

CSCI 480 PROGRAMMING ASSIGNMENT #2
due Tues Nov. 25, 2008, 6:30 PM

Specifications:

In this assignment, you will implement a microshell in C/C++ on the departmental Linux machines. This shell does the following:

- Print a prompt “myshell>” and wait for your input;
- Execute the command you type in after the prompt and print a new prompt.
- The shell understands the commands “quit” and “q” as special commands to exit.
- The shell understands a special symbol “||”, by which you can pipe the output of one command to next command. To simplify, this assignment only requires one pipe between two commands, such as in “cat myfile || sort”. Please note that the standard Linux pipe is “|”, which is different from what our microshell will use.

You can test your program using a text file named “myfile” with the contents of:

```
Illinois  
USA  
DeKalb
```

Output:

```
turing%>mysh  
myshell>cat myfile || sort  
DeKalb  
Illinois  
USA  
myshell>quit
```

```
turing%>mysh  
myshell>less myfile || grep DeKalb  
DeKalb  
myshell>ls -Alst  
myfile (etc.)  
mysh  
myshell>q
```

```
turing%>
```

Background Knowledge:

1. You may need strtok() to parse the command line for you. Read the manual page to understand this function.
2. The parent process needs to call waitpid(..) to wait for the commands to complete.

3. The assignment will need several system calls for process management and IPC such as `fork()`, `exec()`, `pipe()`, `dup()`. Read the manual pages (e.g. "man dup") to understand their usage. You will use `fork()` to create two child processes and use `pipe()` with the help of `dup()` to set up the communication between the child processes. `dup()` is used to duplicate the file descriptors so that you can replace the standard input or output of a process by the file descriptors of a pipe.
4. You need to close all the unneeded file descriptors of the pipes. You need to close the two pipe file descriptors in the parent process, the read end of the pipe for the first child process, and the write end of the pipe for the second child process. After `dup()` is called in each child process, you can close the write end for the first process and the read end for the second process since they are no longer needed.
5. You can use any one of the six exec system calls. If you use `execv()`, `execvp()` or `execve()`, you need to build an array of pointers to the arguments. The last element of the array must be a null pointer.

Suggestions:

Tackle the problem step by step. First make sure that your shell is taking inputs correctly. Then you should test the execution of the commands without involving any pipes. Afterwards you can go ahead and solve the pipe problem.

Requirements:

The programs should 1) work according to the specifications; 2) be comprehensible and well commented; 3) check the return value of the system calls and have proper error handling. Grading will be based on the correctness of the program, as well as documentation and formatting.

Submission:

Same as program #1.

Hints

- How to find the library for a system call.

Check the manual page.

For example: Do “man strtok”. In the synopsis, you will see `#include <strings.h>`.

If you are interested in the exact contents of the header file, you can go to `/usr/include`, and do “less strings.h”.

- How do I use `pipe()`?

You need to declare a pair of file descriptors and then call `pipe()`. Do not forget to check the return value of the system call.

Example:

```
int pfd[2];
if(pipe(pfd) == -1) { //error handling }
```

Normally a process that calls `pipe()` will then call `fork()`, creating an IPC channel.

- How do I use `dup()`?

A `dup()` call allocates the next free entry in the file descriptor table and copies the contents of the entry being duplicated.

A process has file descriptor 0 as the standard input, file descriptor 1 as the standard output. If we do the following in a process:

```
close(1);
dup(pfd[1]);
```

Then the write end of the pipe was copied into entry 1. In other words, the standard output of the process is now the write end of the pipe.

You can now close the original pipe write end file descriptor by

```
close(pfd[1]);
```

If you only plan to use one of the two pipe file descriptors, close the unneeded one as well.

- How do I use `execvp()`?

Execvp() is one of the six exec functions. You can also use others in the assignment. If you plan to use execvp, its syntax is:

```
int execvp(const char *filename, char *const argv[]);
```

You will need to build up the array of argv[], which contains all the arguments, and ends with NULL.

When you call it, it could be something like:

```
execvp(comargs[0],comargs);
```

Where the comargs[0] contains the name of the command, comargs starts with the command, and then all the arguments and the NULL character.

The *execv* function replaces your program with a new program. (The *system* function runs a new program, then continues running your program. You should not use system in the assignment.)

■ How do I use strtok()

strtok() can help you parse the commands you get from input. If you choose to use it, the syntax is:

```
char *strtok(char *s1, const char *s2);
```

Example:

string is “abcd efgh || ijk lmn”

```
number =0;
command[0]=strtok(string,"||");
while((ptr=strtok(NULL, "||")) != NULL)
{
number++;
command[number]=ptr;
}
```

Then command[0] will contain “abcd efgh”, command[1] will contain “ijk lmn”.

■ How do I do error handling?

You need to check the return value of a system call. E.g.: if the return value of -1 indicates an error, you should have a block that handles it. The simplest way is to print out an error and then exit the program.