OS LAB Assignment - 4

Design Report

Group Members

1. Pranav Mehrotra (20CS10085)
2. Saransh Sharma (20CS30065)
3. Anand Manojkumar Parikh (20CS10007)
4. Soni Aditya Bharatbhai (20CS10060)

Report

1. Classes:

a. Node:

Node is the class of individual users of the system, with its user_id, degree, sort_by preference, action_ids, and priorities of neighbor nodes and wall and feed queue and a semaphore for the feed queue. We are using a vector as the data structure of the wall and feed queues, as in the priority-based reading, the actions have to be sorted and hence rather than copying them to a different vector and then sorting, we will just sort the feed vector '**in place**'. We could also use a priority queue instead of a vector, but the time complexity would be the same as a vector, as n insertions in a priority queue are O(n*log n) and Sorting is also O(n*log n). We have also overloaded the '<<' operator to help in printing the user details.
Size of Node: 232 bytes

b. Action:

Action is the class of individual actions, with user_id, action_id, timestamp, and action_type. We have defined the necessary constructors and destructors and also overloaded the '<<' operator to help in printing the actions.
Size of Action: 24 bytes

c. my_semaphore:

```
class my_semaphore{
   private:
      int value, wakeups;
      pthread_mutex_t mutex;
      pthread_cond_t cond;
   public:
      my_semaphore(int val);
      my_semaphore(const my_semaphore &s);
```

```
    ~my_semaphore();
    void _wait();
    void _signal();
};
```

Attributes:

1. value: the initial value of the semaphore
2. wakeups: the number of times signal() was called when some threads were waiting (i.e. while value was negative) to allow the waiting threads access to the shared resource
3. mutex: the mutex lock that needs to be acquired to ensure atomicity (with respect to the lock) in wait() and signal()
4. cond: the condition to wait on in case the value becomes negative during the wait(), i.e. shared resource(s) are occupied

Methods:

1. Constructor: value is initialized to given value (default 1, to imitate a simple lock), wakeups initialized to 0, and the pthread things: mutex and cond are initialized through appropriate syscalls with error checking and default attributes (NULL in place of &pthread_mutexattr_t and &pthread_condattr_t respectively)
2. Copy Constructor: value and wakeups copied. But mutex and cond are freshly initialized, since each semaphore waits on a different lock, obviously.
3. Destructor: mutex and cond are de-initialized through appropriate syscalls with error checking.
4. wait: Best explained through pseudocode-

```
wait(){
        mutex_lock(mutex);              // acquire lock
        value--;                        // decrement value
        if(value < 0){          // if negative, resources exhausted
        // so while no wakeups (i.e. no current user signals)
            while(wakeups == 0){
                cond_wait(cond);   // wait on cond, release lock
```

```
        }
    }
    // lock acquired here
    // if there are more wakeups left, broadcast again
    if(wakeups > 0){
        cond_broadcast(cond);
    }
    mutex_unlock(lock);       // finally release lock
}
```

5. signal: Best explained through pseudocode-

```
signal(){
    mutex_lock(lock);          // acquire lock
    value++                    // increment value
    if(value <= 0){            // if value is non positive
    // then it means some threads attempted to use
    // this semaphore, but all resources were unavailable
    // at that moment, so now it's this thread's responsibility
    // to wake those who were waiting
        wakeups++;             // increment wakeups
        cond_broadcast(cond);  // broadcast to cond
    }
    mutex_unlock(mutex);       // release lock
}
```

[**Note:** Reasons for implementing semaphores using mutex locks and condition variables and then using them rather than directly using mutex locks:

1. Semaphores are more versatile than locks in the sense that we can initialize them with any values. For eg., we can wait() in one thread and signal() from another, which is required for easy solutions to the standard readers-writers and bounded buffer problems (and the variation of these used in this problem, explained later).
2. The critical section of the code is only 5-6 instructions in length! So busy waiting is next to nil if any at all.
3. All pthread functions are called within wait() and signal() so the code is not scattered with syscalls and tedious error checking everywhere.]

2. Precompute Priorities:

We are precomputing the priorities for each user node in the Main thread before creating other threads. We are iterating over all the users and then for all their neighbors (as we only need to compute the priorities of neighbors of a node, as its feed queue can have posts from only its neighbors), we are checking for common neighbors between user and the neighbors of the user by using **'set'** data structure to check in **O(log n)** time if common neighbor exists and therefore incrementing the priority. Additionally, to reduce runtime calculation every time the program is run, if the results were not pre-calculated, then after calculating we are storing them in a file for future reference.

3. Important Data Structures:

a. **vector<vector<int>> graph**: The standard way to store a graph in adjacency list format

b. **vector<Node> users**: users[i] is the Node of the user with user_id = i

c. **vector<vector<Action> * > shared(25**):
This is the shared "queue" between the userSimulator thread and 25 pushUpdate threads. It has 25 pointers, each pointing to a vector of Actions in the heap of the process.
Size of shared queue = 25 * sizeof(pointer) = 200 bytes

The pointers are pointing to the vector of Actions that are stored in the heap, which can have a Maximum size of:
$25 * K * \log_2(\max(\text{degree})) * \text{sizeof(Action)}$ bytes = $25 * 10 * \log_2(9458) * 24$ bytes
[Exact usage and reason for this design explained later]

d. **set<int> feed_updates**:
This set is used by pushUpdate Threads to store only unique nodes to whose feed queue, actions have been pushed, so that we ensure that only one readPost thread can read all the actions from the feed queue of that user and no other thread is kept waiting on the lock of that user's feed queue, thereby minimizing the number of lock

requests and making the execution much faster. The final pushUpdate Thread after completing execution for that iteration will change this set to ‘inform’ shared vector so that readPost threads can read concurrently.

Data Analysis:

While analyzing data we found, that the total count of neighbors of a set of random nodes contains approx. 17% - 28% duplicate nodes, hence by using ‘set’ we ensure that these duplicate nodes are removed, and hence only 1 readPost thread can read all the actions for that thread.

e. **vector<int> * inform:**

This is a pointer to a vector in the heap which is shared by the 25 pushUpdate threads and the 10 readPost threads. Since there are 37700 nodes, readPost threads cannot poll or wait on each of 37700 feed queues. So, pushUpdate threads will put in the indexes of Nodes whose feed queues have been modified to inform readPost threads of these changes.

The vector it points to is dynamically allocated (new) by a pushUpdate thread and deallocated by (delete) by a readPost thread.

Size of inform "queue" = sizeof(pointer) = 8 bytes

The pointer is pointing to a vector in the heap of the process, which can be of the size of the number of neighbors (assuming all are distinct) of the 25 source action nodes. On average, 25 * avg(degree per node) * sizeof(int) = 25 * 8 * 4 bytes

[Exact usage and reason for this design explained later]

4. The Flow of execution, Synchronization, and Concurrency of various Threads

The different threads function as follows:

(how mutual exclusion, progress, and bounded waiting are enforced using locks is described later)

a. userSimulator:

The userSimulator thread runs the following procedure 4 times before going to sleep for 2 minutes:

1. Generate all actions for 25 nodes and store them in a static buffer for their own usage and allocate a dynamic copy for usage by pushUpdate threads (maintaining static copy is necessary, since the dynamic one)
2. Acquire access to the shared queue.
3. Put pointers to these 25 dynamic vectors (each vector has all actions of one node) in the shared queue. Using this technique, we only hold the shared lock while copying 25 pointers, which takes almost no time, rather than copying all actions by value, which would take time linear in the number of actions!
4. Release access to the shared queue and signal pushUpdate threads.
5. Acquire access to output files: sns.log and STDOUT.
6. Print respective data to both
7. Release access to output files.

[**Note:** Reasons exactly 25 Nodes' actions are generated at once:

1. If only 1 Node's actions are generated and then pushed, we need to lock and unlock the same mutex 100 times! This wastes a lot of time.
2. If all 100 Nodes' actions are generated and then pushed at once, parallelism is killed, since all pushUpdates and readPosts are sitting idle for this whole duration! They could have been processing old actions.
3. So, 25 is a good trade-off to guarantee both parallelism and efficiency. Also, the size of data in the heap is not too large.]

b. pushUpdate[25]
   There is some level of synchronization between the pushUpdate threads. pushUpdate threads perform the following procedure forever:
   1. Exactly ONE pushUpdate thread (the first one to start reading) waits for acquiring to the shared queue.
   2. Once this is obtained, all pushUpdate threads are allowed CONCURRENT access to the shared queue, meaning they can parallelly read the shared queue.
   3. Each thread reads a distinct index: the i^{th} thread reads the i^{th} index.

Each index is a pointer to a vector of actions, so the thread first copies this pointer in a local pointer.

4. Exactly ONE pushUpdate thread (the last one to finish reading) releases access to the shared queue and signals the userSimulator thread.
5. Each pushUpdate thread reads its data from its local pointer. These actions are then pushed to all neighbors of the node that generated these actions. This pointer (pointing to vector<Action> in heap) is now de-allocated. [Note that the lock has already been released, so the shared queue can now be accessed by the userSimulator WHILE the pushUpdate threads are reading from the heap and performing pushes to feed queues].
6. Exactly ONE pushUpdate thread (the first one to start writing) waits to acquire access to the feed_updates set.
7. Once this is obtained, all pushUpdate threads are allowed SEQUENTIAL access to this set. Each one inserts the indices of Nodes whose feed queues it updates. So, in case multiple pushUpdate threads add actions to the feed queue of the same node, it appears only once in the set.
8. Exactly ONE pushUpdate thread (the last one to finish writing) releases access to the feed_updates set. However, since multiple readPost threads cannot read simultaneously from a std::set, this thread (the last one) copies the set data into the inform "queue" which is implemented as a vector<Action> *, so a local copy of this vector is dynamically allocated. THEN access to the "inform" queue is obtained, the pointer value is copied and access is released.
9. Each pushUpdate thread acquires access to the output files.
10. Each pushUpdate writes appropriate data to sns.log and STDOUT.
11. Each pushUpdate releases access to the outfiles.

c. readPost[10]

There is some level of synchronization amongst the 10 readPost threads. readPost threads perform the following procedure forever:

1. Exactly ONE readPost thread acquires access to the inform "queue" (the first one to read)

2. Subsequent readPost threads are allowed CONCURRENT access to the inform queue. Now, each readPost thread reads individual indices in a %10 fashion. For eg., the 3rd thread reads indices 2,12,22, etc.
3. Exactly ONE readPost thread releases access to the inform "queue" (the last one to write), AFTER de-allocating the WHOLE inform queue from the heap.
4. Each readPost acquires access to the output files.
5. Appropriate data is output to sns.log and STDOUT.
6. Each readPost releases access to the output files.

5. Organization of Semaphores

The following semaphores (self-implemented using the regular pthread functions: pthread_mutex_lock(), pthread_mutex_unlock(), pthread_cond_wait(), and pthread_cond_broadcast()) are used for the mentioned purposes:
(value in brackets)

1. write_logfile(1) - for writing to sns.log and STDOUT
2. write_shared(1) - for writing to "shared" queue
3. read_shared(1) - for reading from "shared" queue
4. pU_group(1) - for synchronization amongst the 25 pushUpdate threads
5. write_inform(1) - for writing to "inform" queue
6. read_inform(1) - for reading from "inform" queue
7. rP_group(1) - for synchronization amongst the 10 readPost threads
8. feed_sem(1) - one EACH for every Node's feed queue

[Total locks = 37700]
[**Note**: These are only the number of semaphore objects INITIALIZED for usage. Not every semaphore is busy at all times. Having these many semaphores actually improves concurrency.
Fewer semaphores would mean that more resources are needed to be locked together in groups, meaning that all resources in a group are locked while one is being used, which reduces concurrency.
Having a separate semaphore means we can work on each one separately and parallelly.]

Their usage can be explained best using the following pseudocodes for each type of thread:

a. userSimulator:

```
while(1){
        write_shared.wait()         // wait for write access
        // write to shared queue
        read_shared.signal()        // signal pushUpdates for read access

        write_logfile.wait()        // wait for write access
        // write to sns.log and STDOUT
        write_logfile.signal()      // release write access
}
```

b. pushUpdate[id]:

```
start_read = 0      // no. of threads who have started reading from shared
finish_read = 0     // no. of threads who have finished reading from shared
written = 0         // no. of threads who have written to files
done[25] = {0}      // TRUE if the i^th thread has done everything, else FALSE

while(1){
        while(1){
                pU_group.wait()             // wait on pU_group
                if(done[id] == 0){          // if already done
                        pU_group.signal() // then signal pU_group
                        continue            // and go back to waiting
                }
                // else, if not done
                if(start_read == 0){        // if it's the FIRST one
                        // only then it needs to wait on read access for  "shared"
                        read_shared.wait()
                }
                start_read++
// releasing here allows other threads to proceed until this point SEQUENTIALLY
                pU_group.signal()
```

```
        }

        // So, this zone allows CONCURRENT access to all pushUpdate
        threads since it // is not within any critical section
        // also pushUpdates have read access to the shared queue at this
        point
        // so, all will read from shared parallelly

        pU_group.wait()          // wait on pU_group
        finish_read++            // this thread has now finished reading
        if(finish_read == 25){   // if it is the LAST thread
            start_read = 0       // then reset variables
            finish_read = 0
// and now signal write_shared allowing userSimulator to write into it once again
            write_shared.signal()
        }

    // update each feed queue separately by acquiring lock, writing, releasing

        pU_group.wait()          // wait on pU_group
        if(written == 0){        // if it is the FIRST thread to write to inform
            write_inform.wait()      // wait for writing to inform queue
        }

        // This zone allows SEQUENTIAL access to all pushUpdates
        // also, pushUpdates have write access to inform queue at this point
        // So, all will insert indices of Nodes whose feed queues to update in
        // std::set<int> feed_updates

        written++       // this thread has now finished writing to feed_updates
        done[id] = TRUE  // this thread has done everything for this iteration
        if(written == 25){       // if it is the LAST thread to do so
            // then dynamically allocate new vector<int> inform
            // take all from set<int> feed_updates to vector<int> * inform
            read_inform.signal()  // signal readPosts to read from inform
            // reset variables for the next iteration
            written = 0
```

```
            for(i : [0,25]) done[i] = FALSE
        }
        pU_group.signal()          // release group lock

        write_logfile.wait()       // wait for write access
        // write to sns.log and STDOUT
        write_logfile.signal()     // release write access
}
```

c. readPost[10]

readPost code is very similar to pushUpdate (only different semaphore names are used, obviously). All readPost threads are allowed to read from the "inform" queue CONCURRENTLY in the same fashion as pushUpdates are allowed to read from the "shared" queue concurrently.
Moreover, each of them is guaranteed to wait on different feed queues, since we used a set<int> to uniquely identify queues that are updated, providing more parallelism. Output to files sns.log and STDOUT is done in the same way as userSimulator and pushUpdate.

END