Programming Assignment 3

Bilal Berkam Dertli 29267

In this programming assignment, I wrote a rideshare.c file to simulate a rideshare for two team fans: Team A and Team B. In my implementation, I used two semaphores for waiting fans of two teams, and one barrier for synchronization of console outputs which is used in each valid combination of a ride share band. Implementation of semaphores, barrier and threads are all coming from pthread library

For the main thread, a pseudocode can be given as follows:
Get the two arguments of number of fans for Team A and Team B

If number of fans is even for both team, and total number of fans is multiple of 4 input check is valid, continue

Initialize two semaphores **for both teams** (will be explained later)

Initialize one barrier to be used in formation of a rideshare band **to synchronize the console outputs of the threads, barrier count is set to 4 since a band consists of 4 fans!**

Allocate memory from heap for the threads that will be created

Create number of A fans (given as input) of threads and call the function for A fans with the newly-created threads, similarly, create number of B fans (given as input) of threads and call the function for B fans with the newly-created threads.

Wait for all threads to finish their execution

Free the memory of threads and barrier

Print “The main terminates” and terminate

Otherwise directly print “The main terminates” **without forming any child thread**

For any fan thread of Team A:

Grab the global lock so that no two threads try to form a band concurrently, otherwise two thread might try to form a band with the same 3 threads (fans) waiting (called sem_wait()) and data race can occur!

Print the init statement: "Thread ID: <TID>, Team: A, I am looking for a car.\n"

Check first possible case: There are 3 waiting (called sem_wait()) A fans, and this thread is the fourth, so they can form a band!

If true, **the running thread declares itself as the driver(captain)** of the band, and wakes 3 A threads by calling sem_post on A fans semaphore 3 times.

Check second possible case: There are 1 waiting (called sem_wait()) A fan, and two waiting (called sem_wait()) B fans, this thread is the fourth to form a 2 A – 2 B band, so they can form a band!

If true, **the running thread declares itself as the driver(captain) of the band**, and wakes 1 A thread by calling sem_post on semaphore of A fans once, and wakes 2 B threads by calling sem_post on semaphore of B fans twice.

If two cases are not valid, then **current thread cannot form a band** under current conditions, and need to sleep and **wait for another thread to form a band including itself**: Unlock the global lock, and call sem_wait())

//Note: If any thread reaches this stage, this means that the thread has somehow found a band (does not matter if the running thread is the captain or not)

Print the mid statement: "Thread ID: <TID>, Team: A, I have found a spot in a car.\n"

Wait for all members of the band to print the mid statement: **Usage of barrier**

If the running thread is the captain, **after all threads printed the mid message (which is synchronized by barrier),** state that this thread is the captain, and print the car number as well:
"Thread ID: <TID> Team: A, I am the captain and driving the car with ID <CARID>.\n", then terminate.

If the current thread is not the captain, directly terminates.

For any fan thread of Team B:

Grab the global lock so that no two threads try to form a band concurrently, otherwise two thread might try to form a band with the same 3 threads (fans) waiting (called sem_wait()) and data race can occur!

Print the init statement: "Thread ID: <TID>, Team: B, I am looking for a car.\n"

Check first possible case: There are 3 waiting (called sem_wait()) B fans, and this thread is the fourth, so they can form a band!

If true, **the running thread declares itself as the driver(captain)** of the band, and wakes 3 B threads by calling sem_post on B fans semaphore 3 times.

Check second possible case: There are 1 waiting (called sem_wait()) B fan, and two waiting (called sem_wait()) A fans, this thread is the fourth to form a 2 A – 2 B band, so they can form a band!

If true, **the running thread declares itself as the driver(captain) of the band**, and wakes 1 B thread by calling sem_post on semaphore of B fans once, and wakes 2 A threads by calling sem_post on semaphore of A fans twice.

If two cases are not valid, then **current thread cannot form a band** under current conditions, and need to sleep and **wait for another thread to form a band including itself**: Unlock the global lock, and call sem_wait())

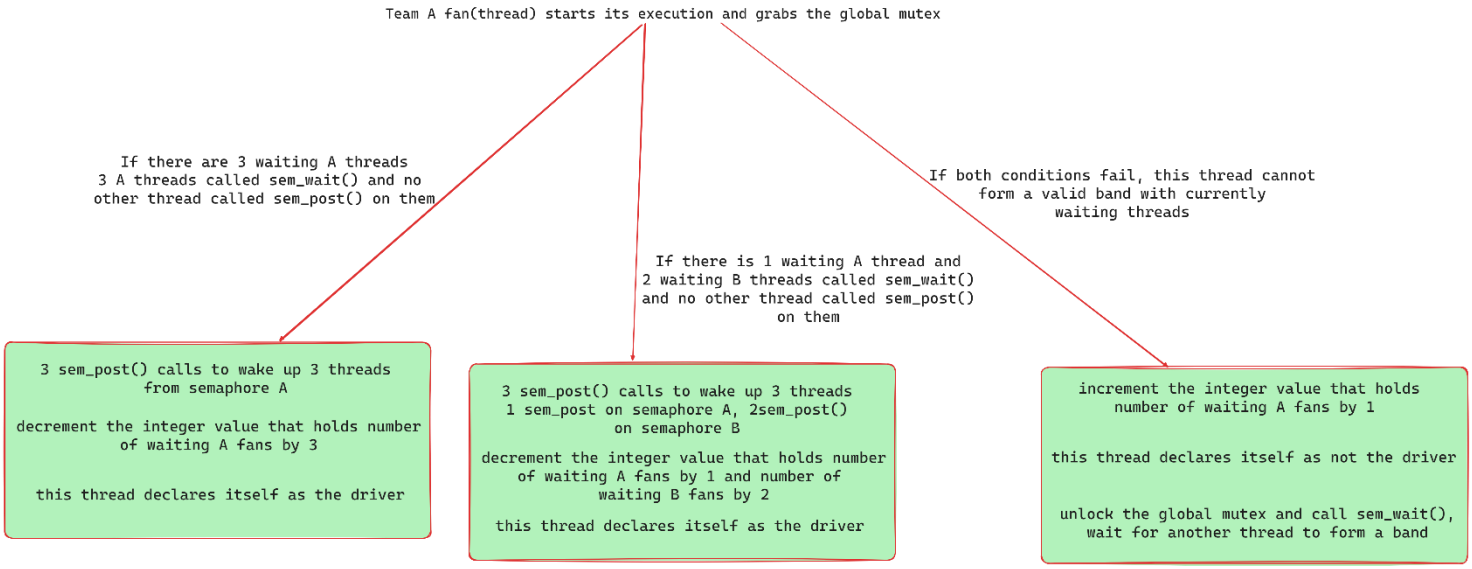
//Note: If any thread reaches this stage, this means that the thread has somehow found a band (does not matter if the running thread is the captain or not)

Print the mid statement: "Thread ID: <TID>, Team: B, I have found a spot in a car.\n"

Wait for all members of the band to print the mid statement: **Usage of barrier**

If the running thread is the captain, **after all threads printed the mid message (which is synchronized by barrier),** state that this thread is the captain, and print the car number as well:
"Thread ID: <TID> Team: B, I am the captain and driving the car with ID <CARID>.\n", then terminate.

If the current thread is not the captain, directly terminates.



Example run of a thread(thread of team A in this case)

Synchronization:

We have two semaphores because of the conditions of band formation: like the producer-consumer example from the lectures, we want to assure that the thread that we wake up is fan of a determined team. Otherwise, we cannot keep track which thread has been waken up, and most probably the threads waken up are not aligning with what we want to do, so deadlocks are inevitable. For each team, a semaphore keeps track of the waiting fans that are ready to form a band. Additionally, the value for the semaphore is set to 0 because initially there is nobody waiting, and we want threads to wait and do not waste CPU cycles when a thread calls sem_wait(). If a thread can form a band with waiting threads, it calls sem_post on the corresponding semaphore(s) so as to wake up the waiting threads, and release execution of them.

In combination with semaphores, two global integer values are used to keep track of the number of waiting threads for each semaphore. This enables the currently running thread to check the number of currently waiting threads for each team, which is crucial to determine if this thread can form a band with waiting threads. Additionally, to prevent data race, a global mutex is used for all fan threads which provides a deadlock-free implementation.

The count for barrier is assigned as 4 since a band consists of 4 fans, and since pthread library barriers are reusable, so no need to refresh it for different bands.

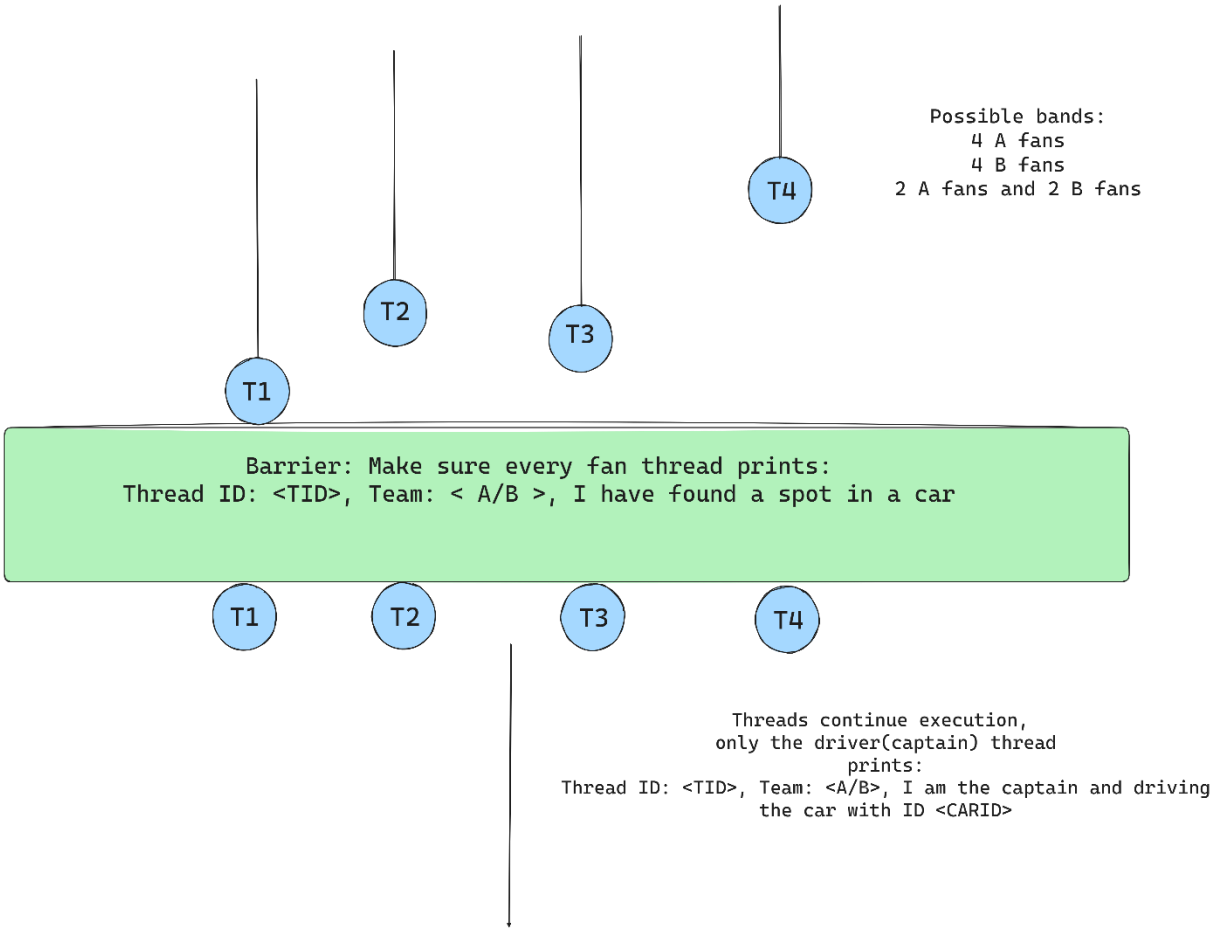


Figure of barrier implementation which provides synchronized outputs

Correctness:

For the main thread, correctness is ensured by checks on two inputs (number of A and B fans). If the validation cannot be satisfied (numbers are not even, or total is not multiple of 4), main thread does not create any children and terminates.

For the fan threads, correctness criteria are as follows:

Number of init strings should be numA + numB: This criterion is satisfied because when a thread grabs the global lock, it prints the init statement firstly. Since main thread waits for all fan threads, we are also sure that each thread runs and grabs the lock once.

Number of mid strings should be numA + numB: This criterion is satisfied because whether a thread is the driver or not, it eventually passes through the checks (used for formation of a valid band, either directly by being the driver or being a passenger which is waken up by the driver), and the mid message is the first thing after the checks. Since main thread checks prevent a thread not to form a band and wait forever (even number of fans from both teams and total number of threads is multiple of 4), every fan thread will come to here and print mid statement.

Number of end and car strings should be (numA + numb) / 4: This criterion is satisfied because if a thread can form a valid band, it declares itself as driver by setting the local Boolean isDriver to true, and after printing the mid message, it goes into the if statement checking if the current thread is driver, and only the driver thread prints end and car string (once in every band – once in every run of 4 threads). If a thread cannot form a valid band, it already has the Boolean set to false, so it does not print the end and car statements.

For each thread, init, mid and end (if exists) must be in this order: This criterion is satisfied because init statement is printed before doing anything. About the order of mid and end, barrier ensures that all fans of a band have printed the mid statement before they continue, so end statement is always later than mid statements for a band.

4 mids, then end, recursively: This is ensured using a global mutex. Each thread locks the global mutex initially. If a valid combination can happen, then the thread wakes up 3 remaining threads to form a group, so that they can continue execution. Since the last thread to form a band declares itself as driver, firstly barrier makes sure that all threads print mid statements, and the driver thread first prints the end and car statements, then unlocks the global mutex, so that no other thread can interleave and print something.

Car id is held as an integer, in each print of end and car statement, it is incremented by 1 for the next band. Again, global mutex makes sure that there is no data race on carid.

Also as a note, number of waiting threads for both fan semaphores are held as two integers named as waitingA and waitingB, which is global and set to 0 initially. Since pthread semaphores do not allow negative *values*, we cannot know the number of waiting fans(threads) for any semaphore, thus these integers are used to keep track of number of waiting fans.