Design of Sleeping Teaching Assistant

Course: CPSC 351 - Operating System Concepts
Assignment: Programming Assignment – Sleeping Teaching Assistant
Language Used: C/C++
Team Members: Christopher Contreras, Alexie Gonzalez, and Britney Lopez

---

1. Introduction

This design document outlines the implementation of the Sleeping Teaching Assistant (TA) problem using POSIX threads, semaphores, and mutex locks in C/C++. The program aims to simulate a scenario where students require help from a TA, who may be either available, assisting another student, or sleeping when not in demand.

The TA office has limited seats, so students must wait or return later if seats are unavailable. This problem focuses on correct synchronization, including one TA and multiple student threads. Also, make sure for mutilation exclusion and thread safety.

2. Design Goals

- **Process Synchronization**: Proper synchronization of threads to manage shared resources (TA and waiting chairs).
- **Mutual Exclusion**: Ensure that no two students simultaneously occupy the same chair or interact with the TA.
- **Efficient Resource Utilization**: Minimize students' time waiting and maximize TA efficiency by correctly managing available seats and help times.

3. Key Components

1. **TA Thread (`TA_Activity`)**: Handles the TA's behavior, such as sleeping, helping students, and managing available chairs.
2. **Student Threads (`Student_Activity`)**: Each student thread simulates a student arriving to get help from the TA.
3. **Main Function (`main`)**: Initializes semaphores, mutex locks, and threads. It creates and manages the lifecycle of the TA and student threads.

4. Program Flow

4.1 TA_Activity Function

This function defines the TA's behavior:

- **Sleep Mode**: The TA thread initially waits for a student thread signal when no students need help.
- **Student Assistance**: When a student requires help, the TA thread assists that student in sequence based on availability.
- **Chair Management**: The TA ensures students are attended to in the order they arrive, using a queue of available chairs.

The functions use sem_wait and sem_post to sleep and wake up the state of TA, which is used with pthread_mutex_lock and pthread_mutex_unlock. This makes sure no two students can alter the ChairsCount variable simultaneously.

4.2 Student_Activity Function

Each student thread follows this behavior:

- **Check for TA Availability**: If the TA sleeps, the student wakes the TA by posting to `TA_Sleep`.
- **Seating Management**: If the TA is helping another student, the student checks for an available chair. If a chair is available, the student sits down and waits for their turn; otherwise, the student leaves to try again later.
- **Getting Help**: Once the TA is ready, the student is helped, and upon completion, they leave, freeing up a chair.

The `sem_wait`, `sem_post`, and mutex locks used here ensure that no two students occupy the same chair and that the chair count is accurately updated.

4.3 Main Function

The `main` function initializes all semaphores and mutexes:

- **Semaphore Initialization**:
  - `TA_Sleep`: Allows the TA to sleep until a student wakes them.
  - `Chair_Sem[]`: Array of semaphores representing the three available chairs.
  - `Student_Sem`: Allows the TA to signal a student when they are ready to help.
- **Mutex Initialization**:
  - `Chairs_Mutex`: Protects the `ChairsCount` variable from race conditions.
- **Thread Creation**: Creates one TA thread and multiple student threads based on user input (or defaults to five students if not specified).
- **Thread Joining**: Ensures all threads are terminated adequately before freeing resources.

5. Synchronization Mechanism

The program uses the following synchronization primitives:

- **Semaphores**:
  - `TA_Sleep`: Controls when the TA sleeps or is woken up by a student.
  - `Chair_Sem[]`: Allows students to wait for a specific chair if the TA is busy.
  - `Student_Sem`: Used by the TA to signal the next student waiting on a chair.
- **Mutex Locks**:
  - `Chairs_Mutex`: Protects access to the `ChairsCount` variable, ensuring that only one student thread can modify the count.

These primitives prevent race conditions and ensure that only one student can access the TA at a time while also managing the limited seating outside the TA's office.

6. Error-Handling Strategy

The program includes error-checking mechanisms for all system calls involving thread creation, semaphore initialization, and mutex operations. This ensures the program can handle unexpected scenarios gracefully and will output appropriate error messages if any issues arise.

7. Assumptions and Limitations

- The program assumes all students arrive within the same timeframe and will try again later if no chairs are available.
- The implementation does not account for priority or urgency in student requests.
- The program assumes the TA will always be available when at least one student is waiting.

8. Testing Strategy

To ensure correctness:

- **Compile and Run**: The program is compiled using G++ and tested by running with different numbers of students.
- **Output Verification**: The output is reviewed to confirm that students either receive help, are correctly seated, or are redirected to come back later.
- **Stress Testing**: Running the program with many students to ensure that synchronization works under load and no deadlock occurs.

9. Summary

This program leverages POSIX threads, semaphores, and mutexes to model a real-world scenario of a teaching assistant helping students. Proper use of synchronization tools ensures thread safety and efficient handling of shared resources, fulfilling the assignment's requirements.