Part 2

c.

After testing both Part 2a and Part 2b multiple times using varying numbers of TA processes, the program always reached the student 9999, and all TA's terminated correctly. In every run, I saw no signs of deadlock or livelock. All processes continued to make progress until the last exam was processed.

In Part 2a, the behaviour is intentionally race condition heavy. None of the shared memory updates are protected, so several TAs can enter the same critical section at the same time. This is why the output looks more chaotic than 2b; multiple TAs correcting the same rubric entry simultaneously, more than one TA marking the same question, and multiple TAs trying to load the next exam at the same moment. Though the results are inconsistent, the key point is that nothing ever blocks. Every operation happens immediately, so each TA process always has a chance to run. Since there are no semaphores or locks at all, none of the deadlock conditions can be met, there is no hold and wait and no circular wait. Because nothing forces one TA to wait for another, deadlock simply cannot occur in Part 2a.

In Part 2b, I added the semaphores around the three actual critical regions; updating the rubric, selecting a question to mark, and loading the next exam. Each semaphore protects exactly one shared resource, and each TA only acquires one semaphore at a time. This lessens the probability of circular wait because TAs don't hold one semaphore while trying to acquire another. It also prevents hold and wait, since the TA finishes the entire critical section before releasing the semaphore and moving on. This setup gives proper mutual exclusion and bounded waiting TAs may briefly wait for a semaphore, but they always eventually get it, and none of them gets stuck indefinitely. Although the interleaving of TA actions is still non-deterministic, the behaviour is now consistent: no duplicated markings, no corrupted rubric letters, and only one TA updates next_exam_index at a time.

In conclusion, both versions eventually terminate, but for different reasons. Part 2a finishes because it never blocks on anything, so deadlock is impossible even though race conditions occur constantly. Part 2b finishes because the semaphores ensure synchronization while avoiding any of the conditions required for deadlock. In both cases, the system maintains progress, but Part 2b does so with well defined concurrency control instead of relying on uncontrolled interleaving.