For this segment, we studied the outcome of the programs we devised for Section 2.a and Section 2.b in an effort to know the deadlock and livelock behaviors in the execution of such programs. Section 2.a, in this case, has no synchronization since we allowed the teaching assistants (TAs) to run their processes in complete isolation. Each TA concurrently loads the exams, reviews them, edits the rubric, and chooses the questions to be marked. Given the absence of mutual exclusion, no TA will wait for other TAs to release their processes to continue, ensuring that no deadlock situation occurs. Furthermore, there are no problems of livelock for Part 2.a, since the processes are able to go on about their activities, but due to the race conditions that allow them to overwrite and disrupt the processes of one another. While the TAs may end up marking the same question or editing the rubric, there is no chance that any will become stuck in repeating a process and waiting indefinitely.

This is where Part 2.b gives a livelock and deadlock solution in the form of real synchronization introduced by means of three semaphores, one for rubric editing, one for question selection, and the other for exam loading. The mutual exclusion of all critical regions is retained but these semaphores are used one at a time. In all test cases executed, no TA had control of more than one semaphore at a time, and the order in which they applied the locks did not change.
A process only ever acquires and later releases a single semaphore, which means the conditions for deadlock (in particular “hold and wait” and “circular wait”) will never occur and therefore deadlock cannot occur in this design. The same is true for livelock: every semaphore is eventually released and no process is left behind. Not every TA is forced to retry the operation as though it is an endless cycling process while others are executing. Each TA, in order, loads an exam, checks or updates the rubric, marks a set of questions, and then terminates once the exam (student 9999) is the one to terminate on.

The order in which the semaphores are to be executed is also consistent and correct. Each TA loads an exam one at a time, and then rubric corrections after in a single order, and TA's question selection and execution is done in a single order a any given time. Each part has, in general, completed a consistent number of execution runs without any deadlocks in the process. Part 2.a has some, albeit race conditions, and has completed execution successfully, while Part 2.b has executed semaphores execution to coordinate TA's processing safely and correctly, and completed execution successfully.

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

In designing this section 2b, we were using three semaphores (one for rubric updates, one for selecting the next question, and one for loading exams) and this covers the three requirements of the critical-section problem (mutual exclusion, progress, and bounded waiting) fully. Concerning mutual exclusion, any access to shared memory due to conflicts, for example, in altering the rubric, claiming a question to mark, or loading the next exam, is guarded by a P()/V() pair preventing two TAs from entering the same critical section at the same time. Regarding progress, there is a critical section of the process in which the TA is waiting, at which point they can enter the section, and there is a critical section which can finish freeing the section from outside access, thus preventing the situation of being "held up". About the bounded waiting, there is a fair queuing of the processes at the semaphores, so the TAs can be sure that they will be able to access the section, and they will not be "waiting" for access forever. In general, the semaphores are allowing a safe, responsive, and equitable environment for the TAs to work in parallel.