CS2106: Introduction to Operating Systems

Lab Assignment 2 (A2)

Process Operations in Unix

IMPORTANT

The deadline of submission through LumiNUS: **5th March, 2023, 11.59 PM Saturday**

The total weightage is 8% (+2% bonus):

- Exercise 1: 3% [Lab demo exercise]
- Exercise 2: 2%
- Exercise 3: 3%
- Exercise 4: 2% (Bonus)

You must ensure the exercises work properly on the VM provided or on the SoC Cluster.

1 Introduction

As programmers, command-line interpreters, also known as **shells**, are an important and ubiquitous part of our lives. A command line interpreter (or command prompt, or shell) allows the execution of multiple user commands with various number of arguments. The user inputs the commands one after another, and the commands are executed by the command line interpreter.

A shell is actually just another program implemented using the process-related system calls discussed in the lectures. In this lab, you are going to implement a simple shell with the following functionalities:

- Running commands in foreground and background
- Chaining commands
- Redirecting input, output, and error streams
- Terminating commands
- Managing the processes launched by this shell

The main purpose of this lab is to familiarize you with:

- advanced aspects of C programming,
- system calls,
- process operations in Unix-based operating systems.

2 Exercises in Lab 2

This lab has a total of **four** exercises.

For exercise 1, only some simple functionalities are required. Take the opportunity to design your code in a modular and extensible way. You may want to look through the rest of the exercises before starting to code.

Shell Implementation

The driver of the shell has been implemented for you in **`driver.c`**. The driver will read and tokenise the user commands for you, where tokens will be separated by spaces (whitespace or tabs).

We provide a structure **`PCBTable`** in **`myshell.h`** as follows:

```
struct PCBTable {
    pid_t pid;
    int status;   // 4: Stopped, 3: Terminating, 2: Running, 1: exited
    int exitCode; // -1 not exit, else exit code status
};
```

where

- `pid` is the process id
- `status` indicates the status of the process ; (e.g., 2: Running, 1: Exited)
- `exitCode` indicate the exit status of process, -1 if still running

Please use the `PCBTable` to maintain the details of all the processes you fork. Define a structure variable of type `PCBTable`, which can be an array, a linked list or other containers. You can assume there won't be more than 50 `PCBTables`.

You should implement the **three** functions in the file **`myshell.c`**:

- **`my_init()`** is called when your shell starts up, before it starts accepting user commands. It should initialise your shell appropriately.
- **`my_process_command()`** is called in the main loop. It should handle every user command except the `quit` command. The function accepts two arguments:
  - `tokens` is an array of tokens with the last element of the array being **`NULL`**.
  - `size` is the size of that array, including the **`NULL`**.

  You may want to print out the **`tokens`** array or look at the **`driver.c`** file for your own understanding. You can also call other user-defined functions to make the program more modular (more details later).
- **`my_quit()`** is called when user inputs the `quit` command.

We have also provided some executable programs with various runtime behaviours in the `programs` folder to aid your testing later. You can also create your own programs and run them with your shell.

- **`programs/goingtosleep`**: Prints "Good night!" and then prints "Going to sleep" 10 times with a 2-second sleep between each print.
- **`programs/lazy`**: Wake up, echo "Good morning..." and loops for 2 minutes. It takes a while (at least 5 seconds) to respond to the SIGTERM signal.
- **`programs/result`**: Takes in a single number **`x`** and exits with a return status of **`x`**.

- **`programs/showCmdArg`**: Shows the command line arguments passed in by the user.
- **`programs/Makefile`**: To compile the above programs.

Preventing Fork Bombs

During implementation, you might mistakenly call `fork` infinitely and ignite a fork bomb. Thus we have provided a fork monitor (in `monitor.c` and `fork-wrapper.c`) to prevent the shell from creating too many subprocesses. If your shell creates too many processes, it will be killed with a "YOU ARE HITTING THE FORK LIMIT" message. Please check your code if you see that message.

Although we have such precautions, you should still take care of the problem. Add in the "`fork()`" call only after thoroughly testing the basic code. You may want to test it separately without putting it in a loop first. For any child process, make sure you have a "`return ...`" or "`exit()`" as the last line of code (even if there is an `exec` before as the `exec` can fail).

If you accidentally ignite a fork bomb on one of the SoC Compute Cluster node, your account may be frozen. Please contact your lab tutor who will then contact SoC Technical Service to unfreeze your account and kill off all your processes.

Build and Run

Unlike lab1, lab2 requires you to have a interative shell. Use the following commands below to build and launch your shell. *Please do not use VSCode for developing as it seem to hog the server. Either use ssh+vim or virtual machine for developing and testing. Finally, test your code on the cluster nodes just before submission.*

Build and Run on Slurm

```
$ salloc                # Obtain a job allocation
$ srun --pty bash       # Request for interactive shell
$ make                  # Build
$ ./monitor myshell     # Launch the shell with fork monitor
```

One you have finished testing the code, please DO NOT forget to exit out the shell.

```
$ exit   # Exit the interative shell
$ exit   # Release job allocated via salloc
```

More information can be found here.

For those coding directly on the remote node, we also **strongly recommend** keeping a copy of the code locally, just in case anything unexpected happens to your data on the remote nodes.

You may add **`-Werror`** into the `CFLAGS` in the `Makefile` to make all warnings errors. You will not be penalised for warnings, but you are strongly encouraged to resolve all warnings. Warnings are indications of potential undefined behaviour which may cause your program to behave in inconsistent and unexpected ways that may not always manifest (i.e., it may work when you test, but it may fail when grading). You are also advised to use **valgrind** to ensure your program is free of memory errors.

On launching your shell, you will then see an **`myshell>`** prompt where you can input commands in a similar manner to a Bash shell. The commands that this shell should accept are elaborated in the following sections.

File Structure

When you unzip `lab2.tar.gz`, you should find the following files:

`myshell.c`	Your implementation.
`myshell.h`	Do not modify. Modifications will be ignored when grading.
`driver.c`	Do not modify. The file will be replaced when grading.
`monitor.c`	Do not modify. Fork Monitor
`fork-wrapper.c`	Do not modify. Fork Monitor
`check_zip.sh`	Do not modify. For you to check your zip file before submission.
`Makefile`	Do not modify. The file will be replaced when grading.
`programs/*`	Do not modify. The file will be replaced when grading. Tiny programs for testing.

Note: We have provided some skeleton functions to help you make the program more modular and easy to debug. You may or may not use. Your implementation will NOT be tested for its structure or modularity.

Terminologies

The Table 1 contains information about some terminologies used in this lab document.

Table 1: Terminology Description

Terminology	Description
`{program}`	A single string with no whitespaces. Will be a **valid file path** that only contains the following characters: a-z, A Z, 0-9, forward slash (/), dash (-), and period (.). Refer to the definition of a valid file path at the end of additional details.
`(args...)`	**Optional** arguments to the program. Each argument will be a single string with no whitespaces, and will only contain the following characters: a-z, A-Z, 0-9, forward slash (/), dash (-), and period (.) If there is more than 1 argument, the arguments will be separated by whitespaces.
`option`	It is numeric numbers (both +ve and -ve)
`{PID}`	The process id of the child process. Your process ids may be different from the sample output, this is expected. Will only contain the following characters: 0-9.
`{file}`	A single string with no whitespaces. Will be a **valid file path** that only contains the following characters: a-z, A Z, 0-9, forward slash (/), dash (-), and period (.). Refer to the definition of a valid file path at the end of additional details.
Any other token, e.g., `quit`, `info`, `&`	They refer to the token itself.

2.1 Exercise 1: Basic Shell (1% demo + 2% submission OR 3% submission) [Optional Demo]

Let us implement a simple shell with limited features in this first exercise.

The shell should accept any of the following commands, in a loop:

`{program} (args...)`

Example commands:

```
/bin/ls
/bin/cat -n file.txt
```

If `{program}` is a readable and executable file:
- Execute `{program}` in a child process with the supplied arguments.
- **Wait** till the child process is done.

Else:
- Print "`{program} not found`" to `stderr`.

`{program} (args...) &`

Note the & symbol at the end of the user command.
If `{program}` is a readable and executable file:
- Execute `{program}` in a child process with the supplied arguments.
- Print "**`Child [PID] in background`**", where **`PID`** is the process id of the child process.
- **Continue to accept user commands.**

Else:
- Print "`{program} not found`" to `stderr`.

`info option`

If `option` is **0**
- Print details of all processes in the order in which they were run. You will need to print their process IDs, their current status (Exited or Running)
- For Exited processes, print their exit codes.
- Please check the sample output for the printing format (**??**).

If `option` is **1**
- Print the **number** of exited process.

If `option` is **2**
- Print the **number** of running process.

For all other cases print "`Wrong command`" to `stderr`.

`quit`

- Terminate **all** RUNNING processes by sending **SIGTERM** signal to them, and print "`Killing [pid]`" for each terminated process.
- Do not wait for the child processes. Print "`Goodbye`" and exit the shell.

Notes:

- **Read Section 2.4** to find out additional details about these requirements.
- Note that **info** can only display a return result after a process has exited.
- You should look at the following C system calls and library functions:
  - `access`
  - `fork`
  - `execv`
  - `wait`
  - `waitpid`
  - `kill`
- You can use the skeleton functions provided in `myshell.c` to make your code modular and understand where to use suggested system calls and library functions.

Sample of this exercise:

See it at https://asciinema.org/a/NZAYfBa6ZvG2TVGlaXbYzu0ZD

```
myshell> info 0
myshell> info
Wrong command
myshell> info -1
Wrong command
myshell> /bin/echo hello
hello
myshell> info 0
[3281554] Exited 0
myshell>
myshell> /bin/notaprogram
/bin/notaprogram not found
myshell>
myshell> info 0
[3281554] Exited 0
myshell> /bin/sleep 10 &
Child [3281571] in background
myshell> info 0
[3281554] Exited 0
[3281571] Running
myshell> info 2
Total running process: 1
myshell> info 0
[3281554] Exited 0
```

```
[3281571] Running
myshell> info 0
[3281554] Exited 0
[3281571] Exited 0
myshell> info 1
Total exited process: 2
myshell>
myshell> ./programs/result 7
myshell> info 0
[3281554] Exited 0
[3281571] Exited 0
[3281579] Exited 7
myshell> info 2
Total running process: 0
myshell> info 1
Total exited process: 3
myshell> ./programs/result 256
myshell> info 0
[3281554] Exited 0
[3281571] Exited 0
[3281579] Exited 7
[3281585] Exited 0
myshell> ./programs/showCmdArg 5 23 1 &
Child [3281597] in background
myshell> [Arg 0]: 5
[Arg 1]: 23
[Arg 2]: 1

myshell> /bin/sleep 15 &
Child [3281599] in background
myshell> quit
Killing [3281599]

Goodbye
```

2.2 Exercise 2: Advanced Shell (2%)

Implement the following commands, in addition to the ones in exercise 1.

`wait {PID}`

Example command:

```
wait 226
```

`{PID}` is a process id created using "`{program} (args...) &`" syntax and has not yet been waited for before.
If the process indicated by the process id is RUNNING, **wait** for it.
Else, continue accepting user commands.
No output should be produced.

`terminate {PID}`

Example command:

```
terminate 226
```

If the process indicated by the process ID `{PID}` is RUNNING:

- Terminate it by sending it the **SIGTERM** signal.
- You should **not** wait for `{PID}`.
- The state of `{PID}` should be "Terminating" until `{PID}` exits.

Continue accepting user commands. No output should be produced.

`{program1} (args1...) ; {program2} (args2...) ; ...`

Example command:

```
/bin/ls ; /bin/sleep 5 ; /bin/pwd ; /bin/ls
```

`;` is an operator that allows multiple "`{program} (args...)`" to be chained together and executed **sequentially**, and in the foreground.

1. If `{program1}` exists and is readable and executable:
   - Run and **wait** for `{program1}`.
   - There might be error output from the program if it fails, which is fine.
2. Else:
   - Print "`{program} not found`" to `stderr`.
3. Go back to step 1 with the next `{program2}`.

Note: There will always be spaces around "`;`". It would be helpful to start with 2 chained commands, before extending your implementation to any number of chained commands. Last command will not have a "`;`"

Extend the following command from exercise 1:

`info option`

Should now have an additional status "Terminating", in addition to the original "Running" and "Exited".
If `option` is **0**

- Print details of all processes in the order in which they were run. You will need to print their process IDs, their current status (Exited or Running)
- For Exited processes:
  - If the child process ended normally, print the exit code of the child process.
  - If the child process ended abnormally, print which signal (the signal number) caused the child process to exit (link).

If `option` is **3**

- Print the **number** of terminating process

Notes:

- **Read Section 2.4** to find out additional details about these requirements.
- You should look at the following C system calls:
  - `wait`
  - `kill`

Sample of this exercise:

2.2.1 `wait` Command

See it at https://asciinema.org/a/K2Fk0SkjysZQdWF6LcY2CB2qk

```
myshell> /bin/sleep 15 &
Child [3285066] in background
myshell> info 0
[3285066] Running
myshell> wait 3285066
myshell> info 0
[3285066] Exited 0
myshell> quit

Goodbye
```

2.2.2 `terminate` Command

See it at https://asciinema.org/a/Fb8y1yXa1TVbDq1bW2c5gw6w1

```
myshell> /bin/sleep 30 &
Child [1004267] in background
myshell> info 0
```

```
[1004267] Running
myshell> terminate 1004267
myshell> info 0
[1004267] Exited 15
myshell> info 2
Total running process: 0
myshell>
myshell> ./programs/lazy &
Child [1005093] in background
myshell> Good morning...
terminate 1005093
myshell> Give me 5 more seconds

myshell> info 0
[1004267] Exited 15
[1005093] Terminating
myshell> info 3
Total terminating process: 1
myshell> info 0
[1004267] Exited 15
[1005093] Exited 0
myshell> info 3
Total terminating process: 0
myshell> quit

Goodbye
```

2.2.3 Chained Commands

See it at https://asciinema.org/a/1Nye0hABvhcQG7M8os5TIdh0G

```
myshell> /bin/sleep 5 ; /bin/echo Hello ; /bin/echo Bye
Hello
Bye
myshell> /bin/sleep notnumber ; /bin/echo Hello ; /bin/echo Bye
/bin/sleep: invalid time interval ‘notnumber’
Try '/bin/sleep --help' for more information.
Hello
Bye
myshell> info 0
[3287915] Exited 0
[3287916] Exited 0
[3287917] Exited 0
[3287945] Exited 1
[3287946] Exited 0
[3287947] Exited 0
myshell> info 1
Total exited process: 6
```

```
myshell> info 2
Total running process: 0
myshell>
myshell> quit

Goodbye
```

2.3 Exercise 3: Redirection (3%)

In this exercise, we will implement the redirection operators for our shell by extending the `{program}` (args...), the `{program}` (args...) & commands, and the ; operator from exercises 1 and 2.

Implement the following user commands:

```
{program} (args...) (< {file}) (> {file}) (2> {file})
```

`(< {file})`, `(> {file})`, and `(2> {file})` are optional and may or may not be present. If there are more than 1 present, **all `{file}`s will be different,** i.e., no reading and writing to the same file.

Example commands

```
/bin/cat a.txt > b.txt
/bin/sort < test.txt > sorted.txt
/bin/sleep 2  2> error.log
```

- If **`(< {file})`** is present:
  - If there exists a file at `{file}`: `{program}` reads the contents of `{file}` as input.
  - Else, print "`{file} does not exist`" to `stderr` and exit the child process with exit code 1.
- If **`(> {file})`** is present:
  - If there does not exist a file at `{file}`, create it, then redirect the **standard output** of `{program}` into `{file}`. The `{file}` should be opened in **write** mode, i.e., the file's existing content will be overwritten.
- If **`(2> {file})`** is present:
  - If there does not exist a file at `{file}`, create it, then redirect the **standard error** of {program} into `{file}`. The `{file}` should be opened in **write** mode, i.e., the file's existing content will be overwritten.

Note:

1. A child process should always be created as long as the `{program}` is an executable file. You should check the validity of the files (i.e, whether the file is opened correctly for either reading, writing, or both) used for redirection within the child process before calling `exec`-family syscalls.
2. There will always be spaces around "<", ">" and "2>".

```
{program} (args...) (< {file}) (> {file}) (2> {file}) &
```

Note the & symbol at the end of the user command. Same behaviour as above, except that the command will be run in the background instead, i.e., **your shell should continue accepting user commands.**

```
{program} (args...) (< {file}) (> {file}) (2> {file}) ; {program}
↪  (args...) (< {file}) (> {file}) (2> {file}) ; ...
```

Example command

```
/bin/printf hello\nworld\n > test.txt ; /bin/sort < test.txt >
↪  sorted.txt ; /bin/cat sorted.txt
```

The rest of the `;` operator's function remains the same as in exercise 2. Note that the same file can be read/written across different programs, as shown above with `test.txt`.
If a file is not found for `(< {file})`, just print "`{file} does not exist`" to `stderr`. Again, continue executing the rest of the programs even if one of the program fails.

Notes:

- For simplicity, you may assume that `(< {file})` always appears before `(> {file})`, and both will always appear before `(2> {file})`. Also, the redirection operators will always appear after `{program} (args...)`. This differs from the actual Bash shell where even something like

  ```
  > a.out < a.in cat
  ```

  is still a valid command.
- Notice that with these new functionalities, you can simulate piping by doing

  ```
  program1 > temp ; program2 < temp
  ```

  Have you wondered how piping actually works? Find out more here!
- **Read Section 2.4** to find out additional details about these requirements.
- You should look at the following C system calls and library functions:
  - `fopen` / `open`
  - `dup2`
- A flow chart about how the chained commands with **input redirection** (only) should be handled is shown in Figure 1.

Sample of this exercise: See it at https://asciinema.org/a/ZoqYgvqeezLsZ5UdkMIWABs3G

```
myshell> /bin/cat ./programs/result > ./a.txt
myshell> /bin/cat ./a.txt
#!/bin/bash
result=$1
exit $result
myshell> info 0
[3289566] Exited 0
[3289596] Exited 0
myshell>
myshell> /bin/sort < ./a.txt > ./b.txt &
Child [3289645] in background
myshell> info 0
[3289566] Exited 0
[3289596] Exited 0
[3289645] Exited 0
```

```
myshell> /bin/cat ./b.txt
#!/bin/bash
exit $result
result=$1
myshell>
myshell> /bin/sort < ./doesnotexit.txt
./doesnotexit.txt does not exist
myshell> info 0
[3289566] Exited 0
[3289596] Exited 0
[3289645] Exited 0
[3289661] Exited 0
[3289776] Exited 1
myshell>
myshell> /bin/printf hello\nworld\n > ./a.txt ; /bin/sort < ./a.txt >
↪  ./b.txt ; /bin/cat ./b.txt
hello
world
myshell> info 0
[3289566] Exited 0
[3289596] Exited 0
[3289645] Exited 0
[3289661] Exited 0
[3289776] Exited 1
[3289795] Exited 0
[3289796] Exited 0
[3289797] Exited 0
myshell>
myshell> /bin/echo hello ; /bin/sort < ./doesnotexit.txt
hello
./doesnotexit.txt does not exist
myshell> info 0
[3289566] Exited 0
[3289596] Exited 0
[3289645] Exited 0
[3289661] Exited 0
[3289776] Exited 1
[3289795] Exited 0
[3289796] Exited 0
[3289797] Exited 0
[3289801] Exited 0
[3289802] Exited 1
myshell>
myshell> /bin/sleep notanumber 2> err.log ; /bin/echo hello
hello
myshell> /bin/cat err.log
/bin/sleep: invalid time interval ‘notanumber’
Try '/bin/sleep --help' for more information.
myshell>
```

```
myshell> quit

Goodbye
```

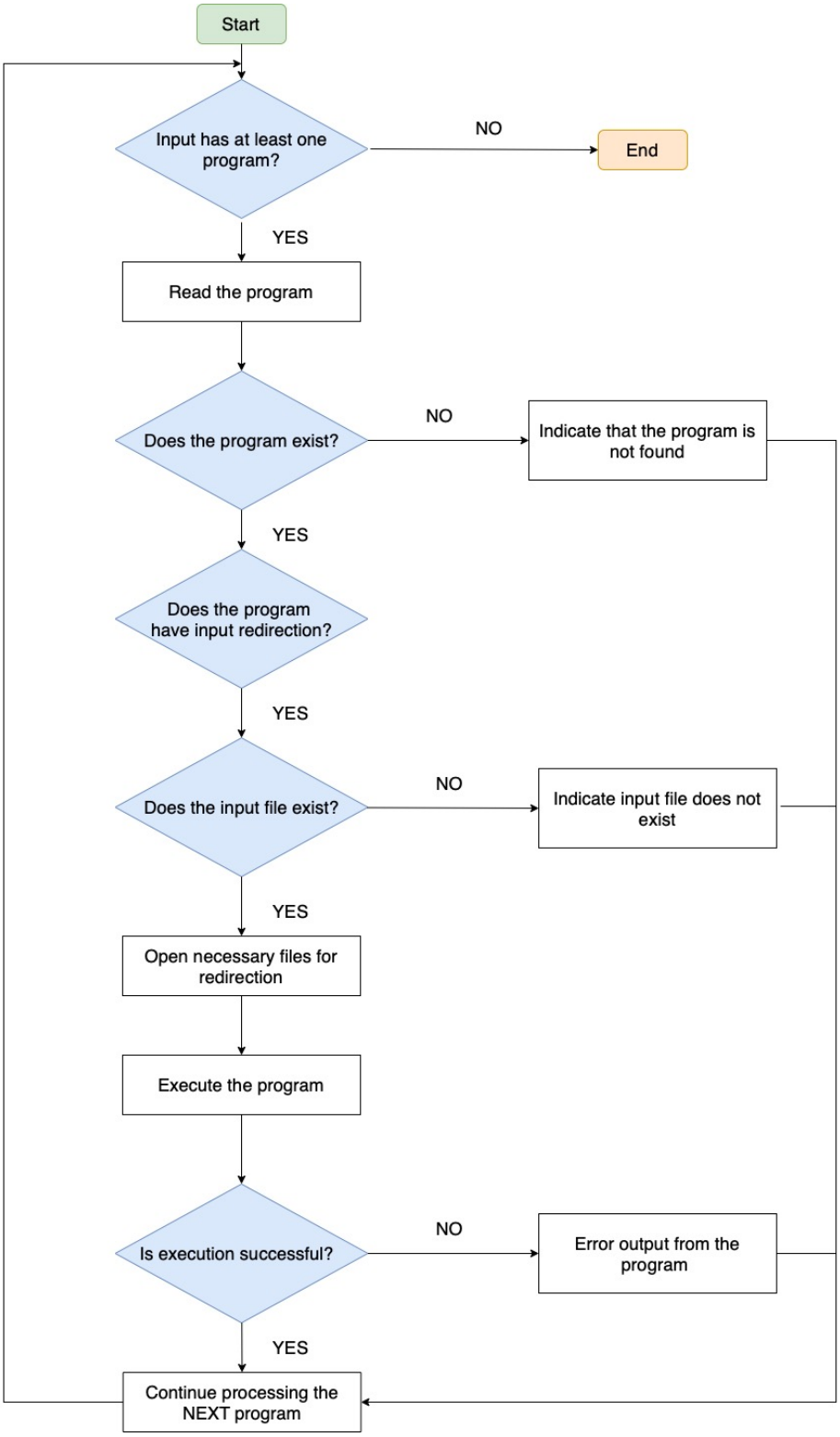

Figure 1: Chained Commands with Input Direction

2.4 Additional details for all exercises

This section will be long and verbose, but it is essential as we need to limit the scope of the lab by defining some constraints. Please read through it, perhaps after you have read through the exercises. The details below apply to all exercises in this lab, unless otherwise stated.

- Any command that does not satisfy the correct syntax details will not be tested on your shell.
- Any command that does not satisfy the formats specified in the various tables shown in sections 2.1 – 2.6 will not be tested on your shell. To give an example, "`/bin/ls & -a`" does not satisfy our command formats, because firstly, the ampersand must appear at the end of the command, and secondly, if we interpret the ampersand as an argument, it will be syntactically invalid by our definitions.
- It is good practice to check the return values of all syscalls (and indeed, all functions) and handle any errors that occur.
- In total, during the entire lifetime of the program, **no more than 50 (<= 50) processes will be run** (counting both currently running processes and processes that have exited).
- Programs with the same names as the user commands will not be run, i.e., there will be no name collisions.
- Programs that require interactive input will not be tested on your shell. For example, programs such as the Python shell, or your shell itself.
- Notice that **background processes** may sometimes mess up the display on your shell, which may cause your output to be slightly different from the sample session. This is fine. Other than these cases, your output, excluding the PID numbers, **should match the sample sessions exactly**.
- At any point of time, if a file is being read from or executed, we will not be running a command that writes to that same file.
- You may assume that there will be no permission issues for files that already exist (files that are not created by your shell). Your shell will have read, write, and execute permissions, where necessary, for these files that already exist. Similarly, there will be no permission issues for directories as well.
- To illustrate what a **valid file path** is for the purposes of this lab, we split a path up into two parts. The first part is the path to the containing directory (we will call it `<dir>`), such that running the command `cd <dir>` in a bash shell always succeeds. The implications of this are that every directory along the path to our file exists and is accessible. This first part is optional, and if omitted defaults to the current working directory. The second part (non-optional) is the file name, which can only contain the following characters: a-z, A-Z, 0-9, dash (-), and period (.). In `<dir>`, there might or might not be a file name that matches our second part, but either way it is still considered a **valid file path**.
- You will have to check if the process can execute the `programs` or not.
- For simplicity, only regular files/directories will be used in our testing. No symbolic links will be used.

2.5 Exercise 4 (BONUS): More Signals (2%)

Right now, when you press Ctrl-Z or Ctrl-C while running your shell, it gets suspended or interrupted respectively. For this exercise, you will intercept the **SIGTSTP** and **SIGINT** signals, which correspond to Ctrl-Z and Ctrl-C respectively (these keys may differ if you're on a Mac).

Please do this bonus using a **copy** of your Exercise 1-3 solutions, in a separate folder as instructed in section 3. The bonus and the main parts will be graded separately.

<Ctrl-Z>

If there is a currently running program that your shell is **waiting** for, send the **SIGTSTP** signal to it, and print "`[PID] stopped`".
Else, do nothing and continue accepting user input.

<Ctrl-C>

If there is a currently running program that your shell is **waiting** for, send the **SIGINT** signal to it, and print "`[PID] interrupted`".
Else, do nothing and continue accepting user input.

`fg {PID}`

If `{PID}` is currently stopped

- Print "`[PID] resumed`".
- Send **SIGCONT** to `{PID}` to get it continue and **wait** for it.

Extend the following commands.

`info option`

Should now have an additional status "Stopped" , in addition to the original "Running", "Exited", and "Terminating".
If `option` is **4**

- Print the **number** of stopped process

`quit`

- Terminates **all** RUNNING and STOPPED processes using SIGTERM.
- Print "`Killing [pid]`" if there exist running or stopped process.
- Print "`Goodbye`" and exit the shell.

Sample of this exercise:

See it at `https://asciinema.org/a/y2dzFLRZ76TcPZTZ2WTCVpkux`

```
myshell> ./programs/lazy
Good morning...
^C[3507490] interrupted
myshell> info 0
[3507490] Exited 2
myshell>
myshell> ./programs/goingtosleep

Good night!
Going to sleep
^Z[3507594] stopped
myshell> info 0
[3507490] Exited 2
[3507594] Stopped
myshell> info 4
Total stopped process: 1
myshell> fg 3507594
[3507594] resumed
Going to sleep
Going to sleep
Going to sleep
Going to sleep
Going to sleep
^Z[3507594] stopped
myshell> info 0
[3507490] Exited 2
[3507594] Stopped
myshell> fg 3507594
[3507594] resumed
Going to sleep
Going to sleep
Going to sleep
Going to sleepmyshell> quit

Goodbye
```

2.6 Exercise 5: Check your archive before submission (0%)

Before you submit your lab assignment, run our check archive script named **`check_zip.sh`**.

The script checks the following:

a. The name or the archive you provide matches the naming convention mentioned in section 3.
b. Your zip file can be unarchived, and the folder structure follows the structure presented in section 3.
c. All files for each exercise with the required names are present.
d. Each exercise can be compiled.

You have the zip files on cluster nodes as follows:

```
$ zip -r E0123456.zip E0123456 % replace with your zip file name
or
$ zip -r E0123456_E0123457.zip E0123456_E0123457
```

Once you have the zip file, you will be able to check it by doing:

```
$ chmod +x ./check_zip.sh
$ ./check_zip.sh E0123456.zip           % replace with your zip file name
or
$ ./check_zip.sh E0123456_E0123457.zip
```

During execution, the script prints if the checks have been successfully conducted, and which checks failed. Successfully passing checks ensures that we can grade your assignment.

3 Submission through Canvas

Zip the following files as `E0123456.zip` (**use your NUSNET id, NOT your student no A012...B, and use capital 'E' as prefix**):

Do **not** add any additional folder structure during zipping.

E0123456.zip contains 1 file, with 1 extra folder if the bonus section is attempted, following this file structure:

```
E0123456
E0123456/myshell.c
E0123456/bonus
E0123456/bonus/myshell.c
```

*The bolded names are folders.

If you have not attempted the bonus, do not submit the bonus folder. Your file structure should just be:

```
E0123456
E0123456/myshell.c
```

You can check your file structure matches the above format by using the following command:

```
$ unzip -l E0123456.zip          % replace with your zip file name
```

If submitting as a pair, *only one member needs to submit*. The folder name should be both partners' NUSNET ids separated by an underscore, i.e., `E0123456_E0123457` instead of `E0123456`. Resultant zip file should be `E0123456_E0123457.zip` instead of `E0123456.zip`.

Upload the zip file to the "Lab 2" folder on Canvas. Note the deadline for the submission is **5th March, 2023, 11.59 PM Saturday**.

Please ensure that you follow the instructions carefully (output format, how to zip the files etc.). Deviations will be penalized.