INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

OPERATING SYSTEMS LABORATORY

Assignment 5 - Report

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

Submitted By: Group 29
Vinod Meena (20CS10074)
Kriti Bhardwaj (20CS30028)
Nirbhay Kumar (20CS10040)
Sonu Kumar Yadav (20CS10061)

DATA STRUCTURES USED

- **class Node** : The Node class represents a user, and each instance of the Node class stores information about the user, such as their ID, the number of actions they have taken, and their action queues (feed and wall).

  Breakdown of each element of the 'Node' class:

(a) **id (integer)** : the unique identifier for the user associated with this node.
(b) **action_cnt (array of integers)**: keeps track of the number of actions of each type (post, like and comment) that the user has taken.
(c) **order (integer)**: randomly initialized to either 0 (for priority) or 1(for chronological) to determine the order in which actions are added to the feed queue.
(d) **wq (queue of action objects)**: the user's wall queue (FIFO), which stores all actions taken by the user
(e) **fq1 (priority queue of action objects)**: the user's feed queue, which is sorted chronologically (by time_stamp)
(f) **fq2 (priority queue of action objects)**: another feed queue for the user, sorted by priority.

- **struct action** : The action struct represents an action taken by a user

  Elements of the action struct:

(a) **user_id**: an integer variable that represents the ID of the user who performed the action
(b) **action_type**: an integer variable that represents the type of action performed by the user. There are three possible action types, and the variable can take value as 0 (post), 1 (comment) or 2 (like)
(c) **action_id**: an integer variable that represents the ID of the action performed by the user e.g. 5th comment, 7th post
(d) **time_stamp**: a time_t variable that represents the time when the action was performed.
(e) **priority**: an integer variable that represents the priority of the action.

- **map<int, set<int>> graph** is used to represent the social network graph. In this representation, each user/node of the network is represented by a unique integer identifier, and the set associated with each user/node contains the integer identifiers of other users/nodes that this user/node is connected to in sorting order

- **map<pair<int, int>, int> neighbours** is used to store the count of mutual friends of a pair of users. This count can be further used to update the feed queue of a user on priority basis.

- **queue<action> shared_queue** : The shared_queue is a queue of actions that have been taken by users, and it is shared between the user simulator thread and the push update thread. The user simulator thread adds new actions to the queue, and the push update thread consumes actions from the queue and updates the feed queues of the affected users i.e. the neighbours.

  Shared_queue is of flexible size.

- **queue <int> cfq** : This FIFO queue stores the id of the users whose feed queue is affected.

LOCKS USED

- There are mutex lock for **shared_queue_lock** for shared queue, **cfq_lock** for cfq and **feed_queue_locks** for feed queues for each user

  Mutexes are used to provide mutual exclusion and ensure that only one thread can access a shared resource at a time. This is done to prevent race conditions and other synchronization issues.
  A conditional variable is a synchronization primitive that allows threads to wait for a certain condition to become true before proceeding.

- The **shared_queue_lock** mutex is used to protect the shared queue. Whenever a thread wants to access the shared queue, it must first acquire the shared_queue_lock mutex. This ensures that only one thread can access the shared queue at a time, avoiding data corruption.

  Conditional variable **shared_queue_cond** is used to wait for the shared queue
  This lock is acquired and released whenever a thread needs to access the shared queue. Hence the shared_queue_lock was used before performing the following tasks:
  (a) to push an action in the shared queue
  (b) to get an action from the shared queue in the pushUpdate thread

  Total shared_queue_lock used = 1

- The **feed_queue_lock** mutex is used to protect the feed queue of each user. Whenever a thread wants to add an item to the feed queue, it must first acquire the feed_queue_lock mutex.

  This lock is acquired and released whenever a thread needs to access

the feed queue of a particular user. Hence the feed_queue_lock was used before performing the following tasks:

(a) to push an action in the feed queue of a particular user
(b) to read a post from an user's feed queue

Total feed_queue_lock used = 37700

- The **cfq_lock mutex** is used to protect the cfq queue

Conditional variable **cfq_cond** is used to wait for the cfq queue

This lock is acquired and released whenever a thread needs to access the cfq. Hence the **cfq_lock** was used before performing the following tasks:

(a) to push the id of the affected user in cfq
(b) to get the user id for reading posts from the user's feed queue

Total cfq_queue_lock used = 1

RATIONALIZING QUEUE SIZE

- The **userSimulator** thread pushes actions to a shared queue, but the size of the queue cannot grow indefinitely. The pushUpdate thread continuously pops elements from the shared queue, so the size of the **shared queue** will be kept small. This ensures that the pushUpdate thread can handle the elements in the shared queue in a timely manner.

- Similarly, the **readPost** thread regularly pops elements from the feed queue of each user, ensuring that the size of the feed queue does not grow arbitrarily. This allows the readPost thread to handle the elements in the feed queue in a timely manner and ensure that the users are receiving new posts in a timely and efficient manner.