CmpE 322 – Operating Systems

Project#2 - Flight Reservation System Simulator

Due Date: Dec 30, 2018 23:55

1 Introduction

In this project, you are going to implement a simple flight reservation system. This program will be run as multi-threaded by taking the synchronization issues into consideration. The scenario of the flight reservation system is that the clients arrive for seat reservation, and the program reserves the seats for them. The requirements, operations, and flow of the program are as follows:

1. The program will take an integer as argument (e.g., "./a.out 60"), between 50 and 100 (including both 50 and 100). This value represents the total number of seats that are available within the plane.
2. Upon execution, the program generates clients **as separate threads**, one client for each seat. The client threads are created by the main thread sequentially, without any waiting time. However, each client thread sleeps for a random time (between 50-200 milliseconds, including both 50 and 200) upon creation. Then, the client thread can start execution.
3. A server thread is created immediately when a client arrives in order to provide service to it.
4. After the creation of a client-server pair and random sleep time, the client requests a random seat for reservation among the available ones at that time.
5. If that request is successfully handled by the server thread, the seat is marked as reserved, by noting the client number (such as: Client1, Client39 etc. Note that client numbers are sorted by their arrival times.)
6. If the requested seat is not available, the client generates another request, again with a random selection of seat until it reserves one.
7. The main thread waits until all of the seats are reserved and it prints out the seat-ID pairs for each seat within the plane.
8. The important thing here is that you need to **consider the conditions when two different clients request the same seat at the same time**.

You need to implement this program in C/C++ and it should be **compilable by gcc/g++**. The detailed list of the submission is available in Section 3.

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). In order to address the synchronization issues, you can use mutex, semaphores, or both.

As stated in the previous section, the executable should take an integer argument that varies between 50 and 100, including both. This argument represents the number of total seats that are available within the plane. The representation of the seats or the variable type is up to you, there is no restriction for this part.

When the program starts executing, the clients start to arrive. Each client should be represented as a separate thread within the process. Besides, there is another rule for the client operations. The main thread should create the client threads sequentially. In other words, the clients arrive one by one, without any waiting time between. However, after creating a client thread, the client should sleep (or wait) for a random time, which should vary between 50-200 milliseconds (including both 50 and 200). You should take a random sleep time for each client, not just one time. As an example case, after the first client arrives, this client should wait, let's say 63 milliseconds, before starting execution. After creating the first client thread, the second one is created immediately, and the second client should wait, 52 milliseconds according to the random number generator.

For each client thread creation, your program should create its peer as a server thread (note that server is not a separate program, it is just a thread to serve the client's request). After the creation of client/server threads and random sleep time, the client randomly selects a seat, among the ones that have not been reserved yet. By considering the synchronization issues, if that request is satisfied by its server thread, the seat is marked as reserved, and the client number is noted (the details are presented in the previous section, item 5). For example, suppose that Client20 requests for Seat23. If it is available and there is no critical section issue, the server marks Seat23 as reserved by Client20.

After the request of a client is handled and its seat is reserved, both the client and the server threads exit. Here is an important case that you should consider: For instance, Client45 checks the available seats and randomly selects Seat67 for a reservation. After selecting this seat, suppose that the client thread is preempted from the CPU, and another client (Client68) arrives. Surprisingly, it chooses Seat67 for a possible reservation, and since it is still available (i.e., it is not reserved by another client), the server thread reserves Seat67 for Client68, and they exit. Then, Client45 takes the CPU again, but its previous choice is no more available since it is reserved by another client (Client68).

We expect you to handle such cases. If a client chooses a seat for reservation, it should not be reserved by another client even it is still not reserved by the server thread. Considering the previous example, after Client45 chooses Seat67 as a possible reservation, even if it is preempted before reservation, Client68 **cannot** choose it for possible reservation. Your program should handle this scenario. Besides, you should avoid the following type of implementation: 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 whole program as a single critical section, but it does not provide the necessary functionality. So **this type of implementation is forbidden**.

During reservation and progress of the program, you should log the reservations to an external file which should be named as **output.txt**. The program should continue to operate until all seats are reserved. Upon completion, the program should exit.

Below, you may find a sample of output.txt file (i.e., we expect you to use the following format):

```
Number of total seats: 78
Client1 reserves Seat33
Client2 reserves Seat44
Client4 reserves Seat1
Client3 reserves Seat76
...
...
All seats are reserved.
```

(We can foresee questions like "There is a single space between : and 78 at the first line in the example. Should it be like that in our program?". The answer is, one simple space is not important, you can do as you wish.). The output log and the lines should be sorted according to the reservation time, not client arrival time. Since the clients wait for a random time upon creation, the arrival order and reservation order may not be the same. For example, Client3 arrives before Client4 but Client4 reserves a seat before Client3 in the example case of output.txt above. It may be due to either OS scheduling or the random sleep times. Besides, the last line of the output.txt ("All seats are reserved") should be written by the main thread of the program, not by any other thread since it represents the end of the program.

The seat number starts from Seat1, you should not create a seat named as Seat0. Same applies for

the clients. The client to arrive first should be named as Client1. The output.txt should be created in the directory where the main function resides.

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

3 Submission Process

- You need to upload a .zip file on Moodle until the deadline specified at the top of this document (30 December 2018, 23:55). Important note: No .rar files or other extensions are accepted.
- Here is the late submission policy. If you submit within following 24 hours after the deadline, there will be 20 points penalty. For another 24 hours will result in 30 points more penalty. No submissions will be accepted after the second day of the late submission. Here are numerical examples: If you submit your project between 30 December 2018 23:55 - 31 December 2018 23:55, there will be 20 points penalty. If you submit your project between 31 December 2018 23:55 - 1 January 2019 23:55, there will be 50 points penalty in total (20+30). Lastly, you cannot submit your project after 1 January 2019 23:55 (in other words, there will be 100 points penalty). The server's time is taken as the basis, so do not postpone your submission to the last few minutes. If the server is overloaded and your submission is delayed, it is not accepted as an excuse.
- This is an individual assignment, no group submission is allowed. Plagiarism policy of the course applies for this project, which means that your code will be analyzed for plagiarism. No excuses will be accepted.
- The name of the zipped file should be **[StudentID].zip** without the brackets (e.g., "2016400132.zip"). The files that must be included in the zipped file are:
  - Code files (C or C++)
  - A Makefile that creates an executable (cmake is **not** accepted for compilation)
  - A short report (it should be in pdf format, other formats are not accepted) which should discuss your design and methodology. Besides, in addition to the Makefile, the short report should include the necessary commands to compile your code files (e.g., g++ std=g++11 *.cpp -o main).
- You should document your code.
- Here is the grading policy:
  - Code files and program (90%)
  - Short report (5%)
  - Code documentation (5%)
- If you conflict with any provided rule in this section, it will negatively affect your grade.
- Good luck!