Lab 3: Script Interpretation

February 2, 2016

In this lab, knowledge of how to create new processes and have those new processes execute other programs will be extended, along with exploration of a few extra topics. Now, not only will the child processes execute other programs, the return value of the programs will need to be evaluated by the parent. Those extra topics will be how to write simple shell scripts using the Bourne Again SHell (Bash), and how to implement some of Bash's built-in functions. Again, some familiarity with how to open a file, read from it, and parse what was read from the file will be needed for this lab.

1 Handling Child Process Return Values

In the previous lab, when a program is run on a child process, the parent process waits on the child process; and once the program on the child process terminates, the parent process doesn't evaluate the return value in the `status` variable, as it was not necessary. This is no longer the case for this lab. Recall for the previous lab how the parent process waits for the child process:

```
...
int status = 0;
...
pid_t child;
child = fork();
if(child == 0){
    //create and populate an argArray
    ...
    execvp(nameOfProg, argArray);
    perror("exec has encountered an error trying to execute nameOfProg");
    return -1;
}else if(child > 0){
    //the parent waits on the child process to terminate
    wait(&status);
    return 0;
}else{
    perror("fork");
    return -1;
}
```

When the child terminates, the return value of the child program, along with the status of the child process is stored in the `status` variable. To determine if the child process terminated normally, encountered an error, or received a signal, there are macros that can parse out that information from the variable `status`. For a list of those macros and what they do, consult `man 2 wait`.

2 A Quick and Simple Guide to Bash Scripting

The expectation of the student for this lab is that the student is at the very least familiar with the use of the Bash shell[1], if not proficient with its use. With this knowledge, writing a simple Bash script is quite simple. For example, here is a 'hello world' Bash script:

```
#! /bin/bash
# written by Kris Hall
#
echo "Hello World"
```

Saving this code into a file called `hello_world.sh` and run it either by typing the command:

```
$ bash hello_world.sh
```

or make the script executable, and run it as such:

```
$ chmod +x hello_world.sh
$ ./hello_world.sh
```

Include the output of running the above script in your lab report.

3 Script file format

With this simple example of `hello_world.sh`, some explanation of what each line in the file `hello_world.sh` does and how this can be extended is provided below.

3.1 First line

The first line of a script file should tell what type of file it is, and which program should interpret it. For example, shell scripts that start with

```
#! /bin/bash
```

or

```
#! /bin/sh
```

are shell scripts meant to be interpreted by Bash and SH respectively. Other scripts can include `#! /bin/python`, `#! /bin/perl`, etc. When a script is executed on the command line the shell will search for the correct interpreter to start using this first line. Therefore with the above example, calling

```
$ ./hello_world.sh
```

is interpreted as

```
$ /bin/bash hello_world.sh
```

[1] The terms shell and terminal are often used interchangeably; however, they are in fact different things. The terminal provides an interface to type commands into the computer. A shell such as Bash is a program which interprets and executed the commands.

3.2 Comments

Any line starting with a `#` and not followed by an `!` is considered a comment line and ignored by Bash. Comments can appear anywhere in the file. Note that most interpreters will not accept partial line comments, e.g.

```
#! /bin/bash
echo "hello world" # print hello world
```

Instead the correct way to write this for maximum portability would be

```
#! /bin/bash
# Print hello world
echo "hello world"
```

3.3 Commands

A command is anything the script is to execute. Script commands are identical to typing commands on the command line. For example the following set of commands:

```
$ git pull upstream/master
$ git status
```

could be replaced with a single shell script, which will be called update-git.sh:

```
#! /bin/bash
git pull upstream/master
git status
```

and then executed using the signal command `./update-git.sh`. Another example of a command would be the line `echo "Hello World"` in the script file `hello_world.sh` created earlier.

Though not very important to the use of the Bash shell, this information will be useful for portability purposes of the script and for the implementation of the task for the lab. So far, commands have been treated as something that will work for every interpreter. This may not be true for all commands that are in the scripts. The set of commands that may not always work for every interpreter are known as 'builtin' commands, or commands that are built-in to the interpreter. These built-in commands are potentially different for different interpreters. The other set of commands are aliases, functions, executables, and keywords; of which we are mainly interested in executables. To check if a command is a builtin or not, simply type into Bash:

```
$ type commandToCheck
```

This will return the type of the command in `commandToCheck`. If a command is a built-in command, `type` will return 'builtin'. If a command is an executable, `type` will either return 'hashed' if the executable has been executed at least once during the session, or it would return the path to the executable if it has not.

Use the `type` command to answer the following questions

- What is the type of the `cd` command?
- What is the type of the `ls` command?
- what is the type of the `python` command?

3.4 Command Line Arguments

Command line arguments can be passed to a shell script just like any c program. The number of command line arguments passed is stored in a variable named `$#` and each argument is stored in `$1`, `$2`,..., `$n`. The variable `$0` is the name of the program by convention. Here is an example:

```
#! /bin/bash
echo $0 'was called with' $# 'arguments'
```

3.5 Further Information

There are far more capabilities to Bash scripting than discussed here. Examples include conditional if statements, loops, and math. An excelent resource to learn more is http://linuxcommand.org/lc3_writing_shell_scripts.php

4 How to Set and Use Environment Variables

When a shell is started, it has to keep track of a lot of settings for resource access and properties. How it keeps track of all of this is through what is called an environment. This is a list of variables that hold all sorts of information for the correct execution of the shell. An interesting property of the environment is that any child shell or process of the shell will inherit the variables when started from the parent shell. To list all environment variables that the shell has access to, the following command is used:

```
$ printenv
```

For better readability use:

```
$ printenv | less
```

The output should show some familiar variables, such as `PATH`, `SHELL`, and `HOME`. In the event that printing out the environment variables are insufficiently interesting, creation of personalized environment variables can be performed as well. To create a new environment variable, use the following command:

```
$ export VAR_TEST=valueForVar
```

This command will place `VAR_TEST` in the list of environment variables with the value of `valueForVar`. Note that `valueForVar` will be interpreted as a string, regardless of what its value is. To use a variable(or more specifically, expand it), place a `$` before the variable name, as such:

```
$ echo $PATH
```

This would expand the `PATH` variable. An example that some students may be familiar with in this regard is appending a directory to the `PATH`:

```
$ export PATH=$PATH:path/to/new/executable/directory
```

This example would add the path `path/to/new/executable/directory` to the PATH variable, allowing the user to access the new executable installed in the directory without typing out the full path to it.

In your lab report research and explain briefly how these environment variables were exploited causing the *Shellshock Bug*.

5 Tasks for this lab

The task for the student for this lab is to implement a script interpreter similar to Bash called the Cyclone Advanced SHell (CASH) that can interpret the bash commands that were outlined in the previous section. The idea is to use all of the information and system calls covered or reviewed in the previous sections to create something that functionally works like a shell script interpreter. The exact specification of the interpreter is as follows:

- The interpreter will have one argument, the script that it will interpret.
- If there is no argument, or the argument is not a script file, it should print out usage information before exiting
- To determine if a file is not a script file, check the first line of the file to see if it is `#!  /bin/cash`
- If any errors are encountered during program execution, the program should print out error information and exit.
- No segmentation faults should be allowed to occur.
- The interpreter should be able to support the following types of commands in the script:

a) `cd`
b) `pwd`
c) `export`
d) `echo`
e) any program that is either user created (if given the path to it), or can be found in one of the directories specified in `$PATH`[for example, the program `gedit` should be able to be executed from the interpreter, along with `anyopen` or even the interpreter that is the task for this lab]

- The interpreter should be able to ignore all text following a comment
- The interpreter should be able to run any command in the background(if the command has a `&` at the end of it), and when the background process is complete, a line should be printed in the following format:

```
[process_id] exit_status        command
```

6 Extra Credit

Extra credit will be given for implementing additional features from Bash into your interpreter. Here are some examples of features you might choose to implement:

- Aliasing commands
- Variables
- Conditions

Make sure to include a readme file explaining any additional features you implemented and an example script that shows them working.

7 Hints

These man pages may be of interest to the students:

- `man 3 getenv`
- `man 3 setenv`
- `man 2 chdir`
- `man 3 getcwd`

8 License