# DB

CS167 Operating Systems

July 6, 2012

*This assignment must be completed by both CS167 and CS169 students.*

## 1 Introduction

In this project you will develop and test a multi-threaded program in C. You will produce a database server that handles any number of clients. The database contains a number of pairs of names of countries and the names of their capitals. Clients may query it, asking for the capital of a county; they may also add new pairs to it and delete pairs from it. We provide you with a single-threaded, single-client version of the program in `/course/cs167/asgn/db`. You are to modify it in stages, first adding support for multiple clients with multiple threads, then making the databse thread-safe, and finally adding clean-termination and timeout features.

Clients interact with the database via `xterm` windows; we provide the windowing code and there is no need for you to modify it (or even look at it). Associated with each client window is a thread that handles all the client's interaction with the database. It waits for input from the client, parses client commands, calls database code as required, and returns results to the client. The database is rather simple – it's a collection of name-value pairs organized as a search tree (not a balanced search tree – making that thread-safe and high-performance is extremely difficult). The program's source consists of six files. The file `server.c` contains the main code for the server; you will modify this file when implementing multiple client windows. The files `db.c` and `db.h` contain the code that manages the database; you will modify these when implementing database locking. The files `window.h`, `window.c` and `interface.h` contain the code for communicating with and managing the windows; you will never need to modify the contents of these three files.

A completely working binary is available in `/course/cs167/demo/db/DemoServer`. If you copy the demo to your working directory, make sure you place it in the same directory as a working `interface` binary. The demo handles all parts of the assignment and takes one argument on the command line: the number of seconds to wait before client threads self-destruct (see the section on timing out).

## 2 Part 1: Multiple Clients

A single-threaded, single-client version of the program is in `/course/cs167/asgn/db`. Your task is to turn it into a multithreaded program that handles multiple clients. For now you are only concerned about queries. Later, you will deal with other concurrency issues.

### 2.1 Multiple Client Windows

The directory contains a `Makefile`. Run the command `make` to produce an executable called `server` and a window will appear (the window does not run a shell, but rather a special program that communicates with the database server). Within this window, run the command `f caps`. This will initialize the database by running a script contained in the `caps` file. You can give it queries of the form `q <country>` and it will respond with that country's capital, if it is in the database. For example, try `q papua_new_guinea`.

Note that all letters are lower case and underscores are used instead of spaces. If a country has a common abbreviation like "usa" or "uk", the abbreviation is used. Typing a line containing only `Ctrl-D` in the window causes the client to self-destruct and the window to disappear.

You should modify `server.c` so that when you start up the program, no window is created automatically. Instead, every time you type enter in the terminal where you're running `server`, a new window should be created along with a new thread to handle it. When you type commands into any of the new windows, they should be handled just as in the single-threaded version of the program. However, commands typed into any of the windows access the same shared database.

You must implement all the changes marked `TODO (Part 1)` in `server.c`. Also note that the program should be terminated with care: one should first terminate the various client windows (by typing `Ctrl-D` in them), and only then terminate the main window (by typing Ctrl+D into it). We will fix this in part 3.

### 2.2 Implementation Details

Note that in the single-threaded version, the (only) thread calls `RunClient` – see the code in `server.c`. In your multithreaded version, you must arrange so that every time enter is typed, rather than the main thread calling `RunClient`, a new thread is created and detached and it calls `RunClient`. This shouldn't require you to write much code, but make sure that you modify `Makefile` so that the `-pthread` flag is passed to `gcc`!

The program is designed in an object-oriented fashion: constructor routines `malloc` and initialize storage, while destructor routines perform cleanup tasks and free storage. In the single-threaded version of the program, the main routine (in `server.c`) calls `Client_constructor` to allocate and initialize an instance of `Client_t`. It then calls `RunClient`, passing it the pointer to the `Client_t` instance, which does the work.

As before, you must implement all the changes marked `TODO (Part 2)` in `server.c`. We strongly suggest you use the following strategy: modify `Client_constructor` so that it creates a thread that calls `RunClient`. Thus each time you type enter, the main thread simply calls `Client_constructor`.

The file `server.c` contains the main routine. You need to modify it to support the creation of additional client threads. The `window.c` file contains the implementation of client windows. Windows are created by calling `window_constructor`. To communicate with the window, a thread calls the `serve` method, which sends the answer of the previous query (or the empty string, if there was no previous query) to the window, and waits for and returns the next query. See the code in `server.c` for an example of how to use these methods.

Note that the only way for the program to cause a window to disappear is to call `window_destructor`, passing it the handle provided by `window_constructor`. `Client_destructor` makes this call: so make sure that client threads call `Client_destructor` when they terminate!

The `db.c` file contains the code for parsing commands and the implementation of the database, which is structured as a collection of items of type `Node_t`.

The `interface.c` file contains the code that is invoked by xterm to handle the communication between you and the database server. There should be no reason for you to modify this code. (It is not multithreaded code and it runs in a separate process.)

## 3 Part 2: Thread Safety

Part 1 was easy. Part 2 starts off less so and gets tougher still. You are to make the database thread-safe. And to help you test your code, you are to add some additional features.

### 3.1 Database Locking

The database consists of a collection of nodes, organized as an unbalanced binary search tree. An empty database consists of a header, pointing to `NULL`. Otherwise the database contains some number of name-value pairs, each stored in a node. Each node contains two pointers to nodes, a left child and a right child. All names in the nodes in the tree pointed to by a node's left child are lexicographically less than the node's name; all names in the nodes in the tree pointed to by the node's right child are lexicographically greater than the node's name. The database implementation is in `db.h` and `db.c`.

The `add` routine calls search to verify that the node to be added is not present. `search` returns in its third argument (an out argument) a pointer to the node that should be the node-to-be-added's parent. `add` then creates the new node and connects it to its parent.

`remove` is a bit more difficult. It first checks to make sure the node we're deleting exists, by calling `search`. `search` returns a pointer to the node (if it exists) and, in its third argument, a pointer to that node's parent. Let's call the node we're deleting `dnode`. If it has no children, it is simply deleted and the pointer that referred to it (in its parent) is set to `NULL`. If it has one `NULL` child pointer, then it's also easy: the pointer field that referred to it in its parent is set to point to the non-`NULL` child. If both children are non-`NULL`, things are a bit tougher.

Consider the subtree headed by `dnode`'s right child. Let's call the node with the lexicographically smallest name in that subtree `xnode`. This node's name is, however, lexicographically greater than all the names of the nodes in the subtree headed by `dnode`'s left child. Suppose we replace `dnode` with `xnode` (i.e., we copy `xnode`'s name and value into `dnode` and remove `xnode` from where it was in the right subtree). The resulting tree is well-formed: everything in `xnode`'s left subtree has a name that's lexicographically less than `xnode`'s, and everything in `xnode`'s right subtree has a name that's lexicographically greater than `xnode`'s. Furthermore, since we've gotten rid of `dnode`, the resulting tree doesn't have `dnode` in it. So, we reduced the problem of deleting `dnode` to that of deleting `xnode`. But, since `xnode` was the smallest node in the left subtree, it's easy to delete, since it has no left child.

There are two ways to make this code thread-safe. The first is the easiest and is known as coarse-grained locking. We have a single readers-writers lock protecting the entire tree. A thread simply locks the readers-writers lock (read-locking it or write-locking it, as appropriate) before doing an operation and unlocks it afterwards. The more interesting approach is known as fine-grained locking, in which a readers-writers lock is associated with each node. Thus only the portion of the database being manipulated is locked.

You are to first implement coarse-grained locking. Then, you are to implement fine-grained locking. What you hand in should be the fine-grained version. Note that doing the fine-grained version is difficult. Make sure your coarse-grained solution works before you attempt a fine-grained one. Don't start working on the fine-grained solution until you've implemented the `start` and `stop` commands. If you can't complete fine-grained locking within the allotted time, partial credit will be given for a working implementation using coarse-grained locking – but don't forget to work on part 3!

To help you test your code, you should further modify `server.c` by adding `start` and `stop` commands: typing `s` in the main window (the one in which you typed server) causes all client threads to stop handling input until `g` (for go) is typed in the same window. In particular, you are required to implement (and call) three functions in `server.c`: `ClientControl_wait`, `ClientControl_stop`, and `ClientControl_release`. See the comments above these functions for a description of their required behavior. We strongly suggest that you use a condition variable (combined with a mutex) to implement this feature.

Within the `tests/` directory are three scripts to help you test your solution: `test1`, `test2`, and `WindowScript`. Note that if you add the `-DAUTOMATED_TEST` to your `Makefile` and rerun `make`, then whenever a client window is created, the client automatically executes the contents of `WindowScript` without your having to type anything in it. This might make it even easier to test your code.

### 3.2 Implementation Details

The name `remove` has recently been turned into what's effectively a reserved word in C – there's a routine of that name used in the standard I/O library. Thus we have renamed our remove function `xremove`.

## 4 Part 3: Shutdowns and Timeouts

Now for the fun part. Three things we discussed in class that we haven't yet used are timeouts, signals, and cancellation. We correct that omission here, by adding three new features to the database program: full shutdown, client-only shutdown, and timeout.

### 4.1 Full Shutdown

An annoying problem with the database program up to now is that one must type `Ctrl-D` in each window to cause them to go away. What would be nicer is if typing `Ctrl-D` in the main window cleanly terminates the program: all client windows close, all client threads terminate, the main thread terminates, and the program goes away cleanly.

Modify your code, using the cancellation facilities of POSIX threads, so that this indeed is the case. Try not to have any storage leaks: if a window is terminated before the control window terminates, everything associated with the terminal window should be cleaned up (and freed if necessary).

Note that `fgets`, which is used in the `interpret_command` routine of `db.c`, is not a cancellation point – you should make an explicit call to `pthread_testcancel` after the call to `fgets` so that a client thread that's cancelled here will find out about it. Also note that `window_constructor` must be executed atomically: cancellation must be disabled within it.

### 4.2 Client-Only Shutdown

Modify your solution to multiple client windows so that when it receives `SIGINT` (generated by typing a `Ctrl-C` in the main window, from which you launched the program), all the client threads stop what they are doing and terminate, but the server thread continues to function. You should have the first thread create a child thread which waits for `SIGINT`s to occur, then cancels the current client threads.

### 4.3 Timeout

Modify your solution even further by associating a timeout with each client thread. Client threads time out after waiting a given period of time without receiving input: they close their window and terminate. The timeout period is given in seconds as an argument to main.

### 4.4 Implementation Details

As before, you must implement all the changes marked `TODO (Part 3)` in `server.c`.

Our strategy for doing the first two features is to maintain a list of all client threads. When a thread is created, it puts itself into the list. When we want to force a thread to terminate, we cancel it. When a thread does terminate (either normally or because it's been cancelled), it pulls itself from the list.

To implement timeouts, we associate with each client thread a *watchdog* thread. Every time a client thread receives input, it stores in a timeout variable the time at which it should be terminated if it receives no more input (thus if the timeout period is 30 seconds and the client thread receives input at 3:01:00, it stores 3:01:30 into the timeout variable). The watchdog thread initially calls `nanosleep` so that it wakes up after the first timeout period. When it wakes up, it checks to see if its client thread has received input. If not, it cancels the client thread and terminates itself. Otherwise it goes to sleep again, waking up at the time specified in the current value of the client's timeout variable. It does this repeatedly until either it cancels the client or the client terminates for some other reason (and cancels the watchdog thread).

Make certain that there are no storage leaks and that all threads are correctly terminated when their time comes. Everything should be properly cleaned up when `Ctrl-D` is pressed in the main window. All client threads, watchdog threads, and windows should disappear when `Ctrl-C` is typed into the main window (generating a `SIGINT`). When a `Ctrl-D` is typed in a client window, the associated client thread and watchdog thread should disappear.

To assist in making your code "bomb-proof", see the `AUTOMATED_TESTING` `Makefile` option mentioned above.

## 5 Handing In

To hand in your finished assignment, please run this command while in the directory containing your code: `/course/cs167/bin/cs167_handin db`.