Технологично училище "Електронни системи"
Технически университет – София

Операционни системи

Задача 5: Команден интерпретатор – част II

Любомир Чорбаджиев
lchorbadjiev@elsys-bg.org

16 май 2012 г.

1 Условие на задачата

1.1 Основна функционалност

Целта на задачата е да се развие интерпретатора `shell` от Задача 4: Команден интерпретатор – част I, като се добавят възможности за пренасочване на стандартния вход и стандартния изход на изпълняваните приложения (команди) и възможност за създаване на pipe между две приложения (команди).

При стартиране на програмата, тя започва да чете редове от стандартния вход и да ги интерпретира. За тази цел програмата трябва да раздели командния ред на думи, като за разделители се използва интервал (`' '`).

Допълнително, програмата трябва да третира по специален начин:

- символите `>` и `<`, които се използват за пренасочване на стандартния вход и изход на стартираното приложение;
- символът `|`, който се използва за създаване на pipe между приложения.

Например, ако на стандартния вход се напише следния ред:

```
/bin/ls -l /usr/include
```

то командния интерпретатор трябва го превърне в следния масив от думи:

```
{"/bin/ls","-l","/usr/include"}
```

Първата дума се интерпретира като име на файл, който трябва да се изпълни. Програмата трябва да се опита да изпълни този файл, а като аргументи от командния ред трябва се предаде масив от всички думи в командния ред.

В разгледания пример, при команден ред:

```
/bin/ls -l /usr/include
```

програмата трябва да се опита да изпълни файла `/bin/ls` и да му предаде следния масив от аргументи:

```
{"/bin/ls","-l","/usr/include"}
```

1.2 Пренасочване на стандартния изход (20 точки)

Командния интерпретатор трябва да поддържа пренасочване на стандартния изход. Например, ако на интерпретатора се подаде следната команда:

```
/bin/ls -l /usr/include > list.txt
```

то той трябва да изпълни командата

```
/bin/ls -l /usr/include
```

като стандартния изход на тази команда трябва да се запише във файлът `list.txt`.

1.3 Пренасочване на стандартния вход (20 точки)

Командния интерпретатор трябва да поддържа пренасочване на стандартния вход. Например, ако на интерпретатора се подаде следната команда:

```
/bin/wc < list.txt
```

то той трябва да изпълни командата

```
/bin/wc
```

като изпълняваната команда трябва да използва за стандартен вход файлът `list.txt`.

1.4 Създаване на pipe (40 точки)

Командния интерпретатор трябва да поддържа създаване на pipe между две приложения. Например, ако на интерпретатора се подаде командата:

```
/bin/ls -l /usr/include | /bin/wc
```

то командния интерпретатор трябва:

- да създаде два процеса;
- да създаде pipe между двата процеса, така че стандартния изход на първия процес да бъде насочен към стандартния вход на втория процес;
- в рамките на първия процес да изпълни командата `/bin/ls -l /usr/include`;
- в рамките на втория процес да изпълни командата `/bin/wc`.

1.5 Допълнителни изисквания

Командния интерпретатор трябва да позволява организиране на pipe между две[1] приложения (команди). За тази цел приложението трябва да поддържа структура, в която да съхранява:

- името на файла, която трябва да се изпълни;
- списъкът с аргументи, които трябва да бъдат предадени на приложението от командния ред;
- файлове, към които трябва да бъде пренасочен стандартния вход и стандартния изход на приложението.

По-долу е дадена примерна реализация на такава структура (или клас):

```
struct Job {
  char* command;
  char** argv;
  int stdin;
  int stdout;
  ...
};
```

[1] Или повече.

където `stdin` и `stdout` са файловите дескриптори на стандартния вход и стандартния изход на приложението. Когато тези дескриптори се различават от `STDIN_FILENO` и `STDOUT_FILENO`, то при изпълнение на приложението стандартния вход/изход трябва да бъде пренасочен.

За пренасочване на стандартния вход/изход трябва да се използва системната функция

```
int dup2(int old,int new);
```

Ако интерпретатора трябва да направи pipe между две приложения, то при обработване на командния ред трябва да бъдат създадени две структури **struct** `Job`, да се стартират двете приложения, да се направи pipe между тях, като стандартния изход на първото приложение трябва да се пренасочи към стандартния вход на второто приложение.

За създаване на pipe трябва да се използва системната функция:

```
int pipe(int filedesc[2]);
```

2 Изисквания към решението и оценяване

1. Решението на задачата трябва да бъде написано на езика `C` съгласно ISO/IEC 9899:1999.

2. Правилата за оценяване са следните. Приемаме, че напълно коректна и написана спрямо изискванията програма получава максималния брой точки — 100% или 80 точки. Ако в решението има пропуски, максималният брой точки ще бъде намален съгласно правилата описани по-долу.

3. Задължително към файловете с решението трябва да е приложен и `Makefile`. Изпълнимият файл, който се създава по време на компилация на решението, трябва да се казва `shell2`.

4. При проверка на решението програмата ви ще бъде компилирана и тествана по следния начин:

   ```
   make
   ./shell2
   ```

   Предходната процедура ще бъде изпълнена няколко пъти с различни входни данни за да се провери дали вашата програма работи коректно.

5. Реализацията на програмата трябва да спазва точно изискванията. Всяко отклонение от изискванията ще доведе до получаване на 0 точки за съответната част от условието.

6. Работи, които са предадени по-късно от обявеното (или не са предадени), ще бъдат оценени с 0 точки.

7. Програмата ви трябва да съдържа достатъчно коментари. Оценката на решения без коментари или с недостатъчно и/или мъгляви коментари ще бъде намалена с 30%.

8. Всеки файл от решението трябва да започва със следният коментар:

```
//------------------------------------------
// NAME: Ivan Ivanov
// CLASS: XIa
// NUMBER: 13
// PROBLEM: #1
// FILE NAME: xxxxxx.yyy.zzz (unix file name)
// FILE PURPOSE:
//  няколко реда, които описват накратко
//  предназначението на файла
//  ...
//------------------------------------------
```

Всяка функция във вашата програма трябва да включва кратко описание в следния формат:

```
//------------------------------------------
// FUNCTION: xxyyzz (име на функцията)
//  предназначение на функцията
// PARAMETERS:
//  списък с параметрите на функцията
//  и тяхното значение
//------------------------------------------
```

9. Лош стил на програмиране и липсващи заглавни коментари ще ви костват 30%.

10. Програми, които не се компилират получават 0 точки. Под „не се компилират“ се има предвид произволна причина, която може да причини неуспешна компилация, включително липсващи файлове, неправилни имена на файлове, синтактични грешки, неправилен или липсващ `Makefile`, и т.н. Обърнете внимание, че в `UNIX` имената на файловете са “case sensitive”.

11. Програми, които се компилират, но не работят, не могат да получат повече от 50%. Под „компилира се, но не работи“ се има предвид, че вие сте се опитали да решите проблема до известна степен, но не сте успели да направите пълно решение. Често срещан проблем, който спада към този случай, е че вашият Makefile генерира изпълним файл, но той е именуван с име, различно от очакваното (т.е. `shell2` в разглеждания случай).

12. Безсмислени или мъгляви програми ще бъдат оценявани с 0 точки, независимо че се компилират.

13. Програми, които дават неправилни или непълни резултати, или програми, в които изходът и/или форматирането се различава от изискванията ще получат не повече от 70%.

14. Всички наказателни точки се сумират. Например, ако вашата програма няма задължителните коментари в началото на файлове и функциите се отнемат 30%, ако няма достатъчно коментари се отнемат още 30%, компилира се, но не работи правилно — още 30%, то тогава резултатът ще бъде: $80 * (100 - 30 - 30 - 30)\% = 80 * 10\% = 8$ точки

15. Работете самостоятелно. Групи от работи, които имат твърде много прилики една с друга, ще бъдат оценявани с 0 точки.