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 so on. 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.

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 frequently appears in 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 enter the built-in command `exit`, at which point it exits.

Specifications

The prompt

The **prompt** is what the shell prints before waiting for you to enter a command. In this lab, you will use the following format (the final underscore represents your cursor; you should **not** print that underscore):

```
[nyush dir]$ _
```

The `dir` is the **basename** of the **current working directory**. For example, if you are in `/home/abc123/os/lab2`, then the prompt should be:

```
[nyush lab2]$ _
```

Note that there is a space after the dollar sign.

The command

In each iteration, the user inputs a command terminated by the “enter” key (*i.e.*, newline). For simplicity, we have the following assumptions:

- Each command has no more than 1000 characters.
- Program arguments, if any, are separated by a single space.
- There is no space within the program name or any command-line argument.
- A command may contain multiple programs separated by the **pipe** (`|`) symbol.
- In each command, the **first** program may **redirect its input** using `<`, and the **last** program may **redirect its output** using `>` or `>>`. If there is only one program in the command, it may redirect both input and output.
- In each command, there may be at most one input redirection and one output redirection.
- Built-in commands (*e.g.*, `cd`) cannot be I/O redirected or piped.

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].
            := '>>' [filename].

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

[cmdname] := A string without any space, tab, > (ASCII 62), < (ASCII 60), | (ASCII 124), * (ASCII 42), !

[arg] := A string without any space, tab, > (ASCII 62), < (ASCII 60), | (ASCII 124), * (ASCII 42), ! (AS

[filename] := A string without any space, tab, > (ASCII 62), < (ASCII 60), | (ASCII 124), * (ASCII 42),
```

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

- A blank line.
- `ls -a -l`

- `cat shell.c | grep main | less`
- `cat < input.txt > output.txt`
- `cat < input.txt >> output.txt`
- `cat < input.txt | cat | cat > output.txt`

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

- `cat <`
- `cat >`
- `cat |`
- `| cat`
- `cat << input.txt`
- `cat < output.txt < output2.txt`
- `cat < output.txt output2.txt`
- `cat > output.txt > output2.txt`
- `cat > output.txt output2.txt`
- `cat > output.txt | cat`
- `cat | cat < input.txt`
- `cd / > output.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]$ _
```

Locating programs

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

An **absolute path** begins with a slash (`/`). If the user specifies an absolute path, then your shell must run the program at that location.

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`.

Otherwise, if the user specifies **only the base name** without any slash (`/`), then your shell must search for the program under `/bin` and `/usr/bin` (in such order). For example, when the user types `ls`, then your shell should try `/bin/ls`. If that fails, try `/usr/bin/ls`. If that also fails, it is an error. In this case, your shell should **not** search the current working directory. If 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`, `SIGTERM`, `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 `SIGKILL` signal.

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 file name. 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 file name. 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.

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` in the following format: `[index] command`. For example:

```
[nyush lab2]$ jobs
[1] cat
[2] top | cat
[3] cat > output.txt
[nyush lab2]$ _
```

A job is the whole command, which may be either one program or multiple programs connected through pipes. 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).

For simplicity, you can assume that there are no more than 100 suspended jobs at one time. Besides, you are not required to handle the case where a suspended job is resumed or terminated by other processes.

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 in the bracket printed by the `jobs` command. For example:

```
[nyush lab2]$ jobs
[1] cat
[2] top | cat
[3] cat > output.txt
[nyush lab2]$ fg 2
```

This command would resume `top | cat` in the foreground.

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
```

Compiling

We will grade your submission on linserv1.cims.nyu.edu, which runs CentOS Linux release 7.9.2009. We will compile your program using `gcc` 9.2.0. You need to run the following command to load it:

```
$ module load gcc-9.2
```

You must provide a `Makefile`, and by running `make`, it should generate an executable file named `nyush` in the current working directory.

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

Testing

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?

By popular demand, we are providing some sample test cases. Note that these cases are **not** exhaustive. The test cases for final grading will be different from the ones provided and will not be shared. Commands like `job` and `fg` are not tested extensively in these test cases, but they will be for the final grading.

Submission

You will submit an archive containing all files needed to compile `nyush`. You can do so with the following command:

```
$ tar cvJf nyush-Your_NetID.tar.xz Makefile *.h *.c
```

Rubric

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

- Compile successfully and can prompt the user for input. (40 points)
- Process creation and termination. (20 points)
- I/O redirection and pipe. (20 points)

- Handling suspended jobs (`jobs` and `fg`). (10 points)
- Built-in commands and error handling. (10 points)

You will get 0 points for this lab if you call the `system()` function.

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

Tips

Don’t procrastinate

This lab requires significant programming effort. Therefore, **start as early as possible!** Don’t wait until the last week.

Step by step

Remember to get the **basic functionality** working first, and build up your shell step-by-step.

Here is how I would tackle this lab:

1. Write a simple command parser.
2. Be able to run a simple program, such as `ls`.
3. Run a program with arguments, such as `ls -l`.
4. Handle simple built-in commands (`cd` and `exit`).

5. Handle output redirection, such as `cat > output.txt`.
6. Handle input redirection, such as `cat < input.txt`.
7. Run two programs with one pipe, such as `cat | cat`.
8. Handle multiple pipes, such as `cat | cat | cat`.
9. Handle suspended jobs.
10. Handle more built-in commands (`jobs` and `fg`).

Feel free to rearrange as you see fit.

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

Read man pages

Man pages are of vital importance for programmers working on Linux and such. They contain a 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 pipe` shows the page in Section 8 by default. If you need to look up the `pipe()` system call, you need to invoke `man 2 pipe`.

Useful system calls

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

- Process management: `fork()`, `exec*()`, `wait()`, `waitpid()`.
- I/O redirection and pipe: `dup2()`, `creat()`, `open()`, `close()`, `pipe()`.
- Signal handling: `signal()`.
- Built-in commands: `chdir()`, `getcwd()`, `kill()`.

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

Check the **return values** of all system calls from the very beginning of your work. This will often catch errors early, and it is a good programming practice.

Parsing the command line

You may 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?

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