# OS LAB ASSIGNMENT-4

# Thread synchronization

## Design Document


## Group No: 17

**Pranav Nyati (20CS30037)**
**Prerit Paliwal (20CS10046)**
**Vibhu (20CS10072)**
**Ashwani Kumar Kamal (20CS10011)**


## Basic Data Structures and Constructs:

### 1.) Struct Action:

- **user_id:** node_id of the node which generates the action
- **action_id:** counter variable associated with each node and specific to each action (like 7th Like by a user, 3rd Comment, 12th Post, etc.)
- **action_type:** 0 for Post, 1 for Like, 2 for Comment
- **timestamp:** time instance (upto precision of seconds) at which the action is generated by the userSimulator thread for a particular user(node)

### 2.) Struct Node:

- **node_id (int):** user_id of the node (is i for the $i^{th}$ node)
- **Degree (int):** degree of node

- **Action_ctr[3] (int array):** An array of 3 counters to store the count of each type of action generated by the node (0 -> Post, 1-> Like, 2 -> Comment)
- **order_by(int):** to decide whether a node wants its feed to be shown in chronological (order_by = 1) or priority-based (order_by = 0)
- **Neighbors (vector <pair<int, int>>):** vector to store the neighbors of the node along with their priority (the first integer in pair is the neighbor, and 2nd int is the priority). The priority between two nodes is **proportional to the number of their common neighbors**. We have considered the prop. constant 1.
- **Wall_queue (queue <Action>):** The wall queue of a node stores the Actions generated by the node itself. The userSim thread pushes all actions generated for a node to its own wall_queue.
  No lock is needed for the wall_queue of a node as only the userSim thread accesses it, so it is not shared with other threads.
- **Feed_queue OR Priority_feed_queue:**
  The **feed_queue** is a simple FIFO queue of STL and is used to store the actions generated by neighbors of a node. This queue is populated for a node whose order_by value = 1 (chronological), in which case the priority_feed_queue won't be populated; as for it posts need to be stored in increasing order of time, which is maintained by the FIFO order of the queue itself, as an Action generated earlier will be pushed to the feed_queue of its neighbor earlier than another node generated later, hence will be popped out first.
  **priority_feed_queue** is a Max heap of pair <int, Action>, where the int value is the priority of the poster node that generated the Action wrt to the node. Actions are ordered as a Max heap based on the first element of the pair (int). Thus, in decreasing order of priority. We have used a custom **myComp** comparator function for defining this comparison criterion. This queue is populated for a node whose order_by value = 0 (priority), in which case the feed_queue(FIFO) won't be populated.

  **Thus, for any node, only 1 of these two feed queues will be populated depending on the feed order preference of the node.**

**Since these are STL queues, the size is auto-incremented by the STL methods, and we need not worry about an exponential increase in the size (or a MAX size) of these queues as they will be popped out (till fully empty) by the pushUpdate/ readPost threads in each iteration.**

- **Feed_mutex (pthread_mutex_t):** This is a mutex lock to allow the mutual exclusion for access to the feed_queue/priority_feed_queue of a node. **Thus, we have used one mutex lock for each graph node's feed queue.**

## 3.) Graph (vector<Node>) :

- It contains all the Nodes in the graph (ith node contains a node with node_id => i; **total 37700 nodes**). Declared globally and is shared by all threads to get attributes of a node).

## 4.) New_action_q (queue<Action>) :

- It is a global queue (STL queue) shared by all threads and is used by the userSim thread to push new Actions so that these actions can be popped by pushUpdate threads and further processed.
- This queue can be modified by the userSim (pushes new Actions to it) and pushUpdate threads (pop Actions from it); hence we need **1** mutex lock for it (**action_q_mutex**).
- **Mechanism:** pushUpdate threads wait (blocking wait) for a broadcast from the userSim thread (userSim generates a broadcast for pushUpdate threads using **pthread_cond_t action_q_cond** once on pushing each new action to the new_action_q, indicating a change in the queue). On receiving the broadcast, the pushUpdate threads pop an Action from it and then push it to the neighbors of the node that generated it.

## 5.) Feed_update_q (queue <int> ):

- It is a global queue (STL queue) between pushUpdate and readPost threads. A pushUpdate thread pushes the node_id of all the nodes

whose feed has changed (due to the pushUpdate thread pushing action to neighbors of a node) and broadcasts (using **pthread_cond_t feed_update_q_cond**) to the readPost threads indicating that the queue has changed. The readPost threads (which wait as blocking) on receiving the broadcast pop a node_id from the feed_update_q to read the feed of that node.

- This queue can be modified by the pushUpdate (pushes nodes whose feed has changed to it) and readPost threads (pop node_id from it); hence we need **1** mutex lock for it (**feed_update_q_mutex**).

**Note: (1) We need not worry about an exponential increase in the size (or a MAX size) of these queues in our implementation as they will be popped out (till fully empty) by the pushUpdate/ readPost threads in each iteration.**
**Note: (2) On running the program several times, the general MAX size that the new_action_q reached in each iteration came out around 3000-4000 (no of total actions of all 100 nodes generated in each iteration on an average), while the MAX size of the feed_update_q came out in the range of 1,00,000 - 2,00,000 (on an average), which can be reasonable handled by STL queues.**

6.) **Logfile (sns.log):** All the threads print their activities in this file. Hence there is **one** lock for this file (logfile_mutex).

# Use of Locks and Synchronization:

## 1.) Locks used:

- **1 for new_action_q** (shared b/w userSim and pushUpdate threads)
- **1 for feed_update_q** (shared b/w pushUpdtae and readPost threads)
- **1 for logfile (sns.log)** (shared b/w all threads)
- **1 for Feed of each node** (37, 700 locks)

**Note (1): The use of 37, 700 locks (one for a feed of each node) is not a very large overhead as at any time during the program's execution, a maximum of 35 different nodes' feed queues can be accessed parallelly (when each of 25 pushUpdate threads and each of 10 readPost threads is accessing a node's Feed). Thus, although there are 37700 locks for the Feeds, at any instance, max of 35 of these Feed locks are being used -> Hence, the large number of locks does not affect the concurrency and parallelism of the program.**

**Note(2): We used one lock for each node's Feed instead of mapping a fixed number of nodes to each pushUpdate (say 37,700/25) and each readPost thread (say 37,700/10), and then using a lock for each thread when it accesses the Feed of any of its mapped nodes, because this way, although the no of locks may be less, parallel running of two or more threads (in different parts of their code) may not take place.**

**For example:** Let's say we mapped Nodes 1-3770 to readPost thread 1, Nodes 3771-7540 to readPost thread 2, and so on. Now suppose the userSim picks 100 random nodes to generate actions such that feeds of 200 nodes in node-set 1-3770 got updated while only of 5 nodes in node-set 3771-7540 got updated. Then, readPost thread 2 will just service these 5 nodes and keep waiting (blocking wait till a further change in feeds of any of the mapped nodes occurs), while the readPost thread 1 will have to serve 200 nodes before it waits again. **Thus, it may lead to a drastic uneven division of work among the readPost threads and prevent them from taking full advantage of the parallel processing of threads.**

**In our approach of using one lock for the Feed of each node, on the other hand, each readPost thread services one node at a time till all nodes whose feed changed have been serviced. This leads to an equal division of work among the 10 readPost threads and allows them to take advantage of parallelism, as any thread can process any node, and there is no restrictive mapping of nodes**.

---