Software Design Document

CSE 130, Fall 2020

Multithreaded server

Authors:

- Scott Melero
- Korbie Sevilla

Sources:

- Linux man pages
- How to use getOpt()
- How to use unordered maps
- Multi-threading servers in C
- Queue linked list implementation in C

Program Structure

- We are building off of our initial server design, which can be found here. This document will not re-explain the basic server structure from asgn1, but will go into detail about all of the new features we implemented to support multi-threading and redundancy.

Data Design

- Thread pool array to store our team of worker threads.
- Mutex pool for each file in our servers directory.
- Int lock count to keep track of how many locks we are creating.
- Pthreads
    - Mutex locks to ensure thread safety, and to protect the critical region
    - Condition Variable to avoid busy waiting and sleep our worker threads while there is no work in the queue.

- Queue linked list to store and sequence the incoming client connections.

Critical Region

- The queue data structure is the program's critical region, and all of the threads have access to it. Before we add or remove elements from the queue, we lock the global queue lock, and unlock once the operation is done.

Pseudocode Outline

- Multi-threading:
    - We don’t want to overload our system with too many threads, so we need to specify how many threads will be created via the command line.
        - Define a new thread pool variable and set that equal to the number given at the command line. We use getopt() to parse the command line arguments.
    - In a loop, create N threads, using pthread_create(), and put them into a buffer of type pthread_t and of size N (pthread_t thread[N]).
        - Each pthread will take in a thread function, which starts the request processing.
    - We want to reuse these threads instead of killing them as they finish processing a request. To do this, we need to create a queue (first in first out). When we receive work for a thread to do (a client request), we call enqueue() to add that connection to the queue.
    - While the head of the queue is not NULL, that means that there is still work to do. The worker threads will constantly be grabbing work off of the queue, until it is empty. Requests will sit in the queue until there is an open thread to process them.
    - The queue data structure is a shared data structure between all of the threads. They modify it whenever we enqueue() or dequeue() a request. To ensure thread safety, we use mutex locks to lock each thread before it adds or removes work from the queue, and unlock it afterwards so that other threads can access the critical region.
    - We also do not want out threads to stand by and constantly check if there is more work on the queue to be done. Doing so will burn up all of our available CPU cycles. To avoid what is known as “busy waiting”, we use conditional variables to put our worker threads to sleep when they are not in use. In main, right we enqueue() a new client request, we need to use pthread_cond_signal in order to “wake up the worker thread”. Before we

are able to dequeue() a request, we need to use pthread_cond_wait to wait for the signal in main.

- Redundancy:
    - If the server is run with the redundancy flag, then we need to operate on 3 separate copies of the specified file. We used getopt to parse the command line arguments and check for the -r flag.
    - PUT with redundancy:
        - In the execute_PUT() function, we initialize an int (lets call it x) that will let us know how many copies of the file to open. If redundancy is selected, x=3, else x=1.
        - With redundancy selected, we need to open the file into 2 folder, copy1, copy2, & copy3. We created a new char[] = "./copy%d/%s" where %d is the folder index and %s is the filename sent by the client.
        - Initialize a new file descriptor array and In a loop, from i = 1 to x, call open on the filename into fd[i].
            - If redundancy, append the file name with the folder path char that we defined. In this loop we need to open the path "./copy[i]/filename" into fd[i].
            - If there is no redundancy, we will just open the file into the servers home directory.
        - To keep track of errors, after each iteration of the open loop we need to check if fd[i] == -1. If it does, we increment our error counter. We also need to check what type of error we got and record the number of them that occurred during the opening process.
        - If a majority of the opens resulted in a -1, then we look for the most common error reported, and return that status message to the client. If all 3 return codes are different, then that is an internal server error.
        - If a majority of the opens were successful, then we need to write() the reshe request body. In a loop, keep recv()'ing from the client until we have no more body. In an inner loop, from i = 1 to x, write the body continents into fd[i].
    - GET with redundancy:
        - This process is the exact same as the put request, up until we need to operate on the file descriptors.

        - If a majority of the open()'s were successful, then we need to check if at least 2 of the files are identical in content. To do this, we are going to test 2 files at a time (file i & file j). Until read returns 0 for both files, we will read their contents into a couple of temporary buffers, and compare the 2 using strncmp(). If strncmp() does not return a 0, then we will break from the loop and compare the next set up files. The file comparison order is as follows
            - Compare "copy1/…" & "copy2/….". Then compare "copy1/…" & "copy3/….". then compare "copy2/…" & "copy3/….".
        - If we complete this revc() loop, then that means that the files i & j are equal in content, and we can return one of the 2. In this case, we will write fd[i] back to the client window.
        - If all of the files are not equal, then we will return a 500.
        - Of course if we do not have redundancy checked, then we will only try to GET the one file that should be in the servers home directory, no need to run the comparison.

System Components

- Void execute_GET():
    - This function will perform the GET request for the server. Based on the command/operation, the appropriate struct will be filled with the correct variables and call the proper status code.
    - Furthermore, we have implemented additional functionality. This function also handles the opening of at most 3 files and will check/record any errors discovered. For instance, if a majority of the files fail to open the proper status will be called. Otherwise, if at least 2 files are equal a copy of one of the correct files will be returned.

- Void make_locks()
    - This function does not take in any arguments and will run before we start listening for client connections.
    - Creates a new mutex lock for each file in the directory. If redundancy is enabled, then it will run on each file in the copy folders. If not, it will run on each file in the servers home directory.

- Myqueue.cpp:
    - Myqueue is a linked list that operates as our queue and contains 2 functions.

        - enqueue();
            - This function takes in a client socket and will add the new incoming request to the tail of the queue.
        - dequeue():
            - This function does not take in any arguments, but will return the head of the list. If the head is not NULL, that means there is work for a thread to do. If it is NULL, there is no more work and all of the threads can go to “sleep”.

- Void execute_PUT():
    - This function will perform the PUT request for the server. The data sent by a client will either be created or truncated into a new file. Similar to GET, the appropriate struct will be filled with the correct variables and call the proper status code.
    - This function also handles the opening of at most 3 files. Here, each file will be checked and scanned for any errors. If an error is determined there will be a call sent to the appropriate statue code. Otherwise, the opening of the files will continue.

- Void * handle_request():
    - This function will take in a pointer to the client socket and proceed to call the following functions: read_http_response, process_request, construct_http_response. The purpose of these function calls is to parse the HTTP message, process the necessary request, and then construct the client response. Once these calls are made the client socket will be closed.

- Void * thread_function():
    - The purpose of this function is to maintain each worker thread(Either specified at the command line or defaulted to 4) with the necessary “work”. The function will first take in void arg pointers, and determine the correct amount of threads specified. A mutex lock is called to allow multiple program threads to share the same resources, such as file access. Although, this does not occur simultaneously. The mutex lock is also responsible for locking the thread before the critical region is entered. Here, each thread is managed through a queue structure to maintain the order of the “thread in line” with the “task” to be accomplished. Once a

thread is assigned a task, the next thread will await for a signal from the current thread to begin work and will be unlocked.

Description of the User Interface

- Server terminal:
    - A user can launch the server by running the following command via a terminal window: *./httpserver address port# -r -N #*. The address must be specified, and an error will be returned if the given arg is not valid.
    - The port number is optional. If it is not defined, the server will default to port 80. You need to run the code as a super user in order to listen on port 80 (sudo ./httpserver address port#).
    - The -r and -N flags are optional.
        - If -r is given, then the server will run in “redundancy mode”. This means that we will operate on the specified file 3 times, each in a separate folder. If -r is not given, then the server will only operate on 1 copy of the specified file in the servers home directory.
        - -N is followed by an int which specifies the number of workers threads to create. If -N is not given, the server will create 4 threads by default.
    - The command “./httpserver localhost 8080 -r -N 4” will create a localhost server listening on port 8080, running in redundancy mode, and with 4 worker threads to handle incoming connections.

- Client Terminal.
    - Once the server is online, a client will then be able to GET files from the server, and PUT files on it via a separate terminal window. All files names **MUST** be 10 character ascii strings containing only the char’s [0-9] & [a/A - b/B]. To perform an operation on the server, a client can run the following cUrl commands:
        - *curl -T file http://address:port#/filename*, to PUT a file onto the server directory.
        - *curl http://address:port#/filename*, to GET a file that is in the server directory.
    - The server will send back a response to the client terminal containing a status code and content length.

Testing

Ensuring Thread Safety

- The Queue:
    - To ensure thread safety when updating the queue, we used a mutex lock from the pthread library. Before we add or remove elements from the queue, we lock the global queue lock, and unlock once the operation is done.
- Files:
    - In order to make sure that 2 threads are not operating on the same file at the same time, we need to initialize a new lock for each file in the directory and lock it before it is operated on.
    - To do this, we run a function (make_locks) before we listen for client connections, which will find the files in the servers directory and each sub directory, and will assign a new mutex lock to it. This data is stored in an unordered map lock_map.
    - When we execute a PUT or GET, we need to lock the corresponding filename's lock. If we are processing a PUT request, and the file does not exist, we need to lock the global newfile_lock, so that we can safely create a new lock for filename. We add this new lock to the file lock array, lock it, and then unlock the global newfile_lock.

Testing Issues

- NO issues at this time.

Classes of tests

- Test the following scenarios by creating shell scripts that run multiple curl commands in parallel:
    - GET file that doesn't exist
    - GET file with no permissions
    - GET file with invalid name
    - GET file with invalid name that doesn't exist
    - GET file with 2/3 files giving no permissions
    - GET file with 3/3 files giving no permissions

    - GET file with 1/3 files giving no permissions, 1/3 files that don't exist, 1/3 files that do exist and we have permission
    - GET file with 1/3 files giving no permissions, 2/3 files that don't exist
    - GET file with 2/3 files giving no permissions, 1/3 files that don't exist
    - GET file missing a folder (copy1 for example)
    - PUT file with 1/3 folders giving no permission
    - PUT file with 2/3 folders giving no permission
    - PUT file missing a folder
    - PUT file with invalid name
    - PUT file with incomplete body (closed early) (should files be created at all or should we first buffer the body to ensure we receive the entire thing?)
    - PUT file with 2/3 files being created, but 1 error (should the other files be deleted?)
    - PUT file with 1/3 files being created, but 2 errors
    - Send 2-4 requests of the same resource at the same time using 1-4 worker threads.
    - Send 2-4 requests of different resources at the same time using 1-4 worker threads.
    - Send multiple requests in the same connection

Expected software response

- Multiple threads:
    - The server is able execute multiple client requests simultaneously
- Redundancy:
    - GET:
        - If 3/3 of the opens returned the same result, and they all have the same content (errors or file content), return any one of the 3. Works as expected
        - If 2/3 of the opens returned the same value, and those 2 have the same content (errors or file content), return one of the 2. Works as expected
        - If all of the open are different(errors or file content), return a status code of 500. Works as expected

Performance bounds

- On top of the initial parsing of simple HTTP headers and management of client requests via PUT and GET, the server can now handle multi-threading to handle multiple and requests and clients at a time and redundancy to maintain having three copies of a file instead of one, as stated in the assignment spec.
- The server performance is limited by the hardware it is running on. We could very well initialize 1000 threads, it would not run very well of the laptops we are using. The ideal number of threads at any one time is 4-6.