Course: Operating Systems - Semester 1 of 5786
Assignment 3

Directions

A. Due Date: 22 December 2025 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 on GitHub.

D. You must begin from the starter code given in the GitHub template repository for the assignment.

E. Each team member must make at least 1 substantive commit of code to the repository. Write good commit messages, following the instructions at this tutorial: `https://www.theodinproject.com/lessons/foundations-commit-messages`

F. The last commit must include the message "Submitted for grading"

G. The instructions for this assignment include a requirement to commit the code in specific stages. If you do not follow the commit instructions, there will be a grade penalty. In certain cases, not following the commit instructions will lead to a 0 grade. For instance, performing only a single commit with your final solution will lead to a 0 grade.

H. The assignment requires you to build a shell. 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. do not print anything extra for debugging,

I. There are 100 points total on this assignment.

J. What to turn in via your GitHub repository:

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

   (b) Completed `shell.c` (where you're going to do your work)

   (c) `tokenizer.h` and `tokenizer.c` from the template (they are already in the repository template, just don't remove them!)

   (d) Any additional files required for compilation.

   (e) README.md file with:

   - Names and TZ of the students
   - Assignment name (Assignment 3)
   - Course name
   - Semester (A/B) and Academic Year
   - How many hours were spent on the assignment's execution (break down by the names of the students submitting)
   - Any personal comments or thoughts you want to add

General Requirements

1. All of the code below must be written in C and compilable and executable in a standard Linux Mint or Linux Ubuntu 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're going to build 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

Open a repository for the assignment using the link provided in Moodle. Once your repository is 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 that 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.

Do the following steps:

1. Add a new built-in command `pwd` that prints the current working directory to standard output.
2. Add a new built-in command `cd` that takes one argument (a directory path) and changes the current working directory to that directory.
   (a) If the desired directory doesn't exist, print "No such file or directory" to standard output
3. Next, add support for commands such as "`cd .`" (stay in the same directory) and "`cd ..`" (move up to the parent directory).
4. Next, add support for `cd` without an argument. When the user enters just "`cd`" followed by an enter, change the working directory to the user's HOME directory (you can get it using the environment variable HOME).
5. When you're done, push your code and makefile to the repository. Write the following message in the commit message: "Add support for cd and pwd"

[1] This assignment is adapted from HW2 of CS162 at UC Berkeley by Stoica.

2.1 Tests

The following tests will be performed to ensure your code works properly:

```
pwd
pwd ..
cd
cd /usr/bin
cd /
cd /tmp
cd .
cd ..
cd tmp
cd ../usr/bin
cd /tmp/temp
```

The outputs should be as follows:

1. `0:` `pwd` The shell must output the current working directory
2. `1:` `pwd ..` The shell must output the current working directory (*e.g.,* ignore the ..)
3. `2:` `cd` The shell must move the current working directory to the user's HOME directory
4. `3:` `cd /usr/bin` The shell must move the current working directory to /usr/bin
5. `4:` `cd /` The shell must move the current working directory to /
6. `5:` `cd /tmp` The shell must move the current working directory to /tmp
7. `6:` `cd .` The shell must leave the current working directory alone
8. `7:` `cd ..` The shell must move the current working directory to the parent of the current working directory (if you followed the steps here, it should now be /)
9. `8:` `cd tmp` The shell must move the current working directory to the tmp subdirectory (if you follow the steps here, it should now be /tmp). If tmp doesn't exist, output "No such file or directory"
10. `9:` `cd ../usr/bin` The shell must move the current working directory up one level and then down to the usr/bin subdirectory. If you follow the steps here, it should now be /usr/bin). If there is no appropriate subdirectory (after moving up one, there is no subdirectory usr with bin as a sub-subdirectory), output "No such file or directory" and leave the current working directory unchanged.
11. `10:` `cd /tmp/temp` The shell must print "No such file or directory" and leave the current working directory unchanged.

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 it so 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, 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`. Right now, 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 (non-complete path) program names like `wc`. However, you can get some points for the assignment by only supporting full paths.

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 must fork a child process, which calls one of the `exec` functions to run the new program. The parent process must *wait* until the child process completes and then continue listening for more commands.

Once you're done with this step, push your code to the repository. Add a commit message with the following: "Add support for basic program execution"

3.1 Tests

The following tests will be performed to ensure your code works properly:

```
/usr/bin/ls
/usr/bin/ls -l
/usr/bin/ls shell.c shell.o -l
/usr/bin/touch file1.txt
/usr/local/bin/apt
/usr/bin/nano file1.txt
/usr/bin/firefox
/usr/bin/curl https://mjmay-kinneret.github.io/syllabuses/OS-Fall-5786-Syllabus.pdf --output
    syllabus.pdf
/usr/bin/wc
/usr/bin/wc file1.txt file2.txt file3.txt file4.txt file5.txt file6.txt
```

The output behavior must be as follows:

1. `0:` `/usr/bin/ls` Print the contents of the current working directory
2. `1:` `/usr/bin/ls -l` Print the contents of the current working directory with long details
3. `2:` `/usr/bin/ls shell.c shell.o -l` Print long details about shell.c and shell.o in the current working directory.
4. `3:` `/usr/bin/touch file1.txt` Must create a file called file1.txt. No output should be shown.
5. `4:` `/usr/local/bin/apt` Print the output from the apt command (about packages and installation)
6. `5:` `/usr/bin/nano file1.txt` Open the nano editor on screen for editing of file1.txt. The shell waits for the nano editor to close to return to interactivity.
7. `6:` `/usr/bin/firefox` Starts the Firefox browser. The shell waits for the Firefox browser to close to return to interactivity.
8. `7:` `/usr/bin/curl https://mjmay-kinneret.github.io/syllabuses/OS-Fall-5786-Syllabus.pdf --output syllabus.pdf` Uses the curl tool to download the specified document. The content of the page is stored at syllabus.pdf.[2]
9. `8:` `/usr/bin/wc` The wc program is started. Standard input goes to the wc program. When Ctrl+D is entered, wc shows the results of the input. The shell waits for the wc program to close to return to interactivity.
10. `9:` `/usr/bin/wc file1.txt file2.txt file3.txt file4.txt file5.txt file6.txt` The wc program is started. It is provided with the files in the list as parameters. When finished, wc outputs it results to standard output. The shell waits for the wc program to close to return to interactivity.

4 Step 4: Path Resolution

You probably found it 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*

[2] If your OS doesn't have curl installed, you can install it using `sudo apt install curl`.

variables, a hashtable of string keys to string values the operating system maintains. 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 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[3].

Modify your shell so 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. If you use execvp or do not do manual path resolution you **will not** receive points for this section.

Once you're done, push your code to the repository. Add a commit message with the following: "Add support for path resolution"

4.1 Tests

The following tests will be performed to ensure your code works properly:

```
ls
ls -l
ls shell.c shell.o -l
touch file2.txt
apt
nano file2.txt
firefox
curl https://mjmay-kinneret.github.io/syllabuses/OS-Fall-5786-Syllabus.pdf --output
    syllabus2.pdf
wc
wc file1.txt file2.txt file3.txt file4.txt file5.txt file6.txt
shell
./shell
cat shell.c
./cat shell.c
```

The output behavior must be as follows:

1. `0:` `ls` Print the contents of the current working directory
2. `1:` `ls -l` Print the contents of the current working directory with long details
3. `2:` `ls shell.c shell.o -l` Print long details of the shell.c and shell.o files in the current working directory.
4. `3:` `touch file2.txt` Must create a file called file2.txt. No output should be shown.
5. `4:` `apt` Print the output from the apt command (about packages and installation)
6. `5:` `nano file2.txt` Open the nano editor on screen for editing of file2.txt. The shell waits for the nano Editor to close before it returns to interactivity.

[3] 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`).

7. `6: firefox` Starts the Firefox browser. The shell waits for the Firefox browser to close to return to interactivity.
8. `7: curl https://mjmay-kinneret.github.io/syllabuses/OS-Fall-5786-Syllabus.pdf --output syllabus2.pdf` Uses the curl tool to download the specified webpage. The content of the page is output to syllabus2.pdf.
9. `8: wc` The wc program is started. Standard input goes to the wc program. When Ctrl+D is entered, wc shows the results of the input. The shell waits for the wc program to close before it returns to interactivity.
10. `9: wc file1.txt file2.txt file3.txt file4.txt file5.txt file6.txt` The wc program is started. It is provided with the files in the list as parameters. When finished, wc outputs it results to standard output. The shell waits for the wc program to close before it returns to interactivity.
11. `10: shell` The command should fail since there is no built in command called "shell" in Linux. The output should include the words "Not found".
12. `11: ./shell` The command should run the `shell` program in the current working directory. The parent shell waits for the child shell to close before it returns to interactivity.
13. `12: cat shell.c` Print the contents of the file `shell.c` in the current working directory.
14. `13: ./cat shell.c` Run the program "cat" in the current working directory. Provide the parameter shell.c to it. The shell waits for the local cat program to close before it returns to interactivity.

5 Step 5: Input/Output Redirection

When running programs, it is sometimes useful to provide input from a file or to redirect 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 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 >.

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

Once you're done, push your code to the repository. Add the following to your commit message: "Add support for input/output redirection."

5.1 Tests

The following tests will be performed to ensure your code works properly:

```
apt > aptOutfile.txt
wc > wcMultipleInFileOut1.txt first-file-to-open.txt second-file-to-open.txt
wc first-file-to-open.txt > wcMultipleInFileOut2.txt  second-file-to-open.txt
wc first-file-to-open.txt second-file-to-open.txt > wcMultipleInFileOut3.txt
wc < wc-testing-file.txt > wcInputOutfile.txt
wc > wcInputOutfile.txt < wc-testing-file.txt
echo Hello there > file1.txt
ls -lart > filelist.txt
cat < filelist.txt
cat file1.txt < filelist.txt
```

The output behavior must be as follows:

1. `0: apt > aptOutfile.txt` The output from the apt command is written to the file aptOutfile.txt

2. `1: wc > wcMultipleInFileOut1.txt first-file-to-open.txt second-file-to-open.txt` wc is provided two files as parameters (first-file-to-open.txt and second-file-to-open.txt). The output from wc is written to wcMultipleInFileOut1.txt.
3. `2: wc first-file-to-open.txt > wcMultipleInFileOut2.txt second-file-to-open.txt` wc is provided two files as parameters (first-file-to-open.txt and second-file-to-open.txt). The output from wc is written to wcMultipleInFileOut2.txt.
4. `3: wc first-file-to-open.txt second-file-to-open.txt > wcMultipleInFileOut3.txt` wc is provided two files as parameters (first-file-to-open.txt and second-file-to-open.txt). The output from wc is written to wcMultipleInFileOut3.txt.
5. `4: wc < wc-testing-file.txt > wcInputOutfile.txt` wc is provided with wc-testing-file.txt as input via standard input. The output from wc is written to wcInputOutfile.txt.
6. `5: wc > wcInputOutfile.txt < wc-testing-file.txt` wc is provided with wc-testing-file.txt as input via standard input. The output from wc is written to wcInputOutfile.txt.
7. `6: echo Hello there > file1.txt` echo program is provided with `Hello there` as parameters. Output from echo is written to file1.txt
8. `7: ls -lart > filelist.txt` ls program is provided with -lart as a parameter. Standard output from ls is written to file1.txt.
9. `8: cat < filelist.txt` cat program is provided with the filelist.txt file as input from standard input. Output is written to standard output (screen).
10. `9: cat file1.txt < filelist.txt` cat program is provided with file1.txt as a parameter and standard input it provided from filelist.txt In practice, cat will use the file1.txt input, ignoring the redirected standard input.

6 Step 6: Pipes

Sometimes 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 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. Add the following to your commit message: "Add support for pipes."

6.1 Tests

The following tests will be performed to ensure your code works properly:

```
ls -lart | grep txt
ls -lart | grep txt | wc
echo I like cats | grep cats | wc
cat /etc/passwd | cut --fields=1,2 --delimiter=: | grep x | sha256sum | grep b
```

The output behavior must be as follows:

1. `0: ls -lart | grep txt` The output from the ls program (standard output) is redirected as input to grep. grep program is provided also with "txt" as a parameter. The result from grep is output to the command line. Effective output to screen is the rows from the list of files in the current working directory with the letters txt in them.

   Sample output on my computer:

   ```
   -rw-rw-r-- 1 mjmay mjmay 16 Nov 10 10:31 file1.txt
   -rw-rw-r-- 1 mjmay mjmay 599 Nov 10 10:31 filelist.txt
   ```

2. `1: ls -lart | grep txt | wc` The output from the ls program (standard output) is redirected as input to grep. grep program looks for the letters txt. The result from grep is output to standard input for wc program. The output from wc in output to the command line. Effective output is a single line with the character, word, and line count from the lines with "txt" in them.

   Sample output on my computer:

   ```
   2 18 113
   ```

3. `2: echo I like cats | grep cats | wc` echo program is provided with `I like cats` as parameters. The output from echo is redirected to standard input for grep. grep is provided with `cats` as a parameter. The output from grep is redirected to standard input for wc program. The output from wc program is output to the command line. Effective output is a single line with the character, word, and line count of the original line (it has the word cats in it, so it matches).

   Sample output on my computer:

   ```
   1 3 12
   ```

4. `3: cat /etc/passwd | cut --fields=1,2 --delimiter=: | grep x | sha256sum | grep b` cat program is provided with /etc/passwd as a parameter. cat program's standard output is passed to cut program's standard input. cut program is also provided with two parameters (`--fields=1,2 --delimiter=:`). cut program's standard output is passed to grep's standard input. grep's output is passed to sha256sum's standard input. sha256sum's output is passed to grep's standard input. grep program's output is output to the command line.

   Sample output on my computer:

   ```
   36b06bcc897bc75be9940980c07056414467df63cfdefe683058d13b4d41b0be -
   ```

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 the 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`. `ps` gives you information about all processes running on the system; `jobs` gives you a list of jobs 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

As mentioned in class, 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()`. You can see more about them on the related man pages.

Keep in mind that when your shell starts a new program it might require multiple processes to function correctly. All of the 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 also have text names that all start with SIG. Some common ones are:

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.

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. Add the following to your commit message: "Add support for signals and process groups"

7.5 Tests

The following tests will be performed to ensure your code works properly:

```
wc
hello there
^C

cat
hello world
^D

bash
^\
```

The output behavior must be as follows:

1. `0: wc`
   `hello there`
   `^C`
   The CTRL-C signal is sent to wc, causing it to quit. The shell then resumes where it left off.
2. `1: cat`
   `hello there`
   `hello world`
   `^D` The CTRL-D signal (EOF) is sent to cat. It then completes and the shell resumes where it left off.
3. `2: bash`
   `mjmay@mjmay-virtualbox:~$ wc`
   `^\ (Quit (core dumped)`
   `mjmay@mjmay-virtualbox:~$ exit`
   `3:`
   The CTRL-\ signal is sent to wc, causing it to quit with a core dump. The internal bash shell then resumes where it left off. After entering exit, the internal bash shell quits and the shell program resumes where it left off.

8 Grading Rules and Notes

The assignment will use the autograding features of GitHub, so most points will be assigned on commit. Some tests (*e.g.,* signals) will be tested manually as well. The assignment points will be divided 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: 10 points
   a. Proper behavior of `pwd`: 3 points
   b. Proper behavior of `cd`: 7 points
3. Step 3: Program execution: 17 points
   a. Executes commands with complete path provided: 12 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: 8 points
   b. Handles redirected output: 7 points
6. Step 6: Pipes: 16 points
   a. Sends output from one process to another. 7 points
   b. Supports multiple chaining pipes. 9 points
7. Step 7: Signal Processing: 15 points
   a. Intercepts signals to the shell. 8 points

   b. Transfers signals to the child process group. 7 points

8. GitHub Usage and README.md: 5 points