CS39002 - OPERATING SYSTEMS LABORATORY

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Laboratory Assignment 4

Authors:
Abhay Kumar Keshari (20CS10001)
Aniket Kumar (20CS10083)
Jay Kumar Thakur (20CS30024)
Tanmay Mohanty (20CS10089)

Date: March 10, 2023

Data Structures Used in the Assignment

Structure — Action

```cpp
class Action {
  static uint64_t postCount;
  static uint64_t commentCount;
  static uint64_t likeCount;
  int userId;
  int actionId;
  enum ACTION_TYPE {
    POST,
    COMMENT,
    LIKE
  } actionType;
  time_t timestamp;
  Action();
  Action(int user, ACTION_TYPE type);
  Action(const Action &action);
  ~Action();
  Action &operator=(const Action &action);
};
```

Analysis

1. Three static variables are used for maintaining the counts of Posts, Likes, and Comments

2. userId stores the ID of the user, who has created the action corresponding to actionId

3. The above data structures helps us in carrying out all the actions generated by any node, efficiently, based on custom priority

Structure — Node

```
class Node {
  struct Friend {
      Node *node;
      int priority;
      Friend();
      Friend(Node *n, int p = 0);
      Friend(const Friend &a);
      Friend &operator=(const Friend &a);
  };

  enum PREFERENCE {
      PRIORITY,
      CHRONOLOGICAL
  };

  int userId;
  const PREFERENCE preference;
  std::map<int, Friend> friendList;
  ActionQueue wall;
  ActionQueue feed;

  Node();
  Node(int user, PREFERENCE pref);
  Node(const Node &node);
  ~Node();

  inline size_t getDegree() const {
      return friendList.size();
  }

  void initPriority();
};
```

Analysis

1. We have designed a data structure named ActionQueue, which is a priority queue that stores actions depending upon the custom comparator.

2. We developed a friendList to store the neighbors of the node corresponding to userId, along with their priorities. This will assist us in constructing the priority queue which will be used by the readPost thread to display the feed.

3. Two ActionQueue are constructed to keep the actions according to the time-stamp in the wall queue and the node preference in the feed queue.

4. To improve the space complexity and efficiency, we only store pointers inside the friend data structure.

Locks

- We have utilized a total of 37,714 locks for the purpose of achieving the targets, and the allocation is as follows:
  - 37,700 locks for the feed queues present at each node
  - 10 locks comprising of 1 lock for each queue associated with a read process thread
  - 2 locks are for sns.log, and cout
  - 1 lock for Global ActionQueue

Justification of Parallelism:

- To understand the underlying parallelism between the threads, we start with analyzing the main thread, which initializes the above-mentioned locks.
- We then consider the lock for the Global ActionQueue, since it is being accessed by 1 SimulateUser and 25 PushUpdate threads. To avoid race conditions, we have allowed only one thread at a time to access the Global ActionQueue, for pushing and popping. To improve efficiency, we have introduced multipop and multipush functions, to maximize the throughput for the thread after the release of the lock.
- To improve parallelism between PushUpdate and ReadPost, we have created 10 queues, one for each thread corresponding to ReadPost. To avoid redundant waits, we acquire the lock corresponding to the feed queue of the node under consideration, when we need to push an action into the feed queue of the node.
- We have also added the nodeId corresponding to the node which received the action in its feed, in one of the threads of ReadPost. We have equally shared the load between each thread of ReadPost by using a modulo 10 function.
- A ReadPost owns its dedicated Queue, hence there is no need to wait for any other ReadPost to release its lock. It can start the process immediately after acquiring lock from any PushUpdate process. Moreover, we can also use multipop to read all the actions in a queue at once.
- The two locks for outputs corresponding to sns.log and cout are globally shared among all the threads of SimulateUser, ReadPost and PushUpdate. Thus, a thread needs to acquire the lock prior to printing on any of these files. This ensures the prevention of race conditions during the generation of the output files.

Queue Sizes

- Our implementation utilizes 11 different Global Queues and 37,700 Feed and Wall Queues. Out of the 11 Global Feed Queues, 10 are associated with all the pushUpdate threads and 1 readPost thread, and the last one is associated with the 25 pushUpdate threads and 1 simulateUser thread.
- To determine the maximum size of the Global ActionQueue which is associated with the simutaeUser and the pushUpdate threads, we can find the maximum sum of the following equation

  $$\sum_{n=0}^{100} 10(1 + \log_2(\text{INDEGREE}))$$

  which, when we evaluate for the given graph, comes out to be 9870 which can be rounded to 10000.
- To determine the size of the 10 readPost Queues we must remember that the Queues are designed in such a way as to balance out a load of any one Queue through a biased cyclic mod thus to find the maximum we can find the nodes with the maximum indegrees, find the Actions generated by them and multiply with the value of Indegree, thus after calculating we acquire a total of 90000.
- As the wall Queue has continuous pushes we can not fix a maximum size for it.
- Finally the Feed Queue can receive input from all the neighbors, thus the node having neighbours of highest Indegree will receive the largest input into the Feed Queue which upon computation comes out to be 9830, which can be rounded to 10000.

Main Thread

- We start with performing preprocessing activities in the main thread. We load the graph into an array of nodes and efficiently calculate the priorities of each neighbor corresponding to each node. Then we proceed towards creating threads for userSimulator, pushUpdate and ReadPost.
- We also created a monitor thread, which keeps track of the number of actions read, created, or pushed by any of the threads spawned previously and outputs the statistics to timer.log.

UserSimulator Thread

- This thread randomly selects 100 nodes and computes the number of actions to be generated by each of them as $10 * (1 + log_2(InDegree))$.
- Next, we select 500 actions randomly from all the actions and push them to the queue. This improves diversity and simulates a closer to real-life ActionQueue.
- After this, the lock is acquired by the thread for file I/O and outputs the required statements.

pushUpdate Thread

- First, the pushUpdate thread acquires the lock to access the Global ActionQueue. It then pops the latest element in the ActionQueue and releases the lock. This method of lock release before pushing into the feed queue allows a parallel run of pushUpdate and simulateUser thread, while this thread is occupied in pushing into the concerned feed queue.
- After this release, the pushUpdate thread pushes to the feed queue of each neighbor of the Node that generated the action. This also requires the acquisition of the lock corresponding to the neighbor's feed queue.
- Simultaneously it pushes the nodeId, whose feed queue has been transformed into the queue of the readPost thread, to balance the workload. This balance is maintained by taking the biased cyclic mod of Action ID and an offset.
- After pushing into the required Feed Queue and the readPost thread Queue, the pushUpdate thread acquires the lock for file IO to print its work.

readPost Thread

- The readPost thread first acquires the lock of its own Queue to check if anything has been pushed into it. Else it goes to sleep and releases the lock.
- If items are present in the Queue, it pops the element and reads their respective feed Queue by acquiring the lock to the necessary feed Queue, and then performing the read in order to avoid race conditions.
- After which the readPost process acquires the lock for file I/O to print its work.
- This method where we have multiple queues running parallelly with other threads allows the maximum amount of parallelization. This is because it does not require any resource apart from the lock corresponding to the feed queue of the graph node.