CS307 - Operating Systems — 2021-2022 Fall

Programming Assignment 3 Report

Doğukan Yıldırım — 28364

The Flow of Threads

(a) Pseudocode

```
void* fan_routine(void* args) {
    Lock the mutex semaphore to access shared variables
    Print "I am looking for a car"
    Wait the 'outer waiting queue' semaphore of own team
    Get the number of threads (fans) waiting from own team (from sem value)
    Get the number of threads (fans) waiting from other team (from sem value)

    If there are more than or equal to 4 fans waiting from the team of the fan:
        Wake 3 threads from the own team waiting at the inner waiting queue
        Wake 4 threads from the own team waiting at the outer waiting queue
        Print "I have found a spot in a car"
        Wait at a barrier until all 4 fans are seated in the car
        Print "I am the captain and driving the car"
        Unlock the mutex semaphore

    Else if there are 2 fans from the own team and 2 or more fans waiting from the
    other team:
        Wake 1 thread from the own team waiting at the inner waiting queue
        Wake 2 threads from the other team waiting at the inner waiting queue
        Wake 2 threads from the own team waiting at the outer waiting queue
        Wake 2 threads from the other team waiting at the outer waiting queue
        Print "I have found a spot in a car"
        Wait at a barrier until all 4 fans are seated in the car
        Print "I am the captain and driving the car"
        Unlock the mutex semaphore

    Else: // No valid combination for car seating
        Unlock the mutex semaphore
        Wait in the inner waiting queue of the own team
        Print "I have found a spot in a car"
        Wait at a barrier until all 4 fans are seated in the car
}
```

(b) Code

```c
// Threads from different teams will use the same routine
void* fan_routine(void* args) {
    struct Fan* fan = (struct Fan*) args;
    sem_wait(&mutex);
    printf("Thread ID: %ld, Team: %c, I am looking for a car\n", pthread_self(),
    fan->team);
    sem_wait(&*(fan->sem_own_team));
    int waiting_own_team, waiting_other_team;
    sem_getvalue(&*(fan->sem_own_team), &waiting_own_team);
    sem_getvalue(&*(fan->sem_other_team), &waiting_other_team);

    if (waiting_own_team <= 0) {
        sem_post(&*(fan->sem_own_team_inner));
        sem_post(&*(fan->sem_own_team_inner));
        sem_post(&*(fan->sem_own_team_inner));

        sem_post(&*(fan->sem_own_team));
        sem_post(&*(fan->sem_own_team));
        sem_post(&*(fan->sem_own_team));
        sem_post(&*(fan->sem_own_team));

        printf("Thread ID: %ld, Team: %c, I have found a spot in a car\n",
        pthread_self(), fan->team);
        pthread_barrier_wait(&barrier);
        printf("Thread ID: %ld, Team: %c, I am the captain and driving the car\n",
        pthread_self(), fan->team);
        sem_post(&mutex);
    }

    else if (waiting_own_team == 2 && waiting_other_team <= 2) {
        sem_post(&*(fan->sem_own_team_inner));
        sem_post(&*(fan->sem_other_team_inner));
        sem_post(&*(fan->sem_other_team_inner));

        sem_post(&*(fan->sem_own_team));
        sem_post(&*(fan->sem_own_team));
        sem_post(&*(fan->sem_other_team));
        sem_post(&*(fan->sem_other_team));

        printf("Thread ID: %ld, Team: %c, I have found a spot in a car\n",
        pthread_self(), fan->team);
        pthread_barrier_wait(&barrier);
        printf("Thread ID: %ld, Team: %c, I am the captain and driving the car\n",
        pthread_self(), fan->team);
        sem_post(&mutex);
    }

    else {
        sem_post(&mutex);
        sem_wait(&*(fan->sem_own_team_inner));
        printf("Thread ID: %ld, Team: %c, I have found a spot in a car\n",
        pthread_self(), fan->team);
        pthread_barrier_wait(&barrier);
    }
}
```

(c) Explanation

Threads from different teams will use the same routine, in other words, each fan thread will use the same function, namely $fan\_routine(args)$, for the purposes of the program. The team that the fan supports, and 4 other pointers to certain semaphores will be passed as arguments, which the details are explained below:

There are 5 semaphores in the implementation: 1 semaphore with initialized value of 1 which behaves like a mutex, 2 semaphores for fans of team A and 2 semaphores for fans of team B. Each team has one semaphore which acts 'outer waiting queue' (referred to as **sem_own_team** or **sem_other_team**), and another one semaphore which acts like a 'inner waiting queue' (referred to as **sem_own_team_inner** or **sem_other_team_inner**). 'Outer waiting queue' semaphores are initialized to value of 4, i.e. there are 4 resources that can be used without causing any waiting, while the 'inner waiting queue' semaphores are initialized to value of 0, i.e. behaves similar to conditional variables.

Each fan thread are provided with pointers that point to these semaphores, i.e. each fan thread can access their own team's waiting queues, and also the other team's waiting queues. This is an integral part of the program.

At first, only maximum number of 4 threads (fans) from the same team can proceed with the program, the other soon-to-arrive threads from the same team will be put into the 'outer waiting queue'. Each thread that proceeds with the program without being put into the 'outer waiting queue' will check if there exists a valid combination for seating. If there doesn't exist a valid combination, then the thread will be put into the 'inner waiting queue'. If there exists a valid combination, then the corresponding threads that form the valid combination which are also stuck waiting in the 'inner waiting queue' will be awakened so that they can go to the barrier (the car in this case) and 4 threads will be awakened from the 'outer waiting queue' so more fan threads can proceed with the program. The barrier's only purpose is to ensure that all fans are seated in the car before the car starts to drive off into the sunset. After all 4 threads are at the barrier, they pass through the barrier and the thread that awakened the other 3 is declared as the captain/driver, then the program continues on with the same steps until there are no more fans (threads) that are looking for a ride share.

My implementation is **correct** due to the facts that:

- The main thread checks the validity of input arguments correctly and as described in the document, and it also creates the correct number of children threads. If there are any issues regarding the validity of inputs or the creation of threads, then the main thread terminates.
- The strings that are printed out by the fan threads are in correct order.
- The number of *init*, *mid* and *end* strings printed out to the console are the same numbers specified in the document: no less, no more.
- *mid*s represent a valid combination.

No sample output of my implementation is included which would have proven the correctness claims above. However, one can check whether the correctness claims are represent the truth or not by just running the program and doing simple checks.