Project Database

Due: December 16, 2014

1 Introduction

When two people love each other very much, they go to the window and whistle for the stork. The stork has a nine month backlog, but when she gets around to their request, she delivers a baby to their front porch. Whistling is the easy part, but finding the perfect name in those nine months is another story. Currently in order to find a name, parents need to go to Baby Names R Us to search through BabyBase, the baby name database, to see all of the different names they can choose. Ideally, parents would be able to search for baby names from home. Unfortunately, BabyBase does not currently support multiple clients. Furthermore, even if it did, it was never written to be thread-safe. The task to fix this has fallen on your high chair.

In this project, you will be creating a simple server to manage a database of key-value pairs. Clients can search for a key in the database, add new entries, and remove existing entries. You are given source code that allows a single client to access the database. You will progressively modify this code, first adding support for multiple clients, then making the database thread-safe, and finally adding thread cancellation and timeout features.

2 Support Code

The support code for this project consists of the following files:

- *server.c*: A C file containing the program's main function and related code.
- *db.h*: A header containing function declarations for the database.
- *db.c*: A C file implementing database functionality.
- *window.h*: A header defining the `Window` abstraction that you will use to set up and communicate with each client.
- *window.c*, *io_functions.h*, *io_functions.c*, *interface.c*: Files containing the implementation of client functionality.
- *timeout.h*: A header defining functions for client timeout functionality to be used in Part 2 of this assignment.
- *timeout.c*: Implementation of timeout functionality.
- *scripts/*: A directory containing database test scripts. Each of these scripts is just a series of client commands in a file, one per line. Feel free to make your own in addition to those provided.
- *Makefile*: A Makefile for the project, which produces the executables *server* and *interface*. You will only ever need to directly run *server*. *interface* is called internally by the *server* support code.

- *database.pdf*: This handout.

You can install this by running

```
cs033_install database
```

The code contained in the handout produces a single-threaded, server-only version of the database. You will be modifying this to allow for multiple clients, a thread-safe database, and client cancellation and timeouts.

You will only need to modify *server.c*, *db.c*, and *Makefile* to complete this project. You will also have to refer to *window.h*, which contains function declarations used to create and interact with clients, and *timeout.h*, which contains function declarations for setting up timeouts. The remaining files contain implementation details that you do not need to worry about.

2.1 Client Interface

To begin with, user interaction with the database occurs only through the REPL of the *server* executable using a simple textual interface. When you implement the functionality for multiple clients, users will also interact with the database via `xterm` windows. These client windows will support only the client commands described below, while the *server* REPL also supports select server-only commands. You are provided with the windowing code for the clients and will not need to modify it (or even look at it) to complete the project. The client interface receives queries from its terminal and forwards them to the server process. The server processes each query and sends an appropriate response to the client. The database itself is represented as a binary search tree, which is not constrained to be balanced (it would be much more difficult to implement fine-grained locking on a balanced binary search tree).

Clients accept the following commands:

- `a <key> <value>`: Adds `<key>` into the database with value `<value>`.
- `q <key>`: Retrieves the value stored with key `<key>`.
- `d <key>`: Deletes the given key and its associated value from the database.
- `p <file>`: Prints the contents of the database to the specified file.
- `f <file>`: Executes the sequence of commands contained in the specified file.

When a client encounters an EOF in its input, it will terminate, causing the window to disappear. You can induce termination by typing CTRL-D on a blank line in the client window. Note that manually closing the client window in another manner, such as clicking the "X" button to close the window, will cause the server to print a non-fatal error message that the client terminated unexpectedly.

2.2 Testing Resources

In the *scripts* directory, you are provided with a pair of files called *names2013.txt* and *names1880.txt* that you can use to quickly initialize the database. This file contains a series of add commands to

insert babyname-frequency pairs into the database. In addition, the *scripts* directory contains two other files, *test1.txt* and *test2.txt*, that you can use to test your database's thread safety in part 2.

You can check that the tree output by the `p` command is correct with the following script:

```
cs033_db_check txtfile
```

where `txtfile` is the file containing the output of the `p` command.

To visualize the tree output by the `p` command, we have provided a script that creates a PNG image of the database. To use this script, enter the following command in a terminal:

```
cs033_db_vis txtfile pngfile
```

where `txtfile` is the file containing the output of the `p` command, and `pngfile` is the filename for the resulting PNG image.

3 Database

The database, implemented in *db.c*, consists of a collection of nodes organized in a binary search tree. Each node contains a pointer to a left and right child, either or both of which may be null pointers. The key associated to a given node is lexicographically greater than all nodes in its left subtree, and lexicographically less than all nodes in its right subtree. (In other words, an in-order traversal of the tree nodes yields a lexicographical ordering of the corresponding keys.)

The database supports the following functions:

3.1 `db_add()`

The `db_add()` function calls `search()` to determine if the given key is already in the database. If the key is not in the database, the function creates a new node with the given key and value and inserts this node into the database as a child of the parent node returned by `search()`.

3.2 `db_query()`

The `db_query()` function calls `search()` to retrieve the node associated with the given key. If such a node is found, the function retrieves the value stored in that node and returns it.

3.3 `db_remove()`

The `db_remove()` function calls `search()` to retrieve the node associated with the given key. If such a node is found, the function must delete it while preserving the tree ordering constraints. There are three cases that may occur, depending on the children of the node to be removed:

- Both children are `NULL`: In this case, the function simply deletes the node and sets the corresponding pointer of the parent node to `NULL`.

- One child is `NULL`: In this case, the function replaces the node with its non-`NULL` child.
- Neither child is `NULL`: In this case, the function finds the leftmost child of the node's right subtree and replaces the removed node with this one. Since the replacement node has no left child, it is easy to remove it from its current position, and since it is the leftmost child of its subtree it can occupy the position of the deleted node and satisfy the tree's ordering constraints.

3.4 `db_print()`

The `db_print()` function performs a pre-order traversal of the tree, printing to a given file a representation of each node followed by that node's left and right subtrees.

4 Assignment

This assignment is split up into two parts. We recommend finishing Part 1 within the week after the assignment is released. A completely functioning database will adhere to the following specifications:

- Part 1
  - Your server must be able to handle multiple clients, each with their own thread.
  - The database must implement fine-grained locking (see below for explanation).
  - Your server must be able to handle "s" (stop) and "g" (go) commands using a condition variable and mutex.
- Part 2
  - You must maintain a list of all client threads.
  - When the server receives an EOF from `stdin`, all clients should be immediately terminated (via cancellation), after which the program should exit cleanly. (Note: This overrides the termination behavior specified previously.)
  - When the server receives a SIGINT, all clients should be immediately terminated via cancellation, after which the server should continue its input loop.
  - Your server must be able to handle the "w" (wait) command using a condition variable and mutex.
  - Your program must take a timeout argument (in seconds) when run from the command line and cancel clients which do not receive any input for more than the number of seconds specified.

5 Part 1

5.1 Multithreading

Your first task is to modify the server code to handle multiple client windows. To accomplish this, modify *server.c* so that in addition to the commands it currently accepts from `stdin` it will

also accept newlines. (Think about how `fgets()` will behave when an empty line is entered.) Whenever the server receives a newline input it should create a new client window, along with a separate thread to handle it. The server itself should remain a functioning client, accepting all database manipulation commands, in addition to this added functionality. All clients should access the same, shared database. [1]

To implement the multithreaded version of the server, we suggest that you modify `client_new()` so that it spawns a new thread which executes `run_client()`. This way, the program's main function simply calls `client_new()` for each empty line of input received. You will need to modify the signature of `run_client()` so that you can pass it as an argument to `pthread_create()`.[2] You will also need to ensure that `client_delete()` is called at some point for each client, probably at the end of `run_client()`.

The server thread should stop accepting input on EOF. If there are client windows still open, these should be allowed to terminate naturally. The easiest way to do this is probably to detach each client thread. Be careful when you clean up at the end – the database should be deleted, but not before all clients are done with it. (Hint: One way to achieve this is to maintain a thread-safe counter of the number of active threads.)

To verify that you are managing threads correctly, your `main()` function must terminate with a call to `pthread_exit()` rather than calling `exit()` or executing a `return` statement. This will prevent the program from terminating until all threads have finished.

5.2 Thread Safety

Now that you have multiple threads, you must modify the database so that it is thread-safe. There are two principal ways to do this: apply *coarse-grained locking* and apply *fine-grained locking*. Both techniques are discussed below, but you must implement fine-grained locking to earn full credit for this assignment.

In addition to making the database thread-safe, you will add some additional features to the server to facilitate testing.

5.2.1 Coarse-Grained Locking

The simplest way to ensure thread safety is to put a read/write lock on the whole database. Each thread should obtain an appropriate type of lock before accessing the database. The `db_print()` function should also lock its output file using the `flock()` function declared in `<sys/file.h>`. For more information, see `man 2 flock`.

We recommend you implement and test coarse-grained locking first to get used to read/write locks before attempting the more difficult fine-grained locking scheme described below.

[1] Note that the database will not yet be thread-safe, so your program may behave incorrectly or crash if it receives input from more than one client at a time. You will fix this shortly.

[2] Casting function pointers should only be done when the functions have compatible argument and return types, as specified by the C standard.

5.2.2 Fine-Grained Locking

Coarse-grained locking is easy to implement, but it is not very efficient. For any modifications to occur, a single thread must obtain exclusive access to the entire database. This strategy ignores the fact that nodes in the tree have some level of independence. A more efficient design would use *fine-grained locking*. In this design, each node in the tree has its own read/write lock. These locks must be carefully managed to maintain database consistency while allowing multiple threads to access and modify the database simultaneously.

To implement fine-grained locking, you should modify the `search()` function in *db.c* to accept an additional parameter specifying read- or write-locking. This function should lock the root node and percolate down the tree, locking each node *before* releasing the lock on its parent. You will also have to modify the other database functions so that they handle locking appropriately. You must think carefully about the operations involved to avoid deadlocks and ensure that the database stays consistent.

5.2.3 Testing Features

To test thread-safety, you should add the following functionality to your server code. When the server encounters the line "`s`", all client threads should temporarily stop handling input. The line "`g`" should resume activity. This feature should allow you to test your database's thread safety by pausing activity, creating several clients, and then running them all at once. To implement this feature, you should use a "stopped" flag with a condition variable (and associated mutex).

The server accepts an optional script name argument, which can be one of those provided in the *scripts/* directory or one of your own creation. If you provide this argument when running the server, clients will execute the instructions in the given script file instead of accepting user input and terminate as soon as the commands in the script are completed. You can use this for testing in conjunction with the the "`s`" and "`g`" commands to test a large number of clients accessing the database concurrently.

6 Part 2

In the second part of this assignment, you will add timeouts, signal handling, and cancellation to your database. Specifically, you must update the database so it supports the following behavior:

- When the server process receives an EOF from `stdin`, all clients should be immediately terminated, after which the program should exit cleanly. (Note: This overrides the termination behavior specified in part 1.)
- When the server process receives a `SIGINT` signal, all existing clients should terminate. The program should continue to accept input and create new clients when requested.
- Client threads time out after waiting a given period of time without receiving input. On timeout, a given client window should close and the associated thread should terminate. The timeout period, in seconds, should be supplied as an argument to `main()`.
- Your server must be able to handle the "`w`" (wait) command to wait for all clients to terminate before continuing with the input loop.

To implement the first two features, you should maintain a list of all client threads. When a thread is created, it should add itself to the list. To terminate all current threads, we cancel all threads on the list. When a thread terminates (either from cancellation or naturally), it removes itself from the list. Note that you must be somewhat careful about the timing involved — if the main thread creates a new client, then reaches an EOF, it may issue a "cancel all" command before the new client has added itself to the list. We recommend using a barrier so that the main thread waits for a new client to add itself to the list before continuing.

When the server encounters the line "`w`", it should wait for all clients to terminate and then continue the server input loop. This feature should not be difficult to implement as it uses code nearly identical to code you have already written. Your server already waits for all clients to terminate before it exits (such as when the server receives an EOF).

6.1 Cancellation

To implement Part 2, you will need to use *cancellation*, a feature of the pthreads library that allows for semi-asynchronous thread termination. A thread may be marked for cancellation at any time using the `pthread_cancel()` function; however, it is not cancelled immediately. When a thread marked for cancellation reaches a *cancellation point*, one of a set of library functions specified by the POSIX standard, it is terminated. You can find a list of POSIX-specified cancellation points using `man 7 pthreads`. Of the support code functions, `get_command()` acts as a cancellation point and `send_response()` may act as a cancellation point in some cases. You cannot add your own cancellation points. It is safe to call `pthread_cancel()` on a thread any number of times until the thread terminates.

Each thread maintains a stack of *cleanup handlers* using the functions `pthread_cleanup_push()` and `pthread_cleanup_pop()`. When a cancelled thread reaches a cancellation point, the current cleanup handlers are executed in first-in-last-out order.

You will need to modify the client-handling threads to properly handle cancellation by installing appropriate cleanup handlers and modifying the cancel state as appropriate. For instance, the following block of code:

```
void foo(pthread_mutex_t my_mutex, pthread_cond_t my_cond) {
    pthread_mutex_lock(&my_mutex);

    while(!(some condition))
        // Cancellation point - the thread could stop executing here
        pthread_cond_wait(&my_cond, &my_mutex);

    // If the thread was cancelled in the pthread_cond_wait, this mutex would
    // never be unlocked
    pthread_mutex_unlock(&my_mutex);
}
```

could be replaced with:

```
// Cleanup handler called when thread is cancelled to unlock mutex so the thread
// doesn't quit while holding the lock
void cleanup_pthread_mutex_unlock(void *arg) {
    pthread_mutex_unlock((pthread_mutex_t *)arg);
}

void foo(pthread_mutex_t my_mutex, pthread_cond_t my_cond) {
    pthread_mutex_lock(&my_mutex);

    // Push the mutex unlock handler onto the stack of cleanup functions in case
    // the thread is cancelled while holding the lock
    pthread_cleanup_push(&cleanup_pthread_mutex_unlock, (void *)(&my_mutex));

    while(!(some condition))
        pthread_cond_wait(&my_cond, &my_mutex);  // Cancellation point

    // Pop and execute cleanup handler to release the lock regardless of whether
    // thread was cancelled
    pthread_cleanup_pop(1);
}
```

This way, the mutex will definitely be unlocked, even if the thread is cancelled while waiting.

Note that cancellation only occurs at most once per thread, so a cleanup handler may safely contain cancellation points. However, cancellation *can* occur while executing a cleanup handler during a call to `pthread_cleanup_pop()`.

Some of the database functions may contain cancellation points, such as the locking functions for read/write locks. Rather than worrying about a thread terminating while accessing the database, you should modify `process_command()` in *server.c* to disable cancellation while calling any database function.

Each thread has a *cancel state*, which sets whether or not a thread can be terminated at a cancellation point. The cancel state can be controlled with the `pthread_setcancelstate()` function.

You should also disable cancellation (or ensure it will not occur) when calling `window_new()` or `window_delete()`.

6.2 Signal Handling

Signals occur at the process level, making them somewhat difficult to handle in multi-threaded code. For this project, you will handle signals in a separate thread that runs alongside the server and client threads. To set this up, you should first mask off `SIGINT` for the entire process using `pthread_sigmask()`. Then, create a special signal monitoring thread that uses the `sigwait()` function to listen for `SIGINT` signals and cancel running clients.

6.3 Timeouts

We have provided support code for timeout functionality; all you must do is call the relevant functions at appropriate places in your code. These functions are declared in *timeout.h*. The basic design of timeouts is that each client thread is paired with a "watchdog" thread which repeatedly sleeps and then checks to see if a given deadline has expired. The client thread updates the deadline each time it receives input. If the client receives no input for too long, the deadline will expire and the watchdog will call a specified timeout function (in this case, probably a wrapper for `pthread_cancel()`).

You should create a new Timeout struct and watchdog thread for each client using `timeout_new()`. The timeout can then be started or reset with `timeout_activate()`, and stopped with `timeout_deactivate()`. When a thread terminates or is cancelled, it should delete its associated Timeout using `timeout_delete()`.

Note that clients should not terminate due to a timeout while they are stopped (with the "`s`" command), but if a client's timeout expires while it is stopped, it is perfectly acceptable for it to terminate immediately after it is resumed using the go command.

7 Demos

We have provided two demos for this project:

- `cs033_db_demo1`: This demo implements a basic multi-client, thread-safe database that satisfies Part 1 of the assignment. The command takes an optional argument of a script to run in client windows.
- `cs033_db_demo2`: This demo fully implements all parts of the assignment. The command requires an argument specifying the timeout duration, which can be followed by an optional script name argument.

8 Handin

To hand in your database implementation, run

```
cs033_handin database
```

from your project working directory. Make sure you include all C files, your Makefile, and a README. If you wish to change your handin, you can do so by re-running the handin script. Only your most recent handin will be graded.

9 Grading

This project will be graded according to how much of the functionality you implement:

- If you implement Part 1 with coarse-grained locking (one lock for the entire database), you can get up to a C.

- If you implement Part 1 with fine-grained locking, you can get up to a B.
- To get an A you must implement Part 1 with fine-grained locking, and also have at least a partially working implementation of Part 2 (cancellation, signals, and timeouts).