CSc 360: Operating Systems (Spring 2015)

Assignment 1 (P1): A Realistic Shell Interpreter (RSI)

Spec Out: Jan. 7, 2015
Code Due: 2:30 pm, Jan. 30, 2015

1 Introduction

In this assignment you will implement a realistic shell interpreter (RSI), interacting with the operating system using system calls. The RSI will be very similar to the Linux shell `bash`: it will support foreground execution of programs, the ability to change directories, and background execution.

You should implement your solution in C. Your work will be tested on machines in ECS 242 (the Linux drop-in lab) or on `linux.csc.uvic.ca`.

Note: `linux.csc.uvic.ca` is a *particular* machine at the UVic Department of Computer Science. It does *not* mean "any Linux machine" at UVic. Even more importantly, it does *not* mean any "Unix-like" machine, such as a Mac OS X machine—many students have developed their programs for their Mac OS X laptops only to find that their code works differently on `linux.csc.uvic.ca` resulting in a *substantial* loss of marks.

You can remote access `linux.csc.uvic.ca` by `ssh username@linux.csc.uvic.ca`. SSH clients are available for a wide variety of operating systems including Linux, Mac OS and Windows.

Be sure to study the `man` pages for the various systems calls and functions suggested in this assignment. The system calls are in Section 2 of the `man` pages, so you should type (for example):

```
$ man 2 waitpid
```

2 Schedule

In order to help you finish this programming assignment successfully, the schedule of this assignment has been synchronized with both the lectures and the tutorials. There are two tutorials arranged during the course of this assignment. The final delivery and demo date is Jan 30, 2015.

Date	Tutorial	Milestones
Jan. 16	P1 spec go-through, design hints, system calls	design and code skeleton
Jan. 23	more on system programming, help, and testing	alpha code done
Jan. 30	—	Final delivery and test

3 Requirements

3.1 Basic Execution (30%)

Your RSI shell shows the prompt

```
linux.csc.uvic.ca:/home/user$ ./rsi
RSI: /home/user >
```

for user input. The prompt includes the current directory name in absolute path, e.g., `/home/user`.

Using `fork()` and `execvp()`, implement the ability for the user to execute arbitrary commands using your shell program. For example, if the user types:

```
RSI: /home/user > ls -l /usr/bin
```

your shell should run the `ls` program with the parameters `-l` and `/usr/bin` — which should list the contents of the `/usr/bin` directory on the screen.

Another example, `RSI: /home/user > mkdir test` will create a directory with the name `test` under `/home/user`, i.e., a directory `/home/user/test` is **actually** created in the host file system.

Note: The example above uses 2 arguments. We will, however, test your RSI shell by invoking programs that take more than 2 arguments.

A well-written shell should support as many arguments as given on the command line.

3.2 Changing Directories (30%)

Using the functions `getcwd()` and `chdir()`, add functionality so that users can:

- change the current working directory using the command `cd`
- print the current working directory using the command `pwd`

The `cd` command should take exactly one argument — the name of the directory to change into. The special argument `..` indicates that the current directory should "move up" by one directory.

That is, if the current directory is `/home/user/subdir` and the user types:

```
RSI: /home/user/subdir > cd ..
```

the current working directory will become `/home/user`.

The special argument ∼ indicates the home directory of the current user. If `cd` is used without any argument, it is equivalent to `cd ~`, i.e., returning to the home directory, e.g., `/home/user`.

The `pwd` command takes no arguments.

Note: There is a program named `pwd`. You can either use `fork()` and `execvp()` to execute it, or use the system call `getcwd()`.

Note: There is no such a program called `cd` in the system that you can run directly (as you did with `ls`) and change the current directory of the **calling** program, even if you created one. You need to use the function `chdir()`.

3.3 Background Execution and Process Management (30%)

Many shells allow programs to be started in the background—that is, the program is running, but the shell continues to accept input from the user.

You will implement a simplified version of background execution that supports executing processes in the background. The maximum number of background processes is not limited.

If the user types: `cat foo.txt &` or `nohup cat foo.txt &`, your RSI shell will start the command `cat` with the argument `foo.txt` in the background. That is, the program will execute and the RSI shell will also continue to execute and give the prompt to accept more commands. Note that if you run a program in the background with `&`, the background process will be automatically terminated when you log out. To avoid this, use `nohup` to run a command immune to hangups.

The command `ps` will have the RSI shell to report a list of all the processes currently running. Note that `ps` has different options to display various information regarding the processes. Check `man ps` for details.

Related to process management, in your RSI shell:

1. The command `ps -e` display all processes using standard syntax.
2. The command `kill` *pid* will send the `TERM` signal to the job with process ID *pid* to terminate that job.

See the `man` page for the `kill()` system call for details.

Your RSI shell must indicate to the user when background jobs have terminated. Read the man page for the `waitpid()` system call. You are suggested to use the `WNOHANG` option.

To summarize, your RSI shell should at least support the following commands: `cd`, `mkdir`, `rmdir`, `ls`, `ps`, `kill`, and `nohup`. Your RSI shell can run any "external" commands/programs provided by the system (e.g., `ls`, `mkdir`, `rmdir`, etc).

If the user types an unrecognized command supported by neither internal nor external commands, an error message is given by the RSI, e.g.,

```
RSI: /home/user > ttest
RSI: ttest:  command not found
```

4 Odds and Ends

4.1 Compilation

You've been provided with a `Makefile` that builds the sample code (in `p1s.tar.gz`). It takes care of linking-in the GNU `readline` library for you. The sample code shows you how to use `readline()` to get input from the user, only if you choose to use the `readline` library.

4.2 Submission

Submit a `tar.gz` archive named `p1.tar.gz` of your assignment to connex assignment 1.

You can create a `tar.gz` archive of the current directory by typing:

```
tar zcvf p1.tar.gz *
```

Please do not submit `.o` files or executable files (`a.out`) files. Erase them before creating the archive.

4.3 Helper Programs

4.3.1 `inf.c`

This program takes two parameters:

tag: a single word which is printed repeatedly

interval: the interval, in seconds, between two printings of the tag

The purpose of this program is to help you with debugging background processes. It acts a trivial background process, whose presence can be "felt" since it prints a tag (specified by you) every few seconds (as specified by you). This program takes a tag so that even when multiple instances of it are executing, you can tell the difference between each instance.

4.3.2 `args.c`

This is a very trivial program which prints out a list of all arguments passed to it.

This program is provided so that you can verify that your RSI shell passes *all* arguments supplied on the command line—often, people have off-by-1 errors in their code and pass one argument less.

4.4 Code Quality 10%

We cannot specify completely the coding style that we would like to see but it includes the following:

1. Proper decomposition of a program into subroutines (and multiple source code files when necessary)—A 500 line program as a single routine won't suffice.
2. Comment—judiciously, but not profusely. Comments also serve to help a marker, in addition to yourself. To further elaborate:
   (a) Your favorite quote from Star Wars or Douglas Adams' Hitch-hiker's Guide to the Galaxy does not count as comments. In fact, they simply count as anti-comments, and will result in a loss of marks.
   (b) Comment your code in English. It is the official language of this university.
3. Proper variable names—`leia` is not a good variable name, it never was and never will be.
4. Small number of global variables, if any. Most programs need a very small number of global variables, if any. (If you have a global variable named `temp`, think again.)
5. **The return values from all system calls and function calls listed in the assignment specification should be checked and all values should be dealt with appropriately.**

If you are in doubt about how to write good C code, you can easily find many C style guides on the 'Net. The Indian Hill Style Guide is an excellent short style guide.

4.5 Plagiarism

This assignment is to be done individually. You are encouraged to discuss the design of your solution with your classmates, but each person must implement their own assignment.

Your markers may submit the code to an automated plagiarism detection program. We add archived solutions from previous semesters (a few years worth) to the plagiarism detector, in order to catch "recycled" solutions.

The End
