Assignment #1

CSC 360: Operating Systems (Fall 2007)

Start Date: September 10, 2007
Due Date (Deliverable 1): September 24, 2007
Due Date (Deliverable 2): October 5, 2007

1 Introduction

In this assignment you will implement a remote shell program.

Deliverable 1 The basic shell will be similar to the Unix shell `bash`: It will support foreground execution of programs, the ability to change directories, and background execution.

Deliverable 2 The shell must also be able to function as a server: it should be able to accept commands sent to it from a client shell.

You may implement your solution in C or C++. Your solution will be tested 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 have been warned.

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 Requirements for Deliverable 1

2.1 Basic Execution (5 marks)

Your shell shows the prompt

```
shell>
```

for user input.

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

For example, if the user types:

```
shell> 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 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.

2.2 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`
- 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/djp/subdir` and the user types:

```
cd ..
```

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

The `pwd` command takes no arguments.

Note: There is a program named `pwd`. Do not use `fork()` and `execvp()` to execute it. You are required to use the system call `getcwd()`.

2.3 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 processing that supports a fixed number of processes (in this case, 5) executing in the background.

If the user types: `bg cat foo.txt` your shell will start the command `cat` with the argument `foo.txt` in the background. That is, the program will execute and the shell will also continue to execute.

The command `bglist` will display a listing of all the commands currently executing in the background, similar to:

```
0:  /home/djp/a1/foo
1:  /home/djp/a1/foo
Total Background jobs:  2
```

In this case, there are 2 background jobs, both running the program `foo`.

In your shell:

1. The command `bgkill` *n* will send the `TERM` signal to job *n* to terminate that job.
2. The command `bgstop` *n* will send the `STOP` signal to job *n* to stop (temporarily) that job.
3. The command `bgstart` *n* will send the `CONT` signal to job *n* to **re**-start that job (which has been previously stopped).

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

Your shell must indicate to the user when background jobs have terminated. Read the man page for the `waitpid()` system call. I suggest using the `WNOHANG` option.

3 Requirements for Deliverable 2 (10 marks)

You will modify your shell to work as a remote shell. Essentially, the program written for deliverable 1 will be modified into *two* parts—a server shell which executes commands, *and* a client shell which sends commands to the servers shell. Communication between the client shell and the server shell will be done via the "sockets" library.

Sockets are a form of IPC (InterProcesss Communication) provided by the UNIX operating system. A socket is one end-point of a connection between two processes. Hence, each connection has two sockets:

1. A "client" socket: which initiated the connection.
2. A "server" socket: which accepted the connection.

Each socket typically has a "name" consisting of a IP address *and* a 16-bit port number.

Much more information will be given in the tutorial—so please attend.

3.1 Invocation

The shell must accept a single command line parameter:

`--server`, which instructs the shell to act as a server which waits for the client to send a command and then executes it, on behalf of the client. This "wait-get-execute" sequence is performed repeatedly until the command `exit` is received, at which point the server process terminates.

`--client`, which instructs the shell to acts as a client, which simply sends the commands typed by the user (at the keyboard) to the server shell.

Note: The shell server is only required to support one shell client.

3.2 Server Requirements

The C library/kernel function calls that you will need to create a server process are:

`socket`, to create a "server" socket—a socket which is capable of accepting connections.

`gethostbyname`, which is used to obtain the IP address of a named host.

`bind`, which binds an IP address to a server-socket.

`listen`, which configures a server-socket to listen for connections.

`accept`, which accepts a connection attempt made by a client, and returns a file description. This file description can be read with functions such as `read`, `write`, and (with a little massaging) `fread` and `fwrite`. Essentially, after an `accept` function, reading from and writing to a socket is no different than from/to a file.

When a server receives a command (typically via `fgets`) it will:

1. Perform the command exactly as if the user had entered the command locally at the keyboard.
2. Write the messages (that it would write to the screen), back to the client via the socket connection.
3. Write the message `END OF MESSAGE` back to the client.

Note: The output of any programs invoked does *not* have to be captured and sent back to the client.

3.3 Client Requirements

The C library/kernel function calls that you will need to create a client process are:

`socket`, to create a "client" socket—a socket which is capable of making connections.

`gethostbyname`, which is used to obtain the IP address of a named host—the remote host which you want to connect to.

`connect`, which connects the client-socket to the named host.

When a client receives a command (typed at the keyboard) it will:

1. Send the command to the server via the socket interface.
2. Print all messages sent from the server until the message `END OF MESSAGE` is received.

4 Bonus Features

Only a simple 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/remote shell (e.g., handling many clients at the same time, capturing and redirecting remote program output, etc).

If you want to design and implement a bonus feature, you should contact the course instructor for permission before the due date of Deliverable 1, and clearly indicate the feature in the submission of Deliverable 2. The credit for correctly implemented bonus features will not exceed 20% of the full marks for this assignment.

5 Odds and Ends

5.1 Compilation

You've been 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.)

5.2 Submission

Submit a `tar` archive named `assign1.tar` of your assignment using the automated submission program at

http://www.csc.uvic.ca/~submit/index.cgi

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

```
tar cvf assign1.tar *
```

Please do not submit `.o` files or executable files (`.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 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 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 favourite 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.

The End
