1 Introduction

This assignment has the following goals:

1. To dust off your C programming skills;
2. To implement an OS command interpreter (shell);
3. To use some system calls (such as fork() and exec()).

This is an *individual* assignment. This assignment requires you to build a simple shell. Everything you do in this warm-up assignment is at the user-level (outside of the operating system kernel).

2 Unix: The Manual

All Unix systems come with an online manual. The "man" command is used to look up pages within the manual. For example, to look at the manual page for the "chdir" system call, type:

```
man chdir
```

Sometimes, the same item appears in separate sections of the manual pages. System calls are documented in Section 2. You can specify a section number before the name of the item for which you documentation. For example, to lookup the `chdir()` system call in Section 2, use the following command:

```
man 2 chdir
```

Section 1 contains information on commands, Section 2 contains information on system calls, and Section 3 contains information on C and Unix library functions.

3 Template Code

In order to level the playing field between those who have and have not taken Computer Organization (CSC 252) at the University of Rochester, you may consult and use the skeleton code (sh-skeleton.c) provided on Blackboard. There is no compulsion to use this skeleton. You can certainly create your own organization and reference this skeleton to determine the kinds of calls and structures you will need to create.

For students less experienced with Unix, there is a working shell on the course web site; it is an executable for Linux x86 platforms. Note that this shell is not a complete solution for this assignment. It lacks support some features that you are required to implement.

Please note that the skeleton code is not well written. Students can use it as a starting point for their shells but should not assume that the naming conventions and commenting style are indicative of the good programming style expected in this course.

4 The Shell

4.1 Unix shells

The OS command interpreter is the program with which people interact in order to launch and control programs. On Unix systems, the command interpreter is often called a *shell*; it is a user-level program that gives people a command-line interface for launching, suspending, and killing other programs. `sh`, `ksh`, `bash`, `csh`, and `tcsh` are all examples of Unix shells. You use a shell like this every time you log into a Linux machine at a URCS computer lab and bring up a terminal. It might be useful to look at the manual pages of these shells. For example, type "man csh" to get information on the csh C shell.

The most rudimentary shell is structured as the following loop:

1. Print out a prompt (e.g., "CSC2/456Shell$ ");
2. Read a line from the user;
3. Parse the line into the program name and an array of parameters;
4. Use the fork() system call to spawn a new child process;
5. The child process then uses the exec() system call (or one of its variants) to launch the specified program while the parent process (the shell) uses the wait() system call (or one of its variants) to wait for the child to terminate. Once the child (the launched program) finishes, the shell repeats the loop by jumping to step 1.

Although most commands people type on the shell prompt are the names of other Unix programs (such as `ps` or `cat`), shells also recognize some special commands (called *internal commands*) that are not program names. For example, the `exit` command terminates the shell, and the `cd` command changes the current working directory. Shells directly make system calls to execute these commands instead of forking a child process to handle them.

4.2 Basic Shell Requirements

Your job is to implement a very primitive shell that knows how to launch new programs in the foreground and the background. It should also recognize a few internal commands. More specifically, it should support the following features.

It should recognize the internal commands `exit`, `cd`, and `jobs`. The `exit` command should use the exit() system call to terminate the shell. cd uses the chdir() system call to change to a new directory. The `jobs` command should print the jobs in the background and should print a) an index number for the command line that was run in the background and b) the command line that was placed into the background. The format should be:

```
<id>: <program name> <arg1> <arg2> .... <argN>
```

If the command line does not indicate any internal commands, it should be in the following form:

```
<program name> <arg1> <arg2> .... <argN> [&]
```

Your shell should invoke the program, passing it the list of arguments on the command line. The shell must wait until the started program completes unless the user runs it in the background (with &). To allow users to pass arguments, you need to parse the input line into words separated by whitespace (spaces and tab characters). You might try to use strtok_r() for parsing (check the manual page of strtok_r() and Google it for examples of using it). In case you wonder, strtok_r() is a user-level library routine and not a system call; it is implemented completely outside the operating system kernel. To make the parsing easy for you, you can assume the '&' token (when used) is separated from the last argument with one or more spaces or tab characters.

The shell runs programs using two core system calls: fork() and execvp(). Read the manual pages to see how to use them. In short, fork() creates an exact copy of the currently running process, and is used by the shell to spawn a new process. The execvp() call is used to overload the currently running program with a new program, which is how the shell turns a forked process into the program it wants to run. In addition, the shell must wait until the previously started program completes unless the user runs it in the background (with &). This is done with the wait() system call or one of its variants (such as waitpid()). All these system calls can fail due to unforeseen reasons (see their manual pages for details). You should check their return status and report errors if they occur.

No input the user gives should cause the shell to exit (except when the user types exit or Ctrl+D). This means your shell should handle errors gracefully, no matter where they occur. Even if an error occurs in the middle of a long pipeline, it should be reported accurately and your shell should recover gracefully. In addition, your shell should not generate leaking open file descriptors. Hint: you can monitor the current open file descriptors of the shell process through the /proc file system.

4.3 Adding Pipes

Your shell must support pipes. Pipes allow the standard inputs (stdins) and standard outputs (stdouts) of a list of programs to be concatenated in a chain. More specifically, the first program's stdout is directed to the stdin of the second program; the second program's stdout is directed to the stdin of the third program; and so on so forth. Multiple piped programs in a command line are separated with the token "|". A command line will therefore have the following form:

```
<program1> <arglist1> | <program2> <arglist2> | ... | <programN> <arglistN> [&]
```

Try an example like this: pick a text file with more than 10 lines (assume it is called textfile) and then type

```
cat textfile | gzip -c | gunzip -c | tail -n 10
```

in a regular shell or in the working shell we provide. Pause a bit to think about what it really does. Note that multiple processes need to be launched for piped commands and all of them should be waited on in a foreground execution. The pipe() and dup2() system calls will be useful; consult their manual pages for details.

4.4 Controlling Terminal

When a terminal reads various control sequences from the keyboard, it sends appropriate signals to the processes associated with that terminal. For example, typing Ctrl-C sends a SIGINT signal while Ctrl-Z sends a SIGTSTP signal. The terminal is known as the *controlling terminal* for those processes.

When a process is run in the background, it should not receive these signals. Likewise, when it is brought back into the foreground, it is attached to the controlling terminal and should receive these signals.

You must add the bg command (which moves a foreground process into the background) and the fg command (which moves a background process into the foreground). Moving processes into/out of the foreground should change their association with the controlling terminal properly.

Details on controlling terminals can be found in W. Richard Steven's book *Advanced Programming in the Unix Envionment;* this book is available online via the library.

5 Administrative Policies

5.1 Permissible Programming Languages

C is the only choice for this assignment and all later programming assignments. Most existing operating system kernels (Linux and other Unix variants) themselves are written in C; the remaining parts are written in assembly language.

Higher-level languages (Java, Perl, ...), while possible, present additional challenges due to their use of automatic memory management and their general inability to address memory directly.

5.2 Turn-in

You are to create a directory named `mp1` and within this directory place the following items:

1. The source code for your shell.
2. A Makefile that will compile your shell. The name of the generated executable should be `shell`. Typing "make" within the `mp1` directory should compile your shell.

You should submit your files as a single GNU zipped tar archive named mp1.tar.gz. When running the following command, we should see a directory named `mp1` with the above files:

```
gunzip --stdout mp1.tar.gz | tar -xvf -
```

If you are unfamiliar with the tar and gzip/gunzip programs, please consult their man pages. You will submit your GNU zippped tar archive via Blackboard.

5.3 Code Comments

As you will learn when you start reading operating system kernel code, *too many* programmers write too few comments. It helped them code faster at the expense of your time and the time of all future kernel programmers to come. Therefore, to help end this cycle of misery, you are required to transcend current industry norms and comment your code.

Any non-trivial function should have a comment before the function describing its purpose, the meaning of its inputs, the meaning of its outputs, and the interpretation of its return value. Even simple functions that return `true` and `false` may need explanation of what those two values mean. Additional comments should exist throughout the code as necessary to explain what the code is doing and why.

In short, if the graders have difficulty understanding your code, then your commenting does not suffice.

5.4 Grading

Please note that students that took CSC 252 at the University of Rochester may only use their own solutions as a starting point (if they choose to do so) and clearly indicate what changes were made in a README file included with the submission.

You will be penalized heavily for failure to pass tests. The tests are not identical to those used in CSC 252. You should test your code thoroughly.

- *30 points:* your shell supports foreground and background execution and correctly reaps child processes (i.e., it does not leak system resources by creating zombie processes)
- *20 points:* your shell supports pipes and does not leak open file descriptors
- *10 points:* your shell supports the exit, cd, and jobs internal commands
- *10 points:* your shell supports correct assignment of the controlling terminal to the foreground job
- *10 points:* your shell handles errors gracefully and does not crash
- *10 points:* your code is properly commented
- *10 points:* your submission follows the submission requirements