SYSC 4001

ASSIGNMENT 3 PART 2 Report

November 30th 2025

Sammy Eyongorock-101246652-Student 1

Paul Harry Theuma-101299749-Student 2

https://github.com/Esammy-88/SYSC4001_A3_P2

1. Introduction

This report examines the design, behaviour, and correctness of a concurrent Teaching Assistant (TA) exam-marking system implemented in C++. The purpose of the project is to demonstrate the effects of race conditions in a multi-threaded program and to show how proper synchronization techniques prevent inconsistent access to shared data.

Two versions of the system were implemented:

1. **Unsynchronized version (Part 2a)** – no mutexes or locks, used to illustrate race conditions.

2. **Synchronized version (Part 2b)** – uses mutexes and a readers-writers pattern to ensure correct access to shared structures.

The system simulates multiple TAs reading a shared rubric, marking questions on student exams, and loading the next available exam.

2. Test Cases

2.1 Input Files

The program uses a set of test files stored in exam_files/. These files represent student exam submissions.

The naming and contents are:

- **exam_1.txt** through **exam_25.txt**

- then a jump directly to:

- **exam_9999.txt**

Each file contains a simple placeholder value:

0001
0002
...
0025
9999

This design forces the system to behave normally through exam 1–25 and then terminate when the last exam file (exam_9999) is reached.

Purpose of Input Structure

- **Sequential testing:** Exams 1 to 25 allow observing the marking cycle many times.
- **Boundary condition:** exam_9999 acts as a sentinel value indicating the end of the marking queue.
- **Thread behaviour monitoring:** Because multiple exams are available in sequence, the system's correctness, concurrency, and load balancing can be observed over many iterations.

3. System Overview

Each TA is represented by a thread. TAs repeat the following cycle:

1. **Read the shared rubric**
    - Multiple TAs may read simultaneously.
    - Occasionally, a TA decides the rubric is incorrect and attempts to correct it.
2. **Mark questions on the current exam**
    - Each exam has 5 questions.
    - TAs independently attempt to claim and mark available questions.
3. **Check if all questions are marked**
    - Once the exam is complete, a TA loads the next exam.
4. **Stop when exam_9999 is reached**

4. Code Structure

4.1 Core Files

- **ta_marking_basic.cpp**
  Contains the unsynchronized implementation.

- **ta_marking_semaphores.cpp**
  Contains the synchronized implementation using mutexes and readers-writers logic.

4.2 Shared Data Structures

Exam Data

Tracks which exam is being marked and which questions are completed.

Key fields:

- student_number
- questions_marked[5]
  - 0 → not started
  - 1 → in progress
  - 2 → completed

Rubric Data

A vector of five lines representing the rubric of an exercise.

1, A
2, B
3, C
4, D
5, E

TAs read these lines and occasionally modify a letter (A→B, B→C, etc.) to simulate rubric corrections.

5. Part 2a – Unsynchronized Implementation

The unsynchronized version intentionally contains **no mutexes**.
This leads to several observable race conditions.

5.1 Race Condition: Rubric Corruption

Two TAs may write to the same rubric line at the same time.

Example:

TA1: changes 'A' to 'B'
TA2: reads 'A' but writes later → changes to 'C'

Because TA2 read old data, the final rubric is incorrect.

5.2 Race Condition: Duplicate Question Marking

Two TAs can check the same question at the same time:

TA1 checks Q1 → sees 0
TA2 checks Q1 → sees 0

Both set to 1
Both mark Q1

Result:

- The question is marked twice.
- Work is duplicated.
- Output becomes interleaved.

5.3 Race Condition: Exam Loading

Multiple TAs detect completion and simultaneously load the next exam.

Example output:

TA1 loading exam 2
TA2 loading exam 2
TA3 loading exam 2

This causes:

- Incorrect exam numbering

- Premature termination
- Skipped exams

6. Part 2b – Synchronized Implementation

The synchronized program uses three major synchronization mechanisms:

1. **Readers–Writers Locks for rubric access**
2. **Per-question mutexes for marking**
3. **A dedicated mutex for loading exams**

6.1 Readers–Writers Pattern

Readers (many) and writers (one) must share the rubric safely.

Rules achieved:

- Multiple TAs may read the rubric simultaneously.
- If one TA wants to write (correct the rubric), all readers finish first.
- During writing, all other TAs must wait.

6.2 Question Marking Locks

Each question has its own mutex, preventing duplicate marking.

Only one TA can claim a question at a time, ensuring:

- No two TAs mark the same question
- The check-then-mark operation is atomic

6.3 Exam Load Lock

Only one TA at a time may load the next exam.

This ensures:

- Exactly one examiner increments the student number
- Exams are not duplicated or skipped

7. Input/Output Behaviour

7.1 Inputs

Our implementation simulates exam files using a shared memory struct.
We included these files for better explanation and context:

exam_1.txt
exam_2.txt
...
exam_25.txt
exam_9999.txt

No manual input is required.

7.2 Outputs

Outputs include TA activity logs such as:

- "TA 1: Reading rubric line 1"
- "TA 2: Corrected rubric for question 3"
- "TA 1: Marking question 4"
- "TA 3: Completed marking question 5"
- "TA 2: All questions marked, loading next exam"

Unsynchronized Output Characteristics

- Interleaving text
- Duplicate markings
- Corrupted rubric lines
- Multiple exam loads

Synchronized Output Characteristics

- Clean, ordered messages
- Correct rubric updates
- No duplicate markings
- Exams loaded exactly once
- TAs stop cleanly at exam 9999

8. Experimental Results

8.1 Behaviour of Part 2a

Race conditions occurred frequently:

Race Condition	Frequency	Effect
Rubric corruption	High	Wrong rubric values
Duplicate marking	High	Same question marked twice
Exam load conflicts	Very high	Incorrect student numbers
Output interleaving	Always	Hard to read logs

8.2 Behaviour of Part 2b

After adding synchronization:

- No rubric corruption
- No duplicated marking
- No exam loading conflicts
- All threads proceed smoothly
- Exam 9999 stops the system correctly

9. Discussion in Relation to Critical-Section Requirements

9.1 Mutual Exclusion

- Achieved using mutexes.
- Writers have exclusive access to the rubric.
- Question marking is protected per question.
- Exam loading is exclusive.

9.2 Progress

- If no one is in a critical section, the next TA immediately enters.
- Locks are held for short periods → no blocking of unrelated code.

9.3 Bounded Waiting

- C++ mutexes are fair.
- Readers-writers logic ensures no starvation.
- Per-question locks ensure every question is eventually assigned.

All three requirements are fully satisfied in Part 2b.

10. Conclusion

The experiment demonstrates the importance of proper synchronization in multi-threaded programs. The unsynchronized version shows real race conditions affecting correctness and consistency. By applying mutexes, per-resource locking, and the readers-writers pattern, the synchronized version eliminates all concurrency issues.

The system behaves predictably, produces correct outputs, and stops properly at exam 9999.

This assignment highlights:

- Why critical section protection is necessary
- How design choices (e.g., per-question locks) improve parallelism

- How to analyze and avoid race conditions, deadlock, and starvation
- How concurrency transforms even simple tasks into complex problems