CS2106: Introduction to Operating Systems

Lab Assignment 1 (A1)
Advanced C Programming and Shell Scripting

Important:

The deadline of submission through LumiNUS: Sat, 4 Sep, 2pm
The total weightage is 7%:

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

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. General Information

Here are some simple guidelines that will come in handy for all future labs.

1.1. Lab Assignment Duration & Lab Demonstration

Each lab assignment spans about **two to three weeks** and consists of multiple exercises. One of the exercises is chosen to be the "lab demo exercise" which you may **choose** to demonstrate to your lab TA. If you choose to demo, part of the marks for that exercise will be given for the demo, while the other part will come from grading. If you choose not to demo, the marks from that exercise will be fully determined via grading.
For example, in this lab,

- If you demo, 1% is allocated through demo, while exercise 1 will be graded out of 1%.
- If you do not demo, exercise 2 will be graded out of 2% (instead of 1%).

This demonstration 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 allocated lab session. The main purpose of the lab session is to demo your exercise and clarify doubts with the lab TAs.

1.2. Lab Setup

Since the classes are largely running online, we do not have a physical lab where you can go and sit down to complete your lab assignments. For this semester:

- You will 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).
- Additionally, 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.
- 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. However, take note that:

Your submissions will be tested on one of the following machines from SoC Compute Cluster: hostnames (nodes) xcne0 - xcne7.

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

Nodes **xcne5**, **xcne6** and **xcne7** have been reserved for CS2106 for the entire semester. However, you may use any available node xcne0 - xcne7 to test your assignments before submissions.

The software configuration for these nodes follows:

```
Ubuntu 20.04
gcc version 9.3.0
GNU bash, version 5.0.17(1)-release
```

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 exercises

For every lab, we will release two files under the LumiNUS “Labs” folder:

- **`labX.pdf`**: A document to describe the lab question, including the specification and the expected output for all the 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 the ***X*** with the actual lab number.
3. The above command should setup the files in the following structure:

ex2/		subdirectory for exercise 2
	ex2.c	skeleton file for exercise 2
	testY.in	sample test inputs, Y= 1, 2, 3, ...
	testZ.out	sample outputs, Z = 1, 2, 3, …
ex3/		
	…	Similar to ex2
ex5/		Similar to ex2
…		

Section 2. Exercises in Lab 1

There are **seven exercises** in this lab. The main motivation for this lab is to familiarize you with:

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

As such, you will need to write a combination of C programs and shell scripts (shell commands) to achieve some simple tasks. The **techniques and shell commands** used in these exercises are quite commonly used in OS related topics, and they will help you in completing the next lab assignments.

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

There are many ways to go about doing the development of your lab assignment. This exercise will run through some basic shell commands, as well as some advanced options that may aid you with your development. You should find what set-up works best for you.

Before we begin, make sure you have gone through Section 1.2, especially the part on the SoC Compute Cluster. We will be using our terminal (for Linux or Mac) or command prompt (for Windows) to connect to the remote nodes. We will also be downloading our assignment files using the terminal. You can think of the terminal or command prompt as an interactive program that allows you to type commands to complete some tasks without a graphical interface.

We will now discuss two possible ways to connect to the remote nodes:

A. Secure Shell (SSH) to xcne0-xcne7 through `sunfire`

Secure Shell (SSH) is a network protocol that allows us to work with remote nodes securely. We will first utilize this protocol to access our SoC Compute Cluster nodes.

To reach the SoC Compute Cluster nodes (xcne0-xcne7), we would normally need to perform the **`ssh`** command twice – once to get access to the `sunfire` node, then once more from the `sunfire` node to the xcne node. The steps are as follows:

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

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

You will be prompted for your SoC account password. Once entered, you should be looking at the shell of the remote `sunfire` node, and you will be placed at the home directory of your `sunfire` account.

2. To access xcne0 – xcne7, you can type e.g., **`ssh xcne6`** for xcne6. You will once again be prompted for your password. Once entered, you should now see that you are now in the xcne node.

Alternatively, instead of using these two steps, we can us a single `ssh` command using the `-J` flag. For example, in your computer's terminal or command prompt, enter:

```
ssh -J <your_soc_account_id>@sunfire.comp.nus.edu.sg
<your_soc_account_id>@xcne6
```

You will be prompted twice for your SoC account password, once for `sunfire`, and another for the xcne6 node. Once entered, you should be looking at the shell of the remote xcne6 node, and you will be placed at the home directory of your account.

B. Secure Shell (SSH) directly to xcne0-xcne7 from the SoC Network

Finding it tedious to have to enter your password twice? You can **`ssh`** directly into the xcne remote node **if you are connected to the SoC VPN** (https://dochub.comp.nus.edu.sg/cf/guides/network/vpn)!

To do so, simply enter the command (using xcne6 as an example):

```
ssh <your_soc_account_id>@xcne6.comp.nus.edu.sg
```

For those interested in further optimizing the developer experience for yourself, 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.

C. Basic Commands in the Terminal

Once you are connected to the remote shell, 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

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

D. Development Set-Up

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

1. code directly on the remote node (xcne0-7),
2. work locally using the virtual machine image provided, or
3. use your own Ubuntu set-up.

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

a. use terminal-based text editors like Vim, which might seem less user-friendly, or
b. set up remote development using SSH with your favorite code editor for a smoother development experience. For example, Visual Studio Code provides a SSH extension that lets you modify files on the xcne0-7 remote nodes as if you were editing a local file (https://code.visualstudio.com/docs/remote/ssh).

Please **do not** develop on the `sunfire` remote node, as it is not running the same OS as the xcne0-7 nodes.

E. File Transfer

Once you set up your development environment, you can download the `lab1` files using the terminal of your development environment. For example, if you are using the virtual machine image, this will be the terminal of the instance. You will need to first fetch the zip files from a download link. You can use either **`wget`** or **`curl`** to do so. Use **`man`** / **`help`** to find out how to use **`wget`** or **`curl`**.

Download link:
https://www.comp.nus.edu.sg/~ccris/cs2106_ay2122s1/lab1.tar.gz

Once the files are downloaded, follow the instructions in Section 1.3 to unpack the files for `lab1`.

To transfer your code to and from the remote nodes, you can either use the **`sftp`** command, which uses the secure file transfer protocol to transfer your files onto the node. Use **`man`** to find out how to use **`sftp`**.

Alternatively, you can create a **private git repository e.g., on GitHub**, to commit and retrieve code. Make sure your **repository is set to private** and that nobody can access your code. We have had cases in the past where students used public repositories and their code was plagiarized. Penalties for plagiarism are applied to both the parties.

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. Once again, you can either use the **`sftp`** command to transfer files from the remote node to local or use a **private git repository**.

F. 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.

2.2. Exercise 2: Circular Linked List in C [Lab Demo Exercise] (1% demo + 1% submission OR 2% submission)

Exercise 2 requires you to implement in C some functionalities for a circular linked list. Circular linked lists are commonly used in the Linux Kernel, such as for process and memory management. This exercise allows you to become more familiar with C syntax and appreciate the challenge behind implementing different parts of the operating system.

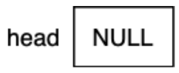
For this exercise, we represent a node in our circular linked list as follows (in `node.h`):

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

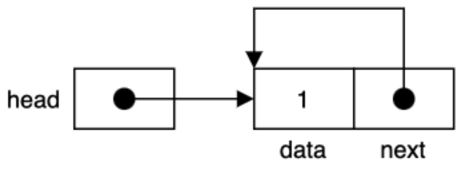
The list representation looks like this:

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

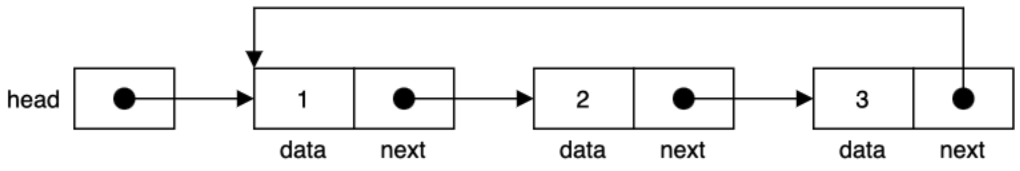
Initially, the list will be empty.



Upon inserting the first node with data equals 1, we will have a circular linked list where the first node's next pointer points back to itself.



Thereafter, every insertion will expand this circular linked list:



You need to implement five functions shown below to work with the circular 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`**.

`void rotate_list(list *lst, int offset)`
This function rotates the entire list by **`offset`** number of nodes. Using the example diagram before, a rotation of offset 1 will result in a list with head pointer to the second node, i.e., the node with data 2 right after the original head node. Assume that **`offset`** is non-negative. You should not be reallocating or modifying the data of the nodes for this rotation.

`void reverse_list(list *lst)`
This function reverses the order of the nodes in **`lst`**. The head pointer of **`lst`** should also now point to the original "tail" node, i.e., the node that had a next pointer to the original head. 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`**.

The folder structure of **ex2** is as follows:

```
/ex2
node.h (not to be modified)
ex2.c (not to be modified)
node.c
*.in / *.out (files used for testing)
Makefile (not to be modified)
```

You should **only** modify **`node.c`** for this exercise. Please take note that changes to any other file will be overwritten when we run the grading script.

After you have finished implementation, use the following command to compile your code:

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

This will produce the executable **`ex2`** by running `gcc` compiler with c99 standard, enabling warnings, and creating an output `ex2`. You may use **`-Werror`** option in `gcc` to make all warnings into errors. We will not penalize marks 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.

```
$ ./ex2 < sample.in | diff sample.out -
```
(Note the **-** at the end!)

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 follow:

- 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 have made use of a **pipe (|)** to pass the output from the first process (running the command **`./ex2 < sample.in`**) into the input of the second process running (running the command **`diff sample.out -`**). The **-** sign stands for the second input file being replaced with standard input (here, with the output produced by `ex2`).

Apart from **`sample.in`**, we have two more test cases to help verify your program. 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 circular linked list that works for all test cases.

Also, take note that the runner will call **`reset_list`** to delete all nodes in the list before it terminates the program. The only file that needs changes is **`node.c`**.

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(lst)
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 2          // rotate_list(lst, 2)
0
4            // reverse_list(lst)
0
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 ]
[ 1 200 100 2 ]
[ 2 100 200 1 ]
[ ]
[ 1000 ]
```

We defined specific macros for each of the functions:

```
#define PRINT_LIST 0
#define INSERT_AT 1
#define DELETE_AT 2
#define ROTATE_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.

2.3. Exercise 3: Function Pointers – 1%

This exercise extends the functionalities of the circular linked list implementation by applying several operations on its nodes. These operations are added by making use of function pointers in C.

Function Pointer Overview

You can refer to this link for more information on function pointers. Unlike normal pointer, which points 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 by that pointer. This technique is commonly used in **system call / interrupt handlers**.

In C, it is possible to define a **function pointer** to refer to a function. For example:

```
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`)".

Let's use the function pointers to define and use a group of functions that can **map** different operations to the circular linked list from exercise 2.

Exercise 3 is an extension of exercise 2. For this exercise, you must write the test runner `ex3.c` and `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 circular linked list.
  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 ROTATE_LIST 3
  #define REVERSE_LIST 4
  #define RESET_LIST 5
  #define MAP 6
  ```

INSERT, DELETE and LIST operations are similar with exercise 2 (no changes are needed). We have added **MAP** and replaced **PRINT_LIST** with **SUM_LIST**.

SUM_LIST function definition is provided in node.h. This function sums the data of all nodes in the list and prints out (at standard output) the sum.

In addition to the functionalities from exercise 2, we define a map function:

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

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

The **map** function (MAP) uses function pointers. You need to implement this function by applying the function func on each element of the circular linked list.

MAP can be used to apply five operations on the list. Indices for these operations are given below:

0	**add_one**
1	**add_two**
2	**multiply_five**
3	**square**
4	**cube**

The implementation for these functions is provided in file 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, you will need to initialize an array of function pointers with indices 0 to 4. This array has already been declared in function_pointers.h, but not initialized. 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 file, 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 runner should be easily extensible to allow for new operations for the map function **without changing the implementation of ex3.c and node.c**.

Adding or removing a new `MAP` operation should be done by modifying only the array of function pointers in `function_pointers.c` and its declaration in `function_pointers.h`.

For this exercise the file structure is as follows:

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

As explained earlier, we provide the functions used by **MAP** in **`functions.c`** and **`functions.h`**. These files should not be modified.

Use the Makefile provided to compile your code:

```
$ make
```

Command `make` uses the instructions found in the Makefile to compile your code. You can also run **`make clean`** to clean up the executable. Note that you must either call **`gcc`** or **`make`** to compile your code (there is no need to call both of them, in sequence).

Test your `ex3` as follows:

```
$ ./ex3 sample.in > res.out
$ diff res.out sample.out
```
(to compare your output with the given 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.

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

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             // sums list and prints it
6 0           // runs map on list with add_one function
0
6 3           // runs map on list with square function
0
6 4           // runs map on list with cube function
0
```

```
sample.out:
6
9
29
4889
```

When we pass **sample.in** to our program, it should output at standard output (console) the result of any **SUM_LIST** instruction found. This is why the **sample.out** file has 4 numbers (4 **SUM_LIST** instructions in our input file).

Some things to take note of for this exercise:

- input will always be valid (there is no need to do input validation on runner)
- the sum of list will be smaller than 2^{63} - 1 and larger than -2^{63} for any test cases we use (notice we use **long** data type for the function definition)

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

In this exercise, we introduce **valgrind**, a tool commonly used to identify and fix any kind of memory errors (memory leak detection / out of bound array access etc.). You might use **valgrind** for future lab assignments.

We want you to try using **valgrind** on the executable from **ex2** and **ex3**. You used **malloc** or **free** a couple of times in the code and there is a possibility of memory leaks occurring if any resource obtained dynamically is not freed.

To use valgrind, **cd** into **ex2** directory and run the below command:

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

You should see output like this:

Sample Output

```
==3368== Memcheck, a memory error detector
==3368== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==3368== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==3368== Command: ./ex2 < sample.in
==3368==
==3368==
==3368== HEAP SUMMARY:
==3368==     in use at exit: 0 bytes in 0 blocks
==3368==   total heap usage: 7 allocs, 7 frees, 8,824 bytes allocated
==3368==
==3368== All heap blocks were freed -- no leaks are possible
==3368==
==3368== For counts of detected and suppressed errors, rerun with: -v
==3368== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

As you can see, **valgrind** has done the hard work of checking for any potential memory leaks for us! Suppose our **ex3** had some memory leaks. **valgrind** detects memory problems and shows how to rerun to see additional details. When running on the sample input, you might see output as follows:

Sample Output

```
==3388== Memcheck, a memory error detector
==3388== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==3388== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==3388== Command: ./ex3 sample.in
==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)
```

This exercise is not graded but take note that 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 more about our system – 2%

In the next two exercises, we will write some shell scripts! A shell script is a list of commands designed to be run by the Linux shell. Earlier when we talked about **`cd`** and **`pwd`**, commands you can use in the terminal (shell). You can think of a shell script as running a sequence of these commands. For this lab, we will be focusing on the **`bash`** shell.

Exercise 5 requires you to write a simple shell script to learn more about your operating system and system. Use the man pages and online search to find out about **`bash`**.

A simple **`bash`** shell script that prints the current working directory is as follows:

`check_dir.sh`

```
#!/bin/bash
dir=$(pwd)
echo Current directory: $dir
```

The first line of the script is known as an *interpreter directive*, and it starts with the magic sequence **`#!`** known as a shebang. The directive is parsed by the kernel and instructs the kernel to run the script using the given program, in this case **`/bin/bash`**. This directive can also be used for other shells (e.g., **`zsh`**, **`fish`**) and for other scripting languages (e.g., Python).

Following this, we just have a variable that takes in the return value of the **`pwd`** command, and we print it out using **`echo`**.

To run the above script, use the following commands:

```
$ chmod +x ./check_dir.sh
$ ./check_dir.sh
```

The first command helps to change the file permissions to make the script executable. The second command runs the script. You should be able to see your current directory being printed onto the screen.

The output of the script you are supposed to write for this exercise should look as follows:

Sample Expected Output

```
Hostname: xcne6
Machine Hardware: Linux x86_64
Max User Processes: 1541954
User Processes: 9
User With Most Processes: root
Memory Free (%): 64.6353
Swap Free (%): 99.6441
```

The actual values differ based on the system you are using. 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 (|)`**

As always, do check the `man` pages to find out more about these commands. You will need to discover the other commands on your own to complete this exercise.

Once you have figured out the commands that you might need, you can run them in the terminal to test them and copy them into the **`bash`** script once they give the output you desire.

2.6. Exercise 6: Know your syscalls – 1%

For this exercise, we will be writing another shell script to help us list the system calls done by a 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**.

Within the **ex6** folder, you may find a sample C program **pid_checker.c** that prints its own process ID and its parent's process ID. We have also given a **bash** script skeleton **check_syscalls.sh** for you to update.

You need to complete the **bash** script to obtain a report on the system calls made by the program and the time spent in each call.

The expected output has the following format (values might differ, depending on the program that you are running):

Expected Output

```
Printing system call report
Process ID: 94510
Parent Process ID: 11815
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
  0.00    0.000000           0         1           read
  0.00    0.000000           0         2           write
  0.00    0.000000           0         2           open
  0.00    0.000000           0         2           close
  0.00    0.000000           0         3           fstat
  0.00    0.000000           0         7           mmap
  0.00    0.000000           0         4           mprotect
  0.00    0.000000           0         1           munmap
  0.00    0.000000           0         3           brk
  0.00    0.000000           0         3         3 access
  0.00    0.000000           0         1           getpid
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         1           arch_prctl
------ ----------- ----------- --------- --------- ----------------
100.00    0.000000                    31         3 total
```

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

2.7. Exercise 7: 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:

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 and/or executed.
e. The output your exercise produces using our sample input matches the expected output.

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. 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

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. The file structure should be:

`E0123456.zip` contains 4 folders, with the following content:

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

*The bolded names are folders.

Upload the zip file to the Student Submissions “Lab 1" folder on LumiNUS. Note the deadline for the submission is **4 Sep, 2pm**.

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