Project 2a: A Unix-Style Shell

Deadline: Feb 22 11:59:59 PM

Tests

python ~cs537-1/testing/p2a/ShellTest.py

A readme file is placed at the same location ~cs537-1/testing/p2a that gives a brief description on the tests that are part of the test cases.

Note that the tests expect "mysh" and so, make sure that your makefile produces the binary with this name.

Teams

This project is intended to be done in **teams of two**. See `piazza` or consult a TA for how to go about this. Teams with members from one lecture or from both lectures are permitted.

Team Advisory: Do not allow your partner to sink you. Each partner must hold the other accountable - do not suffer excuses.

Objectives

There are three objectives to this assignment:

- Familiarize yourself with the Linux programming environment.
- Learn how processes are created, destroyed, and managed.
- Gain exposure to the necessary functionality in shells.

Overview

In this assignment, you will implement a **command line interpreter** (aka a **shell**). The shell should operate in this basic way: when you type in a command (in response to its prompt), the shell creates a child process (or processes) that executes the command you entered and then prompts for more user input when finished.

The shell you implement will be similar to, but simpler than, the one you run every day in Unix. You can find out which shell you are running by typing `"echo $SHELL"`. You may wish to look at the man pages for `csh`, `bash`, or whatever shell you use to learn more about its functionality. For this project, you do not need to implement too much functionality.

Program Specifications

Basic Shell

You should structure your shell such that it creates a **new process for each new command** (there are a few exceptions to this, which we will discuss below). Running commands in a new process protects the main shell process from any errors that occur in the new command.

The shell has to accept commands through standard input (STDIN). Every line of input should be treated as a separate command. For reading lines of input, you may want to look at **fgets() or getline().** Be sure to check the return code of these routines for errors! (If you see an error, the routine **perror()** is useful for displaying the problem. You may find the **strtok()** routine useful for parsing the command line (i.e., for extracting the arguments within a command separated by whitespace or a tab). Some have found **strchr()** useful as well.

Your shell must be able to run **any** program in your path. The `execvp()` system call **does this for you**. For example, if the command is `myfoo a b`, `execvp` will check for a binary named `myfoo` in the contents of `$PATH` (e.g., `/usr/bin:/bin`). **This means that you do not need to add `/bin/` or similar paths to your command name before passing it to `execvp()`.**

Running Programs

Most commands will instruct the shell to run a program with some specified arguments. For example:

```
mysh> prog arg1 arg2 ...
```

In order to execute the programs given as commands to your shell, look into **fork**, **execvp**, and **wait/waitpid** system calls. See the UNIX man pages for these functions, and also read the Advance Programming in the UNIX Environment, **Chapter 8** (specifically, 8.1, 8.2, 8.3, 8.6, 8.10). Before starting this project, you should definitely play around with these functions.

You will note that there are a variety of commands in the `exec` family; for this project, you must use **execvp**. You should **not** use the **system()** call to run a command. Remember that if `execvp()` is successful, it will not return; if it does return, there was an error (e.g., the command does not exist). A challenging part is getting the arguments correctly specified. The first argument specifies the program that should be executed, with the full path specified; this is straight-forward. The second argument, `char *argv[]` matches those that the program sees in its function prototype:

```
int main(int argc, char *argv[]);
```

Note that this argument is an array of strings, or an array of pointers to characters. For example, if you invoke a program with:

```
foo 205 535
```

then argv[0] = "foo", argv[1] = "205" and argv[2] = "535".

Important: the list of arguments must be terminated with a NULL pointer; that is, argv[3] = NULL. We strongly recommend that you carefully check that you are constructing this array correctly!

Built-in Commands

There are three special cases where your **shell should execute a command directly itself instead of running a separate process.**

First, if the user enters **"exit"** as a command, the shell should terminate (either by returning from main, or with a call to `exit(0)`).

Second, if the user enters **"cd dir"**, you should attempt to change the current directory to "dir" by using the `chdir` system call. Users can run programs with paths relative to the working directory without specifying an absolute path. For example, instead of typing "/a/b/c/myprog", a user could type two commands: "cd /a/b/c" followed by "myprog". If the user simply types **"cd"** (no dir specified), change to the user's home directory. The $HOME environment stores the desired path; use `getenv("HOME")` to obtain this.

Third, if the user enters **"pwd"**, print the current working directory. This can be obtained with `getcwd()`.

[UPDATE] The output of running the 'pwd' command in a regular linux shell and invoking `getcwd()` could vary if the location of current directory involves a symbolic link. The testing script will accept either of the output result.

```
pwd: /u/s/a/sankarp
getcwd(): /afs/cs.wisc.edu/u/s/a/sankarp
```

Also, it is fine if your shell(mysh) handles **"pwd"** special command by forking and performing exec on the pwd program already available in Linux. **Solutions based on getcwd() or fork/exec of pwd program are acceptable.**

Special Features

Your shell should have a number of special features:

- overwrite redirection (">"),
- append redirection (">>"),
- pipes ("|") and
- tee ("%")

Overwrite redirection: If the user types "program args > outfile", save the output from running the program to outfile, **overwriting** any file that already exists with that name.

Append redirection: If the user types "program args >> outfile" **append** the

output of the program to the end of the outfile, creating it if it does not already exist.

Pipes: If the user types "program1 args1 | program2 args2", use the output from program1 as the input to program2.

These features are relatively easy to implement. After fork (but before exec), the STDIN and STDOUT file descriptors are already set up to refer to user-typed input and output to the terminal respectively. The `dup2` system call is useful for setting STDIN and STDOUT to refer to other sources/destinations of data. The `pipe` system call may be useful for setting up a pair of file descriptors for piping. You might also want to learn about modes for `open()` like `O_TRUNC` and `O_APPEND`. More on these calls during discussion.

Tee(%): The tee special feature is unique to this project. Specifying the tee operator ("%") will cause your shell program to internally generate a set of pipes and call a program which you must also write. The program shall implement a subset of the standard Linux "tee" program does.

```
mysh> a | b
mysh> a % b
```

The first command executes `a` and `b` sending the output of `a` to the input of `b`. The second command runs `a`, your tee-like program and `b`. The output of `a` is sent to your tee program (which writes all input to `tee.txt` in the current directory) **and** sends that same input to `b`. If `tee.txt` already exists in that directory location, overwrite its contents (similar to the behavior of '>' operator). Note that two programs are included in the command line but three are executed, `a`, `b` and your tee program **which you will call mytee**. **You may not reuse the Linux tee command.**

Multiple Operators: Your shell must handle multiple operators in the same command line and every operator will appear only once except the pipe operator. You shell must handle up to two pipes (and no more than two pipes) per input command line. Note that the "%" (or tee operator) counts as two pipe symbols. Consider:

```
mysh> a | b | c
mysh> a % b
mysh> a | b % c
mysh> a | b >> foo.txt
```

The first command must be supported and sends the output of `a` to the input of `b` and arranges for the output of `b` to be routed to the input of `c`. There are two pipes here.

The second command is allowed and is the same as `a | mytee | b`. This has two pipes.

The third command is **not** supported because it would result in three pipes. These would be `a` piped to `b`, `b` piped to your own tee command (second pipe) and finally the output of your tee command would be piped to `c` (third pipe - not allowed).

The last command line is allowed and must be supported. This command directs the output of `a` to the input of `b` and the output of `b` is appended to the file "foo.txt".

Hint

In a standard shell, execute

```
sleep 1 | sleep 1 | sleep 1 | sleep 1 | ps -l
```

Examine the process ID columns and parent process ID columns. This will suggest to you how to structure your code governing who forks who.

Command Line Syntax Checking

You will be responsible for certain command line syntax checking. As described below, you should print `Error!\n` to standard error and continue running. Here is an example of bad syntax which should cause an error:

```
mysh> a >
```

This is an error because a redirection of the output is called for but no file name is given.

Before you ask, we will not test you on this:

```
mysh> a % b > tee.txt
```

We will not attempt this in our tests because the tee operator already writes to `tee.txt`. You are invited to try it out, but tweeting something about anything is more interesting.

Assumptions

- Commands are no longer than 1024 characters.
- We will not test paths containing tildas (in most shells, "~username" is a shorthand for a user's home directory).
- Whitespace may appear before or after commands, or between command arguments.
- If any problems occur (e.g., a command doesn't follow proper syntax or a program doesn't exist), print "Error!\n" to standard error and continue running.
- You do not need to pipe or redirect built-in commands.

Handin

The handin directory (for Section 1) is `~cs537-1/handin/login/p2/linux` where `login` is your login; for Section 2, it is the same but with a 2 (~cs537-2/handin/login/p2/linux)

Copy all of your .c source files and .h header files into the appropriate hand-in subdirectory. Make sure that your code runs correctly on the Linux machines in the CSL labs. Submit a makefile along with your source files that builds both **`mysh` and `mytee`** and by default, the makefile should also include a clean section.

Grading

We will release **some (70%)** of the test cases we will use for grading before the project due date. If your programs passes these tests, you will get at least 70% of the grade unless you are not following specifications (for example, you will receive a very low score if you use the `system()` function instead of fork/exec). Programs clearly written to just pass the test cases (instead of being built for general input) will also receive little or no credit.

The remaining 30% of your grade will be based on tests not released. These will test whether you have implemented **all** the details in the specifications - so if you miss some corner case, you will likely lose points here.

Hints

Remember to get the **basic functionality** of your program working before worrying about all of the error conditions and corner cases. For example, for your shell, first get a single command running (probably first a command with no arguments, such as "ls"). Then try adding more arguments. Next, try working on multiple commands. Make sure that you are correctly handling all of the cases where there is miscellaneous white space around commands or missing commands. Finally, support for built-in commands, redirection, and pipes.

We strongly recommend that you check the return codes of all system calls from the very beginning of your work. This will often catch errors in how you are invoking these new system calls. And, it's just good programming sense.

Keep versions of your code. More advanced programmers will use a source control system such as CVS. Minimally, when you get a piece of functionality working, make a copy of your .c file (perhaps a subdirectory with a version number, such as v1, v2, etc.). By keeping older, working versions around, you can comfortably work on adding new functionality, safe in the knowledge you can always go back to an older, working version if need be.

README

The README file will take the same format as last time, and it's available @ **~cs537-2/public/samples/README**.

During hand-in, have one partner turn in the code. Both partners will have to hand in the README to their respective hand in directory. This is important

that both partners turn in the readme file so that the grading scripts can score for both of you.

If you choose to work alone on P2A (or subsequent projects), leave the second name field blank on the README file.

Academic Dishonesty Warning!

Placing your code in a public place like a public `github` can lead to cheating which *YOU* will be held accountable for.