Assignment 1: Multithreaded Programming

August 24, 2012

This exercise entails producing a simple 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 country; they may add new pairs to it and delete pairs from it. We provide you with a single-threaded, single-client version of the program. You are to modify it in stages, first adding support for multiple clients with multiple threads, and then making the database thread-safe.

We provide you with a skeletal code for the assignment in C. In it, 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 is a collection of name-value pairs organized as a unbalanced search tree.

The program's source consists of six files. *server.c* contains the mainline code of the server; *db.c* and *db.h* define the database; *window.c*, *window.h* and *interface.c* contain the code for communicating with and managing the windows.

1 Multithreading the server

A single-threaded, single-client version of the program is provided to you. Your task is to turn it into a multithreaded program that handles multiple clients. For now you are only concerned about queries. In the next part you deal with other issues.

The directory contains a *Makefile*. Run the command `make` to produce an executable called *server*. Execute `server`; a window will appear. Within this window, type the command[1] `'f caps'`. This initializes 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. E.g., try `'q papua_new_guinea'` - note that all letters are lower case and underscores are typed where blanks should be. If a country has a common abbreviation, e.g., usa and uk, the abbreviation is used. Typing a line containing only `<CONTROL>+D` in the window causes the client to self-destruct (and for the window to disappear).

Assignment Part A

Modify *server.c* so that when you start up the program, no window is created automatically. Instead, every time you type the command `'e'` (in the window in which you are running server), a new client (and window) is 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.

Note that the program should be terminated with care: one should first terminate the various client windows (by typing `<CONTROL>+D` in them), and only then terminate the main window in the same way.

[1] A command is defined by a single line of input. Thus, console commands are followed by hitting `<ENTER>`.

2 Making the server thread-safe

In this part your task is to make the database thread-safe. To help you test your code, you are to add some additional features.

Database structure and API

The database consists of a collection of nodes, organized as an unbalanced binary search tree. An empty database consists of a dummy head node with no children. 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 smaller 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 can be found 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 parent of the node to be added. `add` then creates the new node and connects it to its parent.

`xremove`[2] is a bit more difficult. It first checks to make sure the node we are 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.

If it has no children, it is simply deleted and the pointer that referred to it (in its parent) is set to `NULL`. For example, consider the tree shown in the figure. If we want to delete node H, we only need to set the right child pointer of G to `NULL` before freeing the memory occupied by H.

If it has only one child (the other child pointer points to `NULL`), then it is also easy: the pointer field that referred to it in its parent is set to point to the non-null child. For example, if we want to delete node G, we set F's right child pointer to H.

If it has two children, things are a bit tougher, for example when we want to delete node B in the shown tree. Consider the subtree headed by B's right child, that is the subtree headed by F. In this tree, the node with the lexicographically smallest name in that subtree is the node C and as such it is lexicographically greater than all the names of the other nodes in the subtree. Suppose we replace node B by node C (that is we overwrite B's values but do not touch the pointers to the children) and remove the old C node from the tree. The resulting tree is well-formed: everything in the left subtree of C has a name that is lexicographically smaller than C, and everything in C's right subtree has a name that is lexicographically greater than C. Furthermore, since we overwrote B's contents, the resulting tree does not have B in it. Thus, we reduced the problem of deleting B to that of deleting C. But, since C was the smallest node in the right subtree, it is easy to delete, since it has no left child.

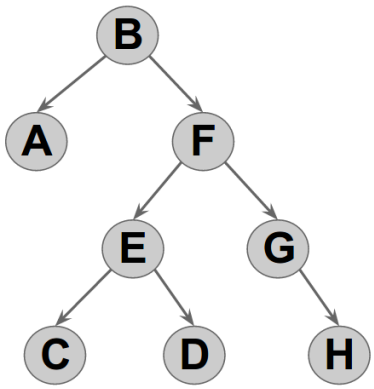

Figure 1: Sample unbalanced binary tree

[2]We avoid naming the method `remove` since this method name is already used by the standard I/O library.

Assignment Part B

In this part you are going to implement coarse-grained locking. That is, 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.

For example, consider there are two threads running (A and B). Thread A executes a query first, and thus read-locks the tree. If thread B wants to execute a query as well, it can do so since the tree is only read-locked (multiple read accesses to the tree are allowed). If thread B wants to modify the tree though, it blocks until the tree is no longer locked since it needs to write-lock the tree for this operation.

In the last part of this assignment (see part C) you will implement a different locking approach. To make switching between the two locking approaches easier (and to support grading the assignment parts independently), you have to embed the code required for coarse-grained locking into `#ifdef COARSE_LOCK` macro blocks. In this way you can easily enable coarse-grained locking by adding the `'-D COARSE_LOCK'` flag to the `CFLAGS` variable in the *Makefile*.

To help you test your code, you should further modify *server.c* by adding start and stop commands: if the command `'s'` is typed in the server window all client threads should stop handling input until `'g'` (for go) is typed in the same window. Use a condition variable (combined with a mutex) to implement this feature.

Within the directory are two scripts to help you test your solution: *test1* and *test2*. There's another lengthy test script named *WindowScript*, which is automatically executed in a newly created client window if you compile your code with the `'-D __RUN_WINDOW_SCRIPT__'` flag. For this, simply add this flag to the `CFLAGS` variable in the *Makefile*.

Assignment Part C

In this part you are to implement fine-grained locking. That is, instead of having a single lock for the whole tree, each node has its own readers-writers lock. Thus only the portion of the database being manipulated should be locked.

Make sure that a node is only locked if this is necessary for consistency. For example, in the sample tree shown earlier, if you want to delete node E, node F needs to be locked as well (since it's child pointer gets modified), but you do not need to lock node A since this part of the tree is never touched.

Embed your fine-grained locking code into `#ifdef FINE_LOCK` macro blocks, and enable this mechanism by using the flag `'-D FINE_LOCK'` instead of `'-D COARSE_LOCK'` in the *Makefile*'s `CFLAGS` specification.

Carefully test your code and make sure that you have the correct understanding of how to locking mechanism is expected to behave. For that, consult the documentation of the functions you use.