Programming Assignment (PA) - 3
Riding to a Soccer Game

CS307 - Operating Systems (Fall 2022)

20 November 2021
DEADLINE: 06 December 2021, 23:55

1 Introduction

As you know, the way of public transportation is changing with the brand new applications that allow people to travel more easily, cheaply and safely. In order to get a share of this huge cake, you are required to develop a simple rideshare application. This application is designed to serve to the fans of soccer clubs. So basically, it is aimed to provide a rideshare to the fans when they are going to the stadium, even though if they are arch rivals. Therefore, you need to be careful at the point of sharing i.e there must be a rule to allow a certain kind of combinations of the fans in order to prevent from clash of these arch rivals :).

In this programming assignment you are asked to implement simple, Unix command line based rideshare application for the fans of soccer clubs. For simplicity, we assume that there are two rival soccer clubs. As stated above, there are some rules at allowing the passengers when they want to share a ride to the stadium. First of all, there must be exactly four people to share the riding. Moreover, it is not allowed to share the riding with one person from one club and three people from the rival, which is a basic rule to avoid clashes among rivals. In a car, either all 4 people must be the fan of the same club or we must have a $2 - 2$ equality.

You will use *POSIX threads (pthread)* to represent the fans. Each fan (or the thread in this case) starts with printing a string stating that the fan is looking for a ride share. Then s/he blocks until there are four threads with the correct combination. After that, each fan will print that s/he has found a spot in a car. Then, only one of the fans (does not matter which of them) becomes a driver and s/he will print a string to state s/he is the captain of the car and s/he will be driving. More technical details and the requirements are given below.

2 Threads

You are required to create a POSIX thread (pthread) for each fan. Fans of each club might run the same method or you might implement different methods for different club supporters. The choice is yours. In addition to fan threads, there will be a main thread creating fan threads.

2.1 The Main Thread

The main thread, which is the thread of the process created by the operating system when the application starts, runs the main method.

The main method will require two arguments to start and function properly. Since there are two groups (the clubs) of fans there will be two integer arguments representing the number of fans of each group. These arguments will be program input when called from the console. An example call of your program is as follows:

```
$ ./ridesharing 2 6
```

where `ridesharing` is your executable (compiled version of your source code), 2 and 6 are the number of Team A and B supporters, respectively.

In the main method, you have to check validity of arguments as well. There are two conditions on the number of supporters:

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

These conditions guarantee that all supporters will eventually find a seat in a car and all cars will have a full valid configuration.

If arguments are valid, then the main method creates the fan threads according to the given arguments. Then, it waits until all of the fan threads terminate. Otherwise, the main thread terminates before creating any child thread. In both cases, it must print `The main terminates` at the end (after all children (if exists) terminate).

2.2 Fan Threads

Below, you can find a typical behaviour of a Team A supporter:

- It first prints: `Thread ID:` $< tid >$`, Team: A, I am looking for a car`.
- If there are 3 more Team A supporters, or 1 more Team A supporter and at least 2 Team B supporters, then a valid combination is possible. As you can guess, a valid combination consists of either 4 Team A supporters or 4 Team B supporters (which is never the case for a Team A supporter at this point) or 2 Team A and 2 Team B supporters. Then, this thread forms a valid band of 4 and wakes the remaining 3 members of the band. Otherwise (if there is no possible valid combination), it waits until a valid combination is formed and the last thread forming the ride share band wakes it up.
  - **Optional:** If this thread is the last one forming the ride share band, it can declare itself as the captain (driver). Of course, there are other ways of choosing a captain. The choice is yours. The only important point is that there must be exactly one captain per ride share band.

- At that point, this thread must be a member of a valid ride share band. Then, it prints `Thread ID:` $< tid >$`, Team: A, I have found a spot in a car`.
- The thread waits until all of the other members of the ride share band completes the previous step.
- At that point, all the other 3 members of the band are in the car. If this thread is the unique captain, then it prints `Thread ID:` $< tid >$`, Team: A, I am the captain and driving the car` and terminates. Otherwise, it terminates doing nothing in this step.

As you can see from the above description, a fan thread's execution consists of three phases. In the first phase, it requests a ride share and waits until a valid combination is possible. In the second phase, it gets into the car and waits until the remaining three band members join it. In the last phase, it starts driving the car if it is the captain, otherwise it does nothing.

The behaviour of a Team B Thread is similar and can be inferred from the description above.

3 Implementation Details

As indicated above, you will use POSIX threads. In order to handle the synchronization issues you can use pthread semaphores and pthread barriers or you can implement your own barrier.

The pthread semaphores behave like Linux Zemaphores, not like the original Dijkstra semaphores we have seen in the lectures. So, before using the pthread semaphores, take a look at the manual page of the command `sem_wait` or check the web page : `https://linux.die.net/man/7/sem_overview`. If you wish, you can implement Dijkstra semaphores using pthread condition variables and mutexes as in the lectures. The choice is yours. In the report you will provide in the submission package, you have to explain which synchronization primitives you used and how you implemented them.

4 Correctness

There are correctness specifications for both the main thread and the children (fan) threads.

The correctness of the main thread is mostly explained in the Section 2.1. First, it should check validity of input arguments as described. If arguments are valid, then it should try creating the correct number of children threads. You do not have to consider the case when a child thread creation fails. However, you have to ensure that the main thread waits until all children terminates if all children are created successfully. The console prompt `The main terminates` by the main thread must always be printed as the last thing to the console.

Correctness of the fan threads depends on the order and interleaving of strings they print to the console. To make things easier, let us give strings they print to the console at various steps and number of supporters some names:

- num_A: The number of Team A supporters
- num_B: The number of Team B supporters
- $init$: 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`

Then, for any execution of your program with valid inputs, the following conditions must be satisfied:

- There must be exactly $num_A + num_B$ $init$ strings printed to the console.
- There must be exactly $num_A + num_B$ mid strings printed to the console.
- There must be exactly $(num_A + num_B)/4$ end strings printed to the console.
- For each thread $init$, mid and end (if exists) must be printed to the console in this order.
- 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 addition, teams of these mids must be a valid combination. Lastly, $init$s corresponding to this 4 mid group ($init$s by the same threads) must be before the first mid of this group.

See Section 5, for some sample output obeying the correctness conditions above.

5 Useful Information & Tips

- First, read this document from beginning to the end. Make sure that you understand what is the problem you need to solve and what is expected from you. You can mark important points and take some notes in this step. Then, develop your solution using pen and paper (maybe by writing an abstract pseudo-code). Then, start implementing the solution considering tips in this section and the grading section. Complete one grading item at a time obeying the preconditions. Make sure that your improvements and refinements do not violate previously completed grading items.
- Take a look at the behavior of the pthread semaphores before using them.
- Ensure that your application works correctly in the case that there are more than one ride share bands like two threads for A and six threads for B.
- Your program should terminate properly, you will lose points if your program can not terminate all the threads safely or in case of missing outputs.
- You have to use pthreads (POSIX Threads) library and submit a C or C++ file. Any other thread library will not be accepted as a solution.

- Do not forget to use *-lpthread* option while compiling.
- In the `flow` machine, C and C++ compilers' default standards are quite old (like 1986). They will not compile semaphore or barrier libraries if you call the compiler without changing the library standard. You have to execute a command like this:

```
gcc rideshare.c -std=gnu99 -lpthread
```

- Under MacOS, `sem_init` method does work properly and it is not supported. You are advised to use `sem_open` and `sem_close` methods instead of `sem_init` and `sem_destroy` methods. See the following stack overflow page fore details: `https://stackoverflow.com/questions/1413785/sem-init-on-os-x`

Submission

You are expected to submit a zip file named *<YourSUUserName>_PA3.zip* until 25 April 2021, 23h55.

The content of the zip file is as follows:

- **report.pdf**: In your report, you must present the flow of your threads as a pseudo code. You discuss which synchronization mechanisms (semaphores or condition variables or any other logical way) you have chosen, how you implemented, used or adapted them to suit your needs and provide formal arguments on why your code satisfies the correctness criteria described above.
- **rideshare.c** or **rideshare.cpp**: Your C or C++ implementation file.

Grading

Some parts of the grading will be automated. If automated tests fail, your code will not be manually inspected for partial points. Some students might be randomly called for an oral exam to explain their implementations and reports.

Submissions will be graded over 100 points:

1. **Compilation (10 pts):** Your program compiles without giving an error. When we run the program, we do not get any runtime errors as well.
2. **Command Line Input (10 pts):** Your program must take numbers of supporters of both teams as inputs from the command line. If your program runs using a hard-coded fixed number of threads or tries to take input calling `scanf`, you will not get any points from this part. Moreover, your program must check the validity of the input and terminate immediately in case it is not valid.
3. **No Deadlock (10 pts):** For valid input, your threads never block due to a deadlock. All fan threads and the main thread runs to completion printing their lines to the console.
4. **Ordered Termination (10 pts):** The main method waits termination of all children threads and always prints `The main terminates` as the last thing to the console. Of course, for a valid input, we expect usual fan thread lines before this line although they do not have the correct order among themselves.
5. **No Problem with One Band (15 pts):** Your program must work correctly when there are threads just enough to form a single ride share band e.g. four threads for group A and zero threads for group B.
6. **No Problem with Multiple Bands (25 pts):** Your program must work correctly when there are threads just enough to form two, three or four ride share bands e.g. four threads for group A and eight threads for group B.

7. **Report (20 pts):** Your report explains your thread methods, how you implemented them, what kind of synchronization mechanisms you used and adapted and why you think that it is correct (-5 pts if your report is not in pdf format).

For each integer $i \in [1, 5]$, item i is a precondition for item $i + 1$.

Sample Output

Due to random nature of threads and random generator, your output could not exactly be same however there is an order among the strings that must be satisfied. As you see below the sample outputs there must be exactly four strings stating the fans have found a spot in a car before the captain runs the car. Order of the strings about the spot among themselves is not important for sure, however there must be exactly four before starting the rideshare.

Sample Run 1: $ ./main 2 2

```
Thread ID: 139889192867584, Team: A, I am looking for a car
Thread ID: 139889184474880, Team: A, I am looking for a car
Thread ID: 139889176082176, Team: B, I am looking for a car
Thread ID: 139889098094336, Team: B, I am looking for a car
Thread ID: 139889176082176, Team: B, I have found a spot in a car
Thread ID: 139889192867584, Team: A, I have found a spot in a car
Thread ID: 139889184474880, Team: A, I have found a spot in a car
Thread ID: 139889098094336, Team: B, I have found a spot in a car
Thread ID: 139889098094336, Team: B, I am the captain and driving the car
The main terminates
```

Sample Run 2: $ ./main 2 4

```
The main terminates
```

Sample Run 3: $ ./main 2 6

```
Thread ID: 139669941450496, Team: A, I am looking for a car
Thread ID: 139669907879680, Team: B, I am looking for a car
Thread ID: 139669899486976, Team: B, I am looking for a car
Thread ID: 139669924665088, Team: B, I am looking for a car
Thread ID: 139669933057792, Team: A, I am looking for a car
Thread ID: 139669933057792, Team: A, I have found a spot in a car
Thread ID: 139669907879680, Team: B, I have found a spot in a car
Thread ID: 139669899486976, Team: B, I have found a spot in a car
Thread ID: 139669941450496, Team: A, I have found a spot in a car
Thread ID: 139669933057792, Team: A, I am the captain and driving the car
Thread ID: 139669916272384, Team: B, I am looking for a car
Thread ID: 139669891094272, Team: B, I am looking for a car
Thread ID: 139669882701568, Team: B, I am looking for a car
```

```
Thread ID: 139669882701568, Team: B, I have found a spot in a car
Thread ID: 139669924665088, Team: B, I have found a spot in a car
Thread ID: 139669891094272, Team: B, I have found a spot in a car
Thread ID: 139669916272384, Team: B, I have found a spot in a car
Thread ID: 139669882701568, Team: B, I am the captain and driving the car
The main terminates
```