SWEN-563/CMPE-663/EEEE-663

Real-Time and Embedded Systems

Project 4 - April 9, 2017

Zachary Weeden | zdw7287@rit.edu

Overview:

We were given the task to simulate a bank’s operation for a complete business day, this entailed a customer queue as well as 3 bank tellers to service each arriving customer in the 7 hour day. Subsequent customers arrived anywhere from 1 to 4 minutes of each other and were added to the customer queue which was then handled by the tellers. Each of these customers had a time associated with their transaction which was how long their business would take once taken by a teller, this time ranged anywhere from 30 seconds to 8 minutes. All customers in the queue were guaranteed service regardless of bank hours. The outcome was to gather various metrics of the customers, tellers and of the day of business.

Areas of Focus:

Zachary Weeden: Responsible for entirety of Project 4 (see [https://github.com/zweed4u/Real-Time-and-Embedded-Systems/](https://github.com/zweed4u/Real-Time-and-Embedded-Systems/tree/master/Projects))

Analysis/Design:

My top level design is 4 threads, 3 of which are dedicated for each individual teller and the remaining thread for the customer queue. In my main I toggle a flag to indicate that the bank is open and I create the threads. My main then sleeps for 42 seconds real time which equates to 7 hours simulated time. (100ms real time = 60 seconds simulated) After 42 seconds of real time has elapsed, the bank flag is toggled once again indicating that the bank is now closed. Conversion and some basic arithmetic then takes place on variables needed for metrics and a function is called to print out the metrics in a user friendly format.

A queue thread is used for the simulation of a customer queue in the bank. In this thread a random arrival and transaction time is generated with the specified bounds as given in the report. (Note: the conversion between real time and simulated is taken into account here) This is done for each customer. The thread then sleeps for the generated arrival time as this simulates the customer not arriving until the dedicated time. After the sleep the customer is enqueued and the epoch is recorded for metrics to be used later. The actual data enqueued is the transaction time needed for that particular customer. Therefore, the generated transaction time for a customer can be thought of as their specific ID. After the customer is enqueued the queue depth is looked upon and max depth is then updated if need be. This process repeats with customers being added until the bank flag is toggled in main. (after 42 real time seconds)

Each of the teller threads are very similar with just variables such as “current\_teller1\_customer” renamed to “current\_teller2\_customer” and “current\_teller3\_customer”. The bank flagged is looked upon as well as the size of the queue to ensure that the teller has processing to do. In each of these the epoch is then noted again. This is used to determine in how long a customer has waited in the queue to be seen/dequeued by a teller. Detection on whether this waited time is a new maximum/total time customers have waited in the queue are recorded. The current customer is then defined for the teller as the first person in the queue. The epoch is then used again with the perspective of the teller in mind; that they have received a customer. This will be described later on. The customer is officially dequeued and the teller thread then sleeps for the transaction time for that customer (the data used for that customer enqueue) Again, the time used is properly converted from real time to simulation time. The transaction time is noted and determined whether if it is the longest transaction thus far and updated. After the sleep has completed, translating to the customer being serviced and business taken care of, the epoch is noted once again. Used in conjunction with epoch recorded just prior to the previous customer’s dequeue in this thread, this is used for the determination of how long the teller has to wait before they receive another customer. The total amount of customers served by the teller is then incremented and then the loop repeats. Upon the bank flag being toggled by main, the thread then jumps to an else clause that also takes into account the size of the queue. This is to say that regardless of the time, customers that have made it into the queue will be seen by a teller. Mutexes are exclusively used in dealing anything with the queue as the queue is manipulated by all other threads in the program.

Block diagram here

Test Plan:

Essentially non-applicable. For purposes of accessibility, testing of the program was done on a non-QNX machine but under Linux. GCC was used for compilation and building of the project. Under a different OS there were slight variations and discrepancies in some of the metrics calculated but nothing too startling.

Project Results:

Describe results and compared to expected results – analysis on results here – include screenshots

Lessons Learned:

Brief summary of what was learned in project and descriptions of difficulties encountered

Threads are very fickle and don’t play too nicely with one another without the involvement of a mutex to handle any shared variables/resources. Timers under the QNX system needed to be researched and clock\_\_gettime function proved very helpful in retrieving the epoch at function call which allowed for arithmetic to determine various elapsed times. . I made use of many variables to allow the threads to not interfere with one another but with more research, time and confidence better thought out mutexes could be used to cut down on the number of thread specific variables.