Project 2: Designing and Implementing a Shell

CS 411

This project gives you the opportunity to study and use numerous Linux system calls, and will give you experience with coarse-grained parallelsim via the `fork(2)` and `wait(2)` systems calls. You don't need your virtual machines for this project — you may do your work on phoenix.

References

The following are all available on, or linked to from, the course web site:

1. *Assignment 1*, from CS 314, Operating Systems, at Stonehill College. This project is an adaptation from the Shell Program project in Gary Nutt's *Kernel Projects for Linux*. **This is the main reference for this project.**
2. `pipe.c`, demonstrating a parent process creating a pipe that two child processes use to communicate.
3. *Section 2 of the Linux man pages, system calls*
4. *Section 3 of the Linux man pages, C library calls*
5. *Tutorial for printf(3)*
6. *Debugging Programs with gdb*, a quick tutorial to gdb, which you'll probably need to use at one point or another (I needed it several times.).

Repository Setup

As with the gdb practice exercise, you'll be starting from a fresh, empty repository. Start by going into Canvas and following the link you'll find there to create your team and repo for this project. Follow this rule when you create your team name:

```
<initials of 1st member>_<initials of 2nd member>_2
```

Example:

```
JLZ_TPK_2
```

If necessary, go back to the gdb exercise description and follow the instructions there for creating your project directory, initializing the repository, etc. Populate this directory with three files: `shell.c`, `README.md` and `.gitignore`. `.gitignore` is used to specify files for `git` to ignore. In this case, you will add an entry to .gitignore to ignore your program binary. See `gitignore(5)` for documentation on the use of this file.

Shell Functionality

The CS 314 project description defines for implementation five functional behaviors of the shell:

1. Running Linux programs for the user. Examples:

   ```
   ls -al /usr/local/bin
   ps
   top -a
   ```

   The shell waits for these programs to terminate before issuing the next prompt.

2. Terminating, by typing the end-of-file character (Ctrl-d) or the `exit` command.

3. Running a program in the background and immediately issuing the next prompt.

   ```
   sleep 5 &
   ```

4. Redirecting `stdin` and/or `stdout`

   ```
   ls /usr/include > /tmp/includes   # stdout redirected
   wc -l < /tmp/includes             # stdin redirected
   ```

5. Piping two commands together, so that `stdout` from the first command is redirected into `stdin` of the second command

   ```
   ls /usr/local/bin | wc -l
   ```

I will add one additional behavior to these five for you to implement:

1. Changing the working directory, using a relative or absolute path

   ```
   cd ..                   # go up one directory
   cd ../..                # go up two directories
   cd ../build             # go up one, then down one
   cd /usr/local/bin       # absolute path
   cd build                # go down one
   cd                      # go to the home directory, as defined by the HOME
                           # environment variable
   ```

   The `PWD` environment variable's value is the absolute path of the working directory, when the working directory is changed, `PWD` must be updated.

Attacking the Project

If you don't attack the project, it will attack you!

Phase One

In this phase you will get your shell to the point where it will run Linux programs for you and will exit on the end-of-file character.

My C program for the shell has the following includes and defines:

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <err.h>
#include <stdlib.h>

#define LINESIZE 1024
#define DELIMS " \n\t\r"
```

I used `fgets(3)` to read a command line from `stdin` to a `char` array with `LINESIZE` elements. You may use the following function to tokenize the command line

```
int tokenize(char *tokens[], char *command)
{
   int i = 0;
   static char line[LINESIZE];

   strcpy(line, command);

   tokens[i] = strtok(line, DELIMS);

   while (tokens[i] != NULL)
      tokens[++i] = strtok(NULL, DELIMS);

   return i;
}
```

Creating a child process to run the command, running the command, and waiting looks something like this:

```
if (fork() == 0 )
   execvp(tokens[0], tokens);
else
   wait(NULL);
```

When I say "looks something like this" I'm serious — you'll need to determine for yourself what code to add to this to handle the error conditions listed later in this document. If you don't add the proper error handling, you'll end up with two shells running after certain errors have occurred.

My phase one shell was 60 lines of code.

Phase Two

In this phase, add the `exit` and `cd` commands. These commands are handled by the shell — no calls to `fork(2)` should be made. In this phase, you'll need to perform a bit of parsing on your tokens. You'll probably find `strcmp(3)` useful:

```
if (strcmp(tokens[0], "cd") == 0)
   ...
```

`chdir(2)` is used to change the working directory:

```
chdir(tokens[1]);
```

`getenv(3)` is used to read the value of an environment variable:

```
chdir(getenv("HOME"));
```

`getcwd(3)` is used to get the current working directory and `setenv(3)` is used to set an environment variable:

```
buf = getcwd(NULL, 0);
setenv("PWD", buf, 1);
free(buf);
```

It took me about 25 lines of code to implement this phase.

Phase Three

In this phase, you'll add background execution. This is only about 10 lines of code, but it's a bit subtle. When the background process completes, it will be one of the processes that will cause the shell's `wait(2)` call to return. If you're not ready for this, you'll get the shell out of sync with foreground processes. This problem manifests itself like this:

```
tsh> sleep 3 &
tsh> sleep 10
tsh>
```

where the `sleep 10` seemingly completes in three seconds. The solution is to keep track of the foreground child's PID and loop the parent's `wait(2)`:

```
while (child != wait(NULL))
   ;
```

Phase Four

In this phase, you'll add I/O redirection. The CS 314 project description provides a pretty good introduction to this phase. Here are a few reminders and some subtle traps to avoid:

1. Redirecting `stdin` will look something like

   ```
   inFD = open(tokens[tokenCount - 1], O_RDONLY);
   close(0);
   dup(inFD);
   close(inFD);
   ```

although my call to `open(2)` was in a function separate from the three following statements.

2. Similarly, redirecting `stdout` will look something like

```
outFD = open(tokens[tokenCount - 1], O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
close(1);
dup(outFD);
close(outFD);
```

3. The subtle point here is where you close either `stdin` or `stdout`. You have two choices: perform the close in the parent process, or perform the close in the child. If you think about the consequences of closing either of these in the parent, you'll make the correct design decision.

To implement this phase, I had to restructure and rewrite some of my code from the earlier phases. While restructuring for this phase, I also designed for the next phase, in which the parent process has to create and control two child processes. The actual functionality implemented in this phase ended up being about 40 lines of code.

Phase Five

In this phase you'll add the ability to execute a two command pipe. See the CS 314 project description and `pipe.c`. You'll have to sharpen up your command line parsing function — it will need to be able to split the one command into two commands, each run as a separate child process. The parent process creates the pipe using `pipe(2)` so that the pipe is available to both children, because a parent's resources are available to its children. As demonstrated in `pipe.c`, the first child in the pipe redirects `stdout` to the write end of the pipe and closes the read end of the pipe. The second child redirects `stdin` to the read end of the pipe and closes the write end of the pipe. The parent closes both ends of the pipe

This phase required about 25 lines of code. The I/O pipe redirections were trivial, because of the generality I had designed into the Phase Four functionality. The trickiest part of this phase was writing a function that kept track of whether or not both children had completed yet, for the parent's `wait(2)` loop. Oh, and I had forgotten to close both ends of the pipe in the parent — that caused the second child to hang because it never saw the end of file condition come through the pipe.

Phase Six

Kick back and celebrate — you just wrote a working shell!

Error Checking/Recovery

Your shell should be able to intelligently handle the following error conditions:

1. An empty command line.
2. Trying to run a program that isn't on the system.
3. Redirecting input from a file that doesn't exist.
4. Redirecting output to a file to which you don't have write permission.

Error messages are typically sent to `stderr` using `fprintf(3)`:

```
fprintf(stderr, "%s: Command not found.\n", tokens[0]);
```

See the *Tutorial for printf(3)* and the documentation for `fprintf(3)` in Section 3 of the Linux man pages.

Project 2 Turn-in

By the project deadline, email to me the GitHub https URL of your project repository. The names of your team members should be listed in the README.md file in the repository. Remember to document assistance you received from others. This can be done in the email you send me your repo URL, in your README.md file, or in comments at the very top of your source code files. This project will be assessed as follows:

- Shell functionality: 90%
- Documentation: 10%