Sabanci University

Operating Systems

CS 307

# Programming Assignment - 3: Synchronization IC: Managing a Study Room with Semaphores

Release Date: 28 November 2025
Deadline: 8 December 2025 23.55

## 1 Introduction

In the lectures you have seen synchronization primitives like mutexes, semaphores, condition variables and barriers which are essential tools for coordination and synchronization of multi-threaded programs. In Programming Assignment 3 (PA3), we ask you to implement a C++ class for organizing a group study room in the Information Center (IC). Students and tutors will be represented by threads and you have to synchronize them using semaphores on the usage of the shared resource of the study room.

Students arrive at the IC, hoping to use the group study room for a collaborative study session. They are allowed into the room if it is not already full and if there is no active study session taking place. For simplicity, we will assume that a session can only begin when there is a fixed group of students ready to participate. Additionally, depending on reservation type, one of the students may act as a tutor for that session.

When a session begins, none of the students inside can leave before the session ends, and they will not let any new students inside. When the session ends and a tutor is present, the tutor must first announce the session is over; only then the students can start leaving. If there is no tutor for that session, the students will agree upon themselves and leave without such an announcement. After everyone else has left, the last student also needs to inform waiting students that the room is now available.

## 2 Problem Description

In this programming assignment you are tasked with preparing a C++ header file named `"Study.h"` implementing the `Study` class. This class will include a constructor, `arrive`, `start` and `leave` member methods. You have to implement all methods except `start` which will change with respect to test cases. The details will be explained later in this section.

In our simulation the students will be represented by threads. These threads will call `arrive`, `start` and `leave` methods sequentially in this order. In these methods threads will use print statements to indicate their state and you are tasked to provide synchronization between threads to make these states obey the following rules:

- When there are enough students for a session inside the room i.e., returned from the `arrive` method, students must start the session.

- There cannot be more students in the room than the number required for a session. 2
- 3 `start` method will represent a student starts passing some time in the room. If there are not enough students for a session, we can interpret time spent in this method as an individual study or scrolling on Tiktok while waiting. When there are enough students and a session starts, the time spent in this method represents an organized session experience. If a student has finished studying (returned from `start` and called `leave`) and a session has not started, they must leave without participating in any group session.
- 4 If there is a tutor, non-tutor students must not return from the `leave` method until the tutor announces that the session has ended. Otherwise, students can leave freely.
- 5 When a session ends, the last student to leave must notify the waiting students that are blocked in the `arrive` method so that they can return from this method and enter the room.
- 6 While a session is in progress, no one can enter the room before the session ends and all students and (if exists) the tutor involved in the session leaves the room.

We expect you to solve synchronization problems above using semaphores and barriers only. When a thread is blocked due to a condition described above, it should not busy-wait in a loop. Please, note that, if you use mutexes instead of semaphores, busy-waiting loops will be inevitable. If you do so, you might lose some points (see Section 9). Basically, `start` and `leave` member methods should not contain any loops.

For other synchronization problems and critical sections that are not described above, you can use mutexes. For instance, if you want to ensure that an increment operation on a shared counter is atomic, you can wrap this operation with mutex lock and unlock methods.

For more information of these rules, how they can be achieved and for a typical behaviour of a student thread please read the next section. You can also find some sample runs at 10.

## 3 Study Class

In your simulation you will implement the `Study` class. In its shared state it must have some fields to keep number of students, whether there is a tutor and who is the tutor if exists, whether there is a study session going on and the necessary synchronization objects. You should initialize and destroy these fields by writing a proper constructor and destructor. This class is supposed to have three public member methods: `arrive`, `start` and `leave`. There will also be another function, `start`, that will be declared inside test cases.

Each student will be represented by a POSIX thread (pthread). These threads will be created and the method they execute will be implemented in the main program. However, you can safely assume that threads will execute `arrive`, `start` and `leave` methods in this order.

### 3.1 Constructor

The constructor requires two input arguments. The first argument is declaring how many students are needed to start a study session and the second argument indicates whether there will be a tutor in the session. The second parameter can be either 0 or 1. An example call of it is as follows:

$ Study(4,1)

In this case, 4 students are needed for a study session and there will be a tutor in the session. So, in total, 5 students (including the tutor) must be in a room to start a study session.

In the constructor, you also need to check the validity of the arguments. There are two conditions:

- First argument must be a positive integer.
- Second argument must be either 0 or 1.

If these conditions do not hold, your program needs to throw an exception.

## 3.2 Arrive Method

Below, you can find a typical behaviour of a student's arrive function:

- It first prints: `Thread ID:` $<$ *tid* $>$ `| Status: Arrived at the IC.`
- It checks if there is an active session already in progress. If so, it blocks, otherwise it enters the room.
  - You can keep a boolean variable to check if a session is in progress or not. You can use a semaphore to block threads in `arrive` method when the room is full and a session is ongoing.
- When it arrives, it checks how many students are in the room. There are two scenarios depending on the tutor option:
  If a session does not require a tutor, the student checks whether, including itself, the number of students in the room has reached the required session size. If so, this arrival completes the group and the session starts.
  If there has to be a tutor, the student checks whether, excluding itself, the number of students already inside equals the required number of students for the session. If this condition holds, this arriving student becomes the tutor, and together with the existing students, they form the complete group. The session then starts.
  If one of these conditions holds, it starts the session and prints

  `Thread ID:` $<$ *tid* $>$`| Status: There are enough students, the study session is starting.`

  Otherwise it prints

  `Thread ID:` $<$ *tid* $>$ `| Status: Only` $<$ *n* $>$ `students inside, studying individually.`

  and passes some time waiting for another student to start the session.
- If you are counting the number of people inside the room, this number must be incremented when someone is entering the room. Note that, increment operation is not atomic and you might need to use a mutex for this purpose.

## 3.3 Start Method

Passing some time is simulated using a `start` method that will be provided by us when running test cases. For your PA you are not responsible for how a student passes its time in the location. You can assume that if the session started when a student was waiting, it automatically joins the session. Please, declare this method in the class signature but do not implement it.

## 3.4 Leave Method

At this point, `start` method finished its execution. Depending on some conditions, this method prints the following lines:

- If a session has not started it prints

  ```
  Thread ID: < tid > | Status: No group study formed
  while I was waiting, I am leaving.
  ```

  and leaves the room. Next items apply if a session is in progress.

- If there is the session tutor, and this student is assigned as a tutor it prints

  ```
  Thread ID: < tid > | Status: Session tutor speaking,
  the session is over.
  ```

- If this student is not the session tutor and the tutor has already left, it prints

  ```
  Thread ID: < tid > | Status: I am a student and I
  am leaving.
  ```

- If there is not any tutor in the session, it prints

  ```
  Thread ID: < tid > | Status: I am a student and I
  am leaving.
  ```

- The last student that leaves the room also prints

  ```
  Thread ID: < tid > | Status: All students have left,
  the new students can come.
  ```

  as its last statement and leaves the room.

## 4 General Considerations and Corner Cases

1 • Multiple threads might try to print at the same time, which might result in garbled output. You are also responsible for preventing this. The easiest way to do so is using `printf`, as it is atomic. If you want to use streams like `cout`, you can do this with the help of synchronization primitives like locks and semaphores.

2 • When a session is in progress at the study room, allow new students to enter only after the last student in the session leaves, by using semaphore API methods.

3 • Consider using a barrier to pick the tutor and printing its output. i.e: when a study group is formed, students should wait in the barrier until a tutor prints its output

4 • If you're using a barrier in your implementation while synchronizing the exit outputs, do not forget to initialize the barrier in the beginning.

• Do not forget that when there are no session on the room, students could arrive and leave freely without blocking.

• Even though a tutor is picked for every session that is formed if tutor is present, it is also referred as a student in the document in general to prevent confusion.

• In order to select the tutor when a session group is formed, you might consider using a special field in your class of type `pthread_t` and set it to the tutor thread's thread ID when You pick the tutor in the arrive method. You might need this information in the `leave` method because the tutor prints a distinct line.

• In `leave` method, in order to understand if the current thread is the tutor, you have to compare `pthread` objects. You should use `pthread_equal` library method instead of "==" operator for the comparison.

## 5 Correctness

There are correctness conditions for the child(student) threads. First of all, it must be ensured that main thread always finishes the last and waits until all of its children threads terminate. Afterwards, main thread should output -*main terminates* to the console as the last thing.

Correctness of the student threads depends on the order and interleaving of strings they print to the console. To make things easier, let us declare some variables and give names to strings they print to the console at various steps first:

- *total_num*: The total number of threads.
- *student_count*: The number of students in a session, excluding the session tutor.
- *all_count*: The number of students in a session, including the study session tutor.
- *num_sessions*: The number of study sessions done
- *init*: The string `"Thread ID:` $< tid >$ `| Status: Arrived at the IC."`
- *enter_passtime*: The string `Thread ID:` $< tid >$ `| Status: Only` $< x >$ `students inside, studying individually.`
- *enter_session*: The string `Thread ID:` $< tid >$ `| Status: There are enough students, the study session is starting.`
- *everybody_left*: The string `Thread ID:` $< tid >$ `| Status: All students have left, the new students can come.`
- *tutor_leaving*: The string `Thread ID:` $< tid >$ `| Status: Session tutor speaking, the session is over.`
- *student_leaving*: The string `Thread ID:` $< tid >$ `| Status: I am a student and I am leaving.`
- *no_session*: The string `Thread ID:` $< tid >$ `| Status: No group study formed while I was waiting, I am leaving.`

Then, for any execution of your program with valid inputs, the following conditions must be satisfied. Note that due to the multi-threaded nature of the program, different executions may produce different valid outputs. Therefore, your output is not expected to match the sample run text files line by line; instead, make sure that your program satisfies below conditions:

- There must be exactly *total_num* times *init* strings printed to the console
- There must be exactly $total\_num - num\_sessions$ times *enter_passtime* strings printed to the console
- There must be exactly *num_sessions* times *enter_session* strings printed to the console
- There must be exactly *num_sessions* times *everybody_left* strings printed to the console
- There must be exactly *num_sessions* times *tutor_leaving* strings printed to the console
- There must be exactly $num\_sessions * student\_count$ times *student_leaving* strings printed to the console
- There must be exactly $total\_num - (num\_sessions * (all\_count))$ times *no_session* strings printed to the console
- For each thread *init*, *enter_passtime* or *enter_session*, *tutor_leaving* or *student_leaving*(if session exists), *no_session*(if no session exists) must be printed to the console in this order
- For each session after *enter_session* string, *tutor_leaving* must be printed to the console before any *student_leaving*
- For each session after *enter_session*, *init* strings might be printed in-between sessions but, no *enter_passtime* nor *enter_session* can be printed to the console before *everybody_left* string

See Section 10, for some sample output obeying the correctness conditions above.

## 6 Useful Information and Tips

- First, read this document from beginning to the end. Make sure that you understand what is the problem you need to solve and what is expected from you. You can mark important points and take some notes in this step. Then, develop your solution using pen and paper (maybe by writing an abstract pseudo-code). Then, start implementing the solution considering tips in this section, corner cases section and the grading section. Complete one grading item at a time obeying the preconditions. Make sure that your improvements and refinements do not violate previously completed grading items.

- If you have no idea on how to solve the the problem or where to use semaphores/barriers, please, start simple by reading and solving problems in Little Book of Semaphores. First read a problem and try to solve it yourself. Once you spend enough time on it, you can move to the solution. Then, you can try to understand whether your solution was correct or why the solution in the book works.

- If you wish to use a barrier, you can use off-the-shelf one like a `pthread barrier` or you can implement your own reusable barrier. Little Book of Semaphores contain semaphore based barrier implementations. Please note that solutions in that book are written in pseudo-code and might not be directly translated into C++ code. We also provide Python implementations for these barrier algorithms and again, Python Threads Library works completely different than `pthreads`. You should take this book for a guidance, not embrace as a solution.

- Take a look at the behavior of the pthread semaphores before using them.

- Ensure your program executions obey the correctness format. See Section  10 for correctness.

- Ensure that your application works correctly in cases where there are multiple sessions, such as eight threads or twelve threads when the session size is four.

- Your program should terminate properly, you will lose points if your program cannot terminate all the threads safely or in case of missing outputs. This requirement also entails that your threads should not have a deadlocking execution.

- You have to use pthreads (POSIX Threads) library and submit C++ file. Do not forget to use `#include<pthread>` statement. Any other thread library will not be accepted as a solution.

- Do not forget to use -*lpthread* option while compiling.

## 7 Supplementary Information

In this section, we provide further resources and tools that might help you during your implementation.

### 7.1 Posix Threads API

The pthread semaphores behave like Linux Zemaphores, not like the original Dijkstra semaphores we have seen in the lectures. So, before using the pthread semaphores, take a look at the manual page of the command `sem_wait` or check the web page. If you wish, you can implement Dijkstra semaphores using pthread condition variables and mutexes as in the lectures but it is discouraged since as stated below Helgrind and TSan can only detect race conditions and Lock-Order Inversions(deadlocks) if and only if semaphores and barriers are created using pthreads api or Annotations are made to user defined Semaphores which is a highly advanced concept that is not expected from you for this course. See Section 7.2 for more details. The choice is yours. In the report you will provide in the submission package, you have to explain which synchronization primitives you used and how you implemented them if you choose to implement them on your own.

### 7.2 Usage of Helgrind and Thread Sanitizer (TSan)

Since you'll solve a synchronization problem for PA3, you may find yourselves dispersed in a non-synchronized environment and it is sometimes overwhelming to search for race conditions in a multi-threaded program by yourself. Even if they are not fool-proof as stated in the recitation document, Helgrind - TSan (please check this link ), these tools will help you detect race conditions and deadlocks much more faster if you are familiar with your code.

## 8 Submission Guidelines

For this homework, you are expected to submit two files.

- **Study.h**: Your **C++** implementation of the header file where your `constructor`, `arrive` and `leave` methods are implemented.
- **report.pdf**: In your report, you must present the flow of your `arrive` and `leave` methods as a pseudo code. You discuss which synchronization mechanisms (semaphores, mutexes and barriers) you have chosen, how you implemented, used or adapted them to suit your needs and provide formal arguments on why your code satisfies the correctness criteria described above.

During the submission of this homework, you will see two different sections on SUCourse. For this assignment you are expected to submit your files **separately**. You should **NOT** zip any of your files. While you are submitting your homework, please submit your **report** to "PA3 – REPORT Submission", and your **code** to "PA3 – CODE Submission". SUCourse will **not** accept if your files are in a different format, so please be careful. If your submission does not fit the format specified above, your grade may be penalized up to 10 points.

## 9 Grading

Grading will be done automatically. If automated tests fail, your code will not be manually inspected for partial points. Some students might be randomly called for an oral exam to explain their implementations and reports.

- **Class Interface (10 pts):** When we include your header file to our cpp program, we can access the `constructor`, `arrive` and `leave` methods, and when we use them in our program we do not get any runtime errors.
- **Exception Handling (10 pts):** When the constructor is called with improper values, your program throws an exception.
- **No Sessions Case (15 pts)**
  - **Without Tutors (10 pts):** When there are no tutors, if the students are never able to participate in a session, your outputs fit our correctness criteria.

- **With Tutors (5 pts):** When there are tutors, if the students are never able to participate in a session, your outputs fit our correctness criteria.

- **At Most 1 Session Case (15 pts)**
  - **Without Tutors (10 pts):** When there are no tutors, if the students are able to participate in at most one session, your outputs fit our correctness criteria.
  - **With Tutors (5 pts):** When there are tutors, if the students are able to participate in at most one session, your outputs fit our correctness criteria.

- **Multiple Sessions Case (15 pts)**
  - **Without Tutors (10 pts):** When there are no tutors, if the students participate in multiple sessions, your outputs fit our correctness criteria.
  - **With Tutors (5 pts):** When there are tutors, if the students participate in multiple sessions, your outputs fit our correctness criteria.

- **Last student in a session informs others (10 pts):** When a session ends, and all the students have left, the last student prints the appropriate string.

- **Report(20 pts):** Your report explains your thread methods, how you implemented them, what kind of synchronization mechanisms you used and adapted and why you think that it is correct (-5 pts if your report is not in pdf format).

**Important Note:** It is possible to solve this problem using only mutexes. In this case, your implementation will suffer from the busy-waiting problem even though you do not use spin-locks. When a student comes when there is a session already in progress, it has to wait in a loop until the current session is over. Since this is one of the fundamental problems we were trying to avoid in the lectures, you will **lose 20 points**, if your implementation suffers from the busy-waiting problem. Note that you do not face such a problem if you use semaphores.

## 10 Sample Runs

Sample runs are included in the PA3 bundle as text files, allowing you to test your solution with various inputs. Each text file is named starting with either ‘study_test‘ or ‘study_test2‘, indicating the corresponding test class. The remainder of the file name follows the format ‘x_y_z‘. To test your solution, you can execute your code using the following command:

```
$ ./study_test x y z
```

i.e. if file name is study_test_4_4_1.txt , then in your comment line you should run

```
$ ./study_test 4 4 1
```