Carleton University
Department of Systems and Computer Engineering
SYSC 4001 Operating Systems Fall 2025

Assignment 2 – Process Scheduling, Memory Management

This assignment must be completed in **teams of two students** and submitted **in two stages**, each with its own deadline: an initial **individual** submission followed by a **final** collaborative submission. In the first stage, one student is responsible for an explanation of the algorithms used Exercise I.a)(i), the other for Exercise I.a)(ii)), and both must individually complete portions of Exercise II.. Any student who fails to submit their individual portion or scores below 50% will receive a **zero for the entire assignment**. For the final submission, teams must submit the **complete** assignment, incorporating both individual and collaborative components. The programming section must be completed using the Pair Programming technique, and the final grade will be calculated as the sum of all assignment parts.

YOU ARE REQUIRED TO FORM GROUPS ON BRIGHTSPACE BEFORE THE SUBMISSION OF THE ASSIGNMENT. To form groups, go to "Tools" > "Groups". Scroll through the list of group options available. The group would be "<Lab section> - Assignment 2". The "Lab Section" will be your respective section. The deadline for forming groups is **October 13th**. Brightspace will auto-assign your groups after this deadline. You will not be allowed to change groups after this deadline for **ANY** reason.

More information can be found here:
https://carleton.ca/brightspace/students/viewing-groups-and-using-groups-locker/

Please submit the assignment using the electronic submission on Brightspace. The submission process will be closed at the deadline. No assignments will be accepted via email.

Part I – Concepts [1.5 marks]

Answer the following questions. Justify your answers. Show all your work.

a) [0.5 mark] Consider the following set of processes. Each process has a single CPU burst and does not perform any I/O.

Process	Arrival Time (ms)	Execution Time (ms)
P1	0	12
P2	5	8
P3	8	3
P4	15	6
P5	20	5

With the help of Gantt charts, draw the execution timeline for the following scheduling algorithms:
(i) FCFS (First Come First Serve). [Student 1: submit a separate document explaining the FCFS algorithm]

(ii) Round Robin (time slice of 4 ms). [Student 2: submit a separate document explaining the RR algorithm]
(iii) Shortest Job First with preemption
(iv) Multiple queues with feedback (high-priority queue: quantum = 2; mid-priority queue: quantum = 3; low-priority queue: FIFO)

For each scheduling algorithm:

1. Draw the Gantt chart showing the execution timeline
2. Calculate the completion time for each process
3. Calculate the turnaround time for each process
4. Calculate the mean turnaround time of all the processes

b) [0.5 mark] Now assume that each process in part a) requests to do an I/O every 2 ms, and the duration of each of these I/O is 0.5 ms. Create new Gantt diagrams considering the I/O operations and repeat all the parts done in part a) using this new input trace.

c) [0.5 mark] Consider a multiprogrammed system that uses multiple partitions (of variable size) for memory management. A linked list of holes (the "free" list) is maintained by the operating system to keep track of the available memory in the system.

At the start of the exercise, the free list consists of holes with the following sizes (in KB):

Position	Hole Size	Status
1	85 KB	Free
2	340 KB	Free
3	28 KB	Free
4	195 KB	Free
5	55 KB	Free
6	160 KB	Free
7	75 KB	Free
8	280 KB	Free

The free list is ordered in the sequence given above: the first hole is 85 KB, followed by 340 KB, and so on.

When the system is in this state, the following jobs arrive; each of them has different memory requirements, and they arrive in the following order:

Job No.	Arrival Time	Memory Requirement
J1	t_1	140 KB
J2	t_2	82 KB
J3	t_3	275 KB
J4	t_4	65 KB
J5	t_5	190 KB

[with $t_1 < t_2 < t_3 < t_4 < t_5$]

1) Determine which free partition will be allocated to each job for the following algorithms:

- (i) First Fit
- (ii) Best Fit
- (iii) Worst Fit

For each algorithm, show the allocation table (which job gets which partition), the remaining free memory after all allocations, the total internal fragmentation and the total external fragmentation.

2) [0.2 marks] Based on your calculations, analyze and compare the three algorithms according to memory utilization efficiency and fragmentation. Based on your analysis, justify which algorithm would be most appropriate for a system with frequent small allocations, and a system with mixed workload sizes

Show all calculations step by step; also draw the memory state after each allocation. Calculate fragmentation metrics for comparison and provide detailed justification for your analysis

Part II – Concurrent Processes in Unix [1 mark]

1. Run two concurrent processes in C/C++ under Linux

[Student 2: explain what the *fork* system call does]

Using the *fork* system call (page 472 of the Linux book), create two independent processes, which run indefinitely. Process 1 will run forever, will initialize a counter at 0, and will increment it in each cycle of an infinite loop. Process 2 will run forever, will initialize a counter at 0, and will increment it in each cycle of an infinite loop. Use delay functions to slow the display speed.

To finish the program, use the kill command (man pages), find the PID of both processes (ps) and kill them.

2. Extend the Processes. Now, process 1 will display only multiples of 3. That is, increment the counter in an infinite loop, calculate if it's a multiple of 3, and display that number. Also, display the number of cycles in each iteration. Example:

   Cycle number: 0 – 0 is a multiple of 3
   Cycle number: 1
   Cycle number: 2
   Cycle number: 3 – 3 is a multiple of 3
   …

   Process 2 will do the same but decrementing the value of the counter.

   Use the exec system call to launch Process 2 (i.e., Process 2 should be a different program/executable).

   Use delay functions to slower the display speed.

   To finish the program execution, use the kill command (man pages), find the pid of both processes (ps) and kill them.

   [Student 1: explain what the exec function does]

3. Extend the processes above once more. Use the wait system call. Process 1 starts as in 2, and when Process 2 starts, it waits for it. Process 2 runs until it reaches a value lower than -500. When this happens, Process 1 should end too.

4. Extend the processes above once more. They should now share memory. The primary functions are listed in the book and course materials. Using shmget, shmctl, shmat, and shmdt, add two common variables shared between the two processes. The first variable contains the value of the multiple (in the example above: 3; that number can be changed and then the program should adapt and display the multiples of the chosen number). The second variable contains the value of the counter used by Process 1, which is now shared with Process 2. Process 2 starts only when the value of this variable is larger than 100.

Each of the processes should now react to the value of the shared variable and display a message identifying themselves and numbers in shared memory. Both processes finish when the value of the shared variable is larger than 500.

5. Extend the processes above once more. They should now protect concurrent access to the shared memory positions. On top of the shm instructions, you should protect the shared memory access using semaphores. Use semget, semop, semctl to protect the shared memory section.

As before, you now have a common variable shared between the two processes, and it is protected from concurrent access using semaphores. The behavior is as in 4.

Information about message passing is on Chapter 14 of the Linux reference book, and in the online materials posted.

Marks to be obtained: 0.2 marks for each correct part.

Put all the relevant files in a repository named *SYSC4001_A2_P2*. Remember to append the student numbers of both the teammates to the filename!

<u>Part III - Design and Implementation of an API Simulator: fork/exec [2.5 marks]</u>

The objective of this assignment is to build a small API simulator.

We will reuse Assignment 1, where we simulated an interrupt system. A template will be provided on GitHub and Brightspace. The use of the template is optional, and you can reuse your Assignment 1 code and start right away.
The template for assignment 2 can be found here: https://github.com/sysc4001/Assignment_2.git

The system has one CPU. The OS uses fixed partitions in memory. The simulator uses the following data structures:

i. Memory Partitions: fixed partitions

You have a simulated space of 100 Mb of user space available, divided into six fixed partitions:

1- 40 Mb
2- 25 Mb
3- 15 Mb
4- 10 Mb
5- 8 Mb
6- 2 Mb

You need to define a table (array of structs, or linked list) with the following structure (provided in the template):

Partition Number	Size	Code
Unsigned int	Unsigned int	String. Only use “free”, “init” or the program name (more details about the code later)

ii. PCB:

Using the textbook/slides, you need to define a table (array of structs, or linked list) with similar contents of those found in a PCB (only include the information needed: PID, CPU, and I/O information, remaining CPU time – needed to represent preemption –, partition number where the process is located). You should add any other information that your simulator needs.

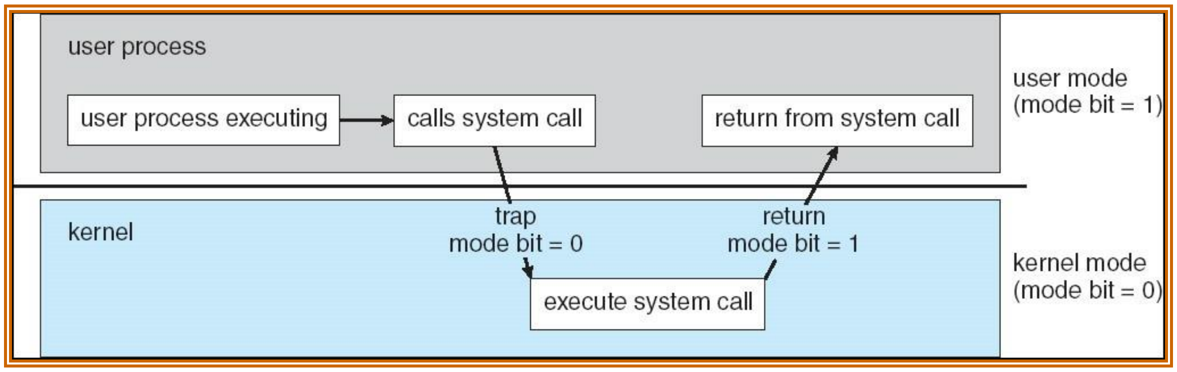
Your simulator must initialize the PCB table and pid 0, called init, which will use Partition 6; it uses 1 Mb of memory.

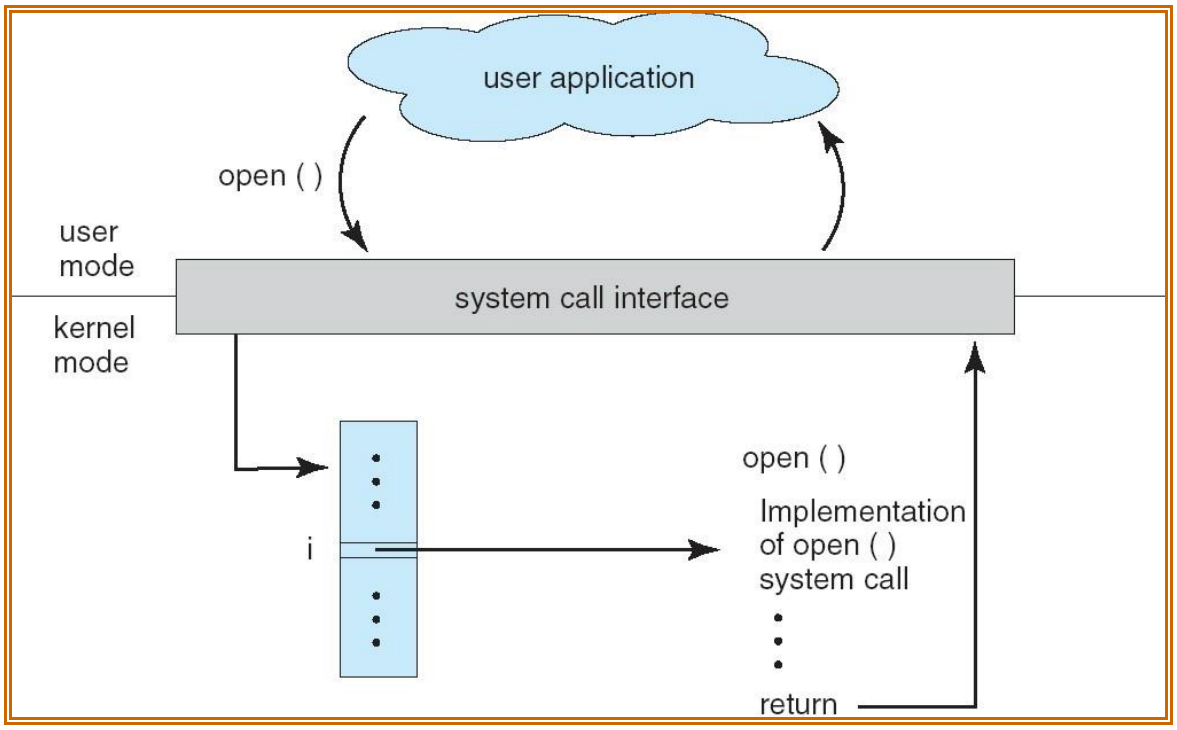
iii. List of External Files

Another table simulates your persistent memory (hard drive, USB, flash drive…). This is implemented as an array of structs, linked list, or your choice of data structure for the table. Its initial contents are loaded from an external text file (*external_files.txt*) at the start of the simulation. It The file (and table) contains a list of programs :

Program Name	Size of the program executable (equal to the size when loaded in memory)
String; 20 chars	Unsigned Integer

The objective of this simulator is to implement simulated versions of the *fork* and *exec* system calls. For all the system calls, the simulation should try to reproduce the behavior of this state diagram (from Silberschatz et al.) which was implemented in Assignment 1, for both system calls and interrupts.





In this simulator, we implement the two system calls as follows:

a. fork () [*will appear in the trace file as FORK, <duration>. Duration is how long the ISR takes*]

a. Executes the SYSCALL defined in Assignment 1
b. Then, the ISR copies the information needed from the PCB of the parent process to the child process
c. At the end of the ISR, you call the routine scheduler (), which is, for now, empty (it just displays "scheduler called").
d. Return from the ISR (as in Assignment 1)

exec (file_name) [*will appear in the trace file as EXEC <program name>, <duration>. Duration is how long the ISR takes, program name is the program you want to load*]

e. Execute the SYSCALL defined in Assignment 1
f. The ISR uses the size of the new executable; it searches the file in the file list, and obtains its memory size
g. It finds an empty partition where the executable fits
h. Simulates the execution of the loader, which reads a byte of the executable from disk (secondary memory) and writes to memory (RAM). It repeats the loading process and when loaded it calculates the remaining space in the chosen partition.
i. It marks the partition as occupied
j. It updates the PCB with the new information
k. At the end of the ISR, you call the routine scheduler (), which is, for now, empty (it just displays "scheduler called"). Return from the ISR (as in Assignment 1)
l. Run the new program

All the activities by fork and exec must be logged by the simulator. Each step is associated with a random execution time between 1 and 10 milliseconds, **except for the loader step in exec**. The duration of that step is a function of how large the program is, say 15ms for every Mb of program (for instance, a 10Mb program would take 150 ms of simulation time until it is fully loaded).

<u>Conditional statements</u>
The following statements allow you to differentiate between child and parent processes

- **IF_CHILD** [*will appear in the trace file as IF_CHILD, 0. Note that the duration is 0, this is done only to stick with the input format, and is not significant]*
    - o Every line in the trace file under 'IF_CHILD, 0' is to be executed by the child until IF_PARENT, 0' or 'ENDIF, 0' is encountered
- **IF_PARENT** [*will appear in the trace file as IF_PARENT, 0*]
    - o Every line under 'IF_PARENT, 0' is to be executed by the parent until an 'ENDIF, 0' is encountered
- **ENDIF** [*will appear in the trace file as ENDIF, 0*]
    - o Every line after 'ENDIF, 0' should be executed by both the parent and the child

The simulator checks the Ready Queue (in the PCB table you defined), grabs the first process available, and simulates its execution. How? By reading the program name from the *external_files* table and loading the appropriate program into the simulator. Other than that, you execute the fork and exec syscall as above.

Assumptions to be made:

- **The child process has a higher priority than the parent process, and no preemption**. Therefore, the child process should be executed until completion before resuming the parent process. This means that the simulator does not account for orphaned processes yet; the child must finish execution before the parent process can terminate. <u>**HINT:**</u> *As soon as you hit a FORK, **start executing the child**. When you hit an IF_PARENT, skip all the lines of the trace file till you get to ENDIF. Once you get to ENDIF, execute till the end of the trace file. This marks the end of the child process; go back to the IF_PARENT line, execute the rest as parent.* Do **NOT** worry about scheduling.
- EXEC is found at vector 3 and for FORK is found at vector 2.
- The init/trace file will not contain nested IF_PARENT...ENDIF blocks. A FORK may appear inside a subprogram reached via EXEC, but you won't see consecutive FORKs within the same (child or parent) flow. That means you don't need to track multiple open IF/ENDIF pairs.

The simulator in this assignment will have 4 input files and 2 output files.

- Input files:
  - Trace file (*trace.txt*):

    Like Assignment 1, The trace information is stored in a table as follows:

Activity	Duration **OR** device number

The first column, **Activity,** refers to the type of activity carried out by the program (CPU burst, I/O etc.), and the second column is either **Duration** (in milliseconds) or **device number** of the device that caused the interrupt; the data is comma separated.

  - List of external files (*external_files.txt*):

    As mentioned above, this is a text file which lists out the name and size of the external programs available to the simulator
  - Vector file

The same as assignment 1, you will need to include a Vector Table as follows (provided to you in *vector_table.txt*):

Interrupt(device) Number	**ISR Address**
...	
7	0x0E
...	
12	0x1B
...	
20	0x28
...	
22	0x16

  - Device delay file

Same as assignment 1, a device table must be inputted into your simulator (provided to you in *device_table.txt*):

Device number	Delay (ms)
...	
7	110
...	
12	600
...	

- Output files:
  - Execution file (*execution.txt*):

    Same as assignment 1, the *execution.txt* file should include the following information, comma separated:

Time of the event	Duration of the event	Event type

  - System status file (*system_status.txt*):

    This file must contain a snapshot of the state of the system. That is, it must display the PCB (of the currently executing process and of the processes in the wait queue), along with the time of the snapshot and trace being executed. This snapshot must be taken after every FORK and EXEC command:

    time: <current time>; current trace: <the trace line that was executed (FORK or EXEC)>

PID	Program name	Partition Number	Size	State
int, PID of the process	string, name of the process	int (between 0 and 5), partition that this program occupies	int, size of the program (from external files)	string, waiting or running.

The following test scenarios would help you understand the input and output files of the simulator better.
Mandatory Test Scenarios

1. ***Only Init is in the system. Init runs this code:***

```
FORK,  10          //fork is called by init
IF_CHILD, 0
EXEC program1, 50 //child executes program1
IF_PARENT, 0
EXEC program2, 25   //parent executes program2
ENDIF, 0     //rest of the trace doesn't really matter (why?)
```

Contents of program1:

```
CPU, 100
```

Contents of program2:

```
SYSCALL, 4
```

Output execution file (comments are for explanation only and should not be included in final output):

```
0, 1, switch to kernel mode      //encounters FORK, 10
1, 10, context saved
11, 1, find vector 2 in memory position 0x0004 //same as syscall
12, 1, load address 0X0695 into the PC
13, 10, cloning the PCB //this duration is taken from the trace
23, 0, scheduler called    //Assume 0 time for scheduling
23, 1, IRET
24, 1, switch to kernel mode      //executing child, encountered EXEC
25, 10, context saved
35, 1, find vector 3 in memory position 0x0006
36, 1, load address 0X042B into the PC
37, 50, Program is 10 Mb large
87, 150, loading program into memory  //10Mb * 15ms/Mb = 150ms
237, 3, marking partition as occupied
240, 6, updating PCB
246, 0, scheduler called
246, 1, IRET
247, 100, CPU Burst   //executing the contents of program1, child done
347, 1, switch to kernel mode     //executing parent, encountered EXEC
...
619, 1, IRET
620, 1, switch to kernel mode    //executing program2
621, 10, context saved
631, 1, find vector 4 in memory position 0x0008
632, 1, load address 0X0292 into the PC
633, 250, SYSCALL ISR
883, 1, IRET
```

Output of system status (comments are for explanation only and should not be included in final output):

```
time: 24; current trace: FORK, 10 //clones init, and runs the child
+------------------------------------------------------+
| PID |program name |partition number | size |   state |
+------------------------------------------------------+
|   1 |        init |               5 |    1 | running |
|   0 |        init |               6 |    1 | waiting |
+------------------------------------------------------+

time: 247; current trace: EXEC program1, 50 //exec is run by child
+------------------------------------------------------+
| PID |program name |partition number | size |   state |
+------------------------------------------------------+
|   1 |    program1 |               4 |   10 | running |
|   0 |        init |               6 |    1 | waiting |
+------------------------------------------------------+

time: 620; current trace: EXEC program2, 25 //exec run by parent, init no
longer exists
+------------------------------------------------------+
| PID |program name |partition number | size |   state |
+------------------------------------------------------+
|   0 |    program2 |               3 |   15 | running |
+------------------------------------------------------+
```

2. ***Only Init is in the system. Init runs this code:***

```
      FORK, 17     //fork is called
      IF_CHILD, 0
      EXEC program1, 16  //exec is called by child
      IF_PARENT,0
      ENDIF, 0
      CPU, 205 //CPU burst being called after the conditionals
```

Contents of program1:

```
      FORK, 15
      IF_CHILD, 0
      IF_PARENT, 0
      ENDIF, 0  //notice how nothing is happening in the conditionals
      EXEC program2, 33 //which process executes this? Why?
```

Contents of program2:

```
      CPU, 53
```

Output execution file (comments are for explanation only and should not be included in final output):

```
0, 1, switch to kernel mode //fork encountered, 2 processes in PCB
      1, 10, context saved
      ...
      30, 1, IRET
      31, 1, switch to kernel mode //exec encountered by child
      ...
      219, 1, IRET
      220, 1, switch to kernel mode //fork encountered (program1) by child, 3
      processes in PCB
      ...
      248, 1, IRET
      249, 1, switch to kernel mode //exec encountered by child of program1
      ...
      529, 1, IRET
      530, 53, CPU Burst //program2 executed by child
      583, 1, switch to kernel mode //exec encountered by parent of program1
      ...
      863, 1, IRET
      864, 53, CPU Burst //program2 executed by parent of program1
```

```
917, 205, CPU Burst //the rest of init executed by parent
```

Output of system status (comments are for explanation only and should not be included in final output):

```
time: 31; current trace: FORK, 17 //here you see that the fork cloned
init (and set it to running because child has priority)
+------------------------------------------------------+
| PID |program name |partition number | size |   state |
+------------------------------------------------------+
|   1 |        init |               5 |    1 | running |
|   0 |        init |               6 |    1 | waiting |
+------------------------------------------------------+

time: 220; current trace: EXEC program1, 16 //exec copied the new process
onto init. Notice how the partition was reassigned
+------------------------------------------------------+
| PID |program name |partition number | size |   state |
+------------------------------------------------------+
|   1 |    program1 |               4 |   10 | running |
|   0 |        init |               6 |    1 | waiting |
+------------------------------------------------------+

time: 249; current trace: FORK, 15 //fork called by program1
+------------------------------------------------------+
| PID |program name |partition number | size |   state |
+------------------------------------------------------+
|   2 |    program1 |               3 |   10 | running |
|   0 |        init |               6 |    1 | waiting |
|   1 |    program1 |               4 |   10 | waiting |
+------------------------------------------------------+

time: 530; current trace: EXEC program2, 33 //exec called by child
+------------------------------------------------------+
| PID |program name |partition number | size |   state |
+------------------------------------------------------+
|   2 |    program2 |               3 |   15 | running |
|   0 |        init |               6 |    1 | waiting |
|   1 |    program1 |               4 |   10 | waiting |
+------------------------------------------------------+

time: 864; current trace: EXEC program2, 33 //same exec called by parent
+------------------------------------------------------+
| PID |program name |partition number | size |   state |
+------------------------------------------------------+
|   1 |    program2 |               3 |   15 | running |
|   0 |        init |               6 |    1 | waiting |
+------------------------------------------------------+
```

3. Only Init is in the system. Init runs this code:

```
FORK, 20
IF_CHILD, 0
IF_PARENT, 0
EXEC program1, 60
ENDIF, 0
CPU, 10
```

Program 1 runs this code:

```
CPU, 50
SYSCALL, 6
CPU, 15
END_IO, 6
```

Output execution file:
```
0, 1, switch to kernel mode //fork encountered, 2 processes in PCB
...
33, 1, IRET
34, 10, CPU Burst //child executes CPU burst, child done.
...
276, 1, IRET
277, 50, CPU Burst //execution of program1
327, 1, switch to kernel mode
...
605, 1, IRET
606, 15, CPU Burst
621, 1, switch to kernel mode
...
899, 1, IRET
```

Output files will be available on GitHub for validation of the simulation results.

Your Test Scenarios

After these three tests, you should create at least 2 other tests to be submitted.

The input to your program:
- List of external files in *external_files.txt*
- Initial trace file in *trace.txt*
- Your vector table in *vector_table.txt*
- Information about the devices *device_table.txt*
- This is not an input, but you need the traces of your external programs accessible by your simulator. The simulator will look at the respective program name from the EXEC call, check memory requirements from the external_files table and fetch the corresponding program.

Test cases and an example document will be posted on Brightspace, as in Assignment 1.

The program must be written in C++ (or C, they are mostly backwards compatible).

You must run numerous simulation tests and discuss the influence of the different components of the simulator. You must submit the two tests above, but you must also create your own, analyze the results, and write a report (1-3 pages long). Submit the report in a *report.pdf* file.

For each of the simulation tests above, you should analyze the results obtained, and based on the simulation logs, explain what the system is doing. Go through the execution steps of your simulation and explain how the actual system call works and how this is done in your simulation, using the log results to discuss each of the steps in detail.

Submission guideline:

Your code for Part III must be pushed to GitHub. Name the repository *SYSC4001_A2_P3*. The repository must contain:
- Interrupts_<student#1>_<student#2>.cpp
- interrupts_<student#1>_<student#2>.hpp
- build.sh

- input_files
  - This is a folder containing your input files
- output_files
  - This is a folder containing your execution files and
- vector_table.txt
- device_table.txt
- external_files.txt
- program1.txt ... programn.txt
- report.pdf

Your code for Part II must be pushed to GitHub. Name the repository *SYSC4001_A2_P2*.

YOU MUST ALSO SUBMIT THE **REPORT.PDF** ON THE BRIGHTSPACE SUBMISSION PAGE. Include the link to both your repositories in this pdf. Also, in the description text box, include the link to both your repositories.