Santa Clara University

COEN 383 - Advanced Operating System

Professor: Ahmed Ezzat

Winter 2021

Group 5: Anh Truong, Quan Bach, Travis Le, Yukun Zhang, Pauldin Bebla

PROJECT 3 REPORT

A. Software Design

The problems posed by the project requirements:

- Multiple threads work collaboratively
- Restriction on the number of seats and seat assignment.
- Restriction on the sale window

In order to simulate the ticket selling process, our team implemented 10 **queues** with 3 types of sellers: H, M, and L (refer to the project requirements for more details). At the beginning of the program, 10 queues will be populated with customers (**class Job** in the implementation); each customer has a **name, arrival time,** and **service time**. All queues will be sorted based on the order of customers' arrival time. The number of customers in each queue is implemented as the input parameter when running the program (command line parameter). Once all the queues are ready, 10 **threads** will be created to simulate 10 **ticket sellers**.

The critical section is also determined in the program. Further details for the critical section are in PART C of the report. In order to have the threads most accurately behave like in a real life situation, our team also implemented an algorithm to keep all the threads in sync with a global time. Details of this implementation can be found in PART D of this report. Once everything is set up and ready, a **wake up function** will broadcast the condition to wake up all the threads. Then they will immediately start serving the customers in their queue.

Thread function implementation: when a thread is woken up, it will use the input parameters from the **thread_create function** to identify its seller type. Then the thread will perform some logistical tasks to prepare to enter the loop to serve customers. As the **main** loop starts, the thread will start serving the customer in a queue, or sleep if there is no customer. In order to access or modify any variables in the critical section, the thread has to acquire a **mutex object** called lock, then unlock it and give access to other waiting threads. This will ensure that the variables are modified consistently among threads. Further details of implementation can be found in the below parts of the report.

B. Simulation Parameters

In order to make sure that the program closely simulates real ticket booths, our team has implemented time synchronization among threads. This implementation simulates a global time for all threads and ensures that no customers will be served if his/her arrival time is greater than the global time. This implementation is further explained in PART D of this report.

C. Shared Data and Critical Region

Figure 1 is the description of our team's implementation of the **critical section**. Before accessing to read or make modification to any of the variables in this region, the thread must successfully acquire a specific **initialized mutex** (declared as lock).

```
/* CRITICAL SECTION */

//number of seats
int num_of_seats = 100;
//keep track of threads
int counter =0;
//now a global timer
int time_now = 0;
// logs for each type of
std::vector<std::tuple<std::string,Seat>> H_log;
std::vector<std::tuple<std::string,Seat>> M_log;
std::vector<std::tuple<std::string,Seat>> L_log;
//chronicle log of events
std::multimap <std::string, std::string> chronological_log;

/* END OF CRITICAL SECTION */
```

Figure 1. Declaration/Implementation of Critical Section

The variables in the critical section are:

- The number of seat available of the concert
- A counter to keep track of thread (this counter is implemented to keep time synchronized among threads)
- A global time for all threads
- A log of each type of sellers
- A log of chronological events

D. Time Synchronization

The variable that had to be kept in sync between threads was the time. TIming is important because we did not want a process to reserve a ticket ahead of time before another process that was earlier in time. By doing so, some queues might have all their jobs finished while others might not finish even one job. In addition, the time stamps would be out of place.

To keep the program in sync, threads had a shared **global timer** and a **local timer**. A global timer kept all the processes in sync while the local timer made it easier to call time without requesting a mutex for the global timer. The global timer will increment its time only if every thread is finished with their time piece. So another variable called **"count"** had to be implemented to see which threads are waiting at the moment. Then all local timers will get the global timer when the global timer increments. In addition, all threads will return when its local timer is past a certain point or after 59 time units. All threads must acquire a different mutex (declared as cond_lock) to access or change the global timer or "count." A thread will wait for the local timer to advance when it's either waiting for a new customer, busy with a customer, waiting for the rest of the queues to complete their jobs, or waiting out the time because there are no more seats. If a thread is in the middle of an

order with a customer after 59 minutes, it will add on the time to the local timer and complete that order and exit the thread. Time does not have to be in sync at the last few moments because new orders are not coming in and seats will be reserved for current customers.

E. Displaying Seating Chart

The program prints out a matrix of 10-by-10 that shows which ticket was assigned to which seat. One seller H fills the seats from top to bottom. Six sellers L fill the seats from bottom to top. Three sellers M fills seats in the following order: 5, 6, 4, 7, 3, 8, 2, 9, 1, and 10. All the seats will be assigned from left to right order. Generally, each seller type will fill one seat after another in its respective region: top for H, middle for M, and bottom for L. Special case occurs when one or more seller types are filling seats faster than the others which will lead to collision when each of them picks the next available seat. As we can see in Figure 2, seller L is filling up its region so fast that it collides with seller M at row 6. So it picks the next available seat in its filling direction which is Row 4, Seat 4 for ticket **L208**. Seller M, which was trying to fill the seat for ticket **M109,** had to skip to the next available seat which was Row 4, Seat 5. The seats keep being filled up in the same manner, intertwining between sellers when collision occurs. The "----" indicates that there is no ticket being assigned to that seat.

```
********************** TICKET SEATING MATRIX **********************
Row 01:  H001  H002  H003  H004  H005  H006  H007  H008  H009  H010
Row 02:  L410  L508  ----  ----  ----  ----  ----  ----  ----  ----
Row 03:  M206  M110  L209  L409  L507  L308  M207  L608  M208  L210
Row 04:  M205  M309  M310  L208  M109  L408  L506  L607  L307  L110
Row 05:  M101  M301  M302  M102  M103  M303  M201  M304  M104  M305
Row 06:  M105  M306  M202  M106  M107  M203  M307  M204  M308  M108
Row 07:  L207  L406  L605  L305  L108  L505  L407  L606  L306  L109
Row 08:  L206  L603  L303  L503  L106  L405  L504  L604  L304  L107
Row 09:  L601  L103  L403  L204  L104  L302  L602  L205  L404  L105
Row 10:  L101  L401  L201  L501  L402  L102  L202  L502  L301  L203
```

Figure 2: Ticketing Seating Matrix (Case N = 10)

F. Statistics

Customers of each seller type got served almost as soon as they arrived at the queues. As a result, we see that the **average response time** is very small for all of the seller types. **Average turnaround time** is influenced mainly by the **service time** here. As seller H has smaller service time (1 to 2 minutes) than sellers M and sellers L, its average turnaround time is also smallest. On the other hand, sellers L have the largest service time of 4 to 7 minutes, therefore the average turnaround time is also largest. The **average throughput** is very much the same for all 3 seller types, roughly around 0.15. This means that a ticket gets issued in about every 9 minutes.

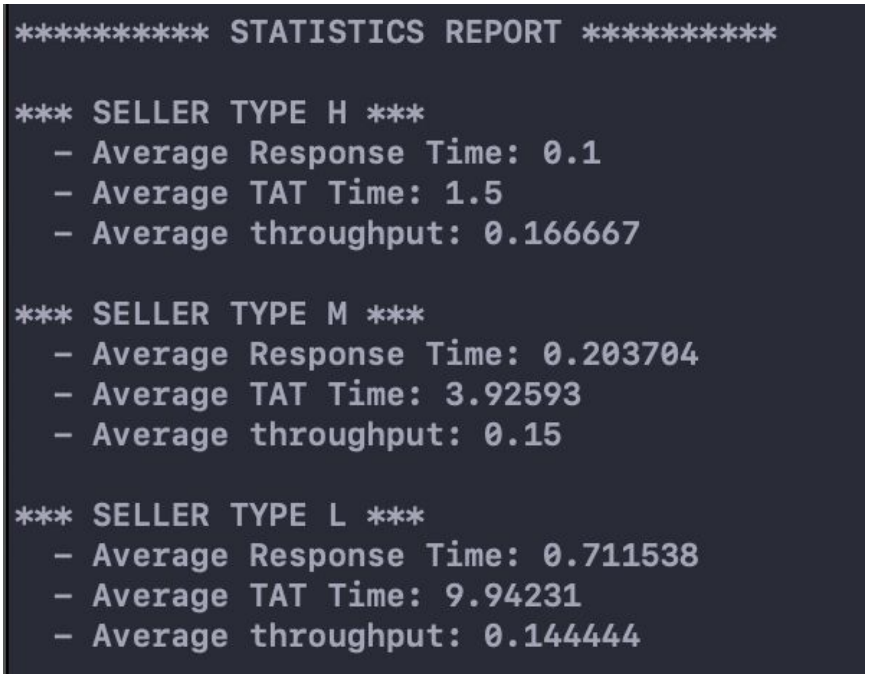

Figure 3: Statistics (Case N = 9)