Hagverdi Ibrahimli
30014

In this report, I will explain the overall flow of my program that is simulating a rideshare application. First of all, I coded the application using the C++ programming language. In addition to that, I utilized the synchronization mechanisms such as the Semaphores and Barriers, readily available in C-libraries. It is important to note that, I did not tweak or change these synchronization primitives and used them as they are by inspecting their corresponding manual pages.

At the start of the program, I initialize and/or declare some global variables such as: a vector to store the thread ids, a *pthread barrier,* 4 semaphores (2 binary semaphores to use as mutex locks and 2 other semaphores to use to make threads sleep according to their team type). Furthermore, I initialize 3 integer counters: 2 to keep track of currently waiting number of fans for team A and team B, respectively. And the last integer counter is to store the unique car ids that should be incremented each time a ride is formed.

Moving to the main part of the program, main thread created by the operating system on program's execution starts by running the *main()* function. Firstly, we check for the validity of the user arguments provided:

1) If the number of arguments provided is not equal to 3, return an error value, terminate the process by printing the appropriate statement.
2) Provided that (1) holds, if the arguments for number of fans of Team A and Team B are negative or not even integers or their total sum is not a multiple of 4, return an error value, terminate the process by printing the appropriate statement.

After validating the arguments, I initialize all of the semaphores and the barrier accordingly: Initialize two of the semaphores with an initial value of 1, use them as binary semaphores: mutex locks in our case. One of them *checkMutex* is to prevent from data race in critical sections and the other *printMutex* is to ensure atomicity of print statements in the program.

Initialize the other two semaphores to an initial value of 0: these semaphores are to keep the fans of the Team A and fans of the Team B sleeping, accordingly. Initially any thread calling *sem_wait()* on this semaphore will be put to sleep. And, once the driver is ready, these threads should be signaled accordingly via *sem_post()* to be awakened. Initialize the *pthread barrier* to a value of 4, so that only after 4 threads hit on this barrier, will they be permitted to continue.

Following these initializations, I create the provided number of threads for fans of team A and team B, respectively, initialize them, assign them to the function *share_ride*, and add their thread ids to the global vector *fanThreads*. As the last part of the main function, the main thread waits for all fan threads to finish their execution and deallocates memory allocated and destroys the barrier, then terminates.

The key part of the program lies in the *share_ride* function, as all fan threads are executing it. Here is the main logic:

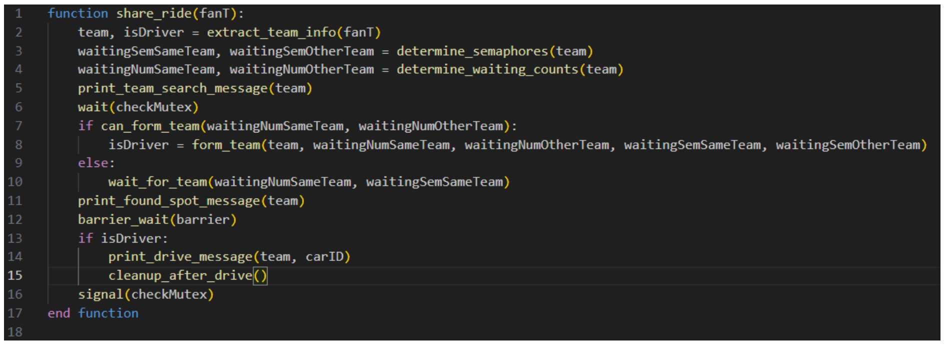
1) Get the current fan thread's team type, and assign its *isDriver* (whether this thread is the driver of the ride or not) attribute to false, by default. Initialize some semaphore and integer pointers to keep track of the global semaphores and integer values. To clarify, each thread needs to know the number of waiting threads of team A and team B, along with currently sleeping threads of the team A and team B accordingly.

2) Based on the team type of the fan thread, assign the variables mentioned above to their corresponding global declarations. Acquire the *printMutex* semaphore and print own thread id, along with a notification that this thread is looking for a car (ride). Release the semaphore by calling *sem_post()* on it. Now, before entering the critical section, acquire the *checkMutex* semaphore. There are two cases to handle at this point. Either this thread will be able to form a ride and declare itself as the captain, or it will sleep since no valid ride can be formed.

<u>Case 1:</u> A ride can be formed because there are either three (or more) people with the same team type as this tread and/or there is one (or more) people with the same team type and 2 (or more) people with the other team type. In both scenarios, assign this thread as the driver and increment the global car id counter. If we formed the ride with only the same team type threads, decrement the global value of threads waiting for the same team type by 3, and signal the according semaphore to wake three of those threads up. On the other hand, if we formed the ride the other way, decrement the global value of threads waiting for the same team type by 1 and the global value of threads waiting for the other team type by 2, signaling 1 person from the same team type and 2 from the other team type, thus waking them up. Then, all these threads will acquire and release the *printMutex* semaphore and print to the console that they have found a spot, along with their thread ids and wait at the barrier until all 4 reaches the barrier. Then, upon the release from the barrier, the thread that is the assigned driver for this ride will print that it is the driver, destroy the current barrier and make a new one for the upcoming (if exists) ride. Finally, it releases the *checkMutex* semaphore so that other threads can enter the critical section.

<u>Case 2:</u> A ride can not be formed because there is no valid configuration as for the number of threads waiting with the same or the other team type as this thread. Thus, this thread increments the global counter of waiting threads of the same team type as its own, releases the *checkMutex* so that others can enter the critical section as well. Lastly, this thread waits on the global semaphore of waiting threads of the same team type as its own.

Also, here is the simple pseudocode implementing the design decisions I explained above:



```
function share_ride(fanT):
    team, isDriver = extract_team_info(fanT)
    waitingSemSameTeam, waitingSemOtherTeam = determine_semaphores(team)
    waitingNumSameTeam, waitingNumOtherTeam = determine_waiting_counts(team)
    print_team_search_message(team)
    wait(checkMutex)
    if can_form_team(waitingNumSameTeam, waitingNumOtherTeam):
        isDriver = form_team(team, waitingNumSameTeam, waitingNumOtherTeam, waitingSemSameTeam, waitingSemOtherTeam)
    else:
        wait_for_team(waitingNumSameTeam, waitingSemSameTeam)
    print_found_spot_message(team)
    barrier_wait(barrier)
    if isDriver:
        print_drive_message(team, carID)
        cleanup_after_drive()
    signal(checkMutex)
end function
```

<u>My code satisfies the correctness criteria because:</u>

num_A: the number of team A fans
num_B: the number of team B fans
init: The string Thread ID: < tid >, Team: < AorB >, I am looking for a car
mid: The string Thread ID: < tid >, Team: < AorB >, I have found a spot in a car
end: The string Thread ID: < tid >, Team: < AorB >, I am the captain and driving the car with ID < cid >

1) The main thread waits for all threads to finish their execution, afterwards printing the appropriate statements and terminating the program.
2) There are exactly *(num_A + num_B)* number of *init* and *mid* strings printed to the console.
3) There are exactly *(num_A + num_B)/4* end statements printed to the console.
4) The value of the car id is unique for each ride and starts by the value of 0.