COSC 301: Operating Systems
Fall 2019
Homework 3: A new shell awakens
Due: 24 October 2019, 11pm

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 with many simplifications. In particular, you do not need to implement pipes or input/output redirection and numerous other standard features in a modern shell such as `bash`.

The goals for this homework 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.

To get started: click on the Github classroom link on Moodle to create a repo in your Github account, then clone the repo on birds.cs.colgate.edu like you've done for other homeworks and labs. To submit your work, you can simply commit and push all your code changes to Github; there's no need to submit anything to Moodle.

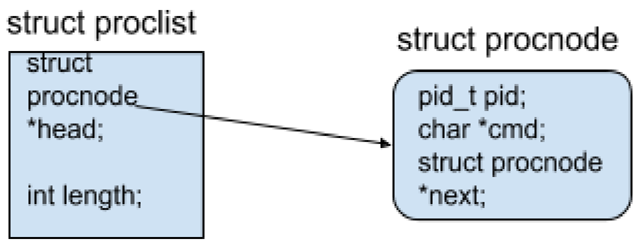
Again, please work in groups of 2. You may choose to work in the same groups as in homeworks 1 or 2, but you may also change groups if you'd like. I'll allow groups of 1 or 3 in special circumstances.

Important note: there is some advice below on how to proceed with building your shell and adding new functionality. I strongly suggest that as you add new features to your shell, you compile and test frequently. Do not try to write *everything* (or even a lot of the code) then go back and test. Take it slow, add a little bit of code at a time, and compile and test as you go. This homework is harder than you might think.

Detailed Description

Linked list helper functions

Before you dive into writing the shell code, you'll need to write a few linked list helper functions to use when you add shell functionality to handle "background" processes. There are 6 functions in `list.c` to complete. You will want to refer to `list.h` which has two struct definitions for implementing the linked list.



A few notes about the list implementation:

- The struct proclist will hold a pointer to the head of the actual list (pointer to a struct node), as well as the length of the list. In your shell, you'll just have one instance of a (pointer to a) struct proclist.
- The linked list will be composed of zero or more struct procnode's. (Notice that the head attribute in the struct proclist points to the first struct procnode in the linked list.) Each struct procnode has:
    - A process id (pid).
    - A pointer to a char (i.e., a C string) with the program running in the process (i.e., the command, which is abbreviated cmd). You only need to include the main program file part of the command (e.g., /bin/ls), not any command-line arguments to a program. **The cmd should be allocated on the heap**! When ever you create a new struct procnode, you should create a new heap allocation and copy the string to the heap (hint: use strdup). When you free up a struct procnode, don't forget to free the cmd!
- You'll need to write functions to:
    - Allocate a new struct proclist on the heap and initialize it;
    - Free a proclist, and any procnodes on the linked list;
    - Add a new procnode to the linked list;
    - Remove a procnode from the linked list;
    - Find a procnode on the linked list given a pid;
    - Print some representation of the processes on the proclist.

Again, these functions will be used when you implement background processes in the shell, which is described under "Advanced shell operation", below.

There are some functions provided to help you test your linked list code. When you type `make`, a program named `listtest` will be created (as long as things successfully compile). You can run this program (`./listtest`) to perform some basic tests on your linked list code. I strongly recommend that you also use scan-build (`scan-build make`) and valgrind (`valgrind ./listtest`) to test your list functions, and to ensure that you don't have any memory leaks or other unsavory memory-related actions going on.

Basic shell operation

Once your linked list functions are working, you can get started on the basic shell operation. All the code you write for your basic (and advanced) shell functionality should go in `main.c`. When you type make, a program named `shell` will be created.

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 your shell, 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.
- You must be able to handle comment strings in your shell. Anything after (and including) a # (hash) character should be ignored.
- There are two built-in commands that you will need to handle for the shell's basic operation. Note that these commands must be handled differently than other commands; they are not to be executed like /bin/ls or other programs invoked through the shell. Here are the two built-in commands you must handle:
    - `exit`
        - 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).

    - `status`
        - When a user types `status` as a command, the shell should print the current amount of CPU time that the shell has been executing in user mode and in system mode. You should print user and system computation time both for the shell process as well as for all child processes. You can get these values through the `getrusage` system call (see below for more on this system call). Here is an example of how your output should look (note that the times reported will almost always be zero, too):

          ```
          prompt> status
          ```

```
Shell: user 0.000000 sys 0.000000
Children: user 0.000000 sys 0.000000
Processes currently active: none
```

To call getrusage, you should just create a struct rusage on the stack, and pass the address of that struct, like:

```
struct rusage usage;
getrusage(RUSAGE_SELF, &usage);
```

Refer to getrusage's man page for more detail (note that you'll have to call `getrusage` twice with a different first parameter to get usage for "self" as well as child processes). The user and system time fields in `struct rusage` are of type `struct timeval`. There are two fields in struct timeval: `tv_sec` and `tv_usec`, for seconds and microseconds, respectively. To print the values of a `struct timeval` in a nice way, you can do something like the following (assume that your `struct timeval` variable is named `tv`):

```
printf("%d.%06d\n", tv.tv_sec, tv.tv_usec);
```
(note that the %06d provides a field width of 6 digits --- appropriate for microseconds, since there are up to 999999 microseconds per second --- and will "pad" the printed value on the left with zeroes)

The `status` command should also cause a list of processes that are currently running in the background to be printed. For the shell's basic operation there will not be any background processes, so running `status` should always result in the shell reporting that there are no processes currently active, as shown in the example above.

- 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 the `/bin/ls`, then once it is done should immediately run the `/bin/ps` program. Spaces or tabs before or after a semicolon should not 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`.)
- 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 do not need to worry about handling "quotes" on a command line like a typical shell must handle. (For example, you can say echo "hello, world" in bash, and bash will identify that you've given a quoted string to "echo" to the shell. You do not need to identify or handle any quoted strings in your shell.)
- You should structure your shell such that it creates a new process for each new command using the `fork` system call. When starting a new command (after creating a new process context in which to run it via `fork`), you *must* use either the `execv` or `execvp` system call. (There are a number of other exec-like calls; please stick with one of these.) Note that `execv()` and `execvp()` take a structure just like that produced by the `parse_tokens` function you wrote in lab 3. Code for this function(written as a function named `tokenify`) has already been included in your `main.c` template to reuse.
- In the basic operation of the shell, after creating a new process you will need to immediately wait for it to complete before running any other commands or processing new input from the command line. You should use either the `wait()` or `waitpid()` commands. `waitpid()` is much more flexible (and its use is necessary for the advanced shell operation, described below). The simpler `wait()` system call is sufficient for the shell basic operation. Note that there are examples of using `fork`, `execv`, and `wait` in lab3. There are also examples with using these functions in Chapter 5 of OSTEP.
- Be careful about handling the situation when `execv` fails. If it fails (which will happen in a child process), you should just call `exit(-1)` to complete the child process.

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 (crash), 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.

Other tips for shell basic operation

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

}
```

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 "blahblah"
```

You can then tokenify on ";", then for each command run tokenify on it (but splitting on spaces). If the command is "empty" (i.e., all spaces) you should just ignore it.

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.)

Advanced shell operation

In the shell's advanced operation, you will add the capability to run processes in the "background". That is, when you start a command, you should be able to type a new command in the shell immediately, without waiting for the child process to finish.

The way you will indicate that a command should be executed as a background process is to use the & character at the end of the command. The spacing between the end of a command (and any options) and the & should not matter, and any spacing after the & should not matter. Here are some example (valid) command lines:

(Notice just one command with a couple options, run in the background)

```
prompt> /bin/ls -lt -r    &
```

(Notice three commands; the first is run in the background, so the second command is immediately run; after /bin/ls completes /bin/sleep 30 is executed in the background; the shell should immediately be printed without waiting for the sleep command to finish, which will take 30 seconds!)

```
prompt> /bin/ps -ef & ; /bin/ls; /bin/sleep 30&
```

When a job is started in the background:

- You should not immediately wait for it to finish (which should go without saying, but I'm saying it anyway).
- You should add the child process's pid and the cmd to the proclist. Use the `proclist_add` function that you already created.
- Each time through the main loop of your shell (i.e., when you are printing the prompt and getting the next line of text), you should scan the linked list of processes. For each process on the list, do the following:
    - Check whether it has completed using the `waitpid` system call, which takes three parameters: the process id, a pointer to an int (which will be the exit status of the child process), and an integer flag. For the flag value you should use WNOHANG, which tells `waitpid` not to "hang" or wait if the process is not finished. If the child is done, `waitpid` will return the child's pid. If the child is not yet done, `waitpid` will return 0. If there's an error with `waitpid`, it will return -1. (See `man waitpid` for more details.)
    - For any child processes that have finished, you'll need to remove them from the process list (and free up the memory used by the linked list node for the child process).
    - Note that since this process list scanning will only happen when you go through the main loop, you might need to hit return on an empty command line for your shell to figure out that a child process has finished. (A real shell has some additional features to find out immediately when a process is done, but you don't need to do any fancy tricks for your shell.)
- When a user types the shell built-in command status, you should print a representation of the background processes running (i.e., nodes on the proclist). Here's an example of what that might look like (but your output can be anything reasonable, as long as it includes the pid and cmd):

```
Processes currently active:
```

```
[30]: /bin/sleep
[29]: /bin/sleep
[28]: /bin/sleep
```

You'll need to implement one new shell built-in command: `kill pid,` where `pid` is some integer. When this built-in command is given, you should check whether a child process with the given pid is running. If a child is running with that pid, you should issue the `kill` system call to kill the child process. You should call it like: `kill(pid, SIGKILL);` There are different values that can be specified for the second parameter, but SIGKILL will kill the child process definitively. See the man page for kill if you want to learn more.

Lastly, 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 just print a warning that child processes are running, and go back to accepting new commands.

Other suggestions

It is strongly recommended that you check the return codes of all system calls from the very beginning of your work. (`printf` is one exception; you don't need to check return values from this function.) 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 homework, 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 ./shell`. Again, `scan-build` will also be helpful in pointing out uninitialized variables and other lurking problems. Run `scan-build make` to invoke that tool.

Submitting your work

When you're done, commit your code to git and push to github. The push command should be:

```
$ git push origin master
```