Интерфейс на UNIX

Пламен Танов
Ненко Табаков
Технологично училище „Електронни системи“
Технически университет – София

версия 0.1

Въведение

- Операционната система UNIX поддържа съвкупност от системни извиквания. Те представляват функции на операционната система, които могат да се извикват от потребителските програми
- В UNIX входът и изходът се осъществява чрез четене и запис във файлове. Всички периферни устройства (usb, серийни портове, CD, т.н.) представляват файлове във файловата система. Комуникацията с тях се извършва като се пише и чете в тези файлове

Отваряне на файл

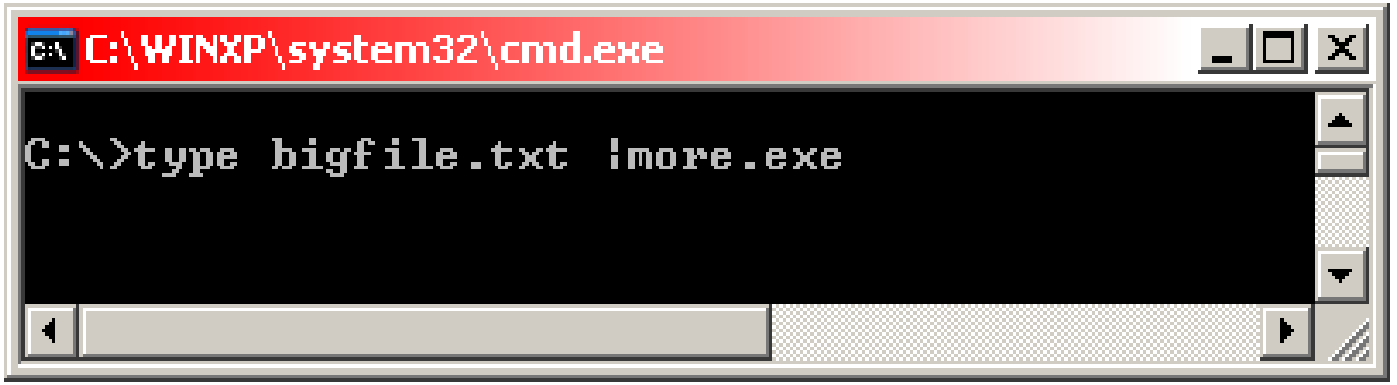
- Системата трябва да бъде уведомена какво ще се прави със съответния файл – дали ще се чете или пише
- Ако ще се записва във файл може да се наложи той да бъде създаден и/или пък предишното му съдържание да бъде изтрито
- Системата проверява дали потребителят има права да извърши съответните действия
- Ако всичко е наред към програмата се връща не отрицателно число. То се нарича файлов дескриптор и служи за достъп до файла

Стандартни дескриптори

При стартирането на дадена програма, се отварят три файла съответно с файлови дескриптори 0, 1 и 2:
0 – стандартен вход
1 – стандартен изход
2 – стандартен изход за грешки

Пренасочване на стандартния вход и изход

Това е функция на операционната система.
При Linux, Windows и DOS:



Вход и изход на ниско ниво

За вход и изход се използват системните извиквания **`read`** и **`write`**, достъпът до които се осъществява посредством две функции, наречени **`read()`** и **`write()`**

В различните системи те са описани на различни места (**`fcntl.h`**, **`unistd.h`**, **`sys/file.h`**, ...)

```
int read (int fd, char *buf, int n);

int write (int fd, char *buf, int n);
```

read()

```
int read (int fd, char *buf, int n);
```

- **`fd`** – файлов дескриптор
- **`buf`** – масив от символи, където ще отидат данните
- **`n`** – брой байтове, които трябва да се прочетат в **`buf`**
- Функцията връща като резултат броя на прочетените байтове. Той може да се окаже по–малък от заявения брой (**`n`**) (ако файлът е по-малък)
- Върната стойност **`0`** означава край на файла
- Върната стойност **`-1`** означава грешка

write()

```
int write (int fd, char *buf, int n);
```

- **`fd`** – файлов дескриптор
- **`buf`** – масив от символи
- **`n`** – брой байтове, които трябва да се запишат от **`buf`** във файла
- Функцията връща като резултат броя на записаните байтове. Ако възникне грешка той е различен от **`n`**

Пример

```
#include <fcntl.h>
#define BUFSIZE 200

int read (int fd, char *buf, int n);
int write (int fd, char *buf, int n);

int main () {
   char buf[BUFSIZE];
   int n;
   while ((n = read(0, buf, BUFSIZE))>0)
      write (1, buf, n);
   return 0;
}
```

Пример

```
#include <fcntl.h>

/* getchar: версия без буфериране */
int getchar (){
   char c;
   return (read(0, &c, 1) == 1)? (unsigned char) c : EOF;
}
```

Пример

```
#include <fcntl.h>
#define BUFSIZE 200

/* getchar: версия с буфериране */
int getchar(void) {
   static char buf[BUFSIZE];
   static char *bufp = buf;
   static int n = 0;
   if (n == 0) { /* буферът е празен */
      n = read(0, buf, sizeof buf);
      bufp = buf;
   }

   return (--n >= 0) ? (unsigned char) *bufp++ : EOF;
}
```

Работа с други файлове

Ако не се използват стандартните вход, изход или поток за грешки, то трябва явно да се отвори файл за четене/писане. За тази цел съществуват системните извиквания – **`open`** и **`creat`** и съответните функции за тях:

```
int open(char *name, int flags, int perms);

int creat(char *name, int perms);
```

open()

```
int open (char *name, int flags, int perms);
```

- Отваря съществуващ файл
- **`name`** – името на файла, който ще бъде отворен
- **`flags`** – режим на отваряне:
  - **`O_RDONLY`** – отваря се само за четене
  - **`O_WRONLY`** – отваря се само за писане
  - **`O_RDWR`** – отваря се едновременно за четене и писане
  - В <**`fcntl.h`**> в System V UNIX и в <**`sys/file.h`**> в Berkeley (BSD)
- **`perms`** – правата върху файла
- Функцията връща файлов дескриптор, **`-1`** при грешка

creat()

```
int creat (char *name, int perms);
```

- Създава и отваря несъществуващ файл или отваря и изтрива съдържанието на съществуващ
- **`name`** – името на файла, който ще бъде отворен
- **`perms`** – правата върху файла (**`1: x`**, **`2: w`**, **`4: r`** за потребител, група, останали)
- Функцията връща файлов дескриптор, -1 при грешка

Пример$_1$

```
#include <stdio.h>
#include <fcntl.h>
#define BUFSIZE 1204
#define PERMS 0666//r+w=2+4=6, за потребител, група и останали
void error (char *, ...);
int main (int argc, char *argv[]) {
   int f1, f2, n;
   char buf[BUFSIZE];
   if (argc != 3)
      error ("Usage: cp from to");
   if ((f1 = open(argv[1], O_RDONLY, 0)) == -1)
      error ("cp: cannot open %s", argv[1]);
   if ((f2 = creat(argv[2], PERMS)) == -1)
      error ("cp: cannot create %s, mode %3o",argv[2],PERMS);
   while ((n = read(f1, buf, BUFSIZE)) > 0)
      if (write (f2, buf, n) != n)
         error ("cp : write error on file %s", argv[2]);
   close(f1); close(f2);//накрая затваряме файловете
   return 0;
}
```

Пример$_2$

```
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>//за exit()

void error (char *fmt, ...) {
   va_list args;

   va_start(args, fmt);
   fprintf(stderr, "error :");

   vfprintf(stderr, fmt, args);
/* printf функциите, чиито имена започват с v означават, че
последният им аргумент ще е от тип va_list, а не ...   */

   fprintf(stderr, "\n");
   va_end(args);

   exit(1);
}
```

close(), unlink(), remove()

```
int close(int fd);
int unlink(const char *name);
int remove(const char *name);
```

- **`close()`** служи за затваряне на файл
- **`unlink()`** и **`remove()`** служат за изтриване на файл от диска
- **`unlink()`** е от UNIX интерфейса
- **`remove()`** - от **`ANSI C`** стандарта

lseek()$_1$

```
long lseek (int fd, long offset, int origin);
```

Входът и изходът обикновено са последователни – **`read()`**/**`write()`** започват от позицията, на която предишната е спряла. Функцията **`lseek()`** предоставя начин, чрез който може да се премества позицията вътре във файла, без да се чете или пише

lseek()$_2$

```
long lseek (int fd, long offset, int origin);
```

- **`fd`** – файлов дескриптор
- **`offset`** – отместването спрямо **`origin`** (положително или отрицателно число)
- **`origin`** – мястото, спрямо което да се направи отместването: **`SEEK_SET, SEEK_CUR и SEEK_END`**
  - **`0`** – **`offset`** ще се изчислява спрямо началото на файла
  - **`1`** – **`offset`** ще се изчислява спрямо текущата позиция
  - **`2`** – **`offset`** ще се изчислява спрямо края на файла
- Функцията връща новата позиция във файла или **`-1`** при грешка

Пример

```
/*прочита n байта от позицията pos и ги записва в buf*/
int get (int fd, long pos, char *buf, int n) {
   if (lseek(fd, pos, 0) >= 0)
      return read(fd, buf, n);
   else
      return -1;
}
```

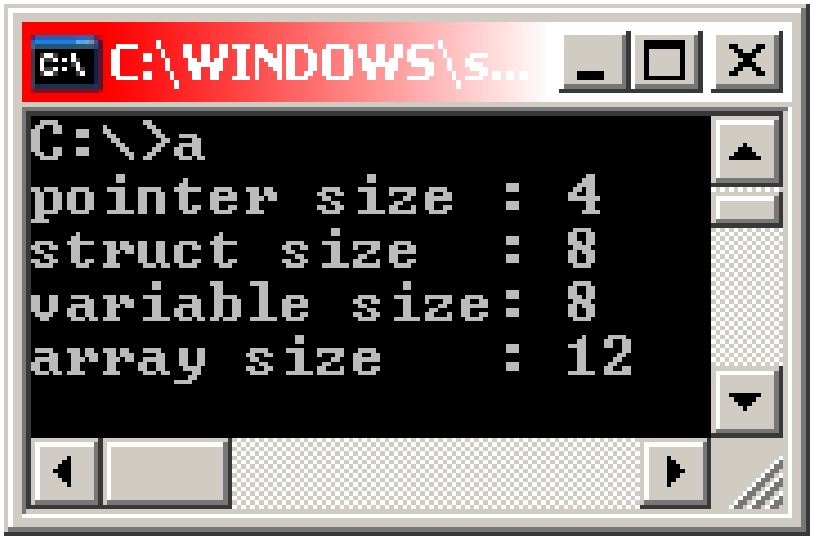
sizeof

- Колко байта заема даден елемент в паметта

```
#include <stdio.h>

typedef struct {
  int a, b;
} TwoInts;

int main () {
  TwoInts i, * pi = &i;
  int arr[3];
  printf("pointer size : %d\n", sizeof(pi));//указател - 4
  printf("struct size  : %d\n", sizeof(TwoInts));//2*4 = 8
  printf("variable size: %d\n", sizeof(i));      //2*4 = 8
  printf("array size   : %d\n", sizeof(arr));    //3*4 = 12
  return 0;
}
```



Динамично заделяне на памет 1

- Динамичното заделяне на памет позволява да се създават масиви, чиято дължина не се знае предварително (динамични масиви)
- Като цяло използвайки памет, заделена по този начин, може програмата да се направи по–малка в сравнение с такава, в която паметта е заделена статично
- В **C** памет се заделя с функциите **`malloc()`** и **`calloc()`**
- Памет заделена с някоя от тези функции трябва да бъде освободена след ползването ѝ чрез извикване на функцията **`free()`**

Динамично заделяне на памет$_2$

- **`malloc()`** прави заявка за памет към операционната система. Тъй като другите елементи от програмата също могат да правят заявки за памет, без да използват този механизъм, паметта предоставена от две последователни извиквания на **`malloc()`** може да не бъде последователно разположена, но в едно извикване на **`malloc()`** заделената памет е последователно разположена
- Незаетата памет се пази като списък от свободни блокове. Всеки блок съдържа размер, колко е голям и указател към следващ празен блок

Динамично заделяне на памет 3

- Когато бъде направена заявка, списъкът със свободната памет се обхожда, докато не бъде открит достатъчно голям блок
- Ако блокът е с размер равен на заявения, той се премахва от списъка и се подава на потребителя
- Ако блокът е по–голям от заявения, той се разделя и желаното количество се дава на потребителя, а останалата част се запазва в списъка като свободна
- Освобождаването на памет също е придружено с претърсване на списъка с празните места, за да се намери подходящо място, където да се постави освободената памет
- Ако освободеният блок се окаже съседен на свободен блок, то двата блока се сливат в един по–голям блок

Динамично заделяне на памет 4

```
void *malloc (size_t size);
```

- Връща указател към място в паметта за обект с големина **size** байта. Ако не може да открие такова място връща **NULL**

```
void *calloc (size_t nobj, size_t size);
```

- Връща указател към място в паметта за масив от **nobj** на брой обекта, всеки от който е с големина **size** байта. Ако не може да открие такова място - връща **NULL**

```
void free (void *p);
```

- Освобождава мястото в паметта, към което сочи **p**. Трябва да се извиква всеки път, когато е заделена памет с някоя от горните две функции. Ако мястото не е заделено с тях последиците са непредвидими

Пример 1

```
#include <stdio.h>
#include <stdlib.h>

int main (){
   int *n;
   int i;
   int count;
   int sum = 0;
   printf("Enter count : ");
   scanf("%d", &count);
   //заделяме памет за въвеждане на елементите:
   n = (int *) malloc(count * sizeof(int));
   printf ("Enter values: \n");
```

Пример$_2$

```
   for (i = 0; i<count; i++) {
      printf ("%d value = ", i+1);

      //заделената памет може да се достъпва като указател:
      scanf ("%d", (n+i));
   }

   for (i = 0; i<count; i++)
      sum += n[i];//може да се достъпва и като масив
   printf ("The sum of entered values is %d\n", sum);

//динамично заетата памет не ни трябва вече - освобождава ме я:
   free(n);
   return 0;
}
```

Пример$_1$

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
   int x;
   int y;
} Point;

int main () {
   Point *points;
   int i;
   int count;
   int sumy = 0;
   int sumx = 0;
   printf ("Enter count : ");
   scanf ("%d", &count);
//заделяме памет за въвеждане на елементите
//те са count на брой, всеки с големина sizeof(Point) байта:
   points = (Point *) calloc(count, sizeof(Point));
```

Пример 2

```
   printf ("Enter values : \n");
   for (i = 0; i<count; i++){
      printf ("%d x = ", i+1);
//достъпваме i'тия елемент като указател:
      scanf ("%d", &((points+i)->x));
      printf ("%d y = ", i+1);
      scanf ("%d", &(points[i].y));//и като масив
   }

   for (i = 0; i<count; i++){
      sumx += (points+i)->x;
      sumy += points[i].y;
   }
   printf ("The sum of entered coordinates over x is %d\n",
sumx);
   printf ("The sum of entered coordinates over y is %d\n",
sumy);

   free(points);//освобождава динамично заетата памет
   return 0;
}
```

Двумерни динамични масиви

- Предсатвя се като едномерен масив от указатели към едномерни масиви
- Първо заделяме място за масива от указатели:

```
arr = malloc(ROWS * sizeof(int *));
```

- След това заделяме място за всеки един ред (едномерен масив):

```
for(i=0;i<ROWS;i++)
   arr[i] = malloc(COLS * sizeof(int));
```

- Освобождаването на памет се извършва в обратен ред:

```
for (i = 0; i < ROWS; i++)
   free(arr[i]);
free(arr);
```

Пример 1

```
#include <stdio.h>
#include <stdlib.h>

#define ROWS 3
#define COLS 5
int main () {

  int ** arr;
  int i, j;

  //правим масив от указатели към int
  if ((arr = malloc(ROWS * sizeof(int *))) == NULL) {
//или:  if ((arr = calloc(ROWS , sizeof(int *))) == NULL) {
    fprintf(stderr, "not enought memory");
    exit(1);
  }
```

Пример$_2$

```
  for(i=0;i<ROWS;i++) {
/* инициализираме всеки един указател към int да сочи към
едномерен масив от int. Забележете, че всеки един едномерен
масив може да е с различна големина!!!*/
    if ((arr[i] = malloc(COLS * sizeof(int))) == NULL){
//или: if ((arr[i] = calloc(COLS , sizeof(int))) == NULL) {
      fprintf(stderr, "not enought memory");
      exit(2);
    }
  }
  //попълваме двумерния масив
  for(i = 0;i<ROWS;i++) {
    for(j = 0;j<COLS;j++) {
        arr[i][j] = 10*i+j;
    }
  }
```

Пример$_3$

```
  //извеждаме двумерния масив:
  for(i = 0;i<ROWS;i++) {
    for(j = 0;j<COLS;j++) {
        printf("%5.2d", arr[i][j]);//*(*(arr+i)+j)
    }
    printf("\n");
  }

  //освобождаваме паметта в обратен ред
  for (i = 0; i < ROWS; i++) {
    free(arr[i]);
  }
  free(arr);

  return 0;
}
```