Sabanci University

Operating Systems

CS 307

Programming Assignment - 3: Synchronization Sightseeing: Managing a Tourist Attraction with Semaphores

Release Date: 22 November 2024
Deadline: 2 December 2024 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 organizing a tourist attraction site. Visitors and guides will be represented by threads and you have to synchronize them using semaphores on the usage of the shared resource of the attraction site.

Visitors arrive at the landmark, hoping to tour and take photographs. They are allowed into the area if it is not already full and if there is no active tour taking place. For simplicity, we will assume that a tour can only begin when there is a fixed group of visitors ready to participate. Additionally, depending on prior arrangements, one of the visitors may be a guide.

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

2 Problem Description

In this programming assignment you are tasked with preparing a C++ header file named `"Tour.h"` implementing the `Tour` 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 visitors 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 visitors for a tour inside the area i.e., returned from the `arrive` method, visitors must start the tour.

- There can not be more visitors in the area than the number required for a tour.
- `start` method will represent a visitor starts passing some time in the landmark. If there are not enough visitors for a tour, we can interpret time spent in this method like spending some time at the entrance or sitting at a coffee shop while waiting. When there are enough visitors and a tour starts, the time spent in this method represents an organized tour experience. If a visitor has finished exploring (returned from `start` and called `leave` and a tour has not started, it must leave without participating in a tour.
- If there is a guide, visitors must not return from the `leave` method until the guide announces that the tour has ended. Otherwise, visitors can leave freely.
- When a tour ends, the last visitor to leave must notify the waiting visitors that are blocked in the `arrive` method so that they can return from this method and enter the area.
- While a tour is in progress, no one can enter the landmark before the tour ends and all visitors and (if exists) the guide involved in the tour leaves the area.

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

3 Tour Class

In your simulation you will implement the `Tour` class. In its shared state it must have some fields to keep number of visitors, whether there is a guide and who is the guide if exists, whether there is a tour 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 visitor 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 visitors are needed to start a tour and the second argument indicates whether there will be a guide in the tour. The second parameter can be either 0 or 1. An example call of it is as follows:

$ Tour(4,1)

In this case, 4 visitors are needed for a tour and there will be a guide in the tour. So, in total, 5 people must be in a location to start a tour.

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 Arrive Method

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

- It first prints: `Thread ID:` $< tid >$ `| Status: Arrived at the location.`
- It checks if there is an active tour already in progress. If so, it blocks, otherwise it enters the attraction site.
  - You can keep a boolean variable to check if a tour is in progress or not. You can use a semaphore to block threads in `arrive` method when the site is full and a tour is in ongoing.
- When it arrives, it checks how many visitors are in the site. There are two scenarios depending on the guide option. If a tour does not require a guide, it checks if, including itself, there are enough visitors to have a tour. If there has to be a guide, it checks if, excluding itself, there are enough people to have a tour. In this case, it is assigned as a guide. If one of these conditions hold, it starts the tour and it prints

  `Thread ID:` $< tid >$ `| Status: There are enough visitors, the tour is starting.`

  Otherwise it prints

  `Thread ID:` $< tid >$ `| Status: Only` $< n >$ `visitors inside, starting solo shots.`

  and passes some time waiting for another visitor to start the tour.
- If you are counting the number of people inside the location, this number must be incremented when someone is entering the attraction site. 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 visitor passes its time in the location. You can assume that if the tour started when a visitor was waiting, it automatically joins the tour. 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 tour has not started it prints

  `Thread ID:` $< tid >$ `| Status: My camera ran out of memory while waiting, I am leaving.`

  and leaves the attraction site. Next items apply if a tour is in progress.

- If there is the tour guide, and this visitor is assigned as a tour guide it prints

  `Thread ID:` $< tid >$ `| Status: Tour guide speaking, the tour is over.`

- If this visitor is not the tour guide and the tour guide has already left, it prints

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

- If there is not any tour guide in the tour, it prints

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

- The last visitor that leaves the tour also prints

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

  as its last statement and leaves the attraction site.

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 a tour is in progress at the attraction site, allow new visitors to enter only after the last visitor in the tour leaves, by using semaphore API methods.
- Consider using a barrier to pick the guide and printing its output. i.e: when a tour group is formed, visitors should wait in the barrier until a guide 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 tours on the site visitors could arrive and leave freely without blocking.
- Even though a guide is picked for every tour that is formed if guide is present, it is also referred as a visitor in the document in general to prevent confusion.
- In order to select the guide when a tour group is formed, you might consider using a special field in your class of type `pthread_t` and set it to the guide thread's thread ID when You pick the guide in the arrive method. You might need this information in the `leave` method because the guide prints a distinct line.
- In `leave` method, in order to understand if the current thread is the guide, 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(visitor) 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 visitor 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.

- *visitor_count*: The number of visitors in a tour, excluding the tour guide.
- *all_count*: The number of visitors in a tour, including the tour guide.
- *num_tours*: The number of tours done
- *init*: The string "`Thread ID:` $< tid >$ `| Status: Arrived at the location.`"
- *enter_passtime*: The string `Thread ID:` $< tid >$ `| Status: Only` $< x >$ `visitors inside, taking solo shots.`
- *enter_tour*: The string `Thread ID:` $< tid >$ `| Status: There are enough visitors, the tour is starting.`
- *everybody_left*: The string `Thread ID:` $< tid >$ `| Status: All visitors have left, the new visitors can come.`
- *tourguide_leaving*: The string `Thread ID:` $< tid >$ `| Status: Tour guide speaking, the tour is over.`
- *visitor_leaving*: The string `Thread ID:` $< tid >$ `| Status: I am a visitor and I am leaving.`
- *no_tour*: The string `Thread ID:` $< tid >$ `| Status: My camera ran out of memory while waiting, I am leaving.`

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 $total\_num - num\_tours$ times *enter_passtime* strings printed to the console
- There must be exactly *num_tours* times *enter_tour* strings printed to the console
- There must be exactly *num_tours* times *everybody_left* strings printed to the console

- There must be exactly num_tours times $tourguide\_leaving$ strings printed to the console
- There must be exactly $num\_tours * tour\_size$ times $visitor\_leaving$ strings printed to the console
- There must be exactly $total\_num - (num\_tours * (all\_count))$ times no_tour strings printed to the console
- For each thread $init$, $enter\_passtime$ or $enter\_tour$, $tourguide\_leaving$ or $visitor\_leaving$(if tour exists), no_tour(if no tour exists) must be printed to the console in this order
- For each tour after $enter\_tour$ string, $tourguide\_leaving$ must be printed to the console before any $visitor\_leaving$
- For each tour after $enter\_tour$, $init$ strings might be printed in-between games but, no $enter\_passtime$ nor $enter\_tour$ 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 cases where there are multiple tours, such as eight threads or twelve threads when the tour 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.

- **Tour.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". 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 2 December 2024, 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`, `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 Tours Case (15 pts)**
  - **Without Guides (10 pts):** When there are no guides, if the visitors are never able to participate in a tour, your outputs fit our correctness criteria.
  - **With Guides (5 pts):** When there are guides, if the visitors are never able to participate in a tour, your outputs fit our correctness criteria.
- **At Most 1 Tour Case (15 pts)**
  - **Without Guides (10 pts):** When there are no guides, if the visitors are able to participate in at most one tour, your outputs fit our correctness criteria.

  - **With Guides (5 pts):** When there are guides, if the visitors are able to participate in at most one tour, your outputs fit our correctness criteria.

- **Multiple Tours Case (15 pts)**
  - **Without Guides (10 pts):** When there are no guides, if the visitors participate in multiple tours, your outputs fit our correctness criteria.
  - **With Guides (5 pts):** When there are guides, if the visitors participate in multiple tours, your outputs fit our correctness criteria.

- **Last visitor in a tour informs others (10 pts):** When a tour ends, and all the visitors have left, the last visitor 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 visitor comes when there is a tour already in progress, it has to wait in a loop until the tour 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 ‘tour_test1‘ or ‘tour_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:

```
$ ./tour_test1 x y z
```

i.e. if file name is tour_test1_4_4_1.txt , then in your comment line you should run

```
$ ./tour_test1 4 4 1
```