

# **Design Discussion**

## **A. Mutual Exclusion**

In this design, the critical sections involve two specific operations:

1. Reading and incrementing the rubric\_answers character array, when a TA decided to modify the rubric.
2. Reading and updating the questions\_marked boolean array and decrementing questions\_remaining, when a TA decided what question to mark next.

In part A which doesn't involve synchronized processes, mutual exclusion was violated. A delay was added (usleep(100)) between reading the state (if !marked) and writing the (marked = true). The output logs with 10 TAs confirm that the same question was indexed simultaneously, proving that multiple processes were executing at the same time. Where the correct execution order would have been for one TA to access the question, and mark it as true, so that the next TA would mark an unmarked question.

In part B implements synchronization with Semaphores initialized to 1. The code SemaphoreWait(exam\_sem, 0) is used before a TA decides what question to mark, where it decrements the semaphore, allowing for a TA to either enter its critical section or wait if it has already been decremented. The relevant exam data is copied to local variables before the lock is released, which prevents a race condition where one TA might load a new exam into shared memory while another TA is still reading the previous exam's data. This way it is not possible for 2 TAs to modify access and modify questions\_marked simultaneously, therefore mutual exclusion is satisfied.

## **B. Progress**

In this design, the remainder section refers to the TA spending time doing work. Such as when it takes 0.5-1.0 seconds to check the rubric, and when it takes 1.0-2.0 seconds to mark the exam.

In part B, SemaphoreSignal was called after the TA selects a question, and before sleep(1) to simulate working. This means that when a TA which is marking a question does not block other TAs from entering their critical sections, therefore the decision to enter the critical section is never blocked by a process doing unrelated work and the design satisfies the Progress condition.

## **C. Bounded Waiting**

This design ensures that a TA cannot continuously reacquire the lock and mark all the questions while another TA waits. When SemaphoreWait blocks a process it is placed in a wait queue which is typically implemented in a FIFO queue. This means that when another TA process unlocks the critical section, the first process which was blocked is placed at the head of the queue and gets to mark the next question in order. Generally, if there are n TAs, a TA will enter a critical section after at most n - 1 other TAs have executed in the critical section. It is not possible for the TA to be excluded indefinitely and therefore the design satisfies the bounded waiting condition.

## **Deadlock and Livelock Analysis**

### **Deadlock Analysis:**

The textbook states that deadlock can occur when four conditions occur simultaneously: mutual exclusions, hold and wait, no preemption, and circular wait. The design used in part B does not allow for deadlock to occur.

Mutual exclusion is applied as stated previously, however the hold and wait and circular wait conditions are prevented. The TAs never hold one lock while waiting for another, a process acquires the rubrinc\_sem, modifies the rubric, and releases it right after. Only then does it attempt to acquire the exam\_sem. Because the locks are acquired one after another, a circular dependency cannot form.

The code further prevents deadlock during the exit mechanism. If a TA discovers that all of the exams have been marked, it releases the lock before breaking out of the while loop, which ensures that the semaphore won't be locked forever.

### **Livelock Analysis:**

The processes cannot be in livelock since the Semaphore implementation blocks incoming processes and places them in a wait queue, while waiting for another process to unlock the critical section. This means that the processes cannot continuously change their states by retrying, ensuring no live lock can occur.

## **Execution Order**

When running part B with 10 TAs, the execution order of the TAs is non-deterministic. As an example, after an exam is loaded, TA 8 and TA 10 correct the rubric before TA 9 marks a question, showing that the processes don't execute in the order they were created. This is because the scheduler determines which processes wake up first when a semaphore is unlocked. This randomness confirms that the concurrency requirement is met.