Assignment 3
CS-351
Fall 2018
Due Date: 12/23/2018 at 11:59 pm (No extensions)

Goals:

1. To experiment with threads and mutexes.
2. To use threads in order to speed up database access operations.
3. To relate the threading, process synchronization, and deadlock concepts covered in class to a real-world problem of database access.
4. To discover the challenges of debugging race conditions and deadlocks.
5. To recognize problems which can solved more efficiently using parallelization.
6. To explain the need for parallel hash tables.
7. To implement a parallel hash table.
8. To observe the interplay of all operating systems concepts covered in class.

Overview

In this assignment you will combine what you learned about threads, mutexes, and message passing 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. It reflects a problem which you you are highly likely to encounter in your career.

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, deadlocks, 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 $mutex_i$.

Specifications:

You are given two programs. A server and a client. The server program maintains a hash table of records and a set of threads. The client requests records from the server by sending record ids over a message queue. The server will retrieve the message from the message queue, look up the requested record in the hash table, and then send back the fetched record to the server. The server will maintain a set of threads that allow it to respond to multiple requests in parallel. The message passing functionality has already been implemented for you.

The server has the following structure and functions:

1. The server is invoked with two command line argument specifying the number of threads.
   - The name of the file storing a list of records.
   - The number of threads used for fetching records.

   For example, `./server namesDB.txt 10`

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(const record& 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. When the server starts the parent thread creates a set of threads responsible for fethcing records (`void createFetcherThreads(const int& numFetchers)`. We will refer to these as the fetcher threads. Each fetcher thread calls function `void* fetecherThreadFunction(void* arg)` where it handles responding to the client's requests.

4. The parent thread also creates `NUM_INSERTERS` (globally defined) threads (`createInserterThreads()`) that constantly generate random records and insert them into the hashtable. This is used to simulate parallel updates to a live database. The function `getHashTableRecord()` that generates a random record had already been implemented for you.

5. The parent thread then listens on a message queue for incoming requests.

6. When a new request arrives the parent thread retrieves the message, adds it to a list of pending requests (globally declared linked list `idsToLookUpList`) waiting to be processed (in function `processIncomingMessages()`), and 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.**

7. One of the threads in `fetcherThreadFunction()` (that execute in parallel with the parent thread) remove a request from the front of the `idsToLookUpList` list (`using getIdsToLookUp()`) and checks the *id* field of the message. The id field indicates that the client requests the record with the respective id.

8. The thread then checks if the record with a given id exists in the table, and if so, retrieves it. 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.***

9. The thread then removes the next message from the list and repeats the same process. If the list is empty, then the thread exits.

10. When the user presses `Ctrl-c`, the server catches the `SIGINT` signal, removes the message queue using `msgctl()`, deallocates any other resources (e.g. mutexes), tells all the threads to exit and calls `pthread_join()` for them. 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 and CS-458 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. record does not exist), then prints the record.
   (e) Repeats the process.

The client shall be invoked as `./client`.

The sample client output is:

```
$./client
messageType=2 id=46 firstName=Daniel lastName=Ariza
messageType=2 id=62 firstName=Michael lastName=Busslinger
messageType=2 id=31 firstName=Tommy lastName=Chao
messageType=2 id=8 firstName=Illianna lastName=Izabal
.  .  .
```

Files provided:

- `server.cpp`: skeleton file for the server. It contains `TODO:` comments which are helpful (but not necessarily exhaustive) guides.
- `client.cpp`: skeleton file for the client. Most functionality was already implemented for you.
- `msg.h and msg.cpp:` contain an implementation of the message queue library that wraps the System V message queue functions for easier access.
- `pthread.cpp`: illustrates the basic usage of pthreads.
- `namesDB.txt`: a sample text file of initial records that you can use for testing.
- `signal.cpp:`illustrates the overriding of default signal handlers.

USEFUL TIPS AND RESOURCES

- All sample files can be compiled using the `make` command.
- You may want to use vector and linked list data structures provided by the C++ Standard Template Library (STL):
- To prevent inserter threads from generating too many records to quickly (and exchausting program memory), you can use `usleep(5)` that will cause the calling thread to sleep 5 microseconds before generating another record.
  - **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
- **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:

Expand the program to support at least three simultaneous clients. Also, make use of the a thread pool where the fetcher threads sleep and are woken up only if there is an incoming request.
You may want to read about condition variables. For example, https://computing.llnl.gov/tutorials/pthreads/.

SUBMISSION GUIDELINES:

- ***This assignment MUST be completed using C or C++ on Linux.***
- ***You may work in groups of 5.***
- ***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 `p4-[userid]` for assignment 3, e.g. `p4-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 p4-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: 65'
- README file: 5'
- Bonus: 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.