PART II - CONCURRENT PROCESSES IN UNIX

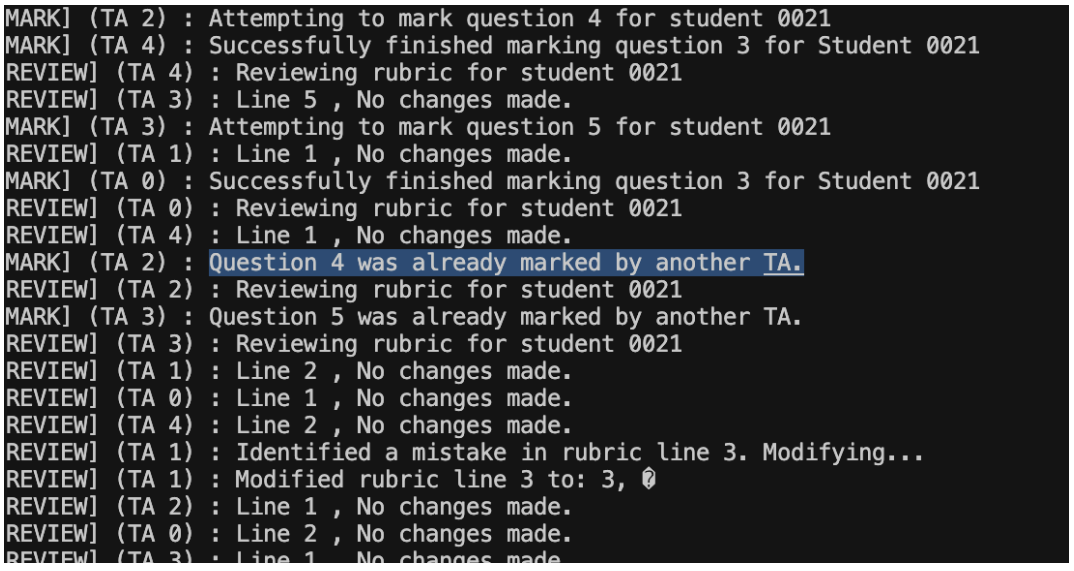
Repository Link: https://github.com/akshavib/SYSC4001_A3_P2.git

2.c) Run the programs above. If (all or some of) the processes are in deadlock or livelock, open the files and check the 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.

When running part2a_marking and part2b_marking, both systems were able to execute from the first exam (exam01.txt) all the way to the last exam (exam20.txt which had 9999 as a student number). There was no deadlock or livelock that was observed. The TA processes exited properly at the end of the execution and the parent process did properly clean up the shared memory at the very end. This is proven at the end, where when a TA finds student 9999, it exits the marking process, and all the other TA's exit the marking process as well.

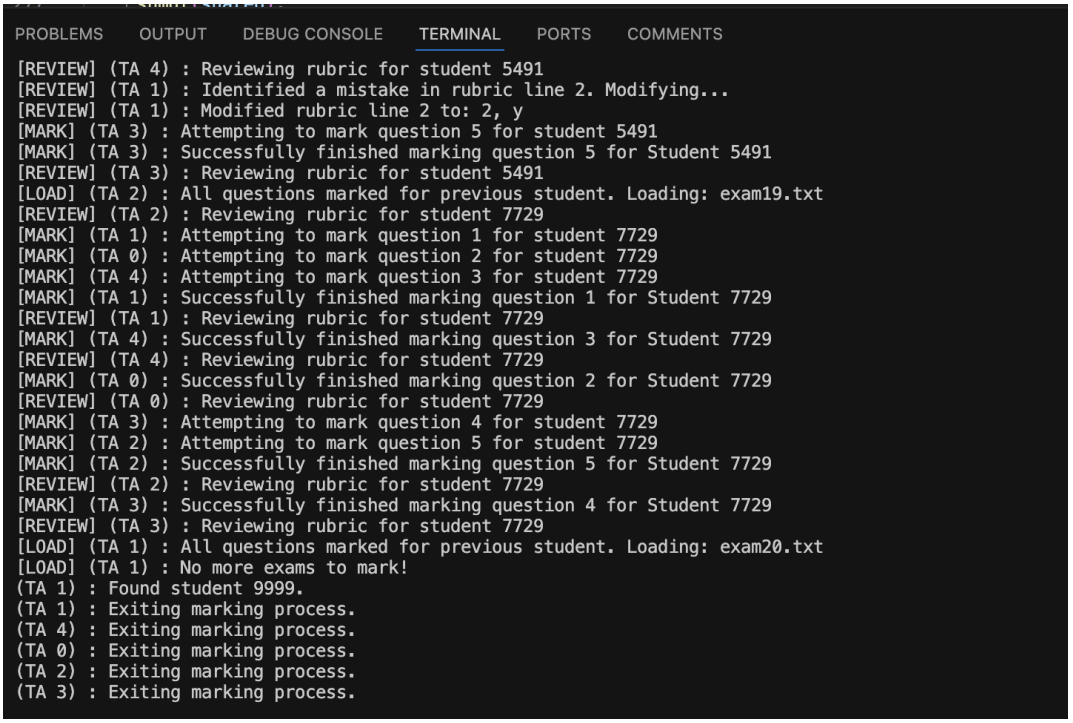
The TA's do not hold the semaphore while waiting for other resources for themselves. They perform an action, and then release it right away. This makes sure that there is no deadlock. There is no livelock because every time the TA got the lock, they made tangible progress and there was no case where there would have been an infinite loop.

In the part2a_marking process, to put it quite simply, there was no execution order. The TA's had overlapping execution, where TAs ended up marking the exact same questions at the same time. This was seen when the TA would print a statement about marking a question, and then the next TA would mark the exact same question. Even more, this process had race conditions. This is seen when a TA prints "Question 3 was already marked by another TA". The order depended entirely on the OS scheduler and the TA delays, and there was no set organization.



Example of race condition in part2a_marking

In the part2b_marking process, the execution order was sequential and serialized. The semaphore acted as a bottleneck mutex). When TAs were delaying, only one TA was able to claim a question and modify the rubric at a time. This ensured that there was no overlapped marking and no race conditions. The TAs maintained marking in the order, and they exited in this same order as well.



End of part2b_marking process, all TA's exit when expected

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

The 3 requirements for a correct critical section solution includes mutual exclusion, progress, and bounded waiting.

For mutual exclusion, if a process is executing in its critical section, no other processes can be executing in their critical sections. This applies to my part 2b code. Semaphores are placed when marking a question, and the logs show how there is a strict interleaving of marking events. For example, TA 2 claims Question 1, finishes it, and only then TA 4 claims Question 2.

For progress, if no process is executing in its critical section and some processes wish to enter, then only those processes that are not in their remainder sections can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely. Part 2b satisfies this requirement because there was no wait in the process where there was no TA requesting access to the shared memory. The logs show continuous activity, where TAs cycled through reviewing, marking, then loading the next exam without stalling (except for this thinking delay time of course).

Lastly, bounded waiting is when there is a milit on the number of times other processes are allowed to enter their critical sections after a process has made a request to enter it. With background knowledge of the semaphores in Linux, it uses a FIFO queue for processes that are on their wait lock. In the part2b process, there was no TA that was starved, and there was a good distribution of work amongst the TAs.

In conclusion, all 3 requirements of the critical section solution were met.