Dance of Threads: Concurrent Programming with Disco Clients

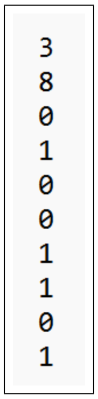
Itziar López Almagro

18/01/2024

1 Description of the program.

I chose to do Project 3 (Concurrent Program with Condition Varible). This is the description of the program I implemented:
We aim to develop a nightclub capacity control program. At this venue, there is a specified limit denoted as N for the maximum number of individuals allowed inside. Additionally, there are two types of clients: VIP clients and ordinary clients. The system should adhere to the following rules:
1. When the number of individuals in the nightclub reaches the maximum allowed limit (N), new clients must wait to enter.
2. VIP clients waiting in line will be given priority over ordinary ones. Ordinary clients will have to wait until all VIPs have entered.
3. Clients will enter one by one in a strict order based on their arrival to the queue, whether they are VIP or ordinary clients.
The system is required to generate M threads, each representing a client in the nightclub. Each thread should store two pieces of data: an identifier (id) based on the order of creation and a boolean value (vip) indicating whether the client is a VIP (1) or not (0).
The program will take a filepath as a command-line argument which will contain a textfile with this format:
1. The first line will contain the maximum number of persons allowed in the discoteque (N).
2. The second line will contain the number of clients the program will have to create (M).
3. The following M lines will take the values 0 or 1 for indicating if that client is VIP (1) or if is not (0).

Example input file:



```
3
8
0
1
0
0
1
1
0
1
```

Figure 1: Input file named 'example.txt' included in the zip.

2 Development enviroment and libraries.

I opted for Eclipse as my development environment for working with the JavaScript language. The packages I used where the following ones:
1.java.util.concurrent.locks.ReentrantLock.
2.java.util.concurrent.locks.Lock.
3.java.util.concurrent.locks.Condition.
4.java.util.Random.
5.import java.util.Scanner.
6.import java.io.File.
7.java.io.FileNotFoundException.

3 Implementation of the program.

In this section I will explain what resources I used and why. Additionally, I will explain the implementations of every class and function.

3.1 Class Disco

This is the class which contains the main. The main will read the data contained in the file and will create all the threads.

```
public static void main(String[] args) throws FileNotFoundException {
  Scanner sc = new Scanner(System.in);
  System.out.print("Enter the path of your textfile which contains the following information:");
  System.out.print("The first line must have the capacity of the disco.");
  System.out.print("The second line must have the number of clients that will be created (M).");
  System.out.println("The following M lines will indicate wether that client is vip (1) or not(0)");
  String path =sc.nextLine();
  Scanner scanner = new Scanner(new File(path));
  int N= scanner.nextInt();//capacity of the disco
  int M = scanner.nextInt();//number of clients that the program will have to create
  Client.N=N;
  for(int i=0;i< M;i++) {
        Client c= new
        Client(scanner.nextInt()==1,i+1);
        c.start();
   }
   sc.close();
   scanner.close();
 }
```

3.2 Class Client

This class extends from the class Thread and fields contained in are the following ones:

```
 private boolean vip;
 private int id;
 public static int N;
 private static int num_per=0;
 private static int num_vips_q=0;
 private static Lock mutex= new ReentrantLock();
 private static Condition q_vips=mutex.newCondition();
 private static Condition q_clients=mutex.newCondition();
```

- The field 'id' stores the identifier of the thread.
- The field 'N' stores the maximum number of persons that can be inside the discoteque.
- The field 'num_per' counts the numer of clients inside the discoteque.
- The field 'num_vips_q' counts the number of vips clients waiting in the queue.
- The field 'mutex' will control the access to the variables 'num _vips _q' and 'num _per'.
- The field 'q _vips' is the conditional variable associated with the mutex. It will 'sleep' the threads of the VIP clients until they have chance of coming in the discoteque.
- The field 'q _clients' is the conditional variable associated with the mutex. It will 'sleep' the threads of the ordinary clients until they have a chance of coming in the discoteque.

Every class that extends from the class Thread should have implemented the function *run*(). This is the function *run*() of the class Client:

```
    public void run() {
      try{
        if(vip)enter_vip_client();
        else enter_normal_client();
        dance();
        exit_client();
        }catch(InterruptedException e) {}
    }
```

The function *enter_normal_client*() is the following one:

```
public void enter_normal_client() throws InterruptedException {
    mutex.lock();
    System.out.println("The client " +id + " is waiting for getting in the disco");
    while(num_per== N || num_vips_q>0) q_clients.await();
    System.out.println("The client " +id + " has just entered in the disco");
    num_per++;
    mutex.unlock();
 }
```

This function will be executed by the clients (or threads) that are not VIP when they are willing to enter to the discoteque. The first instruction is lock the mutex. Once the thread owns the mutex, it will check if number of clients inside the disco is equal to the maximum capacity. If we find that this last condition is true, we will have to wait. In addition, if there are vip clients in the queue this client will have also to wait. When the thread has entered in the discoteque we will increase the number of clients and unlock the mutex to let other threads access.

This is the function *enter_vip_client*() :

```
 public void enter_vip_client() throws InterruptedException {
    mutex.lock();
    System.out.println("The vip client " +id + " is waiting for getting in the disco");
    num_vips_q++;
    while(num_per== N) q_vips.await();
    num_per++;
    num_vips_q--;
    if(num_vips_q==0 && num_per<N)q_clients.signal();
    mutex.unlock();
    System.out.println("The vip client " +id + " has just entered in the disco");
  }
```

This function will be executed by the threads that are VIP when they are willing to enter to the discoteque. Once the thread owns the mutex it will raise one unit the variable 'num _vips _q' . While the number of individuals in the nightclub equals the maximum capacity, the thread will enter a 'sleep' state until the conditional variable 'q _vips' signals the thread that this condition is no longer true. Then ,when the thread has succesfully entered the discoteque, the thread will raise the number of persons in the discoteque 'num _per' and low the variable 'num _vips _q'. In addittion, if there are not more vips in the queue, this thread will warn one thread in the ordinary client's queue that probably now would be able to come in.

This is the function *dance*():

```
public void dance() throws InterruptedException {
    System.out.println("The client "+ id + " is dancing in the disco ");
    int m_seconds=new Random().nextInt(3) + 1;
    m_seconds=m_seconds*1000;
    Thread.sleep(m_seconds);
  }
```

This function will be executed when the threads have successfully entered in the discoteque. The thread will sleep a random number of seconds between 1 and 3.

This is the function *exit_client*():

```
public void exit_client() {
   mutex.lock();
   System.out.println("The client " +id + " got out of  the disco");
    num_per--;
    if(num_vips_q >0)q_vips.signal();
    else q_clients.signal();
     mutex.unlock();
  }
```

This function will be executed after the threads have completed their dance. Once the thread acquires the mutex, it will decrement the variable 'num_per' by one unit. The thread will also signal other threads (using *signal*()) to notify that this client has left, allowing another client to enter the discotheque.

4 Why do we need two condition variables?

These are the reasons why we need two condition variables:

- We need two queues: one for VIP clients and one for normal clients.
- Priority will be given to VIP clients over regular ones; however, we must consider the arrival order of clients within both the VIP and regular queues.
- As long as there are VIP clients who have not entered the discotheque, regular ones will be unable to gain access. It is inefficient for these threads to compete for the mutex under these circumstances.

5 How the program is deadlock and starvation free?

These are the reasons why my program is deadlock free:

- Each function in which the thread requests the mutex initiates only one lock() operation followed by one *unlock*(). Additionally, we employ a Reentrant Lock, ensuring that it is impossible for the thread to block itself.
- When a client is leaving the discotheque, they will consistently notify other clients. This action will awaken all threads, providing each thread with one successfull entrance to the discoteque.

These are the reasons why my program is starvation free:

- The use of condition variables will help to signal waiting threads when the resource becomes available, preventing any single thread from being consistently bypassed.
- The use of the mutex will also help to establish a fair order in which threads can access the protected resource.

Moreover, the employment of mutex ensures the protection of shared data ('num_pers' and 'num_vips_q'), thereby preventing any occurrence of race conditions throughout the code.

6 Optimizations.

Initially, I believed that employing two mutexes—one to safeguard the variable 'num _pers' and other for protecting the variable 'num_vips _q' was a sound approach. However, upon further consideration, I concluded that this strategy didn't yield significant optimization benefits. In fact, it seemed more like an unnecessary resource expenditure, adding unnecessary complexity to the code.

7 Conclusion.

I am confident that this code will not result in any race conditions or undesirable interleaving. The incorporation of two condition variables has enhanced code clarity and facilitated implementation. Without their use, the code would have been notably challenging, perplexing, and less efficient. This approach ensures a more robust and streamlined execution.