PA3 Report

Riding to a Soccer Game

CS307 - Operating Systems

Mert Ekici 26772
ekicimert@sabanciuniv.edu

I. Introduction

In this programming assignment, a simple Unix command line based ***rideshare*** application should be implemented to serve the fans of soccer clubs. The main aim is to provide a rideshare for the fans when they are going to the stadium, even though they are supporters of the rival teams. To prevent any possible conflict or clash that might occur during the ride, there are predefined rules for the application such as;

- There must be exactly four people to share the riding.
- It is not allowed to share the riding with one person from one club and three people from the rival club.
- In a car, either all 4 people must be the fan of the same club or we must have a 2-2 equality.

Fans are represented by ***POSIX threads (pthread)***. Each fan starts by printing a statement that the fan is looking for a ride to share. Then this fan blocks until there are four threads (fans) that can form a correct combination for a ride. After that, each fan will print that they have found a spot in a car. Then, only one of the fans becomes a driver and states that s/he is the captain who is going to drive the car. While fan threads are running the same method, implemented ***pthread mutexes*** and ***pthread semaphores*** will provide synchronization and the block/wait/signal functionalities of the ***barriers***.

II. Input Validity and Output Correctness

First, the program checks the validity of the arguments. There are two important conditions on the number of supporters which is mandatory to form correct combinations of fans;

1. Each group size must be an even number.
2. Total number of supporters must be a multiple of four.

These conditions guarantee that all team supporters can eventually find a seat in a car where all cars have a valid combination. If the arguments are valid, the main method creates the fan threads based on the given arguments. Otherwise, it terminates without creating any child thread. In both cases, "*The main terminates"*, should be printed at the end (after all children terminate if exist).

As an additional requirement, outputs should fit the specified format and order. These correctness specifications should be followed by the ***main thread*** and ***children (fan) threads***. After the ***input check***, the program should create the correct number of children threads if arguments are valid. The main thread must wait until all children terminate if all children are created successfully.

Specified correctness conditions are given below;

- num_a: The number of Team A supporters
- num_b: The number of Team B supporters
- $init$: The string "Thread ID: ***tid***, Team: ***A or B***, I am looking for a car
- mid: The string "Thread ID: ***tid***, Team: ***A or B***, I have found a spot in a car
- end: The string "Thread ID: ***tid***, Team: ***A or B***, I am the captain and driving the car

1. There must be exactly $num_a + num_b$ $\boldsymbol{init}$ strings printed to the console.
2. There must be exactly $num_a + num_b$ $\boldsymbol{mid}$ strings printed to the console.
3. There must be exactly $(num_a + num_b)/4$ $\boldsymbol{end}$ strings printed to the console.
4. For each thread ***init, mid*** and ***end*** (if exists) must be printed to the console in this order.
5. 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 our program, it is the thread that forms the band or the last thread of the possible combination). Lastly, ***inits*** corresponding to this 4 ***mid*** group (***inits*** by the same threads) must be before the first ***mid*** of this group.

Additionally, I added an extra rule as a challenge to myself which provides better-looking output in any circumstance which requires a more complex implementation and additional ***barrier functionality***.

6. The should not be any ***init*** between ***mids***.

All of the ***semaphores, mutexes*** and ***barriers*** are used in ***rideShare function*** to preserve the format within the rules described above.

III. Functionality (rideShare)

As mentioned above, the relevant problem is a ***concurrency*** problem to be solved with ***barrier functionality*** using ***semaphores***. Multiple fan threads will try to find a ride to the game obeying the given rules. These rules are ensured in the ***rideShare()*** function which will be called by the fan threads. Before explaining the function, let's check the following global variables (integers, mutexes, semaphores and barriers);

- ***int moveCount***: number of threads that are ready to move (this variable is used for releasing the ***barrierForm*** after it reaches 4)
- ***int countA***: count Team A supporters during the Phase I (forming a band)
- ***int countB***: count Team B supporters during the Phase I (forming a band)
- ***int inA***: count Team A supporters inside the car during the Phase II (entering the car)
- ***int inB***: count Team B supporters inside the car during the Phase II (entering the car)
- ***pthread_mutex_t lock:*** mutex lock for ***countA***, ***countB*** and ***barrierMove***

- ***pthread_mutex_t lock2:*** mutex lock for ***moveCount*** and ***barrierForm***
- ***pthread_mutex_t lockW:*** mutex lock for ***write*** operation
- ***sem_t barrierForm:*** barrier for ***form phase***
- ***sem_t barrierEnter:*** barrier for ***enter phase***
- ***sem_t barrierA:*** barrier to count the correct number of A-supporters entered the car
- ***sem_t barrierB:*** barrier to count the correct number of B-supporters entered the car
- ***sem_t barrierMove:*** barrier for ***move phase***

The variables described above are defined and initialized at the beginning of the program. Using ***PTHREAD_MUTEX_INITIALIZER***, ***lock – lock2 – lockW*** mutexes are initialized. For the barrier side, they all implemented using ***pthread semaphores*** and initialized using ***sem_init()*** function. Only ***barrierForm*** and ***barrierMove*** are initialized with the value of ***1*** since they will be acquired during the start of the ***rideShare()*** function. Other ***barriers*** are initialized with the value ***0*** and they will start locked and will be updated during the ***rideShare.*** At the very beginning of the ***main()*** function, ***numerical inputs*** are converted to ***integer*** using ***atoi()*** function into the variables ***numA*** and ***numB***. If inputs are valid; first a ***supporter struct*** is created based on the numbers of each team's supporters and then this struct is passed to the thread with ***rideShare*** function using ***pthread_create()*** in a loop. Lastly, created threads are waited (joined) with ***pthread_join()*** and "***The main terminates"*** is printed when all threads are terminated. Each phase will be explained in detail based on the implementation of the ***rideShare*** function in the next chapters.

IV. Phase I - Form

In the first phase, the program tries to form a valid band to start the process with the active threads. Valid combination includes either 2-2 supporters of each team or 4 supporters of one of the teams. Until valid formation is set, threads are blocked by the barrier functionality. We will investigate the ***rideShare*** function sequentially to have a better understanding of the implementation.

First, the function typecasts the ***member variables*** of the ***supporter struct***: ***team*** and ***isCaptain.*** During the thread creation loop, ***correct teams are assigned*** and all ***supporter structs*** start with the ***0*** value of ***isCaptain*** since we do not know the captain yet. Phase I starts with ***sem_wait(&barrierForm)*** and if it is available (it is not available when there is a band that is currently entering a car) it obtains the ***lock***. After that, ***countA or countB*** is updated based on the ***myTeam variable.*** Next step is printing the output and this piece of code is wrapped with ***lockW (write lock).*** When the valid combination is possible considering previous threads (***by checking countA and countB),*** the function marks the current thread as the ***captain thread (captain is the thread that forms the band)*** just after ***waiting for barrierMove (if the previously formed band is still entering the car wait for them to print their outputs and move).*** After the captain is selected, ***barrierEnter*** is incremented by one and ***barrierA-barrierB*** are updated based on the type of valid combination. If the valid combination is still not formed ***sem_post*** is used with ***barrierForm*** to look for other active threads to check for possible combinations and the ***lock*** is released for the other waiting threads. As a critical remark, ***barrierA*** and ***barrierB*** are created to let the correct number of each team's supporters enter the car. In other words they are used to avoid the following problematic cases;

- Team A, I am looking for a car
- Team A, I am looking for a car
- Team A, I am looking for a car
- Team B, I am looking for a car
- Team A, I am looking for a car

- Team B, I am looking for a car
- Team B, I am looking for a car
- Team B, I am looking for a car
- Team A, I am looking for a car
- Team A, I am looking for a car

When the outputs above are printed, the program should not let the supporters enter the car without checking which team they support. For example, if we just enter 4 supporters after one of the cases above, we might end up with a car of 3 A supporters and 1 B supporter or vice versa.

V. Phase II – Enter

For the second phase, the function first checks the membership of each supporter to ensure we have a correct number of supporters for each team in the car because of the problematic cases I mentioned above. Since ***barrierA*** and ***barrierB semaphores*** have the correct number of ***resources*** to ***release*** depending on the correct combination (2-2 or 4-0 or 0-4), ***sem_wait(&barrierAorB)*** will be executed without waiting for the correct number of times. After the ***team check***, the thread consumes ***barrierEnter*** with ***wait*** which was incremented in **Phase I.** Following step is the ***enter prompt with lockW***. Then, ***countA or countB*** are decremented based on ***myTeam*** and relevant ***inA or inB*** is incremented. This step basically moves the thread to the car and decreases the ***count of supporters that are looking for a ride***. If all the correct threads are inside of the car ***sem_post(&barrierMove)*** is called for the initialization of ***Phase III – Move (Captain).*** Otherwise, ***sem_post(&barrierEnter)*** is called and it signals to let next member of the valid band enter to the car.

VI. Phase III – Move (Captain)

In the last phase, each thread starts to move and if the thread is captain (***isCaptain***), it declares that s/he is the captain that will be driving the car. This phase starts with ***sem_wait(&barrierMove)*** which can be activated when each member of the valid combination band has entered the car. If the signal is sent, thread acquires the ***lock2*** which is specifically created to atomize the ***increment*** operations for ***moveCount*** and ***barrierForm.*** After locking the mutex, function checks the thread if it is the ***captain*** of the ride. If the thread is the captain, it prints the relevant output by getting the ***lockW*** and if not, function continues.

Next, ***moveCount*** is incremented and if it equals to 4 after the increment, ***moveCount*** will be reset to ***0*** and ***barrierForm*** is released (***form phase is available again*** for the other waiting threads). Lastly, ***sem_post(&barrierMove)*** is called and now another candidate captain thread can form the band (without this barrier ***5th rule*** might be violated – consecutive ***ends and mids*** could be tangled).