

Shell

Part 1 due 2019-02-11 23:59

Graded files:

- shell.c

Part 2 due 2019-02-18 23:59

Graded files:

- shell.c

Content

Backstory

Notices

Overview

Interaction

Commands

Built-in Commands

External Commands

Logical Operators

Memory

Week 2 Only - Additional Built-In Commands

Grading

Learning Objectives

The learning objectives for Shell are:

- Learning How a Shell Works
- Fork, Exec, Wait
- Signals
- Processes
- Zombie Processes

Backstory

Well, we'll keep it short – you got fired from Macrohard. Your boss brought you in for a code review and was more than disappointed. Apparently, she wanted a C++ style vector: we didn't get the memo. Now, you've decided to work for *insert hot tech company here*, and you got the job! However, there's a catch - all newhires in *insert hot tech company here* apparently have to go through a newcomers test if they want to keep their jobs. The task? Write a shell. So, you're going to drop a 🔥 🔥 shell that is so fancy that your boss will not just keep you in the company, they'll immediately give you a pay raise as well.

The basic function of a shell is to accept commands as inputs and execute the corresponding programs in response. You will be provided the `vector`, `sstring` and `format.c` libraries for your use. Hopefully, this will make things right and you can secure your foothold at *insert hot tech company here*. Feel free to refer to the Unix shell as a rough reference.

Notices

Fork Bombs



fork

If your code fork-bombs on *any* autograder, then you will automatically fail this MP. Please make sure that your code is correct before committing your code for the autograder. (<https://cs241.cs.illinois.edu/assignments/shell.html>)

ulimit

To prevent you from fork bombing your own VM, we recommend looking into <https://ss64.com/bash/ulimit.html> for how many times you can fork.

system

Since a learning objective of this assignment is to use the fork-exec-wait pattern, if you use `fork()` you will automatically fail this MP. (<http://man.3.org/linux/man3/fork.3.html>)

Formatting

Since this MP **requires** your programs to print a variety of things like error messages, we have provided you with our own highly customized formatting library. You should not be printing out to stdout and stderr at all; instead, all output and errors should be printed using the functions provided in `format.c` and `format.h`. In `format.h` you can find documentation about what each function does, and you should use them whenever appropriate.

Note: don't worry if you don't use all of the functions in `format.c`, but you should use them whenever their documented purpose matches the situation.

Overview

The shell is responsible for providing a command line for users to execute programs or scripts. You should be very familiar

bash

with `bash`, which will be the basis for your own shell. (https://www.gnu.org/software/bash/manual/html_node/)

Starting Your Shell

The shell should run in a loop like this executing multiple commands:

- Print a command prompt
- Read the command from standard input
- Print the PID of the process executing the command (with the exception of built-in commands), and run the command

The shell must support the following two optional arguments:

History

`-h` takes the filename of the history file. The shell should load in the history file as its history. If the file does not exist, you should treat it as an empty file and write all the commands that were executed in the terminal to the output file on exit. Upon exit, the exact same history file should be updated, even if the shell is in a different working directory than where it started.

```
./shell -h <filename>
```

The format of the history file stored should be exactly the same as a script file. See below for details.

If the `-h` flag is not specified, the shell will still keep a history of commands run, but will not read/write from/to a history file. Just think of it like private browsing mode for your terminal.

File

`-f` takes the name of the file to be executed by the shell. The shell will both print and run the commands in the file in sequential order until the end of the file. See the following example file and execution:

```
commands.txt :
```

```
cd cs241
echo Hey!
```

```
./shell -f commands.txt
(pid=1234)/home/user$ cd cs241
(pid=1234)/home/user/cs241$ echo Hey!
Command executed by pid=1235
Hey!
```

You have been given a sample script file `test_file.txt`. Your history files and script files should be formatted in the same manner.

If the user supplies an incorrect number of arguments, or the script file cannot be found, your shell should print the appropriate error from `format.h` and exit.

```
getopt
The (http://www.diyet/main/3/gedp0-
```

Interaction

Prompting

When prompting for a command, the shell will print a prompt in the following format (from `format.h`):

```
(pid=<pid>)<path>$
```

`<pid>` is the current process ID, and `<path>` is a path to the current working directory. Note the lack of a newline at the end of this prompt.

Reading in the Command

The shell will read in a command from `stdin` (as a file if one was specified) (`http://www.diyet/main/3/gedp0-`).

Running the Command

The shell should run the command that was read in previously.

If the command is run by a new process, the PID of the process should be printed like this:

```
Command executed by pid=<pid>
```

This should be printed by the process that will run the command, before any of the output of the command is printed (prints to be used are in `format.c/h`).

Keeping History

Your shell should store the command that the user entered, so the user can repeat it later if they wish. Every command should be stored unless otherwise noted. A vector may be useful here.

Backgrounding

An *external* command suffixed with `&` should be run in the background. In other words, the shell should be ready to take the next command before the given command has finished running. There is no limit on the number of background processes you can have running at one time (aside from any limits set by the system).

There may or may not be a single space between the rest of the command and `&`. For example, `pwd&` and `pwd &` are both valid.

Since spawning a background process introduces a race condition, it is okay if the prompt gets misaligned as in the following example:

```
(pid=1873)/home/user$ pwd &
Command executed by pid=1874
(pid=1873)/home/user$
/home/user
```

When I type, it shows up on this line

While the shell should be usable after calling the command, after the process finishes, the parent is still responsible for waiting on the child (hint: catch a signal). Avoid creating zombies! Think about what happens when multiple children finish around the same time.

Backgrounding will **not** be chained with the logical operators.

Catching Ctrl+C

Usually when we do `Ctrl+C`, the current running program will exit. However, we want the shell itself to ignore the `Ctrl+C` signal (`SIGINT`). Instead, it should check if there is a currently running foreground process, and if so, it should kill that foreground process using `SIGINT` (the `kill()` function might come in handy, here and elsewhere).

Note that we want this signal sent to the foreground process, not to any backgrounded processes. As such, you will want to

```
setpgid
(http://man7.org/linux/man-
pages/man2/setpgid.2.html)
use to assign setpgid to child and process to its own process group when forking:
```

```
pid_t pid = fork();
if (pid > 0) {
    // ...

    // assign the child's process group id to be its pid
    if (setpgid(pid, pid) == -1) {
        print_setpgid_failed();
        exit(1);
    }

    // ...
} else if ( ... ) {
    // ...
}
```

Yes, we are giving you code that you will want to put somewhere in your program. Use it appropriately for free points!

Commands

Shell supports two types of commands: built-in and external (i.e. non-built-in). While built-in commands are executed without creating a new process, an external command *must* create a new process to execute the program for that particular command.

Command arguments will be space-separated without trailing whitespace. Your shell does not need to support quotes (for example, `echo "hello there"`).

Built-in Commands

There are several built-in commands your shell is expected to support.

`cd <path>`

Changes the current working directory of the shell to `<path>` . Paths not starting with `/` should be followed relative to the current directory. If the directory does not exist, then print the appropriate error. Unlike your regular shell, the `<path>` argument is mandatory here. A missing path should be treated as a nonexistent directory.


```
(pid=1234)/home/user$ cd code
(pid=1234)/home/user/code$ cd imaginary_directory
imaginary_directory: No such file or directory
(pid=1234)/home/user/code$
```

There is a system call that may be helpful here.

!history

Prints out each command in the history, in order.

```
(pid=1234)/home/user$ !history
0  ls -l
1  pwd
2  ps
(pid=1234)/home/user$
```


 This command is not stored in history.


#<n>

Prints and executes the n th command in history (in chronological order, from earliest to most recent), where n is a non-negative integer. Other values of n will not be tested. The command run should be stored in the history. If n is not a valid index, then print the appropriate error and do not store anything in the history.

The following example assumes a fresh history:

```
(pid=1234)/home/user$ echo Echo This!
Command executed by pid=1235
Echo This!
(pid=1234)/home/user$ echo Another echo
Command executed by pid=1236
Another echo
(pid=1234)/home/user$ !history
0  echo Echo This!
1  echo Another echo
(pid=1234)/home/user$ #1
echo Another echo
Command executed by pid=1237
Another echo
(pid=1234)/home/user$ #9001
Invalid Index
(pid=1234)/home/user$ !history
0  echo Echo This!
1  echo Another echo
2  echo Another echo
(pid=1234)/home/user$
```

 Print out the command before executing if there is a match.

 The `#<n>` command itself is **not** stored in history, but the command being executed (if any) is.

!<prefix>

Prints and executes the last command that has the specified prefix. If no match is found, print the appropriate error and do not store anything in the history. The prefix may be empty. The following example assumes a fresh history:

```
(pid=1234)/home/user$ echo Echo This!
Command executed by pid=1235
Echo This!
(pid=1234)/home/user$ echo Another echo
Command executed by pid=1236
Another echo
(pid=1234)/home/user$ !e
echo Another echo
Command executed by pid=1237
Another echo
(pid=1234)/home/user$ !echo E
echo Echo This!
Command executed by pid=1238
Echo This!
(pid=1234)/home/user$ !d
No Match
(pid=1234)/home/user$ !
echo Echo This!
Command executed by pid=1239
Echo This!
(pid=1234)/home/user$ !history
0      echo Echo This!
1      echo Another echo
2      echo Another echo
3      echo Echo This!
4      echo Echo This!
(pid=1234)/home/user$
```

! Print out the command before executing if there is a match.

! The !<prefix> command itself is **not** stored in history, but the command being executed (if any) is.

exit

(<https://linux.die.net/man/3/exit>)

exit

The shell will exit once it receives the `Ctrl-D` (EOF). An EOF is sent by typing `Ctrl-D` from your terminal. It is also sent automatically from a script file (as used with the `-f` flag) once the end of the file is reached. This should cause your shell to exit with exit status 0.

exit

If there are currently stopped or running background processes when your shell receives `Ctrl-D` (EOF), you should kill and cleanup each of those children before your shell exits. You do not need to worry about SIGTERM.

! If you don't handle EOF to exit, you will fail many of our test cases!

exit

! Do **not** store (http://linux.die.net/man/3/exit)

Invalid Built-in Commands

cd

You should be printing appropriate errors in cases where built-in commands fail; for example, if the user tries to `cd` a nonexistent directory. (<https://linux.die.net/man/3/cd>)

```
(pid=1234)/home/user$ cd /imaginary_directory
/imaginary_directory: No such file or directory
(pid=1234)/home/user$
```

External Commands

For commands that are not built-in, the shell should consider the command name to be the name of a file that contains executable binary code. Such a code must be executed in a process different from the one executing the shell. You must use

```
fork exec wait waitpid
(https://linux.die.net/man/3/waitpid)
```

The fork/exec/wait paradigm is as follows: [fork](https://linux.die.net/man/3/fork) (http://linux.die.net/man/3/fork) to create a child process. The child process must execute the command with

```
wait
exec*, while the parent must wait (http://linux.die.net/man/3/wait) for the child to terminate before printing the next prompt.
```

You are responsible of cleaning up all the child processes upon termination of your program. It is important to note that, upon

a successful execution of the command, [exec](https://linux.die.net/man/3/exec) (http://linux.die.net/man/3/exec) only returns to the child process, [exec](https://linux.die.net/man/3/exec) (http://linux.die.net/man/3/exec) only returns to the child process, [exec](https://linux.die.net/man/3/exec) (http://linux.die.net/man/3/exec)

when the command fails to execute successfully. If any of [fork](https://linux.die.net/man/3/fork) (http://linux.die.net/man/3/fork), [exec](https://linux.die.net/man/3/exec) (http://linux.die.net/man/3/exec), or [wait](https://linux.die.net/man/3/wait) (http://linux.die.net/man/3/wait) fails, an error message should

be printed. The child should [exit](https://linux.die.net/man/3/exit) (http://linux.die.net/man/3/exit) with exit status 1 if it fails to execute a command.

Some external commands you may test to see whether your shell works are:

```
/bin/ls
echo hello
```

It is good practice to flush the standard output stream before the fork to be able to correctly display the output.

!! Please read the disclaimer at the top of the page! We don't want to have to give any failing grades. !!

Logical Operators

bash

Like [your shell should support](https://linux.die.net/man/1/bash) (http://linux.die.net/man/1/bash) and ; in between two commands. This will require only a minimal amount of string parsing that you have to do yourself.

Important: each input can have at most one of && , || , or ; . You do *not* have to support chaining (e.g. x && y || z; w).

Important: you should *not* try to handle the combination of the !history , #<n> , !<prefix> , or [command substitution](https://en.wikipedia.org/wiki/Command_substitution) (https://en.wikipedia.org/wiki/Command_substitution) with any logical operators. Rather, you can assume these commands will always be run on a line by themselves.

AND

&& is the AND operator.

Input: x && y

- The shell first runs x , then checks the exit status.
- If x exited successfully (status = 0), run y .

short-circuiting

(https://en.wikipedia.org/wiki/Short-circuit_evaluation)

- If x did not exit successfully (status ≠ 0), do *not* run y . This is also known as

This mimics short-circuiting AND in boolean algebra: if x is false, we know the result will be false *without* having to run y .

? This is often used to run multiple commands in a sequence and stop early if one fails. For example, make && ./shell

will run your shell only if [make](https://linux.die.net/man/3/make) (https://linux.die.net/man/3/make) succeeds.

OR

|| is the OR operator.

Input: x || y

- The shell first runs `x`, then checks the exit status.
- If `x` exited successfully, the shell does *not* run `y`. This is short-circuiting.
- If `x` did not exit successfully, run `y`.

Boolean algebra: if `x` is true, we can return true right away *without* having to run `y`.

? This is often used to recover after errors. For example, `make || echo 'Make failed!'` will run `echo` (http://linux.die.net) if `make` does not succeed.

Separator

`;` is the command separator.

Input: `x; y`

- The shell first runs `x`.
- The shell then runs `y`.

? The two commands are run regardless of whether the first one succeeds.

Memory

As usual, you may not have any memory leaks or errors.

Week 2 Only - Additional Built-In Commands

`ps`
(<https://linux.die.net/man/1/ps>)

Like our good old `ps` (http://linux.die.net/man/1/ps), your shell should print out information about all currently executing processes. You should include the shell and its immediate children, but don't worry about grandchildren or other processes. Make sure you use `print_process_info_header()` and `print_process_info()` (and maybe some other helper functions)!

Note: while `ps` (http://linux.die.net/man/1/ps) is normally used directly in a shell, it is a built-in command for your shell. (This is not "execing" (http://linux.die.net/man/1/exec) it in the code. Thus you may have to keep track of some information for each process.)

Your version of the `ps` (http://linux.die.net/man/1/ps) should print the following information for each process:

- PID: The pid of the process
- NLWP: The number of threads currently being used in the process
- VSZ: The program size (virtual memory size) of the process, in kilobytes
- STAT: The state of the process
- START: The start time of the process
- TIME: The amount of cpu time that the process has been executed for
- COMMAND: The command that executed the process

Hint: You may find the `/proc` filesystem to be useful, as well as the man pages for it.

Some things to keep in mind:

- The order in which you print the processes does not matter.
- The 'command' for `print_process_info` should be the full command you executed. The `&` for background processes is optional. For the main shell process *only*, you do not need to include the command-line flags.

- You may not exec the `ps` (http://linux.die.net/man/1/ps) binary to complete this part of the assignment.

Example output of this command:


```
(pid=25497)/home/user$ ps
PID      NLWP    VSZ     STAT    START   TIME    COMMAND
25498    1       7328    R       14:03   1:35    dd if=/dev/zero of=/dev/null &
25501    1       7288    S       14:04   0:00    sleep 1000 &
25497    1       7484    R       14:03   0:00    ./shell
```

pfd <pid>

To impress your boss even further, you've decided that you're going to implement a new command that doesn't exist in the regular shell.

The `pfd` (print file descriptor) command that you have come up with will print the following information about each of the open file descriptors of a process:

- The file descriptor number used in the process
- The position of the file descriptor for the process
- The *real path* of the file

Note: This includes `stdin`, `stdout`, and `stderr`.

Hint: You may find the `/proc` filesystem to be useful here as well.

Example output of this command:

```
(pid=29709)/home/user$ pfd 29719
FD_NO    POS      REALPATH
0         0        /dev/pts/2
1         0        /dev/pts/2
2         0        /dev/pts/2
3         4096     /home/user/doc.txt
```

kill <pid>

The ever-useful panic button. Send `SIGTERM` to the specified process.

Use the appropriate prints from `format.h` for:

- Successfully sending `SIGTERM` to process
- No process with `pid` exists
- `kill` was ran without a `pid`

stop <pid>

This command will allow your shell to stop a currently executing process by sending it the `SIGTSTP` signal. It may be resumed by using the command `cont`.

Use the appropriate prints from `format.h` for:

- Process was successfully sent `SIGTSTP`
- No such process exists
- `stop` was ran without a `pid`

cont <pid>

This command resumes the specified process by sending it `SIGCONT`.

Use the appropriate prints from `format.h` for:

- No such process exists
- `cont` was ran without a `pid`

`kill`

Any `<pid>` used in `(http://linux.die.net/man/2/kill)` will kill the process that is a direct child of your shell or a non-existent process. You do not have to worry about killing other processes.

Grading

As you may notice, this MP is split up into two weeks:

- Week 1 (50%): Week 1 tests cover everything up until the week 2 section.
- Week 2 (50%): Week 2 tests cover everything covered in week 1 as well as the week 2 section.

