OPERATING SYSTEMS LABORATORY

CS39002

Assignment 4:

Creating a push-updates mechanism for a social-media site using threads

Designing Documentation

Group No. 21

Nikhil Sarashwat (20CS10039)

Abhijeet Singh (20CS30001)

Amit Kumar (20CS30003)

Gopal (20CS30021)

➢ Data Structures

```
typedef struct Graph
{
    int node_count, edge_count;
    Node nodes[MAX_NODES];
    EdgeData edges[MAX_EDGES];
    Graph(int n)
    {
        node_count = n;
        for (int i = 0; i < n; i++)
        {
            nodes[i].id = i;
        }
        edge_count = 0;
    }
    void addEdge(int from, int to)
    {
        edges[edge_count].to = to;
        edges[edge_count].nxt = nodes[from].head;
        nodes[from].head = edge_count;
        nodes[from].degree++;
        edge_count++;
    }
    ~Graph()
    {
        for (int i = 0; i < node_count; i++)
        {
            pthread_mutex_destroy(&nodes[i].feedQueueMutex);
            pthread_cond_destroy(&nodes[i].condVar);
        }
    }
} Graph;
```

For implementing the graph we defined the above structure. It has the following fields:

- **node_count**: an integer to store the number of nodes in the graph.
- **edge_count**: an integer to store the number of edges in the graph.
- **nodes**: an array of Node structure that stores information about a node in the graph.
- **edges**: an array of EdgeData structure that stores information about an edge in the graph.

- **Graph(int n)**: a constructor that takes the number of nodes in the graph as an argument and initializes the id field of each node to its index.
- **addEdge(int from, int to)**: a method that adds an edge to the graph, taking the indices of the source and destination nodes as arguments. The method adds the new edge to the list of edges that originate from the source node, increments the source node's degree field, and increments the edge_count field of the graph.
- **~Graph()**: destructor to destroy the mutex and condition variable objects associated with each node in the graph.

```
// Node struct to store the information of a graph node
typedef struct Node
{
    int degree;
    int head;
    int id;

    pthread_mutex_t feedQueueMutex;
    pthread_cond_t condVar;

    vector<Action> wallQueue;
    vector<Action> feedQueue;

    int count[3]; // 0: post, 1: like, 2: comment
    int priority; // 1: chronological, 0: popularity

    Node(int id = 0)
    {
        this->degree = 0;
        this->head = -1;

        this->id = id;
        this->count[0] = 0;
        this->count[1] = 0;
        this->count[2] = 0;

        this->priority = random_int(0, 1);

        pthread_mutex_init(&feedQueueMutex, NULL);
        pthread_cond_init(&condVar, NULL);
    }
} Node;
```

The Node structure could be used to represent a user in a social media platform, and it contains several fields that could be used to manage and display the user's feed.

- The **feedQueueMutex** and **condVar** fields would be used to manage the synchronization and blocking of threads that are accessing or modifying the feedQueue vector. The **feedQueue** vector stores the post made by the neighbor of the user.
- **feedQueueMutex** ensures that at a time only one other user can make changes to this user's feedQueue vector.
- **condVar** is a condition variable which is used as a synchronization primitive that allows threads to wait when feedQueue is empty and proceeds when this feedQueue isn't empty.
- **WallQueue** is a vector that holds all the posts made by a particular user. In a social media feed, posts from a user's wall would be displayed in the user's own feed.
- The **count** array could be used to keep track of the number of posts, likes, and comments that the user has made, and then it would be used to calculate their overall activity on the platform and potentially influence the ordering of posts in their feed.
- The purpose of the **priority** field is to determine the ordering of posts in the user's feed. If priority is set to 1, then the posts would be ordered chronologically based on the timestamp of the post. If priority is set to 0, then the posts would be ordered based on their popularity, which could be determined based on factors such as the number of likes or comments they have received.

```
typedef struct EdgeData
{
    int to;
    int nxt; /* nxt: is the index of the next edge */
} EdgeData;
```

The EdgeData structure is used to represent an edge in a graph. It contains two fields:

- **to**: This field represents the destination vertex of the edge. It is an integer value that specifies the index or identifier of the vertex that the edge connects to.
- **nxt**: This field represents the index of the next edge in the adjacency list of the source vertex. It is an integer value that indicates the position of the next edge

in the array. The nxt field is used to represent a linked list of edges that originate from the same source vertex.

```
// Struct to represent an action
typedef struct
{
    int user_id;     // node id
    int action_id;   // 4th like, 5th comment ,etc
    int action_type; // post ,like , comment
    time_t timestamp;
} Action;
```

This struct can be helpful for several reasons:

- In this struct we are incorporating a **timestamp** which would enable displaying of actions in increasing order of time by sorting the actions on the basis of a timestamp in a user's feed.
- Also we are storing **user_id** of the user enabling sorting based on the number of common neighbors between the poster node and reader node as we are able to identify each action by the user that caused the action.
- Additionally, we have **action_id** for uniquely identifying each action and **action_type** to help us distinguish between the three different types of actions.

➢ Rationalize choice

feedQueue

- We are finding the top 100 values of degrees among all the nodes and summing up the value of 10(1+log2(degree)) for the top 100 nodes.
- Taking approximately size of 8700 will suffice as there will be less than 8700 actions using which we found using the formula.

wallQueue

- 8880 will be the sum of 10*(1+log2(degree)) using top 100 log2(degree) nodes.
- And for i queues the size of wall queues combined will be i*8880.

➢ Locks

- A **feedQueueMutex** has been associated with each node and thus since there are 37,700 users(nodes) there will be 37,770 locks corresponding to each user.

- These locks are used to allow only one neighbor to insert into the current user's feedQueue at one time.
- A lock named log_mutex has been kept to ensure only one thread at any given time will be able to write to the file "sns.log".
- A lock named log_mutex_terminal has been kept to ensure only one thread at any given time will be able to write to stdout i.e. terminal.
- There are 25 threads for push_update and each thread has its separate pushQueue and corresponding to each of these queues we have a lock named push_queue_mutex so that UserSimulator and push_update will not change pushQueue simultaneously.
- There are 10 threads for read_post and each thread has its separate set named feedQueueSet and corresponding to each of these queues we have a lock named feed_queue_set_mutex so that threads of read_post and push_update don't change the same feedQueueSet simultaneously.
- Thus there are a total of 37,737 locks.

Preserving Concurrency:

- We are using 25 threads for push_update and each thread has its own queue into which we are pushing the action and each of these 25 queues has its own lock and thus locking one thread will not block other threads from executing and hence we are able to preserve concurrency for push_update threads.
- Similarly for read_post there are there 10 threads and each thread has its own set and each set is controlled by its own mutex lock named feed_queue_set_mutex and since there are different locks for each of its thread hence the blocking of one thread will not affect the other thread and hence its concurrency is preserved.
- Each read_post thread will execute for milliseconds of time because we are writing into a file and terminal and flushing into the terminal. But accessing a lock only requires microseconds of time. Thus probability of accessing locks by two or more read_post threads is negligible.