Course: Operating Systems - Semester 1 of 5781
Assignment 2

Directions

A. Due Date: 6 December 2020 at 11:55pm

B. Groups of up to two (2) students may submit this assignment.

C. Code for this assignment must be submitted via Github using the private repository opened for you in the OSCourse organization.

D. This assignment requires you to build a shell. Please ensure that your solution does not print any extraneous output when `stdin` is not a terminal. That is, any time a built-in or a process is run with your shell, only the output of the built-in or process should be printed. Please do not print anything extra for debugging,

E. There are 100 points total on this assignment.

F. What to turn in:

   (a) `Makefile` for the whole project (requirements below)

   (b) Completed `shell.c`

   (c) Any additional files required for compilation (you need `tokenizer.h` and `tokenizer.c` at least)

   (d) README.txt file with any instructions or comments on the assignment submission

General Requirements

1. All of the code below must be written in C and compilable and executable in a standard Linux Mint (17+) or Linux Ubuntu (16+) environment.
2. All code must have comments - each function must have an introductory comment detailing its purpose, input parameters, and return value.

Shell[1]

In this homework, you'll be building a shell, similar to the Bash shell you use on your Mint virtual machine. The purpose of a shell is to allow users to run and manage programs. The operating system kernel provides well-documented interfaces for building shells. By building your own shell, you'll become more familiar with these interfaces and you'll probably learn more about other shells as well.

1 Step 1: Makefile

Log in to your GitHub account and open up a repository for HW2 using the link provided in Moodle. Once you get your repository working you'll find starter code for your shell. It includes a string tokenizer, which splits a string into words. In order to run the shell, you should be able to run the following from the command line:

```
$ make
$ ./shell
```

In order to terminate the shell after it starts, either type exit or press CTRL-D.

What to do Write a Makefile for the project which includes the following rules:

- An `all` rule which compiles the code into an executable called `shell`. The rule should be the default rule for the `make` tool.
- For each `*.c` file: An intermediate rule to compile it to a `*.o` file.
- A `clean` rule which erases all `*.o` files and the `shell` executable.

2 Step 2: Add support for `cd` and `pwd`

The skeleton code for your shell has a dispatcher for *built-in commands*. Every shell needs to support a number of built-in commands, which are functions in the shell itself, not external programs. For example, the `exit` command needs to be implemented as a built-in command, because it exits the shell itself. So far, the only two built-ins supported are `?`, which brings up the help menu, and `exit`, which exits the shell.

Next, do the following steps:

1. As your first step, add a new built-in `pwd` that prints the current working directory to standard output.
2. Then, add a new built-in `cd` that takes one argument, a directory path, and changes the current working directory to that directory. Be sure to support commands such as "cd ." (stay in the same directory) and "cd .." (move up to the parent directory).
3. Once you're done, push your code and makefile to the repository. Push a commit to your group's repository on the working code so far with the message "Added support for cd and pwd"

You should commit your code periodically and often so you can go back to a previous version of your code if you want to.

[1] This assignment is adapted from HW2 of CS162 at UC Berkeley by Kubiatowicz. The original can be found at: `https://cs162.org/static/homeworks/homework2.pdf`

3 Step 3: Program execution

If you try to type something into your shell that isn't a built-in command, you'll get a message that the shell doesn't know how to execute programs. Modify your shell so that it can *execute programs* when they are entered into the shell. The first word of the command is the name of the program. The rest of the words are the command-line arguments to the program.

For this step, you can assume that the first word of the command will be the full path to the program. So instead of running `wc`, you would have to run `/usr/bin/wc`. In the next section, you will implement support for simple program names like `wc`. But you can get some points for the assignment by only supporting full paths.

You should use the functions defined in `tokenizer.c` for separating the input text into words. You do not need to support any parsing features that are not supported by `tokenizer.c`. Once you implement this step, you should be able to execute programs like this:

```
$ ./shell
0: /usr/bin/wc shell.c
77 262 1843 shell.c
1: exit
```

When your shell needs to execute a program, it should fork a child process, which calls one of the `exec` functions to run the new program. The parent process should *wait* until the child process completes and then continue listening for more commands.

Once you're done, push your code to the repository. Push a commit to your group's repository on the working code so far with the message "Added support for basic program execution"

4 Step 4: Path Resolution

You probably found that it was a pain to test your shell in the previous part because you had to type the full path of every program. Luckily, every program (including your shell program) has access to a set of *environment variables*, which is structured as a hashtable of string keys to string values. One of these environment variables is the PATH variable. You can print the `PATH` variable of your current environment on your Mint or Ubuntu VM: (use `bash` for this, not your shell)

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:...
```

When `bash` or any other shell executes a program like `wc`, it looks for a program called `wc` in each directory on the `PATH` environment variable and runs the first one that it finds. The directories on the path are separated with a colon.

Important: If the user writes the name of a program without a complete or relative path (*e.g.* writing "$ wc" at the command prompt), look for the program **only** in the PATH directories. Do not look for it in the working directory. For instance, if there is a program called "wc" in the working directory and the user types "$ wc", run the "wc" that is **found on the PATH** and not the one in the working directory. As another example, if there is a program called "foo" in the working directory, the user types "$ foo", and there is no program called "foo" found in the PATH directories, **do not** run the "foo" in the working directory[2].

Modify your shell so that it uses the PATH variable from the environment to resolve program names. Typing in the full pathname of the executable should still be supported. Do not use `execvp` (which does the path search for you). Use `execv` instead and implement your own `PATH` resolution.

Once you're done, push your code to the repository. Push a commit to your group's repository on the working code so far with the message "Added support for path resolution"

[2] In order to run "foo" in the working directory, the user must either enter the relative path ($ ./foo) or the complete path ($ /usr/home/mjmay/foo).

5 Step 5: Input/Output Redirection

When running programs, it is sometimes useful to provide input from a file or to direct output to a file. The syntax `[process] > [file]` tells your shell to redirect the process's standard output to a file. Similarly, the syntax `[process] < [file]` tells your shell to feed the contents of a file to the process's standard input.

Modify your shell so that it supports redirecting `stdin` and `stdout` to files.

- You **do not** need to support redirection for shell built-in commands.
- You **do not** need to support `stderr` redirection or appending to files (e.g. `[process] >> [file]`).
- You **can** assume that there will always be spaces around special characters < and >.

Be aware that the < `[file]` or > `[file]` are **not** passed as arguments to the program.

Tests Your program must support calls which redirect both input and output. For example, the following commands must all work:

- `apt > aptOutfile.txt`
  - Single output redirection
- `/usr/bin/wc > wcMultipleInFileOut1.txt first-file-to-open.txt second-file-to-open.txt`
  - Single output redirection (to wcMultipleInFileOut1.txt) with parameters passed (first-file-to-open.txt and second-file-to-open.txt)
- `/usr/bin/wc < wc-testing-file.txt > wcInputOutfile.txt`
  - Redirect of input from wc-testing-file.txt
  - Redirect of output to wcInputOutfile.txt

Once you're done, push your code to the repository. Push a commit to your group's repository on the working code so far with the message "Added support for input/output redirection."

6 Step 6: Pipes

Other times, it is useful to provide the output of a program as the input of another program. The syntax

```
[process A] | [process B]
```

tells your shell to pipe the output of process A to the input of process B. In other words, the output of program A becomes the input of program B. (Think redirecting STDOUT from A to go to STDIN from B).

Modify your shell so that it supports pipes between programs. Assume there will always be spaces around the special character |. Outputs may be piped more than once. For example,

```
[process A] | [process B] | [process C]
```

can be passed as an argument to the program.

Once you're done, push your code to the repository. Push a commit to your group's repository on the working code so far with the message "Added support for pipes."

7 Step 7: Signal Handling and Terminal Control

Most shells let you stop or pause processes with special key strokes. These special keystrokes, such as CTRL-C or CTRL-Z, work by sending signals to the shell's subprocesses. For example, pressing CTRL-C sends the `SIGINT` signal, which usually stops the current program. Pressing CTRL-Z sends the `SIGTSTP` signal, which usually sends the current program to the background. If you try these keystrokes in your shell at this point, the signals are sent directly to the shell process itself. This is not what we want since, for example, attempting to CTRL-Z a subprocess of your shell will also stop the shell itself. We want to have the signals affect only the subprocesses that our shell creates. Before we explain how you can achieve this effect, let's discuss some more operating system concepts.

7.1 Example: Shells in Shells

On your Mint VM, you'll execute a short series of commands to better understand the correct behavior. We'll primarily be making use of two commands, `ps` and `jobs`. Recall that `ps` gives you information about all processes running on the system; `jobs` gives you a list of jobs that the current shell is managing. Enter the following commands in your terminal, and you should see similar output:

```
$ ps
PID TTY TIME CMD
20970 ttys002 0:01.30 -bash
$ sh
sh-3.2$ ps
PID TTY TIME CMD
20970 ttys002 0:00.63 -bash
22323 ttys004 0:00.01 sh
```

At this point, we have started a `sh` shell within our bash shell.

```
sh-3.2$ cat
hello
hello
world
world
^Z
[1]+ Stopped(SIGTSTP) cat
sh-3.2$ ps
PID TTY TIME CMD
20970 ttys004 0:00.63 -bash
22323 ttys004 0:00.02 sh
22328 ttys004 0:00.01 cat
```

Notice how sending a CTRL-Z while the `cat` program was running did not suspend the `sh` or the `bash` shells.

```
sh-3.2$ jobs
[1]+ Stopped(SIGTSTP) cat
sh-3.2$ exit
$ ps
PID TTY TIME CMD
20970 ttys004 0:00.65 -bash
```

Since `exit` terminates the shell it's typed into, we terminated the `sh` program and returned to the bash shell that started `sh`. Enter `exit` again and your terminal will close. Before we explain how you can achieve this effect, let's discuss some more operating system concepts.

7.2 Process groups

We have already established that every process has a unique process ID (**pid**). Every process also has a (possibly non-unique) process group ID (**pgid**) which, by default, is the same as the pgid of its parent process. Processes can get and set their process group ID with `getpgid()`, `setpgid()`, `getpgrp()`, or `setpgrp()`.

Keep in mind that, when your shell starts a new program, that program might require multiple processes to function correctly. All of these processes will inherit the same process group ID of the original process. So, it may be a good idea to put each shell subprocess in its own process group, to simplify your bookkeeping. When you move each subprocess into its own process group, the `pgid` should be equal to the `pid`.

7.3 Foreground terminal

Every terminal has an associated *foreground* process group ID. When you type CTRL-C, your terminal sends a signal to every process inside the foreground process group. You can change which process group is in the foreground of a terminal with `tcsetpgrp(int fd, pid_t pgrp)`. The `fd` should be 0 for `standard input`.

7.4 Overview of signals

Signals are asynchronous messages that are delivered to processes. They are identified by their signal number, but they also have somewhat human-friendly names that all start with SIG. Some common ones include:

1. SIGINT - Delivered when you type CTRL-C. By default, this stops the program.
2. SIGQUIT - Delivered when you type CTRL-\. By default, this also stops the program, but programs treat this signal more seriously than SIGINT. This signal also attempts to produce a core dump of the program before exiting.
3. SIGKILL - There is no keyboard shortcut for this. This signal stops the program forcibly and cannot be overridden by the program. (Most other signals can be ignored by the program.)
4. SIGTERM - There is no keyboard shortcut for this either. It behaves the same way as SIGQUIT.
5. SIGTSTP - Delivered when you type CTRL-Z. By default, this pauses the program. In bash, if you type CTRL-Z, the current program will be paused and bash (which can detect that you paused the current program) will start accepting more commands.
6. SIGCONT - Delivered when you run `fg` or `fg %NUMBER` in bash. This signal resumes a paused program.
7. SIGTTIN - Delivered to a background process that is trying to read input from the keyboard. By default, this pauses the program, since background processes cannot read input from the keyboard. When you resume the background process with SIGCONT and put it in the foreground, it can try to read input from the keyboard again.
8. SIGTTOU - Delivered to a background process that is trying to write output to the terminal console, but there is another foreground process that is using the terminal. Behaves the same as SIGTTIN by default.

In your shell, you can use `kill -XXX PID`, where `XXX` is the human-friendly suffix of the desired signal, to send any signal to the process with process id PID. For example, `kill -TERM PID` sends a SIGTERM to the process with process id PID.

In C, you can use the `sigaction` system call to change how signals are handled by the current process. The shell should basically ignore most of these signals, whereas the shell's subprocesses should respond with the default action. For example, the shell should ignore SIGTTOU, but the subprocesses should not.

Beware: forked processes will inherit the signal handlers of the original process.

Reading `man 2 sigaction` (`http://man7.org/linux/man-pages/man2/sigaction.2.html`) and `man 7 signal` (`http://man7.org/linux/man-pages/man7/signal.7.html`) will provide more information. Be sure to check out the SIG_DFL and SIG_IGN constants. For more information on process group and terminal signaling,please work through the tutorial at https://www.usna.edu/Users/cs/aviv/classes/ic221/s16/lec/19/lec.html.

Your task is to ensure that each program you start is in its own process group. When you start a process, its process group should be placed in the foreground. Stopping signals should only affect the foregrounded program, not the backgrounded shell.

Once you're done, push your code to the repository. Push a commit to your group's repository on the working code so far with the message "Added support for signals and process groups"

8 Grading Rules and Notes

The assignment will be graded as follows.

1. Step 1: Makefile: 7 points
   (a) Build and all rules. 5 points
   (b) Clean rule. 2 points
2. Step 2: cd and pwd: 8 points
   a. Proper addition of `pwd`: 4 points
   b. Proper addition of `cd`: 4 points
3. Step 3: Program execution: 20 points
   a. Executes commands with complete path provided: 15 points
   b. Waits for child process to complete: 5 points
4. Step 4: Path resolution: 15 points
   a. Executes commands on the path: 10 points
   b. Executes local commands with complete path provided (even though there is path support): 5 points
5. Step 5: Redirect I/O: 15 points
   a. Handles redirected input: 7.5 points
   b. Handles redirected output: 7.5 points
6. Step 7: Pipes: 15 points
   a. Sends output from one process to another. 7.5 points
   b. Supports multiple chaining pipes. 7.5 points
7. Step 6: Signal Processing: 20 points
   a. Intercepts signals to the shell. 10 points
   b. Transfers signals to the child process group. 10 points

The following penalties will be assessed:

- Not following instructions for step by step committing of code **(-5) points**
- Runs only commands from the path, not with complete paths. **(-10) points**
- Repository doesn't compile (missing code or header files) (fixable) **(-10) points**
- Repository doesn't compile (not fixable) **0 points**

- Uses `execvp` for Step 3. **0 points for Step 3a**
- Input and output redirect only work at the end of the command, not in the middle or beginning. **(-1) point**
- Can do input and output redirection separately, but not together **(-5) points**
- Submits the < character or the > character as parameters to the called child process. **(-3) points**
- Output redirection or input redirection don't work when there are other parameters. **(-5) points**
- Finds some programs on the path, not all. **(-12) points**
- Function modified by students which doesn't have a comment with purpose and input/output parameters. **(-3) points each**
- Runs commands from the local directory overriding the path. **(-10) points**
- Spurious output (debugging code). **(-2) points**
- When giving a relative path, it still looks in the path for options. **(-5) points**
- Error when running a program with wait (*e.g.* bad free()). **(-3) points**