**SYSC 4001 Operating Systems Fall 2025**

Assignment 3

Part 2

Part C Report & Discussion of Design

SYSC4001 L3 – Group 16

Muhammad Ali – 101291890

Gregory Horvat – 101303925

**Part 2.c)**

Within the various test case outputs for both Part A and B, there is no deadlock observed.

Execution order for Part A is unpredictable and overlap one another. This can be seen in the example output below in Figure 1:

```
Loaded exam for student 1
TA 1: reviewing rubric
TA 4: reviewing rubric
TA 2: reviewing rubric
TA 3: reviewing rubric
TA 5: reviewing rubric
TA 3: correcting rubric question 1
TA 3: rubric question 1 changed from 'A' to 'B'
TA 1: correcting rubric question 1
TA 1: rubric question 1 changed from 'B' to 'C'
TA 4: correcting rubric question 1
TA 4: rubric question 1 changed from 'C' to 'D'
TA 1: correcting rubric question 2
TA 1: rubric question 2 changed from 'B' to 'C'
TA 4: correcting rubric question 2
TA 4: rubric question 2 changed from 'C' to 'D'
TA 2: correcting rubric question 4
TA 2: rubric question 4 changed from 'D' to 'E'
```

*Figure 1: Part A Test Case Output for 5 TAs*

The exact ordering of when the TA's review the rubric and correct a question to the rubric is entirely random. No Deadlock is observed due to the implementation of Part A. There are no locks or semaphores to create the conditions needed for Deadlock. TAs are free to explore any segment of the shared memory whenever they choose.

The design of Part A is free of all semaphore protection, critical sections of functions accessed by multiple TA's at once run the risk of race conditions. The review_rubric function as shown in Figure 2 displays this issue:

```
void review_rubric(int ta_id, SharedData* data) { // TA reviews the rubric
    cout << "TA " << ta_id << ": reviewing rubric" << endl;

    for (int i = 0; i < 5; i++) { // for every exercise in the rubric
        double delay = random_delay(0.5, 1.0); // get the delay
        usleep(delay * 1000000); // function wide sleep

        if (rand() % 5 == 0) {
            cout << "TA " << ta_id << ": correcting rubric question " << (i + 1) << endl;
            char current = data->rubric[i][2];
            data->rubric[i][2] = current + 1; // change ASCII by 1
            cout << "TA " << ta_id << ": rubric question " << (i + 1) << " changed from '" << current << "' to '" << data->rubric
            save_rubric(data);
        }
    }
}
```

*Figure 2: review_rubric Function from Part A .cpp file*

If multiple TAs were to execute on the function simultaneously for the same rubric question, the final new value of the rubric could be misrepresented. I.e. TA1 reads B and updates it to C, TA2 also reads B and incorrectly updates it to C instead of D.

Part A featured no design elements regarding solving the critical section problem, it is purposefully designed poorly to highlight and showcase the issue of not protecting critical sections.

Execution order for Part B is defined by the order of the semaphores/critical sections. TA's must follow the defined order of reviewing, correcting if needed and marking before the next exam can be loaded as shown in Figure 2:

```
loaded exam for student 1
TA 1: reviewing rubric
TA 2: reviewing rubric
TA 1: correcting rubric question 1
TA 1: rubric question 1 changed from 'A' to 'B'
TA 2: correcting rubric question 1
TA 2: rubric question 1 changed from 'B' to 'C'
TA 1: correcting rubric question 4
TA 1: rubric question 4 changed from 'D' to 'E'
TA 1: claiming question 1
TA 1: marking exam 1 question 1
TA 2: claiming question 2
TA 2: marking exam 1 question 2
TA 1: finished marking exam 1 question 1
TA 1: claiming question 3
TA 1: marking exam 1 question 3
TA 1: finished marking exam 1 question 3
TA 1: claiming question 4
TA 1: marking exam 1 question 4
TA 2: finished marking exam 1 question 2
TA 2: claiming question 5
TA 2: marking exam 1 question 5
TA 2: finished marking exam 1 question 5
loaded exam for student 2
```

*Figure 3: Part B Test Case Output for 2 TAs*

Part B featured design elements in regard to solving the critical section problem in the form of semaphores. Semaphore wait and signal functions were defined as shown in Figure 4 to correctly block other processes from accessing critical sections of functions when a given process was currently using them. The use of these wait and signal functions can be seen within the reviewing rubric and question marking functions as shown in Figures 5 and 6.

```cpp
void sem_wait(int semid, int sem_num){
    struct sembuf op;
    op.sem_num = sem_num; // which semaphore to operate on, 0 = rubric, 1 = question marking, 2 = exam loading
    op.sem_op = -1; // if <0, blocks/waits
    op.sem_flg = 0; // if counter == 0, block
    semop(semid, &op, 1); // execute the semaphore op
}

void sem_signal(int semid, int sem_num){
    struct sembuf op;
    op.sem_num = sem_num;
    op.sem_op = 1; // unlocked
    op.sem_flg = 0; // if counter == 0, block
    semop(semid, &op, 1); // execute the semaphore op
}
```

*Figure 4: sem_wait and sem_signal functions shown in Part B .cpp file*

```cpp
void review_rubric(int ta_id, SharedData* data, int semid) { // TA reviews the rubric
    cout << "TA " << ta_id << ": reviewing rubric" << endl;

    for (int i = 0; i < 5; i++) { // for every exercise in the rubric
        double delay = random_delay(0.5, 1.0); // get the delay
        usleep(delay * 1000000); // function wide sleep

        if (rand() % 5 == 0) {
            sem_wait(semid, 0); // semaphore waits for signal to get access to correct rubric
            cout << "TA " << ta_id << ": correcting rubric question " << (i + 1) << endl;
            char current = data->rubric[i][2];
            data->rubric[i][2] = current + 1;
            cout << "TA " << ta_id << ": rubric question " << (i + 1) << " changed from '" << current << "' to '" << data->rubric[i][2] << "'" << endl;
            save_rubric(data);
            sem_signal(semid, 0); // semaphore frees signal
        }
    }
}
```

*Figure 5: review_rubric function with Semaphore signals shown in Part B .cpp file*

```cpp
void mark_questions(int ta_id, SharedData* data, int semid) { // TA marking a exercise
    while (!data->finished) { // while we still have students to mark
        // Check if this TA needs to review rubric for current exam
        if (data->reviewed_rubric[ta_id - 1] == 0) {
            review_rubric(ta_id, data, semid);
            data->reviewed_rubric[ta_id - 1] = 1; // Mark as reviewed
        }

        int question = -1;
        int current_exam_num; // Store exam number locally to avoid race condition

        sem_wait(semid, 1);
        for (int i = 0; i < 5; i++) { // parse through all of the exercises
            if (data->questions_marked[i] == 0) { // if a question isnt marked
                question = i;
                cout << "TA " << ta_id << ": claiming question " << (question + 1) << endl;
                data->questions_marked[i] = 1; // if a question is marked
                break;
            }
        }
        current_exam_num = data->current_student_num; // Capture exam number while protected
        sem_signal(semid, 1);

        if (question == -1) { // if all questions were marked for the current exam
            sem_wait(semid, 2); // wait for semaphore signal
            if (data->current_student_num == 9999) {
                cout << "TA " << ta_id << ": setting finished flag for student 9999" << endl;
                data->finished = true;
                sem_signal(semid, 2);
                return;
            }
            int next_student = get_next_exam(data->current_student_num);
            if (next_student == -1){
                cout << "TA " << ta_id << ": no more exams, setting finished flag" << endl;
                data->finished = true;
                sem_signal(semid, 2);
                return;
            }
            load_exam(data, next_student);
            sem_signal(semid, 2); // release semaphore signla
            continue;
        }

        cout << "TA " << ta_id << ": marking exam " << current_exam_num << " question " << (question + 1) << endl;
        double delay = random_delay(1.0, 2.0); // marking delay
        usleep(delay * 1000000); // function wide sleep
        cout << "TA " << ta_id << ": finished marking exam " << current_exam_num << " question " << (question + 1) << endl;
    }
}
```

*Figure 6: mark_questions function with Semaphore signals shown in Part B .cpp file*