SYSC4001_A3_P2 Part C Report

Here is the link to A3_P2: https://github.com/kyrafingas/SYSC4001_A3_P2

As there was no deadlock in any of the tested runs for our code, we will briefly discuss the execution order for the processes.

Process Explanation
The processes are forked from the parent process in main after some semaphore initialization, shared memory creation, and file reading. One less than how many TAs are specified in the TA file are forked into existence, as the parent file is also being used as a TA. The processes each run run_simulation().

In run_simulation(), the process enters a while loop until it reads all exam files, reaches student id 9999, or one of the other processes terminates (*done==true). For each iteration of the while loop, the TA has two tasks: 1. Read through each rubric line and regrade is necessary(50% chance) and 2. Help mark unmarked exam questions until the exam is finished (possibly loading a new exam).

When the TA exits the while loop, it is immediately terminated unless it is the parent, then the parent cleans up the program.

Execution Order
Execution order for the processes varies per execution because choosing whether to regrade and marking exam questions both take random amounts of time. This causes the processes to reach semaphores at different times to each other and differently per run.

Each process can read the rubric simultaneously, but only one at a time can write at once due to a semaphore in the function correct_rubric(). If more than one process is trying to write at once, the first process will obtain the mutex and the others will have to wait until the running process releases the mutex.

After reviewing the rubric, the processes will begin marking the exam. Each process can mark simultaneously but only one at a time can write to the questions_marked vector of booleans that keeps track of which questions have been marked. If more than one process is trying to write at once, the first process will obtain the mutex and the others will have to wait until the question is set to marked.

This same mutex is also used for loading the next exam. After all the questions have been marked, the final boolean value in current_exam.questions_marked is reserved for whether the next exam has been loaded or not. When a process acquires a mutex to do this, it does not release it until the next exam has been completely loaded, meaning that no questions can be graded while the exam is being loaded into the system.

When all exam files have been read or a student id of 9999 is reached, the process who detects this terminates. The others follow suit by catching these conditions or checking if (*done==true). Each process is then out of the run_simulation() and all children terminate.

Critical Section Problem

There are three requirements associated with the solution to the critical section problem: Mutual Exclusion, Progress, and Bounded Waiting. Our solution for Part 2 considers all three requirements.

Mutual Exclusion means ensuring that only one process can use a critical section at once. Our code uses semaphores to address this requirement, ensuring our critical sections are properly protected.

Progress means allowing processes to enter a critical section if no process is currently there. Our code makes this decision by allowing the first process to reach a SemaphoreWait() call to enter when the semaphore is free. This means that processes are scheduled on a first come first serve basis and ensures progress.

Bounded waiting means there is a limit to how long a process waits before it has a chance to enter a critical section. Although our program does not address starvation directly by implementing some sort of timer, our code keeps the protected sections short and separated. This means that processes are unlikely to get caught in deadlock, livelock, or stuck inside a critical section for a long time. With FCFS scheduling, starvation becomes very unlikely.