CS 307 Programming Assigment 3
Riding to a Soccer Game

İsmail Berat Düzenli

December 19, 2022

1 Pseudo Code

The simple idea is there are 2 types of fans, drivers and non drivers.
Non drivers lock the mutex prevents other interruption checks if either it is driver or not and since it is not it releases mutex lock letting another fan to get in line and it waits at semaphore lock.
A driver comes locks the mutex check either it is driver and then since it is driver it releases correct number of A fans and B fans and waits until all fans sit in car (using barrier) then announce his drivership and releases lock.
Drivership check is done by bookkeeping fans waiting in semaphore threads. If fan distribution is either (3,0), (3,1), (1,2), (1,3) and a fan comes to first group (1, 0) then a car must be arranged. if it is (3,0) or (3,1) driver signals 3 fans from his own team, if it is (1,2) or (1,3) then driver signals 1 from his own team and 2 from rival team.

The pseudo code for the threads in the given code is as follows:
Team: t is 0 or 1
release(t,n) is a function that releases n t team fans looking for a car
waiting is a 2 lenght array holding fun numbers claimed to wait a car.

```
function FAN_THREAD(t)
    driver ← 0
    lock(mutex)
    print(Thread ID: pthread_self(), Team: t)
    if waiting[t] = 1 then
        if waiting[1 − t] ≥ 2 then
            driver ← 1
            release(t, 1)
            release(1 − t, 2)
        else
            waiting[t] ← waiting[t] + 1
            unlock(mutex)
            wait(sem[t])
        end if
    else if waiting[t] = 3 then
        driver ← 1
        release(t, 3)
    else
        waiting[t] ← waiting[t] + 1
        unlock(mutex)
        wait(sem[t])
    end if
    print(Thread ID: pthread_self(), Team: t)
    wait(barrier))
    if driver then
        print(Thread ID: pthread_self(), Team: t)
        unlock(mutex)
    end if
end function
```

2 Synchronization Mechanisms Implementation

The code uses the following synchronization mechanisms:

- Mutex: Mutexes are used to process one thread at a time to preent concurrent access to common memory, especially to array holding number of fans told who went into thread.
- Semaphores: Semaphores are used to hold fans before a car is filled with correct number distribution. Then release correct number of fans after checking waiting[] array.
- Barrier: Barrier is used to prevent driver to announce himself as driver before all selected fans got into the car.

3 Correctness

3.1 Correctness Requirements

- There must be exactly numA + numB init strings printed to the console.
- There must be exactly numA + numB mid strings printed to the console.
- There must be exactly (numA + numB)/4 end strings printed to the console.
- For each thread init, mid and end (if exists) must be printed to the console in this order.
- There must be exactly 4 mids between consecutive ends and before the first end. The thread identifier of the end must be the thread identifier of one of these mids. In addition, teams of these mids must be a valid combination. Lastly, inits corresponding to this 4 mid group (inits by the same threads) must be before the first mid of this group.

3.2 Correctness

- Using parameters given by user, A and B number of threads created using 2 for loop. Every threads displays 1 init message without any condition, so all threads (#A+B) display init message.
- Same as init all threads display mid string without any condition providing A+B number of mid strings.
- end string is displayed only when a threads claims driver role. Driver role is claimed when arrival of given thread causes completion of a car. This only happens when there are at least 3 already waiting threads which are distributed as (3,0) or (1,2) (own team, rival team), and there may be 1 rival fan more. A driver will release 3 already waiting fans and proceed itself as forth. So 1 driver will exist when there are 4 fans placed to a car. (A+B)/4
- Path of an any fan is below
  - **1.** begin $\longrightarrow$ lock $\longrightarrow$ print(init) $\longrightarrow$ unlock $\longrightarrow$ wait(sem[t]) $\longrightarrow$ getsignal(sem[t]) $\longrightarrow$ print(mid) $\longrightarrow$ wait(barrier) $\longrightarrow$ exit
  - **2.** begin $\longrightarrow$ lock $\longrightarrow$ print(init) $\longrightarrow$ unlock $\longrightarrow$ be driver $\longrightarrow$ send signals $\longrightarrow$ print(mid) $\longrightarrow$ wait(barrier) $\longrightarrow$ unlock $\longrightarrow$ exit

  In a car ride there will be four fans where first 3 followed first path and last 1 followed second path. First three will display init and wait sem lock without interruption because of mutex locks, then last one will display init without any mid displayed (all 4 init displayed without any mid or end and this is deterministic).
  Then first 3 waiting fans are signaled and freed while also fourth fan is running. At this point they are only 4 threads running since fourth thread (driver) is holding mutex lock preventing any other threads to proceed. Then all 4 threads will display mid and every one of them wait others to do on the barrier. When barrier is (filled and) opened all 4 threads will be displayed their inits and mids in desired order. Then first three threads doesn't matter since they got nothing to do. Last thread will announce end and will unlock mutex lock.