Shell

Part 1 due **2021-09-20 23:59**
Graded files:

- shell.c

Part 2 due **2021-09-27 23:59**
Graded files:

- shell.c

Content

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, they 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.h` 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.

Important Things to Note

Fork Bombs

🍴💣‼️

To prevent you from fork bombing your own VM, we recommend looking into `ulimit` (https://linux.die.net/man/3/ulimit). This will allow you to set a limit on how many times you can fork. Note that `ulimit` is terminal session specific, so you will need to (https://linux.die.net/man/3/ulimit)

- do it everytime you launch a terminal
- add this to your `~/.bashrc` file (feel free to look up online how to do so), so that it is run every time you log in to your VM.

Note that you should give it a more generous amount (say, 100-200), since the terminal will likely have background processes already running. If you give it too small a limit, you won't be able to launch anything, and you'll need to launch a new terminal.

If you happen to fork bomb your CS Cloud VM, please notify course staff in a private post with your VM number. Note that it may take up to a few hours for us to respond, so try not to fork bomb your VM.

Plan Before You Start

This assignment marks the beginning of a series of projects where you will be given mostly blank files without predefined functions to fill in. Most of the remaining MPs will challenge your design skills to create interesting utilities. Therefore, it is important that you **read the entirety of the documentation (including part 2), as well as the header files** to get a clear idea on what needs to be done. A few reminders about good coding and developing practices that will really help you in the rest of the semester:

- List down the features that you need to implement, as well as the gotchas. Make a to-do list to ensure you don't miss out anything.
- Plan out the entirety of your assignment. Create a skeleton of how your entire code will look like. This will prevent you from needing to restructure your entire code to add in a single new feature.
- Ensure that you fully understand the system calls/library functions you're using - the parameters, the return values, the possible errors, the gotchas and notes.
- Structure your code into modular functions. You do not want to debug a 1500 line `while` loop within `main`. (https://linux.die.net/man/1/while)
- Work incrementally. Implement a feature, test, debug, move on.
- Good naming and spacing will make your code much more readable.
- Try putting `TODO` comments in unfinished portions of your code. They are automatically highlighted in many text editors, which alerts you to incomplete code.

Do Not Use `system` (https://linux.die.net/man/3/system)

Since a learning objective of this assignment is to use the fork-exec-wait pattern, if you use `system` (https://linux.die.net/man/3/system), you will automatically fail this MP.

Input Formatting

Do not worry about irregular spacing in command inputs (i.e. extra whitespace before and after each token). This is considered undefined behavior and will not be tested. You are free to make your code as robust as you want, but we will only test the basic cases without irregular spacing (unless specified).

Output Formatting

Since this MP **requires** your shell and the programs you launch to print a variety of things like output messages and error messages, we have provided you with our own highly customized formatting library. You should not be printing out to `stdout` and `stderr` (https://linux.die.net/man/3/stdout) (https://linux.die.net/man/3/stderr) in your code. Instead, all output should be printed using the functions provided in `format.h`. In `format.h` you can find documentation about what each function does, and you should use them whenever appropriate.

If you place print statements in your debugging code, please remember to remove them before autograding, or use the `#define DEBUG` block to place your print statements.

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

Overview and To-Dos

The shell is responsible for providing a command line for users to execute programs or scripts. You should be very familiar with `bash` (https://linux.die.net/man/1/bash) by now, which will be the basis for your own shell. This is a 2 week MP, and the features you will need to implement are as follows:

Part 1

- Starting up a shell
- Optional arguments when launching shell
- Interaction
- Built-in commands
- Foreground external commands
- Logical operators
- `SIGINT` handling
- Exiting

Part 2

Everything from part 1, and:

- Background external commands
- `ps` (https://linux.die.net/man/1/ps)
- Redirection commands
- Signal commands

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, however, *the order of the arguments does not matter, and should not affect the functionality of your shell. Your shell should be able to handle having none, one or both of these arguments*.

History

Your shell should support storing the history of commands executed across shell sessions. The command is as follows:

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

When provided `-h`, the shell should load in the history file as its history. Upon exiting, the shell should *append the commands of the current session* into the supplied history file, even if the shell is in a different working directory than where it started. If the file does not exist, you should treat it as an empty file. The format of the history file stored should be exactly the same as a script file, where you list a series of commands to be executed. Example:

`history.txt`:

```
cd cs241
Hm
```



```
./shell -h history.txt
(pid=1234)/home/user/cs241$ echo Hey!
Command executed by pid=1235
Hey!
(pid=1234)/home/user/cs241$ exit
```


Updated `history.txt`:

```
cd cs241
Hm
echo Hey!
```

Notes:

- If the 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.
- Every command should be stored into the history file, unless specified.

File

Your shell should also support running a series of commands from a script file. The command is as follows:

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

When provided `-f`, your 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 (this means you can use your history file as a script file in `-f`).

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.

Tip: The `getopt` function may come in handy! (https://linux.die.net/man/3/getopt) 😄

Interaction Within Your Shell

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

Reading In Commands

The shell will read in a command from `stdin` (https://linux.die.net/man/3/stdin). If `-f` was specified, commands should be read from the specified file.

Command Types and Formats

Shell supports two types of commands: built-in and external (i.e. non-built-in). Built-in commands are part of the shell's code, and are executed without creating a new process. External commands *must* be executed by a new process, forked from your shell. If a command is not one of the built-in commands listed, it is an external command.

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

Running the Commands

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.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.

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

The shell will exit once it receives the `exit` command or once it receives an `EOF` **at the beginning of the line**. 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.

If there are currently stopped or running background processes when your shell receives `exit` or `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` or `exit`, you will fail many of our tests!

⚠️ Do **not** store `exit` in your history.

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 kill the currently-running foreground process (if one exists) using `SIGINT`. One way to do this is to use the `kill` (https://linux.die.net/man/3/kill) function in the `signal` (https://linux.die.net/man/3/signal) process PID when `SIGINT` is caught in your shell. However, when a signal is sent to a process, it is sent to all processes in its process group. In this assignment, the shell process is the leader of a process group consisting of all processes that are `fork` (https://linux.die.net/man/3/fork)ed from it. So another way to handle `Ctrl+C` is to simply *do nothing inside the handler for `SIGINT`* if it is caught in the shell - your shell will continue running, but `SIGINT` will automatically propagate to the foreground process and kill it.

However, since we want this signal to be sent to **only** the foreground process, but not to any backgrounded processes, you will want to use `setpgid` (https://linux.die.net/man/3/setpgid) to assign each background process to its own process group after forking. (Note: think about who should be making the `setpgid` call, and why.)

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 executed 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**.

Invalid Built-in Commands

You should be printing appropriate errors in cases where built-in commands fail; for example, if the user tries to `cd` into a nonexistent directory.

(https://linux.die.net/man/1/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/wait) (https://linux.die.net/man/3/waitpid)

The `fork/exec/wait` paradigm is as follows: (https://linux.die.net/man/3/fork) The child process must execute the command with `exec*`, while the parent must wait (https://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 `exec` command, (https://linux.die.net/man/3/exec) never returns to the calling process (https://linux.die.net/man/3/exec) only returns to the.

child process when the command fails to execute successfully. If any of `fork` (https://linux.die.net/man/3/fork), `exec` (https://linux.die.net/man/3/exec), `wait` (https://linux.die.net/man/3/wait), or `exit` (https://linux.die.net/man/3/exit) should be printed. The child should exit with 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
```

Tip: It is good practice to flush the standard output stream before the fork to be able to correctly display the output. This will also prevent duplicate printing from the child process.

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

Logical Operators

Like `bash` (https://www.gnu.org/software/bash/), your shell should support `&&` 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 `!?<substring>?` commands with any logical operators. Rather, you can assume these commands will always be run on a line by themselves.

AND

`&&` is the AND operator. Usage:

```
x && y
```

- The shell first runs `x`, then checks the exit status.
- If `x` exited successfully (status = 0), run `y`.
- If `x` did not exit successfully (status ≠ 0), do *not* run `y`. This is also known as short-circuiting (https://en.wikipedia.org/wiki/Short-circuit_evaluation)

```
(pid=27853)/home/user/semester/shell$ echo hi && echo bye
Command executed by pid=27854
hi
Command executed by pid=27855
bye


(pid=27879)/home/mkrzys2/fa19/shell$ cd /asdf && echo short-circuit
/asdf: No such file or directory!
```

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/ma succeeds.

Tip: You may want to look into the provided macros to read the status of an exited child.

OR

`||` is the OR operator. Usage:

```
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`.

```
(pid=27853)/home/user/semester/shell$ echo hi || echo bye
Command executed by pid=27854
hi
```

```
(pid=1234)/home/user$ cd /asdf || echo runMe
/asdf: No such file or directory
runMe
```


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` only if `make` (https://linux.die.net/man/3/make) fails.

Separator

`;` is the command separator. Usage:

```
x; y
```

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

```
(pid=27879)/home/user/semester/shell$ echo hi; echo bye
Command executed by pid=27883
hi
Command executed by pid=27884
bye
```

```
(pid=27879)/home/user/semester/shell$ cd /asdf; echo runMe
/asdf: No such file or directory
Command executed by pid=27884
runMe
```

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

Memory

As usual, you may not have any memory leaks or errors. Note that still reachable memory blocks do not count as memory leaks.

Background Processes

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 will be a single space between the rest of the command and `&`. For example, `pwd &` is valid while you need not worry about `pwd&`.

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
```

Note this is not the only way your shell may misalign.

While the shell should be usable after calling the command, after the process finishes, the parent is still responsible for waiting on the child. Avoid creating zombies! Do not catch `SIGCHLD`, as catching `SIGCHLD` comes with all sorts of caveats and subtleties that are hard to work around. Instead regularly check to see if your children need reaping (think about placement of this piece of code: where should you put this, and why). Think about what happens when multiple children finish around the same time, and what happens if a foreground/background process finish around the same time.

Backgrounding will **not** be chained with the logical operators nor with redirection operators.

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

Like our good old `ps` (https://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` (https://linux.die.net/man/1/ps) is normally a separate binary, it is a built-in command for your shell. (This is not "execing `ps`" (https://linux.die.net/man/1/ps) code. Thus you may have to keep track of some information for each process.)

Your version of the `ps` (https://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 (1 kilobyte = 1024 bytes)
- STAT: The state of the process
- START: The start time of the process. You will want to add the boot time of the computer, and start time of the process to calculate this. Make sure you are careful while converting from various formats - the man pages for `procfs` have helpful tips.

- TIME: The amount of cpu time that the process has been executed for. This includes time the process has been scheduled in user mode (`utime` (https://linux.die.ne and kernel mode (`stime` (https://linux.die.net/man/2/stime)
- COMMAND: The command that executed the process

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. Ensure that the 'command' does not have trailing whitespace at the end of it.
- You may not exec the `ps` (https://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   0:08    dd if=/dev/zero bs=1M c
ount=123456 of=/dev/null &
25501   1       7288    S       14:04   0:00    sleep 1000 &
25497   1       7484    R       14:03   0:00    ./shell
```

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

Redirection Operators

Your boss wants some way for your shell commands to be able to link together. You decide to implement `>>`, `>`, and `<`. 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 `cd` (https://linux.die.net/man/1/ `history`, `#<n>`, `!<prefix>`, or `exit` (https://linux.die.net/man/3/exit) commands with any redirection operators. Rather, you can assume these commands will always be run on a line by themselves.

Note: Assume that the redirection operator commands will be formatted correctly. Any incorrectly formatted redirection commands is considered undefined behavior.

OUTPUT

`>` places the output of a command into a file. Usage:

```
<cmd> [args ...] > <filename>
```

If the file exists, overwrite the contents of the file with the output of the current command. Example usage:

```
(pid=2777)/home/usr$ echo hello > hey.txt
Command executed by pid=3750
(pid=2777)/home/usr$ cat hey.txt
Command executed by pid=3751
hello
(pid=2777)/home/usr$ echo welcome to cs241 > hey.txt
Command executed by pid=3752
(pid=2777)/home/usr$ cat hey.txt
Command executed by pid=3754
welcome to cs241
```

APPEND

`>>` appends the output of a command into a file. Usage:

```
<cmd> [args ...] >> <filename>
```

If the file does not exist, assume that it is an empty file. Example usage (`hi.txt` does not exist in the directory before these commands are executed):

```
(pid=2777)/home/usr$ echo a >> hi.txt
Command executed by pid=2780
(pid=2777)/home/usr$ cat hi.txt
Command executed by pid=2781
a
(pid=2777)/home/usr$ echo wheeee >> hi.txt
Command executed by pid=2782
(pid=2777)/home/usr$ cat hi.txt
Command executed by pid=2783
a
wheeee
```


INPUT

`<` pipes the contents of a file into a command as its input. Usage:

```
<cmd> [args ...] < <filename>
```

If the file does not exist, it is undefined behavior. Example usage: `hello.txt` contains:

```
welcome to cs241


(pid=3771)/home/usr$ wc < hello.txt
Command executed by pid=3772
 1  3 17
```

Hint: `dup` will be useful for all the remaining commands (https://linux.die.net/man/3/dup)

Signal Commands

Like bash, your shell will support sending signals to its child processes. We require you to implement the 3 signals listed below.

`kill <pid>`

The ever-useful panic button. Sends `SIGKILL` to the specified process.

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

- Successfully sending `SIGKILL` to process
- No process with `pid` exists
- `kill` was ran without a `pid` (https://linux.die.net/man/3/kill)

`stop <pid>`

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

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

- Process was successfully sent `SIGSTOP`
- No process with `pid` 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:

- Process was successfully sent `SIGCONT`
- No such process exists
- `cont` was ran without a `pid`

Note: Any `<pid>` used in `kill` (https://linux.die.net/man/3/kill) will be a 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

Note that Week 1 and Week 2 count as one week of MP grades respectively. See the overview for a list of features required for each week.