Dennis Chen student#101236818
Mithushan Ravichandramohan student#101262467

SYSC4001A Assignment 3 Part 2

2c)

For both programs in 2a) and 2b), there is no deadlock or livelock so instead the execution order will be discussed.

Part 2a

There is no blocking wait on other processes and no semaphores, locks, or busy-waiting loops.

Each TA only does this:

1. Sleeps (timed)
2. Reads/writes shared memory
3. Loads exams from files
4. Loops until the exam file 9999 appears, then exits.

Because no process ever waits for another process (they only wait for time delays or I/O), there is no deadlock since there is no circular wait on resources and no livelock since TAs don't keep reacting to each other without progress.

As long as there is an exam with student 9999 in the sequence, the execution order only involves:

1. The OS scheduler arbitrarily interleaving TAs.
2. They race on rubric updates / question states / exam loading, but eventually all questions of each exam get marked, the exam index advances, exam file 9999 is loaded, terminate is set, and finally all TAs see it and exit.

Thus there is no deadlock/livelock since the program works by arbitrary interleaving with eventual termination.

Part 2b

There are 3 separate binary semaphores and the program never holds more than one at a time:

SEM_RUBRIC for the whole rubric pass,
SEM_QUESTIONS for picking/updating question states,
SEM_EXAMLOAD for loading the next exam or setting terminate.

Deadlock requires: mutual exclusion, hold-and-wait, no preemption, and circular wait. It has mutual exclusion via semaphores. It does not have hold-and-wait because a TA never holds one semaphore and then tries to acquire another; they always acquire at most one at a time, then releases it. Circular wait cannot form, because there is no chain. Ex. TA1 holds A, waits for B; TA2 holds B, waits for A, etc. Thus structurally there is no deadlock since there's no way to build a cycle of processes each holding one semaphore and waiting on another.

Dennis Chen student#101236818
Mithushan Ravichandramohan student#101262467

Livelock would require processes constantly changing state but making no real progress (e.g repeatedly acquiring and releasing semaphores without advancing marking/exams). In this program, each successful entry to a critical section causes real progress:

Rubric Critical Section: rubric is processed and possibly updated once per TA per exam.
Question Critical Section: some question moves from NOT_MARKED → IN_PROGRESS → DONE.
Exam-load Critical Section: the exam index advances or terminate is set.

There's no pattern of retreating and retrying forever. Every pass through the loop either marks something, loads a new exam, or notices terminate and exits. Thus livelock cannot occur.

Execution order with semaphores
The OS scheduler still decides which TA runs next, so interleaving is arbitrary. Semaphores only constrain the order of critical sections.

Rubric: only one TA can be in the rubric phase at a time; others wait.
Questions: only one TA at a time can claim or finish a question.
Exam loading: only one TA can advance to the next exam or set terminate.

Overall execution order:

1. TAs take turns doing rubric passes (SEM_RUBRIC).
2. They take turns claiming and finishing questions (SEM_QUESTIONS).
3. When an exam is done, exactly one TA at a time loads the next exam (SEM_EXAMLOAD).

When the final exam 9999 is reached, terminate is set; all TAs eventually see it and exit.

So for Part 2b there is no deadlock or livelock and execution is arbitrary but safely serialized around the critical sections by semaphores.

Discussion of design
Part 2a
On startup, the rubric and the first exam file are loaded into shared memory. All TA processes use only shared memory (not the files directly). Race conditions are expected.

Each TA reads the rubric. For all 5 rubric entries, it does the following:

- Sleeps 0.5–1.0 s
- Prints before/after each shared-memory read
- Randomly decides whether to correct the rubric
- If correcting, increments the rubric's character (A -> B, C -> D, etc.)
- Prints before/after each shared-memory write
- Saves the updated rubric back to rubric.txt

Dennis Chen student#101236818
Mithushan Ravichandramohan student#101262467

Then the TA marks 5 questions in each exam. TAs select an unmarked question from shared memory. The program prints before/after each read/write. Marking takes 1.0–2.0 s. In this part, multiple TAs may mark the same question.

When all questions are marked, one TA loads the next exam file into shared memory. The program prints all shared-memory writes. If the student number is 9999, sets terminate = 1 and all TAs stop.

It is expected in Part 2a that there are many simultaneous rubric reads/writes, lots of overlapping question markings, and several TAs may attempt to load the next exam. This is all due to race conditions, which are allowed in Part 2a.

Part 2b
The behavior of Part 2b is similar to Part 2a, with these differences:
Rubric access is serialized. All rubric-related work (reading, deciding, correcting, and saving to rubric.txt) is done inside a semaphore-protected critical section. There are no longer overlapping rubric corrections and simultaneous writes to the rubric file. Only one TA at a time can be in the rubric writing phase.

In 2a, two TAs could both see a question as NOT_MARKED and both set it to IN_PROGRESS due to race conditions. In 2B, picking and updating a question's state (NOT_MARKED → IN_PROGRESS → DONE) happens inside a SEM_QUESTIONS critical section. No two TAs will ever claim the same question. Question assignment is exclusive.
Exam loading is now coordinated. Previously in part 2a, multiple TAs might decide the exam is finished and race to load the next exam, potentially overwriting each other. Now, only one TA at a time can load the next exam (SEM_EXAMLOAD). The exam index, student number, and question states are updated once per exam transition, in a controlled way.

Termination becomes cleanly synchronized. Setting terminate = 1 (when 9999 is loaded or no more exams) is done inside the same critical section as exam loading.Once exam 9999 is detected, the shared terminate flag is set and all TAs see a consistent termination decision.

In 2A there are lots of data race conditions on rubric, question states, and exam loading. Logs can look chaotic and logically inconsistent. In part 2b: you still see interleaving of actions from different TAs, but critical updates happen one at a time, so the shared state evolves in a consistent way. There are no double-claimed questions, no overlapping rubric edits, and no competing exam loads.

In the context of the 3 requirements of the critical section problem
Requirement 1: Mutual exclusion
Part 2b adds binary semaphores around each shared resource:
SEM_RUBRIC: protects all rubric access/modification
SEM_QUESTIONS: protects choosing and updating question states

Dennis Chen student#101236818
Mithushan Ravichandramohan student#101262467

SEM_EXAMLOAD: protects loading the next exam

This ensures only one TA at a time can modify each shared structure. Part 2b had race conditions while part 2b enforces proper exclusion.

Requirement 2: Progress
As soon as a TA exits a critical section and executes V(sem), the kernel immediately wakes one waiting TA. There are no extra delays, no busy-waiting, and no process outside the critical section prevents progress. Thus if the critical section is free and a TA wants in, one of the waiting TAs always gets the semaphore next.

Requirement 3: Bounded Waiting
The program relies on the operating system's semaphore queueing: once a TA blocks on P(sem), it joins the semaphore's wait queue and the OS eventually wakes it after a finite number of other semaphore entries. The program doesn't implement explicit ticket ordering but standard SysV semaphore behavior prevents starvation. So each TA waits a bounded number of turns before entering its critical section.