Sabanci University

Operating Systems

CS 307

Programming Assignment - 3: Synchronization Slam Dunk: Implementing a Basketball Court with Semaphores

Release Date: 18 April 2025
Deadline: 28 April 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 a basketball court organization. Users of the court (players and referees) will be represented by threads and you have to synchronize them using semaphores on the usage of the shared basketball court resource.

Users arrive the court hoping to play a match. They will enter the court if the court is not full and a match is not already going on. For the sake of simplicity, we will assume there is a fixed agreed number of players for a match to begin, and the players will not start a match if there are not enough players. Depending on the prior agreement again, one of the players might be a referee.

When a match begins, none of the players inside can leave before the match ends, and they will not let any new players inside. When the match ends and a referee is present, first they need to announce the match is over. After this the players will start to leave as well. If there is no referee for that match the players will agree upon themselves and leave without an order. After everyone else left, the last player also needs to inform waiting players, so that they can enter.

2 Problem Description

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

In our simulation the players will be represented by threads. These threads will call `enter`, `play` 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 players for a match inside the court i.e., returned from the `enter` method, players must start the match.

- There can not be more players in the court than the number required for a match.
- `play` method will represent a player passing some time in the court. If there are not enough players for a match, we can interpret time spent in this method like a free practice. When there are enough players and a match starts, we can think of remaining time spent in this method as playing a match. If a player is done playing (returned form `play` and called `leave` and a match has not started, it must leave without playing in a match.
- If there is a referee, players must not return from the `leave` method until the referee announces that the game is over. Otherwise, players can leave freely.
- When a match finishes, the last player to leave must notify the waiting players that are blocked in the `enter` method so that they can return from this method and enter the court.
- When there is a match, no one can enter the court before the match finishes and all players and (if exists) the referee involved in the match leaves the court.

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, `enter` 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 player thread please read the next section. You can also find some sample runs at 10.

3 Basketball Court Class

In your simulation you will implement the `Court` class. In its shared state it must have some fields to keep number of players, whether there is a referee and who is the referee if exists, whether there is a match 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: `enter`, `play` and `leave`. There will also be another function, `play`, that will be declared inside test cases.

Each player 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 `enter`, `play` and `leave` methods in this order.

3.1 Constructor

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

$ Court(4,1)

In this case, 4 players are needed for a match and there will be a referee in the matches. So, in total, 5 people must be in a court to start a match.

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

3.2 Enter Method

Below, you can find a typical behaviour of a player's enter function:

- It first prints: `Thread ID:` $<tid>$`, I have arrived at the court.`
- It checks if there is a match already being played. If so, it blocks, otherwise it enters the court.
  - You can keep a boolean variable to check if a match is being played or not. You can use a semaphore to block threads in `enter` method when the court is full and a match is ongoing.
- When it enters, it checks how many players are in the court. There are two scenarios depending on the referee option. If a match does not require a referee, it checks if, including itself, there are enough players to play a match. If there has to be a referee, it checks if, excluding itself, there are enough players to play a match. In this case, it is assigned as a referee. If one of these conditions hold, it starts the match and it prints

  `Thread ID:` $<tid>$`, There are enough players, starting a match.`

  Otherwise it prints

  `Thread ID:` $<tid>$`, There are only` $<n>$ `players, passing some time.`

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

3.3 Play Method

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

3.4 Leave Method

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

- If a match has not started it prints

  ```
  Thread ID: < tid >, I was not able to find a match
  and I have to leave.
  ```

  and leaves the court. Next items are for the case where a match is played.

- If there is the referee, and this player is assigned as a referee it prints

  ```
  Thread ID: < tid >, I am the referee and now, match
  is over.  I am leaving.
  ```

- If this player is not the referee and the referee has already left it prints

  ```
  Thread ID: < tid >, I am a player and now, I am leaving.
  ```

- If there is not any referee in the match, it prints

  ```
  Thread ID: < tid >, I am a player and now, I am leaving.
  ```

- The last player that leaves the court also prints

  ```
  Thread ID: < tid >, everybody left, letting any waiting
  people know.
  ```

  as its last statement and leaves the court.

4 General Considerations and Corner Cases

- 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.

- When there is a game on the court, let new players in the court after the last player in the match leaves by using semaphore API methods.
- Consider using a barrier to pick the referee and printing its output. i.e: when a match is formed, players should wait in the barrier until a referee prints its output
- 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 games on the court players could enter and exit freely without blocking.
- Even though a referee is picked for every game that is formed if referee is present, it is also referred as a player in the document in general to prevent confusion.
- In order to select the referee when a game is formed, you might consider using a special field in your class of type `pthread_t` and set it to the referee thread's thread ID when You pick the referee in the enter method. You might need this information in the `leave` method because the referee prints a distinct line.
- In `leave` method, in order to understand if the current thread is the referee, 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(player) 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 player 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.

- *player_count*: The number of players in a match, excluding the referee.
- *all_count*: The number of players in a match, including the referee.
- *num_games*: The number of games formed
- *init*: The string `"Thread ID:` $< tid >$`, I have arrived at the court."`
- *enter_passtime*: The string `Thread ID:` $< tid >$`, There are only` $< x >$ `players, passing some time.`
- *enter_game*: The string `Thread ID:` $< tid >$`, There are enough players, starting a match.`
- *everybody_left*: The string `Thread ID:` $< tid >$`, everybody left, letting any waiting people know.`
- *referee_leaving*: The string `Thread ID:` $< tid >$`, I am the referee and now, match is over.  I am leaving.`
- *player_leaving*: The string `Thread ID:` $< tid >$`, I am a player and now, I am leaving.`
- *no_game*: The string `Thread ID:` $< tid >$`, I was not able to find a match and I have to leave.`

Then, for any execution of your program with valid inputs, the following conditions must be satisfied:

- There must be exactly *total_num* times *init* strings printed to the console
- There must be exactly $total\_num - num\_games$ times *enter_passtime* strings printed to the console
- There must be exactly *num_games* times *enter_game* strings printed to the console
- There must be exactly *num_games* times *everybody_left* strings printed to the console
- There must be exactly *num_games* times *referee_leaving* strings printed to the console

- There must be exactly $num\_games * match\_size$ times $player\_leaving$ strings printed to the console

- There must be exactly $total\_num - (num\_games * (all\_count))$ times no_game strings printed to the console

- For each thread $init$, $enter\_passtime$ or $enter\_game$, $referee\_leaving$ or $player\_leaving$(if game exists), no_game(if no game exists) must be printed to the console in this order

- For each game after $enter\_game$ string, $referee\_leaving$ must be printed to the console before any $player\_leaving$

- For each game after $enter\_game$, $init$ strings might be printed in-between games but, no $enter\_passtime$ nor $enter\_game$ can be printed to the console before $everyone\_left$ 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 the case that there are more than one matches like eight threads or twelve threads when the match size is four.
- Your program should terminate properly, you will lose points if your program can not 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.

- **Court.h**: Your **C++** implementation of the header file where your `constructor`, `enter` and `leave` methods are implemented.
- **report.pdf**: In your report, you must present the flow of your `enter` 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”. The files that you submit should **NOT** contain your name or your id. 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. You are expected to complete your submissions until 28 April 2025, 23.55 .

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`, `enter` 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 Matches Case (15 pts)**
  - **Without Referees (10 pts):** When there are no referees if the players are never able to play a match, your outputs fit our correctness criteria.
  - **With Referees (5 pts):** When there are referees if the players are never able to play a match, your outputs fit our correctness criteria.
- **At Most 1 Match Case (15 pts)**
  - **Without Referees (10 pts):** When there are no referees if the players are able to play at most one match, your outputs fit our correctness criteria.
  - **With Referees (5 pts):** When there are referees if the players are able to play at most one match, your outputs fit our correctness criteria.

- **Multiple Matches Case (15 pts)**
  - **Without Referees (10 pts):** When there are no referees if the players play multiple matches, your outputs fit our correctness criteria.
  - **With Referees (5 pts):** When there are referees if the players play multiple matches, your outputs fit our correctness criteria.
- **Last player in a match informs others(10 pts):** When a match ends, and all the players left, the last player prints 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 player comes when there is a match already being played, it has to wait in a loop until the match 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 'court_test' or 'court_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:

```
$ ./court_test x y z
```

i.e. if file name is court_test2_4_5_1.txt , then in your command line you should run

```
$ ./court_test2 4 5 1
```