Part 2: Concurrent Processes in Unix

SYSC4001

Group member 1: Seham Khalifa (101295726)
Group member 2: Pardis Ehsani (101300400)

Submitted on: Dec 1st, 2025

INTRODUCTION:

In Part 2 of the assignment, we designed and implemented a concurrency problem in Unix using multiple TA processes, shared memory, and semaphores. We created 20 exam files and one rubric file, then loaded the rubric and the current exam into shared memory so that all TA processes could access them. In Part 2.a, we focused on building the processes and shared-memory structure: each TA picked an exam, checked and possibly corrected the rubric, and then marked questions with random delays. When an exam was fully marked, a TA loaded the next exam into shared memory, repeating this until the special student number 9999 was reached, causing the program to terminate. In Part 2.b I added semaphores (sem_rubric and sem_exam) so only one TA at a time can modify the rubric or update the exam, preventing concurrent writes and ensuring proper synchronization. Finally, in Part 2.c we ran the program, and analyzed whether any deadlock or livelock occurred.

<u>Order of whats occurring when in Execution: (This is just a sample please run the code using what's provided in the README to be able to see all the steps in execution)</u>

1- TA1 accesses the rubric and makes several changes
2- TA1 then accesses the loaded exam and starts marking the exam
3- TA2 then is accessing the rubric and making any changes needed
4- TA1 then accesses the loaded exam and starts marking the exam
5- Both TAs repeatedly alternate between reviewing the rubric and marking exams, repeat until student number 9999 is reached

```
seham@Seham-Yoga:/mnt/c/Users/seham/SYSC4001_A3
_P2$ ./src/marking_program 2
TA 1 is reviewing rubric
TA 1 changed rubric Q1: A -> B
TA 1 changed rubric Q3: C -> D
TA 1 changed rubric Q5: E -> F
TA 1 is marking exam 9980
TA 2 is reviewing rubric
TA 1 marked Q1 of exam 9980
TA 1 marked Q2 of exam 9980
TA 2 changed rubric Q4: D -> E
TA 1 marked Q3 of exam 9980
TA 1 marked Q4 of exam 9980
TA 1 marked Q5 of exam 9980
TA 2 is marking exam 9980
TA 1 is reviewing rubric
TA 1 changed rubric Q1: B -> C
TA 1 changed rubric Q3: D -> E
TA 1 changed rubric Q4: E -> F
TA 1 changed rubric Q5: F -> G
TA 1 is marking exam 9981
TA 2 is reviewing rubric
TA 2 changed rubric Q1: C -> D
TA 1 marked Q1 of exam 9981
TA 2 changed rubric Q3: E -> F
TA 1 marked Q2 of exam 9981
TA 2 changed rubric Q4: F -> G
TA 1 marked Q3 of exam 9981
TA 1 marked Q4 of exam 9981
```

```
TA 1 marked Q2 of exam 9997
TA 1 marked Q3 of exam 9997
TA 1 marked Q4 of exam 9997
TA 1 marked Q5 of exam 9997
TA 2 is marking exam 9997
TA 1 is reviewing rubric
TA 1 changed rubric Q2: Y -> Z
TA 1 changed rubric Q5: S -> T
TA 1 is marking exam 9998
TA 2 is reviewing rubric
TA 1 marked Q1 of exam 9998
TA 1 marked Q2 of exam 9998
TA 2 changed rubric Q5: T -> U
TA 1 marked Q3 of exam 9998
TA 1 marked Q4 of exam 9998
TA 1 marked Q5 of exam 9998
TA 2 is marking exam 9998
TA 1 finished all exams.
TA 2 finished all exams.
All exams completed.
seham@Seham-Yoga:/mnt/c/Users/seham/SYSC4001_A3_P2$
```

The new file for the rubric after execution:

1, Q
2, Z
3, X
4, Z
5, U

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

Mutual exclusion: Mutual Exclusion is satisfied because shared resources (rubric and exam) are protected by their own semaphore (/sem_rubric and /sem_exam). A TA must call sem_wait() before entering, which blocks all other TAs until sem_post() is called. This directly follows the idea that if one process is in its critical section, then no other processes can be executing in their critical sections. The output also shows that no two TAs ever modify the same shared resource(rubric or exam) at the same time**.**

Progress: Progress is satisfied because the semaphore guarantees that if no TA is currently inside the critical section, any TA that is waiting will enter immediately. As soon as one TA finishes and calls sem_post(), the next TA blocked on sem_wait() is allowed to enter without delay (semaphore ensures the next waiting TA is admitted as soon as the critical section is free).

Bounded waiting: Bounded waiting is fully satisfied in my design because when TA 1 enters the rubric section, TA 2 is correctly blocked by the semaphore and then immediately allowed to proceed once TA 1 calls sem_post(). This behaviour matches the definition of bounded waiting, which requires a limit on how many times other processes may enter their critical sections after a process has requested access and prevents starvation (ensures that every TA will eventually gain access).

Part 2.c) No deadlock or livelock occurred in any of the program runs. The semaphores allowed the TAs to take turns smoothly when accessing shared data, while still allowing concurrent marking with no conflicts.