Design Doc for Asgn2 Multi-Threaded HTTP server

The server is thread-safe when multiple threads try to access shared resources. In order to manage the incoming requests, we have made a global request queue. Everytime a new request is made, the queue is locked with pthread_mutex_lock(), and the request is then pushed to the back of the queue. After this process is complete, the queue is then unlocked with pthread_mutex_lock. The reason why it's not thread safe is because the queue is a shared resource among threads therefore enabling the possibility of race conditions. Accessing files is also not thread-safe because they are shared among the threads as well. In both cases of PUT and GET requests, files are locked before they are opened, read from, and written to. After the process has finished the files are then unlocked.

The critical section refers to the resources that are shared between the threads in the program but only one thread should access it at a time in order to avoid race conditions. For example, only one thread should be able to access file “a”. When another thread wants to access this same file, that thread must acquire the lock pertaining to that file or else it will not be able to open, read or write to that file.

In order to enable redundancy with our server, the -r flag is scanned on the command line and stored in a boolean variable. If this variable is set to true, then redundancy is enabled and a new set of functions and error checking is run. When the server is first created, the files that exist in the directories won’t have a lock associated with them. We check the folders and files if they exist and if we have the correct permissions. If so, we create locks for each valid file. The name of the file is then paired with the file lock into an std::map variable which we called “globalMap”. This will be further explained and shown in Part 3.

Part 1: Struct HttpObject with Flags and Struct ThreadHandler

The HttpObject struct is used to store and maintain the data from the server and the client across all processes and functions. The ThreadHandler struct will store the necessary values needed to maintain a multithreaded function. The FileHandler struct will store the boolean status for when we check if the file has access to read or write permissions. These structs will be stored in the httpserver.h file:

```
typedef struct HttpObject
{
    char     buffer[BUFFER_SIZE];           // Header will not be larger than 4 KiB
    char     method[METHOD_SIZE];           // PUT, GET
    char     fileName[FILENAME_SIZE];       // The file we are trying to create or read from
    char     httpVersion[HTTPVERSION_SIZE]; // HTTP/1.1
    char     statusText[STATUS_SIZE];       // "OK", "Bad Request", "Not Found", etc..
    char     response[RESPONSE_SIZE];       // Contains the full response send to client
    char     afterExpect100[BUFFER_SIZE];   // Grabs any extra file contents on first read()
    char*    port;                          // port number given on command line
    int      contentLength;                 // size of the client's file
    int      clientSocket;                  // socket the client is on
    int      numberOfThreads;               // number of threads
    int      statusCode;                    // 200, 201, 400, 403, 404, 500
    bool     rFlag;                         // -l flag
    bool     NFlag;                         // -N flag
    bool     hasSent;                       // have we sent the response to the client
    bool     doesExist;                     // Does the file exist for a GET request
    bool     hasReadPermission;             // Do we have read permission for a GET request
    std::vector<std::string> vecFiles;      // stores the filenames in the server folder
    std::vector<std::string> vecCopy1Files; // stores the filenames in folder "copy1"
    std::vector<std::string> vecCopy2Files; // stores the filenames in folder "copy2"
    std::vector<std::string> vecCopy3Files; // stores the filenames in folder "copy3"

} HttpObject;


-------------------------------------------------------------------------------------------
---

typedef struct ThreadHandler
{
       int threadID;
       int numberOfThreads;
       bool NFlag;
       pthread_t thread;
       HttpObject message;

}ThreadHandler;


-------------------------------------------------------------------------------------------
---

typedef struct FileHandler
{
       bool doesExist;
       bool hasReadPermission;
       bool hasWritePermission;
       bool isLocked;
       char filePathAndName[FILEPATH_SIZE];
```

```
}FileHandler;
```

The data that is in these structs will be reused throughout this assignment to minimize repeating data, we used these structs to organize information. The newly added booleans help with the -r and -N flags to figure out if the user inputted these commands in the command line.

Part 2: Implement Server Socket with Thread Creation and Flag Checking in Main

The data that needs to be added to create a successful server and client socket are these:

```
HttpObject msg;
msg.rFlag            = false;
msg.NFlag            = false;
msg.port             = DEFAULT_SERVER_PORT;
msg.numberOfThreads  = DEFAULT_THREAD_POOL_SIZE;

int                enable           = 1;
int                serverSocket     = 0;
char*              address          = argv[1];
socklen_t          clientAddressLength;
struct sockaddr_in serverAddress;
struct sockaddr    clientAddress;
ThreadHandler*     pThreadHandler;
```

The variables that access the HttpObject struct will hold the boolean value for flags, port number, and the number of threads. The enable variable will help the setsockopt function. The address variable will hold the value of whatever the user wants to call the http server with.The server_sockd will hold the socket function value. We will use the sockaddr_in to hold the values for the server address and the sockaddr to hold the client address that needs to be accessed. There will also need to be a socklen_t size to hold the correct address length for the client socket to be used.

We allocate memory for the globalRequestQueue here before memset. A memset is used to set the server_addr to the size of the server_addr along with other calls from the struct sockaddr_in. The ReadCommandLine comes after which will be discussed later in Part 2a.

```
globalRequestQueue = new std::queue<int>();
memset(&serverAddress, 0, sizeof(serverAddress));
ReadCommandLine(argc, argv, &msg);

// initialize pThreadHandler after we give ReadCommandLine() the number of threads
pThreadHandler = new ThreadHandler[msg.numberOfThreads];
serverAddress.sin_family      = AF_INET;
```

```
serverAddress.sin_port        = htons(atoi(msg.port));
serverAddress.sin_addr.s_addr = H_GetAddress(address);
```

Prior to connecting to the client server there will be checks to make sure certain system calls are used correctly, if not exit the code:

```
if((server_sockd = socket(...)) <= 0){
       print (error in socket)
       exit code
}
if((ret = setsockopt(...)) < 0){
       print (error in setsockopt)
       exit code
}

if((ret = bind(...) < 0){
       print (error in bind)
       exit code
}

if( (ret = listen(...)) < 0){
       print (error in listen)
       exit code
}
```

Part 2a: Reading User Command Line Function

This assignment has the user create certain flags needed to access commands on the command line. In the main function, we will access this function:

```
ReadCommandLine(argc, argv, &msg);
```

This function serves to read what the user inputs and access the necessary information for the program to follow. The function is declared:

```
void ReadCommandLine(int argc, char **argv, HttpObject *pMessage){
      int result = 0
      ...
}
```

In this function we will check to see if the user inputs any amount of commands:

```
if(argc == 1){
      print error("this is how you use this command")
      write(...)
      exit(...)
}
else if(argv > 2){
      Given address = argv[2]
      Check if valid input of threads is given
            If not exit
}
```

After checking the statements we will parse through the user inputs to see what is being asked: We will be using the **getopt(...)** function call to read what the user inputs on the command line and access the necessary information when those inputs are called. We will also use a switch case to determine what the next course of action to take when a desired input is called:

```
while( (result = getopt(argc, argv, ...) ) != -1 ){
      switch (result){
            case 'N':
            {
                  NFlag = true
                  printf (NFlag)
                  for (i; i < strlen(tag); i++){
                        Check that tag is an appropriate number
                        if(tag is not good) exit
                  }
                  threads = atoi(tag)
                  break
            }

            case 'r':
            {
                  rFlag = true
                  printf (rFlag)
```

```
                    break;
                }

                default:
                {
                    warn(error)
                    break;
                }
            }
        }
```

Part 2b: Thread Creation Function

In this assignment, we will also be focusing on creating a thread function that can handle multiple threads. In httpserver.h, we have declared a typedef struct called ThreadHandler that will store the necessary components listed in Part 1.

```
ThreadHandler* pThreadHandler;
```

This is declared before ReadCommandLine(...) to save the amount of Threads needed to be created when the user inputs the desired number.

After the ReadCommandLine(...) is called, we will save an array of threads that is declared as:

```
pThreadHandler = new ThreadHandler[msg.numberOfThreads];
```

We will initialize each thread with a function we created called:

```
Thread_Init(pThreadHandler, &msg);
```

Within this function:

```
void Thread_Init(ThreadHandler *pThreadHandler, HttpObject *pMessage){
    ...
}
```

We will initialize the threads with the given values from the command line. A temporary thread handler will be created in order to push values into the main ThreadHandler array after initializing each thread.

```
ThreadHandler tempThreadHandler;
for(i = 0; i < numberOfThreads; i++){
      tempThreadHandler.threadID
      tempThreadHandler.message.NFlag
      tempThreadHandler.message.rFlag
      tempThreadHandler.numberofThreads
}

pThreadHandler[i] = tempThreadHandler;
```

Once all the threads created are initialized, we will use a function called pthread_create to formally create the threads.

```
if(pthread_create(thread, NULL, HandleRequests , (void*)thread == 0){
      printf (success in creating a thread)
}
else{
      printf(error in creating a thread)
}
```

The code above will create threads that will be available to be used but the next question is what is being passed into these threads and when will they be accessed? The function that is created to handle the thread locks and multithreading is in:

```
void *HandleRequest(void *pData)
```

Part 2c: HandleRequest Function with GlobalRequestQueue and Queue Locks

Before HandleRequest is mentioned, there are other pthread calls needed for this function to work. A global pthread condition and pthread mutex are needed for the locks that will be called in the HandleRequest function. A global queue is also used to maintain the multithreading aspect of this program.

```
pthread_cond_t globalConditionVariable =
PTHREAD_COND_INITIALIZER
pthread_mutex_t globalRecursiveLock = PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP
std::queue<int>* globalRequestQueue
```

The condition allows our threads to wait for some condition to occur which is important in handling locks. The mutex is used as a recursive lock for the global Queue.

In the main function we will initialize the globalRequestQueue:

```
globalRequestQueue = new std::queue<int>();
```

Now that we have all the necessary components for the HandleRequest function we will dissect the order of the function to make it thread safe.

```
void *HandleRequests(void *pData){
    ...
}
```

In the HandleRequest function, we will access the ThreadHandler values to use the clientSocket values:

```
ThreadHandler* pThreadHandler = (ThreadHandler *) pData
pThreadHandler->message.clientSocket = 0
```

After these declarations, we will run a while loop that will check the queue and help lock the critical section:

```
while (true){
    pthread_mutex_lock(&globalRecursiveLock)

    while(globalRequestQueue is empty, sleep threads){
        printf(a thread is waiting)
        pthread_cond_wait(&globalConditionVariable, &globalRecursiveLock)
        pThreadHandler.clientSocket = globalRequestQueue.front(value)
```

```
            globalRequestQueue.pop()
        }

        pthread_mutex_unlock(&globalRecursiveLock)
        ...
    }
```

The information after the lock will just be to access all the functions for the httpserver requests and responses by passing in the threads.

```
    ReadHTTPRequest(&pThreadHandler->message)
    ProcessHTTPRequest(...)
    ConstructHTTPResponse(...)
    SendHTTPResponse(...)
    close(...)
```

Part 2d: Scanning Existing Files and Pushing Locks into the Map

The last implementations that are included within the main function are:

```
    ScanExistingFiles(...)
    PushLocksToMap(...)
```

These two functions will look through the files that exist already and apply a lock in each of them. It is important to notice that these functions are called before the creation of threads. The reason to do this is to optimize the amount of locking that is needed to be done within the program. If we do it after the threads are created, then multiple locks are needed to be called in each file that wants to be added into the map.

This functions checks if files that aren't on the vector when the server starts up, push them onto the vector to be added with a lock later.

```
    void ScanExistingFiles(HttpObject* pMessage){
        DIR* directory
        Struct dirent* entry
        const char* folder[3] = {copy1,copy2,copy3}

        for(i = 0; i < folders; i++){
            if(directory.folder != NULL){
```

```
                while(entry != NULL){
                     if(copy1){
                          Push file onto the vector
                     }else if(copy2){
                          Push file onto the vector
                     }else if(copy3){
                          Push file onto the vector
                     }else{
                          print error
                     }
                }
           }else if{directory.file != NULL){
                Push file onto the vector
           }
}
```

After the files are added to the map, this function will add a lock to each file to be accessed later in the program.

```
void PushLocksToMap(HttpObject* pMessage){

     if(redundancy is active){
          for(every file in copy1){
               Add a lock to each file
          }
          for(every file in copy2){
               Add a lock to each file
          }
          for(every file in copy3){
               Add a lock to each file
          }
     }else if(no redundancy){
          Add a lock to the file
     }
}
```

Part 3: Redundancy Application

The last part that this assignment asks for is to check that in three different folders there will be three exact copies of a file in each of them. There is a need to have locks for this implementation because we want the program to access each file synchronously and without any race conditions. A method we are thinking of using is a dictionary, a map, to create and store the value for our files.

We will first create a map that holds the filename and the mutex lock in its dictionary:

```
std::map<std::string, pthread_mutex_t> globalMap;
```

The two main functions we added for the redundancy aspect to this program is to check if the rFlag is true and then discern what action to take after that. We created two new functions that will handle the rFlag call and implement file locking accordingly.

```
void HandlePutRFlag(HttpObject* pMessage){
      ...
}

void HandleGetRFlag(HttpObject* pMessage){
      ...
}
```

These functions are called in SendHTTPResponse(...) through H_Call_Method(...).

Both Put and Get will initialize their own folders and files that they will be accessing in the Filehandler struct.

```
Filehandler folder1;
Filehandler folder2;
Filehandler folder3;

FileHandler file1;
FileHandler file2;
FileHandler file3;
```

Both functions will need to get the right path to print into three different folders. To access these, we chose to use sprintf(...) to append the copy folders with the filename.

```
sprintf(folder1.filePathAndName, "%s", (char*)"copy1");
sprintf(folder2.filePathAndName, "%s", (char*)"copy2");
sprintf(folder3.filePathAndName, "%s", (char*)"copy3");
```

```
sprintf(file1.filePathAndName, ... , “copy1”,
pMessage->fileName)
sprintf(file2.filePathAndName, ... , “copy2”,
pMessage->fileName)
sprintf(file3.filePathAndName, ... , “copy3”,
pMessage->fileName)
```

We gave Put files the permissions to write and the Get files the permissions to read

Put	Get
file1.write	file1.read
file2.write	file2.read
file3.write	file3.read

Part 3a: Helper Functions/Error Checking for HandlePutrFlag & HandleGetrFlag
Before I explain in detail how HandlePutrFlag and HandleGetrFlag works, we must go over the extra error checking and helper functions that help keep these functions file safe.

These two overloaded functions help distinguish which files to check if and when the rFlag is raised. The purpose of these two similar helper functions is to check if a filename is within the map.

```
//For Redundancy
bool H_IsFilenameInMap(FileHandler* pFile){
     use globalMap dictionary to access filePathAndName
     ...
}

//For no Redundancy
bool H_IsFilenameInMap(HttpObject* pMessage){
     use globalMap dictionary to access fileName
     ...
}
```

As you can see these two helper functions are called in two separate locations but are accessed accordingly when the type of data is passed through the parameters.

This Helper function will check and provide a file to have read access if it exists. If it doesn't it will print out the according error checking path and give it no read permission. This function is included in the HandleGetrFlag.

```
void H_SetRedundancyGETFileExistenceAndReadPermission(FileHandler* pFileHandler){
      if(file exists){
            allow file to access(read)
            if(error){ permission denied }
      }else{
            file doesn't exist
      }
      File has read permission
}
```

This Helper function will check and provide the file to have write access if it exists. If it doesn't it will print out the according error checking path and give it no write access. This function is included in the HandlePutrFlag.

```
void H_SetRedundancyPUTFileExistenceAndWritePermission(FileHandler* pFileHandler){
      if(file exists){
            allow file to access(write)
            if(error){ permission denied }
      }else{
            file doesn't exist
      }
      File has write permission
}
```

This Helper function will check to see if the folder and file exists. If they do and the filename is not in the map, create a lock for that file that existed already.

```
void H_RedundancyTryPushingFileLockMap(...){
      if(we have folder read permission)
            if(file exists)
                  if(filename is not in map)
                        Push filelock into map
}
```

This is a simple helper function that just checks if the file exists.

```
void H_CheckRedundancyFileExists(...){
      if(files exists){true}
            Else {false}
```

```
}
```

This function will go through and check to see if files have no read access, and if it doesn't it will return an error message. If ⅔ of the files don't have read permission, send a response and exit the function.

```
void SendResponseBasedOnGETRedundancyFileReadPermission(...){
      if(!file1.Read && !file2.Read){error}
      else if(!file2.Read && !file3.Read){error}
      else if(!file1.Read && !file3.Read){error}
}
```

This function will go through and check that if ⅔ of the files don't exist, it will send an error.

```
void SendResponseBasedOnGETRedundancyFileExistence(...){
      if(!file1.exist && !file2.exist) {error}
      ...file2 & file3 {error}
      ...file1 & file3 {error}
}
```

This function will go through all the errors that could occur when the files have no write access and fail the request.

```
void SendResponseBasedOnPUTRedundancyFileWritePermission(...){
      if(!file1.Write && !file2.Write){error}
      ...(!file2 && !file3){error}
      ...(!file1 && !file3){error}

}
```

This function is another error checking to see if the folder (directory) is accessed correctly. If ⅔ of the folders don't have read permissions, then send an error.

```
void SendResponseBasedOnRedundancyFolderPermission(...){
      if(corresponding folders don't work){
            print error
      }
}
```

This function checks to see if ⅔ of the folder don't exist, it will send an error.

```
void SendResponseBasedOnRedundancyFolderExistence(...){
```

```
        if(!folder1.exist && !folder2.exist){error}
        ...(!folder2 && !folder3){error}
        ...(!folder1 && !folder3){error}
    }
```

This function checks to see if the all three files are different from each other, if it is, send an error.

```
    void SendResponseBasedOnRedundancyGETFileDifference(...){
        if(!files.exist && !files.read && files.readandPermission){error}
        ...(All combinations for failure)
    }
```

This function gives the folders read permissions when accessing the folders. This function also accesses the dirent.h library to use DIR.

```
    void H_SetRedundancyFolderPermission(...){
        if(folder exists){
            allow access(...)
        }
    }
```

This function checks and locks existing files that are shown in the globalmap.

```
    void LockExistingFiles(...){
        if(file1.exist){Lock file1}
        ...file2
        ...file3
    }
```

This function checks and unlocks existing files that are shown in the globalmap.

```
    void UnlockExistingFiles(...){
        if(file1.exist){Unlock file1}
        ...file2
        ...file3
    }
```

This function is used for a Get request with redundancy. The point of this function is to check if the files are identical and return true if contents and or value are the same.

```
bool H_CheckIfGETRedundancyFilesAreIdentical(...){
      if(checks if files exist){...}
      if(checks if files have the same content length){
            if(not same){close files}
      }
      if(checks if both file buffers are the same){
            if(not same){close files}
      }
}
```

This function is finally called in the HandleGetRFlag after all the checks and locks are in place. This prints out the desired output when everything succeeds.

```
void FinalizeHandleGetRFlag(...){
      if(Getfiles.identical(file1, file2)){
            print response
            Call HandleGET
      }
      ...file2 & file3
      ...file1 & file3

}
```

Part 3b: HandlePutRFlag Function

As we step into this function we see that we need a lock for each file that is first accessed. First we check to see if the file is already in the Map.

```
H_RedundancyTryPushingFileLockToMap(folder1, file1)
H_RedundancyTryPushingFileLockToMap(folder2, file2)
H_RedundancyTryPushingFileLockToMap(folder3, file3)
```

Next we give the existing files write permissions for the file that are about to be accessed. This function is stated in Part 3a.

```
H_SetRedundancyPUTFileExistenceAndWritePermission(file1);
H_SetRedundancyPUTFileExistenceAndWritePermission(file2);
H_SetRedundancyPUTFileExistenceAndWritePermission(file3);
```

The reason why we don't need to give these files locks is because in the beginning when we accessed the ScanExistingFile(...) we made sure to give each file a lock already and put it in a map.

After giving the files and folders the permissions to read or write, check and see if the folders exist and are accessible. If not exit the program and return an error. Then lock the files that are selected before writing or rewriting the new files into the folders.

```
SendResponseBasedOnRedundancyFolderExistence(...)

SendResponseBasedOnRedundancyFolderPermission(...)

SendResponseBasedOnPUTRedundancyFileWritePermission(...)

pthread_mutex_lock(globalMap[file1])
... file2
... file3
```

Now that the files are locked, we enter the critical sections for the files that we are going to access. With all the checking that happened above, we will open the files and provide a truncate if the file exists already. If it isn't we will create a new file in each folder.

```
int fd1 = open(file1, O_CREAT | O_WRONLY | O_TRUNC, ...)
int fd2 = open(file2, O_CREAT | O_WRONLY | O_TRUNC, ...)
int fd3 = open(file3, O_CREAT | O_WRONLY | O_TRUNC, ...)
```

Another file checking is added just in case something bad happened when creating the new files.

```
if(fd1 < 0){close fd1}
... fd2
... fd3
```

If the file had empty bytes close the file, unlock it, and give it a created message.

```
if(file is empty){
      close (fd1)
      ... fd2
      ... fd3

      pthread_mutex_unlock(globalMap[file1])
      ... file2
      ... file3

      print created status
}
```

If the file is being written we have to track the buffer and write it to each file that is being written into.

```
Byteswritten1 = write(...)
Byteswritten2 = write(...)
Byteswritten3 = write(...)
```

Another important thing to catch is that if the files have an error that wasn't checked we will need to exchange the bytes that were already written with the tempBytes that were accessed.

```
if(tempByWritten1 != -1){bytesWritten = tempByWritten1}
if(tempByWritten2 != -1){bytesWritten = tempByWritten2}
if(tempByWritten3 != -1){bytesWritten = tempByWritten3}
```

If there was an error, close the file, unlock the file, and print out an error.

```
if(all bytes are < 0){close all files & unlock files}
```

From here on out it's the same process as handling the Get function but with three different repeated steps to catch each file that needs to be written. A common practice in handling file locks is that if there is an error you need to close the file and unlock it to prevent having multiple locks accessed since the file is broken already. If the file is successfully rewritten, then close the file and unlock them. Print out a created status code.

```
Close(files)
UnlockExistingFiles(...)
print created status code
```

Part 3c: HandleGetRFlag Function

As we step into the Get function we see that we need a lock for each file that is first accessed. First we check to see if the file is already in the Map.

```
H_RedundancyTryPushingFileLockToMap(folder1, file1)
H_RedundancyTryPushingFileLockToMap(folder2, file2)
H_RedundancyTryPushingFileLockToMap(folder3, file3)
```

Next we give the existing files read permissions for the file that are about to be accessed. This function is stated in Part 3a.

```
H_SetRedundancyGETFileExistenceAndReadPermission(file1);
H_SetRedundancyGETFileExistenceAndReadPermission(file2);
H_SetRedundancyGETFileExistenceAndReadPermission(file3);
```

After giving the files and folders the permissions to read or write, check and see if the folders exist and are accessible. If not exit the program and return an error. We also need to check to see if the files are different within the folders. If they are identical it will return the corresponding answer that is needed for the answer. Then lock the files that are selected before writing or rewriting the new files into the folders. More details of each function are listed in Part 3a.

```
SendResponseBasedOnRedundancyFolderExistence(...)

SendResponseBasedOnRedundancyFolderPermission(...)

SendResponseBasedOnGETRedundancyFileExistence(...)

SendResponseBasedOnRedundancyGETFileDifference(...)

SendResponseBasedOnGETRedundancyFileReadPermission(...)
```

When all the error checking is processed, lock each file and access the HandleGet() function.

```
LockExistingFiles(...)
FinalizeHandleGetRFlag(...)
UnlockExistingFiles(...)
```

Part 4: Functions for HTTPServer

Creating the process and the execution of the HTTP Server, we will need to identify the “name” of the address required for an IPv4 address which is represented in sockaddr_in. We will resolve this in this function:

```
unsigned long GetAddress(char* name)
```

In this function, provided by Daniel Alves’s sample code, we will use the addrinfo struct to help contain and access the desired address. The getaddrinfo and freeaddrinfo functions will be used as well:

```
if(getaddrinfo(...) != 0 || info == NULL){
       print error message
       write(...)
       exit(1)
}
```

After checking if there is an address info error, we store the address information in res:

```
unsigned long res
res = ((struct sockaddr_in*) info → ai_addr) → sin_addr.s_addr
freeaddrinfo(info)
return res
```

When the server sockets are all set up the next thing is to connect the client and server together. The entire process and execution of HTTP Server consists of these five high level steps:

1. Accept Connection – creates a connection between the client and the server
2. Read http message – parses through the clients request and checks if it is valid
3. Process request – stores the information in a struct
4. Construct response – constructs a response message based on the validity of the request
5. Send message – sends the response back to the client

In order to have these information pass accordingly between the client and the server, these 5 functions are organized to do that.

Part 4a: AcceptConnection

```
void AcceptConnection( HttpObject* pMessage, struct sockaddr* client_addr, socklen_t*
client_addrlen, int server_sockd);
```

Create a client socket and check for errors.
The system call accept(...) is needed to accept the connection between the client socket and server, which is implemented in this function.

```
client_socket = accept(...)
```

We check if there is an error with the client_socket when it's value is less than 0.

```
if(client_socket < 0){
        Contentlength = 0
        print error message
}
```

Part 4b: Read HTTP Message

This function will check and store the method name, file name, and http version. If the request is a PUT then it also stores content length.

```
void ReadHTTPRequest( HttpObject* pMessage);
```

The system call recv(...) is used to check if the message is received into the socket. We also make sure that the bytes read in this function is ssize_t.

```
ssize_t bytesRead = 0

if((bytesRead = recv(...)) < 0){
        print error (in recv(...))
}else if (bytesRead == 0){
        print error (client is disconnected)
}
```

To read the client that was requested, we read the information with the tool sscanf(...) and store them.

```
sscanf(requestBuffer, "%s %s %s", method, tempBuffer, httpversion)
```

With this we parse the information, but we need to check the buffer to see what is being read.

```
if(buffer[0] == '/'){
```

```
        move information to the right location
    }

    if(check if tempBuffer == 10 ascii characters){
        copy filename and the buffer
    }else{
        Print error with error code
    }
```

Now we check to see if the put method was called:

```
    if(method == put){
        store the content length
    }
```

Part 4c: Process Request

This function rescans the stored content to make sure all information is stored correctly and is valid.

```
void ProcessHTTPRequest( HttpObject* pMessage);
```

The information that needs to be checked is if the method name is valid, if the filename is valid, and if the HTTP version is valid.

```
    if(method name is the method name){
        sscanf(error, …, messageStatus)
        print error code
        Send a response to the server
    }

    if(filename is the filename){
        sscanf(error, …, messageStatus)
        print error code
        Send a response to the server
    }

    if(httpversion is the httpversion){
        if( (compare string == httpversion)  !=  0){
            sscanf(error, …, messageStatus)
            print error code
            Send a response to the server
        }
    }
```

Lastly, we look for the GET request:

```
    if(method == GET) {
        if(file exist but empty){
            print success code
```

```
                sscanf(good, …, statusMessage)
        }else{
                print error code
                sscanf(Missing, …, statusMessage)
        }

        sprintf(reponse, Contentlength, httpversion, statusCode, statusMessage,
        content length)
}
```

Part 4d: Construct Response

This function will check that if no errors were thrown, a valid status code and status text are set according to the request.

```
void ConstructHTTPResponse( HttpObject* pMessage);
```

The PUT and GET are checked first:

```
if(method == PUT){
        sscanf(Created, …, statusMessage)
}else if (method == GET){
        sscanf(OK , ..., statusMessage)
}else{
        sscanf(Error, …, statusMessage)
        print error code
        write out the response for the error
}
```

Now we check to see if the file exists and if it does grant access permission to read or write accordingly:

```
if(fileExists){
        sscanf(OK, …, statusMessage)
        print success code
}

if(method == GET){
        ret = access(fileName, file_flag)
        if(file not in directory){
                print error code
                sscanf(Error, …, statusMessage)
                send response to the server
        }else if(file access in wrong area){
                print error code
                sscanf(Error, …, statusMessage)
                send response to the server
        }

        ret = access(fileName, read_flag)
```

```
        if(read is not ok){
                print error code
                sscanf(Error, …, statusMessage)
                send response to the server
        }

        ret = access(fileName, write_flag)

        if(write is not ok){
                print error code
                sscanf(Error, …, statusMessage)
                send response to the server
        }
}
```

Part 4e: Send Response

This function handles the client requests (PUT & GET) and sends the final response as the final step.

```
void SendHTTPResponse( HttpObject* pMessage);
```

To check if there are any errors or problems, we first check the string method is PUT to handle the request:

```
if(method == PUT){
        open file
        if(file doesn't exist){
                print error
                close file
        }

        if(file is empty){
                print success code
                sscanf(create, …, statusMessage)
                close file
        }
        ...
}
```

Now we read the buffer and write out what is given from the file. We check to see what is in the file or exit if there is an error:

```
bytesRead = read(...)

if(bytesRead is wrong){
        print error
}

ssize_t bytesWritten = write(...)
```

```
ssize_t totalBytes = bytesWritten

while(totalBytes < contentlength){
	if(bytesWritten is wrong){
		print error
	}else{
		bytesRead = read(...)
		bytesWritten = write(...)
		totalBytes = totalBytes + bytesWritten
	}
}

close(file);
```

We check if the request is GET and we do the same thing as PUT but we open the file and just read it:

```
if(method == GET){
	if(file exist){
		print success code
	}

	open file(READ)
	if(file has errors){
		close file
		print error code
	}else{
		ssize_t bytesRead = read(...)
		if(bytesRead has error){
			print error code
		}
		ssize_t totalBytes = 0
		ssize_t bytesWritten = 0
		while(totalBytes < contentlength){
			bytesWritten = write(...)
			bytesRead = read(...)
			totalBytes = totalBytes + bytesWritten
		}
	}
	close(file)
}
```

Part 5: Helper Functions for HTTPServer

These helper functions will be used to help organize the 5 main functions needed to run the client and server:

Handles functionality when clients request is GET:

```
void HandleGET( HttpObject* pMessage);
```

This helper function is implemented in `void SendHTTPResponse( HttpObject* pMessage)` to organize the read functions when GET is called in method.

Handles functionality when clients request is PUT:

```
void HandlePUT( HttpObject* pMessage);
```

This helper function is implemented in `void SendHTTPResponse( HttpObject* pMessage)` to organize the read and write functions when PUT is called in method.

Checks the validity of the client's file name:

```
int H_CheckFileName(char* name);
```

This helper function will loop through the given filename and check to see if it is a viable filename. This is achieved by checking each character in the filename to make sure each character is a legal character.

Checks the validity of the client's file method name. A valid method name will only be PUT or GET:

```
int H_CheckMethod( HttpObject* pMessage);
```

This helper function will compare the string in the method to see if the string value is a PUT or GET. If it is PUT return 1 or if it is GET return 2.

Checks the validity of the clients http version. It should only ever be “HTTP/1.1” or else an error is thrown:

```
void H_CheckHTTPVersion( HttpObject* pMessage);
```

This helper function is implemented in `void ProcessHTTPRequest( HttpObject* pMessage)`. This will compare strings if HTTP/1.1 is part of the version. If it is not it will print out an error code with a bad request.

Calls either HandlePUT() or HandleGET():

```
void H_CallMethod( HttpObject* pMessage);
```

This helper function is called in `void SendHTTPResponse( HttpObject* pMessage)`. This just checks if the string is PUT or GET and calls the respected function. If there is an error, send an error code and a bad request.

Parses through the clients request and stores the content length:

```
void H_StoreContentLength( HttpObject* pMessage, char requestBuffer[]);
```

This helper function is called in `void ReadHTTPRequest( HttpObject* pMessage)`. This function uses the strtok() function to help store the content length. This function also checks the buffer and stores it in an array.

This function is used if a file descriptor throws an error. The HTTP status code error is set automatically based on the errno:

```
void H_SetStatusOnError( HttpObject* pMessage);
```

This helper function uses a switch case to organize the desired HTTP status code errors.

Different from H_StoreContentLength(),This function H_ReadContentLength() analyzes the file pointed to by stat() and returns the size by using .st_size. This is stored in the content length for future reference:

```
void H_ReadContentLength( HttpObject* pMessage);
```

Simply checks if the file exists. Makes use of stdbool.h:

```
bool H_CheckFileExists( HttpObject* pMessage);
```

Sets the response that will eventually be sent to the client. This is called frequently throughout the program:

```
void H_SetResponse( HttpObject* pMessage);
```

Sends the response to the client:

```
void H_SendResponse( HttpObject* pMessage);
```

This helper function uses the send() and write() function to send the desired response in the server. The send() value will be stored in ssize_t while the write() function will be stored in size_t.

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.

- Multiple threads reading and writing to the same file could cause race conditions
- For example: T1 could truncate the file with an empty document, while T2 was reading from it causing T2 to read nothing in the middle of a function.

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

- As you increase the number of threads it will allow more jobs to be picked up by each thread. The performance/scalability, however, does not necessarily get better. The program would be concurrent but not parallel because the threads cannot access shared resources if they are locked. If the threads are trying to access the same shared resources, they will have to wait for one another to complete their task.

TESTING

Your design document is also where you'll describe the testing you did on your program and answer any short questions the assignment might ask. The testing can be unit testing (testing of individual functions or smaller pieces of the program) or whole-system testing, which involves running your code in particular scenarios. In particular, we want you to describe testing that you did for:

- Checking that multiple threads can run and process requests concurrently. Specifically, when you increase the number of threads, does this lead to processing simultaneous requests faster?

- When you increase the number of threads, processing concurrent requests will be faster, however only if each thread is working on a request. If the number of threads do not exceed the number of requests, then processing simultaneous requests should be faster.

- 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?

This is mostly a concern for a GET request as we need to check if the files are different

- If ⅔ files do not have read access and it is a GET request, return 403 Forbidden error
- If ⅔ files do not have write access and it is a PUT request, return 403 Forbidden error
- If ⅔ files do not exist and it is a GET request, return 404 Not Found Error
- If ⅔ files do not exist and it is a PUT request, that is fine just create the files
- If 3/3 files are different from each in the same way or different ways, return 500 error
- If at least ⅔ files are the same in existence, length, and content and it is a GET request, return the contents of one of the 2 files
- If 3/3 files are the same in existence, length, and content and it is a GET request, return the contents of one of the 3 files
- If a "copyX" is removed and the file is missing or if the file is missing from "copyX", the program will still check the other two folders to see if it exists. If it does it will still return the contents from one of the two files that are identical.

OUR TESTING

We tested our code with the sample that was given on the piazza post 330 and more. Along with testing the PUT and GET functions that were provided, we made sure that the server could continue to process multiple requests transitioning from one process to the next without crashing. We applied about 1000 concurrent requests all with Large Binary files, running with 4 threads and redundancy on. This took a long time to process but at the end of it, the program could continue to run after the 1000 requests. We also removed permission with “chmod 000 filename” or “chmod 000 foldername” in each folder and file to test to see if the program would return the right error message. The error message is determined if ⅔ of the file has the concurrent error message. For example if 2 of the 3 files are not found it will return 404 Not Found, or if 2 of the 3 files were given no permission, it will return 403 Forbidden. We also compared files with the “diff” system call to see if the two files were identical or not. All of these tests passed.