Concurrent

Programming

Assignment

Report

Bishwendra Chaudhary (150101017)

Tamilselvan S (150101079)

Problem 1: Sock Matching

Classes implemented:

1. Sock:

* This class represents the sock object.
* The class has one attribute: colour(String) and two methods:   
  getColor () and getlock ().
* The class also has a Reentrantlock (var name: lock) to implement synchronization.

1. Heap:

* The Heap is a single instance class.
* This class has one attribute: heap (List<Sock>) and three methods:

getInstance (): returns the instance of the class.

Shuffle (): Shuffles the heap using Collections.shuffle().

Push (): Pushes a sock object into the heap.

* The constructor of this class initializes heap to be a new ArrayList<>

1. Arm:

* This class extends Thread class and implements abstract class runnable.
* The objects of Arm have one attribute: id(int).
* The Arm implements run () by overriding the run () method in Runnable class.

Description of run ():

* + While the index I does not reach heap.size (), the arm tries to get a lock on ith element in the heap.
  + If getting a lock is successful, the sock is pushed to Matcher.heap using ‘synchronized’ keyword on Matcher.heap. The index is incremented
  + Else the index is incremented
  + The run () stops when index reaches end of heap.

1. Matcher:

* The Matcher is a single instance class which extends thread and implements runnable.
* This class has 9 attributes:

1. Heap(List<Sock>)
2. white, black, blue, grey (int)
3. tot\_white, tot\_black, tot\_blue, tot\_grey (int).

* There are 3 methods:

1. getInstance(): returns the instance of the class.
2. Push (): pushes a sock object to the heap.
3. Run ():
4. while true, get a sock object from heap list and check its color with the given colors(white,black,blue,grey), increase increment white, black, blue, grey accordingly. Whenever the count for a color reaches 2 then reinitialize the color count to 0 and increment tot\_white, tot\_black, tot\_blue, tot\_grey accordingly. If heap is empty increment status.
5. If status reaches 10 the while loop breaks and run() prints the values of tot\_white, tot\_black, tot\_blue, tot\_grey.
6. Main:

* The Main class has one method: main ().

Description of main ():

* + There are 5 variables no\_of\_arms, white,black,blue,grey.
  + The main () takes input of no. of arms, and no. of socks in each color.
  + The socks of each color are pushed to Heap.heap .
  + The Heap.heap is shuffled to maintain randomness in picking socks from heap.
  + Each arm object is initialised using constructor and run using start() method.
  + The Matcher class is run using Matcher.getInstance().start() .

1. The role of concurrency and synchronization in the above system.

* Since each robotic arm has to function independent of each other the arms have to run in parallel. In a multi-core system running several arms in parallel reduces the time to empty the heap. This system is implemented using threads which introduces the concept of concurrency i.e. running multiple instructions simultaneously.
* If concurrency had not been implemented, the entire heap will be handled by one arm. Since the entire program runs on a single thread the program can’t advantage of multicore system (i.e. running multiple instructions simultaneously)
* Since multiple threads of arms are running simultaneously there is a need for synchronisation while accessing shared resources such as Heap.heap and Matcher.heap
* If synchronization had not been handled multiple arms can access the same sock and push the sock multiple times to the Matcher.heap.

1. How did you handle it?

Concurrency has been handled by using threads to implement the arms and matcher.

Synchronisation has been handled at multiple places:

1. Applying lock on sock object in Heap.heap so that no other thread accesses it in the future.
2. Synchronised keyword has been used when pushing a sock to Matcher.heap so that at a time only one thread can push a sock to the heap.

Problem 2 – Data Modification in Distributed System

Classes implemented:

1. Entry: Contains the description of each entry in the file.
2. FileClass:
   * This is a single instance class
   * Contains a hashmap of entry objects representing the iput file. The key is of type String and represents the roll no of the student.
   * There are two methods implemented:   
     load() which loads the input file data to the hashmap.  
     unload() sorts the hashmap according to rollno and name write it into two new text files simultaneously.

1. Command:
   * Command class extends thread and implements runnable
   * The object of this class runs a command of incrementing or decrementing a student’s marks in either synchronised or non-synchronised mode.
2. Main:
   * Creates an array of commands based on input of no. of commands.
   * If synchronization is chosen the commands are run by synchronising the entry object to be modified else run otherwise.
   * Once show output option is chosen the unload() method of FileClass generates two required files(sorted based on rollno and names).
3. Why concurrency is important here?

If concurrency is not handled no two teachers can have access to the database in real time. The user commands might have to wait in queue to execute sequentially which is inefficient when two commands are accessing different entries.

1. What are the shared resources?

The entries in FileClass is a single instance class accessed by multiple commands.

1. What may happen if synchronization is not taken care of? Give examples.

If synchronisation is not handled the modified data has a very high risk of becoming inconsistent.

Ex: Suppose TA1and TA2 access the same entry whose marks are 5.

TA1 wants to increase the marks by 2(Thread t1) and TA2 wants to decrease the marks by 1(Thread t2).

The sequence of execution may be of one of the following orders.

Case1:

t1: read data  5

t2: read data  5

t1: new data  7

t2: new data  4

t1: write data  7 (new data)

t2: write data  4 (new data)

result: 4  inconsistent

Case2:

t1: read data  5

t2: read data  5

t1: new data  7

t2: new data  4

t2: write data  4 (new data)

t1: write data  7 (new data)

result: 7  inconsistent

Case3:

t1: read data  5

t1: new data  7

t1: write data  7 (new data)

t2: read data  7

t2: new data  6

t2: write data  6 (new data)

result: 6  consistent

Case3:

t2: read data  5

t2: new data  4

t2: write data  4 (new data)

t1: read data  4

t1: new data  6

t1: write data  6 (new data)

result: 6  consistent

We can see that two of the four cases result in an inconsistent data hence each entry needs to accessed by only one thread at a time.

1. How you handled concurrency and synchronization?

Concurrency has been handled by running each command as a separate thread.

Synchronization has been implemented by synchronising the entry so that no two commands access the same entry at a time.

Problem 3 – Room Delivery Service of Tea/Snacks

In this problem there are two types of users – client and server.

Client.java -

Classes implemented: -

Customers class - First customers try to connect with a localhost through 5000 port number using a socket. If he is connected to server then he will be able to order.

Server file.java -

Classes implemented -:

Server class – Intially server is waiting for a client, when client try to connect with server then server accept is request and store the data sends by client into DataInputStream object.

And send acknowledgement to client.

Run thread for each client which maintain the concurrency between clients.

Stock class – basically used to maintain the current stock.

This is single instance class.

There are 3 methods for 3 items respectively.

We used synchronized keyword for each method of this class.

OrderQueue class – here we follow FCFS technique as mentioned in the question.

This is single instance class.

We push order in a list according to arrival time. Execute the order whose arrival time is early, irrespective of the time taken to complete.

Order class – order class extends Thread implements Runnable.

Now, in run ()

1) If it is not the head of OrderQueue sleep for infinite time.

After completion of the order thread, it is popped from the list and the thread revives the next order in the queue by using thread interrupt.

After completion of order send the invoice to client.

ItemQueue class – this class has arraylist<Item> which contains all items of a current thread order.

Single instance class.

Run a thread for each item (concurrency).

Item class - this class is “extends Thread implements Runnable”.

run() method of this class helps to check whether the required quantity of an item is present in stock or not ?

If Yes, then get it.

else print given item not available.

ClientHandler class – this class is extended by a thread class.

Here we push all the details of the thread order into Order list.

Admin: Creates an admin interface for the server so the admin can monitor the sales. When any of the stocks get low i.e. lesser than 10 the remaining quantity is highlighted in red.

a) What role concurrency plays here?

- Here Concurrency is used to process items in an order in parallel i.e. The server can prepare tea and coffee in parallel

b) Do we need to bother about synchronization? Why? Illustrate with example.

Yes. Each order has been executed sequentially (FCFS)so that no two threads access the stock at the same time.

If no:

There is a chance that the stock shows “available”

for two order threads but has stock to serve only one thread.

c) How you handle both?

We have used synchronised methods to consume the stock so that no threads can call these methods at the same time.

We have run all items in separate threads so that concurrency is maintained.