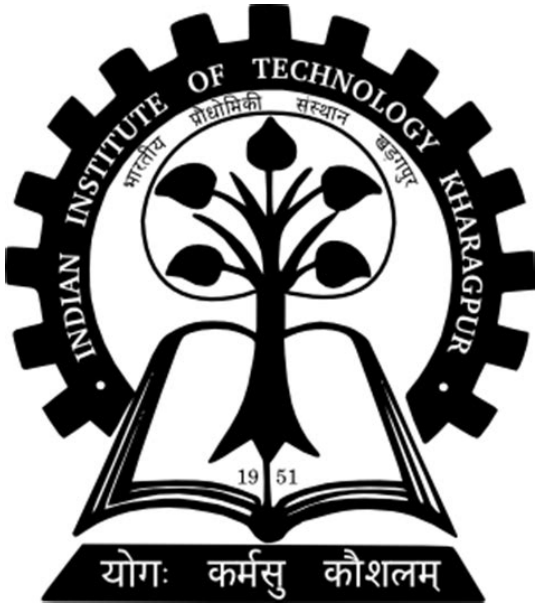
INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING



LABORATORY ASSIGNMENT 4

(Operating Systems Laboratory)

GROUP 31

Likhith Reddy Moreddigari 20CS10037

Shivansh Shukla 20CS10057

Venkata Sai Suvvari 20CS10067

Boorgu Shashank Goud 20CS30013

March 10, 2023

Data Structures

Main Thread

Storing Graph

We have defined a **class Node**, which has the following data members and member function.

```
class Node
{
    public :
    vector<int>neigh;                    // stores neighbour of current node
    int degree;                          // stores degree of current node
    priority_queue<pair<int, Action>> feedQueue;    /* stores a pair of (priority, Action) where
    priority can be defined either in terms of count of mutual friend or timestamp depending upon
    value of typeFeed variable */
    queue<Action> wallQueue;             // stores Action as they appear in wall queue
    bool typeFeed;                       // 0 - priority based, 1 - chronological
    map<int, int> cntMutualFriends;      /* stores count of mutual friend for each node with
    current node */
    Node();
    ~Node();
    void addNeighbour(int);              // to add a node to neigh vector
};
```

Each Node in the graph has it's own Wall queue, Feed queue(priority queue), a map denoting number of mutual friends. Number of Mutual friends are calculated **on fly**. Also, mutual friends between n_1 and n_2 will be calculated only if $n_1.cntMutualFriends.count(n_2) == 0$, which ensures they are calculated **at most once**.

Actions

We have defined a **class Action** which has action_id, user_id, global_action_id, action_type, timestamp and an overloaded $<$ operator used in comapring two FeedQueues if priority is same.

```
class Action
{
    public :
    int user_id;                          // Index of node which generated the action
    int action_id;                        // Action ID specific to this type of action
    int global_action_id;                 // Action ID equalivalent to all types of actions
    int action_type;                      // 0 -> post, 1 -> comment, 2 -> like
    time_t timestamp;                     // time when this action was generated
    Action(int, int, int, int);
    ~Action();
    Action(const Action &);                    // Copy constructor
    Action& operator= (const Action&);    // Copy assignment operator
    bool operator< (const Action&) const;      /* overloading < as it will be required by
    priority_queue */
};
```

We have also overloaded $<$ operator, which will be used in comparing two *pairs* within FeedQueue. For simplicity, we have assumed that if priority of two nodes is same, then we'll prefer the action which occured earlier(chronological order).

Furthermore, we have overloaded << operator to directly print the attributes of an Action object, rather than re-writing it again and again.
The maximum number of actions generated by userSimulator is 8880. This we obtained using the top 100 log2(degree(node)) values which is the worst case.

Responsibility of main thread

The *main* thread first constructs the graph by reading edges from *csv* file. After that, it initialises all the mutex locks, conditional variables appropriately. It then creates an *userSimulator* thread, 25 *pushUpdate* threads and 10 *readPosts* threads and appropriate runner functions are passed while creating threads. Now the *main* thread will wait using *pthread_join*() call for each of them, and after they return *main* thread will destroy all the mutex locks and conditional variables which were created.

pushUpdate Threads

Responsibility of pushUpdate Threads:

The pushUpdate threads first locks the log file and opens it to get access to write to it and unlocks it. Then it locks update queue and checks if it is empty or not. If it is empty then it waits on the condition variable and unlocks the update queue. If it is not empty then we take the action from the update queue and pop it from the queue. Then we unlock the update queue. Now we lock the lock the log file and write some data to it and then unlock it. Then we lock the stdout and write some data to it and then unlock it. Now we are checking all the neighbours of the node that generated the action. For each neighbour we are checking if the its feed queue is of chronological type or priority type. If it is of former type then we are setting the __priority value as negative of current time stamp. If it is of later type then we are setting the __priority value as number of common friends between the node and neighbour. Then we lock the feed queue of the neighbour and insert the pair priority value and action. The priority queue checks the priority value and sorts the queue accordingly in O(logn) time. Now we unlock the feed queue of the neighbour. Now we lock the lock the log file and write some data to it and then unlock it. Then we lock the stdout and write some data to it and then unlock it. Now we randomly select a updNodeFeed out of 10 and lock it and push the index of the index of the neighbour and signal on condUpdNodeFeed of the updNodeFeed queue we selected and then unlock it.

readPost Threads

Responsibility of readPost Threads:

The readPost thread first locks the log file and opens it to get access to write to it and unlocks it. Then it acquires the lock on the update feed queue and check if feed of any of the nodes have changed. If changed, then it gets the index of that node, pops it from the queue and unlocks the update feed queue. By locking the feed queue of the node, the thread prints the feed into log file and terminal by locking and unlocking them in each iteration so that other threads can use them and no waiting for them. After printing the entire feed to the log file and terminal, the thread unlocks the feed queue and so that the push update threads can push the feed into it.

Queues used

```
priority_queue<pair<int, Action>> feedQueue; // used to store feed of a node
queue<Action> wallQueue;   // used to actions generated by a node
queue<Action> updates;     // used as global shared queue between userSimulator and pushUpdates
queue<int> updNodeFeed[];  // used as shared queues between pushUpdates and readPosts
// updNodeFeed[i] is for ith thread of 10 threads present in readPosts
```

feedQueue

It is used to store feed of a node. It is implemented as priority queue of pair (int, Action), this is to ensure that when we push something to the feedQueue it is already sorted with respect to first variable of pair. The first variable of pair here is either global action ID or -numberOfMutualFriends, which can be known from Node.typeFeed(1 if chronological, 0 if priority based). The maximum size of a feedQueue can be at most 8610. The calculations are done as follows:

- First Calculated degree of all the nodes, and stored neighbours of each node in an adjacency list.
- Then for each node we ran a loop and found the top 100 values of $10 * (1 + log_2(degree))$ where neigh is the neighbour of current node.
- We then added those 100 values of each node and then took the maximum of all the values. So we can have at most 8610 actions in a feedQueue.

wallQueue

It is used to store a nodes actions in it's structure. The maximum size of this keeps on increasing as time progresses. Let we ran the code for k iterations. Now, the top 100 log2(degree) values are:

```
13 12 11 11 11 11 10 10 10 10 10 10 10 10 10 10 10 10 10 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9
    9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
    8 8 8 8 8 8 8 8 8 8 8 8 8 8
```

Now sum of $10 * (1 + log_2(degree))$ is 8880.
Therefore, for i iterations maximum size of all wall queues combined will be $i * 8880$

updates

This is used as global shared queue between userSimulator and pushUpdate queues. This can have a maximum size of 8880 same as total number of actions generated. We push a set of actions generated by a node at once and then pop in the pushUpdate queue.

updNodeFeed[]

These are used as shared queue between pushUpdates and readPost threads. It stores the number of node which got an action being pushed in feedQueue in pushUpdate. This will facilitate to know which node has it's feedQueue with some actions, instead of checking all 38000 nodes everytime. A node in pushUpdate whose feedQueue got

updated is mapped to one of the 10 updNodeFeed queues and is pushed into it. Now in readPost, each updNodeFeed is mapped to one of the threads and the thread will pop the node and lock mutexFeedQueue[node] and pops out all actions from that feedQueue and prints it into file, then releases $mutexFeedQueue[node]$.
This will ensure **concurrency** of threads as they operate on different set of nodes at the same time, thanks to the mutexFeedQueue[] array. From here we can see that only **at most 35** of mutexFeedQueue are used at once, in pushUpdate where we lock to push to the feedQueue and unlock it. The number 35 because only 25 threads are there so at most only 25 of the nodes can have their pushUpdate queues getting pushed, and at most 10 of the threads in readPosts can have nodes whose feedQueues getting popped.
The maximum size of updNodeFeed can be calculated as follows:

- updNodeFeed[i] has all the actions pushed in node n's feedQueue for all nodes n such that $i = n/(MAXNODES/10)$
- So size required is sum of top 100 values of $10 * log_2(degree) * degree = 994717$.
- However, on an average, as nodes will be divided among 10 queues, size will be far less than this.

Locks and Conditional Variables Used

```
pthread_mutex_t mutexUpdateQueue, mutexFeedQueue[MAXNODES], mutexUpdNodeFeed[10];
pthread_cond_t condUpdateQueue = PTHREAD_COND_INITIALIZER, condUpdNodeFeed[10];

pthread_mutex_t filelock;
```

Details are as follows:

- **mutexUpdateQueue**: This is lock used for pushing or popping from updates(shared queue between userSimulator and pushUpdates which stores actions). Lock the updates queue in userSimulator then push then unlock. Lock updates queue in pushUpdates, pop the action, then unlock.
- **mutexFeedQueue[MAXNODES]**: Each of the lock is a lock for feedQueue of a particular node. When some n1 locks mutexFeedQueue[n2] then no other node will be able to push into or pop from node n2. Though there are many locks only atmost 35 of them are used at once.
- **mutexUpdNodeFeed[10]**: Each of the lock ensures all the actions from pushUpdates are taken by the 10 threads without any race conditions.
- **condUpdateQueue**: This is the conditional variable for updates(shared queue between userSimulator and pushUpdates) which tells whether updates has some actions or not.
- **condUpdNodeFeed[10]**: These are the condition variables corresponding to each updNodeFeed queues. i_{th} readPosts thread wait on i_{th} condUpdNodeFeed variable for any update in i_{th} updNodeFeed queue.
- **filelock**: This is used to lockfile, so that no other process is writing to file at the same time.

Concurrency

Main Process creates 1 UserSimulator thread, 25 pushUpdate threads, 10 readPost threads.

i. In **userSimulator** we are generating all the actions at once (not in batches, because it will lessen the overhead of locking and unlocking since we are only pushing, generation has happened previously and got stored in a local array). With current number of actions being generated on average, we can generate all actions at once. In case, if the number of actions generated is too large say 100,000 then we can use a buffer of a particular size(typically order of 1000's) and then when it becomes full we can just signal pushUpdate (Similar to bounded buffer).

ii. In **pushUpdate** we lock and unlock the update queue after reading each action. So other threads do not wait for long to access the update queue. Similarly we lock the feed queue of a particular node just before writing into it instead of locking all the neighbours of a node that generated the action. Also any thread waiting on a lock will not wait for more time and will get a signal as soon as the lock is lifted. Here if a thread1 has action generated by node1, thread2 action generated by node2, if both nodes doesn't have any mutual nodes in common then both the threads will run concurrently, but then to print the output to the file and to the standard output, we lock and unlock to print the action so that the action message is not interfered with another. Since we unlock as soon as we print, other threads can print their respective actions. But if the two threads have a mutual friend then one of the thread will push and other threads which want to push in the same nodes feed queue will get blocked. So, the concurrency still persists in the pushUpdate threads.

iii. In **readPost** each thread waits on a different updNodeFeed queue. On average each updNodeFeed queue gets similar number of entries. So each thread will have similar amount of work to do and each thread is independent of other so that they can run concurrently. Each thread then locks a feed queue of the node in the corresponding upNodeFeed queue and then it reads all the posts in that. It prints the feed into both log file and the standard output by locking and unlocking when it print each action in feed so that other threads which want to print the action can do. After it prints all actions in the feed, it unlocks the feed of that node. It is locked during this period so that pushUpdate thread and readPost thread won't go into same critical section.

iv. When the pushUpdate threads are pushing to a node, readUpdate threads can run on different nodes feedQueue and pop some action from it, so even both the type of threads are concurrent and when the userSimulator pushes, readPosts can still run concurrently. In a case where the the userSimulator chose nodes in which node on common with the nodes that the pushUpdate threads are working currently, then all the threads viz., 1 userSimulator thread,

Thread Work Distribution

```
iter: 0
PushUpdate Threads: 3695 2407 2349 2664 2359 2503 2432 2530 2699 2513 2530 2580 2491 2333 2525 2582 2406 2486 4183 2433 2412 2627 2686 2644 2191
ReadPost Threads: 10140 12100 13700 11120 10600 13260 11800 14080 13680 13440
iter: 1
PushUpdate Threads: 4722 4347 4642 4614 5099 4531 4351 5181 4882 4702 4742 4542 4843 5329 4881 4850 4962 4997 4769 4724 5716 4840 5332 4578 4824
ReadPost Threads: 21720 25060 23960 24000 21240 23940 24500 23500 24200 22560
iter: 2
PushUpdate Threads: 2570 2699 2665 2624 2569 3169 2396 2432 2707 3463 3627 2441 2466 2807 2630 2675 2553 2742 2538 2714 2642 2538 2623 2338 2662
ReadPost Threads: 11840 10760 14040 12400 12420 11900 14020 11840 14820 14240
iter: 3
PushUpdate Threads: 1098 1271 1296 1240 1260 1454 1117 1224 1163 1298 1247 1253 1309 3236 1285 1426 1107 1204 1255 2200 1167 1420 1294 1312 1234
ReadPost Threads: 5520 5320 6080 5900 4440 7900 6160 6940 7400 7260
iter: 4
PushUpdate Threads: 6952 5645 6046 6440 6417 6417 6118 6507 6779 5692 5997 5902 6278 6001 6726 6781 6243 5657 5324 5877 5925 6359 6140 5911 6546
ReadPost Threads: 30600 29420 28220 30360 30920 30460 28940 30980 29520 32920
iter: 5
PushUpdate Threads: 2357 2909 2612 2521 2572 2356 2439 2642 2659 2423 2516 2437 2414 2508 2637 2404 2325 2646 2427 2543 2417 2361 4985 2396 2454
ReadPost Threads: 10660 12800 13420 9800 9220 14800 12460 15780 11220 13100
iter: 6
PushUpdate Threads: 4036 4485 4329 4895 4606 4251 3788 4710 5775 3990 5987 3923 4443 4967 5922 4328 5338 4383 4632 3595 3908 4513 4050 5056 3590
ReadPost Threads: 23460 22000 22020 21740 22700 22100 24740 22780 20420 18600
iter: 7
PushUpdate Threads: 2860 2837 2901 2882 2779 3036 2352 2703 2795 2843 2750 2413 2606 2942 2930 2658 2889 2757 2372 2628 2809 2671 2607 2590 2540
ReadPost Threads: 12740 13020 12900 10760 10980 13680 12600 13080 16160 13640
iter: 8
PushUpdate Threads: 3759 3992 4012 3980 4248 4109 4074 4060 4063 3930 3785 4856 3837 4005 4103 4118 4426 4409 4144 4312 4395 3813 4308 4063 3829
ReadPost Threads: 19200 18880 21560 16940 17760 18460 19640 19640 26560 19340
iter: 9
PushUpdate Threads: 3472 3295 3485 3702 3788 3489 3721 3528 3242 3370 3236 3661 3767 3276 3200 3828 3385 3335 3419 3847 3538 3631 3430 3315 3620
ReadPost Threads: 15800 16480 16260 14120 18680 17020 16320 16780 19880 17220
```