Kevin Velcich
Matthew Findlay
Esai Morales

Team #1

COEN 383
Project 3 Report
Feb. 6, 2019

Software Design

We approached the design problem by abstracting buyers, sellers and an auditorium into different entities, more specifically C++ classes. Each class had its own set of specific data variables. For example in *sellers*, each seller is given a type (L, M, or H) as well as its list of buyers. The program takes a command line argument for the amount of buyers to generate for each seller. Furthermore, sellers are modeled as threads given they will concurrently perform jobs i.e. selling to customers and may access the available seats at any given time. A *Buyer* contains an ID, an arrival time (for when they enter the seller's queue), and lastly which seat they are assigned to (which is only set after a ticket is purchased). The buyer's arrival time is randomly generated (from 0 to 59) and is added to the end of its sellers queue when the current time matches their arrival time. Lastly, we also have the *Auditorium* class which essentially stores the grid/seats, keeping track of which customer has purchased which seat, which are still available, etc. Our program generates all of the necessary classes for starting the program. Once this is set in place, the threads are awakened and executed in parallel at the start of the simulation through the use of a conditional flag. Every seller begins processing when there is a buyer in their queue.

What parameters did you adjust to make the simulation run realistically?

In order to run our program as realistically as possible, we aimed to achieve several things: firstly we wanted all sellers to be able to sell at the same time, secondly we wanted to ensure that the seat ordering/retrieving didn't have any particular ordering/favoring and it just needed to follow the correct patterns for the three types of sellers, and lastly we aimed to ensure that the system simulates a realistic timeline. To do these we had to carefully minimize our critical regions while logically separating the rest of the selling logic in separate classes/functions so that everything can behave independently. Additionally, we added "sleep()" calls within our function to simulate a realistic time period. This way you can get an idea of what occurs over time and can comprehend a gap in time where no buyers exist. This allows us to easily modify the system

to wait an actual minute between each clock tick so that we simulate an actual run through with a length of one hour.

What data was shared and what were the critical regions?

In order to make our selling realistic and efficient we wanted to minimize the critical regions. Because of this, the main one we have is with the seating grid (auditorium) data structure was shared between all of the threads. The critical regions are those where the sellers perform a transaction and reserve a seat. To minimize the locking of this region, when the transaction occurs, we initially lock the auditorium. However, we reserve the seat, and then unlock the lock. This allows us to do all of the processing without taking unnecessary control over the lock.

What process synchronization was required?

The first synchronization that we required was to have all sellers start at the same time. This was done by using a lock and a conditional flag. When starting, all sellers with unlock and wait on the conditional flag, and once we broadcast the conditional flag, all sellers can proceed.

The main Process Synchronization was required for data consistency with regards to filled and vacant seats. Essentially, it ensures that no available seat is sold to two different customers by different sellers or that no seat that is currently being sold is given to a different customer. What this required was to create a lock for accessing the auditorium/seats. Therefore when a seller started a transaction they needed to acquire the lock first, and wait if its not available.