Programming Languages Lab

Concurrent Programming Assignment Report

Neel Mittal
Roll - 150101042

Prashansi Kamdar
Roll - 150101047

Sock Matching Problem

- Concurrency and Synchronisation Issues:

  Arms picking up same socks : The need of concurrency arises in the system as the picking up of different socks by different arms from the heap should be allowed. But the same sock should not be picked up by two different arms and hence the need of synchronisation.

  Shelver and Matcher running with no socks : We also need that the matcher and shelver doesn't try to match nothing and shelf nothing respectively, i.e, socks need to be present in the matcher for the matching action to happen and a pair of socks need to be present in the shelver for the shelving action to happen. If the matcher is run with no socks present, it is just going to waste time, similarly for the shelver.

- How the solution is modeled :

  Sock Heap :
  Array of Integers (Each integer signifying a color)

  Arm :
  Multiple threads which randomly generate a number which is the index of a sock in the array, pick up this sock and give it to the matcher thread by use of a blocking queue.

  Matcher :
  Thread which sees socks in the matcher queue, make pairs of the socks and gives it to the shelver by use of another blocking queue.

Shelver :

Thread which takes a pair from the shelver queue and puts it in the correct shelf (variables which count the number of pairs observed).

- Resolution for the problems :

Arms picking up same socks : For allowing concurrent picking up of socks from the heap, we have taken an array of integers (each of the integers signifying one of the four colors). Each arm Thread randomly chooses an index of the array to pick a sock from. If the index is empty(doesn't contain a sock), the arm repeats the process. For synchronisation, we have used the synchronised keyword to lock an index on the array, thus, no two arms will be able to pick up the same sock. If two arms get the same index from the random call, then either one of the arms will be able to acquire a lock on the index. Once an arm gets a lock, the other arm has to wait until the first arm picks up the sock and sets the array at that index empty. Then the first arm will release the lock and the array value at the index will be empty. The second arm will thus not be able to pick up the sock and the issue is resolved.

Shelver and Matcher running with no socks : The problem of scheduling the matcher thread and shelver thread with no socks present is resolved by using blocking queues. There are two blocking queues. One for the matcher and one for the shelver. The arms pick up the socks and insert them in the matcher queue, and the matcher matches the socks from this queue. It then adds the pair to the shelver queue, and the shelver uses this queue to put pairs of socks into corresponding shelves. Blocking queue has the functionality that it blocks the threads trying to pop values when the queue is empty, thereby preventing wastage of time in scheduling threads which are anyway waiting.

Data Modification in Distributed System

- Concurrency and Synchronisation :

The record level modification for two different records can be done simultaneously by two Teachers and both the modifications should be reflected in the final records. Due to this nature of the updation we need to allow concurrent modifications. This data will thus need to be synchronised as well for correct execution.

- How the scenario is modeled :

  Record:

  A class containing all the data present in one record corresponding to a roll number. This class has a reentrant lock object inside it. This lock is always requested by threads before any modification of the records take place. Thus, every record has its own lock.

  Teacher :

  Thread that modifies the records. It can call for two types of modifications, Record Level and File level.

  Records:

  Hashmap of records that is the record level database that is changed when record level modifications are called. Since, this is a collection of record objects and every record object has handled synchronisation issues internally, it would allow multiple threads altering different records simultaneously but changes to the structure of hashmap is not synchronised. But, this is not an issue since CC and TA threads only modify existing records.

  Files:

  A class containing the methods to change the file level records.

- Shared Resources :

  File objects and record objects are to be used/modified by the threads. We also have a map of records, which is passed to each thread for modification and hence,it is also a shared resource. The file objects corresponding to sort_name.txt and sort_roll.txt files are also used by the threads for writing their changes to the disk and hence are shared resources.

- Problems Without Synchronisation :

  If synchronisation is not taken care of some of the operations might go unnoticed for threads.

  Example 1 : An example can be this sequence of execution by two threads (**T1**, **T2**) where **T1** tries to increase 4 marks for roll 174101055 and **T2** decreasing 2 marks for the same record.

Problematic Sequence (Record Updation):

T1: Get map entry record corresponding to roll no 174101055
T2: Get map entry record corresponding to roll no 174101055
T1: Increase marks from 75 to 79
T2:Decrease marks from 75 to 73

In the above sequence **T2** doesn't get the updated value of the record and hence on its own update of the record, the initial modification by **T1** is ignored. For a synchronised update, the record will hold 77 marks and both the updates will be appropriately taken care of.

Example 2 : Another example can be seen while dealing with file_write operations by two threads.

Problematic Sequence (File updation):

T1: Opens file1(sort_name.txt) to write (without append mode).
T1: Writes Record 1 to file1
T2: Opens file1(sort_name.txt) to write (without append mode).
T2: Writes Record 1 to file1
And so on.

In this the **T2** is overwriting the modifications done by **T1** to the file1. Hence, we need to synchronise the file updates.

- Resolving Concurrency and Synchronisation Issues:

For concurrency, we have used a map of records and passed to each thread by reference and hence they can access the records in this map if they are updating two different records simultaneously. This is handled by creating a map of the records and allowing threads to lock the record they are updating before accessing it. Thus, if another thread tries to access a record that thread1 is modifying, it will wait until the complete critical section for **T1** is executed. But if the thread wants to update a record which is not being modified by any other thread, it can do so by accessing that map entry and locking it before modification.
For synchronisation, we have used Reentrant Locks in the objects Record and Files. Reentrant lock is different from the normal lock as it gives more preference to the thread waiting the longest on the resource. When a thread is changing a record, it first locks that record, so that no other thread can modify the record at the same time.

Room Delivery Service

- Concurrency and Synchronisation Issues :

  Multiple clients should be able to **place orders simultaneously**, thus multiple threads need to handle the interface and requests of each user. If this is not done, no other user is able to place an order while another user is using the service. The shopkeeper also needs to be modeled to do accurate delivery time calculations, thus we would need another thread to mimic the shopkeeper behaviour. Hence the need for concurrency. Since the inventory needs to be updated for each order placed, the inventory needs to be synchronised to give accurate representations of availability.

- How the scenario is modeled :

  ServerThread :

  These are threads corresponding to each socket connection at the server. These threads are dedicated to serve requests from each client. They perform inventory queries and quote the prices to the client along with availability data, and they accept orders, validate them and return invoices with expected time of delivery.

  Server :

  The server accepts socket connections and creates ServerThread threads. This class also contain the sales data and restock data, and can be used to view it.

  TeaServer :

  This thread mimics the shopkeeper behaviour of making tea and delivering it, so it is responsible for maintaining the accuracy of delivery time, by decrementing te delivery delay every minute.

  Item :

  This class just holds all the relevant data for an item in the inventory. The inventory is then just a collection of Item objects.

  Client :

  Every user will run an instance of this class to open the GUI for delivery service. This class holds GUI information and initiates connection to the server. It is also responsible for sending requests to the server for a particular user.

- Problems without Synchronisation :

  Example 1 : An example can be this sequence of execution by two threads (**T1**, **T2**) where **T1** tries to place an order for 4 chips packets and **T2** tries to place an order for 7 chips packets.

  Problematic Sequence (Inventory Updation):

  T1: Get the number of chips packets available (10)
  T2: Get the number of chips packets available (10)
  T1: Availability satisfied, reduce the number of chips packets by 4
  T2: Availability satisfied, reduce the number of chips packets by 7
  T1: Order Placed!
  T2: Order Placed!

  In the above sequence **T2** doesn't get the updated number of chips packets and hence decides that 7 chips packets are available, thus placing the order successfully and wrongly setting the value for number of chip packets to 3. This needs to be synchronised. For a synchronised update, the chips packets entry will hold 6 and **T2** will correctly throw a failed order alert to the user.

  Example 2 : Another example can be this sequence of execution by two threads (**T1**, **T2**) where **T1** tries to place an order for 4 cups of tea and **T2** tries to place an order for 7 cups of tea.

  Problematic Sequence (Delivery Time Update and Quote):

  T1: Successfully place the order, fetch current delivery delay (0)
  T2: Successfully place the order, fetch current delivery delay (0)
  T1: Updating the delivery delay to (0 + 4 + 2) , quoting the expected time of delivery 6 mins.
  T2: Updating the delivery delay to (0 + 7 + 2) , quoting the expected time of delivery 9 mins.

  But since **T1**'s order went through first, the order for **T2** will only be prepared after **T1**, thus the delivery time for **T2** would be **15 minutes** instead of **9 minutes**. Also, the value of delivery delay would be now set to **9**, while it would take the shopkeeper **15 minutes** to service these two orders. Thus, the delivery times would be wrongly quoted from here on. This too needs to be synchronised.

- Resolving the Issues :

  To synchronise the inventory we used a **ConcurrentHashMap** to hold the inventory data. This Hash Map implementation is thread-safe and allows for very large degree of concurrency since locks are placed on the buckets not on the entire collection. This manages the inventory updates properly. For the delivery delay issue, we have used an **AtomicInteger** to hold the value of delivery delay. AtomicInteger defines an operation addAndGet(), this operation atomically adds a value to the atomic integer returning the sum. Thus, the issue of two threads reading and updating the delivery time simultaneously is resolved by making the entire operation atomic.