CS105 – Computer Systems Spring 2023

Assignment 7: Shell Lab

Due: Tuesday, March 28, 2023 at 11:59pm

In this homework, you will be building out the core system-call logic of an interactive shell. The goals of this assignment are for you to learn how to manage processes with fork and exec and to understand child process management. As usual, you should complete this assignment with a partner; you may choose your partner for this assignment.

The starter code for this assignment is available both on the course website and on the course VM. You may complete it either on your local machine or on the course VM (the file path is `/cs105/starters/shelllab.tar`. Note that you will need to be connected to the Pomona WiFi network or the Pomona VPN to access the VM.

As usual, you can unpack the starter code with the command

```
tar xvf /cs105/starters/shelllab.tar
```

This will create a subdirectory named `shelllab-handout` containing three files: `Makefile`, `ish.c` (where you'll work), and `snooze.c` (a helper for testing). Fill out your team members in the comment the top of `ish.c`, then run `make` to build the executables `ish` (your shell) and `snooze` (for testing).

It's shell...-ish

The *shell* is the expert's control hatch to the computer, typically run in a terminal or console. In this lab, you'll be writing the core system-call logic of a very tiny shell we're calling `ish`.

The starter code has four parts:

- The `main` function, which consists of a read-then-evaluate loop. It uses `getline` to read a line from the user, and then parses the line into a command and its arguments.
- The `setup_signal_handlers` function for setting up interrupt handlers. These handlers can intercept signals from the operating system.
- The definition of `job_t` and its associated functions, `add_job`, `free_job`, and `check_jobs`. You'll use these functions to keep track of background jobs; you'll need to write `check_jobs` yourself.
- The `parse_line` function, which breaks a line up into an array of 'words', the first of which will be interpreted as a command. You should not have to make any changes to this function.

You have six tasks, which will touch three functions in total (plus one you'll write yourself):

1. Get `ish` to actually run the command. (`main`)
2. Have `ish` print the status if it was non-zero. (`main`)
3. Enable `ish` to run jobs in the background. (`main`)
4. Make `ish` wait for background jobs to finish before exiting the shell. (`main`, `check_jobs`)
5. Make `ish` report on background jobs before each prompt. (`main`, `check_jobs`)
6. Optimize `ish` to only report on jobs when something has changed. (`main`, `setup_signal_handlers`)

I strongly recommend that you read through this entire document before you start, and that you not move on from one task until you're confident you have the right behavior.

Important Note: the shell itself should only print to *standard error* (Unix file descriptor 2), a special output stream that's different from *standard output* (Unix file descriptor 1). You can see examples already in the starter code, where we use the `fprintf` system call with the `FILE` stream `stderr` as its first argument. Make sure you should do that, too. Printing to `stdout` (e.g., with `printf`) will not be considered correct behavior.

1 Running the command

Towards the end of `main`, you'll find the code snippet:

```
// Parse user_input command
int num_words;
char **args = parse_line(user_input, len, &num_words);
assert(args);
assert(args[num_words] == NULL);

// TODO #1: run the user_command
```

At this point in the program, `args` is an array of strings (i.e., an array of `char *` aka a pointer of type `char**`). Your first task is to get `ish` to actually run the command in `args`. There are three steps:

1. Use the `fork` function to create a new process. This function calls the underlying Linux system call.
2. In the child process, use `execve` to run the given program.
3. In the parent process, use the `waitpid` function to wait for the child process to complete. You should use `options=0`.

You'll want to look at the manpages for each of these commands: run `man fork`, etc or use the provided links. You need to read these pages carefully, especially the `RETURN VALUES` section.

The `execve` function can feel a bit odd until you've seen it a few times. The first item of `args` is what you will pass in as the pathname; you will use all of `args` as the second argument to `execve`, `argv`. You should use an empty environment, i.e., an array of `char *` which just has one `NULL` entry (not a null pointer, but a valid `char**` whose first element is `NULL`.

If `execve` fails for some reason, you should indicate failure on standard error; use the `perror` function, following the example below. (Note that `perror` is only for reporting errors. The man pages for perror and errno should help, if you're confused.)

Note that `execve` doesn't do any of the fancy `PATH` lookup that a real shell does, so we'll have to use explicit paths to name programs.

Examples

When you are done, you should able to have the following interaction, where `^D` represents pressing control-D in your terminal (which sends an EOF).

```
¢ /bin/echo hello
hello
¢ /bin/nonesuch
ish: command error: No such file or directory
¢ ^D
Goodbye!
```

2 Printing exit status

Every command has an *exit status*, a number between 0 and 255. Check out `man 3 exit` (where the `3` specifies a section of the manpages, so you see the C function and not the shell command).

A process exits with `0` or `EXIT_SUCCESS` (which is defined to be 0 in `/usr/include/stdlib.h`) to indicate success; anything else indicates failure. While `main` returns an `int` (aka a four byte integer value) on our server, it's convention to stick to values between 0 and 255, as historically different operating systems could do different things.

Your next task is to give an informative message when the command exits with failure. Remember to use `fprintf` with `stderr` as the output stream.

Important note: You can't just use the raw return value from `waitpid`. The `DESCRIPTION` section of `waitpid`'s manpage contains important information about how to extract the exit status from the result of `waitpid`. (Specifically, pay attention to `WIFEXITED` and `WEXITSTATUS`.)

Examples

Here's an interaction on the VM. On your machine, `tar` might give a different error message.

```
¢ /usr/bin/tar
/usr/bin/tar: You must specify one of the ‘-Acdtrux’ or ‘--test-label’  options
Try ‘/usr/bin/tar --help’ or ‘/usr/bin/tar --usage’ for more information.
ish: status 2
¢ /usr/bin/true
¢ /usr/bin/false
ish: status 1
¢ /bin/nonesuch
ish: command error: No such file or directory
¢ ^D
Goodbye!
```

3 Running background jobs

Now that your shell can run and report on jobs in the foreground, it's time to support running background tasks. Like in a real shell, we'll type "&" at the end of a command to mark it as "asynchronous", i.e., a command that should run in the background while the shell continues and accepts the user's next input.

I've provided code that does the parsing logic for you: it sets the variable `run_in_background` to 1 (true) when ish should run the command in the background, and it will be 0 (false) for a normal (synchronous command). It also extracts the command and stores it in a variable `user_command` (**Hint:** you'll need this, too). This logic is right above where you solved tasks 1 and 2.

You don't want to wait for background jobs. Instead, you'll add them to a list (using the provided function `add_job`) so that you can keep track of them.

Hint: Yes, you'll need to modify the code you've already written to complete this task.

Examples

To test background jobs, we need a program that takes some time. A nice way to do this is to write a custom test program—we've provided `snooze.c`, which you should make sure is compiled. In the following example, we run `ish` first running `snooze` in the foreground, then in the background—while typing `/bin/echo hi` as `snooze` is running. Notice how `snooze`'s output is interleaved with our input!

```
¢ ./snooze
Taking a nap...zzzz...zzzz......yawn! What nice nap.
¢ ./snooze &
¢ Taking a nap.../bin/eczzzz...ho hi
hi
¢ zzzz......yawn! What a nice nap.
^D
Goodbye!
```

4 Waiting for background jobs

Next, we should make our shell wait for background jobs to complete before it fully exits. To see why, look at the following interaction:

```
$ ./ish
¢ ./snooze &
¢ Taking a nap...^D
Goodbye!
$ zzzz...zzzz......yawn! What a nice nap.
```

Here `$` is our *actual* shell prompt. And look: somebody is snoring in our terminal!

There are two `TODO` marks for task 4: one in `main` and one in `check_jobs`.

First, let's address the one in `main`. If there are any background jobs, then you should output to stderr: `"Jobs are still running...\n"` and call `check_jobs`.

The `check_jobs` function should iterate through every job in the list, using the `waitpid` system call to see if the job has terminated. If it's terminated successfully, it should print out "job `COMMAND` complete" on its own line; if it ended unsuccessfully, it should print out "job `COMMAND` status `STATUS`". Here `COMMAND` is the command name (i.e., `job->command`) and `STATUS` is the exit status.

Hint: For now, we want to use `waitpid` without any options (i.e., `options = 0`).

Examples

Here, we run `snooze` in the background and immediately exit `ish`. You can see `snooze`'s snoring and wake-up, followed by its wakeup.

```
¢ ./snooze &
¢ Taking a nap...^D
Jobs are still running...
zzzz...zzzz......yawn! What a nice nap.
job './snooze' complete

Goodbye!
```

Here's another, running `snooze 4` followed by `snooze`, both in the background. Note that jobs are waited for in decreasing recency:

```
¢ ./snooze 4 &
¢ Taking a nap..../snoozzzzz...e &
¢ Taking a nap...zzzz...zzzz...^D
Jobs are still running...
...yawn! What a nice nap.
zzzz......yawn! What a nice nap.
job './snooze' complete
job './snooze 4' status 4

Goodbye!
```

5 Reporting on Background Job Statuses

We'd like to update the user about background jobs as they complete. To start with, have main call `check_jobs` before prompting the user and reading the line (`check_jobs(WNOHANG)`).

Now, alter `check_jobs` to support the new option! You'll need to make a few changes:

1. pass the options argument to `waitpid`,

2. save the `pid_t` returned from `waitpid` (`waitpid` does not always return a process ID), and
3. handle the case when the process is still running (i.e., `waitpid` returns 0): print `job 'COMMAND' still running`.

Examples

Here we run a command in the background and hit return occasionally over the course of five seconds. Note that `/bin/sleep` is a real, pre-existing utility that's different from the `./snooze` helper we provided; check `man sleep` for more information:

```
¢ /bin/sleep 5 &
job '/bin/sleep 5' still running
¢
job '/bin/sleep 5' still running
¢
job '/bin/sleep 5' still running
¢
job '/bin/sleep 5' complete
¢ ^D
Goodbye!
```

Here we run two commands in the background, hitting return occasionally.

```
¢ /bin/sleep 5 &
job '/bin/sleep 5' still running
¢ /bin/sleep 3 &
job '/bin/sleep 3' still running
job '/bin/sleep 5' still running
¢
job '/bin/sleep 3' still running
job '/bin/sleep 5' complete
¢
job '/bin/sleep 3' complete
¢ ^D
Goodbye!
```

6 Reporting only when something changed

It's annoying to update the user unnecessarily—we should only report on background jobs when something has changed. To do so, we'll set up a *signal handler* for the `SIGCHLD` signal. Whenever a child process terminates, the parent process receives a `SIGCHLD` signal. By default, processes ignore these signals... but we'll use them to more cleverly notify the user of changes.

To start, you need to set up the new signal handler. Signal handlers are very restricted—you shouldn't run a lot of code in them! The standard thing to do is to set a global variable that says, "Hey, something happened!"

that your program checks at appropriate points. To do this, first define a global `int` variable that defaults to 0. Then define a handler function (takes an `int` and returns `void`) just above the `setup_signal_handlers`; your handler should set your global variable to 1 to indicate that `SIGCHLD` arrived.

Next, you need to modify `setup_signal_handlers` to call your new signal handler when the `SIGCHLD` signal arrives. Currently, `setup_signal_handlers` installs a signal handler to ignore `SIGINT` (i.e., control-C). You'll want to install a second signal handler (after the first, under the `TODO #6` comment). To do this, you'll need to:

1. Zero out the action struct with the command `sigemptyset(&action.sa_mask);` (man page).
2. Set its `sa_flags` to `SA_RESTART`. (If you *don't* set this flag, your shell might behave strangely when `SIGCHLD` comes in the middle of another system call.)
3. Set `action.sa_handler` to the function you want to call when the signal is received.
4. Change the signal action with `sigaction` (man page).

Finally, modify your `main` function to use your global variable to condition whether or not you check for jobs before prompting. Your program should now only report on background jobs if the `SIGCHLD` signal was received since a report was previously made.

Examples

Here we run a background command that will sleep for five seconds. We hit return a few times and get *no* updates. After waiting four or five seconds, we hit return again and *do* get an update.

```
¢ /bin/sleep 5 &
¢
¢
¢
¢
job '/bin/sleep 5' complete
¢ ^D
Goodbye!
```

Here's another example, where we run a command in the interim, and then wait several seconds before hitting return gain.

```
¢ /bin/sleep 5 &
¢ /bin/echo hi
hi
job '/bin/sleep 5' still running
¢
job '/bin/sleep 5' complete
¢ ^D
Goodbye!
```

Notice that we get the job update after the call to `echo`. Why? When `echo` terminates, it will send its own `SIGCHLD`, which will get caught by our handler and cause us to update the user about which tasks are running.

Feedback

Please remember to include a file called `feedback.txt` in your submission that answers the following questions:

1. How long did each of you spend on this assignment?
2. Any comments on this assignment?

As always, how you answer these questions **will not affect your grade**, but whether you answer them will.

Submission

Submit your `ish.c` and `feedback.txt` files as one submission on Gradescope. And remember to tag your partner as a collaborator! Note that the autograder for this assignment is very picky. If you aren't passing all the test cases, compare your output carefully to the expected output (check for linebreaks, extra spaces, missing quotes, etc.).