OPERATING SYSTEM LAB

ASSIGNMENT 4

GROUP: 5

Design and Data Structure Report

MEMBERS:

Abhibhu Prakash 20CS10002

Anirban Haldar 20CS10008

Shah Dhruv Rajendrabhai 20CS10088

Suhas A M 20CS10066

Vivek Jaiswal 20CS10077

Details of Data Structures:

Action Storing Data Structure:

```
typedef struct at
{
   int uid;
   int aid;
   string atype;// 0 : post, 1: comment, 2: like
   time_t t_stamp;
   int pv;// priority value or the number of mutual friends
}action;
```

uid:stores the user who created the action

aid: stores the id of the action

atype: type of action

t_stamp: the time of creation of action

pv: the priority value, used in the readPost part

Node Storing Data Structure:

```
typedef struct node
{
  int id;
   int like=0;
   int post=0;
   int comment=0;
   queue<action>wq;
   queue<action>fq;
   int ptype;  // 0: priority, 1: chronological
}node;
```

id: id of the user node

wq: the wall queue

fq: the feed queue

ptype: the type of priority for the feed queue actions

Graph:

vector<vector<int>>g(37701);

This g stores the whole graph. For each edge, it takes the first node and adds the second node to the corresponding vector of that node. Does the same after changing the order of nodes since the edges are undirected

Storing degree of each node:

vector<int>degree(37701);

This vector stores the degree of each node.

The userSimulator and the postUpdate functions shared data structure:

unordered_set<int> usq;

This set stores the nodes whose neighbors had done the actions. Whenever a action is generated all the neighbors of the action generator node will be added to this set. This is because the feed queue of these nodes will be changed by the pushUpdate function.

Store all the nodes and their queues

vector<node>nodes(tot_nodes);

This vector stores all the nodes with the user defined datatype node.

The data structure stores the actions to be inserted in the node's feed queue.

vector<vector<action>> nodes_action(37701);

This nodes_action stores the action generated by the userSimulator in each of its neighbor's vector. When we access the usq set in the pushUpdate function, we will get the first element of the set and then add the actions stored in nodes_action[element] to its feed queue.

To store the number of common neighbors for all the edges

map<pair<int,int>,int>mp;

This map stores the number of common neighbors between the two nodes present in the pair.

The pushUpdate and readPost functions shared data structure:

unordered_set<int> uset;

This set contains the set of nodes whose feed queue is changed by the pushUpdate functions.

Rationalize your choice for all queue sizes

At the time of initialization, there is no requirement of the size for the queues but analytically we can determine the maximum size of the queue from the sizes of the queue in different runs of the program. And the usq set maximum size will be the maximum number of nodes which whose feed queue is going to change by the generated actions. And according to our analysis, we got that its size should be around 1000-5000 after an iteration of the userSimulator. Now comes the size of the uset which will be the same as that of the usq as same nodes will be added to the uset since their feed queue gets updated. Now comes to the size of queues. Since we are creating actions for 100 users, with maximum degree of any user possible being n=9458, the size of action queue will be less than that considering an action queue with the maximum degree node.

Maxsize= 100*[10(1+logn)]

=> Maxsize = (1+logn)1000.

For the wall queue we cxannot set a size as it can go to infinity as the process is never ending.

Use of Locks:

1) pthread_mutex_t nodes_action_lock[37700];

These locks are used so that no two threads or functions do not access the vector nodes_action[i] at the same time.

2) pthread_mutex_t usq_lock;

This lock is used so that the pushUpdate threads and userSimulator threads do not access the set and elements at the same time.

3) pthread_mutex_t nodes_lock[tot_nodes][2];

These locks are used so that no two threads or functions change the wall queue and the feed queue of the nodes at the same time.

4) pthread_mutex_t print_lock;

This lock is used to print all the lines correctly without mixing the lines.

Justification of the Design used by us:

***userSimulator* Thread:**

It generates random actions as given and then, for every generated action *myaction*, it pushes the action in the user's wall queue (*nodes[<id>].wq*) and every neighbour of the user is inserted in a shared set *usq*.

And, the 2d vector of *action*s (struct) *nodes_action* is updated where *myaction* is pushed in every neighbour's vector in *nodes_action*.

The idea behind this is that the *nodes_action[<id>]* directly gives the vector of actions which needs to be pushed in the <id>'s feed queue. And the set ensures, if another action destined to same id comes, it just keeps the single instance and the *pushUpdate* can have all the actions to be pushed in a feed queue of a node together.

The advantage of having all the actions to be put in a feed queue of a user, together, is explained later when addressing locks.

***pushUpdate* Thread:**

All 25 threads are on wait until the set *usq* has at least an element. And then each thread takes the next user id available in the set and removes it from the set. Then from the *nodes_action*, it has all the actions to be pushed in the feed queue of the user and so does the same.

And to convey the *readPost* about which user's posts are to be read, we use the same strategy as used by *userSimulator* to convey to *pushUpdate* thread. That is, we use a set *uset*. The nodes whose feed queues got updated; those nodes are immediately added in the set *uset*.

***readPost* Thread:**

All threads wait until *uset* is non empty. And then it's just a simple task where each thread takes the next available user id from the set and store the posts (action) of the user's feed queue in a vector and sort them in the priority generated and then read them in that order. Use of set here enables us to implement efficient multithreading by being able to assign the threads the next available user (different threads to different user ids) whose feed queue is to be read.

Mutex locks and conditional variables used:

We have used a mutex lock for each node's feed queue (not for wall queue) named *nodes_lock* and not a single lock for all nodes. And the reason for this is, when the *pushUpdate* threads are pushing into feed queues of a particular node, the pushing to and reading from other nodes should not be blocked.

And for the similar reason, lock is used for every user's vector in *nodes_action* so that *userSimulator* updating a user's vector in *nodes_action* does not block *pushUpdate* updating another user's vector in *nodes_action.*

2 conditional variables are used, *usq_cond* for signaling and waiting of the *usq* and *uset_cond* for signalling and waiting of *uset*.

And the reason for using the method of having all actions destined to a user, together, over using a shared queue (used between *userSimulator* and *pushUpdate*):

If a shared queue is used, then assume the queue has 2 actions consecutively which are destined for the same user. The first thread will acquire the feed queue of the user. And so, the second thread must wait for it to get freed to acquire it, but there are lot many other actions in shared queue that are destined to other users which could have been dealt first. And this can be more than just 2 actions and 2 threads.

However, if we have all the actions destined to a user together for every user, the threads won't collide as each thread could take up a user and acquire its feed queue to push all the required actions in it.