SYSC4001

Assignment 3:

Seydi Cheikh Wade (101323727)

Sean Baldaia (101315064)

Part 2

Part C Report

Livelock Analysis and Execution Order:

Experimental setup:

For both Part 2.a (unsynchronized shared memory) and Part 2.b (semaphore-based solution), I ran the marker program with different numbers of TAs:

- Part 2.a (no semaphores, only shared memory):
  - 2 TAs → partA_2TAs.log
  - 3 TAs → partA_3TAs.log
  - 5 TAs → partA_5TAs.log
- Part 2.b (shared memory + semaphores):
  - 2 TAs → partB_2TAs.log
  - 3 TAs → partB_3TAs.log
  - 5 TAs → partB_5TAs.log

Each run processes the same sequence of exams, with student ID 9999 as the sentinel that triggers termination. The logs show every parent and TA action, including rubric checks/updates, question marking, exam completion, and the final shutdown.

Part 2.a -Unsynchronized shared-memory solution:

In Part 2.a there is **no mutual exclusion** around the shared data structures:

- The rubric array in shared memory can be read and written concurrently by all TAs.
- The per-question state (e.g., "not started / in progress / done") is also updated without locks.
- The parent periodically polls shared memory to decide when an exam appears to be finished and when to load the next one.

Observed behaviour:

Across all three runs (2, 3, and 5 TAs):

- Every exam from the first student up to the sentinel exam is eventually processed.
- All TAs reach a state where they print that all questions "appear done" for the current student.
- The parent ultimately detects the sentinel student 9999, sets the terminate flag, waits for all children, and prints a final "all done" style message.

So, **no deadlock** and **no livelock** were observed in any run of Part 2.a.

However, the logs clearly show **race conditions**:

- Multiple TAs can start working on the same student and the same question at almost the same time.
- Rubric entries are sometimes updated by different TAs in quick succession, so the "current" letter seen by each TA may differ.
- The parent may decide that an exam is done while another TA is still in the middle of reading or updating the rubric.

Despite this, the system keeps making progress because:

- No TA ever blocks while holding a resource; they simply read/write shared variables and continue.
- There is no circular wait for locks (since there are no locks at all).
- Every TA eventually "falls through" each exam and then moves on or terminates.

This gives **nondeterministic and sometimes inconsistent marking**, but it does **not** lead to deadlock or livelock in the observed runs.

Part 2.b -Semaphore-based synchronized solution:

In Part 2.b, semaphores are introduced to coordinate access to shared data:

- mutex_rubric protects the rubric array and the rubric_dirty flag.
- mutex_questions protects the per-question state and the exam_done flag.
- mutex_log serializes access to the shared log counter and printf, so the global Gxxxxx ordering is coherent.
- exam_ready is used by the parent to wake up all TAs when a new exam is loaded.

All semaphores are **process-shared** and stored inside the same shared memory segment as the rubric and question state.

Execution order (high-level):

From the logs for 2, 3, and 5 TAs, a typical exam follows this pattern:

1. **Parent loads exam**
    - The parent reads the next examNN.txt, copies the 4-digit student ID into shared memory, resets question_state[] to "not started", and clears exam_done.

  - For normal exams, terminate is left at 0; for the sentinel student (9999), terminate is set to 1.

2. **Parent wakes TAs**
   - The parent posts exam_ready once per TA process.
   - TAs that were blocked on sem_wait(&exam_ready) wake up and start working on the new student.
3. **TAs review the rubric under mutual exclusion**
   - Each TA loops over questions Q1–Q5.
   - For each question, it briefly locks mutex_rubric, reads the current letter, and unlocks.
   - After a random delay, the TA may decide to increment that letter (again under mutex_rubric) and set rubric_dirty = 1.
   - Parent periodically checks rubric_dirty, and if set, it locks mutex_rubric, writes the updated rubric to the file, clears rubric_dirty, and unlocks.
4. **TAs mark questions under mutual exclusion on question state**
   - To pick a question, a TA locks mutex_questions and scans question_state[].
   - The first entry with state 0 is claimed (set to 1 = "marking").
   - If no questions are left with state 0 and all are 2 ("done"), the TA sets exam_done = 1.
   - After unlocking, the TA marks that question using the current rubric and then sets its state to 2 ("done") under mutex_questions.
5. **Parent loads next exam when exam_done is set**
   - The parent periodically locks mutex_questions, checks exam_done, unlocks, and if the exam is done, loads the next exam and posts exam_ready num_TAs times again.
6. **Termination**
   - When the sentinel exam (student 9999) is loaded, the parent sets terminate = 1.
   - After observing terminate, TAs finish any in-progress work, log that the terminate flag is set, and exit.
   - The parent then wakes any TAs still blocked on exam_ready, waits for all children with wait(), and logs that everything is done.

Because mutex_rubric, mutex_questions, and mutex_log are always acquired separately and held only for short critical sections, and there is a consistent order of acquiring locks (no nested locking in conflicting orders), there is no circular wait.

Deadlock / livelock observation

Across all Part 2.b runs (2, 3, and 5 TAs):

- All exams, including the one with student ID 9999, are eventually processed.
- All TAs print termination messages and exit.
- The parent successfully reaches the final message after waiting for all children.

Therefore, **no deadlock** and **no livelock** occurred in the synchronized solution either.

The main difference compared to Part 2.a is that:

- Rubric updates are now serialized, so the log clearly shows a controlled sequence of rubric corrections.
- Question assignment and completion are consistent, and at most one TA marks a given question for a given exam.
- The parent's "exam done" detection and rubric saving are coordinated with the TAs via the semaphores, making the execution order easier to reason about.

Conclusion:

For both the unsynchronized (Part 2.a) and semaphore-based (Part 2.b) implementations, **no deadlock or livelock** was observed in the collected logs for 2, 3, and 5 TAs:

- Part 2.a exhibits race conditions and nondeterministic orderings, but the absence of locks also means there is no circular wait; processes never block indefinitely.
- Part 2.b uses process-shared semaphores to protect all shared structures and to wake TAs when new exams arrive. Locks are acquired in a consistent order, held briefly, and there is always progress toward finishing each exam and reaching the sentinel student.

As a result, the main effect of adding semaphores in Part 2.b is not to "fix" a deadlock, but to **eliminate race conditions**, guarantee a well-defined execution order on the shared state, and make the behaviour of the concurrent system more predictable and easier to explain.