Httpserver Asgn3

Assignment Questions:

1. How would this backup/recovery functionality be useful in real-world scenarios?
    a. One way this is helpful in real-world scenarios is when you break your laptop and have to get a new one. If you hadn't backed-up your files at all or recently, then you would have lost them all or have really old (maybe irrelevant) files. Another scenario could be if you accidentally do a PUT request for the wrong file. Then, it would be useful to have this recovery functionality especially with a recent previous backup.

Testing

For our testing, we used whole-system testing. We ran our code using different curl commands under different scenarios. We created multiple backups using put requests throughout so that some of the files would vary throughout the backup folders and then called recovery making sure that the files in our server directory match the files in our most recent backup folder. We also changed the file permissions inside our backups and also changed the permissions for our backup folders to see if we would get a correct response. Throughout our testing we a

*the changes we made for assignment 3 are inside of the HTTPParse class, specifically the `HandleBackups(), IsProgramFile(), InBackupDirectory(), GetNewestBackup(),HandleFolderRecovery`

The HTTPParse class:

The main goal of this class is to handle the parsing and sorting of an HTTP header and body. ServerManager will be creating HTTPParse objects to parse requests sent to the server, then use the data parsed by HTTPParse to respond to these requests.

Private Members:

- index: An integer representing the current index in our character stream of the request. Utilized by GetWord().
- request: The string HTTP request from the client. This can either be a header or a body
- requestLength: The string length of request
- requestType: A string representing the type of HTTP request given
- filename: The filename given in the request header
- contentLength: The content length given in the request header
- fd[3]: The file descriptors used when GETting or PUTting
- dr: the DIR used for a GET list request

- bytesUsed[3]: The total number of bytes got or put in each of the files specified by fd
- correctFileIndex: The index of a valid file when doing a GET request
- char* GetWord()
- bool IsValidName(char*)
- int SetupGetRequest()
- int SetupGetListRequest();

Public Members:

- body: The buffer for the data send in the HTTP request's body
- HTTPParse()
- ~HTTPParse()
- int ParseRequestHeader(char*)
- int PutAction(int);
- int GetAction();
- int GetRequestType()
- int ContentLength()
- char* GetFilename()
- int HandleFolderRecovery(char *)
- int HandleFolderRecoveryNewest()
- bool IsProgramFile(const char*)
- int FolderHasPermissions(char*)

int HTTPParse::GetContentLength():

This function returns the content length of a file.

char* HTTPParse::GetWord():

This function will return the next word in the stream of *request*. It uses *index* as it's marker for where the last GetWord() call left off in *request*. A word is defined to be a substring of characters, starting from *index*, up to the next space or newline. When the next word is returned, the index is set to the beginning of the next word.

bool HTTPParse::IsValidName(char*):

This function will look at a string and determine if it is a valid filename for our server. A file needs to have an alphanumeric, 10 character name. If it is, return true, otherwise return false.

int HTTPParse::SetupGetRequest():

This function has a few purposes, all of which occur immediately after parsing the header and finding out that the server has been given a GET request. Universally, this function opens the

specified file for reading to find out it's content length, which is given to contentLength. If redundancy is enabled however, first the function determines the state of the 3 copies of the file on the server. If at least 2 files have the same content, it sets correctFileIndex to one of the two, and finds the content length of that file to put into contentLength. If each of the files contains different content, it returns 500. If for some reason while reading these files the server cannot find the file, it returns 404, and if the file specified exists but cannot be opened it returns 403.

int HTTPParse::PutAction(int s):

This function is designed to finalize the processes of a PUT request. It is called from ServerConnection from inside of a while loop and works to add whatever we receive into the file corresponding to the filename the client has sent us. It will save a file of a specified name with specified data from *body* into the file and increment our counter of bytesUsed. We check for errors and if there aren't any we return 201. (This code will be interpreted and handled by ServerConnection). If redundancy is enabled, it will put the information in 3 files under "copy" subdirectories.

int HTTPParse::PutAction(int s)

```
int fileCount = 1
if (GlobalServerInfo.redundancy)
     fileCount = 3
for (int i = 0, i < fileCount; i++)
     if (fd < 0)
          char* filepath = "copy" + i + "/"
          if (GlobalServerInfo.redundancy)
               filepath = filepath + filename
          else
               filepath = filename
          fd = open(filepath, write only | create if it does
     not exist | remove any data inside)
          If (fd < 0)
               return 403
     Int n = write(fd[i], body, s)
     bytesUsed[i]+=n
     if bytesUsed[i] >= contentLength
          if (close(fd[i]) < 0)
               warn
```

```
                    return 500
        return 201
```

int HTTPParse::GetAction():

This function will put the contents of the file specified by filename SIZE bytes at a time into *body*, where it will be used by serverconnection to send to the client. If redundancy is enabled, it gets the file at the *correctFileIndex*. It returns the number of bytes read this time (no greater than SIZE)

int HTTPParse::GetAction()

```
        if (fd[correctFileIndex] < 0)
              char* filepath = "copy" + correctFileIndex + "/"
              if (globalServerInfo.redundancy)
                    filepath = filepath + filename
              else
                    filepath = filename
              fd[correctFileIndex] = open(filepath, readonly)
              if (fd[correctFileIndex] < 0)
                    return -1
        int n = read(fd[correctFileIndex], body, SIZE)
        bytesUsed[correctFileIndex] += n
        if (bytesUsed[correctFileIndex] >= contentLength)
              if (close(fd[correctFileIndex]) < 0)
                    return -1
        return n
```

HTTPParse::HTTPParse()

This is the constructor for HTTPParse. It initializes the member variables of the class.

HTTPParse::HTTPParse()

index = 0

requestLength = 0

contentLength = -1

requestType = NULL

filename = NULL

contentLength = 0

for (int i = 0; i < 3; i++)

fd[i] = -1

bytesUsed[i] = 0

correctFileIndex = 0

HTTPParse::~HTTPParse()

This is the destructor for HTTPParse. It deletes *requestType* and *filename*.

int HTTPParse::ParseRequestHeader(char*)

This function parses an HTTP header. It will return a code to be handled by ServerManager once it completes its task. If this is the first iteration of a PUT request, it will return 0 to identify that ServerManager is still waiting for the body of the HTTP request.

int HTTPParse::ParseRequestHeader(char* r)

```
index = 0
request = r
requestLength = length of request
requestType = GetWord()
filename = GetWord()
if (filename[0] == '/')
     remove the '/'
else
     return 400
if (GetRequestType() == 0)
     bool isValidFunctionality = length of filename == 1 &&
(filename == "r" || filename == "l" || filename == "b")
     if (length of filename > 1)
          isValidFunctionality = isValidFunctionality ||
(filename[0] == 'r' && filename[1] == '/')
     if (!IsValidFilename && !isValidFunctionality)
          return 400
if (GetRequestType() == 1)
     if (!IsValidFilename())
          return 400

char httpTitle = GetWord()
if (httpTitle != "HTTP/1.1")
     return 400
```

```
delete httpTitle
if (GetRequestType() == -1)
     return 500
if (GetRequestType() == 0)
     int messageCode
     if (filename == "r")
          messageCode = HandleFolderRecoveryNewest()
     else if (filename == "b")
          messageCode == HandleBackups(filename)
     else if (filename == "l")
          messageCode = SetupGetListRequest()
     else
          if (filename[0] == 'r' && filename[1] == '/')
               char newFilename = filename
               remove the r and / from the front
               if (length of newFilename < 1)
                    return 400
               long temp = (long) newFilename
               messageCode = HandleFolderRecovery(temp)
if (GetRequestType() == 1)
     char* contentLengthHeader
     while(contentLengthHeader != "Content-Length:" && index <
requestLength)
          contentLengthHeader = GetWord()
     if (index < requestLength)
          contentLength = (int)GetWord()
     else
          contentLength = -1
return 0
```

int HTTPParse::HandleBackups(char* filename):

This function handles the backup of all the files in our current directory that aren't part of our program (files like httpserver). We created a char array with the names of the files that are used for our program and don't need to backup. First, the function starts by getting the time using `time_t`. Then

we create a string with the backup folder name and create the directory of this name. Then, we open the current directory using the opendir() from the man pages and then use readdir() from the man pages in order to loop through the files in our current directory. As we loop through these files using a while loop, we then use our `IsProgramFile` function to check whether or not this is a file that is part of our program and therefore does not need to be backed up. If it's not a program file, then we create a string with our pathname, which will be something like "`backup-123456789/filename`", and a buffer. We open the file from our current directory. Then, we open (or create if it doesn't exist) a file of the same name in our backup folder using the file path string. Using the file descriptors, we read from the file in our current directory and write to the file in our backup folder using the file path. We do this for each file in our current directory as long as we are looping through our current directory.

```
time_t seconds;
seconds = time(NULL);
std::string folderName = "backup-" + std::to_string(seconds);
const char * directory = folderName.c_str();
int checkCreatedDirectory = mkdir(directory,0777);

DIR *openedSuccessfully = opendir(".");
while ((directoryPointer = readdir(openedSuccessfully)) != NULL)
      std::string file = directoryPointer->d_name;
      const char * fileNameStr = file.c_str();
      if (!IsProgramFile(fileNameStr)) {
            std::string file = directoryPointer->d_name;
              std::cout << "[HTTPParse] file: " << file << '\n';
              std::string pathnamestr = folderName + "/" + file;
              std::cout << "[HTTPParse] pathName: " << pathnamestr <<
'\n';
              const char * pathName = pathnamestr.c_str();

              char b[15872];
              memset(b, 0, sizeof b);

              int fd = open(fileNameStr, O_RDONLY);
              If there aren't any errors
                  int backupFd = open(pathName)
```

```
                    if (backupFd < 0) {
                        warn("error opening backup file %s",
pathName);
                    } else {
                        int fileBytesRead = 1;
                        while (fileBytesRead != 0) {
                                fileBytesRead = read(fd, b, sizeof
b);
                                if (fileBytesRead < 0){
                                    warn("%s", fileNameStr);
                                    close(fd);
                                    break;
                                }

                            int writtenSuccessfully =
write(backupFd, b, fileBytesRead);
                            if (writtenSuccessfully < 0) {
                                warn("%s", "write()");
                                break;
                            }
                        }
                    }
                    Close file descriptors
        }
    }
    Close folder file descriptor
    Return 200
```

int HTTPParse::HandleFolderRecoveryNewest():

This function calls HandleFolderRecovery, passing in a value created by GetNewestBackup(), and returning the value returned by HandleFolderRecovery

long HTTPParse::HandleFolderRecovery(char* newestBackupTime):

This function handles recovery for the specified backup time. First we convert newestBackupTime to a string and append it to “backup-”. Once we have our backup folder name, then

we loop through our backup folder and for each file we open our backup file and the same file in our server. Then, using a while loop we read from the backup file and then write to the server file.

```
      contentLength = 0;
   // check if newestBackupTime is 0 then there is no backup folder
   std::string backupFolderNameStr = "backup-" +
std::to_string(newestBackupTime);
   const char * backupFolderName = backupFolderNameStr.c_str();

   if (FolderHasPermissions(backupFolderName) == 1) {
       return 404;
   }

   if (FolderHasPermissions(backupFolderName) == 2) {
       return 403;
   }

   struct dirent *backupDirent;
   DIR* backupFolder = opendir(backupFolderName);
   // check if folder opened successfully
   while ((backupDirent = readdir(backupFolder)) != NULL) {
       char buffer[SIZE];
       strcpy(buffer, backupFolderName);
       strcat(buffer, "/");
       strcat(buffer, backupDirent->d_name);
       int backupFD = open(buffer, O_RDONLY);
       int serverFD = open(backupDirent->d_name, O_CREAT | O_WRONLY |
O_TRUNC, S_IRWXU);
       while (1) {
           memset(buffer, 0, sizeof buffer);
           int readSize = read(backupFD, buffer, SIZE);
           if (readSize < 1) {
               break;
           }
           int writeSize = write(serverFD, buffer, readSize);
```

```
        if (writeSize != readSize) {
            //return 500;
        }


    }
    close(backupFD);
    close(serverFD);
}
Return 200
```

long HTTPParse::GetNewestBackup():

This function loops through our current directory (where our server is) checks to see if a the file/directory at this loop iteration is a backup folder and finds the most recent backup folder created. We do this but getting the one with the highest integer.

```
long HTTPParse::GetNewestBackup() {
   // would be the max
   long newestBackup = 0;

   struct dirent *directoryPointer;
   DIR *openedSuccessfully = opendir(".");
   // check if opened successfully
   while ((directoryPointer = readdir(openedSuccessfully)) != NULL) {
// manpages
        file = directoryPointer->d_name
        const char * isBackupFolder;
        isBackupFolder = strstr (file,"backup-");
        if (isBackupFolder != NULL) {
            std::string str2 = fileStr.substr (7, strlen(file)); // 7
because 'backup-'
            long temp = atol(str2.c_str())
            if (temp > newestBackup) {
                newestBackup = temp
            }
        }
```

```
    }
    return newestBackup;
}
```

bool HTTPParse::InBackupDirectory(const char * f, const char* folder):

This folder takes in the file and back up folder name in order to check whether a file from our server directory is in our backup folder. We do this by using opendir() and readdir() to look through our backup folder and compare the filenames

```
struct dirent *directoryPointer;
    DIR *openedSuccessfully = opendir(folder);
    // check if folder opened successfully
    std::string fileStr = directoryPointer->d_name;
    while ((directoryPointer = readdir(openedSuccessfully)) != NULL) {
        std::string fileStr = directoryPointer->d_name;
        const char * file = fileStr.c_str();
        if (strcmp(f, file) == 0)
            return true;

    }
    return false;
```

int HTTPParse::FolderHasPermissions(const char * f):

This function is used in order to check if our most recent backup folder has permissions or exists and this is for the case of recovery.

```
int HTTPParse::FolderHasPermissions(const char * backup) {
      DIR *openedSuccessfully = opendir(backup);
    if (openedSuccessfully) {
            closedir(openedSuccessfully);
            return 0;
      } else if (ENOENT == errno) {
            return 1;
      } else {
            return 2;
      }
}
```

bool HTTPParse::IsProgramFile(const char * f):

The purpose of this function is to check if the file in our current directory is one of our program files or not. If it's not, then we back up the file. This function loops through our array of program file names and returns true if the filename we are checking is in our array. Otherwise, it returns false.

```
bool HTTPParse::IsProgramFile(const char * f) {
      char* fname = strdup(f);
      if (!IsValidName(fname)) {
            return true;
      }
      free(fname);
   const char * isBackupFolder;
       isBackupFolder = strstr (f,"backup-");
       if (isBackupFolder != NULL) {
           std::cout << "is backup folder\n";
               return true;
       }

   return false;
}
```

int HTTPParse::GetRequestType():

This function returns an integer based on the value of *requestType*. If its NULL, return -1, if its GET, return 0, if its PUT, return 1.

int HTTPParse::GetContentLength():

This function returns the value stored in *contentLength()*.

char* HTTPParse::GetFilename():

This function returns the value stored in *filename*.

int HTTPParse::SetupGetListRequest():

This function sets up the content length for a GET list request. It goes through all the backups and their timestamps, counts the number of bytes needed for the timestamps, and sets *contentLength* to that value.

```
int HTTPParse::SetupGetListRequest()
      struct dirent *de
      DIR *dir = opendir(".");
```

```
        if (dir == NULL)
              return 500
        int accumulator
        char buffer[SIZE]
        while ((de = readdir(dir)) != NULL)
              clear the buffer
              buffer = substring the first 7 char of de.d_name
              if (buffer == "backup")
                    clear the buffer
                    buffer = the substring of de.d_name without the first
  7 chars
                    int nameLength = length of the buffer
                    accumulator += nameLength + 1
        closedir(dir)
        contentLength = accumulator
        return 200
```

int HTTPParse::GetListAction():

This function puts the timestamp of the next backup into the *body*. It returns the number of bytes it reads into *body*, and is called by BeginRecv() in ServerConnection when the server is processing a GET list request.

```
int HTTPParse::GetListAction()
      struct dirent *de
      if (dr == NULL)
            dr = opendir(".")
            if (dr == NULL)
                  return -1
      int nameLength = 0
      clear the body
      de = readdir(dr)
      char buffer[SIZE]
      if (de != NULL)
            buffer = substring the first 7 char of de.d_name
            if (buffer == "backup-")
```

```
                    body = the substring of de.d_name without the first 7
chars
                    body += "\n"
`                   nameLength = length of body
        else
                closedir(dr)
                return 0
        return nameLength
```

The GlobalServerInfo class

This is a static class that will hold information from each thread in a list. The information will pertain to the processes that each thread is currently handling, more specifically, the files that each thread is currently working with. This is to give each thread a way to easily determine when the other threads are completed working with files they are waiting to handle.

Private Members:

- Struct MutexInfo {
  char * filename
  Pthread_mutex_t mutex;
  }
- static vector mutexInfos;

Public Members (all static):

- boolean redundancy
- Int mutexInfosSize
- Bool addMutexInfo(char*)
- static pthread_mutex_t* GetFileMutex(char*);
- static bool MutexInfoExists(char*);
- static void RemoveMutexInfo();

std::vector<GlobalServerInfo::MutexInfo*> GlobalServerInfo::mutexInfos;

- This is a vector of mutexInfos, mutexes for each filename we receive from the client

bool GlobalServerInfo::redundancy = false;

- We use this so that any thread/connection knows whether or not we are using the redundancy functions we have in httpparse.cpp

int GlobalServerInfo::mutexInfosSize = 0;

- Keeps track of how many mutexes we have so far

bool GlobalServerInfo::AddMutexInfo(char* filename)

If the mutex exists, we return false

Otherwise:

Declare a new MutexInfo struct variable named *mutexInfo*

Initialize a mutex using *pthred_mutex_init*(&mutexInfo->mutex, NULL)

Set the filename property of the mutex

Push this mutex onto the vector

Increase the size of *MutexInfoSize* by 1

pthread_mutex_t* GlobalServerInfo::GetFileMutex(char* f)

```
for (int i = 0; i < mutexInfosSize; i++)
      if (mutexInfos[i]->filename == f) // if same filename
            return &mutex[i] -> mutex
return NULL // no mutex for file f
```

bool GlobalServerInfo::MutexInfoExists(char* f)

```
for (int i = 0; i < mutexInfosSize; i++)
      if (mutexInfos[i]->filename == f) // if same filename
            return true
return false // a mutex for this filename does not exist
```

void GlobalServerInfo::RemoveMutexInfo()

```
for (int i=0; i < mutexInfosSize; i++)
      pthread_mutex_destroy(&mutexInfos[i]->mutex);
      delete mutexInfos[i];
      mutexInfosSize--;
   mutexInfos.clear();
```

The ServerConnection class

The purpose of this class will be to handle a connection to the server. It will act independently from all other connections to the server and wait for requests to be sent to it. The processes of this class will be working on their own separate thread from the class's other instances. This class will be instantiated by ServerManager.

Private Members:

- struct ParseHeaderInfo{
  bool parseHeaderComplete = false;
  bool badRequest = false;
  ServerConnection* thisSC;
  };
- ServerConnectionData* serverConnectionData;
- std::queue<ServerConnection*>* availableServerConnections;
- pthread_mutex_t* standbyMutex;
- char* GenerateMessage(int, int);
- static void* TimeoutBadRequest(void*);

Public Members:

- struct ServerConnectionData {
  int index;
  int comm_fd;
  ServerConnection* thisSC;
  };
- void SetupConnection();
- void Init(std::queue<ServerConnection*>*, pthread_mutex_t*, ServerConnectionData*);
- void BeginRecv();
- static void* toProcess(void*);
- int GetIndex();

char* ServerConnection::Init(std::queue<ServerConnection*> *q, pthread_mutex_t *m, ServerConnectionData *d):

This initializes the variables standbyMutex, ServerConnectionData, and availableServerConnections. AvailableServerConnections represents the available connections on the queue that are ready to be picked up by the client. Because we create the threads at the beginning of our program, StandByMutex is used to pause the threads and wait for a connection. Once a connection is made, the mutex unlocks.

char* ServerConnection::SetupConnection():

This unlocks the standbyMutex mutex.

char* ServerConnection::BeginRecv():

This function handles the requests. It locks the standbyMutex right when we enter the function and creates a new ParseHeaderInfo object. We parse the header information, create a thread for the case where our header is not in the proper format (ex. No "\r\n\r\n"). We recv() from the client inside a while loop and once we hit a newline we break out of the loop. This process is done inside of a thread which waits 20 seconds; if the header hasn't been parsed in 20 seconds then there is an issue with the header we were sent by the client. We check that the header contents are valid (filename, http version, etc) and if so, we continue on to handle either a GET or PUT request.

If it's a GET request, then we send the response back to the client first, then we check if a mutex exists for the file in order to enter the critical region. If the mutex does not exist already, we create a new mutex and if it exists then we get it from our *GlobalServerInfo* class. We lock the mutex, and call our *GetAction*() function while we're in a loop. Every loop iteration we send the file contents we read in *GetAction*() back to the client, and we remain in the loop until we have sent the correct number of bytes.

If our GET request is of type backup or recovery, no information is sent other than a response header. If our GET request is of type list, instead it sends the response followed by sending each backup folder one line at a time seperated by "\n", similarly to how the server sends the file contents in a regular GET request.

If it's a PUT request, then we do the same mutex process as for GET and lock it so we can do the process for our file. Once we lock the mutex, we enter the critical region. Inside the critical region, we have a while loop where in each iteration we receive the body from the client, and call *PutAction*() so it writes the contents we received into the according file. Once we've written the correct number of bytes to the file, then we break out of the while loop and send the appropriate message to the client.

char* ServerConnection::toProcess():

This is a static function that we use in order for our thread to use the function BeginRecv().

char* ServerConnection::GenerateMessage(int, int):

```
char* ServerManager::GenerateMessage(int message, int contentLength)
    if (message == 0)
        return NULL
    else if (message == 200)
        return "HTTP/1.1 200 OK \r\nContent-Length: " + contentLength + "\r\n"
    else if (message == 201)
        return "HTTP/1.1 201 OK\r\nContent-Length: 0\r\n\r\n"
    else if (message == 400)
        return "HTTP/1.1 400 Bad Request\r\nContent-Length: 0\r\n\r\n
    else if (message == 403)
```

return “HTTP/1.1 403 Forbidden\r\nContent-Length: 0\r\n\r\n”
else if (message == 404)
return “HTTP/1.1 404 Not Found\r\nContent-Length: 0\r\n\r\n”
else if (message == 500)
return “HTTP/1.1 500 Internal Server Error\r\nContent-Length: 0\r\n\r\n”
else
return NULL

void* ServerConnection::TimeoutBadRequest():

This function checks whether the header was parsed within 20 seconds. If it wasn’t, then that means we received a badly formatted header from the client.

int ServerConnection::GetIndex():

This function returns the according index of an element in *serverConnectionDatas*.

The ServerManager class:

The purpose of this class is to set up the server and create threadCount number of threads. We are also creating ServerConnections and assigning them to the threads created, and we are creating mutexes for the threads.

Private Members:

- listen_fd: The Listen file descriptor
- std::queue<ServerConnection*>* availableServerConnections;
- unsigned long GetAddress(char*): Returns an Internet address based on specific parameters

Public Members:

- ServerManager(): The constructor of ServerManager
- void Setup(char*, unsigned short): The main process of ServerManager

unsigned long ServerManager::GetAddress(char*):

This function is designed to take an input string referencing an internet address and converts it into a working integer address.

ServerManager::ServerManager():

This creates a new queue of ServerConnections called availableServerConnections.

ServerManager::ServerManager()
availableServerConnections = new std::queue<ServerConnection*>();
listen_fd = 0;

void ServerManager::Setup(`char* address, unsigned short port, int threadCount, bool redundancy`):

The purpose of this function is to be the dispatcher for the various connections being made to the server. Initially, the function will set up every server connection and give each its own thread to be held up by a mutex. When a new connection is made, the dispatcher will pop the next available serverconnection off of the available server connections queue and give it the information it needs for this connection. Once it is given this information, its standby mutex will be lifted and the serverconnection will be permitted to complete its process and push itself back onto the available server connections queue (also reapplying the standby mutex lock).

unsigned long ServerManager::Setup(`char* address, unsigned short port, int threadCount, bool redundancy`)

```
GlobalServerInfo::redundancy = redundancy;

pthread_mutex_t* servConStandbyMutexes = new
pthread_mutex_t[threadCount];
pthread_t* threads = new pthread_t[threadCount];
ServerConnection::ServerConnectionData* serverConnectionDatas = new
ServerConnection::ServerConnectionData[threadCount];

for (int i = 0; i < threadCount; i++)
       serverConnectionDatas[i].thisSC =  new ServerConnection();
       serverConnectionDatas[i].thisSC->Init(availableServerConnections,
&servConStandbyMutexes[i], &serverConnectionDatas[i]);
       serverConnectionDatas[i].index = i;

       pthread_mutex_init(&servConStandbyMutexes[i], NULL);
       pthread_mutex_lock(&servConStandbyMutexes[i]);

       pthread_create(&threads[i], NULL, ServerConnection::toProcess,
&serverConnectionDatas[i]);

 availableServerConnections->push(serverConnectionDatas[i].thisSC);


struct sockaddr_in servaddr
memset(&servaddr, 0, sizeof servaddr)
```

```
listen_fd = socket(AF_INET, SOCK_STREAM, 0)
if (listen_fd < 0)
      Socket error
servaddr.sin_family = AD_INET
servaddr.sin_addr.s_addr = GetAddress(address)
servaddr.sin_port = htons(port)
if (bind(listen_fd, (struct sockaddr*) &servaddr, sizeof servaddr) < 0 )
      bind error
if (listen(listen_fd, 500) < 0)
      listen error
while(1)
      waiting = "waiting for connection\n"
      write waiting to stdout
      comm_fd accept(listen_fd, NULL, NULL)
      if (comm_fd < 0)
            accept error
      if (availableServerConnections->size() > 0) // thread's available
            ServerConnection* servCon =
availableServerConnections->front();
            availableServerConnections->pop();
            // set the file descriptor for that thread
            serverConnectionDatas[servCon->GetIndex()].comm_fd = comm_fd;
            servCon->SetupConnection();
      Else
            Error: No threads are available

GlobalServerInfo::RemoveMutexInfo();

for (int i = 0; i < threadCount; i++) {
      pthread_mutex_destroy(&servConStandbyMutexes[i]);
}
if (servConStandbyMutexes != NULL)
      delete[] servConStandbyMutexes;
if (threads != NULL)
      delete[] threads;
if (serverConnectionDatas != NULL)
      delete[] serverConnectionDatas;
```

```
if (shutdown(listen_fd, SHUT_RDWR) < 0) {
    warn("shutdown()");
}
```

The ServerTools class:

This class was initially intended to have several functions that are used by multiple classes, but as we went through the program we realized we really only needed one of these functions. The function *AppendChar()* is used when we are reading the header from the client. We are appending each character from the buffer we read into the header.

Public Members (all static):

- static void AppendChar(char*&, char);

void ServerTools::AppendChar(char*& string, char character):

```
int strLength = strlen(string);
char* temp = string;
string = new char[strLength + 2];
for (int i = 0; i < strLength; i++) {
      string[i] = temp[i];
}
delete[] temp;
string[strLength] = character;
string[strLength + 1] = '\0';
```

main.cpp:

This file's purpose is to call main() and get our server running. It takes 5 arguments (with a 6th argument optional for redundancy): one for the server's Internet address, one for the port for it to be active on, one indicating the number of threads (-N), the actual number of threads, and optionally redundancy. If nothing is given as an argument for the port, it defaults the port to 80 and the thread count defaults to 4. We have a setValues() function that basically iterates through the arguments passed by the user and sets them accordingly. This function was implemented to handle cases where the arguments are given in different orders. We have an int array called 'wasFound' that we use in order to set the address and port easily. We use a for loop starting at i=0, and once we find the redundancy and thread count arguments, we set the value of setValues[i] to be 1 (meaning its been found). After finishing this loop, we start a new for loop to go through the values in the wasFound array. The first 0 we find is the address, and set that accordingly, and the second 0 we find is the port.

- Argv represents the array of arguments passed by the user
- Argc is the size of this array

void setValues(int argc, char **argv, int &threadCount, bool &redundancy, char *&address, int &port, int *wasFound)

```
For (i =0; i < argc; i++)
      If (argv[i] is "-N")
            Set the thread count to argv[i]
            set the values in the wasFound array at indices i and i+1
      If (argv[i] == "-r")
            Redundancy = true
            Set value in the wasFound array at index i: wasFound[i] = 1
Bool is firstZero = true
For (i=1; i < argc; i++) // start at i =1 because i=0 is ./httpserver
      If (wasFound[i] == 0 && firstZero == true)
            Address = argv[i]
            isFirstZero = false
      Else if (wasFound[i] == 0)
            Port = argv[i]
            Break; // we got everything we needed
```

main(size of array, array of program arguments)

```
int threadCount
bool redundancy = false
char* address
int port = 80

if (argc < 2)
      Error, not enough arguments
if (size of array > 6)
      error, too many arguments, return
if (argc == 2)
      Set the address to the second argument as the address
If (argc == 3)
      Int wasFound = {1, 0, 0}
```

```
      setValues(argc, argv, threadcount, redundancy, address, port,
wasFound)
If (argc ==4)
      Int wasFound = {1, 0, 0, 0}
      setValues(argc, argv, threadcount, redundancy, address, port,
wasFound)
Else If (argc ==5)
      Int wasFound = {1, 0, 0, 0, 0}
      setValues(argc, argv, threadcount, redundancy, address, port,
wasFound)
Else
      Int wasFound = {1, 0, 0, 0, 0, 0}
      setValues(argc, argv, threadcount, redundancy, address, port,
wasFound)
```

References

[1] https://www.geeksforgeeks.org/c-program-list-files-sub-directories-directory/

[2] https://www.tutorialspoint.com/c_standard_library/c_function_ctime.htm