CS 3841 – Teeny Tiny SHell (TTSH)

1 Introduction

A shell is a program that provides a traditional text-based interface to an operating system. In Linux, a user types commands into the shell to be executed similar to the command prompt (cmd.exe) on Windows. The purpose of this lab is to design, code, and test a program that will create a command prompt allowing user to execute commands. It is intended to familiarize yourself with the system calls needed to create processes on Linux.

2 References

There are many different shells available for Linux, arguably the most popular is 'bash' which stands for the Bourne-Again SHell. The shell you will be developing for this lab will work similarly to 'bash' but will have fewer features. For more information about the 'bash' shell, you can check out the Wikipedia page:

https://en.wikipedia.org/wiki/Bash_(Unix_shell)

For this lab, you will be using the fork() system call to create child processes, then running exec() to run the commands specified by the user, finally waiting for the commands to finish executing using the wait() system call. Before you start, it might be helpful to read the 'man' pages for these system calls.

- `man 2 fork`
- `man 2 exec`
- `man 2 wait`

3 The Exercise

The steps for a typical shell are as follows:

1. Print out a prompt to the user – this may include the user's login name followed by their current working directory. Shells typically allow the user to configure this through a settings file. Our shell will use the same prompt every time.
2. Read the command string from the user
3. Invoke the fork() system call to create a child process to run the command
4. Invoke the exec() system call in the child process to replace its address space with the command to run
5. Invoke the wait() system call in the parent process to wait for the child process (running the command) to finish executing
6. Go back to 1

Shells typically run in a loop executing commands and then waiting for more work to do. For this lab you will be developing a shell called Teeny Tiny Shell (ttsh), that will do the same.

Commands on Linux are pre-compiled programs that exist within the directory structure. For example, the 'ls' command that you used in previous labs is located at /usr/bin/ls. The Linux directory structure begins at the root directory. Each directory name is delimited by the slash (/) character. The 'ls' command can be executed by the shell using the absolute path (e.g. /usr/bin/ls), however this isn't required because Linux has a set of paths pre-set that it uses to search for commands. The directory /usr/bin is one of those paths. When a user just types 'ls' for the command to run, Linux searches the set of paths to find where 'ls' is located. If it finds it, it runs the command. If it can't find it, the shell prints out 'command not found'. Your shell will have to be able to handle this just like 'bash' does. Thankfully the search paths are built into the exec() system call for you.

NOTE: if at any time you want to find where a command is located, you can use the 'which' command.

In addition to the commands themselves, the execution of a command also may involve one or more command line arguments. For example, you can run 'ls' to get the directory listing, but you can run 'ls -l' to have the 'ls' command print out a long, more detailed directory listing. Command line arguments are separated by spaces in the user input. Most shells support an unlimited number of command line arguments, however your shell will is only required to support up to 10 command line arguments.

Teeny Tiny Shell supports an additional feature using the semicolon. Commands separated by a semicolon are run sequentially after each other. For example:

```
echo hi ; ls -a ; echo bye
```

will run the 3 commands:

1. 'echo hi' will print 'hi' to the terminal
2. 'ls -a' will print all files in the current directory
3. 'echo bye' will print 'bye' to the terminal

Commands will be executed sequentially: the first command will run to completion (the other two will not run), then the second command will run to completion, finally the third command will run.

While the user might not specify any command line arguments (e.g. if they just execute 'ls' to get the directory listing), any call to exec requires at least one command line argument. The first argument is ALWAYS the command that is being executed. For example, if the user types the following command:

$> ls -l

This is translated to an execution of the 'ls' command with command line arguments: 'ls' and '-l'

4 Development Requirements

1. While most shells allow the user to customize the prompt, you shell will always use the following prompt string:
   `$>`
   A dollar sign ($), followed by a greater than (>), followed by a space ( )
2. Your 'ttsh' must support up to 10 command line arguments for each command (separated by spaces)
3. Your 'ttsh' must support up to 5 commands run sequentially (separated by semicolon)
4. While there many 'flavors' of the exec() system call, for this lab you are required to use execvp(). See the man page for the specifics of how execvp() works.
5. Your shell must support a single internal command called 'quit' which stops the execution loop
6. You must use fork() and execvp() to create child processes and execute commands. The use of the system() function is not allowed for this lab.
7. Your 'ttsh' must be free of memory leaks and segmentation faults.
8. Your 'ttsh' only needs to support a max user input of 256 characters.

5 Getting Started

Parsing the user's input can be kind of complicated. Using the string tokenizer function (strtok) can be helpful. For help on how to use strtok see the man page:

- `man strtok`

In addition, a starter source file is provided for you to help with command parsing

6 Sample Execution

Here is a sample output for an execution of 'ttsh'. Note the first line and last line shows the prompt presented from 'bash' prior to running 'ttsh'.

```
lembke@cs3841:~/Desktop/cs3841/labs$ ./ttsh
$> ls
ttsh   ttsh.c
$> ls -l
total 500
-rwxrwxr-x 1 lembke lembke 17240 Sep 17 12:59 ttsh
-rw-rw-r-- 1 lembke lembke  7053 Sep 17 12:47 ttsh.c
$> echo hello world
hello world
$> ps -f -l
F S UID          PID    PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
0 S lembke     43181   43170  0  80   0 -  5173 do_wai 08:59 pts/0    00:00:00 bash
0 R lembke     46683   43181  0  80   0 -  5029 -      11:59 pts/0    00:00:00 ps -f -l
$> which ls
/usr/bin/ls
$> mkdir testdir
$> ls -l
total 504
-rwxrwxr-x 1 lembke lembke 17240 Sep 17 12:59 ttsh
-rw-rw-r-- 1 lembke lembke  7053 Sep 17 12:47 ttsh.c
drwxrwxr-x 2 lembke lembke 4096 Sep 18 12:02 testdir
$> rmdir testdir
$> ls -l
total 500
-rwxrwxr-x 1 lembke lembke 17240 Sep 17 12:59 ttsh
-rw-rw-r-- 1 lembke lembke  7053 Sep 17 12:47 ttsh.c
$> asdfasdfasdfasdf
asdfasdfasdfasdf: failed to execute command
$> quit
lembke@cs3841:~/Desktop/cs3841/labs$
```

7 Development Tips

Consider breaking the lab up into pieces:

1. Create your shell input/execute loop
2. Implement the 'quit' command
3. Read and parse user input for commands
   a. Break apart commands by semicolon
   b. Break apart command line arguments by spaces
4. Get fork() and execvp() working to create a child process
   a. Build the array of command line arguments
      i. A single command with no command line arguments (remember that even without command line arguments explicitly specified there is always 1 passed to the new process)
      ii. A single command with multiple command line arguments (remember that this include the command name AND the actual command line arguments)
      iii. Make sure you end your array of command line arguments with a NULL pointer
   b. Run multiple commands in sequence with or without command line arguments
5. Handle command not found

Additional tips:

- Your shell has limitations on string length requirements. Use this to your advantage
  - A max of 256 characters of user input
  - A max of 10 command line arguments (this max includes the first argument for the command name itself, effectively making the number that the user can specify to be 9 arguments)
  - A max of 5 commands executed in sequence

- Don’t forget about using gdb to help with debugging
- Using valgrind will be helpful in this lab to ensure you do not have any memory leaks.

8 Testing and Debugging

Make sure your shell supports all the features listed in the “Development Requirements”. Make sure you test boundary cases and print appropriate error messages to the user as needed. For example, if the user attempts to execute a command with more than 10 command line arguments, print an error message to the user and wait for a new command (without executing the command).

9 Deliverables

You will need to include all your source files and any other resources you used to complete lab. Please don’t just google search for a solution, but if you do use google for any help, include a description and URL of what you used to help you.

You should ensure that this program compiles without warning (-Wall and -Wextra) prior to submitting. Using gnu90 standard is allowed (gcc default). If your implementation requires C99 or gnu99, please make that clear in the comments.

Also include a text, doc, or pdf file with a report that provides the following:

- Introduction – describe the lab in your own words
- Design – describe your thought processes in creating your solution
- Build Instructions – what must be executed (gcc, make, etc.) to build your shell. A make file is not required, but can be extremely helpful
- Resources – Describe any resources you used to help you complete the lab
- Conclusion – What specifically was challenging about this lab? What did you like about it? What could we do to improve it for others?

Prepare a zip file with all submitted files and upload the file to Canvas per your instructor’s instructions.