PA06 Report

Programming Assignment #06:
Universal Studies Bounded-Buffer Thread-Pool Problem

ECE437/CS481
Computer Operating Systems
Fall 2022

Professor: Dr. Xiang, Sun
Student: Zachary, Montoya

Submitted: DEC-04-2022

Albuquerque, NM

Executive Summary

The purpose of this report is to document the results of the c programming language code created as part of programming assignment number 6 for course ECE437/CS481 in Fall of 2022. The assignment was to simulate a waiting line at universal studios Jurassic Park ride. The detail requirement specifications can be found in the original assignment issued, see append 1. All requirements stated in the assignment were successfully met, evidence and a description of the source code design is included below in this document.

Table of Contents

Detailed Description

This thread pool is broken into three (3) primary sections, the ParkMaster which oversees opening and closing the park, calling the poissonRandom function based on the time interval to create guest threads. The other two (2) sections are; first the threads for the *Ford Explorers,* herein referred to as Ford Explorers, SUVs, or CARs, and second the threads for the *Guests*. These two (2) types of threads will execute the functions titled CarFunction and GuestFunction, respectively. The guest threads will get in the fixed length waiting line if the waiting line is not at its capacity of 800 people, the SUV threads will then remove guests from the waiting line in a fixed cyclic nature at every minute. In essence this is a classical producer and consumer problem with an exciting amusement park twist, it can be more precisely described as a dynamic-multi producer and static-multi consumer thread pool problem. The crux of this problem is one (1) protecting the critical sections or shared variables without creating a deadlock, and two (2) synchronizing the threads. The second crux, when solved will, improve the first one because the behavior is well defined and controlled.

Synchronization is key between the ParkMaster and CarFunction threads, as the CarFunctions threads are created when the park is opened and loop infinitely until the park is closed. To reduce wasted CPU cycles with the SUVs looping needlessly a PTHREAD_BARRIER is used. The barrier, although not taught in-scope of ECE437/CS481, behaves as the name sounds it, creates a barrier that *waits* all the threads that have approached it until a predetermined quantity of threads have reached it. Semantically, this can be thought of a starting line. The threads won't cross the starting line until *n* number of threads have arrived. In our application *n* is equal to the number of cards + 1, the additional threads being the park master. This barrier is applied so the CARs remove guests in line and get back to the barrier, but the park master has more work to do, it must loop through a nested for loop to count time, it will then release the SUVs at the top of every minute. Thus, achieving synchronization between the master and consumers.

The park master at the top of every minute but prior to beginning of a minute in the timer (above the minute nested for-loop), will determine the number of guest threads that will be attempting to joining the waiting line. The guest threads themselves will document their arrival and determine if to join the waiting line based on its remaining capacity. If the guest threads are rejected, they will document their rejection. If the guest threads enter the waiting line, they will increment a shared waiting line counter variable, store the time they entered the waiting line into an array for starting wait time, increment an indexer for this waiting time array, and wait on a counting semaphore. The guest threads are released (or posted) from the counting semaphore by the SUVs, they will then check the time again, and calculate the time they waited, then signal the SUV to decrement the indexer for the waiting time array.

The SUVs when released from barrier, as described above, will check if the waiting line has enough guest to fill an entire SUV – they are allowed to complete the ride with an SUV that isn't completely full, it will remove the correct number of guests from the line, then it will wait on a binary semaphore to synchronize decrementing of the waiting time array. This binary semaphore is used ensure that the SUV does not decrement the indexer for waiting time array

before the guest thread calculated the waiting time. This method of used a waiting time array and indexer that is incremented and decremented by the producer and consumer was inspired directly from the producer and consumer with a bounded-buffer problem. The waiting time array only works proper because the counting semaphore acts as a queue instead of a stack, FIFO instead of LIFO.

Finally for completeness, the main function simply parses the inputs of -M and -N and has handlers if they are not provided, it initializes and destroys the semaphores and mutexes, and creates the SUV and park master threads in parallel creation. Additionally, there is another function called parkTime that simply formats the secs to HH:MM:SS. Time is tracked in seconds for granularity purposes in the ParkMaster using three for loops, the first one as 10 loops for 10 hours, the second as 60 loops for 60 minutes per hour, and the last loop as 60 loops for 60 seconds per minute. This is a grand total of 36000 seconds the park is open. Lastly, pseudo code describing this system can be found in the subsequent section.