CSc 360: Operating Systems (Fall 2010)

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

Spec Out: September 22, 2010
Code Due: October 5, 2010

1 Introduction

In P0, you have implemented a simple shell interpreter (SSI), good job! But you soon find that the SSI does not actually interact with the operating system; instead, it just emulates some of the common shell commands (it was intended to be a warming up assignment anyway ;-). In this assignment you will implement a more realistic shell interpreter (RSI), interacting with the real 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 can implement your solution in C or C++ and build upon **your own** P0 (you can reuse **your own** P0 code when you see fit). Your work will be tested on `u-*.csc.uvic.ca` in ECS242.

Note: You can remote access the Linux computers in ECS242 Linux Teaching Lab using `ssh`. For a list of Linux computers (u-*.csc.uvic.ca) in that lab, please check
`http://www.csc.uvic.ca/~labspg/ecs242servers.html`

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 on time 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.

Date	Tutorial	Milestones
September 24	P1 spec go-through, design hints, system calls	design and code skeleton
October 1	testing, packing and extra help	close-to-finish deliverable

3 Requirements

3.1 Basic Execution (5 marks)

Your RSI shell shows the prompt

```
u-arch.csc.uvic.ca:/home/user/p1 $ ./rsi
RSI: /home/user/p1 $
```

for user input. The prompt includes the current directory name in absolute path, e.g., `/home/user/p1`.

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/p1 $ 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 in details on the screen.

Another example, `RSI: /home/user/p1 $ mkdir test` will create a directory with the name `test` under `/home/user/p1`, i.e., a directory `/home/user/p1/test` is **actually** created in the host file system, which is very different from the SSI in P0.

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 (5 marks)

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

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

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`.

To get the user's home directory, you need to use `getenv()` to obtain environment variable `HOME`. This home directory is not necessary the directory where `./rsi` was initially executed.

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 have to use the system call `chdir()`.

3.3 Background Execution (10 Marks)

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: `bg 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.

The command `bglist` will have the RSI shell display a list of all the programs currently executing in the background, e.g.,:

```
[1] 123:  running /home/user/p1/foo -i
[2] 456:  stopped /home/user/p1/foo -p
[3] 789:  running /home/user/p1/foo -q
Total background job(s):  3
```

In this case, there are 3 background jobs, all running the program `foo`, the first one with process ID `123`, the second one with `456` but stopped (see below), and the third one with `789`. Note that their running parameters are also listed.

In your RSI shell:

1. The command `bgkill` *pid* will send the `TERM` signal to the job with process ID *pid* to terminate that job.

2. The command `bgstop` *pid* will send the `STOP` signal to the job *pid* to stop (temporarily) that job (e.g., the process with ID `456` in the example above).

3. The command `bgstart` *pid* will send the `CONT` signal to the job *pid* to **re**-start that job (which has been previously stopped).

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

Your RSI shell must indicate to the user when the use interacts with RSI that some background jobs have terminated. For example, when the user inputs `ls`

```
RSI: /home/user/p1 $ ls
foo
[3] 789:  finished /home/user/p1/foo -q
Total background job(s):  2
```

indicating the process with ID `789` has finished already when `ls` finishes. Read the man page for the `waitpid()` system call. You are suggested to use the `WNOHANG` option.

Therefore, your RSI shell should support 6 internal commands: `cd`, `bg`, `bglist`, `bgkill`, `bgstop`, and `bgstart`. 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
```

`man execvp` to see how `execvp()` searches an executable file if the specified file name does not contain a slash (/) character. The search path is the path specified in the environment by the `PATH` variable. It also explains the return or error code.

If the user types in a recognized command but with wrong arguments, you need to give an error message, e.g.,

```
RSI: /home/user $ bgkill foo
RSI: bgkill:  foo:  argument must be process ID
RSI: /home/user $ bgkill 456
[2] 456:  finished /home/user/p1/foo -p
Total background job(s):  1
```

4 Bonus Features

Although more realistic, only a simplified shell with limited functionality is required in this assignment. However, students have the option to extend their design and implementation to include more features in a regular shell or even a remote shell (e.g., capturing and redirecting program output, handling many remote clients at the same time, etc).

If you want to design and implement a bonus feature, you should contact the course instructor for permission **one week before the due date**, and clearly indicate the feature in the submission of your code. The credit for correctly implemented bonus features will not exceed 25% of the full marks for this assignment (the full marks weigh 10% in the final grade).

5 Odds and Ends

5.1 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.

5.2 Submission

Submit a `tar.gz` archive named `p1.tar.gz` of your assignment through http://connex.csc.uvic.ca

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.

5.3 Helper Programs

5.3.1 `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.

This program considerably simplifies the programming of the part of your RSI shell which deals with re-starting, stopping, and killing programs.

5.3.2 `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.

5.4 Code Quality

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.

5.5 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 will 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
