Homework 5 - CSE 320 - Fall 2023

Professor Eugene Stark

Due Date: Friday 12/08/2023 @ 11:59pm

Introduction

The goal of this assignment is to become familiar with low-level POSIX threads, multi-threading safety, concurrency guarantees, and networking. The overall objective is to implement a simple multi-threaded transactional object store. As you will probably find this somewhat difficult, to grease the way we have provided you with a design for the server, as well as binary object files for almost all the modules. This means that you can build a functioning server without initially facing too much complexity. In each step of the assignment, you will replace one of our binary modules with one built from your own source code. If you succeed in replacing all of our modules, you will have completed your own version of the server.

It is probably best if you work on the modules in roughly the order indicated below. Turn in as many modules as you have been able to finish and have confidence in. Don't submit incomplete modules or modules that don't function at some level, as these will negatively impact the ability of the code to be compiled or to pass tests.

Takeaways

After completing this homework, you should:

- Have a basic understanding of socket programming
- Understand thread execution, locks, and semaphores
- Have an advanced understanding of POSIX threads
- Have some insight into the design of concurrent data structures
- Have enhanced your C programming abilities

Hints and Tips

- We strongly recommend you check the return codes of all system calls. This will help you catch errors.
- **BEAT UP YOUR OWN CODE!** Throw lots of concurrent calls at your data structure libraries to ensure safety.
- Your code should **NEVER** crash. We will be deducting points for each time your program crashes during grading. Make sure your code handles invalid usage gracefully.
- You should make use of the macros in `debug.h`. You would never expect a library to print arbitrary statements as it could interfere with the program

using the library. **FOLLOW THIS CONVENTION!** `make debug` is your friend.

:scream: **DO NOT** modify any of the header files provided to you in the base code. These have to remain unmodified so that the modules can interoperate correctly. We will replace these header files with the original versions during grading. You are of course welcome to create your own header files that contain anything you wish.

:nerd: When writing your program, try to comment as much as possible and stay consistent with your formatting.

Helpful Resources

Textbook Readings

You should make sure that you understand the material covered in chapters **11.4** and **12** of **Computer Systems: A Programmer's Perspective (3rd Edition)** before starting this assignment. These chapters cover networking and concurrency in great detail and will be an invaluable resource for this assignment.

pthread Man Pages

The pthread man pages can be easily accessed through your terminal. However, this opengroup.org site provides a list of all the available functions. The same list is also available for semaphores.

Getting Started

Fetch and merge the base code for `hw5` as described in `hw0`. You can find it at this link: https://gitlab02.cs.stonybrook.edu/cse320/hw5. Remember to use the `--stategy-option=theirs` flag for the `git merge` command to avoid merge conflicts in the Gitlab CI file.

The Xacto Server and Protocol: Overview

The "Xacto" server implements a simple transactional object store, designed for use in a network environment. For the purposes of this assignment, an object store is essentially a hash map that maps **keys** to **values**, where both keys and values can be arbitrary data. There are two basic operations on the store: `PUT` and `GET`. As you might expect, the `PUT` operation associates a key with a value, and the `GET` operation retrieves the value associated with a key. Although values may be null, keys may not. The null value is used as the default value associated with a key if no `PUT` has been performed on that key.

A client wishing to make use of the object store first makes a network connection to the server. Upon connection, a **transaction** is created by the server. All op-

erations that the client performs during this session will execute in the scope of that transaction. The client issues a series of `PUT` and `GET` requests to the server, receiving a response from each one. Upon completing the series of requests, the client issues a `COMMIT` request. If all is well, the transaction **commits** and its effects on the store are made permanent. Due to interference of transactions being executed concurrently by other clients, however, it is possible for a transaction to **abort** rather than committing. The effects of an aborted transaction are erased from the store and it is as if the transaction had never happened. A transaction may abort at any time up until it has successfully committed; in particular the response from any `PUT` or `GET` request might indicate that the transaction has aborted. When a transaction has aborted, the server closes the client connection. The client may then open a new connection and retry the operations in the context of a new transaction.

The Xacto server architecture is that of a multi-threaded network server. When the server is started, a **master** thread sets up a socket on which to listen for connections from clients. When a connection is accepted, a **client service thread** is started to handle requests sent by the client over that connection. The client service thread executes a service loop in which it repeatedly receives a **request packet** sent by the client, performs the request, and sends a **reply** packet that indicates the result. Besides request and reply packets, there are also **data** packets that are used to send key and value data between the client and server.

> :nerd: One of the basic tenets of network programming is that a network connection can be broken at any time and the parties using such a connection must be able to handle this situation. In the present context, the client's connection to the Xacto server may be broken at any time, either as a result of explicit action by the client or for other reasons. When disconnection of the client is noticed by the client service thread, the client's transaction is aborted and the client service thread terminates.

The Base Code

Here is the structure of the base code:

```
.
   .gitlab-ci.yml
   hw5
       include
           client_registry.h
           data.h
           debug.h
           protocol.h
           server.h
           store.h
```

```
    transaction.h
lib
    xacto.a
    xacto_debug.a
Makefile
src
    main.c
tests
    xacto_tests.c
util
     client
```

The base code consists of header files that define module interfaces, a library `xacto.a` containing binary object code for our implementations of the modules, and a source code file `main.c` that contains containing a stub for function `main()`. The `Makefile` is designed to compile any existing source code files and then link them against the provided library. The result is that any modules for which you provide source code will be included in the final executable, but modules for which no source code is provided will be pulled in from the library.

The `util` directory contains an executable for a simple test client. When you run this program it will print a help message that summarizes the commands. The client program understands command-line options `-h <hostname>`, `-p <port>`, and `-q`. The `-p` option is required; the others are optional. The `-q` flag tells the client to run without prompting; this is useful if you want to run the client with the standard input redirected from a file.

Task I: Server Initialization

When the base code is compiled and run, it will print out a message saying that the server will not function until `main()` is implemented. This is your first task. The `main()` function will need to do the following things:

- Obtain the port number to be used by the server from the command-line arguments. The port number is to be supplied by the required option `-p <port>`.
- Install a `SIGHUP` handler so that clean termination of the server can be achieved by sending it a `SIGHUP`. Note that you need to use `sigaction()` rather than `signal()`, as the behavior of the latter is not well-defined in a multithreaded context.
- Set up the server socket and enter a loop to accept connections on this socket. For each connection, a thread should be started to run function `xacto_client_service()`.

These things should be relatively straightforward to accomplish, given the information presented in class and in the textbook. If you do them properly, the server should function and accept connections on the specified port, and you

should be able to connect to the server using the test client. Note that if you build the server using `make debug`, then the binaries we have supplied will produce a fairly extensive debugging trace of what they are doing. This, together with the specifications in this document and in the header files, will likely be invaluable to you in understanding the desired behavior of the various modules.

Task II: Send and Receive Functions

The header file `include/protocol.h` defines the format of the packets used in the Xacto network protocol. The concept of a protocol is an important one to understand. A protocol creates a standard for communication so that any program implementing a protocol will be able to connect and operate with any other program implementing the same protocol. Any client should work with any server if they both implement the same protocol correctly. In the Xacto protocol, clients and servers exchange **packets** with each other. Each packet has two parts: a fixed-size **header** that describes the packet, and an optional **payload** that can carry arbitrary data. The fixed-size header of a packet always has the same size and format, which is given by the `XACTO_PACKET` structure; however the payload can be of arbitrary size. One of the fields in the header tells how long the payload is.

- The function `proto_send_packet()` is used to send a packet over a network connection. The `fd` argument is the file descriptor of a socket over which the packet is to be sent. The `pkt` argument is a pointer to the packet header. The `data` argument is a pointer to the data payload, if there is one, otherwise it is `NULL`. The `proto_send_packet` assumes that multi-byte fields in the packet passed to it are in **network byte order**. This means that when multibyte fields in the packet header are set, the values being stored in them must be converted from the normal host byte order to network byte order. The `write()` system call is then used to write the header to the "wire" (i.e. the network connection). If the length field of the header specifies a nonzero payload length, then an additional `write()` call is used to write the payload data to the wire.

  :nerd: Normally, multi-byte values are stored in memory according to **host byte order**, which varies between architectures. For example, Intel-based platforms use "little-endian" byte order, which stores the least-significant byte of a multibyte quantity in the lowest-numbered address, and MIPS-based platforms use "big-endian" byte order, which is the opposite. In order for hosts with different "endianness" to be able to understand each other, multi-byte quantities in fields of packets sent over the network are converted from host byte order to a standard network byte order before sending, and they are converted back to host byte order upon reception. Conversion between host and network byte order can be done using, e.g., the `htonl()` and related functions described in the Linux man pages.

- The function `proto_recv_packet()` reverses the procedure in order to receive a packet. It first uses the `read()` system call to read a fixed-size packet header from the wire. If the length field of the header is nonzero then an additional `read()` is used to read the payload from the wire. The header and payload are stored using pointers supplied by the caller. Multi-byte values in the packet header returned by the `proto_recv_packet()` are stored in network byte order.

NOTE: Remember that it is always possible for `read()` and `write()` to read or write fewer bytes than requested. You must check for and handle these "short count" situations.

Implement these functions in a file `protocol.c`. If you do it correctly, the server should function as before.

Task III: Client Registry

You probably noticed the initialization of the `client_registry` variable in `main()` and the use of the `creg_wait_for_empty()` function in `terminate()`. The client registry provides a way of keeping track of the number of client connections that currently exist, and to allow a "master" thread to forcibly shut down all of the connections and to await the termination of all server threads before finally terminating itself.

The functions provided by a client registry are specified in the `client_registry.h` header file. Provide implementations for these functions in a file `src/client_registry.c`. Note that these functions need to be thread-safe, so synchronization will be required. Use a mutex to protect access to the thread counter data. Use a semaphore to perform the required blocking in the `creg_wait_for_empty()` function. To shut down a client connection, use the `shutdown()` function described in Section 2 of the Linux manual pages. You may assume that any file descriptors managed by the client registry have a value less than the constant `FD_SETSIZE` defined in the header file `<sys/select.h>` (see the man page for `select(2)`).

Implementing the client registry should be a fairly easy warm-up exercise in concurrent programming. If you do it correctly, the Xacto server should still shut down cleanly in response to SIGHUP using your version.

Note: You should test your client registry separately from the server. Create test threads that rapidly call `creg_register()` and `creg_unregister()` methods concurrently and then check that a call to the `creg_wait_for_empty()` function blocks until the number of registered clients reaches zero, and then returns.

Task IV: Data Module

The file `data.h` describes **blobs**, which represent reference-counted chunks of arbitrary data, **keys**, which are blobs packaged together with a hash code so that they can be used as keys in a hash table, and **versions**, which are used to represent versions of data in the transactional store.

A possibly unfamiliar (but essential) concept here is the notion of a **reference count**. The basic idea of reference counting is for an object to maintain a field that counts the total number of pointers to it that exist. Each time a pointer is created (or copied) the reference count is increased. Each time a pointer is discarded, the reference count is decreased. When the reference count reaches zero, there are no outstanding pointers to the object and it can be freed. The reason we need to use reference counting for various objects is because pointers to those objects can be stored in other objects, where they can persist long after the original context in which they were created has terminated. Eventually, we need to be able to determine when the last pointer to an object has been discarded, so that we can free the object and not leak storage.

In a reference counting scheme, the object to which reference counting is to be applied has to provide two functions: one for increasing the reference count and another for decreasing it. The function for decreasing the reference count has the responsibility of freeing the object when the reference count reaches zero. In addition, the code that uses a reference-counted object has to be carefully written so that whenever a pointer is created the method to increase the reference count is called, and whenever a pointer is discarded the method to decrease the reference count is called.

When specifying and implementing functions that work on reference-counted objects, it is important to have a clear idea of exactly when reference counts need to be increased, decreased, or left alone. If a function that receives a reference-counted object passed as a parameter needs to save a pointer to that object somewhere, in general it will need to increase the reference count to account for the additional pointer. The caller is then free to continue to use its own pointer to the object after the function has been called.

On the other hand, in some cases, the caller does not need to use its pointer again once it has been passed to a function. We can then think of the caller transferring "ownership" of its pointer to the called function, which "consumes" the reference it is passed. If a caller's reference is consumed by a function, then the caller must not use that pointer again. Judicious use of the concept of consuming a reference can reduce the number of situations in which a reference count is increased, only to be immediately decreased. However, it is important that the specification of a function clearly indicate whether the function is inheriting a reference or whether it must increase the reference count if it wants to retain a copy of that pointer.

A related piece of terminology we will use in conjunction with non-reference-

counted objects is the concept of a function "inheriting" a pointer from its caller. In this case, the caller transfers to the called function the responsibility for ultimately freeing the pointer. When a called function inherits a pointer from its caller, the caller must not use the pointer again after the call.

With the above information in mind, the implementation of this module should be relatively straightforward. Note that, in a multi-threaded setting, reference counts are shared between threads and therefore need to be protected by mutexes if they are to work reliably.

Task V: Client Service Thread

Next, you should implement the thread function that performs service for a client. This function is called `xacto_client_service`, and you should implement it in the `src/server.c` file.

The `xacto_client_service` function is invoked as the thread function for a thread that is created to service a client connection. The argument is a pointer to the integer file descriptor to be used to communicate with the client. Once this file descriptor has been retrieved, the storage it occupied needs to be freed. The thread must then become detached, so that it does not have to be explicitly reaped, and it must register the client file descriptor with the client registry. Next, a transaction should be created to be used as the context for carrying out client requests. Finally, the thread should enter a service loop in which it repeatedly receives a request packet sent by the client, carries out the request, and sends a reply packet, followed (in the case of a reply to a `GET` request) by a data packet that contains the result. If as a result of carrying out any of the requests, the transaction commits or aborts, then (after sending the required reply to the current request) the service loop should end and the client service thread terminate, closing the client connection.

Task VI: Transaction Manager

Probably the next easiest module to implement is the transaction manager, which is responsible for creating, committing, and aborting transactions. The various functions that have to be implemented are specified in `transaction.h` header file.

For transactions, the function `trans_ref()` is used to increase the reference count on a transaction and the function `trans_unref()` is used to decrease the reference count. The reference count itself is maintained in the `refcnt` field of the `transaction` structure. Because it is accessed concurrently, it needs to be protected by a mutex. When transaction is created by the client service thread, it initially has a reference count of one. The transaction gets passed to various other functions, including operations on the store. If the store needs to save a reference to a transaction (*e.g.* to record the transaction as the "creator" of a version of some data), then it will increase the reference count before doing so

to account for the saved pointer. Ultimately, the transaction will be committed via a call to `trans_commit` or aborted via a call to `trans_abort`. These two calls consume a reference to the transaction (*i.e.* they decrement the reference count), so the caller should not continue to use the transaction subsequently unless an additional reference was previously created.

A transaction is only ever committed by a call to `trans_commit` made by the client service thread in response to a `COMMIT` request received from the client. Once `trans_commit` has been called, it could be that the last reference to the transaction has been consumed and the transaction has been freed, so the transaction object should not referenced again after that (unless another reference has previously been obtained). Calls to operations on the store may cause a transaction to abort, but in that case they arrange to increase the reference count before doing so, so that the net value of the reference count will not be decreased and client service thread will be able to check the status of a transaction after performing a store operation to see whether or not the transaction has aborted.

One other complication in the implementation of transactions is the notion of "dependency". As a result of performing an operation on the store, a transaction may become **dependent** on one or more other transactions. Before a transaction may commit, it must wait for all the transactions on which it depends (*i.e.* the transactions in its **dependency set**) to commit. If any of the transactions in the dependency set aborts, then the transaction itself must abort. In order to manage dependencies, the transaction manager module provides a function `trans_add_dependency`, which is used to add one transaction (the "dependee") to the dependency set of another (the "dependent"). Adding a dependency is done by allocating a "dependency" structure, storing a reference to the dependee transaction in that structure, and adding the structure to the linked list of dependencies for the dependent transaction, after checking to make sure that there is not already an entry for the same dependee in the dependency list. The reference count of the dependee transaction must increased to account for the stored pointer.

When a transaction attempts to commit, it must first call `sem_wait` on the semaphore in the `sem` field of each of the transactions in its dependency set. This will block the transaction until the transaction on which it depends has committed and done `sem_post` on that semaphore. Since a transaction can exist in the dependency set of more than one other transaction, in order to know how many times to call `sem_post` it has to rely on the transaction calling `sem_wait` to have first incremented the `waitcnt` field on the transaction for which it is waiting, so that `waitcnt` always correctly reflects the number of transactions waiting on `sem_wait`. Of course, the `waitcnt` field of a transaction must only be accessed while the mutex lock on that transaction is held.

Once `sem_wait` has been called on all transactions in the dependency set, a transaction can check the status of these transactions to see if any have aborted. If so, then committing is not possible, and the dependent transaction must abort

instead. If all of the dependee transactions have committed, then the dependent transaction can move to the "committed" state.

Task VII: Transactional Store

If you have made it this far, then you can try to implement the transactional store module itself. This is probably the most technical of the various modules, so you probably need to have a good understanding of what is going on before you attempt the implementation. The functions for the store are specified in the `store.h` header file, where some further details of how the store works are given. Besides the fairly straightforward initialization and finalization functions, the store provides functions `store_put` and `store_get`, which in the end are rather similar to each other. Each function uses the hash code of the key passed as argument to identify a particular "bucket" in the hash table. The bucket is then searched to find a map entry whose key is equal to the given key. If an entry is not found, one is created and inserted into the proper bucket.

Once the map entry for the given key has been located (or created), then both `store_put` and `store_get` work by inserting a new "version" into the list of versions contained in that map entry. The concept of a version, and the rules by which version lists are maintained, are detailed in `store.h`. As a result of these rules, it is possible for a transaction calling `store_put` or `store_get` to either abort or become dependent on other transactions. The implementations of these functions have to arrange for these actions to be carried out when necessary.

Note that the only time versions are removed from the transactional store is during the "garbage collection" phase carried out at the beginning of a `store_put` or `store_get` operation. Upon completion of an operation, there will in general be "extra" versions in the store that will get cleaned up on the next call to `store_put` or `store_get`. You will want to be careful to manage the reference counts on the various objects so that you don't end up with any blobs or transactions that hang around forever, even when they are not reachable from anything in the store, and that you don't attempt to decrease any reference count below zero. If you do this all correctly, then when the server terminates (as a result of `SIGHUP`), all objects will be cleanly freed.

Submission Instructions

Make sure your hw5 directory looks similarly to the way it did initially and that your homework compiles (be sure to try compiling both with and without "debug"). Note that you should omit any source files for modules that you did not complete, and that you might have some source and header files in addition to those shown. You are also, of course, encouraged to create Criterion tests for your code. The test file we have provided contains some code to start a server and attempt to connect to it. It will probably be useful while you are working on `main.c`.

It would definitely be a good idea to use `valgrind` to check your program for memory and file descriptor leaks. Keeping track of allocated objects and making sure to free them is one of the more challenging aspects of this assignment.

To submit, run `git submit hw5`.