Kemal Yılmaz 31097

In my rideshare application, I used 4 semaphores from the semaphore.h class. Which locks itself when its value is 0.

The 4 semaphores and their initializations are:

SemA -> The semaphore that holds the A team threads. It is initialized as 0 because it needs to be locked until the right combination is found.

SemB -> The semaphore that holds the B team threads. It is initialized as 0 because it needs to be locked until the right combination is found.

SemP -> The semaphore makes sure the 4 threads that pass from their respected semaphores(the right combination threads) prints their found a car statements before any other thread can print found a car before the 4 thread fills up the car. It is initialized to 4 as the first combination can pass freely then they finish their operations and after the car leader statement is printed the semaphore is posted 4 times so that the next combination can start printing their found a car statements.

mutex -> It is a semaphore used as a mutex lock. It is used to prevent race conditions for when global variables such as A thread num, B thread num and current threads in the car are being incremented. It is initialized to 1 as it needs to start unlocked for a thread to capture it and do its operations.

Pseudocode for thread implementation:

ThreadFunc:

```
lock(mutex) // as mentioned to prevent race conditions in global variables
print(thread id,looking for a car);
if(team == A) numA++; //number of A threads that are sleeping
else numB++; //number of B threads that are sleeping
if(numA == 4){ // 4 A combination
        for(i=0;i<4;i++) sem_post(&semA);
//posts semA for times so 4 A threads that are waiting can continue to run
        numA-=4 // since they are woken up num is decreased
}
else if(numB == 4){ // 4 B combination
        for(i=0;i<4;i++) sem_post(&semB);
//posts semB for times so 4 B threads that are waiting can continue to run
        numB-=4 // since they are woken up num is decreased
}
if(numA >= 2 && numB >= 2){ // 2 A 2B combination
```

```
        for(i=0;i<2;i++) sem_post(&semA);
        for(i=0;i<2;i++) sem_post(&semB);
//posts semA 2 times and semB 2 times so 2 B and 2 A threads continue to run
        numA-=2 // since they are woken up num is decreased
        numB-=2 // since they are woken up num is decreased
}

unlock(&mutex);

if(Team A) sem_wait(semA); // when posted A threads wake up from here
else sem_wait(semB); when posted B threads wake up from here

// this ensures that A threads wait at semA and B threads wait at semB;

// team name is checked via the void* arg of the threads by first turning them into
//char pointers then dereferencing these char pointers

sem_wait(semP) // this is initialized to 4 at the beginning so when the right
combinations leaves the semA and semB those 4 threads can pass from here and
when another correct combination is found it holds them here – when 4 threads pass
semP becomes 0 so it gets locked until it's posted.

lock(mutex) // this is to ensure the 4 threads that pass don't create a race condition
when adding to the threads that are currently in the car.

print(found a car)
threadsInTheCar++;
if(threadsInTheCar == 4){ //which means the last thread is in the car
print(i am the captain,carId) // the last thread in the car becomes captain
carId++;
for(i=0;i<4;i++) sem_post(&semP); // so that the next 4 combo can print found a car;
}
unlock(mutex);
```

As described in the pseudocode here is how my application works but to further explain it also satisfies the correctness conditions.

All children execute before the termination: All threads are joined at the main thread so this is ensured.

For each thread init,mid,end order: This is ensured as init statement and the thread incrementing the sleeping thread count is done in a mutex so a thread can't be in a valid combo before it is initialized after that 4 threads that are in the critical section end part is only printed when 4 threads are in the car so it is also ensured.

4 mids before end: As ensured by semP, only 4 threads can be in the section that prints the mid part and after the 4th mid part is printed end parts are printed in a mutex. Also printed combo is valid as posix semaphores work in FCFS scheduling thus ensuring the correct combo enters the critical section.

Number of prints: Every thread gets a chance to execute and find a valid combo as there are input checks that ensure that.