Assignment 3
CS-351
Fall 2015

Due Date: 12/23/2015 at 11:59 pm (No extensions)
You may work in groups of 7

Goals:

1. To experiment with threads, condition variables, and thread pools.
2. To implement a parallel database using threads.
3. To relate threading, process synchronization, and deadlock concepts covered in class to the real-world problem of database access.
4. To discover the challenges of debugging race conditions, deadlocks, and other problems of synchronization.
5. To recognize problems which can solved more efficiently using parallelization.
6. To observe the interplay of the key operating systems concepts covered in class.

Overview

In this assignment you will combine what you learned about threads, synchronization, thread pools, mutexes, and condition variables in order to implement a real-world, parallel hash table. This assignment is a simplified version of a scheme that your instructor had to design for one of his research projects.

Hash tables are widely used for efficient storage and retrieval of data. When the same hash table is accessed by multiple threads, we must ensure that these concurrent accesses are free from race conditions, dead locks, and other problems of thread/process synchronization.
A naive solution is to prevent concurrent accesses to the hash table altogether. That is, prior to accessing the hash table a thread acquires a mutex, performs the access, and then releases the mutex. Such approach, although simple, is very inefficient. In this assignment you will implement a more efficient solution which permits multiple threads to access the hashtable concurrently, as long as they are accessing different hash locations.
The solution works as follows. Let $H[100]$ represent a hash table of 100 cells. Each cell i stores a linked list of records hashing to cell i. Each cell i is protected by a separate mutex $mutex_i$. When a thread wishes to insert/retrieve a record from location i, it first will lock $mutex_i$ protecting location $H[i]$, insert/retrieve the record, and then release $mutex_i$. For greater efficiency your program shall also make use of a thread pool. The thread pool will be implemented using a pthread condition variable about which we learned in class.

Specifications:

You are given two skeleton programs: a server and a client. The server program maintains a hash table of records and a pool of threads (with the latter implemented using a condition variable). The client requests records from the server by sending the ids of the requested records over the message queue. The server retrieves a request from the message queue and wakes up a thread in the thread pool. The awakened thread then uses the record id supplied in the message to retrieve the corresponding record from the hash table and sends the record to the client over the message queue. The message passing functionality has already been implemented for you.

The server has the following structure and function:

1. The server is invoked with two command line arguments specifying:
   - The name of the file storing a list of records.
   - The number of ID lookup threads.

   For example, `./server namesDB.txt 100`

2. When started, the server reads the specified file, and stores the records in the hash table (the functionality for reading and populating the hashtable was already implemented for you). The hash table has the following structure and function:
   - The hash table is an array (or vector) of 100 cells.
   - Each cell is represented as a class containing two items 1) a lock (mutex) protecting the cell; and 2) a linked-list of records hashing to the cell. See the code below:

   ```
   class hashTableCell
   {
           .
           .
           .
                /* The linked list of records (from C++ Standard
                 * Template Library)
           */
           list<record> recordList;


           /**
            * TODO: declare a cell mutex
            */

   };
   ```

   - Each record is represented using `struct record`:

   ```
   struct record
   {
           /* The record id */
           int id;
   ```

```
        /* The first name */
        string firstName;

        /* The first name */
        string lastName;
};
```

- The hash table supports methods `void addToHashTable(rec)` and `record getHashTableRecord(const int& id)` for inserting and retrieving records, respectively.
- The hash keys of records are computed as `record id` mod `hash table size` where `record id` is the id associated with each record (i.e., the `id` field of the record struct).

3. The server then creates the specified number of threads (in function `createThreads()`, which immediately join the thread pool (in function `threadPoolFunc(void* arg)` by calling `pthread_cond_wait()` on a condition variable. Finally, the parent thread calls `recvMessage()` on the message queue, and waits for messages to arrive from the client (already implemented for you).

4. The main thread spends the majority of its life in the `processIncomingMessages()` function. When a new message arrives, the parent thread retrieves the message, adds it to a list of received messages (a globally declared linked list `idsToLookUpList`) waiting to be processed, and wakes up a thread from the thread pool by signaling the condition variable (i.e., by calling `pthread_cond_signal in wakeUpThread()` on the condition variable). The structure of the message, defined in file `msg.h`, mimics the structure of the record stored in the file. **Please note: since this list may be accessed by concurrently by multiple threads, it will need to be protected by a mutex. That is, parent thread locks the mutex protecting the list, adds the received record id, and then releases the mutex.**

5. The awakened thread removes a message from the front of the list (using `getIdsToLookUp()`) and checks the *id* field of the message. The id field indicates that the client who sent the message wants to retrieve the record associated with the respective id.

6. The thread then checks if the record with a given id exists in the table, and if so, retrieves it (`in threadPoolFunc()`). The retrieved record is then sent to the client over a message queue. If the record does not exist, then the server sends back a record with id field set to -1. This will tell the client that the requested record does not exit. ***Please note: this is the most important part. The thread computes the hash cell index using formula*** `record id` ***mod*** `hash table size`***. It then locks the mutex protecting the cell, searches the linked-list at that location for the record matching the id, retrieves the record (if exists), and releases the mutex.***

7. The thread then removes the next message from the list and repeats the same process. If the list is empty, then the thread goes back to the thread pool by calling pthread_cond_wait() on the condition variable.

8. The server also simulates a scenario where the hash table is constantly being updated with new records. You will simulate this by using **5 separate threads** that periodically (e.g., every 1 second) randomly generate records and insert them into the hash table (function

addNewRecords(). The function for randomly generated records, generateRandomRecord() was already implemented for you.

9. When the user presses Ctrl-c, the server catches the SIGINT signal and calls the tellAllThreadsToExit() signal handler. The signal handler, in turn, tells all threads to exit[1], waits for threads to exit, and finally deallocates other resources (e.g., mutexes, the condition variable, and the message queue etc.) and exits. To intercept the signal, you will need to define a custom signal handler.

Note, a file of initial records, namesDB.txt is included with sample files. It contains the names of students from CS-351 (sections 1, 2, and 3), and CS-456 classes. The record file comprises multiple lines where each line contains the following items:

- a unique numerical id
- the first name
- the last name

The client program shall have the following structure and function (most of which has already been implemented for you).

1. The client program connects to the message queue previously created by the server.
2. If the connection is successful, then the client goes into an infinite loop where it:
   (a) Generates a random number between 0 and 1000 representing the record ID.
   (b) Sends the id to the server via the message queue.
   (c) Waits for the server to reply with the requested record.
   (d) If the id field in the server's reply message is NOT -1 (i.e., -1 means the record does not exist), then prints the record.
   (e) Repeats the process.

The client shall be invoked as ./client.

Files provided:

- server.cpp: skeleton file for the server. It contains TODO: comments which are helpful (but not necessarily exhaustive) guides.
- client.cpp: All functionality was already implemented for you.
- pthread.cpp: Illustrates the basic usage of pthreads.
- condvar.cpp: Illustrates the use of condition variables. Implements a simplified version of producer consumer problem which uses buffer size of 1.
- signal.cpp: Illustrates the overriding of default signal handlers.

[1] Please see the comments in the skeleton code for threadPoolFunc(), addNewRecords(), and processIncoming(), which give hints on how to accomplish this

USEFUL TIPS AND RESOURCES

- All sample files can be compiled using the `make` command.
- ***When running this assignment, you will need two separate terminals: one for the server and one for the client. Naturally, the server needs to be invoked first. This link gives a YouTube video which illustrates (an example of) what your output should look like: https://www.youtube.com/watch?v=nQZXDPYucgI***
- You may want to use vector and linked list data structures provided by the C++ Standard Template Library (STL):
  - **STL vectors:**
    http://www.cplusplus.com/reference/stl/vector/
    http://www.cprogramming.com/tutorial/stl/vector.html
  - **STL linked lists:**
    http://www.cplusplus.com/reference/stl/list/
    http://www.cprogramming.com/tutorial/stl/stllist.html
- Your server program may not compile unless you append the `-lpthread` switch to the end of your compilation line.
- **Signal handlers:** Pressing `Ctrl-c` sends a `SIGINT` signal to the process. In this assignment you will need to override the default signal handler for `SIGINT` (see the specifications for the server). See the link below for more information on overriding signal handlers.
  http://www.cs.utah.edu/dept/old/texinfo/glibc-manual-0.02/library_21.html
- **Pthreads tutorial:**
  http://www.cs.nmsu.edu/ jcook/Tools/pthreads/library.html
- **Condition variables:**
  http://www.qnx.com/developers/docs/6.3.2/neutrino/lib_ref/p/pthread_cond_wait.html
- **System V message queues:**
  http://beej.us/guide/bgipc/output/html/multipage/mq.html
- You can view all allocated message queues using the `ipcs` command. You can delete a message queue from command line using `ipcrm -Q <KEY SHOWN BY IPCS>`. **message queues are a finite resource - please delete all your queues after you are done. If all available queues are used up, then somebody will not be able to complete their assignment.**

BONUS 1:

Implement a multi-threaded quicksort that uses a thread pool similar to the hash table program. Your program must perform parallel sorting.

BONUS 2:

Complete assignment 4, posted on Titanium.

SUBMISSION GUIDELINES:

- ***This assignment MUST be completed using C or C++ on Linux.***
- ***You may work in groups of 6***.
- ***Your assignment must compile and run on the LISP server.*** Please contact the CS office if you need an account.
- Please hand in your source code along with the makefile electronically (do not submit .o or executable code) through **TITANIUM**.
- You must make sure that the code compiles and runs correctly.
- Write a README file (text file, do not submit a .doc file) which contains
  - Your name and email address.
  - The programming language you used (i.e., C or C++).
  - How to execute your program.
  - Whether you implemented the extra credit.
  - Anything special about your submission that we should take note of.
- Place all your files under one directory with a unique name (such as `p3-[userid]` for assignment 3, e.g., `p3-mgofman`).
- Tar the contents of this directory using the following command. `tar -cvf [directory_name].tar [directory_name]` E.g., `tar -cvf p3-mgofman.tar p3-mgofman/`
- Use TITANIUM to upload the tared file you created above.

Grading guideline:

- Program compiles: 10'
- Correct input/output format: 20'
- Correct implementation of parallel hashtable: 35'
- Correct implementation of condition thread pool: 30'
- README file: 5'
- Bonus 1: 15'
- Bonus 2: 15'

- Late submissions shall be penalized 10%. No assignments shall be accepted after 24 hours.

Academic Honesty:

Academic Honesty: All forms of cheating shall be treated with utmost seriousness. You may discuss the problems with other students, however, you must write your **OWN codes and solutions**. Discussing solutions to the problem is **NOT** acceptable (unless specified otherwise). Copying an assignment from another student or allowing another student to copy your work **may lead to an automatic F for this course**. Moss shall be used to detect plagiarism in programming assignments. If you have any questions about whether an act of collaboration may be treated as academic dishonesty, please consult the instructor before you collaborate. Details posted at http://www.fullerton.edu/senate/documents/PDF/300/UPS300-021.pdf.