Operating Systems

Homework Assignment #5

Due 18.6.2015, 23:59

Part 1

In this part of the assignment, you will implement a concurrent list-based Set data structure, as in lecture 6.

For that you are provided with a header file *setos.h*, which contains the functions the Set needs to implement – *add*, *remove*, *contains*, as well as functions for initialization and destruction of the Set. Along with the header file a sample *C* file is attached, *setos.c*, which implements the Set in a non-concurrent way (i.e., it is not thread-safe).

You need to implement a *concurrent* Set that is thread-safe, in 3 ways:

1. *setos_coarse.c* - **coarse-grained locking**.
   In this implementation, the entire Set is protected by a lock so that each operation is done individually, without any actual concurrency.
2. *setos_fine.c* – **fine-grained locking** (hand-over-hand).
   In this implementation, each node in the set has its own lock, and add/remove operations acquire the locks in a "hand-over-hand" fashion to achieve concurrency while ensuring the correctness of the Set.
3. *setos_optimistic.c* - **optimistic**.
   In this implementation, traversal of the Set is done without locking, in an "optimistic" way. Once the relevant 2 nodes are found, they are locked and verified. Then, according to the verification, the operation either proceeds or restarts.

For each of the above you need to provide a complete implementation of the Set, matching the interface provided in the header file *setos.h*.

For information on the list-based concurrent Set data structure and its various implementations, refer to lecture 6.

Guidelines

- You **may not** change the header file in any way! Your code will be tested with the original header file.
- **Do not** submit the provided files! Only submit your implementations.
- **Submit** three C files: `setos_coarse.c`, `setos_fine.c`, `setos_optimistic.c`.

Part 2

In this part of the assignment, you will write a server which acts as a network wrapper for a concurrent Set from Part 1 (it can be compiled against any of the implementations).

The server listens for connections, with worker threads constantly running in the background. Whenever a client connects, its socket is enqueued to a shared queue, to be handled by one of the worker threads.

The server holds a **single shared Set**, which the worker threads access. Connecting clients send commands for the worker threads to execute on the Set, to add, remove, or query (*contains*) items.

Write a program `setos_server`. It will receive two arguments: number of threads and port number.

First, implement a simple shared Queue data structure that is thread-safe:

- Implemented by a linked-list (singly-linked) with a global lock, and head and tail pointers.
- The value of each item in the queue is a socket.
- To enqueue or dequeue, grab the lock and perform the required operation (coarse-grained locking).
- Grab and release the lock in appropriate places! Do not hold the lock for performing operations where the lock isn't needed (allocating a queue item before enqueue, etc.).
- Use condition variables where appropriate.

In the main thread:

- Create the required number of worker threads.
- Create a socket and listen for incoming connections (i.e., *accept*).
- When a connection is made, enqueue the new socket into the shared queue, and repeat (*accept* again).

In each worker thread, repeatedly:

- Dequeue a socket item from the queue.
- Read commands from the client. Each command consists of 5 bytes: a single byte specifying the command (0: add, 1: remove, 2: contains), followed by 4 bytes for the key. The value can be left as *NULL*.
- Clients repeatedly send commands until closing the socket. The worker threads should execute these commands on the shared Set, and send back the results.
- Note that a client does not necessarily wait for the results - it may send several commands at the same time, and even close the socket earlier. You server should handle that without error.
- When finished (*EOF* from client, or any error), close the client socket and repeat.

Note: Pressing CTRL+C (SIGINT) should exit the server **properly!**
i.e., use a signal handler to stop any running threads, close sockets, etc., before terminating.
You can & should wait for threads to finish handling any current clients before terminating (i.e., empty the queue).

Guidelines

- You are supplied with a simple client `setos_client.c` - this is the format you server should support. Use it by running it against your server to make sure it works, and expand it for testing. **Do not submit it**.
- **Submit**: `setos_server.c`.