Lab 2: Shell

Introduction

The **shell** is the main command-line interface between a user and the operating system, and it is an essential part of the daily lives of computer scientists, software engineers, system administrators, and such. It makes heavy use of many OS features. In this lab, you will build a simplified version of the Unix shell called the **New Yet Usable SHell**, or `nyush` for short.

Please review the first lecture of MIT's The Missing Semester of Your CS Education if you are not familiar with the shell.

Objectives

Through this lab, you will:

- Familiarize yourself with the Linux programming environment and the shell, of course.
- Learn how to write an interactive command-line program.
- Learn how processes are created, destroyed, and managed.
- Learn how to handle signals and I/O redirection.
- Get a better understanding of the OS and system calls.
- Be a better C programmer and be better prepared for your future technical job interviews. In particular, the string parsing skill that you will practice in this lab is desired in many interview questions.

Overview

The shell is essentially a command-line interpreter. It works as follows:

1. It prompts you to enter a command.
2. It interprets the command you entered.

- If you entered a **built-in command** (*e.g.*, `cd`), then the shell runs that command.
- If you entered an external **program** (*e.g.*, `/bin/ls`), or multiple programs connected through **pipes** (*e.g.*, `ls -l | less`), then the shell creates child processes, executes these programs, and waits for **all** these processes to either terminate or be suspended.
- If you entered something wrong, then the shell prints an error message.

3. Rinse and repeat until you press `Ctrl-D` to close `STDIN` or enter the built-in command `exit`, at which point the shell exits.

Specifications

Your shell should follow these specifications carefully. These specifications may be slight different from the default Linux shell (`bash`) for simplicity.

The prompt

The **prompt** is what the shell prints before waiting for you to enter a command. In this lab, your prompt must have exactly the following format:

- An opening bracket `[`.
- The word `nyush`.
- A whitespace.
- The **basename** of the **current working directory**.
- A closing bracket `]`.
- A dollar sign `$`.
- Another whitespace.

For example, if you are in `/home/abc123/cs202/lab2`, then the prompt should be:

```
[nyush lab2]$ 
```

If you are in the root directory (`/`), then the prompt should be:

```
[nyush /]$ ▉
```

Note that the final `▉` character in these examples represents your *cursor*; you should **not** print that character in your shell prompt.

Keep in mind that `STDOUT` is line-buffered by default. Therefore, don't forget to **flush** `STDOUT` immediately after you print the prompt. Otherwise, your program may not work correctly with the autograder.

The command

In each iteration, the user inputs a command terminated by the "enter" key (*i.e.*, newline). A command may contain multiple programs separated by the **pipe** (`|`) symbol. A valid command must satisfy the following requirements:

- If there are multiple programs in a command, only the **first** program may **redirect its input** (using `<`), and only the **last** program may **redirect its output** (using `>` or `>>`). If there is only one program in a command, it may redirect both input and output.
- In each command, there are no more than one input redirection and one output redirection.
- Built-in commands (*e.g.*, `cd`) cannot be I/O redirected or piped.

For simplicity, our test cases have the following assumptions:

- Each command has no more than 1000 characters.
- There is always **a single space** separating filenames, arguments, and the pipe and redirection symbols (`|`, `<`, `>`, `>>`).
- There are no spaces within a filename or an argument.

For your reference, here is the grammar for valid commands (don't worry if you can't understand it; just look at the examples below):

```
[command] := ""; or
          := [cd] [arg]; or
          := [exit]; or
          := [fg] [arg]; or
          := [jobs]; or
          := [cmd] '<' [filename] [recursive]; or
          := [cmd] '<' [filename] [terminate]; or
          := [cmd] [recursive]; or
          := [cmd] [terminate] '<' [filename]; or
          := [cmd] [terminate].

[recursive] := '|' [cmd] [recursive]; or
            := '|' [cmd] [terminate].

[terminate] := ""; or
            := '>' [filename]; or
            := '>>' [filename].

[cmd] := [cmdname] [arg]*

[cmdname] := A string without any space, tab, > (ASCII 62), < (ASCII 60), | (A

[arg] := A string without any space, tab, > (ASCII 62), < (ASCII 60), | (ASCII

[filename] := A string without any space, tab, > (ASCII 62), < (ASCII 60), | (
```

Here are some examples of **valid** commands:

- A blank line.
- `/usr/bin/ls -a -l`
- `cat shell.c | grep main | less`
- `cat < input.txt`
- `cat > output.txt`
- `cat >> output.txt`
- `cat < input.txt > output.txt`
- `cat < input.txt >> output.txt`
- `cat > output.txt < input.txt`
- `cat >> output.txt < input.txt`
- `cat < input.txt | cat > output.txt`

- `cat < input.txt | cat | cat >> output.txt`

Here are some examples of **invalid** commands:

- `cat <`
- `cat >`
- `cat |`
- `| cat`
- `cat << file.txt`
- `cat < file.txt < file2.txt`
- `cat < file.txt file2.txt`
- `cat > file.txt > file2.txt`
- `cat > file.txt >> file2.txt`
- `cat > file.txt file2.txt`
- `cat > file.txt | cat`
- `cat | cat < file.txt`
- `cd / > file.txt`

If there is any error in parsing the command, then your shell should print the following error message to `STDERR` and prompt for the next command.

```
Error: invalid command
```

Note that there should be a newline at the end of the error message. For example:

```
[nyush lab2]$ cat <
Error: invalid command
[nyush lab2]$ ▉
```

(Again, the final `▉` character represents your cursor.)

Locating programs

You can specify a program by either an **absolute path**, a **relative path**, or **base name only**.

1. An **absolute path** begins with a slash (`/`). If the user specifies an absolute path, then your shell must run the program at that location.
2. A **relative path** contains, but not begins with, a slash (`/`). If the user specifies a relative path, then your shell should locate the program by following the path from the current working directory. For example, `dir1/dir2/program` is equivalent to `./dir1/dir2/program`.
3. Otherwise, if the user specifies **only the base name** without any slash (`/`), then your shell must search for the program under `/usr/bin`. For example, when the user types `ls`, then your shell should try `/usr/bin/ls`. If that fails, it is an error. In this case, your shell should **not** search the current working directory. For example, suppose there is a program named `hello` in the current working directory. Entering `hello` should result in an error, whereas `./hello` runs the program.

In any case, if the program cannot be located, your shell should print the following error message to `STDERR` and prompt for the next command.

```
Error: invalid program
```

Process termination and suspension

After creating the processes, your shell must wait until **all** the processes have stopped running—either terminated or suspended. Then, your shell should prompt the user for the next command.

Your shell must not leave any zombies in the system when it is ready to read the next command from the user.

Signal handling

If a user presses `Ctrl-C` or `Ctrl-Z`, they don't expect to terminate or suspend the shell. Therefore, your shell should **ignore** the following signals: `SIGINT`, `SIGQUIT`, and `SIGTSTP`. All other signals not listed here should keep the default signal handlers.

Note that only the **shell** itself, not the child processes created by the shell, should ignore these signals. For example,

```
[nyush lab2]$ cat
^C
[nyush lab2]$
```

Here, the signal `SIGINT` generated by `Ctrl-C` terminates only the process `cat`, not the shell itself.

As a side note, if your shell ever hangs and you would like to kill the shell, you can still send it the `SIGTERM` or `SIGKILL` signal. To do so, you can connect to the running Docker container from another terminal window using:

```
docker exec -it cs202 bash
```

...and then kill your shell using:

```
[root@... cs202]# killall nyush
```

I/O redirection

Sometimes, a user would read the input to a program from a file rather than the keyboard, or send the output of a program to a file rather than the screen. Your shell should be able to redirect the **standard input** (`STDIN`) and

the **standard output** (`STDOUT`). *For simplicity, you are not required to redirect the standard error (`STDERR`).*

Input redirection

Input redirection is achieved by a `<` symbol followed by a filename. For example:

```
[nyush lab2]$ cat < input.txt
```

If the file does not exist, your shell should print the following error message to `STDERR` and prompt for the next command.

```
Error: invalid file
```

Output redirection

Output redirection is achieved by `>` or `>>` followed by a filename. For example:

```
[nyush lab2]$ ls -l > output.txt
[nyush lab2]$ ls -l >> output.txt
```

If the file does not exist, a new file should be created. If the file already exists, redirecting with `>` should **overwrite** the file (after truncating it), whereas redirecting with `>>` should **append** to the existing file.

Pipe

A **pipe** (`|`) connects the standard output of the first program to the standard input of the second program. For example:

```
[nyush lab2]$ cat shell.c | wc -l
```

The user may invoke *n* programs chained through (*n* - 1) pipes. Each pipe connects the output of the program immediately before the pipe to the input of the program immediately after the pipe. For example:

```
[nyush lab2]$ cat shell.c | grep main | less
```

Here, the output of `cat shell.c` is the input of `grep main`, and the output of `grep main` is the input of `less`.

Built-in commands

Every shell has a few built-in commands. When the user issues a command, the shell should first check if it is a **built-in command**. If so, it should not be executed like other programs.

In this lab, you will implement four built-in commands: `cd`, `jobs`, `fg`, and `exit`.

`cd <dir>`

This command changes the **current working directory** of the shell. It takes exactly one argument: the directory, which may be an absolute or relative path. For example:

```
[nyush lab2]$ cd /usr/local
[nyush local]$ cd bin
[nyush bin]$ 
```

If `cd` is called with 0 or 2+ arguments, your shell should print the following error message to `STDERR` and prompt for the next command.

```
Error: invalid command
```

If the directory does not exist, your shell should print the following error message to `STDERR` and prompt for the next command.

```
Error: invalid directory
```

`jobs`

This command prints a list of currently **suspended** jobs to `STDOUT`, one job per line. Each line has the following format: `[index] command`. For example:

```
[nyush lab2]$ jobs
[1] ./hello
[2] /usr/bin/top -c
[3] cat > output.txt
[nyush lab2]$
```

(If there are no currently suspended jobs, this command should print nothing.)

A job is the whole command, including any arguments and I/O redirections. A job may be suspended by `Ctrl-Z`, the `SIGTSTP` signal, or the `SIGSTOP` signal. This list is sorted by the time each job is suspended (oldest first), and the index starts from 1.

> ***Note for Windows users***
>
> *If you're using recent versions of Windows, you might need to enable Legacy Console mode in order for `Ctrl-Z` to be properly handled by PowerShell.*

For simplicity, we have the following assumptions:

- There are no more than 100 suspended jobs at one time.

- There are no pipes in any suspended jobs.
- The only way to resume a suspended job is by using the `fg` command (see below). We will not try to resume or terminate a suspended job by other means. We will not try to press `Ctrl-C` or `Ctrl-D` while there are suspended jobs.
- You don't need to worry about "process groups." (If you don't know what process groups are, don't worry.)

The `jobs` command takes no arguments. If it is called with any arguments, your shell should print the following error message to `STDERR` and prompt for the next command.

```
Error: invalid command
```

`fg <index>`

This command resumes a job in the foreground. It takes exactly one argument: the job index, which is the number inside the bracket printed by the `jobs` command. For example:

```
[nyush lab2]$ jobs
[1] ./hello
[2] /usr/bin/top -c
[3] cat > output.txt
[nyush lab2]$ fg 2
```

The last command would resume `/usr/bin/top -c` in the foreground. Note that the job index of `cat > output.txt` would become 2 as a result. Should the job `/usr/bin/top -c` be suspended *again*, it would be inserted to the end of the job list:

```
[nyush lab2]$ jobs
[1] ./hello
[2] cat > output.txt
[3] /usr/bin/top -c
[nyush lab2]$
```

If `fg` is called with 0 or 2+ arguments, your shell should print the following error message to `STDERR` and prompt for the next command.

```
Error: invalid command
```

If the job `index` does not exist in the list of currently suspended jobs, your shell should print the following error message to `STDERR` and prompt for the next command.

```
Error: invalid job
```

`exit`

This command terminates your shell. However, if there are currently suspended jobs, your shell should not terminate. Instead, it should print the following error message to `STDERR` and prompt for the next command.

```
Error: there are suspended jobs
```

The `exit` command takes no arguments. If it is called with any arguments, your shell should print the following error message to `STDERR` and prompt for the next command.

```
Error: invalid command
```

Note that if the `STDIN` of your shell is closed (*e.g.*, by pressing `Ctrl-D` at the prompt), your shell should terminate regardless of whether there are suspended jobs.

Compilation

We will grade your submission in an x86_64 Rocky Linux 8 container on Gradescope. We will compile your program using `gcc` 12.1.1 with the C17 standard and GNU extensions.

You must provide a `Makefile`, and by running `make`, it should generate an executable file named `nyush` in the current working directory. *(Refer to Lab 1 for an example of the `Makefile`.)*

Your program must not call the `system()` function or execute `/bin/sh`. Otherwise, what is the whole point of this lab?

Evaluation

Beat up your own code extensively. Better yet, eat your own dog food. I would happily use `nyush` as my main shell (at least for the duration of this lab), so why wouldn't you?

We are providing a sample autograder with a few test cases. Please extract them inside the Docker container and follow the instructions in the `README` file. *(Refer to Lab 1 for how to extract a `.tar.xz` file.)*

Note that these test cases are **not** exhaustive. The test cases on Gradescope are different from the ones provided and will not be disclosed. Do not try to hack or exploit the autograder.

Submission

You must submit a `.zip` archive containing all files needed to compile `nyush` in the root of the archive. You can create the archive file with the following command in the Docker container:

```
$ zip nyush.zip Makefile *.h *.c
```

(Omit `*.h` if you don't have header files.)

Note that other file formats (*e.g.*, `rar`) will **not** be accepted.

You need to **upload the `.zip` archive to Gradescope**. If you need to acknowledge any influences per our academic integrity policy, write them as comments in your source code.

Rubric

The total of this lab is **100 points**, mapped to **15% of your final grade** of this course.

- Compile successfully and can print the correct prompt. (40 points)
- Process creation and termination. (20 points)
- Simple built-in commands (`cd` and `exit`) and error handling. (10 points)
- Input and output redirection. (10 points)
- Pipes. (10 points)
- Handling suspended jobs (`jobs` and `fg`). (10 points)

You will get 0 points for this lab if you call the `system()` function or execute `/bin/sh`.

Please make sure that your shell prompt and all error messages are **exactly as specified** in this document. Any discrepancy may lead to point deductions.

Tips

Don't procrastinate! Don't procrastinate!! Don't procrastinate!!!

This lab is **significantly more challenging** than Lab 1 and requires **significant programming effort**. Therefore, **start as early as possible!** Don't wait until the last week.

How to get started?

Please review our academic integrity policy carefully before you start.

This lab requires you to **write a complete system from scratch**, so it may be daunting at first. Remember to get the **basic functionality** working first, and build up your shell step-by-step.

Here is how I would tackle this lab:

Milestone 1. Write a simple program that prints the prompt and flushes `STDOUT`. You may use the `getcwd()` system call to get the current working directory.

Milestone 2. Write a loop that repeatedly prints the prompt and gets the user input. You may use the `getline()` library function to obtain user input.

Milestone 3. Extend Milestone 2 to be able to run a simple program, such as `ls`. At this point, what you've written is already a shell! Conceptually, it might look like:

Click to reveal spoiler

Milestone 4. Extend Milestone 3 to run a program with arguments, such as `ls -l`.

Milestone 5. Handle simple built-in commands (`cd` and `exit`). You may use the `chdir()` system call to change the working directory.

Milestone 6. Handle output redirection, such as `cat > output.txt`.

Milestone 7. Handle input redirection, such as `cat < input.txt`.

Milestone 8. Run two programs with one pipe, such as `cat | cat`. The example code in `man 2 pipe` is very helpful.

Milestone 9. Handle multiple pipes, such as `cat | cat | cat`.

Milestone 10. Handle suspended jobs and related built-in commands (`jobs` and `fg`). Read `man waitpid` to see how to wait for a **stopped** child.

Some students may find handling suspended jobs easier than implementing pipes, so feel free to rearrange these milestones as you see fit.

Keep versions of your code! Use `git` or similar tools, but **don't make your repository public**.

Useful system calls

Your shell will make many system calls. Here are a few that you may find useful.

- Process management: `access()`, `fork()`, `execv()` (or other variants), `waitpid()`.
- I/O redirection and pipes: `dup2()`, `creat()`, `open()`, `close()`, `pipe()`.
- Signal handling: `signal()`.
- Built-in commands: `chdir()`, `getcwd()`, `kill()`.

You might not need to use all of them, and you are free to use other system calls not mentioned above.

Check the **return values** (and possibly, `errno`) of all system calls and library function calls from the very beginning of your work! This will often catch errors early, and it is a good programming practice.

Reading man pages

Man pages are of vital importance for programmers working on Linux and such. It's a treasure trove of information.

Man pages are divided into **sections**. Please see `man man` for the description of each section. In particular, Section 2 contains system calls. You will need to look them up a lot in this lab.

Sometimes, you need to specify the section number explicitly. For example, `man kill` shows the `kill` command in Section 1 by default. If you need to look up the `kill()` system call, you need to invoke `man 2 kill`.

Parsing the command line

You might find writing the command parser troublesome. Don't be frustrated. You are not alone. However, it is an essential skill for any programmer, and it often appears in software engineer interviews. Once you get through it, you will never be afraid of it again.

I personally find the `strtok_r()` function extremely helpful. You don't have to use it, but why not give it a try?

In `gdb`, how to debug the child process after `fork()`?

By default, when a program forks, `gdb` will continue to debug the parent process, and the child process will run unimpeded.

If you want to follow the child process instead of the parent process, use the command `set follow-fork-mode child`.

If you want to debug both the parent and child processes, use the command `set detach-on-fork off`. Then, you can use `info inferiors` to show all processes and use `inferior` to switch to another process.

I'm still confused about pipes. Any more hints?

If you can already support I/O redirection for a single process, adding support for pipes shouldn't be too difficult since a pipe is just two I/O redirections. Start by reading the example code in `man 3 pipe` and writing a simple program to see how pipe works.

Here are some additional hints:

Click to reveal spoiler

I'm still confused about suspended jobs. Any more hints?

Click to reveal spoiler

I don't want to reënter `gdb` when I press `Ctrl-Z`. Is it possible?

You can always use `c` to continue the program, or you can use the command `handle SIGTSTP nostop noprint` to prevent `gdb` from handling the signal.

I hate the command line! Can I debug in VS Code?

First, I recommend leaving your comfort zone and embracing the command line. For many of you, it will be an essential skill in your career.

However, if you can't live without a GUI, you can follow these steps to attach VS Code to a running container:

1. Install the Dev Containers extension in VS Code.
2. Start the container in your OS as usual using the `docker run` command.
3. In VS Code, press `Ctrl-Shift-P` (Windows) or `⇧⌘P` (Mac) and select `Dev Containers: Attach to Running Container...`.
4. Select `cs202` from the pop-up.
5. Now, you have connected your VS Code to the container. Remember to install useful extensions inside the container, such as the C/C++ extension, since it is a remote environment.
6. You can configure `gdb` in VS Code.
7. To switch between processes in VS Code, add these lines to `launch.json`.

This lab has borrowed some ideas from Prof. Arpaci-Dusseau and Dr. T. Y. Wong.