COSC 301: Operating Systems
Fall 2014
Project 2: Roll your own shell
Due: 15 October 2014, 11:59:59 pm

Overview

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

The shell you implement will be similar to one that you use every day in Linux (e.g., bash), but will be much simpler. In particular, you do not need to implement pipes or input/output redirection and numerous other standard features in modern shells.

The goals for this project are as follows:

- To gain more experience in a UNIX programming environment.
- To develop your defensive programming skills in C.
- To learn how processes are started, stopped, and otherwise managed through the C API in UNIX.

You can (and should) work in pairs. Only one submission needs to be made per pair, but please make it clear through comments at the top of your code who worked on the project. In those comments, include a short description of who worked on what (1 or 2 sentences are fine).

To get started: fork the git repo at https://github.com/jsommers/cosc301_proj02. This will create a new repo in your account called cosc301_proj01. You should then clone that repo to your virtual machine or other workspace for editing, compiling, and testing the code. To submit your work, you should just commit and push all your changes to github, and post the repository location to Moodle. Note that only one person in a pair needs to do the fork; other github users can be added as collaborators on the newly forked repo, which will help to make collaboration easier.

Important note: there are two stages to achieve in this project. The second stage requires more sophisticated programming and provides more features in your shell. The stages are designed and will be graded such that you can still achieve a good grade even if you only get through stage 1. *Work through the stages sequentially*, i.e., don't try to incorporate stage 2 features unless you're confident that everything works well for stage 1.

And one more thing: some aspects of the project are intentionally left vague. Especially when you get to stage 2, the correct behavior for your shell may not be obvious. Ask questions by posting to Piazza.

Detailed Description

Stage 1 functionality (worth 85 out of 100 points for project correctness)

Your shell should run in an interactive fashion. You should display a prompt (any string you want) and the user of the shell will type a command at the prompt. Your shell should receive both program names to execute, along with program options (e.g., `/bin/ls -l`) or "built-in" commands, such as `exit`. For the first stage of the project, a user will need to type the entire path (i.e., complete location in the file system) for a system command, like `/bin/ls` instead of just `ls`. You can assume a reasonable maximum length of an input line (e.g., 1024 characters is fine). The basic requirements for this stage are as follows:

- Each command (unless it is a shell built-in command) should contain the full path to the executable file (i.e., `/bin/ps` rather than just `ps`). Your shell should be able to handle any arbitrary number of command-line options to different system programs (e.g., `/bin/ls -l -t -r`). Whitespace (tabs and spaces) between command-line arguments shouldn't matter. There are just two built-in commands you'll have to handle, `exit` and `mode`, described below.
- You must be able to handle comment strings in your shell. Anything after a `#` (hash) character should be ignored.
- To quit the shell, the user can type `exit`. The effect should be just to quit the shell (the `exit()` system call may be useful here). Note that `exit` and `mode` are built-in shell commands. They are not to be executed like other programs the user types in.
- Multiple commands may appear on the same command line. Each command must be separated by the `;` character (semicolon). For example, if a user types: `prompt> /bin/ls ; /bin/ps`, the shell should run both the `/bin/ls` and `/bin/ps` programs. Spaces or tabs before or after a semicolon shouldn't matter. For example, the following command string should be valid: `/bin/ls;/bin/ps ;     /bin/ls`. (Note that using a semicolon to separate commands on the same line is also permissible with a standard shell program such as `bash`.)
- The shell should be able to run in two different modes: sequential and parallel. In *sequential* mode, when multiple jobs are listed on a single command line, they should be run one at a time to completion, in left-to-right order. So, for the example above (`/bin/ls ; /bin/ps`), first `/bin/ls` should run to completion, then `/bin/ps`. In contrast, in *parallel* mode, the jobs should all be started in rapid succession and essentially allowed to run in parallel. In both modes, the prompt should not be shown again until all jobs are complete (the `wait()` or `waitpid()` system calls will be useful here).
- To switch into sequential mode, the user types `mode sequential`. Similarly, to switch into parallel mode, the user types `mode parallel`. Shortcuts should be available (e.g., `mode s` and `mode p` should work, too). The shell should begin in sequential mode. If a

user types `mode` but does not provide any other recognized subcommand (parallel or sequential), you should print the current execution mode for the user.

  If one of the commands in the sequence of jobs on one line is to change the mode, you should only change it for the *next* shell input. For example, if the shell is in parallel mode and a user types: `/bin/ls ; mode p ; /bin/ps`, those commands should be all run sequentially, but the *next* command (or sequence of commands) should be run in parallel.

- Your shell should recognize the end of the input stream and terminate when it gets an "end of file" (EOF) notification via a `ctrl+d`. You only need to worry about handling an EOF notification on an input line by itself. (That is, don't worry about a situation in which there are commands entered into the shell, followed by a `ctrl+d`, all on the same line.)
- You should *never* exit the shell while jobs are running. Hence, if some jobs are running and you receive an exit or EOF, you should wait until they complete and then exit. For example, if a command sequence is typed like: `/bin/ls ; mode s ; exit ; /bin/ps`, you should not exit until all other commands are processed. Thus, the output of `/bin/ls` and `/bin/ps` should be printed to the console before you exit.
- You should be able to receive multiple built-in commands (as well as system commands) on a single line, such as `mode` or even `exit`.
- You should structure your shell such that it creates a new process for each new command. 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 easy concurrency; that is, multiple commands can be started and allowed to execute simultaneously (i.e., in parallel mode). You will want to look into the `fork()` system call for creating a new process.
- When starting a new command (after creating a new process context in which to run it), you *must* use the `execv()` system call. (There are a number of other exec-like calls; please stick with `execv()`.) Note that `execv()` takes a structure similar to that produced by the `tokenify` function you wrote in lab 2. You're welcome to reuse that code for this project.
- For collecting the carcass of a child process after it has been spawned and has completed, you should look into the `wait()` and/or `waitpid()` commands. `waitpid()` is much more flexible (and its use will be necessary for stage 2, below). The simpler `wait()` system call is sufficient for stage 1. Note that you wrote a simple fork/exec/wait sequence in lab 3.

Defensive programming is an important concept in operating systems: an OS can't simply fail when it encounters an error; it must check all parameters 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", I mean print an understandable error message and either continue processing or exit, depending upon the situation.

Moreover, your program should not have any memory leaks or any memory corruption problems of any kind. Be sure to test your shell using valgrind to check for memory-related problems. I can guarantee you that I will run your program with valgrind.

If a command that a user types does not exist, you should print an error message to the user and continue processing any further commands.

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

- An empty command line.
- Multiple white spaces on a command line.
- White space before or after the `;` character or extra white space in general.

Hints for stage 1

In this section are a few tips for writing your code for stage 1. You don't strictly need to do things exactly as suggested, so if you think you have a "better" way, go for it.

Your shell is basically a loop: it repeatedly prints a prompt (if in interactive mode), parses the input, executes the command specified on that line of input, and waits for the command to finish, if it is in the foreground. This is repeated until the user types `exit` or ends their input. You can assume a reasonable upper-bound on the length of a command line, like 1024 characters. For example:

```
printf("%s", prompt);
fflush(stdout);  /* if you want the prompt to immediately appear,
                    call fflush.  it 'flushes' the output to screen */
char buffer[1024];
while (fgets(buffer, 1024, stdin) != NULL) {
    // process current command line in buffer

}
```

Some tips for the command-line parsing:

- Handle any comments first; search for the first # char (if one exists) and overwrite it with the C string termination character. At this point, you will have commands separated by semicolons, like:

  ```
  prompt> /bin/ls -l ; /bin/echo "blah, blah, blah"
  ```

- First, you should parse the command line and organize the separate commands in memory. I suggest that for each command, you parse and organize it in memory so that it is represented by an array of pointers to char, with the last element in the array set to NULL. (There's a really good reason for this: the `execv` system call that you'll use to run a command needs the command in this format.) You should use (or adapt) your tokenify function to do this! A code snippet that uses |execv()| is shown below:

```
/* argv is an array of strings, with the last element in the
   array as NULL */
char *argv[] = { "/bin/ls", "-ltr", NULL };

/* morph the child process into running a new program, as specified
   by the command line;  in this example, we hard-code things to
   run the ls program */
if (execv(argv[0], argv) < 0) {
    fprintf(stderr, "execv failed: %s\n", strerror(errno));
}
```

Remember that once you (successfully) call `execv()` the calling process will be running a completely different program!

- You'll have to be able to parse and somehow store multiple sets of commands separated by semicolons on the command line. You could create an array of pointers to each (ahem) array of pointers for each command. The top-level array would have an entry for each separate semicolon-separated command, and could be just terminated with a NULL. You can easily find out how many commands are on a command line by counting the semicolons (and adding 1). (Note that this is an upper bound: a valid command line is `/bin/ls ; ; ;` which really just contains one command.)

Start slowly: get the basic parsing and process creation functionality of your shell working before worrying about various edge cases. For example, focus on getting a single command running in sequential mode, then add support for multiple jobs separated by semicolons on the same line. After that, you can try to get parallel mode working. You should also leverage any of the code you've already written (linked lists, tokenify, etc.)

Stage 2 functionality (worth 15 out of 100 points for project correctness)

First, make sure everything works for stage 1. Stage 2 adds:

- A `PATH` variable-like capability to your shell, and
- The ability to run jobs in the background while continuing to accept new commands in the shell.

`PATH` variable capability for stage 2

For the `PATH` variable capability, your shell should read the file `shell-config`. This file should have a list of directories to search for programs that can be invoked from the shell prompt without giving the explicit path (i.e., this list of directories makes up the `PATH` variable in a standard shell). If this file doesn't exist, your program should still start up, but require a user to give the full directory path to a program to execute as in stage 1.

Each directory to search for executable files should be specified on separate lines of the input file. For example:

```
/bin
/usr/bin/
/sbin
/usr/local/bin
.
```

You can try the following:

1. You can organize your list of directories as a linked list (or some other way).
2. Test whether the command, as given, refers to an actual file. You can use the `stat()` system call to check whether a file exists at a particular path. (`man 2 stat`, and an example is shown below.)
3. If the file as given doesn't exist, you can prepend each path variable element, in order, an test whether *that* file exists. For example, if someone types `prompt> whoami`, you would first check whether the file `whoami` exists using `stat` (which will almost certainly fail.) You can then try `/bin/whoami`, then `/usr/bin/whoami`, then `/sbin/whoami`, etc. For the first command that you find exists in the file system, call the `execv()`.

Using `stat` isn't too hard. Here is a short example:

```
struct stat statresult;
int rv = stat("/usr/bin/ls", &statresult);
if (rv < 0) {
    // stat failed; file definitely doesn't exist
}
```

Background job capability for stage 2

For this part of stage 2, you will improve the process handling capability of your shell. If you are in parallel mode, you must allow jobs to continue "in the background" and accept new commands from the interactive prompt or batch file. As processes that have been started complete, you must print a completion message for the user. Note that sequential mode processing should be identical to stage 1; only parallel mode should change.

For example, if the shell is in parallel mode and you type: `/bin/sleep 120`, a new process should be started to run the sleep program (which will run happily for 120 seconds). *Immediately* after spawning the process to run the sleep program, you should display a new prompt for the user to enter a new command. Once the sleep program completes, a message such as

`Process 5534 (/bin/sleep 120) completed` should be displayed for the user. Your shell should not limit the number of jobs that can be run in the background.

In addition, you should provide three new built-in shell commands:

`jobs`

should print out a list of all processes that are currently running in the background. At minimum, you should print out the Process ID, the command being executed, and the process state (either running or paused).

`pause PID`

the `pause` command should send a signal to the background child process with process ID [PID] in order to pause the process. After pausing a process, running the `jobs` command should show the updated status.

`resume PID`

the `resume` command should send a signal to a background child process that has been paused in order to restart it. Again, running the `jobs` command after resuming a process should show the updated status.

For this stage, you will probably find the system calls `kill()` and `poll()` useful. You can use `kill(pid, SIGSTOP)` to pause a process, and `kill(pid, SIGCONT)` to resume it. Since you'll potentially have processes running in the background, and at the same time need to handle user input, you'll need a way to periodically test for input and handle it, while also periodically checking for child process death. The `poll()` system call is one of the best ways to handle that:

```
// declare an array of a single struct pollfd
struct pollfd pfd[1];
pfd.fd = 0; // stdin is file descriptor 0
pfd.events = POLLIN;
pfd.revents = 0;

// wait for input on stdin, up to 1000 milliseconds
int rv = poll(&pfd, 1, 1000);

// the return value tells us whether there was a
// timeout (0), something happened (>0) or an
// error (<0).

if (rv == 0) {
    printf("timeout\n");
} else if (rv > 0) {
    printf("you typed something on stdin\n");
} else {
    printf("there was some kind of error: %s\n", strerror(errno));
```

```
        }
```

As in stage 1, never exit the shell when there are jobs running. If a user types `exit` and there are processes running, you should simply print an error message. You need not "defer" the exit command until all processes have completed (i.e., if a user wants to exit, he/she must type exit again).

Other suggestions

It is strongly recommended 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.

Beat up your own code! You are the best tester of your code. Throw lots of junk at it and make sure the shell behaves well. Good code comes through testing — you must run all sorts of different tests to make sure things work as desired.

Use `valgrind` to check for memory corruption — you will end up doing lots of string processing in this project, and it's easy to make mistakes that corrupt your address space. Valgrind can help ferret out those types of problems. To use valgrind, just type `valgrind ./proj02`.

How to submit

Just post a link to your git repository on Moodle for submission. That's it.