From CS 537 - Introduction to Operating Systems - Fall 2017

Site: Project2a

Project 2a: The Unix Shell

Update:

- The hint for pipeline implementation has been updated.

Objectives

There are three objectives to this assignment:

- To familiarize yourself with the Linux programming environment.
- To learn how processes are created, destroyed, and managed.
- To gain exposure to the necessary functionality in shells.

Overview

In this assignment, you will implement a command line interpreter (CLI) or, as it is more commonly known, 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 that executes the command you entered and then prompts for more user input when it has finished.

The shells 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` at a prompt. You may then wish to look at the man pages for the shell you are running (probably `bash`) to learn more about all of the functionality that can be present. For this project, you do not need to implement too much functionality.

Program Specifications

Basic Shell

Your basic shell, called `mysh`, is basically an interactive loop: it repeatedly prints a prompt, parses the input, executes the command specified on that line of input, and waits for the command to finish. This is repeated until the user types `exit`. The name of your final executable should be **mysh**. The prompt of mysh should look like `mysh (\!)>` (note the space after the greater-than sign), where `\!` is the history number of the command (starting at 1 and incremented every time a command is entered). For example:

```
% ./mysh
mysh (1)> pwd
/afs/cs.wisc.edu/u/m/j/username
mysh (2)> date > out
mysh (3)> cat out
Tue Sep 26 10:00:00 CDT 2017
mysh (4)> wc < out
      1       6      29
```

```
mysh (5)> exit
%
```

After you print the prompt, you might want to call `fflush(stdout)` to force the stdout buffer to be flushed. Otherwise outputs from child processes might be printed to the screen before the prompt, if the commands are executed very quickly (i.e. when grading with scripts).

You should structure your shell such that it creates a new process for each new command (note that there are a few exceptions to this, which we discuss below). There are two advantages of creating a new process. First, it protects the main shell process from any errors that occur in the new command. Second, it allows for concurrency; that is, multiple commands can be started and allowed to execute simultaneously.

Your basic shell should be able to parse a command, and run the program corresponding to the command. For example, if the user types "`ls -la /tmp`" , your shell should run the program `/bin/ls` with all the given arguments and print the output on the screen. When `ls` exits, your shell should increase the command history number by 1 and print it in the next prompt.

Important: Note that the shell itself does not "implement" `ls` or really many other commands at all. All it does is find those executables and create a new process to run them. More on this in the Hints section.

The maximum length of a line of input to the shell is 128 bytes.

You may notice the following when you run your shell:

```
% ./mysh
mysh (1)> lls^[[D
```

Basically, you cannot use the left arrow key on the keyboard to move backwards if you typed something wrong. To fix this, start your shell with a wrapper as follows:

```
% ~cs537-1/ta/tools/rlwrap ./mysh
```

Built-in commands

Whenever your shell accepts a command, it should check whether the command is a built-in command or not. If it is, it should not be executed like other programs. Instead, your shell will invoke your implementation of the built-in command.

Most Unix shells have many built-in commands, such as `cd`, `echo`, `pwd`, etc. In this project, you will be implementing just the built-in commands **exit**, **cd**, and **pwd**.

The formats for `exit`, `cd`, and `pwd` are:

```
[optionalSpace]exit[optionalSpace]
[optionalSpace]cd[optionalSpace]
[optionalSpace]cd[oneOrMoreSpace]dir[optionalSpace]
[optionalSpace]pwd[optionalSpace]
```

When you run `exit`, your shell should kill background processes that are still running (more on this in Background Execution section), and then simply call `exit()`;

When you run `cd` (without arguments), your shell should change the working directory to the path stored in the `$HOME` environment variable. Use the call `getenv("HOME")` in your source code to obtain this value.

You do not have to support tilde (~). Although in a typical Unix shell you could go to a user's directory by typing `cd ~username`, in this project you do not have to deal with tilde. You should treat it like a common character, i.e., you should just pass the whole word (e.g. "`~username`") to `chdir()`, and `chdir()` will return an error.

Basically, when a user types `pwd`, you simply call `getcwd()`, and show the result. When a user changes the current working directory (e.g. `cd somepath`), you simply call `chdir()`. Hence, if you run your shell, and then run `pwd`, it should look like this:

```
% ./mysh
mysh (1)> cd
mysh (2)> pwd
/afs/cs.wisc.edu/u/m/j/username
```

Redirection

Many times, a shell user prefers to send the output of his/her program to a file rather than to the screen. Usually, a shell provides this nice feature with the ">" character. Formally this is named as redirection of standard output. To make your shell users happy, your shell should include some common redirection features (">" and "<"). Their formats are:

```
some_command[oneOrMoreSpace]>[oneOrMoreSpace]outputfile
some_command[oneOrMoreSpace]<[oneOrMoreSpace]inputfile
```

If `>` appears, the word following it is the name of a file to use as the standard output for the command to be executed. For example, if a user types "`ls -la /tmp > output `" nothing should be printed on the screen. Instead, the standard output of the `ls` program should be rerouted to the "`output`" file.

If the "`output`" file already exists before you run your shell, you should simple overwrite the file (after truncating it).

Similarly, the standard input of a program can also be redirected to be read from a file, rather than from the user interactively. The syntax is to use `<` instead of `>`. This operator will tell the program to use the content of the file as its standard input.

Important: note that a command can be followed by both input and output redirections, and in any order.

Only one file can be specified in redirection, so the following inputs should not work and raise an error message:

```
ls > out1 out2
ls < out1 out2 out3
```

Your shell should also output errors in the following cases:

- No files are specified after the redirection operators
- The file specified by output redirection (>) cannot be created
- The file specified by input redirection (<) does not exists

Pipeline

Sometimes we may want to feed the output of one program to the input of another program without creating an additional file. Unix shells solved this problem by implementing a pipeline. You shell should also support this functionality. Its format is

```
some_command_1[oneOrMoreSpace]|[oneOrMoreSpace]some_command_2
```

The standard output of the first program will become the standard input of the second program, but the output of the second program should still be printed on the terminal. For example,

```
mysh (1)> seq 5 | wc -c
10
```

The two programs should run concurrently, so the second program should begin to process its input even if the first program is still running and generating more outputs. However, if the second program has already exited before the first program finishes, the first program should also exit. For example, `yes` is a program that will continuously generate `"y\n"`, but the `head -1` command only reads one line of input and exits:

```
mysh (1)> yes | head -1
y
```

Your shell should report an error if there is no commands before or after the pipeline, or the commands are invalid.

Background Execution

Sometimes, when using a shell, you want to be able to run multiple jobs concurrently. In most shells, this is implemented by letting you put a job in the "background". This is done as follows:

```
mysh (1)> ls &
```

The & operator will always be the last word of an input. You should not wait for the child process to exit before moving on and printing the next command prompt. Thus, you can start more than one job by repeatedly using the trailing ampersand:

```
mysh (1)> ls &
mysh (2)> ps &
mysh (3)> find . -name *.c -print &
```

Note that outputs are not shown here because they will vary in different systems, but they should be printed to standard output just like normal commands.

Processes that are started and left running in the background this way must be tracked by your shell, however, so that you can kill the unfinished processes when user enters the built-in `exit` command.

Note that a child process will become a zombie (defunct) process if it finishes but the parent does not collect its return values. You shell should avoid leaving zombie processes floating around by checking if any background processes have exited. You can use the `ps` command to check if there are any zombie processes left.

Assumption

You may make the following assumptions (simplifications):

- No redirection, pipeline, or background execution will be used with built-in commands.
- No mechanism for escaping or quoting whitespace (or anything else) is needed.
- The maximum number of background processes is 20.
- All redirection operators and pipelines will be separated by spaces, so it is okay for commands like "`ls>a`" not work.
- No input line will include more than one output redirection operator (`>`), more than one input redirection operator (`<`), or more than one pipeline (`|`).
- Pipeline will **not** be combined with redirections or background execution. (Redirections will be combined with background execution, however.)
- A command that is empty (or all-whitespace) after removing its redirection syntax should be ignored -* don't try to print an error or record it in history, just move on to the next command.

Defensive Programming and Error Messages

Defensive programming is required. Your program should check all parameters, error-codes, etc. before it trusts them. In general, there should be no circumstances in which your C program will core dump, hang indefinitely, or prematurely terminate. Therefore, your program must respond to all input in a reasonable manner; by "reasonable", we mean print the error message (as specified in the next paragraph) and either continue processing or exit, depending upon the situation.

Since your code will be graded with automated testing, you should print this one and only error message whenever you encounter an error of any type:

```
char error_message[30] = "An error has occurred\n";
write(STDERR_FILENO, error_message, strlen(error_message));
```

For this project, the error message should be printed to stderr. Also, do not attempt to add whitespaces or tabs or extra error messages.

There is a difference between errors that your shell catches and those that the program catches. Your shell should catch all the syntax errors specified in this project page. If the syntax of the command looks perfect, you simply run the specified program. If there is any program-related errors (e.g. invalid arguments to ls when you run it, for example), let the program prints its specific error messages in any manner it desires (e.g. could be stdout or stderr). For example,

```
mysh (1)> invalid-command
An error has occurred
mysh (2)> ls
lint  tests  tools  xv6
mysh (3)> ls /bad
ls: cannot access '/bad': No such file or directory
mysh (4)>
```

Here the shell cannot find `invalid-command`, so the only error message is printed. However in the third command, the shell starts `ls` with argument /bad successfully, but `ls` reports its own error message.

You should consider the following situations as errors; in each case, your shell should print the error message to stderr and exit gracefully:

- An incorrect number of command line arguments to your shell program.

For the following situation, you should print the error message to stderr and continue processing:

- A command does not exist or cannot be executed.
- A very long command line (over 128 bytes).

Your shell should also be able to handle the following scenarios below, which are not errors.

- An empty command line.
- Multiple white spaces on a command line.
- Tabs are used in place of spaces.

All of these requirements will be tested extensively.

Hints

Writing your shell in a simple manner is a matter of finding the relevant library routines and calling them properly. To simplify things for you in this assignment, we will suggest a few library routines you may want to use to make your coding easier. You are free to use these routines if you want or to disregard our suggestions. To find information on these library routines, look at the manual pages.

Basic Shell

Parsing: For reading lines of input, check out `fgets()`. 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. But do not print the error message from `perror()` to the screen. You should only print the one and only error message that we have specified above). You may find the `strtok_r()` routine useful for parsing the command line (i.e., for extracting the arguments within a command separated by whitespaces).

Executing Commands: Look into **fork**, **execvp**, and **wait/waitpid**. See the man pages for these functions, and also read book chapter here.

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). The most challenging part is getting the arguments correctly specified. The first argument specifies the program that should be executed; this is straightforward. 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; in our example, this means `argv[3] = NULL`. We strongly recommend that you carefully check that you are constructing this array correctly!

Make sure you use `waitpid()` to wait for the right process. If you start a background job A and then a foreground job B, you should not move on to the next command if A completes and exits while B is still running (i.e. don't use plain `wait()`).

Built-in Commands

For the exit built-in command, after checking background processes, you should simply call `exit()`. The corresponding process will exit, and the parent (i.e. your shell) will be notified.

For managing the current working directory, you should use **getenv**, **chdir**, and **getcwd**. The `getenv()` call is useful when you want to go to your HOME directory. You do not have to manage the PWD environment variable. `getcwd()` system call is useful to know the current working directory; i.e. if a user types pwd, you simply call `getcwd()`. And finally, chdir is useful for moving directories. Fore more information on these topics, read the man pages or the Advanced Unix Programming book Chapters 4 and 7.

Redirection and Pipeline

Redirection is relatively easy to implement. For example, to redirect standard output to a file, `open()` a file and use `dup2()` to connect it to stdout. More on this below.

With a file descriptor, you can perform read and write to a file. Maybe in your life so far, you have only used `fopen()` , `fread()` , and `fwrite()` for reading and writing to a file. Unfortunately, these functions work on `FILE*` , which is more of a C library support; the file descriptors are hidden.

To work on a file descriptor, you should use `open()` , `read()` , and `write()` system calls. These functions perform their work by using file descriptors. To understand more about file I/O and file descriptors you should read the Advanced Unix Programming book Section 3 (specifically, 3.2 to 3.5, 3.7, 3.8, and 3.12), or just read the man pages. Before reading forward, at this point, you should become more familiar with file descriptors.

The idea of redirection is to make the stdout descriptor point to your output file descriptor. First of all, let's understand the STDOUT_FILENO file descriptor. When a command "`ls -la /tmp`" runs, the ls program prints its output to the screen. But obviously, the ls program does not know what a screen is. All it knows is that the screen is basically pointed by the STDOUT_FILENO file descriptor. In other words, you could rewrite `printf("hi")` in this way: `write(STDOUT_FILENO, "hi", 2)` .

For output redirection, the output file should be opened with the `open()` system call using the flags O_WRONLY, O_CREAT, and O_TRUNC (see man 2 open). You should then use the `dup2()` system call to use the resulting file descriptor as the standard output (file descriptor number 1) of the child process in which the command is executed.

For input redirection, open the input file with the O_RDONLY flag and use the resulting file descriptor (again via `dup2()`) as the child process's standard input (file descriptor 0).

For pipeline, use `pipe()` instead of `open()` to get the file descriptor to redirect the file descriptors of both programs. The stdout descriptor of the first program should be connected to the write end of the pipe, while the stdin descriptor of the second program should be connected to the read end.

If `pipe()` is called in the parent process (your shell), remember to close them after you are done with creating child process. Otherwise, the parent process will continue to keep the file descriptors of the pipe, and the pipe will stay open even after all child processes have exited.

Background Execution

If the & operator is given, you should **not** call `wait()` (or anything similar) to wait for the child process to exit. You need to maintain a list of currently-running background processes in your shell. You can use `waitpid()` with

WNOHANG or `kill()` with sig set to 0 to check if any of these processes have exited (see their man pages). When the built-in `exit` command is called, use `kill()` to send signals to terminate all still-running processes.

Testing

It is a good habit to get basic functionalities working before moving to advanced features. To run the test scripts, go to the directory where your source codes reside and type

```
shell% ~cs537-1/ta/tests/2a/runtests
```

You can run an individual test with the following command:

```
shell% ~cs537-1/ta/tests/2a/runtests testname
```
