OS-LAB ASSIGNMENT 4

GROUP 13

Members
Aditya Choudhary (20CS10005)
Vikas Bastewad (20CS10073)
Yashraj Singh (20CS10079)
Rishi Raj (20CS30040)

Important Decisions :-

1. **Sizes of Shared queue and the feed queue**
2. **Critical Sections in the code where locks are imposed**
3. **Data structures used for various tasks**
4. **Explanation of how concurrency is ensured**

ANSWERS :-

1. **Sizes of Shared queue and the feed queue**
   The sizes of the shared queue and the feed queue have been taken such that the queue does not become full by the actions produced by the producers in those respective queues, as that would be wastage of resources and the maximum size we obtained was reasonable to allocate.

   For the shared queue, the producer is the userSimulator thread, which chooses 100 nodes and generates k*(1 + log2(degree_node)), where k is taken as 10. If we take the 100 most dense nodes and calculate this value for each of them and sum it, we obtain 10340 as the size of the shared queue, which is reasonable and can be allocated.

   For the feed queue of each node, each queue can have contributions from its neighbours, ie. its degree number of nodes. Hence, assuming its neighbours have the maximum possible degree, ie. 9000, we get maximum size as (node_degree)*(1+log2(9000)), which is around 15*node_degree. This size is less likely to overflow and also ensures queue size allocation as per the degree of a node.

2. **Critical Sections in the code where locks are imposed**
   Multiple threads reading or writing on the queue has the biggest risk of interleaving. So locks have to be imposed so that only one thread can read or write to the queue, but reading by one thread and writing by one thread on a queue is possible.

   For the shared queue, since there is only one producer (userSimulator), there is no need of a lock on the producer side, but the consumer side (pushUpdate), needs to have a lock to avoid racing conditions on reading from the queue.

   For the feed queues of the nodes, we have implemented a fine grained locking strategy that involved assigning every node a lock of its own,separate for reading the feed queue and writing the feed queue, so when a thread is writing/reading to a feed queue of a node, it will impose the write/read lock of the node and proceed. This is a good strategy as chances of two threads doing the same operation on a same node are less, hence offering more concurrency. It should be noted that two threads can still read and write simultaneously to the feed queue as locks in a node for both reading and writing are different.

   A separate lock also needs to be imposed on output statements to the file and the console and the data structure for book-keeping the updated node for the readPost thread to pick up to avoid race conditions.

   So in all we have 1 lock for the consumer side of the shared queue, a lock for producer and consumer side of the feed queue for every node, a lock for output and a lock for data structure for book-keeping the updated node.

3. **Data structures used for various tasks**
   a. For book-keeping updated node in the feed queue
      For the data structure, we have used the unordered_map, which contains the mapping of the node id and the number of updates in the feed queue. Whenever there is an update in a particular node, we signal a thread which finds out the updated node from the map and gets the updates from the queue as per the priority, displays the output and then goes to sleep.

   b. For storing the common neighbours between two nodes
      For this we have used a map which contains the mapping of a pair of nodes a and b to the number of common neighbours they have in between. This data structure is computed at the beginning.

4. **Explanation of how concurrency is ensured despite locks**

   Concurrency is ensured in the design because of the following reasons :-

   a. Reading and writing can take place in the shared queue at the same time.
   b. If one userUpdate thread wants to write to the feed queue of a node, and one readPost thread wants to read from the same node, they can proceed as read and write to a queue with separate locks.
   c. If one thread wants to write to the feed queue of a node, and other thread wants to write to the feed queue of another node, they can proceed as every node has its own set of locks for reading and writing to the queue.