Assignment 2 Design

I. Introduction

A. Objective

The goal of this assignment is to implement a multi-threaded HTTP server that responds to GET and PUT commands to read and write files, respectively. This server will also have a new feature, redundancy.

B. Statement of Scope

The HTTP server will listen for GET or PUT requests, followed by a 10 character ASCII file name and, optionally, the content-length on a user-specified port. A GET request to the server followed by a filename will return the contents of a file that is stored persistently in the same directory or return an error if the file is not found. A PUT request to the server, followed by a file name and data will send the data to be put into a new file denoted by the file name. Each response header to a GET or PUT request will be a valid HTTP response that specifies the HTTP version, the status code, and content length.

If the "-N" flag is detected with a number after, httpserver will create the specified number of worker threads that will handle one request at a time.

If the "-r" flag is detected, httpserver will copy all the files from the current directory into 3 existing folders named copy1,copy2, and copy3. Each copy will exist in a separate folder. When a PUT request is received, the program will write to all three copies of the file. When a GET request is received, the program checks whether the files are identical before returning the content. If two or three are identical, then the contents of the identical files are returned. If all three copies are different, a 500 error is returned.

II. Restrictions

The program will not utilize standard libraries for HTTP or any of the C library FILE* functions except for when printing to the screen. The system calls: read and write are used for file I/O. The other system calls: socket, bind, listen, accept, connect, send, and recv are used for socket programming. The port numbers that the user can specify are restricted to port numbers above 1024 unless the server is run with sudo. Each thread is also only allowed to be allocated 16 KB of buffer space.

III. Program Discussion

A. Thread Management

a. Our system is thread safe because more than one thread can execute the same code segments without synchronization problems. Code segments that could potentially cause synchronization issues are protected by mutexes. Aside from one condition variable and one mutex, all other variables are "local" to each individual thread and are not shared between other threads. We do not implement any global variables other than SIZE, which is never modified and just specifies the maximum allowed buffer size, and one global lock which ensures thread safety when new files are created.

- **b.** The condition variable **request** and the mutex **queueLock** are shared between the threads.
- **c.** The shared variables are modified in the following situations:
    - **a)** **queueLock** is locked after the dispatch thread accepts a new connection and pushes the accepted socket file descriptor into the queue. **queueLock** is unlocked after the file descriptor is enqueued.
    - **b)** **request** is signaled after the dispatch thread successfully enqueues an accepted connection's file descriptor.
    - **c)** In each worker thread, **request** is used so that the thread waits until there is an available request in the queue. **queueLock** is also "toggled" to the unlocked state while waiting for an available request and back to the locked state after being signalled.
    - **d)** **queueLock** is locked if there is an available request in the queue. The worker thread responsible for that request will then pop the front of the queue, get the file descriptor of the request, and then unlock **queueLock**.
- **d.** The critical sections are in the following segments of code:
    - **a)** In the dispatch thread (located in **main()**), the critical section is where the accepted connection's file descriptor is pushed into the request queue.
    - **b)** In the worker threads' start function (**workerThread()**), the critical section is where the worker thread pops the front of the request queue.
    - **c)** The **workerThread()** start function will call a function named **executeFunctions()**, where there are several critical sections:
        - **(1)** When a file lock does not yet exist, the global lock is locked, the file lock is created and inserted into the map, and then the global lock is unlocked.
        - **(2)** The file lock is locked before **get_request()** or **put_request()** is called and unlocked after those functions have finished.

B. Redundancy Functioning

- **a.** Redundancy works in our program because we are only allowing one thread to access all 3 files inside copy1, copy2, and copy3 at a time. The specific implementation is described below:
- **b.** Inside of **get_request()** and **put_request()** we have an argument named **rflag** which is initialized to a value in **main()** to determine the presence of a -r argument when the program is run.
    - **a)** **rflag** is passed in all the way to **get_request()** and **put_request()** so those functions can handle requests properly.
- **c.** We initialize a **fileLock** for each file that does not have a **fileLock**.

d. We then lock **fileLock** before calling **get_request()** or **put_request()** and unlock **fileLock** after **get_request()** or **put_request()** returns to the parent function.

   a) Only one thread can access **get_request()** and **put_request()** for a specified **fileLock**.

      (1) Only **put_request()** has file manipulation functions

      (2) Only **get_request()** has file comparison functions

   b) **put_request** with **rflag = true**, will treat filename as a pathname "./copy1/filename" instead of "filename" and continue with execution. Additionally, after the **recv()** is done, **copyFiles()** will be called to copy contents of "./copy1/filename" to "./copy2/filename" and "./copy3/filename"

   c) **get_request** with **rflag = true** will also treat filename as the pathname "./copy1/filename". The only addition is that we make sure at least ⅔ copies are identical and return the correct copy.

e. Because we only unlock **fileLock** after **get_request()** or **put_request()** has been called, only one thread is allowed to access all 3 files leading to all changes to be atomic.

IV. Program Walkthrough

1. Global declarations:

   a. Queue **commQ**

      i. A request buffer for storing/taking out requests

      ii. Used by **main()** and ***workerThread()**

   b. Mutex lock **newFileLock**

      i. Prevents creating more than one fileLock for the same file

      ii. Used by **executeFunctions()**

   c. Unordered_map<file name, mutex lock> **fileLock**

      i. Dynamically stores and accesses a unique mutex lock for each file

      ii. Used by **executeFunctions()**

   d. Struct **HttpObject**

      i. Stores a parsed **request** from **commQ**

      ii. Used by almost all functions in ***workerThread()**

   e. Struct **Flags**

      i. Defines specific conditions for a **HttpObject**

      ii. Used by almost same functions as **HttpObject**

   f. Struct **requestLock**

      i. Used to pass in mutex/condition pointers and -r flag

      ii. Used by **main()** to pass into ***workerThread()**

2. Start in **main():** The main initialization/setup before accepting HTTP requests

a. Parse command line args with **getopt()** to find -N and -r. If -N is present, set the number of worker threads to N. If the flag was present but either no integer or 0 is specified, throw an error. If -r is present, set the rflag to true. If another flag is present, throw an error.

b. If rflag is true, call **copyFiles()** to make three copies of all files in the server and place them into separate folders called "copy1", "copy2", and "copy3".

c. If the number of workers variable is still initialized to 0 after the checking done in **getopt()**, set numworkers to the default value, 4.

d. Obtain the host address with **argv[optind]**. The host address should be the "first" command line argument after getopt() is called.

e. Determine which port number to listen from using the function **getPort()**, which takes in argc and argv[]. If **optind++ == argc-1**, then the port number is not specified and should be set to the default value, 80. Else, port number is specified in **argv[optind]**. The user-specified port must be above 1024 and the program will exit if an invalid port is provided.

f. A **sockaddr_in struct** named **server_addr** is declared. **inet_aton()** is used to convert the first command line argument, which specifies a hostname or an IP address, to a network address for the socket. The port number is then assigned to the socket as well.

g. The server socket is created with a file descriptor, **server_socket**, that refers to it.

h. Socket options are set using **setsockopt()** with the option **SO_REUSEADDR** to allow the reuse of addresses and ports.

i. **bind()** is used to assign **server_addr** to the server socket.

j. **listen()** is used to listen for incoming connections on the server socket.

k. A **sockaddr struct** named **client_addr** is declared.

l. A condition variable, **request**, is initialized. This condition variable will be used to ensure that the threads will sleep while waiting for a request to perform.

m. A mutex, **queue**, is initialized. This will be the lock used to maintain the queue of requests.

n. A **requestLock** struct named **sink** is declared. This struct consists of two pointers to the previously declared condition variable **request**, mutex **queueLock**, and the previously determined **rflag**.

o. Declare an instance of the **httpObject struct** named **request** and the flag **struct** named **flag**.

p. A pthread_t array named **tid** will be initialized with **numworkers** amount of indices in **main()**. **pthread_create()** will be called **numworkers** amount of times to create N worker threads. These worker threads will pass the **sink** struct into and execute the start function named ***workerThread()**.

3. In ***workerThread()** initialized by main()**:** The following will be executed before the worker threads wait to be awoken by the dispatcher thread in main():

   a. An infinite while loop will be implemented so that each thread will continuously:
      i. Check if there are requests to handle (check if the queue **commQ** is empty)
         1. If **commQ** is empty, sleep the thread using conditional wait **pthread_cond_wait(newReq, queueLock)** until the conditional signal **request** is received from **main()**
            a. Currently the thread is sleeping and **main()** continues execution in step **4**
         2. If **commQ** is not empty, continue execution in step **5**

4. Back to **main()** to continue execution while *workerThread() is sleeping**:**
   a. An infinite while loop will be implemented to represent the dispatcher thread. This thread will continuously accept client requests until it is terminated. The while loop will then do the following:
      i. Once a connection request is accepted, the mutex **queueLock** will be locked with **pthread_mutex_lock(queueLock).**
      ii. The file descriptor of the accepted connection is pushed into the **commQ** queue.
      iii. Once the comm_fd is pushed onto **commQ, queueLock** is unlocked.
      iv. Then the condition request **pthread_cond_signal(request)** is called to
         1. If any thread(s) are sleeping, **request** will wake up one thread to handle the request
         2. If no threads are sleeping(meaning they are all busy), the next available thread will automatically **pop()** a request to handle instead of sleeping.

5. Rest of ***workerThread()** now continues after the thread is woken up:
   a. We are still in the infinite while loop. We are either after **pthread_cond_signal(request)** is received. After being woken, each thread will:
      i. Lock the **queueLock** with **request** or manually locking **queueLock** if queue **commQ** is not empty
      ii. Take the **front()** of **commQ**, since that file descriptor is the oldest.
      iii. **pop()** the front of **commQ** off.
      iv. Unlock **queueLock**.
      v. Call function **executeFunctions().**
      vi. Close the file descriptor after handling the request successfully.
   b. At this point the while loop reaches the end and repeats (**3a)**.

6. In **executeFunctions()**, the following will occur:

- **a.** HTTP header from the client socket will be received and stored in a buffer.
- **b.** **recv()** once from the client to get/parse the header
- **c.** Store the first line (header) into the request object **request**
- d. Verify that the HTTP version and filename requirements are correct
- **e.** Continue parsing through the first recv for content length or 2 new lines
  - **i.** Set a pointer to end of 2 lines to signify start of body for put request
- **f.** Check whether the filename specified in the request has a lock.
  - **i.** If a lock does not exist, lock **newFileLock**
    - **1.** Create and initialize pthread_mutex_t **fileMutex**
    - **2.** insert the (filename, mutex) into the unordered_map **fileLock**.
    - **3.** Unlock **newFileLock**
- **g.** Lock the mutex **fileLock(name)** and check whether the request type is a GET(**7)**, PUT(8), or something else.
  - **i.** If the request is a GET request, call the function **get_request()**.
  - **ii.** If the request is a PUT request, call the function **put_request()**.
  - **iii.** If the request is not a GET or PUT, then it's a bad request. The HTTP response is constructed with a 500 status code.
- **h.** Unlock the mutex **fileLock(name)**

7. If the request is a **get_request():**

- **a.** The file specified by the file name in the header is opened. If it fails to open, the appropriate status code is determined, put into the httpObject **request**, and the HTTP response is constructed and sent to the client using **constructResponse()**. If the file is opened successfully, the content length is determined, and **rflag** is checked.
  - **i.** If **rflag** is true, check whether or not the contents all three files are identical.
    - **1.** The file paths to the specified file for the folders copy2 and copy3 are constructed.
    - **2.** Each file is opened and checked for a successful **open()**.
    - **3.** If any two copies are opened successfully and **compareFiles()** returns as true, set the **sendfile** flag equal to the file descriptor of one of the identical files.
    - **4.** If none of the copies open successfully, set the status code to 500.
  - **ii.** If **rflag** is false, set the **sendfile** flag equal to the file descriptor of the only file.
  - **iii.** The HTTP response is constructed and sent to the client, then **sendfile** will be continuously read and sent to the client until the end of the file is reached.
  - **iv.** The file is then closed before returning to **executeFunctions()**.

8. If the response is a **put_request():**

a. The file specified by the filename in the header is checked to see whether it already exists or not using **access()**. The file path provided to access() is obtained through the function **pathName()**.

b. **pathName()** does the following:

   i. If **rflag** is true, then a path to the copy1-folder file is created and returned.

   ii. If **rflag** is false, then just return the original file name string obtained from the request header.

c. The flag **exists** in struct **flags** will be updated accordingly.

d. If the put request has no error, the file path is opened; if the open fails, return a 500 status code with the response through **syscallError()**. This is the appropriate status code since, unlike GET, if the named file does not exist, it should be created. This is because we use the **O_CREAT** flag in our **open()** syscall.

   i. We dont want to create a file if there is an error with the put request but we have to **recv()** to clear the buffer for the next request regardless of valid or invalid

e. Then check whether the content length of the file was specified in the header (whether or not it is equal to 0).

   i. If the content length is specified, the server stops receiving data after the specified number of bytes, constructs the HTTP response, sends the response to the client, closes the file, and then returns to **executeFunctions()**. If the content length isn't specified, the server receives data until the client closes the connection or the end of the file is reached.

V. Pseudocode

A. struct httpObject:

```
        char[] type
        char[] filename
        char[] httpversion
        int status code
        content length
END
```

B. struct flags:

```
        bool exists
        bool isFirstParse
        bool nameIsValid
END
```

C. struct requestLock:

```
        pthread_cond_t *newReq
        pthread_mutex_t *queueLock
        bool rflag
END
```

D. bool compareFiles(int file1, int file2):

```
        char buf1[SIZE]
        char buf2[SIZE]
        WHILE file1 != EOF
            IF file2 == EOF
                return false
            IF (!memcmp(buf1, buf2, SIZE)
                RETURN false
        RETURN true
END
```

E. void executeFunctions(comm_fd, httpObjects request, buf, flag):

```
Recv from client into buf
Sscanf from buf (type, filename, httpversion, flag) delimited by space into request
Check if header is valid or not
While loop for parsing through header
        Parse for content-length
        Store content length if found
        If 2 new lines found set body pointer
Create lock for file if file does not have a lock
Call Get or Put request depending on type
END
```

F. void parse_request(comm_fd, httpObject request, char buf, flag):

```
        IF is first time calling parse
            Check http version
            Check if name is valid
        (ELSE parsing for content-length)
        IF content length string exists
```

```
            Store into request.content length
        ELSE (no content length string in header)
            break and leave content length empty
    END
```

G. int main(): //asgn1 focused code omitted, but still elaborated above in section "Design/Walkthrough"

```
        Parse command line arguments for -N and -r flags and set flags and variables accordingly
        IF -r flag is present, make three different copies of all files in the server and put into folders
            copy1, copy2, and copy3
        IF -N is not present, default numworkers = 4
        Initialize condition variable and mutex
        Create numworkers # of worker threads, all with the start function *workerThread()
        WHILE(true)
            Accept incoming connection
            Lock mutex
            Push accepted file descriptor into queue
            Unlock mutex
        END
        Pthread_cond_signal(condition variable)
    END
```

H. void* workerThread(void* arg):

```
        Initialize pointer to condition variable and mutex
        Obtain rflag value
        Create httpObject struct
        Create flags struct
        WHILE(true)
            IF commQ.empty()
                Pthread_cond_wait(condition variable)
            ELSE
                Lock mutex
            Comm_fd = commQ.front()
            commQ.pop()
            Unlock mutex
            Set first_parse flag to true
            Call executeFunctions()
        END
    END
```

I. void get_request(com_fd, httpObject request, buf, flag):

```
        Memset buf since all objects are parsed into request already
        Open file with file path for reading
        IF error
            Set status code to error number
            Construct response
            Send constructed response
        ELSE (get request succeeds)
            Set status code to 200
            Get content length of file being read and store into request
            IF rflag
                Create the file paths for folders copy2 and copy3
                IF two of the files from the folders copy1, copy2, or copy3 were successfully
                    opened and compareFiles() returns true, set sendFile to open one of the
    identical files
                ELSE
                    Set status_code to 500
            ELSE
                Set sendFile to the filename stored in the request struct
            Construct response
            Send constructed response
            Read and send sendFile
        Close sendFile
    END
```

J. void put_request(com_fd, httpObject request, buf, flag):

```
        Check whether file name already exists and set exists flag accordingly
        If no error open file from filename for writing
```

```
            Check to see if open is successful
        If buf is not empty write first from buf to file
        IF content length exists:
            While content length is not 0:
                Write from buf to file
                Content length-bytes received
            IF content length is 0 (meaning all of data was received)
                IF file already existed
                    Set status code to 200
                ELSE
                    Set status code to 201
            ELSE(data was not all received)
                Throw error 500
        ELSE (content length does not exist)
            While client does not close connection(recv != 0)
                Read from buf to file until EOF
        IF rflag
            Close the source file
            Reopen the source file
            Copy contents of source file into files in folders copy2 and copy3 with copyFiles()
        Construct response
        Close file
    END
```

K. void construct_response(comm_fd, httpObject request):

```
        Char length_string[20]
        Convert request→content_length from a long int to to a string and store in
            length_string
        Char response[50]
        Copy request→httpversion to response
        Concatenate “ ” to response
        Concatenate getCode(request→status_code) to response
        Concatenate “\r\n” to response
        Concatenate “Content-Length: ” to response
```

```
          Concatenate length_string to response
          Concatenate “\r\n\r\n” to response
          send(comm_fd, response, strlen(response), 0)
END
```

VI. Testing

A. Classes of Tests

White-box testing will be utilized while testing the functionality of the program. Although the use of C library FILE* functions are restricted for the functionality of the program, they will be used during testing as a convenient way to debug and keep track of values such as loop iterations and filenames. These print statements will be removed before submission to ensure clean and readable code. The program will be incrementally developed, so these tests are performed at every stage. These tests are described as follows:

- **a.** Testing if the command-line arguments are parsed correctly by printing to stdout.
  - **a)** Test with a specified port number, -N, and -r
  - **b)** Test with a specified port number and -r
    - **(1)** Number of threads should default to 4
  - **c)** Test with no port number, -N, and -r
    - **(1)** Port number should default to 80
- **b.** Testing threads, locks, and conditional signaling without redundancy
  - **a)** Test if a single thread can replace the while loop in **main()**
  - **b)** Test if we can pass in requests from newly created dispatch thread in **main()** to a sleeping **workerThread()**
    - **(1)** Test the condition variable **request** called from **main()** to wake up **workerThread()**
    - **(2)** Test the queue lock **queueLock** so that the request queue **commQ** can only be accessed by one thread (including the dispatch thread in **main()**)
  - **c)** Test if we can have multiple **workerThread()**s retrieve/execute requests from **commQ** correctly
    - (1) Test if **queueLock** and **request** work with multiple threads and the dispatch thread
  - **d)** Test the global new file lock and the file locks
    - **(1)** Test the unordered_map **fileLock** for setting a lock to a filename
    - **(2)** Test the global new file lock **newFileLock.** It is for preventing more than one **workerThread()** from creating a **fileLock** for the same file
    - **(3)** Test the functionality of **fileLock** by performing multiple put/get requests without erroring
- **c.** Test redundancy
  - **a)** Test if **main()** copies all files from current directory to the folders copy1, copy2, copy3
  - **b)** Test if the redundancy flag **rflag** can be passed in from **main()** to **workerThreads()**
  - **c)** Test **get_request()** with **rflag = true**
    - **(1)** Check if at least two of the files are the same return one of the files
    - **(2)** If all three are different then return status code 500
  - **d)** Test **put_request()** with **rflag = true**

(1) Check that a put request will change all three files in copy1, 2, and 3 and not change files anywhere else

Black-box testing will also be utilized through manual testing. The code will be tested using a variety of different combinations of inputs. Many curl requests were compiled together in a shell script. The script runs the clients and sends requests back to back in the background (the curls have "&" at the end of the command). The tests include the following:

- **a.** Test **GET** requests
  - **i.** A valid get request
    - **1.** With one thread
    - **2.** With one thread and redundancy
    - **3.** With multiple threads
    - **4.** With multiple threads and redundancy
  - **ii.** A get request with invalid files (multiple threads with redundancy & non-redundancy)
    - **1.** Nonexistent file
    - **2.** Invalid name
      - **a.** File name with a non alphanumeric character
      - **b.** File name less than 10 ASCII
      - **c.** File name more than 10 ASCII
  - **iii.** Files with no read permission
    - **1.** Without redundancy
    - **2.** With redundancy
      - **a.** ⅓ files have no read permission
      - **b.** ⅔ files have no read permission
  - **iv.** One missing copy in copy1, copy2, or copy3
  - **v.** Two missing copies in copy1, copy2, or copy3

- **b.** Test **PUT** requests
  - **i.** A valid put request
    - **1.** With one thread
    - **2.** With one thread and redundancy
    - **3.** With multiple threads
    - **4.** With multiple threads and redundancy
  - **ii.** A put request with invalid files (multiple threads with redundancy & non-redundancy)

        1. Nonexistent file
        2. Invalid name
            a. File name with a non alphanumeric char
            b. File name less than 10 ASCII
            c. File name more than 10 ASCII
    iii. Files with no read permission
        1. Without redundancy
        2. With redundancy
            a. ⅓ files have no read permission
            b. ⅔ files have no read permission
    iv. Files with no write permissions
        1. Without redundancy
        2. With redundancy
            a. ⅓ files have no write permission
            b. ⅔ files have no write permission
    v. One missing copy in copy1, copy2, or copy3
    vi. Two missing copies in copy1, copy2, or copy3
    vii. If header and body are recv()-ed all at once
    viii. If header and body take multiple recv()s to get

B. Expected Output

The following expected outputs will assume that httpserver is using more than one thread because all single threaded httpserver requests work from our assignment 1 httpserver.

a. When httpserver receives a valid **GET** request **without** redundancy enabled, the appropriate response header is sent back to the client, followed by the contents of the specified file stored in the server directory, if GET was successful.

b. When httpserver receives a valid **GET** request **with** redundancy enabled, the appropriate response header is sent back to the client, followed by the contents of one of the identical copies from folders copy1, copy2, or copy3, if GET was successful and at least two copies were identical to each other.

c. When httpserver receives a valid **PUT** request **without** redundancy enabled, the appropriate response header is sent back to the client and the contents of the specified source file will either be written into a new file or overwritten into an existing file on the server directory.

d. When httpserver receives a valid **PUT** request **with** redundancy enabled, the appropriate response header is sent back to the client and the contents of the specified source file will either be written into three new files in the folders copy1, copy2, and copy3, or overwritten into three existing files in the same folders.

VII. Assignment Questions

A. 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.

If we do not hold a global lock when creating a new file, multiple threads may try to create the new file lock at the same time. For example, if two threads are trying to perform a PUT request on a file, let's say file1, which does not exist yet, both threads will check the hashmap, see that a file lock does not yet exist for file1, and try to create the file lock. When we were still trying to implement the global lock into our program, we actually misplaced the global locks and got an out of range error when trying to access the file lock in the hashmap. This makes sense, since there was probably an error when creating the file lock and inserting the key-value pair into the hashmap; thus, when we try to lock the file lock afterwards, we get the out of range error since that key-value pair does not exist in the map.

B. As you increase the number of threads, do you keep getting better performance/scalability indefinitely? Explain why or why not.

No. At some point, the overhead from running so many threads will outscale the performance of running that amount of threads. For example, having a large number of threads when handling a relatively small number requests would render a lot of threads useless. Even in the case that the program has a large number of threads to handle a large number of requests, the program throughput will bottleneck at a certain number of threads, due to its design. Since the dispatch threads and worker threads share some locks, the addition of more worker threads increases the wait time for entering the critical section, for all threads. This is especially detrimental to the dispatch thread (which uses a shared mutex to enqueue requests for the worker threads) since worker threads can only receive and handle requests as fast as the dispatch thread can dispatch requests.

VIII. Citations

A. How to parse through all filenames in directory
https://stackoverflow.com/questions/4204666/how-to-list-files-in-a-directory-in-a-c-program/17683417