Assignment 1: Multithreaded Programming

January 28, 2013

This exercise entails producing a simple database server that handles any number of clients. The database maps a string-valued key to a string-valued value. The assignment code includes a sample configuration that creates a database containing 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. You will implement and contrast three locking strategies in the database: a coarse strategy, a readers/writers locking system, and finally a fine grained readers/writers system.

This is the sort of exercise you would go through in designing any threaded library. The simple client/server setup is a scaffold in which to test your implementation.

For the assignment we will ask you to turn in a UNIX tape archive file (tar file) containing your code and the various write-ups from the subsections. Each subsection will describe what to include, and we summarize at the end.

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_coarse`. Execute `server_coarse`; 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: 15 points

Modify *server.c* so that when you start up the program, no window is created automatically. Instead, you will add a selection of commands that make the server useful for testing your database implementations.

The first command you will add is the command *'e'*. Typing it to the server program (in the window in which you are running textttserver_coarse), will create a new client (and window) 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,

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

shared database. In doing this you will want to use the *pthread_create* library function. In parsing input lines, it will be helpful to use the standard library call *getline* and the word parsing routines in *words.c*.

Starting these threads is only half the battle. You will also need to write code that joins these threads and collects their state and resources. There are a variety of ways to do this, but you will certainly want to use the *pthread_join* call. If you need to share data between threads, for example the number of threads started or stopped, you will need to use the *pthread_mutex_lock* and *pthread_mutex_unlock* calls as well as the *pthread_cond_wait*, *pthread_cond_signal*, and *pthread_cond_broadcast* calls. These implement the monitor functionality we talked about in class.

Of course, those calls are suggestions. You can feel free to use any thread synchronization that you prefer.

The desired behavior is that the server can create one or more windows, each served by a thread. When a window exits (because a user types <CONTROL>+D in it) the server joins the thread serving it and collects its resources. When the user types <CONTROL>+D into the server window, the server prints a message indicating it is terminating and waits for any running threads/windows to terminate. Then it exits. If you start any threads not associated with a window, stop them before termination as well.

Once *e* is working, add a command *E* that takes an input and output file as parameters. Invoke the *client_create_no_window* routine in *server.c* to create a client that reads commands from the first file and prints the results to the second. You will see that *client_create_no_window* does the work of connecting a client object to those files and providing the same interface as other client objects. It is also helpful to know that specifying /dev/stdout as the output file will print to the screen. Each client created with *E* should be served by a separate thread, just as the interactive windows created by *e* are.

The next commands to implement are *s* and *g*. When *s* is given, any running client threads stop processing commands and any new client threads stop before processing their first command. The main server input thread (the one reading these new commands) continues to run. When *g* is given, all stopped threads may proceed.

This will require sharing stop/start data between the threads, so synchronization is needed.

Finally, add a command *w* that suspends the server's command reading loop in order until all currently existing client threads have exited. Issuing *w* must issue an implicit *g* command. Waiting for client threads that are all stopped is not useful.

The commands to implement are:

- *e* create an interactive client in a window
- *E* create a non-interactive client that reads commands from one file and writes results to another
- *s* stop processing command from clients
- *g* continue processing command from clients
- *w* Stop processing server commands (*e, E, s, g*) until currently running clients have terminated.

The combination of these commands will give you a simple scaffold from which to test your database locking system. Running multiple interactive windows will let you test simple coordinated operations. The file-reading and writing clients will be useful in stress testing. The *s* and *g* commands will let you queue up several file-reading clients and start them operating on the database together. Finally the *w* command will let you initialize the database to a known state before letting your clients loose on it.

You will be graded on how well you implement these commands. If you change the semantics or add additional commands, let us know in your write-up. If you change the semantics of any of the required commands, be prepared to argue that your version is more interesting to implement as well as being useful. Extensions do not require any justification.

Things to hand in

A design document describing how you implemented each command. This document must be called *commands.pdf* and be in PDF format. It does not need to be long, but for each command include a few sentences describing your implementation. For example: "the s command sets a shared flag that is protected by locks. Client code checks that flag before processing a command. The functions used are ..."

The modified *server.c* and any other changed files. Any changes to the Makefile. These will be part of the complete code you hand in.

Grading Guidelines

The *e* and *E* commands are worth 10 points, total. 5 points for a correct implementation, 5 points for the design document and comments.

The remaining commands are worth 5 points, total. 3 points for a correct implementation, 2 points for the design document and comments.

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*.

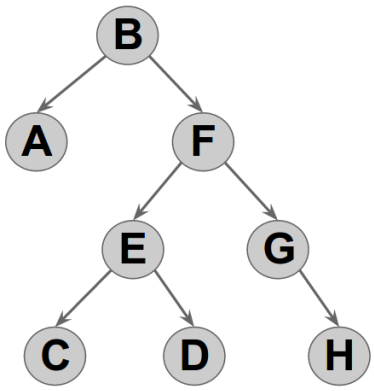

Figure 1: Sample unbalanced binary tree

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.

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

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.

Assignment Part B: 5 Points

In this part you are going to implement coarse-grained locking. Each thread that wants to access the database will acquire an exclusive lock to the tree and carry out their operation. All accesses will be completely serialized.

Though this implementation is the easiest, it is worth designing some of your testing suites and methods here. While you can initially test your changes interactively, you will quickly find it useful to create text files that you can pipe into the server process via redirection to carry out tests.

Make the changes in this phase to *db_coarse.c*. The *server_coarse* program will use them.

We will run our own series of tests to confirm that your locking works correctly under load. If you design a particularly interesting test in this phase or one of the later ones, please let us know. We would be happy to run it against your peers' code.

Things to hand in

A brief description of your design, as in part A. This file must be called *coarse.pdf*. Include any information relevant to your design and implementation, including known defects. We will look for the unknown ones.

The modified code and Makefiles as part of your complete code package.

Grading Guidelines

The coarse locking implementation is worth 5 points, total. 3 points for a correct implementation, 2 points for the design document and comments.

Assignment Part C: 10 Points

In this part you are going to implement coarse-grained locking that differentiates between readers and writers. Many readers can query a database simultaneously, but only one writer can be making changes at a time. Readers can share a lock, but a writer must have exclusive access to the database – no readers or other writers.

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.

Within the directory are two scripts to help you test your solution: *test1* and *test2*. There's another lengthy test script named *WindowScript*. These are not exhaustive tests; some of your assignment is to think about and design tests for these parts.

Make the changes in this phase to *db_rw.c*. The *server_rw* program will use them.

Things to hand in

A brief description of your design, as in earlier parts. This file must be called *rw.pdf* and be in PDF format. Include any information relevant to your design and implementation, including known defects. We will look for the unknown ones.

The modified code and Makefiles as part of your complete code package.

Grading Guidelines

The readers/writers locking implementation is worth 10 points, total. 5 points for a correct implementation, 5 points for the design document and comments.

Assignment Part D: 10 Points

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.

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. You will want to look more deeply into the internals of the database operations here and think about how the locks will be used.

Make the changes in this phase to *db_fine.c*. The *server_fine* program will use them.

Things to hand in

A brief description of your design, as in earlier parts. This file must be called *fine.pdf* and be in PDF format. Include any information relevant to your design and implementation, including known defects. We will look for the unknown ones.

The modified code and Makefiles as part of your complete code package.

Grading Guidelines

The fine-grained readers/writers locking implementation is worth 10 points, total. 5 points for a correct implementation, 5 points for the design document and comments.

Assignment Part E: 5 points

In this part you will look at the performance implications of the different locking implementations. First, generate a database workload that runs faster under each more complex locking system – slowest under coarse locks and fastest under fine grained locks. Explain why your workload exhibits this property.

You will want to create input files that will recreate the load, and measure run times of the various programs under that load. The input files will be both server input files (*E, s, g,* or *w* commands) and client input files (*a, d, q,* or *f* commands).

When you design your workload, make sure that your machine is making use of more than one processor, or you will not see as significant an improvement. Explain why the locking strategy more important on multi-processor machines.

The specific parameters of the workload you generate will depend quite a bit on the specific parameters of your local machine. Significant variation can result from processor and OS features and the VM system will not hide these from you. Because of this, a correct description of the workload and understanding of how the locking interacts with it is worth more points than the demonstration. However, having a demonstration in hand will make you more certain it your understanding is correct.

Things to hand in

Include a description of your workload in natural language, and an explanation of why it performs differently under the three strategies. This can be a qualitative discussion. Include your timing results and where those results were gathered. Answer the question about multiprocessing and uniprocessing as well.

Give the names and functions of your server and client input files so that graders can run your workload. If you add the tests as Makefile targets (which is not required) include that information as well.

The file with all this information must be called *performance.pdf* and be in PDF format.

Grading Guidelines

The performance analysis is worth 5 points. 2 points for the discussion of performance differences, 2 points for the multiprocessor/uniprocessor discussion and 1 for a working example.

3 The Code

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 you can accomplish the assignment without modifying it or looking at it. Interested students are welcome to investigate it, of course. 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:

- *server.c* contains the mainline code of the server. It creates and runs clients, calling into the database code. You will add multithreading and new commands to this file in the first part of the assignment.
- *db.h* defines the database data structures and the server to database interface.
- *db_coarse.c*, *db_rw.c* and *db_fine.c* define the database implementation. You will specialize each of these to implement a different locking strategy. The initial versions in the assignment are the same.

- *window.h* defines the window data structures and the server to client (window) interface. You do not need to modify this file.
- *window.c* contains the implementation of the client and window code. You do not need to modify this code.
- *interface.c* is compiled to a separate executable that is run by the *xterm* program to gather user data from a window and print results. You do not need to modify this code.

Each of these implementation files is commented, both describing the interfaces and the implementation. Additional information about the design is given below.

The Makefile supplied creates three server programs (*server_coarse serve_rw* and *server_fine*, each linked with a the corresponding database implementation (*server_coarse* is linked with *db_coarse.o*, which is generated from *db_coarse.c*). The *interface* program is also generated by the Makefile.

Running

```
make
```

or

```
make all
```

from the code directory will generate the programs. Running

```
make clean
```

will delete the programs and the intermediate files. Running

```
make clean all
```

will delete the intermediates and remake everything. Other details about make and Makefiles are available from google. This is a GNU Makefile.

What To Hand In

A UNIX tar file containing the source code and write-ups of each sub assignment. Only include the source code, not executables.

Submission details are forthcoming.