SET);
  read (d1, buf1, 4);
  read (d2, buf2, 4);
}
```

will read four characters starting with the 1024th character of '`foo`', and then four more characters starting with the 1028th character.

off_t Data Type

This is an arithmetic data type used to represent file sizes. In the GNU system, this is equivalent to `fpos_t` or `long int`.

If the source is compiled with `_FILE_OFFSET_BITS == 64`, this type is transparently replaced by `off64_t`.

off64_t Data Type

This type is used similar to `off_t`. The difference is that even on 32-bit machines, where the `off_t` type would have 32 bits, `off64_t` has 64 bits and so is able to address files up to 2^63 bytes in length.

When compiling with `_FILE_OFFSET_BITS == 64`, this type is available under the name `off_t`.

These aliases for the '`SEEK_...`' constants exist for the sake of compatibility with older BSD systems. They are defined in two different header files: '`fcntl.h`' and '`sys/file.h`'.

`L_SET` An alias for `SEEK_SET`

`L_INCR` An alias for `SEEK_CUR`

`L_XTND` An alias for `SEEK_END`

2.4 Descriptors and Streams

Given an open file-descriptor, you can create a stream for it with the `fdopen` function. You can get the underlying file-descriptor for an existing stream with the `fileno` function. These functions are declared in the header file '`stdio.h`'.

`FILE *` **fdopen** (`int` *filedes*, `const char *`*opentype*) Function

The `fdopen` function returns a new stream for the file descriptor *filedes*.

The *opentype* argument is interpreted in the same way as for the `fopen` function,[5] except that the '`b`' option is not permitted; this is because GNU makes no distinction between text and binary files. Also, '`w`' and '`w+`' do not cause truncation of the file; these have an effect only when opening a file, and in this case the file has already been opened. You must make sure that the *opentype* argument matches the actual mode of the open file descriptor.

The return value is the new stream. If the stream cannot be created (for example, if the modes for the file indicated by the file descriptor do not permit the access specified by the *opentype* argument), a null pointer is returned instead.

In some other systems, `fdopen` may fail to detect that the modes for file descriptor do not permit the access specified by `opentype`. The GNU C Library always checks for this.

For an example showing the use of the `fdopen` function, see Section 4.1 [Creating a Pipe], page 119.

`int` **fileno** (`FILE *`*stream*) Function

This function returns the file descriptor associated with the stream *stream*. If an error is detected (for example, if the *stream* is not valid) or if *stream* does not do I/O to a file, `fileno` returns -1.

`int` **fileno_unlocked** (`FILE *`*stream*) Function

The `fileno_unlocked` function is equivalent to the `fileno` function, except that it does not implicitly lock the stream if the state is `FSETLOCKING_INTERNAL`.

This function is a GNU extension.

[5] Ibid., "Opening Streams".

There are also symbolic constants defined in ‘unistd.h’ for the file descriptors belonging to the standard streams stdin, stdout and stderr.[6]

STDIN_FILENO
: This macro has value 0, which is the file descriptor for standard input.

STDOUT_FILENO
: This macro has value 1, which is the file descriptor for standard output.

STDERR_FILENO
: This macro has value 2, which is the file descriptor for standard error output.

2.5 Dangers of Mixing Streams and Descriptors

You can have multiple file-descriptors and streams (let’s call both streams and descriptors *channels* for short) connected to the same file, but you must take care to avoid confusion between channels. There are two cases to consider: *linked* channels that share a single file position value, and *independent* channels that have their own file positions.

It’s best to use just one channel in your program for actual data transfer to any given file, except when all the access is for input. For example, if you open a pipe (something you can only do at the file descriptor level), either do all I/O with the descriptor, or construct a stream from the descriptor with fdopen, and then do all I/O with the stream.

2.5.1 Linked Channels

Channels that come from a single opening share the same file position; we call them *linked* channels. Linked channels result when you make a stream from a descriptor using fdopen, when you get a descriptor from a stream with fileno, when you copy a descriptor with dup or dup2, and when descriptors are inherited during fork. For files that don’t support random access, such as terminals and pipes, *all* channels are effectively linked. On random-access files, all append-type output streams are effectively linked to each other.

If you have been using a stream for I/O (or have just opened the stream), and you want to do I/O using another channel (either a stream or a descriptor) that is linked to it, you must first *clean up* the stream that you have been using (see Section 2.5.3 [Cleaning Streams], page 30).

Terminating a process, or executing a new program in the process, destroys all the streams in the process. If descriptors linked to these streams persist in other processes, their file positions become undefined as a result. To prevent this, you must clean up the streams before destroying them.

---

[6] Ibid., “Standard Streams”.

2.5.2 Independent Channels

When you open channels (streams or descriptors) separately on a seekable file, each channel has its own file position. These are called *independent channels*.

The system handles each channel independently. Most of the time, this is quite predictable and natural (especially for input)—each channel can read or write sequentially at its own place in the file. However, if some of the channels are streams, you must take these precautions:

- You should clean an output stream after use, before doing anything else that might read or write from the same part of the file.
- You should clean an input stream before reading data that may have been modified using an independent channel. Otherwise, you might read obsolete data that had been in the stream's buffer.

If you do output to one channel at the end of the file, this will certainly leave the other independent channels positioned somewhere before the new end. You cannot reliably set their file positions to the new end of file before writing, because the file can always be extended by another process between when you set the file position and when you write the data. Instead, use an append-type descriptor or stream—they always output at the current end of the file. In order to make the end-of-file position accurate, you must clean the output channel you were using, if it is a stream.

It's impossible for two channels to have separate file pointers for a file that doesn't support random access. Thus, channels for reading or writing such files are always linked, never independent. Append-type channels are also always linked. For these channels, follow the rules for linked channels (see Section 2.5.1 [Linked Channels], page 29).

2.5.3 Cleaning Streams

On the GNU system, you can clean up any stream with `fclean`:

`int` **fclean** (`FILE *`*stream*) Function

Clean up the stream *stream* so that its buffer is empty. If *stream* is doing output, force it out. If *stream* is doing input, give the data in the buffer back to the system, arranging to reread it.

On other systems, you can use `fflush` to clean a stream in most cases.

You can skip the `fclean` or `fflush` if you know the stream is already clean. A stream is clean whenever its buffer is empty. For example, an unbuffered stream is always clean. An input stream that is at end-of-file is clean. A line-buffered stream is clean when the last character output was a newline. However, a just-opened input stream might not be clean, as its input buffer might not be empty.

There is one case in which cleaning a stream is impossible on most systems. This is when the stream is doing input from a file that is not random access. Such streams typically read ahead, and when the file is not random access, there is no

way to give back the excess data already read. When an input stream reads from a random-access file, `fflush` does clean the stream, but leaves the file pointer at an unpredictable place; you must set the file pointer before doing any further I/O. On the GNU system, using `fclean` avoids both of these problems.

Closing an output-only stream also does `fflush`, so this is a valid way of cleaning an output stream. On the GNU system, closing an input stream does `fclean`.

You need not clean a stream before using its descriptor for control operations such as setting terminal modes—these operations don't affect the file position and are not affected by it. You can use any descriptor for these operations, and all channels are affected simultaneously. However, text already "output" to a stream but still buffered by the stream will be subject to the new terminal modes when subsequently flushed. To make sure "past" output is covered by the terminal settings that were in effect at the time, flush the output streams for that terminal before setting the modes (see Section 6.4 [Terminal Modes], page 181).

2.6 Fast Scatter-Gather I/O

Some applications may need to read or write data to multiple buffers, which are separated in memory. Although this can be done easily enough with multiple calls to `read` and `write`, it is inefficient because there is overhead associated with each kernel call.

Instead, many platforms provide special high-speed primitives to perform these *scatter-gather* operations in a single kernel call. The GNU C library will provide an emulation on any system that lacks these primitives, so they are not a portability threat. They are defined in `sys/uio.h`.

These functions are controlled with arrays of `iovec` structures, which describe the location and size of each buffer.

struct iovec Data Type

The `iovec` structure describes a buffer. It contains two fields:

`void *iov_base`
: This contains the address of a buffer.

`size_t iov_len`
: This contains the length of the buffer.

`ssize_t` **readv** (`int` *filedes*, `const struct iovec *`*vector*, `int` *count*) Function

The `readv` function reads data from *filedes* and scatters it into the buffers described in *vector*, which is taken to be *count* structures long. As each buffer is filled, data is sent to the next.

`readv` is not guaranteed to fill all the buffers. It may stop at any point, for the same reasons `read` would.

The return value is a count of bytes (*not* buffers) read, 0 indicating end-of-file, or -1 indicating an error. The possible errors are the same as in `read`.

`ssize_t` **writev** (`int` *filedes*, `const struct iovec` `*`*vector*, `int` *count*) Function

The `writev` function gathers data from the buffers described in *vector*, which is taken to be *count* structures long, and writes them to `filedes`. As each buffer is written, it moves on to the next.

Like `readv`, `writev` may stop midstream under the same conditions `write` would.

The return value is a count of bytes written, or -1 indicating an error. The possible errors are the same as in `write`.

If the buffers are small (under about 1kB), high-level streams may be easier to use than these functions. However, `readv` and `writev` are more efficient when the individual buffers themselves (as opposed to the total output), are large. In that case, a high-level stream would not be able to cache the data effectively.

2.7 Memory-Mapped I/O

On modern operating systems, it is possible to *mmap* (pronounced "em-map") a file to a region of memory. When this is done, the file can be accessed just like an array in the program.

This is more efficient than `read` or `write`, since only the regions of the file that a program actually accesses are loaded. Accesses to not-yet-loaded parts of the mmapped region are handled in the same way as swapped-out pages.

Since mmapped pages can be stored back to their file when physical memory is low, it is possible to mmap files orders of magnitude larger than both the physical memory *and* swap space. The only limit is address space. The theoretical limit is 4GB on a 32-bit machine—however, the actual limit will be smaller since some areas will be reserved for other purposes. If the LFS interface is used, the file size on 32-bit systems is not limited to 2GB (offsets are signed, which reduces the addressable area of 4GB by half); the full 64 bits are available.

Memory mapping only works on entire pages of memory. Thus, addresses for mapping must be page aligned, and length values will be rounded up. To determine the size of a page the machine uses, you should use:

```
size_t page_size = (size_t) sysconf (_SC_PAGESIZE);
```

These functions are declared in '`sys/mman.h`'.

`void *` **mmap** (`void *`*address*, `size_t` *length*, `int` *protect*, `int` *flags*, `int` *filedes*, `off_t` *offset*) Function

The `mmap` function creates a new mapping, connected to bytes (*offset*) to (*offset* + *length* - 1) in the file open on *filedes*. A new reference for the file specified by *filedes* is created, which is not removed by closing the file.

address gives a preferred starting address for the mapping. `NULL` expresses no preference. Any previous mapping at that address is automatically removed. The address you give may still be changed, unless you use the `MAP_FIXED` flag.

protect contains flags that control what kind of access is permitted. They include `PROT_READ`, `PROT_WRITE` and `PROT_EXEC`, which permit reading, writing and execution, respectively. Inappropriate access will cause a segfault (see Section 17.2.1 [Program-Error Signals], page 379).

Most hardware designs cannot support write permission without read permission, and many do not distinguish read and execute permission. Thus, you may receive wider permissions than you ask for, and mappings of write-only files may be denied even if you do not use `PROT_READ`.

flags contains flags that control the nature of the map. One of `MAP_SHARED` or `MAP_PRIVATE` must be specified.

They include:

`MAP_PRIVATE`
: This specifies that writes to the region should never be written back to the attached file. Instead, a copy is made for the process, and the region will be swapped normally if memory runs low. No other process will see the changes.

  Since private mappings effectively revert to ordinary memory when written to, you must have enough virtual memory for a copy of the entire mmapped region if you use this mode with `PROT_WRITE`.

`MAP_SHARED`
: This specifies that writes to the region will be written back to the file. Changes made will be shared immediately with other processes mmaping the same file.

  Actual writing may take place at any time. You need to use `msync`, described below, if it is important that other processes using conventional I/O get a consistent view of the file.

`MAP_FIXED`
: This forces the system to use the exact mapping address specified in *address* and to fail if it can't.

`MAP_ANONYMOUS`
`MAP_ANON`
: This flag tells the system to create an anonymous mapping, not connected to a file. *filedes* and *off* are ignored, and the region is initialized with zeros.

  Anonymous maps are used as the basic primitive to extend the heap on some systems. They are also useful to share data between multiple tasks without creating a file.

  On some systems, using private anonymous mmaps is more efficient than using `malloc` for large blocks. This is not an issue with the GNU C Library, since the included `malloc` automatically uses `mmap` where appropriate.

`mmap` returns the address of the new mapping, or −1 for an error.

Possible errors include:

`EINVAL`
: Either *address* was unusable, or inconsistent *flags* were given.

`EACCES`
: *filedes* was not open for the type of access specified in *protect*.

`ENOMEM`
: Either there is not enough memory for the operation, or the process is out of address space.

`ENODEV`
: This file is of a type that doesn't support mapping.

`ENOEXEC`
: The file is on a file system that doesn't support mapping.

`void *` **mmap64** (`void *`*address*`, size_t` *length*`, int` *protect*`, int` *flags*`, int` *filedes*`, off64_t` *offset*) Function

The `mmap64` function is equivalent to the `mmap` function, but the *offset* parameter is of type `off64_t`. On 32-bit systems, this allows the file associated with the *filedes* descriptor to be larger than 2GB. *filedes* must be a descriptor returned from a call to `open64` or `fopen64` and `freopen64`, where the descriptor is retrieved with `fileno`.

When the sources are translated with `_FILE_OFFSET_BITS == 64`, this function is actually available under the name `mmap`—the new, extended API using 64-bit file sizes and offsets transparently replaces the old API.

`int` **munmap** (`void *`*addr*`, size_t` *length*) Function

`munmap` removes any memory maps from (*addr*) to (*addr* + *length*). *length* should be the length of the mapping.

It is safe to unmap multiple mappings in one command, or include unmapped space in the range. It is also possible to unmap only part of an existing mapping. However, only entire pages can be removed. If *length* is not an even number of pages, it will be rounded up.

It returns 0 for success and −1 for an error.

One error is possible:

`EINVAL` The memory range given was outside the user mmap range or wasn't page aligned.

`int` **msync** (`void *`*address*`, size_t` *length*`, int` *flags*) Function

When using shared mappings, the kernel can write the file at any time before the mapping is removed. To be certain data has actually been written to the file

and will be accessible to non-memory-mapped I/O, it is necessary to use this function.

It operates on the region *address* to (*address* + *length*). It may be used on part of a mapping or multiple mappings; however, the region given should not contain any unmapped space.

flags can contain some options:

`MS_SYNC`
: This flag makes sure the data is actually written *to disk*. Normally `msync` only makes sure that accesses to a file with conventional I/O reflect the recent changes.

`MS_ASYNC`
: This tells `msync` to begin the synchronization, but not to wait for it to complete.

`msync` returns 0 for success and −1 for error. Errors include:

`EINVAL` An invalid region was given, or the *flags* were invalid.

`EFAULT` There is no existing mapping in at least part of the given region.

`void *` **mremap** (`void *`*address*`, size_t` *length*`, size_t` *new_length*`, int` *flag*) Function

This function can be used to change the size of an existing memory area. *address* and *length* must cover a region entirely mapped in the same `mmap` statement. A new mapping with the same characteristics will be returned with the length *new_length*.

One option is possible, `MREMAP_MAYMOVE`. If it is given in *flags*, the system may remove the existing mapping and create a new one of the desired length in another location.

The address of the resulting mapping is returned, or −1. Possible error codes include:

`EFAULT` There is no existing mapping in at least part of the original region, or the region covers two or more distinct mappings.

`EINVAL` The address given is misaligned or inappropriate.

`EAGAIN` The region has pages locked, and if extended it would exceed the process's resource limit for locked pages (see Section 14.2 [Limiting Resource Usage], page 338).

`ENOMEM` The region is private writable, and insufficient virtual memory is available to extend it. Also, this error will occur if `MREMAP_MAYMOVE` is not given and the extension would collide with another mapped region.

This function is only available on a few systems. Except for performing optional optimizations, you should not rely on this function.

Not all file descriptors may be mapped. Sockets, pipes and most devices only allow sequential access and do not fit into the mapping abstraction. In addition, some regular files may not be mmapable, and older kernels may not support mapping at all. Thus, programs using `mmap` should have a fallback method to use should it fail.[7]

`int` **madvise** (`void *`*addr*`, size_t` *length*`, int` *advice*) Function

This function can be used to provide the system with *advice* about the intended usage patterns of the memory region starting at *addr* and extending *length* bytes.

The valid BSD values for *advice* are

`MADV_NORMAL`
: The region should receive no further special treatment.

`MADV_RANDOM`
: The region will be accessed via random page references. The kernel should page-in the minimal number of pages for each page fault.

`MADV_SEQUENTIAL`
: The region will be accessed via sequential page references. This may cause the kernel to aggressively read ahead, expecting further sequential references after any page fault within this region.

`MADV_WILLNEED`
: The region will be needed. The pages within this region may be prefaulted in by the kernel.

`MADV_DONTNEED`
: The region is no longer needed. The kernel may free these pages, causing any changes to the pages to be lost, as well as swapped-out pages to be discarded.

The POSIX names are slightly different, but with the same meanings:

`POSIX_MADV_NORMAL`
: This corresponds with BSD's `MADV_NORMAL`.

`POSIX_MADV_RANDOM`
: This corresponds with BSD's `MADV_RANDOM`.

`POSIX_MADV_SEQUENTIAL`
: This corresponds with BSD's `MADV_SEQUENTIAL`.

`POSIX_MADV_WILLNEED`
: This corresponds with BSD's `MADV_WILLNEED`.

[7] Richard Stallman et al., "Mmap" in *GNU Coding Standards* (January 16, 2004), http://www.gnu.org/prep/standards_toc.html.

`POSIX_MADV_DONTNEED`
: This corresponds with BSD's `MADV_DONTNEED`.

`msync` returns 0 for success and −1 for error. Errors include:

`EINVAL` An invalid region was given, or the *advice* was invalid.

`EFAULT` There is no existing mapping in at least part of the given region.

2.8 Waiting for Input or Output

Sometimes a program needs to accept input on multiple input channels whenever input arrives. For example, some workstations may have devices such as a digitizing tablet, function-button box or dial box that are connected via normal asynchronous serial interfaces; good user interface style requires responding immediately to input on any device. Another example is a program that acts as a server to several other processes via pipes or sockets.

You cannot normally use `read` for this purpose, because this blocks the program until input is available on one particular file descriptor; input on other channels won't wake it up. You could set nonblocking mode and poll each file-descriptor in turn, but this is very inefficient.

A better solution is to use the `select` function. This blocks the program until input or output is ready on a specified set of file descriptors, or until a timer expires, whichever comes first. This facility is declared in the header file '`sys/types.h`'.

In the case of a server socket (see Section 5.9.2 [Listening for Connections], page 155), we say that "input" is available when there are pending connections that could be accepted (see Section 5.9.3 [Accepting Connections], page 155). `accept` for server sockets blocks and interacts with `select` just as `read` does for normal input.

The file-descriptor sets for the `select` function are specified as `fd_set` objects. Here is the description of the data type and some macros for manipulating these objects:

fd_set Data Type
: The `fd_set` data type represents file-descriptor sets for the `select` function. It is actually a bit array.

`int` **FD_SETSIZE** Macro
: The value of this macro is the maximum number of file descriptors that a `fd_set` object can hold information about. On systems with a fixed maximum number, `FD_SETSIZE` is at least that number. On some systems, including GNU, there is no absolute limit on the number of descriptors open, but this macro still has a constant value that controls the number of bits in an `fd_set`; if you get a file descriptor with a value as high as `FD_SETSIZE`, you cannot put that descriptor into an `fd_set`.

`void` **FD_ZERO** (`fd_set *`*set*) Macro

This macro initializes the file-descriptor set *set* to be the empty set.

`void` **FD_SET** (`int` *filedes*, `fd_set *`*set*) Macro

This macro adds *filedes* to the file-descriptor set *set*.

The *filedes* parameter must not have side effects, since it is evaluated more than once.

`void` **FD_CLR** (`int` *filedes*, `fd_set *`*set*) Macro

This macro removes *filedes* from the file-descriptor set *set*.

The *filedes* parameter must not have side effects, since it is evaluated more than once.

`int` **FD_ISSET** (`int` *filedes*, `const fd_set *`*set*) Macro

This macro returns a nonzero value (true) if *filedes* is a member of the file-descriptor set *set*, and 0 (false) otherwise.

The *filedes* parameter must not have side effects, since it is evaluated more than once.

Next, here is the description of the `select` function itself.

`int` **select** (`int` *nfds*, `fd_set *`*read-fds*, `fd_set *`*write-fds*, `fd_set *`*except-fds*, `struct timeval *`*timeout*) Function

The `select` function blocks the calling process until there is activity on any of the specified sets of file descriptors, or until the timeout period has expired.

The file descriptors specified by the *read-fds* argument are checked to see if they are ready for reading; the *write-fds* file descriptors are checked to see if they are ready for writing; and the *except-fds* file-descriptors are checked for exceptional conditions. You can pass a null pointer for any of these arguments if you are not interested in checking for that kind of condition.

A file descriptor is considered ready for reading if it is not at end of file. A server socket is considered ready for reading if there is a pending connection which can be accepted with `accept` (see Section 5.9.3 [Accepting Connections], page 155). A client socket is ready for writing when its connection is fully established (see Section 5.9.1 [Making a Connection], page 153).

"Exceptional conditions" does not mean errors—errors are reported immediately when an erroneous system call is executed, and do not constitute a state of the descriptor. Rather, they include conditions such as the presence of an urgent message on a socket (see Chapter 5 [Sockets], page 125).

The `select` function checks only the first *nfds* file descriptors. The usual thing is to pass `FD_SETSIZE` as the value of this argument.

The *timeout* specifies the maximum time to wait. If you pass a null pointer for this argument, it means to block indefinitely until one of the file descriptors is

ready. Otherwise, you should provide the time in `struct timeval` format.[8] Specify zero as the time (a `struct timeval` containing all zeros) if you want to find out which descriptors are ready without waiting if none are ready.

The normal return value from `select` is the total number of ready file descriptors in all of the sets. Each of the argument sets is overwritten with information about the descriptors that are ready for the corresponding operation. Thus, to see if a particular descriptor *desc* has input, use `FD_ISSET (`*desc*`,` *read-fds*`)` after `select` returns.

If `select` returns because the time-out period expires, it returns a value of 0.

Any signal will cause `select` to return immediately. So if your program uses signals, you can't rely on `select` to keep waiting for the full time specified. If you want to be sure of waiting for a particular amount of time, you must check for `EINTR` and repeat the `select` with a newly calculated time-out based on the current time (see the example below and Section 17.5 [Primitives Interrupted by Signals], page 408).

If an error occurs, `select` returns `-1` and does not modify the argument file-descriptor sets. The following `errno` error conditions are defined for this function:

`EBADF` One of the file-descriptor sets specified an invalid file-descriptor.

`EINTR` The operation was interrupted by a signal (see Section 17.5 [Primitives Interrupted by Signals], page 408).

`EINVAL` The *timeout* argument is invalid; one of the components is negative or too large.

Portability Note: The `select` function is a BSD Unix feature.

Here is an example showing how you can use `select` to establish a time-out period for reading from a file descriptor. The `input_timeout` function blocks the calling process until input is available on the file descriptor, or until the time-out period expires.

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/time.h>



int
input_timeout (int filedes, unsigned int seconds)
{
```

[8] See Loosemore et al., "High-Resolution Calendar" (see chap. 1, n. 1).

```
  fd_set set;
  struct timeval timeout;


  /* Initialize the file-descriptor set. */
  FD_ZERO (&set);
  FD_SET (filedes, &set);

  /* Initialize the timeout data structure. */
  timeout.tv_sec = seconds;
  timeout.tv_usec = 0;


  /* select returns 0 if timeout, 1 if input available, -1 if error. */
  return TEMP_FAILURE_RETRY (select (FD_SETSIZE,
                                     &set, NULL, NULL,
                                     &timeout));
}


int
main (void)
{
  fprintf (stderr, "select returned %d.\n",
           input_timeout (STDIN_FILENO, 5));
  return 0;
}
```

There is another example showing the use of `select` to multiplex input from multiple sockets in Section 5.9.7 [Byte-Stream Connection Server Example], page 161.

2.9 Synchronizing I/O Operations

In most modern operating systems, the normal I/O operations are not executed synchronously; even if a `write` system call returns, this does not mean the data is actually written to the media, e.g., the disk.

In situations where synchronization points are necessary, you can use special functions that ensure that all operations finish before they return.

`int` **sync** (`void`) Function

A call to this function will not return as long as there is data that has not been written to the device. All dirty buffers in the kernel will be written and so an overall consistent system can be achieved (if no other process in parallel writes data).

A prototype for `sync` can be found in '`unistd.h`'.

The return value is 0 to indicate no error.

Programs more often want to ensure that data written to a given file is committed, rather than all data in the system. For this, `sync` is overkill.

`int` **fsync** (`int` *fildes*) Function

The `fsync` function can be used to make sure all data associated with the open file *fildes* is written to the device associated with the descriptor. The function call does not return unless all actions have finished.

A prototype for `fsync` can be found in '`unistd.h`'.

This function is a cancellation point in multithreaded programs. This is a problem if the thread allocates some resources (like memory, file descriptors, semaphores, etc.) at the time `fsync` is called. If the thread gets canceled, these resources stay allocated until the program ends. To avoid this, calls to `fsync` should be protected using cancellation handlers.

The return value of the function is 0 if no error occurred. Otherwise, it is -1 and the global variable *errno* is set to the following values:

`EBADF` The descriptor *fildes* is not valid.

`EINVAL` No synchronization is possible since the system does not implement this.

Sometimes it is not even necessary to write all data associated with a file descriptor. For example, in database files that do not change in size, it is enough to write all the file content data to the device. Metainformation, such as the modification time, is not that important, and leaving such information uncommitted does not prevent a successful recovering of the file in case of a problem.

`int` **fdatasync** (`int` *fildes*) Function

When a call to the `fdatasync` function returns, it is ensured that all of the file data is written to the device. For all pending I/O operations, the parts guaranteeing data integrity finished.

Not all systems implement the `fdatasync` operation. On systems missing this functionality, `fdatasync` is emulated by a call to `fsync` since the performed actions are a superset of those required by `fdatasync`.

The prototype for `fdatasync` is in '`unistd.h`'.

The return value of the function is 0 if no error occurred. Otherwise, it is -1 and the global variable *errno* is set to the following values:

`EBADF` The descriptor *fildes* is not valid.

`EINVAL` No synchronization is possible since the system does not implement this.

2.10 Perform I/O Operations in Parallel

The POSIX.1b standard defines a new set of I/O operations that can significantly reduce the time an application spends waiting at I/O. The new functions allow a program to initiate one or more I/O operations and then immediately resume normal work while the I/O operations are executed in parallel. This functionality is available if the '`unistd.h`' file defines the symbol `_POSIX_ASYNCHRONOUS_IO`.

These functions are part of the library with real-time functions named '`librt`'. They are not actually part of the '`libc`' binary. The implementation of these functions can be done using support in the kernel (if available) or using an implementation based on threads at user level. In the latter case, it might be necessary to link applications with the thread library '`libpthread`' in addition to '`librt`'.

All AIO operations operate on files that were opened previously. There might be an arbitrary number of operations running for one file. The asynchronous I/O operations are controlled using a data structure named `struct aiocb` (AIO *control block*). It is defined in '`aio.h`' as follows:

struct aiocb Data Type

The POSIX.1b standard mandates that the `struct aiocb` structure contain at least the members described in the following table. There might be more elements that are used by the implementation, but depending upon these elements is not portable and is highly deprecated.

`int aio_fildes`
: This element specifies the file descriptor to be used for the operation. It must be a legal descriptor, otherwise the operation will fail.

  The device on which the file is opened must allow the seek operation—it is not possible to use any of the AIO operations on devices like terminals, where an `lseek` call would lead to an error.

`off_t aio_offset`
: This element specifies the offset in the file at which the operation (input or output) is performed. Since the operations are carried out in arbitrary order and more than one operation for one file descriptor can be started, you cannot expect a current read/write position of the file descriptor.

`volatile void *aio_buf`
: This is a pointer to the buffer with the data to be written or the place where the read data is stored.

`size_t aio_nbytes`
: This element specifies the length of the buffer pointed to by `aio_buf`.

`int aio_reqprio`
: If the platform has defined `_POSIX_PRIORITIZED_IO` and `_POSIX_PRIORITY_SCHEDULING`, the AIO requests are processed based on the current scheduling priority. The `aio_reqprio` element can then be used to lower the priority of the AIO operation.

`struct sigevent aio_sigevent`
: This element specifies how the calling process is notified once the operation terminates. If the `sigev_notify` element is `SIGEV_NONE`, no notification is sent. If it is `SIGEV_SIGNAL`, the signal determined by `sigev_signo` is sent. Otherwise, `sigev_notify` must be `SIGEV_THREAD`. In this case, a thread is created that starts executing the function pointed to by `sigev_notify_function`.

`int aio_lio_opcode`
: This element is only used by the `lio_listio` and `lio_listio64` functions. Since these functions allow an arbitrary number of operations to start at once, and each operation can be input or output (or nothing), the information must be stored in the control block. The possible values are

  `LIO_READ`
  : Start a read operation. Read from the file at position `aio_offset` and store the next `aio_nbytes` bytes in the buffer pointed to by `aio_buf`.

  `LIO_WRITE`
  : Start a write operation. Write `aio_nbytes` bytes starting at `aio_buf` into the file starting at position `aio_offset`.

  `LIO_NOP` Do nothing for this control block. This value is useful sometimes when an array of `struct aiocb` values contains holes, i.e., some of the values must not be handled although the whole array is presented to the `lio_listio` function.

  When the sources are compiled using `_FILE_OFFSET_BITS == 64` on a 32-bit machine, this type is in fact `struct aiocb64`, since the LFS interface transparently replaces the `struct aiocb` definition.

For use with the AIO functions defined in the LFS, there is a similar type defined that replaces the types of the appropriate members with larger types but otherwise is equivalent to `struct aiocb`. Particularly, all member names are the same.

struct aiocb64 Data Type

`int aio_fildes`
: This element specifies the file descriptor that is used for the operation. It must be a legal descriptor, since otherwise the operation fails for obvious reasons.

  The device on which the file is opened must allow the seek operation—it is not possible to use any of the AIO operations on devices like terminals, where an `lseek` call would lead to an error.

`off64_t aio_offset`
: This element specifies at which offset in the file the operation (input or output) is performed. Since the operations are carried out in arbitrary order and more than one operation for one file descriptor can be started, you cannot expect a current read/write position of the file descriptor.

`volatile void *aio_buf`
: This is a pointer to the buffer with the data to be written or the place where the read data is stored.

`size_t aio_nbytes`
: This element specifies the length of the buffer pointed to by `aio_buf`.

`int aio_reqprio`
: If for the platform `_POSIX_PRIORITIZED_IO` and `_POSIX_PRIORITY_SCHEDULING` are defined, the AIO requests are processed based on the current scheduling priority. The `aio_reqprio` element can then be used to lower the priority of the AIO operation.

`struct sigevent aio_sigevent`
: This element specifies how the calling process is notified once the operation terminates. If the `sigev_notify` element is `SIGEV_NONE`, no notification is sent. If it is `SIGEV_SIGNAL`, the signal determined by `sigev_signo` is sent. Otherwise, `sigev_notify` must be `SIGEV_THREAD`, in which case a thread that starts executing the function pointed to by `sigev_notify_function`.

`int aio_lio_opcode`
: This element is only used by the `lio_listio` and `[lio_listio64` functions. Since these functions allow an arbitrary number of operations to start at once, and since each operation can be input or output (or nothing), the information must be stored in the control block. See the description of `struct aiocb` for a description of the possible values.

When the sources are compiled using `_FILE_OFFSET_BITS == 64` on a 32-bit machine, this type is available under the name `struct aiocb64`, since the LFS transparently replaces the old interface.

2.10.1 Asynchronous Read and Write Operations

`int` **aio_read** (`struct aiocb` ⋆*aiocbp*) Function

This function initiates an asynchronous read operation. It immediately returns after the operation was enqueued or when an error was encountered.

The first `aiocbp->aio_nbytes` bytes of the file for which `aiocbp->aio_fildes` is a descriptor are written to the buffer starting at `aiocbp->aio_buf`. Reading starts at the absolute position `aiocbp->aio_offset` in the file.

If prioritized I/O is supported by the platform, the `aiocbp->aio_reqprio` value is used to adjust the priority before the request is actually enqueued.

The calling process is notified about the termination of the read request according to the `aiocbp->aio_sigevent` value.

When `aio_read` returns, the return value is 0 if no error occurred that can be found before the process is enqueued. If such an early error is found, the function returns −1 and sets `errno` to one of the following values:

`EAGAIN` The request was not enqueued due to (temporarily) exceeded resource limitations.

`ENOSYS` The `aio_read` function is not implemented.

`EBADF` The `aiocbp->aio_fildes` descriptor is not valid. This condition need not be recognized before enqueueing the request and so this error might also be signaled asynchronously.

`EINVAL` The `aiocbp->aio_offset` or `aiocbp->aio_reqpiro` value is invalid. This condition need not be recognized before enqueueing the request and so this error might also be signaled asynchronously.

If `aio_read` returns 0, the current status of the request can be queried using `aio_error` and `aio_return` functions. As long as the value returned by `aio_error` is `EINPROGRESS`, the operation has not yet completed. If `aio_error` returns 0, the operation successfully terminated, otherwise the value is to be interpreted as an error code. If the function terminated, the result of the operation can be obtained using a call to `aio_return`. The returned value is the same as an equivalent call to `read` would have returned. Possible error codes returned by `aio_error` are

`EBADF` The `aiocbp->aio_fildes` descriptor is not valid.

`ECANCELED` The operation was canceled before the operation was finished (see Section 2.10.4 [Cancellation of AIO Operations], page 52).

`EINVAL` The `aiocbp->aio_offset` value is invalid.

When the sources are compiled with `_FILE_OFFSET_BITS == 64`, this function is in fact `aio_read64`, since the LFS interface transparently replaces the normal implementation.

`int` **aio_read64** (`struct aiocb *`*aiocbp*) Function

This function is similar to the `aio_read` function. The only difference is that on 32-bit machines, the file descriptor should be opened in the large file mode. Internally, `aio_read64` uses functionality equivalent to `lseek64` (see Section 2.3 [Setting the File Position of a Descriptor], page 25) to position the file descriptor correctly for the reading, as opposed to `lseek` functionality used in `aio_read`.

When the sources are compiled with `_FILE_OFFSET_BITS == 64`, this function is available under the name `aio_read` and so transparently replaces the interface for small files on 32-bit machines.

To write data asynchronously to a file, there exists an equivalent pair of functions with a very similar interface.

`int` **aio_write** (`struct aiocb *`*aiocbp*) Function

This function initiates an asynchronous write operation. The function call immediately returns after the operation is enqueued or if, before that happens, an error is encountered.

The first `aiocbp->aio_nbytes` bytes from the buffer starting at `aiocbp->aio_buf` are written to the file for which `aiocbp->aio_fildes` is a descriptor, starting at the absolute position `aiocbp->aio_offset` in the file.

If prioritized I/O is supported by the platform, the `aiocbp->aio_reqprio` value is used to adjust the priority before the request is actually enqueued.

The calling process is notified about the termination of the read request according to the `aiocbp->aio_sigevent` value.

When `aio_write` returns, the return value is 0 if no error occurred that can be found before the process is enqueued. If such an early error is found, the function returns -1 and sets `errno` to one of the following values:

`EAGAIN` The request was not enqueued due to (temporarily) exceeded resource limitations.

`ENOSYS` The `aio_write` function is not implemented.

`EBADF` The `aiocbp->aio_fildes` descriptor is not valid. This condition may not be recognized before enqueueing the request, and so this error might also be signaled asynchronously.

`EINVAL` The `aiocbp->aio_offset` or `aiocbp->aio_reqprio` value is invalid. This condition may not be recognized before enqueueing the request and so this error might also be signaled asynchronously.

When `aio_write` returns 0, the current status of the request can be queried using `aio_error` and `aio_return` functions. As long as the value returned by `aio_error` is `EINPROGRESS`, the operation has not yet completed. If `aio_error` returns 0, the operation successfully terminated, otherwise the value is to be interpreted as an error code. If the function terminated, the result of the operation can be had using a call to `aio_return`. The returned value is the same as an equivalent call to `read` would have returned. Possible error codes returned by `aio_error` are

`EBADF` The `aiocbp->aio_fildes` descriptor is not valid.

`ECANCELED`
The operation was canceled before the operation was finished (see Section 2.10.4 [Cancellation of AIO Operations], page 52).

`EINVAL` The `aiocbp->aio_offset` value is invalid.

When the sources are compiled with `_FILE_OFFSET_BITS == 64`, this function is in fact `aio_write64` since the LFS interface transparently replaces the normal implementation.

`int` **aio_write64** (`struct aiocb *`*aiocbp*) Function

This function is similar to the `aio_write` function. The only difference is that on 32-bit machines, the file descriptor should be opened in the large file mode. Internally, `aio_write64` uses functionality equivalent to `lseek64` (see Section 2.3 [Setting the File Position of a Descriptor], page 25) to position the file descriptor correctly for the writing, as opposed to `lseek` functionality used in `aio_write`.

When the sources are compiled with `_FILE_OFFSET_BITS == 64`, this function is available under the name `aio_write` and so transparently replaces the interface for small files on 32-bit machines.

Besides these functions with the more or less traditional interface, POSIX.1b also defines a function that can initiate more than one operation at a time, and that can handle freely mixed read and write operations. It is therefore similar to a combination of `readv` and `writev`.

`int` **lio_listio** (`int` *mode*`,` `struct aiocb *const` *list*`[],` `int` *nent*`,` `struct sigevent *`*sig*) Function

The `lio_listio` function can be used to enqueue an arbitrary number of read and write requests at one time. The requests can all be meant for the same file, all for different files or every solution in between.

`lio_listio` gets the *nent* requests from the array pointed to by *list*. The operation to be performed is determined by the `aio_lio_opcode` member in each element of *list*. If this field is `LIO_READ`, a read operation is enqueued, similar to a call of `aio_read` for this element of the array (except that the way the termination is signalled is different, as we will see below). If the `aio_lio_opcode` member is `LIO_WRITE`, a write operation is enqueued. Otherwise,

the `aio_lio_opcode` must be `LIO_NOP`, in which case this element of *list* is simply ignored. This "operation" is useful in situations where you have a fixed array of `struct aiocb` elements from which only a few need to be handled at a time. Another situation is where the `lio_listio` call was canceled before all requests are processed (see Section 2.10.4 [Cancellation of AIO Operations], page 52) and the remaining requests have to be reissued.

The other members of each element of the array pointed to by `list` must have values suitable for the operation as described in the documentation for `aio_read` and `aio_write` above.

The *mode* argument determines how `lio_listio` behaves after having enqueued all the requests. If *mode* is `LIO_WAIT`, it waits until all requests are terminated. Otherwise, *mode* must be `LIO_NOWAIT`, and in this case the function returns immediately after having enqueued all the requests and the caller gets a notification of the termination of all requests according to the *sig* parameter. If *sig* is `NULL`, no notification is sent. Otherwise, a signal is sent or a thread is started, just as described in the description for `aio_read` or `aio_write`.

If *mode* is `LIO_WAIT`, the return value of `lio_listio` is 0 when all requests are completed successfully. Otherwise, the function return -1 and `errno` is set accordingly. To find out which request or requests failed, you have to use the `aio_error` function on all the elements of the array *list*.

In case *mode* is `LIO_NOWAIT`, the function returns 0 if all requests were enqueued correctly. The current state of the requests can be found using `aio_error` and `aio_return` as described above. If `lio_listio` returns -1 in this mode, the global variable `errno` is set accordingly. If a request did not yet terminate, a call to `aio_error` returns `EINPROGRESS`. If the value is different, the request is finished and the error value (or 0) is returned, and the result of the operation can be retrieved using `aio_return`.

Possible values for `errno` are

`EAGAIN` The resources necessary to queue all the requests are not available at the moment. The error status for each element of *list* must be checked to determine which request failed.

Another reason could be that the system-wide limit of AIO requests is exceeded. This cannot be the case for the implementation on GNU systems, since no arbitrary limits exist.

`EINVAL` The *mode* parameter is invalid or *nent* is larger than `AIO_LISTIO_MAX`.

`EIO` One or more of the request's I/O operations failed. The error status of each request should be checked to determine which one failed.

`ENOSYS` The `lio_listio` function is not supported.

If the *mode* parameter is `LIO_NOWAIT` and the caller cancels a request, the error status for this request returned by `aio_error` is `ECANCELED`.

When the sources are compiled with `_FILE_OFFSET_BITS == 64`, this function is in fact `lio_listio64`, since the LFS interface transparently replaces the normal implementation.

`int` **lio_listio64** (`int` *mode*, `struct aiocb *const` *list*, `int` *nent*, `struct sigevent *`*sig*) Function

This function is similar to the `lio_listio` function. The only difference is that on 32-bit machines, the file descriptor should be opened in the large file mode. Internally, `lio_listio64` uses functionality equivalent to `lseek64` (see Section 2.3 [Setting the File Position of a Descriptor], page 25) to position the file descriptor correctly for the reading or writing, as opposed to `lseek` functionality used in `lio_listio`.

When the sources are compiled with `_FILE_OFFSET_BITS == 64`, this function is available under the name `lio_listio` and so transparently replaces the interface for small files on 32-bit machines.

2.10.2 Getting the Status of AIO Operations

As already described in the documentation of the functions in the last section, it must be possible to get information about the status of an I/O request. When the operation is performed truly asynchronously (as with `aio_read` and `aio_write` and with `lio_listio` when the mode is `LIO_NOWAIT`), you sometimes need to know whether a specific request already terminated and if so, what the result was. The following two functions allow you to get this kind of information:

`int` **aio_error** (`const struct aiocb *`*aiocbp*) Function

This function determines the error state of the request described by the `struct aiocb` variable pointed to by *aiocbp*. If the request has not yet terminated, the value returned is always `EINPROGRESS`. Once the request has terminated, the value `aio_error` returns is either 0 if the request completed successfully, or the value that would be stored in the `errno` variable if the request would have been done using `read`, `write` or `fsync`.

The function can return `ENOSYS` if it is not implemented. It could also return `EINVAL` if the *aiocbp* parameter does not refer to an asynchronous operation whose return status is not yet known.

When the sources are compiled with `_FILE_OFFSET_BITS == 64`, this function is in fact `aio_error64`, since the LFS interface transparently replaces the normal implementation.

`int` **aio_error64** (`const struct aiocb64 *`*aiocbp*) Function

This function is similar to `aio_error`, with the only difference being that the argument is a reference to a variable of type `struct aiocb64`.

When the sources are compiled with `_FILE_OFFSET_BITS == 64`, this function is available under the name `aio_error` and so transparently replaces the interface for small files on 32-bit machines.

`ssize_t` **aio_return** (`const struct aiocb *`*aiocbp*) Function

This function can be used to retrieve the return status of the operation carried out by the request described in the variable pointed to by *aiocbp*. As long as the error status of this request as returned by `aio_error` is `EINPROGRESS`, the return of this function is undefined.

Once the request is finished, this function can be used exactly once to retrieve the return value. Following calls might lead to undefined behavior. The return value itself is the value that would have been returned by the `read`, `write`, or `fsync` call.

The function can return `ENOSYS` if it is not implemented. It could also return `EINVAL` if the *aiocbp* parameter does not refer to an asynchronous operation whose return status is not yet known.

When the sources are compiled with `_FILE_OFFSET_BITS == 64`, this function is in fact `aio_return64`, since the LFS interface transparently replaces the normal implementation.

`int` **aio_return64** (`const struct aiocb64 *`*aiocbp*) Function

This function is similar to `aio_return`, with the only difference being that the argument is a reference to a variable of type `struct aiocb64`.

When the sources are compiled with `_FILE_OFFSET_BITS == 64`, this function is available under the name `aio_return` and so transparently replaces the interface for small files on 32-bit machines.

2.10.3 Getting into a Consistent State

When dealing with asynchronous operations, it is sometimes necessary to get into a consistent state. This would mean for AIO that you want to know whether a certain request or a group of requests was processed. This could be done by waiting for the notification sent by the system after the operation terminates, but this sometimes would mean wasting resources (mainly computation time). Instead, POSIX.1b defines two functions that will help with most kinds of consistency.

The `aio_fsync` and `aio_fsync64` functions are only available if the symbol `_POSIX_SYNCHRONIZED_IO` is defined in '`unistd.h`'.

`int` **aio_fsync** (`int` *op*, `struct aiocb *`*aiocbp*) Function

Calling this function forces all I/O operations operating queued at the time of the function call operating on the file descriptor `aiocbp->aio_fildes` into the synchronized I/O completion state (see Section 2.9 [Synchronizing I/O Operations], page 40). The `aio_fsync` function returns immediately, but the notification through the method described in `aiocbp->aio_sigevent` will happen only after all requests for this file descriptor have terminated and the file is synchronized. This also means that requests for this very same file-descriptor that are queued after the synchronization request are not affected.

If *op* is `O_DSYNC`, the synchronization happens as with a call to `fdatasync`. Otherwise, *op* should be `O_SYNC`, and the synchronization happens as with `fsync`.

As long as the synchronization has not happened, a call to `aio_error` with the reference to the object pointed to by *aiocbp* returns `EINPROGRESS`. Once the synchronization is done, `aio_error` return 0 if the synchronization was not successful. Otherwise, the value returned is the value to which the `fsync` or `fdatasync` function would have set the `errno` variable. In this case, nothing can be assumed about the consistency for the data written to this file descriptor.

The return value of this function is 0 if the request was successfully enqueued. Otherwise, the return value is -1, and `errno` is set to one of the following values:

`EAGAIN` The request could not be enqueued due to temporary lack of resources.

`EBADF` The file descriptor `aiocbp->aio_fildes` is not valid or not open for writing.

`EINVAL` The implementation does not support I/O synchronization or the *op* parameter is other than `O_DSYNC` and `O_SYNC`.

`ENOSYS` This function is not implemented.

When the sources are compiled with `_FILE_OFFSET_BITS == 64`, this function is in fact `aio_fsync64`, since the LFS interface transparently replaces the normal implementation.

`int` **aio_fsync64** (`int` *op*, `struct aiocb64 *`*aiocbp*) Function

This function is similar to `aio_fsync`, with the only difference being that the argument is a reference to a variable of type `struct aiocb64`.

When the sources are compiled with `_FILE_OFFSET_BITS == 64`, this function is available under the name `aio_fsync` and so transparently replaces the interface for small files on 32-bit machines.

Another method of synchronization is to wait until one or more requests of a specific set terminate. This could be achieved by the `aio_*` functions to notify the initiating process about the termination, but in some situations this is not the ideal solution. In a program that constantly updates clients somehow connected to the server, it is not always the best solution to go round robin since some connections might be slow. On the other hand letting the `aio_*` function notify the caller might not be the best solution either, since whenever the process works on preparing data for one client, it makes no sense for it to be interrupted by a notification because the new client will not be handled before the current client is served. For situations like this, `aio_suspend` should be used.

`int` **aio_suspend** (`const struct aiocb *const` *list*`[], int` *nent*`, const struct timespec *`*timeout*) Function

When calling this function, the calling thread is suspended until at least one of the requests pointed to by the *nent* elements of the array *list* has completed. If any of the requests has already completed at the time `aio_suspend` is called, the function returns immediately. Whether a request has terminated or not is determined by comparing the error status of the request with `EINPROGRESS`. If an element of *list* is `NULL`, the entry is simply ignored.

If no request has finished, the calling process is suspended. If *timeout* is `NULL`, the process is not woken until a request has finished. If *timeout* is not `NULL`, the process remains suspended at least as long as specified in *timeout*. In this case, `aio_suspend` returns with an error.

The return value of the function is 0 if one or more requests from the *list* have terminated. Otherwise, the function returns -1, and `errno` is set to one of the following values:

`EAGAIN` None of the requests from the *list* completed in the time specified by *timeout*.

`EINTR` A signal interrupted the `aio_suspend` function. This signal might also be sent by the AIO implementation while signalling the termination of one of the requests.

`ENOSYS` The `aio_suspend` function is not implemented.

When the sources are compiled with `_FILE_OFFSET_BITS == 64`, this function is in fact `aio_suspend64`, since the LFS interface transparently replaces the normal implementation.

`int` **aio_suspend64** (`const struct aiocb64 *const` *list*`[], int` *nent*`, const struct timespec *`*timeout*) Function

This function is similar to `aio_suspend`, with the only difference being that the argument is a reference to a variable of type `struct aiocb64`.

When the sources are compiled with `_FILE_OFFSET_BITS == 64`, this function is available under the name `aio_suspend` and so transparently replaces the interface for small files on 32-bit machines.

2.10.4 Cancellation of AIO Operations

When one or more requests are asynchronously processed, it might be useful in some situations to cancel a selected operation, e.g., if it becomes obvious that the written data is no longer accurate and would have to be overwritten soon. As an example, assume an application, which writes data in files, in a situation where new incoming data would have to be written in a file that will be updated by an enqueued request. The POSIX AIO implementation provides such a function, but this function is not capable of forcing the cancellation of the request. It is up to the implementation to decide whether it is possible to cancel the operation or not. Therefore, using this function is merely a hint.

`int` **aio_cancel** (`int` *fildes*, `struct aiocb` **aiocbp*) Function

The `aio_cancel` function can be used to cancel one or more outstanding requests. If the *aiocbp* parameter is `NULL`, the function tries to cancel all of the outstanding requests that would process the file descriptor *fildes* (i.e., whose `aio_fildes` member is *fildes*). If *aiocbp* is not `NULL`, `aio_cancel` attempts to cancel the specific request pointed to by *aiocbp*.

For requests that were successfully canceled, the normal notification about the termination of the request should take place; depending on the `struct sigevent` object that controls this, nothing happens, a signal is sent or a thread is started. If the request cannot be canceled, it terminates the usual way after performing the operation.

After a request is successfully canceled, a call to `aio_error` with a reference to this request as the parameter will return `ECANCELED`, and a call to `aio_return` will return -1. If the request wasn't canceled and is still running the error status is still `EINPROGRESS`.

The return value of the function is `AIO_CANCELED` if there were requests that haven't terminated and that were successfully canceled. If there is one or more requests left that couldn't be canceled, the return value is `AIO_NOTCANCELED`. In this case, `aio_error` must be used to find out which of the, perhaps multiple, requests (in *aiocbp* is `NULL`) weren't successfully canceled. If all requests already terminated at the time `aio_cancel` is called, the return value is `AIO_ALLDONE`.

If an error occurred during the execution of `aio_cancel`, the function returns -1, and sets `errno` to one of the following values:

`EBADF` The file descriptor *fildes* is not valid.

`ENOSYS` `aio_cancel` is not implemented.

When the sources are compiled with `_FILE_OFFSET_BITS == 64`, this function is in fact `aio_cancel64`, since the LFS interface transparently replaces the normal implementation.

`int` **aio_cancel64** (`int` *fildes*, `struct aiocb64` **aiocbp*) Function

This function is similar to `aio_cancel`, with the only difference being that the argument is a reference to a variable of type `struct aiocb64`.

When the sources are compiled with `_FILE_OFFSET_BITS == 64`, this function is available under the name `aio_cancel` and so transparently replaces the interface for small files on 32-bit machines.

2.10.5 How to Optimize the AIO Implementation

The POSIX standard does not specify how the AIO functions are implemented. They could be system calls, but it is also possible to emulate them at user level.

At the time of this writing, the available implementation is a user-level implementation that uses threads for handling the enqueued requests. While this implementation requires making some decisions about limitations, hard limitations are

best avoided in the GNU C Library. Therefore, the GNU C Library provides a means for tuning the AIO implementation according to the individual use.

struct aioinit Data Type

This data type is used to pass the configuration or tunable parameters to the implementation. The program has to initialize the members of this struct and pass it to the implementation using the `aio_init` function.

`int aio_threads`
: This member specifies the maximal number of threads that may be used at any one time.

`int aio_num`
: This number provides an estimate on the maximum number of simultaneously enqueued requests.

`int aio_locks`
: This is unused.

`int aio_usedba`
: This is unused.

`int aio_debug`
: This is unused.

`int aio_numusers`
: This is unused.

`int aio_reserved[2]`
: This is unused.

`void` **aio_init** (`const struct aioinit *`*init*) Function

This function must be called before any other AIO function. Calling it is completely voluntary, as it is only meant to help the AIO implementation perform better.

Before calling the `aio_init` function, the members of a variable of type `struct aioinit` must be initialized. Then a reference to this variable is passed as the parameter to `aio_init`, which itself may or may not pay attention to the hints.

The function has no return value, and no error cases are defined. It is an extension that follows a proposal from the SGI implementation in Irix 6. It is not covered by POSIX.1b or Unix98.

2.11 Control Operations on Files

This section describes how you can perform various other operations on file descriptors, such as inquiring about or setting flags describing the status of the file descriptor, and manipulating record locks. All of these operations are performed by the function `fcntl`.

The second argument to the `fcntl` function is a command that specifies which operation to perform. The function and macros that name various flags that are used with it are declared in the header file '`fcntl.h`'. Many of these flags are also used by the `open` function (see Section 2.1 [Opening and Closing Files], page 17).

`int` **fcntl** (`int` *filedes*, `int` *command*, `...`) Function

The `fcntl` function performs the operation specified by *command* on the file descriptor *filedes*. Some commands require additional arguments. These additional arguments, the return value and error conditions are given in the detailed descriptions of the individual commands.

Briefly, here is a list of what the various commands are.

`F_DUPFD` Duplicate the file descriptor—return another file-descriptor pointing to the same open file (see Section 2.12 [Duplicating Descriptors], page 55).

`F_GETFD` Get flags associated with the file descriptor (see Section 2.13 [File-Descriptor Flags], page 57).

`F_SETFD` Set flags associated with the file descriptor (see Section 2.13 [File-Descriptor Flags], page 57).

`F_GETFL` Get flags associated with the open file (see Section 2.14 [File Status Flags], page 59).

`F_SETFL` Set flags associated with the open file (see Section 2.14 [File Status Flags], page 59).

`F_GETLK` Get a file lock (see Section 2.15 [File Locks], page 64).

`F_SETLK` Set or clear a file lock (see Section 2.15 [File Locks], page 64).

`F_SETLKW`
Like `F_SETLK`, but wait for completion (see Section 2.15 [File Locks], page 64).

`F_GETOWN`
Get process or process-group ID to receive `SIGIO` signals (see Section 2.16 [Interrupt-Driven Input], page 68).

`F_SETOWN`
Set process or process-group ID to receive `SIGIO` signals (see Section 2.16 [Interrupt-Driven Input], page 68).

This function is a cancellation point in multithreaded programs. This is a problem if the thread allocates some resources (such as memory, file descriptors or semaphores) at the time `fcntl` is called. If the thread gets canceled, these resources stay allocated until the program ends. To avoid this, calls to `fcntl` should be protected using cancellation handlers.

2.12 Duplicating Descriptors

You can *duplicate* a file descriptor, or allocate another file descriptor that refers to the same open file as the original. Duplicate descriptors share one file position and one set of file status flags (see Section 2.14 [File Status Flags], page 59), but each has its own set of file-descriptor flags (see Section 2.13 [File-Descriptor Flags], page 57).

The major use of duplicating a file descriptor is to implement *redirection* of input or output—to change the file or pipe that a particular file-descriptor corresponds to.

You can perform this operation using the `fcntl` function with the `F_DUPFD` command, but there are also convenient functions `dup` and `dup2` for duplicating descriptors.

The `fcntl` function and flags are declared in '`fcntl.h`', while prototypes for `dup` and `dup2` are in the header file '`unistd.h`'.

`int` **dup** (`int` *old*) Function

This function copies descriptor *old* to the first available descriptor number (the first number not currently open). It is equivalent to `fcntl (`*old*`, F_DUPFD, 0)`.

`int` **dup2** (`int` *old*`, int` *new*) Function

This function copies the descriptor *old* to descriptor number *new*.

If *old* is an invalid descriptor, then `dup2` does nothing; it does not close *new*. Otherwise, the new duplicate of *old* replaces any previous meaning of descriptor *new*, as if *new* were closed first.

If *old* and *new* are different numbers, and *old* is a valid descriptor number, then `dup2` is equivalent to:

```
close (new);
fcntl (old, F_DUPFD, new)
```

However, `dup2` does this atomically; there is no instant in the middle of calling `dup2` at which *new* is closed and not yet a duplicate of *old*.

`int` **F_DUPFD** Macro

This macro is used as the *command* argument to `fcntl`, to copy the file descriptor given as the first argument.

The form of the call in this case is

```
fcntl (old, F_DUPFD, next-filedes)
```

The *next-filedes* argument is of type `int` and specifies that the file descriptor returned should be the next available one greater than or equal to this value.

The return value from `fcntl` with this command is normally the value of the new file-descriptor. A return value of −1 indicates an error. The following `errno` error conditions are defined for this command:

`EBADF` The *old* argument is invalid.

`EINVAL` The *next-filedes* argument is invalid.

`EMFILE` There are no more file descriptors available—your program is already using the maximum. In BSD and GNU, the maximum is controlled by a resource limit that can be changed (see Section 14.2 [Limiting Resource Usage], page 338 for more information about the `RLIMIT_NOFILE` limit).

`ENFILE` is not a possible error code for `dup2` because `dup2` does not create a new opening of a file; duplicate descriptors do not count toward the limit that `ENFILE` indicates. `EMFILE` is possible because it refers to the limit on distinct descriptor numbers in use in one process.

Here is an example showing how to use `dup2` to do redirection. Typically, redirection of the standard streams (like `stdin`) is done by a shell or shell-like program before calling one of the `exec` functions to execute a new program in a child process (see Section 7.5 [Executing a File], page 212). When the new program is executed, it creates and initializes the standard streams to point to the corresponding file-descriptors, before its `main` function is invoked.

So, to redirect standard input to a file, the shell could do something like:

```
pid = fork ();
if (pid == 0)
  {
    char *filename;
    char *program;
    int file;
    ...
    file = TEMP_FAILURE_RETRY (open (filename, O_RDONLY));
    dup2 (file, STDIN_FILENO);
    TEMP_FAILURE_RETRY (close (file));
    execv (program, NULL);
  }
```

There is also a more detailed example showing how to implement redirection in the context of a pipeline of processes in Section 8.6.3 [Launching Jobs], page 228.

2.13 File-Descriptor Flags

file-descriptor flags are miscellaneous attributes of a file descriptor. These flags are associated with particular file descriptors, so that if you have created duplicate file-descriptors from a single opening of a file, each descriptor has its own set of flags.

Currently, there is just one file-descriptor flag: `FD_CLOEXEC`, which causes the descriptor to be closed if you use any of the `exec...` functions (see Section 7.5 [Executing a File], page 212).

The symbols in this section are defined in the header file '`fcntl.h`'.

`int` **F_GETFD** Macro

This macro is used as the *command* argument to `fcntl`, to specify that it should return the file-descriptor flags associated with the *filedes* argument.

The normal return value from `fcntl` with this command is a nonnegative number that can be interpreted as the bit-wise OR of the individual flags (except that currently there is only one flag to use).

In case of an error, `fcntl` returns −1. The following `errno` error conditions are defined for this command:

`EBADF` The *filedes* argument is invalid.

`int` **F_SETFD** Macro

This macro is used as the *command* argument to `fcntl`, to specify that it should set the file-descriptor flags associated with the *filedes* argument. This requires a third `int` argument to specify the new flags, so the form of the call is

```
fcntl (filedes, F_SETFD, new-flags)
```

The normal return value from `fcntl` with this command is an unspecified value other than −1, which indicates an error. The flags and error conditions are the same as for the `F_GETFD` command.

The following macro is defined for use as a file-descriptor flag with the `fcntl` function. The value is an integer constant usable as a bit-mask value.

`int` **FD_CLOEXEC** Macro

This flag specifies that the file descriptor should be closed when an `exec` function is invoked (see Section 7.5 [Executing a File], page 212). When a file descriptor is allocated (as with `open` or `dup`), this bit is initially cleared on the new file-descriptor, meaning that descriptor will survive into the new program after `exec`.

If you want to modify the file-descriptor flags, you should get the current flags with `F_GETFD` and modify the value. Don't assume that the flags listed here are the only ones that are implemented; your program may be run years from now and more flags may exist then. For example, here is a function to set or clear the flag `FD_CLOEXEC` without altering any other flags:

```
/* Set the FD_CLOEXEC flag of desc if value is nonzero,
   or clear the flag if value is 0.
   Return 0 on success, or -1 on error with errno set. */

int
set_cloexec_flag (int desc, int value)
{
  int oldflags = fcntl (desc, F_GETFD, 0);
  /* If reading the flags failed, return error indication now. */
  if (oldflags < 0)
    return oldflags;
```

```
  /* Set just the flag we want to set. */
  if (value != 0)
    oldflags |= FD_CLOEXEC;
  else
    oldflags &= ~FD_CLOEXEC;
  /* Store modified flag word in the descriptor. */
  return fcntl (desc, F_SETFD, oldflags);
}
```

2.14 File Status Flags

File status flags are used to specify attributes of the opening of a file. Unlike the file-descriptor flags (see Section 2.13 [File-Descriptor Flags], page 57), the file status flags are shared by duplicated file-descriptors resulting from a single opening of the file. The file status flags are specified with the *flags* argument to `open` (see Section 2.1 [Opening and Closing Files], page 17).

File status flags fall into three categories, which are described in the following sections.

- *Access modes* (see Section 2.14.1 [File-Access Modes], page 59) specify what type of access is allowed to the file: reading, writing or both. They are set by `open` and are returned by `fcntl`, but cannot be changed.
- *Open-Time flags* (see Section 2.14.2 [Open-Time Flags], page 60) control details of what `open` will do. These flags are not preserved after the `open` call.
- *Operating modes* (see Section 2.14.3 [I/O Operating Modes], page 62) affect how operations such as `read` and `write` are done. They are set by `open` and can be fetched or changed with `fcntl`.

The symbols in this section are defined in the header file '`fcntl.h`'.

2.14.1 File-Access Modes

The file-access modes allow a file descriptor to be used for reading, writing or both. (In the GNU system, they can also allow none of these, and allow execution of the file as a program.) The access modes are chosen when the file is opened, and never change.

`int` **O_RDONLY** Macro
Open the file for read access.

`int` **O_WRONLY** Macro
Open the file for write access.

`int` **O_RDWR** Macro
Open the file for both reading and writing.

In the GNU system (and not in other systems), O_RDONLY and O_WRONLY are independent bits that can be bit-wise-ORed together, and it is valid for either bit to be set or clear. This means that O_RDWR is the same as O_RDONLY|O_WRONLY. A file-access mode of zero is permissible; it allows no operations that do input or output to the file, but does allow other operations such as fchmod. On the GNU system, since "read-only" or "write-only" are misnomers, 'fcntl.h' defines additional names for the file-access modes. These names are preferred when writing GNU-specific code. But most programs will want to be portable to other POSIX.1 systems and should use the POSIX.1 names above instead.

int **O_READ** Macro

Open the file for reading. This is the same as O_RDONLY; it is only defined on GNU.

int **O_WRITE** Macro

Open the file for writing. This is the same as O_WRONLY; it is only defined on GNU.

int **O_EXEC** Macro

Open the file for executing. It is only defined on GNU.

To determine the file-access mode with fcntl, you must extract the access-mode bits from the retrieved file-status flags. In the GNU system, you can just test the O_READ and O_WRITE bits in the flags word. But in other POSIX.1 systems, reading and writing access modes are not stored as distinct bit flags. The portable way to extract the file-access mode bits is with O_ACCMODE.

int **O_ACCMODE** Macro

This macro stands for a mask that can be bit-wise-ANDed with the file-status flag value to produce a value representing the file-access mode. The mode will be O_RDONLY, O_WRONLY or O_RDWR. (In the GNU system, it could also be zero, and it never includes the O_EXEC bit.)

2.14.2 Open-Time Flags

The open-time flags specify options affecting how open will behave. These options are not preserved once the file is open. The exception to this is O_NONBLOCK, which is also an I/O operating mode and so *is* saved (see Section 2.1 [Opening and Closing Files], page 17, for how to call open).

There are two sorts of options specified by open-time flags.

- *File-name translation flags* affect how open looks up the file name to locate the file, and whether the file can be created.
- *Open-time action flags* specify extra operations that open will perform on the file once it is open.

Here are the file-name translation flags:

`int` **O_CREAT** Macro
If set, the file will be created if it doesn't already exist.

`int` **O_EXCL** Macro
If both `O_CREAT` and `O_EXCL` are set, then `open` fails if the specified file already exists. This is guaranteed to never clobber an existing file.

`int` **O_NONBLOCK** Macro
This prevents `open` from blocking for a "long time" to open the file. This is only meaningful for some kinds of files, usually devices such as serial ports; when it is not meaningful, it is harmless and ignored. Often, opening a port to a modem blocks until the modem reports carrier detection; if `O_NONBLOCK` is specified, `open` will return immediately without a carrier.

The `O_NONBLOCK` flag is overloaded as both an I/O operating mode and a file-name translation flag. This means that specifying `O_NONBLOCK` in `open` also sets nonblocking I/O mode (see Section 2.14.3 [I/O Operating Modes], page 62). To open the file without blocking but do normal I/O that blocks, you must call `open` with `O_NONBLOCK` set and then call `fcntl` to turn the bit off.

`int` **O_NOCTTY** Macro
If the named file is a terminal device, don't make it the controlling terminal for the process. (See Chapter 8 [Job Control], page 221, for information about what it means to be the controlling terminal.)

In the GNU system and 4.4 BSD, opening a file never makes it the controlling terminal and `O_NOCTTY` is zero. However, other systems may use a nonzero value for `O_NOCTTY` and set the controlling terminal when you open a file that is a terminal device; so to be portable, use `O_NOCTTY` when it is important to avoid this.

The following three file-name translation flags exist only in the GNU system:

`int` **O_IGNORE_CTTY** Macro
Do not recognize the named file as the controlling terminal, even if it refers to the process's existing controlling terminal device. Operations on the new file-descriptor will never induce job-control signals (see Chapter 8 [Job Control], page 221).

`int` **O_NOLINK** Macro
If the named file is a symbolic link, open the link itself instead of the file it refers to. (`fstat` on the new file-descriptor will return the information returned by `lstat` on the link's name.)

`int` **O_NOTRANS** Macro
If the named file is specially translated, do not invoke the translator. Open the bare file the translator itself sees.

The open-time action flags tell `open` to do additional operations that are not really related to opening the file. The reason to do them as part of `open` instead of in separate calls is that `open` can do them *atomically*.

`int` **O_TRUNC** Macro

Truncate the file to zero length. This option is only useful for regular files, not special files such as directories or FIFOs. POSIX.1 requires that you open the file for writing to use `O_TRUNC`. In BSD and GNU you must have permission to write the file to truncate it, but you need not open for write access.

This is the only open-time action flag specified by POSIX.1. There is no good reason for truncation to be done by `open`, instead of by calling `ftruncate` after. The `O_TRUNC` flag existed in Unix before `ftruncate` was invented, and is retained for backward compatibility.

The remaining operating modes are BSD extensions. They exist only on some systems. On other systems, these macros are not defined.

`int` **O_SHLOCK** Macro

Acquire a shared lock on the file, as with `flock` (see Section 2.15 [File Locks], page 64).

If `O_CREAT` is specified, the locking is done atomically when creating the file. You are guaranteed that no other process will get the lock on the new file first.

`int` **O_EXLOCK** Macro

Acquire an exclusive lock on the file, as with `flock` (see Section 2.15 [File Locks], page 64). This is atomic like `O_SHLOCK`.

2.14.3 I/O Operating Modes

The operating modes affect how input and output operations using a file descriptor work. These flags are set by `open` and can be fetched and changed with `fcntl`.

`int` **O_APPEND** Macro

This is the bit that enables append mode for the file. If set, then all `write` operations write the data at the end of the file, extending it, regardless of the current file position. This is the only reliable way to append to a file. In append mode, you are guaranteed that the data you write will always go to the current end of the file, regardless of other processes writing to the file. Conversely, if you simply set the file position to the end of file and write, then another process can extend the file after you set the file position but before you write, resulting in your data appearing someplace before the real end of file.

`int` **O_NONBLOCK** Macro

This is the bit that enables nonblocking mode for the file. If this bit is set, `read` requests on the file can return immediately with a failure status if there is no

input immediately available, instead of blocking. Likewise, `write` requests can also return immediately with a failure status if the output can't be written immediately.

The `O_NONBLOCK` flag is overloaded as both an I/O operating mode and a file-name translation flag (see Section 2.14.2 [Open-Time Flags], page 60).

`int` **O_NDELAY** Macro

This is an obsolete name for `O_NONBLOCK`, provided for compatibility with BSD. It is not defined by the POSIX.1 standard.

The remaining operating modes are BSD and GNU extensions. They exist only on some systems. On other systems, these macros are not defined.

`int` **O_ASYNC** Macro

This is the bit that enables asynchronous-input mode. If set, then `SIGIO` signals will be generated when input is available (see Section 2.16 [Interrupt-Driven Input], page 68).

Asynchronous-input mode is a BSD feature.

`int` **O_FSYNC** Macro

This is the bit that enables synchronous writing for the file. If set, each `write` call will make sure the data is reliably stored on disk before returning.

Synchronous writing is a BSD feature.

`int` **O_SYNC** Macro

This is another name for `O_FSYNC`. They have the same value.

`int` **O_NOATIME** Macro

If this bit is set, `read` will not update the access time of the file (see Section 3.9.9 [File Times], page 108). This is used by programs that do backups, so that backing a file up does not count as reading it. Only the owner of the file or the superuser may use this bit.

This is a GNU extension.

2.14.4 Getting and Setting File Status Flags

The `fcntl` function can fetch or change file status flags.

`int` **F_GETFL** Macro

This macro is used as the *command* argument to `fcntl`, to read the file status flags for the open file with descriptor *filedes*.

The normal return value from `fcntl` with this command is a nonnegative number that can be interpreted as the bit-wise OR of the individual flags. Since the file-access modes are not single-bit values, you can mask off other bits in the returned flags with `O_ACCMODE` to compare them.

In case of an error, `fcntl` returns −1. The following `errno` error conditions are defined for this command:

`EBADF` The *filedes* argument is invalid.

`int` **F_SETFL** Macro

This macro is used as the *command* argument to `fcntl`, to set the file status flags for the open file corresponding to the *filedes* argument. This command requires a third `int` argument to specify the new flags, so the call looks like this:

```
fcntl (filedes, F_SETFL, new-flags)
```

You can't change the access mode for the file in this way—whether the file descriptor was opened for reading or writing.

The normal return value from `fcntl` with this command is an unspecified value other than −1, which indicates an error. The error conditions are the same as for the `F_GETFL` command.

If you want to modify the file status flags, you should get the current flags with `F_GETFL` and modify the value. Don't assume that the flags listed here are the only ones that are implemented; your program may be run years from now and more flags may exist then. For example, here is a function to set or clear the flag `O_NONBLOCK` without altering any other flags:

```
/* Set the O_NONBLOCK flag of desc if value is nonzero,
   or clear the flag if value is 0.
   Return 0 on success, or -1 on error with errno set. */

int
set_nonblock_flag (int desc, int value)
{
  int oldflags = fcntl (desc, F_GETFL, 0);
  /* If reading the flags failed, return error indication now. */
  if (oldflags == -1)
    return -1;
  /* Set just the flag we want to set. */
  if (value != 0)
    oldflags |= O_NONBLOCK;
  else
    oldflags &= ~O_NONBLOCK;
  /* Store modified flag word in the descriptor. */
  return fcntl (desc, F_SETFL, oldflags);
}
```

2.15 File Locks

The remaining `fcntl` commands are used to support *record locking*, which permits multiple cooperating programs to prevent each other from simultaneously accessing parts of a file in error-prone ways.

An *exclusive* or *write* lock gives a process exclusive access for writing to the specified part of the file. While a write lock is in place, no other process can lock that part of the file.

A *shared* or *read* lock prohibits any other process from requesting a write lock on the specified part of the file. However, other processes can request read locks.

The `read` and `write` functions do not actually check to see whether there are any locks in place. If you want to implement a locking protocol for a file shared by multiple processes, your application must do explicit `fcntl` calls to request and clear locks at the appropriate points.

Locks are associated with processes. A process can only have one kind of lock set for each byte of a given file. When any file descriptor for that file is closed by the process, all of the locks that process holds on that file are released, even if the locks were made using other descriptors that remain open. Likewise, locks are released when a process exits, and are not inherited by child processes created using `fork` (see Section 7.4 [Creating a Process], page 211).

When making a lock, use a `struct flock` to specify what kind of lock and where. This data type and the associated macros for the `fcntl` function are declared in the header file '`fcntl.h`'.

struct flock Data Type

This structure is used with the `fcntl` function to describe a file lock. It has these members:

`short int l_type`
: This specifies the type of the lock; either `F_RDLCK`, `F_WRLCK` or `F_UNLCK`.

`short int l_whence`
: This corresponds to the *whence* argument to `fseek` or `lseek`, and specifies what the offset is relative to. Its value can be one of `SEEK_SET`, `SEEK_CUR` or `SEEK_END`.

`off_t l_start`
: This specifies the offset of the start of the region to which the lock applies and is given in bytes relative to the point specified by the `l_whence` member.

`off_t l_len`
: This specifies the length of the region to be locked. A value of `0` is treated specially; it means the region extends to the end of the file.

`pid_t l_pid`
: This field is the process ID (see Section 7.2 [Process-Creation Concepts], page 210) of the process holding the lock. It is filled in by

calling `fcntl` with the `F_GETLK` command, but is ignored when making a lock.

`int` **F_GETLK** Macro

This macro is used as the *command* argument to `fcntl`, to specify that it should get information about a lock. This command requires a third argument of type `struct flock *` to be passed to `fcntl`, so that the form of the call is

```
fcntl (filedes, F_GETLK, lockp)
```

If there is a lock already in place that would block the lock described by the *lockp* argument, information about that lock overwrites **lockp*. Existing locks are not reported if they are compatible with making a new lock as specified. Thus, you should specify a lock type of `F_WRLCK` if you want to find out about both read and write locks, or `F_RDLCK` if you want to find out about write locks only.

There might be more than one lock affecting the region specified by the *lockp* argument, but `fcntl` only returns information about one of them. The `l_whence` member of the *lockp* structure is set to `SEEK_SET`, and the `l_start` and `l_len` fields are set to identify the locked region.

If no lock applies, the only change to the *lockp* structure is to update the `l_type` to a value of `F_UNLCK`.

The normal return value from `fcntl` with this command is an unspecified value other than -1, which is reserved to indicate an error. The following `errno` error conditions are defined for this command:

`EBADF` The *filedes* argument is invalid.

`EINVAL` Either the *lockp* argument doesn't specify valid lock information, or the file associated with *filedes* doesn't support locks.

`int` **F_SETLK** Macro

This macro is used as the *command* argument to `fcntl`, to specify that it should set or clear a lock. This command requires a third argument of type `struct flock *` to be passed to `fcntl`, so that the form of the call is

```
fcntl (filedes, F_SETLK, lockp)
```

If the process already has a lock on any part of the region, the old lock on that part is replaced with the new lock. You can remove a lock by specifying a lock type of `F_UNLCK`.

If the lock cannot be set, `fcntl` returns immediately with a value of -1. This function does not block waiting for other processes to release locks. If `fcntl` succeeds, it return a value other than -1.

The following `errno` error conditions are defined for this function:

`EAGAIN`
`EACCES` The lock cannot be set because it is blocked by an existing lock on the file. Some systems use `EAGAIN` in this case, and other

systems use `EACCES`; your program should treat them alike, after `F_SETLK`. The GNU system always uses `EAGAIN`.

`EBADF` Either the *filedes* argument is invalid; because you requested a read lock but the *filedes* is not open for read access; or you requested a write lock but the *filedes* is not open for write access.

`EINVAL` Either the *lockp* argument doesn't specify valid lock information, or the file associated with *filedes* doesn't support locks.

`ENOLCK` The system has run out of file-lock resources—there are already too many file locks in place.

Well-designed file systems never report this error, because they have no limitation on the number of locks. However, you must still account for the possibility of this error, as it could result from network access to a file system on another machine.

`int` **F_SETLKW** Macro

This macro is used as the *command* argument to `fcntl`, to specify that it should set or clear a lock. It is just like the `F_SETLK` command, but causes the process to block (or wait) until the request can be specified.

This command requires a third argument of type `struct flock *`, as for the `F_SETLK` command.

The `fcntl` return values and errors are the same as for the `F_SETLK` command, but these additional `errno` error conditions are defined for this command:

`EINTR` The function was interrupted by a signal while it was waiting (see Section 17.5 [Primitives Interrupted by Signals], page 408).

`EDEADLK` The specified region is being locked by another process. But that process is waiting to lock a region that the current process has locked, so waiting for the lock would result in deadlock. The system does not guarantee that it will detect all such conditions, but it lets you know if it notices one.

The following macros are defined for use as values for the `l_type` member of the `flock` structure. The values are integer constants.

`F_RDLCK` This macro is used to specify a read (or shared) lock.

`F_WRLCK` This macro is used to specify a write (or exclusive) lock.

`F_UNLCK` This macro is used to specify that the region is unlocked.

As an example of a situation where file locking is useful, consider a program that can be run simultaneously by several different users, which logs status information to a common file. One example of such a program might be a game that uses a file to keep track of high scores. Another example might be a program that records usage or accounting information for billing purposes.

Having multiple copies of the program simultaneously writing to the file could cause the contents of the file to become mixed up. But you can prevent this kind of problem by setting a write lock on the file before actually writing to the file.

If the program also needs to read the file and wants to make sure that the contents of the file are in a consistent state, then it can also use a read lock. While the read lock is set, no other process can lock that part of the file for writing.

Remember that file locks are only a *voluntary* protocol for controlling access to a file. There is still potential for access to the file by programs that don't use the lock protocol.

2.16 Interrupt-Driven Input

If you set the `O_ASYNC` status flag on a file descriptor (see Section 2.14 [File Status Flags], page 59), a `SIGIO` signal is sent whenever input or output becomes possible on that file descriptor. The process or process group to receive the signal can be selected by using the `F_SETOWN` command to the `fcntl` function. If the file descriptor is a socket, this also selects the recipient of `SIGURG` signals that are delivered when out-of-band data arrives on that socket (see Section 5.9.8 [Out-of-Band Data], page 164). [`SIGURG` is sent in any situation where `select` would report the socket as having an "exceptional condition" (see Section 2.8 [Waiting for Input or Output], page 37).]

If the file descriptor corresponds to a terminal device, then `SIGIO` signals are sent to the foreground process group of the terminal (see Chapter 8 [Job Control], page 221).

The symbols in this section are defined in the header file '`fcntl.h`'.

`int` **F_GETOWN** Macro

This macro is used as the *command* argument to `fcntl`, to specify that it should get information about the process or process group to which `SIGIO` signals are sent. For a terminal, this is actually the foreground process-group ID, which you can get using `tcgetpgrp` (see Section 8.7.3 [Functions for Controlling-Terminal Access], page 241).

The return value is interpreted as a process ID; if negative, its absolute value is the process-group ID.

The following `errno` error condition is defined for this command:

`EBADF` The *filedes* argument is invalid.

`int` **F_SETOWN** Macro

This macro is used as the *command* argument to `fcntl`, to specify that it should set the process or process group to which `SIGIO` signals are sent. This command requires a third argument of type `pid_t` to be passed to `fcntl`, so that the form of the call is

```
fcntl (filedes, F_SETOWN, pid)
```

The *pid* argument should be a process ID. You can also pass a negative number whose absolute value is a process-group ID.

The return value from `fcntl` with this command is −1 in case of error and some other value if successful. The following `errno` error conditions are defined for this command:

`EBADF` The *filedes* argument is invalid.

`ESRCH` There is no process or process group corresponding to *pid*.

2.17 Generic I/O Control Operations

The GNU system can handle most input/output operations on many different devices and objects in terms of a few file primitives: `read`, `write` and `lseek`. However, most devices also have a few peculiar operations that do not fit into this model, such as:

- Changing the character font used on a terminal
- Telling a magnetic tape system to rewind or fast forward (since they cannot move in byte increments, `lseek` is inapplicable).
- Ejecting a disk from a drive
- Playing an audio track from a CD-ROM drive
- Maintaining routing tables for a network

Although some objects such as sockets and terminals have special functions of their own, it would not be practical to create functions for all these cases.[9]

Instead these minor operations, known as *IOCTL*s, are assigned code numbers and multiplexed through the `ioctl` function, defined in `sys/ioctl.h`. The code numbers themselves are defined in many different headers.

`int` **ioctl** (`int` *filedes*`,` `int` *command*`, ...`) Function

The `ioctl` function performs the generic I/O operation *command* on *filedes*.

A third argument is usually present, either a single number or a pointer to a structure. The meaning of this argument, the returned value and any error codes depends upon the command used. Often, −1 is returned for a failure.

On some systems, IOCTLs used by different devices share the same numbers. Thus, although use of an inappropriate IOCTL *usually* only produces an error, you should not attempt to use device-specific IOCTLs on an unknown device.

Most IOCTLs are OS specific and/or are only used in special system utilities, and are thus beyond the scope of this document. For an example of the use of an IOCTL, see Section 5.9.8 [Out-of-Band Data], page 164.

[9] Actually, the terminal-specific functions are implemented with IOCTLs on many platforms.

3 File-System Interface

This chapter describes the GNU C Library's functions for manipulating files. Unlike the input and output functions (see Chapter 2 [Low-Level Input/Output], page 17),[1] these functions are concerned with operating on the files themselves rather than on their contents.

Among the facilities described in this chapter are functions for examining or modifying directories, functions for renaming and deleting files, and functions for examining and setting file attributes such as access permissions and modification times.

3.1 Working Directory

Each process has a directory associated with it, called its *current working directory* or simply *working directory*, which is used in the resolution of relative file names.[2]

When you log in and begin a new session, your working directory is initially set to the home directory associated with your login account in the system user database. You can find any user's home directory using the `getpwuid` or `getpwnam` functions (see Section 10.13 [User Database], page 274).

Users can change the working directory using shell commands like `cd`. The functions described in this section are the primitives used by those commands and by other programs for examining and changing the working directory.

Prototypes for these functions are declared in the header file '`unistd.h`'.

`char *` **getcwd** (`char *`*buffer*`, size_t` *size*) Function

The `getcwd` function returns an absolute file-name representing the current working directory, storing it in the character array *buffer* that you provide. The *size* argument is how you tell the system the allocation size of *buffer*.

The GNU library version of this function also permits you to specify a null pointer for the *buffer* argument. Then `getcwd` allocates a buffer automatically, as with `malloc`.[3] If the *size* is greater than 0, then the buffer is that large; otherwise, the buffer is as large as necessary to hold the result.

The return value is *buffer* on success and a null pointer on failure. The following `errno` error conditions are defined for this function:

`EINVAL` The *size* argument is 0 and *buffer* is not a null pointer.

`ERANGE` The *size* argument is less than the length of the working directory name. You need to allocate a bigger array and try again.

`EACCES` Permission to read or search a component of the file name was denied.

---

[1] See Loosemore et al., "Input/Output on Streams" (see chap. 1, n. 1).

[2] Ibid., "File-Name Resolution".

[3] Ibid., "Unconstrained Allocation".

You could implement the behavior of GNU's `getcwd (NULL, 0)` using only the standard behavior of `getcwd`:

```
char *
gnu_getcwd ()
{
  size_t size = 100;

  while (1)
    {
      char *buffer = (char *) xmalloc (size);
      if (getcwd (buffer, size) == buffer)
        return buffer;
      free (buffer);
      if (errno != ERANGE)
        return 0;
      size *= 2;
    }
}
```

For information about `xmalloc`, which is not a library function but is a customary name used in most GNU software, see Loosemore et al., "Examples of `malloc`" (see chap. 1, n. 1).

`char *` **getwd** (`char *`*buffer*) Deprecated Function

This is similar to `getcwd`, but has no way to specify the size of the buffer. The GNU library provides `getwd` only for backward compatibility with BSD.

The *buffer* argument should be a pointer to an array at least `PATH_MAX` bytes long (see Section 12.6 [Limits on File-System Capacity], page 318). In the GNU system, there is no limit to the size of a file name, so this is not necessarily enough space to contain the directory name. That is why this function is deprecated.

`char *` **get_current_dir_name** (`void`) Function

This `get_current_dir_name` function is basically equivalent to `getcwd (NULL, 0)`. The only difference is that the value of the `PWD` variable is returned if this value is correct. This is a subtle difference that is visible if the path described by the `PWD` value is using one or more symbol links, in which case the value returned by `getcwd` can resolve the symbol links and therefore yield a different result.

This function is a GNU extension.

`int` **chdir** (`const char *`*filename*) Function

This function is used to set the process's working directory to *filename*.

The normal, successful return value from `chdir` is `0`. A value of `-1` is returned to indicate an error. The `errno` error conditions defined for this function are

the usual file-name syntax errors, plus `ENOTDIR` if the file *filename* is not a directory.[4]

`int` **fchdir** (`int` *filedes*) Function

This function is used to set the process's working directory to the directory associated with the file descriptor *filedes*.

The normal, successful return value from `fchdir` is `0`. A value of `-1` is returned to indicate an error. The following `errno` error conditions are defined for this function:

`EACCES` Read permission is denied for the directory named by `dirname`.

`EBADF` The *filedes* argument is not a valid file-descriptor.

`ENOTDIR` The file descriptor *filedes* is not associated with a directory.

`EINTR` The function call was interrupt by a signal.

`EIO` An I/O error occurred.

3.2 Accessing Directories

The facilities described in this section let you read the contents of a directory file. This is useful if you want your program to list all the files in a directory, perhaps as part of a menu.

The `opendir` function opens a *directory stream* whose elements are directory entries. You use the `readdir` function on the directory stream to retrieve these entries, represented as `struct dirent` objects. The name of the file for each entry is stored in the `d_name` member of this structure. There are obvious parallels here to the stream facilities for ordinary files.[5]

3.2.1 Format of a Directory Entry

This section describes what you find in a single directory entry, as you might obtain it from a directory stream. All the symbols are declared in the header file '`dirent.h`'.

struct dirent Data Type

This is a structure type used to return information about directory entries. It contains the following fields:

`char d_name[]`
: This is the null-terminated file-name component. This is the only field you can count on in all POSIX systems.

[4] Ibid., "File-Name Errors".

[5] Ibid., "Input/Output on Streams".

`ino_t d_fileno`
: This is the file serial number. For BSD compatibility, you can also refer to this member as `d_ino`. In the GNU system and most POSIX systems, for most files this the same as the `st_ino` member that `stat` will return for the file (see Section 3.9 [File Attributes], page 93).

`unsigned char d_namlen`
: This is the length of the file name, not including the terminating null character. Its type is `unsigned char` because that is the integer type of the appropriate size.

`unsigned char d_type`
: This is the type of the file, possibly unknown. The following constants are defined for its value:

  `DT_UNKNOWN`
  : The type is unknown. On some systems, this is the only value returned.

  `DT_REG` This is a regular file.

  `DT_DIR` This is a directory.

  `DT_FIFO` This is a named pipe, or FIFO (see Section 4.3 [FIFO Special Files], page 123).

  `DT_SOCK` This is a local-domain socket.

  `DT_CHR` This is a character device.

  `DT_BLK` This is a block device.

  This member is a BSD extension. The symbol `_DIRENT_HAVE_D_TYPE` is defined if this member is available. On systems where it is used, it corresponds to the file-type bits in the `st_mode` member of `struct statbuf`. If the value cannot be determined, the member value is `DT_UNKNOWN`. These two macros convert between `d_type` values and `st_mode` values:

  `int` **IFTODT** (`mode_t` *mode*) Function
  : This returns the `d_type` value corresponding to *mode*.

  `mode_t` **DTTOIF** (`int` *dtype*) Function
  : This returns the `st_mode` value corresponding to *dtype*.

This structure may contain additional members in the future. Their availability is always announced in the compilation environment by a macro named `_DIRENT_HAVE_D_`*xxx* where *xxx* is replaced by the name of the new member.

For instance, the member `d_reclen` available on some systems is announced through the macro `_DIRENT_HAVE_D_RECLEN`.

When a file has multiple names, each name has its own directory entry. The only way you can tell that the directory entries belong to a single file is that they have the same value for the `d_fileno` field.

File attributes such as size, modification times, etc., are part of the file itself, not of any particular directory entry (see Section 3.9 [File Attributes], page 93).

3.2.2 Opening a Directory Stream

This section describes how to open a directory stream. All the symbols are declared in the header file '`dirent.h`'.

DIR Data Type

The `DIR` data type represents a directory stream.

You shouldn't ever allocate objects of the `struct dirent` or `DIR` data types, since the directory-access functions do that for you. Instead, you refer to these objects using the pointers returned by the following functions:

`DIR *` **opendir** (`const char *`*dirname*) Function

The `opendir` function opens and returns a directory stream for reading the directory whose file name is *dirname*. The stream has type `DIR *`.

If unsuccessful, `opendir` returns a null pointer. In addition to the usual file-name errors, the following `errno` error conditions are defined for this function:[6]

`EACCES` Read permission is denied for the directory named by `dirname`.

`EMFILE` The process has too many files open.

`ENFILE` The entire system, or perhaps the file system that contains the directory, cannot support any additional open files at the moment. This problem cannot happen on the GNU system.

The `DIR` type is typically implemented using a file descriptor, and the `opendir` function in terms of the `open` function (see Chapter 2 [Low-Level Input/Output], page 17). Directory streams and the underlying file-descriptors are closed on `exec` (see Section 7.5 [Executing a File], page 212).

In some situations, it can be desirable to get hold of the file descriptor that is created by the `opendir` call. For instance, to switch the current working directory to the directory just read the `fchdir` function could be used. Historically, the `DIR` type was exposed and programs could access the fields. This does not happen in the GNU C Library. Instead, a separate function is provided to allow access.

[6] Ibid., "File-Name Errors".

`int` **dirfd** (`DIR *`*dirstream*) Function

The function `dirfd` returns the file descriptor associated with the directory stream *dirstream*. This descriptor can be used until the directory is closed with `closedir`. If the directory stream implementation is not using file descriptors, the return value is `-1`.

3.2.3 Reading and Closing a Directory Stream

This section describes how to read directory entries from a directory stream, and how to close the stream when you are done with it. All the symbols are declared in the header file '`dirent.h`'.

`struct dirent *` **readdir** (`DIR *`*dirstream*) Function

This function reads the next entry from the directory. It normally returns a pointer to a structure containing information about the file. This structure is statically allocated and can be rewritten by a subsequent call.

Portability Note: On some systems, `readdir` may not return entries for '`.`' and '`..`', even though these are always valid file-names in any directory.[7]

If there are no more entries in the directory or an error is detected, `readdir` returns a null pointer. The following `errno` error condition is defined for this function:

`EBADF` The *dirstream* argument is not valid.

`readdir` is not thread safe. Multiple threads using `readdir` on the same *dirstream* may overwrite the return value. Use `readdir_r` when this is critical.

`int` **readdir_r** (`DIR *`*dirstream*`, struct dirent *`*entry*`, struct dirent **`*result*) Function

This function is the reentrant version of `readdir`. Like `readdir`, it returns the next entry from the directory. But to prevent conflicts between simultaneously-running threads, the result is not stored in statically allocated memory. Instead, the argument *entry* points to a place to store the result.

Normally, `readdir_r` returns 0 and sets `*`*result* to *entry*. If there are no more entries in the directory or an error is detected, `readdir_r` sets `*`*result* to a null pointer and returns a nonzero error code, also stored in `errno`, as described for `readdir`.

Portability Note: On some systems, `readdir_r` may not return a NUL-terminated string for the file name, even when there is no `d_reclen` field in `struct dirent` and the file name is the maximum size allowed. Modern systems all have the `d_reclen` field, and on old systems multithreading is not critical. In any case, there is no such problem with the `readdir` function, so that even on systems without the `d_reclen` member, you could use multiple threads by using external locking.

[7] Ibid., "File-Name Resolution".

It is also important to look at the definition of the `struct dirent` type. Simply passing a pointer to an object of this type for the second parameter of `readdir_r` might not be enough. Some systems don't define the `d_name` element to be sufficiently long. In this case, the user has to provide additional space. There must be room for at least `NAME_MAX + 1` characters in the `d_name` array. Code to call `readdir_r` could look like this:

```
union
{
  struct dirent d;
  char b[offsetof (struct dirent, d_name) + NAME_MAX + 1];
} u;

if (readdir_r (dir, &u.d, &res) == 0)
  ...
```

To support large file-systems on 32-bit machines there are LFS variants of the last two functions.

`struct dirent64 *` **readdir64** (`DIR *`*dirstream*) Function

The `readdir64` function is just like the `readdir` function, except that it returns a pointer to a record of type `struct dirent64`. Some of the members of this data type (notably `d_ino`) might have a different size to allow large file-systems.

In all other aspects, this function is equivalent to `readdir`.

`int` **readdir64_r** (`DIR *`*dirstream*`, struct dirent64 *`*entry*`, struct dirent64 **`*result*) Function

The `readdir64_r` function is equivalent to the `readdir_r` function, except that it takes parameters of base type `struct dirent64` instead of `struct dirent` in the second and third position. The same precautions mentioned in the documentation of `readdir_r` also apply here.

`int` **closedir** (`DIR *`*dirstream*) Function

This function closes the directory stream *dirstream*. It returns `0` on success and `-1` on failure.

The following `errno` error conditions are defined for this function:

`EBADF` The *dirstream* argument is not valid.

3.2.4 Simple Program to List a Directory

Here's a simple program that prints the names of the files in the current working directory:

```
#include <stdio.h>
```

```
#include <sys/types.h>
#include <dirent.h>


int
main (void)
{
  DIR *dp;
  struct dirent *ep;

  dp = opendir ("./");
  if (dp != NULL)
    {
      while (ep = readdir (dp))
        puts (ep->d_name);
      (void) closedir (dp);
    }
  else
    perror ("Couldn't open the directory");

  return 0;
}
```

The order in which files appear in a directory tends to be fairly random (see Section 3.2.6 [Scanning the Content of a Directory], page 79. A more useful program would sort the entries (perhaps by alphabetizing them) before printing them.[8]

3.2.5 Random Access in a Directory Stream

This section describes how to reread parts of a directory that you have already read from an open directory stream. All the symbols are declared in the header file ‘`dirent.h`’.

`void` **rewinddir** (`DIR *`*dirstream*) Function

The `rewinddir` function is used to reinitialize the directory stream *dirstream*, so that if you call `readdir`, it returns information about the first entry in the directory again. This function also notices if files have been added or removed to the directory since it was opened with `opendir`. Entries for these files might or might not be returned by `readdir` if they were added or removed since you last called `opendir` or `rewinddir`.

[8] Ibid., “Array Sort Function”.

`off_t` **telldir** (`DIR *`*dirstream*) Function

The `telldir` function returns the file position of the directory stream *dirstream*. You can use this value with `seekdir` to restore the directory stream to that position.

`void` **seekdir** (`DIR *`*dirstream*`, off_t` *pos*) Function

The `seekdir` function sets the file position of the directory stream *dirstream* to *pos*. The value *pos* must be the result of a previous call to `telldir` on this particular stream; closing and reopening the directory can invalidate values returned by `telldir`.

3.2.6 Scanning the Content of a Directory

A higher-level interface to the directory-handling functions is the `scandir` function. With its help, you can select a subset of the entries in a directory, possibly sort them and get a list of names as the result.

`int` **scandir** (`const char *`*dir*`, struct dirent ***`***`*namelist*`, int (*`*selector*`) (const struct dirent *), int (*`*cmp*`) (const void *, const void *)`) Function

The `scandir` function scans the contents of the directory selected by *dir*. The result in **namelist* is an array of pointers to a structure of type `struct dirent`, which describes all selected directory entries and is allocated using `malloc`. Instead of always getting all directory entries returned, the user-supplied function *selector* can be used to decide which entries are in the result. Only the entries for which *selector* returns a nonzero value are selected.

Finally, the entries in **namelist* are sorted using the user-supplied function *cmp*. The arguments passed to the *cmp* function are of type `struct dirent **`, therefore you cannot directly use the `strcmp` or `strcoll` functions; instead see the functions `alphasort` and `versionsort` below.

The return value of the function is the number of entries placed in **namelist*. If it is `-1`, an error occurred (either the directory could not be opened for reading or the `malloc` call failed), and the global variable `errno` contains more information on the error.

As described above, the fourth argument to the `scandir` function must be a pointer to a sorting function. For the convenience of the programmer, the GNU C Library contains implementations of functions that are very useful for this purpose.

`int` **alphasort** (`const void *`*a*`, const void *`*b*`)` Function

The `alphasort` function behaves like the `strcoll` function.[9] The difference is that the arguments are not string pointers but instead are of type `struct dirent **`.

The return value of `alphasort` is less than, equal to or greater than 0 depending on the order of the two entries *a* and *b*.

[9] Ibid., “String/Array Comparison”.

`int` **versionsort** (`const void *a, const void *b`) Function
The `versionsort` function is like `alphasort`, except that it uses the `strverscmp` function internally.

If the file system supports large files, we cannot use `scandir` anymore, since the `dirent` structure might not be able to contain all the information. The LFS provides the new type `struct dirent64`. To use this, we need a new function.

`int` **scandir64** (`const char *dir, struct dirent64 ***namelist, int (*selector) (const struct dirent64 *), int (*cmp) (const void *, const void *)`) Function
The `scandir64` function works like the `scandir` function, except that the directory entries it returns are described by elements of type `struct dirent64`. The function pointed to by *selector* is again used to select the desired entries, except that *selector* now must point to a function that takes a `struct dirent64 *` parameter.

Similarly, the *cmp* function should expect its two arguments to be of type `struct dirent64 **`.

As *cmp* is now a function of a different type, the functions `alphasort` and `versionsort` cannot be supplied for that argument. Instead, we provide these two replacement functions:

`int` **alphasort64** (`const void *a, const void *b`) Function
The `alphasort64` function behaves like the `strcoll` function.[10] The difference is that the arguments are not string pointers but instead are of type `struct dirent64 **`.

The return value of `alphasort64` is less than, equal to or greater than 0 depending on the order of the two entries *a* and *b*.

`int` **versionsort64** (`const void *a, const void *b`) Function
The `versionsort64` function is like `alphasort64`, except that it uses the `strverscmp` function internally.

It is important not to mix the use of `scandir` and the 64-bit comparison functions or vice versa. There are systems on which this works, but on others it fails miserably.

3.2.7 Simple Program to List a Directory, Mark II

Here is a revised version of the directory lister found in Section 3.2.4 [Simple Program to List a Directory], page 77. Using the `scandir` function we can avoid the functions that work directly with the directory contents. After the call, the returned entries are available for direct use.

[10] Ibid., “String/Array Comparison”.

```
#include <stdio.h>
#include <dirent.h>


static int
one (struct dirent *unused)
{
  return 1;
}

int
main (void)
{
  struct dirent **eps;
  int n;

  n = scandir ("./", &eps, one, alphasort);
  if (n >= 0)
    {
      int cnt;
      for (cnt = 0; cnt < n; ++cnt)
        puts (eps[cnt]->d_name);
    }
  else
    perror ("Couldn't open the directory");

  return 0;
}
```

Note the simple selector function in this example. Since we want to see all directory entries, we always return `1`.

3.3 Working with Directory Trees

The functions described so far for handling the files in a directory have allowed you to either retrieve the information bit by bit, or to process all the files as a group (see `scandir`). Sometimes it is useful to process whole hierarchies of directories and their contained files. The X/Open specification defines two functions to do this. The simpler form is derived from an early definition in System V systems and therefore this function is available on SVID-derived systems. The prototypes and required definitions can be found in the '`ftw.h`' header.

There are four functions in this family: `ftw`, `nftw` and their 64-bit counterparts `ftw64` and `nftw64`. These functions take as one of their arguments a pointer to a callback function of the appropriate type.

__ftw_func_t Data Type

```
int (*) (const char *, const struct stat *, int)
```

This is the type of callback function given to the `ftw` function. The first parameter points to the file name, the second parameter to an object of type `struct stat`, which is filled in for the file named in the first parameter.

The last parameter is a flag giving more information about the current file. It can have the following values:

`FTW_F` The item is either a normal file or a file that does not fit into one of the following categories. This could be special files, sockets, etc.

`FTW_D` The item is a directory.

`FTW_NS` The `stat` call failed, so the information pointed to by the second parameter is invalid.

`FTW_DNR` The item is a directory that cannot be read.

`FTW_SL` The item is a symbolic link. Since symbolic links are normally followed, seeing this value in a `ftw` callback function means the referenced file does not exist. The situation for `nftw` is different.

This value is only available if the program is compiled with `_BSD_SOURCE` or `_XOPEN_EXTENDED` defined before including the first header. The original SVID systems do not have symbolic links.

If the sources are compiled with `_FILE_OFFSET_BITS == 64`, this type is in fact `__ftw64_func_t`, since this mode changes `struct stat` to be `struct stat64`.

For the LFS interface and for use in the function `ftw64`, the header '`ftw.h`' defines another function type.

__ftw64_func_t Data Type

```
int (*) (const char *, const struct stat64 *, int)
```

This type is used just like `__ftw_func_t` for the callback function, but this time is called from `ftw64`. The second parameter to the function is a pointer to a variable of type `struct stat64`, which is able to represent the larger values.

__nftw_func_t Data Type

```
int (*) (const char *, const struct stat *, int, struct FTW *)
```

The first three arguments are the same as for the `__ftw_func_t` type. However for the third argument some additional values are defined to allow finer differentiation:

`FTW_DP` The current item is a directory and all subdirectories have already been visited and reported. This flag is returned instead of `FTW_D` if the `FTW_DEPTH` flag is passed to `nftw` (see below).

`FTW_SLN` The current item is a stale symbolic link. The file it points to does not exist.

The last parameter of the callback function is a pointer to a structure with some extra information as described below.

If the sources are compiled with `_FILE_OFFSET_BITS == 64`, this type is in fact `__nftw64_func_t`, since this mode changes `struct stat` to be `struct stat64`.

For the LFS interface there is also a variant of this data type available, which has to be used with the `nftw64` function.

__nftw64_func_t Data Type

```
int (*) (const char *, const struct stat64 *, int, struct FTW *)
```

This type is used just like `__nftw_func_t` for the callback function, but this time is called from `nftw64`. The second parameter to the function is this time a pointer to a variable of type `struct stat64`, which is able to represent the larger values.

struct FTW Data Type

The information contained in this structure helps in interpreting the name parameter and gives some information about the current state of the traversal of the directory hierarchy.

`int base` The value is the offset into the string passed in the first parameter to the callback function of the beginning of the file name. The rest of the string is the path of the file. This information is especially important if the `FTW_CHDIR` flag was set in calling `nftw`, since then the current directory is the one the current item is found in.

`int level`
While processing, the code tracks how many directories down it has gone to find the current file. This nesting level starts at 0 for files in the initial directory (or is 0 for the initial file if a file was passed).

`int` **ftw** (`const char *`*filename*`, __ftw_func_t` *func*`, int` *descriptors*) Function

The `ftw` function calls the callback function given in the parameter *func* for every item that is found in the directory specified by *filename* and all directories below. The function follows symbolic links if necessary but does not process an item twice. If *filename* is not a directory then it itself is the only object returned to the callback function.

The file name passed to the callback function is constructed by taking the *filename* parameter and appending the names of all passed directories and then the local file-name. So the callback function can use this parameter to access the file. `ftw` also calls `stat` for the file and passes that information on to the callback function. If this `stat` call was not successful the failure is indicated by setting the third argument of the callback function to `FTW_NS`. Otherwise it is set according to the description given in the account of `__ftw_func_t` above.

The callback function is expected to return 0 to indicate that no error occurred and that processing should continue. If an error occurred in the callback function or it wants `ftw` to return immediately, the callback function can return a value other than 0. This is the only correct way to stop the function. The program must not use `setjmp` or similar techniques to continue from another place. This would leave resources allocated by the `ftw` function unfreed.

The *descriptors* parameter to `ftw` specifies how many file descriptors it is allowed to consume. The function runs faster the more descriptors it can use. For each level in the directory hierarchy at most one descriptor is used, but for very deep ones any limit on open file-descriptors for the process or the system may be exceeded. Moreover, file-descriptor limits in a multithreaded program apply to all the threads as a group, and therefore it is a good idea to supply a reasonable limit to the number of open descriptors.

The return value of the `ftw` function is 0 if all callback function calls returned 0 and all actions performed by the `ftw` succeeded. If a function call failed (other than calling `stat` on an item) the function returns -1. If a callback function returns a value other than 0, this value is returned as the return value of `ftw`.

When the sources are compiled with `_FILE_OFFSET_BITS == 64` on a 32-bit system, this function is in fact `ftw64`—the LFS interface transparently replaces the old interface.

`int` **ftw64** (`const char *`*filename*`, __ftw64_func_t` *func*`,` `int` *descriptors*) Function

This function is similar to `ftw` but it can work on file systems with large files. File information is reported using a variable of type `struct stat64`, which is passed by reference to the callback function.

When the sources are compiled with `_FILE_OFFSET_BITS == 64` on a 32-bit system, this function is available under the name `ftw` and transparently replaces the old implementation.

`int` **nftw** (`const char *`*filename*`, __nftw_func_t` *func*`,` `int` *descriptors*`,` `int` *flag*) Function

The `nftw` function works like the `ftw` functions. They call the callback function *func* for all items found in the directory *filename* and below. At most *descriptors* file descriptors are consumed during the `nftw` call.

One difference is that the callback function is of a different type. It is of type `struct FTW *` and provides the callback function with the extra information described above.

A second difference is that `nftw` takes a fourth argument, which is 0 or a bitwise-OR combination of any of the following values:

`FTW_PHYS`
: While traversing the directory, symbolic links are not followed. Instead, symbolic links are reported using the `FTW_SL` value for the type parameter to the callback function. If the file referenced by a symbolic link does not exist, `FTW_SLN` is returned instead.

`FTW_MOUNT`
: The callback function is only called for items that are on the same mounted file-system as the directory given by the *filename* parameter to `nftw`.

`FTW_CHDIR`
: If this flag is given, the current working directory is changed to the directory of the reported object before the callback function is called. When `ntfw` finally returns, the current directory is restored to its original value.

`FTW_DEPTH`
: If this option is specified, all subdirectories and files within them are processed before processing the top directory itself (depth-first processing). This also means the type flag given to the callback function is `FTW_DP` and not `FTW_D`.

The return value is computed in the same way as for `ftw`. `nftw` returns 0 if no failures occurred and all callback functions returned 0. In case of internal errors, such as memory problems, the return value is -1 and *errno* is set accordingly. If the return value of a callback invocation was nonzero, then that value is returned.

When the sources are compiled with `_FILE_OFFSET_BITS == 64` on a 32-bit system, this function is in fact `nftw64`—the LFS interface transparently replaces the old interface.

`int` **nftw64** (`const char *`*filename*`, __nftw64_func_t` *func*`, int` *descriptors*`, int` *flag*) Function

This function is similar to `nftw`, but it can work on file systems with large files. File information is reported using a variable of type `struct stat64`, which is passed by reference to the callback function.

When the sources are compiled with `_FILE_OFFSET_BITS == 64` on a 32-bit system, this function is available under the name `nftw` and transparently replaces the old implementation.

3.4 Hard Links

In POSIX systems, one file can have many names at the same time. All of the names are equally real, and no one of them is preferred to the others.

To add a name to a file, use the `link` function. (The new name is also called a *hard link* to the file.) Creating a new link to a file does not copy the contents of the file; it simply makes a new name by which the file can be known, in addition to the file's existing name or names.

One file can have names in several directories, so the organization of the file system is not a strict hierarchy or tree.

In most implementations, it is not possible to have hard links to the same file in multiple file systems. `link` reports an error if you try to make a hard link to the file from another file system when this cannot be done.

The prototype for the `link` function is declared in the header file '`unistd.h`'.

`int` **link** (`const char *`*oldname*`, const char *`*newname*) Function

The `link` function makes a new link to the existing file named by *oldname*, under the new name *newname*.

This function returns a value of `0` if it is successful and `-1` on failure. In addition to the usual file-name errors for both *oldname* and *newname*, the following `errno` error conditions are defined for this function:[11]

`EACCES` You are not allowed to write to the directory in which the new link is to be written.

`EEXIST` There is already a file named *newname*. If you want to replace this link with a new link, you must remove the old link explicitly first.

`EMLINK` There are already too many links to the file named by *oldname*. The maximum number of links to a file is `LINK_MAX` (see Section 12.6 [Limits on File-System Capacity], page 318).

`ENOENT` The file named by *oldname* doesn't exist. You can't make a link to a file that doesn't exist.

`ENOSPC` The directory or file system that would contain the new link is full and cannot be extended.

`EPERM` In the GNU system and some others, you cannot make links to directories. Many systems allow only privileged users to do so. This error is used to report the problem.

`EROFS` The directory containing the new link can't be modified because it's on a read-only file system.

`EXDEV` The directory specified in *newname* is on a different file system than the existing file.

`EIO` A hardware error occurred while trying to read or write to the file system.

[11] Ibid., "File-Name Errors".

3.5 Symbolic Links

The GNU system supports *soft links* or *symbolic links*. This is a kind of "file" that is essentially a pointer to another file name. Unlike hard links, symbolic links can be made to directories or across file systems with no restrictions. You can also make a symbolic link to a name that is not the name of any file (opening this link will fail until a file by that name is created). Likewise, if the symbolic link points to an existing file that is later deleted, the symbolic link continues to point to the same file name even though the name no longer names any file.

The reason symbolic links work the way they do is that special things happen when you try to open the link. The `open` function realizes you have specified the name of a link, reads the file name contained in the link, and opens that file name instead. The `stat` function likewise operates on the file that the symbolic link points to, instead of on the link itself.

By contrast, other operations, such as deleting or renaming the file, operate on the link itself. The functions `readlink` and `lstat` also refrain from following symbolic links, because their purpose is to obtain information about the link. `link`, the function that makes a hard link, does too. It makes a hard link to the symbolic link, which you rarely want.

Some systems have for some functions operating on files a limit on how many symbolic links are followed when resolving a path name. The limit, if it exists, is published in the '`sys/param.h`' header file.

`int` **MAXSYMLINKS** Macro

The macro `MAXSYMLINKS` specifies how many symlinks some function will follow before returning `ELOOP`. Not all functions behave the same, and this value is not the same as that returned for `_SC_SYMLOOP` by `sysconf`. In fact, the `sysconf` result can indicate that there is no fixed limit even though `MAXSYMLINKS` exists and has a finite value.

Prototypes for most of the functions listed in this section are in '`unistd.h`'.

`int` **symlink** (`const char *`*oldname*`, const char *`*newname*) Function

The `symlink` function makes a symbolic link to *oldname* named *newname*.

The normal return value from `symlink` is `0`. A return value of `-1` indicates an error. In addition to the usual file-name syntax errors, the following `errno` error conditions are defined for this function:[12]

`EEXIST` There is already an existing file named *newname*.

`EROFS` The file *newname* would exist on a read-only file system.

`ENOSPC` The directory or file system cannot be extended to make the new link.

[12] Ibid., "File-Name Errors".

`EIO` A hardware error occurred while reading or writing data on the disk.

`int` **readlink** (`const char *`*filename*`, char *`*buffer*`, size_t` *size*) Function

The `readlink` function gets the value of the symbolic link *filename*. The file name that the link points to is copied into *buffer*. This file-name string is *not* null terminated; `readlink` normally returns the number of characters copied. The *size* argument specifies the maximum number of characters to copy—usually the allocation size of *buffer*.

If the return value equals *size*, you cannot tell whether or not there was room to return the entire name. So make a bigger buffer and call `readlink` again. Here is an example:

```
char *
readlink_malloc (const char *filename)
{
  int size = 100;
  char *buffer = NULL;

  while (1)
    {
      buffer = (char *) xrealloc (buffer, size);
      int nchars = readlink (filename, buffer, size);
      if (nchars < 0)
        {
          free (buffer);
          return NULL;
        }
      if (nchars < size)
        return buffer;
      size *= 2;
    }
}
```

A value of `-1` is returned in case of error. In addition to the usual file-name errors, the following `errno` error conditions are defined for this function:[13]

`EINVAL` The named file is not a symbolic link.

`EIO` A hardware error occurred while reading or writing data on the disk.

In some situations it is desirable to resolve all the symbolic links to get the real name of a file where no prefix names a symbolic link that is followed and no file name in the path is '`.`' or '`..`'. This is desirable if, for example, files have to be compared, in which case different names can refer to the same inode.

[13] Ibid., "File-Name Errors".

`char *` **canonicalize_file_name** (`const char *`*name*) Function

The `canonicalize_file_name` function returns the absolute name of the file named by *name* that contains no '`.`', '`..`' components nor any repeated path separators ('`/`') or symlinks. The result is passed back as the return value of the function in a block of memory allocated with `malloc`. If the result is not used anymore, the memory should be freed with a call to `free`.

If any of the path components except the last one is missing, the function returns a NULL pointer. This is also what is returned if the length of the path reaches or exceeds `PATH_MAX` characters. In any case, `errno` is set accordingly.

`ENAMETOOLONG`
: The resulting path is too long. This error only occurs on systems that have a limit on the file-name length.

`EACCES` At least one of the path components is not readable.

`ENOENT` The input file-name is empty.

`ENOENT` At least one of the path components does not exist.

`ELOOP` More than `MAXSYMLINKS` many symlinks have been followed.

This function is a GNU extension and is declared in '`stdlib.h`'.

The Unix standard includes a similar function that differs from `canonicalize_file_name` in that the user has to provide the buffer where the result is placed in.

`char *` **realpath** (`const char *restrict` *name*`, char *restrict` *resolved*) Function

A call to `realpath` where the *resolved* parameter is `NULL` behaves exactly like `canonicalize_file_name`. The function allocates a buffer for the file name and returns a pointer to it. If *resolved* is not `NULL`, it points to a buffer into which the result is copied. It is the caller's responsibility to allocate a buffer that is large enough. On systems that define `PATH_MAX`, this means the buffer must be large enough for a pathname of this size. For systems without limitations on the pathname length, the requirement cannot be met, and programs should not call `realpath` with anything but `NULL` for the second parameter.

One other difference is that the buffer *resolved* (if nonzero) will contain the part of the path component that does not exist or is not readable if the function returns `NULL` and `errno` is set to `EACCES` or `ENOENT`.

This function is declared in '`stdlib.h`'.

The advantage of using this function is that it is more widely available. The drawback is that it reports failures for long path on systems that have no limits on the file-name length.

3.6 Deleting Files

You can delete a file with `unlink` or `remove`.

Deletion actually deletes a file name. If this is the file's only name, then the file is deleted as well. If the file has other remaining names (see Section 3.4 [Hard Links], page 85), it remains accessible under those names.

`int` **unlink** (`const char *`*filename*) Function

The `unlink` function deletes the file name *filename*. If this is a file's sole name, the file itself is also deleted. If any process has the file open when this happens, deletion is postponed until all processes have closed the file.

The function `unlink` is declared in the header file '`unistd.h`'.

This function returns `0` on successful completion and `-1` on error. In addition to the usual file-name errors, the following `errno` error conditions are defined for this function:[14]

`EACCES` Write permission is denied for the directory from which the file is to be removed, or the directory has the sticky bit set and you do not own the file.

`EBUSY` This error indicates that the file is being used by the system in such a way that it can't be unlinked. For example, you might see this error if the file name specifies the root directory or a mount point for a file system.

`ENOENT` The file name to be deleted doesn't exist.

`EPERM` On some systems, `unlink` cannot be used to delete the name of a directory, or at least can only be used this way by a privileged user. To avoid such problems, use `rmdir` to delete directories. (In the GNU system `unlink` can never delete the name of a directory.)

`EROFS` The directory containing the file name to be deleted is on a read-only file system and can't be modified.

`int` **rmdir** (`const char *`*filename*) Function

The `rmdir` function deletes a directory. The directory must be empty before it can be removed; in other words, it can only contain entries for '`.`' and '`..`'.

In most other respects, `rmdir` behaves like `unlink`. There are two additional `errno` error conditions defined for `rmdir`:

`ENOTEMPTY`
`EEXIST` The directory to be deleted is not empty.

These two error codes are synonymous; some systems use one, and some use the other. The GNU system always uses `ENOTEMPTY`.

The prototype for this function is declared in the header file '`unistd.h`'.

[14] Ibid., "File-Name Errors".

`int` **remove** (`const char *`*filename*) Function

This is the ISO C function to remove a file. It works like `unlink` for files and like `rmdir` for directories. `remove` is declared in '`stdio.h`'.

3.7 Renaming Files

The `rename` function is used to change a file's name.

`int` **rename** (`const char *`*oldname*`, const char *`*newname*) Function

The `rename` function renames the file *oldname* to *newname*. The file formerly accessible under the name *oldname* is afterward accessible as *newname* instead. If the file had any other names aside from *oldname*, it continues to have those names.

The directory containing the name *newname* must be on the same file system as the directory containing the name *oldname*.

One special case for `rename` is when *oldname* and *newname* are two names for the same file. The consistent way to handle this case is to delete *oldname*. However, in this case POSIX requires that `rename` do nothing and report success—which is inconsistent. We don't know what your operating system will do.

If *oldname* is not a directory, then any existing file named *newname* is removed during the renaming operation. However, if *newname* is the name of a directory, `rename` fails in this case.

If *oldname* is a directory, then either *newname* must not exist, or it must name a directory that is empty. In the latter case, the existing directory named *newname* is deleted first. The name *newname* must not specify a subdirectory of the directory `oldname` that is being renamed.

One useful feature of `rename` is that the meaning of *newname* changes "atomically" from any previously existing file by that name to its new meaning (i.e. the file that was called *oldname*). There is no instant at which *newname* is nonexistent "in between" the old meaning and the new meaning. If there is a system crash during the operation, it is possible for both names to still exist; but *newname* will always be intact if it exists at all.

If `rename` fails, it returns `-1`. In addition to the usual file-name errors, the following`errno` error conditions are defined for this function:[15]

`EACCES` One of the directories containing *newname* or *oldname* refuses write permission; or *newname* and *oldname* are directories and write permission is refused for one of them.

`EBUSY` A directory named by *oldname* or *newname* is being used by the system in a way that prevents the renaming from working. This includes directories that are mount points for file systems, and directories that are the current working directories of processes.

[15] Ibid., "File-Name Errors".

`ENOTEMPTY`
`EEXIST` The directory *newname* isn't empty. The `GNU` system always returns `ENOTEMPTY` for this, but some other systems return `EEXIST`.

`EINVAL` *oldname* is a directory that contains *newname*.

`EISDIR` *newname* is a directory but the *oldname* isn't.

`EMLINK` The parent directory of *newname* would have too many links (entries).

`ENOENT` The file *oldname* doesn't exist.

`ENOSPC` The directory that would contain *newname* has no room for another entry, and there is no space left in the file system to expand it.

`EROFS` The operation would involve writing to a directory on a read-only file system.

`EXDEV` The two file names *newname* and *oldname* are on different file systems.

3.8 Creating Directories

Directories are created with the `mkdir` function. (There is also a shell command `mkdir` that does the same thing.)

`int` **mkdir** (`const char *`*filename*`, mode_t` *mode*) Function

The `mkdir` function creates a new, empty directory with name *filename*.

The argument *mode* specifies the file permissions for the new directory file (see Section 3.9.5 [The Mode Bits for Access Permission], page 102).

A return value of `0` indicates successful completion, and `-1` indicates failure. In addition to the usual file-name syntax errors, the following `errno` error conditions are defined for this function:[16]

`EACCES` Write permission is denied for the parent directory in which the new directory is to be added.

`EEXIST` A file named *filename* already exists.

`EMLINK` The parent directory has too many links (entries).

Well-designed file systems never report this error, because they permit more links than your disk could possibly hold. However, you must still take account of the possibility of this error, as it could result from network access to a file system on another machine.

[16] Ibid., "File-Name Errors".

`ENOSPC` The file system doesn't have enough room to create the new directory.

`EROFS` The parent directory of the directory being created is on a read-only file system and cannot be modified.

To use this function, your program should include the header file '`sys/stat.h`'.

3.9 File Attributes

When you issue an '`ls -l`' shell command on a file, it gives you information about the size of the file, who owns it, when it was last modified, etc. These are called the *file attributes*, and are associated with the file itself and not a particular one of its names.

This section contains information about how you can inquire about and modify the attributes of a file.

3.9.1 The Meaning of the File Attributes

When you read the attributes of a file, they come back in a structure called `struct stat`. This section describes the names of the attributes, their data types, and what they mean. For the functions to read the attributes of a file, see Section 3.9.2 [Reading the Attributes of a File], page 97.

The header file '`sys/stat.h`' declares all the symbols defined in this section.

struct stat Data Type

The `stat` structure type is used to return information about the attributes of a file. It contains at least the following members:

`mode_t st_mode`
: This specifies the mode of the file. This includes file-type information (see Section 3.9.3 [Testing the Type of a File], page 99) and the file permission bits (see Section 3.9.5 [The Mode Bits for Access Permission], page 102).

`ino_t st_ino`
: This is the file serial number, which distinguishes this file from all other files on the same device.

`dev_t st_dev`
: This identifies the device containing the file. The `st_ino` and `st_dev`, taken together, uniquely identify the file. The `st_dev` value is not necessarily consistent across reboots or system crashes, however.

`nlink_t st_nlink`
: This is the number of hard links to the file. This count keeps track of how many directories have entries for this file. If the count is ever decremented to 0, then the file itself is discarded as soon as no process still holds it open. Symbolic links are not counted in the total.

`uid_t st_uid`
: This is the user ID of the file's owner (see Section 3.9.4 [File Owner], page 101).

`gid_t st_gid`
: This is the group ID of the file (see Section 3.9.4 [File Owner], page 101).

`off_t st_size`
: This specifies the size of a regular file in bytes. For files that are really devices, this field isn't usually meaningful. For symbolic links, this specifies the length of the file name the link refers to.

`time_t st_atime`
: This is the last access-time for the file (see Section 3.9.9 [File Times], page 108).

`unsigned long int st_atime_usec`
: This is the fractional part of the last access-time for the file (see Section 3.9.9 [File Times], page 108).

`time_t st_mtime`
: This is the time of the last modification to the contents of the file (see Section 3.9.9 [File Times], page 108).

`unsigned long int st_mtime_usec`
: This is the fractional part of the time of the last modification to the contents of the file (see Section 3.9.9 [File Times], page 108).

`time_t st_ctime`
: This is the time of the last modification to the attributes of the file (see Section 3.9.9 [File Times], page 108).

`unsigned long int st_ctime_usec`
: This is the fractional part of the time of the last modification to the attributes of the file (see Section 3.9.9 [File Times], page 108).

`blkcnt_t st_blocks`
: This is the amount of disk space that the file occupies, measured in units of 512-byte blocks.

  The number of disk blocks is not strictly proportional to the size of the file, for two reasons: the file system may use some blocks for internal record keeping; and the file may be sparse—it may have

"holes" that contain zeros but do not actually take up space on the disk.

You can tell (approximately) whether a file is sparse by comparing this value with `st_size`, like this:

```
(st.st_blocks * 512 < st.st_size)
```

This test is not perfect, because a file that is just slightly sparse might not be detected as sparse at all. For practical applications, this is not a problem.

`unsigned int st_blksize`
: The optimal block size for reading or writing this file, in bytes. You might use this size for allocating the buffer space for reading or writing the file. (This is unrelated to `st_blocks`.)

The extensions for the Large File Support (LFS) require, even on 32-bit machines, types that can handle file sizes up to 2^63. Therefore, a new definition of `struct stat` is necessary.

struct stat64 Data Type

The members of this type are the same and have the same names as those in `struct stat`. The only difference is that the members `st_ino`, `st_size` and `st_blocks` have a different type to support larger values.

`mode_t st_mode`
: This specifies the mode of the file. This includes file-type information (see Section 3.9.3 [Testing the Type of a File], page 99) and the file permission bits (see Section 3.9.5 [The Mode Bits for Access Permission], page 102).

`ino64_t st_ino`
: This is the file serial number, which distinguishes this file from all other files on the same device.

`dev_t st_dev`
: This identifies the device containing the file. The `st_ino` and `st_dev`, taken together, uniquely identify the file. The `st_dev` value is not necessarily consistent across reboots or system crashes, however.

`nlink_t st_nlink`
: This is the number of hard links to the file. This count keeps track of how many directories have entries for this file. If the count is ever decremented to 0, then the file itself is discarded as soon as no process still holds it open. Symbolic links are not counted in the total.

`uid_t st_uid`
: This is the user ID of the file's owner (see Section 3.9.4 [File Owner], page 101).

`gid_t st_gid`
: This is the group ID of the file (see Section 3.9.4 [File Owner], page 101).

`off64_t st_size`
: This specifies the size of a regular file in bytes. For files that are really devices, this field isn't usually meaningful. For symbolic links, this specifies the length of the file name the link refers to.

`time_t st_atime`
: This is the last access-time for the file (see Section 3.9.9 [File Times], page 108).

`unsigned long int st_atime_usec`
: This is the fractional part of the last access-time for the file (see Section 3.9.9 [File Times], page 108).

`time_t st_mtime`
: This is the time of the last modification to the contents of the file (see Section 3.9.9 [File Times], page 108).

`unsigned long int st_mtime_usec`
: This is the fractional part of the time of the last modification to the contents of the file (see Section 3.9.9 [File Times], page 108).

`time_t st_ctime`
: This is the time of the last modification to the attributes of the file (see Section 3.9.9 [File Times], page 108).

`unsigned long int st_ctime_usec`
: This is the fractional part of the time of the last modification to the attributes of the file (see Section 3.9.9 [File Times], page 108).

`blkcnt64_t st_blocks`
: This is the amount of disk space that the file occupies, measured in units of 512-byte blocks.

`unsigned int st_blksize`
: This is the optimal block size for reading or writing this file, in bytes. You might use this size for allocating the buffer space for reading or writing the file. It is unrelated to `st_blocks`.

Some of the file attributes have special data-type names that exist specifically for those attributes. They are all aliases for well-known integer types that you know and love. These typedef names are defined in the header file '`sys/types.h`' as well as in '`sys/stat.h`'. Here is a list of them:

mode_t Data Type
This is an integer data type used to represent file modes. In the GNU system, this is equivalent to `unsigned int`.

ino_t Data Type
This is an arithmetic data type used to represent file serial numbers. In Unix jargon, these are sometimes called *inode numbers*. In the GNU system, this type is equivalent to `unsigned long int`.

If the source is compiled with `_FILE_OFFSET_BITS == 64`, this type is transparently replaced by `ino64_t`.

ino64_t Data Type
This is an arithmetic data type used to represent file serial numbers for use in LFS. In the GNU system, this type is equivalent to `unsigned long longint`.

When compiling with `_FILE_OFFSET_BITS == 64`, this type is available under the name `ino_t`.

dev_t Data Type
This is an arithmetic data type used to represent file device numbers. In the GNU system, this is equivalent to `int`.

nlink_t Data Type
This is an arithmetic data type used to represent file link counts. In the GNU system, this is equivalent to `unsigned short int`.

blkcnt_t Data Type
This is an arithmetic data type used to represent block counts. In the GNU system, this is equivalent to `unsigned long int`.

If the source is compiled with `_FILE_OFFSET_BITS == 64`, this type is transparently replaced by `blkcnt64_t`.

blkcnt64_t Data Type
This is an arithmetic data type used to represent block counts for use in LFS. In the GNU system, this is equivalent to `unsigned long long int`.

When compiling with `_FILE_OFFSET_BITS == 64`, this type is available under the name `blkcnt_t`.

3.9.2 Reading the Attributes of a File

To examine the attributes of files, use the functions `stat`, `fstat` and `lstat`. They return the attribute information in a `struct stat` object. All three functions are declared in the header file '`sys/stat.h`'.

`int` **stat** (`const char *`*filename*`, struct stat *`*buf*) Function
The `stat` function returns information about the attributes of the file named by *filename* in the structure pointed to by *buf*.

If *filename* is the name of a symbolic link, the attributes you get describe the file that the link points to. If the link points to a nonexistent file name, then `stat` fails, reporting a nonexistent file.

The return value is `0` if the operation is successful or `-1` on failure. In addition to the usual file-name errors, the following `errno` error condition is defined for this function:[17]

`ENOENT` The file named by *filename* doesn't exist.

When the sources are compiled with `_FILE_OFFSET_BITS == 64`, this function is in fact `stat64`, since the LFS interface transparently replaces the normal implementation.

`int` **stat64** (`const char *`*filename*`, struct stat64 *`*buf*) Function

This function is similar to `stat` but it is also able to work on files larger then 2^31 bytes on 32-bit systems. To be able to do this, the result is stored in a variable of type `struct stat64` to which *buf* must point.

When the sources are compiled with `_FILE_OFFSET_BITS == 64`, this function is available under the name `stat` and so transparently replaces the interface for small files on 32-bit machines.

`int` **fstat** (`int` *filedes*`, struct stat *`*buf*) Function

The `fstat` function is like `stat`, except that it takes an open file-descriptor as an argument instead of a file name (see Chapter 2 [Low-Level Input/Output], page 17).

Like `stat`, `fstat` returns `0` on success and `-1` on failure. The following `errno` error condition is defined for `fstat`:

`EBADF` The *filedes* argument is not a valid file-descriptor.

When the sources are compiled with `_FILE_OFFSET_BITS == 64`, this function is in fact `fstat64`, since the LFS interface transparently replaces the normal implementation.

`int` **fstat64** (`int` *filedes*`, struct stat64 *`*buf*) Function

This function is similar to `fstat` but is able to work on large files on 32-bit platforms. For large files, the file descriptor *filedes* should be obtained by `open64` or `creat64`. The *buf* pointer points to a variable of type `struct stat64` that is able to represent the larger values.

When the sources are compiled with `_FILE_OFFSET_BITS == 64`, this function is available under the name `fstat` and so transparently replaces the interface for small files on 32-bit machines.

`int` **lstat** (`const char *`*filename*`, struct stat *`*buf*) Function

The `lstat` function is like `stat`, except that it does not follow symbolic links. If *filename* is the name of a symbolic link, `lstat` returns information about the link itself. Otherwise, `lstat` works like `stat` (see Section 3.5 [Symbolic Links], page 87).

[17] Ibid., "File-Name Errors".

When the sources are compiled with `_FILE_OFFSET_BITS == 64`, this function is in fact `lstat64`, since the LFS interface transparently replaces the normal implementation.

`int` **lstat64** (`const char *`*filename*`, struct stat64 *`*buf*) Function

This function is similar to `lstat`, but it is also able to work on files larger then 2^31 bytes on 32-bit systems. To be able to do this, the result is stored in a variable of type `struct stat64` to which *buf* must point.

When the sources are compiled with `_FILE_OFFSET_BITS == 64`, this function is available under the name `lstat` and so transparently replaces the interface for small files on 32-bit machines.

3.9.3 Testing the Type of a File

The *file mode*, stored in the `st_mode` field of the file attributes, contains two kinds of information: the file-type code, and the access-permission bits. This section discusses only the type code, which you can use to tell whether the file is a directory, socket, symbolic link, etc. For details about access permissions, see Section 3.9.5 [The Mode Bits for Access Permission], page 102.

There are two ways you can access the file-type information in a file mode. First, for each file type there is a *predicate macro* that examines a given file mode and returns whether it is of that type or not. Second, you can mask out the rest of the file mode to leave just the file-type code, and compare this against constants for each of the supported file-types.

All of the symbols listed in this section are defined in the header file '`sys/stat.h`'.

The following predicate macros test the type of a file, given the value *m*, which is the `st_mode` field returned by `stat` on that file:

`int` **S_ISDIR** (`mode_t` *m*) Macro

This macro returns nonzero if the file is a directory.

`int` **S_ISCHR** (`mode_t` *m*) Macro

This macro returns nonzero if the file is a character special file (a device like a terminal).

`int` **S_ISBLK** (`mode_t` *m*) Macro

This macro returns nonzero if the file is a block special file (a device like a disk).

`int` **S_ISREG** (`mode_t` *m*) Macro

This macro returns nonzero if the file is a regular file.

`int` **S_ISFIFO** (`mode_t` *m*) Macro

This macro returns nonzero if the file is a FIFO special file, or a pipe (see Chapter 4 [Pipes and FIFOs], page 119).

`int` **S_ISLNK** (`mode_t` *m*) Macro
This macro returns nonzero if the file is a symbolic link (see Section 3.5 [Symbolic Links], page 87).

`int` **S_ISSOCK** (`mode_t` *m*) Macro
This macro returns nonzero if the file is a socket (see Chapter 5 [Sockets], page 125).

An alternate non-POSIX method of testing the file type is supported for compatibility with BSD. The mode can be bit-wise ANDed with `S_IFMT` to extract the file-type code, and compared to the appropriate constant. For example:

```
S_ISCHR (mode)
```

is equivalent to:

```
((mode & S_IFMT) == S_IFCHR)
```

`int` **S_IFMT** Macro
This is a bit mask used to extract the file-type code from a mode value.

These are the symbolic names for the different file-type codes:

`S_IFDIR` This is the file-type constant of a directory file.

`S_IFCHR` This is the file-type constant of a character-oriented device file.

`S_IFBLK` This is the file-type constant of a block-oriented device file.

`S_IFREG` This is the file-type constant of a regular file.

`S_IFLNK` This is the file-type constant of a symbolic link.

`S_IFSOCK`
This is the file-type constant of a socket.

`S_IFIFO` This is the file-type constant of a FIFO or pipe.

The POSIX.1b standard introduced a few more objects that can possibly be implemented as objects in the file system. These are message queues, semaphores and shared memory objects. To allow differentiation of these objects from other files, the POSIX standard introduces three new test macros. But unlike the other macros, they do not take the value of the `st_mode` field as the parameter. Instead, they expect a pointer to the whole `struct stat` structure.

`int` **S_TYPEISMQ** (`struct stat *`*s*) Macro
If the system implements POSIX message queues as distinct objects and the file is a message-queue object, this macro returns a nonzero value. In all other cases, the result is 0.

`int` **S_TYPEISSEM** (`struct stat *`*s*) Macro
If the system implements POSIX semaphores as distinct objects and the file is a semaphore object, this macro returns a nonzero value. In all other cases, the result is 0.

`int` **S_TYPEISSHM** (`struct stat *`*s*) Macro

If the system implements POSIX shared memory objects as distinct objects and the file is an shared memory object, this macro returns a nonzero value. In all other cases, the result is 0.

3.9.4 File Owner

Every file has an *owner* that is one of the registered user names defined on the system. Each file also has a *group* that is one of the defined groups. The file owner can often be useful for showing you who edited the file (especially when you edit with GNU Emacs), but its main purpose is for access control.

The file owner and group play a role in determining access because the file has one set of access-permission bits for the owner, another set that applies to users who belong to the file's group, and a third set that applies to everyone else. See Section 3.9.6 [How Your Access to a File is Decided], page 104, for the details of how access is decided based on this data.

When a file is created, its owner is set to the effective user-ID of the process that creates it (see Section 10.2 [The Persona of a Process], page 253). The file's group ID may be set to either the effective group-ID of the process, or the group ID of the directory that contains the file, depending on the system where the file is stored. When you access a remote file system, it behaves according to its own rules, not according to the system your program is running on. Thus, your program must be prepared to encounter either kind of behavior no matter what kind of system you run it on.

You can change the owner and/or group owner of an existing file using the `chown` function. This is the primitive for the `chown` and `chgrp` shell commands.

The prototype for this function is declared in '`unistd.h`'.

`int` **chown** (`const char *`*filename*`, uid_t` *owner*`, gid_t` *group*) Function

The `chown` function changes the owner of the file *filename* to *owner* and its group owner to *group*.

Changing the owner of the file on certain systems clears the set-user-ID and set-group-ID permission bits. This is because those bits may not be appropriate for the new owner. Other file permission bits are not changed.

The return value is `0` on success and `-1` on failure. In addition to the usual file-name errors, the following `errno` error conditions are defined for this function:[18]

`EPERM` This process lacks permission to make the requested change.

Only privileged users or the file's owner can change the file's group. On most file systems, only privileged users can change the file owner; some file systems allow you to change the owner if you

[18] Ibid., "File-Name Errors".

are currently the owner. When you access a remote file system, the behavior you encounter is determined by the system that actually holds the file, not by the system your program is running on.

See Section 12.7 [Optional Features in File Support], page 319, for information about the `_POSIX_CHOWN_RESTRICTED` macro.

`EROFS` The file is on a read-only file system.

`int` **fchown** (`int` *filedes*, `int` *owner*, `int` *group*) Function

This is like `chown`, except that it changes the owner of the open file with descriptor *filedes*.

The return value from `fchown` is `0` on success and `-1` on failure. The following `errno` error codes are defined for this function:

`EBADF` The *filedes* argument is not a valid file-descriptor.

`EINVAL` The *filedes* argument corresponds to a pipe or socket, not an ordinary file.

`EPERM` This process lacks permission to make the requested change. For details, see `chmod` above.

`EROFS` The file resides on a read-only file system.

3.9.5 The Mode Bits for Access Permission

The *file mode*, stored in the `st_mode` field of the file attributes, contains two kinds of information: the file-type code, and the access-permission bits. This section discusses only the access-permission bits, which control who can read or write the file. See Section 3.9.3 [Testing the Type of a File], page 99, for information about the file-type code.

All of the symbols listed in this section are defined in the header file 'sys/stat.h'.

These symbolic constants are defined for the file mode bits that control access permission for the file:

`S_IRUSR`
`S_IREAD` This is the read permission bit for the owner of the file. On many systems, this bit is 0400. `S_IREAD` is an obsolete synonym provided for BSD compatibility.

`S_IWUSR`
`S_IWRITE`
This is the write permission bit for the owner of the file. Usually, it is 0200. `S_IWRITE` is an obsolete synonym provided for BSD compatibility.

`S_IXUSR`
`S_IEXEC` This is the execute (for ordinary files) or search (for directories) permission bit for the owner of the file. It is usually 0100. `S_IEXEC` is an obsolete synonym provided for BSD compatibility.

`S_IRWXU` This is equivalent to '`(S_IRUSR | S_IWUSR | S_IXUSR)`'.

`S_IRGRP` This is the read permission bit for the group owner of the file. Usually, it is 040.

`S_IWGRP` This is the write permission bit for the group owner of the file. Usually, it is 020.

`S_IXGRP` This is the execute or search permission bit for the group owner of the file. Usually, it is 010.

`S_IRWXG` This is equivalent to '`(S_IRGRP | S_IWGRP | S_IXGRP)`'.

`S_IROTH` This is the read permission bit for other users. Usually, it is 04.

`S_IWOTH` This is the write permission bit for other users. Usually, it is 02.

`S_IXOTH` This is the execute or search permission bit for other users. Usually, it is 01.

`S_IRWXO` This is equivalent to '`(S_IROTH | S_IWOTH | S_IXOTH)`'.

`S_ISUID` This is the set-user-ID-on-execute bit. Usually, it is 04000 (see Section 10.4 [How an Application Can Change Persona], page 254).

`S_ISGID` This is the set-group-ID-on-execute bit Usually, it is 02000 (see Section 10.4 [How an Application Can Change Persona], page 254).

`S_ISVTX` This is the *sticky* bit. Usually, it is 01000.

For a directory, it gives permission to delete a file in that directory only if you own that file. Ordinarily, a user can either delete all the files in a directory or cannot delete any of them (based on whether the user has write permission for the directory). The same restriction applies—you must have both write permission for the directory and own the file you want to delete. The one exception is that the owner of the directory can delete any file in the directory, no matter who owns it (provided the owner has given himself write permission for the directory). This is commonly used for the '`/tmp`' directory, where anyone may create files but not delete files created by other users.

Originally, the sticky bit on an executable file modified the swapping policies of the system. Normally, when a program terminated, its pages in core were immediately freed and reused. If the sticky bit was set on the executable file, the system kept the pages in core for a while as if the program were still running. This was advantageous for a program likely to be run many times in succession. This usage is obsolete in modern systems. When a program terminates, its pages always remain in core as long as there is no shortage of memory in the system. When the program is next run, its pages will still be in core if no shortage arose since the last run.

On some modern systems where the sticky bit has no useful meaning for an executable file, you cannot set the bit at all for a nondirectory.

If you try, `chmod` fails with `EFTYPE` (see Section 3.9.7 [Assigning File Permissions], page 104).

Some systems (particularly SunOS) have yet another use for the sticky bit. If the sticky bit is set on a file that is *not* executable, it means the opposite—never cache the pages of this file at all. The main use of this is for the files on an NFS server machine that are used as the swap area of diskless client machines. The idea is that the pages of the file will be cached in the client's memory, so it is a waste of the server's memory to cache them a second time. With this usage, the sticky bit also implies that the file system may fail to record the file's modification time onto disk reliably (the idea being that no one cares for a swap file).

This bit is only available on BSD systems (and those derived from them). Therefore, you have to use the `_BSD_SOURCE` feature-select macro to get the definition (see Section 1.3.4 [Feature-Test Macros], page 8).

The actual bit values of the symbols are listed in the table above so you can decode file mode values when debugging your programs. These bit values are correct for most systems, but they are not guaranteed.

Warning: Writing explicit numbers for file permissions is bad practice. Not only is it not portable, it also requires everyone who reads your program to remember what the bits mean. To make your program clean use the symbolic names.

3.9.6 How Your Access to a File is Decided

Recall that the operating system normally decides access permission for a file based on the effective user and group IDs of the process and its supplementary group-IDs, together with the file's owner, group and permission bits. These concepts are discussed in detail in Section 10.2 [The Persona of a Process], page 253.

If the effective user-ID of the process matches the owner user-ID of the file, then permissions for read, write and execute/search are controlled by the corresponding "user" (or "owner") bits. Likewise, if any of the effective group-ID or supplementary group-IDs of the process matches the group owner ID of the file, then permissions are controlled by the "group" bits. Otherwise, permissions are controlled by the "other" bits.

Privileged users, like '`root`', can access any file regardless of its permission bits. As a special case, for a file to be executable even by a privileged user, at least one of its execute bits must be set.

3.9.7 Assigning File Permissions

The primitive functions for creating files (for example, `open` or `mkdir`) take a *mode* argument, which specifies the file permissions to give the newly created file. This mode is modified by the process's *file creation mask*, or *umask*, before it is used.

The bits that are set in the file-creation mask identify permissions that are always to be disabled for newly created files. For example, if you set all the "other" access bits in the mask, then newly created files are not accessible at all to processes in the "other" category, even if the *mode* argument passed to the create function would permit such access. In other words, the file-creation mask is the complement of the ordinary access-permissions you want to grant.

Programs that create files typically specify a *mode* argument that includes all the permissions that make sense for the particular file. For an ordinary file, this is typically read and write permission for all classes of users. These permissions are then restricted as specified by the individual user's own file-creation mask.

To change the permission of an existing file given its name, call `chmod`. This function uses the specified permission bits and ignores the file creation mask.

In normal use, the file-creation mask is initialized by the user's login shell (using the `umask` shell command), and inherited by all subprocesses. Application programs normally don't need to worry about the file-creation mask. It will automatically do what it is supposed to do.

When your program needs to create a file and bypass the umask for its access-permissions, the easiest way to do this is to use `fchmod` after opening the file, rather than changing the umask. In fact, changing the umask is usually done only by shells. They use the `umask` function.

The functions in this section are declared in '`sys/stat.h`'.

`mode_t` **umask** (`mode_t` *mask*) Function

The `umask` function sets the file-creation mask of the current process to *mask*, and returns the previous value of the file-creation mask.

Here is an example showing how to read the mask with `umask` without changing it permanently:

```
mode_t
read_umask (void)
{
  mode_t mask = umask (0);
  umask (mask);
  return mask;
}
```

However, it is better to use `getumask` if you just want to read the mask value, because it is reentrant (at least if you use the GNU operating system).

`mode_t` **getumask** (`void`) Function

Return the current value of the file-creation mask for the current process. This function is a GNU extension.

`int` **chmod** (`const char *`*filename*`, mode_t` *mode*) Function

The `chmod` function sets the access-permission bits for the file named by *filename* to *mode*.

If *filename* is a symbolic link, `chmod` changes the permissions of the file pointed to by the link, not those of the link itself.

This function returns `0` if successful and `-1` if not. In addition to the usual file-name errors, the following `errno` error conditions are defined for this function:[19]

`ENOENT` The named file doesn't exist.

`EPERM` This process does not have permission to change the access permissions of this file. Only the file's owner (as judged by the effective user ID of the process) or a privileged user can change them.

`EROFS` The file resides on a read-only file system.

`EFTYPE` *mode* has the `S_ISVTX` bit (the *sticky bit*) set, and the named file is not a directory. Some systems do not allow setting the sticky bit on nondirectory files, and some do (and only some of those assign a useful meaning to the bit for nondirectory files).

You only get `EFTYPE` on systems where the sticky bit has no useful meaning for nondirectory files, so it is always safe to just clear the bit in *mode* and call `chmod` again. See Section 3.9.5 [The Mode Bits for Access Permission], page 102, for full details on the sticky bit.

`int` **fchmod** (`int` *filedes*, `int` *mode*) Function

This is like `chmod`, except that it changes the permissions of the currently open file given by *filedes*.

The return value from `fchmod` is `0` on success and `-1` on failure. The following `errno` error codes are defined for this function:

`EBADF` The *filedes* argument is not a valid file-descriptor.

`EINVAL` The *filedes* argument corresponds to a pipe or socket, or something else that doesn't really have access permissions.

`EPERM` This process does not have permission to change the access permissions of this file. Only the file's owner (as judged by the effective user-ID of the process) or a privileged user can change them.

`EROFS` The file resides on a read-only file system.

3.9.8 Testing Permission to Access a File

In some situations, it is desirable to allow programs to access files or devices even if this is not possible with the permissions granted to the user. One possible solution is to set the setuid-bit of the program file. If such a program is started, the *effective* user ID of the process is changed to that of the owner of the program

[19] Ibid., "File-Name Errors".

file. So to allow write access to files like '`/etc/passwd`', which normally can be written only by the superuser, the modifying program will have to be owned by `root` and the setuid-bit set.

But besides the files the program is intended to change, the user should not be allowed to access any file to which he would not have access anyway. The program therefore must explicitly check whether *the user* would have the necessary access to a file, before it reads or writes the file.

To do this, use the function `access`, which checks for access permission based on the process's *real* user-ID rather than the effective user-ID. (The setuid feature does not alter the real user-ID, so it reflects the user who actually ran the program.)

There is another way you could check this access, which is easy to describe, but very hard to use. This is to examine the file mode bits and mimic the system's own access computation. This method is undesirable, because many systems have additional access-control features; your program cannot portably mimic them, and you would not want to try to keep track of the diverse features that different systems have. Using `access` is simple and automatically does whatever is appropriate for the system you are using.

`access` is only appropriate to use in setuid programs. A non-setuid program will always use the effective ID rather than the real ID.

The symbols in this section are declared in '`unistd.h`'.

`int` **access** (`const char *`*filename*`, int` *how*) Function

The `access` function checks to see whether the file named by *filename* can be accessed in the way specified by the *how* argument. The *how* argument either can be the bit-wise OR of the flags `R_OK`, `W_OK`, `X_OK` or the existence test `F_OK`.

This function uses the *real* user and group-IDs of the calling process, rather than the *effective* IDs, to check for access permission. As a result, if you use the function from a `setuid` or `setgid` program (see Section 10.4 [How an Application Can Change Persona], page 254), it gives information relative to the user who actually ran the program.

The return value is `0` if the access is permitted and `-1` otherwise. (In other words, treated as a predicate function, `access` returns true if the requested access is *denied*.)

In addition to the usual file-name errors, the following `errno` error conditions are defined for this function:[20]

`EACCES` The access specified by *how* is denied.

`ENOENT` The file doesn't exist.

`EROFS` Write permission was requested for a file on a read-only file system.

[20] Ibid., "File-Name Errors".

These macros are defined in the header file 'unistd.h' for use as the *how* argument to the access function. The values are integer constants.

int **R_OK** Macro
This is a flag meaning test for read permission.

int **W_OK** Macro
This is a flag meaning test for write permission.

int **X_OK** Macro
This is a flag meaning test for execute/search permission.

int **F_OK** Macro
This is a flag meaning test for existence of the file.

3.9.9 File Times

Each file has three time stamps associated with it: its access time, its modification time and its attribute-modification time. These correspond to the st_atime, st_mtime, and st_ctime members of the stat structure (see Section 3.9 [File Attributes], page 93).

All of these times are represented in calendar-time format, as time_t objects. This data type is defined in 'time.h'.[21]

Reading from a file updates its access-time attribute, and writing updates its modification time. When a file is created, all three time stamps for that file are set to the current time. In addition, the attribute-change-time and modification-time fields of the directory that contains the new entry are updated.

Adding a new name for a file with the link function updates the attribute-change-time field of the file being linked, and both the attribute-change-time and modification-time fields of the directory containing the new name. These same fields are affected if a file name is deleted with unlink, remove or rmdir. Renaming a file with rename affects only the attribute-change-time and modification-time fields of the two parent directories involved, and not the times for the file being renamed.

Changing the attributes of a file (for example, with chmod) updates its attribute-change-time field.

You can also change some of the time stamps of a file explicitly using the utime function—all except the attribute-change time. You need to include the header file 'utime.h' to use this facility.

struct utimbuf Data Type
The utimbuf structure is used with the utime function to specify new access and modification times for a file. It contains the following members:

[21] Ibid., "Calendar Time".

`time_t actime`
: This is the access time for the file.

`time_t modtime`
: This is the modification time for the file.

`int` **utime** (`const char *`*filename*`, const struct utimbuf *`*times*) — Function

This function is used to modify the file times associated with the file named *filename*.

If *times* is a null pointer, then the access and modification times of the file are set to the current time. Otherwise, they are set to the values from the `actime` and `modtime` members (respectively) of the `utimbuf` structure pointed to by *times*.

The attribute-modification time for the file is set to the current time in either case (since changing the time stamps is itself a modification of the file attributes).

The `utime` function returns `0` if successful and `-1` on failure. In addition to the usual file-name errors, the following `errno` error conditions are defined for this function:[22]

`EACCES`
: There is a permission problem in the case where a null pointer was passed as the *times* argument. In order to update the time stamp on the file, you must either be the owner of the file, have write permission for the file or be a privileged user.

`ENOENT`
: The file doesn't exist.

`EPERM`
: If the *times* argument is not a null pointer, you must either be the owner of the file or be a privileged user.

`EROFS`
: The file lives on a read-only file system.

Each of the three time stamps has a corresponding microsecond part, which extends its resolution. These fields are called `st_atime_usec`, `st_mtime_usec` and `st_ctime_usec`; each has a value between 0 and 999,999, which indicates the time in microseconds. They correspond to the `tv_usec` field of a `timeval` structure.[23]

The `utimes` function is like `utime`, but also lets you specify the fractional part of the file times. The prototype for this function is in the header file '`sys/time.h`'.

`int` **utimes** (`const char *`*filename*`, struct timeval` *tvp*`[2])` — Function

This function sets the file access and modification times of the file *filename*. The new file access-time is specified by *tvp*`[0]` and the new modification time

[22] Ibid., "File-Name Errors".

[23] Ibid., "High-Resolution Calendar"

by *tvp*[1]. Similar to utime, if *tvp* is a null pointer, then the access and modification times of the file are set to the current time. This function comes from BSD.

The return values and error conditions are the same as for the utime function.

`int` **lutimes** (`const char *`*filename*`, struct timeval` *tvp*`[2]`) Function

This function is like utimes, except that it does not follow symbolic links. If *filename* is the name of a symbolic link, lutimes sets the file access and modification times of the symbolic link special file itself, as seen by lstat (see Section 3.5 [Symbolic Links], page 87), while utimes sets the file access and modification times of the file the symbolic link refers to. This function comes from FreeBSD and is not available on all platforms (if not available, it will fail with ENOSYS).

The return values and error conditions are the same as for the utime function.

`int` **futimes** (`int *`*fd*`, struct timeval` *tvp*`[2]`) Function

This function is like utimes, except that it takes an open file descriptor as an argument instead of a file name (see Chapter 2 [Low-Level Input/Output], page 17). This function comes from FreeBSD and is not available on all platforms (if not available, it will fail with ENOSYS).

Like utimes, futimes returns 0 on success and −1 on failure. The following errno error conditions are defined for futimes:

EACCES There is a permission problem in the case where a null pointer was passed as the *times* argument. In order to update the time stamp on the file, you must either be the owner of the file, have write permission for the file or be a privileged user.

EBADF The *filedes* argument is not a valid file-descriptor.

EPERM If the *times* argument is not a null pointer, you must either be the owner of the file or be a privileged user.

EROFS The file lives on a read-only file system.

3.9.10 File Size

Normally, file sizes are maintained automatically. A file begins with a size of 0 and is automatically extended when data is written past its end. It is also possible to empty a file completely by an open or fopen call.

However, sometimes it is necessary to *reduce* the size of a file. This can be done with the truncate and ftruncate functions. They were introduced in BSD Unix. ftruncate was later added to POSIX.1.

Some systems allow you to extend a file (creating holes) with these functions. This is useful when using memory-mapped I/O (see Section 2.7 [Memory-Mapped I/O], page 32), where files are not automatically extended. However it is not

portable, but must be implemented if `mmap` allows mapping of files (i.e., `_POSIX_MAPPED_FILES` is defined).

Using these functions on anything other than a regular file gives *undefined* results. On many systems, such a call will appear to succeed, without actually accomplishing anything.

`int` **truncate** (`const char *`*filename*`, off_t` *length*) Function

The `truncate` function changes the size of *filename* to *length*. If *length* is shorter than the previous length, data at the end will be lost. The file must be writable by the user to perform this operation.

If *length* is longer, holes will be added to the end. However, some systems do not support this feature and will leave the file unchanged.

When the source file is compiled with `_FILE_OFFSET_BITS == 64`, the `truncate` function is in fact `truncate64`, and the type `off_t` has 64 bits, which makes it possible to handle files up to 2^63 bytes in length.

The return value is 0 for success and -1 for an error. In addition to the usual file-name errors, the following errors may occur:

`EACCES` The file is a directory or not writable.

`EINVAL` *length* is negative.

`EFBIG` The operation would extend the file beyond the limits of the operating system.

`EIO` A hardware I/O error occurred.

`EPERM` The file is "append-only" or "immutable".

`EINTR` The operation was interrupted by a signal.

`int` **truncate64** (`const char *`*name*`, off64_t` *length*) Function

This function is similar to the `truncate` function. The difference is that the *length* argument is 64 bits wide even on 32-bit machines, which allows the handling of files with sizes up to 2^63 bytes.

When the source file is compiled with `_FILE_OFFSET_BITS == 64` on a 32-bit machine, this function is actually available under the name `truncate` and so transparently replaces the 32-bit interface.

`int` **ftruncate** (`int` *fd*`, off_t` *length*) Function

This is like `truncate`, but it works on a file descriptor *fd* for an opened file instead of a file name to identify the object. The file must be opened for writing to successfully carry out the operation.

The POSIX standard leaves implementation defined what happens if the specified new *length* of the file is bigger than the original size. The `ftruncate` function might simply leave the file alone and do nothing, or it can increase the size to the desired size. In this later case, the extended area should be zero filled. So using `ftruncate` is not a reliable way to increase the file size, but if it is possible, it

is probably the fastest way. The function also operates on POSIX shared memory segments if these are implemented by the system.

`ftruncate` is especially useful in combination with `mmap`. Since the mapped region must have a fixed size, you cannot enlarge the file by writing something beyond the last mapped page. Instead you have to enlarge the file itself and then remap the file with the new size. The example below shows how this works.

When the source file is compiled with `_FILE_OFFSET_BITS == 64`, the `ftruncate` function is in fact `ftruncate64`, and the type `off_t` has 64 bits, which makes it possible to handle files up to 2^63 bytes in length.

The return value is 0 for success and -1 for an error. The following errors may occur:

`EBADF` *fd* does not correspond to an open file.

`EACCES` *fd* is a directory or not open for writing.

`EINVAL` *length* is negative.

`EFBIG` The operation would extend the file beyond the limits of the operating system.

`EIO` A hardware I/O error occurred.

`EPERM` The file is "append-only" or "immutable".

`EINTR` The operation was interrupted by a signal.

`int` **ftruncate64** (`int` *id*, `off64_t` *length*) Function

This function is similar to the `ftruncate` function. The difference is that the *length* argument is 64 bits wide even on 32-bit machines, which allows the handling of files with sizes up to 2^63 bytes.

When the source file is compiled with `_FILE_OFFSET_BITS == 64` on a 32-bit machine, this function is actually available under the name `ftruncate` and so transparently replaces the 32-bit interface.

Here is a little example of how to use `ftruncate` in combination with `mmap`:

```
int fd;
void *start;
size_t len;

int
add (off_t at, void *block, size_t size)
{
  if (at + size > len)
    {
      /* Resize the file and remap.  */
      size_t ps = sysconf (_SC_PAGESIZE);
      size_t ns = (at + size + ps - 1) & ~(ps - 1);
```

```
          void *np;
          if (ftruncate (fd, ns) < 0)
            return -1;
          np = mmap (NULL, ns, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
          if (np == MAP_FAILED)
            return -1;
          start = np;
          len = ns;
        }
      memcpy ((char *) start + at, block, size);
      return 0;
    }
```

The function `add` writes a block of memory at an arbitrary position in the file. If the current size of the file is too small, it is extended. It is extended by a round number of pages. This is a requirement of `mmap`. The program has to keep track of the real size, and when it has finished, a final `ftruncate` call should set the real size of the file.

3.10 Making Special Files

The `mknod` function is the primitive for making special files, such as files that correspond to devices. The GNU library includes this function for compatibility with BSD.

The prototype for `mknod` is declared in '`sys/stat.h`'.

`int` **mknod** (`const char *`*filename*`, int` *mode*`, int` *dev*) Function

The `mknod` function makes a special file with name *filename*. The *mode* specifies the mode of the file, and may include the various special-file bits, such as `S_IFCHR` (for a character special file) or `S_IFBLK` (for a block special file) (see Section 3.9.3 [Testing the Type of a File], page 99).

The *dev* argument specifies which device the special file refers to. Its exact interpretation depends on the kind of special file being created.

The return value is `0` on success and `-1` on error. In addition to the usual file-name errors, the following `errno` error conditions are defined for this function:[24]

`EPERM` The calling process is not privileged. Only the superuser can create special files.

`ENOSPC` The directory or file system that would contain the new file is full and cannot be extended.

`EROFS` The directory containing the new file can't be modified because it's on a read-only file system.

[24] Ibid., "File-Name Errors".

`EEXIST` There is already a file named *filename*. If you want to replace this file, you must remove the old file explicitly first.

3.11 Temporary Files

If you need to use a temporary file in your program, you can use the `tmpfile` function to open it. Or you can use either the `tmpnam` or `tmpnam_r` (better) function to provide a name for a temporary file, and then you can open it in the usual way with `fopen`.

The `tempnam` function is like `tmpnam` but lets you choose what directory temporary files will go in, and something about what their file names will look like. Important for multithreaded programs is that `tempnam` is reentrant, while `tmpnam` is not, since it returns a pointer to a static buffer.

These facilities are declared in the header file '`stdio.h`'.

`FILE *` **tmpfile** (`void`) Function

This function creates a temporary binary file for update mode, as if by calling `fopen` with mode '`wb+`'. The file is deleted automatically when it is closed or when the program terminates. On some other ISO C systems, the file may fail to be deleted if the program terminates abnormally.

This function is reentrant.

When the sources are compiled with `_FILE_OFFSET_BITS == 64` on a 32-bit system, this function is in fact `tmpfile64`—the LFS interface transparently replaces the old interface.

`FILE *` **tmpfile64** (`void`) Function

This function is similar to `tmpfile`, but the stream it returns a pointer to was opened using `tmpfile64`. Therefore, this stream can be used for files larger then $2^3$1 bytes on 32-bit machines.

The return type is still `FILE *`. There is no special `FILE` type for the LFS interface.

If the sources are compiled with `_FILE_OFFSET_BITS == 64` on a 32-bit machine, this function is available under the name `tmpfile` and so transparently replaces the old interface.

`char *` **tmpnam** (`char *`*result*) Function

This function constructs and returns a valid file name that does not refer to any existing file. If the *result* argument is a null pointer, the return value is a pointer to an internal static string, which might be modified by subsequent calls and therefore makes this function nonreentrant. Otherwise, the *result* argument should be a pointer to an array of at least `L_tmpnam` characters, and the result is written into that array.

It is possible for `tmpnam` to fail if you call it too many times without removing previously created files. This is because the limited length of the temporary file

names gives room for only a finite number of different names. If `tmpnam` fails it returns a null pointer.

Warning: Between the time the pathname is constructed and the file is created, another process might have created a file with the same name using `tmpnam`, leading to a possible security hole. The implementation generates names which can hardly be predicted, but when opening the file, you should use the `O_EXCL` flag. Using `tmpfile` or `mkstemp` is a safe way to avoid this problem.

`char *` **tmpnam_r** (`char *`*result*) Function

This function is nearly identical to the `tmpnam` function, except that if *result* is a null pointer, it returns a null pointer.

This guarantees reentrancy because the nonreentrant situation of `tmpnam` cannot happen here.

Warning: This function has the same security problems as `tmpnam`.

`int` **L_tmpnam** Macro

The value of this macro is an integer constant expression that represents the minimum size of a string large enough to hold a file name generated by the `tmpnam` function.

`int` **TMP_MAX** Macro

The macro `TMP_MAX` is a lower bound for how many temporary names you can create with `tmpnam`. You can rely on being able to call `tmpnam` at least this many times before it might fail saying you have made too many temporary file names.

With the GNU library, you can create a very large number of temporary file names. If you actually created the files, you would probably run out of disk space before you ran out of names. Some other systems have a fixed, small limit on the number of temporary files. The limit is never less than `25`.

`char *` **tempnam** (`const char *`*dir*`, const char *`*prefix*) Function

This function generates a unique temporary file name. If *prefix* is not a null pointer, up to five characters of this string are used as a prefix for the file name. The return value is a string newly allocated with `malloc`, so you should release its storage with `free` when it is no longer needed.

Because the string is dynamically allocated, this function is reentrant.

The directory prefix for the temporary file name is determined by testing each of the following in sequence. The directory must exist and be writable.

- The environment variable `TMPDIR`, if it is defined—for security reasons, this only happens if the program is not SUID or SGID enabled
- The *dir* argument, if it is not a null pointer
- The value of the `P_tmpdir` macro
- The directory '`/tmp`'

This function is defined for SVID compatibility.

Warning: Between the time the pathname is constructed and the file is created, another process might have created a file with the same name using `tempnam`, leading to a possible security hole. The implementation generates names that can hardly be predicted, but when opening the file you should use the `O_EXCL` flag. Using `tmpfile` or `mkstemp` is a safe way to avoid this problem.

`char *` **P_tmpdir** SVID Macro

This macro is the name of the default directory for temporary files.

Older Unix systems did not have the functions just described. Instead, they used `mktemp` and `mkstemp`. Both of these functions work by modifying a file-name template string you pass. The last six characters of this string must be '`XXXXXX`'. These six '`X`'s are replaced with six characters that make the whole string a unique file-name. Usually the template string is something like '`/tmp/`*prefix*`XXXXXX`', and each program uses a unique *prefix*.

Because `mktemp` and `mkstemp` modify the template string, you *must not* pass string constants to them. String constants are normally in read-only storage, so your program would crash when `mktemp` or `mkstemp` tried to modify the string. These functions are declared in the header file '`stdlib.h`'.

`char *` **mktemp** (`char *`*template*) Function

The `mktemp` function generates a unique file-name by modifying *template* as described above. If successful, it returns *template* as modified. If `mktemp` cannot find a unique file name, it makes *template* an empty string and returns that. If *template* does not end with '`XXXXXX`', `mktemp` returns a null pointer.

Warning: Between the time the pathname is constructed and the file is created, another process might have created a file with the same name using `mktemp`, leading to a possible security hole. The implementation generates names that can hardly be predicted, but when opening the file you should use the `O_EXCL` flag. Using `mkstemp` is a safe way to avoid this problem.

`int` **mkstemp** (`char *`*template*) Function

The `mkstemp` function generates a unique file-name just as `mktemp` does, but it also opens the file for you with `open` (see Section 2.1 [Opening and Closing Files], page 17). If successful, it modifies *template* in place and returns a file descriptor for that file open for reading and writing. If `mkstemp` cannot create a uniquely named file, it returns `-1`. If *template* does not end with '`XXXXXX`', `mkstemp` returns `-1` and does not modify *template*.

The file is opened using mode `0600`. If the file is meant to be used by other users, this mode must be changed explicitly.

Unlike `mktemp`, `mkstemp` is actually guaranteed to create a unique file that cannot possibly clash with any other program trying to create a temporary file. This is because it works by calling `open` with the `O_EXCL` flag, which says you want to create a new file and get an error if the file already exists.

`char *` **mkdtemp** (`char *`*template*) Function

The `mkdtemp` function creates a directory with a unique name. If it succeeds, it overwrites *template* with the name of the directory, and returns *template*. As with `mktemp` and `mkstemp`, *template* should be a string ending with '`XXXXXX`'.

If `mkdtemp` cannot create an uniquely named directory, it returns `NULL` and sets *errno* appropriately. If *template* does not end with '`XXXXXX`', `mkdtemp` returns `NULL` and does not modify *template*. *errno* will be set to `EINVAL` in this case.

The directory is created using mode `0700`.

The directory created by `mkdtemp` cannot clash with temporary files or directories created by other users. This is because directory creation always works like `open` with `O_EXCL` (see Section 3.8 [Creating Directories], page 92).

The `mkdtemp` function comes from OpenBSD.

4 Pipes and FIFOs

A *pipe* is a mechanism for interprocess communication; data written to the pipe by one process can be read by another process. The data is handled in a first-in, first-out (FIFO) order. The pipe has no name; it is created for one use, and both ends must be inherited from the single process that created the pipe.

A *FIFO special file* is similar to a pipe, but instead of being an anonymous, temporary connection, a FIFO has a name or names like any other file. Processes open the FIFO by name in order to communicate through it.

A pipe or FIFO has to be open at both ends simultaneously. If you read from a pipe or FIFO file that doesn't have any processes writing to it (perhaps because they have all closed the file, or exited), the read returns end-of-file. Writing to a pipe or FIFO that doesn't have a reading process is treated as an error condition; it generates a `SIGPIPE` signal, and fails with error code `EPIPE` if the signal is handled or blocked.

Neither pipes nor FIFO special files allow file positioning. Both reading and writing operations happen sequentially; reading from the beginning of the file and writing at the end.

4.1 Creating a Pipe

The primitive for creating a pipe is the `pipe` function. This creates both the reading and writing ends of the pipe. It is not very useful for a single process to use a pipe to talk to itself. In typical use, a process creates a pipe just before it forks one or more child processes (see Section 7.4 [Creating a Process], page 211). The pipe is then used for communication either between the parent or child processes or between two sibling processes.

The `pipe` function is declared in the header file '`unistd.h`'.

`int` **pipe** (`int` *filedes*`[2]`) Function

The `pipe` function creates a pipe and puts the file descriptors for the reading and writing ends of the pipe, respectively, into *filedes*`[0]` and *filedes*`[1]`.

An easy way to remember that the input end comes first is that file descriptor `0` is standard input, and file descriptor `1` is standard output.

If successful, `pipe` returns a value of `0`. On failure, `-1` is returned. The following `errno` error conditions are defined for this function:

`EMFILE` The process has too many files open.

`ENFILE` There are too many open files in the entire system.[1] This error never occurs in the GNU system.

[1] For more information about `ENFILE`, see Loosemore et al., "Error Codes" (see chap. 1, n. 1).

Here is an example of a simple program that creates a pipe. This program uses the `fork` function to create a child process (see Section 7.4 [Creating a Process], page 211). The parent process writes data to the pipe, which is read by the child process.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

/* Read characters from the pipe and echo them to stdout. */

void
read_from_pipe (int file)
{
  FILE *stream;
  int c;
  stream = fdopen (file, "r");
  while ((c = fgetc (stream)) != EOF)
    putchar (c);
  fclose (stream);
}

/* Write some random text to the pipe. */

void
write_to_pipe (int file)
{
  FILE *stream;
  stream = fdopen (file, "w");
  fprintf (stream, "hello, world!\n");
  fprintf (stream, "goodbye, world!\n");
  fclose (stream);
}

int
main (void)
{
  pid_t pid;
  int mypipe[2];


  /* Create the pipe. */
  if (pipe (mypipe))
    {
```

```
      fprintf (stderr, "Pipe failed.\n");
      return EXIT_FAILURE;
    }


  /* Create the child process. */
  pid = fork ();
  if (pid == (pid_t) 0)
    {
      /* This is the child process.
    Close other end first. */
      close (mypipe[1]);
      read_from_pipe (mypipe[0]);
      return EXIT_SUCCESS;
    }
  else if (pid < (pid_t) 0)
    {
      /* The fork failed. */
      fprintf (stderr, "Fork failed.\n");
      return EXIT_FAILURE;
    }
  else
    {
      /* This is the parent process.
    Close other end first. */
      close (mypipe[0]);
      write_to_pipe (mypipe[1]);
      return EXIT_SUCCESS;
    }
}
```

4.2 Pipe to a Subprocess

A common use of pipes is to send data to or receive data from a program being run as a subprocess. One way of doing this is by using a combination of `pipe` (to create the pipe), `fork` (to create the subprocess), `dup2` (to force the subprocess to use the pipe as its standard input or output channel) and `exec` (to execute the new program). Or, you can use `popen` and `pclose`.

The advantage of using `popen` and `pclose` is that the interface is much simpler and easier to use. But it doesn't offer as much flexibility as using the low-level functions directly.

`FILE *` **popen** (`const char *`*command*`, const char *`*mode*) Function

The `popen` function is closely related to the `system` function (see Section 7.1 [Running a Command], page 209). It executes the shell command *command* as a subprocess. However, instead of waiting for the command to complete, it creates a pipe to the subprocess and returns a stream that corresponds to that pipe.

If you specify a *mode* argument of '`r`', you can read from the stream to retrieve data from the standard output channel of the subprocess. The subprocess inherits its standard input channel from the parent process.

Similarly, if you specify a *mode* argument of '`w`', you can write to the stream to send data to the standard input channel of the subprocess. The subprocess inherits its standard output channel from the parent process.

In the event of an error, `popen` returns a null pointer. This might happen if the pipe or stream cannot be created, if the subprocess cannot be forked or if the program cannot be executed.

`int` **pclose** (`FILE *`*stream*) Function

The `pclose` function is used to close a stream created by `popen`. It waits for the child process to terminate and returns its status value, as for the `system` function.

Here is an example showing how to use `popen` and `pclose` to filter output through another program, in this case the paging program `more`.

```
#include <stdio.h>
#include <stdlib.h>

void
write_data (FILE * stream)
{
  int i;
  for (i = 0; i < 100; i++)
    fprintf (stream, "%d\n", i);
  if (ferror (stream))
    {
      fprintf (stderr, "Output to stream failed.\n");
      exit (EXIT_FAILURE);
    }
}


int
main (void)
{
  FILE *output;
```

```
  output = popen ("more", "w");
  if (!output)
    {
      fprintf (stderr,
               "incorrect parameters or too many files.\n");
      return EXIT_FAILURE;
    }
  write_data (output);
  if (pclose (output) != 0)
    {
      fprintf (stderr,
               "Could not run more or other error.\n");
    }
  return EXIT_SUCCESS;
}
```

4.3 FIFO Special Files

A FIFO special file is similar to a pipe, except that it is created in a different way. Instead of being an anonymous communications channel, a FIFO special file is entered into the file system by calling `mkfifo`.

Once you have created a FIFO special file in this way, any process can open it for reading or writing, in the same way as an ordinary file. However, it has to be open at both ends simultaneously before you can proceed to do any input or output operations on it. Opening a FIFO for reading normally blocks until some other process opens the same FIFO for writing, and vice versa.

The `mkfifo` function is declared in the header file '`sys/stat.h`'.

`int` **mkfifo** (`const char *`*filename*`, mode_t` *mode*) Function

The `mkfifo` function makes a FIFO special file with name *filename*. The *mode* argument is used to set the file's permissions (see Section 3.9.7 [Assigning File Permissions], page 104).

The normal, successful return value from `mkfifo` is `0`. In the case of an error, `-1` is returned. In addition to the usual file-name errors, the following `errno` error conditions are defined for this function:[2]

`EEXIST` The named file already exists.

`ENOSPC` The directory or file system cannot be extended.

`EROFS` The directory that would contain the file resides on a read-only file system.

[2] Ibid., "File-Name Errors".

4.4 Atomicity of Pipe I/O

Reading or writing pipe data is *atomic* if the size of data written is not greater than `PIPE_BUF`. This means that the data transfer seems to be an instantaneous unit, in that nothing else in the system can observe a state in which it is partially complete. Atomic I/O may not begin right away (it may need to wait for buffer space or for data), but once it does begin, it finishes immediately.

Reading or writing a larger amount of data may not be atomic; for example, output data from other processes sharing the descriptor may be interspersed. Also, once `PIPE_BUF` characters have been written, further writes will block until some characters are read.

See Section 12.6 [Limits on File-System Capacity], page 318, for information about the `PIPE_BUF` parameter.

5 Sockets

This chapter describes the GNU facilities for interprocess communication using sockets.

A *socket* is a generalized interprocess communication channel. Like a pipe, a socket is represented as a file descriptor. Unlike pipes, sockets support communication between unrelated processes, and even between processes running on different machines that communicate over a network. Sockets are the primary means of communicating with other machines; `telnet`, `rlogin`, `ftp`, `talk` and the other familiar network programs use sockets.

Not all operating systems support sockets. In the GNU library, the header file '`sys/socket.h`' exists regardless of the operating system, and the socket functions always exist, but if the system does not really support sockets, these functions always fail.

We do not currently document the facilities for broadcast messages or for configuring Internet interfaces. The reentrant functions and some newer functions that are related to IPv6 aren't yet documented either.

5.1 Socket Concepts

When you create a socket, you must specify the style of communication you want to use and the type of protocol that should implement it. The *communication style* of a socket defines the user-level semantics of sending and receiving data on the socket. Choosing a communication style specifies the answers to questions such as these:

- **What are the units of data transmission?** Some communication styles regard the data as a sequence of bytes with no larger structure; others group the bytes into records (which are known in this context as *packets*).
- **Can data be lost during normal operation?** Some communication styles guarantee that all the data sent arrives in the order it was sent (barring system or network crashes); other styles occasionally lose data as a normal part of operation, and may sometimes deliver packets more than once or in the wrong order.

  Designing a program to use unreliable communication styles usually involves taking precautions to detect lost or misordered packets and to retransmit data as needed.
- **Is communication entirely with one partner?** Some communication styles are like a telephone call—you make a *connection* with one remote socket and then exchange data freely. Other styles are like mailing letters—you specify a destination address for each message you send.

You must also choose a *namespace* for naming the socket. A socket name ("address") is meaningful only in the context of a particular namespace. In fact, even the data type to use for a socket name may depend on the namespace. Namespaces

are also called "domains", but we avoid that word as it can be confused with other usage of the same term. Each namespace has a symbolic name that starts with '`PF_`'. A corresponding symbolic name starting with '`AF_`' designates the address format for that namespace.

Finally, you must choose the *protocol* to carry out the communication. The protocol determines what low-level mechanism is used to transmit and receive data. Each protocol is valid for a particular namespace and communication style; a namespace is sometimes called a *protocol family* because of this, which is why the namespace names start with '`PF_`'.

The rules of a protocol apply to the data passing between two programs, perhaps on different computers; most of these rules are handled by the operating system and you need not know about them. What you do need to know about protocols is this:

- In order to have communication between two sockets, they must specify the *same* protocol.
- Each protocol is meaningful with particular style/namespace combinations and cannot be used with inappropriate combinations. For example, the TCP protocol fits only the byte-stream style of communication and the Internet namespace.
- For each combination of style and namespace, there is a *default protocol*, which you can request by specifying 0 as the protocol number. And that's what you should normally do—use the default.

Throughout the following description, at various places variables/parameters to denote sizes are required. And here the trouble starts. In the first implementations, the type of these variables was simply `int`. On most machines at that time, an `int` was 32 bits wide, which created a *de facto* standard requiring 32-bit variables. This is important, since references to variables of this type are passed to the kernel.

Then the POSIX people came and unified the interface with the words "all size values are of type `size_t`". On 64-bit machines, `size_t` is 64 bits wide, so pointers to variables were no longer possible.

The Unix98 specification provides a solution by introducing a type `socklen_t`. This type is used in all of the cases that POSIX changed to use `size_t`. The only requirement of this type is that it be an unsigned type of at least 32 bits. Therefore, implementations that require that references to 32-bit variables be passed can be as happy as implementations that use 64-bit values.

5.2 Communication Styles

The GNU library includes support for several different kinds of sockets, each with different characteristics. This section describes the supported socket types. The symbolic constants listed here are defined in '`sys/socket.h`'.

`int` **SOCK_STREAM** Macro

The `SOCK_STREAM` style is like a pipe (see Chapter 4 [Pipes and FIFOs], page 119). It operates over a connection with a particular remote socket and transmits data reliably as a stream of bytes.

Use of this style is covered in detail in Section 5.9 [Using Sockets with Connections], page 153.

`int` **SOCK_DGRAM** Macro

The `SOCK_DGRAM` style is used for sending individually addressed packets unreliably. It is the diametrical opposite of `SOCK_STREAM`.

Each time you write data to a socket of this kind, that data becomes one packet. Since `SOCK_DGRAM` sockets do not have connections, you must specify the recipient address with each packet.

The only guarantee that the system makes about your requests to transmit data is that it will try its best to deliver each packet you send. It may succeed with the sixth packet after failing with the fourth and fifth packets; the seventh packet may arrive before the sixth, and may arrive a second time after the sixth.

The typical use for `SOCK_DGRAM` is in situations where it is acceptable to simply resend a packet if no response is seen in a reasonable amount of time.

See Section 5.10 [Datagram Socket Operations], page 167, for detailed information about how to use datagram sockets.

`int` **SOCK_RAW** Macro

This style provides access to low-level network protocols and interfaces. Ordinary user programs usually have no need to use this style.

5.3 Socket Addresses

The name of a socket is normally called an *address*. The functions and symbols for dealing with socket addresses were named inconsistently, sometimes using the term “name” and sometimes using “address”. You can regard these terms as synonymous where sockets are concerned.

A socket newly created with the `socket` function has no address. Other processes can find it for communication only if you give it an address. We call this *binding* the address to the socket, and the way to do it is with the `bind` function.

You need be concerned with the address of a socket if other processes are to find it and start communicating with it. You can specify an address for other sockets, but this is usually pointless; the first time you send data from a socket, or use it to initiate a connection, the system assigns an address automatically if you have not specified one.

Occasionally a client needs to specify an address because the server discriminates based on address; for example, the rsh and rlogin protocols look at the client’s socket address and only bypass password checking if it is less than `IPPORT_RESERVED` (see Section 5.6.3 [Internet Ports], page 144).

The details of socket addresses vary depending on what namespace you are using (see Section 5.5 [The Local Namespace], page 132, or Section 5.6 [The Internet Namespace], page 134).

Regardless of the namespace, you use the same functions `bind` and `getsockname` to set and examine a socket's address. These functions use a phony data type, `struct sockaddr *`, to accept the address. In practice, the address lives in a structure of some other data type appropriate to the address format you are using, but you cast its address to `struct sockaddr *` when you pass it to `bind`.

5.3.1 Address Formats

The functions `bind` and `getsockname` use the generic data type `struct sockaddr *` to represent a pointer to a socket address. You can't use this data type effectively to interpret an address or construct one; for that, you must use the proper data type for the socket's namespace.

Thus, the usual practice is to construct an address of the proper namespace-specific type, then cast a pointer to `struct sockaddr *` when you call `bind` or `getsockname`.

The one piece of information that you can get from the `struct sockaddr` data type is the *address format designator*. This tells you which data type to use to understand the address fully.

The symbols in this section are defined in the header file '`sys/socket.h`'.

struct sockaddr Data Type

The `struct sockaddr` type itself has the following members:

`short int sa_family`
: This is the code for the address format of this address. It identifies the format of the data that follows.

`char sa_data[14]`
: This is the actual socket address data, which is format dependent. Its length also depends on the format, and may well be more than 14. The length 14 of `sa_data` is essentially arbitrary.

Each address format has a symbolic name that starts with '`AF_`'. Each of them corresponds to a '`PF_`' symbol that designates the corresponding namespace. Here is a list of address format names:

`AF_LOCAL`
: This designates the address format that goes with the local namespace. (`PF_LOCAL` is the name of that namespace.) See Section 5.5.2 [Details of Local Namespace], page 132, for information about this address format.

`AF_UNIX` This is a synonym for `AF_LOCAL`. Although `AF_LOCAL` is mandated by POSIX.1g, `AF_UNIX` is portable to more systems. `AF_UNIX` was

the traditional name stemming from BSD, so even most POSIX systems support it. It is also the name of choice in the Unix98 specification. The same is true for PF_UNIX vs. PF_LOCAL.

AF_FILE This is another synonym for AF_LOCAL, for compatibility. PF_FILE is likewise a synonym for PF_LOCAL.

AF_INET This designates the address format that goes with the Internet namespace (see Section 5.6.1 [Internet Socket Address Formats], page 135). PF_INET is the name of that namespace.

AF_INET6
This is similar to AF_INET, but refers to the IPv6 protocol. PF_INET6 is the name of the corresponding namespace.

AF_UNSPEC
This designates no particular address format. It is used only in rare cases, such as to clear out the default destination address of a "connected" datagram socket (see Section 5.10.1 [Sending Datagrams], page 167).

The corresponding namespace designator symbol PF_UNSPEC exists for completeness, but there is no reason to use it in a program.

'sys/socket.h' defines symbols starting with 'AF_' for many different kinds of networks, most or all of which are not actually implemented. We will document those that really work as we receive information about how to use them.

5.3.2 Setting the Address of a Socket

Use the bind function to assign an address to a socket. The prototype for bind is in the header file 'sys/socket.h'. For examples of use, see Section 5.5.3 [Example of Local-Namespace Sockets], page 133, or Section 5.6.7 [Internet Socket Example], page 149.

int **bind** (int *socket*, struct sockaddr **addr*, socklen_t *length*) Function

The bind function assigns an address to the socket *socket*. The *addr* and *length* arguments specify the address; the detailed format of the address depends on the namespace. The first part of the address is always the format designator, which specifies a namespace and says that the address is in the format of that namespace.

The return value is 0 on success and −1 on failure. The following errno error conditions are defined for this function:

EBADF The *socket* argument is not a valid file-descriptor.

ENOTSOCK
The descriptor *socket* is not a socket.

`EADDRNOTAVAIL`
: The specified address is not available on this machine.

`EADDRINUSE`
: Some other socket is already using the specified address.

`EINVAL` The socket *socket* already has an address.

`EACCES` You do not have permission to access the requested address. In the Internet domain, only the superuser is allowed to specify a port number in the range 0 through `IPPORT_RESERVED` minus 1 (see Section 5.6.3 [Internet Ports], page 144).

Additional conditions may be possible depending on the particular namespace of the socket.

5.3.3 Reading the Address of a Socket

Use the function `getsockname` to examine the address of an Internet socket. The prototype for this function is in the header file '`sys/socket.h`'.

`int` **getsockname** (`int` *socket*, `struct sockaddr` ⋆*addr*, `socklen_t` ⋆*length-ptr*) — Function

The `getsockname` function returns information about the address of the socket *socket* in the locations specified by the *addr* and *length-ptr* arguments. The *length-ptr* is a pointer; you should initialize it to be the allocation size of *addr*, and on return it contains the actual size of the address data.

The format of the address data depends on the socket namespace. The length of the information is usually fixed for a given namespace, so normally you can know exactly how much space is needed and can provide that much. The usual practice is to allocate a place for the value using the proper data type for the socket's namespace, then cast its address to `struct sockaddr ⋆` to pass it to `getsockname`.

The return value is `0` on success and `-1` on error. The following `errno` error conditions are defined for this function:

`EBADF` The *socket* argument is not a valid file-descriptor.

`ENOTSOCK`
: The descriptor *socket* is not a socket.

`ENOBUFS` There are not enough internal buffers available for the operation.

You can't read the address of a socket in the file namespace. This is consistent with the rest of the system; in general, there's no way to find a file's name from a descriptor for that file.

5.4 Interface Naming

Each network interface has a name. This usually consists of a few letters that relate to the type of interface, which may be followed by a number if there is more than one interface of that type. Examples might be `lo` (the loopback interface) and `eth0` (the first Ethernet interface).

Although such names are convenient for humans, it would be clumsy to have to use them whenever a program needs to refer to an interface. In such situations an interface is referred to by its *index*, which is an arbitrarily-assigned small positive integer.

The following functions, constants and data types are declared in the header file '`net/if.h`'.

`size_t` **IFNAMSIZ** Constant

This constant defines the maximum buffer size needed to hold an interface name, including its terminating zero byte.

`unsigned int` **if_nametoindex** (`const char *ifname`) Function

This function yields the interface index corresponding to a particular name. If no interface exists with the name given, it returns 0.

`char *` **if_indextoname** (`unsigned int ifindex, char *ifname`) Function

This function maps an interface index to its corresponding name. The returned name is placed in the buffer pointed to by `ifname`, which must be at least `IFNAMSIZ` bytes in length. If the index was invalid, the function's return value is a null pointer, otherwise it is `ifname`.

struct if_nameindex Data Type

This data type is used to hold the information about a single interface. It has the following members:

`unsigned int if_index;`
: This is the interface index.

`char *if_name`
: This is the null-terminated index name.

`struct if_nameindex *` **if_nameindex** (`void`) Function

This function returns an array of `if_nameindex` structures, one for every interface that is present. The end of the list is indicated by a structure with an interface of 0 and a null name pointer. If an error occurs, this function returns a null pointer.

The returned structure must be freed with `if_freenameindex` after use.

`void` **if_freenameindex** (`struct if_nameindex *ptr`) Function

This function frees the structure returned by an earlier call to `if_nameindex`.

5.5 The Local Namespace

This section describes the details of the local namespace, whose symbolic name (required when you create a socket) is `PF_LOCAL`. The local namespace is also known as "Unix domain sockets". Another name is file namespace since socket addresses are normally implemented as file names.

5.5.1 Local-Namespace Concepts

In the local namespace, socket addresses are file names. You can specify any file name you want as the address of the socket, but you must have write permission on the directory containing it. It's common to put these files in the '`/tmp`' directory.

One peculiarity of the local namespace is that the name is only used when opening the connection; once open the address is not meaningful and may not exist.

Another peculiarity is that you cannot connect to such a socket from another machine—not even if the other machine shares the file system that contains the name of the socket. You can see the socket in a directory listing, but connecting to it never succeeds. Some programs take advantage of this, such as by asking the client to send its own process ID, and using the process IDs to distinguish between clients. However, we recommend you not use this method in protocols you design, as we might someday permit connections from other machines that mount the same file systems. Instead, send each new client an identifying number if you want it to have one.

After you close a socket in the local namespace, you should delete the file name from the file system. Use `unlink` or `remove` to do this (see Section 3.6 [Deleting Files], page 90).

The local namespace supports just one protocol for any communication style; it is protocol number `0`.

5.5.2 Details of Local Namespace

To create a socket in the local namespace, use the constant `PF_LOCAL` as the *namespace* argument to `socket` or `socketpair`. This constant is defined in '`sys/socket.h`'.

`int` **PF_LOCAL** Macro

This designates the local namespace, in which socket addresses are local names, and its associated family of protocols. `PF_Local` is the macro used by Posix.1g.

`int` **PF_UNIX** Macro

This is a synonym for `PF_LOCAL`, for compatibility's sake.

`int` **PF_FILE** Macro

This is a synonym for `PF_LOCAL`, for compatibility's sake.

The structure for specifying socket names in the local namespace is defined in the header file ‘sys/un.h’:

struct sockaddr_un Data Type

This structure is used to specify local namespace socket addresses. It has the following members:

`short int sun_family`
: This identifies the address family or format of the socket address. You should store the value `AF_LOCAL` to designate the local namespace (see Section 5.3 [Socket Addresses], page 127).

`char sun_path[108]`
: This is the file name to use.

You should compute the *length* parameter for a socket address in the local namespace as the sum of the size of the `sun_family` component and the string length (*not* the allocation size!) of the file-name string. This can be done using the macro `SUN_LEN`:

`int` **SUN_LEN** (*struct sockaddr_un * ptr*) Macro

The macro computes the length of socket address in the local namespace.

5.5.3 Example of Local-Namespace Sockets

Here is an example showing how to create and name a socket in the local namespace.

```
#include <stddef.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>

int
make_named_socket (const char *filename)
{
  struct sockaddr_un name;
  int sock;
  size_t size;

  /* Create the socket.  */
  sock = socket (PF_LOCAL, SOCK_DGRAM, 0);
  if (sock < 0)
    {
```

```
      perror ("socket");
      exit (EXIT_FAILURE);
    }

  /* Bind a name to the socket.  */
  name.sun_family = AF_LOCAL;
  strncpy (name.sun_path, filename, sizeof (name.sun_path));
  name.sun_path[sizeof (name.sun_path) - 1] = '\0';

  /* The size of the address is
  the offset of the start of the file name,
  plus its length,
  plus 1 for the terminating null byte.
  Alternatively you can just do:
  size = SUN_LEN (&name);
 */
  size = (offsetof (struct sockaddr_un, sun_path)
          + strlen (name.sun_path) + 1);

  if (bind (sock, (struct sockaddr *) &name, size) < 0)
    {
      perror ("bind");
      exit (EXIT_FAILURE);
    }

  return sock;
}
```

5.6 The Internet Namespace

This section describes the details of the protocols and socket naming conventions used in the Internet namespace.

Originally, the Internet namespace used only IP version 4 (IPv4). With the growing number of hosts on the Internet, a new protocol with a larger address space was necessary: IP version 6 (IPv6). IPv6 introduces 128-bit addresses (IPv4 has 32-bit addresses) and other features, and will eventually replace IPv4.

To create a socket in the IPv4 Internet namespace, use the symbolic name `PF_INET` of this namespace as the *namespace* argument to `socket` or `socketpair`. For IPv6 addresses, you need the macro `PF_INET6`. These macros are defined in ‘`sys/socket.h`’.

`int` **PF_INET** Macro

This designates the IPv4 Internet namespace and associated family of protocols.

`int` **PF_INET6** Macro

This designates the IPv6 Internet namespace and associated family of protocols.

A socket address for the Internet namespace includes the following components:

- The address of the machine you want to connect to; Internet addresses can be specified in several ways—these are discussed in Section 5.6.1 [Internet Socket Address Formats], page 135; Section 5.6.2 [Host Addresses], page 136 and Section 5.6.2.4 [Host Names], page 141.
- A port number for that machine (see Section 5.6.3 [Internet Ports], page 144)

You must ensure that the address and port number are represented in a canonical format called *network byte order* (see Section 5.6.5 [Byte-Order Conversion], page 147).

5.6.1 Internet Socket Address Formats

In the Internet namespace, for both IPv4 (`AF_INET`) and IPv6 (`AF_INET6`), a socket address consists of a host address and a port on that host. In addition, the protocol you choose effectively serves as a part of the address because local port numbers are meaningful only within a particular protocol.

The data types for representing socket addresses in the Internet namespace are defined in the header file '`netinet/in.h`'.

struct sockaddr_in Data Type

This is the data type used to represent socket addresses in the Internet namespace. It has the following members:

`sa_family_t sin_family`
: This identifies the address family or format of the socket address. You should store the value `AF_INET` in this member (see Section 5.3 [Socket Addresses], page 127).

`struct in_addr sin_addr`
: This is the Internet address of the host machine. See Section 5.6.2 [Host Addresses], page 136, and Section 5.6.2.4 [Host Names], page 141, for how to get a value to store here.

`unsigned short int sin_port`
: This is the port number (see Section 5.6.3 [Internet Ports], page 144).

When you call `bind` or `getsockname`, you should specify `sizeof (struct sockaddr_in)` as the *length* parameter if you are using an IPv4 Internet namespace socket address.

struct sockaddr_in6 Data Type

This is the data type used to represent socket addresses in the IPv6 namespace. It has the following members:

`sa_family_t sin6_family`
: This identifies the address family or format of the socket address. You should store the value of `AF_INET6` in this member (see Section 5.3 [Socket Addresses], page 127).

`struct in6_addr sin6_addr`
: This is the IPv6 address of the host machine. See Section 5.6.2 [Host Addresses], page 136, and Section 5.6.2.4 [Host Names], page 141, for how to get a value to store here.

`uint32_t sin6_flowinfo`
: This is a currently unimplemented field.

`uint16_t sin6_port`
: This is the port number (see Section 5.6.3 [Internet Ports], page 144).

5.6.2 Host Addresses

Each computer on the Internet has one or more *Internet addresses*, numbers which identify that computer among all those on the Internet. Users typically write IPv4 numeric host-addresses as sequences of four numbers, separated by periods, as in '`128.52.46.32`', and IPv6 numeric host-addresses as sequences of up to eight numbers separated by colons, as in '`5f03:1200:836f:c100::1`'.

Each computer also has one or more *host names*, which are strings of words separated by periods, as in '`mescaline.gnu.org`'.

Programs that let the user specify a host typically accept both numeric addresses and host names. To open a connection, a program needs a numeric address, and so must convert a host name to the numeric address it stands for.

5.6.2.1 Internet Host-Addresses

An IPv4 Internet host-address is a number containing 4 bytes of data. Historically, these are divided into two parts, a *network number* and a *local network address number* within that network. In the mid-1990s, classless addresses were introduced that changed this behavior. Since some functions implicitly expect the old definitions, we first describe the class-based network and will then describe classless addresses. IPv6 uses only classless addresses and therefore the following paragraphs don't apply to it.

The class-based IPv4 network number consists of the first 1, 2 or 3 bytes; the rest of the bytes are the local address.

IPv4 network numbers are registered with the Network Information Center (NIC), and are divided into three classes—A, B and C. The local network address numbers of individual machines are registered with the administrator of the particular network.

Class A networks have single-byte numbers in the range 0 to 127. There are only a small number of Class A networks, but they can each support a very large number

of hosts. Medium-sized Class B networks have 2-byte network numbers, with the first byte in the range 128 to 191. Class C networks are the smallest; they have 3-byte network numbers, with the first byte in the range 192-255. Thus, the first 1, 2 or 3 bytes of an Internet address specify a network. The remaining bytes of the Internet address specify the address within that network.

The Class A network 0 is reserved for broadcast to all networks. In addition, the host number 0 within each network is reserved for broadcast to all hosts in that network. These uses are obsolete now but for compatibility reasons, you shouldn't use network 0 and host number 0.

The Class A network 127 is reserved for loopback; you can always use the Internet address '`127.0.0.1`' to refer to the host machine.

Since a single machine can be a member of multiple networks, it can have multiple Internet host-addresses. However, there is never supposed to be more than one machine with the same host address.

There are four forms of the *standard numbers-and-dots notation* for Internet addresses:

a`.`*b*`.`*c*`.`*d* This specifies all 4 bytes of the address individually and is the commonly used representation.

a`.`*b*`.`*c* The last part of the address, *c*, is interpreted as a 2-byte quantity. This is useful for specifying host addresses in a Class B network with network address number *a*`.`*b*.

a`.`*b* The last part of the address, *b*, is interpreted as a 3-byte quantity. This is useful for specifying host addresses in a Class A network with network address number *a*.

a If only one part is given, this corresponds directly to the host-address number.

Within each part of the address, the usual C conventions for specifying the radix apply. In other words, a leading '`0x`' or '`0X`' implies hexadecimal radix; a leading '`0`' implies octal; otherwise, decimal radix is assumed.

Classless Addresses

IPv4 addresses (and IPv6 addresses also) are now considered classless; the distinction between classes A, B and C can be ignored. Instead, an IPv4 host-address consists of a 32-bit address and a 32-bit mask. The mask contains set bits for the network part and cleared bits for the host part. The network part is contiguous from the left, with the remaining bits representing the host. As a consequence, the netmask can simply be specified as the number of set bits. Classes A, B and C are just special cases of this general rule. For example, class A addresses have a netmask of '`255.0.0.0`' or a prefix length of 8.

Classless IPv4 network addresses are written in numbers-and-dots notation with the prefix length appended and a slash as separator. For example the class A network 10 is written as '`10.0.0.0/8`'.

IPv6 Addresses

IPv6 addresses contain 128 bits (IPv4 has 32 bits) of data. A host address is usually written as eight 16-bit hexadecimal numbers that are separated by colons. Two colons are used to abbreviate strings of consecutive zeros. For example, the IPv6 loopback address '`0:0:0:0:0:0:0:1`' can just be written as '`::1`'.

5.6.2.2 Host-Address Data Type

IPv4 Internet host-addresses are represented in some contexts as integers (type `uint32_t`). In other contexts, the integer is packaged inside a structure of type `struct in_addr`. It would be better if the usage were made consistent, but it is not hard to extract the integer from the structure or put the integer into a structure.

You will find older code that uses `unsigned long int` for IPv4 Internet host-addresses instead of `uint32_t` or `struct in_addr`. Historically, `unsigned long int` was a 32-bit number, but with 64-bit machines this has changed. Using `unsigned long int` might break the code if it is used on machines where this type doesn't have 32 bits. `uint32_t` is specified by Unix98 and guaranteed to have 32 bits.

IPv6 Internet host-addresses have 128 bits and are packaged inside a structure of type `struct in6_addr`.

The following basic definitions for Internet addresses are declared in the header file '`netinet/in.h`':

struct in_addr Data Type

This data type is used in certain contexts to contain an IPv4 Internet host-address. It has just one field, named `s_addr`, which records the host-address number as an `uint32_t`.

`uint32_t` **INADDR_LOOPBACK** Macro

You can use this constant to stand for "the address of this machine," instead of finding its actual address. It is the IPv4 Internet address '`127.0.0.1`', which is usually called '`localhost`'. This special constant saves you the trouble of looking up the address of your own machine. Also, the system usually implements `INADDR_LOOPBACK` specially, avoiding any network traffic for the case of one machine talking to itself.

`uint32_t` **INADDR_ANY** Macro

You can use this constant to stand for "any incoming address" when binding to an address (see Section 5.3.2 [Setting the Address of a Socket], page 129). This is the usual address to give in the `sin_addr` member of `struct sockaddr_in` when you want to accept Internet connections.

`uint32_t` **INADDR_BROADCAST** Macro

This constant is the address you use to send a broadcast message.

`uint32_t` **INADDR_NONE** Macro
This constant is returned by some functions to indicate an error.

struct in6_addr Data Type
This data type is used to store an IPv6 address. It stores 128 bits of data, which can be accessed (via a union) in a variety of ways.

`struct in6_addr` **in6addr_loopback** Constant
This constant is the IPv6 address '`::1`', the loopback address. See above for a description of what this means. The macro `IN6ADDR_LOOPBACK_INIT` is provided to allow you to initialize your own variables to this value.

`struct in6_addr` **in6addr_any** Constant
This constant is the IPv6 address '`::`', the unspecified address. See above for a description of what this means. The macro `IN6ADDR_ANY_INIT` is provided to allow you to initialize your own variables to this value.

5.6.2.3 Host-Address Functions

These additional functions for manipulating Internet addresses are declared in the header file '`arpa/inet.h`'. They represent Internet addresses in network byte order, and network numbers and local-address-within-network numbers in host byte order. See Section 5.6.5 [Byte-Order Conversion], page 147, for an explanation of network and host byte order.

`int` **inet_aton** (`const char *`*name*`, struct in_addr *`*addr*) Function
This function converts the IPv4 Internet host-address *name* from the standard numbers-and-dots notation into binary data and stores it in the `struct in_addr` that *addr* points to. `inet_aton` returns nonzero if the address is valid and zero if not.

`uint32_t` **inet_addr** (`const char *`*name*) Function
This function converts the IPv4 Internet host-address *name* from the standard numbers-and-dots notation into binary data. If the input is not valid, `inet_addr` returns `INADDR_NONE`. This is an obsolete interface to `inet_aton`, described immediately above. It is obsolete because `INADDR_NONE` is a valid address (255.255.255.255), and `inet_aton` provides a cleaner way to indicate error return.

`uint32_t` **inet_network** (`const char *`*name*) Function
This function extracts the network number from the address *name*, given in the standard numbers-and-dots notation. The returned address is in host order. If the input is not valid, `inet_network` returns `-1`.

The function works only with traditional IPv4 class A, B and C network types. It doesn't work with classless addresses and shouldn't be used anymore.

`char *` **inet_ntoa** (`struct in_addr` *addr*) Function

This function converts the IPv4 Internet host-address *addr* to a string in the standard numbers-and-dots notation. The return value is a pointer into a statically allocated buffer. Subsequent calls will overwrite the same buffer, so you should copy the string if you need to save it.

In multithreaded programs, each thread has its own statically allocated buffer. But still, subsequent calls of `inet_ntoa` in the same thread will overwrite the result of the last call.

Instead of `inet_ntoa`, the newer function `inet_ntop`, which is described below, should be used since it handles both IPv4 and IPv6 addresses.

`struct in_addr` **inet_makeaddr** (`uint32_t` *net*, `uint32_t` *local*) Function

This function makes an IPv4 Internet host-address by combining the network number *net* with the local-address-within-network number *local*.

`uint32_t` **inet_lnaof** (`struct in_addr` *addr*) Function

This function returns the local-address-within-network part of the Internet host-address *addr*.

The function works only with traditional IPv4 class A, B and C network types. It doesn't work with classless addresses and shouldn't be used anymore.

`uint32_t` **inet_netof** (`struct in_addr` *addr*) Function

This function returns the network number part of the Internet host address *addr*.

The function works only with traditional IPv4 class A, B and C network types. It doesn't work with classless addresses and shouldn't be used anymore.

`int` **inet_pton** (`int` *af*, `const char *`*cp*, `void *`*buf*) Function

This function converts an Internet address (either IPv4 or IPv6) from presentation (textual) to network (binary) format. *af* should be either `AF_INET` or `AF_INET6`, as appropriate for the type of address being converted. *cp* is a pointer to the input string, and *buf* is a pointer to a buffer for the result. It is the caller's responsibility to make sure the buffer is large enough.

`const char *` **inet_ntop** (`int` *af*, `const void *`*cp*, `char *`*buf*, `size_t` *len*) Function

This function converts an Internet address (either IPv4 or IPv6) from network (binary) to presentation (textual) form. *af* should be either `AF_INET` or `AF_INET6`, as appropriate. *cp* is a pointer to the address to be converted. *buf* should be a pointer to a buffer to hold the result, and *len* is the length of this buffer. The return value from the function will be this buffer address.

5.6.2.4 Host Names

Besides the standard numbers-and-dots notation for Internet addresses, you can also refer to a host by a symbolic name. The advantage of a symbolic name is that it is usually easier to remember. For example, the machine with Internet address '158.121.106.19' is also known as 'alpha.gnu.org'; and other machines in the 'gnu.org' domain can refer to it simply as 'alpha'.

Internally, the system uses a database to keep track of the mapping between host names and host numbers. This database is usually either the file '/etc/hosts' or an equivalent provided by a name server. The functions and other symbols for accessing this database are declared in 'netdb.h'. They are BSD features, defined unconditionally if you include 'netdb.h'.

struct hostent Data Type

This data type is used to represent an entry in the hosts database. It has the following members:

`char *h_name`
: This is the "official" name of the host.

`char **h_aliases`
: These are alternative names for the host, represented as a null-terminated vector of strings.

`int h_addrtype`
: This is the host-address type; in practice, its value is always either `AF_INET` or `AF_INET6`, with the latter being used for IPv6 hosts. In principle, other kinds of addresses could be represented in the database as well as Internet addresses; if this were done, you might find a value in this field other than `AF_INET` or `AF_INET6` (see Section 5.3 [Socket Addresses], page 127).

`int h_length`
: This is the length, in bytes, of each address.

`char **h_addr_list`
: This is the vector of addresses for the host. (Recall that the host might be connected to multiple networks and have different addresses on each one.) The vector is terminated by a null pointer.

`char *h_addr`
: This is a synonym for `h_addr_list[0]`; in other words, it is the first host-address.

As far as the host database is concerned, each address is just a block of memory `h_length` bytes long. But in other contexts, there is an implicit assumption that you can convert IPv4 addresses to a `struct in_addr` or an `uint32_t`. Host addresses in a `struct hostent` structure are always given in network byte order (see Section 5.6.5 [Byte-Order Conversion], page 147).

You can use `gethostbyname`, `gethostbyname2` or `gethostbyaddr` to search the hosts database for information about a particular host. The information is returned in a statically allocated structure; you must copy the information if you need to save it across calls. You can also use `getaddrinfo` and `getnameinfo` to obtain this information.

`struct hostent *` **gethostbyname** (`const char *`*name*) Function

The `gethostbyname` function returns information about the host named *name*. If the lookup fails, it returns a null pointer.

`struct hostent *` **gethostbyname2** (`const char *`*name*`, int` *af*) Function

The `gethostbyname2` function is like `gethostbyname`, but allows the caller to specify the desired address family (e.g. `AF_INET` or `AF_INET6`) of the result.

`struct hostent *` **gethostbyaddr** (`const char *`*addr*`, size_t` *length*`, int` *format*) Function

The `gethostbyaddr` function returns information about the host with Internet address *addr*. The parameter *addr* is not really a pointer to char - it can be a pointer to an IPv4 or an IPv6 address. The *length* argument is the size (in bytes) of the address at *addr*. *format* specifies the address format; for an IPv4 Internet address, specify a value of `AF_INET`; for an IPv6 Internet address, use `AF_INET6`.

If the lookup fails, `gethostbyaddr` returns a null pointer.

If the name lookup by `gethostbyname` or `gethostbyaddr` fails, you can find out the reason by looking at the value of the variable `h_errno`. It would be cleaner design for these functions to set `errno`, but use of `h_errno` is compatible with other systems.

Here are the error codes that you may find in `h_errno`:

`HOST_NOT_FOUND`
: No such host is known in the database.

`TRY_AGAIN`
: This condition happens when the name server could not be contacted. If you try again later, you may succeed then.

`NO_RECOVERY`
: A nonrecoverable error occurred.

`NO_ADDRESS`
: The host database contains an entry for the name, but it doesn't have an associated Internet address.

The lookup functions above all have one thing in common: they are not reentrant and so are unusable in multithreaded applications. Therefore, the GNU C Library provides a new set of functions that can be used in this context.

`int` **gethostbyname_r** (`const char *restrict` *name*, `struct hostent *restrict` *result_buf*, `char *restrict` *buf*, `size_t` *buflen*, `struct hostent **restrict` *result*, `int *restrict` *h_errnop*) Function

The `gethostbyname_r` function returns information about the host named *name*. The caller must pass a pointer to an object of type `struct hostent` in the *result_buf* parameter. In addition, the function may need extra buffer space, and the caller must pass a pointer and the size of the buffer in the *buf* and *buflen* parameters.

A pointer to the buffer, in which the result is stored, is available in **result* after the function call successfully returned. If an error occurs or if no entry is found, the pointer **result* is a null pointer. Success is signalled by a zero return value. If the function failed, the return value is an error number. In addition to the errors defined for `gethostbyname`, it can also be `ERANGE`. In this case, the call should be repeated with a larger buffer. Additional error information is not stored in the global variable `h_errno` but instead in the object pointed to by *h_errnop*.

Here's a small example:

```
struct hostent *
gethostname (char *host)
{
  struct hostent hostbuf, *hp;
  size_t hstbuflen;
  char *tmphstbuf;
  int res;
  int herr;

  hstbuflen = 1024;
  /* Allocate buffer, remember to free it to avoid memory leakage.  */
  tmphstbuf = malloc (hstbuflen);

  while ((res = gethostbyname_r (host, &hostbuf, tmphstbuf, hstbuflen,
                                 &hp, &herr)) == ERANGE)
    {
      /* Enlarge the buffer.  */
      hstbuflen *= 2;
      tmphstbuf = realloc (tmphstbuf, hstbuflen);
    }
  /*  Check for errors.  */
  if (res || hp == NULL)
    return NULL;
  return hp;
}
```

`int` **gethostbyname2_r** (`const char *`*name*`, int` *af*`, struct hostent *restrict` *result_buf*`, char *restrict` *buf*`, size_t` *buflen*`, struct hostent **restrict` *result*`, int *restrict` *h_errnop*) Function

The `gethostbyname2_r` function is like `gethostbyname_r`, but allows the caller to specify the desired address family (e.g. `AF_INET` or `AF_INET6`) for the result.

`int` **gethostbyaddr_r** (`const char *`*addr*`, size_t` *length*`, int` *format*`, struct hostent *restrict` *result_buf*`, char *restrict` *buf*`, size_t` *buflen*`, struct hostent **restrict` *result*`, int *restrict` *h_errnop*) Function

The `gethostbyaddr_r` function returns information about the host with Internet address *addr*. The parameter *addr* is not really a pointer to char—it can be a pointer to an IPv4 or an IPv6 address. The *length* argument is the size (in bytes) of the address at *addr*. *format* specifies the address format; for an IPv4 Internet address, specify a value of `AF_INET`; for an IPv6 Internet address, use `AF_INET6`.

Similar to the `gethostbyname_r` function, the caller must provide buffers for the result and memory used internally. In case of success, the function returns 0. Otherwise, the value is an error number where `ERANGE` has the special meaning that the caller-provided buffer is too small.

You can also scan the entire hosts database one entry at a time using `sethostent`, `gethostent` and `endhostent`. Be careful when using these functions, because they are not reentrant.

`void` **sethostent** (`int` *stayopen*) Function

This function opens the hosts database to begin scanning it. You can then call `gethostent` to read the entries.

If the *stayopen* argument is nonzero, this sets a flag so that subsequent calls to `gethostbyname` or `gethostbyaddr` will not close the database (as they usually would). This makes for more efficiency if you call those functions several times, by avoiding reopening the database for each call.

`struct hostent *` **gethostent** (`void`) Function

This function returns the next entry in the hosts database. It returns a null pointer if there are no more entries.

`void` **endhostent** (`void`) Function

This function closes the hosts database.

5.6.3 Internet Ports

A socket address in the Internet namespace consists of a machine's Internet address plus a *port number* that distinguishes the sockets on a given machine (for a given protocol). Port numbers range from 0 to 65,535.

Port numbers less than IPPORT_RESERVED are reserved for standard servers, such as finger and telnet. There is a database that keeps track of these, and you can use the getservbyname function to map a service name onto a port number (see Section 5.6.4 [The Services Database], page 145).

If you write a server that is not one of the standard ones defined in the database, you must choose a port number for it. Use a number greater than IPPORT_USERRESERVED; such numbers are reserved for servers and won't ever be generated automatically by the system. Avoiding conflicts with servers being run by other users is up to you.

When you use a socket without specifying its address, the system generates a port number for it. This number is between IPPORT_RESERVED and IPPORT_USERRESERVED.

On the Internet, it is actually legitimate to have two different sockets with the same port number, as long as they never both try to communicate with the same socket address (host address plus port number). You shouldn't duplicate a port number except in special circumstances where a higher-level protocol requires it. Normally, the system won't let you do it; bind normally insists on distinct port numbers. To reuse a port number, you must set the socket option SO_REUSEADDR (see Section 5.12.2 [Socket-Level Options], page 174).

These macros are defined in the header file 'netinet/in.h'.

int **IPPORT_RESERVED** Macro

Port numbers less than IPPORT_RESERVED are reserved for superuser use.

int **IPPORT_USERRESERVED** Macro

Port numbers greater than or equal to IPPORT_USERRESERVED are reserved for explicit use; they will never be allocated automatically.

5.6.4 The Services Database

The database that keeps track of "well-known" services is usually either the file '/etc/services' or an equivalent from a name server. You can use these utilities, declared in 'netdb.h', to access the services database.

struct servent Data Type

This data type holds information about entries from the services database. It has the following members:

char *s_name

This is the "official" name of the service.

char **s_aliases

These are alternate names for the service, represented as an array of strings. A null pointer terminates the array.

`int s_port`
: This is the port number for the service. Port numbers are given in network byte order (see Section 5.6.5 [Byte-Order Conversion], page 147).

`char *s_proto`
: This is the name of the protocol to use with this service (see Section 5.6.6 [Protocols Database], page 147).

To get information about a particular service, use the `getservbyname` or `getservbyport` functions. The information is returned in a statically allocated structure; you must copy the information if you need to save it across calls.

`struct servent *` **getservbyname** (`const char *`*name*`, const char *`*proto*) — Function

The `getservbyname` function returns information about the service named *name* using protocol *proto*. If it can't find such a service, it returns a null pointer.

This function is useful for servers as well as for clients; servers use it to determine which port they should listen on (see Section 5.9.2 [Listening for Connections], page 155).

`struct servent *` **getservbyport** (`int` *port*`, const char *`*proto*) — Function

The `getservbyport` function returns information about the service at port *port* using protocol *proto*. If it can't find such a service, it returns a null pointer.

You can also scan the services database using `setservent`, `getservent` and `endservent`. Be careful when using these functions, because they are not reentrant.

`void` **setservent** (`int` *stayopen*) — Function

This function opens the services database to begin scanning it.

If the *stayopen* argument is nonzero, this sets a flag so that subsequent calls to `getservbyname` or `getservbyport` will not close the database (as they usually would). This makes for more efficiency if you call those functions several times, by avoiding reopening the database for each call.

`struct servent *` **getservent** (`void`) — Function

This function returns the next entry in the services database. If there are no more entries, it returns a null pointer.

`void` **endservent** (`void`) — Function

This function closes the services database.

5.6.5 Byte-Order Conversion

Different kinds of computers use different conventions for the ordering of bytes within a word. Some computers put the most significant byte within a word first (this is called "big-endian" order), and others put it last ("little-endian" order).

So that machines with different byte-order conventions can communicate, the Internet protocols specify a canonical byte-order convention for data transmitted over the network. This is known as *network byte order*.

When establishing an Internet socket connection, you must make sure that the data in the `sin_port` and `sin_addr` members of the `sockaddr_in` structure are represented in network byte order. If you are encoding integer data in the messages sent through the socket, you should convert this to network byte order too. If you don't do this, your program may fail when running on or talking to other kinds of machines.

If you use `getservbyname` and `gethostbyname` or `inet_addr` to get the port number and host address, the values are already in network byte order, and you can copy them directly into the `sockaddr_in` structure.

Otherwise, you have to convert the values explicitly. Use `htons` and `ntohs` to convert values for the `sin_port` member. Use `htonl` and `ntohl` to convert IPv4 addresses for the `sin_addr` member. (Remember, `struct in_addr` is equivalent to `uint32_t`.) These functions are declared in '`netinet/in.h`'.

`uint16_t` **htons** (`uint16_t` *hostshort*) Function
This function converts the `uint16_t` integer *hostshort* from host byte order to network byte order.

`uint16_t` **ntohs** (`uint16_t` *netshort*) Function
This function converts the `uint16_t` integer *netshort* from network byte order to host byte order.

`uint32_t` **htonl** (`uint32_t` *hostlong*) Function
This function converts the `uint32_t` integer *hostlong* from host byte order to network byte order.

This is used for IPv4 Internet addresses.

`uint32_t` **ntohl** (`uint32_t` *netlong*) Function
This function converts the `uint32_t` integer *netlong* from network byte order to host byte order.

This is used for IPv4 Internet addresses.

5.6.6 Protocols Database

The communications protocol used with a socket controls low-level details of how data are exchanged. For example, the protocol implements things like checksums to detect errors in transmissions, and routing instructions for messages. Normal user programs have little reason to mess with these details directly.

The default communications protocol for the Internet namespace depends on the communication style. For stream communication, the default is TCP ("transmission control protocol"). For datagram communication, the default is UDP ("user datagram protocol"). For reliable datagram communication, the default is RDP ("reliable datagram protocol"). You should nearly always use the default.

Internet protocols are generally specified by a name instead of a number. The network protocols that a host knows about are stored in a database. This is usually either derived from the file '`/etc/protocols`', or it may be an equivalent provided by a name server. You look up the protocol number associated with a named protocol in the database using the `getprotobyname` function.

Here are detailed descriptions of the utilities for accessing the protocols database. These are declared in '`netdb.h`'.

struct protoent Data Type

This data type is used to represent entries in the network protocols database. It has the following members:

`char *p_name`
: This is the official name of the protocol.

`char **p_aliases`
: These are alternate names for the protocol, specified as an array of strings. The last element of the array is a null pointer.

`int p_proto`
: This is the protocol number (in host byte order); use this member as the *protocol* argument to `socket`.

You can use `getprotobyname` and `getprotobynumber` to search the protocols database for a specific protocol. The information is returned in a statically allocated structure; you must copy the information if you need to save it across calls.

`struct protoent *` **getprotobyname** (`const char *`*name*) Function

The `getprotobyname` function returns information about the network protocol named *name*. If there is no such protocol, it returns a null pointer.

`struct protoent *` **getprotobynumber** (`int` *protocol*) Function

The `getprotobynumber` function returns information about the network protocol with number *protocol*. If there is no such protocol, it returns a null pointer.

You can also scan the whole protocols database one protocol at a time by using `setprotoent`, `getprotoent` and `endprotoent`. Be careful when using these functions, because they are not reentrant.

`void` **setprotoent** (`int` *stayopen*) Function

This function opens the protocols database to begin scanning it.

If the *stayopen* argument is nonzero, this sets a flag so that subsequent calls to `getprotobyname` or `getprotobynumber` will not close the database (as they usually would). This makes for more efficiency if you call those functions several times, by avoiding reopening the database for each call.

`struct protoent *` **getprotoent** (`void`) Function

This function returns the next entry in the protocols database. It returns a null pointer if there are no more entries.

`void` **endprotoent** (`void`) Function

This function closes the protocols database.

5.6.7 Internet Socket Example

Here is an example showing how to create and name a socket in the Internet namespace. The newly created socket exists on the machine that the program is running on. Rather than finding and using the machine's Internet address, this example specifies `INADDR_ANY` as the host address; the system replaces that with the machine's actual address.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>

int
make_socket (uint16_t port)
{
  int sock;
  struct sockaddr_in name;

  /* Create the socket. */
  sock = socket (PF_INET, SOCK_STREAM, 0);
  if (sock < 0)
    {
      perror ("socket");
      exit (EXIT_FAILURE);
    }

  /* Give the socket a name. */
  name.sin_family = AF_INET;
  name.sin_port = htons (port);
  name.sin_addr.s_addr = htonl (INADDR_ANY);
```

```
  if (bind (sock, (struct sockaddr *) &name, sizeof (name)) < 0)
    {
      perror ("bind");
      exit (EXIT_FAILURE);
    }

  return sock;
}
```

Here is another example, showing how you can fill in a `sockaddr_in` structure, given a host name string and a port number:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

void
init_sockaddr (struct sockaddr_in *name,
               const char *hostname,
               uint16_t port)
{
  struct hostent *hostinfo;

  name->sin_family = AF_INET;
  name->sin_port = htons (port);
  hostinfo = gethostbyname (hostname);
  if (hostinfo == NULL)
    {
      fprintf (stderr, "Unknown host %s.\n", hostname);
      exit (EXIT_FAILURE);
    }
  name->sin_addr = *(struct in_addr *) hostinfo->h_addr;
}
```

5.7 Other Namespaces

Certain other namespaces and associated protocol families are supported but not documented yet because they are not often used. `PF_NS` refers to the Xerox Network Software protocols. `PF_ISO` stands for Open Systems Interconnect. `PF_CCITT` refers to protocols from CCITT. '`socket.h`' defines these symbols and other naming protocols not actually implemented.

PF_IMPLINK is used for communicating between hosts and Internet Message Processors.[1]

5.8 Opening and Closing Sockets

This section describes the actual library functions for opening and closing sockets. The same functions work for all namespaces and connection styles.

5.8.1 Creating a Socket

The primitive for creating a socket is the socket function, declared in 'sys/socket.h'.

int **socket** (int *namespace*, int *style*, int *protocol*) Function

This function creates a socket and specifies communication style *style*, which should be one of the socket styles listed in Section 5.2 [Communication Styles], page 126. The *namespace* argument specifies the namespace; it must be PF_LOCAL (see Section 5.5 [The Local Namespace], page 132) or PF_INET (see Section 5.6 [The Internet Namespace], page 134). *protocol* designates the specific protocol (see Section 5.1 [Socket Concepts], page 125); zero is usually right for *protocol*.

The return value from socket is the file descriptor for the new socket or -1 in case of error. The following errno error conditions are defined for this function:

EPROTONOSUPPORT
: The *protocol* or *style* is not supported by the *namespace* specified.

EMFILE
: The process already has too many file descriptors open.

ENFILE
: The system already has too many file descriptors open.

EACCES
: The process does not have the privilege to create a socket of the specified *style* or *protocol*.

ENOBUFS
: The system ran out of internal buffer space.

The file descriptor returned by the socket function supports both read and write operations. However, like pipes, sockets do not support file-positioning operations.

For examples of how to call the socket function, see Section 5.5.3 [Example of Local-Namespace Sockets], page 133, or Section 5.6.7 [Internet Socket Example], page 149.

---

[1] For information on this and PF_ROUTE, an occasionally used local-area routing protocol, see Marcus Brinkmann et al., *GNU Hurd Manual* (April 24, 2002), http://www.gnu.org/software/hurd/doc/hurd_toc.html.

5.8.2 Closing a Socket

When you have finished using a socket, you can simply close its file descriptor with `close` (see Section 2.1 [Opening and Closing Files], page 17). If there is still data waiting to be transmitted over the connection, normally `close` tries to complete this transmission. You can control this behavior using the `SO_LINGER` socket option to specify a time-out period (see Section 5.12 [Socket Options], page 173).

You can also shut down only reception or transmission on a connection by calling `shutdown`, which is declared in '`sys/socket.h`'.

`int` **shutdown** (`int` *socket*, `int` *how*) Function

The `shutdown` function shuts down the connection of socket *socket*. The argument *how* specifies what action to perform:

`0` Stop receiving data for this socket. If further data arrives, reject it.

`1` Stop trying to transmit data from this socket. Discard any data waiting to be sent. Stop looking for acknowledgement of data already sent; don't retransmit it if it is lost.

`2` Stop both reception and transmission.

The return value is `0` on success and `-1` on failure. The following `errno` error conditions are defined for this function:

`EBADF` *socket* is not a valid file-descriptor.

`ENOTSOCK` *socket* is not a socket.

`ENOTCONN` *socket* is not connected.

5.8.3 Socket Pairs

A *socket pair* consists of a pair of connected (but unnamed) sockets. It is very similar to a pipe and is used in much the same way. Socket pairs are created with the `socketpair` function, declared in '`sys/socket.h`'. A socket pair is much like a pipe; the main difference is that the socket pair is bidirectional, whereas the pipe has one input-only end and one output-only end (see Chapter 4 [Pipes and FIFOs], page 119).

`int` **socketpair** (`int` *namespace*, `int` *style*, `int` *protocol*, `int` *filedes*`[2]`) Function

This function creates a socket pair, returning the file descriptors in *filedes*`[0]` and *filedes*`[1]`. The socket pair is a full-duplex communications channel, so that both reading and writing may be performed at either end.

The *namespace*, *style* and *protocol* arguments are interpreted as for the `socket` function. *style* should be one of the communication styles listed in Section 5.2

[Communication Styles], page 126. The *namespace* argument specifies the namespace, which must be `AF_LOCAL` (see Section 5.5 [The Local Namespace], page 132); *protocol* specifies the communications protocol, but 0 is the only meaningful value.

If *style* specifies a connectionless communication style, then the two sockets you get are not *connected*, strictly speaking, but each of them knows the other as the default destination address, so they can send packets to each other.

The `socketpair` function returns `0` on success and `-1` on failure. The following `errno` error conditions are defined for this function:

`EMFILE` The process has too many file descriptors open.

`EAFNOSUPPORT`
: The specified namespace is not supported.

`EPROTONOSUPPORT`
: The specified protocol is not supported.

`EOPNOTSUPP`
: The specified protocol does not support the creation of socket pairs.

5.9 Using Sockets with Connections

The most common communication styles involve making a connection to a particular other socket, and then exchanging data with that socket over and over. Making a connection is asymmetric; one side (the *client*) acts to request a connection, while the other side (the *server*) makes a socket and waits for the connection request.

- *Connecting* (see Section 5.9.1 [Making a Connection], page 153) describes what the client program must do to initiate a connection with a server.
- *Listening* (see Section 5.9.2 [Listening for Connections], page 155) and *accepting connections* (see Section 5.9.3 [Accepting Connections], page 155) describe what the server program must do to wait for and act upon connection requests from clients.
- *Transferring data* (see Section 5.9.5 [Transferring Data], page 157) describes how data are transferred through the connected socket.

5.9.1 Making a Connection

In making a connection, the client makes a connection while the server waits for and accepts the connection. Here we discuss what the client program must do with the `connect` function, which is declared in '`sys/socket.h`'.

`int` **connect** (`int` *socket*, `struct sockaddr *`*addr*, `socklen_t` *length*) Function

The `connect` function initiates a connection from the socket with file descriptor *socket* to the socket whose address is specified by the *addr* and *length* argu-

ments. This socket is typically on another machine, and it must be already set up as a server. See Section 5.3 [Socket Addresses], page 127, for information about how these arguments are interpreted.

Normally, `connect` waits until the server responds to the request before it returns. You can set nonblocking mode on the socket *socket* to make `connect` return immediately without waiting for the response (see Section 2.14 [File Status Flags], page 59).

The normal return value from `connect` is `0`. If an error occurs, `connect` returns `-1`. The following `errno` error conditions are defined for this function:

`EBADF` The socket *socket* is not a valid file-descriptor.

`ENOTSOCK`
File descriptor *socket* is not a socket.

`EADDRNOTAVAIL`
The specified address is not available on the remote machine.

`EAFNOSUPPORT`
The namespace of the *addr* is not supported by this socket.

`EISCONN` The socket *socket* is already connected.

`ETIMEDOUT`
The attempt to establish the connection timed out.

`ECONNREFUSED`
The server has actively refused to establish the connection.

`ENETUNREACH`
The network of the given *addr* isn't reachable from this host.

`EADDRINUSE`
The socket address of the given *addr* is already in use.

`EINPROGRESS`
The socket *socket* is nonblocking and the connection could not be established immediately. You can determine when the connection is completely established with `select` (see Section 2.8 [Waiting for Input or Output], page 37). Another `connect` call on the same socket, before the connection is completely established, will fail with `EALREADY`.

`EALREADY`
The socket *socket* is nonblocking and already has a pending connection in progress (see `EINPROGRESS` above).

This function is defined as a cancellation point in multithreaded programs, so you have to be prepared for this and make sure that allocated resources (like memory, files descriptors, semaphores, etc.) are freed even if the thread is canceled.

5.9.2 Listening for Connections

Now let us consider what the server process must do to accept connections on a socket. First it must use the `listen` function to enable connection requests on the socket, then it must accept each incoming connection with a call to `accept` (see Section 5.9.3 [Accepting Connections], page 155). Once connection requests are enabled on a server socket, the `select` function reports when the socket has a connection ready to be accepted (see Section 2.8 [Waiting for Input or Output], page 37).

The `listen` function is not allowed for sockets using connectionless communication styles.

You can write a network server that does not even start running until a connection to it is requested (see Section 5.11.1 [`inetd` Servers], page 172).

In the Internet namespace, there are no special protection mechanisms for controlling access to a port; any process on any machine can make a connection to your server. If you want to restrict access to your server, make it examine the addresses associated with connection requests or implement some other handshaking or identification protocol.

In the local namespace, the ordinary file-protection bits control who has access to connect to the socket.

`int` **listen** (`int` *socket*, `unsigned int` *n*) Function

The `listen` function enables the socket *socket* to accept connections, thus making it a server socket.

The argument *n* specifies the length of the queue for pending connections. When the queue fills, new clients attempting to connect fail with `ECONNREFUSED` until the server calls `accept` to accept a connection from the queue.

The `listen` function returns `0` on success and `-1` on failure. The following `errno` error conditions are defined for this function:

`EBADF` The argument *socket* is not a valid file-descriptor.

`ENOTSOCK`
The argument *socket* is not a socket.

`EOPNOTSUPP`
The socket *socket* does not support this operation.

5.9.3 Accepting Connections

When a server receives a connection request, it can complete the connection by accepting the request. Use the function `accept` to do this.

A socket that has been established as a server can accept connection requests from multiple clients. The server's original socket *does not become part of the connection*; instead, `accept` makes a new socket that participates in the connection. `accept` returns the descriptor for this socket. The server's original socket remains available for listening for further connection requests.

The number of pending connection requests on a server socket is finite. If connection requests arrive from clients faster than the server can act upon them, the queue can fill up and additional requests are refused with an `ECONNREFUSED` error. You can specify the maximum length of this queue as an argument to the `listen` function, although the system may also impose its own internal limit on the length of this queue.

`int` **accept** (`int` *socket*, `struct sockaddr` `*`*addr*, `socklen_t` `*`*length_ptr*) Function

This function is used to accept a connection request on the server socket *socket*.

The `accept` function waits if there are no connections pending, unless the socket *socket* has nonblocking mode set. You can use `select` to wait for a pending connection, with a nonblocking socket (see Section 2.14 [File Status Flags], page 59, for information about nonblocking mode).

The *addr* and *length-ptr* arguments are used to return information about the name of the client socket that initiated the connection (see Section 5.3 [Socket Addresses], page 127, for information about the format).

Accepting a connection does not make *socket* part of the connection. Instead, it creates a new socket that becomes connected. The normal return value of `accept` is the file descriptor for the new socket.

After `accept`, the original socket *socket* remains open and unconnected, and continues listening until you close it. You can accept further connections with *socket* by calling `accept` again.

If an error occurs, `accept` returns `-1`. The following `errno` error conditions are defined for this function:

`EBADF` The *socket* argument is not a valid file-descriptor.

`ENOTSOCK`
The descriptor *socket* argument is not a socket.

`EOPNOTSUPP`
The descriptor *socket* does not support this operation.

`EWOULDBLOCK`
socket has nonblocking mode set, and there are no pending connections immediately available.

This function is defined as a cancellation point in multithreaded programs, so you have to be prepared for this and make sure that allocated resources (like memory, files descriptors, semaphores, etc.) are freed even if the thread is canceled.

The `accept` function is not allowed for sockets using connectionless communication styles.

5.9.4 Who Is Connected to Me?

`int` **getpeername** (`int` *socket*, `struct sockaddr` ⋆*addr*, `socklen_t` ⋆*length-ptr*) Function

The `getpeername` function returns the address of the socket that *socket* is connected to; it stores the address in the memory space specified by *addr* and *length-ptr*. It stores the length of the address in ⋆*length-ptr* (see Section 5.3 [Socket Addresses], page 127, for information about the format of the address). In some operating systems, `getpeername` works only for sockets in the Internet domain.

The return value is `0` on success and `-1` on error. The following `errno` error conditions are defined for this function:

`EBADF` The argument *socket* is not a valid file-descriptor.

`ENOTSOCK` The descriptor *socket* is not a socket.

`ENOTCONN` The socket *socket* is not connected.

`ENOBUFS` There are not enough internal buffers available.

5.9.5 Transferring Data

Once a socket has been connected to a peer, you can use the ordinary `read` and `write` operations to transfer data (see Section 2.2 [Input and Output Primitives], page 20). A socket is a two-way communications channel, so read and write operations can be performed at either end.

There are also some I/O modes that are specific to socket operations. In order to specify these modes, you must use the `recv` and `send` functions instead of the more generic `read` and `write` functions. The `recv` and `send` functions take an additional argument that you can use to specify various flags to control special I/O modes. For example, you can specify the `MSG_OOB` flag to read or write out-of-band data, the `MSG_PEEK` flag to peek at input, or the `MSG_DONTROUTE` flag to control inclusion of routing information on output.

5.9.5.1 Sending Data

The `send` function is declared in the header file '`sys/socket.h`'. If your *flags* argument is 0, you can just as well use `write` instead of `send` (see Section 2.2 [Input and Output Primitives], page 20). If the socket was connected but the connection has broken, you get a `SIGPIPE` signal for any use of `send` or `write` (see Section 17.2.7 [Miscellaneous Signals], page 387).

`int` **send** (`int` *socket*, `void *`*buffer*, `size_t` *size*, `int` *flags*) Function

The `send` function is like `write`, but with the additional flags *flags*. The possible values of *flags* are described in Section 5.9.5.3 [Socket Data Options], page 159.

This function returns the number of bytes transmitted or `-1` on failure. If the socket is nonblocking, then `send` (like `write`) can return after sending just part of the data (see Section 2.14 [File Status Flags], page 59, for information about nonblocking mode).

Note, however, that a successful return value merely indicates that the message has been sent without error, not necessarily that it has been received without error.

The following `errno` error conditions are defined for this function:

`EBADF` The *socket* argument is not a valid file-descriptor.

`EINTR` The operation was interrupted by a signal before any data was sent (see Section 17.5 [Primitives Interrupted by Signals], page 408).

`ENOTSOCK`
The descriptor *socket* is not a socket.

`EMSGSIZE`
The socket type requires that the message be sent atomically, but the message is too large for this to be possible.

`EWOULDBLOCK`
Nonblocking mode has been set on the socket, and the write operation would block. Normally, `send` blocks until the operation can be completed.

`ENOBUFS` There is not enough internal buffer space available.

`ENOTCONN`
You never connected this socket.

`EPIPE` This socket was connected, but the connection is now broken. In this case, `send` generates a `SIGPIPE` signal first; if that signal is ignored or blocked, or if its handler returns, then `send` fails with `EPIPE`.

This function is defined as a cancellation point in multithreaded programs, so you have to be prepared for this and make sure that allocated resources (like memory, files descriptors, semaphores, etc.) are freed even if the thread is canceled.

5.9.5.2 Receiving Data

The `recv` function is declared in the header file '`sys/socket.h`'. If your *flags* argument is 0, you can just as well use `read` instead of `recv` (see Section 2.2 [Input and Output Primitives], page 20).

`int` **recv** (`int` *socket*, `void *`*buffer*, `size_t` *size*, `int` *flags*) Function

The `recv` function is like `read`, but with the additional flags *flags*. The possible values of *flags* are described in Section 5.9.5.3 [Socket Data Options], page 159.

If nonblocking mode is set for *socket*, and no data are available to be read, `recv` fails immediately rather than waiting (see Section 2.14 [File Status Flags], page 59, for information about nonblocking mode).

This function returns the number of bytes received or `-1` on failure. The following `errno` error conditions are defined for this function:

`EBADF` The *socket* argument is not a valid file-descriptor.

`ENOTSOCK`
The descriptor *socket* is not a socket.

`EWOULDBLOCK`
Nonblocking mode has been set on the socket, and the read operation would block. Normally, `recv` blocks until there is input available to be read.

`EINTR` The operation was interrupted by a signal before any data was read (see Section 17.5 [Primitives Interrupted by Signals], page 408).

`ENOTCONN`
You never connected this socket.

This function is defined as a cancellation point in multithreaded programs, so you have to be prepared for this and make sure that allocated resources (like memory, files descriptors, semaphores, etc.) are freed even if the thread is canceled.

5.9.5.3 Socket Data Options

The *flags* argument to `send` and `recv` is a bit mask. You can bit-wise-OR the values of the following macros together to obtain a value for this argument. All are defined in the header file '`sys/socket.h`'.

`int` **MSG_OOB** Macro
Send or receive out-of-band data (see Section 5.9.8 [Out-of-Band Data], page 164).

`int` **MSG_PEEK** Macro
Look at the data, but don't remove it from the input queue. This is only meaningful with input functions such as `recv`, not with `send`.

`int` **MSG_DONTROUTE** Macro
Don't include routing information in the message. This is only meaningful with output operations, and is usually only of interest for diagnostic or routing programs. We don't try to explain it here.

5.9.6 Byte-Stream Socket Example

Here is an example client program that makes a connection for a byte-stream socket in the Internet namespace. It doesn't do anything particularly interesting once it has connected to the server; it just sends a text string to the server and exits.

This program uses `init_sockaddr` to set up the socket address (see Section 5.6.7 [Internet Socket Example], page 149).

```
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define PORT            5555
#define MESSAGE         "Yow!!! Are we having fun yet?!?"
#define SERVERHOST      "mescaline.gnu.org"

void
write_to_server (int filedes)
{
  int nbytes;

  nbytes = write (filedes, MESSAGE, strlen (MESSAGE) + 1);
  if (nbytes < 0)
    {
      perror ("write");
      exit (EXIT_FAILURE);
    }
}


int
main (void)
{
  extern void init_sockaddr (struct sockaddr_in *name,
                             const char *hostname,
                             uint16_t port);
  int sock;
  struct sockaddr_in servername;

  /* Create the socket.  */
```

```
  sock = socket (PF_INET, SOCK_STREAM, 0);
  if (sock < 0)
    {
      perror ("socket (client)");
      exit (EXIT_FAILURE);
    }

  /* Connect to the server. */
  init_sockaddr (&servername, SERVERHOST, PORT);
  if (0 > connect (sock,
                   (struct sockaddr *) &servername,
                   sizeof (servername)))
    {
      perror ("connect (client)");
      exit (EXIT_FAILURE);
    }

  /* Send data to the server. */
  write_to_server (sock);
  close (sock);
  exit (EXIT_SUCCESS);
}
```

5.9.7 Byte-Stream Connection Server Example

The server end is much more complicated. Since we want to allow multiple clients to be connected to the server at the same time, it would be incorrect to wait for input from a single client by simply calling `read` or `recv`. Instead, the right thing to do is to use `select` to wait for input on all of the open sockets (see Section 2.8 [Waiting for Input or Output], page 37). This also allows the server to deal with additional connection requests.

This particular server doesn't do anything interesting once it has gotten a message from a client. It does close the socket for that client when it detects an end-of-file condition (resulting from the client shutting down its end of the connection).

This program uses `make_socket` to set up the socket address (see Section 5.6.7 [Internet Socket Example], page 149).

```
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```
#include <netdb.h>

#define PORT    5555
#define MAXMSG  512

int
read_from_client (int filedes)
{
  char buffer[MAXMSG];
  int nbytes;

  nbytes = read (filedes, buffer, MAXMSG);
  if (nbytes < 0)
    {
      /* Read error */
      perror ("read");
      exit (EXIT_FAILURE);
    }
  else if (nbytes == 0)
    /* End of file */
    return -1;
  else
    {
      /* Data read */
      fprintf (stderr, "Server: got message: `%s'\n", buffer);
      return 0;
    }
}

int
main (void)
{
  extern int make_socket (uint16_t port);
  int sock;
  fd_set active_fd_set, read_fd_set;
  int i;
  struct sockaddr_in clientname;
  size_t size;

  /* Create the socket and set it up to accept connections. */
  sock = make_socket (PORT);
  if (listen (sock, 1) < 0)
    {
      perror ("listen");
```

```
      exit (EXIT_FAILURE);
    }

  /* Initialize the set of active sockets. */
  FD_ZERO (&active_fd_set);
  FD_SET (sock, &active_fd_set);

  while (1)
    {
      /* Block until input arrives on one or more active sockets. */
      read_fd_set = active_fd_set;
      if (select (FD_SETSIZE, &read_fd_set, NULL, NULL, NULL) < 0)
        {
          perror ("select");
          exit (EXIT_FAILURE);
        }

      /* Service all the sockets with input pending. */
      for (i = 0; i < FD_SETSIZE; ++i)
        if (FD_ISSET (i, &read_fd_set))
          {
            if (i == sock)
              {
                /* Connection request on original socket */
                int new;
                size = sizeof (clientname);
                new = accept (sock,
                              (struct sockaddr *) &clientname,
                              &size);
                if (new < 0)
                  {
                    perror ("accept");
                    exit (EXIT_FAILURE);
                  }
                fprintf (stderr,
                         "Server: connect from host %s, port %hd.\n",
                         inet_ntoa (clientname.sin_addr),
                         ntohs (clientname.sin_port));
                FD_SET (new, &active_fd_set);
              }
            else
              {
                /* Data arriving on an already connected socket */
                if (read_from_client (i) < 0)
```

```
                    {
                      close (i);
                      FD_CLR (i, &active_fd_set);
                    }
                }
            }
        }
    }
```

5.9.8 Out-of-Band Data

Streams with connections permit *out-of-band* data that is delivered with higher priority than ordinary data. Typically, the reason for sending out-of-band data is to send notice of an exceptional condition. To send out-of-band data, use `send`, specifying the flag `MSG_OOB` (see Section 5.9.5.1 [Sending Data], page 157).

Out-of-band data are received with higher priority because the receiving process need not read it in sequence; to read the next available out-of-band data, use `recv` with the `MSG_OOB` flag (see Section 5.9.5.2 [Receiving Data], page 158). Ordinary read operations do not read out-of-band data; they read only ordinary data.

When a socket finds that out-of-band data are on their way, it sends a `SIGURG` signal to the owner process or process group of the socket. You can specify the owner using the `F_SETOWN` command to the `fcntl` function (see Section 2.16 [Interrupt-Driven Input], page 68). You must also establish a handler for this signal, as described in Chapter 17 [Signal Handling], page 377, in order to take appropriate action such as reading the out-of-band data.

Alternatively, you can test for pending out-of-band data, or wait until there is out-of-band data, using the `select` function; it can wait for an exceptional condition on the socket (see Section 2.8 [Waiting for Input or Output], page 37).

Notification of out-of-band data (whether with `SIGURG` or with `select`) indicates that out-of-band data are on the way; the data may not actually arrive until later. If you try to read the out-of-band data before it arrives, `recv` fails with an `EWOULDBLOCK` error.

Sending out-of-band data automatically places a "mark" in the stream of ordinary data, showing where in the sequence the out-of-band data "would have been". This is useful when the meaning of out-of-band data is "cancel everything sent so far". Here is how you can test, in the receiving process, whether any ordinary data was sent before the mark:

```
success = ioctl (socket, SIOCATMARK, &atmark);
```

The `integer` variable *atmark* is set to a nonzero value if the socket's read pointer has reached the "mark".

Here's a function to discard any ordinary data preceding the out-of-band mark:

```
int
discard_until_mark (int socket)
```

```
{
  while (1)
    {
      /* This is not an arbitrary limit; any size will do.  */
      char buffer[1024];
      int atmark, success;

      /* If we have reached the mark, return.  */
      success = ioctl (socket, SIOCATMARK, &atmark);
      if (success < 0)
        perror ("ioctl");
      if (result)
        return;

      /* Otherwise, read a bunch of ordinary data and discard it.
         This is guaranteed not to read past the mark
         if it starts before the mark.  */
      success = read (socket, buffer, sizeof buffer);
      if (success < 0)
        perror ("read");
    }
}
```

If you don't want to discard the ordinary data preceding the mark, you may need to read some of it anyway, to make room in internal system buffers for the out-of-band data. If you try to read out-of-band data and get an `EWOULDBLOCK` error, try reading some ordinary data (saving it so that you can use it when you want it) and see if that makes room. Here is an example:

```
struct buffer
{
  char *buf;
  int size;
  struct buffer *next;
};

/* Read the out-of-band data from SOCKET and return it
   as a 'struct buffer', which records the address of the data
   and its size.

   It may be necessary to read some ordinary data
   in order to make room for the out-of-band data.
   If so, the ordinary data are saved as a chain of buffers
   found in the 'next' field of the value.  */

struct buffer *
```

```
read_oob (int socket)
{
  struct buffer *tail = 0;
  struct buffer *list = 0;

  while (1)
    {
      /* This is an arbitrary limit.
         Does anyone know how to do this without a limit?  */
#define BUF_SZ 1024
      char *buf = (char *) xmalloc (BUF_SZ);
      int success;
      int atmark;

      /* Try again to read the out-of-band data.  */
      success = recv (socket, buf, BUF_SZ, MSG_OOB);
      if (success >= 0)
        {
          /* We got it, so return it.  */
          struct buffer *link
            = (struct buffer *) xmalloc (sizeof (struct buffer));
          link->buf = buf;
          link->size = success;
          link->next = list;
          return link;
        }

      /* If we fail, see if we are at the mark.  */
      success = ioctl (socket, SIOCATMARK, &atmark);
      if (success < 0)
        perror ("ioctl");
      if (atmark)
        {
          /* At the mark; skipping past more ordinary data cannot help.
             So just wait a while.  */
          sleep (1);
          continue;
        }

      /* Otherwise, read a bunch of ordinary data and save it.
         This is guaranteed not to read past the mark
         if it starts before the mark.  */
      success = read (socket, buf, BUF_SZ);
      if (success < 0)
```

```
            perror ("read");

          /* Save this data in the buffer list.  */
          {
            struct buffer *link
              = (struct buffer *) xmalloc (sizeof (struct buffer));
            link->buf = buf;
            link->size = success;

            /* Add the new link to the end of the list.  */
            if (tail)
              tail->next = link;
            else
              list = link;
            tail = link;
          }
        }
    }
```

5.10 Datagram Socket Operations

This section describes how to use communication styles that don't use connections (styles `SOCK_DGRAM` and `SOCK_RDM`). Using these styles, you group data into packets, and each packet is an independent communication. You specify the destination for each packet individually.

Datagram packets are like letters—you send each one independently with its own destination address, and they may arrive in the wrong order or not at all.

The `listen` and `accept` functions are not allowed for sockets using connectionless communication styles.

5.10.1 Sending Datagrams

The normal way of sending data on a datagram socket is by using the `sendto` function, declared in '`sys/socket.h`'.

You can call `connect` on a datagram socket, but this only specifies a default destination for further data transmission on the socket. When a socket has a default destination, you can use `send` (see Section 5.9.5.1 [Sending Data], page 157) or even `write` (see Section 2.2 [Input and Output Primitives], page 20) to send a packet there. You can cancel the default destination by calling `connect` using an address format of `AF_UNSPEC` in the *addr* argument (see Section 5.9.1 [Making a Connection], page 153, for more information about the `connect` function).

`int` **sendto** (`int` *socket*, `void *`*buffer*`. size_t` *size*, `int` *flags*, `struct sockaddr *`*addr*, `socklen_t` *length*) Function

The `sendto` function transmits the data in the *buffer* through the socket *socket* to the destination address specified by the *addr* and *length* arguments. The *size* argument specifies the number of bytes to be transmitted.

The *flags* are interpreted the same way as for `send` (see Section 5.9.5.3 [Socket Data Options], page 159).

The return value and error conditions are also the same as for `send`, but you cannot rely on the system to detect errors and report them; the most common error is that the packet is lost or there is no one at the specified address to receive it, and the operating system on your machine usually does not know this.

It is also possible for one call to `sendto` to report an error owing to a problem related to a previous call.

This function is defined as a cancellation point in multithreaded programs, so you have to be prepared for this and make sure that allocated resources (like memory, files descriptors, semaphores, etc.) are freed even if the thread is canceled.

5.10.2 Receiving Datagrams

The `recvfrom` function reads a packet from a datagram socket and also tells you where it was sent from. This function is declared in '`sys/socket.h`'.

`int` **recvfrom** (`int` *socket*, `void *`*buffer*, `size_t` *size*, `int` *flags*, `struct sockaddr *`*addr*, `socklen_t *`*length-ptr*) Function

The `recvfrom` function reads one packet from the socket *socket* into the buffer *buffer*. The *size* argument specifies the maximum number of bytes to be read.

If the packet is longer than *size* bytes, then you get the first *size* bytes of the packet and the rest of the packet is lost. There's no way to read the rest of the packet. Thus, when you use a packet protocol, you must always know how long of a packet to expect.

The *addr* and *length-ptr* arguments are used to return the address where the packet came from (see Section 5.3 [Socket Addresses], page 127). For a socket in the local domain, the address information won't be meaningful, since you can't read the address of such a socket (see Section 5.5 [The Local Namespace], page 132). You can specify a null pointer as the *addr* argument if you are not interested in this information.

The *flags* are interpreted the same way as for `recv` (see Section 5.9.5.3 [Socket Data Options], page 159). The return value and error conditions are also the same as for `recv`.

This function is defined as a cancellation point in multithreaded programs, so you have to be prepared for this and make sure that allocated resources (like memory, files descriptors, semaphores, etc.) are freed even if the thread is canceled.

You can use plain `recv` (see Section 5.9.5.2 [Receiving Data], page 158) instead of `recvfrom` if you don't need to find out who sent the packet (either because you know where it should come from or because you treat all possible senders alike). Even `read` can be used if you don't want to specify *flags* (see Section 2.2 [Input and Output Primitives], page 20).

5.10.3 Datagram Socket Example

Here is a set of example programs that send messages over a datagram stream in the local namespace. Both the client and server programs use the `make_named_socket` function that was presented in Section 5.5.3 [Example of Local-Namespace Sockets], page 133, to create and name their sockets.

First, here is the server program. It sits in a loop waiting for messages to arrive, bouncing each message back to the sender. Obviously this isn't a particularly useful program, but it does show the general ideas involved.

```
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/un.h>

#define SERVER  "/tmp/serversocket"
#define MAXMSG  512

int
main (void)
{
  int sock;
  char message[MAXMSG];
  struct sockaddr_un name;
  size_t size;
  int nbytes;

  /* Remove the file name first; it's ok if the call fails */
  unlink (SERVER);

  /* Make the socket, then loop endlessly. */
  sock = make_named_socket (SERVER);
  while (1)
    {
      /* Wait for a datagram. */
      size = sizeof (name);
      nbytes = recvfrom (sock, message, MAXMSG, 0,
                         (struct sockaddr *) & name, &size);
```

```
      if (nbytes < 0)
        {
          perror ("recfrom (server)");
          exit (EXIT_FAILURE);
        }

      /* Give a diagnostic message.  */
      fprintf (stderr, "Server: got message: %s\n", message);

      /* Bounce the message back to the sender.  */
      nbytes = sendto (sock, message, nbytes, 0,
                       (struct sockaddr *) & name, size);
      if (nbytes < 0)
        {
          perror ("sendto (server)");
          exit (EXIT_FAILURE);
        }
    }
}
```

5.10.4 Example of Reading Datagrams

Here is the client program corresponding to the server above.

It sends a datagram to the server and then waits for a reply. Notice that the socket for the client (as well as for the server) in this example has to be given a name. This is so that the server can direct a message back to the client. Since the socket has no associated connection state, the only way the server can do this is by referencing the name of the client.

```
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/un.h>

#define SERVER  "/tmp/serversocket"
#define CLIENT  "/tmp/mysocket"
#define MAXMSG  512
#define MESSAGE "Yow!!! Are we having fun yet?!?"

int
main (void)
{
```

```
  extern int make_named_socket (const char *name);
  int sock;
  char message[MAXMSG];
  struct sockaddr_un name;
  size_t size;
  int nbytes;

  /* Make the socket. */
  sock = make_named_socket (CLIENT);

  /* Initialize the server socket address. */
  name.sun_family = AF_LOCAL;
  strcpy (name.sun_path, SERVER);
  size = strlen (name.sun_path) + sizeof (name.sun_family);

  /* Send the datagram. */
  nbytes = sendto (sock, MESSAGE, strlen (MESSAGE) + 1, 0,
                   (struct sockaddr *) & name, size);
  if (nbytes < 0)
    {
      perror ("sendto (client)");
      exit (EXIT_FAILURE);
    }

  /* Wait for a reply. */
  nbytes = recvfrom (sock, message, MAXMSG, 0, NULL, 0);
  if (nbytes < 0)
    {
      perror ("recfrom (client)");
      exit (EXIT_FAILURE);
    }

  /* Print a diagnostic message. */
  fprintf (stderr, "Client: got message: %s\n", message);

  /* Clean up. */
  remove (CLIENT);
  close (sock);
}
```

Keep in mind that datagram socket communications are unreliable. In this example, the client program waits indefinitely if the message never reaches the server or if the server's response never comes back. It's up to the user running the program to kill and restart it if desired. A more automatic solution could be to use `select`

(see Section 2.8 [Waiting for Input or Output], page 37), to establish a time-out period for the reply, and in case of time-out either resend the message or shut down the socket and exit.

5.11 The `inetd` Daemon

We've explained above how to write a server program that does its own listening. Such a server must already be running in order for anyone to connect to it.

Another way to provide a service on an Internet port is to let the daemon program `inetd` do the listening. `inetd` is a program that runs all the time and waits (using `select`) for messages on a specified set of ports. When it receives a message, it accepts the connection (if the socket style calls for connections) and then forks a child process to run the corresponding server program. You specify the ports and their programs in the file '`/etc/inetd.conf`'.

5.11.1 `inetd` Servers

Writing a server program to be run by `inetd` is very simple. Each time someone requests a connection to the appropriate port, a new server process starts. The connection already exists at this time; the socket is available as the standard input descriptor and as the standard output descriptor (descriptors 0 and 1) in the server process. Thus the server program can begin reading and writing data right away. Often the program needs only the ordinary I/O facilities; in fact, a general-purpose filter program that knows nothing about sockets can work as a byte-stream server run by `inetd`.

You can also use `inetd` for servers that use connectionless communication styles. For these servers, `inetd` does not try to accept a connection since no connection is possible. It just starts the server program, which can read the incoming datagram packet from descriptor 0. The server program can handle one request and then exit, or you can choose to write it to keep reading more requests until no more arrive, and then exit. You must specify which of these two techniques the server uses when you configure `inetd`.

5.11.2 Configuring `inetd`

The file '`/etc/inetd.conf`' tells `inetd` which ports to listen to and what server programs to run for them. Normally, each entry in the file is one line, but you can split it onto multiple lines provided all but the first line of the entry start with white space. Lines that start with '`#`' are comments.

Here are two standard entries in '`/etc/inetd.conf`':

```
ftp stream tcp nowait root /libexec/ftpd ftpd
talk dgram udp wait root /libexec/talkd talkd
```

An entry has this format:

service style protocol wait user program arguments

The *service* field says which service this program provides. It should be the name of a service defined in '`/etc/services`'. `inetd` uses *service* to decide which port to listen on for this entry.

The fields *style* and *protocol* specify the communication style and the protocol to use for the listening socket. The style should be the name of a communication style, converted to lower case and with '`SOCK_`' deleted—for example, '`stream`' or '`dgram`'. *protocol* should be one of the protocols listed in '`/etc/protocols`'. The typical protocol names are '`tcp`' for byte-stream connections and '`udp`' for unreliable datagrams.

The *wait* field should be either '`wait`' or '`nowait`'. Use '`wait`' if *style* is a connectionless style and the server, once started, handles multiple requests as they come in. Use '`nowait`' if `inetd` should start a new process for each message or request that comes in. If *style* uses connections, then *wait* **must** be '`nowait`'.

user is the user name that the server should run as. `inetd` runs as root, so it can set the user ID of its children arbitrarily. It's best to avoid using '`root`' for *user* if you can; but some servers, such as Telnet and FTP, read a username and password themselves. These servers need to be root initially so they can log in as commanded by the data coming over the network.

program together with *arguments* specifies the command to run to start the server. *program* should be an absolute file name specifying the executable file to run. *arguments* consists of any number of white-space-separated words, which become the command-line arguments of *program*. The first word in *arguments* is argument zero, which should by convention be the program name itself (sans directories).

If you edit '`/etc/inetd.conf`', you can tell `inetd` to reread the file and obey its new contents by sending the `inetd` process the `SIGHUP` signal. You'll have to use `ps` to determine the process ID of the `inetd` process, since it is not fixed.

5.12 Socket Options

This section describes how to read or set various options that modify the behavior of sockets and their underlying communications protocols.

When you are manipulating a socket option, you must specify which *level* the option pertains to. This describes whether the option applies to the socket interface, or to a lower-level communications protocol interface.

5.12.1 Socket Option Functions

Here are the functions for examining and modifying socket options. They are declared in '`sys/socket.h`'.

`int` **getsockopt** (`int` *socket*, `int` *level*, `int` *optname*, `void` `*`*optval*, `socklen_t *`*optlen-ptr*) Function

The `getsockopt` function gets information about the value of option *optname* at level *level* for socket *socket*.

The option value is stored in a buffer that *optval* points to. Before the call, you should supply in `*`*optlen-ptr* the size of this buffer. On return, it contains the number of bytes of information actually stored in the buffer.

Most options interpret the *optval* buffer as a single `int` value.

The actual return value of `getsockopt` is `0` on success and `-1` on failure. The following `errno` error conditions are defined:

`EBADF` The *socket* argument is not a valid file-descriptor.

`ENOTSOCK`
The descriptor *socket* is not a socket.

`ENOPROTOOPT`
The *optname* doesn't make sense for the given *level*.

`int` **setsockopt** (`int` *socket*, `int` *level*, `int` *optname*, `void` `*`*optval*, `socklen_t` *optlen*) Function

This function is used to set the socket option *optname* at level *level* for socket *socket*. The value of the option is passed in the buffer *optval* of size *optlen*.

The return value and error codes for `setsockopt` are the same as for `getsockopt`.

5.12.2 Socket-Level Options

`int` **SOL_SOCKET** Constant

Use this constant as the *level* argument to `getsockopt` or `setsockopt` to manipulate the socket-level options described in this section.

Here is a table of socket-level option names; all are defined in the header file '`sys/socket.h`'.

`SO_DEBUG`
This option toggles recording of debugging information in the underlying protocol modules. The value has type `int`; a nonzero value means "yes".

`SO_REUSEADDR`
This option controls whether `bind` (see Section 5.3.2 [Setting the Address of a Socket], page 129) should permit reuse of local addresses for this socket. If you enable this option, you can actually have two sockets with the same Internet port number; but the system won't allow you to use the two identically named sockets in a way that would

confuse the Internet. The reason for this option is that some higher-level Internet protocols, including FTP, require you to keep reusing the same port number.

The value has type `int`; a nonzero value means "yes".

`SO_KEEPALIVE`
: This option controls whether the underlying protocol should periodically transmit messages on a connected socket. If the peer fails to respond to these messages, the connection is considered broken. The value has type `int`; a nonzero value means "yes".

`SO_DONTROUTE`
: This option controls whether outgoing messages bypass the normal message routing facilities. If set, messages are sent directly to the network interface instead. The value has type `int`; a nonzero value means "yes".

`SO_LINGER`
: This option specifies what should happen when the socket of a type that promises reliable delivery still has untransmitted messages when it is closed (see Section 5.8.2 [Closing a Socket], page 152). The value has type `struct linger`.

struct linger Data Type

This structure type has the following members:

`int l_onoff`
: This field is interpreted as a Boolean. If nonzero, `close` blocks until the data are transmitted, or the time-out period has expired.

`int l_linger`
: This specifies the time-out period, in seconds.

`SO_BROADCAST`
: This option controls whether datagrams may be broadcast from the socket. The value has type `int`; a nonzero value means "yes".

`SO_OOBINLINE`
: If this option is set, out-of-band data received on the socket is placed in the normal input queue. This permits it to be read using `read` or `recv` without specifying the `MSG_OOB` flag (see Section 5.9.8 [Out-of-Band Data], page 164). The value has type `int`; a nonzero value means "yes".

`SO_SNDBUF`
: This option gets or sets the size of the output buffer. The value is a `size_t`, which is the size in bytes.

`SO_RCVBUF`
: This option gets or sets the size of the input buffer. The value is a `size_t`, which is the size in bytes.

`SO_STYLE`
`SO_TYPE`
: This option can be used with `getsockopt` only. It is used to get the socket's communication style. `SO_TYPE` is the historical name, and `SO_STYLE` is the preferred name in GNU. The value has type `int` and its value designates a communication style (see Section 5.2 [Communication Styles], page 126).

`SO_ERROR`
: This option can be used with `getsockopt` only. It is used to reset the error status of the socket. The value is an `int`, which represents the previous error status.

5.13 Networks Database

Many systems come with a database that records a list of networks known to the system developer. This is usually kept either in the file '`/etc/networks`' or in an equivalent from a name server. This database is useful for routing programs such as `route`, but it is not useful for programs that simply communicate over the network. We provide functions to access this database, which are declared in '`netdb.h`'.

struct netent Data Type

This data type is used to represent information about entries in the networks database. It has the following members:

`char *n_name`
: This is the "official" name of the network.

`char **n_aliases`
: These are alternative names for the network, represented as a vector of strings. A null pointer terminates the array.

`int n_addrtype`
: This is the type of the network number; this is always equal to `AF_INET` for Internet networks.

`unsigned long int n_net`
: This is the network number. Network numbers are returned in host byte order (see Section 5.6.5 [Byte-Order Conversion], page 147).

Use the `getnetbyname` or `getnetbyaddr` functions to search the networks database for information about a specific network. The information is returned in a statically allocated structure; you must copy the information if you need to save it.

`struct netent *` **getnetbyname** (`const char *`*name*) Function

The `getnetbyname` function returns information about the network named *name*. It returns a null pointer if there is no such network.

`struct netent *` **getnetbyaddr** (`unsigned long int` *net*, `int` *type*) Function

The `getnetbyaddr` function returns information about the network of type *type* with number *net*. You should specify a value of `AF_INET` for the *type* argument for Internet networks.

`getnetbyaddr` returns a null pointer if there is no such network.

You can also scan the networks database using `setnetent`, `getnetent` and `endnetent`. Be careful when using these functions, because they are not reentrant.

`void` **setnetent** (`int` *stayopen*) Function

This function opens and rewinds the networks database.

If the *stayopen* argument is nonzero, this sets a flag so that subsequent calls to `getnetbyname` or `getnetbyaddr` will not close the database (as they usually would). This makes for more efficiency if you call those functions several times, by avoiding reopening the database for each call.

`struct netent *` **getnetent** (`void`) Function

This function returns the next entry in the networks database. It returns a null pointer if there are no more entries.

`void` **endnetent** (`void`) Function

This function closes the networks database.

6 Low-Level Terminal Interface

This chapter describes functions that are specific to terminal devices. You can use these functions to do things like turn off input echoing; set serial-line characteristics, such as line speed and flow control; and change which characters are used for end of file, command-line editing, sending signals and similar control functions.

Most of the functions in this chapter operate on file descriptors. See Chapter 2 [Low-Level Input/Output], page 17, for more information about what a file descriptor is and how to open a file descriptor for a terminal device.

6.1 Identifying Terminals

The functions described in this chapter only work on files that correspond to terminal devices. You can find out whether a file descriptor is associated with a terminal by using the `isatty` function.

Prototypes for the functions in this section are declared in the header file '`unistd.h`'.

`int` **isatty** (`int` *filedes*) Function

This function returns `1` if *filedes* is a file descriptor associated with an open terminal device and 0 otherwise.

If a file descriptor is associated with a terminal, you can get its associated filename using the `ttyname` function. See also the `ctermid` function, described in Section 8.7.1 [Identifying the Controlling Terminal], page 238.

`char *` **ttyname** (`int` *filedes*) Function

If the file descriptor *filedes* is associated with a terminal device, the `ttyname` function returns a pointer to a statically allocated, null-terminated string containing the file name of the terminal file. The value is a null pointer if the file descriptor isn't associated with a terminal, or if the file name cannot be determined.

`int` **ttyname_r** (`int` *filedes*, `char *`*buf*, `size_t` *len*) Function

The `ttyname_r` function is similar to the `ttyname` function except that it places its result into the user-specified buffer starting at *buf* with length *len*.

The normal return value from `ttyname_r` is 0. Otherwise, an error number is returned to indicate the error. The following `errno` error conditions are defined for this function:

`EBADF` The *filedes* argument is not a valid file-descriptor.

`ENOTTY` The *filedes* is not associated with a terminal.

`ERANGE` The buffer length *len* is too small to store the string to be returned.

6.2 I/O Queues

Many of the remaining functions in this section refer to the input and output queues of a terminal device. These queues implement a form of buffering *within the kernel* independent of the buffering implemented by I/O streams.[1]

The *terminal input queue* is also sometimes referred to as its *typeahead buffer*. It holds the characters that have been received from the terminal but not yet read by any process.

The size of the input queue is described by the `MAX_INPUT` and `_POSIX_MAX_INPUT` parameters (see Section 12.6 [Limits on File-System Capacity], page 318). You are guaranteed a queue size of at least `MAX_INPUT`, but the queue might be larger, and might even dynamically change size. If input flow control is enabled by setting the `IXOFF` input-mode bit (see Section 6.4.4 [Input Modes], page 185), the terminal driver transmits STOP and START characters to the terminal when necessary to prevent the queue from overflowing. Otherwise, input may be lost if it comes in too fast from the terminal. In canonical mode, all input stays in the queue until a newline character is received, so the terminal input queue can fill up when you type a very long line (see Section 6.3 [Two Styles of Input: Canonical or Not], page 180).

The *terminal output queue* is like the input queue, but for output; it contains characters that have been written by processes, but not yet transmitted to the terminal. If output flow control is enabled by setting the `IXON` input-mode bit (see Section 6.4.4 [Input Modes], page 185), the terminal driver obeys START and STOP characters sent by the terminal to stop and restart transmission of output.

Clearing the terminal input queue means discarding any characters that have been received but not yet read. Similarly, clearing the terminal output queue means discarding any characters that have been written but not yet transmitted.

6.3 Two Styles of Input: Canonical or Not

POSIX systems support two basic modes of input: canonical and noncanonical.

In *canonical-input processing* mode, terminal input is processed in lines terminated by newline (`'\n'`), `EOF`, or `EOL` characters. No input can be read until an entire line has been typed by the user, and the `read` function (see Section 2.2 [Input and Output Primitives], page 20) returns at most a single line of input, no matter how many bytes are requested.

In canonical-input mode, the operating system provides input-editing facilities—some characters are interpreted specially to perform editing operations within the current line of text, such as ERASE and KILL (see Section 6.4.9.1 [Characters for Input Editing], page 194).

The constants `_POSIX_MAX_CANON` and `MAX_CANON` parameterize the maximum number of bytes that may appear in a single line of canonical input (see Section 12.6 [Limits on File-System Capacity], page 318). You are guaranteed a

[1] See Loosemore, et al., "Input/Output on Streams" (see chap. 1, n. 1).

maximum line length of at least `MAX_CANON` bytes, but the maximum might be larger, and might even dynamically change size.

In *noncanonical-input processing* mode, characters are not grouped into lines, and ERASE and KILL processing is not performed. The granularity with which bytes are read in noncanonical input mode is controlled by the MIN and TIME settings (see Section 6.4.10 [Noncanonical Input], page 198).

Most programs use canonical-input mode, because this gives the user a way to edit input line by line. The usual reason to use noncanonical mode is when the program accepts single-character commands or provides its own editing facilities.

The choice of canonical- or noncanonical- input is controlled by the `ICANON` flag in the `c_lflag` member of `struct termios` (see Section 6.4.7 [Local Modes], page 189).

6.4 Terminal Modes

This section describes the various terminal attributes that control how input and output are done. The functions, data structures and symbolic constants are all declared in the header file '`termios.h`'.

Don't confuse terminal attributes with file attributes. A device special file that is associated with a terminal has file attributes as described in Section 3.9 [File Attributes], page 93. These are unrelated to the attributes of the terminal device itself, which are discussed in this section.

6.4.1 Terminal Mode Data Types

The entire collection of attributes of a terminal is stored in a structure of type `struct termios`. This structure is used with the functions `tcgetattr` and `tcsetattr` to read and set the attributes.

struct termios Data Type

This is a structure that records all the I/O attributes of a terminal. The structure includes at least the following members:

`tcflag_t c_iflag`
: This is a bit mask specifying flags for input modes (see Section 6.4.4 [Input Modes], page 185).

`tcflag_t c_oflag`
: This is a bit mask specifying flags for output modes (see Section 6.4.5 [Output Modes], page 187).

`tcflag_t c_cflag`
: This is a bit mask specifying flags for control modes (see Section 6.4.6 [Control Modes], page 187).

`tcflag_t c_lflag`
: This is a bit mask specifying flags for local modes (see Section 6.4.7 [Local Modes], page 189).

`cc_t c_cc[NCCS]`
: This is an array specifying which characters are associated with various control functions (see Section 6.4.9 [Special Characters], page 194).

The `struct termios` structure also contains members that encode input and output transmission speeds, but the representation is not specified. See Section 6.4.8 [Line Speed], page 192, for how to examine and store the speed values.

The following sections describe the details of the members of the `struct termios` structure:

tcflag_t Data Type
This is an unsigned integer type used to represent the various bit masks for terminal flags.

cc_t Data Type
This is an unsigned integer type used to represent characters associated with various terminal-control functions.

`int` **NCCS** Macro
The value of this macro is the number of elements in the `c_cc` array.

6.4.2 Terminal Mode Functions

`int` **tcgetattr** (`int` *filedes*, `struct termios *`*termios-p*) Function
This function is used to examine the attributes of the terminal device with file descriptor *filedes*. The attributes are returned in the structure that *termios-p* points to.

If successful, `tcgetattr` returns 0. A return value of −1 indicates an error. The following `errno` error conditions are defined for this function:

`EBADF` The *filedes* argument is not a valid file-descriptor.

`ENOTTY` The *filedes* is not associated with a terminal.

`int` **tcsetattr** (`int` *filedes*, `int` *when*, `const struct termios *`*termios-p*) Function
This function sets the attributes of the terminal device with file descriptor *filedes*. The new attributes are taken from the structure that *termios-p* points to.

The *when* argument specifies how to deal with input and output already queued. It can be one of the following values:

`TCSANOW` Make the change immediately.

`TCSADRAIN`
: Make the change after waiting until all queued output has been written. You should usually use this option when changing parameters that affect output.

`TCSAFLUSH`
: This is like `TCSADRAIN`, but also discards any queued input.

`TCSASOFT`
: This is a flag bit that you can add to any of the above alternatives. Its meaning is to inhibit alteration of the state of the terminal hardware. It is a BSD extension; it is only supported on BSD systems and the GNU system.

  Using `TCSASOFT` is exactly the same as setting the `CIGNORE` bit in the `c_cflag` member of the structure *termios-p* points to. See Section 6.4.6 [Control Modes], page 187, for a description of `CIGNORE`.

If this function is called from a background process on its controlling terminal, normally all processes in the process group are sent a `SIGTTOU` signal, in the same way as if the process were trying to write to the terminal. The exception is if the calling process itself is ignoring or blocking `SIGTTOU` signals, in which case the operation is performed and no signal is sent (see Chapter 8 [Job Control], page 221).

If successful, `tcsetattr` returns 0. A return value of -1 indicates an error. The following `errno` error conditions are defined for this function:

`EBADF` The *filedes* argument is not a valid file-descriptor.

`ENOTTY` The *filedes* is not associated with a terminal.

`EINVAL` Either the value of the `when` argument is not valid, or there is something wrong with the data in the *termios-p* argument.

Although `tcgetattr` and `tcsetattr` specify the terminal device with a file descriptor, the attributes are those of the terminal device itself and not of the file descriptor. This means that the effects of changing terminal attributes are persistent; if another process opens the terminal file later on, it will see the changed attributes, even though it doesn't have anything to do with the open file descriptor you originally specified in changing the attributes.

Similarly, if a single process has multiple or duplicated file descriptors for the same terminal device, changing the terminal attributes affects input and output to all of these file descriptors. This means, for example, that you can't open one file descriptor or stream to read from a terminal in the normal line-buffered, echoed mode; and simultaneously have another file descriptor for the same terminal that you use to read from it in single-character, nonechoed mode. Instead, you have to explicitly switch the terminal back and forth between the two modes.

6.4.3 Setting Terminal Modes Properly

When you set terminal modes, you should call `tcgetattr` first to get the current modes of the particular terminal device, modify only those modes that you are really interested in, and store the result with `tcsetattr`.

It's a bad idea to simply initialize a `struct termios` structure to a chosen set of attributes and pass it directly to `tcsetattr`. Your program may be run years from now, on systems that support members not documented in this manual. The way to avoid setting these members to unreasonable values is to avoid changing them.

What's more, different terminal devices may require different mode settings in order to function properly. So you should avoid blindly copying attributes from one terminal device to another.

When a member contains a collection of independent flags, as the `c_iflag`, `c_oflag` and `c_cflag` members do, even setting the entire member is a bad idea, because particular operating systems have their own flags. Instead, you should start with the current value of the member and alter only the flags whose values matter in your program, leaving any other flags unchanged.

Here is an example of how to set one flag (`ISTRIP`) in the `struct termios` structure while properly preserving all the other data in the structure:

```
int
set_istrip (int desc, int value)
{
  struct termios settings;
  int result;



  result = tcgetattr (desc, &settings);
  if (result < 0)
    {
      perror ("error in tcgetattr");
      return 0;
    }


  settings.c_iflag &= ~ISTRIP;
  if (value)
    settings.c_iflag |= ISTRIP;


  result = tcsetattr (desc, TCSANOW, &settings);
  if (result < 0)
    {
      perror ("error in tcsetattr");
      return 0;
   }
  return 1;
```

```
}
```

6.4.4 Input Modes

This section describes the terminal attribute flags that control fairly low-level aspects of input processing: handling of parity errors, break signals, flow control and RET and LFD characters.

All of these flags are bits in the `c_iflag` member of the `struct termios` structure. The member is an integer, and you change flags using the operators `&`, `|` and `^`. Don't try to specify the entire value for `c_iflag`—instead, change only specific flags and leave the rest untouched (see Section 6.4.3 [Setting Terminal Modes Properly], page 183).

`tcflag_t` **INPCK** Macro

If this bit is set, input parity-checking is enabled. If it is not set, no checking at all is done for parity errors on input; the characters are simply passed through to the application.

Parity checking on input processing is independent of whether parity detection and generation on the underlying terminal hardware is enabled (see Section 6.4.6 [Control Modes], page 187). For example, you could clear the `INPCK` input-mode flag and set the `PARENB` control-mode flag to ignore parity errors on input, but still generate parity on output.

If this bit is set, what happens when a parity error is detected depends on whether the `IGNPAR` or `PARMRK` bits are set. If neither of these bits are set, a byte with a parity error is passed to the application as a `'\0'` character.

`tcflag_t` **IGNPAR** Macro

If this bit is set, any byte with a framing or parity error is ignored. This is only useful if `INPCK` is also set.

`tcflag_t` **PARMRK** Macro

If this bit is set, input bytes with parity or framing errors are marked when passed to the program. This bit is meaningful only when `INPCK` is set and `IGNPAR` is not set.

The way erroneous bytes are marked is with 2 preceding bytes, `377` and `0`. Thus, the program actually reads 3 bytes for 1 erroneous byte received from the terminal.

If a valid byte has the value `0377` and `ISTRIP` (see below) is not set, the program might confuse it with the prefix that marks a parity error. So a valid byte `0377` is passed to the program as 2 bytes, `0377 0377`, in this case.

`tcflag_t` **ISTRIP** Macro

If this bit is set, valid input bytes are stripped to 7 bits. Otherwise, all 8 bits are available for programs to read.

`tcflag_t` **IGNBRK** Macro

If this bit is set, break conditions are ignored.

A *break condition* is defined in the context of asynchronous serial data transmission as a series of 0-value bits longer than a single byte.

`tcflag_t` **BRKINT** Macro

If this bit is set and `IGNBRK` is not set, a break condition clears the terminal input and output queues and raises a `SIGINT` signal for the foreground process group associated with the terminal.

If neither `BRKINT` nor `IGNBRK` are set, a break condition is passed to the application as a single '`\0`' character if `PARMRK` is not set, or otherwise as a three-character sequence '`\377`', '`\0`', '`\0`'.

`tcflag_t` **IGNCR** Macro

If this bit is set, carriage-return characters ('`\r`') are discarded on input. Discarding carriage return may be useful on terminals that send both carriage return and linefeed when you type the ⟨RET⟩ key.

`tcflag_t` **ICRNL** Macro

If this bit is set and `IGNCR` is not set, carriage-return characters ('`\r`') received as input are passed to the application as newline characters ('`\n`').

`tcflag_t` **INLCR** Macro

If this bit is set, newline characters ('`\n`') received as input are passed to the application as carriage-return characters ('`\r`').

`tcflag_t` **IXOFF** Macro

If this bit is set, start/stop control on input is enabled. In other words, the computer sends STOP and START characters as necessary to prevent input from coming in faster than programs are reading it. The idea is that the actual terminal hardware that is generating the input data responds to a STOP character by suspending transmission and to a START character by resuming transmission (see Section 6.4.9.3 [Special Characters for Flow Control], page 197).

`tcflag_t` **IXON** Macro

If this bit is set, start/stop control on output is enabled. In other words, if the computer receives a STOP character, it suspends output until a START character is received. In this case, the STOP and START characters are never passed to the application program. If this bit is not set, then START and STOP can be read as ordinary characters (see Section 6.4.9.3 [Special Characters for Flow Control], page 197).

`tcflag_t` **IXANY** Macro

If this bit is set, any input character restarts output when output has been suspended with the STOP character. Otherwise, only the START character restarts output.

This is a BSD extension; it exists only on BSD systems and the GNU system.

`tcflag_t` **IMAXBEL** Macro

If this bit is set, then filling up the terminal input buffer sends a BEL character (code `007`) to the terminal to ring the bell.

This is a BSD extension.

6.4.5 Output Modes

This section describes the terminal flags and fields that control how output characters are translated and padded for display. All of these are contained in the `c_oflag` member of the `struct termios` structure.

The `c_oflag` member itself is an integer, and you change the flags and fields using the operators `&`, `|` and `^`. Don't try to specify the entire value for `c_oflag`—instead, change only specific flags and leave the rest untouched (see Section 6.4.3 [Setting Terminal Modes Properly], page 183).

`tcflag_t` **OPOST** Macro

If this bit is set, output data is processed in some unspecified way so that it is displayed appropriately on the terminal device. This typically includes mapping newline characters (`'\n'`) onto carriage return and linefeed pairs.

If this bit isn't set, the characters are transmitted as-is.

The following 3 bits are BSD features, and they exist only on BSD systems and the GNU system. They are effective only if `OPOST` is set.

`tcflag_t` **ONLCR** Macro

If this bit is set, convert the newline character on output into a pair of characters, carriage return followed by linefeed.

`tcflag_t` **OXTABS** Macro

If this bit is set, convert tab characters on output into the appropriate number of spaces to emulate a tab stop every eight columns.

`tcflag_t` **ONOEOT** Macro

If this bit is set, discard *C-d* characters (code `004`) on output. These characters cause many dial-up terminals to disconnect.

6.4.6 Control Modes

This section describes the terminal flags and fields that control parameters usually associated with asynchronous serial data transmission. These flags may not make sense for other kinds of terminal ports (such as a network connection pseudoterminal). All of these are contained in the `c_cflag` member of the `struct termios` structure.

The `c_cflag` member itself is an integer, and you change the flags and fields using the operators `&`, `|` and `^`. Don't try to specify the entire value for `c_cflag`—instead, change only specific flags and leave the rest untouched (see Section 6.4.3 [Setting Terminal Modes Properly], page 183).

`tcflag_t` **CLOCAL** Macro

If this bit is set, it indicates that the terminal is connected "locally" and that the modem status lines (such as carrier detect) should be ignored.

On many systems, if this bit is not set and you call `open` without the `O_NONBLOCK` flag set, `open` blocks until a modem connection is established.

If this bit is not set and a modem disconnect is detected, a `SIGHUP` signal is sent to the controlling process group for the terminal (if it has one). Normally, this causes the process to exit (see Chapter 17 [Signal Handling], page 377). Reading from the terminal after a disconnect causes an end-of-file condition, and writing causes an `EIO` error to be returned. The terminal device must be closed and reopened to clear the condition.

`tcflag_t` **HUPCL** Macro

If this bit is set, a modem disconnect is generated when all processes that have the terminal device open have either closed the file or exited.

`tcflag_t` **CREAD** Macro

If this bit is set, input can be read from the terminal. Otherwise, input is discarded when it arrives.

`tcflag_t` **CSTOPB** Macro

If this bit is set, 2 stop bits are used. Otherwise, only 1 stop bit is used.

`tcflag_t` **PARENB** Macro

If this bit is set, generation and detection of a parity bit are enabled. See Section 6.4.4 [Input Modes], page 185, for information on how input parity errors are handled.

If this bit is not set, no parity bit is added to output characters, and input characters are not checked for correct parity.

`tcflag_t` **PARODD** Macro

This bit is only useful if `PARENB` is set. If `PARODD` is set, odd parity is used. Otherwise, even parity is used.

The control-mode flags also include a field for the number of bits per character. You can use the `CSIZE` macro as a mask to extract the value, like this: `settings.c_cflag & CSIZE`.

`tcflag_t` **CSIZE** Macro

This is a mask for the number of bits per character.

`tcflag_t` **CS5** Macro
This specifies 5 bits per byte.

`tcflag_t` **CS6** Macro
This specifies 6 bits per byte.

`tcflag_t` **CS7** Macro
This specifies 7 bits per byte.

`tcflag_t` **CS8** Macro
This specifies 8 bits per byte.

The following 4 bits are BSD extensions; this exists only on BSD systems and the GNU system.

`tcflag_t` **CCTS_OFLOW** Macro
If this bit is set, enable flow control of output based on the CTS wire (RS232 protocol).

`tcflag_t` **CRTS_IFLOW** Macro
If this bit is set, enable flow control of input based on the RTS wire (RS232 protocol).

`tcflag_t` **MDMBUF** Macro
If this bit is set, enable carrier-based flow control of output.

`tcflag_t` **CIGNORE** Macro
If this bit is set, it says to ignore the control modes and line-speed values entirely. This is only meaningful in a call to `tcsetattr`.

The `c_cflag` member and the line-speed values returned by `cfgetispeed` and `cfgetospeed` will be unaffected by the call. `CIGNORE` is useful if you want to set all the software modes in the other members, but leave the hardware details in `c_cflag` unchanged. (This is how the `TCSASOFT` flag to `tcsettattr` works.)

This bit is never set in the structure filled in by `tcgetattr`.

6.4.7 Local Modes

This section describes the flags for the `c_lflag` member of the `struct termios` structure. These flags generally control higher-level aspects of input processing than the input-modes flags described in Section 6.4.4 [Input Modes], page 185, such as echoing, signals and the choice of canonical- or noncanonical-input.

The `c_lflag` member itself is an integer, and you change the flags and fields using the operators `&`, `|` and `^`. Don't try to specify the entire value for `c_lflag`—instead, change only specific flags and leave the rest untouched (see Section 6.4.3 [Setting Terminal Modes Properly], page 183).

`tcflag_t` **ICANON** Macro
This bit, if set, enables canonical-input processing mode. Otherwise, input is processed in noncanonical mode (see Section 6.3 [Two Styles of Input: Canonical or Not], page 180).

`tcflag_t` **ECHO** Macro
If this bit is set, echoing of input characters back to the terminal is enabled.

`tcflag_t` **ECHOE** Macro
If this bit is set, echoing indicates erasure of input with the ERASE character by erasing the last character in the current line from the screen. Otherwise, the character erased is re-echoed to show what has happened (suitable for a printing terminal).

This bit only controls the display behavior; the `ICANON` bit by itself controls actual recognition of the ERASE character and erasure of input, without which `ECHOE` is simply irrelevant.

`tcflag_t` **ECHOPRT** Macro
This bit is like `ECHOE`—it enables display of the ERASE character in a way that is geared to a hard copy terminal. When you type the ERASE character, a '`\`' character is printed followed by the first character erased. Typing the ERASE character again just prints the next character erased. Then, the next time you type a normal character, a '`/`' character is printed before the character echoes.

This is a BSD extension, and exists only in BSD systems and the GNU system.

`tcflag_t` **ECHOK** Macro
This bit enables special display of the KILL character by moving to a new line after echoing the KILL character normally. The behavior of `ECHOKE` (below) is nicer to look at.

If this bit is not set, the KILL character echoes just as it would if it were not the KILL character. Then it is up to the user to remember that the KILL character has erased the preceding input; there is no indication of this on the screen.

This bit only controls the display behavior; the `ICANON` bit by itself controls actual recognition of the KILL character and erasure of input, without which `ECHOK` is simply irrelevant.

`tcflag_t` **ECHOKE** Macro
This bit is similar to `ECHOK`. It enables special display of the KILL character by erasing on the screen the entire line that has been killed. This is a BSD extension, and exists only in BSD systems and the GNU system.

`tcflag_t` **ECHONL** Macro
If this bit is set and the `ICANON` bit is also set, then the newline (`'\n'`) character is echoed even if the `ECHO` bit is not set.

`tcflag_t` **ECHOCTL** Macro

If this bit is set and the `ECHO` bit is also set, echo control characters with '^' followed by the corresponding text character. Thus, *`Control-A`* echoes as '^A'. This is usually the preferred mode for interactive input, because echoing a control character back to the terminal could have some undesired effect on the terminal.

This is a BSD extension, and exists only in BSD systems and the GNU system.

`tcflag_t` **ISIG** Macro

This bit controls whether the INTR, QUIT and SUSP characters are recognized. The functions associated with these characters are performed if and only if this bit is set. Being in canonical- or noncanonical- input mode has no effect on the interpretation of these characters.

You should use caution when disabling recognition of these characters. Programs that cannot be interrupted interactively are very user-unfriendly. If you clear this bit, your program should provide some alternate interface that allows the user to interactively send the signals associated with these characters, or to escape from the program (see Section 6.4.9.2 [Characters that Cause Signals], page 196).

`tcflag_t` **IEXTEN** Macro

POSIX.1 gives `IEXTEN` implementation-defined meaning, so you cannot rely on this interpretation on all systems.

On BSD systems and the GNU system, it enables the LNEXT and DISCARD characters (see Section 6.4.9.4 [Other Special Characters], page 198).

`tcflag_t` **NOFLSH** Macro

Normally, the INTR, QUIT and SUSP characters cause input and output queues for the terminal to be cleared. If this bit is set, the queues are not cleared.

`tcflag_t` **TOSTOP** Macro

If this bit is set and the system supports job control, then `SIGTTOU` signals are generated by background processes that attempt to write to the terminal (see Section 8.4 [Access to the Controlling Terminal], page 223).

The following bits are BSD extensions; they exist only in BSD systems and the GNU system.

`tcflag_t` **ALTWERASE** Macro

This bit determines how far the WERASE character should erase. The WERASE character erases back to the beginning of a word; the question is, where do words begin?

If this bit is clear, then the beginning of a word is a non-white-space character following a white-space character. If the bit is set, then the beginning of a word

is an alphanumeric character or underscore following a character which is none of those.

See Section 6.4.9.1 [Characters for Input Editing], page 194, for more information about the WERASE character.

`tcflag_t` **FLUSHO** Macro

This is the bit that toggles when the user types the DISCARD character. While this bit is set, all output is discarded (see Section 6.4.9.4 [Other Special Characters], page 198).

`tcflag_t` **NOKERNINFO** Macro

Setting this bit disables handling of the STATUS character (see Section 6.4.9.4 [Other Special Characters], page 198).

`tcflag_t` **PENDIN** Macro

If this bit is set, it indicates that there is a line of input that needs to be reprinted. Typing the REPRINT character sets this bit; the bit remains set until reprinting is finished (see Section 6.4.9.1 [Characters for Input Editing], page 194).

6.4.8 Line Speed

The terminal line speed tells the computer how fast to read and write data on the terminal.

If the terminal is connected to a real serial line, the terminal speed you specify actually controls the line—if it doesn't match the terminal's own idea of the speed, communication does not work. Real serial ports accept only certain standard speeds. Also, particular hardware may not support even all the standard speeds. Specifying a speed of 0 hangs up a dial-up connection and turns off modem control signals.

If the terminal is not a real serial line (for example, if it is a network connection), then the line speed won't really affect data-transmission speed, but some programs will use it to determine the amount of padding needed. It's best to specify a line-speed value that matches the actual speed of the actual terminal, but you can safely experiment with different values to vary the amount of padding.

There are actually two line speeds for each terminal—one for input and one for output. You can set them independently, but most often terminals use the same speed for both directions.

The speed values are stored in the `struct termios` structure, but don't try to access them in the `struct termios` structure directly. Instead, you should use the following functions to read and store them:

`speed_t` **cfgetospeed** (`const struct termios *`*termios-p*) Function

This function returns the output line-speed stored in the structure **termios-p*.

`speed_t` **cfgetispeed** (`const struct termios *`*termios-p*) Function
This function returns the input line-speed stored in the structure **termios-p*.

`int` **cfsetospeed** (`struct termios *`*termios-p*`, speed_t` *speed*) Function
This function stores *speed* in **termios-p* as the output speed. The normal return value is 0; a value of −1 indicates an error. If *speed* is not a speed, `cfsetospeed` returns −1.

`int` **cfsetispeed** (`struct termios *`*termios-p*`, speed_t` *speed*) Function
This function stores *speed* in **termios-p* as the input speed. The normal return value is 0; a value of −1 indicates an error. If *speed* is not a speed, `cfsetospeed` returns −1.

`int` **cfsetspeed** (`struct termios *`*termios-p*`, speed_t` *speed*) Function
This function stores *speed* in **termios-p* as both the input and output speeds. The normal return value is 0; a value of −1 indicates an error. If *speed* is not a speed, `cfsetspeed` returns −1. This function is an extension in 4.4 BSD.

speed_t Data Type
The `speed_t` type is an unsigned-integer data type used to represent line speeds.

The functions `cfsetospeed` and `cfsetispeed` report errors only for speed values that the system simply cannot handle. If you specify a speed value that is basically acceptable, then those functions will succeed. But they do not check that a particular hardware device can actually support the specified speeds—in fact, they don't know which device you plan to set the speed for. If you use `tcsetattr` to set the speed of a particular device to a value that it cannot handle, `tcsetattr` returns −1.

Portability Note: In the GNU library, the functions above accept speeds measured in bits per second as input, and they return speed values measured in bits per second. Other libraries require speeds to be indicated by special codes. For POSIX.1 portability, you must use one of the following symbols to represent the speed; their precise numeric values are system dependent, but each name has a fixed meaning: `B110` stands for 110 bps, `B300` for 300 bps, and so on. There is no portable way to represent any speed but these, but these are the only speeds that typical serial lines can support.

```
B0  B50  B75  B110  B134  B150  B200
B300  B600  B1200  B1800  B2400  B4800
B9600  B19200  B38400  B57600  B115200
B230400  B460800
```

BSD defines two additional speed symbols as aliases: `EXTA` is an alias for `B19200` and `EXTB` is an alias for `B38400`. These aliases are obsolete.

6.4.9 Special Characters

In canonical input, the terminal driver recognizes a number of special characters that perform various control functions. These include the ERASE character (usually ⟨DEL⟩) for editing input, and other editing characters. The INTR character (normally `C-c`) for sending a `SIGINT` signal, and other signal-raising characters, may be available in either canonical- or noncanonical- input mode. All these characters are described in this section.

The particular characters used are specified in the `c_cc` member of the `struct termios` structure. This member is an array; each element specifies the character for a particular role. Each element has a symbolic constant that stands for the index of that element—for example, `VINTR` is the index of the element that specifies the INTR character, so storing `'='` in *termios*`.c_cc[VINTR]` specifies '=' as the INTR character.

On some systems, you can disable a particular special-character function by specifying the value `_POSIX_VDISABLE` for that role. This value is unequal to any possible character code. See Section 12.7 [Optional Features in File Support], page 319, for more information about how to tell whether the operating system you are using supports `_POSIX_VDISABLE`.

6.4.9.1 Characters for Input Editing

These special characters are active only in canonical-input mode (see Section 6.3 [Two Styles of Input: Canonical or Not], page 180).

`int` **VEOF** Macro

This is the subscript for the EOF character in the special control character array. *termios*`.c_cc[VEOF]` holds the character itself.

The EOF character is recognized only in canonical-input mode. It acts as a line terminator in the same way as a newline character, but if the EOF character is typed at the beginning of a line, it causes `read` to return a byte count of 0, indicating end of file. The EOF character itself is discarded.

Usually, the EOF character is `C-d`.

`int` **VEOL** Macro

This is the subscript for the EOL character in the special control character array. *termios*`.c_cc[VEOL]` holds the character itself.

The EOL character is recognized only in canonical-input mode. It acts as a line terminator, just like a newline character. The EOL character is not discarded; it is read as the last character in the input line.

You don't need to use the EOL character to make ⟨RET⟩ end a line. Just set the ICRNL flag. In fact, this is the default state of affairs.

`int` **VEOL2** Macro

This is the subscript for the EOL2 character in the special control character array. *termios*`.c_cc[VEOL2]` holds the character itself.

The EOL2 character works just like the EOL character (see above), but it can be a different character. Thus, you can specify two characters to terminate an input line, by setting EOL to one of them and EOL2 to the other.

The EOL2 character is a BSD extension; it exists only on BSD systems and the GNU system.

`int` **VERASE** Macro

This is the subscript for the ERASE character in the special control character array. *termios*`.c_cc[VERASE]` holds the character itself.

The ERASE character is recognized only in canonical-input mode. When the user types the erase character, the previous character typed is discarded. If the terminal generates multibyte character sequences, this may cause more than 1 byte of input to be discarded. This cannot be used to erase past the beginning of the current line of text. The ERASE character itself is discarded.

Usually, the ERASE character is ⟨DEL⟩.

`int` **VWERASE** Macro

This is the subscript for the WERASE character in the special control character array. *termios*`.c_cc[VWERASE]` holds the character itself.

The WERASE character is recognized only in canonical mode. It erases an entire word of prior input and any white space after it; white-space characters before the word are not erased.

The definition of a "word" depends on the setting of the `ALTWERASE` mode (see Section 6.4.7 [Local Modes], page 189).

If the `ALTWERASE` mode is not set, a word is defined as a sequence of any characters except space or tab.

If the `ALTWERASE` mode is set, a word is defined as a sequence of characters containing only letters, numbers and underscores, optionally followed by one character that is not a letter, number or underscore.

The WERASE character is usually `C-w`.

This is a BSD extension.

`int` **VKILL** Macro

This is the subscript for the KILL character in the special control character array. *termios*`.c_cc[VKILL]` holds the character itself.

The KILL character is recognized only in canonical-input mode. When the user types the kill character, the entire contents of the current line of input are discarded. The kill character itself is discarded too.

The KILL character is usually `C-u`.

`int` **VREPRINT** Macro

This is the subscript for the REPRINT character in the special control character array. *termios*`.c_cc[VREPRINT]` holds the character itself.

The REPRINT character is recognized only in canonical mode. It reprints the current input line. If some asynchronous output has come while you are typing, this lets you see the line you are typing clearly again.

The REPRINT character is usually `C-r`.

This is a BSD extension.

6.4.9.2 Characters that Cause Signals

These special characters may be active in either canonical- or noncanonical-input mode, but only when the `ISIG` flag is set (see Section 6.4.7 [Local Modes], page 189).

`int` **VINTR** Macro

This is the subscript for the INTR character in the special control character array. *termios*`.c_cc[VINTR]` holds the character itself.

The INTR (interrupt) character raises a `SIGINT` signal for all processes in the foreground job associated with the terminal. The INTR character itself is then discarded. See Chapter 17 [Signal Handling], page 377, for more information about signals.

Typically, the INTR character is `C-c`.

`int` **VQUIT** Macro

This is the subscript for the QUIT character in the special control character array. *termios*`.c_cc[VQUIT]` holds the character itself.

The QUIT character raises a `SIGQUIT` signal for all processes in the foreground job associated with the terminal. The QUIT character itself is then discarded. See Chapter 17 [Signal Handling], page 377, for more information about signals.

Typically, the QUIT character is `C-\`.

`int` **VSUSP** Macro

This is the subscript for the SUSP character in the special control character array. *termios*`.c_cc[VSUSP]` holds the character itself.

The SUSP (suspend) character is recognized only if the implementation supports job control (see Chapter 8 [Job Control], page 221). It causes a `SIGTSTP` signal to be sent to all processes in the foreground job associated with the terminal. The SUSP character itself is then discarded. See Chapter 17 [Signal Handling], page 377, for more information about signals.

Typically, the SUSP character is `C-z`.

Few applications disable the normal interpretation of the SUSP character. If your program does this, it should provide some other mechanism for the user to stop the job. When the user invokes this mechanism, the program should send a `SIGTSTP` signal to the process group of the process, not just to the process itself (see Section 17.6.2 [Signaling Another Process], page 410).

`int` **VDSUSP** Macro

This is the subscript for the DSUSP character in the special control character array. *termios*`.c_cc[VDSUSP]` holds the character itself.

The DSUSP (suspend) character is recognized only if the implementation supports job control (see Chapter 8 [Job Control], page 221). It sends a `SIGTSTP` signal, like the SUSP character, but not right away—only when the program tries to read it as input. Not all systems with job control support DSUSP; only BSD-compatible systems (including the GNU system).

See Chapter 17 [Signal Handling], page 377, for more information about signals.

Typically, the DSUSP character is *C-y*.

6.4.9.3 Special Characters for Flow Control

These special characters may be active in either canonical- or noncanonical-input mode, but their use is controlled by the flags `IXON` and `IXOFF` (see Section 6.4.4 [Input Modes], page 185).

`int` **VSTART** Macro

This is the subscript for the START character in the special control character array. *termios*`.c_cc[VSTART]` holds the character itself.

The START character is used to support the `IXON` and `IXOFF` input modes. If `IXON` is set, receiving a START character resumes suspended output; the START character itself is discarded. If `IXANY` is set, receiving any character at all resumes suspended output; the resuming character is not discarded unless it is the START character. If `IXOFF` is set, the system may also transmit START characters to the terminal.

The usual value for the START character is *C-q*. You may not be able to change this value—the hardware may insist on using *C-q* regardless of what you specify.

`int` **VSTOP** Macro

This is the subscript for the STOP character in the special control character array. *termios*`.c_cc[VSTOP]` holds the character itself.

The STOP character is used to support the `IXON` and `IXOFF` input modes. If `IXON` is set, receiving a STOP character causes output to be suspended; the STOP character itself is discarded. If `IXOFF` is set, the system may also transmit STOP characters to the terminal, to prevent the input queue from overflowing.

The usual value for the STOP character is `C-s`. You may not be able to change this value—the hardware may insist on using `C-s` regardless of what you specify.

6.4.9.4 Other Special Characters

These special characters exist only in BSD systems and the GNU system.

`int` **VLNEXT** Macro

This is the subscript for the LNEXT character in the special control character array. *termios*`.c_cc[VLNEXT]` holds the character itself.

The LNEXT character is recognized only when `IEXTEN` is set, but in both canonical and noncanonical mode. It disables any special significance of the next character the user types. Even if the character would normally perform some editing function or generate a signal, it is read as a plain character. This is the analog of the `C-q` command in Emacs. "LNEXT" stands for "literal next."

The LNEXT character is usually `C-v`.

`int` **VDISCARD** Macro

This is the subscript for the DISCARD character in the special control character array. *termios*`.c_cc[VDISCARD]` holds the character itself.

The DISCARD character is recognized only when `IEXTEN` is set, but in both canonical and noncanonical mode. Its effect is to toggle the discard-output flag. When this flag is set, all program output is discarded. Setting the flag also discards all output currently in the output buffer. Typing any other character resets the flag.

`int` **VSTATUS** Macro

This is the subscript for the STATUS character in the special control character array. *termios*`.c_cc[VSTATUS]` holds the character itself.

The STATUS character's effect is to print out a status message about how the current process is running.

The STATUS character is recognized only in canonical mode, and only if `NOKERNINFO` is not set.

6.4.10 Noncanonical Input

In noncanonical-input mode, the special editing characters such as ERASE and KILL are ignored. The system facilities for the user to edit input are disabled in noncanonical mode, so that all input characters (unless they are special for signal or flow-control purposes) are passed to the application program exactly as typed. It is up to the application program to give the user ways to edit the input, if appropriate.

Noncanonical mode offers special parameters called MIN and TIME for controlling whether and how long to wait for input to be available. You can even use them

to avoid ever waiting—to return immediately with whatever input is available, or with no input.

The MIN and TIME are stored in elements of the `c_cc` array, which is a member of the `struct termios` structure. Each element of this array has a particular role, and each element has a symbolic constant that stands for the index of that element. `VMIN` and `VMAX` are the names for the indices in the array of the MIN and TIME slots.

`int` **VMIN** Macro

This is the subscript for the MIN slot in the `c_cc` array. Thus, *termios*`.c_cc[VMIN]` is the value itself.

The MIN slot is only meaningful in noncanonical-input mode; it specifies the minimum number of bytes that must be available in the input queue in order for `read` to return.

`int` **VTIME** Macro

This is the subscript for the TIME slot in the `c_cc` array. Thus, *termios*`.c_cc[VTIME]` is the value itself.

The TIME slot is only meaningful in noncanonical-input mode; it specifies how long to wait for input before returning, in units of 0.1 seconds.

The MIN and TIME values interact to determine the criterion for when `read` should return; their precise meanings depend on which of them are nonzero. There are four possible cases:

- Both TIME and MIN are nonzero.

  In this case, TIME specifies how long to wait after each input character to see if more input arrives. After the first character received, `read` keeps waiting until either MIN bytes have arrived in all, or TIME elapses with no further input.

  `read` always blocks until the first character arrives, even if TIME elapses first. `read` can return more than MIN characters if more than MIN happen to be in the queue.

- Both MIN and TIME are 0.

  In this case, `read` always returns immediately with as many characters as are available in the queue, up to the number requested. If no input is immediately available, `read` returns a value of 0.

- MIN is 0 but TIME has a nonzero value.

  In this case, `read` waits for time TIME for input to become available; the availability of a single byte is enough to satisfy the read request and cause `read` to return. When it returns, it returns as many characters as are available, up to the number requested. If no input is available before the timer expires, `read` returns a value of 0.

- TIME is 0 but MIN has a nonzero value.

  In this case, `read` waits until at least MIN bytes are available in the queue. At that time, `read` returns as many characters as are available, up to the number requested. `read` can return more than MIN characters if more than MIN happen to be in the queue.

What happens if MIN is 50 and you ask to read just 10 bytes? Normally, `read` waits until there are 50 bytes in the buffer (or, more generally, until the wait condition described above is satisfied), and then reads 10 of them, leaving the other 40 buffered in the operating system for a subsequent call to `read`.

Portability Note: On some systems, the MIN and TIME slots are actually the same as the EOF and EOL slots. This causes no serious problem because the MIN and TIME slots are used only in noncanonical input and the EOF and EOL slots are used only in canonical input, but it isn't very clean. The GNU library allocates separate slots for these uses.

`void` **cfmakeraw** (`struct termios` ⋆*termios-p*) Function

This function provides an easy way to set up ⋆*termios-p* for what has traditionally been called "raw mode" in BSD. This uses noncanonical input, and turns off most processing to give an unmodified channel to the terminal.

It does exactly this:

```
termios-p->c_iflag &= ~(IGNBRK|BRKINT|PARMRK|ISTRIP
                        |INLCR|IGNCR|ICRNL|IXON);
termios-p->c_oflag &= ~OPOST;
termios-p->c_lflag &= ~(ECHO|ECHONL|ICANON|ISIG|IEXTEN);
termios-p->c_cflag &= ~(CSIZE|PARENB);
termios-p->c_cflag |= CS8;
```

6.5 BSD Terminal Modes

The usual way to get and set terminal modes is with the functions described in Section 6.4 [Terminal Modes], page 181. However, on some systems you can use the BSD-derived functions in this section to do some of the same things. On many systems, these functions do not exist. Even with the GNU C Library, the functions simply fail with `errno` = `ENOSYS` with many kernels, including Linux.

The symbols used in this section are declared in '`sgtty.h`'.

struct sgttyb Data Type

This structure is an input or output parameter list for `gtty` and `stty`.

`char sg_ispeed`
: Line speed for input

`char sg_ospeed`
: Line speed for output

`char sg_erase`
: Erase character

`char sg_kill`
: Kill character

`int sg_flags`
: Various flags

`int` **gtty** (`int` *filedes*, `struct sgttyb` **attributes*) Function

This function gets the attributes of a terminal.

`gtty` sets **attributes* to describe the terminal attributes of the terminal that is open with file descriptor *filedes*.

`int` **stty** (`int` *filedes*, `struct sgttyb * attributes`) Function

This function sets the attributes of a terminal.

`stty` sets the terminal attributes of the terminal that is open with file descriptor *filedes* to those described by **filedes*.

6.6 Line Control Functions

These functions perform miscellaneous control actions on terminal devices. As regards terminal access, they are treated like doing output: if any of these functions is used by a background process on its controlling terminal, normally all processes in the process group are sent a `SIGTTOU` signal. The exception is if the calling process itself is ignoring or blocking `SIGTTOU` signals, in which case the operation is performed and no signal is sent (see Chapter 8 [Job Control], page 221).

`int` **tcsendbreak** (`int` *filedes*, `int` *duration*) Function

This function generates a break condition by transmitting a stream of zero bits on the terminal associated with the file descriptor *filedes*. The duration of the break is controlled by the *duration* argument. If zero, the duration is between 0.25 and 0.5 seconds. The meaning of a nonzero value depends on the operating system.

This function does nothing if the terminal is not an asynchronous serial data port.

The return value is normally 0. In the event of an error, a value of -1 is returned. The following `errno` error conditions are defined for this function:

`EBADF` The *filedes* is not a valid file-descriptor.

`ENOTTY` The *filedes* is not associated with a terminal device.

`int` **tcdrain** (`int` *filedes*) Function

The `tcdrain` function waits until all queued output to the terminal *filedes* has been transmitted.

This function is a cancellation point in multithreaded programs. This is a problem if the thread allocates some resources (like memory, file descriptors, semaphores or whatever) at the time `tcdrain` is called. If the thread gets canceled, these resources stay allocated until the program ends. To avoid this, calls to `tcdrain` should be protected using cancellation handlers.

The return value is normally 0. In the event of an error, a value of −1 is returned. The following `errno` error conditions are defined for this function:

`EBADF` The *filedes* is not a valid file-descriptor.

`ENOTTY` The *filedes* is not associated with a terminal device.

`EINTR` The operation was interrupted by delivery of a signal (see Section 17.5 [Primitives Interrupted by Signals], page 408).

`int` **tcflush** (`int` *filedes*, `int` *queue*) Function

The `tcflush` function is used to clear the input and/or output queues associated with the terminal file *filedes*. The *queue* argument specifies which queue(s) to clear, and can be one of the following values:

`TCIFLUSH`
Clear any input data received but not yet read.

`TCOFLUSH`
Clear any output data written but not yet transmitted.

`TCIOFLUSH`
Clear both queued input and output.

The return value is normally 0. In the event of an error, a value of −1 is returned. The following `errno` error conditions are defined for this function:

`EBADF` The *filedes* is not a valid file-descriptor.

`ENOTTY` The *filedes* is not associated with a terminal device.

`EINVAL` A bad value was supplied as the *queue* argument.

It is unfortunate that this function is named `tcflush`, because the term "flush" is normally used for quite another operation—waiting until all output is transmitted—and using it for discarding input or output would be confusing. Unfortunately, the name `tcflush` comes from POSIX and we cannot change it.

`int` **tcflow** (`int` *filedes*, `int` *action*) Function

The `tcflow` function is used to perform operations relating to XON/XOFF flow control on the terminal file specified by *filedes*.

The *action* argument specifies what operation to perform, and can be one of the following values:

`TCOOFF` Suspend transmission of output.

`TCOON` Restart transmission of output.

`TCIOFF` Transmit a STOP character.

`TCION` Transmit a START character.

For more information about the STOP and START characters, see Section 6.4.9 [Special Characters], page 194.

The return value is normally 0. In the event of an error, a value of −1 is returned. The following `errno` error conditions are defined for this function:

`EBADF` The *filedes* is not a valid file-descriptor.

`ENOTTY` The *filedes* is not associated with a terminal device.

`EINVAL` A bad value was supplied as the *action* argument.

6.7 Noncanonical-Mode Example

Here is an example program that shows how you can set up a terminal device to read single characters in noncanonical-input mode, without echo.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <termios.h>

/* Use this variable to remember original terminal attributes. */

struct termios saved_attributes;

void
reset_input_mode (void)
{
  tcsetattr (STDIN_FILENO, TCSANOW, &saved_attributes);
}

void
set_input_mode (void)
{
  struct termios tattr;
  char *name;

  /* Make sure stdin is a terminal. */
  if (!isatty (STDIN_FILENO))
    {
      fprintf (stderr, "Not a terminal.\n");
```

```
      exit (EXIT_FAILURE);
    }

  /* Save the terminal attributes so we can restore them later. */
  tcgetattr (STDIN_FILENO, &saved_attributes);
  atexit (reset_input_mode);


  /* Set the funny terminal modes. */
  tcgetattr (STDIN_FILENO, &tattr);
  tattr.c_lflag &= ~(ICANON|ECHO); /* Clear ICANON and ECHO. */
  tattr.c_cc[VMIN] = 1;
  tattr.c_cc[VTIME] = 0;
  tcsetattr (STDIN_FILENO, TCSAFLUSH, &tattr);
}


int
main (void)
{
  char c;

  set_input_mode ();

  while (1)
    {
      read (STDIN_FILENO, &c, 1);
      if (c == '\004')          /* C-d */
        break;
      else
        putchar (c);
    }

  return EXIT_SUCCESS;
}
```

This program is careful to restore the original terminal modes before exiting or terminating with a signal. It uses the `atexit` function[2] to make sure this is done by `exit`.

The shell is supposed to take care of resetting the terminal modes when a process is stopped or continued (see Chapter 8 [Job Control], page 221). But some existing

[2] See Loosemore, et al., "Clean-Ups on Exit" (see chap. 1, n. 1).

shells do not actually do this, so you may wish to establish handlers for job control signals that reset terminal modes. The above example does so.

6.8 Pseudoterminals

A *pseudoterminal* is a special interprocess-communication channel that acts like a terminal. One end of the channel is called the *master* side or *master pseudoterminal device*, the other side is called the *slave* side. Data written to the master side is received by the slave side as if it was the result of a user typing at an ordinary terminal, and data written to the slave side is sent to the master side as if it was written on an ordinary terminal.

Pseudoterminals are the way programs like `xterm` and `emacs` implement their terminal-emulation functionality.

6.8.1 Allocating Pseudoterminals

This subsection describes functions for allocating a pseudoterminal, and for making this pseudoterminal available for actual use. These functions are declared in the header file '`stdlib.h`'.

`int` **getpt** (`void`) Function

The `getpt` function returns a new file-descriptor for the next available master pseudoterminal. The normal return value from `getpt` is a nonnegative-integer file-descriptor. In the case of an error, a value of -1 is returned instead. The following `errno` condition is defined for this function:

`ENOENT` There are no free master pseudoterminals available.

This function is a GNU extension.

`int` **grantpt** (`int` *filedes*) Function

The `grantpt` function changes the ownership and access permission of the slave pseudoterminal device corresponding to the master pseudoterminal device associated with the file descriptor *filedes*. The owner is set from the real user-ID of the calling process (see Section 10.2 [The Persona of a Process], page 253), and the group is set to a special group (typically *tty*) or from the real group-ID of the calling process. The access permission is set such that the file is both readable and writable by the owner and only writable by the group.

On some systems, this function is implemented by invoking a special `setuid` root program (see Section 10.4 [How an Application Can Change Persona], page 254). As a consequence, installing a signal handler for the `SIGCHLD` signal (see Section 17.2.5 [Job Control Signals], page 385) may interfere with a call to `grantpt`.

The normal return value from `grantpt` is 0; a value of -1 is returned in case of failure. The following `errno` error conditions are defined for this function:

`EBADF` The *filedes* argument is not a valid file-descriptor.

EINVAL The *filedes* argument is not associated with a master pseudoterminal device.

EACCES The slave pseudoterminal device corresponding to the master associated with *filedes* could not be accessed.

`int` **unlockpt** (`int` *filedes*) Function

The `unlockpt` function unlocks the slave pseudoterminal device corresponding to the master pseudoterminal device associated with the file descriptor *filedes*. On many systems, the slave can only be opened after unlocking, so portable applications should always call `unlockpt` before trying to open the slave.

The normal return value from `unlockpt` is 0; a value of −1 is returned in case of failure. The following `errno` error conditions are defined for this function:

EBADF The *filedes* argument is not a valid file-descriptor.

EINVAL The *filedes* argument is not associated with a master pseudoterminal device.

`char *` **ptsname** (`int` *filedes*) Function

If the file descriptor *filedes* is associated with a master pseudoterminal device, the `ptsname` function returns a pointer to a statically allocated, null-terminated string containing the file name of the associated slave pseudoterminal file. This string might be overwritten by subsequent calls to `ptsname`.

`int` **ptsname_r** (`int` *filedes*, `char *`*buf*, `size_t` *len*) Function

The `ptsname_r` function is similar to the `ptsname` function, except that it places its result into the user-specified buffer starting at *buf* with length *len*.

This function is a GNU extension.

Portability Note: On System V derived systems, the file returned by the `ptsname` and `ptsname_r` functions may be STREAMS-based, and therefore require additional processing after opening before it actually behaves as a pseudoterminal.

Typical usage of these functions is illustrated by the following example:

```
int
open_pty_pair (int *amaster, int *aslave)
{
  int master, slave;
  char *name;

  master = getpt ();
  if (master < 0)
    return 0;

  if (grantpt (master) < 0 || unlockpt (master) < 0)
```

```
    goto close_master;
  name = ptsname (master);
  if (name == NULL)
    goto close_master;

  slave = open (name, O_RDWR);
  if (slave == -1)
    goto close_master;

  if (isastream (slave))
    {
      if (ioctl (slave, I_PUSH, "ptem") < 0
          || ioctl (slave, I_PUSH, "ldterm") < 0)
        goto close_slave;
    }

  *amaster = master;
  *aslave = slave;
  return 1;

close_slave:
  close (slave);

close_master:
  close (master);
  return 0;
}
```

6.8.2 Opening a Pseudoterminal Pair

These functions, derived from BSD, are available in the separate 'libutil' library, and declared in 'pty.h'.

`int` **openpty** (`int` **amaster*, `int` **aslave*, `char` **name*, `struct termios` **termp*, `struct winsize` **winp*) Function

This function allocates and opens a pseudoterminal pair, returning the file descriptor for the master in **amaster*, and the file descriptor for the slave in **aslave*. If the argument *name* is not a null pointer, the file name of the slave pseudoterminal device is stored in `*name`. If *termp* is not a null pointer, the terminal attributes of the slave are set to the ones specified in the structure that *termp* points to (see Section 6.4 [Terminal Modes], page 181). Likewise, if the *winp* is not a null pointer, the screen size of the slave is set to the values specified in the structure that *winp* points to.

The normal return value from `openpty` is 0; a value of -1 is returned in case of failure. The following `errno` condition is defined for this function:

`ENOENT` There are no free pseudoterminal pairs available.

Warning: Using the `openpty` function with *name* not set to `NULL` is *very dangerous* because it provides no protection against overflowing the string *name*. You should use the `ttyname` function on the file descriptor returned in **slave* to find out the file name of the slave pseudoterminal device instead.

`int` **forkpty** (`int` **amaster*, `char` **name*, `struct termios` **termp*, `struct winsize` **winp*) Function

This function is similar to the `openpty` function, but in addition, it forks a new process (see Section 7.4 [Creating a Process], page 211) and makes the newly opened slave pseudoterminal device the controlling terminal (see Section 8.3 [Controlling Terminal of a Process], page 222) for the child process.

If the operation is successful, there are then both parent and child processes, and both see `forkpty` return, but with different values—it returns a value of 0 in the child process and returns the child's process ID in the parent process.

If the allocation of a pseudoterminal pair or the process creation failed, `forkpty` returns a value of -1 in the parent process.

Warning: The `forkpty` function has the same problems with respect to the *name* argument as `openpty`.

7 Processes

Processes are the primitive units for allocation of system resources. Each process has its own address space and (usually) one thread of control. A process executes a program; you can have multiple processes executing the same program, but each process has its own copy of the program within its own address space and executes it independently of the other copies.

Processes are organized hierarchically. Each process has a *parent process* that explicitly arranged to create it. The processes created by a given parent are called its *child processes*. A child inherits many of its attributes from the parent process.

This chapter describes how a program can create, terminate and control child processes. Actually, there are three distinct operations involved: creating a new child process, causing the new process to execute a program and coordinating the completion of the child process with the original program.

The `system` function provides a simple, portable mechanism for running another program; it does all three steps automatically. If you need more control over the details of how this is done, you can use the primitive functions to do each step individually instead.

7.1 Running a Command

The easy way to run another program is to use the `system` function. This function does all the work of running a subprogram, but it doesn't give you much control over the details: you have to wait until the subprogram terminates before you can do anything else.

`int` **system** (`const char *`*command*) Function

This function executes *command* as a shell command. In the GNU C Library, it always uses the default shell `sh` to run the command. In particular, it searches the directories in `PATH` to find programs to execute. The return value is `-1` if it wasn't possible to create the shell process, and otherwise is the status of the shell process (see Section 7.6 [Process Completion], page 215, for details on how this status code can be interpreted).

If the *command* argument is a null pointer, a return value of 0 indicates that no command processor is available.

This function is a cancellation point in multithreaded programs. This is a problem if the thread allocates some resources (like memory, file descriptors, semaphores, etc.) at the time `system` is called. If the thread gets canceled, these resources stay allocated until the program ends. To avoid this, calls to `system` should be protected using cancellation handlers.

The `system` function is declared in the header file '`stdlib.h`'.

Portability Note: Some C implementations may not have any notion of a command processor that can execute other programs. You can determine whether a

command processor exists by executing `system (NULL)`; if the return value is nonzero, a command processor is available.

The `popen` and `pclose` functions (see Section 4.2 [Pipe to a Subprocess], page 121) are closely related to the `system` function. They allow the parent process to communicate with the standard input and output channels of the command being executed.

7.2 Process-Creation Concepts

This section gives an overview of processes and of the steps involved in creating a process and making it run another program.

Each process is named by a *process ID* number. A unique process-ID is allocated to each process when it is created. The *lifetime* of a process ends when its termination is reported to its parent process; at that time, all of the process resources, including its process ID, are freed.

Processes are created with the `fork` system call (so the operation of creating a new process is sometimes called *forking* a process). The *child process* created by `fork` is a copy of the original *parent process*, except that it has its own process ID.

After forking a child process, both the parent and child processes continue to execute normally. If you want your program to wait for a child process to finish executing before continuing, you must do this explicitly after the fork operation, by calling `wait` or `waitpid` (see Section 7.6 [Process Completion], page 215). These functions give you limited information about why the child terminated—for example, its exit status code.

A newly forked child process continues to execute the same program as its parent process, at the point where the `fork` call returns. You can use the return value from `fork` to tell whether the program is running in the parent process or the child.

Having several processes run the same program is only occasionally useful. But the child can execute another program using one of the `exec` functions (see Section 7.5 [Executing a File], page 212). The program that the process is executing is called its *process image*. Starting execution of a new program causes the process to forget all about its previous process image; when the new program exits, the process exits too, instead of returning to the previous process image.

7.3 Process Identification

The `pid_t` data type represents process IDs. You can get the process ID of a process by calling `getpid`. The function `getppid` returns the process ID of the parent of the current process (this is also known as the *parent process ID*). Your program should include the header files '`unistd.h`' and '`sys/types.h`' to use these functions.

pid_t Data Type

The `pid_t` data type is a signed-integer type that is capable of representing a process ID. In the GNU library, this is an `int`.

`pid_t` **getpid** (`void`) Function

The `getpid` function returns the process ID of the current process.

`pid_t` **getppid** (`void`) Function

The `getppid` function returns the process ID of the parent of the current process.

7.4 Creating a Process

The `fork` function is the primitive for creating a process. It is declared in the header file '`unistd.h`'.

`pid_t` **fork** (`void`) Function

The `fork` function creates a new process.

If the operation is successful, there are then both parent and child processes. Both see `fork` return, but with different values—it returns a value of `0` in the child process and returns the child's process ID in the parent process.

If process creation failed, `fork` returns a value of `-1` in the parent process. The following `errno` error conditions are defined for `fork`:

`EAGAIN` There aren't enough system resources to create another process, or the user already has too many processes running. This means exceeding the `RLIMIT_NPROC` resource limit, which can usually be increased (see Section 14.2 [Limiting Resource Usage], page 338).

`ENOMEM` The process requires more space than the system can supply.

The specific attributes of the child process that differ from the parent process are

- The child process has its own unique process-ID.
- The parent process ID of the child process is the process ID of its parent process.
- The child process gets its own copies of the parent process's open file-descriptors. Subsequently changing attributes of the file descriptors in the parent process won't affect the file descriptors in the child, and vice versa (see Section 2.11 [Control Operations on Files], page 54). However, the file position associated with each descriptor is shared by both processes.[1]
- The elapsed processor times for the child process are set to 0.[2]
- The child doesn't inherit file locks set by the parent process (see Section 2.11 [Control Operations on Files], page 54).
- The child doesn't inherit alarms set by the parent process.[3]

[1] See Loosemore et al., "File Position" (see chap. 1, n.1).
[2] Ibid., "Processor Time Inquiry".
[3] Ibid., "Setting an Alarm".

- The set of pending signals (see Section 17.1.3 [How Signals Are Delivered], page 378) for the child process is cleared. The child process inherits its mask of blocked signals and signal actions from the parent process.

`pid_t` **vfork** (`void`) Function

The `vfork` function is similar to `fork`, but on some systems it is more efficient. However, there are restrictions you must follow to use it safely.

While `fork` makes a complete copy of the calling process's address space and allows both the parent and child to execute independently, `vfork` does not make this copy. Instead, the child process created with `vfork` shares its parent's address space until it calls `_exit` or one of the `exec` functions. In the meantime, the parent process suspends execution.

You must be very careful not to allow the child process created with `vfork` to modify any global data or even local variables shared with the parent. Furthermore, the child process cannot return from (or do a long jump out of) the function that called `vfork`! This would leave the parent process's control information very confused. If in doubt, use `fork` instead.

Some operating systems don't really implement `vfork`. The GNU C Library permits you to use `vfork` on all systems, but actually executes `fork` if `vfork` isn't available. If you follow the proper precautions for using `vfork`, your program will still work even if the system uses `fork` instead.

7.5 Executing a File

This section describes the `exec` family of functions, for executing a file as a process image. You can use these functions to make a child process execute a new program after it has been forked.[4]

The functions in this family differ in how you specify the arguments, but otherwise they all do the same thing. They are declared in the header file '`unistd.h`'.

`int` **execv** (`const char *`*filename*`, char *const` *argv*`[])` Function

The `execv` function executes the file named by *filename* as a new process image.

The *argv* argument is an array of null-terminated strings that is used to provide a value for the `argv` argument to the `main` function of the program to be executed. The last element of this array must be a null pointer. By convention, the first element of this array is the file name of the program sans directory names.[5]

The environment for the new process image is taken from the `environ` variable of the current process image.[6]

[4] To see the effects of `exec` from the point of view of the called program, see Loosemore et al., "The Basic Program/System Interface".

[5] For full details on how programs can access these arguments, see Loosemore et al., "Program Arguments".

[6] Ibid., "Environment Variables".

`int` **execl** (`const char *`*filename*`, const char *`*arg0*`, ...`) Function
This is similar to `execv`, but the *argv* strings are specified individually instead of as an array. A null pointer must be passed as the last such argument.

`int` **execve** (`const char *`*filename*`, char *const` *argv*`[], char *const` *env*`[]`) Function
This is similar to `execv`, but it permits you to specify the environment for the new program explicitly as the *env* argument. This should be an array of strings in the same format as for the `environ` variable.[7]

`int` **execle** (`const char *`*filename*`, const char *`*arg0*`, char *const` *env*`[], ...`) Function
This is similar to `execl`, but it permits you to specify the environment for the new program explicitly. The environment argument is passed following the null pointer that marks the last *argv* argument, and should be an array of strings in the same format as for the `environ` variable.

`int` **execvp** (`const char *`*filename*`, char *const` *argv*`[]`) Function
The `execvp` function is similar to `execv`, except that it searches the directories listed in the `PATH` environment variable[8] to find the full file-name of a file from *filename* if *filenambe* does not contain a slash.

This function is useful for executing system utility programs, because it looks for them in the places that the user has chosen. Shells use it to run the commands that users type.

`int` **execlp** (`const char *`*filename*`, const char *`*arg0*`, ...`) Function

This function is like `execl`, except that it performs the same file-name searching as the `execvp` function.

The size of the argument list and environment list taken together must not be greater than `ARG_MAX` bytes (see Section 12.1 [General Capacity-Limits], page 303). In the `GNU` system, the size (which compares against `ARG_MAX`) includes, for each string, the number of characters in the string, plus the size of a `char *`, plus 1, rounded up to a multiple of the size of a `char *`. Other systems may have somewhat different rules for counting.

These functions normally don't return, since execution of a new program causes the currently executing program to go away completely. A value of `-1` is returned in the event of a failure. In addition to the usual file-name errors, the following `errno` error conditions are defined for these functions:[9]

---

[7] Ibid., "Environment Access".
[8] Ibid., "Standard Environment Variables".
[9] Ibid., "File-Name Errors".

`E2BIG` The combined size of the new program's argument list and environment list is larger than `ARG_MAX` bytes. The GNU system has no specific limit on the argument list size, so this error code cannot result, but you may get `ENOMEM` instead if the arguments are too big for available memory.

`ENOEXEC` The specified file can't be executed, because it isn't in the right format.

`ENOMEM` Executing the specified file requires more storage than is available.

If execution of the new file succeeds, it updates the access-time field of the file as if the file had been read (see Section 3.9.9 [File Times], page 108).

The point at which the file is closed again is not specified, but is at some point before the process exits or before another process image is executed.

Executing a new process image completely changes the contents of memory, copying only the argument and environment strings to new locations. But many other attributes of the process are unchanged:

- The process ID and the parent process-ID (see Section 7.2 [Process-Creation Concepts], page 210)
- Session and process group membership (see Section 8.1 [Concepts of Job Control], page 221)
- Real user-ID and group-ID, and supplementary group IDs (see Section 10.2 [The Persona of a Process], page 253)
- Pending alarms[10]
- Current working directory and root directory (see Section 3.1 [Working Directory], page 71)[11]
- File mode creation mask (see Section 3.9.7 [Assigning File Permissions], page 104)
- Process signal-mask (see Section 17.7.3 [Process Signal-Mask], page 416)
- Pending signals (see Section 17.7 [Blocking Signals], page 414)
- Elapsed processor time associated with the process[12]

If the set-user-ID and set-group-ID mode bits of the process-image file are set, this affects the effective user ID and effective group-ID, respectively, of the process. These concepts are discussed in detail in Section 10.2 [The Persona of a Process], page 253.

Signals that are set to be ignored in the existing process image are also set to be ignored in the new process image. All other signals are set to the default action in the new process image. For more information about signals, see Chapter 17 [Signal Handling], page 377.

---

[10] Ibid., "Setting an Alarm".

[11] In the GNU system, the root directory is not copied when executing a setuid program; instead, the system default root directory is used for the new program.

[12] Ibid., "Processor Time Inquiry".

File descriptors open in the existing process image remain open in the new process image, unless they have the `FD_CLOEXEC` (close-on-exec) flag set. The files that remain open inherit all attributes of the open file description from the existing process image, including file locks. File descriptors are discussed in Chapter 2 [Low-Level Input/Output], page 17.

Streams, by contrast, cannot survive through `exec` functions, because they are located in the memory of the process itself. The new process image has no streams except those it creates afresh. Each of the streams in the pre-`exec` process image has a descriptor inside it, and these descriptors do survive through `exec` (provided that they do not have `FD_CLOEXEC` set). The new process image can reconnect these to new streams using `fdopen` (see Section 2.4 [Descriptors and Streams], page 28).

7.6 Process Completion

The functions described in this section are used to wait for a child process to terminate or stop, and determine its status. These functions are declared in the header file '`sys/wait.h`'.

`pid_t` **waitpid** (`pid_t` *pid*, `int` **status-ptr*, `int` *options*) Function

The `waitpid` function is used to request status information from a child process whose process ID is *pid*. Normally, the calling process is suspended until the child process makes status information available by terminating.

Other values for the *pid* argument have special interpretations. A value of `-1` or `WAIT_ANY` requests status information for any child process; a value of `0` or `WAIT_MYPGRP` requests information for any child process in the same process group as the calling process; and any other negative value—*pgid* requests information for any child process whose process group ID is *pgid*.

If status information for a child process is available immediately, this function returns immediately without waiting. If more than one eligible child process has status information available, one of them is chosen randomly, and its status is returned immediately. To get the status from the other eligible child processes, you need to call `waitpid` again.

The *options* argument is a bit mask. Its value should be the bit-wise OR (that is, the '`|`' operator) of zero or more of the `WNOHANG` and `WUNTRACED` flags. You can use the `WNOHANG` flag to indicate that the parent process shouldn't wait; and the `WUNTRACED` flag to request status information from stopped processes as well as processes that have terminated.

The status information from the child process is stored in the object that *status-ptr* points to, unless *status-ptr* is a null pointer.

This function is a cancellation point in multithreaded programs. This is a problem if the thread allocates some resources (like memory, file descriptors, semaphores, etc.) at the time `waitpid` is called. If the thread gets canceled, these resources stay allocated until the program ends. To avoid this, calls to `waitpid` should be protected using cancellation handlers.

The return value is normally the process ID of the child process whose status is reported. If there are child processes but none of them is waiting to be noticed, `waitpid` will block until one is. However, if the `WNOHANG` option was specified, `waitpid` will return 0 instead of blocking.

If a specific PID to wait for was given to `waitpid`, it will ignore all other children (if any). Therefore if there are children waiting to be noticed but the child whose PID was specified is not one of them, `waitpid` will block or return 0 as described above.

A value of `-1` is returned in case of error. The following `errno` error conditions are defined for this function:

`EINTR` The function was interrupted by delivery of a signal to the calling process (see Section 17.5 [Primitives Interrupted by Signals], page 408).

`ECHILD` There are no child processes to wait for, or the specified *pid* is not a child of the calling process.

`EINVAL` An invalid value was provided for the *options* argument.

These symbolic constants are defined as values for the *pid* argument to the `waitpid` function.

`WAIT_ANY`
: This constant macro (whose value is `-1`) specifies that `waitpid` should return status information about any child process.

`WAIT_MYPGRP`
: This constant (with value `0`) specifies that `waitpid` should return status information about any child process in the same process group as the calling process.

These symbolic constants are defined as flags for the *options* argument to the `waitpid` function. You can bit-wise-OR the flags together to obtain a value to use as the argument.

`WNOHANG`
: This flag specifies that `waitpid` should return immediately instead of waiting, if there is no child process ready to be noticed.

`WUNTRACED`
: This flag specifies that `waitpid` should report the status of any child processes that have been stopped as well as those that have terminated.

`pid_t` **wait** (`int *`*status-ptr*) Function

This is a simplified version of `waitpid`, and is used to wait until any one child process terminates. The call:

```
wait (&status)
```

is exactly equivalent to:

```
waitpid (-1, &status, 0)
```

This function is a cancellation point in multithreaded programs. This is a problem if the thread allocates some resources (like memory, file descriptors, semaphores, etc.) at the time `wait` is called. If the thread gets canceled, these resources stay allocated until the program ends. To avoid this, calls to `wait` should be protected using cancellation handlers.

`pid_t` **wait4** (`pid_t` *pid*, `int` ⋆*status-ptr*, `int` *options*, `struct rusage` ⋆*usage*) Function

If *usage* is a null pointer, `wait4` is equivalent to `waitpid (`*pid*, *status-ptr*, *options*`)`.

If *usage* is not null, `wait4` stores usage figures for the child process in ⋆*rusage* (but only if the child has terminated, not if it has stopped) (see Section 14.1 [Resource Usage], page 335).

This function is a BSD extension.

Here's an example of how to use `waitpid` to get the status from all child processes that have terminated, without ever waiting. This function is designed to be a handler for `SIGCHLD`, the signal that indicates that at least one child process has terminated.

```
void
sigchld_handler (int signum)
{
  int pid, status, serrno;
  serrno = errno;
  while (1)
    {
      pid = waitpid (WAIT_ANY, &status, WNOHANG);
      if (pid < 0)
        {
          perror ("waitpid");
          break;
        }
      if (pid == 0)
        break;
      notice_termination (pid, status);
    }
  errno = serrno;
}
```

7.7 Process-Completion Status

If the exit status value[13] of the child process is 0, then the status value reported by `waitpid` or `wait` is also 0. You can test for other kinds of information encoded in the returned status value using the following macros. These macros are defined in the header file '`sys/wait.h`'.

`int` **WIFEXITED** (`int` *status*) Macro
This macro returns a nonzero value if the child process terminated normally with `exit` or `_exit`.

`int` **WEXITSTATUS** (`int` *status*) Macro
If `WIFEXITED` is true of *status*, this macro returns the low-order 8 bits of the exit-status value from the child process.[14]

`int` **WIFSIGNALED** (`int` *status*) Macro
This macro returns a nonzero value if the child process terminated because it received a signal that was not handled (see Chapter 17 [Signal Handling], page 377).

`int` **WTERMSIG** (`int` *status*) Macro
If `WIFSIGNALED` is true of *status*, this macro returns the signal number of the signal that terminated the child process.

`int` **WCOREDUMP** (`int` *status*) Macro
This macro returns a nonzero value if the child process terminated and produced a core dump.

`int` **WIFSTOPPED** (`int` *status*) Macro
This macro returns a nonzero value if the child process is stopped.

`int` **WSTOPSIG** (`int` *status*) Macro
If `WIFSTOPPED` is true of *status*, this macro returns the signal number of the signal that caused the child process to stop.

7.8 BSD Process Wait Functions

The GNU library also provides these related facilities for compatibility with BSD Unix. BSD uses the `union wait` data type to represent status values rather than an `int`. The two representations are actually interchangeable; they describe the same bit patterns. The GNU C Library defines macros such as `WEXITSTATUS` so that they will work on either kind of object, and the `wait` function is defined to accept either type of pointer as its *status-ptr* argument.

These functions are declared in '`sys/wait.h`'.

[13] Ibid., "Program Termination".
[14] Ibid., "Exit Status".

union wait Data Type

This data type represents program-termination status values. It has the following members:

`int w_termsig`
: The value of this member is the same as that of the `WTERMSIG` macro.

`int w_coredump`
: The value of this member is the same as that of the `WCOREDUMP` macro.

`int w_retcode`
: The value of this member is the same as that of the `WEXITSTATUS` macro.

`int w_stopsig`
: The value of this member is the same as that of the `WSTOPSIG` macro.

Instead of accessing these members directly, you should use the equivalent macros.

The `wait3` function is the predecessor to `wait4`, which is more flexible. `wait3` is now obsolete.

`pid_t` **wait3** (`union wait *`*status-ptr*`, int` *options*`, struct rusage *`*usage*) Function

If *usage* is a null pointer, `wait3` is equivalent to `waitpid (-1,` *status-ptr*`,` *options*`)`.

If *usage* is not null, `wait3` stores usage figures for the child process in `*`*rusage* (but only if the child has terminated, not if it has stopped) (see Section 14.1 [Resource Usage], page 335).

7.9 Process-Creation Example

Here is an example program showing how you might write a function similar to the built-in `system`. It executes its *command* argument using the equivalent of '`sh -c` *command*'.

```
#include <stddef.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

/* Execute the command using this shell program.  */
#define SHELL "/bin/sh"
```

```
int
my_system (const char *command)
{
  int status;
  pid_t pid;


  pid = fork ();
  if (pid == 0)
    {
      /* This is the child process.  Execute the shell command.  */
      execl (SHELL, SHELL, "-c", command, NULL);
      _exit (EXIT_FAILURE);
    }
  else if (pid < 0)
    /* The fork failed.  Report failure.  */
    status = -1;
  else
    /* This is the parent process.  Wait for the child to complete.  */
    if (waitpid (pid, &status, 0) != pid)
      status = -1;
  return status;
}
```

There are a couple of things you should pay attention to in this example.

Remember that the first `argv` argument supplied to the program represents the name of the program being executed. That is why, in the call to `execl`, `SHELL` is supplied once to name the program to execute and a second time to supply a value for `argv[0]`.

The `execl` call in the child process doesn't return if it is successful. If it fails, you must do something to make the child process terminate. Just returning a bad status code with `return` would leave two processes running the original program. Instead, the right behavior is for the child process to report failure to its parent process.

Call `_exit` to accomplish this. The reason for using `_exit` instead of `exit` is to avoid flushing fully buffered streams such as `stdout`. The buffers of these streams probably contain data that was copied from the parent process by the `fork`, data that will be output eventually by the parent process. Calling `exit` in the child would output the data twice.[15]

[15] Ibid., "Termination Internals".

8 Job Control

Job control refers to the protocol for allowing a user to move between multiple *process groups* (or *jobs*) within a single *login session*. The job control facilities are set up so that appropriate behavior for most programs happens automatically and they need not do anything special about job control. So you can probably ignore the material in this chapter unless you are writing a shell or login program.

You need to be familiar with concepts relating to process creation (see Section 7.2 [Process-Creation Concepts], page 210) and signal handling (see Chapter 17 [Signal Handling], page 377) in order to understand the material presented in this chapter.

8.1 Concepts of Job Control

The fundamental purpose of an interactive shell is to read commands from the user's terminal and create processes to execute the programs specified by those commands. It can do this using the `fork` (see Section 7.4 [Creating a Process], page 211) and `exec` (see Section 7.5 [Executing a File], page 212) functions.

A single command may run just one process—but often one command uses several processes. If you use the '`|`' operator in a shell command, you explicitly request several programs in their own processes. But even if you run just one program, it can use multiple processes internally. For example, a single compilation command such as '`cc -c foo.c`' typically uses four processes (though normally only two at any given time). If you run `make`, its job is to run other programs in separate processes.

The processes belonging to a single command are called a *process group* or *job*. This is so that you can operate on all of them at once. For example, typing `C-c` sends the signal `SIGINT` to terminate all the processes in the foreground process group.

A *session* is a larger group of processes. Normally, all the processes that stem from a single login belong to the same session.

Every process belongs to a process group. When a process is created, it becomes a member of the same process group and session as its parent process. You can put it in another process group using the `setpgid` function, provided the process group belongs to the same session.

The only way to put a process in a different session is to make it the initial process of a new session, or a *session leader*, using the `setsid` function. This also puts the session leader into a new process group, and you can't move it out of that process group again.

Usually, new sessions are created by the system login program, and the session leader is the process running the user's login shell.

A shell that supports job control must arrange to control which job can use the terminal at any particular time. Otherwise, there might be multiple jobs trying to read from the terminal at once and confusion about which process should receive

the input typed by the user. To prevent this, the shell must cooperate with the terminal driver using the protocol described in this chapter.

The shell can give unlimited access to the controlling terminal to only one process group at a time. This is called the *foreground job* on that controlling terminal. Other process groups managed by the shell that are executing without such access to the terminal are called *background jobs*.

If a background job needs to read from its controlling terminal, it is *stopped* by the terminal driver; if the `TOSTOP` mode is set, likewise for writing. The user can stop a foreground job by typing the SUSP character (see Section 6.4.9 [Special Characters], page 194) and a program can stop any job by sending it a `SIGSTOP` signal. It's the responsibility of the shell to notice when jobs stop, to notify the user about them, and to provide mechanisms for allowing the user to interactively continue stopped jobs and switch jobs between foreground and background.

See Section 8.4 [Access to the Controlling Terminal], page 223, for more information about I/O to the controlling terminal,

8.2 Job Control Is Optional

Not all operating systems support job control. The GNU system does support job control, but if you are using the GNU library on some other system, that system may not support job control itself.

You can use the `_POSIX_JOB_CONTROL` macro to test at compile-time whether the system supports job control (see Section 12.2 [Overall System Options], page 305).

If job control is not supported, then there can be only one process group per session, which behaves as if it were always in the foreground. The functions for creating additional process groups simply fail with the error code `ENOSYS`.

The macros naming the various job-control signals (see Section 17.2.5 [Job Control Signals], page 385) are defined even if job control is not supported. However, the system never generates these signals, and attempts to send a job-control signal or examine or specify their actions report errors or do nothing.

8.3 Controlling Terminal of a Process

One of the attributes of a process is its controlling terminal. Child processes created with `fork` inherit the controlling terminal from their parent process. In this way, all the processes in a session inherit the controlling terminal from the session leader. A session leader that has control of a terminal is called the *controlling process* of that terminal.

You generally do not need to worry about the exact mechanism used to allocate a controlling terminal to a session, since it is done for you by the system when you log in.

An individual process disconnects from its controlling terminal when it calls `setsid` to become the leader of a new session (see Section 8.7.2 [Process-Group Functions], page 239).

8.4 Access to the Controlling Terminal

Processes in the foreground job of a controlling terminal have unrestricted access to that terminal; background processes do not. This section describes in more detail what happens when a process in a background job tries to access its controlling terminal.

When a process in a background job tries to read from its controlling terminal, the process group is usually sent a `SIGTTIN` signal. This normally causes all of the processes in that group to stop (unless they handle the signal and don't stop themselves). However, if the reading process is ignoring or blocking this signal, then `read` fails with an `EIO` error instead.

Similarly, when a process in a background job tries to write to its controlling terminal, the default behavior is to send a `SIGTTOU` signal to the process group. However, the behavior is modified by the `TOSTOP` bit of the local-modes flags (see Section 6.4.7 [Local Modes], page 189). If this bit is not set (which is the default), then writing to the controlling terminal is always permitted without sending a signal. Writing is also permitted if the `SIGTTOU` signal is being ignored or blocked by the writing process.

Most other terminal operations that a program can do are treated as reading or as writing. The description of each operation should say which.

For more information about the primitive `read` and `write` functions, see Section 2.2 [Input and Output Primitives], page 20.

8.5 Orphaned Process-Groups

When a controlling process terminates, its terminal becomes free and a new session can be established on it. (In fact, another user could log in on the terminal.) This could cause a problem if any processes from the old session are still trying to use that terminal.

To prevent problems, process groups that continue running even after the session leader has terminated are marked as *orphaned process-groups*.

When a process group becomes an orphan, its processes are sent a `SIGHUP` signal. Ordinarily, this causes the processes to terminate. However, if a program ignores this signal or establishes a handler for it (see Chapter 17 [Signal Handling], page 377), it can continue running as in the orphan process group even after its controlling process terminates; but it still cannot access the terminal any more.

8.6 Implementing a Job-Control Shell

This section describes what a shell must do to implement job control, by presenting an extensive sample program to illustrate the concepts involved.

8.6.1 Data Structures for the Shell

All of the program examples included in this chapter are part of a simple shell program. This section presents data structures and utility functions that are used throughout the example.

The sample shell deals mainly with two data structures. The `job` type contains information about a job, which is a set of subprocesses linked together with pipes. The `process` type holds information about a single subprocess. Here are the relevant data-structure declarations:

```
/* A process is a single process.  */
typedef struct process
{
  struct process *next;       /* next process in pipeline */
  char **argv;                /* for exec */
  pid_t pid;                  /* process ID */
  char completed;             /* true if process has completed */
  char stopped;               /* true if process has stopped */
  int status;                 /* reported status value */
} process;


/* A job is a pipeline of processes.  */
```

```
typedef struct job
{
  struct job *next;           /* next active job */
  char *command;              /* command line, used for messages */
  process *first_process;     /* list of processes in this job */
  pid_t pgid;                 /* process-group ID */
  char notified;              /* true if user told about stopped job */
  struct termios tmodes;      /* saved terminal modes */
  int stdin, stdout, stderr;  /* standard i/o channels */
} job;

/* The active jobs are linked into a list.  This is its head.   */
job *first_job = NULL;
```

Here are some utility functions that are used for operating on `job` objects:

```
/* Find the active job with the indicated pgid.  */
job *
find_job (pid_t pgid)
{
  job *j;

  for (j = first_job; j; j = j->next)
    if (j->pgid == pgid)
      return j;
  return NULL;
}
```

```
/* Return true if all processes in the job have stopped or completed.  */
int
job_is_stopped (job *j)
{
  process *p;

  for (p = j->first_process; p; p = p->next)
    if (!p->completed && !p->stopped)
      return 0;
  return 1;
}
```

```
/* Return true if all processes in the job have completed.  */
int
job_is_completed (job *j)
{
  process *p;

  for (p = j->first_process; p; p = p->next)
    if (!p->completed)
      return 0;
  return 1;
}
```

8.6.2 Initializing the Shell

When a shell program that normally performs job control is started, it has to be careful, in case it has been invoked from another shell that is already doing its own job control.

A subshell that runs interactively has to ensure that it has been placed in the foreground by its parent shell before it can enable job control itself. It does this by getting its initial process-group ID with the `getpgrp` function, and comparing it to the process-group ID of the current foreground job associated with its controlling terminal (which can be retrieved using the `tcgetpgrp` function).

If the subshell is not running as a foreground job, it must stop itself by sending a `SIGTTIN` signal to its own process group. It may not arbitrarily put itself into the foreground; it must wait for the user to tell the parent shell to do this. If the subshell is continued again, it should repeat the check and stop itself again if it is still not in the foreground.

Once the subshell has been placed into the foreground by its parent shell, it can enable its own job control. It does this by calling `setpgid` to put itself into its own process group, and then calling `tcsetpgrp` to place this process group into the foreground.

When a shell enables job control, it should set itself to ignore all the job control stop signals so that it doesn't accidentally stop itself. You can do this by setting the action for all the stop signals to `SIG_IGN`.

A subshell that runs noninteractively cannot and should not support job control. It must leave all processes it creates in the same process group as the shell itself; this allows the noninteractive shell and its child processes to be treated as a single job by the parent shell. This is easy to do—just don't use any of the job-control primitives—but you must remember to make the shell do it.

Here is the initialization code for the sample shell that shows how to do all of this:

```
/* Keep track of attributes of the shell.  */
```

```
#include <sys/types.h>
#include <termios.h>
#include <unistd.h>

pid_t shell_pgid;
struct termios shell_tmodes;
int shell_terminal;
int shell_is_interactive;


/* Make sure the shell is running interactively as the foreground job
   before proceeding. */

void
init_shell ()
{

  /* See if we are running interactively.  */
  shell_terminal = STDIN_FILENO;
  shell_is_interactive = isatty (shell_terminal);

  if (shell_is_interactive)
    {
      /* Loop until we are in the foreground.  */
      while (tcgetpgrp (shell_terminal) != (shell_pgid = getpgrp ()))
        kill (- shell_pgid, SIGTTIN);

      /* Ignore interactive and job-control signals.  */
      signal (SIGINT, SIG_IGN);
      signal (SIGQUIT, SIG_IGN);
      signal (SIGTSTP, SIG_IGN);
      signal (SIGTTIN, SIG_IGN);
      signal (SIGTTOU, SIG_IGN);
      signal (SIGCHLD, SIG_IGN);

      /* Put ourselves in our own process group.  */
      shell_pgid = getpid ();
      if (setpgid (shell_pgid, shell_pgid) < 0)
        {
          perror ("Couldn't put the shell in its own process group");
          exit (1);
        }

      /* Grab control of the terminal.  */
```

```
          tcsetpgrp (shell_terminal, shell_pgid);

          /* Save default terminal attributes for the shell.  */
          tcgetattr (shell_terminal, &shell_tmodes);
        }
    }
```

8.6.3 Launching Jobs

Once the shell has taken responsibility for performing job control on its controlling terminal, it can launch jobs in response to commands typed by the user.

To create the processes in a process group, you use the same `fork` and `exec` functions described in Section 7.2 [Process-Creation Concepts], page 210. Since there are multiple child processes involved, though, things are a little more complicated and you must be careful to do things in the right order. Otherwise, nasty race conditions can result.

You have two choices for how to structure the tree of parent-child relationships among the processes. You can either make all the processes in the process group be children of the shell process, or you can make one process in a group be the ancestor of all the other processes in that group. The sample shell program presented in this chapter uses the first approach because it makes bookkeeping somewhat simpler.

As each process is forked, it should put itself in the new process group by calling `setpgid` (see Section 8.7.2 [Process-Group Functions], page 239). The first process in the new group becomes its *process group leader*, and its process ID becomes the *process group ID* for the group.

The shell should also call `setpgid` to put each of its child processes into the new process group. This is because there is a potential timing problem: each child process must be put in the process group before it begins executing a new program, and the shell depends on having all the child processes in the group before it continues executing. If both the child processes and the shell call `setpgid`, this ensures that the right things happen no matter which process gets to it first.

If the job is being launched as a foreground job, the new process group also needs to be put into the foreground on the controlling terminal using `tcsetpgrp`. Again, this should be done by the shell as well as by each of its child processes, to avoid race conditions.

The next thing each child process should do is to reset its signal actions.

During initialization, the shell process set itself to ignore job-control signals (see Section 8.6.2 [Initializing the Shell], page 226). As a result, any child processes it creates also ignore these signals by inheritance. This is definitely undesirable, so each child process should explicitly set the actions for these signals back to `SIG_DFL` just after it is forked.

Since shells follow this convention, applications can assume that they inherit the correct handling of these signals from the parent process. But every application has a responsibility not to mess up the handling of stop signals. Applications that

disable the normal interpretation of the SUSP character should provide some other mechanism for the user to stop the job. When the user invokes this mechanism, the program should send a `SIGTSTP` signal to the process group of the process, not just to the process itself (see Section 17.6.2 [Signaling Another Process], page 410).

Finally, each child process should call `exec` in the normal way. This is also the point at which redirection of the standard input and output channels should be handled (see Section 2.12 [Duplicating Descriptors], page 55).

Here is the function from the sample shell program that is responsible for launching a program. The function is executed by each child process immediately after it has been forked by the shell, and never returns.

```
void
launch_process (process *p, pid_t pgid,
                int infile, int outfile, int errfile,
                int foreground)
{
  pid_t pid;

  if (shell_is_interactive)
    {
      /* Put the process into the process group and give the process group
         the terminal, if appropriate.
         This has to be done both by the shell and in the individual
         child processes because of potential race conditions.  */
      pid = getpid ();
      if (pgid == 0) pgid = pid;
      setpgid (pid, pgid);
      if (foreground)
        tcsetpgrp (shell_terminal, pgid);

      /* Set the handling for job control signals back to the default.  */
      signal (SIGINT, SIG_DFL);
      signal (SIGQUIT, SIG_DFL);
      signal (SIGTSTP, SIG_DFL);
      signal (SIGTTIN, SIG_DFL);
      signal (SIGTTOU, SIG_DFL);
      signal (SIGCHLD, SIG_DFL);
    }

  /* Set the standard input/output channels of the new process.  */
  if (infile != STDIN_FILENO)
    {
      dup2 (infile, STDIN_FILENO);
      close (infile);
    }
```

```
      if (outfile != STDOUT_FILENO)
        {
          dup2 (outfile, STDOUT_FILENO);
          close (outfile);
        }
      if (errfile != STDERR_FILENO)
        {
          dup2 (errfile, STDERR_FILENO);
          close (errfile);
        }

      /* Exec the new process.  Make sure we exit.  */
      execvp (p->argv[0], p->argv);
      perror ("execvp");
      exit (1);
    }
```

If the shell is not running interactively, this function does not do anything with process groups or signals. Remember that a shell not performing job control must keep all of its subprocesses in the same process group as the shell itself.

Next, here is the function that actually launches a complete job. After creating the child processes, this function calls some other functions to put the newly created job into the foreground or background (see Section 8.6.4 [Foreground and Background], page 232).

```
void
launch_job (job *j, int foreground)
{
  process *p;
  pid_t pid;
  int mypipe[2], infile, outfile;

  infile = j->stdin;
  for (p = j->first_process; p; p = p->next)
    {
      /* Set up pipes, if necessary.  */
      if (p->next)
        {
          if (pipe (mypipe) < 0)
            {
              perror ("pipe");
              exit (1);
            }
          outfile = mypipe[1];
        }
      else
```

```
          outfile = j->stdout;

        /* Fork the child processes.  */
        pid = fork ();
        if (pid == 0)
          /* This is the child process.  */
          launch_process (p, j->pgid, infile,
                          outfile, j->stderr, foreground);
        else if (pid < 0)
          {
            /* The fork failed.  */
            perror ("fork");
            exit (1);
          }
        else
          {
            /* This is the parent process.  */
            p->pid = pid;
            if (shell_is_interactive)
              {
                if (!j->pgid)
                  j->pgid = pid;
                setpgid (pid, j->pgid);
              }
          }

        /* Clean up after pipes.  */
        if (infile != j->stdin)
          close (infile);
        if (outfile != j->stdout)
          close (outfile);
        infile = mypipe[0];
      }

    format_job_info (j, "launched");

    if (!shell_is_interactive)
      wait_for_job (j);
    else if (foreground)
      put_job_in_foreground (j, 0);
    else
      put_job_in_background (j, 0);
  }
```

8.6.4 Foreground and Background

Now let's consider what actions must be taken by the shell when it launches a job into the foreground, and how this differs from what must be done when a background job is launched.

When a foreground job is launched, the shell must first give it access to the controlling terminal by calling `tcsetpgrp`. Then, the shell should wait for processes in that process group to terminate or stop (see Section 8.6.5 [Stopped and Terminated Jobs], page 233).

When all of the processes in the group have either completed or stopped, the shell should regain control of the terminal for its own process group by calling `tcsetpgrp` again. Since stop signals caused by I/O from a background process or a SUSP character typed by the user are sent to the process group, normally all the processes in the job stop together.

The foreground job may have left the terminal in a strange state, so the shell should restore its own saved terminal modes before continuing. In case the job is merely stopped, the shell should first save the current terminal modes so that it can restore them later if the job is continued. The functions for dealing with terminal modes are `tcgetattr` and `tcsetattr` (see Section 6.4 [Terminal Modes], page 181).

Here is the sample shell's function for doing all of this:

```
/* Put job j in the foreground.  If cont is nonzero,
   restore the saved terminal modes and send the process group a
   SIGCONT signal to wake it up before we block.  */

void
put_job_in_foreground (job *j, int cont)
{
  /* Put the job into the foreground.  */
  tcsetpgrp (shell_terminal, j->pgid);



  /* Send the job a continue signal, if necessary.  */
  if (cont)
    {
      tcsetattr (shell_terminal, TCSADRAIN, &j->tmodes);
      if (kill (- j->pgid, SIGCONT) < 0)
        perror ("kill (SIGCONT)");
    }


  /* Wait for it to report.  */
```

```
  wait_for_job (j);

  /* Put the shell back in the foreground.  */
  tcsetpgrp (shell_terminal, shell_pgid);


  /* Restore the shell's terminal modes.  */
  tcgetattr (shell_terminal, &j->tmodes);
  tcsetattr (shell_terminal, TCSADRAIN, &shell_tmodes);
}
```

If the process group is launched as a background job, the shell should remain in the foreground itself and continue to read commands from the terminal.

In the sample shell, there is not much that needs to be done to put a job into the background. Here is the function it uses:

```
/* Put a job in the background.  If the cont argument is true, send
   the process group a SIGCONT signal to wake it up.  */

void
put_job_in_background (job *j, int cont)
{
  /* Send the job a continue signal, if necessary.  */
  if (cont)
    if (kill (-j->pgid, SIGCONT) < 0)
      perror ("kill (SIGCONT)");
}
```

8.6.5 Stopped and Terminated Jobs

When a foreground process is launched, the shell must block until all of the processes in that job have either terminated or stopped. It can do this by calling the `waitpid` function (see Section 7.6 [Process Completion], page 215). Use the `WUNTRACED` option so that status is reported for processes that stop as well as processes that terminate.

The shell must also check on the status of background jobs so that it can report terminated and stopped jobs to the user; this can be done by calling `waitpid` with the `WNOHANG` option. A good place to put a such a check for terminated and stopped jobs is just before prompting for a new command.

The shell can also receive asynchronous notification that there is status information available for a child process by establishing a handler for `SIGCHLD` signals (see Chapter 17 [Signal Handling], page 377).

In the sample shell program, the `SIGCHLD` signal is normally ignored. This is to avoid reentrancy problems involving the global data structures the shell manipulates. But at specific times when the shell is not using these data structures—such

as when it is waiting for input on the terminal—it makes sense to enable a handler for `SIGCHLD`. The same function that is used to do the synchronous status checks (`do_job_notification`, in this case) can also be called from within this handler.

Here are the parts of the sample shell program that deal with checking the status of jobs and reporting the information to the user:

```
/* Store the status of the process pid that was returned by waitpid.
   Return 0 if all went well, nonzero otherwise.  */

int
mark_process_status (pid_t pid, int status)
{
  job *j;
  process *p;



  if (pid > 0)
    {
      /* Update the record for the process.  */
      for (j = first_job; j; j = j->next)
        for (p = j->first_process; p; p = p->next)
          if (p->pid == pid)
            {
              p->status = status;
              if (WIFSTOPPED (status))
                p->stopped = 1;
              else
                {
                  p->completed = 1;
                  if (WIFSIGNALED (status))
                    fprintf (stderr, "%d: Terminated by signal %d.\n",
                             (int) pid, WTERMSIG (p->status));
                }
              return 0;
            }
      fprintf (stderr, "No child process %d.\n", pid);
      return -1;
    }


  else if (pid == 0 || errno == ECHILD)
    /* No processes ready to report  */
```

```
    return -1;
  else {
    /* Other weird errors  */
    perror ("waitpid");
    return -1;
  }
}
```

```
/* Check for processes that have status information available,
   without blocking.  */

void
update_status (void)
{
  int status;
  pid_t pid;

  do
    pid = waitpid (WAIT_ANY, &status, WUNTRACED|WNOHANG);
  while (!mark_process_status (pid, status));
}
```

```
/* Check for processes that have status information available,
   blocking until all processes in the given job have reported.  */

void
wait_for_job (job *j)
{
  int status;
  pid_t pid;

  do
    pid = waitpid (WAIT_ANY, &status, WUNTRACED);
  while (!mark_process_status (pid, status)
         && !job_is_stopped (j)
         && !job_is_completed (j));
}
```

```
/* Format information about job status for the user to look at.  */

void
format_job_info (job *j, const char *status)
{
  fprintf (stderr, "%ld (%s): %s\n", (long)j->pgid, status, j->command);
}



/* Notify the user about stopped or terminated jobs.
   Delete terminated jobs from the active job list.  */

void
do_job_notification (void)
{
  job *j, *jlast, *jnext;
  process *p;

  /* Update status information for child processes.  */
  update_status ();

  jlast = NULL;
  for (j = first_job; j; j = jnext)
    {
      jnext = j->next;

      /* If all processes have completed, tell the user the job has
         completed and delete it from the list of active jobs.  */
      if (job_is_completed (j)) {
        format_job_info (j, "completed");
        if (jlast)
          jlast->next = jnext;
        else
          first_job = jnext;
        free_job (j);
      }

      /* Notify the user about stopped jobs,
         marking them so that we won't do this more than once.  */
      else if (job_is_stopped (j) && !j->notified) {
        format_job_info (j, "stopped");
        j->notified = 1;
        jlast = j;
```

```
        }

        /* Don't say anything about jobs that are still running.  */
        else
          jlast = j;
      }
  }
```

8.6.6 Continuing Stopped Jobs

The shell can continue a stopped job by sending a `SIGCONT` signal to its process group. If the job is being continued in the foreground, the shell should first invoke `tcsetpgrp` to give the job access to the terminal, and restore the saved terminal settings. After continuing a job in the foreground, the shell should wait for the job to stop or complete, as if the job had just been launched in the foreground.

The sample shell program handles both newly created and continued jobs with the same pair of functions, `put_job_in_foreground` and `put_job_in_background`. The definitions of these functions were given in Section 8.6.4 [Foreground and Background], page 232. When continuing a stopped job, a nonzero value is passed as the *cont* argument to ensure that the `SIGCONT` signal is sent and the terminal modes reset, as appropriate.

This leaves only a function for updating the shell's internal bookkeeping about the job being continued:

```
/* Mark a stopped job J as running again.  */

void
mark_job_as_running (job *j)
{
  Process *p;

  for (p = j->first_process; p; p = p->next)
    p->stopped = 0;
  j->notified = 0;
}



/* Continue the job J.  */

void
continue_job (job *j, int foreground)
{
```

```
    mark_job_as_running (j);
    if (foreground)
      put_job_in_foreground (j, 1);
    else
      put_job_in_background (j, 1);
  }
```

8.6.7 The Missing Pieces

The code extracts for the sample shell included in this chapter are only a part of the entire shell program. In particular, nothing at all has been said about how `job` and `program` data structures are allocated and initialized.

Most real shells provide a complex user interface that has support for a command language—variables, abbreviations, substitutions, pattern matching on file names, etc. All of this is far too complicated to explain here! Instead, we have concentrated on showing how to implement the core process-creation and job-control functions that can be called from such a shell.

Here is a table summarizing the major entry points we have presented:

`void init_shell (void)`
: Initialize the shell's internal state (see Section 8.6.2 [Initializing the Shell], page 226).

`void launch_job (job *j, int foreground)`
: Launch the job *j* as either a foreground or background job (see Section 8.6.3 [Launching Jobs], page 228).

`void do_job_notification (void)`
: Check for and report any jobs that have terminated or stopped. It can be called synchronously or within a handler for `SIGCHLD` signals (see Section 8.6.5 [Stopped and Terminated Jobs], page 233).

`void continue_job (job *j, int foreground)`
: Continue the job *j* (see Section 8.6.6 [Continuing Stopped Jobs], page 237).

Of course, a real shell would also want to provide other functions for managing jobs. For example, it would be useful to have commands to list all active jobs or to send a signal (such as `SIGKILL`) to a job.

8.7 Functions for Job Control

This section contains detailed descriptions of the functions relating to job control.

8.7.1 Identifying the Controlling Terminal

You can use the `ctermid` function to get a file name that you can use to open the controlling terminal. In the GNU library, it returns the same string all the time:

'/dev/tty'. That is a special "magic" file-name that refers to the controlling terminal of the current process (if it has one). To find the name of the specific terminal device, use ttyname (see Section 6.1 [Identifying Terminals], page 179).

The function ctermid is declared in the header file 'stdio.h'.

char * **ctermid** (char **string*) Function

The ctermid function returns a string containing the file name of the controlling terminal for the current process. If *string* is not a null pointer, it should be an array that can hold at least L_ctermid characters; the string is returned in this array. Otherwise, a pointer to a string in a static area is returned, which might get overwritten on subsequent calls to this function.

An empty string is returned if the file name cannot be determined for any reason. Even if a file name is returned, access to the file it represents is not guaranteed.

int **L_ctermid** Macro

The value of this macro is an integer constant expression that represents the size of a string large enough to hold the file name returned by ctermid.

See also the isatty and ttyname functions, in Section 6.1 [Identifying Terminals], page 179.

8.7.2 Process-Group Functions

Here are descriptions of the functions for manipulating process groups. Your program should include the header files 'sys/types.h' and 'unistd.h' to use these functions.

pid_t **setsid** (void) Function

The setsid function creates a new session. The calling process becomes the session leader, and is put in a new process group whose process-group ID is the same as the process ID of that process. There are initially no other processes in the new process group, and no other process groups in the new session.

This function also makes the calling process have no controlling terminal.

The setsid function returns the new process-group ID of the calling process if successful. A return value of -1 indicates an error. The following errno error condition is defined for this function:

EPERM The calling process is already a process group leader, or there is already another process group around that has the same process-group ID.

pid_t **getsid** (pid_t *pid*) Function

The getsid function returns the process-group ID of the session leader of the specified process. If a *pid* is 0, the process-group ID of the session leader of the current process is returned.

In case of error, `-1` is returned and `errno` is set. The following `errno` error conditions are defined for this function:

`ESRCH` There is no process with the given process ID *pid*.

`EPERM` The calling process and the process specified by *pid* are in different sessions, and the implementation doesn't allow access to the process-group ID of the session leader of the process with ID *pid* from the calling process.

The `getpgrp` function has two definitions: one derived from BSD Unix, and one from the POSIX.1 standard. The feature-test macros you have selected (see Section 1.3.4 [Feature-Test Macros], page 8) determine which definition you get. Specifically, you get the BSD version if you define `_BSD_SOURCE`; otherwise, you get the POSIX version if you define `_POSIX_SOURCE` or `_GNU_SOURCE`. Programs written for old BSD systems will not include '`unistd.h`', which defines `getpgrp` specially under `_BSD_SOURCE`. You must link such programs with the `-lbsd-compat` option to get the BSD definition.

`pid_t` **getpgrp** (`void`) POSIX.1 Function

The POSIX.1 definition of `getpgrp` returns the process-group ID of the calling process.

`pid_t` **getpgrp** (`pid_t` *pid*) BSD Function

The BSD definition of `getpgrp` returns the process-group ID of the process *pid*. You can supply a value of `0` for the *pid* argument to get information about the calling process.

`int` **getpgid** (`pid_t` *pid*) System V Function

`getpgid` is the same as the BSD function `getpgrp`. It returns the process-group ID of the process *pid*. You can supply a value of `0` for the *pid* argument to get information about the calling process.

In case of error, `-1` is returned and `errno` is set. The following `errno` error condition is defined for this function:

`ESRCH` There is no process with the given process ID *pid*. The calling process and the process specified by *pid* are in different sessions, and the implementation doesn't allow access to the process-group ID of the process with ID *pid* from the calling process.

`int` **setpgid** (`pid_t` *pid*`,` `pid_t` *pgid*) Function

The `setpgid` function puts the process *pid* into the process group *pgid*. As a special case, either *pid* or *pgid* can be 0 to indicate the process ID of the calling process.

This function fails on a system that does not support job control (see Section 8.2 [Job Control Is Optional], page 222).

If the operation is successful, `setpgid` returns 0. Otherwise, it returns `-1`. The following `errno` error conditions are defined for this function:

`EACCES` The child process named by *pid* has executed an `exec` function since it was forked.

`EINVAL` The value of the *pgid* is not valid.

`ENOSYS` The system doesn't support job control.

`EPERM` The process indicated by the *pid* argument is a session leader, or is not in the same session as the calling process, or the value of the *pgid* argument doesn't match a process-group ID in the same session as the calling process.

`ESRCH` The process indicated by the *pid* argument is not the calling process or a child of the calling process.

`int` **setpgrp** (`pid_t` *pid*, `pid_t` *pgid*) Function

This is the BSD Unix name for `setpgid`. Both functions do exactly the same thing.

8.7.3 Functions for Controlling-Terminal Access

These are the functions for reading or setting the foreground process group of a terminal. You should include the header files '`sys/types.h`' and '`unistd.h`' in your application to use these functions.

Although these functions take a file-descriptor argument to specify the terminal device, the foreground job is associated with the terminal file itself and not a particular open file-descriptor.

`pid_t` **tcgetpgrp** (`int` *filedes*) Function

This function returns the process-group ID of the foreground process group associated with the terminal open on descriptor *filedes*.

If there is no foreground process group, the return value is a number greater than `1` that does not match the process group ID of any existing process group. This can happen if all of the processes in the job that was formerly the foreground job have terminated and no other job has yet been moved into the foreground.

In case of an error, a value of `-1` is returned. The following `errno` error conditions are defined for this function:

`EBADF` The *filedes* argument is not a valid file-descriptor.

`ENOSYS` The system doesn't support job control.

`ENOTTY` The terminal file associated with the *filedes* argument isn't the controlling terminal of the calling process.

`int` **tcsetpgrp** (`int` *filedes*, `pid_t` *pgid*) Function

This function is used to set a terminal's foreground process-group ID. The argument *filedes* is a descriptor that specifies the terminal; *pgid* specifies the process

group. The calling process must be a member of the same session as *pgid* and must have the same controlling terminal.

For terminal-access purposes, this function is treated as output. If it is called from a background process on its controlling terminal, normally all processes in the process group are sent a `SIGTTOU` signal. The exception is if the calling process itself is ignoring or blocking `SIGTTOU` signals, in which case the operation is performed and no signal is sent.

If successful, `tcsetpgrp` returns `0`. A return value of `-1` indicates an error. The following `errno` error conditions are defined for this function:

`EBADF` The *filedes* argument is not a valid file-descriptor.

`EINVAL` The *pgid* argument is not valid.

`ENOSYS` The system doesn't support job control.

`ENOTTY` The *filedes* isn't the controlling terminal of the calling process.

`EPERM` The *pgid* isn't a process group in the same session as the calling process.

`pid_t` **tcgetsid** (`int` *fildes*) Function

This function is used to obtain the process-group ID of the session for which the terminal specified by *fildes* is the controlling terminal. If the call is successful, the group ID is returned. Otherwise, the return value is `(pid_t) -1` and the global variable *errno* is set to one of the following values:

`EBADF` The *filedes* argument is not a valid file-descriptor.

`ENOTTY` The calling process does not have a controlling terminal, or the file is not the controlling terminal.

9 System Databases and Name-Service Switch

Various functions in the C library need to be configured to work correctly in the local environment. Traditionally, this was done by using files (e.g., '`/etc/passwd`'), but other name services (like the Network Information Service (NIS) and the Domain Name Service (DNS)) became popular, and were hacked into the C library, usually with a fixed search order.[1]

The GNU C Library contains a cleaner solution to this problem. It is designed after a method used by Sun Microsystems in the C library of Solaris 2. GNU C Library follows their name and calls this scheme *Name Service Switch* (NSS).

Though the interface might be similar to Sun's version, there is no common code. The developers never saw any source code of Sun's implementation, and so the internal interface is incompatible. This also manifests in the file names we use as we will see later.

9.1 NSS Basics

The basic idea is to put the implementation of the different services offered to access the databases in separate modules. This has some advantages:

1. Contributors can add new services without adding them to GNU C Library.
2. The modules can be updated separately.
3. The C library image is smaller.

To fulfill the first goal above, the ABI of the modules will be described below. For getting the implementation of a new service right, it is important to understand how the functions in the modules get called. They are in no way designed to be used by the programmer directly. Instead, the programmer should only use the documented and standardized functions to access the databases.

The databases available in the NSS are

`aliases` Mail aliases

`ethers` Ethernet numbers

`group` Groups of users (see Section 10.14 [Group Database], page 277)

`hosts` Host names and numbers (see Section 5.6.2.4 [Host Names], page 141)

`netgroup` Network-wide list of host and users (see Section 10.16 [Netgroup Database], page 281)

[1] *The Jargon File*, version 4.4.7. "frobnicate" (December 29, 2003), `http://www.catb.org/~esr/jargon/html/F/frobnicate.html`.

`networks`
: Network names and numbers (see Section 5.13 [Networks Database], page 176)

`protocols`
: Network protocols (see Section 5.6.6 [Protocols Database], page 147)

`passwd`
: User passwords (see Section 10.13 [User Database], page 274)

`rpc`
: Remote procedure call names and numbers

`services`
: Network services, see Section 5.6.4 [The Services Database], page 145

`shadow`
: Shadow user-passwords

There will be some more added later (`automount`, `bootparams`, `netmasks` and `publickey`).

9.2 The NSS Configuration File

Somehow the NSS code must be told about the wishes of the user. For this reason, there is the file '`/etc/nsswitch.conf`'. For each database, this file contains a specification for how the lookup process should work. The file could look like this:

```
# /etc/nsswitch.conf
#
# Name Service Switch configuration file.
#

passwd:     db files nis
shadow:     files
group:      db files nis

hosts:      files nisplus nis dns
networks:   nisplus [NOTFOUND=return] files

ethers:     nisplus [NOTFOUND=return] db files
protocols:  nisplus [NOTFOUND=return] db files
rpc:        nisplus [NOTFOUND=return] db files
services:   nisplus [NOTFOUND=return] db files
```

The first column is the database, as you can guess from the table above. The rest of the line specifies how the lookup process works. You specify the way it works for each database individually. This cannot be done with the old way of a monolithic implementation.

The configuration specification for each database can contain two different items:

- The service specification like `files`, `db` or `nis`
- The reaction on lookup result like `[NOTFOUND=return]`

9.2.1 Services in the NSS Configuration File

The above example file mentions four different services: `files`, `db`, `nis` and `nisplus`. This does not mean these services are available on all sites, nor does it mean these are all the services that will ever be available.

In fact, these names are simply strings that the NSS code uses to find the implicitly addressed functions. The internal interface will be described later. Visible to the user are the modules that implement an individual service.

Assume the service *name* will be used for a lookup. The code for this service is implemented in a module called '`libnss_`*name*'. On a system supporting shared libraries, this is in fact a shared library with the name (for example) '`libnss_`*name*`.so.2`'. The number at the end is the currently used version of the interface, which will not change frequently. Normally the user should not have to be cognizant of these files, since they should be placed in a directory where they are found automatically. Only the names of all available services are important.

9.2.2 Actions in the NSS Configuration

The second item in the specification gives the user much finer control on the lookup process. Action items are placed between two service names and are written within brackets. The general form is

`[` (`!`? *status* `=` *action*)+ `]`

where:

status ⇒ `success | notfound | unavail | tryagain`
action ⇒ `return | continue`

The case of the keywords is insignificant. The *status* values are the results of a call to a lookup function of a specific service. They mean:

'`success`'
: No error occurred and the wanted entry is returned. The default action for this is `return`.

'`notfound`'
: The lookup process worked ok, but the needed value was not found. The default action is `continue`.

'`unavail`'
: The service is permanently unavailable. This can either mean the needed file is not available, or, for DNS, the server is not available or does not allow queries. The default action is `continue`.

‘tryagain’
: The service is temporarily unavailable. This could mean a file is locked or a server currently cannot accept more connections. The default action is `continue`.

If we have a line like:

```
ethers: nisplus [NOTFOUND=return] db files
```

this is equivalent to:

```
ethers: nisplus [SUCCESS=return NOTFOUND=return UNAVAIL=continue
                 TRYAGAIN=continue]
        db      [SUCCESS=return NOTFOUND=continue UNAVAIL=continue
                 TRYAGAIN=continue]
        files
```

(except that it would have to be written on one line). The default value for the actions are normally what you want, and only need to be changed in exceptional cases.

If the optional ‘`!`’ is placed before the *status*, this means the following action is used for all statuses but *status* itself. In other words, ‘`!`’ is negation, as it is in the C language (and others).

Obviously, it makes no sense to add another action item after the `files` service. Since there is no other service following, the action *always* is `return`.

Now, why is this `[NOTFOUND=return]` action useful? To understand this, we should know that the `nisplus` service is often complete; i.e., if an entry is not available in the NIS+ tables, it is not available anywhere else. This is what is expressed by this action item—it is useless to examine further services, since they will not give us a result.

The situation would be different if the NIS+ service were not available because the machine is booting. In this case, the return value of the lookup function is not `notfound` but instead `unavail`. And as you can see in the complete form above, in this situation the `db` and `files` services are used. The system administrator need not pay special attention to the times when the system is not completely ready to work (during booting, shutdown or network problems).

9.2.3 Notes on the NSS Configuration File

The NSS implementation is not completely helpless if ‘`/etc/nsswitch.conf`’ does not exist. For all supported databases there is a default value, so it should normally be possible to get the system running even if the file is corrupted or missing.

For the `hosts` and `networks` databases, the default value is `dns [!UNAVAIL=return] files`. The system is prepared for the DNS service not to be available, but if it is available the answer it returns is definitive.

The `passwd`, `group` and `shadow` databases are traditionally handled in a special way. The appropriate files in the ‘`/etc`’ directory are read, but if an entry with

a name starting with a + character is found, NIS is used. This kind of lookup remains possible by using the special lookup-service `compat`, and the default value for the three databases above is `compat [NOTFOUND=return] files`.

For all other databases, the default value is `nis [NOTFOUND=return] files`. This solution has the best chance to be correct, since NIS and file based lookup is used.

The user should try to optimize the lookup process. The different services have different response times. A simple file lookup on a local file could be fast, but if the file is long and the needed entry is near the end of the file, this may take quite some time. In this case, it might be better to use the `db` service, which allows fast local access to large data sets.

Often, some global information like NIS must be used, so it is unavoidable to use service entries like `nis`. But you should avoid slow services like this if possible.

9.3 NSS Module Internals

The functions contained in a module are identified by their names—there is no jump table. How this is done is of no interest here; those interested in this topic should research dynamic linking.

9.3.1 The Naming Scheme of the NSS Modules

The name of each function consist of various parts:

_nss_*service*_*function*

service corresponds to the name of the module this function is found in.[2] The *function* part is derived from the interface function in the C library itself. If the user calls the function `gethostbyname` and the service used is `files`, the function:

```
_nss_files_gethostbyname_r
```

in the module:

```
libnss_files.so.2
```

is used. Actually, the NSS modules only contain reentrant versions of the lookup functions—if the user would call the `gethostbyname_r` function, this also would end in the above function. For all user interface functions, the C library maps this call to a call to the reentrant function. For reentrant functions, this is trivial since the interface is (nearly) the same. For the nonreentrant version, the library keeps internal buffers that are used to replace the user-supplied buffer.

In other words, the reentrant functions *can* have counterparts. No service module is forced to have functions for all databases and all kinds to access them. If a function is not available, it is simply treated as if the function would return `unavail` (see Section 9.2.2 [Actions in the NSS Configuration], page 245).

[2] This information is duplicated because we want to make it possible to link directly with these shared objects.

The file name ‘libnss_files.so.2’ would, on a Solaris 2 system, be ‘nss_files.so.2’. This is the difference mentioned above. Sun’s NSS modules are usable only as modules that get indirectly loaded.

The NSS modules in the GNU C Library are prepared to be used as normal libraries themselves. This is *not* true at the moment, though. However, the organization of the name space in the modules does not make it impossible like it is for Solaris. Now you can see why the modules are still libraries.

9.3.2 The Interface of the Function in NSS Modules

Now we know about the functions contained in the modules. It is now time to describe the types. Because of the reentrant versions of the functions mentioned above, there are some additional arguments (compared with the standard, nonreentrant version). The prototypes for the nonreentrant and reentrant versions of our function above are

```
struct hostent *gethostbyname (const char *name)

int gethostbyname_r (const char *name, struct hostent *result_buf,
                     char *buf, size_t buflen, struct hostent **result,
                     int *h_errnop)
```

The actual prototype of the function in the NSS modules in this case is

```
enum nss_status _nss_files_gethostbyname_r (const char *name,
                                            struct hostent *result_buf,
                                            char *buf, size_t buflen,
                                            int *errnop, int *h_errnop)
```

The interface function is in fact the reentrant function with the change of the return value and the omission of the *result* parameter. While the user-level function returns a pointer to the result, the reentrant function returns an `enum nss_status` value:

`NSS_STATUS_TRYAGAIN`
: Numeric value `-2`

`NSS_STATUS_UNAVAIL`
: Numeric value `-1`

`NSS_STATUS_NOTFOUND`
: Numeric value `0`

`NSS_STATUS_SUCCESS`
: Numeric value `1`

Now you see where the action items of the ‘/etc/nsswitch.conf’ file are used.

If you study the source code, you will find there is a fifth value, `NSS_STATUS_RETURN`. This is an internal-use-only value, used by a few functions in places

where none of the above values can be used. If necessary, the source code should be examined to learn about the details.

In case the interface function has to return an error, it is important that the correct error code is stored in **errnop*. Some return status values have only one associated error code, others have more.

NSS_STATUS_TRYAGAIN	EAGAIN	One of the functions used ran temporarily out of resources, or a service is currently not available.
	ERANGE	The provided buffer is not large enough. The function should be called again with a larger buffer.
NSS_STATUS_UNAVAIL	ENOENT	A necessary input file cannot be found.
NSS_STATUS_NOTFOUND	ENOENT	The requested entry is not available.

These are proposed values. There can be other error codes, and the described error codes can have different meaning. There is one exception: when returning NSS_STATUS_TRYAGAIN, the error code ERANGE *must* mean that the user-provided buffer is too small. Everything is noncritical.

The above function has something special that is missing from almost all the other module functions—there is an argument *h_errnop*. This points to a variable that will be filled with the error code if the execution of the function fails for some reason. The reentrant function cannot use the global variable *h_errno*; gethostbyname calls gethostbyname_r with the last argument set to &h_errno.

The get*XXX*by*YYY* functions are the most important functions in the NSS modules. But there are others that implement the different ways to access system databases. For the password database, for example, there is setpwent, getpwent and endpwent). These will be described in more detail later. Here we give a general way to determine the signature of the module function:

- The return value is int.
- The name is as explained in Section 9.3.1 [The Naming Scheme of the NSS Modules], page 247.
- The first arguments are identical to the arguments of the nonreentrant function.
- The next three arguments are

  STRUCT_TYPE *result_buf
  : This is a pointer to the buffer where the result is stored. STRUCT_TYPE is normally a struct that corresponds to the database.

  char *buffer
  : This is a pointer to a buffer where the function can store additional data for the result.

  size_t buflen
  : This is the length of the buffer pointed to by *buffer*.

- There could also be a last argument, *h_errnop*, for the host name and network-name lookup functions.

This table is correct for all functions except the `set...ent` and `end...ent` functions.

9.4 Extending NSS

One of the advantages of NSS mentioned above is that it can be extended quite easily. There are two ways in which the extension can happen: adding another database or adding another service. The former is normally done only by the C library developers. It is here only important to remember that adding another database is independent from adding another service, because a service need not support all databases or lookup functions.

A designer/implementor of a new service is therefore free to choose the databases she is interested in and leave the rest for later (or completely aside).

9.4.1 Adding Another Service to NSS

The sources for a new service need not (and should not) be part of the GNU C Library itself. The developer retains complete control over the sources and its development. The links between the C library and the new service module consist solely of the interface functions.

Each module is designed following a certain interface specification. For now, the version is 2 (the interface in version 1 was not adequate), and this manifests in the version number of the shared library object of the NSS modules—they have the extension `.2`. If the interface changes again in an incompatible way, this number will be increased. Modules using the old interface will still be usable.

Developers of a new service will have to make sure that their module is created using the correct interface number. This means the file itself must have the correct name and on ELF systems the *soname* (Shared Object Name) must also have this number. Building a module from a bunch of object files on an ELF system using GNU CC could be done like this:

```
gcc -shared -o libnss_NAME.so.2 -Wl,-soname,libnss_NAME.so.2 OBJECTS
```

noindent See Richard M. Stallman and the GCC Developer Community, "Options for Linking" in *Using GCC: The GNU Compiler Collection Reference Manual* (Boston, MA: GNU Press, October 2003), http://gcc.gnu.org/onlinedocs/gcc-3.3.2/gcc/, to learn more about this command line.

To use the new module, the library must be able to find it. This can be achieved by using options for the dynamic linker so that it will search the directory where the binary is placed. For an ELF system, this could be done by adding the wanted directory to the value of `LD_LIBRARY_PATH`.

But this is not always possible since some programs (those that run under IDs that do not belong to the user) ignore this variable. Therefore, the stable version of the module should be placed into a directory that is searched by the dynamic linker.

Normally, this should be the directory '`$prefix/lib`', where '`$prefix`' corresponds to the value given to configure using the `--prefix` option. But be careful—this should only be done if it is clear the module does not cause any harm. System administrators should be careful.

9.4.2 Internals of the NSS Module Functions

Until now we only provided the syntactic interface for the functions in the NSS module. In fact there is not much more we can say, since the implementation obviously is different for each function. But a few general rules must be followed by all functions.

There are four kinds of different functions that may appear in the interface. All derive from the traditional ones for system databases. *db* in the following table is normally an abbreviation for the database (e.g., it is `pw` for the password database).

`enum nss_status _nss_`*database*`_set`*db*`ent (void)`
: This function prepares the service for operations that will follow.. For a simple file-based lookup, this means files could be opened. For other services, this function is simply a no-op.

  One special case for this function is that it takes an additional argument for some *database*s (i.e., the interface is `int set`*db*`ent (int)`). See Section 5.6.2.4 [Host Names], page 141, which describes the `sethostent` function.

  The return value should be *NSS_STATUS_SUCCESS*, or according to the table above in case of an error (see Section 9.3.2 [The Interface of the Function in NSS Modules], page 248).

`enum nss_status _nss_`*database*`_end`*db*`ent (void)`
: This function simply closes all files that are still open or removes buffer caches. If there are no files or buffers to remove, this is again a simple noop.

  There normally is no return value other than *NSS_STATUS_SUCCESS*.

`enum nss_status _nss_`*database*`_get`*db*`ent_r (`*STRUCTURE* `*result, char *buffer, size_t buflen, int *errnop)`
: Since this function will be called several times in a row to retrieve one entry after the other, it must keep some kind of state. But this also means the functions are not really reentrant. They are reentrant only in that simultaneous calls to this function will not try to write the retrieved data in the same place (as would be the case for the nonreentrant functions); instead, they write to the structure pointed to by the *result* parameter. But the calls share a common state, and in the case of a file access this means they return neighboring entries in the file.

  The buffer of length *buflen* pointed to by *buffer* can be used for storing some additional data for the result. It is *not* guaranteed that the same

buffer will be passed for the next call of this function, so you must not misuse this buffer to save some state information from one call to another.

Before the function returns, the implementation should store the value of the local *errno* variable in the variable pointed to be *errnop*. This is important to guarantee the module working in statically linked programs.

As explained above, this function could also have an additional last argument. This depends on the database used; it happens only for `host` and `networks`.

The function shall return `NSS_STATUS_SUCCESS` as long as there are more entries. When the last entry was read, it should return `NSS_STATUS_NOTFOUND`. When the buffer given as an argument is too small for the data to be returned, `NSS_STATUS_TRYAGAIN` should be returned. When the service was not formerly initialized by a call to `_nss_`*DATABASE*`_set`*db*`ent`, any return values allowed for this function can also be returned here.

`enum nss_status _nss_`*DATABASE*`_get`*db*`by`*XX*`_r (`*PARAMS*`, `*STRUCTURE*` `*`result, char *`buffer, size_t buflen, int *`errnop)`

This function will return the entry from the database that is addressed by the *PARAMS*. The type and number of these arguments vary. It must be individually determined by looking to the user-level interface functions. All arguments given to the nonreentrant version are here described by *PARAMS*.

The result must be stored in the structure pointed to by *result*. If there is additional data to return (such as strings, where the *result* structure only contains pointers), the function must use the *buffer* or length *buflen*. There must not be any references to nonconstant global data.

The implementation of this function should honor the *stayopen* flag set by the `set`*DB*`ent` function whenever this makes sense.

Before the function returns, the implementation should store the value of the local *errno* variable in the variable pointed to by *errnop*. This is important to guarantee that the module work in statically linked programs.

Again, this function takes an additional last argument for the `host` and `networks` database.

The return value should, as always, follow the rules given above (see Section 9.3.2 [The Interface of the Function in NSS Modules], page 248).

10 Users and Groups

Every user who can log in on the system is identified by a unique number called the *user ID*. Each process has an effective user-ID that says which user's access-permissions it has.

Users are classified into *groups* for access-control purposes. Each process has one or more *group-ID values* that say which groups the process can use for access to files.

The effective user and group-IDs of a process collectively form its *persona*. This determines which files the process can access. Normally, a process inherits its persona from the parent process, but under special circumstances a process can change its persona, and thus change its access permissions.

Each file in the system also has a user ID and a group ID. Access control works by comparing the user- and group-IDs of the file with those of the running process.

The system keeps a database of all the registered users, and another database of all the defined groups. There are library functions you can use to examine these databases.

10.1 User- and Group-IDs

Each user account on a computer system is identified by a *user name* (or *login name*) and *user ID*. Normally, each user name has a unique user-ID, but it is possible for several login names to have the same user-ID. The user names and corresponding user-IDs are stored in a database that you can access as described in Section 10.13 [User Database], page 274.

Users are classified in *groups*. Each user name belongs to one *default group* and may also belong to any number of *supplementary groups*. Users who are members of the same group can share resources (such as files) that are not accessible to users who are not a member of that group. Each group has a *group name* and *group ID*. See Section 10.14 [Group Database], page 277, for how to find information about a group ID or group name.

10.2 The Persona of a Process

At any time, each process has an *effective user ID*, an *effective group ID*, and a set of *supplementary group IDs*. These IDs determine the privileges of the process. They are collectively called the *persona* of the process, because they determine "who it is" for purposes of access control.

Your login shell starts out with a persona that consists of your user ID, your default group-ID and your supplementary group-IDs (if you are in more than one group). In normal circumstances, all your other processes inherit these values.

A process also has a *real user-ID*, which identifies the user who created the process, and a *real group-ID*, which identifies that user's default group. These values

do not play a role in access control, so we do not consider them part of the persona. But they are also important.

Both the real and effective user-ID can be changed during the lifetime of a process (see Section 10.3 [Why Change the Persona of a Process?], page 254).

For details on how a process's effective user-ID and group-IDs affect its permission to access files, see Section 3.9.6 [How Your Access to a File is Decided], page 104.

The effective user-ID of a process also controls permissions for sending signals using the `kill` function (see Section 17.6.2 [Signaling Another Process], page 410).

Finally, there are many operations that can only be performed by a process whose effective user-ID is 0. A process with this user ID is a *privileged process*. Commonly, the user name `root` is associated with user ID 0, but there may be other user names with this ID.

10.3 Why Change the Persona of a Process?

The most obvious situation where it is necessary for a process to change its user- and/or group-IDs is the `login` program. When `login` starts running, its user ID is `root`. Its job is to start a shell whose user- and group-IDs are those of the user who is logging in. (To accomplish this fully, `login` must set the real user- and group-IDs as well as its persona. But this is a special case.)

The more common case of changing persona is when an ordinary user program needs access to a resource that wouldn't ordinarily be accessible to the user actually running it.

For example, you may have a file that is controlled by your program but that shouldn't be read or modified directly by other users, either because it implements some kind of locking protocol, or because you want to preserve the integrity or privacy of the information it contains. This kind of restricted access can be implemented by having the program change its effective user or group ID to match that of the resource.

Thus, imagine a game program that saves scores in a file. The game program itself needs to be able to update this file no matter who is running it, but if users can write the file without going through the game, they can give themselves any scores they like. Some people consider this undesirable, or even reprehensible. It can be prevented by creating a new user-ID and login name (say, `games`) to own the scores file, and making the file writable only by this user. Then, when the game program wants to update this file, it can change its effective user-ID to be that for `games`. In effect, the program must adopt the persona of `games` so it can write the scores file.

10.4 How an Application Can Change Persona

The ability to change the persona of a process can be a source of unintentional privacy violations, or even intentional abuse. Because of the potential for problems, changing persona is restricted to special circumstances.

You can't arbitrarily set your user ID or group ID to anything you want; only privileged processes can do that. Instead, the normal way for a program to change its persona is that it has been set up in advance to change to a particular user or group. This is the function of the setuid and setgid bits of a file's access mode (see Section 3.9.5 [The Mode Bits for Access Permission], page 102).

When the setuid bit of an executable file is on, executing that file gives the process a third user-ID—the *file user-ID*. This ID is set to the owner ID of the file. The system then changes the effective user-ID to the file user-ID. The real user-ID remains as it was. Likewise, if the setgid bit is on, the process is given a *file group ID* equal to the group ID of the file, and its effective group-ID is changed to the file group-ID.

If a process has a file ID (user or group), then it can at any time change its effective ID to its real ID and back to its file ID. Programs use this feature to relinquish their special privileges except when they actually need them. This makes it less likely that they can be tricked into doing something inappropriate with their privileges.

Portability Note: Older systems do not have file IDs. To determine if a system has this feature, you can test the compiler define `_POSIX_SAVED_IDS`. (In the POSIX standard, file IDs are known as saved IDs.)

See Section 3.9 [File Attributes], page 93, for a more general discussion of file modes and accessibility.

10.5 Reading the Persona of a Process

Here are detailed descriptions of the functions for reading the user- and group-IDs of a process, both real and effective. To use these facilities, you must include the header files '`sys/types.h`' and '`unistd.h`'.

uid_t Data Type

This is an integer data type used to represent user IDs. In the GNU library, this is an alias for `unsigned int`.

gid_t Data Type

This is an integer data type used to represent group IDs. In the GNU library, this is an alias for `unsigned int`.

`uid_t` **getuid** (`void`) Function

The `getuid` function returns the real user-ID of the process.

`gid_t` **getgid** (`void`) Function
The `getgid` function returns the real group-ID of the process.

`uid_t` **geteuid** (`void`) Function
The `geteuid` function returns the effective user-ID of the process.

`gid_t` **getegid** (`void`) Function
The `getegid` function returns the effective group-ID of the process.

`int` **getgroups** (`int` *count*, `gid_t *`*groups*) Function
The `getgroups` function is used to inquire about the supplementary group-IDs of the process. Up to *count* of these group IDs are stored in the array *groups*; the return value from the function is the number of group IDs actually stored. If *count* is smaller than the total number of supplementary group-IDs, then `getgroups` returns a value of `-1`, and `errno` is set to `EINVAL`.

If *count* is 0, then `getgroups` just returns the total number of supplementary group-IDs. On systems that do not support supplementary groups, this will always be 0.

Here's how to use `getgroups` to read all the supplementary group IDs:

```
gid_t *
read_all_groups (void)
{
  int ngroups = getgroups (0, NULL);
  gid_t *groups
    = (gid_t *) xmalloc (ngroups * sizeof (gid_t));
  int val = getgroups (ngroups, groups);
  if (val < 0)
    {
      free (groups);
      return NULL;
    }
  return groups;
}
```

10.6 Setting the User ID

This section describes the functions for altering the user ID, real and/or effective, of a process. To use these facilities, you must include the header files '`sys/types.h`' and '`unistd.h`'.

`int` **seteuid** (`uid_t` *neweuid*) Function
This function sets the effective user-ID of a process to *newuid*, provided that the process is allowed to change its effective user-ID. A privileged process (effective

user-ID zero) can change its effective user-ID to any legal value. An unprivileged process with a file user-ID can change its effective user-ID to its real user-ID or to its file user-ID. Otherwise, a process may not change its effective user-ID at all.

The `seteuid` function returns a value of `0` to indicate successful completion, and a value of `-1` to indicate an error. The following `errno` error conditions are defined for this function:

`EINVAL` The value of the *newuid* argument is invalid.

`EPERM` The process may not change to the specified ID.

Older systems (those without the `_POSIX_SAVED_IDS` feature) do not have this function.

`int` **setuid** (`uid_t` *newuid*) Function

If the calling process is privileged, this function sets both the real and effective user-ID of the process to *newuid*. It also deletes the file user-ID of the process, if any. *newuid* may be any legal value. Once this has been done, there is no way to recover the old effective user-ID.

If the process is not privileged, and the system supports the `_POSIX_SAVED_IDS` feature, then this function behaves like `seteuid`.

The return values and error conditions are the same as for `seteuid`.

`int` **setreuid** (`uid_t` *ruid*, `uid_t` *euid*) Function

This function sets the real user-ID of the process to *ruid* and the effective user-ID to *euid*. If *ruid* is `-1`, it means not to change the real user-ID; likewise if *euid* is `-1`, it means not to change the effective user-ID.

The `setreuid` function exists for compatibility with 4.3 BSD Unix, which does not support file IDs. You can use this function to swap the effective and real user-IDs of the process. (Privileged processes are not limited to this particular usage.) If file IDs are supported, you should use that feature instead of this function (see Section 10.8 [Enabling and Disabling Setuid Access], page 260).

The return value is `0` on success and `-1` on failure. The following `errno` error condition is defined for this function:

`EPERM` The process does not have the appropriate privileges; you do not have permission to change to the specified ID.

10.7 Setting the Group IDs

This section describes the functions for altering the group IDs, real and effective, of a process. To use these facilities, you must include the header files ‘`sys/types.h`’ and ‘`unistd.h`’.

`int` **setegid** (`gid_t` *newgid*) Function

This function sets the effective group-ID of the process to *newgid*, provided that the process is allowed to change its group ID. Just as with `seteuid`, if the process is privileged, it may change its effective group-ID to any value. If it isn't, but it has a file group-ID, then it may change to its real group-ID or file group-ID. Otherwise, it may not change its effective group-ID.

A process is only privileged if its effective *user*-ID is zero. The effective group-ID only affects access permissions.

The return values and error conditions for `setegid` are the same as those for `seteuid`.

This function is only present if `_POSIX_SAVED_IDS` is defined.

`int` **setgid** (`gid_t` *newgid*) Function

This function sets both the real and effective group-ID of the process to *newgid*, provided that the process is privileged. It also deletes the file group-ID, if any.

If the process is not privileged, then `setgid` behaves like `setegid`.

The return values and error conditions for `setgid` are the same as those for `seteuid`.

`int` **setregid** (`gid_t` *rgid*, `gid_t` *egid*) Function

This function sets the real group-ID of the process to *rgid* and the effective group-ID to *egid*. If *rgid* is `-1`, it means not to change the real group-ID; likewise if *egid* is `-1`, it means not to change the effective group-ID.

The `setregid` function is provided for compatibility with 4.3 BSD Unix, which does not support file IDs. You can use this function to swap the effective and real group-IDs of the process. (Privileged processes are not limited to this usage.) If file IDs are supported, you should use that feature instead of using this function (see Section 10.8 [Enabling and Disabling Setuid Access], page 260).

The return values and error conditions for `setregid` are the same as those for `setreuid`.

`setuid` and `setgid` behave differently depending on whether the effective user-ID at the time is zero. If it is not zero, they behave like `seteuid` and `setegid`. If it is, they change both effective and real IDs and delete the file ID. To avoid confusion, we recommend you always use `seteuid` and `setegid` except when you know the effective user-ID is zero, and your intent is to change the persona permanently. This case is rare—most of the programs that need it, such as `login` and `su`, have already been written.

If your program is setuid to some user other than `root`, there is no way to drop privileges permanently.

The system also lets privileged processes change their supplementary group-IDs. To use `setgroups` or `initgroups`, your programs should include the header file '`grp.h`'.

`int` **setgroups** (`size_t` *count*, `gid_t *`*groups*) Function

This function sets the process's supplementary group-IDs. It can only be called from privileged processes. The *count* argument specifies the number of group IDs in the array *groups*.

This function returns `0` if successful and `-1` on error. The following `errno` error condition is defined for this function:

`EPERM` The calling process is not privileged.

`int` **initgroups** (`const char *`*user*, `gid_t` *group*) Function

The `initgroups` function sets the process's supplementary group IDs to be the normal default for the user name *user*. The group *group* is automatically included.

This function works by scanning the group database for all the groups *user* belongs to. It then calls `setgroups` with the list it has constructed.

The return values and error conditions are the same as for `setgroups`.

If you are interested in the groups a particular user belongs to, but do not want to change the process's supplementary group-IDs, you can use `getgrouplist`. To use `getgrouplist`, your programs should include the header file '`grp.h`'.

`int` **getgrouplist** (`const char *`*user*, `gid_t` *group*, `gid_t *`*groups*, `int *`*ngroups*) Function

The `getgrouplist` function scans the group database for all the groups *user* belongs to. Up to **ngroups* group IDs corresponding to these groups are stored in the array *groups*; the return value from the function is the number of group IDs actually stored. If **ngroups* is smaller than the total number of groups found, then `getgrouplist` returns a value of `-1` and stores the actual number of groups in **ngroups*. The group *group* is automatically included in the list of groups returned by `getgrouplist`.

Here's how to use `getgrouplist` to read all supplementary groups for *user*:

```
gid_t *
supplementary_groups (char *user)
{
  int ngroups = 16;
  gid_t *groups
    = (gid_t *) xmalloc (ngroups * sizeof (gid_t));
  struct passwd *pw = getpwnam (user);

  if (pw == NULL)
    return NULL;

  if (getgrouplist (pw->pw_name, pw->pw_gid, groups, &ngroups) < 0)
    {
```

```
      groups = xrealloc (ngroups * sizeof (gid_t));
      getgrouplist (pw->pw_name, pw->pw_gid, groups, &ngroups);
    }
  return groups;
}
```

10.8 Enabling and Disabling Setuid Access

A typical setuid program does not need its special access all of the time. It's a good idea to turn off this access when it isn't needed, to avoid giving unintended access.

If the system supports the `_POSIX_SAVED_IDS` feature, you can accomplish this with `seteuid`. When the game program starts, its real user-ID is `jdoe`, its effective user-ID is `games`, and its saved user-ID is also `games`. The program should record both user ID values once at the beginning, like this:

```
user_user_id = getuid ();
game_user_id = geteuid ();
```

Then it can turn off game file access with

```
seteuid (user_user_id);
```

and turn it on with

```
seteuid (game_user_id);
```

Throughout this process, the real user-ID remains `jdoe` and the file user-ID remains `games`, so the program can always set its effective user-ID to either one.

On other systems that don't support file user-IDs, you can turn setuid access on and off by using `setreuid` to swap the real and effective user-IDs of the process, as follows:

```
setreuid (geteuid (), getuid ());
```

This special case is always allowed—it cannot fail.

Why does this have the effect of toggling the setuid access? Suppose a game program has just started, and its real user-ID is `jdoe` while its effective user-ID is `games`. In this state, the game can write the scores file. If it swaps the two uids, the real becomes `games` and the effective becomes `jdoe`; now the program has only `jdoe` access. Another swap brings `games` back to the effective user-ID and restores access to the scores file.

In order to handle both kinds of systems, test for the saved user-ID feature with a preprocessor conditional, like this:

```
#ifdef _POSIX_SAVED_IDS
  seteuid (user_user_id);
#else
  setreuid (geteuid (), getuid ());
#endif
```

10.9 Setuid Program Example

Here's an example showing how to set up a program that changes its effective user-ID.

This is part of a game program called `caber-toss` that manipulates a file '`scores`' that should be writable only by the game program itself. The program assumes that its executable file will be installed with the setuid bit set and owned by the same user as the '`scores`' file. Typically, a system administrator will set up an account like `games` for this purpose.

The executable file is given mode `4755`, so that doing an '`ls -l`' on it produces output like:

```
-rwsr-xr-x   1 games    184422 Jul 30 15:17 caber-toss
```

The setuid bit shows up in the file modes as the '`s`'.

The scores file is given mode `644`, and doing an '`ls -l`' on it shows:

```
-rw-r--r--  1 games           0 Jul 31 15:33 scores
```

Here are the parts of the program that show how to set up the changed user-ID. This program is conditionalized so that it makes use of the file IDs feature if it is supported, and otherwise uses `setreuid` to swap the effective and real user-IDs.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>


/* Remember the effective and real UIDs. */

static uid_t euid, ruid;


/* Restore the effective UID to its original value. */

void
do_setuid (void)
{
  int status;

#ifdef _POSIX_SAVED_IDS
  status = seteuid (euid);
#else
  status = setreuid (ruid, euid);
#endif
  if (status < 0) {
    fprintf (stderr, "Couldn't set uid.\n");
```

```
    exit (status);
    }
}



/* Set the effective UID to the real UID.  */

void
undo_setuid (void)
{
  int status;

#ifdef _POSIX_SAVED_IDS
  status = seteuid (ruid);
#else
  status = setreuid (euid, ruid);
#endif
  if (status < 0) {
    fprintf (stderr, "Couldn't set uid.\n");
    exit (status);
    }
}



/* Main program.  */

int
main (void)
{
  /* Remember the real and effective user-IDs.  */
  ruid = getuid ();
  euid = geteuid ();
  undo_setuid ();

  /* Do the game and record the score.  */
  ...
}
```

Notice that the first thing the `main` function does is set the effective user-ID back to the real user-ID. This is so that any other file accesses that are performed while the user is playing the game use the real user-ID for determining permissions. Only when the program needs to open the scores file does it switch back to the file user-ID, like this:

```
/* Record the score.  */
```

```
int
record_score (int score)
{
  FILE *stream;
  char *myname;

  /* Open the scores file. */
  do_setuid ();
  stream = fopen (SCORES_FILE, "a");
  undo_setuid ();


  /* Write the score to the file. */
  if (stream)
    {
      myname = cuserid (NULL);
      if (score < 0)
        fprintf (stream, "%10s: Couldn't lift the caber.\n", myname);
      else
        fprintf (stream, "%10s: %d feet.\n", myname, score);
      fclose (stream);
      return 0;
    }
  else
    return -1;
}
```

10.10 Tips for Writing Setuid Programs

It is easy for setuid programs to give the user access that isn't intended—in fact, if you want to avoid this, you need to be careful. Here are some guidelines for preventing unintended access and minimizing its consequences when it does occur:

- Don't have `setuid` programs with privileged user-IDs such as `root` unless it is absolutely necessary. If the resource is specific to your particular program, it's better to define a new, nonprivileged user-ID or group ID just to manage that resource. It's better if you can write your program to use a special group rather than a special user.
- Be cautious about using the `exec` functions in combination with changing the effective user-ID. Don't let users of your program execute arbitrary programs under a changed user-ID. Executing a shell is especially bad news. Less obviously, the `execlp` and `execvp` functions are a potential risk (since the program they execute depends on the user's `PATH` environment variable).

If you must `exec` another program under a changed ID, specify an absolute file-name[1] for the executable, and make sure that the protections on that executable and *all* containing directories are such that ordinary users cannot replace it with some other program.

You should also check the arguments passed to the program to make sure they do not have unexpected effects. Likewise, you should examine the environment variables. Decide which arguments and variables are safe, and reject all others.

You should never use `system` in a privileged program, because it invokes a shell.

- Only use the user ID controlling the resource in the part of the program that actually uses that resource. When you're finished with it, restore the effective user-ID back to the actual user's user-ID (see Section 10.8 [Enabling and Disabling Setuid Access], page 260).
- If the `setuid` part of your program needs to access other files besides the controlled resource, it should verify that the real user would ordinarily have permission to access those files. You can use the `access` function (see Section 3.9.6 [How Your Access to a File is Decided], page 104) to check this; it uses the real user- and group-IDs, rather than the effective IDs.

10.11 Identifying Who Is Logged In

You can use the functions listed in this section to determine the login name of the user who is running a process, and the name of the user who is logged in in the current session. See also the function `getuid` and friends (see Section 10.5 [Reading the Persona of a Process], page 255). How this information is collected by the system and how to control, add and remove information from the background storage is described in Section 10.12 [The User-Accounting Database], page 265.

The `getlogin` function is declared in '`unistd.h`', while `cuserid` and `L_cuserid` are declared in '`stdio.h`'.

`char *` **getlogin** (`void`) Function

The `getlogin` function returns a pointer to a string containing the name of the user logged in on the controlling terminal of the process, or a null pointer if this information cannot be determined. The string is statically allocated and might be overwritten on subsequent calls to this function or to `cuserid`.

`char *` **cuserid** (`char *`*string*) Function

The `cuserid` function returns a pointer to a string containing a user name associated with the effective ID of the process. If *string* is not a null pointer, it should be an array that can hold at least `L_cuserid` characters; the string is returned in this array. Otherwise, a pointer to a string in a static area is returned.

[1] See Loosemore et al., "File-Name Resolution" (see chap. 1, n. 1).

This string is statically allocated and might be overwritten on subsequent calls to this function or to `getlogin`.

The use of this function is deprecated since it is marked to be withdrawn in XPG 4.2 and has already been removed from newer revisions of POSIX.1.

`int` **L_cuserid** Macro

An integer constant that indicates how long an array you might need to store a user name.

These functions let your program identify positively the user who is running or the user who is logged in in this session. These can differ when setuid programs are involved (see Section 10.2 [The Persona of a Process], page 253). The user cannot do anything to fool these functions.

For most purposes, it is more useful to use the environment variable `LOGNAME` to find out who the user is. This is more flexible precisely because the user can set `LOGNAME` arbitrarily.[2]

10.12 The User-Accounting Database

Most Unix-like operating systems keep track of logged-in users by maintaining a user-accounting database. This user-accounting database stores for each terminal, who has logged on, at what time, the process ID of the user's login shell, etc., but it also stores information about the run level of the system, the time of the last system reboot, and possibly more. The user-accounting database typically lives in '`/etc/utmp`', '`/var/adm/utmp`' or '`/var/run/utmp`'. However, these files should *never* be accessed directly. For reading information from and writing information to the user-accounting database, the functions described in this section should be used.

10.12.1 Manipulating the User-Accounting Database

These functions and the corresponding data structures are declared in the header file '`utmp.h`'.

struct exit_status Data Type

The `exit_status` data structure is used to hold information about the exit status of processes marked as `DEAD_PROCESS` in the user-accounting database.

`short int e_termination`
: This is the exit status of the process.

`short int e_exit`
: This is the exit status of the process.

[2] Ibid., "Standard Environment Variables".

struct utmp Data Type

The `utmp` data structure is used to hold information about entries in the user-accounting database. On the GNU system, it has the following members:

`short int ut_type`
: This specifies the type of login; one of `EMPTY`, `RUN_LVL`, `BOOT_TIME`, `OLD_TIME`, `NEW_TIME`, `INIT_PROCESS`, `LOGIN_PROCESS`, `USER_PROCESS`, `DEAD_PROCESS` or `ACCOUNTING`.

`pid_t ut_pid`
: This is the process ID-number of the login process.

`char ut_line[]`
: This is the device name of the tty (without '`/dev/`').

`char ut_id[]`
: This is the inittab ID of the process.

`char ut_user[]`
: This is the user's login name.

`char ut_host[]`
: This is the name of the host from which the user logged in.

`struct exit_status ut_exit`
: This is the exit status of a process marked as `DEAD_PROCESS`.

`long ut_session`
: This is the session ID, used for windowing.

`struct timeval ut_tv`
: This is the time the entry was made. For entries of type `OLD_TIME`, this is the time when the system clock changed. For entries of type `NEW_TIME`, this is the time the system clock was set to.

`int32_t ut_addr_v6[4]`
: This is the Internet address of a remote host.

The `ut_type`, `ut_pid`, `ut_id`, `ut_tv` and `ut_host` fields are not available on all systems. Portable applications therefore should be prepared for these situations. To help in this, the '`utmp.h`' header provides macros `_HAVE_UT_TYPE`, `_HAVE_UT_PID`, `_HAVE_UT_ID`, `_HAVE_UT_TV` and `_HAVE_UT_HOST` if the respective fields are available. The programmer can handle the situations by using `#ifdef` in the program code.

The following macros are defined for use as values for the `ut_type` member of the `utmp` structure. The values are integer constants.

`EMPTY` This macro is used to indicate that the entry contains no valid user-accounting information.

`RUN_LVL` This macro is used to identify the system's runlevel.

`BOOT_TIME`
: This macro is used to identify the time of system boot.

`OLD_TIME`
: This macro is used to identify the time when the system clock changed.

`NEW_TIME`
: This macro is used to identify the time after the system changed.

`INIT_PROCESS`
: This macro is used to identify a process spawned by the init process.

`LOGIN_PROCESS`
: This macro is used to identify the session leader of a logged-in user.

`USER_PROCESS`
: This macro is used to identify a user process.

`DEAD_PROCESS`
: This macro is used to identify a terminated process.

`ACCOUNTING`
: ???

The size of the `ut_line`, `ut_id`, `ut_user` and `ut_host` arrays can be found using the `sizeof` operator.

Many older systems have, instead of an `ut_tv` member, an `ut_time` member, usually of type `time_t`, for representing the time associated with the entry. Therefore, for backward compatibility only, '`utmp.h`' defines `ut_time` as an alias for `ut_tv.tv_sec`.

`void` **setutent** (`void`) Function

This function opens the user-accounting database to begin scanning it. You can then call `getutent`, `getutid` or `getutline` to read entries and `pututline` to write entries.

If the database is already open, it resets the input to the beginning of the database.

`struct utmp *` **getutent** (`void`) Function

The `getutent` function reads the next entry from the user-accounting database. It returns a pointer to the entry, which is statically allocated and may be overwritten by subsequent calls to `getutent`. You must copy the contents of the structure if you wish to save the information, or you can use the `getutent_r` function, which stores the data in a user-provided buffer.

A null pointer is returned if no further entry is available.

`void` **endutent** (`void`) Function

This function closes the user-accounting database.

`struct utmp *` **getutid** (`const struct utmp *`*id*) Function

This function searches forward from the current point in the database for an entry that matches *id*. If the `ut_type` member of the *id* structure is one of `RUN_LVL`, `BOOT_TIME`, `OLD_TIME` or `NEW_TIME`, the entries match if the `ut_type` members are identical. If the `ut_type` member of the *id* structure is `INIT_PROCESS`, `LOGIN_PROCESS`, `USER_PROCESS` or `DEAD_PROCESS`, the entries match if the `ut_type` member of the entry read from the database is one of these four and the `ut_id` members match. However, if the `ut_id` member of either the *id* structure or the entry read from the database is empty, it checks if the `ut_line` members match instead. If a matching entry is found, `getutid` returns a pointer to the entry, which is statically allocated, and may be overwritten by a subsequent call to `getutent`, `getutid` or `getutline`. You must copy the contents of the structure if you wish to save the information.

A null pointer is returned if the end of the database is reached without a match.

The `getutid` function may cache the last read entry. Therefore, if you are using `getutid` to search for multiple occurrences, it is necessary to zero out the static data after each call. Otherwise, `getutid` could just return a pointer to the same entry over and over again.

`struct utmp *` **getutline** (`const struct utmp *`*line*) Function

This function searches forward from the current point in the database until it finds an entry whose `ut_type` value is `LOGIN_PROCESS` or `USER_PROCESS`, and whose `ut_line` member matches the `ut_line` member of the *line* structure. If it finds such an entry, it returns a pointer to the entry that is statically allocated, and may be overwritten by a subsequent call to `getutent`, `getutid` or `getutline`. You must copy the contents of the structure if you wish to save the information.

A null pointer is returned if the end of the database is reached without a match.

The `getutline` function may cache the last read entry. Therefore, if you are using `getutline` to search for multiple occurrences, it is necessary to zero out the static data after each call. Otherwise, `getutline` could just return a pointer to the same entry over and over again.

`struct utmp *` **pututline** (`const struct utmp *`*utmp*) Function

The `pututline` function inserts the entry `*`*utmp* at the appropriate place in the user-accounting database. If it finds that it is not already at the correct place in the database, it uses `getutid` to search for the position to insert the entry. However, this will not modify the static structure returned by `getutent`, `getutid` and `getutline`. If this search fails, the entry is appended to the database.

The `pututline` function returns a pointer to a copy of the entry inserted in the user-accounting database, or a null pointer if the entry could not be added. The following `errno` error condition is defined for this function:

`EPERM` The process does not have the appropriate privileges—you cannot modify the user-accounting database.

All the `get*` functions mentioned before store the information they return in a static buffer. This can be a problem in multithreaded programs, since the data returned for the request is overwritten by the return-value data in another thread. Therefore, the GNU C Library provides as extensions three more functions that return the data in a user-provided buffer.

`int` **getutent_r** (`struct utmp *`*buffer*`, struct utmp **`*result*) Function

The `getutent_r` is equivalent to the `getutent` function. It returns the next entry from the database. But instead of storing the information in a static buffer, it stores it in the buffer pointed to by the parameter *buffer*.

If the call was successful, the function returns `0` and the pointer variable pointed to by the parameter *result* contains a pointer to the buffer that contains the result (probably the same value as *buffer*). If something went wrong during the execution of `getutent_r`, the function returns `-1`.

This function is a GNU extension.

`int` **getutid_r** (`const struct utmp *`*id*`, struct utmp *`*buffer*`, struct utmp **`*result*) Function

This function retrieves, just like `getutid`, the next entry matching the information stored in *id*. But the result is stored in the buffer pointed to by the parameter *buffer*.

If successful, the function returns `0` and the pointer variable pointed to by the parameter *result* contains a pointer to the buffer with the result (probably the same as *result*). If not successful, the function returns `-1`.

This function is a GNU extension.

`int` **getutline_r** (`const struct utmp *`*line*`, struct utmp *`*buffer*`, struct utmp **`*result*) Function

This function retrieves, just like `getutline`, the next entry matching the information stored in *line*. But the result is stored in the buffer pointed to by the parameter *buffer*.

If successful, the function returns `0` and the pointer variable pointed to by the parameter *result* contains a pointer to the buffer with the result (probably the same as *result*). If not successful, the function returns `-1`.

This function is a GNU extension.

In addition to the user-accounting database, most systems keep a number of similar databases. For example, most systems keep a log file with all previous logins (usually in '`/etc/wtmp`' or '`/var/log/wtmp`').

For specifying which database to examine, the following function should be used:

`int` **utmpname** (`const char *`*file*) Function

The `utmpname` function changes the name of the database to be examined to *file*, and closes any previously opened database. By default, `getutent`, `getutid`, `getutline` and `pututline` read from and write to the user-accounting database.

The following macros are defined for use as the *file* argument:

`char *` **_PATH_UTMP** Macro

This macro is used to specify the user-accounting database.

`char *` **_PATH_WTMP** Macro

This macro is used to specify the user-accounting log file.

The `utmpname` function returns a value of `0` if the new name was successfully stored, and a value of `-1` to indicate an error. `utmpname` does not try to open the database, and therefore the return value does not say anything about whether the database can be successfully opened.

For maintaining log-like databases, the GNU C Library provides the following function:

`void` **updwtmp** (`const char *`*wtmp_file*`, const struct utmp *`*utmp*) Function

The `updwtmp` function appends the entry **utmp* to the database specified by *wtmp_file*. For possible values for the *wtmp_file* argument, see the `utmpname` function.

Portability Note: Although many operating systems provide a subset of these functions, they are not standardized. There are often subtle differences in the return types, and there are considerable differences between the various definitions of `struct utmp`. When programming for the GNU system, it is probably best to stick with the functions described in this section. If however, you want your program to be portable, consider using the XPG functions described in Section 10.12.2 [XPG User-Accounting Database Functions], page 270, or take a look at the BSD-compatible functions in Section 10.12.3 [Logging In and Out], page 273.

10.12.2 XPG User-Accounting Database Functions

These functions, described in the *X/Open Portability Guide*, are declared in the header file '`utmpx.h`'.[3]

struct utmpx Data Type

The `utmpx` data structure contains at least the following members:

[3] X/Open Company, *X/Open Portability Guide,* Issue 4 (Reading, UK: X/Open Company, Ltd., 1992).

`short int ut_type`
: This specifies the type of login; one of `EMPTY`, `RUN_LVL`, `BOOT_TIME`, `OLD_TIME`, `NEW_TIME`, `INIT_PROCESS`, `LOGIN_PROCESS`, `USER_PROCESS` or `DEAD_PROCESS`.

`pid_t ut_pid`
: This is the process-ID number of the login process.

`char ut_line[]`
: This is the device name of the tty (without '`/dev/`').

`char ut_id[]`
: This is the inittab ID of the process.

`char ut_user[]`
: This is the user's login name.

`struct timeval ut_tv`
: This is the time the entry was made. For entries of type `OLD_TIME`, this is the time when the system clock changed. For entries of type `NEW_TIME`, this is the time the system clock was set to.

On the GNU system, `struct utmpx` is identical to `struct utmp`, except that including '`utmpx.h`' does not make visible the declaration of `struct exit_status`.

The following macros are defined for use as values for the `ut_type` member of the `utmpx` structure. The values are integer constants and are, on the GNU system, identical to the definitions in '`utmp.h`'.

`EMPTY` This macro is used to indicate that the entry contains no valid user-accounting information.

`RUN_LVL` This macro is used to identify the system's runlevel.

`BOOT_TIME`
: This macro is used to identify the time of system boot.

`OLD_TIME`
: This macro is used to identify the time when the system clock changed.

`NEW_TIME`
: This macro is used to identify the time after the system changed.

`INIT_PROCESS`
: This macro is used to identify a process spawned by the init process.

`LOGIN_PROCESS`
: This macro is used to identify the session leader of a logged-in user.

`USER_PROCESS`
: This macro is used to identify a user process.

`DEAD_PROCESS`
This macro is used to identify a terminated process.

The size of the `ut_line`, `ut_id` and `ut_user` arrays can be found using the `sizeof` operator.

`void` **setutxent** (`void`) Function
This function is similar to `setutent`. On the GNU system, it is simply an alias for `setutent`.

`struct utmpx *` **getutxent** (`void`) Function
The `getutxent` function is similar to `getutent`, but returns a pointer to a `struct utmpx` instead of `struct utmp`. On the GNU system, it simply is an alias for `getutent`.

`void` **endutxent** (`void`) Function
This function is similar to `endutent`. On the GNU system, it is simply an alias for `endutent`.

`struct utmpx *` **getutxid** (`const struct utmpx *`*id*) Function
This function is similar to `getutid`, but uses `struct utmpx` instead of `struct utmp`. On the GNU system, it is simply an alias for `getutid`.

`struct utmpx *` **getutxline** (`const struct utmpx *`*line*) Function
This function is similar to `getutid`, but uses `struct utmpx` instead of `struct utmp`. On the GNU system, it is simply an alias for `getutline`.

`struct utmpx *` **pututxline** (`const struct utmpx *`*utmp*) Function

The `pututxline` function is functionally identical to `pututline`, but uses `struct utmpx` instead of `struct utmp`. On the GNU system, `pututxline` is simply an alias for `pututline`.

`int` **utmpxname** (`const char *`*file*) Function
The `utmpxname` function is functionally identical to `utmpname`. On the GNU system, `utmpxname` is simply an alias for `utmpname`.

You can translate between a traditional `struct utmp` and an XPG `struct utmpx` with the following functions. On the GNU system, these functions are merely copies, since the two structures are identical.

`int` **getutmp** (`const struct utmpx *utmpx, struct utmp *utmp`) Function
`getutmp` copies the information, insofar as the structures are compatible, from *utmpx* to *utmp*.

`int` **getutmpx** (`const struct utmp *utmp, struct utmpx *utmpx`) Function

`getutmpx` copies the information, insofar as the structures are compatible, from *utmp* to *utmpx*.

10.12.3 Logging In and Out

These functions, derived from BSD, are available in the separate '`libutil`' library, and are declared in '`utmp.h`'.

The `ut_user` member of `struct utmp` is called `ut_name` in BSD. Therefore, `ut_name` is defined as an alias for `ut_user` in '`utmp.h`'.

`int` **login_tty** (`int` *filedes*) Function

This function makes *filedes* the controlling terminal of the current process, redirects standard input, standard output and standard error output to this terminal, and closes *filedes*.

This function returns `0` on successful completion and `-1` on error.

`void` **login** (`const struct utmp *`*entry*) Function

The `login` functions inserts an entry into the user-accounting database. The `ut_line` member is set to the name of the terminal on standard input. If standard input is not a terminal, `login` uses standard output or standard error output to determine the name of the terminal. If `struct utmp` has a `ut_type` member, `login` sets it to `USER_PROCESS`, and if there is an `ut_pid` member, it will be set to the process ID of the current process. The remaining entries are copied from *entry*.

A copy of the entry is written to the user-accounting log file.

`int` **logout** (`const char *`*ut_line*) Function

This function modifies the user-accounting database to indicate that the user on *ut_line* has logged out.

The `logout` function returns `1` if the entry was successfully written to the database and `0` on error.

`void` **logwtmp** (`const char *`*ut_line*`, const char *`*ut_name*`, const char *`*ut_host*) Function

The `logwtmp` function appends an entry to the user-accounting log file, for the current time and the information provided in the *ut_line*, *ut_name* and *ut_host* arguments.

Portability Note: The BSD `struct utmp` only has the `ut_line`, `ut_name`, `ut_host` and `ut_time` members. Older systems do not even have the `ut_host` member.

10.13 User Database

This section describes how to search and scan the database of registered users. The database itself is kept in the file '`/etc/passwd`' on most systems, but on some systems a special network server gives access to it.

10.13.1 The Data Structure That Describes a User

The functions and data structures for accessing the system user-database are declared in the header file '`pwd.h`'.

struct passwd Data Type

The `passwd` data-structure is used to hold information about entries in the system user-database. It has at least the following members:

`char *pw_name`
: This is the user's login name.

`char *pw_passwd.`
: This is the encrypted password string.

`uid_t pw_uid`
: This is the user ID number.

`gid_t pw_gid`
: This is the user's default group-ID number.

`char *pw_gecos`
: This is a string typically containing the user's real name and possibly other information, such as a phone number.

`char *pw_dir`
: This is the user's home directory, or initial working directory. This might be a null pointer, in which case the interpretation is system dependent.

`char *pw_shell`
: This is the user's default shell, or the initial program run when the user logs in. This might be a null pointer, indicating that the system default should be used.

10.13.2 Looking Up One User

You can search the system user-database for information about a specific user using `getpwuid` or `getpwnam`. These functions are declared in '`pwd.h`'.

`struct passwd *` **getpwuid** (`uid_t` *uid*) Function

This function returns a pointer to a statically allocated structure containing information about the user whose user ID is *uid*. This structure may be overwritten on subsequent calls to `getpwuid`.

A null pointer value indicates there is no user in the database with user ID *uid*.

`int` **getpwuid_r** (`uid_t` *uid*, `struct passwd` ⋆*result_buf*, `char` ⋆*buffer*, `size_t` *buflen*, `struct passwd` ⋆⋆*result*) Function

This function is similar to `getpwuid` in that it returns information about the user whose user ID is *uid*. However, it fills the user-supplied structure pointed to by *result_buf* with the information instead of using a static buffer. The first *buflen* bytes of the additional buffer pointed to by *buffer* are used to contain additional information, normally strings which are pointed to by the elements of the result structure.

If a user with ID *uid* is found, the pointer returned in *result* points to the record that contains the wanted data (i.e., *result* contains the value *result_buf*). If no user is found or if an error occurred, the pointer returned in *result* is a null pointer. The function returns 0 or an error code. If the buffer *buffer* is too small to contain all the needed information, the error code `ERANGE` is returned and *errno* is set to `ERANGE`.

`struct passwd` ⋆ **getpwnam** (`const char` ⋆*name*) Function

This function returns a pointer to a statically allocated structure containing information about the user whose user name is *name*. This structure may be overwritten on subsequent calls to `getpwnam`.

A null pointer return indicates there is no user named *name*.

`int` **getpwnam_r** (`const char` ⋆*name*, `struct passwd` ⋆*result_buf*, `char` ⋆*buffer*, `size_t` *buflen*, `struct passwd` ⋆⋆*result*) Function

This function is similar to `getpwnam` in that is returns information about the user whose user name is *name*. However, like `getpwuid_r`, it fills the user-supplied buffers in *result_buf* and *buffer* with the information instead of using a static buffer.

The return values are the same as for `getpwuid_r`.

10.13.3 Scanning the List of All Users

This section explains how a program can read the list of all users in the system, one user at a time. The functions described here are declared in '`pwd.h`'.

You can use the `fgetpwent` function to read user entries from a particular file.

`struct passwd` ⋆ **fgetpwent** (`FILE` ⋆*stream*) Function

This function reads the next user entry from *stream* and returns a pointer to the entry. The structure is statically allocated and is rewritten on subsequent calls to `fgetpwent`. You must copy the contents of the structure if you wish to save the information.

The stream must correspond to a file in the same format as the standard password database file.

`int` **fgetpwent_r** (`FILE *`*stream*`, struct passwd *`*result_buf*`, char *`*buffer*`, size_t` *buflen*`, struct passwd **`*result*) Function

This function is similar to `fgetpwent` in that it reads the next user entry from *stream*, but the result is returned in the structure pointed to by *result_buf*. The first *buflen* bytes of the additional buffer pointed to by *buffer* are used to contain additional information, normally strings that are pointed to by the elements of the result structure.

The stream must correspond to a file in the same format as the standard password-database file.

If the function returns 0, *result* points to the structure with the wanted data (normally this is in *result_buf*). If errors occurred, the return value is nonzero and *result* contains a null pointer.

The way to scan all the entries in the user database is with `setpwent`, `getpwent` and `endpwent`.

`void` **setpwent** (`void`) Function

This function initializes a stream that `getpwent` and `getpwent_r` use to read the user database.

`struct passwd *` **getpwent** (`void`) Function

The `getpwent` function reads the next entry from the stream initialized by `setpwent`. It returns a pointer to the entry. The structure is statically allocated and is rewritten on subsequent calls to `getpwent`. You must copy the contents of the structure if you wish to save the information.

A null pointer is returned when no more entries are available.

`int` **getpwent_r** (`struct passwd *`*result_buf*`, char *`*buffer*`,` `int` *buflen*`, struct passwd **`*result*) Function

This function is similar to `getpwent` in that it returns the next entry from the stream initialized by `setpwent`. Like `fgetpwent_r`, it uses the user-supplied buffers in *result_buf* and *buffer* to return the information requested.

The return values are the same as for `fgetpwent_r`.

`void` **endpwent** (`void`) Function

This function closes the internal stream used by `getpwent` or `getpwent_r`.

10.13.4 Writing a User Entry

`int` **putpwent** (`const struct passwd *`*p*`, FILE *`*stream*) Function

This function writes the user entry *p* to the stream *stream*, in the format used for the standard user-database file. The return value is 0 on success and nonzero on failure.

This function exists for compatibility with SVID. We recommend that you avoid using it, because it makes sense only on the assumption that the `struct passwd` structure has no members except the standard ones; on a system that merges the traditional Unix database with other extended information about users, adding an entry using this function would inevitably leave out much of the important information.

The function `putpwent` is declared in '`pwd.h`'.

10.14 Group Database

This section describes how to search and scan the database of registered groups. The database itself is kept in the file '`/etc/group`' on most systems, but on some systems a special network service provides access to it.

10.14.1 The Data Structure for a Group

The functions and data structures for accessing the system group-database are declared in the header file '`grp.h`'.

struct group Data Type

The `group` structure is used to hold information about an entry in the system group-database. It has at least the following members:

`char *gr_name`
: This is the name of the group.

`gid_t gr_gid`
: This is the group ID of the group.

`char **gr_mem`
: This is a vector of pointers to the names of users in the group. Each user name is a null-terminated string, and the vector itself is terminated by a null pointer.

10.14.2 Looking Up One Group

You can search the group database for information about a specific group using `getgrgid` or `getgrnam`. These functions are declared in '`grp.h`'.

`struct group *` **getgrgid** (`gid_t` *gid*) Function

This function returns a pointer to a statically allocated structure containing information about the group whose group ID is *gid*. This structure may be overwritten by subsequent calls to `getgrgid`.

A null pointer indicates there is no group with ID *gid*.

`int` **getgrgid_r** (`gid_t` *gid*, `struct group` `*`*result_buf*, `char` `*`*buffer*, `size_t` *buflen*, `struct group` `**`*result*) Function

This function is similar to `getgrgid` in that it returns information about the group whose group ID is *gid*. However, it fills the user-supplied structure pointed to by *result_buf* with the information instead of using a static buffer. The first *buflen* bytes of the additional buffer pointed to by *buffer* are used to contain additional information, normally strings that are pointed to by the elements of the result structure.

If a group with ID *gid* is found, the pointer returned in *result* points to the record that contains the wanted data (i.e., *result* contains the value *result_buf*). If no group is found or if an error occurred, the pointer returned in *result* is a null pointer. The function returns 0 or an error code. If the buffer *buffer* is too small to contain all the needed information, the error code `ERANGE` is returned and *errno* is set to `ERANGE`.

`struct group *` **getgrnam** (`const char *`*name*) Function

This function returns a pointer to a statically allocated structure containing information about the group whose group name is *name*. This structure may be overwritten by subsequent calls to `getgrnam`.

A null pointer indicates there is no group named *name*.

`int` **getgrnam_r** (`const char *`*name*, `struct group` `*`*result_buf*, `char *`*buffer*, `size_t` *buflen*, `struct group` `**`*result*) Function

This function is similar to `getgrnam` in that is returns information about the group whose group name is *name*. Like `getgrgid_r`, it uses the user-supplied buffers in *result_buf* and *buffer*, not a static buffer.

The return values are the same as for `getgrgid_r` `ERANGE`.

10.14.3 Scanning the List of All Groups

This section explains how a program can read the list of all groups in the system, one group at a time. The functions described here are declared in '`grp.h`'.

You can use the `fgetgrent` function to read group entries from a particular file.

`struct group *` **fgetgrent** (`FILE *`*stream*) Function

The `fgetgrent` function reads the next entry from *stream*. It returns a pointer to the entry. The structure is statically allocated and is overwritten on subsequent calls to `fgetgrent`. You must copy the contents of the structure if you wish to save the information.

The stream must correspond to a file in the same format as the standard group-database file.

`int` **fgetgrent_r** (`FILE *`*stream*`, struct group *`*result_buf*`, char *`*buffer*`, size_t` *buflen*`, struct group **`*result*) Function

This function is similar to `fgetgrent` in that it reads the next user entry from *stream*, but the result is returned in the structure pointed to by *result_buf*. The first *buflen* bytes of the additional buffer pointed to by *buffer* are used to contain additional information, normally strings that are pointed to by the elements of the result structure.

This stream must correspond to a file in the same format as the standard group-database file.

If the function returns 0, *result* points to the structure with the wanted data (normally this is in *result_buf*). If errors occurred, the return value is nonzero and *result* contains a null pointer.

The way to scan all the entries in the group database is with `setgrent`, `getgrent` and `endgrent`.

`void` **setgrent** (`void`) Function

This function initializes a stream for reading from the group database. You use this stream by calling `getgrent` or `getgrent_r`.

`struct group *` **getgrent** (`void`) Function

The `getgrent` function reads the next entry from the stream initialized by `setgrent`. It returns a pointer to the entry. The structure is statically allocated and is overwritten on subsequent calls to `getgrent`. You must copy the contents of the structure if you wish to save the information.

`int` **getgrent_r** (`struct group *`*result_buf*`, char *`*buffer*`, size_t` *buflen*`, struct group **`*result*) Function

This function is similar to `getgrent` in that it returns the next entry from the stream initialized by `setgrent`. Like `fgetgrent_r`, it places the result in user-supplied buffers pointed to *result_buf* and *buffer*.

If the function returns 0, *result* contains a pointer to the data (normally equal to *result_buf*). If errors occurred, the return value is nonzero and *result* contains a null pointer.

`void` **endgrent** (`void`) Function

This function closes the internal stream used by `getgrent` or `getgrent_r`.

10.15 User- and Group- Database Example

Here is an example program showing the use of the system database-inquiry functions. The program prints some information about the user running the program.

```
#include <grp.h>
#include <pwd.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

int
main (void)
{
  uid_t me;
  struct passwd *my_passwd;
  struct group *my_group;
  char **members;

  /* Get information about the user ID.  */
  me = getuid ();
  my_passwd = getpwuid (me);
  if (!my_passwd)
    {
      printf ("Couldn't find out about user %d.\n", (int) me);
      exit (EXIT_FAILURE);
    }

  /* Print the information.  */
  printf ("I am %s.\n", my_passwd->pw_gecos);
  printf ("My login name is %s.\n", my_passwd->pw_name);
  printf ("My uid is %d.\n", (int) (my_passwd->pw_uid));
  printf ("My home directory is %s.\n", my_passwd->pw_dir);
  printf ("My default shell is %s.\n", my_passwd->pw_shell);

  /* Get information about the default group ID.  */
  my_group = getgrgid (my_passwd->pw_gid);
  if (!my_group)
    {
      printf ("Couldn't find out about group %d.\n",
              (int) my_passwd->pw_gid);
      exit (EXIT_FAILURE);
    }

  /* Print the information.  */
  printf ("My default group is %s (%d).\n",
          my_group->gr_name, (int) (my_passwd->pw_gid));
  printf ("The members of this group are:\n");
  members = my_group->gr_mem;
```

```
  while (*members)
    {
      printf ("  %s\n", *(members));
      members++;
    }

  return EXIT_SUCCESS;
}
```

Here is some output from this program:

```
I am Throckmorton Snurd.
My login name is snurd.
My uid is 31093.
My home directory is /home/fsg/snurd.
My default shell is /bin/sh.
My default group is guest (12).
The members of this group are:
  friedman
  tami
```

10.16 Netgroup Database

10.16.1 Netgroup Data

Sometimes it is useful to group users according to other criteria (see Section 10.14 [Group Database], page 277). It is useful to associate a certain group of users with a certain machine. But grouping of host names is not yet supported.

In Sun Microsystems SunOS, a new kind of database appeared—the netgroup database. It allows grouping hosts, users and domains freely, giving them individual names. To be more concrete, a netgroup is a list of triples consisting of a host name, a user name, and a domain name where any of the entries can be a wild-card entry matching all inputs. A last possibility is that names of other netgroups can also be given in the list specifying a netgroup. So you can construct arbitrary hierarchies without loops.

Sun's implementation allows netgroups only for the `nis` or `nisplus` service (see Section 9.2.1 [Services in the NSS Configuration File], page 245). The implementation in the GNU C Library has no such restriction. An entry in either of the input services must have the following form:

groupname `(` *groupname* `|` `(`*hostname*`,`*username*`,`domainname`)` `)+`

Any of the fields in the triple can be empty, which means anything matches. While describing the functions, we will see that the opposite case is useful as well—there may be entries that will not match any input. For entries like this, a name consisting of the single character '–' will be used.

10.16.2 Looking Up One Netgroup

The lookup functions for netgroups are a bit different from all other system database-handling functions. Since a single netgroup can contain many entries, a two-step process is needed. First you select a single netgroup, and then you can iterate over all entries in this netgroup. These functions are declared in 'netdb.h'.

`int` **setnetgrent** (`const char *`*netgroup*) Function

A call to this function initializes the internal state of the library to allow calls of the `getnetgrent` to iterate over all entries in the netgroup with name *netgroup*.

When the call is successful (when a netgroup with this name exists) the return value is `1`. When the return value is `0`, no netgroup of this name is known or some other error occurred.

It is important to remember that there is only one single state for iterating the netgroups. Even if the programmer uses the `getnetgrent_r` function, the result is not really reentrant, since only one single netgroup at a time can ever be processed. If the program needs to process more than one netgroup simultaneously, the programmer must protect this by using external locking. This problem was introduced in the original netgroups implementation in SunOS, and since the GNU C Library must stay compatible, it is not possible to change this.

Some other functions also use the netgroups state. Currently, these are the `innetgr` function and parts of the implementation of the `compat` service part of the NSS implementation.

`int` **getnetgrent** (`char **`*hostp*`, char **`*userp*`, char **`*domainp*) Function

This function returns the next unprocessed entry of the currently selected netgroup. The string pointers, in which addresses are passed in the arguments *hostp*, *userp* and *domainp*, will contain after a successful call pointers to appropriate strings. If the string in the next entry is empty, the pointer has the value `NULL`. The returned string-pointers are only valid if none of the netgroup-related functions are called.

The return value is `1` if the next entry was successfully read. A value of `0` means no further entries exist or internal errors occurred.

`int` **getnetgrent_r** (`char **`*hostp*`, char **`*userp*`, char **`*domainp*`, char *`*buffer*`, int` *buflen*) Function

This function is similar to `getnetgrent` with only one exception—the strings the three string pointers *hostp*, *userp* and *domainp* point to are placed in the buffer of *buflen* bytes starting at *buffer*. This means the returned values are valid even after other netgroup-related functions are called.

The return value is `1` if the next entry was successfully read and the buffer contains enough room to place the strings in it. `0` is returned in case no more entries are found, the buffer is too small or internal errors occurred.

This function is a GNU extension. The original implementation in the SunOS libc does not provide this function.

`void` **endnetgrent** (`void`) Function

This function frees all buffers that were allocated to process the last selected netgroup. As a result, all string pointers returned by calls to `getnetgrent` are invalid afterward.

10.16.3 Testing for Netgroup Membership

It is often not necessary to scan the whole netgroup, since often the only interesting question is whether a given entry is part of the selected netgroup.

`int` **innetgr** (`const char *`*netgroup*`, const char *`*host*`, const char *`*user*`, const char *`*domain*) Function

This function tests whether the triple specified by the parameters *hostp*, *userp* and *domainp* is part of the netgroup *netgroup*. Using this function has these two advantages:

1. No other netgroup function can use the global netgroup state since internal locking is used.
2. The function is implemented more efficiently than successive calls to the other `set`/`get`/`endnetgrent` functions.

Any of the pointers *hostp*, *userp* or *domainp* can be `NULL`, which means any value is accepted in this position. This is also true for the name '`-`' which should not match any other string otherwise.

The return value is `1` if an entry matching the given triple is found in the netgroup. The return value is `0` if the netgroup itself is not found, the netgroup does not contain the triple or internal errors occurred.

11 System Management

This chapter describes facilities for controlling the system that underlies a process, including the operating system and hardware, and for getting information about it. Anyone can generally use the informational facilities, but usually only a properly privileged process can make changes.

To get information on parameters of the system that are built into the system, such as the maximum length of a file name, Chapter 12 [System-Configuration Parameters], page 303.

11.1 Host Identification

This section explains how to identify the particular system on which your program is running. First, let's review the various ways computer systems are named, which is a little complicated because of the history of the development of the Internet.

Every Unix system (also known as a host) has a host name, whether it's connected to a network or not. In its simplest form, as used before computer networks were an issue, it's just a word like '`chicken`'.

But any system attached to the Internet or any network like it conforms to a more rigorous naming convention as part of the Domain Name System (DNS). In DNS, every host name is composed of both a hostname and a domain name:

You will note that "hostname" looks a lot like "host name", but it is not the same thing. People often incorrectly refer to entire host names as "domain names."

In DNS, the full host name is properly called the FQDN (Fully Qualified Domain Name) and consists of the hostname, then a period, then the domain name. The domain name itself usually has multiple components separated by periods. So for example, a system's hostname may be '`chicken`' and its domain name might be '`ai.mit.edu`', so its FQDN (which is its host name) is '`chicken.ai.mit.edu`'.

Adding to the confusion, though, is that DNS is not the only namespace in which a computer needs to be known. Another namespace is the NIS (aka YP) namespace. For NIS purposes, there is another domain name, which is called the NIS domain name or the YP domain name. It need not have anything to do with the DNS domain name.

Confusing things even more is the fact that in DNS, it is possible for multiple FQDNs to refer to the same system. However, there is always exactly one of them that is the true host name, and it is called the canonical FQDN.

In some contexts, the host name is called a "node name."

For more information on DNS host naming, see Section 5.6.2.4 [Host Names], page 141.

Prototypes for these functions appear in '`unistd.h`'.

The programs `hostname`, `hostid` and `domainname` work by calling these functions.

`int` **gethostname** (`char *`*name*`, size_t` *size*) Function

This function returns the host name of the system on which it is called, in the array *name*. The *size* argument specifies the size of this array, in bytes. This is *not* the DNS hostname. If the system participates in DNS, this is the FQDN (see above).

The return value is `0` on success and `-1` on failure. In the GNU C Library, `gethostname` fails if *size* is not large enough; then you can try again with a larger array. The following `errno` error condition is defined for this function:

`ENAMETOOLONG`
The *size* argument is less than the size of the host name plus 1.

On some systems, there is a symbol for the maximum possible host name length, `MAXHOSTNAMELEN`. It is defined in '`sys/param.h`'. But you can't count on this to exist, so it is cleaner to handle failure and try again.

`gethostname` stores the beginning of the host name in *name* even if the host name won't entirely fit. For some purposes, a truncated host name is good enough. If it is, you can ignore the error code.

`int` **sethostname** (`const char *`*name*`, size_t` *length*) Function

The `sethostname` function sets the host name of the system that calls it to *name*, a string with length *length*. Only privileged processes are permitted to do this.

Usually `sethostname` gets called just once, at system boot time. Often, the program that calls it sets it to the value it finds in the file `/etc/hostname`.

Be sure to set the host name to the full host name, not just the DNS hostname (see above).

The return value is `0` on success and `-1` on failure. The following `errno` error condition is defined for this function:

`EPERM` This process cannot set the host name because it is not privileged.

`int` **getdomainnname** (`char *`*name*`, size_t` *length*) Function

`getdomainname` returns the NIS (aka YP) domain name of the system on which it is called. This is not the more popular DNS domain name. Get that with `gethostname`.

The specifics of this function are analogous to `gethostname`, above.

`int` **setdomainname** (`const char *`*name*`, size_t` *length*) Function

`getdomainname` sets the NIS (aka YP) domain name of the system on which it is called. This is not the more popular DNS domain name. Set that with `sethostname`.

The specifics of this function are analogous to `sethostname`, above.

`long int` **gethostid** (`void`) Function

This function returns the "host ID" of the machine the program is running on. By convention, this is usually the primary Internet IP address of that machine, converted to a `long int`. However, on some systems it is a meaningless but unique number that is hard coded for each machine.

This is not widely used. It arose in BSD 4.2, but was dropped in BSD 4.4. It is not required by POSIX.

The proper way to query the IP address is to use `gethostbyname` on the results of `gethostname`. For more information on IP addresses, see Section 5.6.2 [Host Addresses], page 136.

`int` **sethostid** (`long int` *id*) Function

The `sethostid` function sets the "host ID" of the host machine to *id*. Only privileged processes are permitted to do this. Usually it happens just once, at system boot time.

The proper way to establish the primary IP address of a system is to configure the IP address resolver to associate that IP address with the system's host name as returned by `gethostname`. For example, put a record for the system in '`/etc/hosts`'.

See `gethostid` above for more information on host ids.

The return value is `0` on success and `-1` on failure. The following `errno` error conditions are defined for this function:

`EPERM` This process cannot set the host name because it is not privileged.

`ENOSYS` The operating system does not support setting the host ID. On some systems, the host ID is a meaningless but unique number hard coded for each machine.

11.2 Platform-Type Identification

You can use the `uname` function to find out some information about the type of computer your program is running on. This function and the associated data type are declared in the header file '`sys/utsname.h`'.

As a bonus, `uname` also gives some information identifying the particular system your program is running on. This is the same information that you can get with functions targetted to this purpose described in Section 11.1 [Host Identification], page 285.

struct utsname Data Type

The `utsname` structure is used to hold information returned by the `uname` function. It has the following members:

`char sysname[]`
This is the name of the operating system in use.

`char release[]`
: This is the current release level of the operating system implementation.

`char version[]`
: This is the current version level within the release of the operating system.

`char machine[]`
: This is a description of the type of hardware that is in use.

  Some systems provide a mechanism to interrogate the kernel directly for this information. On systems without such a mechanism, the GNU C library fills in this field based on the configuration name that was specified when building and installing the library.

  GNU uses a three-part name to describe a system configuration; the three parts are *cpu*, *manufacturer* and *system-type*, and they are separated with dashes. Any possible combination of three names is potentially meaningful, but most such combinations are meaningless in practice and even the meaningful ones are not necessarily supported by any particular GNU program.

  Since the value in `machine` is supposed to describe just the hardware, it consists of the first two parts of the configuration name: '*cpu*`-`*manufacturer*'. For example, it might be one of these:

  `"sparc-sun"`, `"i386-`*anything*`"`, `"m68k-hp"`, `"m68k-sony"`, `"m68k-sun"`, `"mips-dec"`

`char nodename[]`
: This is the host name of this particular computer. In the GNU C library, the value is the same as that returned by `gethostname` (see Section 11.1 [Host Identification], page 285).

  gethostname() is implemented with a call to uname().

`char domainname[]`
: This is the NIS or YP domain name. It is the same value returned by `getdomainname` (see Section 11.1 [Host Identification], page 285). This element is a relatively recent invention and use of it is not as portable as use of the rest of the structure.

`int` **uname** (`struct utsname *`*info*) Function

The `uname` function fills in the structure pointed to by *info* with information about the operating system and host machine. A nonnegative value indicates that the data was successfully stored.

`-1` as the value indicates an error. The only error possible is `EFAULT`, which we normally don't mention as it is always a possibility.

11.3 Controlling and Querying Mounts

All files are in file systems, and before you can access any file, its file system must be mounted. Because of Unix's concept of *Everything is a file*, mounting of file systems is central to doing almost anything. This section explains how to find out what file systems are currently mounted, what file systems are available for mounting and how to change what is mounted.

The classic file-system is the contents of a disk drive. The concept is considerably more abstract, though, and lots of things other than disk drives can be mounted.

Some block devices don't correspond to traditional devices like disk drives. For example, a loop device is a block device whose driver uses a regular file in another file-system as its medium. So if that regular file contains appropriate data for a file system, you can essentially mount a regular file by mounting the loop device.

Some file systems aren't based on a device of any kind. The "proc" file-system, for example, contains files whose data is made up by the file-system driver on the fly whenever you ask for it. And when you write to it, the data you write causes changes in the system. No data gets stored.

11.3.1 Mount Information

For some programs, it is desirable and necessary to access information about whether a certain file-system is mounted and, if it is, where, or simply to get lists of all the available file-systems. The GNU libc provides some functions to retrieve this information portably.

Traditionally, Unix systems have a file named '`/etc/fstab`' that describes all possibly mounted file-systems. The `mount` program uses this file to mount at start-up time of the system all the necessary file-systems. The information about all the file systems actually mounted is normally kept in a file named either '`/var/run/mtab`' or '`/etc/mtab`'. Both files share the same syntax and it is crucial that this syntax is followed all the time. Therefore it is best to never directly write the files. The functions described in this section can do this, and they also provide the functionality to convert the external textual representation to the internal representation.

The '`fstab`' and '`mtab`' files are maintained on a system by *convention*. It is possible for the files not to exist or not to be consistent with what is really mounted or available to mount, if the system's administration policy allows it. But programs that mount and unmount file systems typically maintain and use these files as described herein.

The filenames given above should never be used directly. The portable way to handle these files is to use the macro `_PATH_FSTAB`, defined in '`fstab.h`', or `_PATH_MNTTAB`, defined in '`mntent.h`' and '`paths.h`', for '`fstab`'; and the macro `_PATH_MOUNTED`, also defined in '`mntent.h`' and '`paths.h`', for '`mtab`'. The alternate macro names `FSTAB`, `MNTTAB` and `MOUNTED` are also defined, but these names are deprecated and kept only for backward compatibility. The names `_PATH_MNTTAB` and `_PATH_MOUNTED` should always be used.

11.3.1.1 The ‘fstab’ File

The internal representation for entries of the file is `struct fstab`, defined in ‘`fstab.h`’.

struct fstab Data Type

This structure is used with the `getfsent`, `getfsspec`, and `getfsfile` functions.

`char *fs_spec`
: This element describes the device from which the file system is mounted. Normally, this is the name of a special device, such as a hard-disk partition, but it could also be a more or less generic string. For *NFS* it would be a hostname and directory-name combination.

  Even though the element is not declared `const`, it shouldn't be modified. The missing `const` has historic reasons, since this function predates ISO C. The same is true for the other string elements of this structure.

`char *fs_file`
: This describes the mount point on the local system. Accessing any file in this file system has this string implicitly or explicitly as a prefix.

`char *fs_vfstype`
: This is the type of the file system. Depending on what the underlying kernel understands, it can be any string.

`char *fs_mntops`
: This is a string containing options passed to the kernel with the `mount` call. Again, this can be almost anything. There can be more than one option, separated from the others by a comma. Each option consists of a name and an optional value part, introduced by an ‘=’ character.

  If the value of this element must be processed it should ideally be done using the `getsubopt` function.[1]

`const char *fs_type`
: This name is poorly chosen. This element points to a string (possibly in the `fs_mntops` string) that describes the modes with which the file system is mounted. ‘`fstab`’ defines five macros to describe the possible values:

  `FSTAB_RW`
  : The file system gets mounted with read and write enabled.

[1] See Loosemore et al., “Parsing of Suboptions” (see chap. 1, n. 1).

`FSTAB_RQ`
: The file system gets mounted with read and write enabled. Write access is restricted by quotas.

`FSTAB_RO`
: The file system gets mounted read-only.

`FSTAB_SW`
: This is not a real file-system, it is a swap device.

`FSTAB_XX`
: This entry from the '`fstab`' file is totally ignored.

Testing for equality with these values must happen using `strcmp`, since these are all strings. Comparing the pointer will probably always fail.

`int fs_freq`
: This element describes the dump frequency in days.

`int fs_passno`
: This element describes the pass number on parallel dumps. It is closely related to the `dump` utility used on Unix systems.

To read the entire content of the of the '`fstab`' file, the GNU libc contains a set of three functions which are designed in the usual way.

`int` **setfsent** (`void`) Function

This function makes sure that the internal read pointer for the '`fstab`' file is at the beginning of the file. This is done by either opening the file or resetting the read pointer.

Since the file handle is internal to the libc, this function is not threadsafe.

This function returns a nonzero value if the operation was successful and the `getfs*` functions can be used to read the entries of the file.

`void` **endfsent** (`void`) Function

This function makes sure that all resources acquired by a prior call to `setfsent` (explicitly or implicitly by calling `getfsent`) are freed.

`struct fstab *` **getfsent** (`void`) Function

This function returns the next entry of the '`fstab`' file. If this is the first call to any of the functions handling '`fstab`' since program start or the last call of `endfsent`, the file will be opened.

The function returns a pointer to a variable of type `struct fstab`. This variable is shared by all threads, and therefore this function is not threadsafe. If an error occurred, `getfsent` returns a `NULL` pointer.

`struct fstab *` **getfsspec** (`const char *`*name*) Function

This function returns the next entry of the '`fstab`' file that has a string equal to *name* pointed to by the `fs_spec` element. Since there is normally exactly one entry for each special device, it makes no sense to call this function more than once for the same argument. If this is the first call to any of the functions handling '`fstab`' since program start or the last call of `endfsent`, the file will be opened.

The function returns a pointer to a variable of type `struct fstab`. This variable is shared by all threads, and therefore this function is not threadsafe. If an error occurred, `getfsent` returns a `NULL` pointer.

`struct fstab *` **getfsfile** (`const char *`*name*) Function

This function returns the next entry of the '`fstab`' file that has a string equal to *name* pointed to by the `fs_file` element. Since there is normally exactly one entry for each mount point, it makes no sense to call this function more than once for the same argument. If this is the first call to any of the functions handling '`fstab`' since program start or the last call of `endfsent`, the file will be opened.

The function returns a pointer to a variable of type `struct fstab`. This variable is shared by all threads, and therefore this function is not threadsafe. If an error occurred, `getfsent` returns a `NULL` pointer.

11.3.1.2 The '`mtab`' File

The following functions and data structure access the '`mtab`' file.

struct mntent Data Type

This structure is used with the `getmntent`, `getmntent_t`, `addmntent` and `hasmntopt` functions.

`char *mnt_fsname`
: This element contains a pointer to a string describing the name of the special device from which the file system is mounted. It corresponds to the `fs_spec` element in `struct fstab`.

`char *mnt_dir`
: This element points to a string describing the mount point of the file system. It corresponds to the `fs_file` element in `struct fstab`.

`char *mnt_type`
: `mnt_type` describes the file-system type, and is therefore equivalent to `fs_vfstype` in `struct fstab`. '`mntent.h`' defines a few symbolic names for some of the values this string can have. But since the kernel can support arbitrary file-systems, it does not make much sense to give them symbolic names. If you know the symbol name, you also know the file-system name. Nevertheless, here is the list of the symbols provided in '`mntent.h`':

MNTTYPE_IGNORE
: This symbol expands to "ignore". The value is sometime used in 'fstab' files to make sure entries are not used without removing them.

MNTTYPE_NFS
: This symbol expands to "nfs". Using this macro sometimes could make sense since it names the default NFS implementation, in case both version 2 and 3 are supported.

MNTTYPE_SWAP
: This symbol expands to "swap". It names the special 'fstab' entry that names one of the possibly multiple swap partitions.

char *mnt_opts
: The element contains a string describing the options used while mounting the file system. As for the equivalent element fs_mntops of struct fstab, it is best to use the function getsubopt to access the parts of this string.[2]

  The 'mntent.h' file defines a number of macros with string values that correspond to some of the options understood by the kernel. There might be many more possible options, so it doesn't make much sense to rely on these macros, but to be consistent here is the list:

  MNTOPT_DEFAULTS
  : This symbol expands to "defaults". This option should be used alone since it indicates all values for the customizable values are chosen to be the default.

  MNTOPT_RO
  : This symbol expands to "ro". See the FSTAB_RO value; it means the file system is mounted read-only.

  MNTOPT_RW
  : This symbol expands to "rw". See the FSTAB_RW value; it means the file system is mounted with read and write permissions.

  MNTOPT_SUID
  : This symbol expands to "suid". This means that the SUID bit (see Section 10.4 [How an Application Can Change Persona], page 254) is respected when a program from the file system is started.

[2] Ibid., "Parsing of Suboptions".

`MNTOPT_NOSUID`
: This symbol expands to `"nosuid"`. This is the opposite of `MNTOPT_SUID`. The SUID bit for all files from the file system is ignored.

`MNTOPT_NOAUTO`
: This symbol expands to `"noauto"`. At start-up time, the `mount` program will ignore this entry if it is started with the `-a` option to mount all file-systems mentioned in the '`fstab`' file.

As for the `FSTAB_*` entries introduced above, it is important to use `strcmp` to check for equality.

`mnt_freq`
: This element corresponds to `fs_freq` and also specifies the frequency in days in which dumps are made.

`mnt_passno`
: This element is equivalent to `fs_passno` with the same meaning, which is uninteresting for all programs beside `dump`.

For the '`mtab`' file, there is again a set of three functions to access all entries in a row. Unlike the functions to handle '`fstab`', these functions do not access a fixed file, and there is even a threadsafe variant of the get function. Besides this, the GNU libc contains functions to alter the file and test for specific options.

`FILE *` **setmntent** (`const char *`*file*`, const char *`*mode*) Function

The `setmntent` function prepares the file named *FILE*, which must be in the format of an '`fstab`' and '`mtab`' file, for the upcoming processing through the other functions of the family. The *mode* parameter can be chosen in the way the *opentype* parameter for `fopen` can be chosen.[3] If the file is opened for writing, the file is also allowed to be empty.

If the file was successfully opened, `setmntent` returns a file descriptor for future use. Otherwise, the return value is `NULL` and `errno` is set accordingly.

`int` **endmntent** (`FILE *`*stream*) Function

This function takes for the *stream* parameter a file handle that was previously returned from the `setmntent` call. `endmntent` closes the stream and frees all resources.

The return value is 1 unless an error occurred, in which case it is 0.

`struct mntent *` **getmntent** (`FILE *`*stream*) Function

The `getmntent` function takes as the parameter a file handle previously returned by successful call to `setmntent`. It returns a pointer to a static variable

[3] Ibid., "Opening Streams".

of type `struct mntent` that is filled with the information from the next entry from the file currently read.

The file format used prescribes the use of spaces or tab characters to separate the fields. This makes it harder to use names containing one of these characters (e.g., mount points using spaces). Therefore, these characters are encoded in the files and the `getmntent` function takes care of the decoding while reading the entries back in. '`\040`' is used to encode a space character, '`\012`' to encode a tab character and '`\\`' to encode a backslash.

If there was an error or the end of the file is reached, the return value is `NULL`.

This function is not threadsafe, since all calls to this function return a pointer to the same static variable. `getmntent_r` should be used in situations where multiple threads access the file.

`struct mntent *` **getmntent_r** (`FILE *`*stream*`, struct mentent *`*result*`, char *`*buffer*`, int` *bufsize*) Function

The `getmntent_r` function is the reentrant variant of `getmntent`. It also returns the next entry from the file and returns a pointer. The actual variable the values are stored in is not static, though. Instead, the function stores the values in the variable pointed to by the *result* parameter. Additional information (e.g., the strings pointed to by the elements of the result) are kept in the buffer of size *bufsize* pointed to by *buffer*.

Escaped characters (space, tab, backslash) are converted back in the same way as for `getmentent`.

The function returns a `NULL` pointer in error cases. Errors could be

- There was an error while reading the file.
- End of file was reached.
- *bufsize* is too small for reading a complete new entry.

`int` **addmntent** (`FILE *`*stream*`, const struct mntent *`*mnt*) Function

The `addmntent` function allows the addition of a new entry to the file previously opened with `setmntent`. The new entries are always appended—even if the position of the file descriptor is not at the end of the file, this function does not overwrite an existing entry following the current position.

The implication of this is that to remove an entry from a file, you have to create a new file while leaving out the entry to be removed, and after closing the file, remove the old one and rename the new file to the chosen name.

This function takes care of spaces and tab characters in the names to be written to the file. It converts them and the backslash character into the format describe in the `getmntent` description above.

This function returns 0 if the operation was successful. Otherwise, the return value is 1 and `errno` is set appropriately.

`char *` **hasmntopt** (`const struct mntent *`*mnt*`, const char *`*opt*) Function

This function can be used to check whether the string pointed to by the `mnt_opts` element of the variable pointed to by *mnt* contains the option *opt*. If this is true, a pointer to the beginning of the option in the `mnt_opts` element is returned. If no such option exists, the function returns `NULL`.

This function is useful to test whether a specific option is present, but when all options have to be processed, one is better off with using the `getsubopt` function to iterate over all options in the string.

11.3.1.3 Other (Non-libc) Sources of Mount Information

On a system with a Linux kernel and the `proc` file-system, you can get information on currently mounted file-systems from the file '`mounts`' in the `proc` file-system. Its format is similar to that of the '`mtab`' file, but represents what is truly mounted without relying on facilities outside the kernel to keep '`mtab`' up to date.

11.3.2 Mount, Unmount, Remount

This section describes the functions for mounting, unmounting and remounting file systems.

Only the superuser can mount, unmount or remount a file system.

These functions do not access the '`fstab`' and '`mtab`' files. You should maintain and use these separately (see Section 11.3.1 [Mount Information], page 289).

The symbols in this section are declared in '`sys/mount.h`'.

`int` **mount** (`const char *`*special_file*`, const char *`*dir*`, const char *`*fstype*`, unsigned long int` *options*`, const void *`*data*) Function

`mount` mounts or remounts a file system. The two operations are quite different and are merged rather unnaturally into this one function. The `MS_REMOUNT` option, explained below, determines whether `mount` mounts or remounts.

For a mount, the file system on the block device represented by the device special file named *special_file* gets mounted over the mount point *dir*. This means that the directory *dir* (along with any files in it) is no longer visible; in its place (and still with the name *dir*) is the root directory of the file system on the device.

As an exception, if the file-system type (see below) is one that is not based on a device (e.g. "proc"), `mount` instantiates a file system and mounts it over *dir* and ignores *special_file*.

For a remount, *dir* specifies the mount point where the file system to be remounted is (and remains) mounted, and *special_file* is ignored. Remounting a file system means changing the options that control operations on the file system while it is mounted. It does not mean unmounting and mounting again.

For a mount, you must identify the type of the file system as *fstype*. This type tells the kernel how to access the file system and can be thought of as the name of a file-system driver. The acceptable values are system dependent. On a system with a Linux kernel and the `proc` file-system, the list of possible values is in the file '`filesystems`' in the `proc` file-system (type *`cat /proc/filesystems`* to see the list). With a Linux kernel, the types of file systems that `mount` can mount and their type names depend on what file-system drivers are configured into the kernel or loaded as loadable kernel modules. An example of a common value for *fstype* is `ext2`.

For a remount, `mount` ignores *fstype*.

options specifies a variety of options that apply until the file system is unmounted or remounted. The precise meaning of an option depends on the file system and with some file-systems, an option may have no effect at all. Furthermore, for some file-systems, some of these options (but never `MS_RDONLY`) can be overridden for individual file accesses via `ioctl`.

options is a bit string with bit fields defined using the following mask and masked value macros:

`MS_MGC_MASK`
: This multibit field contains a magic number. If it does not have the value `MS_MGC_VAL`, `mount` assumes all the following bits are zero and the *data* argument is a null string, regardless of their actual values.

`MS_REMOUNT`
: This bit on means to remount the file system. Off means to mount it.

`MS_RDONLY`
: When this bit is on, it specifies that no writing to the file system will be allowed while it is mounted. This cannot be overridden by `ioctl`. This option is available on nearly all file systems.

`S_IMMUTABLE`
: When this bit is on, it specifies that no writing to the files in the file system will be allowed while it is mounted. This can be overridden for a particular file access by a properly privileged call to `ioctl`. This option is a relatively new invention and is not available on many file systems.

`S_APPEND`
: When this bit is on. it specifies that the only file-writing that will be allowed while the file system is mounted is appending. Some file systems allow this to be overridden for a particular process by a properly privileged call to `ioctl`. This is a relatively new invention and is not available on many file systems.

`MS_NOSUID`
: When this bit is on, it specifies that Setuid and Setgid permissions on files in the file system will be ignored while it is mounted.

`MS_NOEXEC`
: When this bit is on, it specifies that no files in the file system will be executed while the file system is mounted.

`MS_NODEV`
: When this bit is on, it specifies that no device special files in the file system will be accessible while the file system is mounted.

`MS_SYNCHRONOUS`
: When this bit is on, it specifies that all writes to the file system while it is mounted will be synchronous—data will be synced before each write completes rather than be held in the buffer cache.

`MS_MANDLOCK`
: When this bit is on, it specifies that mandatory locks on files will be permitted while the file system is mounted.

`MS_NOATIME`
: When this bit is on, it specifies that access times of files will not be updated when the files are accessed while the file system is mounted.

`MS_NODIRATIME`
: When this bit is on, it specifies that access times of directories will not be updated when the directories are accessed while the file system in mounted.

Any bits not covered by the above masks should be set off; otherwise, results are undefined.

The meaning of *data* depends on the file system type and is controlled entirely by the file-system driver in the kernel.

Here is an example:

```
#include <sys/mount.h>

mount("/dev/hdb", "/cdrom", MS_MGC_VAL | MS_RDONLY | MS_NOSUID, "");

mount("/dev/hda2", "/mnt", MS_MGC_VAL | MS_REMOUNT, "");
```

Appropriate arguments for `mount` are conventionally recorded in the '`fstab`' table (see Section 11.3.1 [Mount Information], page 289).

The return value is 0 if the mount or remount is successful. Otherwise, it is `-1` and `errno` is set appropriately. The values of `errno` are file-system dependent, but here is a general list:

`EPERM`

- The process is not superuser.

`ENODEV`

- The file-system type *fstype* is not known to the kernel.

`ENOTBLK`

- The file *dev* is not a block device special file.

`EBUSY`

- The device is already mounted.
- The mount point is busy—it is some process's working directory or has a file system mounted on it already.
- The request is to remount read-only, but there are files open for write.

`EINVAL`

- A remount was attempted, but there is no file system mounted over the specified mount point.
- The supposed file system has an invalid superblock.

`EACCES`

- The file system is inherently read-only (possibly due to a switch on the device) and the process attempted to mount it read/write (by setting the `MS_RDONLY` bit off).
- *special_file* or *dir* is not accessible due to file permissions.
- *special_file* is not accessible because it is in a file system that is mounted with the `MS_NODEV` option.

`EM_FILE`

- The table of dummy devices is full. `mount` needs to create a dummy device (aka "unnamed" device) if the file system being mounted is not one that uses a device.

`int` **umount2** (`const char *`*file*`,` `int` *flags*) Function

`umount2` unmounts a file system.

You can identify the file system to unmount either by the device special file that contains the file system or by the mount point. The effect is the same. Specify either as the string *file*.

flags contains the 1-bit field identified by the following mask macro:

`MNT_FORCE`

This bit on means to force the unmounting even if the file system is busy, by making it unbusy first. If the bit is off and the file system is busy, `umount2` fails with `errno` = `EBUSY`. Depending on the file system, this may override all, some, or no busy conditions.

All other bits in *flags* should be set to zero; otherwise, the result is undefined.

Here is an example:

```
#include <sys/mount.h>

umount2("/mnt", MNT_FORCE);

umount2("/dev/hdd1", 0);
```

After the file system is unmounted, the directory that was the mount point is visible, as are any files in it.

As part of unmounting, `umount2` syncs the file system.

If the unmounting is successful, the return value is 0. Otherwise, it is `-1` and `errno` is set accordingly:

`EPERM` The process is not superuser.

`EBUSY` The file system cannot be unmounted because it is busy—it contains a directory that is some process's working directory or a file that some process has open. With some file-systems in some cases, you can avoid this failure with the `MNT_FORCE` option.

`EINVAL` *file* validly refers to a file, but that file is neither a mount point nor a device special file of a currently mounted file-system.

This function is not available on all systems.

`int` **umount** (`const char *`*file*) Function

`umount` does the same thing as `umount2` with *flags* set to zeroes. It is more widely available than `umount2`, but since it lacks the ability to forcefully unmount a file system, it is deprecated when `umount2` is also available.

11.4 System Parameters

This section describes the `sysctl` function, which gets and sets a variety of system parameters.

The symbols used in this section are declared in the file '`sysctl.h`'.

`int` **sysctl** (`int *`*names*`, int` *nlen*`, void *`*oldval*`,` size_t *oldlenp*, void **newval*, size_t *newlen*) Function

`sysctl` gets or sets a specified system parameter. There are so many of these parameters that it is not practical to list them all here, but here are some examples:

- Network domain name
- Paging parameters

- Network Address Resolution Protocol time-out time
- Maximum number of files that may be open
- Root file-system device
- When the kernel was built

The set of available parameters depends on the kernel configuration and can change while the system is running, particularly when you load and unload loadable kernel modules.

The system parameters with which `syslog` is concerned are arranged in a hierarchical structure like a hierarchical file-system. To identify a particular parameter, you specify a path through the structure in a way analogous to specifying the pathname of a file. Each component of the path is specified by an integer and each of these integers has a macro defined for it by '`sysctl.h`'. *names* is the path, in the form of an array of integers. Each component of the path is one element of the array, in order. *nlen* is the number of components in the path.

For example, the first component of the path for all the paging parameters is the value `CTL_VM`. For the free-page thresholds, the second component of the path is `VM_FREEPG`. So to get the free-page threshold values, make *names* an array containing the two elements `CTL_VM` and `VM_FREEPG`, and make *nlen* = 2.

The format of the value of a parameter depends on the parameter. Sometimes it is an integer, sometimes it is an ASCII string; sometimes it is an elaborate structure. In the case of the free-page thresholds used in the example above, the parameter value is a structure containing several integers.

In any case, you identify a place to return the parameter's value with *oldval* and specify the amount of storage available at that location as **oldlenp*. **oldlenp* does double-duty because it is also the output location that contains the actual length of the returned value.

If you don't want the parameter value returned, specify a null pointer for *oldval*.

To set the parameter, specify the address and length of the new value as *newval* and *newlen*. If you don't want to set the parameter, specify a null pointer as *newval*.

If you get and set a parameter in the same `sysctl` call, the value returned is the value of the parameter before it was set.

Each system parameter has a set of permissions similar to the permissions for a file (including the permissions on directories in its path) that determine whether you may get or set it. For the purposes of these permissions, every parameter is considered to be owned by the superuser and Group 0 so processes with that effective uid or gid may have more access to system parameters. Unlike with files, the superuser does not invariably have full permission to all system parameters, because some of them are designed never to be changed.

`sysctl` returns a zero return value if it succeeds. Otherwise, it returns `-1` and sets `errno` appropriately. Besides the failures that apply to all system calls, the following are the `errno` codes for all possible failures:

`EPERM` The process is not permitted to access one of the components of the path of the system parameter or is not permitted to access the system parameter itself in the way (read or write) that it requested.

`ENOTDIR` There is no system parameter corresponding to *name*.

`EFAULT` *oldval* is not null, which means the process wanted to read the parameter, but **oldlenp* is 0, so there is no place to return it.

`EINVAL`

The process attempted to set a system parameter to a value that is not valid for that parameter.

Or, the space provided for the return of the system parameter is not the right size for that parameter.

`ENOMEM` This value may be returned instead of the more correct `EINVAL` in some cases where the space provided for the return of the system parameter is too small.

If you have a Linux kernel with the `proc` file-system, you can get and set most of the same parameters by reading and writing to files in the `sys` directory of the `proc` file-system. In the `sys` directory, the directory structure represents the hierarchical structure of the parameters, so you can display the free-page thresholds with:

```
cat /proc/sys/vm/freepages
```

Some more traditional and more widely available, though less general, GNU C Library functions for getting and setting some of the same system parameters are

- `getdomainname`, `setdomainname`
- `gethostname`, `sethostname` (see Section 11.1 [Host Identification], page 285)
- `uname` (see Section 11.2 [Platform-Type Identification], page 287)
- `bdflush`

12 System-Configuration Parameters

The functions and macros listed in this chapter give information about configuration parameters of the operating system—for example, capacity limits, presence of optional POSIX features and the default path for executable files (see Section 12.12 [String-Valued Parameters], page 324).

12.1 General Capacity-Limits

The POSIX.1 and POSIX.2 standards specify a number of parameters that describe capacity limitations of the system. These limits can be fixed constants for a given operating system, or they can vary from machine to machine. For example, some limit values may be configurable by the system administrator, either at run time or by rebuilding the kernel, and this should not require recompiling application programs.

Each of the following limit parameters has a macro that is defined in '`limits.h`' only if the system has a fixed, uniform limit for the parameter in question. If the system allows different file-systems or files to have different limits, then the macro is undefined; use `sysconf` to find out the limit that applies at a particular time on a particular machine (see Section 12.4 [Using `sysconf`], page 306).

Each of these parameters also has another macro, with a name starting with '`_POSIX`', which gives the lowest value that the limit is allowed to have on *any* POSIX system (see Section 12.5 [Minimum Values for General Capacity-Limits], page 317).

`int` **ARG_MAX** Macro

If defined, this is the unvarying maximum combined length of the *argv* and *environ* arguments that can be passed to the `exec` functions.

`int` **CHILD_MAX** Macro

If defined, this is the unvarying maximum number of processes that can exist with the same real user-ID at any one time. In BSD and GNU, this is controlled by the `RLIMIT_NPROC` resource limit (see Section 14.2 [Limiting Resource Usage], page 338).

`int` **OPEN_MAX** Macro

If defined, this is the unvarying maximum number of files that a single process can have open simultaneously. In BSD and GNU, this is controlled by the `RLIMIT_NOFILE` resource limit (see Section 14.2 [Limiting Resource Usage], page 338).

`int` **STREAM_MAX** Macro

If defined, this is the unvarying maximum number of streams that a single process can have open simultaneously.[1]

`int` **TZNAME_MAX** Macro

If defined, this is the unvarying maximum length of a time zone name.[2]

These limit macros are always defined in '`limits.h`'.

`int` **NGROUPS_MAX** Macro

This is the maximum number of supplementary group-IDs that one process can have.

The value of this macro is actually a lower bound for the maximum—that is, you can count on being able to have that many supplementary group-IDs, but a particular machine might let you have even more. You can use `sysconf` to see whether a particular machine will let you have more (see Section 12.4 [Using sysconf], page 306).

`int` **SSIZE_MAX** Macro

This is the largest value that can fit in an object of type `ssize_t`. Effectively, this is the limit on the number of bytes that can be read or written in a single operation.

This macro is defined in all POSIX systems because this limit is never configurable.

`int` **RE_DUP_MAX** Macro

This is the largest number of repetitions guaranteed to be allowed in the construct '`\{`*min*`,` *max*`\}`' in a regular expression.

The value of this macro is actually a lower bound for the maximum—that is, you can count on being able to have that many repetitions, but a particular machine might let you have even more. You can use `sysconf` to see whether a particular machine will let you have more (see Section 12.4 [Using sysconf], page 306). Even the value that `sysconf` tells you is just a lower bound—larger values might work.

This macro is defined in all POSIX.2 systems, because POSIX.2 says it should always be defined even if there is no specific imposed limit.

---

[1] See Loosemore et al., "Opening Streams" (see chap. 1, n.1).

[2] Ibid., "Functions and Variables for Time Zones".

12.2 Overall System Options

POSIX defines certain system-specific options that not all POSIX systems support. Since these options are provided in the kernel, not in the library, simply using the GNU C Library does not guarantee that any of these features will be supported; it will depend on the system you are using.

You can test for the availability of a given option using the macros in this section, together with the function `sysconf`. The macros are defined only if you include '`unistd.h`'.

For the following macros, if the macro is defined in '`unistd.h`', then the option is supported. Otherwise, the option may or may not be supported; use `sysconf` to find out (see Section 12.4 [Using sysconf], page 306).

`int` **_POSIX_JOB_CONTROL** Macro

If this symbol is defined, it indicates that the system supports job control. Otherwise, the implementation behaves as if all processes within a session belong to a single process-group. (See Chapter 8 [Job Control], page 221.)

`int` **_POSIX_SAVED_IDS** Macro

If this symbol is defined, it indicates that the system remembers the effective user- and group-IDs of a process before it executes an executable file with the set-user-ID or set-group-ID bits set, and that explicitly changing the effective user- or group-IDs back to these values is permitted. If this option is not defined, then if a nonprivileged process changes its effective user- or group-ID to the real user- or group-ID of the process, it can't change it back again (see Section 10.8 [Enabling and Disabling Setuid Access], page 260).

For the following macros, if the macro is defined in '`unistd.h`', then its value indicates whether the option is supported. A value of `-1` means no, and any other value means yes. If the macro is not defined, then the option may or may not be supported; use `sysconf` to find out (see Section 12.4 [Using sysconf], page 306).

`int` **_POSIX2_C_DEV** Macro

If this symbol is defined, it indicates that the system has the POSIX.2 C compiler command, `c89`. The GNU C Library always defines this as `1`, on the assumption that you would not have installed it if you didn't have a C compiler.

`int` **_POSIX2_FORT_DEV** Macro

If this symbol is defined, it indicates that the system has the POSIX.2 Fortran compiler command, `fort77`. The GNU C Library never defines this, because we don't know what the system has.

`int` **_POSIX2_FORT_RUN** Macro

If this symbol is defined, it indicates that the system has the POSIX.2 `asa` command to interpret Fortran carriage control. The GNU C library never defines this, because we don't know what the system has.

`int` **_POSIX2_LOCALEDEF** Macro

If this symbol is defined, it indicates that the system has the POSIX.2 `localedef` command. The GNU C Library never defines this, because we don't know what the system has.

`int` **_POSIX2_SW_DEV** Macro

If this symbol is defined, it indicates that the system has the POSIX.2 commands `ar`, `make` and `strip`. The GNU C Library always defines this as `1`, on the assumption that you had to have `ar` and `make` to install the library, and it's unlikely that `strip` would be absent when those are present.

12.3 Which Version of POSIX is Supported

`long int` **_POSIX_VERSION** Macro

This constant represents the version of the POSIX.1 standard to which the implementation conforms. For an implementation conforming to the 1995 POSIX.1 standard, the value is the integer `199506L`.

`_POSIX_VERSION` is always defined (in '`unistd.h`') in any POSIX system.

Don't try to test whether the system supports POSIX by including '`unistd.h`' and then checking whether `_POSIX_VERSION` is defined. On a non-POSIX system, this will probably fail because there is no '`unistd.h`'. We do not know of *any* way you can reliably test at compilation time whether your target system supports POSIX or whether '`unistd.h`' exists.

The GNU C Compiler predefines the symbol `__POSIX__` if the target system is a POSIX system. Provided you do not use any other compilers on POSIX systems, testing `defined (__POSIX__)` will reliably detect such systems.

`long int` **_POSIX2_C_VERSION** Macro

This constant represents the version of the POSIX.2 standard that the library and system kernel support. We don't know what value this will be for the first version of the POSIX.2 standard, because the value is based on the year and month in which the standard is officially adopted.

The value of this symbol says nothing about the utilities installed on the system.

You can use this macro to tell whether a POSIX.1 system library supports POSIX.2 as well. Any POSIX.1 system contains '`unistd.h`', so include that file and then test `defined (_POSIX2_C_VERSION)`.

12.4 Using `sysconf`

When your system has configurable system-limits, you can use the `sysconf` function to find out the value that applies to any particular machine. The function and the associated *parameter* constants are declared in the header file '`unistd.h`'.

12.4.1 Definition of `sysconf`

`long int` **sysconf** (`int` *parameter*) Function

This function is used to inquire about run-time system parameters. The *parameter* argument should be one of the '`_SC_`' symbols listed below.

The normal return value from `sysconf` is the value you requested. A value of `-1` is returned both if the implementation does not impose a limit and in case of an error.

The following `errno` error condition is defined for this function:

`EINVAL` The value of the *parameter* is invalid.

12.4.2 Constants for `sysconf` Parameters

Here are the symbolic constants for use as the *parameter* argument to `sysconf`. The values are all integer constants (more specifically, enumeration type values).

`_SC_ARG_MAX`
: Inquire about the parameter corresponding to `ARG_MAX`.

`_SC_CHILD_MAX`
: Inquire about the parameter corresponding to `CHILD_MAX`.

`_SC_OPEN_MAX`
: Inquire about the parameter corresponding to `OPEN_MAX`.

`_SC_STREAM_MAX`
: Inquire about the parameter corresponding to `STREAM_MAX`.

`_SC_TZNAME_MAX`
: Inquire about the parameter corresponding to `TZNAME_MAX`.

`_SC_NGROUPS_MAX`
: Inquire about the parameter corresponding to `NGROUPS_MAX`.

`_SC_JOB_CONTROL`
: Inquire about the parameter corresponding to `_POSIX_JOB_CONTROL`.

`_SC_SAVED_IDS`
: Inquire about the parameter corresponding to `_POSIX_SAVED_IDS`.

`_SC_VERSION`
: Inquire about the parameter corresponding to `_POSIX_VERSION`.

`_SC_CLK_TCK`
: Inquire about the parameter corresponding to `CLOCKS_PER_SEC`.[3]

[3] Ibid., "CPU Time Inquiry".

`_SC_CHARCLASS_NAME_MAX`
: Inquire about the parameter corresponding to maximum length allowed for a character-class name in an extended locale specification. These extensions are not yet standardized, so this option is not standardized either.

`_SC_REALTIME_SIGNALS`
: Inquire about the parameter corresponding to `_POSIX_REALTIME_SIGNALS`.

`_SC_PRIORITY_SCHEDULING`
: Inquire about the parameter corresponding to `_POSIX_PRIORITY_SCHEDULING`.

`_SC_TIMERS`
: Inquire about the parameter corresponding to `_POSIX_TIMERS`.

`_SC_ASYNCHRONOUS_IO`
: Inquire about the parameter corresponding to `_POSIX_ASYNCHRONOUS_IO`.

`_SC_PRIORITIZED_IO`
: Inquire about the parameter corresponding to `_POSIX_PRIORITIZED_IO`.

`_SC_SYNCHRONIZED_IO`
: Inquire about the parameter corresponding to `_POSIX_SYNCHRONIZED_IO`.

`_SC_FSYNC`
: Inquire about the parameter corresponding to `_POSIX_FSYNC`.

`_SC_MAPPED_FILES`
: Inquire about the parameter corresponding to `_POSIX_MAPPED_FILES`.

`_SC_MEMLOCK`
: Inquire about the parameter corresponding to `_POSIX_MEMLOCK`.

`_SC_MEMLOCK_RANGE`
: Inquire about the parameter corresponding to `_POSIX_MEMLOCK_RANGE`.

`_SC_MEMORY_PROTECTION`
: Inquire about the parameter corresponding to `_POSIX_MEMORY_PROTECTION`.

`_SC_MESSAGE_PASSING`
: Inquire about the parameter corresponding to `_POSIX_MESSAGE_PASSING`.

_SC_SEMAPHORES
: Inquire about the parameter corresponding to _POSIX_SEMAPHORES.

_SC_SHARED_MEMORY_OBJECTS
: Inquire about the parameter corresponding to _POSIX_SHARED_MEMORY_OBJECTS.

_SC_AIO_LISTIO_MAX
: Inquire about the parameter corresponding to _POSIX_AIO_LISTIO_MAX.

_SC_AIO_MAX
: Inquire about the parameter corresponding to _POSIX_AIO_MAX.

_SC_AIO_PRIO_DELTA_MAX
: Inquire about the value by which a process can decrease its asynchronous I/O priority level from its own scheduling priority. This corresponds to the run-time invariant value AIO_PRIO_DELTA_MAX.

_SC_DELAYTIMER_MAX
: Inquire about the parameter corresponding to _POSIX_DELAYTIMER_MAX.

_SC_MQ_OPEN_MAX
: Inquire about the parameter corresponding to _POSIX_MQ_OPEN_MAX.

_SC_MQ_PRIO_MAX
: Inquire about the parameter corresponding to _POSIX_MQ_PRIO_MAX.

_SC_RTSIG_MAX
: Inquire about the parameter corresponding to _POSIX_RTSIG_MAX.

_SC_SEM_NSEMS_MAX
: Inquire about the parameter corresponding to _POSIX_SEM_NSEMS_MAX.

_SC_SEM_VALUE_MAX
: Inquire about the parameter corresponding to _POSIX_SEM_VALUE_MAX.

_SC_SIGQUEUE_MAX
: Inquire about the parameter corresponding to _POSIX_SIGQUEUE_MAX.

_SC_TIMER_MAX
: Inquire about the parameter corresponding to _POSIX_TIMER_MAX.

_SC_PII Inquire about the parameter corresponding to _POSIX_PII.

_SC_PII_XTI
Inquire about the parameter corresponding to _POSIX_PII_XTI.

_SC_PII_SOCKET
Inquire about the parameter corresponding to _POSIX_PII_SOCKET.

_SC_PII_INTERNET
Inquire about the parameter corresponding to _POSIX_PII_INTERNET.

_SC_PII_OSI
Inquire about the parameter corresponding to _POSIX_PII_OSI.

_SC_SELECT
Inquire about the parameter corresponding to _POSIX_SELECT.

_SC_UIO_MAXIOV
Inquire about the parameter corresponding to _POSIX_UIO_MAXIOV.

_SC_PII_INTERNET_STREAM
Inquire about the parameter corresponding to _POSIX_PII_INTERNET_STREAM.

_SC_PII_INTERNET_DGRAM
Inquire about the parameter corresponding to _POSIX_PII_INTERNET_DGRAM.

_SC_PII_OSI_COTS
Inquire about the parameter corresponding to _POSIX_PII_OSI_COTS.

_SC_PII_OSI_CLTS
Inquire about the parameter corresponding to _POSIX_PII_OSI_CLTS.

_SC_PII_OSI_M
Inquire about the parameter corresponding to _POSIX_PII_OSI_M.

_SC_T_IOV_MAX
Inquire about the value of the value associated with the T_IOV_MAX variable.

_SC_THREADS
Inquire about the parameter corresponding to _POSIX_THREADS.

`_SC_THREAD_SAFE_FUNCTIONS`
: Inquire about the parameter corresponding to `_POSIX_THREAD_SAFE_FUNCTIONS`.

`_SC_GETGR_R_SIZE_MAX`
: Inquire about the parameter corresponding to `_POSIX_GETGR_R_SIZE_MAX`.

`_SC_GETPW_R_SIZE_MAX`
: Inquire about the parameter corresponding to `_POSIX_GETPW_R_SIZE_MAX`.

`_SC_LOGIN_NAME_MAX`
: Inquire about the parameter corresponding to `_POSIX_LOGIN_NAME_MAX`.

`_SC_TTY_NAME_MAX`
: Inquire about the parameter corresponding to `_POSIX_TTY_NAME_MAX`.

`_SC_THREAD_DESTRUCTOR_ITERATIONS`
: Inquire about the parameter corresponding to `_POSIX_THREAD_DESTRUCTOR_ITERATIONS`.

`_SC_THREAD_KEYS_MAX`
: Inquire about the parameter corresponding to `_POSIX_THREAD_KEYS_MAX`.

`_SC_THREAD_STACK_MIN`
: Inquire about the parameter corresponding to `_POSIX_THREAD_STACK_MIN`.

`_SC_THREAD_THREADS_MAX`
: Inquire about the parameter corresponding to `_POSIX_THREAD_THREADS_MAX`.

`_SC_THREAD_ATTR_STACKADDR`
: Inquire about the parameter corresponding to a `_POSIX_THREAD_ATTR_STACKADDR`.

`_SC_THREAD_ATTR_STACKSIZE`
: Inquire about the parameter corresponding to `_POSIX_THREAD_ATTR_STACKSIZE`.

`_SC_THREAD_PRIORITY_SCHEDULING`
: Inquire about the parameter corresponding to `_POSIX_THREAD_PRIORITY_SCHEDULING`.

`_SC_THREAD_PRIO_INHERIT`
: Inquire about the parameter corresponding to `_POSIX_THREAD_PRIO_INHERIT`.

`_SC_THREAD_PRIO_PROTECT`
: Inquire about the parameter corresponding to `_POSIX_THREAD_PRIO_PROTECT`.

`_SC_THREAD_PROCESS_SHARED`
: Inquire about the parameter corresponding to `_POSIX_THREAD_PROCESS_SHARED`.

`_SC_2_C_DEV`
: Inquire about whether the system has the POSIX.2 C compiler command, `c89`.

`_SC_2_FORT_DEV`
: Inquire about whether the system has the POSIX.2 Fortran compiler command, `fort77`.

`_SC_2_FORT_RUN`
: Inquire about whether the system has the POSIX.2 `asa` command to interpret Fortran carriage-control.

`_SC_2_LOCALEDEF`
: Inquire about whether the system has the POSIX.2 `localedef` command.

`_SC_2_SW_DEV`
: Inquire about whether the system has the POSIX.2 commands `ar`, `make` and `strip`.

`_SC_BC_BASE_MAX`
: Inquire about the maximum value of `obase` in the `bc` utility.

`_SC_BC_DIM_MAX`
: Inquire about the maximum size of an array in the `bc` utility.

`_SC_BC_SCALE_MAX`
: Inquire about the maximum value of `scale` in the `bc` utility.

`_SC_BC_STRING_MAX`
: Inquire about the maximum size of a string constant in the `bc` utility.

`_SC_COLL_WEIGHTS_MAX`
: Inquire about the maximum number of weights that can necessarily be used in defining the collating sequence for a locale.

`_SC_EXPR_NEST_MAX`
: Inquire about the maximum number of expressions nested within parentheses when using the `expr` utility.

`_SC_LINE_MAX`
: Inquire about the maximum size of a text line that the POSIX.2 text utilities can handle.

`_SC_EQUIV_CLASS_MAX`
: Inquire about the maximum number of weights that can be assigned to an entry of the `LC_COLLATE` category '`order`' keyword in a locale definition. The GNU C Library does not presently support locale definitions.

`_SC_VERSION`
: Inquire about the version number of POSIX.1 that the library and kernel support.

`_SC_2_VERSION`
: Inquire about the version number of POSIX.2 that the system utilities support.

`_SC_PAGESIZE`
: Inquire about the virtual-memory page size of the machine. `getpagesize` returns the same value (see Section 14.4.2 [How to Get Information About the Memory Subsystem?], page 355).

`_SC_NPROCESSORS_CONF`
: Inquire about the number of configured processors.

`_SC_NPROCESSORS_ONLN`
: Inquire about the number of processors online.

`_SC_PHYS_PAGES`
: Inquire about the number of physical pages in the system.

`_SC_AVPHYS_PAGES`
: Inquire about the number of available physical pages in the system.

`_SC_ATEXIT_MAX`
: Inquire about the number of functions that can be registered as termination functions for `atexit`.[4]

`_SC_XOPEN_VERSION`
: Inquire about the parameter corresponding to `_XOPEN_VERSION`.

`_SC_XOPEN_XCU_VERSION`
: Inquire about the parameter corresponding to `_XOPEN_XCU_VERSION`.

`_SC_XOPEN_UNIX`
: Inquire about the parameter corresponding to `_XOPEN_UNIX`.

`_SC_XOPEN_REALTIME`
: Inquire about the parameter corresponding to `_XOPEN_REALTIME`.

[4] Ibid., "Clean-Ups on Exit".

`_SC_XOPEN_REALTIME_THREADS`
: Inquire about the parameter corresponding to `_XOPEN_REALTIME_THREADS`.

`_SC_XOPEN_LEGACY`
: Inquire about the parameter corresponding to `_XOPEN_LEGACY`.

`_SC_XOPEN_CRYPT`
: Inquire about the parameter corresponding to `_XOPEN_CRYPT`.

`_SC_XOPEN_ENH_I18N`
: Inquire about the parameter corresponding to `_XOPEN_ENH_I18N`.

`_SC_XOPEN_SHM`
: Inquire about the parameter corresponding to `_XOPEN_SHM`.

`_SC_XOPEN_XPG2`
: Inquire about the parameter corresponding to `_XOPEN_XPG2`.

`_SC_XOPEN_XPG3`
: Inquire about the parameter corresponding to `_XOPEN_XPG3`.

`_SC_XOPEN_XPG4`
: Inquire about the parameter corresponding to `_XOPEN_XPG4`.

`_SC_CHAR_BIT`
: Inquire about the number of bits in a variable of type `char`.

`_SC_CHAR_MAX`
: Inquire about the maximum value that can be stored in a variable of type `char`.

`_SC_CHAR_MIN`
: Inquire about the minimum value that can be stored in a variable of type `char`.

`_SC_INT_MAX`
: Inquire about the maximum value that can be stored in a variable of type `int`.

`_SC_INT_MIN`
: Inquire about the minimum value that can be stored in a variable of type `int`.

`_SC_LONG_BIT`
: Inquire about the number of bits in a variable of type `long int`.

`_SC_WORD_BIT`
: Inquire about the number of bits in a variable of a register word.

`_SC_MB_LEN_MAX`
: Inquire about the maximum length of a multibyte representation of a wide-character value.

`_SC_NZERO`
: Inquire about the value used to internally represent the zero priority level for the process execution.

`SC_SSIZE_MAX`
: Inquire about the maximum value that can be stored in a variable of type `ssize_t`.

`_SC_SCHAR_MAX`
: Inquire about the maximum value that can be stored in a variable of type `signed char`.

`_SC_SCHAR_MIN`
: Inquire about the minimum value that can be stored in a variable of type `signed char`.

`_SC_SHRT_MAX`
: Inquire about the maximum value that can be stored in a variable of type `short int`.

`_SC_SHRT_MIN`
: Inquire about the minimum value that can be stored in a variable of type `short int`.

`_SC_UCHAR_MAX`
: Inquire about the maximum value that can be stored in a variable of type `unsigned char`.

`_SC_UINT_MAX`
: Inquire about the maximum value that can be stored in a variable of type `unsigned int`.

`_SC_ULONG_MAX`
: Inquire about the maximum value that can be stored in a variable of type `unsigned long int`.

`_SC_USHRT_MAX`
: Inquire about the maximum value that can be stored in a variable of type `unsigned short int`.

`_SC_NL_ARGMAX`
: Inquire about the parameter corresponding to `NL_ARGMAX`.

`_SC_NL_LANGMAX`
: Inquire about the parameter corresponding to `NL_LANGMAX`.

`_SC_NL_MSGMAX`
: Inquire about the parameter corresponding to `NL_MSGMAX`.

`_SC_NL_NMAX`
: Inquire about the parameter corresponding to `NL_NMAX`.

`_SC_NL_SETMAX`
: Inquire about the parameter corresponding to `NL_SETMAX`.

`_SC_NL_TEXTMAX`
: Inquire about the parameter corresponding to `NL_TEXTMAX`.

12.4.3 Examples of `sysconf`

We recommend that you first test for a macro definition for the parameter you are interested in, and call `sysconf` only if the macro is not defined. For example, here is how to test whether job control is supported:

```
int
have_job_control (void)
{
#ifdef _POSIX_JOB_CONTROL
  return 1;
#else
  int value = sysconf (_SC_JOB_CONTROL);
  if (value < 0)
    /* If the system is that badly wedged,
       there's no use trying to go on.  */
    fatal (strerror (errno));
  return value;
#endif
}
```

Here is how to get the value of a numeric limit:

```
int
get_child_max ()
{
#ifdef CHILD_MAX
  return CHILD_MAX;
#else
  int value = sysconf (_SC_CHILD_MAX);
  if (value < 0)
    fatal (strerror (errno));
  return value;
#endif
}
```

12.5 Minimum Values for General Capacity-Limits

Here are the names for the POSIX minimum upper bounds for the system limit parameters. The significance of these values is that you can safely push to these limits without checking whether the particular system you are using can go that far.

`_POSIX_AIO_LISTIO_MAX`
: The value of this macro is the most restrictive limit permitted by POSIX for the maximum number of I/O operations that can be specified in a list I/O call. The value of this constant is `2`; thus you can add up to two new entries to the list of outstanding operations.

`_POSIX_AIO_MAX`
: The value of this macro is the most restrictive limit permitted by POSIX for the maximum number of outstanding asynchronous I/O operations. The value of this constant is `1`. So you cannot expect that you can issue more than one operation and immediately continue with the normal work, receiving the notifications asynchronously.

`_POSIX_ARG_MAX`
: The value of this macro is the most restrictive limit permitted by POSIX for the maximum combined length of the *argv* and *environ* arguments that can be passed to the `exec` functions. Its value is `4096`.

`_POSIX_CHILD_MAX`
: The value of this macro is the most restrictive limit permitted by POSIX for the maximum number of simultaneous processes per real user-ID. Its value is `6`.

`_POSIX_NGROUPS_MAX`
: The value of this macro is the most restrictive limit permitted by POSIX for the maximum number of supplementary group-IDs per process. Its value is `0`.

`_POSIX_OPEN_MAX`
: The value of this macro is the most restrictive limit permitted by POSIX for the maximum number of files that a single process can have open simultaneously. Its value is `16`.

`_POSIX_SSIZE_MAX`
: The value of this macro is the most restrictive limit permitted by POSIX for the maximum value that can be stored in an object of type `ssize_t`. Its value is `32767`.

`_POSIX_STREAM_MAX`
: The value of this macro is the most restrictive limit permitted by POSIX for the maximum number of streams that a single process can have open simultaneously. Its value is `8`.

`_POSIX_TZNAME_MAX`
: The value of this macro is the most restrictive limit permitted by POSIX for the maximum length of a time zone name. Its value is `3`.

`_POSIX2_RE_DUP_MAX`
: The value of this macro is the most restrictive limit permitted by POSIX for the numbers used in the '`\{`*min*`,`*max*`\}`' construct in a regular expression. Its value is `255`.

12.6 Limits on File-System Capacity

The POSIX.1 standard specifies a number of parameters that describe the limitations of the file system. It's possible for the system to have a fixed, uniform limit for a parameter, but this isn't the usual case. On most systems, it's possible for different file systems (and, for some parameters, even different files) to have different maximum limits. For example, this is very likely if you use NFS to mount some of the file systems from other machines.

Each of the following macros is defined in '`limits.h`' only if the system has a fixed, uniform limit for the parameter in question. If the system allows different file systems or files to have different limits, then the macro is undefined; use `pathconf` or `fpathconf` to find out the limit that applies to a particular file (see Section 12.9 [Using `pathconf`], page 321).

Each parameter also has another macro, with a name starting with '`_POSIX`', that gives the lowest value that the limit is allowed to have on *any* POSIX system (see Section 12.8 [Minimum Values for File-System Limits], page 320).

`int` **LINK_MAX** Macro
: This is the uniform system-limit, if any, for the number of names for a given file (see Section 3.4 [Hard Links], page 85).

`int` **MAX_CANON** Macro
: This is the uniform system-limit, if any, for the amount of text in a line of input when input editing is enabled (see Section 6.3 [Two Styles of Input: Canonical or Not], page 180).

`int` **MAX_INPUT** Macro
: This is the uniform system-limit, if any, for the total number of characters typed ahead as input (see Section 6.2 [I/O Queues], page 180).

`int` **NAME_MAX** Macro
: This is the uniform system-limit, if any, for the length of a file-name component.

`int` **PATH_MAX** Macro

This is the uniform system-limit, if any, for the length of an entire file-name (that is, the argument given to system calls such as `open`).

`int` **PIPE_BUF** Macro

This is the uniform system-limit, if any, for the number of bytes that can be written atomically to a pipe. If multiple processes are writing to the same pipe simultaneously, output from different processes might be interleaved in chunks of this size (see Chapter 4 [Pipes and FIFOs], page 119).

These are alternative macro names for some of the same information.

`int` **MAXNAMLEN** Macro

This is the BSD name for `NAME_MAX`. It is defined in '`dirent.h`'.

`int` **FILENAME_MAX** Macro

The value of this macro is an integer constant expression that represents the maximum length of a file-name string. It is defined in '`stdio.h`'.

Unlike `PATH_MAX`, this macro is defined even if there is no actual limit imposed. In such a case, its value is typically a very large number. *This is always the case on the* GNU *system.*

Don't use `FILENAME_MAX` as the size of an array in which to store a file name, because you can't possibly make an array that big. Use dynamic allocation instead.[5]

12.7 Optional Features in File Support

POSIX defines certain system-specific options in the system calls for operating on files. Some systems support these options and others do not. Since these options are provided in the kernel, not in the library, simply using the GNU C Library does not guarantee that any of these features is supported; it depends on the system you are using. They can also vary between file systems on a single machine.

This section describes the macros you can test to determine whether a particular option is supported on your machine. If a given macro is defined in '`unistd.h`', then its value says whether the corresponding feature is supported. A value of `-1` indicates no; any other value indicates yes. If the macro is undefined, it means particular files may or may not support the feature.

Since all the machines that support the GNU C Library also support NFS, one can never make a general statement about whether all file systems support the `_POSIX_CHOWN_RESTRICTED` and `_POSIX_NO_TRUNC` features. So these names are never defined as macros in the GNU C Library.

[5] Ibid., "Allocating Storage for Program Data".

`int` **_POSIX_CHOWN_RESTRICTED** Macro

If this option is in effect, the `chown` function is restricted so that the only change permitted to nonprivileged processes is to change the group owner of a file to be either the effective group-ID of the process, or one of its supplementary group-IDs (see Section 3.9.4 [File Owner], page 101).

`int` **_POSIX_NO_TRUNC** Macro

If this option is in effect, file-name components longer than `NAME_MAX` generate an `ENAMETOOLONG` error. Otherwise, file-name components that are too long are silently truncated.

`unsigned char` **_POSIX_VDISABLE** Macro

This option is only meaningful for files that are terminal devices. If it is enabled, then handling for special control-characters can be disabled individually (see Section 6.4.9 [Special Characters], page 194).

If one of these macros is undefined, that means that the option might be in effect for some files and not for others. To inquire about a particular file, call `pathconf` or `fpathconf` (see Section 12.9 [Using `pathconf`], page 321).

12.8 Minimum Values for File-System Limits

Here are the names for the POSIX minimum upper bounds for some of the above parameters. The significance of these values is that you can safely push to these limits without checking whether the particular system you are using can go that far. In most cases, GNU systems do not have these strict limitations. The actual limit should be requested if necessary.

`_POSIX_LINK_MAX`

This is the most restrictive limit permitted by POSIX for the maximum value of a file's link count. The value of this constant is `8`; thus, you can always make up to eight names for a file without running into a system limit.

`_POSIX_MAX_CANON`

This is the most restrictive limit permitted by POSIX for the maximum number of bytes in a canonical-input line from a terminal device. The value of this constant is `255`.

`_POSIX_MAX_INPUT`

This is the most restrictive limit permitted by POSIX for the maximum number of bytes in a terminal-device input queue (or type-ahead buffer) (see Section 6.4.4 [Input Modes], page 185). The value of this constant is `255`.

`_POSIX_NAME_MAX`

This is the most restrictive limit permitted by POSIX for the maximum number of bytes in a file-name component. The value of this constant is `14`.

`_POSIX_PATH_MAX`
: This is the most restrictive limit permitted by POSIX for the maximum number of bytes in a file name. The value of this constant is `256`.

`_POSIX_PIPE_BUF`
: This is the most restrictive limit permitted by POSIX for the maximum number of bytes that can be written atomically to a pipe. The value of this constant is `512`.

`SYMLINK_MAX`
: This is the maximum number of bytes in a symbolic link.

`POSIX_REC_INCR_XFER_SIZE`
: This is the recommended increment for file-transfer sizes between the `POSIX_REC_MIN_XFER_SIZE` and `POSIX_REC_MAX_XFER_SIZE` values.

`POSIX_REC_MAX_XFER_SIZE`
: This is the maximum recommended file-transfer size.

`POSIX_REC_MIN_XFER_SIZE`
: This is the minimum recommended file-transfer size.

`POSIX_REC_XFER_ALIGN`
: This is recommended file-transfer buffer alignment.

12.9 Using `pathconf`

When your machine allows different files to have different values for a file-system parameter, you can use the functions in this section to find out the value that applies to any particular file.

These functions and the associated constants for the *parameter* argument are declared in the header file '`unistd.h`'.

`long int` **pathconf** (`const char *`*filename*`, int` *parameter*) Function

This function is used to inquire about the limits that apply to the file named *filename*.

The *parameter* argument should be one of the '`_PC_`' constants listed below.

The normal return value from `pathconf` is the value you requested. A value of `-1` is returned both if the implementation does not impose a limit and in case of an error. In the former case, `errno` is not set, while in the latter case, `errno` is set to indicate the cause of the problem. So the only way to use this function robustly is to store `0` into `errno` just before calling it.

Besides the usual file-name errors, the following error condition is defined for this function:[6]

`EINVAL` The value of *parameter* is invalid, or the implementation doesn't support the *parameter* for the specific file.

[6] Ibid., "File-Name Errors".

`long int` **fpathconf** (`int` *filedes*, `int` *parameter*) Function

This is just like `pathconf`, except that an open file-descriptor is used to specify the file for which information is requested, instead of a file name.

The following `errno` error conditions are defined for this function:

`EBADF` The *filedes* argument is not a valid file-descriptor.

`EINVAL` The value of *parameter* is invalid, or the implementation doesn't support the *parameter* for the specific file.

Here are the symbolic constants that you can use as the *parameter* argument to `pathconf` and `fpathconf`. The values are all integer constants.

`_PC_LINK_MAX`
: Inquire about the value of `LINK_MAX`.

`_PC_MAX_CANON`
: Inquire about the value of `MAX_CANON`.

`_PC_MAX_INPUT`
: Inquire about the value of `MAX_INPUT`.

`_PC_NAME_MAX`
: Inquire about the value of `NAME_MAX`.

`_PC_PATH_MAX`
: Inquire about the value of `PATH_MAX`.

`_PC_PIPE_BUF`
: Inquire about the value of `PIPE_BUF`.

`_PC_CHOWN_RESTRICTED`
: Inquire about the value of `_POSIX_CHOWN_RESTRICTED`.

`_PC_NO_TRUNC`
: Inquire about the value of `_POSIX_NO_TRUNC`.

`_PC_VDISABLE`
: Inquire about the value of `_POSIX_VDISABLE`.

`_PC_SYNC_IO`
: Inquire about the value of `_POSIX_SYNC_IO`.

`_PC_ASYNC_IO`
: Inquire about the value of `_POSIX_ASYNC_IO`.

`_PC_PRIO_IO`
: Inquire about the value of `_POSIX_PRIO_IO`.

`_PC_SOCK_MAXBUF`
: Inquire about the value of `_POSIX_PIPE_BUF`.

`_PC_FILESIZEBITS`
: Inquire about the availability of large files on the file system.

`_PC_REC_INCR_XFER_SIZE`
: Inquire about the value of `POSIX_REC_INCR_XFER_SIZE`.

`_PC_REC_MAX_XFER_SIZE`
: Inquire about the value of `POSIX_REC_MAX_XFER_SIZE`.

`_PC_REC_MIN_XFER_SIZE`
: Inquire about the value of `POSIX_REC_MIN_XFER_SIZE`.

`_PC_REC_XFER_ALIGN`
: Inquire about the value of `POSIX_REC_XFER_ALIGN`.

12.10 Utility Program Capacity-Limits

The POSIX.2 standard specifies certain system-limits that you can access through `sysconf` that apply to utility behavior rather than the behavior of the library or the operating system.

The GNU C Library defines macros for these limits, and `sysconf` returns values for them if you ask; but these values convey no meaningful information. They are simply the smallest values that POSIX.2 permits.

`int` **BC_BASE_MAX** Macro
This is the largest value of `obase` that the `bc` utility is guaranteed to support.

`int` **BC_DIM_MAX** Macro
This is the largest number of elements in one array that the `bc` utility is guaranteed to support.

`int` **BC_SCALE_MAX** Macro
This is the largest value of `scale` that the `bc` utility is guaranteed to support.

`int` **BC_STRING_MAX** Macro
This is the largest number of characters in one string constant that the `bc` utility is guaranteed to support.

`int` **COLL_WEIGHTS_MAX** Macro
This is the largest number of weights that can for certain be used in defining the collating sequence for a locale.

`int` **EXPR_NEST_MAX** Macro
This is the maximum number of expressions that can be nested within parentheses by the `expr` utility.

`int` **LINE_MAX** Macro
This is the largest text line that the text-oriented POSIX.2 utilities can support. If you are using the GNU versions of these utilities, then there is no actual limit except that imposed by the available virtual memory, but there is no way that the library can tell you this.

`int` **EQUIV_CLASS_MAX** Macro
This is the maximum number of weights that can be assigned to an entry of the `LC_COLLATE` category '`order`' keyword in a locale definition. The GNU C Library does not presently support locale definitions.

12.11 Minimum Values for Utility Limits

`_POSIX2_BC_BASE_MAX`
This is the most restrictive limit permitted by POSIX.2 for the maximum value of `obase` in the `bc` utility. Its value is `99`.

`_POSIX2_BC_DIM_MAX`
This is the most restrictive limit permitted by POSIX.2 for the maximum size of an array in the `bc` utility. Its value is `2048`.

`_POSIX2_BC_SCALE_MAX`
This is the most restrictive limit permitted by POSIX.2 for the maximum value of `scale` in the `bc` utility. Its value is `99`.

`_POSIX2_BC_STRING_MAX`
This is the most restrictive limit permitted by POSIX.2 for the maximum size of a string constant in the `bc` utility. Its value is `1000`.

`_POSIX2_COLL_WEIGHTS_MAX`
This is the most restrictive limit permitted by POSIX.2 for the maximum number of weights that can necessarily be used in defining the collating sequence for a locale. Its value is `2`.

`_POSIX2_EXPR_NEST_MAX`
This is the most restrictive limit permitted by POSIX.2 for the maximum number of expressions nested within parentheses when using the `expr` utility. Its value is `32`.

`_POSIX2_LINE_MAX`
This is the most restrictive limit permitted by POSIX.2 for the maximum size of a text line that the text utilities can handle. Its value is `2048`.

`_POSIX2_EQUIV_CLASS_MAX`
This is the most restrictive limit permitted by POSIX.2 for the maximum number of weights that can be assigned to an entry of the `LC_COLLATE` category '`order`' keyword in a locale definition. Its value is `2`. The GNU C Library does not presently support locale definitions.

12.12 String-Valued Parameters

POSIX.2 defines a way to get string-valued parameters from the operating system with the function `confstr`:

`size_t` **confstr** (`int` *parameter*, `char *`*buf*, `size_t` *len*) Function

This function reads the value of a string-valued system parameter, storing the string into *len* bytes of memory space starting at *buf*. The *parameter* argument should be one of the '`_CS_`' symbols listed below.

The normal return value from `confstr` is the length of the string value that you asked for. If you supply a null pointer for *buf*, then `confstr` does not try to store the string; it just returns its length. A value of `0` indicates an error.

If the string you asked for is too long for the buffer (that is, longer than *len* `- 1`), then `confstr` stores just that much (leaving room for the terminating null character). You can tell that this has happened because `confstr` returns a value greater than or equal to *len*.

The following `errno` error condition is defined for this function:

`EINVAL` The value of the *parameter* is invalid.

Currently there is just one parameter you can read with `confstr`:

`_CS_PATH`
: This parameter's value is the recommended default path for searching for executable files. This is the path that a user has by default just after logging in.

`_CS_LFS_CFLAGS`
: The returned string specifies which additional flags must be given to the C compiler if a source is compiled using the `_LARGEFILE_SOURCE` feature-select macro (see Section 1.3.4 [Feature-Test Macros], page 8).

`_CS_LFS_LDFLAGS`
: The returned string specifies which additional flags must be given to the linker if a source is compiled using the `_LARGEFILE_SOURCE` feature-select macro (see Section 1.3.4 [Feature-Test Macros], page 8).

`_CS_LFS_LIBS`
: The returned string specifies which additional libraries must be linked to the application if a source is compiled using the `_LARGEFILE_SOURCE` feature-select macro (see Section 1.3.4 [Feature-Test Macros], page 8).

`_CS_LFS_LINTFLAGS`
: The returned string specifies which additional flags must be given to the lint tool if a source is compiled using the `_LARGEFILE_SOURCE` feature-select macro (see Section 1.3.4 [Feature-Test Macros], page 8).

`_CS_LFS64_CFLAGS`
: The returned string specifies which additional flags must be given to the C compiler if a source is compiled using the `_LARGEFILE64_`

SOURCE feature-select macro (see Section 1.3.4 [Feature-Test Macros], page 8).

_CS_LFS64_LDFLAGS
: The returned string specifies which additional flags must be given to the linker if a source is compiled using the _LARGEFILE64_SOURCE feature-select macro (see Section 1.3.4 [Feature-Test Macros], page 8).

_CS_LFS64_LIBS
: The returned string specifies which additional libraries must be linked to the application if a source is compiled using the _LARGEFILE64_SOURCE feature-select macro (see Section 1.3.4 [Feature-Test Macros], page 8).

_CS_LFS64_LINTFLAGS
: The returned string specifies which additional flags must be given to the lint tool if a source is compiled using the _LARGEFILE64_SOURCE feature-select macro (see Section 1.3.4 [Feature-Test Macros], page 8).

The way to use confstr without any arbitrary limit on string size is to call it twice: first call it to get the length, allocate the buffer accordingly, and then call confstr again to fill the buffer, like this:

```
char *
get_default_path (void)
{
  size_t len = confstr (_CS_PATH, NULL, 0);
  char *buffer = (char *) xmalloc (len);

  if (confstr (_CS_PATH, buf, len + 1) == 0)
    {
      free (buffer);
      return NULL;
    }

  return buffer;
}
```

13 DES Encryption and Password Handling

On many systems, it is unnecessary to have any kind of user authentication. For instance, a workstation that is not connected to a network probably does not need any user authentication, because to use the machine an intruder must have physical access.

Sometimes, however, it is necessary to be sure that a user is authorized to use some service a machine provides—for instance, to log in as a particular user ID (see Chapter 10 [Users and Groups], page 253). One traditional way of doing this is for each user to choose a secret *password*; then, the system can ask someone claiming to be a user what the user's password is, and if the person gives the correct password, the system can grant the appropriate privileges.

If all the passwords are just stored in a file somewhere, then this file has to be very carefully protected. To avoid this, passwords are run through a *one-way function*, a function that makes it difficult to work out what its input was by looking at its output, before being stored in the file.

The GNU C Library provides a one-way function that is compatible with the behavior of the `crypt` function introduced in FreeBSD 2.0. It supports two one-way algorithms: one based on the MD5 message-digest algorithm that is compatible with modern BSD systems, and the other based on the Data Encryption Standard (DES) that is compatible with Unix systems.

It also provides support for Secure RPC, and some library functions that can be used to perform normal DES encryption.

13.1 Legal Problems

Because of the continuously changing state of the law, it's not possible to provide a definitive survey of the laws affecting cryptography. Instead, this section warns you of some of the known trouble spots; this may help you when you try to find out what the laws of your country are.

Some countries require that you have a license to use, possess or import cryptography. These countries are believed to include Byelorussia, Burma, India, Indonesia, Israel, Kazakhstan, Pakistan, Russia and Saudi Arabia.

Some countries restrict the transmission of encrypted messages by radio; some telecommunications carriers restrict the transmission of encrypted messages over their network.

Many countries have some form of export control for encryption software. The Wassenaar Arrangement is a multilateral agreement between 33 countries (Argentina, Australia, Austria, Belgium, Bulgaria, Canada, the Czech Republic, Denmark, Finland, France, Germany, Greece, Hungary, Ireland, Italy, Japan, Luxembourg, the Netherlands, New Zealand, Norway, Poland, Portugal, the Republic of Korea, Romania, the Russian Federation, the Slovak Republic, Spain, Sweden, Switzerland, Turkey, Ukraine, the United Kingdom and the United States) that restricts some kinds of encryption exports. Different countries apply the arrangement

in different ways; some do not allow the exception for certain kinds of "public domain" software (which would include this library), some only restrict the export of software in tangible form, and others impose significant additional restrictions.

The United States has additional rules. This software would generally be exportable under 15 CFR 740.13(e), which permits exports of "encryption source code" that is "publicly available" and that is "not subject to an express agreement for the payment of a licensing fee or royalty for commercial production or sale of any product developed with the source code", to most countries.

The rules in this area are continuously changing. If you know of any information in this manual that is out of date, please report it using the `glibcbug` script (see Section C.6 [Reporting Bugs], page 541).

13.2 Reading Passwords

When reading in a password, it is desirable to avoid displaying it on the screen, to help keep it secret. The following function handles this in a convenient way.

`char *` **getpass** (`const char *`*prompt*) Function

`getpass` outputs *prompt*, then reads a string in from the terminal without echoing it. It tries to connect to the real terminal, '`/dev/tty`', if possible, to encourage users not to put plaintext passwords in files. Otherwise, it uses `stdin` and `stderr`. `getpass` also disables the INTR, QUIT and SUSP characters on the terminal using the `ISIG` terminal attribute (see Section 6.4.7 [Local Modes], page 189). The terminal is flushed before and after `getpass`, so that characters of a mistyped password are not accidentally visible.

In other C libraries, `getpass` may only return the first `PASS_MAX` bytes of a password. The GNU C Library has no limit, so `PASS_MAX` is undefined.

The prototype for this function is in '`unistd.h`'. `PASS_MAX` would be defined in '`limits.h`'.

This precise set of operations may not suit all possible situations. In this case, it is recommended that users write their own `getpass` substitute. For instance, a very simple substitute is as follows:

```
#include <termios.h>
#include <stdio.h>

ssize_t
my_getpass (char **lineptr, size_t *n, FILE *stream)
{
  struct termios old, new;
  int nread;

  /* Turn echoing off and fail if we can't. */
  if (tcgetattr (fileno (stream), &old) != 0)
```

```
    return -1;
  new = old;
  new.c_lflag &= ~ECHO;
  if (tcsetattr (fileno (stream), TCSAFLUSH, &new) != 0)
    return -1;

  /* Read the password.  */
  nread = getline (lineptr, n, stream);

  /* Restore terminal.  */
  (void) tcsetattr (fileno (stream), TCSAFLUSH, &old);

  return nread;
}
```

The substitute takes the same parameters as `getline`;[1] the user must print any prompt desired.

13.3 Encrypting Passwords

`char *` **crypt** (`const char *`*key*`, const char *`*salt*) Function

The `crypt` function takes a password, *key*, as a string, and a *salt* character array, which is described below, and returns a printable ASCII string that starts with another salt. It is believed that, given the output of the function, the best way to find a *key* that will produce that output is to guess values of *key* until the original value of *key* is found.

The *salt* parameter does two things. Firstly, it selects which algorithm is used, the MD5-based one or the DES-based one. Secondly, it makes life harder for someone trying to guess passwords against a file containing many passwords; without a *salt*, an intruder can make a guess, run `crypt` on it once, and compare the result with all the passwords. With a *salt*, the intruder must run `crypt` once for each different salt.

For the MD5-based algorithm, the *salt* should consist of the string `$1$`, followed by up to eight characters, terminated by either another `$` or the end of the string. The result of `crypt` will be the *salt*, followed by a `$` if the salt didn't end with 1, followed by 22 characters from the alphabet `./0-9A-Za-z`, up to 34 characters total. Every character in the *key* is significant.

For the DES-based algorithm, the *salt* should consist of two characters from the alphabet `./0-9A-Za-z`, and the result of `crypt` will be those two characters followed by eleven more from the same alphabet, thirteen in total. Only the first eight characters in the *key* are significant.

[1] See Loosemore et al., "Line-Oriented Input" (see chap. 1, n. 1).

The MD5-based algorithm has no limit on the useful length of the password used, and is slightly more secure. It is therefore preferred over the DES-based algorithm.

When the user enters her password for the first time, the *salt* should be set to a new string that is reasonably random. To verify a password against the result of a previous call to `crypt`, pass the result of the previous call as the *salt*.

The following short program is an example of how to use `crypt` the first time a password is entered. The *salt* generation is just barely acceptable; in particular, it is not unique between machines, and in many applications it would not be acceptable to let an attacker know what time the user's password was last set.

```
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include <crypt.h>

int
main(void)
{
  unsigned long seed[2];
  char salt[] = "$1$........";
  const char *const seedchars =
    "./0123456789ABCDEFGHIJKLMNOPQRST"
    "UVWXYZabcdefghijklmnopqrstuvwxyz";
  char *password;
  int i;

  /* Generate a (not very) random seed.
  You should do it better than this... */
  seed[0] = time(NULL);
  seed[1] = getpid() ^ (seed[0] >> 14 & 0x30000);

  /* Turn it into printable characters from 'seedchars'. */
  for (i = 0; i < 8; i++)
    salt[3+i] = seedchars[(seed[i/5] >> (i%5)*6) & 0x3f];

  /* Read in the user's password and encrypt it. */
  password = crypt(getpass("Password:"), salt);

  /* Print the results. */
  puts(password);
  return 0;
}
```

The next program shows how to verify a password. It prompts the user for a password and prints "Access granted." if the user types `GNU libc manual`.

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <crypt.h>

int
main(void)
{
  /* Hashed form of "GNU libc manual" */
  const char *const pass = "$1$/iSaq7rB$EoUw5jJPPvAPECNaaWzMK/";

  char *result;
  int ok;


  /* Read in the user's password and encrypt it,
  passing the expected password in as the salt. */
  result = crypt(getpass("Password:"), pass);


  /* Test the result. */
  ok = strcmp (result, pass) == 0;

  puts(ok ? "Access granted." : "Access denied.");
  return ok ? 0 : 1;
}
```

`char *` **crypt_r** (`const char *`*key*`, const char *`*salt*`, struct crypt_data *` *data*) Function

The `crypt_r` function does the same thing as `crypt`, but takes an extra parameter that includes space for its result (among other things), so it can be reentrant. `data->initialized` must be cleared to zero before the first time `crypt_r` is called.

The `crypt_r` function is a GNU extension.

The `crypt` and `crypt_r` functions are prototyped in the header '`crypt.h`'.

13.4 DES Encryption

The Data Encryption Standard is described in the US Government Federal Information Processing Standards (FIPS) 46-3 published by the National Institute of

Standards and Technology.[2] The DES has been very thoroughly analyzed since it was developed in the late 1970s, and no new significant flaws have been found.

However, the DES uses only a 56-bit key (plus 8 parity bits), and a machine has been built in 1998 that can search through all possible keys in about 6 days, which cost about US$200,000; faster searches would be possible with more money. This makes simple DES insecure for most purposes, and NIST no longer permits new US government systems to use simple DES.

For serious encryption functionality, it is recommended that one of the many free encryption libraries be used instead of these routines.

The DES is a reversible operation that takes a 64-bit block and a 64-bit key and produces another 64-bit block. Usually, the bits are numbered so that the most significant bit, the first bit of each block, is numbered 1.

Under that numbering, every eighth bit of the key (the eighth, sixteenth, and so on) is not used by the encryption algorithm itself. But the key must have odd parity; that is, out of bits one through eight, and nine through sixteen, and so on, there must be an odd number of '1' bits, and this completely specifies the unused bits.

`void` **setkey** (`const char *`*key*) Function

The `setkey` function sets an internal data structure to be an expanded form of *key*. *key* is specified as an array of 64 bits each stored in a `char`. The first bit is `key[0]` and the 64th bit is `key[63]`. The *key* should have the correct parity.

`void` **encrypt** (`char *`*block*`, int` *edflag*) Function

The `encrypt` function encrypts *block* if *edflag* is 0. Otherwise, it decrypts *block*, using a key previously set by `setkey`. The result is placed in *block*.

Like `setkey`, *block* is specified as an array of 64 bits each stored in a `char`, but there are no parity bits in *block*.

`void` **setkey_r** (`const char *`*key*`, struct crypt_data *` *data*) Function

`void` **encrypt_r** (`char *`*block*`, int` *edflag*`, struct crypt_data *` *data*) Function

These are reentrant versions of `setkey` and `encrypt`. The only difference is the extra parameter, which stores the expanded version of *key*. Before calling `setkey_r` the first time, `data->initialized` must be cleared to zero.

The `setkey_r` and `encrypt_r` functions are GNU extensions. `setkey`, `encrypt`, `setkey_r` and `encrypt_r` are defined in '`crypt.h`'.

[2] National Institute of Standards and Technology, *Federal Information Processing Standards 46-3* (October 1999), http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf.

`int` **ecb_crypt** (`char *`*key*`, char *`*blocks*`, unsigned` *len*`, unsigned` *mode*) Function

The function `ecb_crypt` encrypts or decrypts one or more blocks using DES. Each block is encrypted independently.

The *blocks* and the *key* are stored packed in 8-bit bytes, so that the first bit of the key is the most significant bit of `key[0]` and the 63rd bit of the key is stored as the least significant bit of `key[7]`. The *key* should have the correct parity.

len is the number of bytes in *blocks*. It should be a multiple of 8, so that there is a whole number of blocks to encrypt. *len* is limited to a maximum of `DES_MAXDATA` bytes.

The result of the encryption replaces the input in *blocks*.

The *mode* parameter is the bit-wise OR of two of the following:

`DES_ENCRYPT`
: This constant, used in the *mode* parameter, specifies that *blocks* is to be encrypted.

`DES_DECRYPT`
: This constant, used in the *mode* parameter, specifies that *blocks* is to be decrypted.

`DES_HW` This constant, used in the *mode* parameter, asks to use a hardware device. If no hardware device is available, encryption happens anyway, but in software.

`DES_SW` This constant, used in the *mode* parameter, specifies that no hardware device is to be used.

The result of the function will be one of these values:

`DESERR_NONE`
: The encryption succeeded.

`DESERR_NOHWDEVICE`
: The encryption succeeded, but there was no hardware device available.

`DESERR_HWERROR`
: The encryption failed because of a hardware problem.

`DESERR_BADPARAM`
: The encryption failed because of a bad parameter, for instance *len* is not a multiple of eight or *len* is larger than `DES_MAXDATA`.

`int` **DES_FAILED** (`int` *err*) Function

This macro returns 1 if *err* is a 'success' result code from `ecb_crypt` or `cbc_crypt`, and 0 otherwise.

`int` **cbc_crypt** (`char *`*key*`, char *`*blocks*`, unsigned` *len*`, unsigned` *mode*`, char *`*ivec*) Function

The function `cbc_crypt` encrypts or decrypts one or more blocks using DES in Cipher Block Chaining mode.

For encryption in CBC mode, each block is exclusive-ORed with *ivec* before being encrypted, then *ivec* is replaced with the result of the encryption, then the next block is processed. Decryption is the reverse of this process.

This has the advantage that blocks that are the same before being encrypted are very unlikely to be the same after being encrypted, making it much harder to detect patterns in the data.

Usually, *ivec* is set to 8 random bytes before encryption starts. Then the 8 random bytes are transmitted along with the encrypted data (without themselves being encrypted), and passed back in as *ivec* for decryption. Another possibility is to set *ivec* to eight zeroes initially, and have the first block encrypted consist of 8 random bytes.

Otherwise, all the parameters are similar to those for `ecb_crypt`.

`void` **des_setparity** (`char *`*key*) Function

The function `des_setparity` changes the 64-bit *key*, stored packed in 8-bit bytes, to have odd parity by altering the low bits of each byte.

The `ecb_crypt`, `cbc_crypt` and `des_setparity` functions and their accompanying macros are all defined in the header ‘`rpc/des_crypt.h`’.

14 Resource Usage and Limitation

This chapter describes functions for examining how much of various kinds of resources (CPU time, memory, etc.) a process has used, and for getting and setting limits on future usage.

14.1 Resource Usage

The function `getrusage` and the data-type `struct rusage` are used to examine the resource usage of a process. They are declared in '`sys/resource.h`'.

`int` **getrusage** (`int` *processes*`,` `struct rusage` `*`*rusage*) Function

This function reports resource usage totals for processes specified by *processes*, storing the information in `*`*rusage*.

In most systems, *processes* has only two valid values:

`RUSAGE_SELF`
: This means just the current process.

`RUSAGE_CHILDREN`
: This specifies all child processes, direct and indirect, that have already terminated.

In the GNU system, you can also inquire about a particular child-process by specifying its process ID.

The return value of `getrusage` is zero for success and `-1` for failure.

`EINVAL` The argument *processes* is not valid.

One way of getting resource usage for a particular child-process is with the function `wait4`, which returns totals for a child when it terminates (see Section 7.8 [BSD Process Wait Functions], page 218).

struct rusage Data Type

This data type stores various resource usage statistics. It has the following members, and possibly others:

`struct timeval ru_utime`
: This is the time spent executing user instructions.

`struct timeval ru_stime`
: This is the time spent in operating system code on behalf of *processes*.

`long int ru_maxrss`
: This is the maximum resident set size used, in kilobytes. That is, the maximum number of kilobytes of physical memory that *processes* used simultaneously.

`long int ru_ixrss`
: This is an integral value expressed in kilobytes times ticks of execution, which indicates the amount of memory used by text that was shared with other processes.

`long int ru_idrss`
: This is an integral value expressed the same way, which is the amount of unshared memory used for data.

`long int ru_isrss`
: This is an integral value expressed the same way, which is the amount of unshared memory used for stack space.

`long int ru_minflt`
: This is the number of page faults that were serviced without requiring any I/O.

`long int ru_majflt`
: This is the number of page faults that were serviced by doing I/O.

`long int ru_nswap`
: This is the number of times *processes* was swapped entirely out of main memory.

`long int ru_inblock`
: This is the number of times the file system had to read from the disk on behalf of *processes*.

`long int ru_oublock`
: This is the number of times the file system had to write to the disk on behalf of *processes*.

`long int ru_msgsnd`
: This is the number of IPC messages sent.

`long int ru_msgrcv`
: This is the number of IPC messages received.

`long int ru_nsignals`
: This is the number of signals received.

`long int ru_nvcsw`
: This is the number of times *processes* voluntarily invoked a context switch (usually to wait for some service).

`long int ru_nivcsw`
: The number of times an involuntary context switch took place (because a time slice expired, or another process of higher priority was scheduled).

`vtimes` is a historical function that does some of what `getrusage` does. `getrusage` is a better choice.

`vtimes` and its `vtimes` data structure are declared in '`sys/vtimes.h`'.

`int` **vtimes** (`struct vtimes` *current*, `struct vtimes` *child*) Function

`vtimes` reports resource-usage totals, for a process.

If *current* is nonnull, `vtimes` stores resource usage totals for the invoking process alone, in the structure to which it points. If *child* is nonnull, `vtimes` stores resource-usage totals for all past children (that have terminated) of the invoking process in the structure to which it points.

struct vtimes Data Type

This data type contains information about the resource usage of a process. Each member corresponds to a member of the `struct rusage` data type described above.

`vm_utime`
: This is user CPU time. It is analogous to `ru_utime` in `struct rusage`.

`vm_stime`
: This is system CPU time. It is analogous to `ru_stime` in `struct rusage`.

`vm_idsrss`
: This is data and stack memory, the sum of the values that would be reported as `ru_idrss` and `ru_isrss` in `struct rusage`.

`vm_ixrss`
: This is shared memory. It is analogous to `ru_ixrss` in `struct rusage`.

`vm_maxrss`
: This is maximum resident set size. It is analogous to `ru_maxrss` in `struct rusage`.

`vm_majflt`
: This is major page faults. It is analogous to `ru_majflt` in `struct rusage`.

`vm_minflt`
: This is minor page faults. It is analogous to `ru_minflt` in `struct rusage`.

`vm_nswap`
: This is the swap count. It is analogous to `ru_nswap` in `struct rusage`.

`vm_inblk`
: This is disk reads. It is analogous to `ru_inblk` in `struct rusage`.

`vm_oublk`
: This is disk writes. It is analogous to `ru_oublk` in `struct rusage`.

The return value is 0 if the function succeeds and `-1` otherwise.

An additional historical function for examining resource usage, `vtimes`, is supported but not documented here. It is declared in '`sys/vtimes.h`'.

14.2 Limiting Resource Usage

You can specify limits for the resource usage of a process. When the process tries to exceed a limit, it may get a signal, or the system call by which it tried to do so may fail, depending on the resource. Each process initially inherits its limit values from its parent, but it can subsequently change them.

There are two per-process limits associated with a resource:

current limit
: The current limit is the value the system will not allow usage to exceed. It is also called the "soft limit" because the process being limited can generally raise the current limit at will.

maximum limit
: The maximum limit is the maximum value to which a process is allowed to set its current limit. It is also called the "hard limit" because there is no way for a process to get around it. A process may lower its own maximum limit, but only the superuser may increase a maximum limit.

The symbols for use with `getrlimit`, `setrlimit`, `getrlimit64` and `setrlimit64` are defined in '`sys/resource.h`'.

`int` **getrlimit** (`int` *resource*, `struct rlimit` **rlp*) Function

Read the current and maximum limits for the resource *resource* and store them in **rlp*.

The return value is `0` on success and `-1` on failure. The only possible `errno` error condition is `EFAULT`.

When the sources are compiled with `_FILE_OFFSET_BITS == 64` on a 32-bit system, this function is in fact `getrlimit64`. Thus, the LFS interface transparently replaces the old interface.

`int` **getrlimit64** (`int` *resource*, `struct rlimit64` **rlp*) Function

This function is similar to `getrlimit`, but its second parameter is a pointer to a variable of type `struct rlimit64`, which allows it to read values that wouldn't fit in the member of a `struct rlimit`.

If the sources are compiled with `_FILE_OFFSET_BITS == 64` on a 32-bit machine, this function is available under the name `getrlimit` and so transparently replaces the old interface.

`int` **setrlimit** (`int` *resource*, `const struct rlimit` **rlp*) Function

Store the current and maximum limits for the resource *resource* in **rlp*.

The return value is `0` on success and `-1` on failure. The following `errno` error condition is possible:

`EPERM`

- The process tried to raise a current limit beyond the maximum limit.
- The process tried to raise a maximum limit, but is not superuser.

When the sources are compiled with `_FILE_OFFSET_BITS == 64` on a 32-bit system, this function is in fact `setrlimit64`. Thus, the LFS interface transparently replaces the old interface.

`int` **setrlimit64** (`int` *resource*, `const struct rlimit64` **rlp*) Function

This function is similar to `setrlimit`, but its second parameter is a pointer to a variable of type `struct rlimit64` that allows it to set values that wouldn't fit in the member of a `struct rlimit`.

If the sources are compiled with `_FILE_OFFSET_BITS == 64` on a 32-bit machine, this function is available under the name `setrlimit` and so transparently replaces the old interface.

struct rlimit Data Type

This structure is used with `getrlimit` to receive limit values, and with `setrlimit` to specify limit values for a particular process and resource. It has two fields:

`rlim_t rlim_cur`
: This is the current limit.

`rlim_t rlim_max`
: This is the maximum limit.

For `getrlimit`, the structure is an output; it receives the current values. For `setrlimit`, it specifies the new values.

For the LFS functions, a similar type is defined in '`sys/resource.h`'.

struct rlimit64 Data Type

This structure is analogous to the `rlimit` structure above, but its components have wider ranges. It has two fields:

`rlim64_t rlim_cur`
: This is analogous to `rlimit.rlim_cur`, but with a different type.

`rlim64_t rlim_max`
: This is analogous to `rlimit.rlim_max`, but with a different type.

Here is a list of resources for which you can specify a limit. Memory and file sizes are measured in bytes.

`RLIMIT_CPU`
: This is the maximum amount of CPU time the process can use. If it runs for longer than this, it gets a signal, `SIGXCPU`. The value is measured in seconds (see Section 17.2.6 [Operation-Error Signals], page 387).

`RLIMIT_FSIZE`
: This is the maximum size of file the process can create. Trying to write a larger file causes a signal, `SIGXFSZ` (see Section 17.2.6 [Operation-Error Signals], page 387).

`RLIMIT_DATA`
: This is the maximum size of data memory for the process. If the process tries to allocate data memory beyond this amount, the allocation function fails.

`RLIMIT_STACK`
: This is the maximum stack size for the process. If the process tries to extend its stack past this size, it gets a `SIGSEGV` signal (see Section 17.2.1 [Program-Error Signals], page 379).

`RLIMIT_CORE`
: This is the maximum size core file that this process can create. If the process terminates and would dump a core file larger than this, then no core file is created. So setting this limit to 0 prevents core files from ever being created.

`RLIMIT_RSS`
: This is the maximum amount of physical memory that this process should get. This parameter is a guide for the system's scheduler and memory allocator; the system may give the process more memory when there is a surplus.

`RLIMIT_MEMLOCK`
: This is the maximum amount of memory that can be locked into physical memory (so it will never be paged out).

`RLIMIT_NPROC`
: This is the maximum number of processes that can be created with the same user-ID. If you have reached the limit for your user-ID, `fork` will fail with `EAGAIN` (see Section 7.4 [Creating a Process], page 211).

`RLIMIT_NOFILE`
`RLIMIT_OFILE`
: The maximum number of files that the process can open. If it tries to open more files than this, its open attempt fails with `errno` `EMFILE`.[1] Not all systems support this limit; GNU does, and 4.4 BSD does.

`RLIMIT_AS`
: This is the maximum size of total memory that this process should get. If the process tries to allocate more memory beyond this amount with, for example, `brk`, `malloc`, `mmap` or `sbrk`, the allocation function fails.

`RLIM_NLIMITS`
: This is the number of different resource limits. Any valid *resource* operand must be less than `RLIM_NLIMITS`.

`int` **RLIM_INFINITY** Constant

This constant stands for a value of "infinity" when supplied as the limit value in `setrlimit`.

The following are historical functions to do some of what the functions above do. The functions above are better choices.

`ulimit` and the command symbols are declared in '`ulimit.h`'.

`int` **ulimit** (`int` *cmd*, `...`) Function

`ulimit` gets the current limit or sets the current and maximum limit for a particular resource for the calling process according to the command *cmd*.a

If you are getting a limit, the command argument is the only argument. If you are setting a limit, there is a second argument: `long int` *limit*, which is the value to which you are setting the limit.

The *cmd* values and the operations they specify are:

`GETFSIZE`
: Get the current limit on the size of a file, in units of 512 bytes.

`SETFSIZE`
: Set the current and maximum limit on the size of a file to *limit* * 512 bytes.

There are also some other *cmd* values that may do things on some systems, but they are not supported.

Only the superuser may increase a maximum limit.

When you successfully get a limit, the return value of `ulimit` is that limit, which is never negative. When you successfully set a limit, the return value is 0. When the function fails, the return value is `-1` and `errno` is set according to the reason:

[1] See Loosemore et al., "Error Codes" (see chap. 1, n. 1).

`EPERM` A process tried to increase a maximum limit, but is not superuser.

`vlimit` and its resource symbols are declared in '`sys/vlimit.h`'.

`int` **vlimit** (`int` *resource*, `int` *limit*) Function

`vlimit` sets the current limit for a resource for a process.

resource identifies the resource:

`LIM_CPU` This is the maximum CPU time. It is the same as `RLIMIT_CPU` for `setrlimit`.

`LIM_FSIZE`
This is the maximum file size. It is the same as `RLIMIT_FSIZE` for `setrlimit`.

`LIM_DATA`
This is the maximum data memory. It is the same as `RLIMIT_DATA` for `setrlimit`.

`LIM_STACK`
This is the maximum stack size. It is the same as `RLIMIT_STACK` for `setrlimit`.

`LIM_CORE`
This is the maximum core file size. It is the same as `RLIMIT_COR` for `setrlimit`.

`LIM_MAXRSS`
This is the maximum physical memory. It is the same as `RLIMIT_RSS` for `setrlimit`.

The return value is 0 for success and `-1` with `errno` set accordingly for failure:

`EPERM` The process tried to set its current limit beyond its maximum limit.

14.3 Process CPU Priority and Scheduling

When multiple processes simultaneously require CPU time, the system's scheduling policy and process CPU priorities determine which processes get it. This section describes how that determination is made and GNU C Library functions to control it.

It is common to refer to CPU scheduling simply as scheduling and a process's CPU priority simply as the process's priority, with the CPU resource being implied. Bear in mind, though, that CPU time is not the only resource a process uses or that processes contend for. In some cases, it is not even particularly important. Giving a process a high "priority" may have very little effect on how fast a process runs with respect to other processes. The priorities discussed in this section apply only to CPU time.

CPU scheduling is a complex issue and different systems do it in wildly different ways. New ideas continually develop and find their way into the intricacies of the

various systems' scheduling algorithms. This section discusses the general concepts, some specifics of systems that commonly use the GNU C Library, and some standards.

For simplicity, we talk about CPU contention as if there is only one CPU in the system. But all the same principles apply when a processor has multiple CPUs, and knowing that the number of processes that can run at any one time is equal to the number of CPUs, you can easily extrapolate the information.

The functions described in this section are all defined by the POSIX.1 and POSIX.1b standards (the `sched...` functions are POSIX.1b). However, POSIX does not define any semantics for the values that these functions get and set. In this chapter, the semantics are based on the Linux kernel's implementation of the POSIX standard. As you will see, the Linux implementation is quite the inverse of what the authors of the POSIX syntax had in mind.

14.3.1 Absolute Priority

Every process has an absolute priority, and it is represented by a number. The higher the number, the higher the absolute priority.

On systems of the past, and most systems today, all processes have absolute priority 0 and this section is irrelevant. In that case, see Section 14.3.4 [Traditional Scheduling], page 349. Absolute priorities were invented to accommodate real-time systems, in which it is vital that certain processes be able to respond to external events happening in real time, which means they cannot wait around while some other process that *wants to*, but doesn't *need to* run occupies the CPU.

When two processes are in contention to use the CPU at any instant, the one with the higher absolute priority always gets it. This is true even if the process with the lower priority is already using the CPU—the scheduling is preemptive. Of course, we're only talking about processes that are running or "ready to run", which means they are ready to execute instructions right now. The term "runnable" is a synonym for "ready to run." When a process blocks to wait for something like I/O, its absolute priority is irrelevant.

When two processes are running or ready to run, and both have the same absolute priority, it's more interesting. In that case, who gets the CPU is determined by the scheduling policy. If the processes have absolute priority 0, the traditional scheduling policy described in Section 14.3.4 [Traditional Scheduling], page 349 applies. Otherwise, the policies described in Section 14.3.2 [Real-Time Scheduling], page 345, apply.

You normally give an absolute priority above 0 only to a process that can be trusted not to hog the CPU. Such processes are designed to block (or terminate) after relatively short CPU runs.

A process begins life with the same absolute priority as its parent process. Functions described in Section 14.3.3 [Basic Scheduling Functions], page 346 can change it.

Only a privileged process can change a process's absolute priority to something other than `0`. Only a privileged process or the target process's owner can change its absolute priority at all.

POSIX requires absolute priority values used with the real-time scheduling policies to be consecutive with a range of at least 32. On Linux, they are 1 through 99. The functions `sched_get_priority_max` and `sched_set_priority_min` portably tell you what the range is on a particular system.

14.3.1.1 Using Absolute Priority

One thing you must keep in mind when designing real-time applications is that having higher absolute priority than any other process doesn't guarantee the process can run continuously. Two things that can wreck a good CPU run are interrupts and page faults.

Interrupt handlers live in that limbo between processes. The CPU is executing instructions, but they aren't part of any process. An interrupt will stop even the highest priority process. So you must allow for slight delays and make sure that no device in the system has an interrupt handler that could cause too long a delay between instructions for your process.

Similarly, a page fault causes what looks like a straightforward sequence of instructions to take a long time. The fact that other processes get to run while the page faults in is of no consequence, because as soon as the I/O is complete, the high-priority process will kick them out and run again, but the wait for the I/O itself could be a problem. To neutralize this threat, use `mlock` or `mlockall`.

There are a few ramifications of the absoluteness of this priority on a single-CPU system that you need to keep in mind when you choose to set a priority and also when you're working on a program that runs with high absolute priority. Consider a process that has higher absolute priority than any other process in the system and due to a bug in its program, it gets into an infinite loop. It will never cede the CPU. You can't run a command to kill it because your command would need to get the CPU in order to run. The errant program is in complete control. It controls the vertical, it controls the horizontal.

There are two ways to avoid this: 1) keep a shell running somewhere with a higher absolute priority; 2) keep a controlling terminal attached to the high-priority process group. All the priority in the world won't stop an interrupt handler from running and delivering a signal to the process if you hit Control-C.

Some systems use absolute priority as a means of allocating a fixed percentage of CPU time to a process. To do this, a super-high-priority privileged process constantly monitors the process's CPU usage and raises its absolute priority when the process isn't getting its entitled share and lowers it when the process is exceeding it.

The absolute priority is sometimes called the "static priority". We don't use that term in this manual because it misses the most important feature of the absolute priority—its absoluteness.

14.3.2 Real-Time Scheduling

Whenever two processes with the same absolute priority are ready to run, the kernel has a decision to make, because only one can run at a time. If the processes have absolute priority 0, the kernel makes this decision as described in Section 14.3.4 [Traditional Scheduling], page 349. Otherwise, the decision is as described in this section.

If two processes are ready to run but have different absolute priorities, the decision is much simpler, and is described in Section 14.3.1 [Absolute Priority], page 343.

Each process has a scheduling policy. For processes with absolute priority other than 0, there are two available:

1. First-come first-served
2. Round-robin

The most sensible case is where all the processes with a certain absolute priority have the same scheduling policy. We'll discuss that first.

Under a round-robin policy, processes share the CPU, each one running for a small quantum of time ("time slice") and then yielding to another in a circular fashion. Of course, only processes that are ready to run and have the same absolute priority are in this circle.

Under a first-come first-served policy, the process that has been waiting the longest to run gets the CPU, and it keeps it until it voluntarily relinquishes the CPU, runs out of things to do (blocks), or gets preempted by a higher priority process.

First-come first-served, along with maximum absolute priority and careful control of interrupts and page faults, is the one to use when a process absolutely, positively has to run at full CPU speed or not at all.

Judicious use of `sched_yield` function invocations by processes with first-come first-served scheduling policy forms a good compromise between round-robin and first-come first-served.

To understand how scheduling works when processes of different scheduling policies occupy the same absolute priority, you have to know the nitty-gritty details of how processes enter and exit the ready-to-run list.

In both cases, the ready-to-run list is organized as a true queue, where a process gets pushed onto the tail when it becomes ready to run and is popped off the head when the scheduler decides to run it. Ready to run and running are two mutually exclusive states. When the scheduler runs a process, that process is no longer ready to run and no longer in the ready-to-run list. When the process stops running, it may go back to being ready to run again.

The only difference between a process that is assigned the round-robin scheduling policy and a process that is assigned first-come first-served is that in the former case, the process is automatically booted off the CPU after a certain amount of time. When that happens, the process goes back to being ready to run, which means it enters the queue at the tail. The time quantum we're talking about is small—really small. This is not your father's timesharing. For example, with the Linux kernel,

the round-robin time slice is a thousand times shorter than its typical time-slice for traditional scheduling.

A process begins life with the same scheduling policy as its parent process. Functions described in Section 14.3.3 [Basic Scheduling Functions], page 346 can change it.

Only a privileged process can set the scheduling policy of a process that has absolute priority higher than 0.

14.3.3 Basic Scheduling Functions

This section describes functions in the GNU C Library for setting the absolute priority and scheduling policy of a process.

Portability Note: On systems that have the functions in this section, the macro _POSIX_PRIORITY_SCHEDULING is defined in '<unistd.h>'.

When the scheduling policy is traditional scheduling, more functions to fine tune the scheduling are in Section 14.3.4 [Traditional Scheduling], page 349.

Don't try to make too much out of the naming and structure of these functions. They don't match the concepts described in this manual because the functions are as defined by POSIX.1b, but the implementation on systems that use the GNU C Library is the inverse of what the POSIX structure contemplates. The POSIX scheme assumes that the primary scheduling parameter is the scheduling policy and that the priority value, if any, is a parameter of the scheduling policy. In the implementation, though, the priority value is king and the scheduling policy, if anything, only fine-tunes the effect of that priority.

The symbols in this section are declared by including file 'sched.h'.

struct sched_param Data Type

This structure describes an absolute priority.

`int sched_priority`
: This is the absolute priority value.

`int` **sched_setscheduler** (`pid_t` *pid*, `int` *policy*, `const struct sched_param *`*param*) Function

This function sets both the absolute priority and the scheduling policy for a process.

It assigns the absolute priority value given by *param* and the scheduling policy *policy* to the process with Process ID *pid*, or the calling process if *pid* is 0. If *policy* is negative, `sched_setscheduler` keeps the existing scheduling policy.

The following macros represent the valid values for *policy*:

`SCHED_OTHER`
: This is traditional scheduling.

`SCHED_FIFO`
: This is first-in first-out.

`SCHED_RR`
: This is round-robin.

On success, the return value is `0`. Otherwise, it is `-1` and `ERRNO` is set accordingly. The `errno` values specific to this function are

`EPERM`
- The calling process does not have `CAP_SYS_NICE` permission and *policy* is not `SCHED_OTHER` (or it's negative and the existing policy is not `SCHED_OTHER`).
- The calling process does not have `CAP_SYS_NICE` permission and its owner is not the target process's owner. The effective uid of the calling process is neither the effective nor the real uid of process *pid*.

`ESRCH` There is no process with PID *pid* and *pid* is not 0.

`EINVAL`
- *policy* does not identify an existing scheduling policy.
- The absolute priority value identified by **param* is outside the valid range for the scheduling policy *policy* (or the existing scheduling policy if *policy* is negative) or *param* is null. `sched_get_priority_max` and `sched_get_priority_min` tell you what the valid range is.
- *pid* is negative.

`int` **sched_getscheduler** (`pid_t` *pid*) Function

This function returns the scheduling policy assigned to the process with process ID (PID) *pid*, or the calling process if *pid* is 0.

The return value is the scheduling policy. See `sched_setscheduler` for the possible values.

If the function fails, the return value is instead `-1` and `errno` is set accordingly.

The `errno` values specific to this function are

`ESRCH` There is no process with PID *pid* and it is not 0.

`EINVAL` *pid* is negative.

This function is not an exact mate to `sched_setscheduler`, because while that function sets the scheduling policy and the absolute priority, this function gets only the scheduling policy. To get the absolute priority, use `sched_getparam`.

`int` **sched_setparam** (`pid_t` *pid*, `const struct sched_param` **param*) Function

This function sets a process's absolute priority.

It is functionally identical to `sched_setscheduler` with *policy* = `-1`.

`int` **sched_getparam** (`pid_t` *pid*, `const struct sched_param *`*param*) Function

This function returns a process's absolute priority.

pid is the process ID (PID) of the process whose absolute priority you want to know.

param is a pointer to a structure in which the function stores the absolute priority of the process.

On success, the return value is `0`. Otherwise, it is `-1` and `ERRNO` is set accordingly. The `errno` values specific to this function are

`ESRCH` There is no process with PID *pid* and it is not 0.

`EINVAL` *pid* is negative.

`int` **sched_get_priority_min** (`int *`*policy*); Function

This function returns the lowest absolute priority value that is allowable for a process with scheduling policy *policy*.

On Linux, it is 0 for SCHED_OTHER and 1 for everything else.

On success, the return value is `0`. Otherwise, it is `-1` and `ERRNO` is set accordingly. The `errno` value specific to this function is

`EINVAL` *policy* does not identify an existing scheduling policy.

`int` **sched_get_priority_max** (`int *`*policy*); Function

This function returns the highest absolute priority value that is allowable for a process with scheduling policy *policy*.

On Linux, it is 0 for SCHED_OTHER and 99 for everything else.

On success, the return value is `0`. Otherwise, it is `-1` and `ERRNO` is set accordingly. The `errno` value specific to this function is

`EINVAL` *policy* does not identify an existing scheduling policy.

`int` **sched_rr_get_interval** (`pid_t` *pid*, `struct timespec *`*interval*) Function

This function returns the length of the quantum (time slice) used with the round-robin scheduling policy, if it is used, for the process with process ID *pid*.

It returns the length of time as *interval*.

With a Linux kernel, the round-robin time slice is always 150 microseconds, and *pid* need not even be a real PID.

The return value is `0` on success and in the pathological case that it fails, the return value is `-1` and `errno` is set accordingly. There is nothing specific that can go wrong with this function, so there are no specific `errno` values.

`int` **sched_yield** (`void`) Function

This function voluntarily gives up the process's claim on the CPU.

Technically, `sched_yield` causes the calling process to be made immediately ready to run (as opposed to running, which is what it was before). This means that if it has absolute priority higher than 0, it gets pushed onto the tail of the queue of processes that share its absolute priority and are ready to run, and it will run again when its turn next arrives. If its absolute priority is 0, it is more complicated, but still has the effect of yielding the CPU to other processes.

If there are no other processes that share the calling process's absolute priority, this function doesn't have any effect.

To the extent that the containing program is oblivious to what other processes in the system are doing and how fast it executes, this function appears as a no-op.

The return value is `0` on success and in the pathological case that it fails, the return value is `-1` and `errno` is set accordingly. There is nothing specific that can go wrong with this function, so there are no specific `errno` values.

14.3.4 Traditional Scheduling

This section is about the scheduling among processes whose absolute priority is 0. When the system hands out the scraps of CPU time that are left over after the processes with higher absolute priority have taken all they want, the scheduling described herein determines who among the great unwashed processes gets them.

14.3.4.1 Introduction to Traditional Scheduling

Long before there was absolute priority (see Section 14.3.1 [Absolute Priority], page 343), Unix systems were scheduling the CPU using this system. When Posix came in like the Romans and imposed absolute priorities to accommodate the needs of real-time processing, it left the indigenous Absolute Priority Zero processes to govern themselves by their own familiar scheduling policy.

Indeed, absolute priorities higher than 0 are not available on many systems today and are not typically used when they are, being intended mainly for computers that do real-time processing. So this section describes the only scheduling many programmers need to be concerned about.

But just to be clear about the scope of this scheduling: Any time a process with a absolute priority of 0 and a process with an absolute priority higher than 0 are ready to run at the same time, the one with absolute priority 0 does not run. If it's already running when the higher priority ready-to-run process comes into existence, it stops immediately.

In addition to its absolute priority of 0, every process has another priority, which we will refer to as *dynamic priority* because it changes over time. The dynamic priority is meaningless for processes with an absolute priority higher than 0.

The dynamic priority sometimes determines who gets the next turn on the CPU. Sometimes it determines how long turns last. Sometimes it determines whether a process can kick another off the CPU.

In Linux, the value is a combination of these things, but mostly it just determines the length of the time slice. The higher a process's dynamic priority, the longer a

shot it gets on the CPU when it gets one. If it doesn't use up its time slice before giving up the CPU to do something like wait for I/O, it is favored for getting the CPU back when it's ready for it, to finish out its time slice. Other than that, selection of processes for new time slices is basically round robin. But the scheduler does throw a bone to the low-priority processes: A process's dynamic priority rises every time it is snubbed in the scheduling process.

The fluctuation of a process's dynamic priority is regulated by another value—the *nice* value. The nice value is an integer, usually in the range -20 to 20, and represents an upper limit on a process's dynamic priority. The higher the nice number, the lower that limit.

On a typical Linux system, for example, a process with a nice value of 20 can get only 10 milliseconds on the CPU at a time, whereas a process with a nice value of -20 can achieve a high enough priority to get 400 milliseconds.

The idea of the nice value is deferential courtesy. In the beginning, in the Unix garden of Eden, all processes shared equally in the bounty of the computer system. But not all processes really need the same share of CPU time, so the nice value gave a courteous process the ability to refuse its equal share of CPU time that others might prosper. Hence, the higher a process's nice value, the nicer the process is.

Dynamic priorities tend upward and downward with an objective of smoothing out allocation of CPU time and giving quick response time to infrequent requests. But they never exceed their nice limits, so on a heavily loaded CPU, the nice value effectively determines how fast a process runs.

In keeping with the socialistic heritage of Unix process priority, a process begins life with the same nice value as its parent process and can raise it at will. A process can also raise the nice value of any other process owned by the same user (or effective user). But only a privileged process can lower its nice value. A privileged process can also raise or lower another process's nice value.

GNU C Library functions for getting and setting nice values are described in Section 14.3.4.2 [Functions for Traditional Scheduling], page 350.

14.3.4.2 Functions for Traditional Scheduling

This section describes how you can read and set the nice value of a process. All these symbols are declared in '`sys/resource.h`'.

The function and macro names are defined by POSIX, and refer to "priority", but the functions actually have to do with nice values, as the terms are used both in the manual and POSIX.

The range of valid nice values depends on the kernel, but typically it runs from `-20` to `20`. A lower nice value corresponds to higher priority for the process. These constants describe the range of priority values:

`PRIO_MIN`
: This is the lowest valid nice value.

`PRIO_MAX`
: This is the highest valid nice value.

`int` **getpriority** (`int` *class*, `int` *id*) Function

Return the nice value of a set of processes; *class* and *id* specify which ones (see below). If the processes specified do not all have the same nice value, this returns the lowest value that any of them has.

On success, the return value is `0`. Otherwise, it is `-1` and `ERRNO` is set accordingly. The `errno` values specific to this function are

`ESRCH` The combination of *class* and *id* does not match any existing process.

`EINVAL` The value of *class* is not valid.

If the return value is `-1`, it could indicate failure, or it could be the nice value. The only way to make certain is to set `errno = 0` before calling `getpriority`, then use `errno != 0` afterward as the criterion for failure.

`int` **setpriority** (`int` *class*, `int` *id*, `int` *niceval*) Function

Set the nice value of a set of processes to *niceval*; *class* and *id* specify which ones (see below).

The return value is `0` on success, and `-1` on failure. The following `errno` error condition are possible for this function are

`ESRCH` The combination of *class* and *id* does not match any existing process.

`EINVAL` The value of *class* is not valid.

`EPERM` The call would set the nice value of a process that is owned by a different user than the calling process (i.e. the target process's real or effective uid does not match the calling process's effective uid) and the calling process does not have `CAP_SYS_NICE` permission.

`EACCES` The call would lower the process's nice value and the process does not have `CAP_SYS_NICE` permission.

The arguments *class* and *id* together specify a set of processes in which you are interested. These are the possible values of *class*:

`PRIO_PROCESS`
: This is one particular process. The argument *id* is a process ID (PID).

`PRIO_PGRP`
: This is all the processes in a particular process-group. The argument *id* is a process-group ID (pgid).

`PRIO_USER`
: This is all the processes owned by a particular user (i.e. whose real uid indicates the user). The argument *id* is a user ID (uid).

If the argument *id* is 0, it stands for the calling process, its process group or its owner (real uid), according to *class*.

`int` **nice** (`int` *increment*) Function

Increment the nice value of the calling process by *increment*. The return value is the new nice value on success and `-1` on failure. In the case of failure, `errno` will be set to the same values as for `setpriority`.

Here is an equivalent definition of `nice`:

```
int
nice (int increment)
{
  int result, old = getpriority (PRIO_PROCESS, 0);
  result = setpriority (PRIO_PROCESS, 0, old + increment);
  if (result != -1)
      return old + increment;
  else
      return -1;
}
```

14.3.5 Limiting Execution to Certain CPUs

On a multiprocessor system, the operating system usually distributes the different processes that are runnable on all available CPUs in a way that allows the system to work most efficiently. Which processes and threads run can be to some extent controlled with the scheduling functionality described in the last sections. But which CPU finally executes which process or thread is not covered.

There are a number of reasons why a program might want to have control over this aspect of the system as well:

- One thread or process is responsible for absolutely critical that which under no circumstances must be interrupted or hindered from making progress by other processes or threads using CPU resources. In this case, the special process would be confined to a CPU that no other process or thread is allowed to use.
- The access to certain resources (RAM, I/O ports) has different costs from different CPUs. This is the case in NUMA (Nonuniform Memory Architecture) machines. Preferably memory should be accessed locally but this requirement is usually not visible to the scheduler. Therefore, forcing a process or thread to the CPUs that have local access to the mostly used memory helps to significantly boost the performance.
- In controlled run-times, resource allocation and bookkeeping work (like garbage collection) are performance-local to processors. This can help to reduce locking costs if the resources do not have to be protected from concurrent accesses from different processors.

The POSIX standard up to this date is of not much help to solve this problem. The Linux kernel provides a set of interfaces to allow specifying *affinity sets* for a process. The scheduler will schedule the thread or process on on CPUs specified by the affinity masks. The interfaces that the GNU C Library define follow to some extent the Linux kernel interface.

cpu_set_t Data Type
This data set is a bitset where each bit represents a CPU. How the system's CPUs are mapped to bits in the bitset is system dependent. The data type has a fixed size; in the unlikely case that the number of bits are not sufficient to describe the CPUs of the system, a different interface has to be used.

This type is a GNU extension and is defined in '`sched.h`'.

To manipulate the bitset, to set and reset bits, a number of macros are defined. Some of the macros take a CPU number as a parameter. Here it is important to never exceed the size of the bitset. The following macro specifies the number of bits in the `cpu_set_t` bitset:

`int` **CPU_SETSIZE** Macro
The value of this macro is the maximum number of CPUs that can be handled with a `cpu_set_t` object.

The type `cpu_set_t` should be considered opaque; all manipulation should happen via the next four macros.

`void` **CPU_ZERO** (`cpu_set_t *`*set*) Macro
This macro initializes the CPU set *set* to be the empty set.

This macro is a GNU extension and is defined in '`sched.h`'.

`void` **CPU_SET** (`int` *cpu*, `cpu_set_t *`*set*) Macro
This macro adds *cpu* to the CPU set *set*.

The *cpu* parameter must not have side effects, since it is evaluated more than once.

This macro is a GNU extension and is defined in '`sched.h`'.

`void` **CPU_CLR** (`int` *cpu*, `cpu_set_t *`*set*) Macro
This macro removes *cpu* from the CPU set *set*.

The *cpu* parameter must not have side effects, since it is evaluated more than once.

This macro is a GNU extension and is defined in '`sched.h`'.

`int` **CPU_ISSET** (`int` *cpu*, `const cpu_set_t *`*set*) Macro
This macro returns a nonzero value (true) if *cpu* is a member of the CPU set *set*, and 0 (false) otherwise.

The *cpu* parameter must not have side effects, since it is evaluated more than once.

This macro is a GNU extension and is defined in '`sched.h`'.

CPU bitsets can be constructed from scratch or the currently installed affinity mask can be retrieved from the system.

`int` **sched_getaffinity** (`pid_t` *pid*, `cpu_set_t *`*cpuset*) Function

This function stores the CPU affinity mask for the process or thread with the ID *pid* in the memory pointed to by *cpuset*. If successful, the function always initializes all bits in the `cpu_set_t` object and returns 0.

If *pid* does not correspond to a process or thread on the system or the function fails for some other reason, it returns `-1` and `errno` is set to represent the error condition.

`ESRCH` No process or thread with the given ID is found.

`EFAULT` The pointer *cpuset* is does not point to a valid object.

This function is a GNU extension and is declared in '`sched.h`'.

It is not portably possible to use this information to retrieve the information for different POSIX threads. A separate interface must be provided for that.

`int` **sched_setaffinity** (`pid_t` *pid*, `const cpu_set_t *`*cpuset*) Function

This function installs the affinity mask pointed to by *cpuset* for the process or thread with the ID *pid*. If successful, the function returns 0 and the scheduler will in future take the affinity information into account.

If the function fails, it will return `-1` and `errno` is set to the error code:

`ESRCH` No process or thread with the given ID is found.

`EFAULT` The pointer *cpuset* is does not point to a valid object.

`EINVAL` The bitset is not valid. This might mean that the affinity set might not leave a processor for the process or thread to run on.

This function is a GNU extension and is declared in '`sched.h`'.

14.4 Querying Memory-Available Resources

The amount of memory available in the system and the way it is organized determines oftentimes the way programs can and have to work. For functions like `mmap`, it is necessary to know about the size of individual memory pages, and knowing how much memory is available enables a program to select appropriate sizes for, say, caches. Before we get into these details, a few words about memory subsystems in traditional Unix systems will be given.

14.4.1 Overview of Traditional Unix Memory-Handling

Unix systems normally provide processes virtual-address spaces. This means that the addresses of the memory regions do not have to correspond directly to the addresses of the actual physical memory that stores the data. An extra level of indirection is introduced that translates virtual addresses into physical addresses. This is normally done by the hardware of the processor.

Using a virtual address space has several advantages. The most important is process isolation. The different processes running on the system cannot interfere directly with each other. No process can write into the address space of another process (except when shared memory is used, in which case this is desired and controlled).

Another advantage of virtual memory is that the address space the processes see can actually be larger than the physical memory available. The physical memory can be extended by storage on an external media where the content of currently unused memory regions is stored. The address translation can then intercept accesses to these memory regions and make memory content available again by loading the data back into memory. This concept makes it necessary that programs that have to use lots of memory know the difference between available virtual-address space and available physical memory. If the working set of virtual memory of all the processes is larger than the available physical memory the system will slow down dramatically due to constant swapping of memory content from the memory to the storage media and back. This is called *thrashing*.

A final aspect of virtual memory that is important and follows from what is said in the last paragraph is the granularity of the virtual-address space handling. Although the virtual-address handling stores memory content externally, it cannot do this on a byte-by-byte basis. The administrative overhead does not allow this (not to mention the processor hardware). Instead, several thousand bytes are handled together and form a *page*. The size of each page is always a power of 2 bytes. The smallest page size in use today is 4096, with 8192, 16384 and 65536 being other popular sizes.

14.4.2 How to Get Information About the Memory Subsystem?

It is essential in several situations to know the page size of the virtual memory the process sees. Some programming interfaces (e.g., `mmap`; see Section 2.7 [Memory-Mapped I/O], page 32) require the user to provide information adjusted to the page size. In the case of `mmap` it is necessary to provide a length argument that is a multiple of the page size. Another place where knowledge about the page size is useful is in memory allocation. If you allocate pieces of memory in larger chunks that are then subdivided by the application code, it is useful to adjust the size of the larger blocks to the page size. If the total memory requirement for the block is close to (but not larger than) a multiple of the page size, the kernel's memory-handling can work more effectively since it only has to allocate memory pages that are fully used. To do this optimization, it is necessary to know a bit about the memory allocator, which will require a bit of memory itself for each block, and this overhead must not push the total size over the page size multiple.

The page size traditionally was a compile-time constant. But recent development of processors changed this. Processors now support different page sizes, and they can possibly even vary among different processes on the same system. There-

fore, the system should be queried at run-time about the current page-size, and no assumptions (except about it being a power of 2) should be made.

The correct interface to query about the page size is `sysconf` (see Section 12.4.1 [Definition of sysconf], page 307) with the parameter `_SC_PAGESIZE`. There is a much older interface available, too.

`int` **getpagesize** (`void`) Function

The `getpagesize` function returns the page size of the process. This value is fixed for the runtime of the process but can vary in different runs of the application.

The function is declared in '`unistd.h`'.

Widely available on System V-derived systems is a method to get information about the physical memory the system has. The call:

```
sysconf (_SC_PHYS_PAGES)
```

returns the total number of pages of physical the system has. This does not mean all this memory is available. This information can be found using:

```
sysconf (_SC_AVPHYS_PAGES)
```

These two values help to optimize applications. The value returned for `_SC_AVPHYS_PAGES` is the amount of memory the application can use without hindering any other process (given that no other process increases its memory usage). The value returned for `_SC_PHYS_PAGES` is more or less a hard limit for the working set. If all applications together constantly use more than that amount of memory, the system is in trouble.

The GNU C Library provides two functions in addition to the ways already described to get this information. They are declared in the file '`sys/sysinfo.h`'. Programmers should prefer to use the `sysconf` method described above.

`long int` **get_phys_pages** (`void`) Function

The `get_phys_pages` function returns the total number of pages of physical memory the system has. To get the amount of memory, this number has to be multiplied by the page size.

This function is a GNU extension.

`long int` **get_avphys_pages** (`void`) Function

The `get_phys_pages` function returns the number of available pages of physical memory the system has. To get the amount of memory, this number has to be multiplied by the page size.

This function is a GNU extension.

14.5 Learn About the Processors Available

The use of threads or processes with shared memory allows an application to take advantage of all the processing power a system can provide. If the task can be

parallelized, the optimal way to write an application is to have at any time as many processes running as there are processors. To determine the number of processors available to the system you can run:

```
sysconf (_SC_NPROCESSORS_CONF)
```

which returns the number of processors the operating system configured. But it might be possible for the operating system to disable individual processors, and so the call:

```
sysconf (_SC_NPROCESSORS_ONLN)
```

returns the number of processors that are currently in-line (i.e., available).

The GNU C Library also provides functions to get this information directly. The functions are declared in ‘`sys/sysinfo.h`’.

`int` **get_nprocs_conf** (`void`) Function

The `get_nprocs_conf` function returns the number of processors the operating system configured.

This function is a GNU extension.

`int` **get_nprocs** (`void`) Function

The `get_nprocs` function returns the number of available processors.

This function is a GNU extension.

Before starting more threads, it should be checked whether the processors are not already overused. Unix systems calculate something called the *load average*. This is a number indicating how many processes were running. This number is average over different periods of times (normally 1, 5 and 15 minutes).

`int` **getloadavg** (`double` *loadavg*`[],` `int` *nelem*) Function

This function gets the 1-, 5- and 15-minute load averages of the system. The values are placed in *loadavg*. `getloadavg` will place at most *nelem* elements into the array but never more than three elements. The return value is the number of elements written to *loadavg*, or -1 on error.

This function is declared in ‘`stdlib.h`’.

15 Syslog

This chapter describes facilities for issuing and logging messages of system-administration interest. This chapter has nothing to do with programs issuing messages to their own users or keeping private logs.[1]

Most systems have a facility called "Syslog" that allows programs to submit messages of interest to system administrators and can be configured to pass these messages on in various ways, such as printing on the console, mailing to a particular person, or recording in a log file for future reference.

A program uses the facilities in this chapter to submit such messages.

15.1 Overview of Syslog

System administrators have to deal with lots of different kinds of messages from a plethora of subsystems within each system, and usually lots of systems as well. For example, an FTP server might report every connection it gets. The kernel might report hardware failures on a disk drive. A DNS server might report usage statistics at regular intervals.

Some of these messages need to be brought to a system administrator's attention immediately. And it may not be just any system administrator—there may be a particular system administrator who deals with a particular kind of message. Other messages just need to be recorded for future reference if there is a problem. Still others may need to have information extracted from them by an automated process that generates monthly reports.

To deal with these messages, most Unix systems have a facility called "Syslog." It is generally based on a daemon called "Syslogd" Syslogd listens for messages on a Unix domain socket named '`/dev/log`'. Based on classification information in the messages and its configuration file (usually '`/etc/syslog.conf`'), Syslogd routes them in various ways. Some of the popular routings are

- Write to the system console.
- Mail to a specific user.
- Write to a log file.
- Pass to another daemon.
- Discard.

Syslogd can also handle messages from other systems. It listens on the `syslog` UDP port as well as the local socket for messages.

Syslog can handle messages from the kernel itself. But the kernel doesn't write to '`/dev/log`'; rather, another daemon (sometimes called "Klogd") extracts messages from the kernel and passes them on to Syslog as any other process would (and it properly identifies them as messages from the kernel).

[1] You would typically do that with the facilities described in Loosemore et al., "Input/Output on Streams" (see chap. 1, n. 1).

Syslog can even handle messages that the kernel issued before Syslogd or Klogd was running. A Linux kernel, for example, stores start-up messages in a kernel message ring and they are normally still there when Klogd later starts up. Assuming Syslogd is running by the time Klogd starts, Klogd then passes everything in the message ring to it.

In order to classify messages for disposition, Syslog requires any process that submits a message to it to provide two pieces of classification information with it:

- *facility* This identifies who submitted the message. There are a small number of facilities defined. The kernel, the mail subsystem, and an FTP server are examples of recognized facilities. For the complete list, see Section 15.2.2 [syslog, vsyslog], page 362. Keep in mind that these are essentially arbitrary classifications. "Mail subsystem" doesn't have any more meaning than the system administrator gives to it.
- *priority* This tells how important the content of the message is. Examples of defined priority values are: debug, informational, warning, critical. For the complete list, see Section 15.2.2 [syslog, vsyslog], page 362. Except for the fact that the priorities have a defined order, the meaning of each of these priorities is entirely determined by the system administrator.

A *facility/priority* is a number that indicates both the facility and the priority.

This terminology is not universal. Some people use *level* to refer to the priority and *priority* to refer to the combination of facility and priority. A Linux kernel has a concept of a message *level*, which corresponds both to a Syslog priority and to a Syslog facility/priority (It can be both because the facility code for the kernel is 0, and that makes priority and facility/priority the same value).

The GNU C Library provides functions to submit messages to Syslog. They do it by writing to the '`/dev/log`' socket (see Section 15.2 [Submitting Syslog Messages], page 360).

The GNU C Library functions only work to submit messages to the Syslog facility on the same system. To submit a message to the Syslog facility on another system, use the socket I/O functions to write a UDP datagram to the `syslog` UDP port on that system (see Chapter 5 [Sockets], page 125).

15.2 Submitting Syslog Messages

The GNU C Library provides functions to submit messages to the Syslog facility.

These functions only work to submit messages to the Syslog facility on the same system. To submit a message to the Syslog facility on another system, use the socket I/O functions to write a UDP datagram to the `syslog` UDP port on that system (see Chapter 5 [Sockets], page 125).

15.2.1 openlog

The symbols referred to in this section are declared in the file '`syslog.h`'.

`void` **openlog** (`const char *`*ident*`, int` *option*`, int` *facility*) Function

`openlog` opens or reopens a connection to Syslog in preparation for submitting messages.

ident is an arbitrary identification string that future `syslog` invocations will prefix to each message. This is intended to identify the source of the message, and people conventionally set it to the name of the program that will submit the messages.

If *ident* is NULL, or if `openlog` is not called, the default identification string used in Syslog messages will be the program name, taken from argv[0].

Please note that the string pointer *ident* will be retained internally by the Syslog routines. You must not free the memory that *ident* points to. It is also dangerous to pass a reference to an automatic variable since leaving the scope would mean ending the lifetime of the variable. If you want to change the *ident* string, you must call `openlog` again; overwriting the string pointed to by *ident* is not threadsafe.

You can cause the Syslog routines to drop the reference to *ident* and go back to the default string (the program name taken from argv[0]), by calling `closelog` (see Section 15.2.3 [closelog], page 365).

In particular, if you are writing code for a shared library that might get loaded and then unloaded (like a PAM module), and you use `openlog`, you must call `closelog` before any point where your library might get unloaded, as in this example:

```
#include <syslog.h>

void
shared_library_function (void)
{
  openlog ("mylibrary", option, priority);

  syslog (LOG_INFO, "shared library has been invoked");

  closelog ();
}
```

Without the call to `closelog`, future invocations of `syslog` by the program using the shared library may crash, if the library gets unloaded and the memory containing the string `"mylibrary"` becomes unmapped. This is a limitation of the BSD Syslog interface.

`openlog` may or may not open the '`/dev/log`' socket, depending on *option*. If it does, it tries to open it and connect it as a stream socket. If that doesn't work, it tries to open it and connect it as a datagram socket. The socket has the "Close on Exec" attribute, so the kernel will close it if the process performs an exec.

You don't have to use `openlog`. If you call `syslog` without having called `openlog`, `syslog` just opens the connection implicitly and uses defaults for the information in *ident* and *options*.

options is a bit string, with the bits as defined by the following single-bit masks:

`LOG_PERROR`
: If on, `openlog` sets up the connection so that any `syslog` on this connection writes its message to the calling process's standard error stream, in addition to submitting it to Syslog. If off, `syslog` does not write the message to standard error.

`LOG_CONS`
: If on, `openlog` sets up the connection so that a `syslog` on this connection that fails to submit a message to Syslog writes the message instead to system console. If off, `syslog` does not write to the system console (but of course Syslog may write messages it receives to the console).

`LOG_PID`
: When on, `openlog` sets up the connection so that a `syslog` on this connection inserts the calling process's process ID (PID) into the message. When off, `openlog` does not insert the PID.

`LOG_NDELAY`
: When on, `openlog` opens and connects the '`/dev/log`' socket. When off, a future `syslog` call must open and connect the socket.

  Portability note: In early systems, the sense of this bit was exactly the opposite.

`LOG_ODELAY`
: This bit does nothing. It exists for backward compatibility.

If any other bit in *options* is on, the result is undefined.

facility is the default facility-code for this connection. A `syslog` on this connection that specifies default facility causes this facility to be associated with the message. See `syslog` for possible values. A value of 0 means the default default, which is `LOG_USER`.

If a Syslog connection is already open when you call `openlog`, `openlog` "reopens" the connection. Reopening is like opening except that if you specify 0 for the default facility code, the default facility code simply remains unchanged and if you specify LOG_NDELAY and the socket is already open and connected, `openlog` just leaves it that way.

15.2.2 syslog, vsyslog

The symbols referred to in this section are declared in the file '`syslog.h`'.

`void` **syslog** (`int` *facility_priority*`, char *`*format*`, ...`) Function

`syslog` submits a message to the Syslog facility. It does this by writing to the Unix domain socket '`/dev/log`'.

`syslog` submits the message with the facility and priority indicated by *facility_priority*. The macro `LOG_MAKEPRI` generates a facility/priority from a facility and a priority, as in the following example:

```
LOG_MAKEPRI(LOG_USER, LOG_WARNING)
```

The possible values for the facility code are (macros):

`LOG_USER`
: This is a miscellaneous user process.

`LOG_MAIL`
: This is mail.

`LOG_DAEMON`
: This is a miscellaneous system daemon.

`LOG_AUTH`
: This is security (authorization).

`LOG_SYSLOG`
: This is syslog.

`LOG_LPR` This is the central printer.

`LOG_NEWS`
: This is network news (e.g. Usenet).

`LOG_UUCP`
: This is UUCP.

`LOG_CRON`
: This is cron and At.

`LOG_AUTHPRIV`
: This is private security (authorization).

`LOG_FTP` This is an ftp server.

`LOG_LOCAL0`
: This is locally defined.

`LOG_LOCAL1`
: This is locally defined.

`LOG_LOCAL2`
: This is locally defined.

`LOG_LOCAL3`
: This is locally defined.

`LOG_LOCAL4`
: This is locally defined.

`LOG_LOCAL5`
: This is locally defined.

`LOG_LOCAL6`
: This is locally defined.

`LOG_LOCAL7`
: This is locally defined.

Results are undefined if the facility code is anything else.

`syslog` recognizes one other facility code—that of the kernel. But you can't specify that facility code with these functions. If you try, it looks the same to `syslog` as if you are requesting the default facility. But you wouldn't want to anyway, because any program that uses the GNU C Library is not the kernel.

You can use just a priority code as *facility_priority*. In that case, `syslog` assumes the default facility established when the Syslog connection was opened (see Section 15.2.5 [Syslog Example], page 366).

The possible values for the priority code are (macros):

`LOG_EMERG`
: The message says the system is unusable.

`LOG_ALERT`
: Action on the message must be taken immediately.

`LOG_CRIT`
: The message states a critical condition.

`LOG_ERR` The message describes an error.

`LOG_WARNING`
: The message is a warning.

`LOG_NOTICE`
: The message describes a normal but important event.

`LOG_INFO`
: The message is purely informational.

`LOG_DEBUG`
: The message is only for debugging purposes.

Results are undefined if the priority code is anything else.

If the process does not presently have a Syslog connection open (i.e. it did not call `openlog`), `syslog` implicitly opens the connection the same as `openlog` would, with the defaults for information that would otherwise be included in an `openlog` call. The default identification-string is the program name. The default default facility is `LOG_USER`. The default for all the connection options in *options* is as if those bits were off. `syslog` leaves the Syslog connection open.

If the ‘dev/log’ socket is not open and connected, syslog opens and connects it, the same as openlog with the LOG_NDELAY option would.

syslog leaves ‘/dev/log’ open and connected unless its attempt to send the message failed, in which case syslog closes it, with the hope that a future implicit open will restore the Syslog connection to a usable state.

Here is an example:

```
#include <syslog.h>
syslog (LOG_MAKEPRI(LOG_LOCAL1, LOG_ERROR),
        "Unable to make network connection to %s.  Error=%m", host);
```

`void` **vsyslog** (`int` *facility_priority*, `char *`*format*, `va_list` `arglist`) Function

This is functionally identical to syslog, with the BSD-style variable length argument.

15.2.3 closelog

The symbols referred to in this section are declared in the file ‘syslog.h’.

`void` **closelog** (`void`) Function

closelog closes the current Syslog connection, if there is one. This includes closing the ‘dev/log’ socket, if it is open. closelog also sets the identification string for Syslog messages back to the default, if openlog was called with a non-NULL argument to *ident*. The default identification string is the program name taken from argv[0].

If you are writing shared library code that uses openlog to generate custom syslog output, you should use closelog to drop the GNU C Library's internal reference to the *ident* pointer when you are done (see Section 15.2.1 [openlog], page 360).

closelog does not flush any buffers. You do not have to call closelog before reopening a Syslog connection with initlog. Syslog connections are automatically closed on exec or exit.

15.2.4 setlogmask

The symbols referred to in this section are declared in the file ‘syslog.h’.

`int` **setlogmask** (`int` *mask*) Function

setlogmask sets a mask (the “logmask”) that determines which future syslog calls should be ignored. If a program has not called setlogmask, syslog doesn't ignore any calls. You can use setlogmask to specify that messages of particular priorities shall be ignored in the future.

A `setlogmask` call overrides any previous `setlogmask` call.

Note that the logmask exists entirely independently of opening and closing of Syslog connections.

Setting the logmask has a similar effect to, but is not the same as, configuring Syslog. The Syslog configuration may cause Syslog to discard certain messages it receives, but the logmask causes certain messages never to get submitted to Syslog in the first place.

mask is a bit string with 1 bit corresponding to each of the possible message priorities. If the bit is on, `syslog` handles messages of that priority normally. If it is off, `syslog` discards messages of that priority. Use the message priority macros described in Section 15.2.2 [syslog, vsyslog], page 362 and the `LOG_MASK` to construct an appropriate *mask* value, as in this example:

```
LOG_MASK(LOG_EMERG) | LOG_MASK(LOG_ERROR)
```

or:

```
~(LOG_MASK(LOG_INFO))
```

There is also a `LOG_UPTO` macro, which generates a mask with the bits on for a certain priority and all priorities above it:

```
LOG_UPTO(LOG_ERROR)
```

The unfortunate naming of the macro is due to the fact that internally, higher numbers are used for lower message priorities.

15.2.5 Syslog Example

Here is an example of `openlog`, `syslog` and `closelog`:

This example sets the logmask so that debug and informational messages get discarded without ever reaching Syslog. So the second `syslog` in the example does nothing.

```
#include <syslog.h>

setlogmask (LOG_UPTO (LOG_NOTICE));

openlog ("exampleprog", LOG_CONS | LOG_PID | LOG_NDELAY, LOG_LOCAL1);

syslog (LOG_NOTICE, "Program started by User %d", getuid ());
syslog (LOG_INFO, "A tree falls in a forest");

closelog ();
```

16 Nonlocal Exits

Sometimes when your program detects an unusual situation inside a deeply nested set of function calls, you would like to be able to immediately return to an outer level of control. This section describes how you can do such *nonlocal exits* using the `setjmp` and `longjmp` functions.

16.1 Introduction to Nonlocal Exits

As an example of a situation where a nonlocal exit can be useful, suppose you have an interactive program that has a "main loop" that prompts for and executes commands. Suppose the "read" command reads input from a file, doing some lexical analysis and parsing of the input while processing it. If a low-level input error is detected, it would be useful to be able to return immediately to the "main loop" instead of having to make each of the lexical analysis, parsing and processing phases all have to explicitly deal with error situations initially detected by nested calls.

On the other hand, if each of these phases has to do a substantial amount of clean-up when it exits—such as closing files, deallocating buffers or other data structures, etc.—then it can be more appropriate to do a normal return and have each phase do its own clean-up, because a nonlocal exit would bypass the intervening phases and their associated clean-up code entirely. Alternatively, you could use a nonlocal exit but do the clean-up explicitly either before or after returning to the "main loop".

In some ways, a nonlocal exit is similar to using the '`return`' statement to return from a function. But while '`return`' abandons only a single function call, transferring control back to the point at which it was called, a nonlocal exit can potentially abandon many levels of nested function calls.

You identify return points for nonlocal exits by calling the function `setjmp`. This function saves information about the execution environment in which the call to `setjmp` appears in an object of type `jmp_buf`. Execution of the program continues normally after the call to `setjmp`, but if an exit is later made to this return point by calling `longjmp` with the corresponding `jmp_buf` object, control is transferred back to the point where `setjmp` was called. The return value from `setjmp` is used to distinguish between an ordinary return and a return made by a call to `longjmp`, so calls to `setjmp` usually appear in an '`if`' statement.

Here is how the example program described above might be set up:

```
#include <setjmp.h>
#include <stdlib.h>
#include <stdio.h>

jmp_buf main_loop;

void
abort_to_main_loop (int status)
```

```
{
  longjmp (main_loop, status);
}

int
main (void)
{
  while (1)
    if (setjmp (main_loop))
      puts ("Back at main loop....");
    else
      do_command ();
}


void
do_command (void)
{
  char buffer[128];
  if (fgets (buffer, 128, stdin) == NULL)
    abort_to_main_loop (-1);
  else
    exit (EXIT_SUCCESS);
}
```

The function `abort_to_main_loop` causes an immediate transfer of control back to the main loop of the program, no matter where it is called from.

The flow of control inside the `main` function may appear a little mysterious at first, but it is actually a common idiom with `setjmp`. A normal call to `setjmp` returns 0, so the "else" clause of the conditional is executed. If `abort_to_main_loop` is called somewhere within the execution of `do_command`, then it actually appears as if the *same* call to `setjmp` in `main` were returning a second time with a value of `-1`.

So, the general pattern for using `setjmp` looks something like:

```
if (setjmp (buffer))
  /* Code to clean up after premature return. */
  ...
else
  /* Code to be executed normally after setting up the return point. */
  ...
```

16.2 Details of Nonlocal Exits

Here are the details on the functions and data structures used for performing nonlocal exits. These facilities are declared in '`setjmp.h`'.

jmp_buf Data Type

Objects of type `jmp_buf` hold the state information to be restored by a nonlocal exit. The contents of a `jmp_buf` identify a specific place to return to.

`int` **setjmp** (`jmp_buf` *state*) Macro

When called normally, `setjmp` stores information about the execution state of the program in *state* and returns 0. If `longjmp` is later used to perform a nonlocal exit to this *state*, `setjmp` returns a nonzero value.

`void` **longjmp** (`jmp_buf` *state*, `int` *value*) Function

This function restores the current execution to the state saved in *state*, and continues execution from the call to `setjmp` that established that return point. Returning from `setjmp` by means of `longjmp` returns the *value* argument that was passed to `longjmp`, rather than `0`. (But if *value* is given as `0`, `setjmp` returns `1`.

There are a lot of obscure but important restrictions on the use of `setjmp` and `longjmp`. Most of these restrictions are present because nonlocal exits require a fair amount of magic on the part of the C compiler and can interact with other parts of the language in strange ways.

The `setjmp` function is actually a macro without an actual function definition, so you shouldn't try to '`#undef`' it or take its address. In addition, calls to `setjmp` are safe in only the following contexts:

- As the test expression of a selection or iteration statement (such as '`if`', '`switch`' or '`while`')
- As one operand of an equality or comparison operator that appears as the test expression of a selection or iteration statement—the other operand must be an integer constant expression
- As the operand of a unary '`!`' operator, that appears as the test expression of a selection or iteration statement
- By itself as an expression statement

Return points are valid only during the dynamic extent of the function that called `setjmp` to establish them. If you `longjmp` to a return point that was established in a function that has already returned, unpredictable and disastrous things are likely to happen.

You should use a nonzero *value* argument to `longjmp`. While `longjmp` refuses to pass back a zero argument as the return value from `setjmp`, this is intended as a safety net against accidental misuse and is not really good programming style.

When you perform a nonlocal exit, accessible objects generally retain whatever values they had at the time `longjmp` was called. The exception is that the values of automatic variables local to the function containing the `setjmp` call that have been changed since the call to `setjmp` are indeterminate, unless you have declared them `volatile`.

16.3 Nonlocal Exits and Signals

In BSD Unix systems, `setjmp` and `longjmp` also save and restore the set of blocked signals (see Section 17.7 [Blocking Signals], page 414). However, the POSIX.1 standard requires `setjmp` and `longjmp` not to change the set of blocked signals, and provides an additional pair of functions (`sigsetjmp` and `siglongjmp`) to get the BSD behavior.

The behavior of `setjmp` and `longjmp` in the GNU library is controlled by feature-test macros (see Section 1.3.4 [Feature-Test Macros], page 8). The default in the GNU system is the POSIX.1 behavior rather than the BSD behavior.

The facilities in this section are declared in the header file '`setjmp.h`'.

sigjmp_buf Data Type
This is similar to `jmp_buf`, except that it can also store state information about the set of blocked signals.

`int` **sigsetjmp** (`sigjmp_buf` *state*, `int` *savesigs*) Function
This is similar to `setjmp`. If *savesigs* is nonzero, the set of blocked signals is saved in *state* and will be restored if a `siglongjmp` is later performed with this *state*.

`void` **siglongjmp** (`sigjmp_buf` *state*, `int` *value*) Function
This is similar to `longjmp` except for the type of its *state* argument. If the `sigsetjmp` call that set this *state* used a nonzero *savesigs* flag, `siglongjmp` also restores the set of blocked signals.

16.4 Complete Context Control

The Unix standard provides one more set of functions to control the execution path, and these functions are more powerful than those discussed in this chapter so far. These function were part of the original System V API and by this route were added to the Unix API. Besides on branded Unix implementations, these interfaces are not widely available. Not all platforms and/or architectures the GNU C Library is available on provide this interface. Use '`configure`' to detect the availability.

Similar to the `jmp_buf` and `sigjmp_buf` types used for the variables to contain the state of the `longjmp` functions, the interfaces of interest here have an appropriate type as well. Objects of this type are normally much larger since more information is contained. The type is also used in a few more places, as we will

see. The types and functions described in this section are all defined and declared respectively in the 'ucontext.h' header file.

ucontext_t Data Type

The ucontext_t type is defined as a structure with as least the following elements:

ucontext_t *uc_link
: This is a pointer to the next context structure that is used if the context described in the current structure returns.

sigset_t uc_sigmask
: This is the set of signals that are blocked when this context is used.

stack_t uc_stack
: This is the stack used for this context. The value need not be (and normally is not) the stack pointer (see Section 17.9 [Using a Separate Signal-Stack], page 424).

mcontext_t uc_mcontext
: This element contains the actual state of the process. The mcontext_t type is also defined in this header but the definition should be treated as opaque. Any use of knowledge of the type makes applications less portable.

Objects of this type have to be created by the user. The initialization and modification happens through one of the following functions:

int **getcontext** (ucontext_t *ucp) Function

The getcontext function initializes the variable pointed to by *ucp* with the context of the calling thread. The context contains the content of the registers, the signal mask, and the current stack. Executing the contents would start at the point where the getcontext call just returned.

The function returns 0 if successful. Otherwise, it returns -1 and sets *errno* accordingly.

The getcontext function is similar to setjmp, but it does not provide an indication of whether the function returns for the first time or whether the initialized context was used, and the execution is resumed at just that point. If this is necessary, the user has to determine this herself. This must be done carefully, since the context contains registers that might contain register variables. This is a good situation in which to define variables with volatile.

Once the context variable is initialized, it can be used as is or it can be modified. The latter is normally done to implement co-routines or similar constructs. The makecontext function is what has to be used to do that.

`void` **makecontext** (`ucontext_t *`*ucp*`, void (*`*func*`) (void), int` *argc*`, ...`) Function

The *ucp* parameter passed to `makecontext` should be initialized by a call to `getcontext`. The context will be modified so that if the context is resumed, it will start by calling the function `func`, which gets *argc* integer arguments passed. The integer arguments that are to be passed should follow the *argc* parameter in the call to `makecontext`.

Before the call to this function, the `uc_stack` and `uc_link` element of the *ucp* structure should be initialized. The `uc_stack` element describes the stack that is used for this context. No two contexts that are used at the same time should use the same memory region for a stack.

The `uc_link` element of the object pointed to by *ucp* should be a pointer to the context to be executed when the function *func* returns, or it should be a null pointer. See `setcontext` for more information about the exact use.

While allocating the memory for the stack, you have to be careful. Most modern processors keep track of whether a certain memory region is allowed to contain code that is executed or not. Data segments and heap memory are normally not tagged to allow this. The result is that programs would fail. Examples for such code include the calling sequences the GNU C Compiler generates for calls to nested functions. Safe ways to allocate stacks correctly include using memory on the original threads stack or explicitly allocating memory tagged for execution using memory-mapped I/O (see Section 2.7 [Memory-Mapped I/O], page 32).

Compatibility Note: The current Unix standard is very imprecise about the way the stack is allocated. All implementations seem to agree that the `uc_stack` element must be used but the values stored in the elements of the `stack_t` value are unclear. The GNU C Library and most other Unix implementations require the `ss_sp` value of the `uc_stack` element to point to the base of the memory region allocated for the stack and the size of the memory region to be stored in `ss_size`. There are implementations out there that require `ss_sp` to be set to the value the stack pointer will have (which can be different depending on the direction the stack grows). This difference makes the `makecontext` function hard to use, and it requires detection of the platform at compile time.

`int` **setcontext** (`const ucontext_t *`*ucp*) Function

The `setcontext` function restores the context described by *ucp*. The context is not modified and can be reused as often as wanted.

If the context was created by `getcontext`, execution resumes with the registers filled with the same values and the same stack as if the `getcontext` call just returned.

If the context was modified with a call to `makecontext`, execution continues with the function passed to `makecontext`, which gets the specified parameters passed. If this function returns, execution is resumed in the context that was referenced by the `uc_link` element of the context structure passed to `makecontext` at the time of the call. If `uc_link` was a null pointer, the application terminates in this case.

Since the context contains information about the stack, no two threads should use the same context at the same time. The result in most cases would be disastrous.

The `setcontext` function does not return unless an error occurred, in which case it returns `-1`.

The `setcontext` function simply replaces the current context with the one described by the *ucp* parameter. This is often useful, but there are situations where the current context has to be preserved.

`int` **swapcontext** (`ucontext_t *restrict` *oucp*`, const ucontext_t *restrict` *ucp*) Function

The `swapcontext` function is similar to `setcontext`, but instead of just replacing the current context, the latter is first saved in the object pointed to by *oucp* as if this were a call to `getcontext`. The saved context would resume after the call to `swapcontext`.

Once the current context is saved, the context described in *ucp* is installed, and execution continues as described in this context.

If `swapcontext` succeeds, the function does not return unless the context *oucp* is used without prior modification by `makecontext`. The return value in this case is `0`. If the function fails it returns `-1` and set *errno* accordingly.

Example for SVID Context-Handling

The easiest way to use the context-handling functions is as a replacement for `setjmp` and `longjmp`. The context contains on most platforms more information, which might lead to less surprises, but this also means using these functions is more expensive (besides being less portable).

```
int
random_search (int n, int (*fp) (int, ucontext_t *))
{
  volatile int cnt = 0;
  ucontext_t uc;

  /* Safe current context.  */
  if (getcontext (&uc) < 0)
    return -1;

  /* If we have not tried n times try again.  */
  if (cnt++ < n)
    /* Call the function with a new random number
       and the context.  */
    if (fp (rand (), &uc) != 0)
      /* We found what we were looking for.  */
```

```
      return 1;

  /* Not found  */
  return 0;
}
```

Using contexts in such a way enables emulating exception handling. The search functions passed in the *fp* parameter could be very large, nested, and complex, which would make it complicated (or at least would require a lot of code) to leave the function with an error value that has to be passed down to the caller. By using the context, it is possible to leave the search function in one step and allow restarting the search, which also has the nice side effect that it can be significantly faster.

Something that is harder to implement with `setjmp` and `longjmp` is to switch temporarily to a different execution path and then resume where execution was stopped.

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <ucontext.h>
#include <sys/time.h>

/* Set by the signal handler */
static volatile int expired;

/* The contexts */
static ucontext_t uc[3];

/* We do only a certain number of switches. */
static int switches;


/* This is the function doing the work.  It is just a
  skeleton; real code has to be filled in. */
static void
f (int n)
{
  int m = 0;
  while (1)
    {
      /* This is where the work would be done. */
      if (++m % 100 == 0)
        {
          putchar ('.');
          fflush (stdout);
        }
```

```
      /* The expire variable must be checked regularly. */
      if (expired)
        {
          /* We do not want the program to run forever. */
          if (++switches == 20)
            return;

          printf ("\nswitching from %d to %d\n", n, 3 - n);
          expired = 0;
          /* Switch to the other context, saving the current one. */
          swapcontext (&uc[n], &uc[3 - n]);
        }
    }
}

/* This is the signal handler that simply sets the variable. */
void
handler (int signal)
{
  expired = 1;
}


int
main (void)
{
  struct sigaction sa;
  struct itimerval it;
  char st1[8192];
  char st2[8192];

  /* Initialize the data structures for the interval timer. */
  sa.sa_flags = SA_RESTART;
  sigfillset (&sa.sa_mask);
  sa.sa_handler = handler;
  it.it_interval.tv_sec = 0;
  it.it_interval.tv_usec = 1;
  it.it_value = it.it_interval;

  /* Install the timer and get the context we can manipulate. */
  if (sigaction (SIGPROF, &sa, NULL) < 0
      || setitimer (ITIMER_PROF, &it, NULL) < 0
      || getcontext (&uc[1]) == -1
```

```
    || getcontext (&uc[2]) == -1)
  abort ();

/* Create a context with a separate stack that causes the
function f to be called with the parameter 1.
The uc_link points to the main context,
which will cause the program to terminate once the function
returns. */
uc[1].uc_link = &uc[0];
uc[1].uc_stack.ss_sp = st1;
uc[1].uc_stack.ss_size = sizeof st1;
makecontext (&uc[1], (void (*) (void)) f, 1, 1);

/* This is similar to the above, but 2 is passed as the parameter to f. */
uc[2].uc_link = &uc[0];
uc[2].uc_stack.ss_sp = st2;
uc[2].uc_stack.ss_size = sizeof st2;
makecontext (&uc[2], (void (*) (void)) f, 1, 2);

/* Start running. */
swapcontext (&uc[0], &uc[1]);
putchar ('\n');

return 0;
}
```

This an example how the context functions can be used to implement coroutines or cooperative multithreading. All that has to be done is to call `swapcontext` every once in a while to continue running a different context. It is not permissible to do the context switching from the signal handler directly, since neither `setcontext` nor `swapcontext` are functions that can be called from a signal handler. But setting a variable in the signal handler and checking it in the body of the functions which are executed. Since `swapcontext` is saving the current context, it is possible to have multiple different scheduling-points in the code. Execution will always resume where it was left.

17 Signal Handling

A *signal* is a software interrupt delivered to a process. The operating system uses signals to report exceptional situations to an executing program. Some signals report errors such as references to invalid memory-addresses; others report asynchronous events, such as disconnection of a phone line.

The GNU C Library defines a variety of signal types, each for a particular kind of event. Some kinds of events make it inadvisable or impossible for the program to proceed as usual, and the corresponding signals normally abort the program. Other kinds of signals that report harmless events are ignored by default.

If you anticipate an event that causes signals, you can define a handler function and tell the operating system to run it when that particular type of signal arrives.

Finally, one process can send a signal to another process; this allows a parent process to abort a child, or two related processes to communicate and synchronize.

17.1 Basic Concepts of Signals

This section explains basic concepts of how signals are generated, what happens after a signal is delivered, and how programs can handle signals.

17.1.1 Some Kinds of Signals

A signal reports the occurrence of an exceptional event. These are some of the events that can cause (or *generate*, or *raise*) a signal:

- A program error such as dividing by 0 or issuing an address outside the valid range
- A user request to interrupt or terminate the program.; most environments are set up to let a user suspend the program by typing `C-z`, or terminate it with `C-c`. Whatever key sequence is used, the operating system sends the proper signal to interrupt the process.
- The termination of a child process
- Expiration of a timer or alarm
- A call to `kill` or `raise` by the same process
- A call to `kill` from another process; signals are a limited but useful form of interprocess communication.
- An attempt to perform an I/O operation that cannot be done; examples are reading from a pipe that has no writer (see Chapter 4 [Pipes and FIFOs], page 119), and reading or writing to a terminal in certain situations (see Chapter 8 [Job Control], page 221).

Each of these kinds of events (excepting explicit calls to `kill` and `raise`) generates its own particular kind of signal. The various kinds of signals are listed and described in detail in Section 17.2 [Standard Signals], page 379.

17.1.2 Concepts of Signal Generation

In general, the events that generate signals fall into three major categories: errors, external events and explicit requests.

An error means that a program has done something invalid and cannot continue execution. But not all kinds of errors generate signals—in fact, most do not. For example, opening a nonexistent file is an error, but it does not raise a signal; instead, `open` returns `-1`. In general, errors that are necessarily associated with certain library functions are reported by returning a value that indicates an error. The errors that raise signals are those that can happen anywhere in the program, not just in library calls. These include division by 0 and invalid memory-addresses.

An external event generally has to do with I/O or other processes. These include the arrival of input, the expiration of a timer and the termination of a child process.

An explicit request means the use of a library function such as `kill` whose purpose is specifically to generate a signal.

Signals may be generated *synchronously* or *asynchronously*. A synchronous signal pertains to a specific action in the program, and is delivered (unless blocked) during that action. Most errors generate signals synchronously, and so do explicit requests by a process to generate a signal for that same process. On some machines, certain kinds of hardware errors (usually floating-point exceptions) are not reported completely synchronously, but may arrive a few instructions later.

Asynchronous signals are generated by events outside the control of the process that receives them. These signals arrive at unpredictable times during execution. External events generate signals asynchronously, and so do explicit requests that apply to some other process.

A given type of signal is either typically synchronous or typically asynchronous. For example, signals for errors are typically synchronous because errors generate signals synchronously. But any type of signal can be generated synchronously or asynchronously with an explicit request.

17.1.3 How Signals Are Delivered

When a signal is generated, it becomes *pending*. Normally, it remains pending for just a short period of time and then is *delivered* to the process that was signaled. However, if that kind of signal is currently *blocked*, it may remain pending indefinitely—until signals of that kind are *unblocked*. Once unblocked, it will be delivered immediately (see Section 17.7 [Blocking Signals], page 414).

When the signal is delivered, whether right away or after a long delay, the *specified action* for that signal is taken. For certain signals, such as `SIGKILL` and `SIGSTOP`, the action is fixed, but for most signals, the program has a choice: ignore the signal, specify a *handler function* or accept the *default action* for that kind of signal. The program specifies its choice using functions such as `signal` or `sigaction` (see Section 17.3 [Specifying Signal Actions], page 389). We sometimes say that a handler *catches* the signal. While the handler is running, that particular signal is normally blocked.

If the specified action for a kind of signal is to ignore it, then any such signal that is generated is discarded immediately. This happens even if the signal is also blocked at the time. A signal discarded in this way will never be delivered, not even if the program subsequently specifies a different action for that kind of signal and then unblocks it.

If a signal arrives that the program has neither handled nor ignored, its *default action* takes place. Each kind of signal has its own default action, documented below (see Section 17.2 [Standard Signals], page 379). For most kinds of signals, the default action is to terminate the process. For certain kinds of signals that represent "harmless" events, the default action is to do nothing.

When a signal terminates a process, its parent process can determine the cause of termination by examining the termination status-code reported by the `wait` or `waitpid` functions. (This is discussed in more detail in Section 7.6 [Process Completion], page 215.) The information it can get includes the fact that termination was due to a signal and the kind of signal involved. If a program you run from a shell is terminated by a signal, the shell typically prints some kind of error message.

The signals that normally represent program errors have a special property: when one of these signals terminates the process, it also writes a *core-dump file* that records the state of the process at the time of termination. You can examine the core dump with a debugger to investigate what caused the error.

If you raise a "program error" signal by explicit request, and this terminates the process, it makes a core-dump file just as if the signal had been due directly to an error.

17.2 Standard Signals

This section lists the names for various standard kinds of signals and describes what kind of event they mean. Each signal name is a macro that stands for a positive integer—the *signal number* for that kind of signal. Your programs should never make assumptions about the numeric code for a particular kind of signal, but rather refer to them always by the names defined here. This is because the number for a given kind of signal can vary from system to system, but the meanings of the names are standardized and fairly uniform.

The signal names are defined in the header file '`signal.h`'.

`int` **NSIG** Macro

The value of this symbolic constant is the total number of signals defined. Since the signal numbers are allocated consecutively, `NSIG` is also 1 greater than the largest defined signal-number.

17.2.1 Program-Error Signals

The following signals are generated when a serious program error is detected by the operating system or the computer itself. In general, all of these signals are

indications that your program is seriously broken in some way, and there's usually no way to continue the computation that encountered the error.

Some programs handle program-error signals in order to tidy up before terminating; for example, programs that turn off echoing of terminal input should handle program-error signals in order to turn echoing back on. The handler should end by specifying the default action for the signal that happened and then reraising it; this will cause the program to terminate with that signal, as if it had not had a handler (see Section 17.4.2 [Handlers That Terminate the Process], page 398).

Termination is the sensible ultimate outcome from a program error in most programs. However, programming systems such as Lisp that can load compiled user-programs might need to keep executing even if a user program incurs an error. These programs have handlers that use `longjmp` to return control to the command level.

The default action for all of these signals is to cause the process to terminate. If you block or ignore these signals or establish handlers for them that return normally, your program will probably break horribly when such signals happen, unless they are generated by `raise` or `kill` instead of a real error.

When one of these program-error signals terminates a process, it also writes a *core-dump file* that records the state of the process at the time of termination. The core dump file is named '`core`' and is written in whichever directory is current in the process at the time. On the GNU system, you can specify the file name for core dumps with the environment variable `COREFILE`. The purpose of core-dump files is so that you can examine them with a debugger to investigate what caused the error.

`int` **SIGFPE** Macro

The `SIGFPE` signal reports a fatal arithmetic error. Although the name is derived from "floating-point exception", this signal actually covers all arithmetic errors, including division by 0 and overflow. If a program stores integer data in a location that is then used in a floating-point operation, this often causes an "invalid operation" exception, because the processor cannot recognize the data as a floating-point number.

Actual floating-point exceptions are a complicated subject because there are many types of exceptions with subtly different meanings, and the `SIGFPE` signal doesn't distinguish between them. The *IEEE Standard for Binary Floating-Point Arithmetic (*ANSI/IEEE *Std 754-1985 and* ANSI/IEEE *Std 854-1987)* defines various floating-point exceptions and requires conforming computer systems to report their occurrences. However, this standard does not specify how the exceptions are reported, or what kinds of handling and control the operating system can offer to the programmer.

BSD systems provide the `SIGFPE` handler with an extra argument that distinguishes various causes of the exception. In order to access this argument, you must define the handler to accept two arguments, which means you must cast it to a one-argument function type in order to establish the handler. The GNU library does

provide this extra argument, but the value is meaningful only on operating systems that provide the information (BSD systems and GNU systems).

`FPE_INTOVF_TRAP`
: Integer overflow (impossible in a C program unless you enable overflow trapping in a hardware-specific fashion)

`FPE_INTDIV_TRAP`
: Integer division by 0

`FPE_SUBRNG_TRAP`
: Subscript range (something that C programs never check for)

`FPE_FLTOVF_TRAP`
: Floating overflow trap

`FPE_FLTDIV_TRAP`
: Floating/decimal division by 0.

`FPE_FLTUND_TRAP`
: Floating underflow trap; trapping on floating underflow is not normally enabled.

`FPE_DECOVF_TRAP`
: Decimal overflow trap; only a few machines have decimal arithmetic, and C never uses it.

`int` **SIGILL** Macro

The name of this signal is derived from "illegal instruction"; it usually means your program is trying to execute garbage or a privileged instruction. Since the C compiler generates only valid instructions, `SIGILL` typically indicates that the executable file is corrupted, or that you are trying to execute data. Some common ways of getting into the latter situation are by passing an invalid object where a pointer to a function was expected, or by writing past the end of an automatic array (or similar problems with pointers to automatic variables) and corrupting other data on the stack such as the return address of a stack frame.

`SIGILL` can also be generated when the stack overflows, or when the system has trouble running the handler for a signal.

`int` **SIGSEGV** Macro

This signal is generated when a program tries to read or write outside the memory that is allocated for it, or to write memory that can only be read. Actually, the signals only occur when the program goes far enough outside to be detected by the system's memory protection mechanism. The name is an abbreviation for "segmentation violation".

Common ways of getting a `SIGSEGV` condition include dereferencing a null or uninitialized pointer, or when you use a pointer to step through an array, but fail to check for the end of the array. It varies among systems whether dereferencing a null pointer generates `SIGSEGV` or `SIGBUS`.

`int` **SIGBUS** Macro

This signal is generated when an invalid pointer is dereferenced. Like `SIGSEGV`, this signal is typically the result of dereferencing an uninitialized pointer. The difference between the two is that `SIGSEGV` indicates an invalid access to valid memory, while `SIGBUS` indicates an access to an invalid address. In particular, `SIGBUS` signals often result from dereferencing a misaligned pointer, such as referring to a four-word integer at an address not divisible by 4. Each kind of computer has its own requirements for address alignment.

The name of this signal is an abbreviation for "bus error".

`int` **SIGABRT** Macro

This signal indicates an error detected by the program itself and reported by calling `abort`.[1]

`int` **SIGIOT** Macro

This is generated by the PDP-11 "iot" instruction. On most machines, this is just another name for `SIGABRT`.

`int` **SIGTRAP** Macro

This is generated by the machine's breakpoint instruction, and possibly other trap instructions. This signal is used by debuggers. Your program will probably only see `SIGTRAP` if it is somehow executing bad instructions.

`int` **SIGEMT** Macro

This indicates an emulator trap; it results from certain unimplemented instructions that might be emulated in software, or the operating system's failure to properly emulate them.

`int` **SIGSYS** Macro

This indicates a bad system call. The instruction to trap to the operating system was executed, but the code number for the system call to perform was invalid.

17.2.2 Termination Signals

These signals are all used to tell a process to terminate, in one way or another. They have different names because they're used for slightly different purposes, and programs might want to handle them differently.

The reason for handling these signals is usually so your program can tidy up as appropriate before actually terminating. For example, you might want to save state information, delete temporary files, or restore the previous terminal modes. Such a handler should end by specifying the default action for the signal that happened

[1] See Loosemore et al., "Aborting a Program" (see chap. 1, n. 1).

and then reraising it; this will cause the program to terminate with that signal, as if it had not had a handler (see Section 17.4.2 [Handlers That Terminate the Process], page 398).

The (obvious) default action for all of these signals is to cause the process to terminate.

`int` **SIGTERM** Macro

The `SIGTERM` signal is a generic signal used to cause program termination. Unlike `SIGKILL`, this signal can be blocked, handled and ignored. It is the normal way to politely ask a program to terminate.

The shell command `kill` generates `SIGTERM` by default.

`int` **SIGINT** Macro

The `SIGINT` ("program interrupt") signal is sent when the user types the INTR character (normally *C-c*). See Section 6.4.9 [Special Characters], page 194, for information about terminal driver support for *C-c*.

`int` **SIGQUIT** Macro

The `SIGQUIT` signal is similar to `SIGINT`, except that it's controlled by a different key—the QUIT character, usually *C-*—and produces a core dump when it terminates the process, just like a program-error signal. You can think of this as a program-error condition "detected" by the user.

See Section 17.2.1 [Program-Error Signals], page 379, for information about core dumps, and Section 6.4.9 [Special Characters], page 194, for information about terminal-driver support.

Certain kinds of clean-ups are best omitted in handling `SIGQUIT`. For example, if the program creates temporary files, it should handle the other termination requests by deleting the temporary files. But it is better for `SIGQUIT` not to delete them, so that the user can examine them in conjunction with the core dump.

`int` **SIGKILL** Macro

The `SIGKILL` signal is used to cause immediate program termination. It cannot be handled or ignored, and is therefore always fatal. It is also not possible to block this signal.

This signal is usually generated only by explicit request. Since it cannot be handled, you should generate it only as a last resort, after first trying a less drastic method such as *C-c* or `SIGTERM`. If a process does not respond to any other termination signals, sending it a `SIGKILL` signal will almost always cause it to go away.

In fact, if `SIGKILL` fails to terminate a process, that by itself constitutes an operating system bug that you should report.

The system will generate `SIGKILL` for a process itself under some unusual conditions where the program cannot possibly continue to run (even to run a signal handler).

`int` **SIGHUP** Macro

The `SIGHUP` ("hang-up") signal is used to report that the user's terminal is disconnected, perhaps because a network or telephone connection was broken (see Section 6.4.6 [Control Modes], page 187).

This signal is also used to report the termination of the controlling process on a terminal to jobs associated with that session; this termination effectively disconnects all processes in the session from the controlling terminal.[2]

17.2.3 Alarm Signals

These signals are used to indicate the expiration of timers.[3]

The default behavior for these signals is to cause program termination. This default is rarely useful, but no other default would be useful; most of the ways of using these signals would require handler functions in any case.

`int` **SIGALRM** Macro

This signal typically indicates expiration of a timer that measures real or clock time. It is used by the `alarm` function, for example.

`int` **SIGVTALRM** Macro

This signal typically indicates expiration of a timer that measures CPU time used by the current process. The name is an abbreviation for "virtual time alarm".

`int` **SIGPROF** Macro

This signal typically indicates expiration of a timer that measures both CPU time used by the current process, and CPU time expended on behalf of the process by the system. Such a timer is used to implement code-profiling facilities, hence the name of this signal.

17.2.4 Asynchronous-I/O Signals

The signals listed in this section are used in conjunction with asynchronous I/O facilities. You have to take explicit action by calling `fcntl` to enable a particular file-descriptor to generate these signals (see Section 2.16 [Interrupt-Driven Input], page 68). The default action for these signals is to ignore them.

`int` **SIGIO** Macro

This signal is sent when a file descriptor is ready to perform input or output.

---

[2] Ibid., "Termination Internals".

[3] For information about functions that cause these signals to be sent see Loosemore et al., "Setting an Alarm".

On most operating systems, terminals and sockets are the only kinds of files that can generate `SIGIO`; other kinds, including ordinary files, never generate `SIGIO` even if you ask them to.

In the GNU system, `SIGIO` will always be generated properly if you successfully set asynchronous mode with `fcntl`.

`int` **SIGURG** Macro

This signal is sent when "urgent" or out-of-band data arrives on a socket (see Section 5.9.8 [Out-of-Band Data], page 164).

`int` **SIGPOLL** Macro

This is a System V signal name, more or less similar to `SIGIO`. It is defined only for compatibility.

17.2.5 Job Control Signals

These signals are used to support job control. If your system doesn't support job control, then these macros are defined but the signals themselves can't be raised or handled.

You should generally leave these signals alone unless you really understand how job control works (see Chapter 8 [Job Control], page 221).

`int` **SIGCHLD** Macro

This signal is sent to a parent process whenever one of its child processes terminates or stops.

The default action for this signal is to ignore it. If you establish a handler for this signal while there are child processes that have terminated but not reported their status via `wait` or `waitpid` (see Section 7.6 [Process Completion], page 215), whether your new handler applies to those processes or not depends on the particular operating system.

`int` **SIGCLD** Macro

This is an obsolete name for `SIGCHLD`.

`int` **SIGCONT** Macro

You can send a `SIGCONT` signal to a process to make it continue. This signal is special—it always makes the process continue if it is stopped, before the signal is delivered. The default behavior is to do nothing else. You cannot block this signal. You can set a handler, but `SIGCONT` always makes the process continue regardless.

Most programs have no reason to handle `SIGCONT`; they simply resume execution without realizing they were ever stopped. You can use a handler for `SIGCONT` to make a program do something special when it is stopped and continued—for example, to reprint a prompt when it is suspended while waiting for input.

`int` **SIGSTOP** Macro

The `SIGSTOP` signal stops the process. It cannot be handled, ignored or blocked.

`int` **SIGTSTP** Macro

The `SIGTSTP` signal is an interactive stop-signal. Unlike `SIGSTOP`, this signal can be handled and ignored.

Your program should handle this signal if you have a special need to leave files or system tables in a secure state when a process is stopped. For example, programs that turn off echoing should handle `SIGTSTP` so they can turn echoing back on before stopping.

This signal is generated when the user types the SUSP character (normally *C-z*). For more information about terminal driver support, see Section 6.4.9 [Special Characters], page 194.

`int` **SIGTTIN** Macro

A process cannot read from the user's terminal while it is running as a background job. When any process in a background job tries to read from the terminal, all of the processes in the job are sent a `SIGTTIN` signal. The default action for this signal is to stop the process. For more information about how this interacts with the terminal driver, see Section 8.4 [Access to the Controlling Terminal], page 223.

`int` **SIGTTOU** Macro

This is similar to `SIGTTIN`, but is generated when a process in a background job attempts to write to the terminal or set its modes. Again, the default action is to stop the process. `SIGTTOU` is only generated for an attempt to write to the terminal if the `TOSTOP` output-mode is set (see Section 6.4.5 [Output Modes], page 187).

While a process is stopped, no more signals can be delivered to it until it is continued, except `SIGKILL` signals and (obviously) `SIGCONT` signals. The signals are marked as pending, but not delivered until the process is continued. The `SIGKILL` signal always causes termination of the process and can't be blocked, handled or ignored. You can ignore `SIGCONT`, but it always causes the process to be continued anyway if it is stopped. Sending a `SIGCONT` signal to a process causes any pending stop-signals for that process to be discarded. Likewise, any pending `SIGCONT` signals for a process are discarded when it receives a stop signal.

When a process in an orphaned process-group (see Section 8.5 [Orphaned Process-Groups], page 223) receives a `SIGTSTP`, `SIGTTIN` or `SIGTTOU` signal and does not handle it, the process does not stop. Stopping the process would

probably not be very useful, since there is no shell program that will notice and allow the user to continue it. What happens instead depends on the operating system you are using. Some systems may do nothing; others may deliver another signal instead, such as SIGKILL or SIGHUP. In the GNU system, the process dies with SIGKILL; this avoids the problem of many stopped, orphaned processes lying around the system.

17.2.6 Operation-Error Signals

These signals are used to report various errors generated by an operation done by the program. They do not necessarily indicate a programming error in the program, but an error that prevents an operating system call from completing. The default action for all of them is to cause the process to terminate.

`int` **SIGPIPE** Macro

There is a broken pipe. If you use pipes or FIFOs, you have to design your application so that one process opens the pipe for reading before another starts writing. If the reading process never starts, or terminates unexpectedly, writing to the pipe or FIFO raises a SIGPIPE signal. If SIGPIPE is blocked, handled or ignored, the offending call fails with EPIPE instead.

Pipes and FIFO special files are discussed in more detail in Chapter 4 [Pipes and FIFOs], page 119.

Another cause of SIGPIPE is when you try to output to a socket that isn't connected (see Section 5.9.5.1 [Sending Data], page 157).

`int` **SIGLOST** Macro

The resource was lost. This signal is generated when you have an advisory lock on an NFS file, and the NFS server reboots and forgets about your lock.

In the GNU system, SIGLOST is generated when any server program dies unexpectedly. It is usually fine to ignore the signal; whatever call was made to the server that died just returns an error.

`int` **SIGXCPU** Macro

The CPU-time limit exceeded. This signal is generated when the process exceeds its soft resource-limit on CPU time (see Section 14.2 [Limiting Resource Usage], page 338).

`int` **SIGXFSZ** Macro

The file size limit was exceeded. This signal is generated when the process attempts to extend a file so it exceeds the process's soft resource-limit on file size (see Section 14.2 [Limiting Resource Usage], page 338).

17.2.7 Miscellaneous Signals

These signals are used for various other purposes. In general, they will not affect your program unless it explicitly uses them for something.

`int` **SIGUSR1** Macro
`int` **SIGUSR2** Macro

The `SIGUSR1` and `SIGUSR2` signals are set aside for you to use any way you want. They're useful for simple interprocess communication, if you write a signal handler for them in the program that receives the signal.

There is an example showing the use of `SIGUSR1` and `SIGUSR2` in Section 17.6.2 [Signaling Another Process], page 410.

The default action is to terminate the process.

`int` **SIGWINCH** Macro

This signals a window size change. This is generated on some systems (including GNU) when the terminal driver's record of the number of rows and columns on the screen is changed. The default action is to ignore it.

If a program does full-screen display, it should handle `SIGWINCH`. When the signal arrives, it should fetch the new screen-size and reformat its display accordingly.

`int` **SIGINFO** Macro

Information request. In 4.4 BSD and the GNU system, this signal is sent to all the processes in the foreground process-group of the controlling terminal when the user types the STATUS character in canonical mode (see Section 6.4.9.2 [Characters that Cause Signals], page 196).

If the process is the leader of the process group, the default action is to print some status information about the system and what the process is doing. Otherwise, the default is to do nothing.

17.2.8 Signal Messages

We mentioned above that the shell prints a message describing the signal that terminated a child process. The clean way to print a message describing a signal is to use the functions `strsignal` and `psignal`. These functions use a signal number to specify which kind of signal to describe. The signal number may come from the termination status of a child process (see Section 7.6 [Process Completion], page 215), or it may come from a signal handler in the same process.

`char *` **strsignal** (`int` *signum*) Function

This function returns a pointer to a statically allocated string containing a message describing the signal *signum*. You should not modify the contents of this string; and, since it can be rewritten on subsequent calls, you should save a copy of it if you need to reference it later.

This function is a GNU extension, declared in the header file '`string.h`'.

`void` **psignal** (`int` *signum*, `const char *`*message*) Function

This function prints a message describing the signal *signum* to the standard error output stream `stderr`.[4]

If you call `psignal` with a *message* that is either a null pointer or an empty string, `psignal` just prints the message corresponding to *signum*, adding a trailing newline.

If you supply a nonnull *message* argument, then `psignal` prefixes its output with this string. It adds a colon and a space character to separate the *message* from the string corresponding to *signum*.

This function is a BSD feature, declared in the header file '`signal.h`'.

There is also an array `sys_siglist` that contains the messages for the various signal codes. This array exists on BSD systems, unlike `strsignal`.

17.3 Specifying Signal Actions

The simplest way to change the action for a signal is to use the `signal` function. You can specify a built-in action (such as to ignore the signal), or you can *establish a handler*.

The GNU library also implements the more versatile `sigaction` facility. This section describes both facilities and gives suggestions on which to use when.

17.3.1 Basic Signal-Handling

The `signal` function provides a simple interface for establishing an action for a particular signal. The function and associated macros are declared in the header file '`signal.h`'.

sighandler_t Data Type

This is the type of signal-handler functions. Signal handlers take one integer argument specifying the signal number, and have return type `void`. So, you should define handler functions like this:

```
void handler (int signum) { ... }
```

The name `sighandler_t` for this data type is a GNU extension.

`sighandler_t` **signal** (`int` *signum*, `sighandler_t` *action*) Function

The `signal` function establishes *action* as the action for the signal *signum*.

The first argument, *signum*, identifies the signal whose behavior you want to control, and should be a signal number. The proper way to specify a signal number is with one of the symbolic signal-names (see Section 17.2 [Standard

[4] Ibid., "Standard Streams".

Signals], page 379)—don't use an explicit number, because the numerical code for a given kind of signal may vary from operating system to operating system.

The second argument, *action*, specifies the action to use for the signal *signum*. This can be one of the following:

`SIG_DFL` `SIG_DFL` specifies the default action for the particular signal. The default actions for various kinds of signals are stated in Section 17.2 [Standard Signals], page 379.

`SIG_IGN` `SIG_IGN` specifies that the signal should be ignored.

Your program generally should not ignore signals that represent serious events or that are normally used to request termination. You cannot ignore the `SIGKILL` or `SIGSTOP` signals at all. You can ignore program-error signals like `SIGSEGV`, but ignoring the error won't enable the program to continue executing meaningfully. Ignoring user requests such as `SIGINT`, `SIGQUIT` and `SIGTSTP` is unfriendly.

When you do not wish signals to be delivered during a certain part of the program, the thing to do is to block them, not ignore them (see Section 17.7 [Blocking Signals], page 414).

handler Supply the address of a handler function in your program, to specify running this handler as the way to deliver the signal.

For more information about defining signal-handler functions, see Section 17.4 [Defining Signal-Handlers], page 396.

If you set the action for a signal to `SIG_IGN`, or if you set it to `SIG_DFL` and the default action is to ignore that signal, then any pending signals of that type are discarded (even if they are blocked). Discarding the pending signals means that they will never be delivered, not even if you subsequently specify another action and unblock this kind of signal.

The `signal` function returns the action that was previously in effect for the specified *signum*. You can save this value and restore it later by calling `signal` again.

If `signal` can't honor the request, it returns `SIG_ERR` instead. The following `errno` error condition is defined for this function:

`EINVAL` You specified an invalid *signum*; or you tried to ignore or provide a handler for `SIGKILL` or `SIGSTOP`.

Compatibility Note: A problem encountered when working with the `signal` function is that it has different semantics on BSD and SVID systems. The difference is that on SVID systems, the signal handler is deinstalled after signal delivery. On BSD systems, the handler must be explicitly deinstalled. In the GNU C Library, we use the BSD version by default. To use the SVID version, you can either use the function `sysv_signal` (see below), or use the `_XOPEN_SOURCE` feature-select macro (see Section 1.3.4 [Feature-Test Macros], page 8). In general, use of these

functions should be avoided because of compatibility problems. It is better to use `sigaction` if it is available, since the results are much more reliable.

Here is a simple example of setting up a handler to delete temporary files when certain fatal signals happen:

```
#include <signal.h>

void
termination_handler (int signum)
{
  struct temp_file *p;

  for (p = temp_file_list; p; p = p->next)
    unlink (p->name);
}

int
main (void)
{
  ...
  if (signal (SIGINT, termination_handler) == SIG_IGN)
    signal (SIGINT, SIG_IGN);
  if (signal (SIGHUP, termination_handler) == SIG_IGN)
    signal (SIGHUP, SIG_IGN);
  if (signal (SIGTERM, termination_handler) == SIG_IGN)
    signal (SIGTERM, SIG_IGN);
  ...
}
```

If a given signal was previously set to be ignored, this code avoids altering that setting. This is because non-job-control shells often ignore certain signals when starting children, and it is important for the children to respect this.

We do not handle `SIGQUIT` or the program-error signals in this example because these are designed to provide information for debugging (a core dump), and the temporary files may give useful information.

`sighandler_t` **sysv_signal** (`int` *signum*, `sighandler_t` *action*) Function

The `sysv_signal` implements the behavior of the standard `signal` function as found on SVID systems. The difference with BSD systems is that the handler is deinstalled after a delivery of a signal.

Compatibility Note: As said above for `signal`, this function should be avoided when possible. `sigaction` is the preferred method.

`sighandler_t` **ssignal** (`int` *signum*, `sighandler_t` *action*) Function

The `ssignal` function does the same thing as `signal`; it is provided only for compatibility with SVID.

`sighandler_t` **SIG_ERR** Macro

The value of this macro is used as the return value from `signal` to indicate an error.

17.3.2 Advanced Signal-Handling

The `sigaction` function has the same basic effect as `signal`—to specify how a signal should be handled by the process. However, `sigaction` offers more control, at the expense of more complexity. In particular, `sigaction` allows you to specify additional flags to control when the signal is generated and how the handler is invoked.

The `sigaction` function is declared in '`signal.h`'.

struct sigaction Data Type

Structures of type `struct sigaction` are used in the `sigaction` function to specify all the information about how to handle a particular signal. This structure contains at least the following members:

`sighandler_t sa_handler`
: This is used in the same way as the *action* argument to the `signal` function. The value can be `SIG_DFL`, `SIG_IGN` or a function pointer (see Section 17.3.1 [Basic Signal-Handling], page 389).

`sigset_t sa_mask`
: This specifies a set of signals to be blocked while the handler runs (see Section 17.7.5 [Blocking Signals for a Handler], page 418). The signal that was delivered is automatically blocked by default before its handler is started; this is true regardless of the value in `sa_mask`. If you want that signal not to be blocked within its handler, you must write code in the handler to unblock it.

`int sa_flags`
: This specifies various flags that can affect the behavior of the signal (see Section 17.3.5 [Flags for `sigaction`], page 395).

`int` **sigaction** (`int` *signum*, `const struct sigaction *restrict` *action*, `struct sigaction *restrict` *old-action*) Function

The *action* argument is used to set up a new action for the signal *signum*, while the *old-action* argument is used to return information about the action previously associated with this symbol. In other words, *old-action* has the same purpose as

the `signal` function's return value—you can check to see what the old action in effect for the signal was, and restore it later if you want.

Either *action* or *old-action* can be a null pointer. If *old-action* is a null pointer, this simply suppresses the return of information about the old action. If *action* is a null pointer, the action associated with the signal *signum* is unchanged; this allows you to inquire about how a signal is being handled without changing that handling.

The return value from `sigaction` is 0 if it succeeds and `-1` on failure. The following `errno` error condition is defined for this function:

`EINVAL` The *signum* argument is not valid, or you are trying to trap or ignore `SIGKILL` or `SIGSTOP`.

17.3.3 Interaction of `signal` and `sigaction`

It's possible to use both the `signal` and `sigaction` functions within a single program, but you have to be careful because they can interact in slightly strange ways.

The `sigaction` function specifies more information than the `signal` function, so the return value from `signal` cannot express the full range of `sigaction` possibilities. Therefore, if you use `signal` to save and later reestablish an action, it may not be able to reestablish properly a handler that was established with `sigaction`.

To avoid having problems as a result, always use `sigaction` to save and restore a handler if your program uses `sigaction` at all. Since `sigaction` is more general, it can properly save and reestablish any action, regardless of whether it was established originally with `signal` or `sigaction`.

On some systems, if you establish an action with `signal` and then examine it with `sigaction`, the handler address that you get may not be the same as what you specified with `signal`. It may not even be suitable for use as an action argument with `signal`. But you can rely on using it as an argument to `sigaction`. This problem never happens on the GNU system.

So, you're better off using one or the other of the mechanisms consistently within a single program.

Portability Note: The basic `signal` function is a feature of ISO C, while `sigaction` is part of the POSIX.1 standard. If you are concerned about portability to non-POSIX systems, then you should use the `signal` function instead.

17.3.4 `sigaction` Function Example

In Section 17.3.1 [Basic Signal-Handling], page 389, we gave an example of establishing a simple handler for termination signals using `signal`. Here is an equivalent example using `sigaction`:

```
#include <signal.h>
```

```
void
termination_handler (int signum)
{
  struct temp_file *p;

  for (p = temp_file_list; p; p = p->next)
    unlink (p->name);
}

int
main (void)
{
  ...
  struct sigaction new_action, old_action;

  /* Set up the structure to specify the new action. */
  new_action.sa_handler = termination_handler;
  sigemptyset (&new_action.sa_mask);
  new_action.sa_flags = 0;

  sigaction (SIGINT, NULL, &old_action);
  if (old_action.sa_handler != SIG_IGN)
    sigaction (SIGINT, &new_action, NULL);
  sigaction (SIGHUP, NULL, &old_action);
  if (old_action.sa_handler != SIG_IGN)
    sigaction (SIGHUP, &new_action, NULL);
  sigaction (SIGTERM, NULL, &old_action);
  if (old_action.sa_handler != SIG_IGN)
    sigaction (SIGTERM, &new_action, NULL);
  ...
}
```

The program just loads the `new_action` structure with the desired parameters and passes it in the `sigaction` call. The usage of `sigemptyset` is described later (see Section 17.7 [Blocking Signals], page 414).

As in the example using `signal`, we avoid handling signals previously set to be ignored. Here we can avoid altering the signal handler even momentarily, by using the feature of `sigaction` that lets us examine the current action without specifying a new one.

Here is another example. It retrieves information about the current action for `SIGINT` without changing that action.

```
struct sigaction query_action;

if (sigaction (SIGINT, NULL, &query_action) < 0)
  /* sigaction returns -1 in case of error. */
```

```
else if (query_action.sa_handler == SIG_DFL)
  /* SIGINT is handled in the default, fatal manner. */
else if (query_action.sa_handler == SIG_IGN)
  /* SIGINT is ignored. */
else
  /* A programmer-defined signal-handler is in effect. */
```

17.3.5 Flags for `sigaction`

The `sa_flags` member of the `sigaction` structure is a catch-all for special features. Most of the time, `SA_RESTART` is a good value to use for this field.

The value of `sa_flags` is interpreted as a bit mask. Thus, you should choose the flags you want to set, OR those flags together and store the result in the `sa_flags` member of your `sigaction` structure.

Each signal number has its own set of flags. Each call to `sigaction` affects one particular signal-number, and the flags that you specify apply only to that particular signal.

In the GNU C Library, establishing a handler with `signal` sets all the flags to 0 except for `SA_RESTART`, whose value depends on the settings you have made with `siginterrupt` (see Section 17.5 [Primitives Interrupted by Signals], page 408).

These macros are defined in the header file '`signal.h`'.

`int` **SA_NOCLDSTOP** Macro

This flag is meaningful only for the `SIGCHLD` signal. When the flag is set, the system delivers the signal for a terminated child-process but not for one that is stopped. By default, `SIGCHLD` is delivered for both terminated children and stopped children.

Setting this flag for a signal other than `SIGCHLD` has no effect.

`int` **SA_ONSTACK** Macro

If this flag is set for a particular signal-number, the system uses the signal stack when delivering that kind of signal (see Section 17.9 [Using a Separate Signal-Stack], page 424). If a signal with this flag arrives and you have not set a signal stack, the system terminates the program with `SIGILL`.

`int` **SA_RESTART** Macro

This flag controls what happens when a signal is delivered during certain primitives (such as `open`, `read` or `write`), and the signal handler returns normally. There are two alternatives: the library function can resume, or it can return failure with error code `EINTR`.

The choice is controlled by the `SA_RESTART` flag for the particular kind of signal that was delivered. If the flag is set, returning from a handler resumes the library function. If the flag is clear, returning from a handler makes the function fail (see Section 17.5 [Primitives Interrupted by Signals], page 408).

17.3.6 Initial Signal Actions

When a new process is created (see Section 7.4 [Creating a Process], page 211), it inherits handling of signals from its parent process. However, when you load a new process image using the `exec` function (see Section 7.5 [Executing a File], page 212), any signals that you've defined your own handlers for revert to their `SIG_DFL` handling. This makes sense; the handler functions from the old program are specific to that program, and aren't even present in the address space of the new program image. Of course, the new program can establish its own handlers.

When a program is run by a shell, the shell normally sets the initial actions for the child process to `SIG_DFL` or `SIG_IGN`, as appropriate. It's a good idea to check to make sure that the shell has not set up an initial action of `SIG_IGN` before you establish your own signal-handlers.

Here is an example of how to establish a handler for `SIGHUP`, but not if `SIGHUP` is currently ignored:

```
...
struct sigaction temp;

sigaction (SIGHUP, NULL, &temp);

if (temp.sa_handler != SIG_IGN)
  {
    temp.sa_handler = handle_sighup;
    sigemptyset (&temp.sa_mask);
    sigaction (SIGHUP, &temp, NULL);
  }
```

17.4 Defining Signal-Handlers

This section describes how to write a signal-handler function that can be established with the `signal` or `sigaction` functions.

A signal handler is just a function that you compile together with the rest of the program. Instead of directly invoking the function, you use `signal` or `sigaction` to tell the operating system to call it when a signal arrives. This is known as *establishing* the handler (see Section 17.3 [Specifying Signal Actions], page 389).

There are two basic strategies you can use in signal-handler functions:

- You can have the handler function note that the signal arrived by tweaking some global data structures, and then return normally.
- You can have the handler function terminate the program or transfer control to a point where it can recover from the situation that caused the signal.

You need to take special care in writing handler-functions, because they can be called asynchronously. A handler might be called at any point in the program, unpredictably. If two signals arrive during a very short interval, one handler can run within another. This section describes what your handler should do, and what you should avoid.

17.4.1 Signal Handlers That Return

Handlers that return normally are usually used for signals such as `SIGALRM` and the I/O and interprocess-communication signals. But a handler for `SIGINT` might also return normally after setting a flag that tells the program to exit at a convenient time.

It is not safe to return normally from the handler for a program-error signal, because the behavior of the program when the handler function returns is not defined after a program error (see Section 17.2.1 [Program-Error Signals], page 379).

Handlers that return normally must modify some global variable in order to have any effect. Typically, the variable is one that is examined periodically by the program during normal operation. Its data type should be `sig_atomic_t` for reasons described in Section 17.4.7 [Atomic Data-Access and Signal-Handling], page 406.

Here is a simple example of such a program. It executes the body of the loop until it has noticed that a `SIGALRM` signal has arrived. This technique is useful because it allows the iteration in progress when the signal arrives to complete before the loop exits.

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

/* This flag controls termination of the main loop. */
volatile sig_atomic_t keep_going = 1;

/* The signal handler just clears the flag and re-enables itself. */
void
catch_alarm (int sig)
{
  keep_going = 0;
  signal (sig, catch_alarm);
}

void
do_stuff (void)
{
  puts ("Doing stuff while waiting for alarm....");
}
```

```
int
main (void)
{
  /* Establish a handler for SIGALRM signals. */
  signal (SIGALRM, catch_alarm);

  /* Set an alarm to go off in a little while. */
  alarm (2);

  /* Check the flag once in a while to see when to quit. */
  while (keep_going)
    do_stuff ();

  return EXIT_SUCCESS;
}
```

17.4.2 Handlers That Terminate the Process

Handler functions that terminate the program are typically used to cause orderly clean-up or recovery from program-error signals and interactive interrupts.

The cleanest way for a handler to terminate the process is to raise the same signal that ran the handler in the first place. Here is how to do this:

```
volatile sig_atomic_t fatal_error_in_progress = 0;

void
fatal_error_signal (int sig)
{

  /* Since this handler is established for more than one kind of signal,
     it might still get invoked recursively by delivery of some other kind
     of signal.  Use a static variable to keep track of that. */
  if (fatal_error_in_progress)
    raise (sig);
  fatal_error_in_progress = 1;



  /* Now do the clean up actions:
     - reset terminal modes
     - kill child processes
     - remove lock files */
  ...
```

```
  /* Now reraise the signal.  We reactivate the signal's
     default handling, which is to terminate the process.
     We could just call exit or abort,
     but reraising the signal sets the return status
     from the process correctly.  */
  signal (sig, SIG_DFL);
  raise (sig);
}
```

17.4.3 Nonlocal Control-Transfer in Handlers

You can do a nonlocal transfer of control out of a signal handler using the `setjmp` and `longjmp` facilities (see Chapter 16 [Nonlocal Exits], page 367).

When the handler does a nonlocal control-transfer, the part of the program that was running will not continue. If this part of the program was in the middle of updating an important data structure, the data structure will remain inconsistent. Since the program does not terminate, the inconsistency is likely to be noticed later on.

There are two ways to avoid this problem. One is to block the signal for the parts of the program that update important data structures. Blocking the signal delays its delivery until it is unblocked, once the critical updating is finished (see Section 17.7 [Blocking Signals], page 414).

The other way is to reinitialize the crucial data structures in the signal handler, or make their values consistent.

Here is a rather schematic example showing the reinitialization of one global variable:

```
#include <signal.h>
#include <setjmp.h>

jmp_buf return_to_top_level;

volatile sig_atomic_t waiting_for_input;

void
handle_sigint (int signum)
{
  /* We may have been waiting for input when the signal arrived,
     but we are no longer waiting once we transfer control.  */
  waiting_for_input = 0;
```

```
  longjmp (return_to_top_level, 1);
}



int
main (void)
{
  ...
  signal (SIGINT, sigint_handler);
  ...
  while (1) {
    prepare_for_command ();
    if (setjmp (return_to_top_level) == 0)
      read_and_execute_command ();
  }
}



/* Imagine this is a subroutine used by various commands. */
char *
read_data ()
{
  if (input_from_terminal) {
    waiting_for_input = 1;
    ...
    waiting_for_input = 0;
  } else {
    ...
  }
}
```

17.4.4 Signals Arriving While a Handler Runs

What happens if another signal arrives while your signal-handler function is running?

When the handler for a particular signal is invoked, that signal is automatically blocked until the handler returns. That means that if two signals of the same kind arrive close together, the second one will be held until the first has been handled. The handler can explicitly unblock the signal using `sigprocmask`, if you want to allow more signals of this type to arrive (see Section 17.7.3 [Process Signal-Mask], page 416).

However, your handler can still be interrupted by delivery of another kind of signal. To avoid this, you can use the `sa_mask` member of the action structure passed to `sigaction` to explicitly specify which signals should be blocked while the signal handler runs. These signals are in addition to the signal for which the handler was invoked, and any other signals that are normally blocked by the process (see Section 17.7.5 [Blocking Signals for a Handler], page 418).

When the handler returns, the set of blocked signals is restored to the value it had before the handler ran. So using `sigprocmask` inside the handler only affects what signals can arrive during the execution of the handler itself, not what signals can arrive once the handler returns.

Portability Note: Always use `sigaction` to establish a handler for a signal that you expect to receive asynchronously, if you want your program to work properly on System V Unix. On this system, the handling of a signal whose handler was established with `signal` automatically sets the signal's action back to `SIG_DFL`, and the handler must reestablish itself each time it runs. This practice, while inconvenient, does work when signals cannot arrive in succession. However, if another signal can arrive right away, it may arrive before the handler can reestablish itself. Then the second signal would receive the default handling, which could terminate the process.

17.4.5 Signals Close Together Merge into One

If multiple signals of the same type are delivered to your process before your signal-handler has a chance to be invoked at all, the handler may only be invoked once, as if only a single signal had arrived. In effect, the signals merge into one. This situation can arise when the signal is blocked, or in a multiprocessing environment where the system is busy running some other processes while the signals are delivered. This means, for example, that you cannot reliably use a signal handler to count signals. The only distinction you can reliably make is whether at least one signal has arrived since a given time in the past.

Here is an example of a handler for `SIGCHLD` that compensates for the fact that the number of signals received may not equal the number of child processes that generate them. It assumes that the program keeps track of all the child processes with a chain of structures as follows:

```
struct process
{
  struct process *next;
  /* The process ID of this child.  */
  int pid;
  /* The descriptor of the pipe or pseudoterminal
     on which output comes from this child.  */
  int input_descriptor;
  /* Nonzero if this process has stopped or terminated  */
  sig_atomic_t have_status;
  /* The status of this child; 0 if running,
```

```
      otherwise a status value from waitpid  */
  int status;
};

struct process *process_list;
```

This example also uses a flag to indicate whether signals have arrived since some time in the past—whenever the program last cleared it to 0.

```
/* Nonzero means some child's status has changed,
   so look at process_list for the details.  */
int process_status_change;
```

Here is the handler itself:

```
void
sigchld_handler (int signo)
{
  int old_errno = errno;

  while (1) {
    register int pid;
    int w;
    struct process *p;

    /* Keep asking for a status until we get a definitive result.  */
    do
      {
        errno = 0;
        pid = waitpid (WAIT_ANY, &w, WNOHANG | WUNTRACED);
      }
    while (pid <= 0 && errno == EINTR);

    if (pid <= 0) {
      /* A real failure means there are no more
         stopped or terminated child processes, so return.  */
      errno = old_errno;
      return;
    }

    /* Find the process that signaled us, and record its status.  */

    for (p = process_list; p; p = p->next)
      if (p->pid == pid) {
        p->status = w;
        /* Indicate that the status field
           has data to look at.  We do this only after storing it.  */
```

```
            p->have_status = 1;

            /* If process has terminated, stop waiting for its output.  */
            if (WIFSIGNALED (w) || WIFEXITED (w))
              if (p->input_descriptor)
                FD_CLR (p->input_descriptor, &input_wait_mask);

            /* The program should check this flag from time to time
               to see if there is any news in process_list.  */
            ++process_status_change;
          }

        /* Loop around to handle all the processes
           that have something to tell us.  */
      }
  }
```

Here is the proper way to check the flag `process_status_change`:

```
if (process_status_change) {
  struct process *p;
  process_status_change = 0;
  for (p = process_list; p; p = p->next)
    if (p->have_status) {
      ... Examine p->status ...
    }
}
```

It is vital to clear the flag before examining the list; otherwise, if a signal were delivered just before the clearing of the flag, and after the appropriate element of the process list had been checked, the status change would go unnoticed until the next signal arrived to set the flag again. You could, of course, avoid this problem by blocking the signal while scanning the list, but it is much more elegant to guarantee correctness by doing things in the right order.

The loop that checks process status avoids examining `p->status` until it sees that status has been validly stored. This is to make sure that the status cannot change in the middle of accessing it. Once `p->have_status` is set, it means that the child process is stopped or terminated, and in either case, it cannot stop or terminate again until the program has taken notice. See Section 17.4.7.3 [Atomic Usage-Patterns], page 407, for more information about coping with interruptions during accesses of a variable.

Here is another way you can test whether the handler has run since the last time you checked. This technique uses a counter that is never changed outside the handler. Instead of clearing the count, the program remembers the previous value and sees whether it has changed since the previous check. The advantage of this method is that different parts of the program can check independently, each part checking whether there has been a signal since that part last checked.

```
sig_atomic_t process_status_change;

sig_atomic_t last_process_status_change;

...
{
  sig_atomic_t prev = last_process_status_change;
  last_process_status_change = process_status_change;
  if (last_process_status_change != prev) {
    struct process *p;
    for (p = process_list; p; p = p->next)
      if (p->have_status) {
        ... Examine p->status ...
      }
  }
}
```

17.4.6 Signal Handling and Nonreentrant Functions

Handler functions usually don't do very much. The best practice is to write a handler that does nothing but set an external variable that the program checks regularly, and leave all serious work to the program. This is best because the handler can be called asynchronously, at unpredictable times—perhaps in the middle of a primitive function, or even between the beginning and the end of a C operator that requires multiple instructions. The data structures being manipulated might therefore be in an inconsistent state when the handler function is invoked. Even copying one `int` variable into another can take two instructions on most machines.

This means you have to be very careful about what you do in a signal handler.

- If your handler needs to access any global variables from your program, declare those variables `volatile`. This tells the compiler that the value of the variable might change asynchronously, and inhibits certain optimizations that would be invalidated by such modifications.
- If you call a function in the handler, make sure it is *reentrant* with respect to signals, or else make sure that the signal cannot interrupt a call to a related function.

A function can be nonreentrant if it uses memory that is not on the stack.

- If a function uses a static variable or a global variable, or a dynamically allocated object that it finds for itself, then it is nonreentrant and any two calls to the function can interfere.

  For example, suppose that the signal handler uses `gethostbyname`. This function returns its value in a static object, reusing the same object each time. If the signal happens to arrive during a call to `gethostbyname`, or even after one (while the program is still using the value), it will clobber the value that the program asked for.

However, if the program does not use `gethostbyname` or any other function that returns information in the same object, or if it always blocks signals around each use, then you are safe.

There are a large number of library functions that return values in a fixed object, always reusing the same object in this fashion, and all of them cause the same problem. Function descriptions in this manual always mention this behavior.

- If a function uses and modifies an object that you supply, then it is potentially nonreentrant; two calls can interfere if they use the same object.

  This case arises when you do I/O using streams. Suppose that the signal handler prints a message with `fprintf`. Suppose that the program was in the middle of an `fprintf` call using the same stream when the signal was delivered. Both the signal handler's message and the program's data could be corrupted, because both calls operate on the same data structure—the stream itself.

  However, if you know that the stream that the handler uses cannot possibly be used by the program at a time when signals can arrive, then you are safe. It is no problem if the program uses some other stream.

- On most systems, `malloc` and `free` are not reentrant, because they use a static data-structure that records what memory blocks are free. As a result, no library functions that allocate or free memory are reentrant. This includes functions that allocate space to store a result.

  The best way to avoid the need to allocate memory in a handler is to allocate in advance space for signal handlers to use.

  The best way to avoid freeing memory in a handler is to flag or record the objects to be freed, and have the program check from time to time whether anything is waiting to be freed. But this must be done with care, because placing an object on a chain is not atomic, and if it is interrupted by another signal handler that does the same thing, you could "lose" one of the objects.

- Any function that modifies `errno` is nonreentrant, but you can correct for this. In the handler, save the original value of `errno` and restore it before returning normally. This prevents errors that occur within the signal handler from being confused with errors from system calls at the point the program is interrupted to run the handler.

  This technique is generally applicable; if you want to call in a handler a function that modifies a particular object in memory, you can make this safe by saving and restoring that object.

- Merely reading from a memory object is safe provided that you can deal with any of the values that might appear in the object at a time when the signal can be delivered. Keep in mind that assignment to some data types requires more than one instruction, which means that the handler could run "in the middle of" an assignment to the variable if its type is not atomic (see Section 17.4.7 [Atomic Data-Access and Signal-Handling], page 406).

- Merely writing into a memory object is safe as long as a sudden change in the value, at any time when the handler might run, will not disturb anything.

17.4.7 Atomic Data-Access and Signal-Handling

Whether the data in your application concerns atoms, or mere text, you have to be careful about the fact that access to a single datum is not necessarily *atomic*. This means that it can take more than one instruction to read or write a single object. In such cases, a signal handler might be invoked in the middle of reading or writing the object.

There are three ways you can cope with this problem. You can use data types that are always accessed atomically; you can carefully arrange that nothing untoward happens if an access is interrupted, or you can block all signals around any access that had better not be interrupted (see Section 17.7 [Blocking Signals], page 414).

17.4.7.1 Problems with Nonatomic Access

Here is an example which shows what can happen if a signal handler runs in the middle of modifying a variable. Interrupting the reading of a variable can also lead to paradoxical results, but here we only show writing.

```
#include <signal.h>
#include <stdio.h>

struct two_words { int a, b; } memory;

void
handler(int signum)
{
   printf ("%d,%d\n", memory.a, memory.b);
   alarm (1);
}


int
main (void)
{
   static struct two_words zeros = { 0, 0 }, ones = { 1, 1 };
   signal (SIGALRM, handler);
   memory = zeros;
   alarm (1);
   while (1)
     {
       memory = zeros;
       memory = ones;
     }
```

```
}
```

This program fills `memory` with zeros, ones, zeros, ones, alternating forever; meanwhile, once per second, the alarm signal-handler prints the current contents. (Calling `printf` in the handler is safe in this program because it is certainly not being called outside the handler when the signal happens.)

Clearly, this program can print a pair of zeros or a pair of ones. But that's not all it can do! On most machines, it takes several instructions to store a new value in `memory`, and the value is stored one word at a time. If the signal is delivered in between these instructions, the handler might find that `memory.a` is 0 and `memory.b` is 1 (or vice versa).

On some machines it may be possible to store a new value in `memory` with just one instruction that cannot be interrupted. On these machines, the handler will always print two zeros or two ones.

17.4.7.2 Atomic Types

To avoid uncertainty about interrupting access to a variable, you can use a particular data type for which access is always atomic: `sig_atomic_t`. Reading and writing this data type is guaranteed to happen in a single instruction, so there's no way for a handler to run "in the middle" of an access.

The type `sig_atomic_t` is always an integer data type, but which one it is, and how many bits it contains, may vary from machine to machine.

sig_atomic_t Data Type

This is an integer data type. Objects of this type are always accessed atomically.

In practice, you can assume that `int` and other integer types no longer than `int` are atomic. You can also assume that pointer types are atomic; that is very convenient. Both of these assumptions are true on all of the machines that the GNU C Library supports and on all POSIX systems we know of.

17.4.7.3 Atomic Usage-Patterns

Certain patterns of access avoid any problem even if an access is interrupted. For example, a flag that is set by the handler, and tested and cleared by the main program from time to time, is always safe even if access actually requires two instructions. To show that this is so, we must consider each access that could be interrupted, and show that there is no problem if it is interrupted.

An interrupt in the middle of testing the flag is safe because either it's recognized to be nonzero, in which case the precise value doesn't matter, or it will be seen to be nonzero the next time it's tested.

An interrupt in the middle of clearing the flag is no problem because either the value ends up 0, which is what happens if a signal comes in just before the flag is cleared, or the value ends up nonzero, and subsequent events occur as if the signal

had come in just after the flag was cleared. As long as the code handles both of these cases properly, it can also handle a signal in the middle of clearing the flag. This is an example of the sort of reasoning you need to do to figure out whether nonatomic usage is safe.

Sometimes you can insure uninterrupted access to one object by protecting its use with another object, perhaps one whose type guarantees atomicity (see Section 17.4.5 [Signals Close Together Merge into One], page 401 for an example).

17.5 Primitives Interrupted by Signals

A signal can arrive and be handled while an I/O primitive such as `open` or `read` is waiting for an I/O device. If the signal handler returns, the system faces the question: what should happen next?

POSIX specifies one approach—make the primitive fail right away. The error code for this kind of failure is `EINTR`. This is flexible, but usually inconvenient. Typically, POSIX applications that use signal handlers must check for `EINTR` after each library function that can return it, in order to try the call again. Often programmers forget to check, which is a common source of error.

The GNU library provides a convenient way to retry a call after a temporary failure, with the macro `TEMP_FAILURE_RETRY`:

TEMP_FAILURE_RETRY (*expression*) Macro

This macro evaluates *expression* once. If it fails and reports error code `EINTR`, `TEMP_FAILURE_RETRY` evaluates it again, and over and over until the result is not a temporary failure.

The value returned by `TEMP_FAILURE_RETRY` is whatever value *expression* produced.

BSD avoids `EINTR` entirely and provides a more convenient approach: to restart the interrupted primitive, instead of making it fail. If you choose this approach, you need not be concerned with `EINTR`.

You can choose either approach with the GNU library. If you use `sigaction` to establish a signal handler, you can specify how that handler should behave. If you specify the `SA_RESTART` flag, return from that handler will resume a primitive; otherwise, return from that handler will cause `EINTR` (see Section 17.3.5 [Flags for `sigaction`], page 395).

Another way to specify the choice is with the `siginterrupt` function (see Section 17.10.1 [BSD Function to Establish a Handler], page 426).

When you don't specify with `sigaction` or `siginterrupt` what a particular handler should do, it uses a default choice. The default choice in the GNU library depends on the feature-test macros you have defined. If you define `_BSD_SOURCE` or `_GNU_SOURCE` before calling `signal`, the default is to resume primitives; otherwise, the default is to make them fail with `EINTR`. (The library contains alternate versions of the `signal` function, and the feature-test macros determine which one you really call.) See Section 1.3.4 [Feature-Test Macros], page 8.

The description of each primitive affected by this issue lists `EINTR` among the error codes it can return.

There is one situation where resumption never happens no matter which choice you make—when a data-transfer function such as `read` or `write` is interrupted by a signal after transferring part of the data. In this case, the function returns the number of bytes already transferred, indicating partial success.

This might at first appear to cause unreliable behavior on record-oriented devices (including datagram sockets; see Section 5.10 [Datagram Socket Operations], page 167), where splitting one `read` or `write` into two would read or write two records. Actually, there is no problem, because interruption after a partial transfer cannot happen on such devices; they always transfer an entire record in one burst, with no waiting once data transfer has started.

17.6 Generating Signals

Besides signals that are generated as a result of a hardware trap or interrupt, your program can explicitly send signals to itself or to another process.

17.6.1 Signaling Yourself

A process can send itself a signal with the `raise` function. This function is declared in '`signal.h`'.

`int` **raise** (`int` *signum*) Function

The `raise` function sends the signal *signum* to the calling process. It returns 0 if successful and a nonzero value if it fails. About the only reason for failure would be if the value of *signum* is invalid.

`int` **gsignal** (`int` *signum*) Function

The `gsignal` function does the same thing as `raise`; it is provided only for compatibility with SVID.

One convenient use for `raise` is to reproduce the default behavior of a signal that you have trapped. For instance, suppose a user of your program types the SUSP character (usually `C-z`; see Section 6.4.9 [Special Characters], page 194) to send it an interactive stop signal (`SIGTSTP`), and you want to clean up some internal data buffers before stopping. You might set this up like this:

```
#include <signal.h>

/* When a stop signal arrives, set the action back to the default
   and then resend the signal after doing clean-up actions. */

void
tstp_handler (int sig)
{
```

```
  signal (SIGTSTP, SIG_DFL);
  /* Do clean-up actions here. */
  ...
  raise (SIGTSTP);
}

/* When the process is continued again, restore the signal handler. */

void
cont_handler (int sig)
{
  signal (SIGCONT, cont_handler);
  signal (SIGTSTP, tstp_handler);
}


/* Enable both handlers during program initialization. */

int
main (void)
{
  signal (SIGCONT, cont_handler);
  signal (SIGTSTP, tstp_handler);
  ...
}
```

Portability Note: `raise` was invented by the ISO C committee. Older systems may not support it, so using `kill` may be more portable.

17.6.2 Signaling Another Process

The `kill` function can be used to send a signal to another process. In spite of its name, it can be used for a lot of things other than causing a process to terminate. Some examples of situations where you might want to send signals between processes are

- A parent process starts a child to perform a task—perhaps having the child running an infinite loop—and then terminates the child when the task is no longer needed.
- A process executes as part of a group, and needs to terminate or notify the other processes in the group when an error or other event occurs.
- Two processes need to synchronize while working together.

This section assumes that you know a little bit about how processes work. For more information on this subject, see Chapter 7 [Processes], page 209.

The `kill` function is declared in '`signal.h`'.

`int` **kill** (`pid_t` *pid*, `int` *signum*) Function

The `kill` function sends the signal *signum* to the process or process group specified by *pid*. Besides the signals listed in Section 17.2 [Standard Signals], page 379, *signum* can also have a value of zero to check the validity of the *pid*.

The *pid* specifies the process or process group to receive the signal:

pid > 0 Specify the process whose identifier is *pid*.

pid == 0 Specify all processes in the same process group as the sender.

pid < −1 Specify the process group whose identifier is −*pid*.

pid == −1 If the process is privileged, send the signal to all processes except for some special system-processes. Otherwise, send the signal to all processes with the same effective user-ID.

A process can send a signal to itself with a call like `kill (getpid(), signum)`. If `kill` is used by a process to send a signal to itself, and the signal is not blocked, then `kill` delivers at least one signal (which might be some other pending unblocked signal instead of the signal *signum*) to that process before it returns.

The return value from `kill` is 0 if the signal can be sent successfully. Otherwise, no signal is sent, and a value of −1 is returned. If *pid* specifies sending a signal to several processes, `kill` succeeds if it can send the signal to at least one of them. There's no way you can tell which of the processes got the signal or whether all of them did.

The following `errno` error conditions are defined for this function:

`EINVAL` The *signum* argument is an invalid or unsupported number.

`EPERM` You do not have the privilege to send a signal to the process or any of the processes in the process group named by *pid*.

`ESCRH` The *pid* argument does not refer to an existing process or group.

`int` **killpg** (`int` *pgid*, `int` *signum*) Function

This is similar to `kill`, but sends signal *signum* to the process group *pgid*. This function is provided for compatibility with BSD; using `kill` to do this is more portable.

As a simple example of `kill`, the call `kill (getpid (), sig)` has the same effect as `raise (sig)`.

17.6.3 Permission for Using `kill`

There are restrictions that prevent you from using `kill` to send signals to any random process. These are intended to prevent antisocial behavior such as arbitrarily killing off processes belonging to another user. In typical use, `kill` is used to pass signals between parent, child and sibling processes, and in these situations you normally do have permission to send signals. The only common exception is

when you run a setuid program in a child process; if the program changes its real UID as well as its effective UID, you may not have permission to send a signal. The `su` program does this.

Whether a process has permission to send a signal to another process is determined by the user IDs of the two processes. This concept is discussed in detail in Section 10.2 [The Persona of a Process], page 253.

Generally, for a process to be able to send a signal to another process, either the sending process must belong to a privileged user (like '`root`'), or the real or effective user-ID of the sending process must match the real or effective user-ID of the receiving process. If the receiving process has changed its effective user-ID from the set-user-ID mode bit on its process image file, then the owner of the process image file is used in place of its current effective user-ID. In some implementations, a parent process might be able to send signals to a child process even if the user IDs don't match, and other implementations might enforce other restrictions.

The `SIGCONT` signal is a special case. It can be sent if the sender is part of the same session as the receiver, regardless of user IDs.

17.6.4 Using `kill` for Communication

Here is a longer example showing how signals can be used for interprocess communication. This is what the `SIGUSR1` and `SIGUSR2` signals are provided for. Since these signals are fatal by default, the process that is supposed to receive them must trap them through `signal` or `sigaction`.

In this example, a parent process forks a child process and then waits for the child to complete its initialization. The child process tells the parent when it is ready by sending it a `SIGUSR1` signal, using the `kill` function.

```
#include <signal.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>


/* When a SIGUSR1 signal arrives, set this variable. */
volatile sig_atomic_t usr_interrupt = 0;

void
synch_signal (int sig)
{
  usr_interrupt = 1;
}

/* The child process executes this function. */
void
```

```
child_function (void)
{
  /* Perform initialization. */
  printf ("I'm here!!!  My pid is %d.\n", (int) getpid ());

  /* Let parent know you're done. */
  kill (getppid (), SIGUSR1);

  /* Continue with execution. */
  puts ("Bye, now....");
  exit (0);
}

int
main (void)
{
  struct sigaction usr_action;
  sigset_t block_mask;
  pid_t child_id;

  /* Establish the signal handler. */
  sigfillset (&block_mask);
  usr_action.sa_handler = synch_signal;
  usr_action.sa_mask = block_mask;
  usr_action.sa_flags = 0;
  sigaction (SIGUSR1, &usr_action, NULL);

  /* Create the child process. */
  child_id = fork ();
  if (child_id == 0)
    child_function ();          /* Does not return. */


  /* Busy wait for the child to send a signal. */
  while (!usr_interrupt)
    ;


  /* Now continue execution. */
  puts ("That's all, folks!");

  return 0;
}
```

This example uses a busy wait, which is bad, because it wastes CPU cycles that other programs could otherwise use. It is better to ask the system to wait until the signal arrives (see the example in Section 17.8 [Waiting for a Signal], page 421).

17.7 Blocking Signals

Blocking a signal means telling the operating system to hold it and deliver it later. Generally, a program does not block signals indefinitely—it might as well ignore them by setting their actions to `SIG_IGN`. But it is useful to block signals briefly, to prevent them from interrupting sensitive operations. For instance:

- You can use the `sigprocmask` function to block signals while you modify global variables that are also modified by the handlers for these signals.
- You can set `sa_mask` in your `sigaction` call to block certain signals while a particular signal-handler runs. This way, the signal handler can run without being interrupted itself by signals.

17.7.1 Why Blocking Signals Is Useful

Temporary blocking of signals with `sigprocmask` gives you a way to prevent interrupts during critical parts of your code. If signals arrive in that part of the program, they are delivered later, after you unblock them.

One example where this is useful is for sharing data between a signal handler and the rest of the program. If the type of the data is not `sig_atomic_t` (see Section 17.4.7 [Atomic Data-Access and Signal-Handling], page 406), then the signal handler could run when the rest of the program has only half-finished reading or writing the data. This would lead to confusing consequences.

To make the program reliable, you can prevent the signal handler from running while the rest of the program is examining or modifying that data—by blocking the appropriate signal around the parts of the program that touch the data.

Blocking signals is also necessary when you want to perform a certain action only if a signal has not arrived. Suppose that the handler for the signal sets a flag of type `sig_atomic_t`; you would like to test the flag and perform the action if the flag is not set. This is unreliable. Suppose the signal is delivered immediately after you test the flag, but before the consequent action—then the program will perform the action even though the signal has arrived.

The only way to test reliably for whether a signal has yet arrived is to test while the signal is blocked.

17.7.2 Signal Sets

All of the signal-blocking functions use a data structure called a *signal set* to specify what signals are affected. Thus, every activity involves two stages: creating the signal set, and then passing it as an argument to a library function.

These facilities are declared in the header file ‘`signal.h`’.

sigset_t Data Type

The `sigset_t` data type is used to represent a signal set. Internally, it may be implemented as either an integer or structure type.

For portability, use only the functions described in this section to initialize, change and retrieve information from `sigset_t` objects—don't try to manipulate them directly.

There are two ways to initialize a signal set. You can initially specify it to be empty with `sigemptyset` and then add specified signals individually. Or you can specify it to be full with `sigfillset` and then delete specified signals individually.

You must always initialize the signal set with one of these two functions before using it in any other way. Don't try to set all the signals explicitly, because the `sigset_t` object might include some other information, like a version field, that needs to be initialized as well. In addition, it's not wise to put into your program an assumption that the system has no signals aside from the ones you know about.

`int` **sigemptyset** (`sigset_t *`*set*) Function

This function initializes the signal set *set* to exclude all of the defined signals. It always returns `0`.

`int` **sigfillset** (`sigset_t *`*set*) Function

This function initializes the signal set *set* to include all of the defined signals. Again, the return value is `0`.

`int` **sigaddset** (`sigset_t *`*set*, `int` *signum*) Function

This function adds the signal *signum* to the signal set *set*. All `sigaddset` does is modify *set*; it does not block or unblock any signals.

The return value is `0` on success and `-1` on failure. The following `errno` error condition is defined for this function:

`EINVAL` The *signum* argument doesn't specify a valid signal.

`int` **sigdelset** (`sigset_t *`*set*, `int` *signum*) Function

This function removes the signal *signum* from the signal set *set*. All `sigdelset` does is modify *set*; it does not block or unblock any signals. The return value and error conditions are the same as for `sigaddset`.

Finally, there is a function to test what signals are in a signal set:

`int` **sigismember** (`const sigset_t *`*set*, `int` *signum*) Function

The `sigismember` function tests whether the signal *signum* is a member of the signal set *set*. It returns `1` if the signal is in the set, `0` if not and `-1` if there is an error.

The following `errno` error condition is defined for this function:

`EINVAL` The *signum* argument doesn't specify a valid signal.

17.7.3 Process Signal-Mask

The collection of signals that are currently blocked is called the *signal mask*. Each process has its own signal mask. When you create a new process (see Section 7.4 [Creating a Process], page 211), it inherits its parent's mask. You can block or unblock signals with total flexibility by modifying the signal mask.

The prototype for the `sigprocmask` function is in '`signal.h`'.

You must not use `sigprocmask` in multithreaded processes, because each thread has its own signal mask and there is no single process signal-mask. According to POSIX, the behavior of `sigprocmask` in a multithreaded process is "unspecified". Instead, use `pthread_sigmask`. (See Section 18.9 [Threads and Signal-Handling], page 447.)

`int` **sigprocmask** (`int` *how*, `const sigset_t *restrict` *set*, `sigset_t *restrict` *oldset*) — Function

The `sigprocmask` function is used to examine or change the calling process's signal-mask. The *how* argument determines how the signal mask is changed, and must be one of the following values:

`SIG_BLOCK`
: Block the signals in `set`—add them to the existing mask. In other words, the new mask is the union of the existing mask and *set*.

`SIG_UNBLOCK`
: Unblock the signals in *set*—remove them from the existing mask.

`SIG_SETMASK`
: Use *set* for the mask; ignore the previous value of the mask.

The last argument, *oldset*, is used to return information about the old process signal-mask. If you just want to change the mask without looking at it, pass a null pointer as the *oldset* argument. Similarly, if you want to know what's in the mask without changing it, pass a null pointer for *set* (in this case the *how* argument is not significant). The *oldset* argument is often used to remember the previous signal-mask in order to restore it later. Since the signal mask is inherited over `fork` and `exec` calls, you can't predict what its contents are when your program starts running.

If invoking `sigprocmask` causes any pending signals to be unblocked, at least one of those signals is delivered to the process before `sigprocmask` returns. The order in which pending signals are delivered is not specified, but you can control the order explicitly by making multiple `sigprocmask` calls to unblock various signals one at a time.

The `sigprocmask` function returns `0` if successful and `-1` to indicate an error. The following `errno` error condition is defined for this function:

`EINVAL` The *how* argument is invalid.

You can't block the `SIGKILL` and `SIGSTOP` signals, but if the signal set includes these, `sigprocmask` just ignores them instead of returning an error status.

Remember, too, that blocking program-error signals such as `SIGFPE` leads to undesirable results for signals generated by an actual program error (as opposed to signals sent with `raise` or `kill`). This is because your program may be too broken to be able to continue executing to a point where the signal is unblocked again (see Section 17.2.1 [Program-Error Signals], page 379).

17.7.4 Blocking to Test for Delivery of a Signal

Now for a simple example. Suppose you establish a handler for `SIGALRM` signals that sets a flag whenever a signal arrives, and your main program checks this flag from time to time and then resets it. You can prevent additional `SIGALRM` signals from arriving in the meantime by wrapping the critical part of the code with calls to `sigprocmask`, like this:

```
/* This variable is set by the SIGALRM signal-handler. */
volatile sig_atomic_t flag = 0;

int
main (void)
{
  sigset_t block_alarm;

  ...

  /* Initialize the signal mask. */
  sigemptyset (&block_alarm);
  sigaddset (&block_alarm, SIGALRM);


  while (1)
    {
      /* Check if a signal has arrived; if so, reset the flag. */
      sigprocmask (SIG_BLOCK, &block_alarm, NULL);
      if (flag)
        {
          actions-if-not-arrived
          flag = 0;
        }
      sigprocmask (SIG_UNBLOCK, &block_alarm, NULL);

      ...
    }
```

```
}
```

17.7.5 Blocking Signals for a Handler

When a signal handler is invoked, you usually want it to be able to finish without being interrupted by another signal. From the moment the handler starts until the moment it finishes, you must block signals that might confuse it or corrupt its data.

When a handler function is invoked on a signal, that signal is automatically blocked (in addition to any other signals that are already in the process's signal mask) during the time the handler is running. If you set up a handler for `SIGTSTP`, for instance, then the arrival of that signal forces further `SIGTSTP` signals to wait during the execution of the handler.

However, by default, other kinds of signals are not blocked; they can arrive during handler execution.

The reliable way to block other kinds of signals during the execution of the handler is to use the `sa_mask` member of the `sigaction` structure.

Here is an example:

```
#include <signal.h>
#include <stddef.h>

void catch_stop ();

void
install_handler (void)
{
  struct sigaction setup_action;
  sigset_t block_mask;

  sigemptyset (&block_mask);
  /* Block other terminal-generated signals while handler runs. */
  sigaddset (&block_mask, SIGINT);
  sigaddset (&block_mask, SIGQUIT);
  setup_action.sa_handler = catch_stop;
  setup_action.sa_mask = block_mask;
  setup_action.sa_flags = 0;
  sigaction (SIGTSTP, &setup_action, NULL);
}
```

This is more reliable than blocking the other signals explicitly in the code for the handler. If you block signals explicitly in the handler, you can't avoid at least a short interval at the beginning of the handler where they are not yet blocked.

You cannot remove signals from the process's current mask using this mechanism. However, you can make calls to `sigprocmask` within your handler to block or unblock signals as you wish.

In any case, when the handler returns, the system restores the mask that was in place before the handler was entered. If any signals that become unblocked by this restoration are pending, the process will receive those signals immediately, before returning to the code that was interrupted.

17.7.6 Checking for Pending Signals

You can find out which signals are pending at any time by calling `sigpending`. This function is declared in '`signal.h`'.

`int` **sigpending** (`sigset_t *`*set*) Function

The `sigpending` function stores information about pending signals in *set*. If there is a pending signal that is blocked from delivery, then that signal is a member of the returned set. You can test whether a particular signal is a member of this set using `sigismember` (see Section 17.7.2 [Signal Sets], page 414).

The return value is `0` if successful and `-1` on failure.

Testing whether a signal is pending is not often useful. Testing when that signal is not blocked is almost certainly bad design.

Here is an example:

```
#include <signal.h>
#include <stddef.h>

sigset_t base_mask, waiting_mask;

sigemptyset (&base_mask);
sigaddset (&base_mask, SIGINT);
sigaddset (&base_mask, SIGTSTP);

/* Block user interrupts while doing other processing. */
sigprocmask (SIG_SETMASK, &base_mask, NULL);
...

/* After a while, check to see whether any signals are pending. */
sigpending (&waiting_mask);
if (sigismember (&waiting_mask, SIGINT)) {
  /* The user has tried to kill the process. */
}
else if (sigismember (&waiting_mask, SIGTSTP)) {
  /* The user has tried to stop the process. */
}
```

Remember that if there is a particular signal pending for your process, additional signals of that same type that arrive in the meantime might be discarded. For example, if a `SIGINT` signal is pending when another `SIGINT` signal arrives, your program will probably only see one of them when you unblock this signal.

Portability Note: The `sigpending` function is new in POSIX.1. Older systems have no equivalent facility.

17.7.7 Remembering a Signal to Act on Later

Instead of blocking a signal using the library facilities, you can get almost the same results by making the handler set a flag to be tested later, when you "unblock". Here is an example:

```
/* If this flag is nonzero, don't handle the signal right away. */
volatile sig_atomic_t signal_pending;

/* This is nonzero if a signal arrived and was not handled. */
volatile sig_atomic_t defer_signal;

void
handler (int signum)
{
  if (defer_signal)
    signal_pending = signum;
  else
    ... /* "Really" handle the signal. */
}

...

void
update_mumble (int frob)
{
  /* Prevent signals from having immediate effect. */
  defer_signal++;
  /* Now update mumble, without worrying about interruption. */
  mumble.a = 1;
  mumble.b = hack ();
  mumble.c = frob;
  /* We have updated mumble. Handle any signal that came in. */
  defer_signal--;
  if (defer_signal == 0 && signal_pending != 0)
    raise (signal_pending);
}
```

Note how the particular signal that arrives is stored in `signal_pending`. That way, we can handle several types of inconvenient signals with the same mechanism.

We increment and decrement `defer_signal` so that nested critical sections will work properly; thus, if `update_mumble` were called with `signal_pending` already nonzero, signals would be deferred not only within `update_`

`mumble`, but also within the caller. This is also why we do not check `signal_pending` if `defer_signal` is still nonzero.

The incrementing and decrementing of `defer_signal` each require more than one instruction; it is possible for a signal to happen in the middle. But that does not cause any problem. If the signal happens early enough to see the value from before the increment or decrement, that is equivalent to a signal that came before the beginning of the increment or decrement, which is a case that works properly.

It is absolutely vital to decrement `defer_signal` before testing `signal_pending`, because this avoids a subtle bug. If we did these things in the other order, like this:

```
if (defer_signal == 1 && signal_pending != 0)
  raise (signal_pending);
defer_signal--;
```

then a signal arriving in between the `if` statement and the decrement would be effectively "lost" for an indefinite amount of time. The handler would merely set `defer_signal`—but the program having already tested this variable, it would not test the variable again.

Bugs like these are called *timing errors*. They are especially bad because they happen only rarely and are nearly impossible to reproduce. You can't expect to find them with a debugger as you would find a reproducible bug. So it is worth being especially careful to avoid them.

(You would not be tempted to write the code in this order, given the use of `defer_signal` as a counter which must be tested along with `signal_pending`. After all, testing for 0 is cleaner than testing for 1. But if you did not use `defer_signal` as a counter, and gave it values of 0 and 1 only, then either order might seem equally simple. This is a further advantage of using a counter for `defer_signal`: it will reduce the chance you will write the code in the wrong order and create a subtle bug.)

17.8 Waiting for a Signal

If your program is driven by external events, or uses signals for synchronization, then when it has nothing to do, it should probably wait until a signal arrives.

17.8.1 Using `pause`

The simple way to wait until a signal arrives is to call `pause`. Please read about its disadvantages, in the following section, before you use it.

`int` **pause** () Function

The `pause` function suspends program execution until a signal arrives whose action is either to execute a handler function or to terminate the process.

If the signal causes a handler function to be executed, then `pause` returns. This is considered an unsuccessful return (since "successful" behavior would be to

suspend the program forever), so the return value is `-1`. Even if you specify that other primitives should resume when a system handler returns (see Section 17.5 [Primitives Interrupted by Signals], page 408), this has no effect on `pause`; it always fails when a signal is handled.

The following `errno` error condition is defined for this function:

`EINTR` The function was interrupted by delivery of a signal.

If the signal causes program termination, `pause` doesn't return (obviously).

This function is a cancellation point in multithreaded programs. This is a problem if the thread allocates some resources (like memory, file descriptors, semaphores or whatever) at the time `pause` is called. If the thread gets cancelled these resources stay allocated until the program ends. To avoid this calls to `pause` should be protected using cancellation handlers.

The `pause` function is declared in '`unistd.h`'.

17.8.2 Problems with `pause`

The simplicity of `pause` can conceal serious timing errors that can make a program hang mysteriously.

It is safe to use `pause` if the real work of your program is done by the signal handlers themselves, and the "main program" does nothing but call `pause`. Each time a signal is delivered, the handler will do the next batch of work that is to be done, and then return, so that the main loop of the program can call `pause` again.

You can't safely use `pause` to wait until one more signal arrives, and then resume real work. Even if you arrange for the signal handler to cooperate by setting a flag, you still can't use `pause` reliably. Here is an example of this problem:

```
/* usr_interrupt is set by the signal handler.  */
if (!usr_interrupt)
  pause ();

/* Do work once the signal arrives.  */
...
```

This has a bug—the signal could arrive after the variable `usr_interrupt` is checked, but before the call to `pause`. If no further signals arrive, the process would never wake up again.

You can put an upper limit on the excess waiting by using `sleep` in a loop instead of using `pause`.[5] Here is what this looks like:

```
/* usr_interrupt is set by the signal handler.
while (!usr_interrupt)
  sleep (1);

/* Do work once the signal arrives.  */
```

[5] For more about `sleep`, see Loosemore et al., "Sleeping".

```
...
```

For some purposes, that is good enough. But with a little more complexity, you can wait reliably until a particular signal-handler is run, using `sigsuspend`.

17.8.3 Using `sigsuspend`

The clean and reliable way to wait for a signal to arrive is to block it and then use `sigsuspend`. By using `sigsuspend` in a loop, you can wait for certain kinds of signals, while letting other kinds of signals be handled by their handlers.

`int` **sigsuspend** (`const sigset_t *`*set*) Function

This function replaces the process's signal-mask with *set* and then suspends the process until a signal is delivered whose action is either to terminate the process or invoke a signal-handling function. In other words, the program is effectively suspended until one of the signals that is not a member of *set* arrives.

If the process is woken up by delivery of a signal that invokes a handler function, and the handler function returns, then `sigsuspend` also returns.

The mask remains *set* only as long as `sigsuspend` is waiting. The function `sigsuspend` always restores the previous signal-mask when it returns.

The return value and error conditions are the same as for `pause`.

With `sigsuspend`, you can replace the `pause` or `sleep` loop in the previous section with something completely reliable:

```
sigset_t mask, oldmask;

...

/* Set up the mask of signals to temporarily block. */
sigemptyset (&mask);
sigaddset (&mask, SIGUSR1);

...

/* Wait for a signal to arrive. */
sigprocmask (SIG_BLOCK, &mask, &oldmask);
while (!usr_interrupt)
  sigsuspend (&oldmask);
sigprocmask (SIG_UNBLOCK, &mask, NULL);
```

This last piece of code is a little tricky. The key point to remember here is that when `sigsuspend` returns, it resets the process's signal-mask to the original value, the value from before the call to `sigsuspend`—in this case, the `SIGUSR1` signal is once again blocked. The second call to `sigprocmask` is necessary to explicitly unblock this signal.

You may be wondering why the `while` loop is necessary at all, since the program is apparently only waiting for one `SIGUSR1` signal. The answer is that the

mask passed to `sigsuspend` permits the process to be woken up by the delivery of other kinds of signals, as well—for example, job-control signals. If the process is woken up by a signal that doesn't set `usr_interrupt`, it just suspends itself again until the "right" kind of signal eventually arrives.

This technique takes a few more lines of preparation, but that is needed just once for each kind of wait criterion you want to use. The code that actually waits is just four lines.

17.9 Using a Separate Signal-Stack

A signal stack is a special area of memory to be used as the execution stack during signal handlers. It should be fairly large, to avoid any danger that it will overflow in turn; the macro `SIGSTKSZ` is defined to a canonical size for signal stacks. You can use `malloc` to allocate the space for the stack. Then call `sigaltstack` or `sigstack` to tell the system to use that space for the signal stack.

You don't need to write signal handlers differently in order to use a signal stack. Switching from one stack to the other happens automatically. (Some non-GNU debuggers on some machines may get confused if you examine a stack trace while a handler that uses the signal stack is running.)

There are two interfaces for telling the system to use a separate signal stack. `sigstack` is the older interface, which comes from 4.2 BSD. `sigaltstack` is the newer interface, and comes from 4.4 BSD. The `sigaltstack` interface has the advantage that it does not require your program to know which direction the stack grows, which depends on the specific machine and operating system.

stack_t Data Type

This structure describes a signal stack. It contains the following members:

`void *ss_sp`
: This points to the base of the signal stack.

`size_t ss_size`
: This is the size (in bytes) of the signal stack that '`ss_sp`' points to. You should set this to however much space you allocated for the stack.

  There are two macros defined in '`signal.h`' that you should use in calculating this size:

  `SIGSTKSZ`
  : This is the canonical size for a signal stack. It is judged to be sufficient for normal uses.

  `MINSIGSTKSZ`
  : This is the amount of signal stack-space the operating system needs just to implement signal delivery. The size of a signal stack **must** be greater than this.

For most cases, just using `SIGSTKSZ` for `ss_size` is sufficient. But if you know how much stack space your program's signal handlers will need, you may want to use a different size. In this case, you should allocate `MINSIGSTKSZ` additional bytes for the signal stack and increase `ss_size` accordingly.

`int ss_flags`
: This field contains the bit-wise OR of these flags:

  `SS_DISABLE`
  : This tells the system that it should not use the signal stack.

  `SS_ONSTACK`
  : This is set by the system, and indicates that the signal stack is currently in use. If this bit is not set, then signals will be delivered on the normal user stack.

`int` **sigaltstack** (`const stack_t *restrict` *stack*, `stack_t *restrict` *oldstack*) Function

The `sigaltstack` function specifies an alternate stack for use during signal handling. When a signal is received by the process and its action indicates that the signal stack is used, the system arranges a switch to the currently installed signal-stack while the handler for that signal is executed.

If *oldstack* is not a null pointer, information about the currently installed signal-stack is returned in the location it points to. If *stack* is not a null pointer, then this is installed as the new stack for use by signal handlers.

The return value is `0` on success and `-1` on failure. If `sigaltstack` fails, it sets `errno` to one of these values:

`EINVAL` You tried to disable a stack that was in fact currently in use.

`ENOMEM` The size of the alternate stack was too small. It must be greater than `MINSIGSTKSZ`.

Here is the older `sigstack` interface. You should use `sigaltstack` instead on systems that have it.

struct sigstack Data Type

This structure describes a signal stack. It contains the following members:

`void *ss_sp`
: This is the stack pointer. If the stack grows downward on your machine, this should point to the top of the area you allocated. If the stack grows upward, it should point to the bottom.

`int ss_onstack`
: This field is true if the process is currently using this stack.

`int` **sigstack** (`const struct sigstack *`*stack*`, struct sigstack *`*oldstack*) Function

The `sigstack` function specifies an alternate stack for use during signal handling. When a signal is received by the process and its action indicates that the signal stack is used, the system arranges a switch to the currently installed signal-stack while the handler for that signal is executed.

If *oldstack* is not a null pointer, information about the currently installed signal-stack is returned in the location it points to. If *stack* is not a null pointer, then this is installed as the new stack for use by signal handlers.

The return value is `0` on success and `-1` on failure.

17.10 BSD Signal-Handling

This section describes alternative signal-handling functions derived from BSD Unix. These facilities were an advance, in their time; today, they are mostly obsolete, and supported mainly for compatibility with BSD Unix.

There are many similarities between the BSD and POSIX signal-handling facilities, because the POSIX facilities were inspired by the BSD facilities. Besides having different names for all the functions to avoid conflicts, the main differences between the two are

- BSD Unix represents signal masks as an `int` bit mask, rather than as a `sigset_t` object.
- The BSD facilities use a different default for whether an interrupted primitive should fail or resume. The POSIX facilities make system calls fail unless you specify that they should resume. With the BSD facility, the default is to make system calls resume unless you say they should fail (see Section 17.5 [Primitives Interrupted by Signals], page 408).

The BSD facilities are declared in '`signal.h`'.

17.10.1 BSD Function to Establish a Handler

struct sigvec Data Type

This data type is the BSD equivalent of `struct sigaction` (see Section 17.3.2 [Advanced Signal-Handling], page 392); it is used to specify signal actions to the `sigvec` function. It contains the following members:

`sighandler_t sv_handler`
: This is the handler function.

`int sv_mask`
: This is the mask of additional signals to be blocked while the handler function is being called.

`int sv_flags`
: This is a bit mask used to specify various flags that affect the behavior of the signal. You can also refer to this field as `sv_onstack`.

These symbolic constants can be used to provide values for the `sv_flags` field of a `sigvec` structure. This field is a bit-mask value, so you bit-wise-OR the flags of interest to you together.

`int` **SV_ONSTACK** Macro

If this bit is set in the `sv_flags` field of a `sigvec` structure, it means to use the signal stack when delivering the signal.

`int` **SV_INTERRUPT** Macro

If this bit is set in the `sv_flags` field of a `sigvec` structure, it means that system calls interrupted by this kind of signal should not be restarted if the handler returns; instead, the system calls should return with a `EINTR` error status (see Section 17.5 [Primitives Interrupted by Signals], page 408).

`int` **SV_RESETHAND** Macro

If this bit is set in the `sv_flags` field of a `sigvec` structure, it means to reset the action for the signal back to `SIG_DFL` when the signal is received.

`int` **sigvec** (`int` *signum*, `const struct sigvec` ⋆*action*, `struct sigvec` ⋆*old-action*) Function

This function is the equivalent of `sigaction` (see Section 17.3.2 [Advanced Signal-Handling], page 392); it installs the action *action* for the signal *signum*, returning information about the previous action in effect for that signal in *old-action*.

`int` **siginterrupt** (`int` *signum*, `int` *failflag*) Function

This function specifies which approach to use when certain primitives are interrupted by handling signal *signum*. If *failflag* is false, signal *signum* restarts primitives. If *failflag* is true, handling *signum* causes these primitives to fail with error code `EINTR` (see Section 17.5 [Primitives Interrupted by Signals], page 408).

17.10.2 BSD Functions for Blocking Signals

`int` **sigmask** (`int` *signum*) Macro

This macro returns a signal mask that has the bit for signal *signum* set. You can bit-wise-OR the results of several calls to `sigmask` together to specify more than one signal. For example:

```
(sigmask (SIGTSTP) | sigmask (SIGSTOP)
 | sigmask (SIGTTIN) | sigmask (SIGTTOU))
```

specifies a mask that includes all the job-control stop signals.

`int` **sigblock** (`int` *mask*) Function

This function is equivalent to `sigprocmask` (see Section 17.7.3 [Process Signal-Mask], page 416) with a *how* argument of `SIG_BLOCK`—it adds the

signals specified by *mask* to the calling process's set of blocked signals. The return value is the previous set of blocked signals.

`int` **sigsetmask** (`int` *mask*) Function

This function equivalent to `sigprocmask` (see Section 17.7.3 [Process Signal Mask], page 416) with a *how* argument of `SIG_SETMASK`—it sets the calling process's signal mask to *mask*. The return value is the previous set of blocked signals.

`int` **sigpause** (`int` *mask*) Function

This function is the equivalent of `sigsuspend` (see Section 17.8 [Waiting for a Signal], page 421)—it sets the calling process's signal mask to *mask*, and waits for a signal to arrive. On return, the previous set of blocked signals is restored.

18 POSIX Threads

This chapter describes the pthreads (POSIX threads) library. This library provides support functions for multithreaded programs: thread primitives, synchronization objects, and so forth. It also implements POSIX 1003.1b semaphores (not to be confused with System V semaphores).

The threads operations ('`pthread_*`') do not use *errno*. Instead they return an error code directly. The semaphore operations do use *errno*.

18.1 Basic Thread Operations

These functions are the thread equivalents of `fork`, `exit` and `wait`.

`int` **pthread_create** (`pthread_t *` *thread*, `pthread_attr_t *` *attr*, `void *` (**start_routine*)(`void *`), `void *` *arg*) Function

`pthread_create` creates a new thread of control that executes concurrently with the calling thread. The new thread calls the function *start_routine*, passing it *arg* as first argument. The new thread terminates either explicitly, by calling `pthread_exit`, or implicitly, by returning from the *start_routine* function. The latter case is equivalent to calling `pthread_exit` with the result returned by *start_routine* as exit code.

The *attr* argument specifies thread attributes to be applied to the new thread (see Section 18.2 [Thread Attributessection Thread Attributes], page 430). The *attr* argument can also be `NULL`, in which case default attributes are used: the created thread is joinable (not detached) and has an ordinary (not real-time) scheduling policy.

On success, the identifier of the newly created thread is stored in the location pointed by the *thread* argument, and a 0 is returned. On error, a nonzero error code is returned.

This function may return the following errors

`EAGAIN` There are not enough system resources to create a process for the new thread, or more than `PTHREAD_THREADS_MAX` threads are already active.

`void` **pthread_exit** (`void *`*retval*) Function

`pthread_exit` terminates the execution of the calling thread. All clean-up handlers (see Section 18.4 [Clean-Up Handlers], page 435) that have been set for the calling thread with `pthread_cleanup_push` are executed in reverse order (the most recently pushed handler is executed first). Finalization functions for thread-specific data are then called for all keys that have non-`NULL` values associated with them in the calling thread (see Section 18.8 [Thread-Specific Data], page 445). Finally, execution of the calling thread is stopped.

The *retval* argument is the return value of the thread. It can be retrieved from another thread using `pthread_join`.

The `pthread_exit` function never returns.

`int` **pthread_cancel** (`pthread_t` *thread*) Function

`pthread_cancel` sends a cancellation request to the thread denoted by the *thread* argument. If there is no such thread, `pthread_cancel` fails and returns `ESRCH`. Otherwise, it returns 0 (see Section 18.3 [Cancellation], page 433).

`int` **pthread_join** (`pthread_t` *th*, `void **`*thread_return*) Function

`pthread_join` suspends the execution of the calling thread until the thread identified by *th* terminates, either by calling `pthread_exit` or by being canceled.

If *thread_return* is not `NULL`, the return value of *th* is stored in the location pointed to by *thread_return*. The return value of *th* is either the argument it gave to `pthread_exit`, or `PTHREAD_CANCELED` if *th* was canceled.

The joined thread `th` must be in the joinable state—it must not have been detached using `pthread_detach` or the `PTHREAD_CREATE_DETACHED` attribute to `pthread_create`.

When a joinable thread terminates, its memory resources (thread descriptor and stack) are not deallocated until another thread performs `pthread_join` on it. Therefore, `pthread_join` must be called once for each joinable thread created to avoid memory leaks.

At most one thread can wait for the termination of a given thread. Calling `pthread_join` on a thread *th* on which another thread is already waiting for termination returns an error.

`pthread_join` is a cancellation point. If a thread is canceled while suspended in `pthread_join`, the thread execution resumes immediately and the cancellation is executed without waiting for the *th* thread to terminate. If cancellation occurs during `pthread_join`, the *th* thread remains not joined.

On success, the return value of *th* is stored in the location pointed to by *thread_return*, and 0 is returned. On error, one of the following values is returned:

`ESRCH` No thread could be found corresponding to that specified by *th*.

`EINVAL` The *th* thread has been detached, or another thread is already waiting on termination of *th*.

`EDEADLK` The *th* argument refers to the calling thread.

18.2 Thread Attributessection Thread Attributes

Threads have a number of attributes that may be set at creation time. This is done by filling a thread-attribute object *attr* of type `pthread_attr_t`, then passing it

as second argument to `pthread_create`. Passing `NULL` is equivalent to passing a thread-attribute object with all attributes set to their default values.

Attribute objects are consulted only when creating a new thread. The same attribute-object can be used for creating several threads. Modifying an attribute object after a call to `pthread_create` does not change the attributes of the thread previously created.

`int` **pthread_attr_init** (`pthread_attr_t *`*attr*) Function

`pthread_attr_init` initializes the thread-attribute object *attr* and fills it with default values for the attributes. (The default values for each attribute are listed below.)

Each attribute *attrname* (see below for a list of all attributes) can be individually set using the function `pthread_attr_set`*attrname* and retrieved using the function `pthread_attr_get`*attrname*.

`int` **pthread_attr_destroy** (`pthread_attr_t *`*attr*) Function

`pthread_attr_destroy` destroys the attribute object pointed to by *attr*, releasing any resources associated with it. *attr* is left in an undefined state, and you must not use it again in a call to any pthreads function until it has been reinitialized.

`int` **pthread_attr_setattr** (`pthread_attr_t *`*obj*`, int` *value*) Function

Set attribute *attr* to *value* in the attribute object pointed to by *obj*. See below for a list of possible attributes and the values they can take.

On success, these functions return 0. If *value* is not meaningful for the *attr* being modified, they will return the error code `EINVAL`. Some of the functions have other failure modes; see below.

`int` **pthread_attr_getattr** (`const pthread_attr_t *`*obj*`, int *`*value*) Function

Store the current setting of *attr* in *obj* into the variable pointed to by *value*.

These functions always return 0.

The following thread attributes are supported:

‘`detachstate`’

Choose whether the thread is created in the joinable state (value `PTHREAD_CREATE_JOINABLE`) or in the detached state (`PTHREAD_CREATE_DETACHED`). The default is `PTHREAD_CREATE_JOINABLE`.

In the joinable state, another thread can synchronize on the thread termination and recover its termination code using `pthread_join`,

but some of the thread resources are kept allocated after the thread terminates, and reclaimed only when another thread performs `pthread_join` on that thread.

In the detached state, the thread resources are immediately freed when it terminates, but `pthread_join` cannot be used to synchronize on the thread termination.

A thread created in the joinable state can later be put in the detached thread using `pthread_detach`.

'`schedpolicy`'
Select the scheduling policy for the thread, one of `SCHED_OTHER` (regular, non-real-time scheduling), `SCHED_RR` (real-time, round-robin) or `SCHED_FIFO` (real-time, first-in first-out). The default is `SCHED_OTHER`.

The real-time scheduling policies `SCHED_RR` and `SCHED_FIFO` are available only to processes with superuser privileges. `pthread_attr_setschedparam` will fail and return `ENOTSUP` if you try to set a real-time policy when you are unprivileged.

The scheduling policy of a thread can be changed after creation with `pthread_setschedparam`.

'`schedparam`'
Change the scheduling parameter (the scheduling priority) for the thread. The default is 0.

This attribute is not significant if the scheduling policy is `SCHED_OTHER`; it only matters for the real-time policies `SCHED_RR` and `SCHED_FIFO`.

The scheduling priority of a thread can be changed after creation with `pthread_setschedparam`.

'`inheritsched`'
Choose whether the scheduling policy and scheduling parameter for the newly created thread are determined by the values of the *schedpolicy* and *schedparam* attributes (value `PTHREAD_EXPLICIT_SCHED`) or are inherited from the parent thread (value `PTHREAD_INHERIT_SCHED`). The default is `PTHREAD_EXPLICIT_SCHED`.

'`scope`' Choose the scheduling-contention scope for the created thread. The default is `PTHREAD_SCOPE_SYSTEM`, meaning that the threads contend for CPU time with all processes running on the machine. In particular, thread priorities are interpreted relative to the priorities of all other processes on the machine. The other possibility, `PTHREAD_SCOPE_PROCESS`, means that scheduling contention occurs only between the threads of the running process—thread priorities are interpreted relative to the priorities of the other threads of the process, regardless of the priorities of other processes.

`PTHREAD_SCOPE_PROCESS` is not supported in LinuxThreads. If you try to set the scope to this value, `pthread_attr_setscope` will fail and return `ENOTSUP`.

'`stackaddr`'
Provide an address for an application-managed stack. The size of the stack must be at least `PTHREAD_STACK_MIN`.

'`stacksize`'
Change the size of the stack created for the thread. The value defines the minimum-stack size, in bytes.

If the value exceeds the system's maximum stack-size, or is smaller than `PTHREAD_STACK_MIN`, `pthread_attr_setstacksize` will fail and return `EINVAL`.

'`stack`' Provide both the address and size of an application-managed stack to use for the new thread. The base of the memory area is *stackaddr* with the size of the memory area, *stacksize*, measured in bytes.

If the value of *stacksize* is less than `PTHREAD_STACK_MIN`, or greater than the system's maximum stack size, or if the value of *stackaddr* lacks the proper alignment, `pthread_attr_setstack` will fail and return `EINVAL`.

'`guardsize`'
Change the minimum size in bytes of the guard area for the thread's stack. The default size is a single page. If this value is set, it will be rounded up to the nearest page size. If the value is set to 0, a guard area will not be created for this thread. The space allocated for the guard area is used to catch stack overflow. Therefore, when allocating large structures on the stack, a larger guard area may be required to catch a stack overflow.

If the caller is managing their own stacks (if the `stackaddr` attribute has been set), then the `guardsize` attribute is ignored.

If the value exceeds the `stacksize`, `pthread_atrr_setguardsize` will fail and return `EINVAL`.

18.3 Cancellation

Cancellation is the mechanism by which a thread can terminate the execution of another thread. More precisely, a thread can send a cancellation request to another thread. Depending on its settings, the target thread can then either ignore the request, honor it immediately, or defer it till it reaches a cancellation point. When threads are first created by `pthread_create`, they always defer cancellation requests.

When a thread eventually honors a cancellation request, it behaves as if `pthread_exit(PTHREAD_CANCELED)` was called. All clean-up handlers are executed in reverse order, finalization functions for thread-specific data are

called, and finally the thread stops executing. If the canceled thread was joinable, the return value `PTHREAD_CANCELED` is provided to whichever thread calls *pthread_join* on it (see Section 18.1 [Basic Thread Operations], page 429).

Cancellation points are the points where the thread checks for pending cancellation-requests and performs them. The POSIX threads functions `pthread_join`, `pthread_cond_wait`, `pthread_cond_timedwait`, `pthread_testcancel`, `sem_wait` and `sigwait` are cancellation points. In addition, these system calls are cancellation points:

```
accept          open            sendmsg
close           pause           sendto
connect         read            system
fcntl           recv            tcdrain
fsync           recvfrom        wait
lseek           recvmsg         waitpid
msync           send            write
nanosleep
```

All library functions that call these functions (such as `printf`) are also cancellation points.

`int` **pthread_setcancelstate** (`int` *state*, `int *`*oldstate*) Function

`pthread_setcancelstate` changes the cancellation state for the calling thread—that is, whether cancellation requests are ignored or not. The *state* argument is the new cancellation state—either `PTHREAD_CANCEL_ENABLE` to enable cancellation, or `PTHREAD_CANCEL_DISABLE` to disable cancellation (cancellation requests are ignored).

If *oldstate* is not `NULL`, the previous cancellation-state is stored in the location pointed to by *oldstate*, and can thus be restored later by another call to `pthread_setcancelstate`.

If the *state* argument is not `PTHREAD_CANCEL_ENABLE` or `PTHREAD_CANCEL_DISABLE`, `pthread_setcancelstate` fails and returns `EINVAL`. Otherwise, it returns 0.

`int` **pthread_setcanceltype** (`int` *type*, `int *`*oldtype*) Function

`pthread_setcanceltype` changes the type of responses to cancellation requests for the calling thread: asynchronous (immediate) or deferred. The *type* argument is the new cancellation-type—either `PTHREAD_CANCEL_ASYNCHRONOUS` to cancel the calling thread as soon as the cancellation request is received, or `PTHREAD_CANCEL_DEFERRED` to keep the cancellation request pending until the next cancellation-point. If *oldtype* is not `NULL`, the previous cancellation-state is stored in the location pointed to by *oldtype*, and can thus be restored later by another call to `pthread_setcanceltype`.

If the *type* argument is not `PTHREAD_CANCEL_DEFERRED` or `PTHREAD_CANCEL_ASYNCHRONOUS`, `pthread_setcanceltype` fails and returns `EINVAL`. Otherwise, it returns 0.

`void` **pthread_testcancel** (*void*) Function

`pthread_testcancel` does nothing except testing for pending cancellation and executing it. Its purpose is to introduce explicit checks for cancellation in long sequences of code that do not call cancellation point functions otherwise.

18.4 Clean-Up Handlers

Clean-up handlers are functions that get called when a thread terminates, either by calling `pthread_exit` or because of cancellation. Clean-up handlers are installed and removed following a stack-like discipline.

The purpose of clean-up handlers is to free the resources that a thread may hold at the time it terminates. In particular, if a thread exits or is canceled while it owns a locked mutex, the mutex will remain locked forever and prevent other threads from executing normally. The best way to avoid this is, just before locking the mutex, to install a clean-up handler whose effect is to unlock the mutex. Clean-up handlers can be used similarly to free blocks allocated with `malloc` or close file descriptors on thread termination.

Here is how to lock a mutex *mut* in such a way that it will be unlocked if the thread is canceled while *mut* is locked:

```
pthread_cleanup_push(pthread_mutex_unlock, (void *) &mut);
pthread_mutex_lock(&mut);
/* do some work */
pthread_mutex_unlock(&mut);
pthread_cleanup_pop(0);
```

Equivalently, the last two lines can be replaced by:

```
pthread_cleanup_pop(1);
```

Notice that the code above is safe only in deferred-cancellation mode (see `pthread_setcanceltype`). In asynchronous-cancellation mode, a cancellation can occur between `pthread_cleanup_push` and `pthread_mutex_lock`, or between `pthread_mutex_unlock` and `pthread_cleanup_pop`, resulting in both cases in the thread trying to unlock a mutex not locked by the current thread. This is the main reason why asynchronous cancellation is difficult to use.

If the code above must also work in asynchronous-cancellation mode, then it must switch to deferred mode for locking and unlocking the mutex:

```
pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, &oldtype);
pthread_cleanup_push(pthread_mutex_unlock, (void *) &mut);
pthread_mutex_lock(&mut);
/* do some work */
pthread_cleanup_pop(1);
pthread_setcanceltype(oldtype, NULL);
```

The code above can be rewritten in a more compact and efficient way, using the nonportable functions `pthread_cleanup_push_defer_np` and `pthread_cleanup_pop_restore_np`:

```
pthread_cleanup_push_defer_np(pthread_mutex_unlock, (void *) &mut);
pthread_mutex_lock(&mut);
/* do some work */
pthread_cleanup_pop_restore_np(1);
```

`void` **pthread_cleanup_push** (`void` (**routine*) (`void *`), `void *`*arg*) Function

`pthread_cleanup_push` installs the *routine* function with argument *arg* as a clean-up handler. From this point on to the matching `pthread_cleanup_pop`, the function *routine* will be called with arguments *arg* when the thread terminates, either through `pthread_exit` or by cancellation. If several clean-up handlers are active at that point, they are called in LIFO order—the most recently installed handler is called first.

`void` **pthread_cleanup_pop** (`int` *execute*) Function

`pthread_cleanup_pop` removes the most recently installed clean-up handler. If the *execute* argument is not 0, it also executes the handler, by calling the *routine* function with arguments *arg*. If the *execute* argument is 0, the handler is only removed but not executed.

Matching pairs of `pthread_cleanup_push` and `pthread_cleanup_pop` must occur in the same function, at the same level of block nesting. Actually, `pthread_cleanup_push` and `pthread_cleanup_pop` are macros, and the expansion of `pthread_cleanup_push` introduces an open brace `{` with the matching closing brace `}` being introduced by the expansion of the matching `pthread_cleanup_pop`.

`void` **pthread_cleanup_push_defer_np** (`void` (**routine*) (`void *`), `void *`*arg*) Function

`pthread_cleanup_push_defer_np` is a nonportable extension that combines `pthread_cleanup_push` and `pthread_setcanceltype`. It pushes a clean-up handler just as `pthread_cleanup_push` does, but also saves the current cancellation type and sets it to deferred cancellation. This ensures that the clean-up mechanism is effective even if the thread was initially in asynchronous cancellation mode.

`void` **pthread_cleanup_pop_restore_np** (`int` *execute*) Function

`pthread_cleanup_pop_restore_np` pops a clean-up handler introduced by `pthread_cleanup_push_defer_np`, and restores the cancellation type to its value at the time `pthread_cleanup_push_defer_np` was called.

`pthread_cleanup_push_defer_np` and `pthread_cleanup_pop_restore_np` must occur in matching pairs, at the same level of block nesting.

The sequence:

```
pthread_cleanup_push_defer_np(routine, arg);
...
pthread_cleanup_pop_defer_np(execute);
```

is functionally equivalent to (but more compact and efficient than):

```
{
  int oldtype;
  pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, &oldtype);
  pthread_cleanup_push(routine, arg);
  ...
  pthread_cleanup_pop(execute);
  pthread_setcanceltype(oldtype, NULL);
}
```

18.5 Mutexes

A mutex is a MUTual EXclusion device, and is useful for protecting shared data-structures from concurrent modifications, and implementing critical sections and monitors.

A mutex has two possible states: unlocked (not owned by any thread), and locked (owned by one thread). A mutex can never be owned by two different threads simultaneously. A thread attempting to lock a mutex that is already locked by another thread is suspended until the owning thread unlocks the mutex first.

None of the mutex functions is a cancellation point, not even `pthread_mutex_lock`, in spite of the fact that it can suspend a thread for arbitrary durations. This way, the status of mutexes at cancellation points is predictable, allowing cancellation handlers to unlock precisely those mutexes that need to be unlocked before the thread stops executing. Consequently, threads using deferred cancellation should never hold a mutex for extended periods of time.

It is not safe to call mutex functions from a signal handler. In particular, calling `pthread_mutex_lock` or `pthread_mutex_unlock` from a signal handler may deadlock the calling thread.

`int` **pthread_mutex_init** (`pthread_mutex_t *`*mutex*`, const pthread_mutexattr_t *`*mutexattr*) Function

`pthread_mutex_init` initializes the mutex object pointed to by *mutex* according to the mutex attributes specified in *mutexattr*. If *mutexattr* is `NULL`, default attributes are used instead.

The LinuxThreads implementation supports only one mutex attribute, the *mutex type*, which is either "fast", "recursive", or "error checking". The type of a mutex determines whether it can be locked again by a thread that already owns it. The default type is "fast".

Variables of type `pthread_mutex_t` can also be initialized statically, using the constants `PTHREAD_MUTEX_INITIALIZER` (for timed mutexes), `PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP` (for recursive

mutexes), PTHREAD_ADAPTIVE_MUTEX_INITIALIZER_NP (for fast mutexes) and PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP (for error-checking mutexes).

pthread_mutex_init always returns 0.

int **pthread_mutex_lock** (pthread_mutex_t *mutex)) Function

pthread_mutex_lock locks the given mutex. If the mutex is currently unlocked, it becomes locked and owned by the calling thread, and pthread_mutex_lock returns immediately. If the mutex is already locked by another thread, pthread_mutex_lock suspends the calling thread until the mutex is unlocked.

If the mutex is already locked by the calling thread, the behavior of pthread_mutex_lock depends on the type of the mutex. If the mutex is of the "fast" type, the calling thread is suspended. It will remain suspended forever, because no other thread can unlock the mutex. If the mutex is of the "error-checking" type, pthread_mutex_lock returns immediately with the error code EDEADLK. If the mutex is of the "recursive" type, pthread_mutex_lock succeeds and returns immediately, recording the number of times the calling thread has locked the mutex. An equal number of pthread_mutex_unlock operations must be performed before the mutex returns to the unlocked state.

int **pthread_mutex_trylock** (pthread_mutex_t *mutex) Function

pthread_mutex_trylock behaves identically to pthread_mutex_lock, except that it does not block the calling thread if the mutex is already locked by another thread (or by the calling thread in the case of a "fast" mutex). Instead, pthread_mutex_trylock returns immediately with the error code EBUSY.

int **pthread_mutex_timedlock** (pthread_mutex_t *mutex, const struct timespec *abstime) Function

The pthread_mutex_timedlock is similar to the pthread_mutex_lock function, but instead of blocking for in indefinite time if the mutex is locked by another thread, it returns when the time specified in *abstime* is reached.

This function can only be used on standard ("timed") and "error-checking" mutexes. It behaves just like pthread_mutex_lock for all other types.

If the mutex is successfully locked, the function returns 0. If the time specified in *abstime* is reached without the mutex being locked, ETIMEDOUT is returned.

This function was introduced in the POSIX.1d revision of the POSIX standard.

int **pthread_mutex_unlock** (pthread_mutex_t *mutex) Function

pthread_mutex_unlock unlocks the given mutex. The mutex is assumed to be locked and owned by the calling thread on entrance to pthread_mutex_unlock. If the mutex is of the "fast" type, pthread_mutex_unlock always returns it to the unlocked state. If it is of the "recursive" type,

it decrements the locking count of the mutex (number of `pthread_mutex_lock` operations performed on it by the calling thread), and only when this count reaches 0 is the mutex actually unlocked.

On "error-checking" mutexes, `pthread_mutex_unlock` actually checks at run-time that the mutex is locked on entrance, and that it was locked by the same thread that is now calling `pthread_mutex_unlock`. If these conditions are not met, `pthread_mutex_unlock` returns `EPERM`, and the mutex remains unchanged. "Fast" and "recursive" mutexes perform no such checks, thus allowing a locked mutex to be unlocked by a thread other than its owner. This is nonportable behavior and must not be relied upon.

`int` **pthread_mutex_destroy** (`pthread_mutex_t *`*mutex*) Function

`pthread_mutex_destroy` destroys a mutex object, freeing the resources it might hold. The mutex must be unlocked on entrance. In the LinuxThreads implementation, no resources are associated with mutex objects, thus `pthread_mutex_destroy` actually does nothing except check that the mutex is unlocked.

If the mutex is locked by some thread, `pthread_mutex_destroy` returns `EBUSY`. Otherwise, it returns 0.

If any of the above functions (except `pthread_mutex_init`) is applied to an uninitialized mutex, they will simply return `EINVAL` and do nothing.

A shared global-variable *x* can be protected by a mutex as follows:

```
int x;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
```

All accesses and modifications to *x* should be bracketed by calls to `pthread_mutex_lock` and `pthread_mutex_unlock` as follows:

```
pthread_mutex_lock(&mut);
/* operate on x */
pthread_mutex_unlock(&mut);
```

Mutex attributes can be specified at mutex creation time, by passing a mutex-attribute object as second argument to `pthread_mutex_init`. Passing `NULL` is equivalent to passing a mutex attribute object with all attributes set to their default values.

`int` **pthread_mutexattr_init** (`pthread_mutexattr_t *`*attr*) Function

`pthread_mutexattr_init` initializes the mutex-attribute object *attr* and fills it with default values for the attributes.

This function always returns 0.

`int` **pthread_mutexattr_destroy** (`pthread_mutexattr_t *`*attr*) Function

`pthread_mutexattr_destroy` destroys a mutex-attribute object, which must not be reused until it is reinitialized. `pthread_mutexattr_destroy` does nothing in the LinuxThreads implementation.

This function always returns 0.

LinuxThreads supports only one mutex attribute: the mutex type, which is either `PTHREAD_MUTEX_ADAPTIVE_NP` for "fast" mutexes, `PTHREAD_MUTEX_RECURSIVE_NP` for "recursive" mutexes, `PTHREAD_MUTEX_TIMED_NP` for "timed" mutexes or `PTHREAD_MUTEX_ERRORCHECK_NP` for "error-checking" mutexes. As the `NP` suffix indicates, this is a nonportable extension to the POSIX standard and should not be employed in portable programs.

The mutex type determines what happens if a thread attempts to lock a mutex it already owns with `pthread_mutex_lock`. If the mutex is of the "fast" type, `pthread_mutex_lock` simply suspends the calling thread forever. If the mutex is of the "error-checking" type, `pthread_mutex_lock` returns immediately with the error code `EDEADLK`. If the mutex is of the "recursive" type, the call to `pthread_mutex_lock` returns immediately with a success return code. The number of times the thread owning the mutex has locked it is recorded in the mutex. The owning thread must call `pthread_mutex_unlock` the same number of times before the mutex returns to the unlocked state.

The default mutex type is "timed", that is, `PTHREAD_MUTEX_TIMED_NP`.

`int` **pthread_mutexattr_settype** (`pthread_mutexattr_t *`*attr*`, int` *type*) Function

`pthread_mutexattr_settype` sets the mutex type attribute in *attr* to the value specified by *type*.

If *type* is not `PTHREAD_MUTEX_ADAPTIVE_NP`, `PTHREAD_MUTEX_RECURSIVE_NP`, `PTHREAD_MUTEX_TIMED_NP` or `PTHREAD_MUTEX_ERRORCHECK_NP`, this function will return `EINVAL` and leave *attr* unchanged.

The standard Unix98 identifiers `PTHREAD_MUTEX_DEFAULT`, `PTHREAD_MUTEX_NORMAL`, `PTHREAD_MUTEX_RECURSIVE` and `PTHREAD_MUTEX_ERRORCHECK` are also permitted.

`int` **pthread_mutexattr_gettype** (`const pthread_mutexattr_t *`*attr*`, int *`*type*) Function

`pthread_mutexattr_gettype` retrieves the current value of the mutex type attribute in *attr* and stores it in the location pointed to by *type*.

This function always returns 0.

18.6 Condition Variables

A condition (short for "condition variable") is a synchronization device that allows threads to suspend execution until some predicate on shared data is satisfied. The basic operations on conditions are signal the condition (when the predicate becomes true), and wait for the condition, suspending the thread execution until another thread signals the condition.

A condition variable must always be associated with a mutex, to avoid the race condition where a thread prepares to wait on a condition variable and another thread signals the condition just before the first thread actually waits on it.

`int` **pthread_cond_init** (`pthread_cond_t *`*cond*`,` `pthread_condattr_t *`*cond_attr*) Function

`pthread_cond_init` initializes the condition variable *cond*, using the condition attributes specified in *cond_attr*, or default attributes if *cond_attr* is `NULL`. The LinuxThreads implementation supports no attributes for conditions, hence the *cond_attr* parameter is actually ignored.

Variables of type `pthread_cond_t` can also be initialized statically, using the constant `PTHREAD_COND_INITIALIZER`.

This function always returns 0.

`int` **pthread_cond_signal** (`pthread_cond_t *`*cond*) Function

`pthread_cond_signal` restarts one of the threads that are waiting on the condition variable *cond*. If no threads are waiting on *cond*, nothing happens. If several threads are waiting on *cond*, exactly one is restarted, but it is not specified which.

This function always returns 0.

`int` **pthread_cond_broadcast** (`pthread_cond_t *`*cond*) Function

`pthread_cond_broadcast` restarts all the threads that are waiting on the condition variable *cond*. Nothing happens if no threads are waiting on *cond*.

This function always returns 0.

`int` **pthread_cond_wait** (`pthread_cond_t *`*cond*`,` `pthread_mutex_t *`*mutex*) Function

`pthread_cond_wait` atomically unlocks the *mutex* (as per `pthread_unlock_mutex`) and waits for the condition variable *cond* to be signaled. The thread execution is suspended and does not consume any CPU time until the condition variable is signaled. The *mutex* must be locked by the calling thread on entrance to `pthread_cond_wait`. Before returning to the calling thread, `pthread_cond_wait` reacquires *mutex* (as per `pthread_lock_mutex`).

Unlocking the mutex and suspending on the condition variable is done atomically. Thus, if all threads always acquire the mutex before signaling the condition, this guarantees that the condition cannot be signaled (and thus ignored)

between the time a thread locks the mutex and the time it waits on the condition variable.

This function always returns 0.

`int` **pthread_cond_timedwait** (`pthread_cond_t *`*cond*`, pthread_mutex_t *`*mutex*`, const struct timespec *`*abstime*) Function

`pthread_cond_timedwait` atomically unlocks *mutex* and waits on *cond*, as `pthread_cond_wait` does, but it also bounds the duration of the wait. If *cond* has not been signaled before time *abstime*, the mutex *mutex* is reacquired and `pthread_cond_timedwait` returns the error code `ETIMEDOUT`. The wait can also be interrupted by a signal; in that case, `pthread_cond_timedwait` returns `EINTR`.

The *abstime* parameter specifies an absolute time, with the same origin as `time` and `gettimeofday`: an *abstime* of 0 corresponds to 00:00:00 GMT, January 1, 1970.

`int` **pthread_cond_destroy** (`pthread_cond_t *`*cond*) Function

`pthread_cond_destroy` destroys the condition variable *cond*, freeing the resources it might hold. If any threads are waiting on the condition variable, `pthread_cond_destroy` leaves *cond* untouched and returns `EBUSY`. Otherwise it returns 0, and *cond* must not be used again until it is reinitialized.

In the LinuxThreads implementation, no resources are associated with condition variables, so `pthread_cond_destroy` actually does nothing.

`pthread_cond_wait` and `pthread_cond_timedwait` are cancellation points. If a thread is canceled while suspended in one of these functions, the thread immediately resumes execution, relocks the mutex specified by *mutex*, and finally executes the cancellation. Consequently, clean-up handlers are assured that *mutex* is locked when they are called.

It is not safe to call the condition variable functions from a signal handler. In particular, calling `pthread_cond_signal` or `pthread_cond_broadcast` from a signal handler may deadlock the calling thread.

Consider two shared variables *x* and *y*, protected by the mutex *mut*, and a condition variable *cond* that is to be signaled whenever *x* becomes greater than *y*.

```
int x,y;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Waiting until *x* is greater than *y* is performed as follows:

```
pthread_mutex_lock(&mut);
while (x <= y) {
        pthread_cond_wait(&cond, &mut);
}
/* operate on x and y */
```

```
pthread_mutex_unlock(&mut);
```

Modifications on *x* and *y* that may cause *x* to become greater than *y* should signal the condition if needed:

```
pthread_mutex_lock(&mut);
/* modify x and y */
if (x > y) pthread_cond_broadcast(&cond);
pthread_mutex_unlock(&mut);
```

If it can be proved that at most one waiting thread needs to be awakened (for instance, if there are only two threads communicating through *x* and *y*), `pthread_cond_signal` can be used as a slightly more efficient alternative to `pthread_cond_broadcast`. When in doubt, use `pthread_cond_broadcast`.

To wait for *x* to becomes greater than *y* with a time-out of 5 seconds, do:

```
struct timeval now;
struct timespec timeout;
int retcode;

pthread_mutex_lock(&mut);
gettimeofday(&now);
timeout.tv_sec = now.tv_sec + 5;
timeout.tv_nsec = now.tv_usec * 1000;
retcode = 0;
while (x <= y && retcode != ETIMEDOUT) {
        retcode = pthread_cond_timedwait(&cond, &mut, &timeout);
}
if (retcode == ETIMEDOUT) {
        /* timeout occurred */
} else {
        /* operate on x and y */
}
pthread_mutex_unlock(&mut);
```

Condition attributes can be specified at condition-creation time, by passing a condition-attribute object as a second argument to `pthread_cond_init`. Passing `NULL` is equivalent to passing a condition-attribute object with all attributes set to their default values.

The LinuxThreads implementation supports no attributes for conditions. The functions on condition attributes are included only for compliance with the POSIX standard.

`int` **pthread_condattr_init** (`pthread_condattr_t` **attr*) Function

`int` **pthread_condattr_destroy** (`pthread_condattr_t` **attr*) Function

`pthread_condattr_init` initializes the condition-attribute object *attr* and fills it with default values for the attributes. `pthread_condattr_destroy` destroys the condition-attribute object *attr*.

Both functions do nothing in the LinuxThreads implementation.

`pthread_condattr_init` and `pthread_condattr_destroy` always return 0.

18.7 POSIX Semaphores

Semaphores are counters for resources shared between threads. The basic operations on semaphores are increment the counter atomically, and wait until the counter is nonnull and decrement it atomically.

Semaphores have a maximum value past which they cannot be incremented. The macro `SEM_VALUE_MAX` is defined to be this maximum value. In the GNU C Library, `SEM_VALUE_MAX` is equal to `INT_MAX` (see Section A.5.2 [Range of an Integer Type], page 465), but it may be much smaller on other systems.

The pthreads library implements POSIX 1003.1b semaphores. These should not be confused with System V semaphores (`ipc`, `semctl` and `semop`).

All the semaphore functions and macros are defined in '`semaphore.h`'.

`int` **sem_init** (`sem_t *`*sem*`, int` *pshared*`, unsigned int` *value*) Function

`sem_init` initializes the semaphore object pointed to by *sem*. The count associated with the semaphore is set initially to *value*. The *pshared* argument indicates whether the semaphore is local to the current process (*pshared* is 0) or is to be shared between several processes (*pshared* is not 0).

On success, `sem_init` returns 0. On failure, it returns -1 and sets *errno* to one of the following values:

`EINVAL` *value* exceeds the maximum counter-value `SEM_VALUE_MAX`.

`ENOSYS` *pshared* is not 0. LinuxThreads currently does not support process-shared semaphores. This will eventually change.

`int` **sem_destroy** (`sem_t *` *sem*) Function

`sem_destroy` destroys a semaphore object, freeing the resources it might hold. If any threads are waiting on the semaphore when `sem_destroy` is called, it fails and sets *errno* to `EBUSY`.

In the LinuxThreads implementation, no resources are associated with semaphore objects, thus `sem_destroy` actually does nothing except check that no thread is waiting on the semaphore. This will change when process-shared semaphores are implemented.

`int` **sem_wait** (`sem_t *` *sem*) Function

`sem_wait` suspends the calling thread until the semaphore pointed to by *sem* has nonzero count. It then atomically decreases the semaphore count.

`sem_wait` is a cancellation point. It always returns 0.

`int` **sem_trywait** (`sem_t *` *sem*) Function

`sem_trywait` is a nonblocking variant of `sem_wait`. If the semaphore pointed to by *sem* has nonzero count, the count is atomically decreased and `sem_trywait` immediately returns 0. If the semaphore count is 0, `sem_trywait` immediately returns -1 and sets errno to `EAGAIN`.

`int` **sem_post** (`sem_t *` *sem*) Function

`sem_post` atomically increases the count of the semaphore pointed to by *sem*. This function never blocks.

On processors supporting atomic compare-and-swap (Intel 486, Pentium and later, Alpha, PowerPC, MIPS II, Motorola 68k, Ultrasparc), the `sem_post` function can safely be called from signal handlers. This is the only thread-synchronization function provided by POSIX threads that is async-signal safe. On the Intel 386 and earlier Sparc chips, the current LinuxThreads implementation of `sem_post` is not async-signal safe, because the hardware does not support the required atomic operations.

`sem_post` always succeeds and returns 0, unless the semaphore count would exceed `SEM_VALUE_MAX` after being incremented. In that case, `sem_post` returns -1 and sets *errno* to `EINVAL`. The semaphore count is left unchanged.

`int` **sem_getvalue** (`sem_t *` *sem*, `int *` *sval*) Function

`sem_getvalue` stores in the location pointed to by *sval* the current count of the semaphore *sem*. It always returns 0.

18.8 Thread-Specific Data

Programs often need global or static variables that have different values in different threads. Since threads share one memory space, this cannot be achieved with regular variables. Thread-specific data is the POSIX threads answer to this need.

Each thread possesses a private memory-block, the thread-specific data area, or TSD area for short. This area is indexed by TSD keys. The TSD area associates values of type `void *` to TSD keys. TSD keys are common to all threads, but the value associated with a given TSD key can be different in each thread.

For concreteness, the TSD areas can be viewed as arrays of `void *` pointers, TSD keys as integer indices into these arrays, and the value of a TSD key as the value of the corresponding array-element in the calling thread.

When a thread is created, its TSD area initially associates `NULL` with all keys.

`int` **pthread_key_create** (`pthread_key_t *`*key*, `void (*destr_function) (void *)`) Function

`pthread_key_create` allocates a new TSD key. The key is stored in the location pointed to by *key*. There is a limit of `PTHREAD_KEYS_MAX` on the number of keys allocated at a given time. The value initially associated with the returned key is `NULL` in all currently executing threads.

The *destr_function* argument, if not NULL, specifies a destructor function associated with the key. When a thread terminates via pthread_exit or by cancellation, *destr_function* is called on the value associated with the key in that thread. The *destr_function* is not called if a key is deleted with pthread_key_delete or a value is changed with pthread_setspecific. The order in which destructor functions are called at thread-termination time is unspecified.

Before the destructor function is called, the NULL value is associated with the key in the current thread. A destructor function might, however, re-associate non-NULL values to that key or some other key. To deal with this, if after all the destructors have been called for all non-NULL values, there are still some non-NULL values with associated destructors, then the process is repeated. The LinuxThreads implementation stops the process after PTHREAD_DESTRUCTOR_ITERATIONS iterations, even if some non-NULL values with associated descriptors remain. Other implementations may loop indefinitely.

pthread_key_create returns 0 unless PTHREAD_KEYS_MAX keys have already been allocated, in which case it fails and returns EAGAIN.

int **pthread_key_delete** (pthread_key_t *key*) Function

pthread_key_delete deallocates a TSD key. It does not check whether non-NULL values are associated with that key in the currently executing threads, nor call the destructor function associated with the key.

If there is no such key *key*, it returns EINVAL. Otherwise, it returns 0.

int **pthread_setspecific** (pthread_key_t *key*, const void **pointer*) Function

pthread_setspecific changes the value associated with *key* in the calling thread, storing the given *pointer* instead.

If there is no such key *key*, it returns EINVAL. Otherwise, it returns 0.

void * **pthread_getspecific** (pthread_key_t *key*) Function

pthread_getspecific returns the value currently associated with *key* in the calling thread.

If there is no such key *key*, it returns NULL.

The following code fragment allocates a thread-specific array of 100 characters, with automatic reclamation at thread exit:

```
/* Key for the thread-specific buffer */
static pthread_key_t buffer_key;

/* Once-only initialization of the key */
static pthread_once_t buffer_key_once = PTHREAD_ONCE_INIT;

/* Allocate the thread-specific buffer. */
void buffer_alloc(void)
```

```
{
  pthread_once(&buffer_key_once, buffer_key_alloc);
  pthread_setspecific(buffer_key, malloc(100));
}

/* Return the thread-specific buffer */
char * get_buffer(void)
{
  return (char *) pthread_getspecific(buffer_key);
}

/* Allocate the key. */
static void buffer_key_alloc()
{
  pthread_key_create(&buffer_key, buffer_destroy);
}

/* Free the thread-specific buffer */
static void buffer_destroy(void * buf)
{
  free(buf);
}
```

18.9 Threads and Signal-Handling

`int` **pthread_sigmask** (`int` *how*, `const sigset_t` **newmask*, `sigset_t` **oldmask*) Function

`pthread_sigmask` changes the signal mask for the calling thread as described by the *how* and *newmask* arguments. If *oldmask* is not `NULL`, the previous signal-mask is stored in the location pointed to by *oldmask*.

The meaning of the *how* and *newmask* arguments is the same as for `sigprocmask`. If *how* is `SIG_SETMASK`, the signal mask is set to *newmask*. If *how* is `SIG_BLOCK`, the signals specified to *newmask* are added to the current signal-mask. If *how* is `SIG_UNBLOCK`, the signals specified to *newmask* are removed from the current signal mask.

Recall that signal masks are set on a per-thread basis, but signal actions and signal handlers, as set with `sigaction`, are shared between all threads.

The `pthread_sigmask` function returns 0 on success and one of the following error codes on error:

`EINVAL` *how* is not one of `SIG_SETMASK`, `SIG_BLOCK` or `SIG_UNBLOCK`.

`EFAULT` *newmask* or *oldmask* point to invalid addresses.

`int` **pthread_kill** (`pthread_t` *thread*, `int` *signo*) Function

`pthread_kill` sends signal number *signo* to the thread *thread*. The signal is delivered and handled as described in Chapter 17 [Signal Handling], page 377.

`pthread_kill` returns 0 on success and one of the following error codes on error:

`EINVAL` *signo* is not a valid signal-number.

`ESRCH` The thread *thread* does not exist (e.g. it has already terminated).

`int` **sigwait** (`const sigset_t *`*set*, `int *`*sig*) Function

`sigwait` suspends the calling thread until one of the signals in *set* is delivered to the calling thread. It then stores the number of the signal received in the location pointed to by *sig* and returns. The signals in *set* must be blocked and not ignored on entrance to `sigwait`. If the delivered signal has a signal-handler function attached, that function is *not* called.

`sigwait` is a cancellation point. It always returns 0.

For `sigwait` to work reliably, the signals being waited for must be blocked in all threads, not only in the calling thread, since otherwise the POSIX semantics for signal delivery do not guarantee that it's the thread doing the `sigwait` that will receive the signal. The best way to achieve this is block those signals before any threads are created, and never unblock them in the program other than by calling `sigwait`.

Signal handling in LinuxThreads departs significantly from the POSIX standard. According to the standard, "asynchronous" (external) signals are addressed to the whole process (the collection of all threads), which then delivers them to one particular thread. The thread that actually receives the signal is any thread that does not currently block the signal.

In LinuxThreads, each thread is actually a kernel process with its own PID, so external signals are always directed to one particular thread. If, for instance, another thread is blocked in `sigwait` on that signal, it will not be restarted.

The LinuxThreads implementation of `sigwait` installs dummy signal-handlers for the signals in *set* for the duration of the wait. Since signal handlers are shared between all threads, other threads must not attach their own signal-handlers to these signals, or alternatively they should all block these signals (which is recommended anyway).

18.10 Threads and Fork

It's not intuitively obvious what should happen when a multithreaded POSIX process calls `fork`. Not only are the semantics tricky, but you may need to write code that does the right thing at fork time even if that code doesn't use the `fork` function. Moreover, you need to be aware of interaction between `fork` and some library features like `pthread_once` and stdio streams.

When `fork` is called by one of the threads of a process, it creates a new process that is a copy of the calling process. Effectively, in addition to copying certain system objects, the function takes a snapshot of the memory areas of the parent process, and creates identical areas in the child. To make matters more complicated, with threads it's possible for two or more threads to concurrently call fork to create two or more child processes.

The child process has a copy of the address space of the parent, but it does not inherit any of its threads. Execution of the child process is carried out by a new thread that returns from the `fork` function with a return value of 0; it is the only thread in the child process. Because threads are not inherited across fork, issues arise. At the time of the call to `fork`, threads in the parent process other than the one calling `fork` may have been executing critical regions of code. As a result, the child process may get a copy of objects that are not in a well-defined state. This potential problem affects all components of the program.

Any program component that will continue being used in a child process must correctly handle its state during `fork`. For this purpose, the POSIX interface provides the special function `pthread_atfork` for installing pointers to handler functions that are called from within `fork`.

`int` **pthread_atfork** (`void` (⋆*prepare*)(`void`), `void` (⋆*parent*)(`void`), `void` (⋆*child*)(`void`)) Function

`pthread_atfork` registers handler functions to be called just before and just after a new process is created with `fork`. The *prepare* handler will be called from the parent process, just before the new process is created. The *parent* handler will be called from the parent process, just before `fork` returns. The *child* handler will be called from the child process, just before `fork` returns.

`pthread_atfork` returns 0 on success and a nonzero error-code on error.

One or more of the three handlers *prepare*, *parent* and *child* can be given as `NULL`, meaning that no handler needs to be called at the corresponding point.

`pthread_atfork` can be called several times to install several sets of handlers. At `fork` time, the *prepare* handlers are called in LIFO order (last added with `pthread_atfork`, first called before `fork`), while the *parent* and *child* handlers are called in FIFO order (first added, first called).

If there is insufficient memory available to register the handlers, `pthread_atfork` fails and returns `ENOMEM`. Otherwise, it returns 0.

The functions `fork` and `pthread_atfork` must not be regarded as reentrant from the context of the handlers. That is to say, if a `pthread_atfork` handler invoked from within `fork` calls `pthread_atfork` or `fork`, the behavior is undefined.

Registering a triplet of handlers is an atomic operation with respect to fork. If new handlers are registered at about the same time as a fork occurs, either all three handlers will be called, or none of them will be called.

The handlers are inherited by the child process, and there is no way to remove them, short of using `exec` to load a new process-image.

To understand the purpose of `pthread_atfork`, recall that `fork` duplicates the whole memory space, including mutexes in their current locking-state, but only the calling thread—other threads are not running in the child process. The mutexes are not usable after the `fork` and must be initialized with `pthread_mutex_init` in the child process. This is a limitation of the current implementation and might or might not be present in future versions.

To avoid this, install handlers with `pthread_atfork` as follows. Have the *prepare* handler lock the mutexes (in locking order), and the *parent* handler unlock the mutexes. The *child* handler should reset the mutexes using `pthread_mutex_init`, as well as any other synchronization objects such as condition variables.

Locking the global mutexes before the fork ensures that all other threads are locked out of the critical regions of code protected by those mutexes. Thus when `fork` takes a snapshot of the parent's address space, that snapshot will copy valid, stable data. Resetting the synchronization objects in the child process will ensure they are properly cleansed of any artifacts from the threading subsystem of the parent process. For example, a mutex may inherit a wait queue of threads waiting for the lock; this wait queue makes no sense in the child process. Initializing the mutex takes care of this.

18.11 Streams and Fork

The GNU standard I/O library has an internal mutex that guards the internal linked list of all standard C FILE objects. This mutex is properly taken care of during `fork` so that the child receives an intact copy of the list. This allows the `fopen` function, and related stream-creating functions, to work correctly in the child process, since these functions need to insert into the list.

However, the individual stream locks are not completely taken care of. Thus unless the multithreaded application takes special precautions in its use of `fork`, the child process might not be able to safely use the streams that it inherited from the parent. In general, for any given open stream in the parent that is to be used by the child process, the application must ensure that that stream is not in use by another thread when `fork` is called. Otherwise an inconsistent copy of the stream object be produced. An easy way to ensure this is to use `flockfile` to lock the stream prior to calling `fork` and then unlock it with `funlockfile` inside the parent process, provided that the parent's threads properly honor these locks. Nothing special needs to be done in the child process, since the library internally resets all stream locks.

Note that the stream locks are not shared between the parent and child. For example, even if you ensure that, say, the stream `stdout` is properly treated and can be safely used in the child, the stream locks do not provide an exclusion mechanism between the parent and child. If both processes write to `stdout`, strangely interleaved output may result regardless of the explicit use of `flockfile` or implicit locks.

Also note that these provisions are a GNU extension; other systems might not provide any way for streams to be used in the child of a multithreaded process. POSIX requires that such a child process confine itself to calling only asynchronous-safe functions, which excludes much of the library, including standard I/O.

18.12 Miscellaneous Thread Functions

`pthread_t` **pthread_self** (*void*) Function

`pthread_self` returns the thread identifier for the calling thread.

`int` **pthread_equal** (`pthread_t thread1, pthread_t thread2`) Function

`pthread_equal` determines if two thread-identifiers refer to the same thread.

A nonzero value is returned if *thread1* and *thread2* refer to the same thread. Otherwise, 0 is returned.

`int` **pthread_detach** (`pthread_t` *th*) Function

`pthread_detach` puts the thread *th* in the detached state. This guarantees that the memory resources consumed by *th* will be freed immediately when *th* terminates. However, this prevents other threads from synchronizing on the termination of *th* using `pthread_join`.

A thread can be created initially in the detached state, using the `detachstate` attribute to `pthread_create`. In contrast, `pthread_detach` applies to threads created in the joinable state, and which need to be put in the detached state later.

After `pthread_detach` completes, subsequent attempts to perform `pthread_join` on *th* will fail. If another thread is already joining the thread *th* at the time `pthread_detach` is called, `pthread_detach` does nothing and leaves *th* in the joinable state.

On success, 0 is returned. On error, one of the following codes is returned:

`ESRCH` No thread could be found corresponding to that specified by *th*.

`EINVAL` The thread *th* is already in the detached state.

`void` **pthread_kill_other_threads_np** (*void*) Function

`pthread_kill_other_threads_np` is a nonportable LinuxThreads extension. It causes all threads in the program to terminate immediately, except the calling thread, which proceeds normally. It is intended to be called just before a thread calls one of the `exec` functions, e.g. `execve`.

Termination of the other threads is not performed through `pthread_cancel` and completely bypasses the cancellation mechanism. Hence, the current settings for cancellation state and cancellation type are ignored, and the clean-up handlers are not executed in the terminated threads.

According to POSIX 1003.1c, a successful `exec*` in one of the threads should automatically terminate all other threads in the program. This behavior is

not yet implemented in LinuxThreads. Calling `pthread_kill_other_threads_np` before `exec*` achieves much of the same behavior, except that if `exec*` ultimately fails, then all other threads are already killed.

`int` **pthread_once** (`pthread_once_t *once_control, void (*init_routine) (void)`) Function

The purpose of `pthread_once` is to ensure that a piece of initialization code is executed at most one time. The *once_control* argument points to a static or extern variable statically initialized to `PTHREAD_ONCE_INIT`.

The first time `pthread_once` is called with a given *once_control* argument, it calls *init_routine* with no argument and changes the value of the *once_control* variable to record that initialization has been performed. Subsequent calls to `pthread_once` with the same `once_control` argument do nothing.

If a thread is cancelled while executing *init_routine*, the state of the *once_control* variable is reset so that a future call to `pthread_once` will call the routine again.

If the process forks while one or more threads are executing `pthread_once` initialization routines, the states of their respective *once_control* variables will appear to be reset in the child process so that if the child calls `pthread_once`, the routines will be executed.

`pthread_once` always returns 0.

`int` **pthread_setschedparam** (`pthread_t target_thread, int policy, const struct sched_param *param`) Function

`pthread_setschedparam` sets the scheduling parameters for the thread *target_thread* as indicated by *policy* and *param*. *policy* can be either `SCHED_OTHER` (regular, non-real-time scheduling), `SCHED_RR` (real-time, round-robin) or `SCHED_FIFO` (real-time, first-in first-out). *param* specifies the scheduling priority for the two real-time policies (see Section 14.3.4 [Traditional Scheduling], page 349).

The real-time scheduling policies `SCHED_RR` and `SCHED_FIFO` are available only to processes with superuser privileges.

On success, `pthread_setschedparam` returns 0. On error it returns one of the following codes:

`EINVAL` *policy* is not one of `SCHED_OTHER`, `SCHED_RR` or `SCHED_FIFO`, or the priority value specified by *param* is not valid for the specified policy.

`EPERM` Real-time scheduling was requested, but the calling process does not have sufficient privileges.

`ESRCH` The *target_thread* is invalid or has already terminated.

`EFAULT` *param* points outside the process memory space.

`int` **pthread_getschedparam** (`pthread_t target_thread, int *policy, struct sched_param *param`) Function

`pthread_getschedparam` retrieves the scheduling policy and scheduling parameters for the thread *target_thread* and stores them in the locations pointed to by *policy* and *param*, respectively.

`pthread_getschedparam` returns 0 on success, or one of the following error codes on failure:

`ESRCH` The *target_thread* is invalid or has already terminated.

`EFAULT` *policy* or *param* point outside the process memory space.

`int` **pthread_setconcurrency** (`int level`) Function

`pthread_setconcurrency` is unused in LinuxThreads due to the lack of a mapping of user threads to kernel threads. It exists for source compatibility. It does store the value *level* so that it can be returned by a subsequent call to `pthread_getconcurrency`. However, it takes no other action.

`int` **pthread_getconcurrency** () Function

`pthread_getconcurrency` is unused in LinuxThreads due to the lack of a mapping of user threads to kernel threads. It exists for source compatibility. However, it will return the value that was set by the last call to `pthread_setconcurrency`.

Appendix A C Language Facilities in the Library

Some of the facilities implemented by the C library really should be thought of as parts of the C language itself. These facilities ought to be documented in the C Language Manual, not in the library manual; but since we don't have the language manual yet, and documentation for these features has been written, we are publishing it here.

A.1 Explicitly Checking Internal Consistency

When you're writing a program, it's often a good idea to put in checks at strategic places for "impossible" errors or violations of basic assumptions. These kinds of checks are helpful in debugging problems with the interfaces between different parts of the program, for example.

The `assert` macro, defined in the header file '`assert.h`', provides a convenient way to abort the program while printing a message about where in the program the error was detected.

Once you think your program is debugged, you can disable the error checks performed by the `assert` macro by recompiling with the macro `NDEBUG` defined. This means you don't actually have to change the program source code to disable these checks.

But disabling these consistency checks is undesirable unless they make the program significantly slower. All else being equal, more error checking is good no matter who is running the program. A wise user would rather have a program crash, visibly, than have it return nonsense without indicating anything might be wrong.

`void` **assert** (`int` *expression*) Macro

Verify the programmer's belief that *expression* is nonzero at this point in the program.

If `NDEBUG` is not defined, `assert` tests the value of *expression*. If it is false (0), `assert` aborts the program after printing a message of the form:[1]

'*file*':*linenum*: *function*: `Assertion '`*expression*`' failed.`

on the standard error stream `stderr`.[2] The file name and line number are taken from the C preprocessor macros `__FILE__` and `__LINE__` and specify where the call to `assert` was made. When using the GNU C Compiler, the name of the function that calls `assert` is taken from the built-in variable `__PRETTY_FUNCTION__`; with older compilers, the function name and following colon are omitted.

---

[1] See Loosemore et al., "Aborting a Program" (see chap. 1, n. 1).

[2] Ibid., "Standard Streams".

If the preprocessor macro `NDEBUG` is defined before '`assert.h`' is included, the `assert` macro is defined to do absolutely nothing.

Warning: Even the argument expression *expression* is not evaluated if `NDEBUG` is in effect. So never use `assert` with arguments that involve side effects. For example, `assert (++i > 0);` is a bad idea, because `i` will not be incremented if `NDEBUG` is defined.

Sometimes the "impossible" condition you want to check for is an error return from an operating system function. Then it is useful to display not only where the program crashes, but also what error was returned. The `assert_perror` macro makes this easy.

`void` **assert_perror** (`int` *errnum*) Macro

This is similar to `assert`, but it verifies that *errnum* is 0.

If `NDEBUG` is not defined, `assert_perror` tests the value of *errnum*. If it is nonzero, `assert_perror` aborts the program after printing a message of the form:

'*file*':*linenum*: *function*: *error text*

on the standard error stream. The file name, line number, and function name are as for `assert`. The error text is the result of `strerror (`*errnum*`)`.[3]

Like `assert`, if `NDEBUG` is defined before '`assert.h`' is included, the `assert_perror` macro does absolutely nothing. It does not evaluate the argument, so *errnum* should not have any side effects. It is best for *errnum* to be just a simple variable reference; often it will be `errno`.

This macro is a GNU extension.

Usage Note: The `assert` facility is designed for detecting *internal inconsistency*; it is not suitable for reporting invalid input or improper usage by the *user* of the program.

The information in the diagnostic messages printed by the `assert` and `assert_perror` macro is intended to help you, the programmer, track down the cause of a bug, but is not really useful for telling a user of your program why his input was invalid or why a command could not be carried out. What's more, your program should not abort when given invalid input, as `assert` would do—it should exit with nonzero status[4] after printing its error messages, or perhaps read another command or move on to the next input file.[5]

A.2 Variadic Functions

ISO C defines a syntax for declaring a function to take a variable number or type of arguments. (Such functions are referred to as *varargs functions* or *variadic*

[3] Ibid., "Error Messages".

[4] Ibid., "Exit Status".

[5] For more information on printing error messages for problems that *do not* represent bugs in the program see Loosemore et al., "Error Messages".

functions.) However, the language itself provides no mechanism for such functions to access their nonrequired arguments; instead, you use the variable arguments macros defined in '`stdarg.h`'.

This section describes how to declare variadic functions, how to write them, and how to call them properly.

Compatibility Note: Many older C dialects provide a similar, but incompatible, mechanism for defining functions with variable numbers of arguments, using '`varargs.h`'.

A.2.1 Why Variadic Functions Are Used

Ordinary C functions take a fixed number of arguments. When you define a function, you specify the data type for each argument. Every call to the function should supply the expected number of arguments, with types that can be converted to the specified ones. Thus, if the function '`foo`' is declared with `int foo (int, char *);` then you must call it with two arguments: a number (any kind will do) and a string pointer.

But some functions perform operations that can meaningfully accept an unlimited number of arguments.

In some cases, a function can handle any number of values by operating on all of them as a block. For example, consider a function that allocates a one-dimensional array with `malloc` to hold a specified set of values. This operation makes sense for any number of values, as long as the length of the array corresponds to that number. Without facilities for variable arguments, you would have to define a separate function for each possible array size.

The library function `printf`[6] is an example of another class of function where variable arguments are useful. This function prints its arguments (which can vary in type as well as number) under the control of a format template string.

These are good reasons to define a *variadic* function that can handle as many arguments as the caller chooses to pass.

Some functions such as `open` take a fixed set of arguments, but occasionally ignore the last few. Strict adherence to ISO C requires these functions to be defined as variadic; in practice, however, the GNU C Compiler and most other C compilers let you define such a function to take a fixed set of arguments—the most it can ever use—and then only *declare* the function as variadic (or not declare its arguments at all).

A.2.2 How Variadic Functions Are Defined and Used

Defining and using a variadic function involves three steps:

- *Define* the function as variadic, using an ellipsis ('`...`') in the argument list, and using special macros to access the variable arguments (see Section A.2.2.2 [Receiving the Argument Values], page 458).

[6] Ibid., "Formatted Output."

- *Declare* the function as variadic, using a prototype with an ellipsis (‘...’), in all the files that call it (see Section A.2.2.1 [Syntax for Variable Arguments], page 458).
- *Call* the function by writing the fixed arguments followed by the additional variable arguments (see Section A.2.2.4 [Calling Variadic Functions], page 460).

A.2.2.1 Syntax for Variable Arguments

A function that accepts a variable number of arguments must be declared with a prototype that says so. You write the fixed arguments as usual, and then tack on ‘...’ to indicate the possibility of additional arguments. The syntax of ISO C requires at least one fixed argument before the ‘...’. For example:

```
int
func (const char *a, int b, ...)
{
  ...
}
```

defines a function `func` that returns an `int` and takes two required arguments, a `const char *` and an `int`. These are followed by any number of anonymous arguments.

Portability Note: For some C compilers, the last required argument must not be declared `register` in the function definition. Furthermore, this argument’s type must be *self-promoting*—the default promotions must not change its type. This rules out array and function types, as well as `float`, `char` (whether signed or not) and `short int` (whether signed or not). This is actually an ISO C requirement.

A.2.2.2 Receiving the Argument Values

Ordinary fixed arguments have individual names, and you can use these names to access their values. But optional arguments have no names—nothing but ‘...’. How can you access them?

The only way to access them is sequentially, in the order they were written, and you must use special macros from ‘`stdarg.h`’ in the following three-step process:

1. You initialize an argument pointer variable of type `va_list` using `va_start`. The argument pointer, when initialized, points to the first optional argument.
2. You access the optional arguments by successive calls to `va_arg`. The first call to `va_arg` gives you the first optional argument, the next call gives you the second, and so on.

   You can stop at any time if you wish to ignore any remaining optional arguments. It is perfectly all right for a function to access fewer arguments than

were supplied in the call, but you will get garbage values if you try to access too many arguments.

3. You indicate that you are finished with the argument pointer variable by calling `va_end`.

   In practice, with most C compilers, calling `va_end` does nothing. This is always true in the GNU C Compiler. But you might as well call `va_end`, just in case your program is someday compiled with a peculiar compiler.

See Section A.2.2.5 [Argument-Access Macros], page 460, for the full definitions of `va_start`, `va_arg` and `va_end`.

Steps 1 and 3 must be performed in the function that accepts the optional arguments. However, you can pass the `va_list` variable as an argument to another function and perform all or part of step 2 there.

You can perform the entire sequence of three steps multiple times within a single function invocation. If you want to ignore the optional arguments, you can do these steps zero times.

You can have more than one argument pointer variable if you like. You can initialize each variable with `va_start` when you wish, and then you can fetch arguments with each argument pointer as you wish. Each argument pointer variable will sequence through the same set of argument values, but at its own pace.

Portability Note: With some compilers, once you pass an argument pointer value to a subroutine, you must not keep using the same argument pointer value after that subroutine returns. For full portability, you should just pass it to `va_end`. This is actually an ISO C requirement, but most ANSI C compilers work happily regardless.

A.2.2.3 How Many Arguments Were Supplied

There is no general way for a function to determine the number and type of the optional arguments it was called with. So whoever designs the function typically designs a convention for the caller to specify the number and type of arguments. It is up to you to define an appropriate calling convention for each variadic function and write all calls accordingly.

One kind of calling convention is to pass the number of optional arguments as one of the fixed arguments. This convention works provided all of the optional arguments are of the same type.

A similar alternative is to have one of the required arguments be a bit mask, with a bit for each possible purpose for which an optional argument might be supplied. You would test the bits in a predefined sequence; if the bit is set, fetch the value of the next argument, otherwise use a default value.

A required argument can be used as a pattern to specify both the number and types of the optional arguments. The format-string argument to `printf` is one example of this.[7]

[7] Ibid., “Formatted Output Functions”.

Another possibility is to pass an "end-marker" value as the last optional argument. For example, for a function that manipulates an arbitrary number of pointer arguments, a null pointer might indicate the end of the argument list. (This assumes that a null pointer isn't otherwise meaningful to the function.) The `execl` function works in just this way (see Section 7.5 [Executing a File], page 212).

A.2.2.4 Calling Variadic Functions

You don't have to do anything special to call a variadic function. Just put the arguments (required arguments, followed by optional ones) inside parentheses, separated by commas, as usual. But you must declare the function with a prototype and know how the argument values are converted.

In principle, functions that are *defined* to be variadic must also be *declared* to be variadic using a function prototype whenever you call them (see Section A.2.2.1 [Syntax for Variable Arguments], page 458). This is because some C compilers use a different calling convention to pass the same set of argument values to a function depending on whether that function takes variable arguments or fixed arguments.

In practice, the GNU C Compiler always passes a given set of argument types in the same way regardless of whether they are optional or required. So, as long as the argument types are self-promoting, you can safely omit declaring them. Usually it is a good idea to declare the argument types for variadic functions, and indeed for all functions. But there are a few functions that are convenient to not have to declare as variadic—for example, `open` and `printf`.

Since the prototype doesn't specify types for optional arguments, in a call to a variadic function the *default argument promotions* are performed on the optional argument values. This means the objects of type `char` or `short int` (whether signed or not) are promoted to either `int` or `unsigned int`, as appropriate; and that objects of type `float` are promoted to type `double`. So, if the caller passes a `char` as an optional argument, it is promoted to an `int`, and the function can access it with `va_arg (`*ap*`, int)`.

Conversion of the required arguments is controlled by the function prototype in the usual way—the argument expression is converted to the declared argument type as if it were being assigned to a variable of that type.

A.2.2.5 Argument-Access Macros

Here are descriptions of the macros used to retrieve variable arguments. These macros are defined in the header file '`stdarg.h`'.

va_list Data Type

The type `va_list` is used for argument pointer variables.

`void` **va_start** (`va_list` *ap*, *last-required*) Macro

This macro initializes the argument pointer variable *ap* to point to the first of the optional arguments of the current function; *last-required* must be the last required argument to the function.

See Section A.2.3.1 [Old-Style Variadic Functions], page 462, for an alternate definition of `va_start` found in the header file '`varargs.h`'.

type **va_arg** (`va_list` *ap*, *type*) Macro

The `va_arg` macro returns the value of the next optional argument, and modifies the value of *ap* to point to the subsequent argument. Thus, successive uses of `va_arg` return successive optional arguments.

The type of the value returned by `va_arg` is *type* as specified in the call. *type* must be a self-promoting type (not `char` or `short int` or `float`) that matches the type of the actual argument.

`void` **va_end** (`va_list` *ap*) Macro

This ends the use of *ap*. After a `va_end` call, further `va_arg` calls with the same *ap* may not work. You should invoke `va_end` before returning from the function in which `va_start` was invoked with the same *ap* argument.

In the GNU C Library, `va_end` does nothing, and you need not ever use it except for reasons of portability.

Sometimes you have to parse the list of parameters more than once or you want to remember a certain position in the parameter list. To do this, you will have to make a copy of the current value of the argument. But `va_list` is an opaque type and you cannot necessarily assign the value of one variable of type `va_list` to another variable of the same type.

`void` **__va_copy** (`va_list` *dest*, `va_list` *src*) Macro

The `__va_copy` macro allows copying of objects of type `va_list` even if this is not an integral type. The argument pointer in *dest* is initialized to point to the same argument as the pointer in *src*.

This macro is a GNU extension, but we hope it will also be available in the next update of the ISO C standard.

If you want to use `__va_copy`, you should always be prepared for the possibility that this macro will not be available. On architectures where a simple assignment is invalid, you hope `__va_copy` *will* be available, so you should always write something like this:

```
{
  va_list ap, save;
  ...
#ifdef __va_copy
  __va_copy (save, ap);
#else
  save = ap;
#endif
  ...
}
```

A.2.3 Example of a Variadic Function

Here is a complete sample function that accepts a variable number of arguments. The first argument to the function is the count of remaining arguments, which are added up and the result returned. While trivial, this function is sufficient to illustrate how to use the variable arguments facility.

```
#include <stdarg.h>
#include <stdio.h>

int
add_em_up (int count,...)
{
  va_list ap;
  int i, sum;

  va_start (ap, count);          /* Initialize the argument list. */

  sum = 0;
  for (i = 0; i < count; i++)
    sum += va_arg (ap, int);     /* Get the next argument value. */

  va_end (ap);                   /* Clean up. */
  return sum;
}

int
main (void)
{
  /* This call prints 16. */
  printf ("%d\n", add_em_up (3, 5, 5, 6));

  /* This call prints 55. */
  printf ("%d\n", add_em_up (10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10));

  return 0;
}
```

A.2.3.1 Old-Style Variadic Functions

Before ISO C, programmers used a slightly different facility for writing variadic functions. The GNU C Compiler still supports it; currently, it is more portable than the ISO C facility, since support for ISO C is still not universal. The header file that defines the old-fashioned variadic facility is called ‘`varargs.h`’.

Using ‘varargs.h’ is almost the same as using ‘stdarg.h’. There is no difference in how you call a variadic function (see Section A.2.2.4 [Calling Variadic Functions], page 460). The only difference is in how you define them. First of all, you must use old-style nonprototype syntax, like this:

```
tree
build (va_alist)
     va_dcl
{
```

Secondly, you must give va_start only one argument, like this:

```
  va_list p;
  va_start (p);
```

These are the special macros used for defining old-style variadic functions:

va_alist Macro

This macro stands for the argument name list required in a variadic function.

va_dcl Macro

This macro declares the implicit argument or arguments for a variadic function.

`void` **va_start** (`va_list` *ap*) Macro

This macro, as defined in ‘varargs.h’, initializes the argument pointer variable *ap* to point to the first argument of the current function.

The other argument macros, va_arg and va_end, are the same in ‘varargs.h’ as in ‘stdarg.h’ (see Section A.2.2.5 [Argument-Access Macros], page 460 for details).

It does not work to include both ‘varargs.h’ and ‘stdarg.h’ in the same compilation—they define va_start in conflicting ways.

A.3 Null-Pointer Constant

The null-pointer constant is guaranteed not to point to any real object. You can assign it to any pointer variable, since it has type `void *`. The preferred way to write a null-pointer constant is with NULL.

`void *` **NULL** Macro

This is a null-pointer constant.

You can also use `0` or `(void *)0` as a null-pointer constant, but using NULL is cleaner because it makes the purpose of the constant more evident.

If you use the null-pointer constant as a function argument, then for complete portability you should make sure that the function has a prototype declaration. Otherwise, if the target machine has two different pointer representations, the compiler won’t know which representation to use for that argument. You can avoid the problem by explicitly casting the constant to the proper pointer-type, but we recommend instead adding a prototype for the function you are calling.

A.4 Important Data-Types

The result of subtracting two pointers in C is always an integer, but the precise data-type varies from C compiler to C compiler. Likewise, the data type of the result of `sizeof` also varies between compilers. ISO defines standard aliases for these two types, so you can refer to them in a portable fashion. They are defined in the header file '`stddef.h`'.

ptrdiff_t Data Type

This is the signed integer type of the result of subtracting two pointers. For example, with the declaration `char *p1, *p2;`, the expression `p2 - p1` is of type `ptrdiff_t`. This will probably be one of the standard signed integer types (`short int`, `int` or `long int`), but might be a nonstandard type that exists only for this purpose.

size_t Data Type

This is an unsigned integer type used to represent the sizes of objects. The result of the `sizeof` operator is of this type, and functions such as `malloc`[8] and `memcpy`[9] accept arguments of this type to specify object sizes.

Usage Note: `size_t` is the preferred way to declare any arguments or variables that hold the size of an object.

In the GNU system, `size_t` is equivalent to either `unsigned int` or `unsigned long int`. These types have identical properties on the GNU system and, for most purposes, you can use them interchangeably. However, they are distinct as data types, which makes a difference in certain contexts.

For example, when you specify the type of a function argument in a function prototype, it makes a difference which one you use. If the system header files declare `malloc` with an argument of type `size_t` and you declare `malloc` with an argument of type `unsigned int`, you will get a compilation error if `size_t` happens to be `unsigned long int` on your system. To avoid any possibility of error, when a function argument or value is supposed to have type `size_t`, never declare its type in any other way.

Compatibility Note: Implementations of C before the advent of ISO C generally used `unsigned int` for representing object sizes and `int` for pointer subtraction results. They did not necessarily define either `size_t` or `ptrdiff_t`. Unix systems did define `size_t`, in '`sys/types.h`', but the definition was usually a signed type.

A.5 Data-Type Measurements

Most of the time, if you choose the proper C data-type for each object in your program, you need not be concerned with just how it is represented or how many

[8] Ibid., "Unconstrained Allocation".

[9] Ibid., "Copying and Concatenation".

bits it uses. When you do need such information, the C language itself does not provide a way to get it. The header files ‘`limits.h`’ and ‘`float.h`’ contain macros that give you this information in full detail.

A.5.1 Computing the Width of an Integer Data Type

The most common reason that a program needs to know how many bits are in an integer type is for using an array of `long int` as a bit vector. You can access the bit at index *n* with:

```
vector[n / LONGBITS] & (1 << (n % LONGBITS))
```

provided you define `LONGBITS` as the number of bits in a `long int`.

There is no operator in the C language that can give you the number of bits in an integer data type. But you can compute it from the macro `CHAR_BIT`, defined in the header file ‘`limits.h`’.

`CHAR_BIT`
: This is the number of bits in a `char`—8, on most systems. The value has type `int`.

  You can compute the number of bits in any data type *type* like this:

  ```
  sizeof (type) * CHAR_BIT
  ```

A.5.2 Range of an Integer Type

Suppose you need to store an integer value that can range from 0 to 1,000,000. Which is the smallest type you can use? There is no general rule; it depends on the C compiler and target machine. You can use the ‘`MIN`’ and ‘`MAX`’ macros in ‘`limits.h`’ to determine which type will work.

Each signed integer type has a pair of macros that give the smallest and largest values that it can hold. Each unsigned integer type has one such macro, for the maximum value; the minimum value is, of course, 0.

The values of these macros are all integer constant expressions. The ‘`MAX`’ and ‘`MIN`’ macros for `char` and `short int` types have values of type `int`. The ‘`MAX`’ and ‘`MIN`’ macros for the other types have values of the same type described by the macro—thus, `ULONG_MAX` has type `unsigned long int`.

`SCHAR_MIN`
: This is the minimum value that can be represented by a `signed char`.

`SCHAR_MAX`
`UCHAR_MAX`
: These are the maximum values that can be represented by a `signed char` and `unsigned char`, respectively.

`CHAR_MIN`
: This is the minimum value that can be represented by a `char`. It’s equal to `SCHAR_MIN` if `char` is signed, or 0 otherwise.

`CHAR_MAX`
: This is the maximum value that can be represented by a `char`. It's equal to `SCHAR_MAX` if `char` is signed, or `UCHAR_MAX` otherwise.

`SHRT_MIN`
: This is the minimum value that can be represented by a `signed short int`. On most machines that the GNU C Library runs on, `short` integers are 16-bit quantities.

`SHRT_MAX`
`USHRT_MAX`
: These are the maximum values that can be represented by a `signed short int` and `unsigned short int`, respectively.

`INT_MIN`
: This is the minimum value that can be represented by a `signed int`. On most machines that the GNU C system runs on, an `int` is a 32-bit quantity.

`INT_MAX`
`UINT_MAX`
: These are the maximum values that can be represented by, respectively, the type `signed int` and the type `unsigned int`.

`LONG_MIN`
: This is the minimum value that can be represented by a `signed long int`. On most machines that the GNU C system runs on, `long` integers are 32-bit quantities, the same size as `int`.

`LONG_MAX`
`ULONG_MAX`
: These are the maximum values that can be represented by a `signed long int` and `unsigned long int`, respectively.

`LONG_LONG_MIN`
: This is the minimum value that can be represented by a `signed long long int`. On most machines that the GNU C system runs on, `long long` integers are 64-bit quantities.

`LONG_LONG_MAX`
`ULONG_LONG_MAX`
: These are the maximum values that can be represented by a `signed long long int` and `unsigned long long int`, respectively.

`WCHAR_MAX`
: This is the maximum value that can be represented by a `wchar_t`.[10]

[10] Ibid., "Introduction to Extended Characters".

The header file '`limits.h`' also defines some additional constants that parameterize various operating-system and file-system limits. These constants are described in Chapter 12 [System-Configuration Parameters], page 303.

A.5.3 Floating-Type Macros

The specific representation of floating-point numbers varies from machine to machine. Because floating-point numbers are represented internally as approximate quantities, algorithms for manipulating floating-point data often need to take account of the precise details of the machine's floating-point representation.

Some of the functions in the C library itself need this information; for example, the algorithms for printing and reading floating-point numbers[11] and for calculating trigonometric and irrational functions[12] use it to avoid round-off error and loss of accuracy. User programs that implement numerical analysis techniques also often need this information in order to minimize or compute error bounds.

The header file '`float.h`' describes the format used by your machine.

A.5.3.1 Floating-Point Representation Concepts

This section introduces the terminology for describing floating-point representations.

You are probably already familiar with most of these concepts in terms of scientific or exponential notation for floating-point numbers. For example, the number `123456.0` could be expressed in exponential notation as `1.23456e+05`, a shorthand notation indicating that the mantissa `1.23456` is multiplied by the base `10` raised to power `5`.

More formally, the internal representation of a floating-point number can be characterized in terms of the following parameters:

- The *sign* is either `-1` or `1`.
- The *base* or *radix* for exponentiation is an integer greater than `1`. This is a constant for a particular representation.
- The *exponent* to which the base is raised: the upper and lower bounds of the exponent value are constants for a particular representation.

  Sometimes, in the actual bits representing the floating-point number, the exponent is *biased* by adding a constant to it, to make it always be represented as an unsigned quantity. This is only important if you have some reason to pick apart the bit fields making up the floating-point number by hand, which is something for which the GNU library provides no support. So this is ignored in the discussion that follows.
- The *mantissa* or *significand* is an unsigned integer that is a part of each floating-point number.

[11] Ibid., "Input/Output on Streams".

[12] Ibid., "Mathematics".

- The *precision* of the mantissa: If the base of the representation is *b*, then the precision is the number of base-*b* digits in the mantissa. This is a constant for a particular representation.

  Many floating-point representations have an implicit *hidden bit* in the mantissa. This is a bit that is present virtually in the mantissa, but not stored in memory because its value is always 1 in a normalized number. The precision figure (see above) includes any hidden bits.

  Again, the GNU library provides no facilities for dealing with such low-level aspects of the representation.

The mantissa of a floating-point number represents an implicit fraction whose denominator is the base raised to the power of the precision. Since the largest representable mantissa is 1 less than this denominator, the value of the fraction is always strictly less than `1`. The mathematical value of a floating-point number is then the product of this fraction, the sign, and the base raised to the exponent.

We say that the floating-point number is *normalized* if the fraction is at least `1/`*b*, where *b* is the base. In other words, the mantissa would be too large to fit if it were multiplied by the base. Nonnormalized numbers are sometimes called *denormal*; they contain less precision than the representation normally can hold.

If the number is not normalized, then you can subtract `1` from the exponent while multiplying the mantissa by the base, and get another floating-point number with the same value. *Normalization* consists of doing this repeatedly until the number is normalized. Two distinct normalized floating-point numbers cannot be equal in value.

There is an exception to this rule: if the mantissa is 0, it is considered normalized. Another exception happens on certain machines where the exponent is as small as the representation can hold. Then it is impossible to subtract `1` from the exponent, so a number may be normalized even if its fraction is less than `1/`*b*.

A.5.3.2 Floating-Point Parameters

These macro definitions can be accessed by including the header file '`float.h`' in your program.

Macro names starting with '`FLT_`' refer to the `float` type, while names beginning with '`DBL_`' refer to the `double` type, and names beginning with '`LDBL_`' refer to the `long double` type. (If GCC does not support `long double` as a distinct data-type on a target machine, then the values for the '`LDBL_`' constants are equal to the corresponding constants for the `double` type.)

Of these macros, only `FLT_RADIX` is guaranteed to be a constant expression. The other macros listed here cannot be reliably used in places that require constant expressions, such as '`#if`' preprocessing directives or in the dimensions of static arrays.

Although the ISO C standard specifies minimum and maximum values for most of these parameters, the GNU C implementation uses whatever values describe the floating-point representation of the target machine. So in principle, GNU C actually

satisfies the ISO C requirements only if the target machine is suitable. In practice, all the machines currently supported are suitable.

`FLT_ROUNDS`
: This value characterizes the rounding mode for floating-point addition. The following values indicate standard rounding-modes:

  `-1` The mode is indeterminable.

  `0` Rounding is toward 0.

  `1` Rounding is to the nearest number.

  `2` Rounding is toward positive infinity.

  `3` Rounding is toward negative infinity.

  Any other value represents a machine-dependent nonstandard rounding-mode.

  On most machines, the value is `1`, in accordance with the IEEE standard for floating-point.

  Here is a table showing how certain values round for each possible value of `FLT_ROUNDS`, if the other aspects of the representation match the IEEE single-precision standard.

	0	1	2	3
1.00000003	1.0	1.0	1.00000012	1.0
1.00000007	1.0	1.00000012	1.00000012	1.0
-1.00000003	-1.0	-1.0	-1.0	-1.00000012
-1.00000007	-1.0	-1.00000012	-1.0	-1.00000012

`FLT_RADIX`
: This is the value of the base, or radix, of the exponent representation. This is guaranteed to be a constant expression, unlike the other macros described in this section. The value is 2 on all machines we know of except the IBM 360 and derivatives.

`FLT_MANT_DIG`
: This is the number of base-`FLT_RADIX` digits in the floating-point mantissa for the `float` data type. The following expression yields `1.0` (even though mathematically it should not) due to the limited number of mantissa digits:

```
float radix = FLT_RADIX;


1.0f + 1.0f / radix / radix / ... / radix
```

  where `radix` appears `FLT_MANT_DIG` times.

`DBL_MANT_DIG`
`LDBL_MANT_DIG`
: This is the number of base-`FLT_RADIX` digits in the floating-point mantissa for the data types `double` and `long double`, respectively.

`FLT_DIG`

This is the number of decimal digits of precision for the `float` data type. Technically, if p and b are the precision and base (respectively) for the representation, then the decimal precision q is the maximum number of decimal digits such that any floating-point number with q base-10 digits can be rounded to a floating-point number with p base b digits and back again, without change to the q decimal digits.

The value of this macro is supposed to be at least `6`, to satisfy ISO C.

`DBL_DIG`
`LDBL_DIG`

These are similar to `FLT_DIG`, but for the data types `double` and `long double`, respectively. The values of these macros are supposed to be at least `10`.

`FLT_MIN_EXP`

This is the smallest possible exponent value for type `float`. More precisely, is the minimum negative integer such that the value `FLT_RADIX` raised to this power minus 1 can be represented as a normalized floating-point number of type `float`.

`DBL_MIN_EXP`
`LDBL_MIN_EXP`

These are similar to `FLT_MIN_EXP`, but for the data types `double` and `long double`, respectively.

`FLT_MIN_10_EXP`

This is the minimum negative integer such that `10` raised to this power minus 1 can be represented as a normalized floating-point number of type `float`. This is supposed to be `-37` or even less.

`DBL_MIN_10_EXP`
`LDBL_MIN_10_EXP`

These are similar to `FLT_MIN_10_EXP`, but for the data types `double` and `long double`, respectively.

`FLT_MAX_EXP`

This is the largest possible exponent value for type `float`. More precisely, this is the maximum positive integer such that value `FLT_RADIX` raised to this power minus 1 can be represented as a floating-point number of type `float`.

`DBL_MAX_EXP`
`LDBL_MAX_EXP`

These are similar to `FLT_MAX_EXP`, but for the data types `double` and `long double`, respectively.

`FLT_MAX_10_EXP`
: This is the maximum positive integer such that `10` raised to this power minus 1 can be represented as a normalized floating-point number of type `float`. This is supposed to be at least `37`.

`DBL_MAX_10_EXP`
`LDBL_MAX_10_EXP`
: These are similar to `FLT_MAX_10_EXP`, but for the data types `double` and `long double`, respectively.

`FLT_MAX`
: The value of this macro is the maximum number representable in type `float`. It is supposed to be at least `1E+37`. The value has type `float`.

  The smallest representable number is `- FLT_MAX`.

`DBL_MAX`
`LDBL_MAX`
: These are similar to `FLT_MAX`, but for the data types `double` and `long double`, respectively. The type of the macro's value is the same as the type it describes.

`FLT_MIN`
: The value of this macro is the minimum normalized positive floating-point number that is representable in type `float`. It is supposed to be no more than `1E-37`.

`DBL_MIN`
`LDBL_MIN`
: These are similar to `FLT_MIN`, but for the data types `double` and `long double`, respectively. The type of the macro's value is the same as the type it describes.

`FLT_EPSILON`
: This is the minimum positive floating-point number of type `float` such that `1.0 + FLT_EPSILON != 1.0` is true. It's supposed to be no greater than `1E-5`.

`DBL_EPSILON`
`LDBL_EPSILON`
: These are similar to `FLT_EPSILON`, but for the data types `double` and `long double`, respectively. The type of the macro's value is the same as the type it describes. The values are not supposed to be greater than `1E-9`.

A.5.3.3 IEEE Floating-Point

Here is an example showing how the floating-type measurements come out for the most common floating-point representation, specified by the IEEE *Standard for Binary Floating-Point Arithmetic (*ANSI/IEEE *Std 754-1985).* Nearly all computers designed since the 1980s use this format.

The IEEE single-precision float representation uses a base of 2. There is a sign bit, a mantissa with 23 bits plus 1 hidden bit (so the total precision is 24 base-2 digits), and an 8-bit exponent that can represent values in the range -125 to 128, inclusive.

So, for an implementation that uses this representation for the `float` data type, appropriate values for the corresponding parameters are

```
FLT_RADIX                          2
FLT_MANT_DIG                      24
FLT_DIG                            6
FLT_MIN_EXP                     -125
FLT_MIN_10_EXP                   -37
FLT_MAX_EXP                      128
FLT_MAX_10_EXP                   +38
FLT_MIN              1.17549435E-38F
FLT_MAX              3.40282347E+38F
FLT_EPSILON          1.19209290E-07F
```

Here are the values for the `double` data type:

```
DBL_MANT_DIG                      53
DBL_DIG                           15
DBL_MIN_EXP                    -1021
DBL_MIN_10_EXP                   -307
DBL_MAX_EXP                      1024
DBL_MAX_10_EXP                    308
DBL_MAX         1.7976931348623157E+308
DBL_MIN         2.2250738585072014E-308
DBL_EPSILON     2.220460492503131E-016
```

A.5.4 Structure Field Offset Measurement

You can use `offsetof` to measure the location within a structure type of a particular structure member.

`size_t` **offsetof** (*type, member*) Macro

This expands to a integer constant expression that is the offset of the structure member named *member* in a the structure type *type.* For example, `offsetof (struct s, elem)` is the offset, in bytes, of the member `elem` in a `struct s`.

This macro won't work if *member* is a bit field; you get an error from the C compiler in that case.

Appendix B Summary of Library Facilities

This appendix is a complete list of the facilities declared within the header files supplied with the GNU C Library. Each entry also lists the standard or other source from which each facility is derived, and tells you where in the manual you can find more information about how to use it.

`int accept (int` *socket*`, struct sockaddr *`*addr*`, socklen_t *`*length_ptr*`)`
'`sys/socket.h`' (BSD): Section 5.9.3 [Accepting Connections], page 155.

`int access (const char *`*filename*`, int` *how*`)`
'`unistd.h`' (POSIX.1): Section 3.9.8 [Testing Permission to Access a File], page 106.

`ACCOUNTING`
'`utmp.h`' (SVID): Section 10.12.1 [Manipulating the User-Accounting Database], page 265.

`int addmntent (FILE *`*stream*`, const struct mntent *`*mnt*`)`
'`mntent.h`' (BSD): Section 11.3.1.2 [The '`mtab`' File], page 292.

`AF_FILE`
'`sys/socket.h`' (GNU): Section 5.3.1 [Address Formats], page 128.

`AF_INET`
'`sys/socket.h`' (BSD): Section 5.3.1 [Address Formats], page 128.

`AF_INET6`
'`sys/socket.h`' (IPv6 Basic API): Section 5.3.1 [Address Formats], page 128.

`AF_LOCAL`
'`sys/socket.h`' (POSIX): Section 5.3.1 [Address Formats], page 128.

`AF_UNIX`
'`sys/socket.h`' (BSD, Unix98): Section 5.3.1 [Address Formats], page 128.

`AF_UNSPEC`
'`sys/socket.h`' (BSD): Section 5.3.1 [Address Formats], page 128.

`int aio_cancel (int` *fildes*`, struct aiocb *`*aiocbp*`)`
'`aio.h`' (POSIX.1b): Section 2.10.4 [Cancellation of AIO Operations], page 52.

`int aio_cancel64 (int` *fildes*`, struct aiocb64 *`*aiocbp*`)`
'`aio.h`' (Unix98): Section 2.10.4 [Cancellation of AIO Operations], page 52.

`int aio_error (const struct aiocb *`*aiocbp*`)`
'`aio.h`' (POSIX.1b): Section 2.10.2 [Getting the Status of AIO Operations], page 49.

`int aio_error64 (const struct aiocb64 *`*aiocbp*`)`
'`aio.h`' (Unix98): Section 2.10.2 [Getting the Status of AIO Operations], page 49.

`int aio_fsync (int` *op*`, struct aiocb *`*aiocbp*`)`
'`aio.h`' (POSIX.1b): Section 2.10.3 [Getting into a Consistent State], page 50.

`int aio_fsync64 (int` *op*`, struct aiocb64 *`*aiocbp*`)`
'`aio.h`' (Unix98): Section 2.10.3 [Getting into a Consistent State], page 50.

`B110`
: ‘`termios.h`’ (POSIX.1): Section 6.4.8 [Line Speed], page 192.

`B115200`
: ‘`termios.h`’ (GNU): Section 6.4.8 [Line Speed], page 192.

`B1200`
: ‘`termios.h`’ (POSIX.1): Section 6.4.8 [Line Speed], page 192.

`B134`
: ‘`termios.h`’ (POSIX.1): Section 6.4.8 [Line Speed], page 192.

`B150`
: ‘`termios.h`’ (POSIX.1): Section 6.4.8 [Line Speed], page 192.

`B1800`
: ‘`termios.h`’ (POSIX.1): Section 6.4.8 [Line Speed], page 192.

`B19200`
: ‘`termios.h`’ (POSIX.1): Section 6.4.8 [Line Speed], page 192.

`B200`
: ‘`termios.h`’ (POSIX.1): Section 6.4.8 [Line Speed], page 192.

`B230400`
: ‘`termios.h`’ (GNU): Section 6.4.8 [Line Speed], page 192.

`B2400`
: ‘`termios.h`’ (POSIX.1): Section 6.4.8 [Line Speed], page 192.

`B300`
: ‘`termios.h`’ (POSIX.1): Section 6.4.8 [Line Speed], page 192.

`B38400`
: ‘`termios.h`’ (POSIX.1): Section 6.4.8 [Line Speed], page 192.

`B460800`
: ‘`termios.h`’ (GNU): Section 6.4.8 [Line Speed], page 192.

`B4800`
: ‘`termios.h`’ (POSIX.1): Section 6.4.8 [Line Speed], page 192.

`B50`
: ‘`termios.h`’ (POSIX.1): Section 6.4.8 [Line Speed], page 192.

`B57600`
: ‘`termios.h`’ (GNU): Section 6.4.8 [Line Speed], page 192.

`B600`
: ‘`termios.h`’ (POSIX.1): Section 6.4.8 [Line Speed], page 192.

`B75`
: ‘`termios.h`’ (POSIX.1): Section 6.4.8 [Line Speed], page 192.

`B9600`
: ‘`termios.h`’ (POSIX.1): Section 6.4.8 [Line Speed], page 192.

`void cfmakeraw (struct termios *termios-p)`
'`termios.h`' (BSD): Section 6.4.10 [Noncanonical Input], page 198.

`int cfsetispeed (struct termios *termios-p, speed_t speed)`
'`termios.h`' (POSIX.1): Section 6.4.8 [Line Speed], page 192.

`int cfsetospeed (struct termios *termios-p, speed_t speed)`
'`termios.h`' (POSIX.1): Section 6.4.8 [Line Speed], page 192.

`int cfsetspeed (struct termios *termios-p, speed_t speed)`
'`termios.h`' (BSD): Section 6.4.8 [Line Speed], page 192.

`CHAR_BIT`
'`limits.h`' (ISO): Section A.5.1 [Computing the Width of an Integer Data Type], page 465.

`CHAR_MAX`
'`limits.h`' (ISO): Section A.5.2 [Range of an Integer Type], page 465.

`CHAR_MIN`
'`limits.h`' (ISO): Section A.5.2 [Range of an Integer Type], page 465.

`int chdir (const char *filename)`
'`unistd.h`' (POSIX.1): Section 3.1 [Working Directory], page 71.

`int CHILD_MAX`
'`limits.h`' (POSIX.1): Section 12.1 [General Capacity-Limits], page 303.

`int chmod (const char *filename, mode_t mode)`
'`sys/stat.h`' (POSIX.1): Section 3.9.7 [Assigning File Permissions], page 104.

`int chown (const char *filename, uid_t owner, gid_t group)`
'`unistd.h`' (POSIX.1): Section 3.9.4 [File Owner], page 101.

`tcflag_t CIGNORE`
'`termios.h`' (BSD): Section 6.4.6 [Control Modes], page 187.

`tcflag_t CLOCAL`
'`termios.h`' (POSIX.1): Section 6.4.6 [Control Modes], page 187.

`int close (int filedes)`
'`unistd.h`' (POSIX.1): Section 2.1 [Opening and Closing Files], page 17.

`int closedir (DIR *dirstream)`
'`dirent.h`' (POSIX.1): Section 3.2.3 [Reading and Closing a Directory Stream], page 76.

`void closelog (void)`
'`syslog.h`' (BSD): Section 15.2.3 [closelog], page 365.

`int COLL_WEIGHTS_MAX`
'`limits.h`' (POSIX.2): Section 12.10 [Utility Program Capacity-Limits], page 323.

`size_t confstr (int parameter, char *buf, size_t len)`
'`unistd.h`' (POSIX.2): Section 12.12 [String-Valued Parameters], page 324.

`int connect (int socket, struct sockaddr *addr, socklen_t length)`
'`sys/socket.h`' (BSD): Section 5.9.1 [Making a Connection], page 153.

`_CS_LFS_CFLAGS`
: ‘`unistd.h`’ (Unix98): Section 12.12 [String-Valued Parameters], page 324.

`_CS_LFS_LDFLAGS`
: ‘`unistd.h`’ (Unix98): Section 12.12 [String-Valued Parameters], page 324.

`_CS_LFS_LIBS`
: ‘`unistd.h`’ (Unix98): Section 12.12 [String-Valued Parameters], page 324.

`_CS_LFS_LINTFLAGS`
: ‘`unistd.h`’ (Unix98): Section 12.12 [String-Valued Parameters], page 324.

`_CS_PATH`
: ‘`unistd.h`’ (POSIX.2): Section 12.12 [String-Valued Parameters], page 324.

`tcflag_t CSTOPB`
: ‘`termios.h`’ (POSIX.1): Section 6.4.6 [Control Modes], page 187.

`char * ctermid (char *`*string*`)`
: ‘`stdio.h`’ (POSIX.1): Section 8.7.1 [Identifying the Controlling Terminal], page 238.

`char * cuserid (char *`*string*`)`
: ‘`stdio.h`’ (POSIX.1): Section 10.11 [Identifying Who Is Logged In], page 264.

`DBL_DIG`
: ‘`float.h`’ (ISO): Section A.5.3.2 [Floating-Point Parameters], page 468.

`DBL_EPSILON`
: ‘`float.h`’ (ISO): Section A.5.3.2 [Floating-Point Parameters], page 468.

`DBL_MANT_DIG`
: ‘`float.h`’ (ISO): Section A.5.3.2 [Floating-Point Parameters], page 468.

`DBL_MAX`
: ‘`float.h`’ (ISO): Section A.5.3.2 [Floating-Point Parameters], page 468.

`DBL_MAX_10_EXP`
: ‘`float.h`’ (ISO): Section A.5.3.2 [Floating-Point Parameters], page 468.

`DBL_MAX_EXP`
: ‘`float.h`’ (ISO): Section A.5.3.2 [Floating-Point Parameters], page 468.

`DBL_MIN`
: ‘`float.h`’ (ISO): Section A.5.3.2 [Floating-Point Parameters], page 468.

`DBL_MIN_10_EXP`
: ‘`float.h`’ (ISO): Section A.5.3.2 [Floating-Point Parameters], page 468.

`DBL_MIN_EXP`
: ‘`float.h`’ (ISO): Section A.5.3.2 [Floating-Point Parameters], page 468.

`DEAD_PROCESS`
: ‘`utmp.h`’ (SVID): Section 10.12.1 [Manipulating the User-Accounting Database], page 265.

`DEAD_PROCESS`
: ‘`utmpx.h`’ (XPG4.2): Section 10.12.2 [XPG User-Accounting Database Functions], page 270.

`tcflag_t ECHOK`
: ‘`termios.h`’ (POSIX.1): Section 6.4.7 [Local Modes], page 189.

`tcflag_t ECHOKE`
: ‘`termios.h`’ (BSD): Section 6.4.7 [Local Modes], page 189.

`tcflag_t ECHONL`
: ‘`termios.h`’ (POSIX.1): Section 6.4.7 [Local Modes], page 189.

`tcflag_t ECHOPRT`
: ‘`termios.h`’ (BSD): Section 6.4.7 [Local Modes], page 189.

`EMPTY`
: ‘`utmp.h`’ (SVID): Section 10.12.1 [Manipulating the User-Accounting Database], page 265.

`EMPTY`
: ‘`utmpx.h`’ (XPG4.2): Section 10.12.2 [XPG User-Accounting Database Functions], page 270.

`void encrypt (char *`*block*`, int` *edflag*`)`
: ‘`crypt.h`’ (BSD, SVID): Section 13.4 [DES Encryption], page 331.

`void encrypt_r (char *`*block*`, int` *edflag*`, struct crypt_data *` *data*`)`
: ‘`crypt.h`’ (GNU): Section 13.4 [DES Encryption], page 331.

`void endfsent (void)`
: ‘`fstab.h`’ (BSD): Section 11.3.1.1 [The ‘`fstab`’ File], page 290.

`void endgrent (void)`
: ‘`grp.h`’ (SVID, BSD): Section 10.14.3 [Scanning the List of All Groups], page 278.

`void endhostent (void)`
: ‘`netdb.h`’ (BSD): Section 5.6.2.4 [Host Names], page 141.

`int endmntent (FILE *`*stream*`)`
: ‘`mntent.h`’ (BSD): Section 11.3.1.2 [The ‘`mtab`’ File], page 292.

`void endnetent (void)`
: ‘`netdb.h`’ (BSD): Section 5.13 [Networks Database], page 176.

`void endnetgrent (void)`
: ‘`netdb.h`’ (BSD): Section 10.16.2 [Looking Up One Netgroup], page 282.

`void endprotoent (void)`
: ‘`netdb.h`’ (BSD): Section 5.6.6 [Protocols Database], page 147.

`void endpwent (void)`
: ‘`pwd.h`’ (SVID, BSD): Section 10.13.3 [Scanning the List of All Users], page 275.

`void endservent (void)`
: ‘`netdb.h`’ (BSD): Section 5.6.4 [The Services Database], page 145.

`void endutent (void)`
: ‘`utmp.h`’ (SVID): Section 10.12.1 [Manipulating the User-Accounting Database], page 265.

`void endutxent (void)`
: ‘`utmpx.h`’ (XPG4.2): Section 10.12.2 [XPG User-Accounting Database Functions], page 270.

`int FD_SETSIZE`
'`sys/types.h`' (BSD): Section 2.8 [Waiting for Input or Output], page 37.

`int F_DUPFD`
'`fcntl.h`' (POSIX.1): Section 2.12 [Duplicating Descriptors], page 55.

`void FD_ZERO (fd_set *`*set*`)`
'`sys/types.h`' (BSD): Section 2.8 [Waiting for Input or Output], page 37.

`int F_GETFD`
'`fcntl.h`' (POSIX.1): Section 2.13 [File-Descriptor Flags], page 57.

`int F_GETFL`
'`fcntl.h`' (POSIX.1): Section 2.14.4 [Getting and Setting File Status Flags], page 63.

`struct group * fgetgrent (FILE *`*stream*`)`
'`grp.h`' (SVID): Section 10.14.3 [Scanning the List of All Groups], page 278.

`int fgetgrent_r (FILE *`*stream*`, struct group *`*result_buf*`, char *`*buffer*`, size_t` *buflen*`, struct group **`*result*`)`
'`grp.h`' (GNU): Section 10.14.3 [Scanning the List of All Groups], page 278.

`int F_GETLK`
'`fcntl.h`' (POSIX.1): Section 2.15 [File Locks], page 64.

`int F_GETOWN`
'`fcntl.h`' (BSD): Section 2.16 [Interrupt-Driven Input], page 68.

`struct passwd * fgetpwent (FILE *`*stream*`)`
'`pwd.h`' (SVID): Section 10.13.3 [Scanning the List of All Users], page 275.

`int fgetpwent_r (FILE *`*stream*`, struct passwd *`*result_buf*`, char *`*buffer*`, size_t` *buflen*`, struct passwd **`*result*`)`
'`pwd.h`' (GNU): Section 10.13.3 [Scanning the List of All Users], page 275.

`int FILENAME_MAX`
'`stdio.h`' (ISO): Section 12.6 [Limits on File-System Capacity], page 318.

`int fileno (FILE *`*stream*`)`
'`stdio.h`' (POSIX.1): Section 2.4 [Descriptors and Streams], page 28.

`int fileno_unlocked (FILE *`*stream*`)`
'`stdio.h`' (GNU): Section 2.4 [Descriptors and Streams], page 28.

`FLT_DIG`
'`float.h`' (ISO): Section A.5.3.2 [Floating-Point Parameters], page 468.

`FLT_EPSILON`
'`float.h`' (ISO): Section A.5.3.2 [Floating-Point Parameters], page 468.

`FLT_MANT_DIG`
'`float.h`' (ISO): Section A.5.3.2 [Floating-Point Parameters], page 468.

`FLT_MAX`
'`float.h`' (ISO): Section A.5.3.2 [Floating-Point Parameters], page 468.

`FLT_MAX_10_EXP`
'`float.h`' (ISO): Section A.5.3.2 [Floating-Point Parameters], page 468.

`FPE_SUBRNG_TRAP`
: ‘`signal.h`’ (BSD): Section 17.2.1 [Program-Error Signals], page 379.

`F_RDLCK`
: ‘`fcntl.h`’ (POSIX.1): Section 2.15 [File Locks], page 64.

`int F_SETFD`
: ‘`fcntl.h`’ (POSIX.1): Section 2.13 [File-Descriptor Flags], page 57.

`int F_SETFL`
: ‘`fcntl.h`’ (POSIX.1): Section 2.14.4 [Getting and Setting File Status Flags], page 63.

`int F_SETLK`
: ‘`fcntl.h`’ (POSIX.1): Section 2.15 [File Locks], page 64.

`int F_SETLKW`
: ‘`fcntl.h`’ (POSIX.1): Section 2.15 [File Locks], page 64.

`int F_SETOWN`
: ‘`fcntl.h`’ (BSD): Section 2.16 [Interrupt-Driven Input], page 68.

`int fstat (int` *filedes*`, struct stat *`*buf*`)`
: ‘`sys/stat.h`’ (POSIX.1): Section 3.9.2 [Reading the Attributes of a File], page 97.

`int fstat64 (int` *filedes*`, struct stat64 *`*buf*`)`
: ‘`sys/stat.h`’ (Unix98): Section 3.9.2 [Reading the Attributes of a File], page 97.

`int fsync (int` *fildes*`)`
: ‘`unistd.h`’ (POSIX): Section 2.9 [Synchronizing I/O Operations], page 40.

`int ftruncate (int` *fd*`, off_t` *length*`)`
: ‘`unistd.h`’ (POSIX): Section 3.9.10 [File Size], page 110.

`int ftruncate64 (int` *id*`, off64_t` *length*`)`
: ‘`unistd.h`’ (Unix98): Section 3.9.10 [File Size], page 110.

`int ftw (const char *`*filename*`, __ftw_func_t` *func*`, int` *descriptors*`)`
: ‘`ftw.h`’ (SVID): Section 3.3 [Working with Directory Trees], page 81.

`int ftw64 (const char *`*filename*`, __ftw64_func_t` *func*`, int` *descriptors*`)`
: ‘`ftw.h`’ (Unix98): Section 3.3 [Working with Directory Trees], page 81.

`__ftw64_func_t`
: ‘`ftw.h`’ (GNU): Section 3.3 [Working with Directory Trees], page 81.

`__ftw_func_t`
: ‘`ftw.h`’ (GNU): Section 3.3 [Working with Directory Trees], page 81.

`F_UNLCK`
: ‘`fcntl.h`’ (POSIX.1): Section 2.15 [File Locks], page 64.

`int futimes (int *`*fd*`, struct timeval` *tvp*`[2])`
: ‘`sys/time.h`’ (BSD): Section 3.9.9 [File Times], page 108.

`F_WRLCK`
: ‘`fcntl.h`’ (POSIX.1): Section 2.15 [File Locks], page 64.

`struct hostent * gethostbyaddr (const char *addr, size_t length, int format)`
'`netdb.h`' (BSD): Section 5.6.2.4 [Host Names], page 141.

`int gethostbyaddr_r (const char *addr, size_t length, int format, struct hostent *restrict result_buf, char *restrict buf, size_t buflen, struct hostent **restrict result, int *restrict h_errnop)`
'`netdb.h`' (GNU): Section 5.6.2.4 [Host Names], page 141.

`struct hostent * gethostbyname (const char *name)`
'`netdb.h`' (BSD): Section 5.6.2.4 [Host Names], page 141.

`struct hostent * gethostbyname2 (const char *name, int af)`
'`netdb.h`' (IPv6 Basic API): Section 5.6.2.4 [Host Names], page 141.

`int gethostbyname2_r (const char *name, int af, struct hostent *restrict result_buf, char *restrict buf, size_t buflen, struct hostent **restrict result, int *restrict h_errnop)`
'`netdb.h`' (GNU): Section 5.6.2.4 [Host Names], page 141.

`int gethostbyname_r (const char *restrict name, struct hostent *restrict result_buf, char *restrict buf, size_t buflen, struct hostent **restrict result, int *restrict h_errnop)`
'`netdb.h`' (GNU): Section 5.6.2.4 [Host Names], page 141.

`struct hostent * gethostent (void)`
'`netdb.h`' (BSD): Section 5.6.2.4 [Host Names], page 141.

`long int gethostid (void)`
'`unistd.h`' (BSD): Section 11.1 [Host Identification], page 285.

`int gethostname (char *name, size_t size)`
'`unistd.h`' (BSD): Section 11.1 [Host Identification], page 285.

`int getloadavg (double loadavg[], int nelem)`
'`stdlib.h`' (BSD): Section 14.5 [Learn About the Processors Available], page 356.

`char * getlogin (void)`
'`unistd.h`' (POSIX.1): Section 10.11 [Identifying Who Is Logged In], page 264.

`struct mntent * getmntent (FILE *stream)`
'`mntent.h`' (BSD): Section 11.3.1.2 [The '`mtab`' File], page 292.

`struct mntent * getmntent_r (FILE *stream, struct mntent *result, char *buffer, int bufsize)`
'`mntent.h`' (BSD): Section 11.3.1.2 [The '`mtab`' File], page 292.

`struct netent * getnetbyaddr (unsigned long int net, int type)`
'`netdb.h`' (BSD): Section 5.13 [Networks Database], page 176.

`struct netent * getnetbyname (const char *name)`
'`netdb.h`' (BSD): Section 5.13 [Networks Database], page 176.

`struct netent * getnetent (void)`
'`netdb.h`' (BSD): Section 5.13 [Networks Database], page 176.

`int getnetgrent (char **hostp, char **userp, char **domainp)`
'`netdb.h`' (BSD): Section 10.16.2 [Looking Up One Netgroup], page 282.

`int getnetgrent_r (char **hostp, char **userp, char **domainp, char *buffer, int buflen)`
'`netdb.h`' (GNU): Section 10.16.2 [Looking Up One Netgroup], page 282.

`struct passwd * getpwnam (const char *`*name*`)`
'`pwd.h`' (POSIX.1): Section 10.13.2 [Looking Up One User], page 274.

`int getpwnam_r (const char *`*name*`, struct passwd *`*result_buf*`, char *`*buffer*`, size_t` *buflen*`, struct passwd **`*result*`)`
'`pwd.h`' (POSIX.1c): Section 10.13.2 [Looking Up One User], page 274.

`struct passwd * getpwuid (uid_t` *uid*`)`
'`pwd.h`' (POSIX.1): Section 10.13.2 [Looking Up One User], page 274.

`int getpwuid_r (uid_t` *uid*`, struct passwd *`*result_buf*`, char *`*buffer*`, size_t` *buflen*`, struct passwd **`*result*`)`
'`pwd.h`' (POSIX.1c): Section 10.13.2 [Looking Up One User], page 274.

`int getrlimit (int` *resource*`, struct rlimit *`*rlp*`)`
'`sys/resource.h`' (BSD): Section 14.2 [Limiting Resource Usage], page 338.

`int getrlimit64 (int` *resource*`, struct rlimit64 *`*rlp*`)`
'`sys/resource.h`' (Unix98): Section 14.2 [Limiting Resource Usage], page 338.

`int getrusage (int` *processes*`, struct rusage *`*rusage*`)`
'`sys/resource.h`' (BSD): Section 14.1 [Resource Usage], page 335.

`struct servent * getservbyname (const char *`*name*`, const char *`*proto*`)`
'`netdb.h`' (BSD): Section 5.6.4 [The Services Database], page 145.

`struct servent * getservbyport (int` *port*`, const char *`*proto*`)`
'`netdb.h`' (BSD): Section 5.6.4 [The Services Database], page 145.

`struct servent * getservent (void)`
'`netdb.h`' (BSD): Section 5.6.4 [The Services Database], page 145.

`pid_t getsid (pid_t` *pid*`)`
'`unistd.h`' (SVID): Section 8.7.2 [Process-Group Functions], page 239.

`int getsockname (int` *socket*`, struct sockaddr *`*addr*`, socklen_t *`*length-ptr*`)`
'`sys/socket.h`' (BSD): Section 5.3.3 [Reading the Address of a Socket], page 130.

`int getsockopt (int` *socket*`, int` *level*`, int` *optname*`, void *`*optval*`, socklen_t` *`*optlen-ptr*`)`
'`sys/socket.h`' (BSD): Section 5.12.1 [Socket Option Functions], page 173.

`uid_t getuid (void)`
'`unistd.h`' (POSIX.1): Section 10.5 [Reading the Persona of a Process], page 255.

`mode_t getumask (void)`
'`sys/stat.h`' (GNU): Section 3.9.7 [Assigning File Permissions], page 104.

`struct utmp * getutent (void)`
'`utmp.h`' (SVID): Section 10.12.1 [Manipulating the User-Accounting Database], page 265.

`int getutent_r (struct utmp *`*buffer*`, struct utmp **`*result*`)`
'`utmp.h`' (GNU): Section 10.12.1 [Manipulating the User-Accounting Database], page 265.

`struct utmp * getutid (const struct utmp *`*id*`)`
'`utmp.h`' (SVID): Section 10.12.1 [Manipulating the User-Accounting Database], page 265.

`uint32_t htonl (uint32_t` *hostlong*`)`
: 'netinet/in.h' (BSD): Section 5.6.5 [Byte-Order Conversion], page 147.

`uint16_t htons (uint16_t` *hostshort*`)`
: 'netinet/in.h' (BSD): Section 5.6.5 [Byte-Order Conversion], page 147.

`tcflag_t HUPCL`
: 'termios.h' (POSIX.1): Section 6.4.6 [Control Modes], page 187.

`tcflag_t ICANON`
: 'termios.h' (POSIX.1): Section 6.4.7 [Local Modes], page 189.

`tcflag_t ICRNL`
: 'termios.h' (POSIX.1): Section 6.4.4 [Input Modes], page 185.

`tcflag_t IEXTEN`
: 'termios.h' (POSIX.1): Section 6.4.7 [Local Modes], page 189.

`void if_freenameindex (struct if_nameindex *ptr)`
: 'net/if.h' (IPv6 basic API): Section 5.4 [Interface Naming], page 130.

`char * if_indextoname (unsigned int ifindex, char *ifname)`
: 'net/if.h' (IPv6 basic API): Section 5.4 [Interface Naming], page 130.

`struct if_nameindex * if_nameindex (void)`
: 'net/if.h' (IPv6 basic API): Section 5.4 [Interface Naming], page 130.

`unsigned int if_nametoindex (const char *ifname)`
: 'net/if.h' (IPv6 basic API): Section 5.4 [Interface Naming], page 130.

`size_t IFNAMSIZ`
: 'net/if.h' (net/if.h): Section 5.4 [Interface Naming], page 130.

`int IFTODT (mode_t` *mode*`)`
: 'dirent.h' (BSD): Section 3.2.1 [Format of a Directory Entry], page 73.

`tcflag_t IGNBRK`
: 'termios.h' (POSIX.1): Section 6.4.4 [Input Modes], page 185.

`tcflag_t IGNCR`
: 'termios.h' (POSIX.1): Section 6.4.4 [Input Modes], page 185.

`tcflag_t IGNPAR`
: 'termios.h' (POSIX.1): Section 6.4.4 [Input Modes], page 185.

`tcflag_t IMAXBEL`
: 'termios.h' (BSD): Section 6.4.4 [Input Modes], page 185.

`struct in6_addr in6addr_any`
: 'netinet/in.h' (IPv6 basic API): Section 5.6.2.2 [Host-Address Data Type], page 138.

`struct in6_addr in6addr_loopback`
: 'netinet/in.h' (IPv6 basic API): Section 5.6.2.2 [Host-Address Data Type], page 138.

`uint32_t INADDR_ANY`
: 'netinet/in.h' (BSD): Section 5.6.2.2 [Host-Address Data Type], page 138.

`uint32_t INADDR_BROADCAST`
: 'netinet/in.h' (BSD): Section 5.6.2.2 [Host-Address Data Type], page 138.

`tcflag_t INPCK`
: ‘`termios.h`’ (POSIX.1): Section 6.4.4 [Input Modes], page 185.

`INT_MAX`
: ‘`limits.h`’ (ISO): Section A.5.2 [Range of an Integer Type], page 465.

`INT_MIN`
: ‘`limits.h`’ (ISO): Section A.5.2 [Range of an Integer Type], page 465.

`int ioctl (int` *filedes*`, int` *command*`, ...)`
: ‘`sys/ioctl.h`’ (BSD): Section 2.17 [Generic I/O Control Operations], page 69.

`int IPPORT_RESERVED`
: ‘`netinet/in.h`’ (BSD): Section 5.6.3 [Internet Ports], page 144.

`int IPPORT_USERRESERVED`
: ‘`netinet/in.h`’ (BSD): Section 5.6.3 [Internet Ports], page 144.

`int isatty (int` *filedes*`)`
: ‘`unistd.h`’ (POSIX.1): Section 6.1 [Identifying Terminals], page 179.

`tcflag_t ISIG`
: ‘`termios.h`’ (POSIX.1): Section 6.4.7 [Local Modes], page 189.

`_ISOC99_SOURCE`
: (GNU): Section 1.3.4 [Feature-Test Macros], page 8.

`tcflag_t ISTRIP`
: ‘`termios.h`’ (POSIX.1): Section 6.4.4 [Input Modes], page 185.

`tcflag_t IXANY`
: ‘`termios.h`’ (BSD): Section 6.4.4 [Input Modes], page 185.

`tcflag_t IXOFF`
: ‘`termios.h`’ (POSIX.1): Section 6.4.4 [Input Modes], page 185.

`tcflag_t IXON`
: ‘`termios.h`’ (POSIX.1): Section 6.4.4 [Input Modes], page 185.

`jmp_buf`
: ‘`setjmp.h`’ (ISO): Section 16.2 [Details of Nonlocal Exits], page 369.

`int kill (pid_t` *pid*`, int` *signum*`)`
: ‘`signal.h`’ (POSIX.1): Section 17.6.2 [Signaling Another Process], page 410.

`int killpg (int` *pgid*`, int` *signum*`)`
: ‘`signal.h`’ (BSD): Section 17.6.2 [Signaling Another Process], page 410.

`int L_ctermid`
: ‘`stdio.h`’ (POSIX.1): Section 8.7.1 [Identifying the Controlling Terminal], page 238.

`int L_cuserid`
: ‘`stdio.h`’ (POSIX.1): Section 10.11 [Identifying Who Is Logged In], page 264.

`int LINE_MAX`
: ‘`limits.h`’ (POSIX.2): Section 12.10 [Utility Program Capacity-Limits], page 323.

`int link (const char *`*oldname*`, const char *`*newname*`)`
: ‘`unistd.h`’ (POSIX.1): Section 3.4 [Hard Links], page 85.

`int lstat (const char *filename, struct stat *buf)`
: ‘`sys/stat.h`’ (BSD): Section 3.9.2 [Reading the Attributes of a File], page 97.

`int lstat64 (const char *filename, struct stat64 *buf)`
: ‘`sys/stat.h`’ (Unix98): Section 3.9.2 [Reading the Attributes of a File], page 97.

`int L_tmpnam`
: ‘`stdio.h`’ (ISO): Section 3.11 [Temporary Files], page 114.

`int lutimes (const char *filename, struct timeval tvp[2])`
: ‘`sys/time.h`’ (BSD): Section 3.9.9 [File Times], page 108.

`int madvise (void *addr, size_t length, int advice)`
: ‘`sys/mman.h`’ (POSIX): Section 2.7 [Memory-Mapped I/O], page 32.

`void makecontext (ucontext_t *ucp, void (*func) (void), int argc, ...)`
: ‘`ucontext.h`’ (SVID): Section 16.4 [Complete Context Control], page 370.

`int MAX_CANON`
: ‘`limits.h`’ (POSIX.1): Section 12.6 [Limits on File-System Capacity], page 318.

`int MAX_INPUT`
: ‘`limits.h`’ (POSIX.1): Section 12.6 [Limits on File-System Capacity], page 318.

`int MAXNAMLEN`
: ‘`dirent.h`’ (BSD): Section 12.6 [Limits on File-System Capacity], page 318.

`int MAXSYMLINKS`
: ‘`sys/param.h`’ (BSD): Section 3.5 [Symbolic Links], page 87.

`tcflag_t MDMBUF`
: ‘`termios.h`’ (BSD): Section 6.4.6 [Control Modes], page 187.

`int mkdir (const char *filename, mode_t mode)`
: ‘`sys/stat.h`’ (POSIX.1): Section 3.8 [Creating Directories], page 92.

`char * mkdtemp (char *template)`
: ‘`stdlib.h`’ (BSD): Section 3.11 [Temporary Files], page 114.

`int mkfifo (const char *filename, mode_t mode)`
: ‘`sys/stat.h`’ (POSIX.1): Section 4.3 [FIFO Special Files], page 123.

`int mknod (const char *filename, int mode, int dev)`
: ‘`sys/stat.h`’ (BSD): Section 3.10 [Making Special Files], page 113.

`int mkstemp (char *template)`
: ‘`stdlib.h`’ (BSD): Section 3.11 [Temporary Files], page 114.

`char * mktemp (char *template)`
: ‘`stdlib.h`’ (Unix): Section 3.11 [Temporary Files], page 114.

`void * mmap (void *address, size_t length, int protect, int flags, int filedes, off_t offset)`
: ‘`sys/mman.h`’ (POSIX): Section 2.7 [Memory-Mapped I/O], page 32.

`void * mmap64 (void *address, size_t length, int protect, int flags, int filedes, off64_t offset)`
: ‘`sys/mman.h`’ (LFS): Section 2.7 [Memory-Mapped I/O], page 32.

`mode_t`
: ‘`sys/types.h`’ (POSIX.1): Section 3.9.1 [The Meaning of the File Attributes], page 93.

`FILE * popen (const char *command, const char *mode)`
: 'stdio.h' (POSIX.2, SVID, BSD): Section 4.2 [Pipe to a Subprocess], page 121.

`_POSIX2_BC_BASE_MAX`
: 'limits.h' (POSIX.2): Section 12.11 [Minimum Values for Utility Limits], page 324.

`_POSIX2_BC_DIM_MAX`
: 'limits.h' (POSIX.2): Section 12.11 [Minimum Values for Utility Limits], page 324.

`_POSIX2_BC_SCALE_MAX`
: 'limits.h' (POSIX.2): Section 12.11 [Minimum Values for Utility Limits], page 324.

`_POSIX2_BC_STRING_MAX`
: 'limits.h' (POSIX.2): Section 12.11 [Minimum Values for Utility Limits], page 324.

`int _POSIX2_C_DEV`
: 'unistd.h' (POSIX.2): Section 12.2 [Overall System Options], page 305.

`_POSIX2_COLL_WEIGHTS_MAX`
: 'limits.h' (POSIX.2): Section 12.11 [Minimum Values for Utility Limits], page 324.

`long int _POSIX2_C_VERSION`
: 'unistd.h' (POSIX.2): Section 12.3 [Which Version of POSIX is Supported], page 306.

`_POSIX2_EQUIV_CLASS_MAX`
: 'limits.h' (POSIX.2): Section 12.11 [Minimum Values for Utility Limits], page 324.

`_POSIX2_EXPR_NEST_MAX`
: 'limits.h' (POSIX.2): Section 12.11 [Minimum Values for Utility Limits], page 324.

`int _POSIX2_FORT_DEV`
: 'unistd.h' (POSIX.2): Section 12.2 [Overall System Options], page 305.

`int _POSIX2_FORT_RUN`
: 'unistd.h' (POSIX.2): Section 12.2 [Overall System Options], page 305.

`_POSIX2_LINE_MAX`
: 'limits.h' (POSIX.2): Section 12.11 [Minimum Values for Utility Limits], page 324.

`int _POSIX2_LOCALEDEF`
: 'unistd.h' (POSIX.2): Section 12.2 [Overall System Options], page 305.

`_POSIX2_RE_DUP_MAX`
: 'limits.h' (POSIX.2): Section 12.5 [Minimum Values for General Capacity-Limits], page 317.

`int _POSIX2_SW_DEV`
: 'unistd.h' (POSIX.2): Section 12.2 [Overall System Options], page 305.

`_POSIX_AIO_LISTIO_MAX`
: 'limits.h' (POSIX.1): Section 12.5 [Minimum Values for General Capacity-Limits], page 317.

`POSIX_REC_MAX_XFER_SIZE`
: ‘`limits.h`’ (POSIX.1): Section 12.8 [Minimum Values for File-System Limits], page 320.

`POSIX_REC_MIN_XFER_SIZE`
: ‘`limits.h`’ (POSIX.1): Section 12.8 [Minimum Values for File-System Limits], page 320.

`POSIX_REC_XFER_ALIGN`
: ‘`limits.h`’ (POSIX.1): Section 12.8 [Minimum Values for File-System Limits], page 320.

`int _POSIX_SAVED_IDS`
: ‘`unistd.h`’ (POSIX.1): Section 12.2 [Overall System Options], page 305.

`_POSIX_SOURCE`
: (POSIX.1): Section 1.3.4 [Feature-Test Macros], page 8.

`_POSIX_SSIZE_MAX`
: ‘`limits.h`’ (POSIX.1): Section 12.5 [Minimum Values for General Capacity-Limits], page 317.

`_POSIX_STREAM_MAX`
: ‘`limits.h`’ (POSIX.1): Section 12.5 [Minimum Values for General Capacity-Limits], page 317.

`_POSIX_TZNAME_MAX`
: ‘`limits.h`’ (POSIX.1): Section 12.5 [Minimum Values for General Capacity-Limits], page 317.

`unsigned char _POSIX_VDISABLE`
: ‘`unistd.h`’ (POSIX.1): Section 12.7 [Optional Features in File Support], page 319.

`long int _POSIX_VERSION`
: ‘`unistd.h`’ (POSIX.1): Section 12.3 [Which Version of POSIX is Supported], page 306.

`ssize_t pread (int` *filedes*`, void *`*buffer*`, size_t` *size*`, off_t` *offset*`)`
: ‘`unistd.h`’ (Unix98): Section 2.2 [Input and Output Primitives], page 20.

`ssize_t pread64 (int` *filedes*`, void *`*buffer*`, size_t` *size*`, off64_t` *offset*`)`
: ‘`unistd.h`’ (Unix98): Section 2.2 [Input and Output Primitives], page 20.

`PRIO_MAX`
: ‘`sys/resource.h`’ (BSD): Section 14.3.4.2 [Functions for Traditional Scheduling], page 350.

`PRIO_MIN`
: ‘`sys/resource.h`’ (BSD): Section 14.3.4.2 [Functions for Traditional Scheduling], page 350.

`PRIO_PGRP`
: ‘`sys/resource.h`’ (BSD): Section 14.3.4.2 [Functions for Traditional Scheduling], page 350.

`PRIO_PROCESS`
: ‘`sys/resource.h`’ (BSD): Section 14.3.4.2 [Functions for Traditional Scheduling], page 350.

PRIO_USER
: 'sys/resource.h' (BSD): Section 14.3.4.2 [Functions for Traditional Scheduling], page 350.

void psignal (int *signum*, const char *message*)
: 'signal.h' (BSD): Section 17.2.8 [Signal Messages], page 388.

int pthread_atfork (void (*prepare)(void), void (*parent)(void), void (*child)(void))
: 'pthread.h' (POSIX): Section 18.10 [Threads and Fork], page 448.

int pthread_attr_destroy (pthread_attr_t *attr)
: 'pthread.h' (POSIX): Section 18.2 [Thread Attributessection Thread Attributes], page 430.

int pthread_attr_getattr (const pthread_attr_t *obj, int *value)
: 'pthread.h' (POSIX): Section 18.2 [Thread Attributessection Thread Attributes], page 430.

int pthread_attr_init (pthread_attr_t *attr)
: 'pthread.h' (POSIX): Section 18.2 [Thread Attributessection Thread Attributes], page 430.

int pthread_attr_setattr (pthread_attr_t *obj, int value)
: 'pthread.h' (POSIX): Section 18.2 [Thread Attributessection Thread Attributes], page 430.

int pthread_cancel (pthread_t *thread*)
: 'pthread.h' (POSIX): Section 18.1 [Basic Thread Operations], page 429.

void pthread_cleanup_pop (int *execute*)
: 'pthread.h' (POSIX): Section 18.4 [Clean-Up Handlers], page 435.

void pthread_cleanup_pop_restore_np (int *execute*)
: 'pthread.h' (GNU): Section 18.4 [Clean-Up Handlers], page 435.

void pthread_cleanup_push (void (*routine) (void *), void *arg)
: 'pthread.h' (POSIX): Section 18.4 [Clean-Up Handlers], page 435.

void pthread_cleanup_push_defer_np (void (*routine) (void *), void *arg)
: 'pthread.h' (GNU): Section 18.4 [Clean-Up Handlers], page 435.

int pthread_condattr_init (pthread_condattr_t *attr)
: 'pthread.h' (POSIX): Section 18.6 [Condition Variables], page 441.

int pthread_cond_broadcast (pthread_cond_t *cond)
: 'pthread.h' (POSIX): Section 18.6 [Condition Variables], page 441.

int pthread_cond_destroy (pthread_cond_t *cond)
: 'pthread.h' (POSIX): Section 18.6 [Condition Variables], page 441.

int pthread_cond_init (pthread_cond_t *cond, pthread_condattr_t *cond_attr)
: 'pthread.h' (POSIX): Section 18.6 [Condition Variables], page 441.

int pthread_cond_signal (pthread_cond_t *cond)
: 'pthread.h' (POSIX): Section 18.6 [Condition Variables], page 441.

int pthread_cond_timedwait (pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime)
: 'pthread.h' (POSIX): Section 18.6 [Condition Variables], page 441.

`int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex)`
'`pthread.h`' (POSIX): Section 18.6 [Condition Variables], page 441.

`int pthread_create (pthread_t * thread, pthread_attr_t * attr, void * (*start_routine) (void *), void * arg)`
'`pthread.h`' (POSIX): Section 18.1 [Basic Thread Operations], page 429.

`int pthread_detach (pthread_t th)`
'`pthread.h`' (POSIX): Section 18.12 [Miscellaneous Thread Functions], page 451.

`int pthread_equal (pthread_t thread1, pthread_t thread2)`
'`pthread.h`' (POSIX): Section 18.12 [Miscellaneous Thread Functions], page 451.

`void pthread_exit (void *retval)`
'`pthread.h`' (POSIX): Section 18.1 [Basic Thread Operations], page 429.

`int pthread_getconcurrency ()`
'`pthread.h`' (POSIX): Section 18.12 [Miscellaneous Thread Functions], page 451.

`int pthread_getschedparam (pthread_t target_thread, int *policy, struct sched_param *param)`
'`pthread.h`' (POSIX): Section 18.12 [Miscellaneous Thread Functions], page 451.

`void * pthread_getspecific (pthread_key_t key)`
'`pthread.h`' (POSIX): Section 18.8 [Thread-Specific Data], page 445.

`int pthread_join (pthread_t th, void **thread_return)`
'`pthread.h`' (POSIX): Section 18.1 [Basic Thread Operations], page 429.

`int pthread_key_create (pthread_key_t *key, void (*destr_function) (void *))`
'`pthread.h`' (POSIX): Section 18.8 [Thread-Specific Data], page 445.

`int pthread_key_delete (pthread_key_t key)`
'`pthread.h`' (POSIX): Section 18.8 [Thread-Specific Data], page 445.

`int pthread_kill (pthread_t thread, int signo)`
'`pthread.h`' (POSIX): Section 18.9 [Threads and Signal-Handling], page 447.

`void pthread_kill_other_threads_np (void)`
'`pthread.h`' (GNU): Section 18.12 [Miscellaneous Thread Functions], page 451.

`int pthread_mutexattr_destroy (pthread_mutexattr_t *attr)`
'`pthread.h`' (POSIX): Section 18.5 [Mutexes], page 437.

`int pthread_mutexattr_gettype (const pthread_mutexattr_t *attr, int *type)`
'`pthread.h`' (POSIX): Section 18.5 [Mutexes], page 437.

`int pthread_mutexattr_init (pthread_mutexattr_t *attr)`
'`pthread.h`' (POSIX): Section 18.5 [Mutexes], page 437.

`int pthread_mutexattr_settype (pthread_mutexattr_t *attr, int type)`
'`pthread.h`' (POSIX): Section 18.5 [Mutexes], page 437.

`int pthread_mutex_destroy (pthread_mutex_t *mutex)`
'`pthread.h`' (POSIX): Section 18.5 [Mutexes], page 437.

`int ptsname_r (int` *filedes*`, char *`*buf*`, size_t` *len*`)`
'`stdlib.h`' (GNU): Section 6.8.1 [Allocating Pseudoterminals], page 205.

`int putpwent (const struct passwd *`*p*`, FILE *`*stream*`)`
'`pwd.h`' (SVID): Section 10.13.4 [Writing a User Entry], page 276.

`struct utmp * pututline (const struct utmp *`*utmp*`)`
'`utmp.h`' (SVID): Section 10.12.1 [Manipulating the User-Accounting Database], page 265.

`struct utmpx * pututxline (const struct utmpx *`*utmp*`)`
'`utmpx.h`' (XPG4.2): Section 10.12.2 [XPG User-Accounting Database Functions], page 270.

`ssize_t pwrite (int` *filedes*`, const void *`*buffer*`, size_t` *size*`, off_t` *offset*`)`
'`unistd.h`' (Unix98): Section 2.2 [Input and Output Primitives], page 20.

`ssize_t pwrite64 (int` *filedes*`, const void *`*buffer*`, size_t` *size*`, off64_t` *offset*`)`
'`unistd.h`' (Unix98): Section 2.2 [Input and Output Primitives], page 20.

`int raise (int` *signum*`)`
'`signal.h`' (ISO): Section 17.6.1 [Signaling Yourself], page 409.

`ssize_t read (int` *filedes*`, void *`*buffer*`, size_t` *size*`)`
'`unistd.h`' (POSIX.1): Section 2.2 [Input and Output Primitives], page 20.

`struct dirent * readdir (DIR *`*dirstream*`)`
'`dirent.h`' (POSIX.1): Section 3.2.3 [Reading and Closing a Directory Stream], page 76.

`struct dirent64 * readdir64 (DIR *`*dirstream*`)`
'`dirent.h`' (LFS): Section 3.2.3 [Reading and Closing a Directory Stream], page 76.

`int readdir64_r (DIR *`*dirstream*`, struct dirent64 *`*entry*`, struct dirent64 **`*result*`)`
'`dirent.h`' (LFS): Section 3.2.3 [Reading and Closing a Directory Stream], page 76.

`int readdir_r (DIR *`*dirstream*`, struct dirent *`*entry*`, struct dirent **`*result*`)`
'`dirent.h`' (GNU): Section 3.2.3 [Reading and Closing a Directory Stream], page 76.

`int readlink (const char *`*filename*`, char *`*buffer*`, size_t` *size*`)`
'`unistd.h`' (BSD): Section 3.5 [Symbolic Links], page 87.

`ssize_t readv (int` *filedes*`, const struct iovec *`*vector*`, int` *count*`)`
'`sys/uio.h`' (BSD): Section 2.6 [Fast Scatter-Gather I/O], page 31.

`char * realpath (const char *restrict` *name*`, char *restrict` *resolved*`)`
'`stdlib.h`' (XPG): Section 3.5 [Symbolic Links], page 87.

`int recv (int` *socket*`, void *`*buffer*`, size_t` *size*`, int` *flags*`)`
'`sys/socket.h`' (BSD): Section 5.9.5.2 [Receiving Data], page 158.

`int recvfrom (int` *socket*`, void *`*buffer*`, size_t` *size*`, int` *flags*`, struct sockaddr *`*addr*`, socklen_t *`*length-ptr*`)`
'`sys/socket.h`' (BSD): Section 5.10.2 [Receiving Datagrams], page 168.

`int recvmsg (int` *socket*`, struct msghdr *`*message*`, int` *flags*`)`
'`sys/socket.h`' (BSD): Section 5.10.2 [Receiving Datagrams], page 168.

`int scandir (const char *dir, struct dirent ***namelist, int (*selector) (const struct dirent *), int (*cmp) (const void *, const void *))`
'`dirent.h`' (BSD/SVID): Section 3.2.6 [Scanning the Content of a Directory], page 79.

`int scandir64 (const char *dir, struct dirent64 ***namelist, int (*selector) (const struct dirent64 *), int (*cmp) (const void *, const void *))`
'`dirent.h`' (GNU): Section 3.2.6 [Scanning the Content of a Directory], page 79.

`_SC_ARG_MAX`
'`unistd.h`' (POSIX.1): Section 12.4.2 [Constants for `sysconf` Parameters], page 307.

`_SC_ASYNCHRONOUS_IO`
'`unistd.h`' (POSIX.1): Section 12.4.2 [Constants for `sysconf` Parameters], page 307.

`_SC_ATEXIT_MAX`
'`unistd.h`' (GNU): Section 12.4.2 [Constants for `sysconf` Parameters], page 307.

`_SC_AVPHYS_PAGES`
'`unistd.h`' (GNU): Section 12.4.2 [Constants for `sysconf` Parameters], page 307.

`_SC_BC_BASE_MAX`
'`unistd.h`' (POSIX.2): Section 12.4.2 [Constants for `sysconf` Parameters], page 307.

`_SC_BC_DIM_MAX`
'`unistd.h`' (POSIX.2): Section 12.4.2 [Constants for `sysconf` Parameters], page 307.

`_SC_BC_SCALE_MAX`
'`unistd.h`' (POSIX.2): Section 12.4.2 [Constants for `sysconf` Parameters], page 307.

`_SC_BC_STRING_MAX`
'`unistd.h`' (POSIX.2): Section 12.4.2 [Constants for `sysconf` Parameters], page 307.

`_SC_CHAR_BIT`
'`unistd.h`' (X/Open): Section 12.4.2 [Constants for `sysconf` Parameters], page 307.

`_SC_CHARCLASS_NAME_MAX`
'`unistd.h`' (GNU): Section 12.4.2 [Constants for `sysconf` Parameters], page 307.

`_SC_CHAR_MAX`
'`unistd.h`' (X/Open): Section 12.4.2 [Constants for `sysconf` Parameters], page 307.

`_SC_CHAR_MIN`
'`unistd.h`' (X/Open): Section 12.4.2 [Constants for `sysconf` Parameters], page 307.

`_SC_CHILD_MAX`
'`unistd.h`' (POSIX.1): Section 12.4.2 [Constants for `sysconf` Parameters], page 307.

`_SC_CLK_TCK`
: 'unistd.h' (POSIX.1): Section 12.4.2 [Constants for sysconf Parameters], page 307.

`_SC_COLL_WEIGHTS_MAX`
: 'unistd.h' (POSIX.2): Section 12.4.2 [Constants for sysconf Parameters], page 307.

`_SC_DELAYTIMER_MAX`
: 'unistd.h' (POSIX.1): Section 12.4.2 [Constants for sysconf Parameters], page 307.

`_SC_EQUIV_CLASS_MAX`
: 'unistd.h' (POSIX.2): Section 12.4.2 [Constants for sysconf Parameters], page 307.

`_SC_EXPR_NEST_MAX`
: 'unistd.h' (POSIX.2): Section 12.4.2 [Constants for sysconf Parameters], page 307.

`_SC_FSYNC`
: 'unistd.h' (POSIX.1): Section 12.4.2 [Constants for sysconf Parameters], page 307.

`_SC_GETGR_R_SIZE_MAX`
: 'unistd.h' (POSIX.1): Section 12.4.2 [Constants for sysconf Parameters], page 307.

`_SC_GETPW_R_SIZE_MAX`
: 'unistd.h' (POSIX.1): Section 12.4.2 [Constants for sysconf Parameters], page 307.

`SCHAR_MAX`
: 'limits.h' (ISO): Section A.5.2 [Range of an Integer Type], page 465.

`SCHAR_MIN`
: 'limits.h' (ISO): Section A.5.2 [Range of an Integer Type], page 465.

`int sched_getaffinity (pid_t` *pid*`, cpu_set_t *`*cpuset*`)`
: 'sched.h' (GNU): Section 14.3.5 [Limiting Execution to Certain CPUs], page 352.

`int sched_getparam (pid_t` *pid*`, const struct sched_param *`*param*`)`
: 'sched.h' (POSIX): Section 14.3.3 [Basic Scheduling Functions], page 346.

`int sched_get_priority_max (int *`*policy*`);`
: 'sched.h' (POSIX): Section 14.3.3 [Basic Scheduling Functions], page 346.

`int sched_get_priority_min (int *`*policy*`);`
: 'sched.h' (POSIX): Section 14.3.3 [Basic Scheduling Functions], page 346.

`int sched_getscheduler (pid_t` *pid*`)`
: 'sched.h' (POSIX): Section 14.3.3 [Basic Scheduling Functions], page 346.

`int sched_rr_get_interval (pid_t` *pid*`, struct timespec *`*interval*`)`
: 'sched.h' (POSIX): Section 14.3.3 [Basic Scheduling Functions], page 346.

`int sched_setaffinity (pid_t` *pid*`, const cpu_set_t *`*cpuset*`)`
: 'sched.h' (GNU): Section 14.3.5 [Limiting Execution to Certain CPUs], page 352.

_SC_PII
: 'unistd.h' (POSIX.1g): Section 12.4.2 [Constants for sysconf Parameters], page 307.

_SC_PII_INTERNET
: 'unistd.h' (POSIX.1g): Section 12.4.2 [Constants for sysconf Parameters], page 307.

_SC_PII_INTERNET_DGRAM
: 'unistd.h' (POSIX.1g): Section 12.4.2 [Constants for sysconf Parameters], page 307.

_SC_PII_INTERNET_STREAM
: 'unistd.h' (POSIX.1g): Section 12.4.2 [Constants for sysconf Parameters], page 307.

_SC_PII_OSI
: 'unistd.h' (POSIX.1g): Section 12.4.2 [Constants for sysconf Parameters], page 307.

_SC_PII_OSI_CLTS
: 'unistd.h' (POSIX.1g): Section 12.4.2 [Constants for sysconf Parameters], page 307.

_SC_PII_OSI_COTS
: 'unistd.h' (POSIX.1g): Section 12.4.2 [Constants for sysconf Parameters], page 307.

_SC_PII_OSI_M
: 'unistd.h' (POSIX.1g): Section 12.4.2 [Constants for sysconf Parameters], page 307.

_SC_PII_SOCKET
: 'unistd.h' (POSIX.1g): Section 12.4.2 [Constants for sysconf Parameters], page 307.

_SC_PII_XTI
: 'unistd.h' (POSIX.1g): Section 12.4.2 [Constants for sysconf Parameters], page 307.

_SC_PRIORITIZED_IO
: 'unistd.h' (POSIX.1): Section 12.4.2 [Constants for sysconf Parameters], page 307.

_SC_PRIORITY_SCHEDULING
: 'unistd.h' (POSIX.1): Section 12.4.2 [Constants for sysconf Parameters], page 307.

_SC_REALTIME_SIGNALS
: 'unistdh.h' (POSIX.1): Section 12.4.2 [Constants for sysconf Parameters], page 307.

_SC_RTSIG_MAX
: 'unistd.h' (POSIX.1): Section 12.4.2 [Constants for sysconf Parameters], page 307.

_SC_SAVED_IDS
: 'unistd.h' (POSIX.1): Section 12.4.2 [Constants for sysconf Parameters], page 307.

`int setegid (gid_t` *newgid*`)`
'`unistd.h`' (POSIX.1): Section 10.7 [Setting the Group IDs], page 257.

`int seteuid (uid_t` *neweuid*`)`
'`unistd.h`' (POSIX.1): Section 10.6 [Setting the User ID], page 256.

`int setfsent (void)`
'`fstab.h`' (BSD): Section 11.3.1.1 [The '`fstab`' File], page 290.

`int setgid (gid_t` *newgid*`)`
'`unistd.h`' (POSIX.1): Section 10.7 [Setting the Group IDs], page 257.

`void setgrent (void)`
'`grp.h`' (SVID, BSD): Section 10.14.3 [Scanning the List of All Groups], page 278.

`int setgroups (size_t` *count*`, gid_t *`*groups*`)`
'`grp.h`' (BSD): Section 10.7 [Setting the Group IDs], page 257.

`void sethostent (int` *stayopen*`)`
'`netdb.h`' (BSD): Section 5.6.2.4 [Host Names], page 141.

`int sethostid (long int` *id*`)`
'`unistd.h`' (BSD): Section 11.1 [Host Identification], page 285.

`int sethostname (const char *`*name*`, size_t` *length*`)`
'`unistd.h`' (BSD): Section 11.1 [Host Identification], page 285.

`int setjmp (jmp_buf` *state*`)`
'`setjmp.h`' (ISO): Section 16.2 [Details of Nonlocal Exits], page 369.

`void setkey (const char *`*key*`)`
'`crypt.h`' (BSD, SVID): Section 13.4 [DES Encryption], page 331.

`void setkey_r (const char *`*key*`, struct crypt_data *` *data*`)`
'`crypt.h`' (GNU): Section 13.4 [DES Encryption], page 331.

`int setlogmask (int` *mask*`)`
'`syslog.h`' (BSD): Section 15.2.4 [setlogmask], page 365.

`FILE * setmntent (const char *`*file*`, const char *`*mode*`)`
'`mntent.h`' (BSD): Section 11.3.1.2 [The '`mtab`' File], page 292.

`void setnetent (int` *stayopen*`)`
'`netdb.h`' (BSD): Section 5.13 [Networks Database], page 176.

`int setnetgrent (const char *`*netgroup*`)`
'`netdb.h`' (BSD): Section 10.16.2 [Looking Up One Netgroup], page 282.

`int setpgid (pid_t` *pid*`, pid_t` *pgid*`)`
'`unistd.h`' (POSIX.1): Section 8.7.2 [Process-Group Functions], page 239.

`int setpgrp (pid_t` *pid*`, pid_t` *pgid*`)`
'`unistd.h`' (BSD): Section 8.7.2 [Process-Group Functions], page 239.

`int setpriority (int` *class*`, int` *id*`, int` *niceval*`)`
'`sys/resource.h`' (BSD,POSIX): Section 14.3.4.2 [Functions for Traditional Scheduling], page 350.

`void setprotoent (int` *stayopen*`)`
'`netdb.h`' (BSD): Section 5.6.6 [Protocols Database], page 147.

`int S_ISBLK (mode_t m)`
: ‘`sys/stat.h`’ (POSIX): Section 3.9.3 [Testing the Type of a File], page 99.

`int S_ISCHR (mode_t m)`
: ‘`sys/stat.h`’ (POSIX): Section 3.9.3 [Testing the Type of a File], page 99.

`int S_ISDIR (mode_t m)`
: ‘`sys/stat.h`’ (POSIX): Section 3.9.3 [Testing the Type of a File], page 99.

`int S_ISFIFO (mode_t m)`
: ‘`sys/stat.h`’ (POSIX): Section 3.9.3 [Testing the Type of a File], page 99.

`S_ISGID`
: ‘`sys/stat.h`’ (POSIX): Section 3.9.5 [The Mode Bits for Access Permission], page 102.

`int S_ISLNK (mode_t m)`
: ‘`sys/stat.h`’ (GNU): Section 3.9.3 [Testing the Type of a File], page 99.

`int S_ISREG (mode_t m)`
: ‘`sys/stat.h`’ (POSIX): Section 3.9.3 [Testing the Type of a File], page 99.

`int S_ISSOCK (mode_t m)`
: ‘`sys/stat.h`’ (GNU): Section 3.9.3 [Testing the Type of a File], page 99.

`S_ISUID`
: ‘`sys/stat.h`’ (POSIX): Section 3.9.5 [The Mode Bits for Access Permission], page 102.

`S_ISVTX`
: ‘`sys/stat.h`’ (BSD): Section 3.9.5 [The Mode Bits for Access Permission], page 102.

`S_IWGRP`
: ‘`sys/stat.h`’ (POSIX.1): Section 3.9.5 [The Mode Bits for Access Permission], page 102.

`S_IWOTH`
: ‘`sys/stat.h`’ (POSIX.1): Section 3.9.5 [The Mode Bits for Access Permission], page 102.

`S_IWRITE`
: ‘`sys/stat.h`’ (BSD): Section 3.9.5 [The Mode Bits for Access Permission], page 102.

`S_IWUSR`
: ‘`sys/stat.h`’ (POSIX.1): Section 3.9.5 [The Mode Bits for Access Permission], page 102.

`S_IXGRP`
: ‘`sys/stat.h`’ (POSIX.1): Section 3.9.5 [The Mode Bits for Access Permission], page 102.

`S_IXOTH`
: ‘`sys/stat.h`’ (POSIX.1): Section 3.9.5 [The Mode Bits for Access Permission], page 102.

`struct sgttyb`
: ‘`termios.h`’ (BSD): Section 6.5 [BSD Terminal Modes], page 200.

`struct sigaction`
: ‘`signal.h`’ (POSIX.1): Section 17.3.2 [Advanced Signal-Handling], page 392.

`struct sigstack`
: ‘`signal.h`’ (BSD): Section 17.9 [Using a Separate Signal-Stack], page 424.

`struct sigvec`
: ‘`signal.h`’ (BSD): Section 17.10.1 [BSD Function to Establish a Handler], page 426.

`struct sockaddr`
: ‘`sys/socket.h`’ (BSD): Section 5.3.1 [Address Formats], page 128.

`struct sockaddr_in`
: ‘`netinet/in.h`’ (BSD): Section 5.6.1 [Internet Socket Address Formats], page 135.

`struct sockaddr_un`
: ‘`sys/un.h`’ (BSD): Section 5.5.2 [Details of Local Namespace], page 132.

`struct stat`
: ‘`sys/stat.h`’ (POSIX.1): Section 3.9.1 [The Meaning of the File Attributes], page 93.

`struct stat64`
: ‘`sys/stat.h`’ (LFS): Section 3.9.1 [The Meaning of the File Attributes], page 93.

`struct termios`
: ‘`termios.h`’ (POSIX.1): Section 6.4.1 [Terminal Mode Data Types], page 181.

`struct utimbuf`
: ‘`time.h`’ (POSIX.1): Section 3.9.9 [File Times], page 108.

`struct utsname`
: ‘`sys/utsname.h`’ (POSIX.1): Section 11.2 [Platform-Type Identification], page 287.

`int stty (int` *filedes*`, struct sgttyb * attributes)`
: ‘`sgtty.h`’ (BSD): Section 6.5 [BSD Terminal Modes], page 200.

`int S_TYPEISMQ (struct stat *`*s*`)`
: ‘`sys/stat.h`’ (POSIX): Section 3.9.3 [Testing the Type of a File], page 99.

`int S_TYPEISSEM (struct stat *`*s*`)`
: ‘`sys/stat.h`’ (POSIX): Section 3.9.3 [Testing the Type of a File], page 99.

`int S_TYPEISSHM (struct stat *`*s*`)`
: ‘`sys/stat.h`’ (POSIX): Section 3.9.3 [Testing the Type of a File], page 99.

`int SUN_LEN (`*struct sockaddr_un * ptr*`)`
: ‘`sys/un.h`’ (BSD): Section 5.5.2 [Details of Local Namespace], page 132.

`_SVID_SOURCE`
: (GNU): Section 1.3.4 [Feature-Test Macros], page 8.

`int SV_INTERRUPT`
: ‘`signal.h`’ (BSD): Section 17.10.1 [BSD Function to Establish a Handler], page 426.

`TCSADRAIN`
: ‘`termios.h`’ (POSIX.1): Section 6.4.2 [Terminal Mode Functions], page 182.

`TCSAFLUSH`
: ‘`termios.h`’ (POSIX.1): Section 6.4.2 [Terminal Mode Functions], page 182.

`TCSANOW`
: ‘`termios.h`’ (POSIX.1): Section 6.4.2 [Terminal Mode Functions], page 182.

`TCSASOFT`
: ‘`termios.h`’ (BSD): Section 6.4.2 [Terminal Mode Functions], page 182.

`int tcsendbreak (int` *filedes*`, int` *duration*`)`
: ‘`termios.h`’ (POSIX.1): Section 6.6 [Line Control Functions], page 201.

`int tcsetattr (int` *filedes*`, int` *when*`, const struct termios *`*termios-p*`)`
: ‘`termios.h`’ (POSIX.1): Section 6.4.2 [Terminal Mode Functions], page 182.

`int tcsetpgrp (int` *filedes*`, pid_t` *pgid*`)`
: ‘`unistd.h`’ (POSIX.1): Section 8.7.3 [Functions for Controlling-Terminal Access], page 241.

`off_t telldir (DIR *`*dirstream*`)`
: ‘`dirent.h`’ (BSD): Section 3.2.5 [Random Access in a Directory Stream], page 78.

`TEMP_FAILURE_RETRY (`*expression*`)`
: ‘`unistd.h`’ (GNU): Section 17.5 [Primitives Interrupted by Signals], page 408.

`char * tempnam (const char *`*dir*`, const char *`*prefix*`)`
: ‘`stdio.h`’ (SVID): Section 3.11 [Temporary Files], page 114.

`FILE * tmpfile (void)`
: ‘`stdio.h`’ (ISO): Section 3.11 [Temporary Files], page 114.

`FILE * tmpfile64 (void)`
: ‘`stdio.h`’ (Unix98): Section 3.11 [Temporary Files], page 114.

`int TMP_MAX`
: ‘`stdio.h`’ (ISO): Section 3.11 [Temporary Files], page 114.

`char * tmpnam (char *`*result*`)`
: ‘`stdio.h`’ (ISO): Section 3.11 [Temporary Files], page 114.

`char * tmpnam_r (char *`*result*`)`
: ‘`stdio.h`’ (GNU): Section 3.11 [Temporary Files], page 114.

`tcflag_t TOSTOP`
: ‘`termios.h`’ (POSIX.1): Section 6.4.7 [Local Modes], page 189.

`int truncate (const char *`*filename*`, off_t` *length*`)`
: ‘`unistd.h`’ (X/Open): Section 3.9.10 [File Size], page 110.

`int truncate64 (const char *`*name*`, off64_t` *length*`)`
: ‘`unistd.h`’ (Unix98): Section 3.9.10 [File Size], page 110.

`TRY_AGAIN`
: ‘`netdb.h`’ (BSD): Section 5.6.2.4 [Host Names], page 141.

`char * ttyname (int` *filedes*`)`
: ‘`unistd.h`’ (POSIX.1): Section 6.1 [Identifying Terminals], page 179.

Appendix C Installing the GNU C Library

Before you do anything else, you should read the file 'FAQ' located at the top level of the source tree. This file answers common questions and describes problems you may experience with compilation and installation. It is updated more frequently than this manual.

Features can be added to GNU libc via *add-on* bundles. These are separate tar files, which you unpack into the top level of the source tree. Then you give `configure` the '`--enable-add-ons`' option to activate them, and they will be compiled into the library. As of the 2.2 release, one important component of glibc is distributed as "official" add-ons—the LinuxThreads add-on. Unless you are doing an unusual installation, you should get this.

Support for POSIX threads is maintained by someone else, so it's in a separate package. It is only available for GNU/Linux systems, but this will change in the future. Get it from the same place you got the main bundle; the file is '`glibc-linuxthreads-`*VERSION*`.tar.gz`'.

You will need recent versions of several GNU tools—definitely GCC and GNU Make, and possibly others (see Section C.3 [Recommended Tools for Compilation], page 538).

C.1 Configuring and Compiling GNU libc

GNU libc can be compiled in the source directory, but we strongly advise building it in a separate build directory. For example, if you have unpacked the glibc sources in '`/src/gnu/glibc-2.3`', create a directory '`/src/gnu/glibc-build`' to put the object files in. This allows removing the whole build directory in case an error occurs, which is the safest way to get a fresh start and should always be done.

>From your object directory, run the shell script '`configure`' located at the top level of the source tree. In the scenario above, you'd type:

```
$ ../glibc-2.3/configure args...
```

Please note that even if you're building in a separate build directory, the compilation needs to modify a few files in the source directory, especially some files in the manual subdirectory.

`configure` takes many options, but you can get away with knowing only two: '`--prefix`' and '`--enable-add-ons`'. The `--prefix` option tells `configure` where you want glibc installed. This defaults to '`/usr/local`'. The '`--enable-add-ons`' option tells `configure` to use all the add-on bundles it finds in the source directory. Since important functionality is provided in add-ons, you should always specify this option.

It may also be useful to set the *CC* and *CFLAGS* variables in the environment when running `configure`. *CC* selects the C compiler that will be used, and *CFLAGS* sets optimization options for the compiler.

The following list describes all of the available options for `configure`:

‘`--prefix=`*directory*’
: Install machine-independent data files in subdirectories of ‘*directory*’. The default is to install in ‘`/usr/local`’.

‘`--exec-prefix=`*directory*’
: Install the library and other machine-dependent files in subdirectories of ‘*directory*’. The default is to the ‘`--prefix`’ directory if that option is specified, or ‘`/usr/local`’ otherwise.

‘`--with-headers=`*directory*’
: Look for kernel header files in *directory*, not ‘`/usr/include`’. Glibc needs information from the kernel’s private header files. Glibc will normally look in ‘`/usr/include`’ for them, but if you specify this option, it will look in *DIRECTORY* instead.

  This option is primarily of use on a system where the headers in ‘`/usr/include`’ come from an older version of glibc. Conflicts can occasionally happen in this case. Linux libc5 qualifies as an older version of glibc. You can also use this option if you want to compile glibc with a newer set of kernel headers than the ones found in ‘`/usr/include`’.

‘`--enable-add-ons[=`*list*`]`’
: Enable add-on packages in your source tree. If this option is specified with no list, it enables all the add-on packages it finds. If you do not wish to use some add-on packages that you have present in your source tree, give this option a list of the add-ons that you *do* want used, like this: ‘`--enable-add-ons=linuxthreads`’

‘`--enable-kernel=`*version*’
: This option is currently only useful on GNU/Linux systems. The *version* parameter should have the form X.Y.Z and describes the smallest version of the Linux kernel the generated library is expected to support. The higher the *version* number is, the less compatibility code is added, and the faster the code gets.

‘`--with-binutils=`*directory*’
: Use the binutils (assembler and linker) in ‘*directory*’, not the ones the C compiler would default to. You can use this option if the default binutils on your system cannot deal with all the constructs in the GNU C Library. In that case, `configure` will detect the problem and suppress these constructs, so that the library will still be usable, but functionality may be lost—for example, you can’t build a shared libc with old binutils.

‘`--without-fp`’
: Use this option if your computer lacks hardware floating-point support and your operating system does not emulate an FPU.

‘`--disable-shared`’
: Don’t build shared libraries even if it is possible. Not all systems support shared libraries; you need ELF support and (currently) the GNU linker.

‘`--disable-profile`’
: Don’t build libraries with profiling information. You may want to use this option if you don’t plan to do profiling.

‘`--enable-omitfp`’
: Use maximum optimization for the normal (static and shared) libraries, and compile separate static libraries with debugging information and no optimization. We recommend not doing this. The extra optimization doesn’t gain you much, it may provoke compiler bugs, and you won’t be able to trace bugs through the C library.

‘`--disable-versioning`’
: Don’t compile the shared libraries with symbol version information. Doing this will make the resulting library incompatible with old binaries, so it’s not recommended.

‘`--enable-static-nss`’
: Compile static versions of the NSS (Name-Service Switch) libraries. This is not recommended because it defeats the purpose of NSS; a program linked statically with the NSS libraries cannot be dynamically reconfigured to use a different name database.

‘`--without-tls`’
: By default, the C library is built with support for thread-local storage if the tools used support it. By using ‘`--without-tls`’ this can be prevented, though there generally is no reason, since using this option creates compatibility problems.

‘`--build=`*build-system*’
‘`--host=`*host-system*’
: These options are for cross-compiling. If you specify both options and *build-system* is different from *host-system*, `configure` will prepare to cross-compile glibc from *build-system* to be used on *host-system*. You’ll probably need the ‘`--with-headers`’ option too, and you may have to override `configure`’s selection of the compiler and/or binutils.

  If you only specify ‘`--host`’, `configure` will prepare for a native compile but use what you specify instead of guessing what your system is. This is most useful to change the CPU submodel. For example, if `configure` guesses your machine as `i586-pc-linux-gnu` but you want to compile a library for 386es, give ‘`--host=i386-pc-linux-gnu`’ or just ‘`--host=i386-linux`’ and add the appropriate compiler flags (‘`-mcpu=i386`’ will do the trick) to *CFLAGS*.

If you specify just '`--build`', `configure` will get confused.

To build the library and related programs, type `make`. This will produce a lot of output, some of which may look like errors from `make` but isn't. Look for error messages from `make` containing '`***`'. Those indicate that something is seriously wrong.

The compilation process can take several hours. Expect at least two hours for the default configuration on i586 for GNU/Linux. For Hurd, times are much longer. Some complex modules may take a very long time to compile, as much as several minutes on slower machines. Do not panic if the compiler appears to hang.

If you want to run a parallel make, simply pass the '`-j`' option with an appropriate numeric parameter to `make`. You need a recent GNU `make` version, though.

To build and run test programs that exercise some of the library facilities, type `make check`. If it does not complete successfully, do not use the built library, and report a bug after verifying that the problem is not already known (see Section C.6 [Reporting Bugs], page 541). Some of the tests assume they are not being run by `root`. We recommend you compile and test glibc as an unprivileged user.

Before reporting bugs, make sure there is no problem with your system. The tests (and later installation) use some preexisting files of the system such as '`/etc/passwd`', '`/etc/nsswitch.conf`' and others. These files must all contain correct and sensible content.

To format the GNU C Library reference manuals for printing, type `make dvi`. You need a working TEX installation to do this. The distribution already includes the on-line formatted version of the manual, as Info files. You can regenerate those with `make info`, but it shouldn't be necessary.

The library has a number of special-purpose configuration parameters that you can find in '`Makeconfig`'. These can be overwritten with the file '`configparms`'. To change them, create a '`configparms`' in your build directory and add values as appropriate for your system. The file is included and parsed by `make` and has to follow the conventions for makefiles.

It is easy to configure the GNU C Library for cross-compilation by setting a few variables in '`configparms`'. Set `CC` to the cross-compiler for the target you configured the library for; it is important to use this same `CC` value when running `configure`, like this: '`CC=`*target*`-gcc configure` *target*'. Set `BUILD_CC` to the compiler to use for programs run on the build system as part of compiling the library. You may need to set `AR` and `RANLIB` to cross-compiling versions of `ar` and `ranlib` if the native tools are not configured to work with object files for the target you configured for.

C.2 Installing the C Library

To install the library and its header files, and the Info files of the manual, type `env LANGUAGE=C LC_ALL=C make install`. This will build things, if necessary, before installing them; however, you should still compile everything first. If you are installing glibc as your primary C library, we recommend that you shut the

system down to single-user mode first, and reboot afterward. This minimizes the risk of breaking things when the library changes out from underneath.

If you're upgrading from Linux libc5 or some other C library, you need to replace the '`/usr/include`' with a fresh directory before installing it. The new '`/usr/include`' should contain the Linux headers, but nothing else.

You must first build the library ('`make`'), optionally check it ('`make check`'), switch the include directories, and then install ('`make install`'). The steps must be done in this order. Not moving the directory before install will result in an unusable mixture of header files from both libraries, but configuring, building, and checking the library requires the ability to compile and run programs against the old library.

If you are upgrading from a previous installation of glibc 2.0 or 2.1, '`make install`' will do the entire job. You do not need to remove the old includes—if you want to do so anyway, you must then follow the order given above.

You may also need to reconfigure GCC to work with the new library. The easiest way to do that is to figure out the compiler switches to make it work again ('`-Wl,--dynamic-linker=/lib/ld-linux.so.2`' should work on GNU/Linux systems) and use them to recompile gcc. You can also edit the specs file ('`/usr/lib/gcc-lib/`*TARGET*`/`*VERSION*`/specs`'), but that is a bit of a black art.

You can install glibc somewhere other than where you configured it to go by setting the `install_root` variable on the command line for '`make install`'. The value of this variable is prepended to all the paths for installation. This is useful when setting up a chroot environment or preparing a binary distribution. The directory should be specified with an absolute file-name.

Glibc 2.2 includes a daemon called `nscd`, which you may or may not want to run. `nscd` caches name-service lookups; it can dramatically improve performance with NIS+, and may help with DNS as well.

One auxiliary program, '`/usr/libexec/pt_chown`', is installed setuid `root`. This program is invoked by the `grantpt` function; it sets the permissions on a pseudoterminal so it can be used by the calling process. This means programs like `xterm` and `screen` do not have to be setuid to get a pty. (There may be other reasons why they need privileges.) If you are using a 2.1 or newer Linux kernel with the `devptsfs` or `devfs` file-systems providing pty slaves, you don't need this program; otherwise you do. The source for '`pt_chown`' is in '`login/programs/pt_chown.c`'.

After installation, you might want to configure the time zone and locale installation of your system. The GNU C Library comes with a locale database that gets configured with `localedef`. For example, to set up a German locale with name `de_DE`, simply issue the command '`localedef -i de_DE -f ISO-8859-1 de_DE`'. To configure all locales that are supported by glibc, you can issue from your build directory the command '`make localedata/install-locales`'.

To configure the locally used timezone, set the `TZ` environment variable. The script `tzselect` helps you to select the right value. As an example, for Germany,

`tzselect` would tell you to use '`TZ='Europe/Berlin''`'. For a system-wide installation (the given paths are for an installation with '`--prefix=/usr`'), link the time zone file that is in '`/usr/share/zoneinfo`' to the file '`/etc/localtime`'. For Germany, you might execute '`ln -s /usr/share/zoneinfo/Europe/Berlin /etc/localtime`'.

C.3 Recommended Tools for Compilation

We recommend installing the following GNU tools before attempting to build the GNU C Library:

- GNU `make` 3.79 or newer

  You need the latest version of GNU `make`. Modifying the GNU C Library to work with other `make` programs would be so difficult that we recommend you port GNU `make` instead. We recommend GNU `make` version 3.79. All earlier versions have severe bugs or lack features.

- GCC 3.2 or newer

  The GNU C Library can only be compiled with the GNU C Compiler family. As of the 2.3 release, GCC 3.2 or higher is required. As of this writing, GCC 3.2 is the compiler we advise to use.

  You can use whatever compiler you like to compile programs that use GNU libc, but be aware that both GCC 2.7 and 2.8 have bugs in their floating-point support that may be triggered by the math library.

  Check the FAQ for any special compiler issues on particular platforms.

- GNU `binutils` 2.13 or later

  You must use GNU `binutils` (as and ld) to build the GNU C Library. No other assembler and linker has the necessary functionality at the moment.

- GNU `texinfo` 3.12f

  To correctly translate and install the Texinfo documentation, you need this version of the `texinfo` package. Earlier versions do not understand all the tags used in the document, and the installation mechanism for the info files is not present or works differently.

- GNU `awk` 3.0, or some other POSIX awk

  `Awk` is used in several places to generate files. The scripts should work with any POSIX-compliant `awk` implementation; `gawk` 3.0 and `mawk` 1.3 are known to work.

- Perl 5

  Perl is not required, but it is used if present to test the installation. We may decide to use it elsewhere in the future.

- GNU `sed` 3.02 or newer

  `Sed` is used in several places to generate files. Most scripts work with any version of `sed`. The known exception is the script `po2test.sed` in the `intl` subdirectory, which is used to generate `msgs.h` for the test suite. This

script works correctly only with GNU `sed` 3.02. If you like to run the test suite, you should definitely upgrade `sed`.

- If you change any of the '`configure.in`' files, you will also need GNU `autoconf` 2.12 or higher.
- If you change any of the message translation files, you will need GNU `gettext` 0.10.36 or later.

You may also need these packages if you upgrade your source tree using patches, although we try to avoid this.

C.4 Supported Configurations

The GNU C Library currently supports configurations that match the following patterns:

```
alpha*-*-linux
arm-*-linux
cris-*-linux
hppa-*-linux
ix86-*-gnu
ix86-*-linux
ia64-*-linux
m68k-*-linux
mips*-*-linux
powerpc-*-linux
s390-*-linux
s390x-*-linux
sparc-*-linux
sparc64-*-linux
```

Former releases of this library (version 2.1 and/or 2.0) used to run on the following configurations:

```
arm-*-linuxaout
arm-*-none
```

Very early releases (version 1.09.1 and perhaps earlier versions) used to run on the following configurations:

```
alpha-dec-osf1
alpha-*-linuxecoff
ix86-*-bsd4.3
ix86-*-isc2.2
ix86-*-isc3.n
ix86-*-sco3.2
ix86-*-sco3.2v4
ix86-*-sysv
ix86-*-sysv4
ix86-force_cpu386-none
```

```
ix86-sequent-bsd
i960-nindy960-none
m68k-hp-bsd4.3
m68k-mvme135-none
m68k-mvme136-none
m68k-sony-newsos3
m68k-sony-newsos4
m68k-sun-sunos4.n
mips-dec-ultrix4.n
mips-sgi-irix4.n
sparc-sun-solaris2.n
sparc-sun-sunos4.n
```

Since no one has volunteered to test and fix these configurations, they are not supported at the moment. They probably don't compile; they definitely don't work anymore. Porting the library is not hard. If you are interested in doing a port, please contact the glibc maintainers by sending electronic mail to bug-glibc@gnu.org.

Valid cases of '`ix86`' include '`i386`', '`i486`', '`i586`' and '`i686`'. All of those configurations produce a library that can run on this processor and newer processors. The GCC compiler by default generates code that's optimized for the machine it's configured for and will use the instructions available on that machine. For example, if your GCC is configured for '`i686`', gcc will optimize for '`i686`' and might issue some '`i686`' specific instructions. To generate code for other models, you have to configure for that model and give GCC the appropriate '`-march=`' and '`-mcpu=`' compiler switches via *CFLAGS*.

C.5 Specific Advice for GNU/Linux Systems

If you are installing GNU libc on a GNU/Linux system, you need to have the header files from a 2.2 or newer kernel around for reference. For some architectures, like ia64, sh and hppa, you need at least headers from kernel 2.3.99 (sh and hppa) or 2.4.0 (ia64). You do not need to use that kernel, just have its headers where glibc can access them. The easiest way to do this is to unpack it in a directory such as '`/usr/src/linux-2.2.1`'. In that directory, run '`make config`' and accept all the defaults. Then run '`make include/linux/version.h`'. Finally, configure glibc with the option '`--with-headers=/usr/src/linux-2.2.1/include`'. Use the most recent kernel you can get your hands on.

An alternate tactic is to unpack the 2.2 kernel and run '`make config`' as above; then, rename or delete '`/usr/include`', create a new '`/usr/include`', and make symbolic links of '`/usr/include/linux`' and '`/usr/include/asm`' into the kernel sources. You can then configure glibc with no special options. This tactic is recommended if you are upgrading from libc5, since you need to get rid of the old header files anyway.

After installing GNU libc, you may need to remove or rename '`/usr/include/linux`' and '`/usr/include/asm`', and replace them with copies of '`include/linux`' and '`include/asm-$`*ARCHITECTURE*' taken from the Linux source package that supplied kernel headers for building the library. *ARCHITECTURE* will be the machine architecture for which the library was built, such as '`i386`' or '`alpha`'. You do not need to do this if you did not specify an alternate kernel header source using '`--with-headers`'. The intent here is that these directories should be copies of, **not** symlinks to, the kernel headers used to build the library.

'`/usr/include/net`' and '`/usr/include/scsi`' should **not** be symlinks into the kernel sources. GNU libc provides its own versions of these files.

GNU/Linux expects some components of the libc installation to be in '`/lib`' and some in '`/usr/lib`'. This is handled automatically if you configure glibc with '`--prefix=/usr`'. If you set some other prefix or allow it to default to '`/usr/local`', then all the components are installed there.

If you are upgrading from libc5, you need to recompile every shared library on your system against the new library for the sake of new code, but keep the old libraries around for old binaries to use. This is complicated and difficult. Consult the Glibc2 HOWTO at `http://www.imaxx.net/~thrytis/glibc` for details.[1]

You cannot use `nscd` with 2.0 kernels, due to bugs in the kernel-side thread support. `nscd` happens to hit these bugs particularly hard, but you might have problems with any threaded program.

C.6 Reporting Bugs

There are probably bugs in the GNU C Library. There are certainly errors and omissions in this manual. If you report them, they will get fixed. If you don't, no one will ever know about them and they will remain unfixed for all eternity, if not longer.

It is a good idea to verify that the problem has not already been reported. Bugs are documented in two places: The file '`BUGS`' describes a number of well known bugs and the bug tracking system has a WWW interface at `http://www-gnats.gnu.org:8080/cgi-bin/wwwgnats.pl`. The WWW interface gives you access to open and closed reports. A closed report normally includes a patch or a hint on solving the problem.

To report a bug, first you must find it. With any luck, this will be the hard part. Once you've found a bug, make sure it's really a bug. A good way to do this is to see if the GNU C Library behaves the same way some other C library does. If so, probably you are wrong and the libraries are right (but not necessarily). If not, one of the libraries is probably wrong. It might not be the GNU library. Many historical Unix C libraries permit things that we don't, such as closing a file twice.

[1] The HOWTO is no longer maintained at this site. However, as of this printing, older editions are still available there, as well as contact information for the new maintainer.

If you think you have found some way in which the GNU C Library does not conform to the ISO and POSIX standards (see Section 1.2 [Standards and Portability], page 1), that is definitely a bug. Report it!

Once you're sure you've found a bug, try to narrow it down to the smallest test case that reproduces the problem. In the case of a C library, you really only need to narrow it down to one library function call, if possible. This should not be too difficult.

The final step when you have a simple test case is to report the bug. Do this using the `glibcbug` script. It is installed with libc, or if you haven't installed it, will be in your build directory. Send your test case, the results you got, the results you expected, and what you think the problem might be (if you've thought of anything). `glibcbug` will insert the configuration information we need to see, and ship the report off to `bugs@gnu.org`. Don't send a message there directly; it is fed to a program that expects mail to be formatted in a particular way. Use the script.

If you are not sure how a function should behave, and this manual doesn't tell you, that's a bug in the manual. Report that too! If the function's behavior disagrees with the manual, then either the library or the manual has a bug, so report the disagreement. If you find any errors or omissions in this manual, please report them to the Internet address `bug-glibc-manual@gnu.org`. If you refer to specific sections of the manual, please include the section names for easier identification.

Appendix D Library Maintenance

D.1 Adding New Functions

The process of building the library is driven by the makefiles, which make heavy use of special features of GNU `make`. The makefiles are very complex, and you probably don't want to try to understand them. But what they do is fairly straightforward, and only requires that you define a few variables in the right places.

The library sources are divided into subdirectories, grouped by topic.

The '`string`' subdirectory has all the string-manipulation functions, '`math`' has all the mathematical functions, etc.

Each subdirectory contains a simple makefile, called '`Makefile`', which defines a few `make` variables and then includes the global makefile '`Rules`' with a line like:

```
include ../Rules
```

The basic variables that a subdirectory makefile defines are

`subdir` This is the name of the subdirectory, for example '`stdio`'. This variable *must* be defined.

`headers` This has the names of the header files in this section of the library, such as '`stdio.h`'.

`routines`
`aux` These are the names of the modules (source files) in this section of the library. These should be simple names, such as '`strlen`' (rather than complete file-names, such as '`strlen.c`'). Use `routines` for modules that define functions in the library, and `aux` for auxiliary modules containing things like data definitions. But the values of `routines` and `aux` are just concatenated, so there really is no practical difference.

`tests` This has the names of test programs for this section of the library. These should be simple names, such as '`tester`' (rather than complete file names, such as '`tester.c`'). '`make tests`' will build and run all the test programs. If a test program needs input, put the test data in a file called '*test-program*`.input`'; it will be given to the test program on its standard input. If a test program wants to be run with arguments, put the arguments (all on a single line) in a file called '*test-program*`.args`'. Test programs should exit with zero status when the test passes, and nonzero status when the test indicates a bug in the library or error in building.

`others` This has the names of "other" programs associated with this section of the library. These are programs that are not tests per se, but are other small programs included with the library. They are built by '`make others`'.

`install-lib`
`install-data`
`install` These are files to be installed by '`make install`'. Files listed in '`install-lib`' are installed in the directory specified by '`libdir`' in '`configparms`' or '`Makeconfig`' (see Appendix C [Installing the GNU C Library], page 533). Files listed in `install-data` are installed in the directory specified by '`datadir`' in '`configparms`' or '`Makeconfig`'. Files listed in `install` are installed in the directory specified by '`bindir`' in '`configparms`' or '`Makeconfig`'.

`distribute`
This has other files from this subdirectory that should be put into a distribution tar file. You need not list here the makefile itself or the source and header files listed in the other standard variables. Only define `distribute` if there are files used in an unusual way that should go into the distribution.

`generated`
This has Files that are generated by '`Makefile`' in this subdirectory. These files will be removed by '`make clean`', and they will never go into a distribution.

`extra-objs`
This has extra object files that are built by '`Makefile`' in this subdirectory. This should be a list of file names like '`foo.o`'; the files will actually be found in whatever directory object files are being built in. These files will be removed by '`make clean`'. This variable is used for secondary object files needed to build `others` or `tests`.

D.2 Porting the GNU C Library

The GNU C Library is written to be easily portable to a variety of machines and operating systems. Machine- and operating system-dependent functions are well separated to make it easy to add implementations for new machines or operating systems. This section describes the layout of the library source tree and explains the mechanisms used to select machine-dependent code to use.

All the machine-dependent and operating system-dependent files in the library are in the subdirectory '`sysdeps`' under the top-level library source directory. This directory contains a hierarchy of subdirectories (see Section D.2.1 [Layout of the '`sysdeps`' Directory Hierarchy], page 547).

Each subdirectory of '`sysdeps`' contains source files for a particular machine or operating system, or for a class of machine or operating system (for example, systems by a particular vendor, or all machines that use IEEE 754 floating-point format). A configuration specifies an ordered list of these subdirectories. Each subdirectory implicitly appends its parent directory to the list. For example, specifying the list '`unix/bsd/vax`' is equivalent to specifying the list '`unix/bsd/vax`

`unix/bsd unix`'. A subdirectory can also specify that it implies other subdirectories that are not directly above it in the directory hierarchy. If the file '`Implies`' exists in a subdirectory, it lists other subdirectories of '`sysdeps`' that are appended to the list, appearing after the subdirectory containing the '`Implies`' file. Lines in an '`Implies`' file that begin with a '`#`' character are ignored as comments. For example, '`unix/bsd/Implies`' contains:

```
# BSD has Internet-related things.
unix/inet
```

and '`unix/Implies`' contains:

```
posix
```

So the final list is '`unix/bsd/vax unix/bsd unix/inet unix posix`'.

'`sysdeps`' has a "special" subdirectory called '`generic`'. It is always implicitly appended to the list of subdirectories, so you needn't put it in an '`Implies`' file, and you should not create any subdirectories under it intended to be new specific categories. '`generic`' serves two purposes. First, the makefiles do not bother to look for a system-dependent version of a file that's not in '`generic`'. This means that any system-dependent source file must have an analogue in '`generic`', even if the routines defined by that file are not implemented on other platforms. Second, the '`generic`' version of a system-dependent file is used if the makefiles do not find a version specific to the system you're compiling for.

If it is possible to implement the routines in a '`generic`' file in machine-independent C, using only other machine-independent functions in the C library, then you should do so. Otherwise, make them stubs. A *stub* function is a function that cannot be implemented on a particular machine or operating system. Stub functions always return an error, and set `errno` to `ENOSYS` (Function not implemented).[1] If you define a stub function, you must place the statement `stub_warning(`*function*`)`, where *function* is the name of your function, after its definition; also, you must include the file `<stub-tag.h>` into your file. This causes the function to be listed in the installed `<gnu/stubs.h>`, and makes GNU ld warn when the function is used.

Some rare functions are only useful on specific systems and aren't defined at all on others; these do not appear anywhere in the system-independent source code or makefiles (including the '`generic`' directory), only in the system-dependent '`Makefile`' in the specific system's subdirectory.

If you come across a file that is in one of the main source directories ('`string`', '`stdio`', etc.), and you want to write a machine- or operating system-dependent version of it, move the file into '`sysdeps/generic`' and write your new implementation in the appropriate system-specific subdirectory. If a file is to be system-dependent, it *must not* appear in one of the main source directories.

There are a few special files that may exist in each subdirectory of '`sysdeps`':

'`Makefile`'
: This is a makefile for this machine or operating system, or class of machine or operating system. This file is included by the library makefile

[1] See Loosemore et al., "Error Reporting" (see chap. 1, n. 1).

'Makerules', which is used by the top-level makefile and the subdirectory makefiles. It can change the variables set in the including makefile or add new rules. It can use GNU make conditional directives based on the variable 'subdir' (see above) to select different sets of variables and rules for different sections of the library. It can also set the make variable 'sysdep-routines', to specify extra modules to be included in the library. You should use 'sysdep-routines' rather than adding modules to 'routines' because the latter is used in determining what to distribute for each subdirectory of the main source tree.

Each makefile in a subdirectory in the ordered list of subdirectories to be searched is included in order. Since several system-dependent makefiles may be included, each should append to 'sysdep-routines' rather than simply setting it:

```
sysdep-routines := $(sysdep-routines) foo bar
```

'Subdirs'

This file contains the names of new whole subdirectories under the top-level library source tree that should be included for this system. These subdirectories are treated just like the system-independent subdirectories in the library source tree, such as 'stdio' and 'math'.

Use this when there are completely new sets of functions and header files that should go into the library for the system this subdirectory of 'sysdeps' implements. For example, 'sysdeps/unix/inet/Subdirs' contains 'inet'; the 'inet' directory contains various network-oriented operations that only make sense to put in the library on systems that support the Internet.

'Dist'

This file contains the names of files (relative to the subdirectory of 'sysdeps' in which it appears) that should be included in the distribution. List any new files used by rules in the 'Makefile' in the same directory, or header files used by the source files in that directory. You don't need to list files that are implementations (either C or assembly source) of routines whose names are given in the machine-independent makefiles in the main source tree.

'configure'

This file is a shell script fragment to be run at configuration time. The top-level 'configure' script uses the shell . command to read the 'configure' file in each system-dependent directory chosen, in order. The 'configure' files are often generated from 'configure.in' files using Autoconf.

A system-dependent 'configure' script will usually add things to the shell variables 'DEFS' and 'config_vars'; see the top-level 'configure' script for details. The script can check for '--with-*package*' options that were passed to the

top-level ‘configure’. For an option ‘--with-*package*=*value*’ ‘configure’, sets the shell variable ‘with_*package*’ (with any dashes in *package* converted to underscores) to *value*; if the option is just ‘--with-*package*’ (no argument), then it sets ‘with_*package*’ to ‘yes’.

‘configure.in’
: This file is an Autoconf input fragment to be processed into the file ‘configure’ in this subdirectory.[2] You should write either ‘configure’ or ‘configure.in’, but not both. The first line of ‘configure.in’ should invoke the m4 macro ‘GLIBC_PROVIDES’. This macro does several AC_PROVIDE calls for Autoconf macros that are used by the top-level ‘configure’ script; without this, those macros might be invoked again unnecessarily by Autoconf.

That is the general system for how system dependencies are isolated. The next section explains how to decide what directories in ‘sysdeps’ to use. Section D.2.2 [Porting the GNU C Library to Unix Systems], page 549, has some tips on porting the library to Unix variants.

D.2.1 Layout of the ‘sysdeps’ Directory Hierarchy

A GNU configuration name has three parts: the CPU type, the manufacturer’s name, and the operating system. ‘configure’ uses these to pick the list of system-dependent directories to look for. If the ‘--nfp’ option is *not* passed to ‘configure’, the directory ‘*machine*/fpu’ is also used. The operating system often has a *base operating system*; for example, if the operating system is ‘Linux’, the base operating system is ‘unix/sysv’. The algorithm used to pick the list of directories is simple: ‘configure’ makes a list of the base operating system, manufacturer, CPU type and operating system, in that order. It then concatenates all these together with slashes in between, to produce a directory name; for example, the configuration ‘i686-linux-gnu’ results in ‘unix/sysv/linux/i386/i686’. ‘configure’ then tries removing each element of the list in turn, so ‘unix/sysv/linux’ and ‘unix/sysv’ are also tried, among others. Since the precise version number of the operating system is often not important, and it would be very inconvenient, for example, to have identical ‘irix6.2’ and ‘irix6.3’ directories, ‘configure’ tries successively less specific operating-system names by removing trailing suffixes starting with a period.

As an example, here is the complete list of directories that would be tried for the configuration ‘i686-linux-gnu’ (with the ‘crypt’ and ‘linuxthreads’ add-on):

---

[2] See David MacKenzie et al., “Introduction” in *Autoconf: Generating Automatic Configuration Scripts* (December 2002), http://www.gnu.org/software/autoconf/manual/autoconf-2.57/autoconf.html.

```
sysdeps/i386/elf
crypt/sysdeps/unix
linuxthreads/sysdeps/unix/sysv/linux
linuxthreads/sysdeps/pthread
linuxthreads/sysdeps/unix/sysv
linuxthreads/sysdeps/unix
linuxthreads/sysdeps/i386/i686
linuxthreads/sysdeps/i386
linuxthreads/sysdeps/pthread/no-cmpxchg
sysdeps/unix/sysv/linux/i386
sysdeps/unix/sysv/linux
sysdeps/gnu
sysdeps/unix/common
sysdeps/unix/mman
sysdeps/unix/inet
sysdeps/unix/sysv/i386/i686
sysdeps/unix/sysv/i386
sysdeps/unix/sysv
sysdeps/unix/i386
sysdeps/unix
sysdeps/posix
sysdeps/i386/i686
sysdeps/i386/i486
sysdeps/libm-i387/i686
sysdeps/i386/fpu
sysdeps/libm-i387
sysdeps/i386
sysdeps/wordsize-32
sysdeps/ieee754
sysdeps/libm-ieee754
sysdeps/generic
```

Different machine architectures are conventionally subdirectories at the top level of the ‘`sysdeps`’ directory tree. For example, ‘`sysdeps/sparc`’ and ‘`sysdeps/m68k`’. These contain files specific to those machine architectures, but not specific to any particular operating system. There might be subdirectories for specializations of those architectures, such as ‘`sysdeps/m68k/68020`’. Code that is specific to the floating-point coprocessor used with a particular machine should go in ‘`sysdeps/`*machine*`/fpu`’.

There are a few directories at the top level of the ‘`sysdeps`’ hierarchy that are not for particular machine architectures.

‘`generic`’
: As described above (see Section D.2 [Porting the GNU C Library], page 544), this is the subdirectory that every configuration implicitly uses after all others.

‘ieee754’
: This directory is for code using the IEEE 754 floating-point format, where the C type `float` is IEEE 754 single-precision format, and `double` is IEEE 754 double-precision format. Usually, this directory is referred to in the ‘`Implies`’ file in a machine architecture-specific directory, such as ‘`m68k/Implies`’.

‘libm-ieee754’
: This directory contains an implementation of a mathematical library usable on platforms which use IEEE 754-conformant floating-point arithmetic.

‘libm-i387’
: This is a special case. Ideally, the code should be in ‘`sysdeps/i386/fpu`’, but for various reasons it is kept aside.

‘posix’
: This directory contains implementations of things in the library in terms of POSIX.1 functions. This includes some of the POSIX.1 functions themselves. Of course, POSIX.1 cannot be completely implemented in terms of itself, so a configuration using just ‘`posix`’ cannot be complete.

‘unix’
: This is the directory for Unix-like things (see Section D.2.2 [Porting the GNU C Library to Unix Systems], page 549). ‘`unix`’ implies ‘`posix`’. There are some special-purpose subdirectories of ‘`unix`’:

  ‘unix/common’
  : This directory is for things common to both BSD and System V release 4. Both ‘`unix/bsd`’ and ‘`unix/sysv/sysv4`’ imply ‘`unix/common`’.

  ‘unix/inet’
  : This directory is for `socket` and related functions on Unix systems. ‘`unix/inet/Subdirs`’ enables the ‘`inet`’ top-level subdirectory. ‘`unix/common`’ implies ‘`unix/inet`’.

‘mach’
: This is the directory for things based on the Mach microkernel from CMU (including the GNU operating system). Other basic operating systems (VMS, for example) would have their own directories at the top level of the ‘`sysdeps`’ hierarchy, parallel to ‘`unix`’ and ‘`mach`’.

D.2.2 Porting the GNU C Library to Unix Systems

Most Unix systems are fundamentally very similar. There are variations between different machines, and variations in what facilities are provided by the kernel. But the interface to the operating system facilities is, for the most part, pretty uniform and simple.

The code for Unix systems is in the directory ‘unix’, at the top level of the ‘sysdeps’ hierarchy. This directory contains subdirectories (and subdirectory trees) for various Unix variants.

The functions that are system calls in most Unix systems are implemented in assembly code, which is generated automatically from specifications in files named ‘syscalls.list’. There are several such files, one in ‘sysdeps/unix’ and others in its subdirectories. Some special system-calls are implemented in files that are named with a suffix of ‘.S’; for example, ‘_exit.S’. Files ending in ‘.S’ are run through the C preprocessor before being fed to the assembler.

These files all use a set of macros that should be defined in ‘sysdep.h’. The ‘sysdep.h’ file in ‘sysdeps/unix’ partially defines them; a ‘sysdep.h’ file in another directory must finish defining them for the particular machine and operating system variant. See ‘sysdeps/unix/sysdep.h’ and the machine-specific ‘sysdep.h’ implementations to see what these macros are and what they should do.

The system-specific makefile for the ‘unix’ directory (‘sysdeps/unix/Makefile’) gives rules to generate several files from the Unix system you are building the library on (which is assumed to be the target system you are building the library *for*). All the generated files are put in the directory where the object files are kept; they should not affect the source tree itself. The files generated are ‘ioctls.h’, ‘errnos.h’, ‘sys/param.h’, and ‘errlist.c’ (for the ‘stdio’ section of the library).

Appendix E Contributors to the GNU C Library

The GNU C Library was written originally by Roland McGrath, and is currently maintained by Ulrich Drepper. Some parts of the library were contributed or worked on by other people.

- The `getopt` function and related code was written by Richard Stallman, David J. MacKenzie and Roland McGrath.
- The merge sort function `qsort` was written by Michael J. Haertel.
- The quick sort function used as a fallback by `qsort` was written by Douglas C. Schmidt.
- The memory-allocation functions `malloc`, `realloc` and `free` and related code were written by Michael J. Haertel, Wolfram Gloger and Doug Lea.
- Fast implementations of many of the string functions (`memcpy`, `strlen`, etc.) were written by Torbjörn Granlund.
- The '`tar.h`' header file was written by David J. MacKenzie.
- The port to the MIPS DECStation running Ultrix 4 (`mips-dec-ultrix4`) was contributed by Brendan Kehoe and Ian Lance Taylor.
- The DES encryption function `crypt` and related functions were contributed by Michael Glad.
- The `ftw` and `nftw` functions were contributed by Ulrich Drepper.
- The start-up code to support SunOS shared libraries was contributed by Tom Quinn.
- The `mktime` function was contributed by Paul Eggert.
- The port to the Sequent Symmetry running Dynix version 3 (`i386-sequent-bsd`) was contributed by Jason Merrill.
- The time zone support code is derived from the public-domain time zone package by Arthur David Olson and his many contributors.
- The port to the DEC Alpha running OSF/1 (`alpha-dec-osf1`) was contributed by Brendan Kehoe, using some code written by Roland McGrath.
- The port to SGI machines running Irix 4 (`mips-sgi-irix4`) was contributed by Tom Quinn.
- The port of the Mach and Hurd code to the MIPS architecture (`mips-`*anything*`-gnu`) was contributed by Kazumoto Kojima.
- The floating-point printing function used by `printf` and friends and the floating-point reading function used by `scanf`, `strtod` and friends were written by Ulrich Drepper. The multiprecision integer functions used in those functions are taken from GNU MP, which was contributed by Torbjörn Granlund.
- The internationalization support in the library, and the support programs `locale` and `localedef`, were written by Ulrich Drepper. Ulrich Drepper adapted the support code for message catalogs ('`libintl.h`', etc.) from

the GNU `gettext` package, which he also wrote. He also contributed the `catgets` support and the entire suite of multibyte and wide-character support functions (‘`wctype.h`’, ‘`wchar.h`’, etc.).

- The implementations of the ‘`nsswitch.conf`’ mechanism and the files and DNS backends for it were designed and written by Ulrich Drepper and Roland McGrath, based on a backend interface defined by Peter Eriksson.
- The port to Linux i386/ELF (`i386-`*anything*`-linux`) was contributed by Ulrich Drepper, based in large part on work done in Hongjiu Lu’s Linux version of the GNU C Library.
- The port to Linux/m68k (`m68k-`*anything*`-linux`) was contributed by Andreas Schwab.
- The ports to Linux/ARM (`arm-`*ANYTHING*`-linuxaout`) and ARM standalone (`arm-`*ANYTHING*`-none`), as well as parts of the IPv6 support code, were contributed by Philip Blundell.
- Richard Henderson contributed the ELF dynamic-linking code and other support for the Alpha processor.
- David Mosberger-Tang contributed the port to Linux/Alpha (`alpha-`*anything*`-linux`).
- The port to Linux on PowerPC (`powerpc-`*anything*`-linux`) was contributed by Geoffrey Keating.
- Miles Bader wrote the argp argument-parsing package, and the argz/envz interfaces.
- Stephen R. van den Berg contributed a highly-optimized `strstr` function.
- Ulrich Drepper contributed the `hsearch` and `drand48` families of functions; reentrant ‘`..._r`’ versions of the `random` family; System V shared memory and IPC support code; and several highly-optimized string functions for i*x*86 processors.
- The math functions are taken from `fdlibm-5.1` by Sun Microsystems, as modified by J.T. Conklin, Ian Lance Taylor, Ulrich Drepper, Andreas Schwab and Roland McGrath.
- The `libio` library used to implement `stdio` functions on some platforms was written by Per Bothner and modified by Ulrich Drepper.
- Eric Youngdale and Ulrich Drepper implemented versioning of objects on the symbol level.
- Thorsten Kukuk provided an implementation for NIS (YP) and NIS+, securelevel 0, 1 and 2.
- Andreas Jaeger provided a test suite for the math library.
- Mark Kettenis implemented the utmpx interface and an utmp daemon.
- Ulrich Drepper added character conversion functions (`iconv`).
- Thorsten Kukuk provided an implementation for a caching daemon for NSS (nscd).

- Tim Waugh provided an implementation of the POSIX.2 wordexp function family.
- Mark Kettenis provided a Hesiod NSS module.
- The Internet-related code (most of the 'inet' subdirectory) and several other miscellaneous functions and header files have been included from 4.4 BSD with little or no modification. The copying permission notice for this code can be found in the file 'LICENSES' in the source distribution.
- The random-number generation functions random, srandom, setstate and initstate, which are also the basis for the rand and srand functions, were written by Earl T. Cohen for the University of California at Berkeley and are copyrighted by the Regents of the University of California. They have undergone minor changes to fit into the GNU C Library and to fit the ISO C standard, but the functional code is Berkeley's.
- The DNS-resolver code is taken directly from BIND 4.9.5, which includes copyrighted code from UC Berkeley and from Digital Equipment Corporation. See the file 'LICENSES' for the text of the DEC license.
- The code to support Sun RPC is taken verbatim from Sun's RPCSRC-4.0 distribution; see the file 'LICENSES' for the text of the license.
- Some of the support code for Mach is taken from Mach 3.0 by CMU; the file if_ppp.h is also copyright by CMU, but under a different license; see the file 'LICENSES' for the text of the licenses.
- Many of the IA64 math functions are taken from a collection of "Highly Optimized Mathematical Functions for Itanium" that Intel makes available under a free license; see the file 'LICENSES' for details.
- The getaddrinfo and getnameinfo functions and supporting code were written by Craig Metz; see the file 'LICENSES' for details on their licensing.
- Many of the IEEE 64-bit double precision math functions (in the 'sysdeps/ieee754/dbl-64' subdirectory) come from the IBM Accurate Mathematical Library, contributed by IBM.

Appendix F Free Software Needs Free Documentation

The biggest deficiency in the free software community today is not in the software—it is the lack of good free documentation that we can include with the free software. Many of our most important programs do not come with free reference manuals and free introductory texts. Documentation is an essential part of any software package; when an important free software package does not come with a free manual and a free tutorial, that is a major gap. We have many such gaps today.

Consider Perl, for instance. The tutorial manuals that people normally use are nonfree. How did this come about? Because the authors of those manuals published them with restrictive terms—no copying, no modification, source files not available—which exclude them from the free software world.

That wasn't the first time this sort of thing happened, and it was far from the last. Many times we have heard a GNU user eagerly describe a manual that he is writing, his intended contribution to the community, only to learn that he had ruined everything by signing a publication contract to make it nonfree.

Free documentation, like free software, is a matter of freedom, not price. The problem with the nonfree manual is not that publishers charge a price for printed copies—that in itself is fine. (The Free Software Foundation sells printed copies of manuals, too.) The problem is the restrictions on the use of the manual. Free manuals are available in source code form, and give you permission to copy and modify. Nonfree manuals do not allow this.

The criteria of freedom for a free manual are roughly the same as for free software. Redistribution (including the normal kinds of commercial redistribution) must be permitted, so that the manual can accompany every copy of the program, both on-line and on paper.

Permission for modification of the technical content is crucial too. When people modify the software, adding or changing features, if they are conscientious they will change the manual too—so they can provide accurate and clear documentation for the modified program. A manual that leaves you no choice but to write a new manual to document a changed version of the program is not really available to our community.

Some kinds of limits on the way modification is handled are acceptable. For example, requirements to preserve the original author's copyright notice, the distribution terms, or the list of authors, are ok. It is also no problem to require modified versions to include notice that they were modified. Even entire sections that may not be deleted or changed are acceptable, as long as they deal with nontechnical topics (like this one). These kinds of restrictions are acceptable because they don't obstruct the community's normal use of the manual.

However, it must be possible to modify all the *technical* content of the manual, and then distribute the result in all the usual media, through all the usual channels. Otherwise, the restrictions obstruct the use of the manual, it is not free, and we need another manual to replace it.

Please spread the word about this issue. Our community continues to lose manuals to proprietary publishing. If we spread the word that free software needs free reference manuals and free tutorials, perhaps the next person who wants to contribute by writing documentation will realize, before it is too late, that only free manuals contribute to the free software community.

If you are writing documentation, please insist on publishing it under the GNU Free Documentation License or another free documentation license. Remember that this decision requires your approval—you don't have to let the publisher decide. Some commercial publishers will use a free license if you insist, but they will not propose the option; it is up to you to raise the issue and say firmly that this is what you want. If the publisher you are dealing with refuses, please try other publishers. If you're not sure whether a proposed license is free, write to licensing@gnu.org.

You can encourage commercial publishers to sell more free, copylefted manuals and tutorials by buying them, and particularly by buying copies from the publishers that paid for their writing or for major improvements. Meanwhile, try to avoid buying nonfree documentation at all. Check the distribution terms of a manual before you buy it, and insist that whoever seeks your business must respect your freedom. Check the history of the book, and try reward the publishers that have paid or pay the authors to work on it.

The Free Software Foundation maintains a list of free documentation published by other publishers, at http://www.fsf.org/doc/other-free-books.html.

Appendix G GNU Lesser General Public License

Version 2.1, February 1999

[This is the first released version of the Lesser GPL. It also counts as the successor of the GNU Library Public License, version 2, hence the version number 2.1.]

G.0.1 Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some specially designated software—typically libraries—of the Free Software Foundation and other authors who decide to use it. You can use it too, but we suggest you first think carefully about whether this license or the ordinary General Public License is the better strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish); that you receive source code or can get it if you want it; that you can change the software and use pieces of it in new free programs; and that you are informed that you can do these things.

To protect your rights, we need to make restrictions that forbid distributors to deny you these rights or to ask you to surrender these rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into nonfree programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the *Lesser* General Public License because it does *Less* to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing nonfree programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, nonfree programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used nonfree libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in nonfree programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in nonfree programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and

a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

G.0.2 TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

   A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

   The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

   "Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. The modified work must itself be a software library.

   b. You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

c. You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

d. If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

   (For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

   Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

   This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms

of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

   However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

   When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

   If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

   Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

   You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

   a. Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete

machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

b. Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.

c. Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

d. If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

e. Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

   a. Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

   b. Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.
10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.
11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

    If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

    It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

    This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.
12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder

who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF

THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

G.0.3 How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> *one line to give the library's name and an idea of what it does.*
> Copyright (C) *year name of author*
>
> This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.
>
> This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.
>
> You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301, USA.

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the library, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the library 'Frob' (a library for tweaking knobs) written by James Random Hacker.
>
> *signature of Ty Coon*, 1 April 1990
> Ty Coon, President of Vice

That's all there is to it!

Appendix H GNU Free Documentation License

Version 1.1, March 2000

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

   You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

   You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title

equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one

of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

H.0.1 ADDENDUM: How to Use This License for Your Documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (C) *year* *your name*.
> Permission is granted to copy, distribute and/or modify this document
> under the terms of the GNU Free Documentation License, Version 1.1
> or any later version published by the Free Software Foundation;
> with the Invariant Sections being *list their titles*, with the
> Front-Cover Texts being *list*, and with the Back-Cover Texts being *list*.
> A copy of the license is included in the section entitled ``GNU
> Free Documentation License''.

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being *list*"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Concept Index

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Type Index

T

U

V

Function and Macro Index

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Variable and Constant Macro Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

Program and File Index

T

U

V