

Assignment 7: Shell Lab

Due: Tuesday, April 6, 2021 at 11:59pm PDT

In this homework, you will be building out the core system-call logic of an interactive shell. You may choose your partner for this assignment, but, as usual, they must be in your learning community.

The material for this lab in a tar file, available on the course web page or on the course VM in the `/data` directory. You can unpack it directly by going to your home directory and running:

```
tar xvf /data/shell-handout.tar
```

The directory `shell-handout` will be created, with 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).

When you have completed this homework, you will turn in two files: `ish.c` and `feedback.txt`. Submit it in the usual way on Gradescope, as a joint submission. You may, of course, submit several times—just be sure that all the submissions are the same team and all submissions include both files.

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 three parts, which we list here in order of importance (and reverse order of appearance in the file):

- The `main` function, which consists of a *read/evaluate* loop. It calls `init` to set things up, then it enters a loop where it prompts and uses `getline` to read a line from the user, and then it parses the line into a command and arguments.
- 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. Run background jobs in the background. (`main`)
4. Wait for background jobs to finish before exiting the shell. (`main`, `check_jobs`)
5. Check on jobs before each prompt. (`main`, `check_jobs`)
6. Only check on jobs when something has changed. (`main`, `init`)

We recommend that you read through the entire document, but that you not move on from one task until you're confident you have the right behavior. Each task gives some examples of correct behavior at the end. There are 52 points available total.

Note that, throughout, the shell itself only prints to *standard error*, a special output stream that's different from *standard output*. 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. You should do that, too.

1 Running the command

(10 points)

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

```
int num_words = 0;
char **args = parse_line(buf, len, &num_words);
assert(args);
assert(args[num_words] == NULL);

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

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

1. Use the `fork` system call to create a new process.
2. In the child process, use the `execve` system call to run the given program.
3. In the parent process, use the `waitpid` system call to wait for the child process to complete.

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

In particular, `execve` is tricky to use. You should use the first argument of `args` as the command, but `execve` needs all of the `args` as the second argument. You should use an empty environment, i.e., an array of `char *` which just has one `NULL` entry.

If `execve` fails for some reason, you should indicate failure on standard error; use `perror`, 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 be using explicit paths to name programs.

Examples

When you are done, you should be 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

(8 points)

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`, it's best to stick to values between 0 and 255, as different operating systems 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.

The `DESCRIPTION` section of `waitpid`'s manpage contains important information about how to extract the exit status from the result of `waitpid`.

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 '-Acdrux' 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
⌘ ^D
Goodbye!
```

3 Running background jobs

(10 points)

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

We’ve already done the parsing logic for you: the variable `bg` will be set to 1 when the command should be run in the background, and it will be 0 when it’s not. (The logic is right above where you solved tasks 1 and 2).

You’ll want to change a few things: you *don’t* want to wait for background jobs, and you’ll need to add them to the job list for the next tasks. (If you don’t do it now, you’ll need to do it later.)

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

(8 points)

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 `check_jobs`.

First, let’s address the one in `main`. If there are any background jobs, you should output on a new line, “Jobs are still running...”, and then give output information for each remaining job as it completes: you can use `check_jobs` to wait for each job in turn.

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.

For now, we want to use waitpid without any options (i.e., options = 0); later on, we’ll reuse this code setting options WNOHANG.

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 Checking for completed jobs

(8 points)

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. Specify the options as WNOHANG.

Now, change check_jobs to support the new option! You’ll need to make two changes: first, make sure your options argument is going to the waitpid call. Then make sure that you’re saving the pid_t returned from waitpid—when you pass WNOHANG, it might not be a process ID at all! Read waitpid’s manpage closely. Finally, you’ll need to add a case that prints out “job ‘COMMAND’ still running”—but only when WNOHANG is set.

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 Checking only when something changed

(5 points)

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 record that something happened, and so the user should be updated.

To start, you need to update `init` with a 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.

Define a handler function (takes an `int` and returns `void`) and a global `int` variable, which defaults to 0. Your handler should set it to 1 to indicate that `SIGCHLD` arrived.

Currently, `init` installs a signal handler to ignore `SIGINT` (i.e., control-C). You'll want to set the `action`'s `sa_handler` field to your function, and you'll want to 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.)

Finally, use your global variable to condition whether or not you check for jobs before prompting.

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 again.

```
$ /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

(3 points)

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?
3. Did you attend your learning community?

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!