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

Assignment 3 – Concurrency, Shared Memory, Virtual Memory, Files

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 Exercises in Part III, as done for Assignments 1 and 2. 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 3". The "Lab Section" will be your respective to the section. The deadline to form groups is **November 14th**. 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.

Deadlines:

- **Individual submission, November 21st at midnight.**
- **Group Submission: December 1st at 8:00 am**

Part I - The objective of this part of the assignment is to build a small Scheduler simulator [2.5 marks total].

We will reuse Assignment 1 and 2, where we simulated an interrupt system, API and system calls.

The system has one CPU. The OS use 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 in six fixed partitions:

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

For Memory Management simulation, you need to define a table (array of structs, linked list) with the following structure:

Partition Number	Size	Code
Unsigned int	Unsigned int	String. Only use *free*, *init* or the program name (this will be clear as you read through the assignment)

ii. PCB:

Using the textbook/slides, you need to define a table (array of structs, 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 will initialize the PCB table and *pid 0*, called *init*, which will be assumed to be stored in Partition 6; this process is assumed to use 1 Mb of memory.

iii. Input data

The simulator will receive, as an input, a list of processes to run with their trace information, as follows:

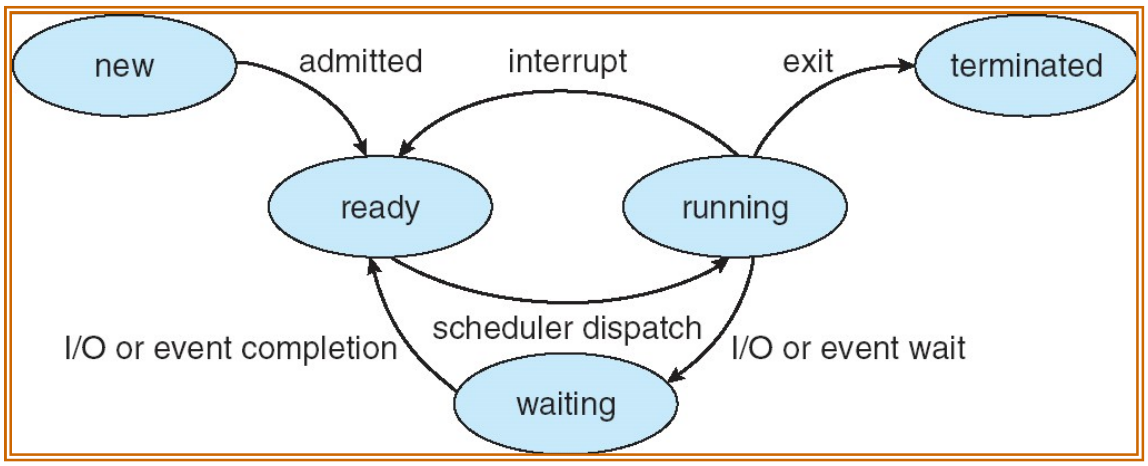
Pid	Memory size	Arrival Time	Total CPU Time	I/O Frequency	I/O Duration

- Pid: a unique identifier for the process
- Memory size: the size of the process when loaded in memory
- Arrival Time: the initial simulated time is 0, and the arrival time can be at 0 or afterwards. The time units to be used are milliseconds.
- I/O Frequency: the processes are assumed to make an input/output with this frequency
- I/O duration: this is the duration for the I/O for each of the processes (assumed to be the same for all the I/O operations)

The information will be stored in a file (to be submitted).

Using the information on the input file, you must define a data structure in memory (array of structs, linked list) similar to a PCB (only include the information needed: PID, CPU and I/O information, remaining CPU time – needed to represent preemption –).

The simulation should try to reproduce the behavior of this state diagram (from Silberschatz et al.):



You must call the simulation code in Assignment 1 every time an interrupt or system call is activated (including the time taken by the ISRs, context switch, etc. as in Assignment 1 and 2). Every time the simulation changes states, the following state information should be included in an output file.

Time of transition	Pid	Old State	New State

- Time: the simulation starts at simulated time 0, and it is measured in milliseconds.
- Pid: the id for the process that has changed state
- Old State/New State: the state of the process before and after the transition

When a process starts, it will demand the memory needed; if not available, the process cannot start, and it must wait until memory is available. Every time a new process starts, the simulator must record:

a. The total amount of memory used
b. The used/free partitions
c. The total amount of free memory available
d. The total amount of "usable" memory available (i.e., total amount of memory available in partitions; **do not add** the internal fragments, as they cannot be used).

Assume that you always have an I/O device available (that is, do not worry about waiting queues at the I/O devices: whenever you request an I/O, the I/O starts immediately).

Schedulers to implement:

1. External Priorities (no preemption). Modify your PCB as needed [0.5 mark].

2. Round Robin with a 100 ms timeout [0.5 mark].

3. A combination of both: external Priorities including preemption and a 100ms timeout, in Round-Robin fashion [0.5 mark].

Test Scenarios

Test cases and an example document will be posted on Brightspace.

iii) Simulation Execution [1 mark]

Run at least 20 different simulation scenarios for each of the different schedulers you defined. Analyze the results obtained in the simulation and write a report (2 pages minimum – can be longer if you find interesting results and you want to elaborate) discussing the results of the simulation execution.

To do that, you should first compute different metrics based on your simulation results: Throughput, Average Wait Time, Average Turnaround time, Average Response Time (i.e., the time between 2 I/O operations). Use the metrics to compare how the algorithms perform with mostly I/O bound, mostly CPU-bound processes, or processes with similar I/O and CPU bursts. The metrics can be calculated externally (python script, C program) or be a part of your simulator (but this will increase the simulator complexity).

You must run the different simulation scenarios, collect simulation results, and analyze the metrics above for the different simulation scenarios. Discuss the results you obtained and compare the algorithms based on your simulations. You should analyze the metrics and the simulation scenarios and discuss the results obtained, thinking why each of the scheduling algorithms favor the different kinds of processes.

The program must be written in C. You must submit an executable, the source code, all files needed to compile, test scenarios, and scripts to run them. Include, at least, two scripts that will be used to run 2 different tests automatically.

[1 bonus mark] Record the use of memory in your simulator and analyze the results obtained. Discuss the results in a short report.

Part 2 – Concurrent Processes in Unix [1 mark]

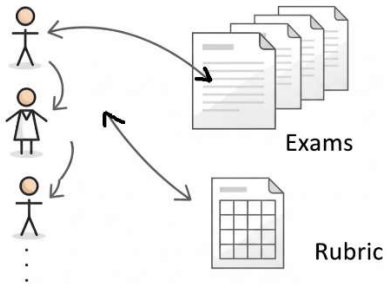
Design and implementation of a Concurrency problem:

Objectives: this part of the assignment is devoted to understanding concurrency issues using Unix/Linux shared memory, semaphores, and processes.

Problem description:

There is a variable number of Teaching Assistants (TA) assigned to mark exams. When they meet to mark, they use a pile of exams written on paper. The system you will develop in this part of the assignment, which will grant access to different files needed to mark the exam.

There are two sets of documents, as in the figure below: (a) the pile of exams, and (b) an individual file with the exam rubric. All the TAs can read the rubric concurrently, but if a TA detects a problem in the rubric, and it needs to be changed, **only one TA** can modify the rubric at a given time. Exams can be read or written by any TA at any time.



The TAs have two tasks at hand: they first pick an exam. Then, they check the rubric, and mark only one exercise in the exam. Then, the TA writes the mark for that student in the exam file. If in the process of marking an error is detected in the rubric, the TA modifies the rubric. All the TAs can be marking their exams concurrently.

Part 2.a) [0.5 marks]

Write a solution to the problem written in C/C++ running under Linux.

Your solution must have n >=2 processes running concurrently (one process per TA; the number of processes/TAs should be an argument passed to your executable program at runtime; you can use argv/argc or a file; this value should **not** be hardcoded).

Each of the processes should inform what they are doing on screen (marking, accessing the rubric, which exam is being access). At this point do not worry about critical sections. **That is, the executing program could run into race conditions.** This first part only focuses on building the processes and the shared data structures needed to communicate.

You should first create the list of exams; use any text editor to create at least 20 text files; each file should include one line with four digits, representing the student numbers (i.e., 0001-9999; we will use the file containing student 9999 to finish the processes). You can add any extra contents you wish to each file, but we will focus on processing the student numbers.

You should also create a rubric text file. The rubric file contains five lines, each of them representing the rubric of an exercise, as follows:

1,A
2,B
3,C
4,D
5,E

In this case, it this means that the rubric for Exercise 1 in the exam has the text "A" as a rubric; Exercise 2 uses "B", etc. As above, you can add more information to each line, but we will focus on using the first number (representing the exercise number) and the first character after the comma (which representing the text that describes the rubric for that exercise).

When each process representing a TA accesses the files, it first opens the file, then it reads the exam file. Then, it identifies the student number, saves it in a local variable, after which it opens and reads the rubric file. The TA process iterates through the five lines of the rubric file. The process waits approximately 1 second (use a combination of a random number generator and a delay). It then decides if the rubric must be corrected (at random too). If, the random selection is that we need to change the rubric, the first character after the comma is then replaced with the next ASCII code (i.e., if the value of the first character is 'C', then it should be replaced 'D', which is 'C'+1).

After this step, the process waits approximately 3 seconds (again, use a random number generator and a delay function). It then displays the student number that was marked, and the exercise that was marked.

Each TA works concurrently with the others.

When a file with a student number value 9999 is reached, the whole execution finishes.

HINT: work incrementally. Start with a program with 2 TAs and. You can also use extra processes for "other activities" (i.e., starting the process or ending it).

Part 2.b) [0.3 marks]

Now you will synchronize the execution of all the processes using semaphores. The objective is to convert Part A) into a Semaphore-based solution with shared memory. Use semaphores for process coordination. Other Linux system calls may be used for process creation, termination, creation and deletion of shared memory etc.

Your solution must have *n* processes (one per TA) running concurrently, and shared memory. **Do not worry about deadlock/livelock**. Each of the processes should inform what they are doing on screen and work as in Part 2.a.

Part 2.d) [0.2 marks]

Run the programs above. If (all or some of) the processes seem to be in deadlock or livelock, open the files and check the current status. Report the conditions that lead to the deadlock or livelock. If there is no deadlock on any of your runs, briefly discuss the execution order for the processes. Write your answers in a file named "reportPartD.txt"

Hand In

You must create a github repository SYSC4001_A3P2 to submit the following:

- Your programs with your student numbers appended at the end.

- A README file describing how to compile and run your program with the various test cases.

- A discussion of your design in the context of the three requirements associated with the solution to the critical section problem.

Marking Scheme for Part I and II programming exercises:

Correctness (including error checking and final cleaning up): 65%
Documentation and output: 25% (You need to print **CLEARLY**, especially before and after each reading or writing to the shared data area)
Program structure: 5%
Style and readability: 5%

Part 3 – Concepts [1.5 marks]

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

1. **[0.6 marks]** Consider the following page reference string in an Operating System with a Demand Paging memory management strategy:

415, 305, 502, 417, 305, 415, 502, 518, 417, 305, 415, 502, 520, 518, 417, 305, 502, 415, 520, 518

(i) **[0.3 marks]** Assume we have 3 Frames Allocated. How many page faults will occur with 3 frames allocated to the program using the following page replacement algorithms?

(a) FIFO (First-In-First-Out)
(b) LRU (Least Recently Used) [Student 1: explain the LRU algorithm]
(c) Optimal

Show your work for each of the algorithms. Calculate the hit ratio for each of the algorithms.

(ii) **[0.1 marks]** Assume that the case above is repeated with 4 Frames Allocated. Repeat all three algorithms from Part (i) with 4 frames allocated to the program. Show your work for each of the algorithms. Calculate the hit ratio for each of the algorithms.

(iii) **[0.2 marks]** Based on your results, answer the following questions: Which algorithm performs best with 3 frames and why? Which algorithm performs best with 4 frames and why? How do the results change when more frames are allocated? What is the relationship? Why is the Optimal algorithm impractical in real-world operating systems? Compare the performance of FIFO and LRU. When might FIFO be better or worse than LRU?

(practice exercise for the final: repeat with LFU) [Student 2: explain the LFU algorithm]

2**. [0.3 marks]** Consider a system with memory mapping done on a page basis. Assume that the necessary page table is always in main memory. A single main memory access takes 120 nanoseconds (ns).

(a) **[0.1 marks]** How long does a paged memory reference take in this system without a TLB? Explain your answer.

(b) **[0.1 marks]** If we add a Translation Lookaside Buffer (TLB) that imposes an overhead of 20 ns on a hit or a miss. If we assume a TLB hit ratio of 95%, what is the Effective Memory Access Time? Explain your answer.

(c) **[0.1 marks]** Why does adding an extra layer, the TLB, generally improve performance? Are there situations where the performance may be worse with a TLB than without one? Explain all cases.

3**. [0.3 marks]** Consider a system with a paged logical address space composed of 128 pages of 4 Kbytes each, mapped into a 512 Kbytes physical memory space. Answer the following questions and justify your answers.

(a) [0.1 marks] What is the format and size (in bits) of the processor's logical address?

(b) [0.1 marks] What is the required length (number of entries) and width (size of each entry in bits, disregarding control bits) of the page table?

(c) [0.1 marks] What is the effect on the page table width if now the physical memory space is reduced by half (from 512 Kbytes to 256 Kbytes)? Assume that the number of page entries and page size remain the same.

[Student 1: explain what is the physical memory space]
[Student 2: explain what is the logical memory space]

4. **[0.1 marks]** Explain, in detail, the sequence of operations and file system data structure accesses that occur when a process executes the **lseek(fd, offset, SEEK_END)** system call. Consider a system using a hierarchical directory structure and assume the file described by the file descriptor (fd) is not currently open by any other process.

5. File System Organization

a) **[0.1 marks]** (from Silberschatz) Consider a file system that uses inodes to represent files. Disk blocks are 8Kb in size, and a pointer to a disk block requires 4 bytes. This file system has 12 direct disk blocks, as well as single, double, and triple indirect disk blocks. What is the maximum size of a file that can be stored in this file system?

b) **[0.1 marks]** Explain what you can do in case (a) if you need to store a file that is larger than the maximum size computed. Give an example showing how you can define a larger file, and what the size of that file would be.