CS307 PA3

Gizem Topsakal

December 20, 2022

1 Pseudocode

Flow of the thread function as a pseudocode:

```
initialize all the global variables
int A_fans, B_fans, A_fan_count, B_fan_count, captain
pthread_mutex_t mutex_A = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex_B = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex_AB = PTHREAD_MUTEX_INITIALIZER;
sem_t sem_A, sem_B;
void * rideshare(void * argv)
if team is A then
    lock
    A_fans++
    if B has more than 2 and A has 2 then
        post 2 A sems
        post 2 B sems
        A_fans-2
        B_fans-2
    end if
    if B has 4 then
        post 4 B sems
        B_fans-4
    end if
    if A has 4 then
        post 4 A sems
        A_fans-4
    end if
    unlock
    semwait
end if
if team is B then
    lock
```

```
        B_fans++
        if A has more than 2 and B has 2 then
            post 2 A sems
            post 2 B sems
            A_fans-2
            B_fans-2
        end if
        if B has 4 then
            post 4 B sems
            B_fans-4
        end if
        if A has 4 then
            post 4 A sems
            A_fans-4
        end if
        unlock
        semwait
    end if
    if team is A or B then
        lock
        captain++
        if captain is divisible by 4 then
            it is captain
        end if
        unlock
    end if
```

2 Description

This code is a program that simulates a ride-sharing service for fans of two different sports teams, team A and team B. The program uses threads to represent individual fans, and it includes mutexes and semaphores to synchronize the threads and ensure that the fans can find rides in an organized manner.

The program starts by initializing several variables, including the number of fans for each team, a "captain" variable to track the number of fans who have found rides, and various mutexes and semaphores.

The main function then takes in the number of fans for each team as command line arguments and creates a thread for each fan using the pthread_create() function. The threads are created by passing a pointer to the thread, an attribute object (which is set to NULL in this case), a pointer to the function that the thread will execute, and a void pointer to the argument passed to the function. In this case, the function that is executed by the threads is rideshare(), and the argument passed to this function is a character representing the team that the fan belongs to ('A' or 'B').

After creating the threads, the main function calls pthread_join() on each of them, which waits for the threads to complete before proceeding. The rideshare() function is where the logic for assigning fans to cars is implemented. It takes a void pointer to an argument as input and casts this argument to a character, which represents the team that the fan belongs to.

The function first acquires a lock on the mutex corresponding to the team that the fan belongs to using pthread_mutex_lock(). The function then increments the count of fans for that team and prints a message indicating that the fan is currently looking for a ride. The function then checks the number of fans of the other team looking for a ride and takes the appropriate action based on the following cases:

1. If there are at least 2 fans of team B and 2 fans of team A looking for a ride, the function assigns 2 fans of each team to a car by signaling the availability of a car to the semaphores corresponding to both teams using sem_post(). The count of fans looking for a ride is then decremented by 2 for both teams.

2. If there are at least 4 fans of team B looking for a ride, the function assigns all of them to a car by signaling the availability of a car to the semaphore corresponding to team B using sem_post(). The count of fans looking for a ride is then decremented by 4 for team B.

3. If there are at least 4 fans of team A looking for a ride, the function assigns all of them to a car by signaling the availability of a car to the semaphore corresponding to team A using sem_post(). The count of fans looking for a ride is then decremented by 4 for team A.

After completing these actions, the function releases the lock on the mutex using pthread_mutex_unlock() and waits for a car to become available by calling sem_wait() on the semaphore corresponding to the team that the fan belongs to. Once a car becomes available, the function prints a message indicating that the fan has found a ride and exits.

The semaphores and mutexes used in this code are used to synchronize the threads and protect access to shared resources. The semaphores are used to signal the availability of a car to the waiting fans, and the mutexes are used to ensure that only one thread can access the global variables that keep track of the number of fans of each team looking for a ride at a time. This prevents race conditions, where two or more threads try to access the same resource simultaneously, leading to unpredictable behavior.

In summary, this code implements a ride-sharing program for fans of two different teams, using threads, mutexes, and semaphores to synchronize the threads and ensure that the fans can find rides in an organized and predictable way. The rideshare() function is responsible for assigning fans to cars, using mutexes to protect access to shared variables and semaphores to signal the availability of rides to waiting fans. The program uses these synchronization tools to prevent race conditions and ensure that the shared resources are accessed in a controlled and coordinated manner.