CS2106: Introduction to Operating Systems

# Lab Assignment 2 (A2)
# Process Operations in Unix

## Important

The deadline of submission through LumiNUS: Wed, 22 Sep, 2pm
The total weightage is 7%:

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

*You must ensure the exercises work properly on the SoC Compute Cluster*, h*ostnames: xcne0 - xcne7, Ubuntu 20.04, x86_64, gcc 9.3.0*.

## Section 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.

In this lab, you'll be implementing a **shell** by emulating a subset of the Bash shell's functionality. Specifically, your shell has to implement functions related to running commands (in foreground and background, and chained), and redirecting their input and output.

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.

## Section 2. Exercises in Lab 2

There are a total of **four exercises** in this lab.

By now you have issued several commands through the shell on the Compute Cluster nodes as part of your Lab 1. The shell is actually just another program implemented using the process-related system calls discussed in the lectures. In the exercises, 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

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.

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

| | |
|---|---|
| **`Makefile`** (do not modify) | Used to compile your files. Just run `make`. (It will be replaced when grading) |
| **`check_zip.sh`** (do not modify) | For you to check your zip file before submission. |
| **`driver.c`** (do not modify) | Prewritten code that handles the main loop and tokenization of user input. (It will be replaced when grading) |
| **`myshell.h`** (do not modify) | Prewritten header file. (It will be replaced when grading) |
| **`myshell.c`** | Your definitions should go in here. Feel free to create your own functions and variables.<br><br>**`my_init()`** will be called when your shell starts up, before it starts accepting user commands.<br><br>**`my_process_command()`** will be called in the main loop. It should handle every user command except the **`quit`** command, which will be handled by **`my_quit()`**. The arguments passed into **`my_process_command()`** will be elaborated in the next section. |
| **`/programs`** | A folder containing sample tiny Bash programs with various runtime behaviors to aid your testing later. Feel free to open them up and take a look.<br><br>You can also create your own programs and run them with your shell. |

| | |
|---|---|
| - **`groot`** | Echoes “I am groot” for 2 minutes with a small random delay. |
| - **`infinite`** | Goes into an infinite loop, never terminates. Only used for the **bonus** exercise. |
| - **`lazy`** | Wakes up, echoes “Good morning...” and loops for 2 minutes. Takes a while (at least 5 seconds) to respond to the SIGTERM signal. |
| - **`result`** | Takes in a single number **`x`** and exits with a return status of **`x`**. |
| - **`showCmdArg`** | Shows the command line arguments passed in by the user. |

To compile the shell, either run **`make`** or invoke GCC directly:

```
$ gcc -std=c99 -Wall -Wextra -D_POSIX_C_SOURCE=200809L -D_GNU_SOURCE -o myshell driver.c myshell.c
```

You may use **`-Werror`** to make all warnings errors. You will not be penalized for warnings, but you are strongly encouraged to resolve all warnings. Warnings are indications of potential undefined behavior 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 make sure your program is free of memory errors.

To run the program, simply run:

```
$ ./myshell
```

You will then see an **`myshell>`** prompt where you can input commands in a similar manner with a Bash shell. The commands that should be accepted by this shell are elaborated in the following sections.

## Syntax Tokens for the Shell Commands

The reading and parsing of user commands for the shell **has been handled for you**. You are nonetheless encouraged to read and understand the provided code.

Tokens will be separated by whitespace (spaces or tabs). Tokenization is handled by **`driver.c`**, and passed to **`my_process_command()`** in **`myshell.c`** as an array of tokens with the last element of the array being **`NULL`**. The size of this array, including the **`NULL`** token, is passed in as **`num_tokens`**. You may want to print out the **`tokens`** array or look at the **`driver.c`** file for your own understanding.

The following table contains information about some terminologies used in this lab document.

| **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 (.).<br><br>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 (.)<br><br>If there is more than 1 argument, the arguments will be separated by whitespaces. |
| `{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 (.).<br><br>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. |

Example commands, as well as sample outputs, have been provided for most of the exercises to try and clear up any ambiguity.

## 2.1. Exercise 1: Basic Shell (1% demo + 2% submission OR 3% submission)

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:

| User command | Explanation |
|---|---|
| `{program} (args...)` | Example commands:<br>`/bin/ls`<br>`/bin/cat -n file.txt`<br><br>If there exists a file at `{program}`:<br>- Execute the file in a child process with the supplied arguments.<br>- **Wait** till the child process is done.<br><br>Else, print "`{program} not found`". |
| `{program} (args...) &` | **Note the <u>&</u> symbol at the end of the user command.**<br><br>If there exists a file at `{program}`:<br>- Run `{program}` in a child process with the supplied arguments.<br>- Print "**`Child[PID] in background`**", where **`PID`** is the process id of the child process.<br>- **Continue to accept user commands.**<br><br>Else, print "`{program} not found`". |
| `info` | Print all processes in the order in which they were run. You will need to print their process ids, their **current status** (Exited or Running), and for Exited processes, their exit status. |
| `quit` | Print "Goodbye!" and exit the shell. |

**Sample session**

```
myshell> info
```
*Nothing is printed here as there are no processes; this line is for visualization only*
```
myshell> /bin/echo hello
hello
myshell> info
[225] Exited 0
myshell> /bin/notaprogram
/bin/notaprogram not found
myshell> /bin/sleep 10 &
Child[226] in background
myshell> info
```
*This command is executed before the 10 seconds are up*
```
[225] Exited 0
[226] Running
```

```
myshell> info
        This command is executed after 10 seconds; note the change in state of [226]
[225] Exited 0
[226] Exited 0
myshell> ./programs/result 7
myshell> info
[225] Exited 0
[226] Exited 0
[227] Exited 7
myshell> ./programs/result 256
myshell> info
[225] Exited 0
[226] Exited 0
[227] Exited 7
[228] Exited 0
                                        This is not a typo; the exit status is 0.
myshell> ./programs/showCmdArg 5 23 1 &
Child[229] in background
myshell> [Arg 0]: 5
[Arg 1]: 23
[Arg 2]: 1
quit
                   Background processes can mess up your terminal input as shown here.
                     "quit" is another input for the shell and should be executed normally.
Goodbye!
```

**Notes:**

- Helpful comments for each sample session are written in *this style*, they **do not** represent a new line or any kind of output.
- **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 library functions and their family of related functions:
  - **`fork`**
  - **`wait`**
  - **`exec`**
  - **`stat/access`**

**Preventing fork bombs:**

- Add in the "`fork()`" call only after you have thoroughly tested 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 SoC Technical Service techsvc@comp.nus.edu.sg to ask them (nicely!) to unfreeze your account and kill off all your processes.

## 2.2. Exercise 2: Advanced Shell (2%)

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

| User command | Explanation |
|---|---|
| `wait {PID}` | Example command:<br>`wait 226`<br><br>`{PID}` is a process id created using "`{program} (args...)` &" syntax and has not yet been waited before.<br><br>If the process indicated by the process id is running, **wait** for it.<br><br>Else, continue accepting user commands.<br>No output should be produced. |
| `terminate {PID}` | Example command:<br>`terminate 226`<br><br>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}` is "Terminating" until `{PID}` is waited, or its state is checked again at "`info`".<br><br>Else, continue accepting user commands.<br>No output should be produced. |
| `{program1} (args1...) && {program2} (args2...) && ...` | Example command:<br>`/bin/ls && /bin/sleep 5 && /bin/pwd && /bin/ls`<br><br>`&&` is an operator that allows multiple "`{program} (args...)`" to be chained together and executed **sequentially**, and in the foreground.<br><br>It would be helpful to start with 2 chained commands, before extending your implementation to any number of chained commands.<br><br>1. If `{program1}` exists:<br>- Run and **wait** for `{program1}`.<br>- If `{program1}` was executed successfully (i.e., exit status 0), go back to step 1 with the next `{program2}`.<br>- Else, print "`{program1} failed`". There might be error output from the program that failed, which is fine. Do not continue executing the rest of the programs.<br><br>2. Else, print "`{program1} not found`". Do not continue executing the rest of the programs. |

**Extend** the following commands from exercise 1:

| User command | Explanation |
|---|---|
| `info` | Should now have an additional status "Terminating", in addition to the original "Running" and "Exited". |
| `quit` | **Terminate** all running processes by sending the **SIGTERM** signal and **wait** for them until they exit. Lastly, print "Goodbye" and exit the shell. |

**Sample session**

```
myshell> /bin/sleep 10 &
Child[226] in background
myshell> wait 226
            There should be a long pause here before the shell prompt appears again
myshell> info
[226] Exited 0
myshell> /bin/sleep 30 &
Child[227] in background
myshell> terminate 227
myshell> info
[226] Exited 0
[227] Exited 0
myshell> ./programs/lazy &
Child[228] in background
myshell> Good morning...
info
[226] Exited 0
[227] Exited 0
[228] Running
myshell> terminate 228
myshell> Give me 5 more seconds
info
                         This "info" command is run before the 5 seconds are up
[226] Exited 0
[227] Exited 0
[228] Terminating
myshell> info
                          This "info" command is run after the 5 seconds are up

[226] Exited 0
[227] Exited 0
[228] Exited 0myshell> /bin/sleep 3 && /bin/echo Hi &&
/bin/echo Bye
```

*There should be a 3 second pause before the next two lines are printed*

```
Hi
Bye
myshell> info
[226] Exited 0
[227] Exited 0
[228] Exited 0
[229] Exited 0
[230] Exited 0
[231] Exited 0
myshell> /bin/notaprogram && /bin/echo Hello
/bin/notaprogram not found
```

*"Hello" should not be printed*

```
myshell> info
[226] Exited 0
[227] Exited 0
[228] Exited 0
[229] Exited 0
[230] Exited 0
[231] Exited 0
myshell> /bin/sleep notanumber && /bin/echo Hello
/bin/sleep: invalid time interval 'notanumber'
Try '/bin/sleep --help' for more information.
/bin/sleep failed
```

*"Hello" should not be printed*

```
myshell> info
[226] Exited 0
[227] Exited 0
[228] Exited 0
[229] Exited 0
[230] Exited 0
[231] Exited 0
[232] Exited 1
myshell> /bin/echo Hi && /bin/notaprogram && /bin/echo Bye
Hi
/bin/notaprogram not found
myshell> info
[226] Exited 0
[227] Exited 0
[228] Exited 0
[229] Exited 0
[230] Exited 0
```

```
[231] Exited 0
[232] Exited 1
[233] Exited 0
myshell> /bin/echo A && ./programs/result 7 && /bin/echo B
A
./programs/result failed
myshell> /bin/echo A && ./programs/result 256 && /bin/echo B
A
B
myshell> ./programs/groot &
Child[238] in background
myshell> I am groot
I am groot
quit
Goodbye!
```

*"I am groot" should not be printed anymore once the shell quits*

**Notes:**

- **Read section 2.4.** to find out additional details about these requirements.
- You should look at the following C library functions and their family of related functions:
  - `wait`
  - `kill`

## 2.3. Exercise 3: Redirection (2%)

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.

| User command | Explanation |
|---|---|
| `{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.<br><br>Example commands:<br>`/bin/cat a.txt > b.txt`<br>`/bin/sort < test.txt > sorted.txt`<br>`/bin/sleep 2 2> error.log`<br><br>If `(< {file})` is present:<br>- If there exists a file at `{file}`:<br>  - `{program}` reads the contents of `{file}` as input.<br>- Else, print "`{file} does not exist`" and abort this command. Do not spawn the child process.<br>If `(> {file})` is present:<br>- 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.<br>If `(2> {file})` is present:<br>- 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.<br><br>Note: When you create a file, you should ensure that it has both read and write access for the current user, and read access for everyone else.. |
| `{program} (args...) (< {file}) (> {file}) (2> {file}) &` | **Note the & symbol at the end of the user command.**<br><br>Same behavior 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:<br>`/bin/printf hello\nworld\n > test.txt && /bin/sort < test.txt > sorted.txt && /bin/cat sorted.txt`<br><br>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`.<br><br>If a file is not found for (`< {file}`), just print "`{file} does not exist`". **A child process should not be spawned for this program**, therefore, we do not print "`{program} failed`" for this specific case.<br><br>Again, do not continue executing the rest of the programs once a program fails. |

**Sample session**

```
myshell> /bin/cat ./programs/result > ./a.txt
myshell> /bin/cat ./a.txt
#!/bin/bash
result=$1
exit $result
myshell> info
[299] Exited 0
[300] Exited 0
myshell> /bin/sort < ./a.txt > ./b.txt &
Child[301] in background
myshell> info
[299] Exited 0
[300] Exited 0
[301] Exited 0
myshell> /bin/cat ./b.txt
#!/bin/bash
exit $result
result=$1
myshell> /bin/sort < ./doesnotexist.txt
./doesnotexist.txt does not exist
myshell> info
[299] Exited 0
[300] Exited 0
```

```
[301] Exited 0
[302] Exited 0
myshell> /bin/printf world\nhello\n > ./a.txt && /bin/sort
< ./a.txt > ./b.txt && /bin/cat ./b.txt
hello
world
myshell> info
[299] Exited 0
[300] Exited 0
[301] Exited 0
[302] Exited 0
[303] Exited 0
[304] Exited 0
[305] Exited 0
myshell> /bin/echo hello && /bin/sort < ./doesnotexist.txt
hello
./doesnotexist.txt does not exist
myshell> info
[299] Exited 0
[300] Exited 0
[301] Exited 0
[302] Exited 0
[303] Exited 0
[304] Exited 0
[305] Exited 0
[306] Exited 0
```

*Note that a process for /bin/echo is created but NOT for /bin/sort due to invalid file*

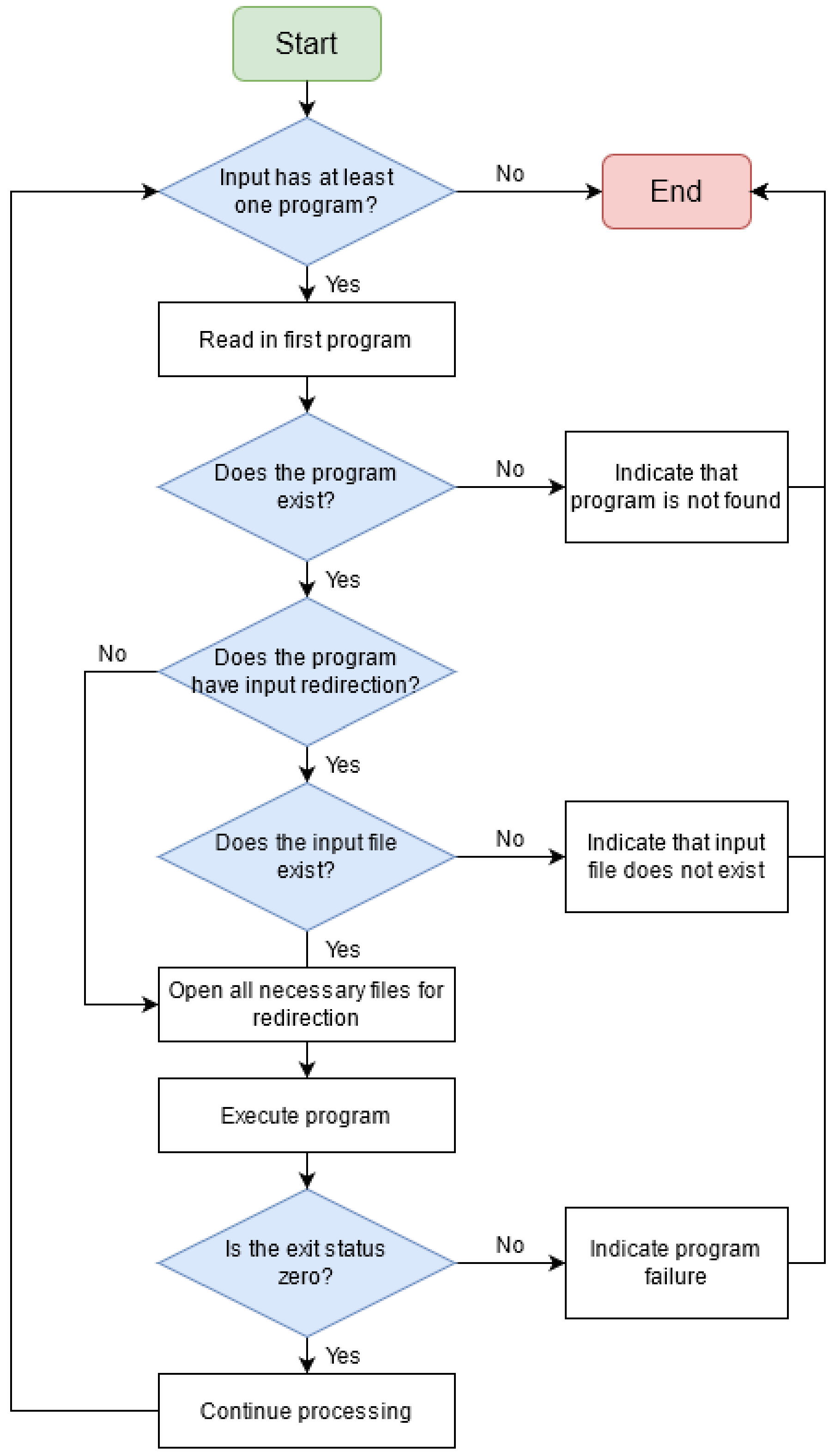
```
myshell> /bin/sleep notanumber 2> ./a.txt && /bin/echo Hello
/bin/sleep failed
myshell> /bin/cat ./a.txt
/bin/sleep: invalid time interval ‘notanumber’
Try '/bin/sleep --help' for more information.
myshell> quit
Goodbye!
```

**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 library functions and their family of related functions:
    - **open**

- **dup**

A flow chart about how the chained commands with redirection should be handled is shown below.


Start
Input has at least one program?
No
End
Yes
Read in first program
Does the program exist?
No
Indicate that program is not found
Yes
No
Does the program have input redirection?
Yes
Does the input file exist?
No
Indicate that input file does not exist
Yes
Open all necessary files for redirection
Execute program
Is the exit status zero?
No
Indicate program failure
Yes
Continue processing

## 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.

- Non-terminating programs such as `programs/infinite` will only be tested on the bonus exercise.

- Notice that **background processes** may sometimes mess up the display on your shell (there are some examples of this in our sample sessions), 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**.

- 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.

| **User command** | **Explanation** |
|---|---|
| `<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`".<br><br>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`".<br><br>Else, do nothing and continue accepting user input. |
| `fg {PID}` | If `{PID}` is currently stopped, get it to continue and **wait** for it. |

**Extend** the following command from exercise 3.

| **User command** | **Explanation** |
|---|---|
| `info` | Should now have an additional status "Stopped" , in addition to the original "Running", "Exited", and "Terminating". |

**Sample session**

```
myshell> ./programs/groot
I am groot
^Z
[267] stopped
myshell> info
[267] Stopped
myshell> /bin/sleep 100 &
Child[268] in background
myshell> wait 268
^Z
[268] stopped
myshell> info
[267] Stopped
[268] Stopped
myshell> ^Z^Z^Z
```

*Does nothing when Ctrl-Z is pressed multiple times when not waiting for a process*

*We press 'enter' here to get a clean prompt for the 'fg 267' command below*

```
myshell> fg 267
I am groot
I am groot
^Z
[267] stopped
myshell> ./programs/infinite
^C
[269] interrupted
myshell> info
[267] Stopped
[268] Stopped
[269] Exited 0
myshell> quit
Goodbye!
```

**Notes:**

- Again, please do this bonus using a **copy** of your Exercise 1-3 files. You will be submitting this bonus separately. Refer to Section 3 for the submission format.

### 2.6. 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.

Each exercise can be compiled.
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)
```

During execution, the script prints if the checks have been successfully conducted, and which checks failed. Successfully passing checks a. - d. ensures that we can grade your assignment. **Points might be deducted if you fail these checks.**

```
Expected Successful Output
Checking zip file....
Unzipping file: E0123456.zip
Transferring necessary skeleton files
[…]
All checks have passed successfully
```

### 2.7. FAQ

Frequently asked questions (FAQ) received from students for this assignment will be answered here. The most recent questions will be added at the beginning of the file, preceded by the date label. **Check this file before asking your questions.**

If there are any questions regarding the assignment, please post on the LumiNUS forum.

## Section 3. Submission through LumiNUS

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:

```
myshell.c
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:

```
myshell.c
```

Upload the zip file to the "Student Submissions Lab 2" folder on LumiNUS. Note the deadline for the submission is **Wed 22 Sep, 2pm**.

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