Operating Systems Laboratory (CS39002)
Spring Semester 2021-2022

Assignment 3: Creating a push-updates mechanism for a social media site using threads.

Assignment given on: Mar 01, 2023
Assignment deadline: Mar 10, 2023, 11:55 pm

In this assignment you will learn how to use threads to create an efficient system. We will add a "push update" mechanism for a social media website. The idea is simple: In almost every functional social media website today you see a 'push' mechanism, when your friends post something on their wall, or perform an activity like liking or commenting on a post, the update comes to your feed. In fact the feed can be sorted by time or by priorities of friends, signifying the order in which you would read the post. We will implement this mechanism using multiple threads for multiple tasks. The threads you need to implement and their functionality is below:

1. **Main thread**

   Functionality: The main thread will load a static graph into memory. Use the graph available from this link. It's a social network graph of Github developers with 37,700 users and 289,003 edges. Take only the "musae_git_edges.csv" file and consider it an undirected graph. Each node U should maintain two queues:

   - Wall Queue: Will store the actions generated by the node U herself (see below *userSimulator* thread function).
   - Feed Queue: Will store the actions generated by all neighbors of U (see below *readPost* and *pushUpdate* functionalities).

   Each node should also have a variable mentioning "priority" (value 0) or "chronological" (value 1) signifying order of reading elements from the feed queue. The value of the variable is set randomly between the two choices.

   Then the main thread will create a **userSimulator** thread, a pool of 10 **readPost** threads and another pool of 25 **pushUpdate** threads and wait for the threads. You can also create *additional data structures* suitable for your functionalities. The *node* will be designed by you, so you can be flexible with the elements stored in a node.

2. **userSimulator**

   Functionality: This thread will just simulate user actions. Here we consider that users can do three actions: *post, comment, like.* Here is what the userSimulator thread should do. It will choose 100 random nodes from the graph (between node ids 1 and 37,700). Then for each node, generate n actions, n needs to be proportional to the *log_2(degree of the node)*, since more popular nodes will do more actions. For each action, *action_type* will be chosen randomly from the fixed list of three actions (*post, comment, like*).

   Each action will be a structure having the following fields

   {user_id, action_id, action_type, timestamp (action time)}

There can be more fields too, but the above 4 are compulsory. user_id is simply node id who did the action, *action_id* is a counter variable associated with each node and specific to each action (e.g., 4th like, 5th post etc.), *action_type* and timestamp is simple unix timestamp of generating the action.

Once an action is generated the *userSimulator* thread will

- push it to the *wall* queue of the user node
- push it to a queue monitored by *pushUpdate* threads (see below).

userSimulator will continue till all actions are generated and pushed. Then it will go to sleep for two minutes. After waking up, *userSimulator* will repeat this process.

You have to write the messages of each functionality of the *userSimulator* thread into a log file (sns.log) as well to the terminal. The log file (and stdout) should contain: which random nodes are being selected by pushUpdate, how many actions per random node to be generated and the degree of those nodes (to show the activities are proportional to the degree), and the actions generated.

3. **pushUpdate**

   Functionality: This set of 25 threads will push the *new actions* in the wall queues to the neighbors of the poster node. To do this, *pushUpdate* threads will monitor a shared queue which will be populated by the userSimulator thread. Once there is a new action in the shared queue, a worker thread will dequeue it and go ahead to push updates to all the neighbors of the associated node. For pushing updates the thread will simply add the element to the *feed queue* of each of the neighbors. Note that the feed queue is also shared by multiple threads (see readPost below).

   Also to make the monitoring efficient instead of checking the data structure again and again, use the *pthread_cond_broadcast*, *pthread_cond_signal* and *pthread_cond_wait* functionalities using conditional variables. Naturally the pushUpdate should start acting only when new actions are added by userSimulator.

   Similar to the earlier case, here too you have to write the messages of each functionality of the *pushUpdate* thread into the same log file (sns.log) as well to the terminal.

4. **readPost**

   Functionality: This set of 10 threads will simulate the neighbors reading the actions generated by *userSimulator* thread and pushed by *pushUpdate*. So these threads will monitor the neighbors whose feed queues might have changed (you can be smart on how to detect it) and then deque the actions, print messages of the form “I read action number XX of type YY posted by user ZZ at time TT” and continue till there is no message. It should come back whenever there are more messages to be read.

   Two things to note:

   - Again, to make the monitoring efficient instead of checking the data structure again and again use the pthread_cond_broadcast,

pthread_cond_signal and pthread_cond_wait functionalities using conditional variables.
- Based on whether the node variable is set as "priority" (value 0) or "chronological"(value 1), the order of reading elements from the queue will vary.
  - If the variable value is set as "chronological", then the actions should be read in increasing order of time.
  - If the variable value is set as "priority", then the actions should be read according to the priority of the node which did the action. The priority will be proportional to the number of *common neighbors* between the poster node and reader node (you can precompute it in the main process if you wish).

Similar to the earlier case, here too you have to write the messages of each functionality of the *readPost* thread into the same log file (sns.log) as well to the terminal.

NOTES

- There might be additional data structures needed in this assignment, think judiciously about the design before you start coding.
- You should realize that there are a lot of shared data structures shared between a large number of threads with different functionalities in this assignment. So to avoid race conditions, you need to use locks. However, using too many locks (or a global lock) will slow down all the functionalities and make everything sequential. Not using any locks will create unpredictable answers. So you need to design locks and the functionalities carefully.
- You need to submit a design doc too on
  - Details of what data structures did you use for each functionality (e.g., size of wall queue, feed queue), along with justification?
  - Rationalize your choice for all queue sizes (e.g., analytically or using data)
  - How many locks are you using and for what purpose? Justify how the concurrency (parallel running of threads) is preserved in your design.

Submission Guideline:

- Create a program file (in C or CPP) called sns.c / sns.cpp. The program file can use multiple other files implementing functionalities of userSimulator, pushUpdated and readPost. You can also write everything in a single file. Also create a Makefile that will compile all programs.
- Include a DESIGNDOC.txt file.
- Zip all the files into Ass4_<groupno>_<roll no. 1>_< roll no. 2>_< roll no. 3>_< roll no. 4>.zip (replace <groupno> and <roll no.> by your group number and roll numbers), and upload it on Moodle.
- You must show the running version of the program(s) to your assigned TA during the lab hours.
- **You cannot use any graph library in the assignment.**
- **[IMPORTANT] Please make only one submission per group.**

Evaluation Guidelines:

Total marks for this assignment: 70.

We will check features of your code (including but not limited to) if the computation is correct, you are creating and using the threads correctly, there is no race condition, your code is not mostly sequential and your log contains meaningful and logically valid messages.

Items	**Marks**
Correctness of the main thread (loading the initial graph, created threads, data structures)	15
userSimulator (create actions and push them into queues like wall and other data structures, waking pushupdate)	15
Correctness of pushUpdate (push new actions to neighbors and wake up readPost)	15
Correctness of readPost (in both chronological and priority case)	15
DESIGN DOC (stating the data structures, rationalizing design decision for queue sizes and the design of how locks are used to ensure no race condition and concurrency)	10
Total	**70**