# CS2106: Introduction to Operating Systems

## Lab Assignment 1

## Advanced C Programming and Shell Scripting

**IMPORTANT**

The deadline of submission through Canvas: **11th Feb, 2023, 11.59 PM Saturday**

The total weightage is 8%:

- Exercise 1: 0.5%
- Exercise 2: 2% [Lab demo exercise]
- Exercise 3: 1%
- Exercise 4: 0%
- Exercise 5: 1%
- Exercise 6: 1%
- Exercise 7: 2%
- Exercise 8: 0.5%

Ensure the exercises work properly on the SoC Cluster.

# 1 General Information

Each lab assignment spans **two to three weeks** and consists of multiple exercises. Labs may be completed *individually or in a group of two*. Only one of the partner need to submit. Grade given will be the same for both partners.

## 1.1 Lab Demonstration

The 'lab demo exercise' is a specified exercise for each lab which can be demonstrated to your lab TA during lab sessions. Demoing the exercise is optional . If you choose to demo, part of the marks for that exercise will be given for the demo, with the remaining marks coming from grading. If not, demo exercise is graded for the total assigned grade. For e.g., exercise 2 has total grade of 2%.

- If you demo, 1% is allocated to the demo and exercise 2 is graded out of 1%.
- If you do not demo, exercise 2 is graded out of 2% (instead of 1%).

The demo serves as a good way to "kick start" your lab assignment efforts. You are **strongly encouraged to** finish the demo exercise before coming to the lab.

The remaining lab exercises are usually quite intensive. Do not expect to finish the exercise during the lab sessions. The main purpose of the lab session is to demo your exercise and clarify doubts with the lab TAs.

## 1.2 Lab Setup

For this semester:

- We provide a virtual machine image with Lubuntu (i.e. Ubuntu 20.04 with LXQt instead of GNOME) installation that can be used with VirtualBox on your personal computer (link).
- You can use the SoC Compute Cluster to test your assignments. You can also develop your assignments on these machines (more will be shared in exercise 1).
- Alternatively, you might install Ubuntu 20.04 natively on your personal computer.

You may use any of the three methods above to develop your assignments. Take note that: **Your submissions will be tested on the SoC cluster.**

To read details about the nodes on the SoC Compute Cluster check: `https://dochub.comp.nus.edu.sg/cf/guides/compute-cluster/hardware`.

You must use your SoC account to use these nodes. If you do not have an SoC account, you can retrieve or create an account using the link provided here: `https://mysoc.nus.edu.sg/~newacct/`

To use these nodes, you **must enable SoC Compute Cluster** from your MySoC Account page (`https://mysoc.nus.edu.sg/~myacct/services.cgi`).

Exercise 1 will run through the basics of connecting to these nodes remotely, along with steps to set up a development environment that will be useful for all future lab assignments.

## 1.3 Setting up the exercise

For every lab, we will release two files under the Canvas "**Labs**" folder:

- **`labX.pdf`**: Lab question document, describing the specification and expected output for all exercises.
- **`labX.tar.gz`**: An archive for setting up the directories and skeleton files given for the lab.

For unpacking the archive:

1. Copy or download the archive `labX.tar.gz` into your account.
2. Enter the following command in the terminal (console):

   ```
   tar -xf labX.tar.gz
   ```

   Remember to replace X with actual lab number.
3. The above command should setup the files in the following structure:

   ```
   ex2/                subdirectory for exercise 2
       ex2.c           skeleton files for exercise 2
       sample.in       sample test inputs
       sample.out      sample test outputs
   ex3/
       ...
   ex4/
   ```

# 2 Exercises in Lab 1

There are **eight exercises** in this lab. This lab should familiarize you with:

- SoC Compute Cluster and Slurm,
- compiling and running C programs in Linux,
- the Linux shell,
- C programming and some advanced aspects.

You will be writing C programs and shell scripts (shell commands) to achieve simple tasks. The techniques and shell commands used in these exercises are commonly used in OS related topics and will help you to complete future lab assignments.

## 2.1 Exercise 1: Setting up your development environment – 0.5%

We can connect to the remote nodes on the Soc Compute Cluster using our terminal (Linux or Mac) or command prompt (Windows). Think of the terminal or command prompt as an interactive program that allows you to type commands to complete tasks without a graphical interface.

### 2.1.1 Using the Virtual Machine

The simplest option to set up your development environment is by using the virtual machine (VM) image provided. However, note that is provided as an easy alternative for you to implement the assignment. You will HAVE to check your implementation on the cluster nodes before submitting.

Please download the image from the Canvas Assignment. Then, install the virtual box (link) on your computer. After installing the virtual box, load the given image by **Machine →Add →CS2106-2010-VM**. Finally, start the VM. Refer to (link) on how to share files between host and VM.

**For Mac OS as host:** Sometime you may come across "Kernel not found error". In this case you will need to give permission to Oracle virtual box in your computer. Go to System Preference →Security & Privacy. Then, **Allow** "System Software from developers Oracle America, Inc was blocked from loading" (link) option.

We have collected possible errors you may encountered while using VM in the following document (link). Please check the document for your error before asking the TA.

**Note:** VM may not work for the latest MacBook M1 Pro or M2. Please use the SoC cluster or Ubuntu installed system in that case.

### 2.1.2 SoC Computer Cluster

SoC computer cluster uses Slurm (link), a scalable cluster management and job scheduling system. You must login to Slurm login nodes (xlog0, xlog1, or xlog2) to run the assignment on SoC cluster nodes. There are two ways to connect to the login nodes:

#### A Secure Shell (SSH) to xlog[0 2] through `stu.comp.nus.edu.sg`

Secure Shell (SSH) is a network protocol that allows us to work with remote nodes securely. To reach the SoC Compute Cluster nodes, we need to perform the **`ssh`** command twice – once to get access to the `stu.comp.nus.edu.sg` node (link), then once more from the `stu` node to the xlog[0 2] node. The steps are as follows:

1. In your computer's terminal or command prompt, enter

   ```
   ssh <your_soc_account_id>@stu.comp.nus.edu.sg
   ```

   You will be prompted for your SoC account password. Once entered, you should see the shell of the remote `stu` node.
2. To access xlog[0 2], you can type e.g., `ssh xlog0` for xlog0. You will again be prompted for your password. Once entered, you should see the shell of the remote xlog0 node, placed at the home directory of your account.

Alternatively, we can replace the above two steps with a single `ssh` command using the `-J` flag. You will be prompted twice for your SoC account password, once for `stu`, and another for xlog0.

```
ssh -J <your_soc_account_id>@stu.comp.nus.edu.sg <your_soc_account_id>@xlog0
```

#### B Secure Shell (SSH) directly to xlog[0 2] from the SoC Network

Finding it tedious to have to enter your password twice? You can `ssh` directly into the xlog remote node **if you are connected to the SoC VPN**![1]

After connecting to SoC VPN, simply enter the command (using xlog0 as an example):

```
ssh <your_soc_account_id>@xlog0.comp.nus.edu.sg
```

For those interested in further optimizing your developer experience, you can read up on **hostnames** and **SSH keys**, to eliminate the need to retype the long host addresses and your SoC account password respectively.

### 2.1.3 Working with Slurm

Once you have logged into the xlog nodes, there are many cluster commands which will help you to run your program as jobs on the nodes. Please refer to the following documentation to help you get started. (link). You should have some basic shell scripting commands to compile and execute the exercises (more details in ex2).

### 2.1.4 Basic Commands in the Terminal

Once you have setup your VM or connected to cluster, you can navigate around. To do so in the shell, you need to be familiar with basic commands such as:

- pwd: print current working directory
- `cd`: change directory
- `ls`: list files in current directory

[1] https://dochub.comp.nus.edu.sg/cf/guides/network/vpn

You can also type `man <command>` on the terminal to view the user manual of the command and read more about it.

#### 2.1.5 Development Set-Up

When programming your assignments, you have a few options. You may choose to:

1. Code directly on the remote node (xlog[0 2]), or
2. Use your own Ubuntu set-up.

For coding directly on the remote node, you can choose to either:

1. Use terminal-based text editors like Vim, which might seem less user-friendly, or
2. Setup remote development using SSH with your favourite code editor for a smoother development experience. For example, Visual Studio Code provides a SSH extension that lets you modify files on the xlog[0 2] remote nodes as if you were editing a local file[2].

**Note:** Please **do not** develop on the `stu.comp.nus.edu.sg` remote node directly.

#### 2.1.6 File Transfer

After the development environment is set up, you can download the `lab1` files from Canvas. Follow the instructions in Section 1.3 to unpack the files for Lab 1.

To transfer your code to and from the remote nodes, you can either use the `sftp` command or `scp` command, to transfer your files onto the node. Use `man` to find out how to use `sftp`.

**Note:** 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.

#### 2.1.7 Grading

Simply by submitting this lab assignment, you will obtain the 0.5% for this first exercise! That being said, do take the time to run through this first exercise fully, as your development set-up and your familiarity with the various commands directly affects your efficiency for this lab and all future labs.

[2] https://code.visualstudio.com/docs/remote/ssh

## 2.2 Exercise 2: Singly Linked List in C [Lab Demo] (1% demo + 1% submission or 2% submission)

Exercise 2 requires you to implement functionalities for a singly linked list. You will be more familiar with C syntax and appreciate the challenge behind implementing different parts of the operating system.

### Singly Linked Lists

Singly linked lists are commonly used in the Linux Kernel, such as for process and memory management. For this exercise, we represent a node in our linked list as follows (in `node.h`):

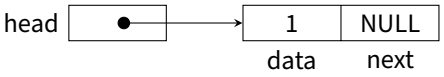
```
typedef struct NODE {
    int data;
    struct NODE *next;
} node;
```

The list representation looks like this:

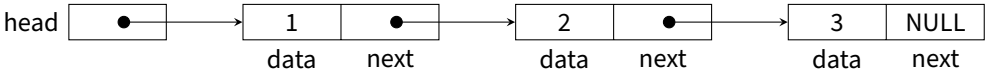
```
typedef struct {
    node *head;
} list;
```

Initially, the list will be empty:


Upon inserting the first node with data equals 1, we will have a singly linked list:




Thereafter, every insertion will expand this singly linked list:




#### 2.2.1 Task

You need to implement five functions shown below to work with the singly linked list:

`void insert_node_at(list *lst, int index, int data)`

This function inserts a new node that contains `data` at `index` of `lst` counting from the head (index starts from 0). You should use `malloc` to allocate memory to create the new node. Assume that `index` is between 0 and length of the list, both inclusive.

`void delete_node_at(list *lst, int index)`

This function deletes the node at `index` of `lst` counting from the head (index starts from 0). You should use `free` to delete memory allocated. Assume that `index` is between 0 inclusive and the length of the list exclusive. If the head node is deleted, the next node at index 1 (before deletion) should be the next head, if such a node exists. If no next node exists, the head of `lst` should be set to `NULL`.

`int search_list(list *lst, int element)`

This function searches for the `element` in the entire list and **returns** the index of the `element` in the list if present, else it returns -1. If the list is empty, it return -2. (**Note:** `print_index()`, which is already provided in `ex2.c` is automatically called to print the index returned).

`void reverse_list(list *lst)`

This function reverses the order of the nodes in `lst`. The head pointer of `lst` should now point to the original "tail" node. You should not be reallocating memory for the nodes or modifying the data of the nodes, but instead **modify the pointers** for this reversal. You may use additional memory to store temporary data during the operation, though it is not necessary.

`void reset_list(list *lst)`

This function deletes all nodes in `lst` and resets its head to `NULL`.

We defined specific macros for each of the functions:

```
#define PRINT_LIST      0
#define INSERT_AT       1
#define DELETE_AT       2
#define SEARCH_LIST     3
#define REVERSE_LIST    4
#define RESET_LIST      5
```

The function `print_list` has already been written for you within the runner `ex2.c`. Feel free to look at the `ex2.c` to understand how the runner works. Understanding how the runner works will help greatly when doing the next exercise.

The folder structure of `ex2` is as follows:

/ex2

| | |
|---|---|
| ex2.c | (not to be modified) |
| node.h | (not to be modified) |
| node.c | (modify) |
| *.in | (files used for testing) |
| *.out | (files used for testing) |
| Makefile | (not to be modified) |

You should **only modify node.c** for this exercise. Changes to any other file will be overwritten when we run the grading script.

### 2.2.2 Compiling and Testing on Cluster Nodes (Method 1)

After you have finished implementation, use the following command to compile your code. This produces the executable ex2 by running gcc compiler with c99 standard, enabling warnings, and creating an output ex2.

```
$ srun gcc -std=c99 -Wall -Wextra node.c ex2.c -o ex2
```

Alternatively, you can use Makefile provided to compile your code:

```
$ srun make
```

You may use -Werror option in gcc to make all warnings into errors. Marks will not be penalized for having warnings, but we may use them to aid us in finding bugs in your programs.

You can use the sample test case we have provided to test your code. (Note the - at the end!)

```
$ srun ./ex2 < sample.in | diff -Z sample.out -
```

### 2.2.3 Compiling and Testing on Cluster Nodes using Shell Script (Method 2)

Alternatively, you can write a simple shell script to compile and run your job on cluster node. First, open an editor and create a file called run.sh. Write the following code in run.sh to test ex2 on cluster nodes.

run.sh

```
#!/bin/sh

make
./ex2 < sample.in | diff -Z sample.out -
```

The file run.sh should be in the ex2 folder. You can run the script on the xlog terminal as follows:

```
$ chmod +x run.sh  (needs to be done only once)
$ srun run.sh
```

### 2.2.4 Compiling and Testing on VM

If you are using the VM for your assignment, you can compile and run the code as follows:

```
$ make
$ ./ex2 < sample.in | diff -Z sample.out -
```

**Note:** The above bash command passes the `sample.in` input file into the test runner and compares the output against the expected. Details about how this command runs:

- The bash spawns two processes. The first process runs `ex2` and the second process runs `diff`.
- We used **input redirection (<)** to replace the standard input (stdin) with the file `sample.in` for the first process (running `ex2`)
- We used a **pipe (|)** to pass the output of the first process (running `./ex2 < sample.in`) into the input of the second process (running `diff sample.out -`). The `-` sign stands for the second input file being replaced with standard input (or in this case, the output of `ex2`).

Apart from `sample.in`, two more test cases have been given to help verify your program. Also, take note that the runner will call `reset_list` to delete all nodes in the list before it terminates the program.

*Please test your code rigorously on your own. The actual grading after submission will be more rigorous and will check more test cases (including corner cases).*

Refer below for an explanation of the sample input. The input file uses a specific numbering scheme to refer to the five functions defined above:

sample.in

```
1 0 1     // insert_node_at(lst, 0, 1)
0         // print_list
1 0 3     // insert_node_at(lst, 0, 3)
0
1 1 2
0
1 0 100
0
1 4 200
0
2 1       // delete_node_at(lst, 1)
0
3 100     // search_list(lst, 100)
4         // reverse_list(lst)
0
3 2
5         // reset_list(lst)
0
1 0 1000
0
```

sample.out

```
[ 1 ]
[ 3 1 ]
[ 3 2 1 ]
[ 100 3 2 1 ]
[ 100 3 2 1 200 ]
[ 100 2 1 200 ]
{0}
[ 200 1 2 100 ]
{2}
[ ]
[ 1000 ]
```

#### 2.2.5 Demo Grade

To get the demo exercise grade for this lab, you have to show your lab TA that you are familiar with basic bash syntax and have a working singly linked list that works for all given test cases.

## 2.3 Exercise 3: Function Pointer – 1%

Exercise 3 extends the functionalities of the singly linked list implementation of exercise 2 by adding some operations on its nodes. We can do this by using function pointers.

### Function Pointers

Unlike normal pointers, which point to memory location for **data storage**, a function pointer points to a **piece of code (function)**. By dereferencing a function pointer, we **invoke the function** that is referred to by that pointer. This technique is commonly used in **system call / interrupt handlers**.

For example, we can define a function pointer:

```
void (*fptr) (int);
```

To understand this declaration (check out this rather handy website), imagine if you replace `(*fptr)` as F, then you have:

```
void F (int);
```

So, F is "a function that takes an integer as input and returns nothing (`void`)". Now, since `(*fptr)` is F, `fptr` is "**a pointer to** a function that takes an integer as input and returns nothing (`void`)". Refer to this link for more information on function pointers.

Let's use function pointers to define and use a group of functions that can **map** different operations to the singly linked list.

#### 2.3.1 Task

You need to implement the test runner `ex3.c` and three functions in `node.c`. The runner:

- **reads the input file provided as a command line argument** (reading from a file can be done using any library. We recommend using `stdio` (`fopen`, `fclose`, `fread`). Make sure that you gracefully handle an invalid file name), and
- **applies the operations listed in the input file** on the singly linked list.

The runner should also free any memory it allocates and reset the list before the program terminates.

The macros corresponding to the functions that can be applied on the list are:

```
#define SUM_LIST        0
#define INSERT_AT       1
#define DELETE_AT       2
#define SEARCH_LIST     3
#define REVERSE_LIST    4
#define RESET_LIST      5
#define LIST_LEN        6
#define MAP             7
```

INSERT, DELETE, SEARCH, REVERSE, and RESET operations are from exercise 2 (no changes needed). We have added **MAP** and **LIST_LEN** and replaced **PRINT_LIST** with **SUM_LIST**. Implement the three functions:

```
long sum_list(list *lst)
```

This function sums the data of all nodes in the `lst` and returns the sum. If `lst` is empty, return 0.

```
int list_len(list *lst)
```

This function finds the number of nodes in the `lst` and returns the length.

```
void map(list *lst, int (*func) (int))
```

This function updates `lst` by applying `func` to the `data` element of every node.

While `sum_list` and `list_len` are straightforward, the `map` function (MAP) uses function pointers, applying the given function `func` on each element of the singly linked list.

**MAP** can be used to apply five operations on the list. The indices for these operations are:

| | |
|---|---|
| 0 | add_one |
| 1 | add_two |
| 2 | multiply_five |
| 3 | square |
| 4 | cube |

The implementation for these functions is provided in `functions.c`. The runner `ex3.c` simply calls the right function based on the index given in the input file.

To apply the five MAP operations, initialize an array of function pointers with indices 0 to 4. This array has already been declared (but not initialized) in `function_pointers.h`. Use the index from the input file to call the corresponding map function. This array of function pointers is:

- named **`func_list`**,
- has been declared in `function_pointers.h`, and
- **will need to be initialized in `function_pointers.c`**. You may choose to use `update_functions` (defined in `function_pointers.h` and implemented in `function_pointers.c`) to help you with the initialization. `update_functions` is called in the main function of `ex3.c` (please do not modify this call).

The folder structure of `ex3` is given below. Do not modify files unnecessarily.

/ex3

| | |
|---|---|
| ex3.c | (modify) |
| node.c | (modify) |
| node.h | (not to be modified) |
| function_pointers.c | (modify) |
| function_pointers.h | (not to be modified) |
| functions.c | (not to be modified) |
| functions.h | (not to be modified) |
| *.in | (files used for testing) |
| *.out | (files used for testing) |
| Makefile | (not to be modified) |

### 2.3.2 Compiling and Testing

Use the Makefile provided to compile your code:

```
$ srun make
```

Command make uses the instructions found in the Makefile to compile your code. Note that you must either call gcc or make to compile your code (there is no need to call both of them).

Test your ex3 as follows:

```
$ srun ./ex3 sample.in > res.out
$ srun diff -Z res.out sample.out # (compare your output with expected
↪  output)
```

Apart from the sample, we have provided two more test cases for testing. Do take note that getting the right answer for these files will not guarantee full marks for this exercise (see **Exercise 4**). *Please test your code rigorously on your own. Other test cases will be used during grading.*

A sample input and output are shown below:

sample.in

```
1 0 1    // first three same as ex2
1 0 3
1 1 2
0        // sum list and prints it
7 0      // runs maps on list with add_one function
0
7 2      // runs maps on list with multiply_five function
0
1 2 4
1 3 5
0
6        // find length of list and prints it
7 4
0
```

sample.out

```
6
9
45
54
5
12564
```

## 2.4 Exercise 4: Checking for Memory Errors – 0%

Exercise 4 introduces `valgrind`, a tool commonly used to identify and fix memory errors (memory leak detection / out of bound array access etc).

We used `malloc` or `free` a couple of times in ex2 and ex3 so there is a possibility of memory leaks occurring if any resource obtained dynamically is not freed.

To check for these errors using valgrind, `cd` into `ex2` directory and run the below command:

```
$ srun valgrind ./ex2 < sample.in > res.out
```

An output without errors should look as follows:

Sample Output

```
==12539== Memcheck, a memory error detector
==12539== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==12539== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright
↪  info
==12539== Command: ./ex2
==12539==
==12539==
==12539== HEAP SUMMARY:
==12539==     in use at exit: 0 bytes in 0 blocks
==12539==   total heap usage: 9 allocs, 9 frees, 16,488 bytes allocated
==12539==
==12539== All heap blocks were freed -- no leaks are possible
==12539==
==12539== For lists of detected and suppressed errors, rerun with: -s
==12539== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

`valgrind` has done the hard work of checking for any potential memory leaks for us!

Now suppose our `ex3` had some memory leaks, `valgrind` will detect these problems and show us how to rerun it to view additional details. An output with errors might look as follows:

Sample Output

```
==3388== Memcheck, a memory error detector
==3388== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3388== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright
↪  info
==3388== Command: ./ex3
==3388==
==3388==
==3388== HEAP SUMMARY:
==3388==        in use at exit: 8 bytes in 1 blocks
==3388==        total heap usage: 7 allocs, 6 frees, 8,824 bytes allocated
==3388==
==3388== LEAK SUMMARY:
==3388==        definitely lost: 8 bytes in 1 blocks
==3388==        indirectly lost: 0 bytes in 0 blocks
==3388==        possibly lost: 0 bytes in 0 blocks
==3388==        still reachable: 0 bytes in 0 blocks
==3388==        suppressed: 0 bytes in 0 blocks
==3388== Rerun with --leak-check=full to see details of leaked memory
==3388==
==3388== For counts of detected and suppressed errors, rerun with: -v
==3388== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

#### 2.4.1 Task

You need to ensure that there are no memory errors in ex2 and ex3 with `valgrind`.

This exercise is not graded but we will deduct marks if there are memory errors in either `ex2` or `ex3` (even if the expected output is correct).

## 2.5 Exercise 5: Shell scripting to find out about our system – 1%

In the next three exercises, we will write some shell scripts on the bash shell! In exercise 5, we write a simple shell script to learn more about our operating system.

### Shell Scripts

A shell script is a list of commands designed to be run by the Linux shell. Earlier we mentioned `cd` and `pwd`, examples of commands you can use in the terminal (shell). Think of a shell script as running a sequence of these commands.

Refer to the Shell Scripting Primer.pdf document provided to understand the basics of shell scripting. You can also refer to the following link, or use the man pages and online search to find out more about bash.

#### 2.5.1 Task

You need to write a shell script that outputs the following details about your system. Use the resources mentioned above to find the relevant commands to call to obtain the information needed.

| Detail | Description |
|---|---|
| Hostname | Name given to each device on a network. Used for identification |
| Machine Hardware | Hardware architecture of a system |
| Max Process ID | Maximum value of PID, at which PIDs wraparound |
| User Processes | Number of processes currently running for *this* user |
| User With Most Processes | Name of user with the most processes currently running |
| Memory Free (%) | Percentage of total memory that is currently free |

The output of the script should look as follows (actual values differ based on your system):

Sample Expected Output

```
Hostname: cs2106-2010-vm
Machine Hardware: Linux x86_64
Max Process ID: 4194304
User Processes: 32
User With Most Processes: root
Memory Free (%): 80.9471
```

We have provided in `ex5` folder a skeleton bash script `check_system.sh` that you can use to start work on this. You do not have to install anything else on your system to get the above information.

To get you started, here are some helpful commands that may be of use to you:

- `sort`
- `awk`

- pipe in shell (`|`)

**Tip:** Once you have figured out the commands that you might need, you can run them directly in the terminal to test them first before using them in the `bash` script

## 2.6 Exercise 6: Know your syscalls – 1%

In exercise 6, we will write another shell script that lists the system calls made by a given C program.

A system call allows a user program to request services that are provided by the operating system. To find out what system calls are made by a program, we use a tool known as `strace`.

In the `ex6` folder, you will find a sample C program `pid_checker.c` that prints its own process ID and its parent's process ID. Our script will run `strace` on the executable compiled from this code. Do not modify `pid_checker.c`. A bash script skeleton `check_syscalls.sh` has been provided for you to fill in.

### 2.6.1 Task

You need to complete bash script `check_syscalls.sh` to obtain a report on the system calls made by the program compiled from `pid_checker.c` and the time spent in each call.

The expected output has the following format (values may differ):

Sample Expected Output

```
Printing system call report
Process ID: 12785
Parent Process ID: 12782
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 21.97    0.000266          38         7           mmap
 15.11    0.000183          30         6           pread64
  9.66    0.000117          39         3           fstat
  9.17    0.000111          37         3           mprotect
  7.60    0.000092          30         3           brk
  6.94    0.000084          42         2           openat
  5.62    0.000068          34         2           close
  4.95    0.000060          30         2         1 arch_prctl
  3.88    0.000047          47         1           munmap
  2.97    0.000036          36         1           getpid
  2.89    0.000035          35         1         1 access
  2.73    0.000033          33         1           read
  2.64    0.000032          32         1           execve
  2.39    0.000029          29         1           getppid
  1.49    0.000018           9         2           write
  ------ ----------- ----------- --------- --------- ----------------
100.00    0.001211                    36         2 total
```

Again, reading the Linux manual for `strace` should allow you to do this exercise rather quickly.

## 2.7 Exercise 7: More Fun with Shell Scripting – 2%

In exercise 7, we will write a bash script allowing a user to input a prefix and some numbers to create dummy files with the inputted prefix and numbers. Then, we will rename the newly created files with a new prefix inputted by the user again.

Again, you may refer to the Shell Scripting Primer.pdf document provided or the following link to understand the basics of shell scripting. Additionally, here are some commands you may find useful:

- bash `if` statement
- bash `for` loop
- `touch`, `cat`
- `mv`

### 2.7.1 Task

A bash script skeleton `file_renaming.sh` has been provided in the `ex7` folder. Complete the script to do the following:

- Read prefix (`prefix`) from the user. If the prefix contains any characters other than [a-z or A-Z] print INVALID and request for a valid prefix.
  e.g., ABC
- Read the number of files (N) to create from the user.
  e.g., `N = 2`
- Read the N numbers ($num_1$, $num_2$,.., $num_N$) from the user.
  e.g., `123`, `3456`
- Create N files with the filenames as follows: `prefix_num`$_N$`.txt`. Print INVALID and request for a valid number if $num_1$ contains any other character other than numbers between [0-9].
  e.g., `ABC_123.txt`, `ABC_3456.txt`
- Read a new prefix (`new_prefix`) from the user (same constraints apply).
  e.g., XYZ
- Rename all the files with the new prefix: `new_prefix_num`$_N$`.txt`
  e.g., renamed files: `XYZ_123.txt`, `XYZ_3456.txt`

To simplify your script, assume that the user will enter an INVALID number or prefix only on the first attempt. The user's second attempt will always be a valid entry.

**Note:** Please use the prompts (i.e., echo statements) provided in the skeleton code. The lab is auto-graded and deviations from expected prompts will lead to wrong grading. Any deviations will be penalized.

The expected input/output are as follows:

Sample Expected Output

```
Enter prefix (only alphabets):
ABC
Number of files to create:
2
Enter 2 numbers:
123
3456

Files Created
ABC_123.txt ABC_3456.txt

Enter NEW prefix (only alphabets):
XYZ

Files Renamed
XYZ_123.txt  XYZ_3456.txt
```

Sample Expected Output

```
Enter prefix (only alphabets):
A123
INVALID
Please enter a valid prefix [a-z A-Z]:
AbC
Number of files to create:
2
Enter 2 numbers:
12df
INVALID
Please enter a valid number [0-9]:
123
34b
INVALID
Please enter a valid number [0-9]:
3456

Files Created
AbC_123.txt AbC_3456.txt

Enter NEW prefix (only alphabets):
XYZ

Files Renamed
XYZ_123.txt  XYZ_3456.txt
```

## 2.8 Exercise 8: Check your archive before submission – 0.5%

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

The script checks the following:

1. The name or the archive you provide matches the naming convention mentioned in Section 3
2. Your zip file can be unarchived, and the folder structure follows the structure presented in Section 3
3. All files for each exercise with the required names are present
4. Each exercise can be compiled and/or executed.
5. The output your exercise produces using our sample input matches the expected output.

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. You will receive 0.5% simply for having a valid submission file!

Expected successful output

```
Checking zip file....
Unzipping file: E0123456.zip Transferring necessary skeleton files
ex2: Success
ex3: Success
ex5: Success
ex6: Success
ex7: Success
```

Expected successful output for pairs

```
Checking zip file....
Submitting as a pair
Unzipping file: E0123456_E0234567.zip Transferring necessary skeleton
↪  files
ex2: Success
ex3: Success
ex5: Success
ex6: Success
ex7: Success
```

# 3 Submission through Canvas

Zip the entire folder `E0123456` as `E0123456.zip` (**use your NUSNET id, NOT your student no A012...B, and use capital 'E' as prefix**). The folder `E0123456` should have five sub folders. Do **not** add any additional folder structure during zipping. Note the top-level folder (i.e., `E0123456`). The file structure after unzipping `E0123456.zip` should be:

```
E0123456/
  ex2/
    node.c
  ex3/
    function_pointers.c
    node.c
    ex3.c
  ex5/
    check_system.sh
  ex6/
    check_syscalls.sh
  ex7/
    file_renaming.sh
```

You can check your file structure 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 1" Assignment on Canvas. Note the deadline for the submission is **11th Feb, 2023, 11.59 PM Saturday**.

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