UĞUR KAĞAN ÇAKIR 27058

PA_3 REPORT

Pseudocode of the threads:

Global variables needed by threads:

1. Semaphore variable “car” for keeping track of a car and when it becomes full
2. Semaphore variables “barrier_a” and “barrier_b” to use as semaphore barriers preventing threads to move forward before finding a suitable car for a fan
3. Semaphore variable mutex for atomically interpreting some operations after passing the barrier
4. Integers count_a and count_b for keeping track of waiting fans
5. Integer finishedthreads for keeping track of threads(fans) that found a seat
6. Integer isCarFull for keeping track of emptiness status of a car

```
Inside the fan_transport function

Initialize info to keep the team of the thread

Sem_wait the car

    Print that this fan is waiting for a car

    Update count_a or count_b depending on the team thread supports

    If after adding this thread, we have 4 B supporters

         Car becomes full

        Count_b = 0

        Sem_post 3 threads stuck at barrier_b

    Else if after adding this thread, we have 4 A supporters

         Car becomes full

        Count_a = 0

        Sem_post 3 threads stuck at barrier_a

    Else if after adding this thread, we have 2 A and 2 B supporters

         Car becomes full

        Count_b -= 2

        Count_a = 0

        If the team that this fan supports = “A”
```

```
                Sem_post a thread stuck at barrier_b
            If the team that this fan supports = "A"
                Sem_post a thread stuck at barrier_a
            Sem_post a thread stuck at barrier_a
            Sem_post a thread stuck at barrier_b
        Else if after adding this thread, we have 2 A and more than 3 B supporters
            Car becomes full
            Count_b -= 2
            Count_a = 0
            Sem_post a thread stuck at barrier_a
            Sem_post 2 threads stuck at barrier_b
        Else if after adding this thread, we have 2 B and more than 3 A supporters
            Car becomes full
            Count_a -= 2
            Count_b = 0
            Sem_post 2 threads stuck at barrier_a
            Sem_post a thread stuck at barrier_b
        Else
            If this fan supports A
                Sem_post car //release car's lock
                Sem_wait in the barrier_a
            If this fan supports B
                Sem_post car //release car's lock
                Sem_wait in the barrier_b
    Sem_wait the mutex
        Print that this fan has found a car
        Finishedthreads++
        If( finishedthreads == 4 ) //if all the seats of the car are filled
            Finished_thread = 0
            Print that the last fan to find a seat in this car is the captain
            isCarFull = 0
            sem_post the car //release car's lock
    Sem_post the mutex
```

Discussion of the synchronization mechanisms:

First of all, I used pthreads, initialized each thread by pthread_create functions. I also used pthread_join function to prevent the main thread to finish after the children threads.

In my solution I used semaphores as synchronization mechanisms. I used in total 4 semaphore variables to which I used 2 of them as regular semaphore variables (car and mutex) and 2 of them as barriers (barrier_a and barrier_b). When the fan threads initialize, they first come to the sem_wait(&car) line, at this line the thread may go on executing the next lines if the car semaphore is not being held by any other thread. The lock of the semaphore car is held by the thread if it can pass through sem_wait(&car) line, the lock is released if the thread reaches to the else part of the if-else statement. If a thread reaches the else statement this means we are yet to find a group of fans that can transport together. After releasing the lock at the else part, the threads stuck at one of the barriers, and they will stay stuck until some thread wakes them up. On the other hand, if the thread does not reach to the else part, the lock will be held until some thread reaches the sem_post(&car) line inside the mutex semaphore. Whenever the else statement after the car semaphore is not executed (i.e. the lock of the car is held by some thread), it means that we have found a suitable group of 4 people that can travel with this car. When a fan thread finds a suitable group of people to travel with, that fan wakes up the other threads that will travel with him. At this point only 4 threads can run further because the lock of the car semaphore is still held by a thread and it is not being released yet. After this point each of the 4 fan threads that will use the car executes what is between sem_wait(&mutex) and sem_post(&mutex) in an atomical way. When the last thread of the 4 threads executes these commands it also becomes the captain, updates some information about the car, and finally releases the lock of the car semaphore before ending its executions completely.

My implementation satisfies the needs described in the homework since, the atomicity of the operations is preserved, the order of the operations is correct and there is no deadlock situation. Since, in each if-else statement correct amount of threads are woken up, we won't face any thread that sleeps forever. Also, the lock of the car semaphore is being released at the end after the threads found their seats, so the new threads that will want to get a seat in a car can start their operations. The order of the operations are also correct since, the sem_wait(&car) is passable whenever there is a need to find more people to get into the car we are working on, and whenever we find a suitable group of people, the lock of the car is held until everybody gets their seat and a captain is chosen. The atomicity of the operations are also preserved, the critical sections are either wrapped around between sem_wait(&car) and sem_post(&car) lines or between sem_wait(&mutex) and sem_post(&mutex) lines.