PROGRAMMING ASSIGNMENT (PA) 3  
DEMO SESSION SIMULATION

CS307 OPERATING SYSTEMS

SPRING 2022-2023

DEADLINE: 18 MAY 2023

REPORT BY

ZEYNEP KURTULUS

29045

Introduction

In the context of PA3, we implement a C++ program that stimulates a demo room. In this demo room there are two types of participants: assistants and students and each of them is represented with threads. Each participant/thread will enter the classroom, participate in the demo and leave the classroom. Demo room has some rules and we need to perform the required checks in our code and depending on these rules we experience some synchronization problems. There are multiple students and assistants who want to enter a classroom. The program uses POSIX threads and POSIX semaphores to address these synchronization problems. The details regarding the C++ implementation are stated in the following parts of this report.

C++ Program Implementation

At the beginning of the program we simply receive the input from the arguments entered by the user to the console. In the code **argv** is an array of strings that contains the command line arguments passed to the program; **argv[0]** is the name of the program, **argv[1]** is the number of assistants and **argv[2]** is the number of students. The program then performs an input check because the number of assistants must be positive and the number of students must be two times the number of assistants, if that is not the case we display an error message.

Moving on, we create two arrays of thread for students and assistants to keep track of the student and assistant threads. Also, **student\_ids** and **assistant\_ids** are created with the means of storing the unique identifiers to the corresponding threads. After that, **sem\_init** function is called to initialize the semaphores four semaphores (**student\_sem, assistant\_sem, demo\_sem, demo\_sem2 and mutex**). The semaphores are used for synchronization and coordination between the student and assistant threads. Here's an explanation of each semaphore and its usage:

* **student\_sem:** This semaphore controls the entry of students into the classroom. Its initial value is set to the number of assistants multiplied by 3 (**num\_assistants \*3**). Students call **sem\_wait(&student\_sem)** to request entry into the classroom and **sem\_post(&student\_sem)** to release the semaphore when they leave the classroom.
* **assistant\_sem:** This semaphore controls the participation of assistants in the classroom. It starts with an initial value of 0. When there are at least two waiting students and one waiting assistant, the assistant thread calls **sem\_post(&assistant\_sem)** to allow one assistant to participate. Assistant threads also call **sem\_wait(&assistant\_sem)** to participate and **sem\_post(&assistant\_sem)** to release the semaphore when they finish.**demo\_sem**: This semaphore is used to synchronize the demonstration process. It starts with an initial value of 0. When two students have entered the classroom, the assistant thread calls **sem\_post(&demo\_sem)** to allow the demonstration to start. After each demonstration, the assistant thread calls **sem\_wait(&demo\_sem)** twice to synchronize the next demonstration. By calling **sem\_wait(&demo\_sem)** twice, the assistant thread ensures that it waits until both students have finished and called **sem\_post(&demo\_sem)** before proceeding. The first **sem\_wait(&demo\_sem)** call waits for the first student to finish and release the semaphore. The second **sem\_wait(&demo\_sem)** call waits for the second student to finish and release the semaphore. Once both students have finished and called **sem\_post(&demo\_sem),** the assistant thread can proceed to the next iteration of the loop and start the next demonstration.
* **demo\_sem2**: This semaphore is used to ensure that assistants wait until all students have left the classroom before ending the demonstration. It starts with an initial value of 1. The assistant thread calls **sem\_wait(&demo\_sem2)** before ending the demonstration and **sem\_post(&demo\_sem2)** when all students have left the classroom.
* **mutex**: This semaphore is used as a mutex to ensure mutual exclusion when accessing shared variables. It starts with an initial value of 1. The assistant and student threads use **sem\_wait(&mutex)** to lock the mutex before updating the waiting student and assistant counts and **sem\_post(&mutex)** to release the mutex after updating.

The program then creates **num\_students** student threads and **num\_assistants** assistant threads. It initializes arrays of student and assistant ids and passes them as arguments to the corresponding threads.

The purpose of the **student\_thread** function is to simulate the behavior of a student thread in the classroom. Specifically, it simulates a student entering the classroom, participating in the demonstration, and leaving the classroom. The function starts by printing the thread ID and role (student) using printf to indicate that the student thread wants to enter the classroom. The student thread calls **sem\_wait(&student\_sem)** to wait for the student semaphore. This semaphore controls the number of students allowed to enter the classroom simultaneously. If the semaphore value is greater than zero, the thread can proceed; otherwise, it will be blocked until a signal is received. After waiting for the student semaphore, the thread prints a message indicating that it has entered the classroom.

Update waiting students count: The thread then calls **sem\_wait(&mutex)** to acquire the mutex semaphore, which ensures that the increment operations on the shared **waiting\_students** variable are done atomically. It increments the **waiting\_students** count by one. If there are at least two waiting students (**waiting\_students >= 2**) and at least one waiting assistant (**waiting\_assistants > 0**), the thread can proceed with the demonstration. It decrements **waiting\_students** by two and **waiting\_assistants** by one. It then signals the assistant thread to proceed by calling **sem\_post(&assistant\_sem)** and **sem\_post(&demo\_sem).** The thread calls **sem\_post(&mutex)** to release the mutex semaphore, allowing other threads to access the shared variables. The thread prints a message indicating that it is now participating in the demonstration. Signal demo\_sem2 if no waiting students: If the **waiting\_students** count is less than or equal to zero, meaning there are no more waiting students, the thread signals the assistant thread by calling **sem\_post(&demo\_sem2)**. The thread calls **sem\_wait(&demo\_sem)** to wait for the assistant thread to signal the start of the demonstration. It then calls **sem\_post(&demo\_sem)** to signal the end of its participation. Finally, it calls **sem\_wait(&demo\_sem)** again to wait for the assistant thread to complete its participation.

After the assistant thread completes its participation, the student thread prints a message indicating that it has left the classroom. The thread calls **sem\_post(&student\_sem)** to release the student semaphore, allowing other students to enter the classroom. It also calls **sem\_post(&demo\_sem2)** to signal that it has left the classroom.

Lastly, the purpose of assistant\_thread is to simulate the behavior of an assistant thread in the classroom. The assistant thread is responsible for participating in the demonstration sessions with the students. The function starts by printing the thread ID and role (assistant) using printf to indicate that the assistant thread has entered the classroom. Then, The assistant thread calls **sem\_wait(&mutex)** to acquire the mutex semaphore, which locks access to the shared variables. The thread increments the **waiting\_assistants** count by one and if there are at least two waiting students (**waiting\_students >= 2**), the thread can proceed with the demonstration. It decrements **waiting\_students** by two and **waiting\_assistants** by one. It then signals the student thread to proceed by calling **sem\_post(&assistant\_sem)** and **sem\_post(&demo\_sem).** Moving on, the thread calls **sem\_post(&mutex)** to release the mutex semaphore, allowing other threads to access the shared variables. The thread waits for the student thread to signal the start of the demonstration by calling **sem\_wait(&assistant\_sem),** it then prints a message indicating that it is now participating in the demonstration. After that, it signals the student thread that it has completed its participation by calling **sem\_post(&assistant\_sem).** Then, the thread waits for the **demo\_sem2** semaphore signal, indicating that all students have left the classroom. It waits here to ensure synchronization with the students before ending the demonstration. After receiving the **demo\_sem2** signal, the thread prints a message indicating that the demonstration is over. The thread signals the end of the demonstration by calling **sem\_post(&demo\_sem)** and waits for the student semaphore twice using **sem\_wait(&student\_sem)** to ensure that two students have left the classroom. Finally, after the students have left, the assistant thread prints a message indicating that it has left the classroom.

After creating all the student threads, the code uses a for loop to join each student thread using **pthread\_join.** This ensures that the main thread waits for all student threads to complete before proceeding. Following the same pattern as the student threads, the code uses another for loop to join each assistant thread using **pthread\_join**. This ensures that the main thread waits for all assistant threads to complete. After all threads have completed, the code calls **sem\_destroy** to destroy the semaphores (**student\_sem, assistant\_sem, demo\_sem, mutex**). This frees any resources associated with the semaphores. Lastly, the code uses printf to print a message indicating that the main thread has terminated.