Project Database

Due: December 15, 2017 at 11:59pm

1 Introduction

Due to recent staffing changes at Monsters, Inc., Mike and Sulley decide that their employee database needs some major updates. Currently, the database is only accessible from the CEO's office, which makes updating the database extremely inconvenient. Mike and Sulley would like for the Monsters, Inc. administrative staff to work with the database in their offices but unfortunately, the database does not support multiple clients. Furthermore, even if it did, it was never written to be thread safe. The task to upgrade the company's database has fallen to you.

In this project you will be creating a simple server to manage an employee database of key value pairs over a network. Administrators should be able to search for employees in the database, add new entries, and remove existing entries.

2 Support Code

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

- *client.c:* A C file that produces the *client* program, including the client REPL and client-side networking.
- *server.c:* A C file containing the *server* program.
- *comm.h:* A header defining abstraction for server-side networking helper functions.
- *comm.c:* A C file that implements the server-side networking helper functions.
- *db.h:* A header containing function declarations for the database.
- *db.c:* A C file implementing database 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.
- *database.pdf:* This handout.

In addition, macOS may not implement some pthread functionality such as read/write locks and barriers. To accommodate this, we have attached the following files:

- *pthread_OSX.c:* A C file implementing missing OSX pthread functionality.
- *pthread_OSX.h:* A header file defining declarations and structs for OSX pthread functionality.

Note that you will need to write a Makefile for the project, which produces the executables *server* and *client*. You will only ever need to directly run *server* to start the server and run *client* to connect to the server.

You can install this project by running

```
cs0330_install database
```

2.1 Client Interface

The client program has been implemented for you. In the provided stencil, the client is written to connect to your server and interact with the database. The client supports client-only commands described below, while the server supports server-only commands. You can input queries to the client interface from either a script file or a REPL and the client will send them to the server. The server then processes each query and sends an appropriate response to the client.

The client can interact with the database using 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.
- **`f <file>`**: Executes the sequence of commands contained in the specified file.

A client will close its connection to the server and exit upon reaching the end of a script file, or on **`CTL+D`** if reading from the command line.

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. These files contain series of add commands to insert (name, frequency) pairs for babies born in the indicated years into the database. In addition, the *scripts* directory contains *test1.txt*, which you can use to test your database's thread safety in Part 2.

You can check tree output, using the **`p`** command (described below). You can then check whether this output is correct with the following script:

```
cs0330_db_check <txtfile>
```

where **`<txtfile>`** is the file containing the output of the **`p`** command. For example, if you input **`p tree.txt`** in the server REPL, **`cs033_db_check tree.txt`** would check that the database's tree is valid.

To visualize the tree output from 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:

```
cs0330_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 (which is not necessarily balanced). 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 the node's representation and then recursively printing its left and right subtrees. It will attempt to print to a file with the given filename, or stdout if none is provided. Be sure to remove trailing whitespace (e.g. newlines) from the filename!

3.5 `db_cleanup()`

The **`db_cleanup()`** function **`free`**s all dynamically-allocated nodes in the database. You should call this function before exiting on the server, and *only* when you are certain that no other threads are using or will use the database. You should use the variables in the **`server_control_t`** struct located near the top of *server.c* to ensure that all threads are terminated *before* you call **`db_cleanup`** in your main thread.

3.6 `interpret_command()`

The **`interpret_command()`** function gets called by the server to interpret the command, call database functions and store the response.

4 Assignment

You will be modifying the code to allow for multiple clients (multi-threading), thread-safety, signal handling and client cancellation. While the client-side code is provided for you, you will be implementing the server-side. You will only need to modify *server.c*, *db.c,* and possibly *db.h* to complete the code-component of this project.

This assignment is split up into four parts. We recommend starting this project early because you'll be applying a lot of different concepts.

- **Part 1: Networking**
    - All the client-side *networking* functions have been implemented for you, along with the entire client program, in `client.c`.
    - All the server-side *networking* functions have been implemented for you in `comm.c`. However, you are responsible for implementing the rest of the server program. Thus, you will be using the networking functions from `comm.c` to help set up your server in `server.c`.
    - As part of your assignment, you will submit written answers to questions (listed here) related to the networking portion of this project.
- **Part 2: Multiple Clients (Multi-threading)**

    - Your server must be able to handle multiple clients. Each connection from a client to the server should be serviced by a thread. You must maintain a list of these client connection threads.

- **Part 3: Thread Safety**
    - The database must implement fine-grained locking (see below for explanation).
    - Your server must be able to handle **"s"** (stop), **"g"** (go) commands using a condition variable and mutex, and **"p"** (print).
- **Part 4: Cancellation and Signal Handling**
    - When the server receives an **`EOF`** from **`stdin`**, all client connection threads should be immediately terminated (via cancellation), after which the server should exit cleanly. You must also ensure that no new **`client_t`** structs are added to the thread list after you call **`delete_all()`** from the main thread when your server REPL terminates. Allowing this would cause your server to terminate with a non-empty thread list which would result in a memory leak! When debugging this you may find it useful to **`assert`** that the list is empty before calling **`db_cleanup()`** (Note, when a client loses connection to the server, it should shut itself down).
    - When the server receives a **`SIGINT`**, all client connections should be immediately terminated via cancellation, after which the server should continue its input loop. Your implementation must allow users to spawn new clients after a **`SIGINT`** is delivered to the server. The server should behave exactly as it did before receiving a **`SIGINT`** (only now the database is potentially non-empty because of operations requested by previous clients).

5 Part 1: Networking

Your first task is to establish a TCP connection between a client and the server. Similarly to the Networking Lab, you will be implementing a server using sockets.[1]

To establish the connection, the client must find the server's internet address using **`getaddrinfo()`** and then set up a TCP connection with it. After finding the appropriate socket, the client creates an I/O stream to communicate with the server. Finally, the client uses **`fputs()`**, **`fgets()`** and **`fflush()`** to transmit and receive data.

[1] We are providing a fully functional *client.c*, but you are welcome to dissect and tamper with it. Keep in mind, however, that if you do, it will become more difficult for us to help you with bugs!

We provide networking support code in *comm.c*. You will use these functions to handle multiple client connections. All clients should access the same, shared database.[2]

There are also questions below about the purpose and implementation of the support functions for you to answer. Take a look at the demo server provided with the first Network Programming lecture for a refresher if you need one!

5.1 Questions

1. Consider the thread created in **`start_listener`**. How many threads with this functionality should be running on the server at any given time?
2. In the **`listener`** function, what do each of the following fields of the **`sockaddr_in`** struct represent**:** **`sin_family`**, **`sin_port`**, and **`sin_addr.s_addr`**?
3. What is the purpose of each of the following function calls in **`listener: socket`**, **`bind`**, **`listen`**, and **`accept`**? You should explain what each call does and what the effect would be if it were not called.
4. Which protocol (TCP or UDP) is used for communication? Why is this protocol used? (Hint: see line 36 in *comm.c*)
5. Describe what **`comm_serve`** does. How are the **`response`** and **`command`** parameters used? What would happen if the stream pointed to by **`cxstr`** were closed?
6. Describe, in detail, what happens during each iteration of the while loop in the **`listener`** function. Be sure to include explanations of all significant function calls.

6 Part 2: Multithreading

To implement the multithreaded version of the server, we suggest that you start with **`client_constructor()`** so that it spawns a new thread which executes **`run_client()`**. You will also need to ensure that **`client_destructor()`** is called at some point for each client, probably at the end of **`run_client()`**.

The server thread should stop accepting input on **`EOF`**. 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.)

7 Part 3: 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.

[2] 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.

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

7.1 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.

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.

7.2 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. Be sure to update **`db_print()`** because it does not use **`search()`**. You must think carefully about the operations involved to avoid deadlocks and ensure that the database stays consistent.

7.3 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).

When the server encounters **`“p”`**, it should call **`db_print()`** on the database. This will help you test by allowing you to see the state of the database. You may also provide a filename to **`db_print()`** if you want the output in a file instead of standard out.

8 Part 4: Cancellation and Signals

In the last part of this assignment, you will add 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.
- When the server process receives a **`SIGINT`** signal, all existing clients should terminate. The program should continue to accept input and create connections to new clients when requested.
- When working with this part of the assignment, it's important to ensure that your program is not leaking any memory support client thread data. You should ensure that the client thread data is cleaned up appropriately and that your thread list is completely empty before the server cleans up the database upon termination. To determine if a leak detected by valgrind is occurring due to this library behavior, check the trace of the reported leak; if the trace begins at **`pthread_cancel()`** or **`pthread_cancel_init()`**, and only references lines of code in library C files, then it's not a result of your code. You may also notice it if running a variable number of clients against your server produces the same roughly constant leak. This apparent leak is visible in the demo we have provided. If you are still unsure about a leak in your code, please talk to a TA or post on Piazza.

To implement these two features, you should maintain a list of all client connection 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 careful about the timing involved — if the listener thread creates a new client, then the main (REPL) thread reaches an **`EOF`**, it may issue a “cancel all" command before the new client connection has added itself to the list. To address this, you should have some thread-safe mechanism for the server to indicate that it is no longer accepting new clients, and each new client should ensure that the server is still accepting clients before it adds itself to the list of clients and begins to serve the client with which it's associated.

8.1 Cancellation

To implement Part 3, 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, **`comm_serve()`** acts

as a cancellation point. 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 that is executed as a result of explicit cancellation via **`pthread_cancel()`** may safely contain cancellation points. However, cancellation *can* occur while executing a cleanup handler that is explicitly called via **`pthread_cleanup_pop()`**.

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. Consider when it may be necessary to call this function in your implementation.

Note: The **`pthreads`** manual page states that **`pthread_rwlock_{wr,rd}lock()`** functions **may** be cancellation points. These functions **are not** implemented as cancellation points on Linux, therefore you may assume that they are not cancellation points in your implementation.

8.2 Signal Handling

Signals occur at the process level, making them somewhat difficult to handle in multithreaded 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.

9 Demos

We have provided a demo binary for this project: **`cs0330_db_demo_server`**. This is a fully functional DB server, with all parts of the assignment implemented, for you to get a better feel for the finished result. We have also provided a demo client: **`cs0330_db_demo_client`**. This models a fully functional client that can interact with the demo server.

10 Tips

As you have probably seen, debugging programs involving multiple processes can be much more confusing than debugging a single component on its own. However, GDB has several tools that may be able to come to the rescue!

A basic functionality of GDB in general is to pause program execution at a specified point and investigate the state of the system. The best way to do that here is to view the current threads with **`info threads`**. The current thread (GDB is able to watch the stack of a single thread of execution at a time) will have a star next it, and you can then switch to a thread with **`thread <thread number>`**. Each thread can give you a **`backtrace`**, and if you want to see them all at once, you can try something like **`thread apply all backtrace`**, which will give you the backtrace for all threads. This works for other commands as well!

If you are facing a deadlock, you will want to interrupt execution with **`CTRL+Z`** and dig your way through mutexes and so forth to identify which resources are causing the problem. You can

move through a trace with **`frame <frame number>`**. If you find that a particular thread is stuck in, say, **`pthread_mutex_lock()`**, you'll be able to see which mutex is being waited on, and which thread owns it. Then switch to that thread and see what it is waiting for, and so on. For more on debugging deadlocks, visit this helpful link.

Keep in mind other common debugging techniques like setting watchpoints, calling functions within gdb, setting conditional breakpoints, etc. and you'll have a much better time!

11 Handin

To hand in your database implementation, run

```
cs0330_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.

12 Grading

This project will be graded according to the following categories, ordered by weight:

- *Functionality*
    - Your client and server should communicate correctly.
    - You should cleanly start and close connections.
    - You should correctly implement fine-grained locking.
- *Correctness*
    - Your server and database should be able to correctly handle multiple clients at once.
    - Your database should be memory safe and should work.
- *Style*
    - Your code should be readable and commented and have a README.