Class: CSE 130
Professor: Faisal Nawab
Assignment 2
Fall 2020

Design Document: multi-threaded HTTP server with redundancy

Contributors:
Adriel Naranjo - adjnaran
Abbas Engineer - afengine

Objective & Synopsis

Modify the HTTP server that is already implemented to have two additional features:

1) Multi-Threading - The server must be able to handle multiple requests simultaneously, each in its own thread.

2) Redundancy - Each file will have three copies instead of one, because if one copy is corrupted the reader has access to 2 others. Using synchronization techniques, service multiple requests at once, while ensuring that different copies of the same file are consistent.

In this project we utilize the tools provided by C/C++ libraries which are listed below and should be looked into before considering the rest of the design. Emphasis on the following:
`<sys/socket.h>, <netdb.h>, <netinet/in.h>, <errno.h>, <sys/stat.h>`
`<stdio.h>, <stdlib.h>, <sys/types.h>, <fcntl.h>, <unistd.h>, <string.h>, <unistd.h>, <iostream>, and <ctype.h>`

Initial Assumptions:

- Each request will process in its own thread - using worker threads.
- Worker threads may not be “busy wait” or “spin lock” - instead they must sleep by waiting on the lock/condition variable/semaphore.
- Allocate 16 KiB of buffer space for each thread.
- Dispatch thread listens to the connection, alerts one thread to handle the connection.
- If there are no threads available, it must wait until one is available to hand off the request.
- The httpserver must be able to process requests in parallel using multiple threads.
- Maintain a separate lock for each file.
- Use getopt(3) to parse options from the command line.

Let's get started:

To create a client-server model in C/C++ there are 3 major steps that need to be taken but now since we are implementing multithreaded requests and redundancy.

Step 1: create a root socket that can listen for incoming connections.

Get the information needed in order to create a host socket. A lot of references of this step can be found at: https://man7.org/linux/man-pages/man3/getaddrinfo.3.html. Other functions seen in this step can also be found in their own manpage(s).

Arguments to create server will come in as the following:

```
“./httpserver localhost: 8080 -N 6”
```

Furthermore the ‘-N’ is NOT required and there is a default value of 4 in that case. If -N not following by a number, should probably result in an error using perror.
Make these global variables so whole system behaves accordingly

```
Int thread_count = 0; //How many threads there are to be simult. Working.
Int threads_requested = 0;
```

this will control the thread count to ensure mutual
exclusion to resources in the critical section

```
      Sem_t mutex;
      Int main()
      // just as before we needed to create a server
      // initialize the mutex since we initialize a lot of values
      // for the server here anyways
      sem_init(mutex, 1);
      root_socket_init()
                case 1:
                      preferred_port = argv[2];
                      check_alphabet(preferred_port);
                      // just checks if the port is all digits 0-9
                case 2: preferred_port = "80";

// this is to support the multithreading
If argc == 5:
      If argc[5] is a numeric characters only
            Threads_requested = argc[5]
// default number of threads is 4
Else:
      Threads_requested = 4;

// Check the manpages for these, understanding these structures will help you
// create various types of sockets.

      struct addrinfo *addrs, hints;
      getaddrinfo(hostname,preferred_port, &hints, &addrs);
```

```
// initialize the root socket using socket()
root_socket = socket(AF_INET, SOCK_STREAM, 0);
Set_socket_attributes(root_socket);
// see link at beginning of step 1 to see how to do
// After validating the socket, put the socket to use by
// binding and listening.
bind(root_socket, address);
listen(root_socket);
```

Step 2: accept the connection and create threads for each of the requests and pass to subroutine which handles the request

```
// since this server is now multithreaded the way the server handles
// requests must ensure the fidelity of the data is kept in tack.
// still we accept any incoming connection…
while(New_connection = accept(root_socket, address)){
      // create a new thread for every request
      pthread_t a_request_thread
      // the program will then process the request with handle_requests
      pthread_create(a_request_thread, handle_requests, new_sock)
}
```

Step 3: Use subroutine to interpret and process the request in a thread.

```
Void *handle_requests(void root_socket)

      Int sd = *((int *)root_socket)
      // points to a message in the server
      // stick whatever is in the socket into the client_message
      // for parsing
      Char client_message[BUFFER_SIZE]
      recv(sd, client_message)
      sem_wait(
      // check manpages, sscanf extracts info from
      // formatted messages from http
      sscanf(client_message, "%s %s %s", request, file, protocol);

      sem_wait(mutex) // decrement resource
      Thread_count++ // keeps track of how many threads we are handling

// Due to the restrictions of the file name we must first
// validate the file resources that meet the constraints of this project
```

If the file is longer than 10 characters long:

Response 400 Bad request

If the file contains any special characters, non-alphabet, non-numeric

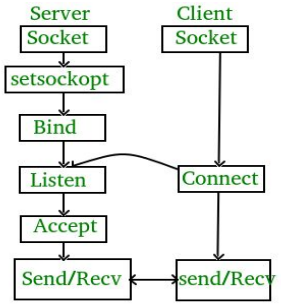
Response 400 Bad request

```
// Next, since we know we have a valid file resource name we must check if
// the request type is either a PUT or a GET. Once this information is
// obtained we can continue responding depending on the context.

      If GET:
            Create buffer to read from file if opened successfully
            get_file_info () // use stat() check manpages
            open(file);
            If permission to open file is not granted
                  Response forbidden 403
            If file does not exist in server directory
                  Response not found 404
            If file has permissions and exists
                  Create 200 OK Response and read file
                  contents from the file directory
                  into the buffer. Then write buffer
                  contents to the socket to give the client
                  the requested data.
      If PUT:
            Retrieve information about the
            content so we can use it to create the response.
            Create a buffer for data transfer.
            open(file); // this is the file
            that exists in the server to be written
            recv(new_socket, buf)
            // information to be PUT into file is received into the buffer
            write(fd, buf)
            // write the information into the file that exists on the server
      If None:
            Response 400 Bad request


      close(new_connection)
```

Diagram Citation: https://www.geeksforgeeks.org/socket-programming-cc/



Design Questions to keep in mind:

- Why is your system thread-safe?
    - The system is thread-safe, since we tried to prevent race conditions from happening when threads share the same data structures. If the mutex is used correctly then there will be synchronization in order to access the critical region, which will properly protect the thread from hazardous modifications. As long as all of our functions are thread-safe and all the shared data is protected then the server should be thread safe. Another way of keeping our systems threads safe is eliminating any global variables that may be used between threads. Additionally methods like **`strtok()`** where changed to the thread safe version, **`strtok_r()`**
- Which variables are shared between threads? When are they modified?
    - When a thread holds a lock over a file within the shared memory, it can make whatever modifications it wants to before it unlocks itself, freeing up that updated memory to be accessed by another thread.
- Where are the critical regions?
    - Anytime there is shared data that multiple threads are trying to access, whether it is files or a buffer, or a counter that two threads operate on. In our design we mainly protected our file resources so that when one thread was doing a read, that it would be reading the most recently written data from that document. So any time that resource is being accessed, the critical region must be initialized by locking the resource before making any changes to the resource, then unlocking the lock on the resource when the operations have been done on the resource.

I/O:
Using large and small files consisting of binary data and other strings as inputs that we can run GET and PUT requests concurrently on.
The output should return any error that could have occurred from the commands and or the errors mentioned above. Else should return something like `'HTTP/1.1 200 OK'`.

Worker Thread:

```
void * workerThread(void* pclient){
    int new_socket = 0;
}
```

The threads here will be obtained from the command line input based on the number following the -N argument. While doing so we initialized the “new_socket” which will be used to keep track of the socket fd.

```
while(1){
        pthread_mutex_lock(&mutex);
```

```
            while(myqueue.empty() == 1){
                  pthread_cond_wait(&cond, &mutex);
            }
            new_socket = myqueue.front();
            myqueue.pop();
            pthread_mutex_unlock(&mutex);
      }
```

Mutex locks are used to block access to a variable or set of variables from threads, if they are currently being used by one.
The mutex object is locked by calling **`pthread_mutex_lock()`**. Putting the threads to sleep until a socket fd is pushed onto the queue
Once the queue starts to fill, the **new_socket** variable grabs the first element in the queue and **`pthread_mutex_unlock(&mutex)`** unlocks the mutex allowing another thread to get hold of it and lock it.

Main Function:

```
      int main(int argc, char*argv[]){

            int opt, num_of_threads;
            int flag = 0;
            while((opt = getopt(argc, argv, "N:r")) != -1){
                  switch(opt){
                        case 'N':
                        flag = 1;
                        num_of_threads = int(atoi(optarg));
                        break;
                        case 'r':
                              is_redundant = true;
                              Break;
                  }
            }
      }
```

Within the main function, we check to see if the flags “-N” and/or “-r” are retrieved from the command line. Using function getopt() permutes into the command line grabbing argv
The colon ensures that we grab “-N”.
If N is in the command line:
Then we check to set the number of threads to the argument grabbed from getopt() function.
If N is not in the command line:
Set a flag so that the server uses the default number of threads.
The extra case is to check if there is redundancy, with the “-r”.

Question:

- If we do not hold a global lock when creating a new file, what kind of synchronization problem can occur? Describe a scenario of the problem.
    - A global lock is how all threads can communicate with one another. So when a thread holds a global lock when creating a new file. It basically is saying telling all the other threads that “I am making some changes to this, do not try and touch it right now”. Incorrect use of locks can result in interleaving Race conditions. If 2 threads are trying to access the same shared memory space for creating a new file then they are in a race with one another. With the global lock, that should only occur when a new file is created, it is important for the thread to maintain its lock while doing so, so that other threads do not try to access memory that is “under construction” and is prone to change.
    - Deadlock can also occur. An example of this If the clients fill up the servers buffer with requests to create the same file, therefore creating multiple global locks, and then blocks itself waiting to add another, the server might fill up the clients buffer with responses then block itself. Both would be waiting on the other and neither would be making any progress.

- As you increase the number of threads, do you keep getting better performance/scalability indefinitely? Explain why or why not.
    - The answer is a conditional yes, The increased number of threads can maximize performance and scalability. However there is a maximum threads value that will start to inhibit maximum performance of the machine. This is where software and hardware coincide. The maximum number of threads that can be used depends on the hardware of the machine. The number of processors and the number of cores within each processor.
    - It is like a roller coaster ride. If you have 4 seats available then filling all 4 seats with 4 people will maximize its performance, 1-3 people wastes the resources capabilities, and going to 5 or more people will cause the additional riders to have to wait in a queue, which doesn’t improve performance or scalability, slows down the efficiency.

Testing Issues:

Through modularization, unit testing was the most effective way to test. With breaking into 2 parent components of: Multi-Threading and Redundancy, each of them with their subcomponents we are able to reduce the amount of time we might have needed to dedicate to testing and debugging.

- Checking that multiple threads can run and process requests concurrently. When you increase the number of threads, does this lead to processing simultaneous requests faster?
    - Increasing the number of threads does not necessarily process each request faster but can lead to processing simultaneously more requests per timeframe, to

a certain extent. If the hardware can support it, there can be more threads run concurrently without slowing down the processing rate of requests. However, after a certain point, the hardware can no longer support the amount of threads required to process all the requests. This can in fact result in an inverted trend slowing down the processing of simultaneous requests.

- Checking that the redundancy mechanism works. What happens if one file is different from the other two? What happens when all three files are different from each other? What happens if a copy of a file does not exist in some of the “copyX” folders?
    - When GET:
      As long as one file has uncorrupted data then that copy will be sent to the client. Otherwise an error will occur depending on the circumstances
      file with invalid name that doesn't exist -> 400
      file with 2/3 files giving no permissions -> 403
      file with 3/3 files giving no permissions -> 403
      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 -> 500
      file with 1/3 files giving no permissions, 2/3 files that don't exist -> 404
      file with 2/3 files giving no permissions, 1/3 files that don't exist -> 403
    - When PUT:
      All three files will always be written to if the permissions are granted. If the file does not yet exist, it will be created and put into the server wherever directed to i.e copy[1-3]. If the write to any of the files is not permitted or has been corrupted then none of them will be written to because if a GET request is sent for these files we must understand that some of them will be corrupted so we should not send them if that is going to be the case.

Conclusion:

When testing with concurrent threads we learned that it can be difficult to debug, sometimes these concurrent bugs are not reproducible. One thing we would try to ask ourselves when designing is, if the tests are too complex our design is probably not right. Minimizing the interaction between threads is another key to making testing more efficient.
One issue with our implementation of redundancy: with the PUT consistently writing the message into each copy, because we are supposed to write information into each file. The function recv() would not allow us to read multiple times through the socket, receive the message from the socket, read from that buffer and write to the other files.