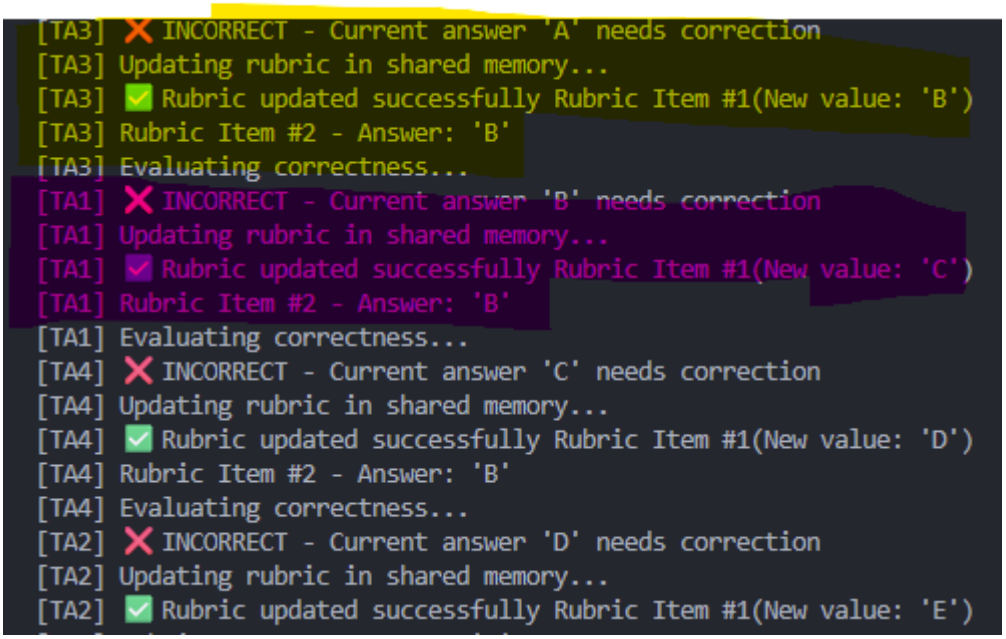
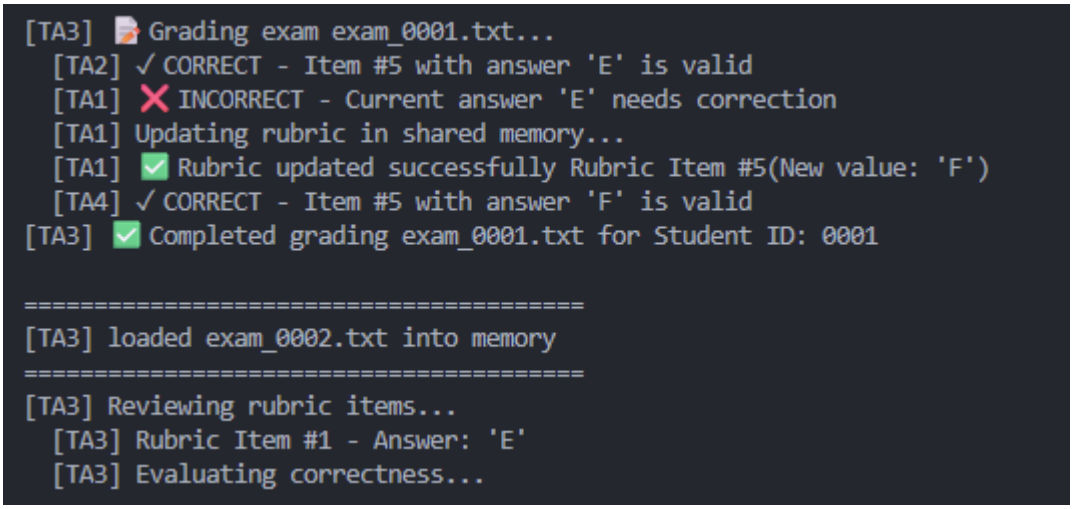
Question: If there is no deadlock on any of your runs, briefly discuss the execution order for the processes.

When a TA process starts, it first accesses the shared rubric. For each rubric item, the TA simulates a grading delay between **0.5 and 1 second**, then generates a random boolean value representing whether the item is correct.

If the generated value is **false**, the rubric item is considered incorrect. The TA must then enter a **critical section** protected by the rubric semaphore. Inside this section, it updates the shared rubric in memory and writes the modified rubric back to the rubric file. During this time, all other processes attempting to access the rubric remain blocked. This behavior is evident in the output: once the semaphore is released, the next TA resumes exactly at the rubric item it was waiting to read before it was blocked.



After finishing all rubric checks, the TA acquires the **exam mutex** to ensure exclusive access to the exam file. Within this critical section, the TA records its name in the exam file, increments the exam filename (e.g., exam_0001.txt → exam_0002.txt), loads the next exam into shared memory, and only then releases the exam mutex. This ensures that the TA grading a given exam is also the one responsible for loading the next one.



This order of operations ensures:

- TAs take turns accessing and updating shared resources,
- The output shows clear transitions: when a TA releases a mutex, the next TA resumes work and can read changed data,
- The same TA who finishes grading one exam typically loads the next exam due to critical section logic.

Question: A discussion of your design in the context of the three requirements associated with the solution to the critical section problem.

Accessing the Rubric Resource

To satisfy the requirements of the critical section problem, access to the shared rubric is strictly controlled. A TA enters the critical section **immediately before evaluating whether a rubric item needs to be corrected**, and the semaphore is released **only after the update to the rubric has been completed**. This ensures that no two processes modify the shared rubric simultaneously, maintaining mutual exclusion and preventing race conditions.

```
float randomNumber = randomNumGenerator(0.5, 1);
            delay(randomNumber);
            bool decision = randomBool();

            if (decision == false)
            {
                std::cout << "  [" << ta_name << "] ❌ INCORRECT - Current answer
'" << shm_ptr->rubric[i + 3] << "' needs correction" << std::endl;
                std::cout << "  [" << ta_name << "] Updating rubric in shared
memory..." << std::endl;

                // Use rubric_mutex for rubric modifications
                sem_wait(&shm_ptr->rubric_mutex);
                shm_ptr->rubric[i + 3] += 1;

                // Convert char array to string for writing to file
                std::string rubricString = charArrayToString(shm_ptr->rubric);

                // Write updated rubric back to rubric file
                FILE *rubric_file = fopen("rubric", "w");
                if (rubric_file != NULL)
                {

                    fputs(rubricString.c_str(), rubric_file);
                    fclose(rubric_file);
                }
                std::cout << "  [" << ta_name << "] ✅ Rubric updated successfully
" << "Rubric Item #"<< shm_ptr->rubric[i] << " (New value: '" << shm_ptr->rubric[i
+ 3] << "')" << std::endl;
                sem_post(&shm_ptr->rubric_mutex);
            }
            else
            {
                std::cout << "  [" << ta_name << "] ✓ CORRECT - Item #" <<
shm_ptr->rubric[i] << " with answer '" << shm_ptr->rubric[i + 3] << "' is valid"
<< std::endl;
            }
```

Accessing the Rubric Resource

Access to the exam file is also protected using a mutex to ensure mutual exclusion. After completing the rubric checks, the TA acquires the exam mutex before interacting with the current exam file. The mutex is released in two situations: first, if the global done flag indicates that grading is complete; and second, if the TA encounters an exam file containing the terminal value 9999, in which case the process terminates. Otherwise, the TA updates the exam file, loads the next exam_XXXX.txt filename into shared memory, and then releases the exam mutex. This guarantees that only one TA at a time advances the exam sequence and writes to the exam files.

```
sem_wait(&shm_ptr->exam_mutex);
        std::string path = shm_ptr->exams;
        std::string filename =
std::filesystem::path(path).filename().string();
        std::string file_number = filename.substr(5, 4);

        // Mark the current exam file (using local path variable)
        std::cout << "\n[" << ta_name << "] 📝 Grading exam " <<
filename << "..." << std::endl;
        // Get next exam file in critical section

        FILE *check_file = fopen(shm_ptr->exams, "r");
        char first_line[256];

        if (check_file == NULL)
        {
            std::cout << "\n[" << ta_name << "] File not found: " <<
filename << " - Terminating." << std::endl;
            sem_post(&shm_ptr->exam_mutex);
            break;
        }

        if (fgets(first_line, 256, check_file) != NULL)
        {
            first_line[strcspn(first_line, "\n")] = '\0';

            if (strcmp(first_line, "9999") == 0)
            {
                std::cout << "\n[" << ta_name << "] *** TERMINATION
SIGNAL DETECTED (9999) - Stopping work ***" << std::endl;
                fclose(check_file);
                sem_post(&shm_ptr->exam_mutex);
                exit(0);
```

```
                break;
            }
        }

        fclose(check_file);

        float randomMarkNumber = randomNumGenerator(1.0, 2.0);
        delay(randomMarkNumber); // Delay outside any lock

        // Write to exam file (no lock needed - each TA works on
different files)
        FILE *file_ptr = fopen(path.c_str(), "a");
        if (file_ptr == NULL)
        {
            perror("Error opening file");
            break;
        }

        char data[50];
        fputs("\n", file_ptr);
        snprintf(data, sizeof(data), "graded by: %s ", ta_name);
        fputs(data, file_ptr);
        fclose(file_ptr);

        // Read back the student ID from the exam file
        FILE *file_to_read = fopen(path.c_str(), "r");
        char file_content[256];
        if (fgets(file_content, 256, file_to_read) != NULL)
        {
            file_content[strcspn(file_content, "\n")] = '\0'; // Remove
newline
            std::cout << "[" << ta_name << "] ✅ Completed grading " <<
filename << " for Student ID: " << file_content << std::endl;
            // Increment to next exam while still holding lock
        }
        else
        {
            std::cout << "[" << ta_name << "] ⚠️  Warning: Could not
read student ID from file" << std::endl;
        }
        loadNewExam(ta_name, shm_ptr->exams);
        fclose(file_to_read);
        sem_post(&shm_ptr->exam_mutex);
```