DESIGN document
Assignment 2: multithreaded HTTP server with redundancy

Purpose

The Objective for this assignment was to implement multithreading and redundancy as two additional features, building on top of assignment 1. Our new HTTP server will be able to handle multiple GET and PUT requests concurrently. We use synchronization techniques such as mutexes and locks to deal with race condition issues (which will be discussed further in this document). Our server will also have a redundancy feature which makes 3 separate copies of each file present in our server and if one copy is corrupted the server relies on the other backup copies.

Design

Functions and data structures

The `Main()` function:

The first thing we did in the main function was to implement the **getopt()** functionality to account for **-N -r** arguments in the command line. The getopt() function is a built in C/C++ function that is used for parsing command line arguments. The synopsis is shown below.

```
#include <unistd.h>
#include <getopt.h>

int getopt(int argc, char * const argv[], const char *optstring);
```

Getopt() takes as arguments argc, argv[] and options (opstring). These formatted string options then go by a switch-case logic depending on whether -N or -r is detected. If -N is detected we get the next argument in the command line to know the number of threads and set it to that number. If -r is detected we know there'll be redundancy files implemented.

The function returns -1 if there are no options to process from the command line.

Redundancy
The next thing we implement after `getopt()` is whether the redundancy flag was on or not. If redundancy was enabled we copy the files in the server to the 3 directories called copy1, copy2 and copy3 (which already exist), if we have a PUT command. In the case of GET, it compares the content of the files in the folders byte by byte, and returns the content if they are identical. If two are identical, it returns the content of one of the identical copies. If all three copies are different it throws a 500 server error.

In main(), we open our current directory using the `opendir()` function defined in `dirent.h`. This is to check all the files that are present in our current http server directory. Then we make sure the directory is not null and continuously read through all the entries present in the current directory. We use the `readdir()` function to achieve this. We conduct a quick check to make sure that the correct files/entries are being copied. If redundancy is enabled, we update it in our global structure and handle it in the process_request() function depending on what method it is. Then we finally close the directory using the `closedir()` function.

Synopsis for `dirent.h`

```
#include <dirent.h>

int           closedir(DIR *);
DIR           *opendir(const char *);
struct dirent *readdir(DIR *);
```

The above functions were the ones we used for our program, defined in the dirent.h library.

Multithreading

The next thing we had to ensure was that our design was multithreaded. To do so we first created threads in a loop with the number of threads specified in the command line arguments. Note the default value is 4 if -N is not provided. We used the `pthread_create()` function to create our threads. The synopsis is stated below.

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
void *(*start_routine) (void *), void *arg);
```

In our case, the first argument is a pointer to the threads that are created. The attributes are NULL in our case. The third argument takes a pointer to the `threadProcess()` function, which works kind of like a consumer function (Note: main function works as dispatcher). The last argument is a `thread_data` argument which is a data structure shared between threads that has some attributes which are explained in the data structures section.

After creating our threads, we have the same functionalities as assignment 1. We create the socket address structure with the server information. Then we `create()` our socket, `bind()` the address of the socket and `listen()` for incoming connections. Then in an always true while loop we `accept()` our connections which creates the client socket file descriptor.

We then `Enqueue()` these client connections to a Queue and make sure that this is done in a **critical region**. We use the mutex variable to lock our threads if it is already in use and another thread is trying to access that set of instructions (i.e. Enqueue client socket descriptors).

After we accept client connections we enter a critical region using the function `pthread_mutex_lock(&mutex)`. The client socket is then put into the Queue. Then we wait on the condition variable (`pthread_cond_signal()`) to signal that new work has been added to the Queue. We are then finally free to unlock the mutex using the `pthread_mutex_unlock()` function.

The `threadProcess()` function:

In the `threadProcess()` or worker function, we first get the thread ID (defined in a data structure we created) for our purposes to track the threads. Then in an always true while loop, we first set the client socket to -1 to indicate no new client sockets.
We then enter the critical region. This is done by locking the mutex using the `pthread_mutex_lock(&mutex)` function. We check the Queue status. If the Queue is empty then there is no work to be done by the worker thread so it waits on the condition variable. We use the function `pthread_cond_wait()`.
If the Queue is not empty and there is work to do, we get the client socket from the Queue and `Dequeue()` it. Note the Queue() uses a FIFO structure so we get the front of the Queue. We are then finally free to leave the critical section by unlocking the mutex. This is using the `pthread_mutex_unlock()` function.

Then we check once more to make sure the client socket exists and that client socket is then passed off to our assignment 1 functions to be handled. So we simply `read_http_response()` and `process_request()`.

Note: Each of these synchronization functions (for mutexes) are defined in the `pthread.h` library.

The `read_http_response()` function:

This function is the same as assignment 1. The high level idea here is to `recv()` the message from the client socket and store it in the buffer. The buffer is defined in the structure `httpObject`. We then parse the buffer string and store the method, filename and httpversion strings into the global structure. These attributes are later used in our other functions.

The `process_request()` function:

The process request function is also mostly the same as assignment 1 with some minor modifications. We first check to see if we have a valid http method, otherwise we throw 500

internal server errors. Provided we have a valid method we check to see if http method is a PUT or GET request.
If we have a PUT request, we first check if the filename we put is valid or not. We have a separate function called `valid_filename()` to check that (same as assignment 1). If we have a valid filename, we then check if it is in a list of known files. If it is, we lock the mutex for that file and execute our assignment 1 program. We open() the file with the filename we are trying to PUT, check for errors (same as assignment 1) and respond to client socket accordingly. If there are no errors in opening the new file descriptor, we continuously read() bytes from the client socket and write() to the new file descriptor. Note, if redundancy is enabled we need to copy the file that we want to PUT from the server directory into the copy directories. We have a copydata() function to implement that. Then we unlock the mutex for that specific file in the know files array list.
Note: If the file to be put is not in the known file array list, we lock the mutex for list of mutexes for each file and then we add the file to our list of known files so we can process it.
If our Http method is a GET, we first check if the filename to get is valid or not and respond accordingly. If it is valid, we check to see if it is in the known files list. If it is we, lock the mutex for that file to execute its operation. We check to see if redundancy is implemented and if it is, we compare the bytes in the files in the copy folders to see if they are similar to the file we are GETTING, and return the contents of the file if they are identical. If the files are not, we throw the 500 error message. The rest is pretty similar to assignment 1. We basically read data from the new fd referred to by the filename we are getting and write() that data out to client socket. After we are done with our file operation we have to unlock the mutex for the files in the known files list.

The `known_files()` function:

This is a very quick simple check to make sure that the files are in our list. We basically iterate through all the number of files and check if it's there, and return a non negative number (index) if it's in the list. If not the function returns -1.

The `should_copy()` and `copydata()` function:

These are simple functions implemented for redundancy. The should_copy() function has a list of the strings that are automatically opened from the opendir() function that we do not want. The function returns 1 if it is a file we want, otherwise it returns 0.
The copydata() function takes in the filename (from the current directory) and the pathname where it is to copy the data to. We open the source filename (from the current directory) for reading only and copy the data to pathname by write() to that destination file descriptor.

Data Structures used for this program (Non trivial)

We used a shared data structure to hold all the data and attributes for the threads. We have a threads[] array to store all the threads we create from our pthread_create() function. We have mutexes for the Queue (for entering and leaving critical regions when we Enqueue or Dequeue),

a pthread condition variable to signal whether there is work or no work, a Queue object called taskQueue to reference when we need to throughout the program. We also store an array of known files and mutexes for those files. We also have a redundancy flag to indicate if it's enabled or not. And finally we also store the port and address information in our struct.

```
struct shared_data {
    pthread_t threads[MAXNUMTHREADS];  //array of threads
    pthread_mutex_t queue_mutex;
    pthread_cond_t queue_cond;
    Queue taskQueue;

    int numFiles;
    char *known_files[MAXNUMFILES];
    pthread_mutex_t files_mutex[MAXNUMFILES];
    pthread_mutex_t new_file_mutex;

    int redundancy;

    int n;                              //number of threads
initialized
    const char* port;                   //port number
    const char* address;                //address
};
```

Abstract Data Type (Queue.cpp)

In our program we also used a Queue ADT, which was cited from Patrick Tantalo's CS101 class. The ADT was configured to work as a .cpp source file instead of a .c file. The Queue basically follows a FIFO structure and really helps in modularizing the program.

Note: Other structures used are the same as assignment 1. We are still using our httpObject struct to store message attributes which are the same.

Pseudo Code

Int main():

-Get options (-N and -r)
-Set values for N, r, port#, address#
-Open current directory and read entries
    While entry exists copy entries/filenames in a file array.
-Initialize Shared data struct with attributes
-Create N threads and files from file array into shared data struct
-Set socketaddress info
-Create socket, bind(), listen()

-while (true)
-accept() client sockets
-lock mutex
-Enqueue() client socket
-wait on mutex condition
-unlock mutex

Void threadProcess():
-Get thread ID and print
-while (true)
- set client socket to -1
- lock mutex
- if Queue == Empty, wait on mutex condition
- else getFront() to get the client socket then Dequeue()
- unlock mutex

If client socket is not -1,
read_http_response()
process_request()

Void read_http_response():
//Same as assignment 1.
-recv() from client socket
-parse the buffer
-store method, filename and httpversion by parsing the header format and updating to our global struct httpObject.

Void process_request():
-if invalid method throw 500 server error
-if method == PUT
-Check if valid filename, else throw error message
-if filename is in know files []
-Lock file mutex
- open() filename with the O_Create flag = fd
- if fd == -1 check errno and respond accordingly
- else read() from fd, and write() to client socket.
- if red == 1, copydata() into copy folders
- Unlock the file mutex
-else if not in known file[], lock a global mutex and add to the known file[] and unlock after execution.
-if method == GET
-Check if valid filename, otherwise throw error.
- lock file mutex.

-if red == 1

-compare (copy1, copy2) or compare (copy1, copy3) or compare(copy2,copy3) and return error if they are different, Otherwise open filename to read() data.

- read() filename to GET by reading from the fd file descriptor.

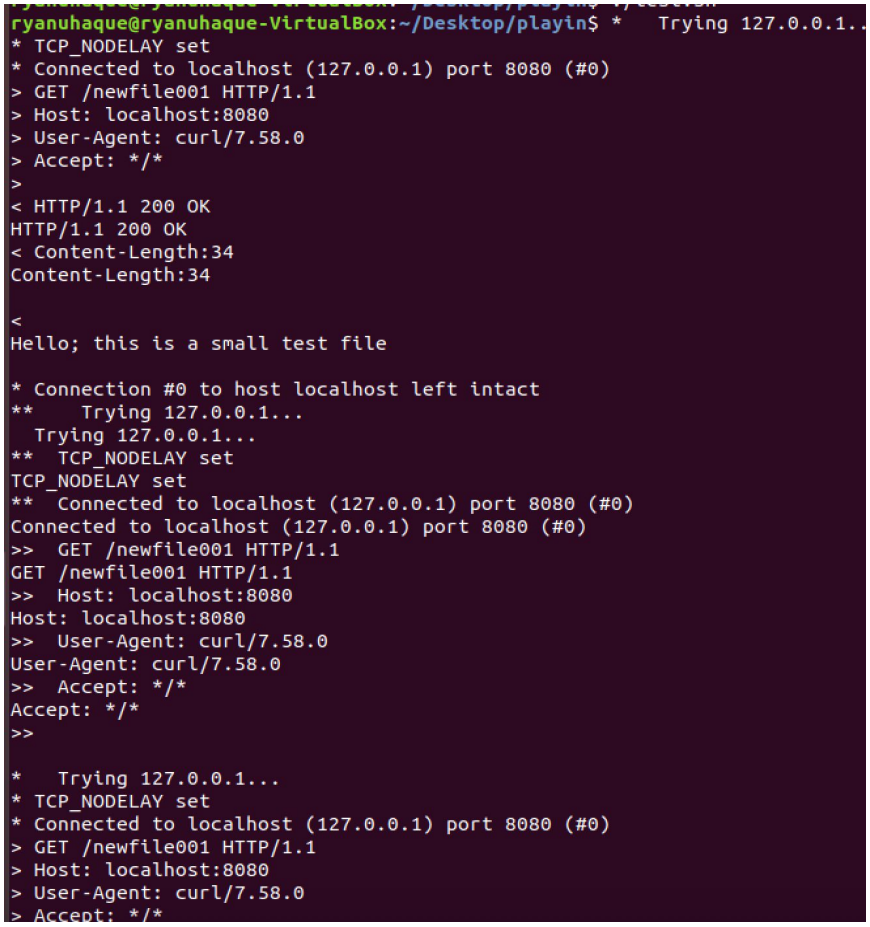
- if there are bytes to read(), continuously read() from fd and write() to client socket fd().

-unlock file mutex after program execution.

Testing

----------BREAK the Program----------

Just for fun and curiosity, I tried to comment out all the mutex functions. So in other words I was trying to run multiple requests without the use of any critical sections. Surely enough this gave out some random errors. The HTTP response seemed to have responded to BOTH client socket and also STDOUT from what I experimented. Seems like there was some interweaving. This is shown on the screenshot below.



Testing Multithreading

Note: Created shell script with the curl command for each test to make testing easier. In order to run shell script we do "**chmod +x (script_name).sh**" followed by "**./(script_name).sh**".

Test 1: Simultaneous GET requests with small file and large file. (get1.sh)

We created 2 files (SmallFile1 and LargeFile1) on our server.
Using one terminal we ran the command "**./httpserver localhost 8080 -N 4**" and on the other (client) terminal we ran "**(curl http://localhost:8080/SmallFile1 -v > out1 & curl http://localhost:8080/LargeFile1 -v > out2)**".

Result: As expected both requests were done simultaneously and the two files out1 and out2 were created with the respective data in them.

Test 2: Simultaneous GET requests with a small file and a small binary file. (get2.sh)

We created 2 files (SmallFile1 and binaryfile) on our server.
Using one terminal we ran the command "**./httpserver localhost 8080 -N 4**" and on the other (client) terminal we ran "**(curl http://localhost:8080/SmallFile1 -v > out3 & curl http://localhost:8080/binaryfile -v > out4)**".

Result: As expected both requests were done simultaneously and the two files out3 and out4 were created with the respective data in them. Note that out4 is not viewable so we ran the diff command between out4 and binaryfile to ensure that it is in fact the exact file.

Test3: Simple GET request to run N threads in a loop (get.sh)

We ran the curl command in a for loop in our shell script, to basically run multiple GET requests. As usual the client and server are run on different terminals.

Result: As expected, our server was able to handle all the GET requests and respond back to our client without any interweaving between requests.

Test4: Simultaneous PUT requests with a small file and a large file. (put1.sh)

We have 2 files present in the server (SmallFile1 and LargeFile1). Using one terminal we ran the command "**./httpserver localhost 8080 -N 4**".
On the other (client) terminal we ran the command "**(curl -T SmallFile1 http://localhost:8080/t123456789 -v & curl -T LargeFile1 http://localhost:8080/b123456789 -v)**".

Result: As expected 2 files t123456789 and b123456789 were created simultaneously. The data in the files were as expected. SmallFile1 contained t123456789 and LargeFile1 contained b123456789 respectively.

Test5: Simultaneous PUT requests with a small file and a binary file. (put2.sh)

We have 2 files present in the server (SmallFile1 and binaryfile). Using one terminal we ran the command "**./httpserver localhost 8080 -N 4**".
On the other (client) terminal we ran the command "**(curl -T SmallFile1 http://localhost:8080/t123456789 -v & curl -T binaryfile http://localhost:8080/b123456789 -v)**".

Result: As expected 2 files t123456789 and b123456789 were created simultaneously. The data in the files were as expected. SmallFile1 contained t123456789 and binaryfile contained

b123456789 respectively. Please note to test the binary file we used the diff command since the file cannot be viewed.

Test6: Redundancy GET request with the file in the copy1 folder different from the other copies

After creating three copies of SmallFile1 in each of the three copy folders, we started up the server with "**./httpserver localhost 8080 -N 4 -r"**. We then altered the version of SmallFile1 which was in the copy1 folder.
On the other (client) terminal we ran the command "**curl http://localhost:8080/SmallFile1".**

Result: As per the redundancy requirements, the server responded to the client with the version of the file which was present in the copy2 and copy3 folders, not the altered version in copy1.

Test7: Redundancy GET request with different versions of a file in all copy folders

After creating three copies of SmallFile1 in each of the three copy folders, we started up the server with "**./httpserver localhost 8080 -N 4 -r"**. We then altered all three versions of SmallFile1 in each of the copy folders (copy1, copy2, and copy3).
On the other (client) terminal we ran the command "**curl http://localhost:8080/SmallFile1".**

Result: As per the redundancy requirements, the server responded to the client with a 500 Internal Server Error.

Design Question:

- If we do not hold a global lock when creating a new file, our multithreaded server could enter a race condition where multiple threads attempt to create and write to the same file simultaneously. Imagine our server received the following two PUT requests: PUT MyBestFile -> TestFile12 & PUT MyLameFile -> TestFile12. Assume TestFile12 did not exist on the server prior to these two requests. TestFile12 would not have its own lock so if these two requests were processed at the same time on two different threads, both threads would attempt to create and write to TestFile12. In this scenario, TestFile12 could become corrupted with data from MyBestFile AND MyLameFile. A global lock prevents this possibility.
- As you keep increasing the number of threads, more processes can occur simultaneously which leads to better performance. This increased performance is limited by the hardware the server is running on, so performance cannot practically be improved indefinitely by adding more threads.