PA3 Report

Name, Surname: Beren Aydoğan

ID: 31938

Program Overview

Court.h simulates a basketball court where a game only starts with a specified number of players and an optional referee, ensuring that all necessary participants are present before starting a match and that they leave in order when the match concludes. It assumes threads as the players and synchronizes them to order access and protect shared variables. To achieve this, the program utilizes synchronization tools semaphores and barriers.

Class Structure

Private Variables

- playerCount
    - Type: “int”
    - Description: Stores the number of players required to conduct a match.
- currentPlayers
    - Type: “int”
    - Description: Stores the number of players currently on the court.
- leftBeforeReferee
    - Type: “int”
    - Description: Tracks the number of players waiting for the referee to leave.
    - Its default value is “0”.
- hasReferee
    - Type: “bool”
    - Description: Flag indicating whether there is a referee present.
- matchInProgress
    - Type: “bool”
    - Description: Flag indicating whether a match is currently in progress.
    - Its default value is “false”.

- hasRefereeLeft
    - Type: “bool”
    - Description: Flag indicating whether the referee has left the court.
    - Its default value is “false”.
- courtAccess:
    - Type: “sem_t”
    - Description: Semaphore to control access to the court.
- mutex:
    - Type: “sem_t”
    - Description: Binary semaphore to provide atomicity for the “enter” and “leave” methods.
- matchEnd:
    - Type: “sem_t”
    - Description: Semaphore to make sure the players entering the enter method sleep while there is an ongoing match.
- referee:
    - Type: “sem_t”
    - Description: Semaphore to ensure players leave after the referee.
- gameStartBarrier:
    - Type: “pthread_barrier_t”
    - Description: Barrier to synchronize the start of the game.
- refereeID:
    - Type: “pthread_t”
    - Description: Stores the ID of the referee thread if present.

Public Methods

- Constructor
    - Prototype: “Court(int playersNeeded, int refereeNeeded)”
    - Description: Initializes the “Court” with the specified number of players and an optional referee. It sets up the semaphores (via sem_init) and the barrier (via pthread_barrier_init). Throws an exception if the entered arguments are invalid.

    - Parameters:
        - “playersNeeded”: Number of players required for a match
        - “refereeNeeded”: Flag indicating whether a referee is needed (0 or 1).
- Destructor
    - Prototype: “~Court()”
    - Description: Cleans up by destroying semaphores (via sem_destroy) and barriers (via pthread_barrier_destroy) to release resources.
- enter Method
    - Prototype: “void enter()”
    - Description: Handles the entry of players and the referee into the court. It manages the synchronization using semaphores and barriers to ensure that there are only enough threads on the court and that the match starts only when the required number of players and the referee (if needed) are present.
- leave Method
    - Prototype: “void leave()”
    - Description: Manages the exit of players and the referee from the court. It ensures synchronization so that the referee (if present) leaves before all players and that the match ends only after all players and the referee (if present) have left the court.
- play Method
    - Prototype: “void play()”
    - Description: Placeholder method intended to be overridden by test cases.

<u>Pseudo Code Descriptions of Key Methods</u>

Enter Method:

1. Announce the arrival of the thread.
2. If a match is in progress, wait on matchEnd semaphore until the match ends.
3. Wait on courtAccess semaphore to ensure there is space on the court.
4. Acquire mutex to ensure atomicity while modifying shared resources.
5. Increment currentPlayers count.
6. Check if the number of players matches the required amount:

```
        a. If yes:
                i. Set matchInProgress to true.
                ii. Check if a referee is needed
                        1. If yes:
                                a. Set refereeID to the current thread’s ID.
                        2. If no:
                                a. Set refereeID to 0.
                iii. Announce the start of the match.
                iv. Wait on gameStartBarrier to ensure all players start at the same time.
                v. Release the mutex.
        b. If no:
                i. Announce that we are waiting for more players.
                ii. Release the mutex.
```

Leave Method:

```
1. Acquire mutex to ensure atomicity while modifying shared resources.
2. Check if matchInProgress:
        a. If no:
                i. Decrement currentPlayers.
                ii. Announce departure since no match was found.
                iii. Release courtAccess to open a spot on the court.
                iv. Release the mutex.
        b. If yes:
                i. If the thread is the referee:
                        1. Set hasRefereeLeft to true.
                        2. Decrement currentPlayers.
                        3. Release all players waiting on the referee semaphore.
                        4. Release the mutex.
                ii. If the thread is a player:
                        1. If waiting for the referee to leave
                                a. Increment leftBeforeReferee.
```

    b. Release the mutex.
    c. Wait on referee semaphore if the referee has not left.
    d. Once releasing the referee semaphore, acquire the mutex.
2. Decrement currentPlayers.
3. Announce departure.
4. If all players have left:
    a. Set matchInProgress to false.
    b. Announce the end of the match.
    c. Signal end of the match and release courtAccess to open a spot on the court.
    d. Reset match-related variables.
5. Release the mutex.

Detailed Descriptions of Key Methods

Enter Method:

The method begins by initializing a pthread_t variable called “tid” and setting it to the thread ID of the calling thread for further use and prints the arrival of the thread using printf for atomicity. Then, it checks if a match is in progress by evaluating the variable “matchInProgress”.

If there is a match already in progress, it calls sem_wait(&matchEnd) and sleeps until it is signaled.

If there isn’t a match is in progress, the method calls sem_wait(&courtAccess), which decrements the “courtAccess” semaphore if its value is positive. This semaphore controls the number of players and the referee entering the court by blocking the thread if the semaphore's value is zero, ensuring that only a limited number of participants can enter. Next, sem_wait(&mutex) is called to decrement the mutex semaphore, ensuring atomic access to the critical section that follows. Since mutex is a binary semaphore, only one thread can enter before the thread that entered calls sem_post. This prevents race conditions by allowing only one thread to modify shared variables at a time. The “currentPlayers” count is then incremented to reflect the entry of a participant.

If the number of current players reaches the required count (including the referee if present), “matchInProgress” is set to true, if a referee is needed the last entering player’s ID is stored as the referee's thread ID in “refereeID” else it is set to 0, and pthread_barrier_wait(&gameStartBarrier) is called. This barrier ensures that the last player is waited. Finally, sem_post(&mutex) releases the mutex.

If the count is not reached yet, the thread prints a message related to it and also releases the mutex via sem_post(&mutex).

Leave Method:

The method begins by initializing a pthread_t variable called “tid” and setting it to the thread id of the calling thread for further use. Then, it calls sem_wait(&mutex), obtaining mutex semaphore to ensure atomic access to the critical section and protecting the modification of shared variables. Inside the critical section, the method checks if a match is in progress by evaluating the variable “matchInProgress”.

If a match is not in progress, the thread decrements “currentPlayers”, prints a message informing that it couldn’t find a match, releases access to court by calling sem_post(&courtAccess) to allow another player to enter the court if any want to, and releases the mutex by calling sem_post(&mutex).

If a match is in progress and the current thread is the referee “pthread_equal(tid, refereeID)”, the method sets “hasRefereeLeft” to true to make sure the other threads are aware of the fact that the referee left so they can exit freely and decrements the “currentPlayers” count. Then, it releases any players waiting for the referee to leave by signaling to the referee semaphore for each waiting player via sem_post(&referee) by using the variable “leftBeforeReferee” which keeps the count of the threads who tried to leave before the referee. Finally, it calls sem_post(&mutex) and releases the mutex.

If a match is in progress and the current thread is not the referee and the referee has not left yet, the method increments “leftBeforeReferee” and releases the mutex by posting to sem_post(&mutex), allowing other threads to enter the critical section. The thread then waits for the referee to leave by calling sem_wait(&referee) and sleeping. When it is released by sem_post(&referee) issued by the referee, it re-acquires the mutex with another sem_wait(&mutex) call. Then this thread (or any other thread that came without being blocked on the referee semaphore since hasRefereeLeft is set to false), it decremenst "currentPlayers" and prints a message that indicates it is leaving. If all players have left the court (currentPlayers == 0), the method sets "matchInProgress" to false and prints a message to indicate the end of the match. It then posts to the "matchEnd" and "courtAccess" semaphores to signal the end of the match to the waiting threads and release access to the court for waiting threads. Finally, the method resets "leftBeforeReferee" and "hasRefereeLeft" to their initial states and releases the mutex with sem_post(&mutex).

Synchronization Mechanisms Used

Semaphores:

- courtAccess: Controls access to the court, ensuring that no more than the required number of players and the referee are on the court at the same time.
  - Initial value of the semaphore is set to the number of players in the game plus the referee, if it is required.
- mutex: Provides mutual exclusion for the modification of shared state variables, ensuring that these operations are atomic and prevent race conditions.
  - Used to protect the bodies of enter() and leave() methods. Ensures the atomicity of the modification of shared variables like currentPlayers, matchInProgress, leftBeforeReferee etc.
  - Implemented as a binary semaphore to be used as a regular lock by setting the initial value of the semaphore to 1.
- matchEnd: Used to block players from entering the court when a match is already in progress.
  - Initial value of the semaphore is set to 0 to block anyone and to release them by sem_post().
- referee: Used to ensure players wait for the referee to leave first if there is one.

  - Initial value of the semaphore is set to 0 to block anyone and to release them by sem_post().

Barrier:

- gameStartBarrier: Ensures the last player (the referee, if present) is present to start the game simultaneously.

Correctness Criteria

Invalid Arguments: In any case of invalid arguments, an exception is thrown, indicating that there is an issue.

Entering the Court: All players can only enter if there is space in the court and there is no ongoing match. The implementation prevents more players than specified from entering the court simultaneously. The use of the "courtAccess" semaphore ensures that entry is controlled, preventing more than the allowed number of players from entering the court. If there is an ongoing match, any thread that wants to access the court is blocked by the utilization of the "matchEnd" semaphore. Then, in the leave method, the desired number of players is signaled via sem_post in the "matchEnd" and "courtAccess" semaphore to control court access.

Starting the Game: Matches start only when the required number of players (including a referee, if needed) is present. The barrier ensures that no player starts before the others by waiting for the very last player.

Atomicity: The implementation also ensures the atomicity of any operation where atomicity is desired. The use of mutex semaphore makes sure that the modification of shared variables like "currentPlayers", "matchInProgress", "leftBeforeReferee" etc. is atomic. Also, the use of printf() instead of cout stream guarantees that there are not any output mix-ups.

Leaving the Court: Players leave in a specific order, with the referee (if present) leaving first. To ensure that no player leaves before the referee, in case it exists, the "referee" semaphore is used to block any player that intends to leave before the referee. Upon the leaving of all players, the last player notifies others that the court is available. "matchEnd" and "courtAccess" semaphores are signaled to make sure to wake up any blocked threads and get an appropriate number of players into the court.

No Busy-Waiting: Using semaphores and barriers instead of regular locks helps prevent busy-waiting.