CS307 AS3 : Study Room Synchronization

Yavuz Can Atalay

12 December 2025

1 Introduction

The purpose of this assignment is to design and implement a thread–safe synchronization model that simulates the behavior of students arriving at, participating in, and leaving a study session inside an Information Center. The core logic of the system is implemented in `Study.h`, where the `Study` class posessing all shared state and synchronization mechanisms. Provided source files (`study_test.cpp`, `study_test2.cpp`) create multiple worker threads that execute the `arrive()` `start()` and `leave()` methods concurrently, enabling the testing of race condition scenarios, barrier behavior, session formation, and proper exit handling.

Fundamentally, the challenge arises from the need to manage multiple threads representing students who may arrive in any order and at any time. The code must therefore coordinate access to shared variables, detect when a full study session can begin, ensure correct blocking behavior for students who must wait, and allow students to leave without causing deadlocks or inconsistent state. Without proper synchronization, threads may interfere with each other, leading to race conditions, double session starts, missed wake-ups, or threads waiting indefinitely.

To address these issues, the implementation relies on several key synchronization methods. POSIX mutexes are used to guarantee exclusive access to shared variables, preventing concurrent modifications that could corrupt program state. Semaphores serve as counting and signaling mechanisms to limit room capacity and to implement a thread barrier that controls when a the room is reavailable after session s terminated. These constructs together enforce ordering constraints between threads, ensure that only the last required student can trigger session formation, and allow all participating threads to proceed in a coordinated manner. Through the combined use of mutexes, semaphores, and careful state management, the program aims to achieve a correct and deadlock free simulation of concurrent student behavior.

2 Start

```
int sessionSize;
int tutorPresent;
int totalCapacity;
int current_students;
bool sessionstarted;

pthread_t tutor_ID;

sem_t Capacity;        // Room capacity
sem_t TutorSpeaking;   // Students wait on this in leave() if tutor exists, works as a barrier
pthread_mutex_t mtx;   // Protects shared state variables such as current_students and sessionstarted

// Caller thread is tutor or not?mç

bool isTutor() {
    if (tutorPresent == 0) return false;
    return pthread_equal(pthread_self(), tutor_ID); // Necessary???
}
```

Figure 1: Variables

Shared variables and a helper function are declared at the beginning of the class definition to manage the room's state. sessionSize represents the room's student capacity (excluding the tutor), while tutorPresent indicates whether a tutor is required. totalCapacity is the sum of sessionSize and tutorPresent, representing the total number of people (students and tutor) allowed in the room. current_students

tracks the number of students currently inside. Lastly, sessionStarted is bool flag raised by last student when room is full and session starts.

Additionally, I declared mutexes and semaphores to establish a properly synchronized environment for multiple threads. tutor_ID is used within the isTutor() helper functioning to verify the current thread's role. The Capacity semaphore is a counting semaphore that limits entry to ensure the room is not overfilled. TutorSpeaking acts as a barrier (binary semaphore) to synchronize departure, ensuring students remain inside until the tutor announces the session is over. Finally, mtx is a mutex used to protect critical sections, such as updating the current_students variable.

3 Constructor and Deconstructor

```
Study(int sessionSize, int tutorPresent) :
    sessionSize(sessionSize), tutorPresent(tutorPresent) {

    if (sessionSize <= 0) {
        throw std::invalid_argument("An error occurred.");
    }
    if (tutorPresent != 0 && tutorPresent != 1) {
        throw std::invalid_argument("An error occurred.");
    }
    else{
    this->totalCapacity = sessionSize + tutorPresent; // Include tutor as a student if present
    this->current_students = 0;
    this->sessionstarted = false;

    sem_init(&Capacity, 0, totalCapacity);
    sem_init(&TutorSpeaking, 0, 0);
    pthread_mutex_init(&mtx, NULL);
    }
}

~Study() {
    sem_destroy(&Capacity);
    sem_destroy(&TutorSpeaking);
    pthread_mutex_destroy(&mtx);
}
```

Figure 2: Constructor and Deconstructor

The Constructor initializes the study room's state and the synchronization primitives required to manage concurrency, which are already listed above. First, it validates the input arguments sessionSize and tutorPresent to ensure they are within valid ranges (positive integers and binary flags, respectively). If the numbers are not desired, an exception is thrown.totalCapacity is calculated by adding the tutor (if present) to the session size, determining exactly how many threads are required to fill the room.

Crucial part for this assigment is to create a concurrent environment.So the constructor initializes two semaphores and one mutex. The Capacity semaphore is initialized with totalCapacity, effectively acting as a "permits" counter that allows exactly that many threads to enter the room before blocking subsequent arrivals. The TutorSpeaking semaphore is initialized to 0; this acts as a barrier or a signal mechanism where students will wait until the tutor "posts" to it. A mutex mtx is initialized to protect critical sections where shared variables like current_students and sessionstarted are modified. The Destructor simply cleans up these resources to prevent memory leaks.

4 Arrive Method

```
void arrive() {
    printf("Thread ID: %lu | Status: Arrived at the IC.\n", pthread_self());

    // If a session is running, this blocks the thread here until the session ends.
    // Check if all students should wait outside unlimitedly or not
    sem_wait(&Capacity);

    // 3. Enter Critical Section
    pthread_mutex_lock(&mtx);
    current_students++;

    // If session isn't full
    if (current_students != totalCapacity) {
        printf("Thread ID: %lu | Status: Only %i students inside, studying individually.\n", pthread_self(), current_students);
    } else {
        // Session can start if capacity is full
        sessionstarted = true;
        if (tutorPresent) {
            tutor_ID = pthread_self(); // The last arrival becomes the tutor, but a student
        }
        printf("Thread ID: %lu | Status: There are enough students, the study session is starting.\n", pthread_self());
    }
    pthread_mutex_unlock(&mtx);
}
```

Figure 3: Arrive Function

The arrive method handles the entry logic and group formation. The code block here require a specific order of operations to prevent deadlock and ensure correct output. First, the thread prints its "Arrived" status before attempting to acquire any synchronization locks. This ensures that even if the room is full and the thread is about to be blocked by a semaphore, it successfully announces its presence, maintaining the chronological log of events, like it is demonstrated in the example output files.

After printing, the thread performs sem_wait(&Capacity). If the room is full, the thread blocks here, waiting for ongoing session to be finished. Otherwise, it proceeds to the critical section protected by mtx. Inside the lock, the current_students counter is incremented. The logic then checks if the room is full (current_students == totalCapacity). If it is, implying there are enough threads filling the room, the boolean flag sessionstarted is set to true. If a tutor is required, the thread identifies itself as the tutor using pthread_self() as the last student. Finally, the thread prints whether it is waiting for more students or if the session is starting, before releasing the mutex

5 Leave Method

The leave method is the most complex part of the synchronization, handling three distinct case scenarios: leaving when no session formed, leaving after session is finished without a tutor and with a tutor.

5.1 Scenario 1: No Session Formed

```
// CASE 1: Session hasn't started while waiting

if (!sessionstarted) {
    printf("Thread ID: %lu | Status: No group study formed while I was waiting, I am leaving.\n", pthread_self());
    current_students--;
    // Since they are leaving individually, we release one spot for the next person
    sem_post(&Capacity);
    pthread_mutex_unlock(&mtx);
    return;
}
```

Figure 4: No Session formed

If the thread wakes up and finds sessionstarted is false, it implies the group hasn't have enough student to be formed at the moment. In this case, synchronization is simple: the thread decrements current_students, prints a leaving message, and immediately calls sem_post(&Capacity). This releases a single spot in the room for a new student immediately, as there is no group coherence required. Before returning, it release the lock to prevent deadlock.

5.2 Scenario 2: Session Active (With Tutor)

```
// CASE 2: Session Started
if (tutorPresent) { // With Tutor
    if (isTutor() == true) {
        // Tutor speaks first
        printf("Thread ID: %lu | Status: Session tutor speaking, the session is over.\n", pthread_self());

        // Wake up all students waiting for the tutor
        // We post (totalCapacity - 1) times because that is how many students are waiting, exclude tutor with -1
        for(int i = 0; i < (totalCapacity - 1); i++) {
            sem_post(&TutorSpeaking);
        }
    } else {
        // student block, wait for tutor to speak
        // unlock mutex while waiting to avoid deadlock
        pthread_mutex_unlock(&mtx);
        sem_wait(&TutorSpeaking);
        pthread_mutex_lock(&mtx); // Re-acquire lock to print and decrement

        printf("Thread ID: %lu | Status: I am a student and I am leaving.\n", pthread_self());
    }
} else {
    // No tutor, everyone just leaves naturally
    printf("Thread ID: %lu | Status: I am a student and I am leaving.\n", pthread_self());
}
```

Figure 5: Tutor Case

If the session is active and a tutor is present, strict ordering is enforced. If the thread is the Tutor, which is controlled with helper function isTutor(), it anounce that session is over. It then acts as the release mechanism for the barrier by looping (totalCapacity - 1) times and calling sem_post(&TutorSpeaking). This signals every waiting student thread that they may proceed. Important part is Since all students must wait the announcement of the tutor before leaving, they call sem_wait(&TutorSpeaking) which can be only opened by tutor, So it works as a barrier forcing preprocesses threads to wait for the tutor thread. The crucial consideration here is looping sem_post(&TutoSpeaking), acting like allowing each thread to leave the session not simultaneously but gradually. However since it is protected by a mutex, it does not cause any interleaving problem where a waiting student trying to enter the room when session is over and working students starts leaving.

In the code, If the thread is a Student, it must wait for the tutor. To avoid holding the lock while waiting (which would cause a deadlock), the student explicitly unlocks mtx, calls sem_wait(&TutorSpeaking), and then re-locks mtx. This "unlock-wait-relock" pattern ensures the student is paused until the tutor speaks, but safely re-enters the critical section to update the student counter and print their leaving message.

5.3 Scenario 3 : Session Active (Without Tutor)

If there is no tutor inside the session, students may leave however they want after study session is over. There is no order that should be kept in control in non-Tutor case. Just last student, in this case acts as the gate opener by calling sem_post enabling the room for waiting students. After the loop is terminated, it release the lock and let the waiting students enter the room.

```
    } else {
        // No tutor, everyone just leaves naturally
        printf("Thread ID: %lu | Status: I am a student and I am leaving.\n", pthread_self());
    }
    current_students--;
    //last student to leave resets the session
    if (current_students == 0) {
        printf("Thread ID: %lu | Status: All students have left, the new students can come.\n", pthread_self());
        sessionstarted = false;

        // Restore capacity ONLY when the room is empty.
        // This allows waiting students to enter.
        for(int i = 0; i < totalCapacity; i++) {
            sem_post(&Capacity);
        }
    }

    pthread_mutex_unlock(&mtx);
}
```

Figure 6: No Tutor Case

5.4 Last Student Cleanup

I already elaborated on this but this is the most crucial part of the homework so I felt like explaining explicitly. Regardless of the scenario, the very last thread to leave the room (when current_students == 0) performs last adjustments. It resets sessionstarted to false and prints the "All students have left" message. Crucially, it does not just release one semaphore; it loops through totalCapacity and performs sem_post(&Capacity) for every spot in the room, which can be observed in both Tutor and NonTutor examples. This effectively "opens the study for another session," allowing the next waiting student group to enter the room together. All this is performed inside a mutex lock to ensure concurrency and cease possible race conditions.

6 Conclusion

In this assignment, a robust and thread-safe implementation of the Study class was aimed to be developed to manage the synchronization of students in an study room. By utilizing POSIX mutexes and certain semaphores(counting and binary semaphores), certain challenges of concurrent programming, including race conditions, deadlocks, and busy-waiting are tried to be handled. In the end, I believe a well constructed concurrent class is created solving problems mentioned in the assignment.