

CS 214 Homework 3

Fall 2021

Introduction

In this assignment, you'll be working with processes and signals by implementing a shell. Your primary `.c` file should be called `shell.c`.

- Please be careful to follow the I/O formats exactly.
- You can assume that inputs will be correctly formatted and that all commands will be a single line of input.

Program execution

Your shell should print a `>` prompt (greater sign and a space) and read input from the user. It should be able to execute other programs, possibly with command-line options, whether they're specified with an absolute path:

```
> /bin/ls -l
```

or a relative path:

```
> ../src/hello
```

If the file doesn't exist, print the path and executable, followed by "No such file or directory". For example,

```
> ../src/hello
../src/hello: No such file or directory
```

If a command is given with no absolute/relative path (e.g., "ls"), and it's not a built-in command (see below), your shell should look in `/usr/bin` and `/bin` (in that order) and execute the first such program found. If it's not found in either location, your shell should print "commandName: command not found" (with `commandName` replaced by the actual command). For example,

```
> burrito
burrito: command not found
```

If the command line ends in an ampersand, your shell should run the program in the background. Otherwise, it should run it in the foreground and wait for the executed program to complete before reading another line of input.

Each line of input defines a job and should be assigned a job ID, starting at 1. Job IDs are written with a %, as in %1, to distinguish them from process IDs (PIDs). Jobs also have a status of running, stopped, or terminated.

If a command is run in the background, you should print the job ID and process ID, as in:

```
> ./hello &
[1] 4061
```

The command will consist of a program and arguments separated by whitespace (possibly leading or trailing). No commands or arguments will use quoted strings or escape characters. For example, these are all valid inputs (with spaces shown explicitly):

```
>   ./hello&
> ./hello&
> ./hello
>   ./hello
>   ./hello-a-b-c&
```

Your shell should reap all zombie children created by any executed process. If any child process terminates due to an unhandled signal, you should print a message with the relevant job ID, process ID, and signal:

```
[1] 1464 terminated by signal 2
```

Signals

- `ctrl-c` should send `SIGINT` to the foreground job and any of its child processes
- `ctrl-z` should send `SIGTSTP` to the foreground job and any of its child processes

If there is no current foreground job, `ctrl-c` and `ctrl-z` do nothing.

`SIGINT` (by default) will cause the receiving job to terminate. It should not terminate your shell.

`SIGTSTP` will suspend the receiving job until it receives the `SIGCONT` signal. It should not suspend your shell. In your shell, you can resume a suspended job in the background (with built-in command `bg`) or foreground (with built-in command `fg`), or kill it (with built-in command `kill`).

Built-in commands

Your shell should recognize these built-in commands:

- `bg <jobID>`
Run a suspended job in the background.

- `cd [path]`

Change current directory to the given (absolute or relative) path. If no path is given, use the value of environment variable `HOME`. Your shell should update the environment variable `PWD` with the (absolute) present working directory after running `cd`.

- `exit`

Exit the shell. The shell should also exit if the user hits `ctrl-d` on an empty input line.

When the shell exits, it should first send `SIGCONT` followed by `SIGHUP` to any stopped jobs, and `SIGHUP` to any running jobs.

- `fg <jobID>`

Run a suspended or background job in the foreground.

- `jobs`

List current jobs, including their job ID, process ID, current status, and command. If no jobs exist, this should print nothing.

```
> jobs
[1] 1432 Running ./test1 &
[2] 1487 Running ./hello &
[3] 1504 Stopped ./foo
```

Background jobs should have an ampersand at the end of the command, whether they were initially run that way or were suspended and then placed in the background.

- `kill <jobID>`

Send `SIGTERM` to the given job.

Examples

To get into the shell, we run

```
$ ./shell  # on the normal bash prompt
>         # now at your shell prompt, waiting for input
```

Then we can run a program:

```
> /bin/echo hello
hello
>
```

We can run something longer and kill it with `ctrl-c`:

```
> /bin/sleep 100
^C
[1] 1464 terminated by signal 2
>
```

We can run `sleep` again and suspend it with `ctrl-z`:

```
> /bin/sleep 100
^Z
>
```

Then jobs should show the suspended job:

```
> jobs
[1] 1234 Stopped /bin/sleep 100
>
```

Then we could foreground it (fg %1), background it (bg %1), or kill it (kill %1).

```
> bg %1
> jobs
[1] 1234 Running /bin/sleep 100 &
> kill %1
[1] 1234 terminated by signal 15
> jobs
>
```

Running it with & runs it in the background:

```
> /bin/sleep 100 &
> jobs
[1] 1248 Running /bin/sleep 100 &
>
```

If we foreground it, we have to wait for it to complete (or suspend it again):

```
> /bin/sleep 100 &
> jobs
[1] 1248 Running /bin/sleep 100 &
> fg %1      # ...and wait...
>
```

Hints

- Read <https://resources.cs.rutgers.edu/docs/preventing-fork-bomb-on-linux/> to avoid accidentally exhausting all resources in the case of a buggy fork loop.
- Some possibly useful functions: access, getenv, setenv, wait, waitpid, kill, signal, fork, execve, setpgid, sigprocmask, sleep.
- man 7 signal describes all signals.
- Your shell needs to keep track of the processes it creates. In addition to wait/waitpid, it may be useful to handle SIGCHLD.
- You will probably want to handle SIGINT and SIGTSTP.
- Remember that signals are (and should be) sent to all processes in a process group.
- Ordering of process execution after fork is not guaranteed, so the child process could terminate before the parent gets a chance to execute a single further line of code.
- If there are regions of your code where receiving a signal could be disruptive, you can block/unblock signals using sigprocmask. Child processes inherit the blocked signal settings from their parent.

- Once you have most parts working, write test programs and ensure your shell handles them correctly (e.g., a program that infinite loops, terminates quickly, segfaults, sends SIGINT to itself, creates many child processes, creates orphan child processes, creates zombie child processes, etc.).
- Note that many things that work in bash won't work in your shell, since we're only implementing a subset of bash's functionality. For example, "`echo hello`" should work, but "`echo $HOME`" will just print "`$HOME`". Similarly, "`ls shell.c`" will show `shell.c`, but "`ls *.c`" will give an error that there's no file named "`*.c`". See `man bash` for more details on bash's functionality.

Compiling and testing

You should use similar CFLAGS to gcc as in previous homeworks, e.g.:

```
-g -Wall -Wvla -fsanitize=address
```

You should create a Makefile so that running "`make`" or "`make all`" builds your program.

Submission

If you develop on your local machine, please be sure to test your code on ilab before submitting.

Please submit the assignment on Canvas as a tar file `hw3.tar` that, when expanded, produces a `hw3` directory (possibly with additional `.c/.h` files):

```
hw3
├─ Makefile
└─ shell.c
```

If you work in a group, please include a `README.txt` file with the names and netIDs of everyone in your group. Only one person needs to submit on Canvas.