CSc 360: Operating Systems (Fall 2019)

Programming Assignment 1
P1: A Simple Shell Interpreter (SSI)

Spec Out: Sept 13, 2019
Code Due: Oct 4, 2019

1 Introduction

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

You can implement your solution in C or C++, and C is highly recommended. Your work will be tested on `linux.csc.uvic.ca`, to which you can remote login by `ssh`. You can also access Linux computers in ECS labs in person or remotely by following `https://itsupport.cs.uvic.ca/`

Be sure to test your code on `linux.csc.uvic.ca` before submission. 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.

Be sure to study the `man` pages for the various systems calls and functions suggested in this assignment. These functions 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 three tutorials arranged during the course of this assignment.

Date	Tutorial	Milestones
Sept 17/19/20	P1 spec go-through, design hints, system calls	design and code skeleton
Sept 24/26/27	system calls, system programming and testing	code almost done
Oct 1/3/4	final testing and last-minute help	final deliverable

3 Requirements

3.1 Basic Execution (5 marks)

Your SSI shows the prompt

```
SSI: username@hostname:  /home/user >
```

for user input. The prompt includes the current directory name in absolute path, e.g., `/home/user`. You can use `getcwd()` to obtain the current directory, `getlogin()` to get username, and `gethostname()` to find the name of the host running the program.

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

```
SSI: username@hostname:  /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.

Note: The example above uses 2 arguments. We will, however, test your SSI 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`

Note that SSI always shows the current directory at prompt.

The `cd` command should take no more than one argument—the name of the directory to change into—and ignore all others. 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:

```
SSI: username@hostname:  /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`.

Q: how do you know the user's home directory location?
H: from the environment variable with `getenv()`.

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 (why?)—i.e., you have to use the system call `chdir()`.

3.3 Background Execution (5 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 SSI shell will start the command `cat` with the argument `foo.txt` in the background. That is, the program will execute and the SSI shell will also continue to execute and give the prompt to accept more commands.

The command `bglist` will have the SSI shell display a list of all the programs, including their execution arguments, currently executing in the background, e.g.,:

```
123:  /home/user/a1/foo 1
456:  /home/user/a1/foo 2
Total Background jobs:  2
```

In this case, there are 2 background jobs, both running the program `foo`, the first one with process ID `123` and execution argument `1` and the second one with PID `456` and argument `2`.

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

```
SSI: username@hostname:  /home/user/subdir > cd
456:  /home/user/a1/foo 2 has terminated.
SSI: username@hostname:  /home/user >
```

Q: how do you make sure your SSI has this behavior?

H: check the list of background processes every time processing a user input.

4 Bonus Features

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 a remote shell (e.g., kill/pause/resuming background processes, 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 by email for permission one week before the due date, and clearly indicate the feature in the submission of your code. The credit for approved and correctly implemented bonus features will not exceed 20% of the full marks for this assignment.

5 Odds and Ends

5.1 Compilation

You will be provided with a `Makefile` that builds the sample code. 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 `readline` library.

5.2 Submission

Submit a `tar.gz` archive named `p1.tar.gz` of your assignment through `connex`, with the `Makefile`

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 SSI shell. You can find the running process by `ps -ef` and `kill -9` a process by its process ID `pid`.

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 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 — A 500 line program as a single routine won't suffice.
2. Comment—judiciously, but not profusely. Comments serve to help a marker. 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 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, including those on github.

The End
