
# 1 Introduction

In this project, you are going to implement a theater ticket reservation system simulation. This program will be run as multi-threaded by considering synchronization and data consistency issues. The scenario of the simulation is that the clients arrive at the ticket office for seat reservation and tellers reserve the seats for them. The requirements, operations, and flow of the program are as follows:

1. There are 3 different theater halls in the system with the given available number of seats:
   (a) ODA TIYATROSU, 60 seats
   (b) USKUDAR STUDYO SAHNE, 80 seats
   (c) KUCUK SAHNE, 200 seats
2. Each simulation will run on a specific theater hall determined by the configuration file.
3. Each theater hall has a ticket office with 3 tellers; A, B, and C. Tellers make reservations on behalf of the clients.
4. Each ticket office has a common queue, so there are no separate queues for each teller. Once a teller becomes available, he/she serves the client at the head of the queue.
5. Tellers must be present in the ticket office before the first clients' arrival.
6. Each client provides four fields of information (via the configuration file): To whose name will the ticket be registered, arrival time, service time, and the desired seat number.
7. The number of clients is specified by the input configuration file. You can assume that the number of clients is always a positive integer value that does not exceed 300.
8. The clients arrive at the theater hall sequentially but wait until his/her arrival time.
9. After waiting some time, the client goes to the ticket office and waits for his/her turn in the queue. Then, he/she goes to one teller and requests the seat stated in the configuration file for the reservation. Please check Figure 1 to see the simulation in detail.
10. If a requested seat is available, the teller successfully handles the client's request within the client's service time. Then, the teller marks the seat as reserved and gives a ticket to the client.
11. If the requested seat is not available, the teller checks the currently available seats and reserves the seat with the lowest number, then gives a ticket. If the teller cannot find such a seat, the client leaves the ticket office.
12. The important thing here is that you need to consider conditions when two different clients request the same seat at the same time or two tellers try to reserve the same seat at the same time.

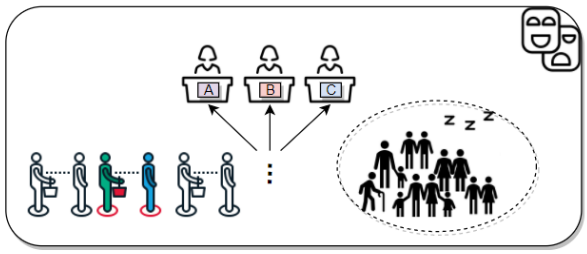


Figure 1: Big picture.

The project will be evaluated automatically in the Linux environment (Ubuntu Version 20.04.1) with a gcc/g++ compiler (Version 9.3.0). Please follow all the requirements specified in the description. Your submissions will be compiled and tested via automatic scripts. Therefore, it is crucial that you follow the protocol defined in the project document. Failure in the compilation and/or execution is your responsibility. You should use the file names, parameters, etc. as mentioned in the project specifications.

# 2 Details of Implementation

You need to use POSIX threads provided by Linux-based systems or macOS for the implementation of this project in order to create a multi-threaded environment (Windows is not allowed). To address the synchronization issues, you can use mutex and semaphores.

## 2.1 Configuration File

All the information required for the simulation to progress is in the configuration file. Your program should take the configuration file as input.

- The first line of the configuration file represents the *Name of the Theater Hall* (i.e., OdaTiyatrosu, UskudarStudyoSahne, and KucukSahne).
- The second line represents the *total number of clients* to be simulated.
- The rest of the configuration file represents the clients. The order of the lines should be aligned with the order of client thread creation. A particular line includes the following information about a client that arrives for a reservation: <client name, arrival time, service time, seat number>. The format of the configuration file is fixed. For example, there will be no white-spaces within lines, and there will be always $NumberOfClients + 2$ lines in the configuration file.

## 2.2 Client-Teller Threads

When the program starts executing, the clients start arriving. The tellers are created in advance, and they are ready before the clients' arrival.

- The main thread should create the client threads sequentially. In other words, the client threads are created one by one, without any waiting time in between. However, after creating a client thread, the client should sleep (or wait) until his/her arrival time.

- In the ticket office, a newly arriving client should enter the queue and wait until the requests of all previous clients in the queue are completed, and the client at the head of the queue is dispatched to the first teller that becomes available. Therefore, the queue mechanism of a ticket office should be implemented as a FIFO queue. (Hint: The mutex unlock operation behaves like FIFO queue in Linux and macOS pthread implementation. Therefore, you don't need to implement an extra linked list or similar structure to imitate the FIFO queue functionality.)
- The tellers should serve the clients in order. If all tellers are available, Teller A gets the first client, Teller B gets the second client, and Teller C gets the third client from the head of the queue.
- After the creation of the client thread and the sleep duration, the client waits for his/her turn and when it's his/her turn, the client selects a seat as stated in the configuration file. Teller thread handles the request of the client, reserves the seat, and sleeps for the client's service time. After sleeping period, client thread exits. If such a seat cannot be reserved, the client thread exits after sleeping period without reservation. (Teller's sleeping period can be considered as ticket preparation time, whether a reservation successful or not Teller thread sleeps before client's leave.)
- The main thread waits until no clients are left to be served.
- The representation of the seats or the variable type is up to you, there is no restriction for this part. Note that the seat number starts from 1, you should not create Seat 0.
- You should avoid implementing the whole program as a single critical section. In order not to restrict the functionality of the program, different critical sections should be implemented as it should be. Of course, it is possible to implement the whole program as a single critical section, but it does not provide the necessary functionality. So this type of implementation is forbidden.

## 2.3 Example Scenarios

Some example scenarios are listed below:

- **Example case 1:** Suppose that Alice arrives at the theater hall with an arrival time of 10 milliseconds, service time of 5 milliseconds, and a request for Seat 23. After waiting for 10 milliseconds in the hall, she goes to the ticket office and gets in the line. Let's say she is the first client in the queue, thus, makes her request from the first unoccupied teller immediately, which is Teller A. Teller A checks Seat 23's availability, and if it is available, and there is no critical section issue, Teller A marks Seat 23 as reserved by Alice. Then, Teller A sleeps for 5 milliseconds, which also makes Alice sleep because she is being served by Teller A. Finally, Teller A prints her ticket as
  ***Alice requests seat 23, reserves seat 23. Signed by Teller A***. Note that, ticket information should be logged to the output file.
- **Example case 2:** Suppose that Bob arrives at the line and goes to Teller B to request Seat 6. While making his reservation he gets an urgent call and leaves the counter for a second. In the meantime, Jane arrives at Teller C to request Seat 6. Since there is an ongoing operation for Seat 6, Teller C cannot reserve Seat 6 for Jane and reserves another seat for her. So, even if a client is preempted from the CPU before the reservation, you should handle the case and do not allow other clients to get a previously chosen seat.

- **Example case 3:** Suppose that Harry gets in the line as the 4th client. The three clients who have arrived earlier occupy the tellers one by one, that is, Teller A gets the first client at the head of the queue, Teller B gets the second client, and Teller C gets the third client at the queue. Whoever finishes the reservation first handles Harry's request. However, if Teller A and B finish their jobs at the same time, Teller A gets the client. You should handle the case when two tellers are available at the same time and a client inaccurately selects both of them or more than one customer arrives at one the counter at the same time.
- **Example case 4:** Suppose that Ron arrives at the line and goes to Teller C to request Seat 23. Teller C cannot reserve seat 23 since it is already reserved by Alice. Thus, Teller C reserves Seat 1 for Ron, since it is the available seat with the lowest number. Ron gets the ticket as ***Ron requests seat 23, reserves seat 1. Signed by Teller C.***
- **Example case 5:** Suppose that Hermione123 arrives at the line and goes to Teller A to request Seat 2. Teller A checks the seats and cannot find an available one. Teller A prints a message as ***Hermione123 requests seat 2, reserves None. Signed by Teller A.***
- **Example Case 6:** Suppose that Jim's arrival time is 15, and Pam's arrival time is 20. Jim and Pam enter the theater hall and sleep 15 and 20 milliseconds respectively. After 15 milliseconds, Jim goes to the ticket office and after 5 milliseconds Pam also goes to the ticket office. So, both threads are created immediately but they should wait for some time before their executions.

### 2.4 Logging of Operations

During the reservation and progress of the program, you should log the reservations to an external file. The path for the output file will be given as an argument to your executable.

- Once the system starts, ***Welcome to the Sync-Ticket!*** should be written by the **main thread** of the program to the output file.
- When teller threads are created, they log their names as follows: ***Teller A has arrived.***, ***Teller B has arrived.***, and ***Teller C has arrived.***.
- After completing a reservation, teller threads log the ticket details to the output file as follows: ***<X> requests seat <Y>, reserves <Y/Z/None>. Signed by Teller <A/B/C>***.
- The program should continue to operate until all clients received service. The main thread should write ***All clients received service.*** to the output file as the last line.
- Pay attention to the synchronization while writing to the same file.

The rest of the implementation is up to you. You can use your own design as long as it provides the necessary functionality, does not conflict with the forbidden cases like defining the whole program as a single critical section, and satisfies the specified format restrictions.

## 3 Input & Output

Your code must read the path of the input and output files from the command line. You should create a a **Makefile** to create the executable (cmake files are not accepted). The name of the executable should be **simulation.o**. We will run your code as follows:

```
> make
> ./simulation.o configuration_path output_path
```

## 3.1 Input

The file structure for the evaluation is provided below. Please design your code and zipped file accordingly. We'll provide you absolute paths, so you do not need to create directories except for the *src* file. Please place all the files in a folder (named as <YOUR_ID>), then zip it (<YOUR_ID>).

```
evalution_script.py
testcases
    inputs
        configuration_file_1.txt
        configuration_file_2.txt
submissions
    2018400XXX.zip
    2018400XXX
        src
            code.c/code.cpp
            anothercode.c/anothercode.cpp
            Makefile
        testcases
            outputs
                configuration_file_1.txt
                configuration_file_2.txt
```

## 3.2 Output

Below you can see an expected output file for the given configuration file.

| OdaTiyatrosu<br>10 |
|---|
| Client1,10,50,5 |
| Client2,20,20,1 |
| Client3,30,20,5 |
| Client4,45,10,1 |
| Client5,65,10,5 |
| Client6,70,10,10 |
| Client7,72,10,10 |
| Client8,100,20,10 |
| Client9,95,20,10 |
| Client10,90,20,10 |

Table 1: Sample configuration file for *Oda Tiyatrosu* with 10 clients.

| Welcome to the Sync-Ticket! |
|---|
| Teller A has arrived. |
| Teller B has arrived. |
| Teller C has arrived. |
| Client2 requests seat 1, reserves seat 1. Signed by Teller B. |
| Client3 requests seat 5, reserves seat 2. Signed by Teller C. |
| Client4 requests seat 1, reserves seat 3. Signed by Teller B. |
| Client1 requests seat 5, reserves seat 5. Signed by Teller A. |
| Client5 requests seat 5, reserves seat 4. Signed by Teller A. |
| Client6 requests seat 10, reserves seat 10. Signed by Teller B. |
| Client7 requests seat 10, reserves seat 6. Signed by Teller C. |
| Client10 requests seat 10, reserves seat 7. Signed by Teller A. |
| Client9 requests seat 10, reserves seat 8. Signed by Teller B |
| Client8 requests seat 10, reserves seat 9. Signed by Teller C. |
| All clients received service. |

Table 2: Expected output of the simulation for the sample configuration file.

# 4 Grading

Corresponding points are **tentatively** specified in parentheses.

1. **Code Documentation:** Explain your code in your own sentences and provide additional information about you and the project. *(10 pts)*
2. **Coding & Implementation:** Try not to conflict with the provided rules. *(40 pts)*
3. **Test Cases:** Your program will be tested using several test cases. *(40 pts)*
4. **Auto-runnable submission:** Submit your code that runs smoothly with the specified commands on the Linux environment. Each re-submission deduces 5 points. *(10 pts)*

# 5 Submission

Submissions will be through Moodle. Submit a single .zip file named with your student ID (e.g. 2018400XXX.zip). The zipped project should contain code files and a Makefile to create the executable. Please pay attention to the file naming. Note that your project will be inspected for plagiarism with previous years' materials as well as this year's. Any sign of **plagiarism** will be **penalized**.