به نام خدا

گزارش پروژه اول درس شبکه های کامپیوتری

سارا رضائی منش ۸۱۰۱۹۸۵۷۶

برنا توسلی ۸۱۰۱۹۸۳۷۴

عنوان پروژه

طراحی یک سرور FTP با قابلیت سرویس دادن همزمان به چند کلاینت با استفاده از socket ها

ساختار پروژه

ساختار درختی فایل ها به صورت زیر می‌باشد:

```
├── configuration
│   ├── config.json
│   └── log.txt
├── include
│   ├── IncludeAndDefine.hpp
│   ├── Parser.hpp
│   ├── Server.hpp
│   └── User.hpp
├── Makefile
└── src
    ├── Client.cpp
    ├── Network.cpp
    ├── Parser.cpp
    ├── Server.cpp
    └── User.cpp

3 directories, 12 files
```

نحوه اجرا

برای ساختن فایل های خروجی سرور و کلاینت از دستور زیر استفاده کنید:

```
make
```

سپس می‌توان با استفاده از دستور های زیر به ترتیب سرور و کلاینت را اجرا کرد:

```
s.out
c.out
```

توضیحات فایل ها

فایل Parser.cpp

همانطور که اشاره شد، این فایل حاوی توابعی است که برای parse کردن فایل config.json نوشته شده اند و در هر دو بخش Server و Client از آن استفاده شده است.

تابع اصلی این فایل تابع parseJson است:

```cpp
#include "../include/Parser.hpp"
using namespace std;

bool hasCharacter(string line, char c) {
   return find(line.begin(), line.end(), c) != line.end() ||
       find(line.begin(), line.end(), c) != line.end();
}
void removeCharacter(string& str, char c){
   str.erase(remove(str.begin(), str.end(), c), str.end());
}

// !!!!!!works with all customizable json files with any permutation!!!!!!!
map<string, vector<string>> parseJson(string fileName) {
   ifstream file;
   file.open(fileName);
   string line, listString = "", field, value, dump;
   map <string, vector<string>> inputs;
   while(getline(file, line)) {
       stringstream ss(line);

       if(hasCharacter(line, '{') || hasCharacter(line, '}') || hasCharacter(line, ']')){
           listString = "";
           continue;
       }
       else if (hasCharacter(line, '[')){
           getline(ss, dump, '"');
           getline(ss, listString, '"');
       }
       else if (listString != ""){
           getline(ss, dump, '"');
           getline(ss, field, '"');
           inputs[listString].push_back(field);
       }
       else{
           getline(ss, dump, '"');
           getline(ss, field, '"');
           getline(ss, value, ',');
           removeCharacter(value, '"');
           removeCharacter(value, ' ');
           removeCharacter(value, ':');
```

```
            if(value != "")
                inputs[field].push_back(value);
        }
    }
    file.close();
    return inputs;
}
```

این تابع نام یک فایل json را به عنوان ورودی دریافت می‌کند و یک داده ساختار map از string به vector حاوی تمام key ها و value های مربوط به آنها به عنوان خروجی برمی‌گرداند. در این تابع vector ها برای ذخیره داده های لیست های فایل config.json مورد استفاده قرار می‌گیرند و در صورتی که یک key تنها یک value داشته باشد، آن value در خانه اول vector ذخیره می‌شود.

لازم به ذکر است این تابع به گونه ای نوشته شده که توانایی parse کردن فایل json به هر فرمت نوشتاری و با هر جایگشت دلخواهی از key ها می‌تواند map با مقادیر صحیحی را به عنوان خروجی برگرداند.

فایل User.cpp و User.hpp

تعریف کلاس User فایل User.hpp به صورت زیر می‌باشد:

```
class User {
public:
    User(std::string username, std::string pass, std::string isAdminS, std::string
datalimitS);
    void printUser();
    bool isValid(std::string username_);
    bool fdMatches(int fd);
    bool login(std::string pass_);
    void updateFd(int fd);
    bool isLoggedIn();
    void logout();
    char* getCurrDir();
    std::string getRelativeDir(std::string path);
    bool updateDir(std::string path);
    bool resetDir();
    bool canDownload(int fileSize);
    void updateDataLimit(int fileSize);
    bool isAdmin();
private:
    std::string username, pass;
    bool admin, loginStatus;
    char currDir[BUFFER_SIZE], originDir[BUFFER_SIZE];
    int datalimit, id;
};
```

این کلاس اطلاعات مربوط به هر user در آرایه users در فایل config.json را ذخیره می‌کند.

فیلد ها

- فیلد های isAdmin ،pass ،username و datalimit برای ذخیره اطلاعات فایل کانفیگ هستند.
- فیلد id برای هر user در ابتدا ۱- است و پس از اینکه یک user به صورت موفقیت آمیز لاگین کرد، به fd آن user تغییر پیدا می‌کند.
- فیلد بولین loginStatus نیز مشخص می‌کند یک user لاگین کرده است یا خیر. در صورتی که مقدار این فیلد true باشد، user اجازه اجرای دستوراتی به جز user و pass را خواهد داشت.
- فیلد currDir پوشه ای که در حال حاضر کاربر در آن قرار دارد را مشخص می‌کند.
- فیلد originDir پوشه ای که کاربر در ابتدا در آن قرار دارد را مشخص می‌کند.

توابع

- سازنده(Constructor) کلاس ابتدا مقادیری که از تابع config.json دریافت شده را در فیلد های متناظر قرار می‌دهد. سپس مقدار id را ۱- می‌کند که نشان می‌دهد کاربری با مشخصات داده شده وارد سیستم نشده و در پایان پوشه فعلی را در فیلد های originDir و currDir قرار می‌دهد. از آنجایی که در روند اجرای دستور دانلود فایل، حجم فایل به بایت محاسبه می‌شود و در فایل کانفیگ سقف اینترنت مصرفی به صورت کیلوبایت است، قبل از این مقدار در فیلد datalimit، آن را در 1000 ضرب می‌کنیم تا به بایت تبدیل شود.

```
User(string username, string pass, string isAdminS, string datalimitS) :
username(username), pass(pass) {
	isAdmin = (isAdminS == "yes");
	datalimit = stoi(datalimitS)*1000;
	id = -1;
	getcwd(currDir, 1024);
	strcpy(originDir, currDir);
}
```

- تابع printUser اطلاعات کانفیگ هر کاربر در یک خط چاپ می‌کند.(این تابع برای دیباگ مورد استفاده قرار گرفته است.)
- تابع isValid با گرفتن یک رشته، بررسی می‌کند که آیا این رشته با username اش برابر است یا خیر.
- تابع fdMatches با گرفتن یک عدد، بررسی می‌کند آیا آن عدد با id اش مقایسه می‌کند. همانطور که در قبل گفته

شد، این id نشاندهنده fd کاربری است که با نام کاربری و رمز عبور متناظر به آن instance از کلاس login کرده است.

- تابع login با گرفتن یک رشته به عنوان password، آن را با فیلد pass خود مقایسه می‌کند و نتیجه را در فیلد loginStatus ذخیره می‌کند و برمی‌گرداند. همچنین با هر بار login مجدد کاربر، فولدری که در آن قرار دارد به حالت اولیه برمی‌گردد و تغییراتی که در لاگین قبلی خود داده مجدد اعمال نمی‌گردد.

```
bool User::login(string pass_){
   loginStatus = (pass_ == pass);
   resetDir();
   return loginStatus;
}
```

- تابع updateFd، با گرفتن یک عدد به عنوان fd، فیلد id را به آن عدد تغییر می‌دهد. این تابع در صورتی مورد استفاده قرار می‌گیرد که یک کاربر خارج شده و مجدد با fd دیگر اما با همان نام کاربری و رمز عبور داخل شود.
- تابع isLoggedIn مقدار فیلد loginStatus را برمی‌گرداند.
- تابع logOut وضعیت فیلد های کلاس را به حالت اولیه برمی‌گرداند. به این صورت که id را به ۱- و loginStatus را به false تغییر می‌دهد. همچنین کاربر را به فولدر اولیه برمی‌گرداند.
- تابع getCurrDir، مقدار فیلد currDir را به عنوان پوشه ای که در حال حاضر کاربر در آن قرار دارد برمی‌گرداند.
- توابع updateDir و resetDir هر دو برای اجرای دستور cwd مورد استفاده قرار می‌گیرند. تابع updateDir، یک رشته به عنوان آدرس دریافت کرده و مکان فعلی کاربر را به آن مسیر تغییر می‌دهد. سپس مقدار فیلد currDir را به مسیر جدید تغییر می‌دهد. در صورتی که pwd بدون هیچ آرگومانی صدا زده شود، تابع resetDir فراخوانی می‌شود. این تابع مکان فعلی کاربر را به مسیر ذخیره شده در originDir تغییر داده و مقدار ذخیره شده در currDir را به مقدار ذخیره شده در originDir تغییر می‌دهد.

```
bool User::updateDir(string argument) {
   bool success = chdir(argument.c_str());
   getcwd(currDir, 1024);
   return !success;
}
bool User::resetDir() {
       strcpy(currDir, originDir);
       return !chdir(originDir);
   }
```

- تابع canDownload با دریافت یک عدد به عنوان حجم فایلی که کاربر می‌خواهد دانلود کند، آن را با حجم دیتای باقی مانده کاربر مقایسه می‌کند و در صورتی که حجم فایل دانلودی کمتر بود، مقدار true و در غیر اینصورت مقدار false برمی‌گرداند.

- تابع updateDataLimit در صورتی که امکان دانلود یک فایل وجود داشت فراخوانی می‌شود و حجم آن فایل را متغیر dataLimit که نشان دهنده حجم باقیمانده کاربر است، کم می‌کند.
- تابع isAdmin مشخص می‌کند که آیا کاربر admin هست یا خیر.

فایل Network.cpp

این فایل تنها شامل یک تابع main می‌باشد که مسئولیت دریافت map ساخته توسط تابع parseJson و بالا آوردن سرور با استفاده از آن را دارد. پس از اجرای دستور run، تا زمانی که در ترمینال ctrl+c وارد نشود، سرور دستورات کاربران اجرا می‌کند. هنگام وارد شدن ctrl+c، سیگنال فعال می‌شود و قبل از خاتمه کامل برنامه، signal_callback_handler اجرا می‌شود. این تابع در فایل Server.hpp تعریف شده است و در ادامه توضیح داده شده است.

```
int main(int argc, char const *argv[]) {
   auto inputs = parseJson(CONFIG_FILE);
   Server server(inputs);
   signal(SIGINT, signal_callback_handler);

   //uncomment to see server info ("config.json")
   //server.printServer();

   server.run();
}
```

فایل Server.cpp و Server.hpp

کلاس Server در فایل Server.hpp به صورت زیر تعریف شده است:

```
class Server {
public:
   Server(std::map<std::string, std::vector<std::string>> inputs);
   void printServer();
   int acceptClient(int port);
   std::string handleCommand(std::string command, std::string argument, int userFd, int
userDataFd);
   void run();

private:
   std::string curr_log, log_path;
   int cmdChannelPort, dataChannelPort, serverDataFd, serverCmdFd;
   std::vector<User> users;
   std::vector <std::string> adminFiles;
   std::map<int, std::string> fdLastRequest, fdLoggedInUser;
```

```
    User* findUserByFd(int userFd);
    User* findUserByName(std::string username_);
    bool loginUser(std::string curr_user, int fd, std::string lastUser);
    bool hasFileAccess(User* currUser, std::string file);
    bool hasDirectoryAccess(User* currUser, std::string file);
    std::string handleUser(int userFd, std::string argument1);
    std::string handlePass(int userFd, std::string argument1);
    std::string handlePwd(User* currUser);
    std::string handleMkd(User* currUser, std::string argument);
    std::string handleDele(User* currUser, std::string argument1, std::string argument2);
    std::string handleFileDel(User* currUser, std::string argument2);
    std::string handleDirDel(User* currUser, std::string argument2);
    std::string handleLs(User* currUser, int dataFd);
    std::string handleCwd(User* currUser, std::string argument1);
    std::string handleRetr(User* currUser, std::string argument1, int dataFd);
    std::string makeRes(int code, std::string msg);
    std::string handleRename(User* currUser, std::string argument1, std::string argument2);
    std::string handleHelp();
    std::string handleQuit(User* currUser, int userFd);
    int setupServer(int port);
    int getFileSize(std::string fileName);
    void sendData(std::string file, int dataFd);
    void convertConfig(std::map<std::string, std::vector<std::string>> inputs);
    void writeLog();
};
void exitLog(std::string log);
void signal_callback_handler(int signum);
```

(توابع اصلی مربوط به هندل کردن دستورات ذکر شده در صورت پروژه در زیر توضیح داده شده‌اند و از توضیحات توابعی که توسط این توابع مورد استفاده قرار می‌گیرند خودداری شده است.)

توضیحات کلی در مورد کلاس و نحوه ورود به سرور

- این کلاس به صورت کلی وظیفه بالا آوردن سرور، listen کردن روی socket ها و اجرای دستورات کلاینت ها را دارد.
- تابع setUpServer، یک سوکت ایجاد می‌کند. همانطور که در کد بالا مشخص است، در constructor کلاس Server این تابع دوبار برای ساختن سوکت داده و دستور فراخوانی می‌شود. در این تابع عملیات ساخت سوکت، bind کردن آن و listen کردن روی آن انجام می‌شود. توضیحات این تابع در کلاس داده شده است.

```
int Server::setupServer(int port) {
    struct sockaddr_in address;
    int server_fd;
    server_fd = socket(AF_INET, SOCK_STREAM, 0);
```

```
    int opt = 1;
    setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(port);

    bind(server_fd, (struct sockaddr *)&address, sizeof(address));
    // only for command
    listen(server_fd, 4);

    return server_fd;
}
```

- تابع acceptClient برای قبول کردن یک کاربر که می‌خواهد به سوکت های سرور متصل شود مورد استفاده قرار می‌گیرد. از آنجایی که دو سوکت برای ارتباط بین کاربر و سرور داریم، هنگام افزوده شدن کاربر جدید این تابع دوبار فراخوانی می‌شود. یکبار با شماره پورت داده و یکبار با شماره پورت دستور فراخوانی می‌شود. این تابع با گرفتن یک شماره پورت، درخواست اتصال کاربر را accept می‌کند و سپس fd مربوط به کاربر، که برای ارتباط با آن مورد استفاده قرار می‌گیرد را بر می‌گرداند.

```
int Server::acceptClient(int port) {
    int client_fd;
    struct sockaddr_in client_address;
    int address_len = sizeof(client_address);
    client_fd = accept(port, (struct sockaddr *)&client_address, (socklen_t*)
                        &address_len);

    return client_fd;
}
```

- تابع run وظیفه پاسخگویی به درخواست های کاربر ها را دارد. این تابع با استفاده از فراخوانی سیستمی select و یک حلقه for بدون بلاک شدن روی یک کاربر، می‌تواند به چندین کاربر به صورت همزمان سرویس دهی کند. از آنجایی که در حالت عادی نباید پیامی روی پورت داده از طرف کاربران ارسال شود، عملیات select و حلقه for تنها برای fd های مربوط به پورت دستور انجام می‌شوند. در قطعه کد زیر ابتدا cmd_master_set که یک داده ساختار set شامل fd های مربوط به دستور هست را خالی می‌کنیم. سپس cmd_fd(command fd) را به آن اضافه می‌کنیم. با استفاده از فراخوانی سیستمی select همه fd های موجود در cmd_master_set را مانیتور می‌کنیم و هر گاه یکی یا بیشتر از آنها آماده ارسال پیام بودند وارد حلقه فور می‌شویم تا با پیمایش مجموعه fd های دستور، fd هایی که select تشخیص داده است را پیدا کنیم و به دستورات آنها پاسخگویی کنیم. ماکروی FD_ISSET بررسی می‌کند که آیا fd فعلی در

حلقه پیامی ارسال کرده است یا خیر.

```
  FD_ZERO(&cmd_master_set);
  cmd_max_sd = cmd_fd;
  FD_SET(cmd_fd, &cmd_master_set);

  printf("Server is up!\n");

  while (1) {
      cmd_working_set = cmd_master_set;
      select(cmd_max_sd + 1, &cmd_working_set, NULL, NULL, NULL);
      for(int i = 0; i <= cmd_max_sd; i++) {
          memset(buffer, 0, 1024);
          if (FD_ISSET(i, &cmd_working_set)) { ...
```

در این تابع در صورتی که fd ای که می‌خواهد پیام ارسال کند همان fd مربوط به پورت دستور سرور باشد(cmd_fd) یعنی یک کاربر جدید می‌خواهد به سرور متصل شود. در این حالت دو بار تابع acceptClient را برای دو پورت داده و دستور صدا می‌زنیم. Fd جدید مربوط به پورت دستور را به cmd_master_set اضافه می‌کنیم و همچنین fd جدید دستور و داده را به یک map به نام fds اضافه می‌کنیم. این map در ادامه برای این استفاده می‌شود که تشخیص دهیم هر پورت داده مربوط به کدام پورت دستور است تا اگر پاسخ دستور فرستاده شده روی یک پورت دستور مربوط به یک کاربر حاوی داده بود، بتوانیم fd داده مربوط به آن کاربر را برای فرستادن داده ها پیدا کنیم.

```
if (i == cmd_fd) {  // new client
                    new_cmd_socket = acceptClient(cmd_fd);
                    new_data_socket = acceptClient(data_fd);
                    FD_SET(new_cmd_socket, &cmd_master_set);
                    fds[new_cmd_socket] = new_data_socket;
                    if (new_cmd_socket > cmd_max_sd)
                              cmd_max_sd = new_cmd_socket;
}
```

در صورتی که درخواست دهنده همان cmd_fd نبود یعنی یک کاربر دستور جدید وارد کرده است که در این صورت در این صورت دستور دریافت شده و با تابع handleCommand به آن رسیدگی می‌شود. دقت شود در صورتی که fd پیامی فرستاده باشد اما تابع recv چیزی دریافت نکند یعنی پیام فرستاده نشانه disconnect شدن کاربر است. در این صورت هر دو fd دستور و داده کاربر بسته می‌شوند، fd دستور از cmd_master_set و fds حذف می‌شود و دستور continue صدا زده می‌شود.

```
else { // client sending msg
          int bytes_received;
          bytes_received = recv(i , buffer, BUFFER_SIZE, 0);
          if (bytes_received == 0) { // EOF
          if (fdLoggedInUser.find(i) != fdLoggedInUser.end()){
```

```
                findUserByName(fdLoggedInUser[i])->logout();
                fdLoggedInUser.erase(i);
            }
            fdLastRequest.erase(i);
            curr_log += "Client on fd " + to_string(i) + " disconnected";
            writeLog();

            write(1, "Client disconnected!\n", 22);
            close(i);
            close(cmdDataFd[i]);
            FD_CLR(i, &cmd_master_set);
            auto it = cmdDataFd.find (i);
            cmdDataFd.erase(it);
            continue;
            }
            stringstream ss(buffer);
            string command, argument;
            getline(ss, command, ' ');
            getline(ss, argument, '\n');
            if (command.back() == '\n')
                command.pop_back();

            strcpy(buffer, handleCommand(command, argument, i, cmdDataFd[i]).c_str());
            curr_log += "Server response (fd = " + to_string(i) + ") was: "+ writeLog();

            send(i, buffer, strlen(buffer), 0);
}
```

- فیلد users یک آرایه از همه user هایی است که در فایل config.json آمده است.
- فیلد های cmdChannelPort آدرس پورت داده را در خود نگه می‌دارد و data_fd و cmd_fd نیز به ترتیب fd های مربوط به داده و دستور را برای ارتباط با کلاینت ذخیره می‌کنند.
- تابع convertConfig برای تبدیل کردن map داده شده از تابع parser به فیلد های یوزر است و در constructor سرور مورد استفاده قرار می گیرد.

```
void Server::convertConfig(map<string, vector<string>> inputs) {
   cmdChannelPort = stoi(inputs[COMMAND_CHANNEL_PORT][0]);
   dataChannelPort = stoi(inputs[DATA_CHANNEL_PORT][0]);

   for (auto x: inputs[CONFIG_ADMIN_FILES])
       adminFiles.push_back(x);

   for(int i = 0; i < inputs[CONFIG_USER].size(); i++) {
       User newUser(inputs[CONFIG_USER][i], inputs[CONFIG_PASSWORD][i],
                    inputs[CONFIG_IS_ADMIN][i], inputs[CONFIG_SIZE][i]);
       users.push_back(newUser);
   }
}
```

- تابع handleCommand برای اجرای همه دستورات مورد استفاده قرار می‌گیرد. به این صورت که دستور و آرگومان ها مربوط به آن به این تابع داده می‌شوند و این تابع بر اساس نوع دستور و با استفاده از توابع موجود در کلاس سرور، دستور ها را هندل می‌کند.
- برای اجرای دستور user از تابع findUser استفاده می‌کنیم. این تابع در بین همه یوزر های موجود در سرور پیمایش کرده و در صورتی که موفق به پیدا کردن آرگومان دستور بین نام های یوزر ها شد، یک پوینتر به آن یوزر و در غیر اینصورت مقدار null را برمی‌گرداند. در commandHandler در صورتی که مقدار بازگشتی null نبود، fd به همراه username وارد شده وارد یک map به نام fdLastRequest می‌شوند و پیام مربوط به ورود پسورد به کاربر داده می‌شود.

```
fdLastRequest(map)
Key: fd(int) → username(string)
```

- از این map در ادامه برای این استفاده می‌شود که تشخیص دهیم آیا کلاینتی که دستور pass را وارد کرده است، دستور user را هم وارد کرده است یا خیر. در صورتی که بعد از دستور user هر دستور دیگری وارد شود، کلاینت از map حذف می‌شود.
- از نام کاربری ذخیره شده در این map برای این استفاده می شود که بعد از اینکه یک pass توسط fd وارد شد، نام کاربری و پسورد دوباره چک شوند و در صورتی که دو کاربر پسورد مشابه داشتند به جای هم نتوانند وارد شوند و یا اگر کاربر دو درخواست user پشت سر هم زد، آرگومان درخواست user ثانویه به عنوان معیار برای صحت سنجی دستور pass در نظر گرفته شود.
- برای اجرای دستور `pass` از تابع loginUser استفاده می‌کنیم. این تابع در صورتی که کاربر قبل از آن دستور user را وارد نکرده باشد، خطای "bad sequence" می‌دهد و در غیر اینصورت، اگر پسورد داده شده صحیح بود، کاربر را به یک map به نام fdLoggedInUser اضافه می‌کند.

```
bool Server::loginUser(string password, int fd, string lastUser) {
   for(auto& user : users) {
       if(user.fdMatches(fd) && user.isValid(lastUser)){
           if(user.login(password)){
               fdLoggedInUser[fd] = lastUser;
               return true;
           }
           else
               return false;
       }
   }
   return false;
}
```

مپ به صورت زیر می‌باشد:

```
fdLoggedInUser(map)
Key: fd(int) → username(string)
```

- این map در ادامه به این منظور استفاده می‌شود که تشخیص دهیم کلاینتی که دستورات دیگری به جز user و pass را وارد می‌کند، قبلا عملیات login را انجام داده است یا خیر.
- در صورتی که کاربر با وارد کردن ctrl+c یا دستور quit از سیستم خارج شود، از این map حذف می‌گردد.
- هندلر مربوط به هر دستور در این کلاس به صورت handle+commandName نامگذاری شده است. هر هندلر با دریافت آرگومان های مورد نیاز خود، یک رشته به عنوان خروجی برمی‌گرداند که نشان می‌دهد عملیات موفقیت آمیز بوده است یا خیر.
- تابع handleHelp برای اجرای دستور help فراخوانی می‌شود. این دستور برای نمایش دادن همه دستور های سیستم و توضیحات آنها مورد استفاده قرار می‌گیرد. در این تابع صرفا یک استرینگ به نام help تشکیل می‌دهیم و یک استرینگ طولانی شامل دستورات و توضیحات آنها را به آن اضافه کرده و برمی‌گردانیم.

توابع لاگر

- این تابع برای نوشتن log در یک فایل که در پوشه configuration قرار دارد مورد استفاده قرار می‌گیرد و با هر بار فراخوانی، فیلد curr_log کلاس سرور را بررسی می‌کند. و در صورتی که خالی نبود، پیام را به همراه زمان ثبت نام یادداشت می‌کند. رشته هایی که در curr_log قرار دارند، عموما شامل دستور کاربر و پاسخ سرور به دستور می‌باشد(بدون داده ها).

```
void Server::writeLog(){
    if (curr_log == "")
        return;

    time_t now = time(0);
    char* dt = strtok(ctime(&now), "\n");
    string newLog = "[";
    newLog += dt;
    newLog += "] ";
    newLog += curr_log;
    newLog += "\n";

    ofstream logFile;
    logFile.open(log_path, ios_base::app);
    logFile << newLog;
```

```
    logFile.close();
    curr_log = "";
}
```

- هنگام بستن سرور با دستور ctrl+c، یک سیگنال فعال می‌شود که تابع exitLog را با پیام Server is offline فراخوانی می‌کند. تابع exitlog پیام دریافتی را در log می‌نویسد و سپس برنامه بسته می‌شود. به این صورت زمان offline شدن سرور هم در لاگ ثبت می‌شود.

```
void exitLog(string log){
    time_t now = time(0);
    char* dt = strtok(ctime(&now), "\n");
    string newLog = "[";
    newLog += dt;
    newLog += "] ";
    newLog += log;
    newLog += "\n";

    ofstream logFile;
    logFile.open(string(SERVER_ABSOLUTE_PATH) + LOG_FILE_NAME, ios_base::app);
    logFile << newLog;
    logFile.close();
}

void signal_callback_handler(int signum) {
   exitLog("Server is offline.");
   printf("\n");
   exit(signum);
}
```

- تابع sendData برای فرستادن بخش داده دستوراتی که داده بازگشتی از سمت سرور دارند مورد استفاده قرار می‌گیرد. این تابع داده ای که باید به سمت کاربر فرستاده شود را به صورت یک رشته دریافت کرده و ارسال می‌کند.

```
void Server::sendData(string file, int dataFd){
    send(dataFd, file.c_str(), strlen(file.c_str()), 0);
}
```

توابع مربوط به دستور های پس از لاگین

- تابع findUserByFd قبل از اجرای همه دستورات این بخش فراخوانی می‌شود. این تابع در میان همه کاربر ها می‌گردد و در صورتی که کاربری را پیدا کرد که مقدار فیلد id آن با fd یکسان است(فراخوانی تابع fdMatches)، یک پوینتر به آن کاربر را برمی‌گرداند و در غیر اینصورت مقدار NULL را برمی‌گرداند. NULL بودن مقدار بازگشتی

این تابع یعنی fd درخواست دهنده عملیات login را انجام نداده است و در نتیجه ارور مربوط به login نمایش داده می‌شود. تابع findUserByName به صورت مشابهی عمل می‌کند اما یک رشته دریافت می‌کند و آن را به کاربر می‌دهد تا با نام خود مقایسه کند.

```
User* Server::findUserByFd(int fd) {
    for(auto& user : users)
        if(user.fdMatches(fd))
            return &user;
    return NULL;
}
```

و فراخوانی آن در تابع commandHandler به صورت زیر می‌باشد:

```
User* currUser = findUserByFd(userFd);
```

- تابع hasDirectoryAccess برای اجرای دستور حذف پوشه مورد استفاده قرار می‌گیرد. این تابع با گرفتن یک پوینتر به کاربر درخواست دهنده و نام پوشه‌ای که می‌خواهد حذف کند، آن پوشه و تمام زیرفایل ها و زیرپوشه های آن را بررسی می‌کند و در صورتی که در پوشه‌ای که کاربر می‌خواهد حذف کند، فایلی وجود داشت که کاربر اجازه دسترسی به آن را نداشت، فرآیند حذف متوقف می‌شود و پیام ارور مربوطه نمایش داده می‌شود.

```
bool Server::hasDirectoryAccess(User* currUser, string file){
    DIR *dir; struct dirent *diread;
    if((dir = opendir(currUser->getRelativeDir(file).c_str())) != nullptr) {
        while((diread = readdir(dir)) != nullptr) {
            if (diread->d_name == ".." || diread->d_name == ".")
                continue;
            if (!hasFileAccess(currUser, diread->d_name))
                return false;
        }
        closedir(dir);
    } else {
        perror("opendir");
        return false;
    }
    return true;
}
```

- تابع hasFileAccess برای اجرای دستورات حذف فایل و یا تغییر نام فایل مورد استفاده قرار می‌گیرد. این تابع با گرفتن یک پوینتر به کاربر درخواست دهنده و نام فایلی که می‌خواهد حذف کند، بررسی می‌کند که آیا این فایل توسط کاربر قابل دسترسی است یا خیر. در صورتی که قابل دسترسی نباشد، اجرای دستور متوقف می‌شود و پیام ارور مربوطه نمایش داده می‌شود.

```
bool Server::hasFileAccess(User* currUser, string file){
```

```
    return (!count(adminFiles.begin(), adminFiles.end(), file) || currUser->isAdmin());
}
```

- تابع handlePwd برای اجرای دستور pwd مورد استفاده قرار می‌گیرد. این دستور آدرس پوشه فعلی که کاربر در آن قرار دارد را نشان می‌دهد. برای دریافت این آدرس، از instance کلاس user متناظر با fd که دستور را وارد کرده است تابع getCurrDir را فراخوانی کرده و نمایش می‌دهد.

```
string Server::handlePwd(User* currUser) {
    return "257: "+string(currUser->getCurrDir());
}
```

- تابع handleMkd برای اجرای دستور mkd مورد استفاده قرار می‌گیرد. این دستور در آدرس داده شده یک پوشه جدید ایجاد می‌کند. در این تابع برای ایجاد پوشه جدید از یک کامند bash به نام mkdir استفاده می‌کنیم. از آنجایی که کاربر آدرس پوشه ای که می‌خواهد ساخته شود را نسبت به پوشه ای که سرور در آن قرار دارد وارد می‌کند، برای اینکه آدرس مطلق را بدست آوریم، مقدار ذخیره شده در currDir کاربر را با تابع getCurrDir دریافت می‌کنیم و قبل از آرگومان وارد شده توسط کاربر اضافه می‌کنیم. با اضافه کردن دستور "mkdir" قبل از این آدرس، یک دستور bash ایجاد می‌شود که system() می‌تواند آن را اجرا کند و در صورت موفقیت آمیز بودن عملیات، مقدار صفر برگرداند. پس برای تعیین مقدار بازگشتی تابع handleMkd، چک می‌کنیم که آیا مقدار بازگشتی system صفر هست یا خیر و در هر صورت پیام مناسب را برمی‌گردانیم.

```
string Server::handleMkd(User* currUser, string argument) {
    string bashCommand = "mkdir " + string(currUser->getCurrDir()) + argument;
    return !system(bashCommand.c_str()) ? "257: " + argument + " created." : "503: No such
file or directory!";
}
```

- تابع handleDele برای اجرای دستور dele فراخوانی می‌شود. این دستور برای حذف یک فایل و یا پوشه استفاده می‌شود. این دستور دو flag متفاوت دارد که برای اجرای هر کدام از آنها یک تابع جدا فراخوانی می‌شود.

```
string Server::handleDele(User* currUser, string argument1, string argument2) {
    if(argument1 == "-f")
        return handleFileDel(currUser, argument2);
    else if(argument1 == "-d")
        return handleDirDel(currUser, argument2);
    else
        return SYNTAX_ERROR;
}
```

- تابع handleFileDel برای اجرا دستور dele با فلگ f- فراخوانی می‌شود و برای حذف یک فایل با نام داده شده مورد استفاده قرار می‌گیرد. در این تابع ابتدا برای دریافت آدرس مطلق فایل، آدرس پوشه فعلی کاربر را گرفته و با کاراکتر "/" قبل از اسم فایل وارد شده توسط کاربر قرار می‌دهد. سپس با استفاده از تابع remove آن فایل را حذف می‌کند. remove در صورت موفقیت آمیز بودن عملیات، مقدار صفر برمی‌گرداند.

این تابع قبل از اجرای عملیات حذف فایل، با استفاده از تابع hasFileAccess چک می‌کند که آیا کاربر به فایلی که می‌خواهد آن را حذف کند دسترسی دارد یا خیر. در صورتی که دسترسی نداشت، اجرای دستور متوقف می‌شود.

```
string Server::handleFileDel(User* currUser, string argument2) {
   if (!hasFileAccess(currUser, argument2))
       return FILE_UNAVAILABLE;
   char path[BUFFER_SIZE];
   strcpy(path, currUser->getRelativeDir(argument2).c_str());
   return !remove(path) ? "250: " + argument2 + " deleted." : SYNTAX_ERROR;
}
```

- تابع handleDirDel برای اجرای دستور dele با فلگ d- فراخوانی می‌شود و برای حذف یک پوشه مورد استفاده قرار می‌گیرد. با توجه به پیچیدگی عملیات حذف پوشه هایی که خالی نیستند و ممکن است زیرپوشه ها و زیرفایل های زیادی داشته باشند، این دستور را به دستور bash تبدیل کرده و با استفاده از system آن را اجرا می‌کنیم. دستور bash به صورت rm -r است که فلگ r- به نشانه recursive مشخص می‌کند که فایل داده شده باید با تمام پوشه ها و فایل های داخلش حذف شود. مشابه توابع قبلی، آدرس مطلق ساخته شده و به عنوان آرگومان این دستور اضافه می‌شود.

این تابع قبل از اجرای فرآیند حذف، با استفاده از تابع hasDirectoryAccess چک می‌کند آیا در پوشه ای که کاربر می‌خواهد حذف کند و زیرپوشه های آن فایلی وجود دارد ه کاربر به آن دسترسی نداشته باشد یا خیر. در صورت وجود چنین فایلی، اجرای دستور متوقف می‌شود.

```
string Server::handleDirDel(User* currUser, string argument2) {
   if (!hasDirectoryAccess(currUser, argument2))
       return FILE_UNAVAILABLE;
   string bashCommand = "rm -r " + currUser->getRelativeDir(argument2);
   return !system(bashCommand.c_str()) ? "257: " + argument2 + "      deleted." :
   SYNTAX_ERROR;
}
```

- تابع handleLs برای اجرای دستور ls فراخوانی می‌شود. این دستور برای نشان دادن همه فایل ها و پوشه های موجود در پوشه فعلی که کاربر در آن قرار دارد مورد استفاده قرار می‌گیرد. در این تابع پوشه فعلی که کاربر در آن قرار دارد را از

تابع getCurrDir دریافت می‌کنیم و با استفاده از تابع opendir یک پوینتر به stream مربوط به آن دریافت می‌کنیم و نام فایل ها را از این stream دریافت می‌کنیم. Opendir در صورتی که موفق به باز کردن پوشه نشود، پوینتر خالی برمی‌گرداند. پس عملیات خواندن از stream را در صورتی که مقدار بازگشتی null نباشد ادامه می‌دهیم. در پایان نام فایل ها را با کاراکتر "|" به هم چسبانده و برمی‌گردانیم.

```
string Server::handleLs(User* currUser, int dataFd) {
   string filesString = "";
   DIR *dir; struct dirent *diread;
   vector<char *> files;
   if((dir = opendir(currUser->getCurrDir())) != nullptr) {
       while((diread = readdir(dir)) != nullptr) {
           files.push_back(diread->d_name);
       }
       closedir(dir);
   } else {
       perror("opendir");
       return ERROR;
   }
   for(auto file : files)
       filesString += "|" + string(file) + " ";
   sendData(filesString, dataFd);
   return LIST_TRANSFER_DONE;
}
```

- تابع handleCwd برای اجرای دستور cwd فراخوانی می‌شود. این دستور برای تغییر پوشه مورد استفاده قرار می‌گیرد. در صورتی که این دستور بدون آرگومان وارد شود، کاربر باید به پوشه اولیه نسبی منتقل شود. در این صورت تابع resetDir و در غیر اینصورت تابع updateDir را از کلاس user را اجرا می‌کنیم. هر دو این توابع در صورت موفقیت آمیز بودن مقدار true را برمی‌گردانند.(توضیح توابع در بخش کلاس user داده شده‌اند.)

```
string Server::handleCwd(User* currUser, string argument1) {
   if(argument1 == "") {
       if(currUser->resetDir())
           return SUCCESSFUL_CHANGE;
       else
           return ERROR;
   }
   else if(currUser->updateDir(argument1))
       return SUCCESSFUL_CHANGE;

   return SYNTAX_ERROR;
}
```

- تابع handleRetr برای اجرای دستور retr فراخوانی می‌شود. این دستور برای دانلود یک فایل مورد استفاده قرار

می‌گیرد. در این تابع ابتدا آدرس مطلق فایل را می‌سازیم و سپس با استفاده از تابع getFileSize، سایز این فایل را به بایت دریافت می‌کنیم. با استفاده از تابع canDownload کلاس user بررسی می‌کنیم که آیا کاربر امکان دانلود کردن فایل با این حجم را دارد یا خیر و در صورتی که این امکان وجود داشت، با استفاده از تابع updateDataLimit کلاس user، حجم فایل را از حجم دیتا کاربر کم می‌کنیم. در مرحله بعد محتویات فایل را در صورت وجود در رشته filesString ذخیره می‌کنیم(این رشته به صورت by value پاس داده شده است.) و در صورتی که همه عملیات های فوق موفقیت آمیز بودند پیام موفق بودن عملیات را برمی‌گردانیم.

این تابع قبل از اجرای عملیات دانلود فایل، با استفاده از تابع hasFileAccess چک می‌کنیم که آیا کاربر درخواست دهنده به فایلی که می‌خواهد نام آن را تغییر دهد دسترسی دارد یا خیر. در صورتی که دسترسی وجود نداشت اجرای دستور متوقف می‌شود.

```
string Server::handleRetr(User* currUser, string argument1, int dataFd)  {
   if(!hasFileAccess(currUser, argument1))
       return FILE_UNAVAILABLE;
   string fileString;
   string path = currUser->getRelativeDir(argument1);
   ifstream ifs(path);
   int fileSize = getFileSize(argument1);
   if(!currUser->canDownload(fileSize))
       return CANT_OPEN_DATA_CONNECTION;

   currUser->updateDataLimit(fileSize);
   stringstream buffer;
   buffer << ifs.rdbuf();
   if(!buffer.str().size())
       return SYNTAX_ERROR;

   fileString = buffer.str();
   string res = argument1 + ":\n" + fileString;
   if(fileString == "")
       res = "";
   sendData(res, dataFd);
   return SUCCESSFUL_DOWNLOAD; }
```

- تابع getFileSize با گرفتن یک رشته حاوی نام فایل، سایز آن فایل را به بیت برمی‌گرداند.

```
int Server::getFileSize(string fileName) {
   ifstream in_file(fileName, ios::binary);
   in_file.seekg(0, ios::end);
   int file_size = in_file.tellg();
   return file_size;
}
```

- تابع handleRename برای اجرای دستور rename فراخوانی می‌شود. این دستور برای تغییر نام یک فایل مورد استفاده قرار می‌گیرد. برای اجرای این عملیات از تابع rename استفاده می‌کنیم. این تابع دو آرگومان به ترتیب به عنوان نام قدیمی و نام جدید دریافت می‌کند و نام فایل را از نام قدیمی به نام جدید تغییر می‌دهد. سپس در صورت موفقیت آمیز بودن این عملیات مقدار صفر را برمی‌گرداند.

  این تابع قبل از اجرای عملیات تغییر نام با استفاده از تابع hasFileAccess چک می‌کنیم که آیا کاربر درخواست دهنده به فایلی که می‌خواهد نام آن را تغییر دهد دسترسی دارد یا خیر. در صورتی که دسترسی وجود نداشت اجرای دستور متوقف می‌شود.

```
string Server::handleRename(User* currUser, string argument1, string argument2) {
    if(!hasFileAccess(currUser, argument1))
        return FILE_UNAVAILABLE;
    if(!rename(argument1.c_str(), argument2.c_str()))
        return SUCCESSFUL_CHANGE;
    return SYNTAX_ERROR;
}
```

- تابع handleQuit برای اجرای دستور quit فراخوانی می‌شود. این دستور برای log out کردن کاربر از سیستم مورد استفاده قرار می‌گیرد. در این تابع ابتدا fd کاربر را از map نگهدارنده fd و نام کاربر هایی که login کرده‌اند حذف می‌کنیم و سپس تابع logout در کلاس user را فراخوانی می‌کنیم.(توضیحات این تابع در بخش کلاس user) آورده شده است. در پایان پیام مربوط به خروج موفقیت آمیز کاربر نمایش داده می‌شود.

  لازم به ذکر است که اگر کاربر قبل از login کردن از این دستور استفاده کند، هنگام فراخوانی findUserById متوجه این موضوع می‌شویم و در غیر اینصورت حالتی وجود ندارد که عملیات های انجام شده در این تابع موفقیت آمیز نباشند.

```
string Server::handleQuit(User* currUser, int userFd) {
    fdLoggedInUser.erase(userFd);
    currUser->logout();
    return SUCCESSFUL_QUIT;
}
```

- تابع main در سرور ابتدا تابع parseJson را صدا می‌زند و سپس یک instance از کلاس سرور با مپ دریافتی از این تابع ایجاد می‌کند. در پایان با فراخوانی تابع run از سرور، سرویس دهی به کاربر ها شروع می‌شود.

```
int main(int argc, char const *argv[]) {
    auto inputs = parseJson("config.json");
    Server server(inputs);
    server.run();
```

```
}
```

فایل Client.cpp

در این فایل تنها از دو تابع connectServer و main استفاده شده است که وظیفه وصل شدن به سرور، فرستادن دستور، دریافت پاسخ و نشان دادن آن را دارند.

- تابع connectServer شماره پورتی که باید به آن متصل شود را دریافت کرده و برای آن درخواست اتصال را ارسال می‌کند. در صورت پذیرفته شدن درخواست، fd مربوطه برای ارتباط با سرور برگردانده می‌شود.

```
int connectServer(int port) {
   int fd;
   struct sockaddr_in server_address;

   fd = socket(AF_INET, SOCK_STREAM, 0);

   server_address.sin_family = AF_INET;
   server_address.sin_port = htons(port);
   server_address.sin_addr.s_addr = inet_addr("127.0.0.1");

   if (connect(fd, (struct sockaddr *)&server_address, sizeof(server_address)) < 0) { //
checking for errors
       printf("Error in connecting to server\n");
   }

   return fd;
}
```

- تابع main در ابتدا با خواندن فایل config.json پورت های داده و دستور را دریافت می‌کند و با دوبار فراخوانی تابع connectServer برای آنها درخواست ارسال می‌فرستد.

```
int data_socket, cmd_socket;
char cmdBuff[1024] = {0}, dataBuff[1024] = {0};
char msgtoServer[1024] = {0};

auto inputs = parseJson("config.json");
int cmd_port = stoi(inputs["commandChannelPort"][0]);
int data_port = stoi(inputs["dataChannelPort"][0]);

data_socket = connectServer(data_port);
cmd_socket = connectServer(cmd_port);
```

سپس در یک حلقه بی‌نهایت می‌تواند تا زمانی که دستور ctrl+c وارد نشده است می‌تواند دستور ارسال کرده و یا دریافت کند. در این حلقه در صورتی که دستورات فرستاده شده، حاوی پاسخ داده ای نبودند، تنها پیام ارسال شده توسط سرور چاپ می‌شود و در غیر این صورت تابع recv روی fd داده نیز اجرا می‌شود تا داده های بازگشتی نیز

دریافت شوند.

```
recv(cmd_socket, cmdBuff, 1024, 0);
       printf("%s\n", cmdBuff);
       if(string(currCommand) == LS_COMMAND || string(currCommand) == RETR_COMMAND) {
           memset(dataBuff, 0, 1024);
           recv(data_socket, dataBuff, 1024, 0);
           if(!dataBuff)
               continue;
           cout << "The data received from server is as follows\n";
           printf("%s\n", dataBuff);
}
```

در این پروژه دستور retr به دو صورت هندل شده است. هم اطلاعات فایل دانلود شده به صورت یک رشته برای کاربر فرستاده شده و نمایش داده می‌شود و هم یک فایل جدید به صورت کپی از فایل دانلود شده در فولدر فعلی کاربر ساخته می‌شود.

```
if(string(currCommand) == RETR_COMMAND) {
           stringstream ss(dataBuff);
           string fileName, dump;
           getline(ss, fileName, ':');
           getline(ss, dump, '\n');
           int sizeOfFilename = fileName.size()+1;
           string res = string(dataBuff).substr(sizeOfFilename+1);
           ofstream out("copy_of_"+string(fileName));
           out << res;
           out.close();
}
```

در پایان تابع main در صورت خارج شدن از حلقه، هر دو fd بسته می‌شوند.

```
close(data_socket);
close(cmd_socket);
```

فایل IncludeAndDefine.hpp

شامل کتابخانه ها و ثابت های استفاده شده در این پروژه می‌باشد.