CS307 PA3 Report

Alpay Naçar

31133

Synchronization Primitives:

I used pthread semaphores and mutexes in my programming assignment to ensure synchronization. It is explained more detailed in correctness of threads part but let me give conclusion here. There is 2 points where threads wait each other, first one is to form share rides, second one is for captain to wait others to print “I have found a spot in a car”. To handle both I used counters, semaphores, and mutexes. Mutexes ensure that there won’t be any read write conflict. Every thread increments the counter and checks if counter reached to the wanted amount, if it does posts semaphores for the size of barrier. After that waits on the semaphore. When the thread get past semaphore it continues to the next section. To ensure there is no mid section of other cars, in between mid section of current ride and end section of current ride, I used mutex1, it will be unlocked at the end of all prints.

Correctness of main:

First of all, at the beginning of my main, I checked if arguments can be converted to integers by the isInteger function and then I converted them to integers, then checked if both arguments are multiple of 2 or not, then checked if sum of these two arguments is multiple of 4. If this is the case, I started creating my threads.

I put all threads that I created into an array of threads, so that after my main creates all threads, it can wait for all to join back, then it will terminate.

Correctness of Threads:

In my thread function, there are two barrier like structures, and both of them uses mutexes for mutual exclusion of critical regions. First one checks if there exist valid enough fans for a share ride (4-0 or 2-2), if not unlocks mutex and waits using semaphore of that team. I used two semaphores for both teams distinctly because I wanted to be able to wake up threads with teams that I can use in the share ride. If I only use one semaphore, I can’t wake up threads with the team I want. If there exists a valid enough threads to form a share ride, I wake up corresponding fan threads and assign this waker (the thread that comes in the end and realizing that there is enough fans to form a share ride) thread a car_id, which will be later used to identify captain. This waker also got caught to the first semaphore, but because it already posts one more space for itself, this thread can get past this semaphore directly. We know that there can’t be more threads waiting in waker’s teams semaphore because if there is, it should wake up all other threads without using waker.

Waker thread didn’t unlock the first mutex because I don’t want other rides to get mixed by current ride. It will be unlocked after it prints that this thread is the captain. But before printing “I am the captain” it

should wait all threads to write “I have found a spot in a car”, and to ensure that I used another counter and a semaphore. Just like we saw, in recitations, I incremented the counter one by one, and checked if counter reached to 4, if it does, I posted all 4 threads. There won’t be any dirty read because I am using a mutex and doing all of these critical operations inside this mutex.

In the end I safely exited my threads.

Thread function pseudocode:

```
Thread Function
    Print “Thread ID <tid>, Team <team>, I am looking for a car”

    Lock Mutex1
    Increment teamCount[team]
    If valid amount of passengers (4-0 or 2-2)
        For i = 1 to 4
            Post Semaphore1[corresponding team]
        Decrement corresponding teamCounts by 4
        Set as captain
    Else
        Unlock Mutex1
    Wait Semaphore1[team]

    Print “found a spot in a car”

    Lock Mutex2
    Increment count2
    If count2 equals 4
        Set count2 to 0
        For i = 1 to 4
            Post Semaphore2
    Unlock Mutex2
    Wait Semaphore2

    If captain
        Print “I am the captain driving car “ + carId
        Unlock Mutex1
```