Broadclub cuttlefish (Sepia latimanus)

CS0330 Gear Up:

Database

Project Overview

- Use and extend your knowledge of concurrency to implement a performing database server
  - Networking code is provided in *comm.c*; you must understand how it works and demonstrate your knowledge by answering the required questions. You are responsible for servicing the clients once the connection is established.
  - The database code is provided, but it is not thread-safe, so you will use your knowledge of locking to make it so.
- Further, your server will need to handle signals appropriately: `^C` will cancel all clients but allow the server to proceed.

Topics You Should Review

- Running and keeping track of multiple threads
- Fine-grained locking
- Signal handling using threads
- Thread cancellation
- Mutex locks and conditional variables
- `pthread_*` library functions
- Networking system calls

What You'll Learn

- How to create a multi-client database server
- How to manage thread safety and thread cleanup
- How to implement fine-grained locking of complex data structures
- How to use threads to manage signals in large processes

Roadmap: Getting Started

- Install the project using `cs033_install database`
- Source files:
  - `client.c`: We have given you the source for the client. Allows a REPL or the creation of several processes, each sending queries from a specified text file.
  - `comm.c`: We have given you the source code for the server-side networking functions. It is important you understand this code to answer the required questions and use it in the correct manner.
  - `server.c`: The server that will receive a request from a client, access the database to fulfill it, and forward the result back to the client.
  - `db.c`: Implements the database as a binary search tree. The file contains a fully-functional BST structure. You will need to add fine-grained locking to the data structure to make it thread-safe.

Roadmap: The Server-Client Connection

- Keep in mind that this project consists of two executables, which you should run in separate terminal windows.
  - The `server.c`, `db.c`, and `comm.c` files produce `server`, which sets up a server on the given port, with name given by the name of the machine.
  - The `client.c` file will produce `client`, which will look for a server specified by a name and a port number.
- When testing your program, run the server first, as the client will exit if it cannot find the server. You can do this over SSH or in the lab, but may have a harder time on your own computer. Ask for help if you are having trouble!

Roadmap: The Server-Client Connection (Cont'd)

- We recommend having your server run `start_listener()` from `comm.c` to start a thread that executes `listener()` which waits for connections.
- Once a connection is made, the server should spawn a new thread which executes `run_client()`.
- The server will get requests and respond to the client until it has finished and then clean up using `client_destructor()`.
- We recommend using `comm_serve()` from `comm.c` for getting commands and sending responses to the client.

Roadmap: Making Your DB Thread-Safe

- The file `db.c` contains an implementation of a Binary Search Tree that specifies how things in the database are stored
  - Key-value pairs organized into the tree by key
- You will need to employ your knowledge of read/write locks to add fine-grained locking and make the structure thread-safe.
- In general, you should:
  - Ensure that you lock the next node (child) *before* unlocking the previous node (parent)

Roadmap: Server REPL

- Implement the following commands on the server-side:
  - **'s'**: Should stop all client threads
  - **'g'**: Should resume all client threads
  - **'p'**: Should invoke `db_print()` to print out the database. You may optionally provide a file to `db_print()` as an output to write to. Otherwise, the database will be printed to the command line output.
  - Think about using a 'stopped' flag with an associated condition variable and mutex
- Test this by running multiple clients on a file of commands all at once using the 's' and 'g' commands, and make sure that the database is in a valid state when you stop

Roadmap: Thread Cancellation

- When should the server manage thread cancellation?
  - If the server receives an EOF (^D), it should cancel all the client threads and exit cleanly.
  - If your server receives SIGINT (^C), it should cancel all the client threads but continue to run.
    - But what happens to the many threads in the server process when SIGINT is caught?
    - One approach might be to mask off SIGINT for the entire process, then create a special signal monitoring thread that listens for SIGINT and cancels the running threads.

Sidebar: Thread Cancellation

- What happens when `pthread_cancel` is called on a thread?
  - Threads will not necessarily cancel immediately — they will continue operating until they reach a cancellation point.
    - You may assume that calls to lock/unlock mutexes and read-write locks are *not* cancellation points on Linux.
  - Push a cleanup handler using `pthread_cleanup_push()`.
    - Note that cleanup handlers are pushed to a LIFO stack (popped in opposite order of addition).
- `man 3 pthread_cancel` and `man 7 pthreads` will be very helpful!

GDB with Threads

- GDB is a lifesaver for debugging deadlocks!
  - `info threads`
  - `thread <threadno>`
  - As always, `backtrace`, `frame`, and `print` are your friends
- Remember that pressing `^Z` in GDB will suspend the process, allowing you to enter GDB commands

Demo

Server demo:

- `cs0330_db_demo_server <port number>`
- We've provided you with a fully functional demo of the server, as with Shell.
- First, run the server demo. Then run the client, using the hostname of your current computer (or `localhost`) in place of the server name.
- Example: `cs0330_db_demo_client cslab2g 1234 input.txt 6`

Testing

- We provide two files, `names2013.txt` and `names1880.txt`, to initialize the database. These files will add name-frequency pairs to your database.
- There are also files to help you test thread safety: `test1.txt`
- If you use the ‘p’ command to print your database to an output file using `p <output_file>` , you can then use `cs0330_db_check <output_file>` to check that the state of your tree is consistent.
- You can visualize the database in the output file using `cs0330_db_vis <output file> <png file>`.

Tips

- Understand *comm.c* before beginning *server.c*.
  - We recommend completing the required questions before beginning the rest of the project. You will need to utilize the functions in *comm.c* when creating the server.
- When implementing fine-grained locking make sure you lock and unlock in a consistent order.
  - Always lock the child *before* unlocking the parent, otherwise another thread may get through and access memory in the subtree, which may corrupt the data.
- Use the visualizers to test your code! Just because you are able to exit cleanly does not mean that everything is working correctly.

More Tips

- When you create a mutex, ask yourself: *"if this thread were cancelled while holding this mutex, what would happen?"*
  - If the answer is *"It would stay locked forever and everything would be on fire,"* you should create and push cleanup handlers to ensure that the mutex will get unlocked even if the thread cancels.
  - To pop a cleanup handler after you are done with the mutex, call `pthread_cleanup_pop()`. If you pass in `1` to the "execute" parameter, your cleanup code will also be executed. Neat!
  - If any of this doesn't make perfect sense, read the man pages!

One Last Tip

- Complete each part of the project sequentially, and make sure to test thoroughly after finishing each part.
  - It is much harder to debug after writing all of parts 1, 2, 3, and 4, rather than incrementally testing as you go.

Questions? We’re Here to Help!