Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Кафедра компьютерных систем и программных технологий

Отчёт по лабораторной работе

Дисциплина: ЭВМ и телекоммуникации

Тема: Организация сетевого взаимодействия. Протоколы TCP и UDP

Выполнил студент гр. 43501/1 А.В. Пузанов

Руководитель К.Д. Вылегжанина

“ ” 2016 г.

Санкт -Петербург
2016

1. Цели работы

Изучение принципов программирования сокетов с использованием протоколов TCP и UDP.

2. Программа работы

Разработать приложение-клиент и приложение сервер электронной почты.

Необходимые операции для сервера:

- Прием почтового сообщения от одного клиента для другого
- Хранение электронной почты для клиентов
- Посылка клиенту почтового сообщения по запросу с последующим удалением сообщения
- Посылка клиенту сведений о состоянии почтового ящика.

Необходимые функции для клиента:

- Передача электронного письма на сервер для другого клиента
- Проверка состояния своего почтового ящика
- Получение конкретного письма с сервера

2.1. Реализация серверного и клиентского приложений

Протокол взаимодействия является текстовым. Названия команд максимально близки к командам протокола POP3. Список команд представлен в табл.1.

Название команды	Параметры	Описание
login	<name> <password>	Вход зарегестрированного пользователя
reg	<name> <password>	Регистрация нового пользователя
list	[<letter number>]	Запрос информации о письме (письмах)
stat	-	Запрос количества писем
retr	<letter number>	Запрос на получение письма
delete	<letter number>	Удаление письма
top	<letter number lines count>	Запрос на получение первых строк письма
exit	-	Завершение работы программы
logout	-	Завершение работы текущего пользователя
send	<recipient>	Отправка письма

Табл.1. Список команд протокола.

В ответ на команды клиента сервер отвечает одни из двух типов сообщений:
«+OK: дополнительная информация».
«-ERR: дополнительная информация».

Кроме команды протокола присутствуют команды управления сервером и клиентом. Их список представлен в табл.2

Название команды	Параметры	Описание
Команды клиента		
Menu	-	Вывод на экрана меню с доступными командами клиента

Команды сервера		
Menu	-	Вывод на экрана меню с доступными командами сервера
Show	-	Вывод на экран списка всех подключенных клиентов
Kill	<number of client>	Отключение выбранного клиента
Exit	-	Завершение работы сервера

Табл.2. Список консольных команд.

Для корректной работы сервера в рабочей директории должна быть создана папка mail. В этой папке сервер сохраняет файл с информацией о зарегистрированных пользователях и папки с письмами для каждого пользователя.

Файл с информацией о пользователях называется users.txt и хранит пары имя пользователя и пароль в открытом виде. При создании нового пользователя его регистрационные данные заносятся в этот файл.

При регистрации нового пользователя для него создается собственная папка, куда будут сохраняться все адресованные ему письма.

Письма хранятся в текстовых файлах, которые разделены на две части – служебная часть(заголовки) и собственно тело письма. В заголовках хранится информация об отправителе письма, получателе и времени создания.

В начале работы на сервере создается два потока – один для обработки консольного ввода и один для ожидания входящих соединений. При приеме очередного соединения сервер проверяет, не превышен ли лимит клиентов и если нет, то добавляет нового клиента в список подключенных. При этом для хранения данных о клиенте создается структура типа user и новый поток, выполняющий взаимодействие с клиентом.

Структура типа user имеет следующие поля:

```
typedef struct {
char name[DEFAULT_BUFLEN]; // имя пользователя
int sock; // сокет, через который ведется взаимодействие с пользователем
int isLogged; // флаг, показывающий авторизирован ли пользователь
int index; // индекс в общем массиве обрабатываемых пользователей
char *ipAddr[DEFAULT_BUFLEN]; // IP адрес пользователя
int port; // порт который использует пользовательское приложение
} user;
```

Максимальное число обрабатываемых одновременно подключений задается константой MAX_USERS.

Алгоритм работы сервера представлен на рисунке 1.

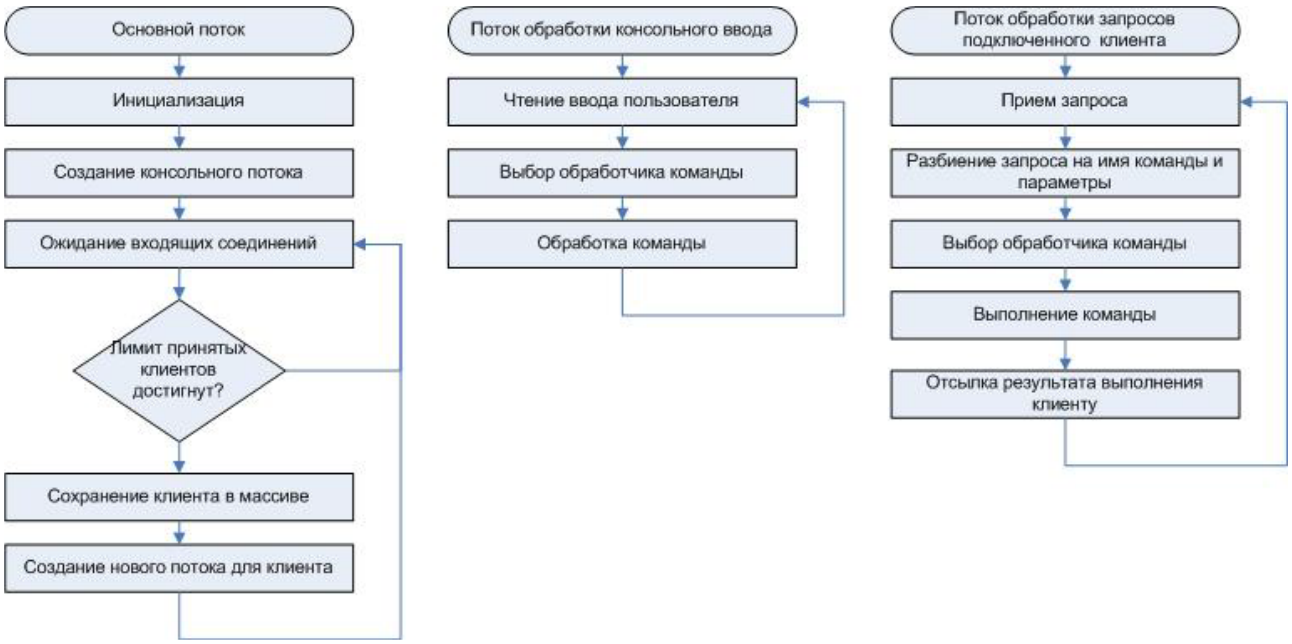

Рис. 1. Алгоритм работы серверных потоков

Алгоритм работы клиента представлен на рисунке 2

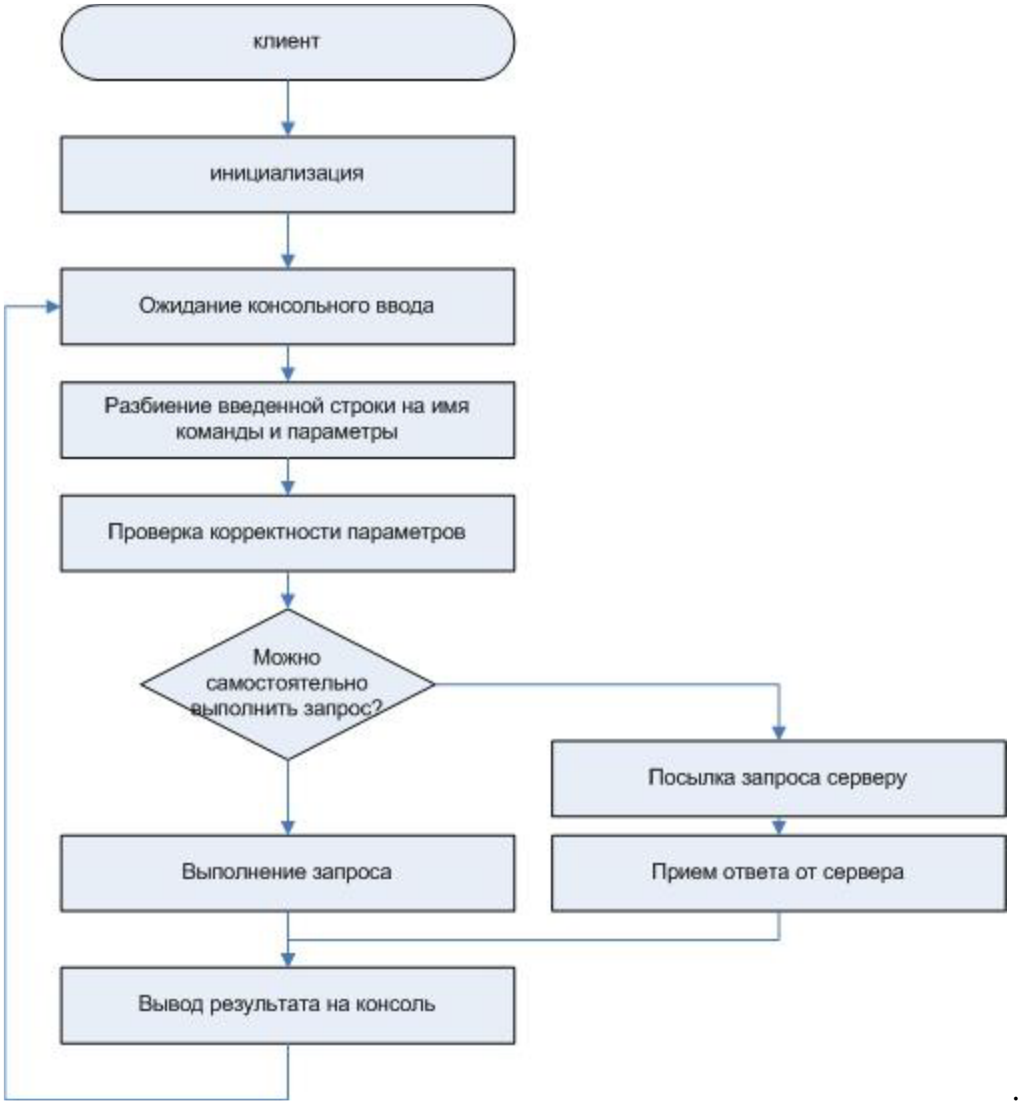

Рис.2. Алгоритм работы клиента

Исходный код программы представлен в приложении 1.
Пример работы с программой:

Со стороны клиента:

```
$ ./tcpmail.exe C
Using defalut ip and port: 127.0.0.1: 7500
        MAIL CLIENT

Main menu:

 |=====Commands:==============|============================|
 | register new user ->       reg <NAME> <PASSWORD>
 | login             ->     login <NAME> <PASSWORD>
 | send letter       ->      send <RECIPIENT>
 | number of letters ->      stat
 | info about letter ->      list [<N>]
 | body of letter    ->      retr <N>
 | delete letter     ->    delete <N>
 | first <M> strings ->       top <N> <M>
 | finish work       ->      exit
 | show menu         ->      menu
 | logout            ->    logout
 |============================|========================================|
```

```
login dm dm
Command is login
        +OK: login successfull
Enter command
stat
Command is stat
        +OK: 3
Enter command
list 1
Command is list
                Letter # 1
        from=jul
        to=dm
        Tue Jan 29 13:04:13 2013
Enter command
retr 1
Command is retr
                Letter # 1
        how are you

Enter command
send dm
Command is send
Input letter. Empty string to finish
first string
second

        +OK: letter saved
Enter command
stat
Command is stat
        +OK: 4
Enter command
top 4 1
Command is top
        first string
Enter command
exit
Command is exit
Sending last message to server
        +OK: bye-bye
```

Со стороны сервера:

```
$ ./tcpmail.exe S
Using default port: 7500
Server's menu:

¦=====Commands:==============¦============================¦
¦ show user list     ->      show
¦ kill user          ->      kill <N>
¦ finish work        ->      exit
¦ show menu          ->      menu
¦============================¦========================================¦
show
        Connected users are:
show
        Connected users are:
User # 1
IP:PORT = 127.0.0.1:49650
Not authorized
User # 2
IP:PORT = 127.0.0.1:49651
Not authorized
show
        Connected users are:
User # 1
IP:PORT = 127.0.0.1:49650
Name: dm
User # 2
IP:PORT = 127.0.0.1:49651
Name: jul
kill dm
Wrong index
kill 1
show
        Connected users are:
User # 2
IP:PORT = 127.0.0.1:49651
Name: jul
exit
Bye-bye
```

3. Выводы

Для создания приложения, взаимодействующих по сети используется технология сокетов. Прикладной протокол передачи данных обычно реализовывается поверх одного их транспортных протоколов стека TCP\IP – протокола TCP или протокола UDP.

Протокол TCP удобно использовать, если нет задачи максимизировать скорость передачи и нужно обеспечить надежность доставки данных.

Протокол UDP позволяет получить более быструю передачу пакетов, но не обеспечивает надежности. При необходимости эту функцию реализуют вышележащие протоколы.

Грамотное проектирование структуры программы очень важно, так как позволяет менять транспортные протоколы, не затрагивая большей части приложения.

Приложение 1. Почта поверх TCP

Client.c

```
#include "constants.h"
#include "clientCommandHandlers.h"
#include "stdio.h"
#include "connectionFunctions.h"
#include "commands.h"
void printMainMenu();
int clientMain(char* addr, char* port) {
    // создаем сокет и подключаем его к серверу
    int sock = createClient(addr, port);
    if(sock == -1) {
        printf("Can't connect to server\n");
        exit(1);
    }
    // переменные, для хранения ввода пользователя
    int res = 0;
    command cmd;
    initCommand(&cmd);
    dataList request, response;
    initDataList(&request);
    initDataList(&response);
    printMainMenu();
    for (;;) {
        clearCommand(&cmd);
        clearList(&request);
        clearList(&response);

        printf("Enter command\n");
        fgets(cmd.commandLine, DEFAULT_BUFLEN, stdin);
        cmd.commandLine[strlen(cmd.commandLine) - 1] = '\0';
        if (strlen(cmd.commandLine) == 0)
            continue;

        parseCommand(&cmd); // разбиваем команду на слова
        res = prepareRequest(&cmd, &request); // проверяем корректность ввода, и
если можем, то сами выполняем запрос
        if (res == -1 || res == 1) { // -1 - не корректный ввод, 1 - запрос сами
выполнили, не надо просить сервер
            continue;
        }
        res = sendAndRecv(sock, &request, &response);
        if (res == -1) {
            printf("Sorry. Connection is lost\n");
            break;
        }
        printDataList(&response); // пишем ответ сервера
        // проверяем не сообщение ли о завершении
        if (strcmp(response.data[0], BYE_STRING) == 0) {
            break;
        }
    }
    return 0;
}

void printMainMenu() {
    printf("\tMAIL CLIENT \n \n");
    printf("Main menu: \n \n");
    printf("|=====Commands:==============|==========================|\n");
    printf("| register new user -> %8s <NAME> <PASSWORD>    \n",REG_CMD);
```

```
    printf("| login             -> %8s <NAME> <PASSWORD>    \n",LOGIN_CMD);
    printf("| send letter       -> %8s <RECIPIENT>          \n",SEND_CMD);
    printf("| number of letters -> %8s                      \n",COUNT_CMD);
    printf("| info about letter -> %8s [<N>]                \n",INFO_CMD);
    printf("| body of letter    -> %8s <N>                  \n",BODY_CMD);
    printf("| delete letter     -> %8s <N>                  \n",DEL_CMD);
    printf("| first <M> strings -> %8s <N> <M>             \n",FIRST_LINES_CMD);
    printf("| finish work       -> %8s                      \n",EXIT_CMD);
    printf("| show menu         -> %8s                      \n",MENU_CMD);
    printf("| logout            -> %8s                      \n",LOGOUT_CMD);

printf("|===========================|=======================================|
\n");
}
```

clientCommandHandlers.c

```
#include <stdlib.h>
#include "clientCommandHandlers.h"
#include "stdio.h"
#include "commands.h"
extern void printMainMenu();
int checkCommand(command* c) {
    printf("Command is %s\n",c->arguments[0]);
    if (strcmp(c->arguments[0], LOGIN_CMD) == 0) {
        return c->argumentsSize == 3 ? 1 : 0;
    }
    if (strcmp(c->arguments[0], REG_CMD) == 0) {
        return c->argumentsSize == 3 ? 1 : 0;
    }
    if (strcmp(c->arguments[0], INFO_CMD) == 0) {
        if (c->argumentsSize == 1)
            return 1;
        if (c->argumentsSize != 2 || atoi(c->arguments[1]) <= 0) // если нет
аргумента, или аргумент не число
            return 0;
        return 1;
    }
    if (strcmp(c->arguments[0], COUNT_CMD) == 0) {
        if (c->argumentsSize != 1)
            return 0;
        return 1;
    }
    if (strcmp(c->arguments[0], BODY_CMD) == 0) {
        if (c->argumentsSize != 2)
            return 0;
        if (atoi(c->arguments[1]) <= 0)
            return 0;
        return 1;
    }
    if (strcmp(c->arguments[0], DEL_CMD) == 0) {
        if (c->argumentsSize != 2)
            return 0;
        if (atoi(c->arguments[1]) <= 0)
            return 0;
        return 1;
    }
    if (strcmp(c->arguments[0], FIRST_LINES_CMD) == 0) {
        if (c->argumentsSize != 3)
            return 0;
        if (atoi(c->arguments[1]) <= 0)
            return 0;
```

```
        if (atoi(c->arguments[2]) <= 0)
            return 0;
        return 1;
    }
    if (strcmp(c->arguments[0], MENU_CMD) == 0) {
        return c->argumentsSize == 1 ? 1 : 0;
    }
    if (strcmp(c->arguments[0], EXIT_CMD) == 0) {
        return c->argumentsSize == 1 ? 1 : 0;
    }
    if (strcmp(c->arguments[0], LOGOUT_CMD) == 0) {
        return c->argumentsSize == 1 ? 1 : 0;
    }
    if (strcmp(c->arguments[0], SEND_CMD) == 0) {
        return c->argumentsSize == 2 ? 1 : 0;
    }
    return 0;
}
/* Подготавливает запрос к отправке или в простейшем случае сам выполняет его
* возвращает -1 в случае ошибки, 1 - если сам выполнил и 0 если запрос подготовлен
* и его надо отослать
*/
int prepareRequest(command* c, dataList* request) {
    if (!checkCommand(c)) {
        printf("\tIncorrect command or arguments\n");
        return -1;
    }
    if (strcmp(c->arguments[0], MENU_CMD) == 0) { // если просят показать меню,
просто показываем и ничего серверу не шлем
        printMainMenu();
        return 1;
    }
    if (strcmp(c->arguments[0], EXIT_CMD) == 0) { //
        printf("Sending last message to server\n");
    }
    addToList(request, c->commandLine);
    if (strcmp(c->arguments[0], SEND_CMD) == 0) { // если хотят отправить письмо,
то сначала считываем его тело из консоли
        readLetter(request);
    }
    return 0;}
int readLetter(dataList* dl) {
    printf("Input letter. Empty string to finish\n");
    char line[DEFAULT_BUFLEN];
    for (;;) {
        bzero(line, DEFAULT_BUFLEN);
        fgets(line, DEFAULT_BUFLEN, stdin);
        line[strlen(line) - 1] = '\0';
        if (strlen(line) == 0) {
            addToList(dl, ".");
            break;
        }
        addToList(dl, line);
    }
    return 1;
}
```

commands.h

```
#ifndef COMMANDS_H
#define COMMANDS_H
// команды, передающиеся клиентом серверу (если больше 8 символов -> поправить
функцию printMainMenu
#define LOGIN_CMD "login"
#define REG_CMD "reg"
#define INFO_CMD "list"
#define COUNT_CMD "stat"
#define BODY_CMD "retr"
#define DEL_CMD "delete"
#define FIRST_LINES_CMD "top"
#define MENU_CMD "menu"
#define EXIT_CMD "exit"
#define LOGOUT_CMD "logout"
#define SEND_CMD "send"
// команды консоли сервера (если больше 8 символов -> поправить функцию
printServerMenu
#define SHOW_SERV_CMD "show"
#define KICK_SERV_CMD "kill"
#define EXIT_SERV_CMD "exit"
#define MENU_SERV_CMD "menu"
// команды, которые шлет сервер клиенту
#define BYE_STRING "+OK: bye-bye"
#endif /* COMMANDS_H */
```

connectionFunction.h

```
#ifndef CONNECTIONFUNCTIONS_H
#define     CONNECTIONFUNCTIONS_H

#include <errno.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <pthread.h>
#include <time.h>

#include <netdb.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/uio.h>

#include "myTypes.h"


int createClient(char* hostName, char* portName);
int createServer(char* hostName, char* portName);


void setAddress(char* hostName, char* portName, struct sockaddr_in* info, char*
protocol);

int sendResponse(int sock, dataList* dl);
int sendLine(int sock, char* line);
int recvLine(int sock, char* buffer, int buffSize);

int sendAndRecv(int sock, dataList* request, dataList* response);
```

```
void printAddrInfo(struct sockaddr_in* addr);
int recvFix(int sock, char* buf, int size, int flags);
#endif      /* CONNECTIONFUNCTIONS_H */
```

connectionFunction.c

```
#include "connectionFunctions.h"


/* шлем клиенту все, что хранится в dataList,после каждой посланной строчки
 * ожидаем очередного запроса, кроме строчки вида ".", сигнализирующей о конце
передачи
 * Если послать не удалось - возвращает -1
 */
int sendResponse(int sock, dataList* dl) {
    int res = 0;
    int i = 0;
    for (i = 0; i < dl->length; ++i) {
        res = sendLine(sock,dl->data[i]);
        if (res == -1) {
            return -1;
        }
    }
    return 0;
}


/*используется клиентом. Шлет в общем случае несколько строк запроса из параметра
 * request  и принимает в общем случае несколько строк ответа, которые сохраняет
в
 * параметр response
 */
int sendAndRecv(int sock, dataList* request, dataList* response) {
    int i = 0;
    int res = 0;
    char buf[DEFAULT_BUFLEN];
    for (i = 0; i < request->length; ++i) {
        bzero(buf, DEFAULT_BUFLEN);
        int res = sendLine(sock, request->data[i]);
        if (res <= 0)
            return -1;
        res = recvLine(sock, buf, DEFAULT_BUFLEN);
        if (res <= 0)
            return -1;

        if (strcmp(buf, ".") == 0) { // если сервер говорит об окончании передачи,
то шлем ему следующую строку
            continue;
        }
        addToList(response, buf); // сохраняем первую строку ответа и дочитываем
остальной
        for (;;) { // если пришла не точка, значит сервер еще будет досылать
ответ. Ждем его
            bzero(buf, DEFAULT_BUFLEN);
            res = recvLine(sock, buf, DEFAULT_BUFLEN);
            if (res <= 0)
                return -1;
            // если сервер говорит об окончании передачи, выходим из внутреннего
цикла и шлем оставшиеся строки запроса
            if (strcmp(buf, ".") == 0) {
                break;
            }
```

```
            addToList(response, buf);
        }
    }
}

void printAddrInfo(struct sockaddr_in* addr) {
    printf("Addr        info:        %s:%d\n",        inet_ntoa(addr->sin_addr),
ntohs(addr->sin_port));
}

int sendLine(int sock, char* str) {

    char tempBuf[DEFAULT_BUFLEN];
    strcpy(tempBuf, str);
    char lineDelim[2];
    sprintf(lineDelim, "%c", LINE_DELIM);
    strcat(tempBuf, lineDelim);

    int res = send(sock, tempBuf, strlen(tempBuf), 0);
    return res;
}

int recvFix(int sock, char* buf, int size, int flags) {
    if (size == 0)
        return 0;
    return recv(sock, buf, size, flags | MSG_WAITALL);
}

int createClient(char* hostName, char* portName) {
    struct sockaddr_in clientInfo;
    int clientSocket;
    setAddress(hostName, portName, &clientInfo, "tcp");
    clientSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (clientSocket < 0) {
        perror("Client: error while using socket");
        exit(1);
    }
    if   (connect(clientSocket,   (struct   sockaddr*)   &clientInfo,   sizeof
(clientInfo))) {
        perror("Client: error while using connect");
        exit(1);
    }
    return clientSocket;
}

void setAddress(char* hostName, char* portName, struct sockaddr_in* info, char*
protocol) {
    bzero(info, sizeof (info));
    info->sin_family = AF_INET;
    if (hostName != NULL) {
        if (!inet_aton(hostName, &info->sin_addr)) {
            // mb change
            perror("Unknown host ");
            exit(1);
        }
    } else {
        info->sin_addr.s_addr = htonl(INADDR_ANY);
    }
    char* endptr;
    short port = strtol(portName, &endptr, 0);
    if (*endptr == '\0')
        info->sin_port = htons(port);
    else {
```

```
        // mb change
        perror("Unknown port ");
        exit(1);
    }
}

int createServer(char* hostName, char* portName) {
    struct sockaddr_in servInfo;
    int servSocket;
    setAddress(hostName, portName, &servInfo, "tcp");
    servSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (servSocket < 0) {
        perror("Server: Error while using socket");
        exit(1);
    }
    if (bind(servSocket, (struct sockaddr*) &servInfo, sizeof (servInfo))) {
        perror("Server: Error while using bind");
        exit(1);
    }
    if (listen(servSocket, BACKLOG)) {
        perror("Server: Error while using listen");
        exit(1);
    }
    return servSocket;
}

int recvLine(int sock, char* buffer, int buffSize) {
    char* buff = buffer;
    char* currPosPointer;
    int count = 0;
    int strSize = 0;
    char tempBuf[1500];
    char currChar;
    while (--buffSize > 0) {
        if (--count <= 0) {
            count = recvFix(sock, tempBuf, strSize, 0);
            if (count != strSize) {
                printf("Very strange error... readLine\n");
                return -1;
            }
            count = recv(sock, tempBuf, sizeof (tempBuf), MSG_PEEK);

            if (count <= 0)
                return count;
            currPosPointer = tempBuf;
            strSize = 0;
        }
        currChar = *currPosPointer++;
        *buffer++ = currChar;
        ++strSize;
        if (currChar == LINE_DELIM) {
            *(buffer - 1) = '\0';
            count = recvFix(sock, tempBuf, strSize, 0);
            if (count != strSize) {
                printf("Very strange error... readLine\n");
                return -1;
            }
            return buffer - buff;
        }
    }
    return -1;
}
```

constants.h

```
#ifndef CONSTANTS_H
#define     CONSTANTS_H

#include "stddef.h"
#include "string.h"
#define LINE_DELIM '\n'

#define DEFAULT_PORT "7500"
#define DEFAULT_SERV_ADDR "127.0.0.1"

#define DEFAULT_BUFLEN 512
#define MAX_ARGS 4 // максимальное количество аргументов в команде
#define MAX_ARG_SIZE 256 // максимальная длинна одного аргумента команды
#define NUM_MSG_LENGTH 7 // максимальное количество разрядов в номере сообщения
#define REQUEST_TIMEOUT 1000 // максимальное время между запросами от клиента.
(после этого кикаем)
#define REQUEST_TRIES 4 // сколько раз клиент пытается получить ответ на свой
запрос. (после каждой попытки таймаут увеличивается на 1 секуду)
#define BACKLOG 5

#define MAX_USERS 2
#define USERS_FILE "mail/users.txt"
#define MAX_LETTERS_STRINGS 40

#define HEADERS_DELIM "----"
#define HEADERS_DELIM_LENGTH 4

#endif      /* CONSTANTS_H */
```

fileSystemFunctions.c

```
#include "fileSystemFunctions.h"
#include "clientCommandHandlers.h"
#include "dirent.h"
#include "stdio.h"
#include "time.h"
#include "pthread.h"
extern pthread_mutex_t file_system_lock;
// возвращает количество писем пользователя. Если не может подсчитать, к примеру
нет папки, возвращает -1
int getLettersCount(char* userName) {
    // открываем папку пользователя
    char pathToFolder[DEFAULT_BUFLEN];
    bzero(pathToFolder, DEFAULT_BUFLEN);
    sprintf(pathToFolder, "mail/%s", userName);
    DIR* d = opendir(pathToFolder);
    if (d == NULL) {
        return -1;
    }
    int lettersCount = 0;
    // получаем все записи в папке, и если это текстовый файл, увеличиваем счетчик
http://www.ibm.com/developerworks/ru/library/au-unix-readdir/
    struct dirent entry;
    struct dirent* result;
    readdir_r(d, &entry, &result);
    while (result) {
        //получаем инфу о файле( неизвестно есть тако поле на всех системах !!!!!)
        if                (entry.d_type                ==                DT_REG)
//http://www.delorie.com/gnu/docs/glibc/libc_270.html
            lettersCount++;
```

```
        // получаем инфу о следующем файле
        readdir_r(d, &entry, &result);
    }
    closedir(d);
    return lettersCount;
}

// записывает заголовки письма в стурктуру типа dataList
// ничего не проверяет, т.к. вызывающая функция должна была уже это сделать
void getInfoAboutLetter(user* client, dataList* dl, int letterNumber) {
    dataList tempArray;
    clearList(&tempArray);
    // получаем все письмо во временный массив
    getWholeLetter(client, letterNumber, &tempArray);
    // проверяем удалось ли прочитать файл
    if (tempArray.length == 0) {
        addToList(dl, " -ERR: can't read file or file is empty");
        return;
    }
    int i;
    char tb[DEFAULT_BUFLEN];
    bzero(tb, DEFAULT_BUFLEN);
    sprintf(tb, "\tLetter # %d", letterNumber);
    addToList(dl, tb);
    for (i = 0; i < tempArray.length; ++i) {
        // если дошли до конца заголовков то прекращаем(подразумевается что
заголовки есть ВСЕГДА!)
        if (strncmp(tempArray.data[i], HEADERS_DELIM, HEADERS_DELIM_LENGTH) == 0)
{
            break;
        }
        // иначе копируем в основной результирующий массив
        addToList(dl, tempArray.data[i]);
    }
}
// ничего не проверяет, т.к. вызывающая функция должна была уже это сделать

void getLetterBody(user* client, int letterNumber, dataList* dl) {
    dataList tempArray;
    clearList(&tempArray);
    // получаем все письмо во временный массив
    getWholeLetter(client, letterNumber, &tempArray);
    // проверяем удалось ли прочитать файл
    if (tempArray.length == 0) {
        addToList(dl, " -ERR: can't read file");
        return;
    }
    char tb[DEFAULT_BUFLEN];
    bzero(tb, DEFAULT_BUFLEN);
    sprintf(tb, "\tLetter # %d", letterNumber);
    addToList(dl, tb);
    int i;
    int isHeader = 1; // флаг, показывающий что сейчас читаются заголовки
    for (i = 0; i < tempArray.length; ++i) {
        // если наткнулись на конец заголовков
        if (strncmp(tempArray.data[i], HEADERS_DELIM, HEADERS_DELIM_LENGTH) == 0
&& isHeader == 1) {
            isHeader = 0;
            continue;
        }
        if (isHeader == 1)
            continue; //  пропускаем заголовок
        // сюда приходим, если читаем само тело письма
```

```
        addToList(dl, tempArray.data[i]);
    }
}

//возвращает 0, если удалось удалить или -1 если нет
int deleteLetter(char* userName, int letterNumber) {
    char fileName[DEFAULT_BUFLEN];
    bzero(fileName, DEFAULT_BUFLEN);
    sprintf(fileName, "mail/%s/%d", userName, letterNumber);
    if (remove(fileName)) {
        return -1;
    }
    pthread_mutex_lock(&file_system_lock);
    shiftFiles(userName, letterNumber);
    pthread_mutex_unlock(&file_system_lock);
    return 0;
}

// возвращает файл в виде массива строк, проверок не выполняет!
// если возникла ошибка, ничего не добавляет в массив, а просто прекращает
выполнение
void getWholeLetter(user* client, int letterNumber, dataList* result) {
    // пытаемся открыть файл
    char buf[DEFAULT_BUFLEN];
    bzero(buf, DEFAULT_BUFLEN);
    sprintf(buf, "mail/%s/%d", client->name, letterNumber);
    FILE* f = fopen(buf, "r");
    if (f == NULL) {
        return;
    }
    // если файл все же открыли читаем построчно и сохраняем в массив
    while (!feof(f)) {
        bzero(buf, DEFAULT_BUFLEN);
        fgets(buf, DEFAULT_BUFLEN, f);
        buf[strlen(buf) - 1] = '\0'; // а если ничего не прочитали?
        addToList(result, buf);
    }
    fclose(f);
}
// записывает в массив result первые maxStrings ТЕЛА письма
void getLetterHead(user* client, int letterNumber, int maxStrings, dataList*
result) {
    dataList tempArray;
    clearList(&tempArray);
    // получаем все письмо во временный массив
    getWholeLetter(client, letterNumber, &tempArray);
    // проверяем удалось ли прочитать файл
    if (tempArray.length == 0) {
        addToList(result, " -ERR: can't read file");
        return;
    }
    int i;
    int curStrings = 0;
    int isHeader = 1; // флаг, показывающий что сейчас читаются заголовки
    for (i = 0; i < tempArray.length; ++i) {
        // если наткнулись на конец заголовков
        if (strncmp(tempArray.data[i], HEADERS_DELIM, HEADERS_DELIM_LENGTH) == 0
&& isHeader == 1) {
            isHeader = 0;
            continue;
        }
        if (isHeader == 1)
            continue; //  пропускаем заголовок
```

```
        // сюда приходим, если читаем само тело письма
        if (curStrings >= maxStrings) {
            break;
        } else {
            addToList(result, tempArray.data[i]);
            curStrings++;
        }
    }
}
int saveLetter(char* from, char* to, dataList* tmpLetter) {
    int i = getFirstFreeNumber(to);
    char fileName[DEFAULT_BUFLEN];
    bzero(fileName, DEFAULT_BUFLEN);
    sprintf(fileName, "mail/%s/%d", to, i);
    FILE* f = fopen(fileName, "w");
    if (f == NULL)
        return -1;
    writeHeaders(f, from, to);
    for (i = 0; i < tmpLetter->length; ++i) {
        fputs(tmpLetter->data[i], f);
        fputs("\n", f);
    }
    fclose(f);
    return 0;
}
void writeHeaders(FILE* f, char* from, char* to) {
    char buf[DEFAULT_BUFLEN];
    bzero(buf, DEFAULT_BUFLEN);

    sprintf(buf, "from=%s\n", from);
    fputs(buf, f);

    bzero(buf, DEFAULT_BUFLEN);
    sprintf(buf, "to=%s\n", to);
    fputs(buf, f);

    bzero(buf, DEFAULT_BUFLEN);
    getCurTime(buf);
    fputs(buf, f);

    bzero(buf, DEFAULT_BUFLEN);
    sprintf(buf, "%s\n", HEADERS_DELIM);
    fputs(buf, f);
}

void getCurTime(char* buf) {
    time_t rawtime;
    struct tm * timeinfo;
    time(&rawtime);
    timeinfo = localtime(&rawtime);
    strcpy(buf, (asctime) (timeinfo));

}

int getFirstFreeNumber(char* to) {
    int i = 1;
    FILE *f;
    char buf[DEFAULT_BUFLEN];
    for (;;) {
        bzero(buf, DEFAULT_BUFLEN);
        sprintf(buf, "mail/%s/%d", to, i);
        f = fopen(buf, "r");
        if (f == NULL)
```

```
            return i;
        fclose(f);
        ++i;
    }
}
void shiftFiles(char* userName, int letterNumber) {
    char newFileName[DEFAULT_BUFLEN];
    bzero(newFileName, DEFAULT_BUFLEN);
    char oldFileName[DEFAULT_BUFLEN];
    bzero(oldFileName, DEFAULT_BUFLEN);

    sprintf(newFileName, "mail/%s/%d", userName, letterNumber);
    sprintf(oldFileName, "mail/%s/%d", userName, letterNumber + 1);
    int i = letterNumber + 1;
    while (!rename(oldFileName, newFileName)) {
        ++i;
        bzero(newFileName, DEFAULT_BUFLEN);
        bzero(oldFileName, DEFAULT_BUFLEN);
        strcpy(newFileName, oldFileName);
        sprintf(newFileName, "mail/%s/%d", userName, i);
    }
}
```

fileSystemFunctions.h

```
#ifndef FILESYSTEMFUNCTIONS_H
#define     FILESYSTEMFUNCTIONS_H

#include "myTypes.h"
#include "constants.h"
#include "stdio.h"

int getLettersCount(char* userName);
void getLetterHead(user* client, int letterNumber, int maxStrings, dataList*
result);
void getInfoAboutLetter(user* client, dataList* dl, int letterNumber);
void getLetterBody(user* client, int letterNumber, dataList* dl);
void getWholeLetter(user* client, int letterNumber, dataList* result);

int deleteLetter(char* userName, int letterNumber);
int saveLetter(char* from, char* to, dataList* tmpLetter);

// вспомогательные функции
void writeHeaders(FILE* f, char* from, char* to);
void getCurTime(char* buf);
int getFirstFreeNumber(char* to);
void shiftFiles(char* userName, int letterNumber);

#endif      /* FILESYSTEMFUNCTIONS_H */
```

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "connectionFunctions.h"

int main(int argc, char** argv) {
    if (argc < 2) {
        printf("You should use at least key S or C");
       exit(0);
    }
    // если хотят запустить сервер
```

```
    if (strncmp(argv[1], "S", 1) == 0) {
        //если не задан порт
        if (argc == 2) {
            printf("Using default port: %s\n", DEFAULT_PORT);
            serverMain(DEFAULT_PORT);
        } else {
            serverMain(argv[2]);
        }
        // если хотят запустить клиент
    } else if (strncmp(argv[1], "C", 1) == 0) {
        // если не задано ни адреса ни порта
        if (argc == 2) {
            printf("Using defalut ip and port: %s: %s\n", DEFAULT_SERV_ADDR,
DEFAULT_PORT);
            clientMain(DEFAULT_SERV_ADDR, DEFAULT_PORT);
        } else if (argc == 3) {
            // если задан только адрес
            printf("Using defalut port: %s:\n", DEFAULT_PORT);
            clientMain(argv[2], DEFAULT_PORT);
        } else {
            clientMain(argv[2], argv[3]);
        }
    } else {
        // если неизвестный ключ запуска
        printf("Illegal first argument '%s'. First argument must be 'S' or 'C'\n",
argv[1]);
        exit(0);
    }
    return 0;
}
```

myTypes.c

```
#include <malloc.h>
#include "myTypes.h"

void addToList(dataList* dl, char* string) {
    if (dl->length == MAX_LETTERS_STRINGS) {
        printf("Error: trying to add string to full list\n");
        return;
    }
    strcpy(dl->data[(dl->length)], string);
    ++(dl->length);
}

void clearList(dataList* dl) {
    dl->length = 0;
    int i;
    for (i = 0; i < MAX_LETTERS_STRINGS; ++i) {
        bzero(dl->data[i], DEFAULT_BUFLEN); // можно оптимизировать
    }
}

void printDataList(dataList* dl) {
    int i;
    for (i = 0; i < dl->length; ++i) {
        printf("\t%s\n", dl->data[i]);
    }
}

void initCommand(command* c) {
    //   printf("Inside init command\n");
```

```
    bzero(c->commandLine, DEFAULT_BUFLEN);
    int i = 0;
    //   printf("\tgoing to malloc %d arguments\n",MAX_ARGS);
    c->arguments = (char**) malloc(MAX_ARGS * sizeof (char*));
    for (i = 0; i < MAX_ARGS; ++i) {
        c->arguments[i] = (char*) malloc(MAX_ARG_SIZE * sizeof (char));
    }
    c->argumentsSize = 0;
    //   printf("Exiting init command\n");
}

void initDataList(dataList* dl) {
    //    printf("Inside init data list\n");
    dl->length = 0;
    int i;
    for (i = 0; i < MAX_LETTERS_STRINGS; ++i) {
        bzero(dl->data[1], DEFAULT_BUFLEN);
    }
}

void clearCommand(command* c) {
    bzero(c->commandLine, DEFAULT_BUFLEN);
    int i;
    for (i = 0; i < c->argumentsSize; ++i) {
        bzero(c->arguments[i], DEFAULT_BUFLEN);
    }
}

void parseCommand(command* c) {
    if (c->commandLine == NULL || strlen(c->commandLine) == 0) {
        printf("Error. Tring to parse empty command\n");
        return;
    }
    c->argumentsSize = 0;
    //   о   функции   strtok_r   -   http://unixcoding.blogspot.ru/2010/10/blog-
post_679.html
    char *tmpCommand = strdup(c->commandLine);
    char *last;
    char* token = strtok_r(tmpCommand, " ", &last); // считываем саму команду
    strcpy(c->arguments[0], token); // и копируем ее как первый аргумент
    ++c->argumentsSize;
    int i = 0;
    for (i = 1; i < MAX_ARGS; ++i) {
        token = strtok_r(NULL, " ", &last);
        if (token == NULL)
            break;
        strcpy(c->arguments[i], token);
        ++c->argumentsSize;
    }
    free(tmpCommand);
}
```

myTypes.h

```
#ifndef MYTYPES_H
#define     MYTYPES_H

#include "constants.h"

typedef struct {
    char name[DEFAULT_BUFLEN];
```

```
    int sock;
    int isLogged;
    int index;
    char *ipAddr[DEFAULT_BUFLEN];
    int port;
    char* serverIP[DEFAULT_BUFLEN];
    int serverPort;
} user;

typedef struct {
    int length;
    char data[MAX_LETTERS_STRINGS][DEFAULT_BUFLEN];
} dataList;

typedef struct {
    char commandLine[DEFAULT_BUFLEN];
    char** arguments;
    int argumentsSize;
} command;

typedef int (*servCmdHndlrPtr)(user* client, command* cmd, dataList* response);
typedef int (*consoleCmdHndlrPtr)(command* cmd);
typedef int (*cmdHndlrPtr)(command* c, dataList* dl);

void addToList(dataList* dl, char* string);
void printDataList(dataList* dl);
void initDataList(dataList* dl);
void clearList(dataList* dl);
void initCommand(command* c);
void clearCommand(command* c);
void parseCommand(command* c);
#endif      /* MYTYPES_H */
```

server.c

```
#include <unistd.h>
#include "constants.h"
#include "serverCommandHandlers.h"
#include "stdio.h"
#include "serverConsoleCommandHandlers.h"
#include "connectionFunctions.h"
#include "serverUtils.h"
#include "pthread.h"
int initUserList();
int getFreePlace();
int addClient(struct sockaddr_in* clientAddr, int clientLen);

static void *clientHandler(void *args);
static void *consoleHandler(void *args);

user userlist[MAX_USERS];
pthread_mutex_t userlist_lock;
pthread_mutex_t users_file_lock;
pthread_mutex_t file_system_lock;

int serverMain(char* port) {
    pthread_mutex_init(&userlist_lock, NULL);
    pthread_mutex_init(&users_file_lock, NULL);
    pthread_mutex_init(&file_system_lock, NULL);
    initUserList();
    // создаем поток, обрабатывающий консольный ввод
    pthread_t ConsoleThread;
```

```
    if (pthread_create(&ConsoleThread, NULL, consoleHandler, NULL) != 0) {
        printf("Can't create console thread\n");
        exit(1);
    }
    // создаем слушающий сокет
    int listenSocket = createServer(NULL, port);
    char buf[DEFAULT_BUFLEN];
    // принимаем подключения
    struct sockaddr_in clientAddr;
    int clientAddrSize = sizeof (clientAddr);
    for (;;) {
        int newClient = accept(listenSocket, (struct sockaddr*) &clientAddr,
&clientAddrSize);
        if (newClient < 0) {
            perror("error while using accept");
            break;
        }
        int i = getFreePlace();
        if (i == -1) {
            close(newClient);
            continue;
        }
        pthread_mutex_lock(&userlist_lock);

        userlist[i].sock = newClient;
        strcpy(userlist[i].ipAddr, inet_ntoa(clientAddr.sin_addr));
        userlist[i].port = ntohs(clientAddr.sin_port);
        userlist[i].index = i;

        pthread_mutex_unlock(&userlist_lock);
        pthread_t tr1;
        if (pthread_create(&tr1, NULL, clientHandler, (void*) i) != 0) {
            printf("cant create clients thread\n");
            exit(0); //ìîæåò íå ñòîèò òàê ðåçêî ïî exit âûõîäèòü?
        }
    }
    return 0;
}

static void *clientHandler(void *args) {
    int index = (int) args;
    pthread_mutex_lock(&userlist_lock);
    user client = userlist[index]; // don't need mutex,?
    pthread_mutex_unlock(&userlist_lock);

    // структура для хранения принятой команды
    command cmd;
    initCommand(&cmd);
    // структура для хранения ответа пользователю
    dataList response;
    initDataList(&response);
    // указатель на функцию, обработчик текущей команды
    servCmdHndlrPtr currentCommand;
    // флаг, сигнализирующий о конце работы
    int end = 0;
    do {
        clearCommand(&cmd); // очищаем прошлую команду
        clearList(&response);
        int res = recvLine(client.sock,cmd.commandLine,DEFAULT_BUFLEN);
        if (res == -1) {
            break;
        }
        parseCommand(&cmd); // выделяем аргументы команды
```

```
        currentCommand = selectHandler(cmd.arguments[0]); // получаем указатель
на функцию обработчик комады
        if  ((*currentCommand)(&client,  &cmd,  &response)  ==  -1)  //  выполняем
команду
            end = 1; // если команда вернула ненулевой код, значит возникли ошибки
при передаче данных
    } while (end == 0);
    // закрываем сокет клиента и сигнализируем что новый клиент может подключится
    closeClientByIndex(index);
    return 0;
}

static void *consoleHandler(void *args) {
    // делаем stdin неблокирующи
    int flags = fcntl(F_GETFL, 0);
    flags = flags | O_NONBLOCK;
    fcntl(0, F_SETFL, flags);

    // создаем структуру, где будем хранить пользовательский ввод
    command cmd;
    initCommand(&cmd);
    consoleCmdHndlrPtr currentCommand; // указатель на текущую команду
    printServerMenu();
    int end = 0;
    do {
        clearCommand(&cmd); // очищаем прошлую команду
        fgets(cmd.commandLine, DEFAULT_BUFLEN, stdin); // считываем новую команду
        cmd.commandLine[strlen(cmd.commandLine) - 1] = '\0'; // fgets считывает
вместе с концом строки, который нам не нужен
        if (strlen(cmd.commandLine) == 0)
            continue;
        parseCommand(&cmd); // выделяем аргументы команды
        currentCommand  =  selectConsoleCommand(cmd.arguments[0]);  //  получаем
указатель на функцию обработчик комады
        if ((*currentCommand)(&cmd) != 0) // выполняем команду
            end = 1;
    } while (end == 0);
    closeAllSockets();
    printf("Bye-bye\n");
    exit(0);
}

int initUserList() {
    int i = 0;
    for (i = 0; i < MAX_USERS; i++) {
        userlist[i].sock = -1;
        userlist[i].port = -1;
        userlist[i].serverPort = -1;
        userlist[i].index = -1;
        userlist[i].isLogged = 0;
        bzero(userlist[i].ipAddr, sizeof (userlist[i].ipAddr));
        bzero(userlist[i].name, sizeof (userlist[i].name));
        bzero(userlist[i].serverIP, sizeof (userlist[i].ipAddr));
    }
}
// возвращаем первое свободное место в массиве клиентов
int getFreePlace() {
    // mutex вроде не нужен, это ведь только чтение + менять может только этот
поток
    int i;
    pthread_mutex_lock(&userlist_lock);
    for (i = 0; i < MAX_USERS; i++) {
        if (userlist[i].sock == -1) {
```

```
            pthread_mutex_unlock(&userlist_lock);
            return i;
        }
    }
    pthread_mutex_unlock(&userlist_lock);
    return -1;
}
```

serverCommandHandlers.c

```
#include "serverCommandHandlers.h"
#include "myTypes.h"
#include "connectionFunctions.h"
#include "serverUtils.h"
#include "fileSystemFunctions.h"
#include "commands.h"
extern pthread_mutex_t userlist_lock;
extern user userlist[MAX_USERS];
// выбирает обработчик команды, заданной в параметре command
servCmdHndlrPtr selectHandler(char* command) {

    if (strcmp(command, LOGIN_CMD) == 0)
        return loginCmdHandler;
    if (strcmp(command, REG_CMD) == 0)
        return regCmdHandler;
    if (strcmp(command, INFO_CMD) == 0)
        return infoCmdHandler;
    if (strcmp(command, COUNT_CMD) == 0)
        return countCmdHandler;
    if (strcmp(command, BODY_CMD) == 0)
        return bodyCmdHandler;
    if (strcmp(command, DEL_CMD) == 0)
        return delCmdHandler;
    if (strcmp(command, FIRST_LINES_CMD) == 0)
        return firstLinesCmdHandler;
    if (strcmp(command, EXIT_CMD) == 0)
        return exitCmdHandler;
    if (strcmp(command, LOGOUT_CMD) == 0)
        return logoutCmdHandler;
    if (strcmp(command, SEND_CMD) == 0)
        return sendCmdHandler;
    return defaultCmdHandler;
}
/* обработчики команд. Список команд в файле commands.h
 * Выполняют обработку команды заданной параметром request и отсылают результат
пользователю
 * данные о котором хранит параметр client. Параметр response используется для
хранения
 * промежуточного результата и номера сообщения для посылки клиенту
 * Возвращают -1 если связь с клиентом утеряна
 */
int loginCmdHandler(user *client, command* request, dataList* response) {
    pthread_mutex_lock(&userlist_lock);
    int alreadyWorking = isUserPresentNow(request->arguments[1]);
    pthread_mutex_unlock(&userlist_lock);
    if (alreadyWorking == 1) {
        addToList(response, "-ERR: user with this name is working now");
    } else if (client->isLogged == 1) {
        addToList(response, "-ERR: logout first");
    } else if (checkPass(request->arguments[1], request->arguments[2])) {
        addToList(response, "-ERR: wrong user name or password");
    } else {
```

```
        addToList(response, "+OK: login successfull");
        userLoggin(client, request->arguments[1]);
    }
    addToList(response, ".");
    return sendResponse(client->sock, response);
}
int regCmdHandler(user *client, command* request, dataList* response) {
    if (client->isLogged == 1) {
        addToList(response, "-ERR: logout first");
    } else if (addUsr(request->arguments[1], request->arguments[2]) != 0) {
        addToList(response, "-ERR: user with this name already presents");
    } else {
        addToList(response, "+OK: register successfull");
        userLoggin(client, request->arguments[1]);
    }
    addToList(response, ".");
    return sendResponse(client->sock, response);
}
int countCmdHandler(user *client, command* request, dataList* response) {
    if (client->isLogged == 0) {
        addToList(response, "-ERR: login first");
    } else {
        int lettersCount = getLettersCount(client->name);
        char tmp[DEFAULT_BUFLEN];
        lettersCount == -1 ? sprintf(tmp, "-ERR: can't read directory") :
sprintf(tmp, "+OK: %d", lettersCount);
        addToList(response, tmp);
    }
    addToList(response, ".");
    return sendResponse(client->sock, response);
}
int infoCmdHandler(user* client, command* request, dataList* response) {
    if (client->isLogged == 0) {
        addToList(response, "-ERR: login first");
    } else {
        //проверяем не пуста ли папка
        int numLetters = getLettersCount(client->name);
        if (numLetters == 0) {
            addToList(response, "+OK: directory is empty");
        } else {
            // если просят информацию обо всех письмах
            if (request->argumentsSize == 1) {
                int i;
                for (i = 1; i <= numLetters; ++i) { //TODO CHECKKKKK!
                    getInfoAboutLetter(client, response, i);
                    printDataList(response);
                    if (sendResponse(client->sock, response) == -1) {
                        return -1;
                    }
                    clearList(response);
                }
            } else { // если просят информацию только об одном письме
                // проверяем корректный ли номер письма
                if (atoi(request->arguments[1]) > numLetters ||
atoi(request->arguments[1]) <= 0) { // вторая проверка на всякий случай
                    addToList(response, "-ERR: no letter with this number");
                } else
                    getInfoAboutLetter(client, response,
atoi(request->arguments[1]));
            }
        }
    }
    addToList(response, ".");
```

```
    return sendResponse(client->sock, response);
}
int bodyCmdHandler(user *client, command* request, dataList * response) {
    if (client->isLogged == 0) {
        addToList(response, "-ERR: login first");
    } else {
        // проверяем есть ли вобще такое письмо
        int letterNumber = atoi(request->arguments[1]);
        int lettersCount = getLettersCount(client->name);
        if (letterNumber > lettersCount) { // пусть письма нумеруются с 1, главное
не забыть потом
            addToList(response, "-ERR: there is no letter with this number");
        } else {
            // если такое письмо есть, надо его послать
            getLetterBody(client, letterNumber, response);
        }
    }
    addToList(response, ".");
    return sendResponse(client->sock, response);
}
int delCmdHandler(user *client, command* request, dataList * response) {
    if (client->isLogged == 0) {
        addToList(response, "-ERR: login first");
    } else {
        // проверяем есть ли вобще такое письмо
        int letterNumber = atoi(request->arguments[1]);
        int lettersCount = getLettersCount(client->name);
        if (letterNumber > lettersCount) { // пусть письма нумеруются с 1, главное
не забыть потом
            addToList(response, "-ERR: there is no letter with this number");
        } else { // пытаемся удалить писмьо
            int res = deleteLetter(client->name, letterNumber);
            addToList(response, res == 0 ? "+OK: letter deleted" : "-ERR: some
error while deleting letter");
        }
    }
    addToList(response, ".");
    return sendResponse(client->sock, response);
}
int firstLinesCmdHandler(user *client, command* request, dataList * response) {
    if (client->isLogged == 0) { // если не зарегистрирован, шлем ошибку
        addToList(response, "-ERR: login first");
    } else {
        // проверяем есть ли вобще такое письмо
        int letterNumber = atoi(request->arguments[1]);
        int lettersCount = getLettersCount(client->name);
        if (letterNumber > lettersCount) { // пусть письма нумеруются с 1, главное
не забыть потом
            addToList(response, "-ERR: there is no letter with this number");
        } else {
            // если есть, надо отдать первые M строк этого письма
            getLetterHead(client,   letterNumber,   atoi(request->arguments[2]),
response);
        }
    }
    addToList(response, ".");
    return sendResponse(client->sock, response);
}
int defaultCmdHandler(user *client, command* request, dataList * response) {
    addToList(response, "-ERR: unrecognized command");
    addToList(response, ".");
    return sendResponse(client->sock, response);
```

```
}
int exitCmdHandler(user *client, command* request, dataList * response) {
    addToList(response,BYE_STRING);
    addToList(response, ".");
    sendResponse(client->sock, response);
    return -1;
}
int logoutCmdHandler(user *client, command* request, dataList * response) {
    if (client->isLogged == 0) {
        addToList(response, "-ERR: login first");
    } else {
        userLoggout(client);
        addToList(response, "+OK: you are not logged now");
    }
    addToList(response, ".");
    return sendResponse(client->sock, response);
}
int sendCmdHandler(user *client, command* request, dataList * response) {
    if (client->isLogged == 0) { // если не зарегистрирован, шлем ошибку
        addToList(response, "-ERR: login first");
        addToList(response, ".");
        return sendResponse(client->sock, response);
    }
    // проверяем зарегистрирован ли получател
    int res = isUsrRegistred(request->arguments[1]);
    if (res != 0) {// если получатель не зарегестрирован у нас
        addToList(response, "-ERR: no such recipient");
        addToList(response, ".");
        return sendResponse(client->sock, response);
    }
    // шлем подтвержение полученного запроса
    addToList(response, ".");
    if (sendResponse(client->sock, response) == -1) {
        return -1;
    }
    char buf[DEFAULT_BUFLEN];
    bzero(buf,DEFAULT_BUFLEN);
    dataList tmpLetter;
    initDataList(&tmpLetter);
    // принимаем тело письма
    for (;;) {
        clearList(response);
        bzero(buf,DEFAULT_BUFLEN);
        // получаем очередную строку данных
        if (recvLine(client->sock,buf,DEFAULT_BUFLEN) == -1) {
            return -1;
        } else {
            if (strcmp(buf, ".") == 0) { // если точка значит письмо кончилось
                break;
            }
            addToList(response, "."); // и шлем подтверждение что мы приняли
очередную строку
            if (sendResponse(client->sock, response) == -1) {
                return -1;
            }
            addToList(&tmpLetter, buf); // сохраняем строку во временное хранилище
        }
    }
    res = saveLetter(client->name, request->arguments[1], &tmpLetter); //
сохраняем пиьсмо уже на диск
    res == -1 ? addToList(response, "-ERR: can't save letter") :
addToList(response, "+OK: letter saved");
    addToList(response, ".");
```

```
    return sendResponse(client->sock, response);
}
```

serverCommandHandlers.h

```
#ifndef SERVERCOMMANDHANDLERS_H
#define     SERVERCOMMANDHANDLERS_H

#include "myTypes.h"
#include "constants.h"
servCmdHndlrPtr selectHandler(char* command);

int loginCmdHandler(user *client, command* request, dataList* response);
int regCmdHandler(user *client, command* request, dataList* response);
int countCmdHandler(user *client, command* request, dataList* response);
int infoCmdHandler(user *client, command* request, dataList* response);
int bodyCmdHandler(user *client, command* request, dataList* response);
int delCmdHandler(user *client, command* request, dataList* response);
int firstLinesCmdHandler(user *client, command* request, dataList* response);
int defaultCmdHandler(user *client, command* request, dataList* response);
int exitCmdHandler(user *client, command* request, dataList* response);
int logoutCmdHandler(user *client, command* request, dataList* response);
int sendCmdHandler(user *client, command* request, dataList* response);
#endif      /* SERVERCOMMANDHANDLERS_H */
```

serverConsoleCommandHandlers.c

```
#include <unistd.h>
#include "serverConsoleCommandHandlers.h"
#include "myTypes.h"
#include "commands.h"
extern user userlist[];
extern pthread_mutex_t userlist_lock;
extern pthread_mutex_t users_file_lock;

int consoleShow(command* consoleCommand) {
    printf("\tConnected users are:\n");
    int i = 0;
    user u;
    for (i = 0; i < MAX_USERS; ++i) {
        pthread_mutex_lock(&userlist_lock);
        u = userlist[i];
        pthread_mutex_unlock(&userlist_lock);
        if (u.sock == -1) {
            continue;
        }
        printf("User # %d\n", u.index + 1);
        printf("IP:PORT = %s:%d\n", u.ipAddr, u.port);
        u.isLogged  ==  1  ?  printf("Name:  %s\n",  u.name)  :  printf("Not
authorized\n");
    }
    return 0;
}

int consoleKill(command* consoleCommand) {
    if (consoleCommand->argumentsSize != 2) {
        printf("Need 1 argument\n");
        return 0;
    }
    int sockIndex = atoi(consoleCommand->arguments[1]);
```

```
    if (sockIndex <= 0 || sockIndex >= MAX_USERS) {
        printf("Wrong index\n");
        return 0;
    }
    closeClientByIndex(sockIndex - 1);
    return 0;
}

int consoleExit(command* consoleCommand) {
    return 1;
}

int consoleMenu(command* consoleCommand) {
    printServerMenu();
    return 0;
}

int consoleDefault(command* consoleCommand) {
    printf("Unknown command\n");
    return 0;
}

int closeClientByIndex(int index) {
    pthread_mutex_lock(&userlist_lock);
    if(userlist[index].sock == -1) {
        pthread_mutex_unlock(&userlist_lock);
        printf("No user with this index\n");
        return -1;
    }
    close(userlist[index].sock);
    userlist[index].sock = -1; // показываем, что место свободно
    bzero(userlist[index].ipAddr, sizeof (userlist[index].ipAddr));
    userlist[index].port = -1;
    userlist[index].index = -1;
    pthread_mutex_unlock(&userlist_lock);
}

void printServerMenu() {
    printf("Server's menu: \n \n");
    printf("|=====Commands:==============|==========================|\n");
    printf("| show user list    -> %8s     \n",SHOW_SERV_CMD);
    printf("| kill user         -> %8s <N> \n",KICK_SERV_CMD);
    printf("| finish work       -> %8s     \n",EXIT_SERV_CMD);
    printf("| show menu         -> %8s     \n",MENU_SERV_CMD);

printf("|===========================|=======================================|
\n");
}

consoleCmdHndlrPtr selectConsoleCommand(char* command) {
   if (strcmp(command, SHOW_SERV_CMD) == 0)
      return consoleShow;
   if (strcmp(command, KICK_SERV_CMD) == 0)
      return consoleKill;
   if (strcmp(command, EXIT_SERV_CMD) == 0)
      return consoleExit;
    if (strcmp(command, MENU_SERV_CMD) == 0)
        return consoleMenu;
    return consoleDefault;
}
```

serverConsoleCommandHandlers.h

```
#ifndef SERVERCONSOLECOMMANDHANDLERS_H
#define     SERVERCONSOLECOMMANDHANDLERS_H

#include <errno.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <pthread.h>
#include <time.h>

#include <netdb.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/uio.h>

#include "myTypes.h"
#include "constants.h"

consoleCmdHndlrPtr selectConsoleCommand(char* command);

int consoleShow(command * consoleCommand);
int consoleKill(command * consoleCommand);
int consoleExit(command * consoleCommand);
int consoleMenu(command * consoleCommand);
int consoleDefault(command * consoleCommand);

int closeClientByIndex(int index);
void printServerMenu();

#endif      /* SERVERCONSOLECOMMANDHANDLERS_H */
```

serverUtils.c

```
#include "serverUtils.h"
#include "dirent.h"
#include "serverConsoleCommandHandlers.h"

extern user userlist[];
extern pthread_mutex_t userlist_lock;
extern pthread_mutex_t users_file_lock;

void userLoggin(user *client, char* userName) {
    // сохраняем имя пользователя, и тот факт что он авторизировался
    client->isLogged = 1;
    strcpy(client->name, userName);
    pthread_mutex_lock(&userlist_lock);
    userlist[client->index].isLogged = 1;
    strcpy(userlist[client->index].name, userName);
    pthread_mutex_unlock(&userlist_lock);
}

void userLoggout(user *client) {
    client->isLogged = 0;
    bzero(client->name, DEFAULT_BUFLEN);
    pthread_mutex_lock(&userlist_lock);
    userlist[client->index].isLogged = 0;
    bzero(userlist[client->index].name, DEFAULT_BUFLEN);
```

```
    pthread_mutex_unlock(&userlist_lock);
}

int addUsr(char* userName, char*pass) {
    // добавляет пользователя, если такого еще не существует, создает ему папку
    if (isUsrRegistred(userName) == 0) // если такой пользователь уже есть
        return -1;
    pthread_mutex_lock(&users_file_lock);
    FILE *f = fopen(USERS_FILE, "a");
    if (f == NULL) {
        pthread_mutex_unlock(&users_file_lock);
        return -1;
    }
    // дописываем в файл пользователя
    char buf[DEFAULT_BUFLEN];
    bzero(buf, DEFAULT_BUFLEN);
    sprintf(buf, "%s:%s\n", userName, pass);
    fputs(buf, f);
    fclose(f);
    pthread_mutex_unlock(&users_file_lock);
    // создаем папку для почты
    bzero(buf, DEFAULT_BUFLEN);
    sprintf(buf, "mail/%s", userName);
    int res = mkdir(buf, S_IRWXU | S_IRWXG | S_IRWXO);
    return res == 0 ? 0 : -1;
}

int isUsrRegistred(char* userName) {
    pthread_mutex_lock(&users_file_lock);
    FILE *f = fopen(USERS_FILE, "r");
    if (f == NULL) {
        pthread_mutex_unlock(&users_file_lock);
        return -1;
    }
    // читаем файл построчно и сравниваем с текущим именем пользователя
    char buf[DEFAULT_BUFLEN];
    bzero(buf, DEFAULT_BUFLEN);
    char *last;
    while (!feof(f)) {
        bzero(buf, DEFAULT_BUFLEN);
        fgets(buf, DEFAULT_BUFLEN, f);
        // выделяем имя пользователя.
        char* curUserName = strtok_r(buf, ":", &last);
        if (curUserName == NULL) {
            fclose(f);
            pthread_mutex_unlock(&users_file_lock);
            return -1;
        }
        if (strcmp(curUserName, userName) == 0) { // и проверяем совпадает ли оно
с переданным
            fclose(f);
            pthread_mutex_unlock(&users_file_lock);
            return 0;
        }
    }
    fclose(f);
    pthread_mutex_unlock(&users_file_lock);
    return -1;
}

int checkPass(char* userName, char*password) {
    pthread_mutex_lock(&users_file_lock);
    FILE *f = fopen(USERS_FILE, "r");
```

```
    if (f == NULL) {
        pthread_mutex_unlock(&users_file_lock);
        return -1;
    }
    char curUser[DEFAULT_BUFLEN];
    bzero(curUser, DEFAULT_BUFLEN);
    sprintf(curUser, "%s:%s", userName, password);
    char buf[DEFAULT_BUFLEN];
    // читаем файл построчно и сравниваем с текущим именем пользователя и паролем
    while (!feof(f)) {
        bzero(buf, DEFAULT_BUFLEN);
        fgets(buf, DEFAULT_BUFLEN, f);
        if (strlen(buf) == 0) {
            fclose(f);
            pthread_mutex_unlock(&users_file_lock);
            return -1;
        }
        buf[strlen(buf) - 1] = '\0';
        if (strcmp(curUser, buf) == 0) { // и проверяем совпадает ли оно с
переданным
            fclose(f);
            pthread_mutex_unlock(&users_file_lock);
            return 0;
        }
    }
    fclose(f);
    pthread_mutex_unlock(&users_file_lock);
    return -1;
}

void closeAllSockets() {
    int i;
    for (i = 0; i < MAX_USERS; ++i) {
        if (userlist[i].sock == -1)
            continue;
        closeClientByIndex(i);
    }
}

int getClientByIpAndPort(char* ip, int port) {
    int i = 0;
    for (i = 0; i < MAX_USERS; ++i) {
        if (strcmp(userlist[i].ipAddr, ip) == 0 && userlist[i].port == port) {
            return i;
        }
    }
    return -1;
}

int isUserPresentNow(char* userName) {
    int i;
    for(i = 0; i < MAX_USERS; ++i) {
        if(userlist[i].sock == -1 || strlen(userlist[i].name) == 0)
            continue;
        if(strcmp(userlist[i].name,userName) == 0)
            return 1;
    }
    return 0;
}
```

serverUtils.h

```
#ifndef SERVERUTILS_H
#define     SERVERUTILS_H

#include <errno.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <pthread.h>
#include <time.h>

#include <netdb.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/uio.h>

#include "myTypes.h"


void userLoggin(user* client, char* userName);
void userLoggout(user* client);
int addUsr(char* userName, char*pass);
int isUsrRegistred(char* userName);
int checkPass(char* userName, char*password);

void closeAllSockets();
int getClientByIpAndPort(char* ip, int port);
int isUserPresentNow(char* userName);

#endif      /* SERVERUTILS_H */
```

Приложение 2. Почта поверх UDP

Client.c

```
#include "constants.h"
#include "myTypes.h"
#include "clientCommandHandlers.h"
#include "stdio.h"
#include "connectionFunctions.h"
#include "commands.h"
void printMainMenu();
int clientMain(char* addr, char* port) {
// создаем сокет и подключаем его к серверу
int sock = createClient(addr, port);
if(sock == -1) {
printf("Can't connect to server\n");
exit(1);
}
// переменные, для хранения ввода пользователя
int res = 0;
command cmd;
initCommand(&cmd);
dataList request, response;
initDataList(&request);
initDataList(&response);
printMainMenu();
for (;;) {
clearCommand(&cmd);
clearList(&request);
clearList(&response);
printf("Enter command\n");
fgets(cmd.commandLine, DEFAULT_BUFLEN, stdin);
cmd.commandLine[strlen(cmd.commandLine) - 1] = '\0';
if (strlen(cmd.commandLine) == 0)
continue;
parseCommand(&cmd); // разбиваем команду на слова
res = prepareRequest(&cmd, &request); // проверяем корректность ввода, и если
можем, то сами выполняем запрос
if (res == -1 || res == 1) { // -1 - не корректный ввод, 1 - запрос сами
выполнили, не надо просить сервер
continue;
}
res = sendAndRecv(sock, &request, &response);
if (res == -1) {
printf("Sorry. Connection is lost\n");
break;
}
printDataList(&response); // пишем ответ сервера
// проверяем не сообщение ли о завершении
if (strcmp(response.data[0], BYE_STRING) == 0) {
break;
}
}
return 0;
}
void printMainMenu() {
printf("\tMAIL CLIENT \n \n");
printf("Main menu: \n \n");
printf("|=====Commands:==============|==========================|\n");
printf("| register new user -> %8s <NAME> <PASSWORD> \n",REG_CMD);
printf("| login -> %8s <NAME> <PASSWORD> \n",LOGIN_CMD);
```

55

```
printf("| send letter -> %8s <RECIPIENT> \n",SEND_CMD);
printf("| number of letters -> %8s \n",COUNT_CMD);
printf("| info about letter -> %8s [<N>] \n",INFO_CMD);
printf("| body of letter -> %8s <N> \n",BODY_CMD);
printf("| delete letter -> %8s <N> \n",DEL_CMD);
printf("| first <M> strings -> %8s <N> <M> \n",FIRST_LINES_CMD);
printf("| finish work -> %8s \n",EXIT_CMD);
printf("| show menu -> %8s \n",MENU_CMD);
printf("| logout -> %8s \n",LOGOUT_CMD);
printf("|============================|=======================================|
\n");
}
```

clientCommandHandlers.c

```
#include "clientCommandHandlers.h"
#include "constants.h"
#include "stdio.h"
#include "commands.h"
extern void printMainMenu();
int checkCommand(command* c) {
if (strcmp(c->arguments[0], LOGIN_CMD) == 0) {
return c->argumentsSize == 3 ? 1 : 0;
}
if (strcmp(c->arguments[0], REG_CMD) == 0) {
return c->argumentsSize == 3 ? 1 : 0;
}
if (strcmp(c->arguments[0], INFO_CMD) == 0) {
if (c->argumentsSize == 1)
return 1;
if (c->argumentsSize != 2 || atoi(c->arguments[1]) <= 0) // если нет аргумента,
или аргумент не число
return 0;
return 1;
}
if (strcmp(c->arguments[0], COUNT_CMD) == 0) {
if (c->argumentsSize != 1)
return 0;
return 1;
}
if (strcmp(c->arguments[0], BODY_CMD) == 0) {
if (c->argumentsSize != 2)
return 0;
if (atoi(c->arguments[1]) <= 0)
return 0;
return 1;
}
if (strcmp(c->arguments[0], DEL_CMD) == 0) {
if (c->argumentsSize != 2)
return 0;
if (atoi(c->arguments[1]) <= 0)
return 0;
return 1;
}
if (strcmp(c->arguments[0], FIRST_LINES_CMD) == 0) {
if (c->argumentsSize != 3)
return 0;
if (atoi(c->arguments[1]) <= 0)
return 0;
if (atoi(c->arguments[2]) <= 0)
return 0;
return 1;
```

```
}
return c->argumentsSize == 1 ? 1 : 0;
if (strcmp(c->arguments[0], MENU_CMD) == 0) { 56
}
if (strcmp(c->arguments[0], EXIT_CMD) == 0) {
return c->argumentsSize == 1 ? 1 : 0;
}
if (strcmp(c->arguments[0], LOGOUT_CMD) == 0) {
return c->argumentsSize == 1 ? 1 : 0;
}
if (strcmp(c->arguments[0], SEND_CMD) == 0) {
return c->argumentsSize == 2 ? 1 : 0;
}
return 0;
}
/* Подготавливает запрос к отправке или в простейшем случае сам выполняет его
* возвращает -1 в случае ошибки, 1 - если сам выполнил и 0 если запрос
подготовлен
* и его надо отослать
*/
int prepareRequest(command* c, dataList* request) {
if (!checkCommand(c)) {
printf("\tIncorrect command or arguments\n");
return -1;
}
if (strcmp(c->arguments[0], MENU_CMD) == 0) { // если просят показать меню,
просто показываем и ничего серверу не шлем
printMainMenu();
return 1;
}
if (strcmp(c->arguments[0], EXIT_CMD) == 0) { //
printf("Sending last message to server\n");
}
addToList(request, c->commandLine);
if (strcmp(c->arguments[0], SEND_CMD) == 0) { // если хотят отправить письмо, то
сначала считываем его тело из консоли
readLetter(request);
}
return 0;
}
int readLetter(dataList* dl) {
printf("Input letter. Empty string to finish\n");
char line[DEFAULT_BUFLEN];
for (;;) {
bzero(line, DEFAULT_BUFLEN);
fgets(line, DEFAULT_BUFLEN, stdin);
line[strlen(line) - 1] = '\0';
if (strlen(line) == 0) {
addToList(dl, ".");
break;
}
addToList(dl, line);
}
return 1;
}
```

clientCommandHandlers.h

```
#ifndef CLIENTCOMMANDHANDLERS_H
#define CLIENTCOMMANDHANDLERS_H
#include "myTypes.h"
int checkCommand(command* c);
```

```
int prepareRequest(command* c, dataList* request);
int readLetter(dataList* dl);
#endif /* CLIENTCOMMANDHANDLERS_H */
```

commands.h

```
#ifndef COMMANDS_H
#define COMMANDS_H
// команды, передающиеся клиентом серверу (если больше 8 символов -> поправить
функцию printMainMenu
#define LOGIN_CMD "login"
#define REG_CMD "reg"
#define INFO_CMD "list"
#define COUNT_CMD "stat"
#define BODY_CMD "retr"
#define DEL_CMD "delete"
#define FIRST_LINES_CMD "top"
#define MENU_CMD "menu"
#define EXIT_CMD "exit"
#define LOGOUT_CMD "logout"
#define SEND_CMD "send"
// команды консоли сервера (если больше 8 символов -> поправить функцию
printServerMenu
#define SHOW_SERV_CMD "show"
#define KICK_SERV_CMD "kill"
#define EXIT_SERV_CMD "exit"
#define MENU_SERV_CMD "menu"
// команды, которые шлет сервер клиенту
#define BYE_STRING "+OK: bye-bye"
#endif /* COMMANDS_H */
```

connectionFunction.c

```
#include "connectionFunctions.h"
#include "constants.h"
int createClient(char* hostName, char* portName) {
// printf("Inside createClient\n");
int serv;
serv = socket(AF_INET, SOCK_DGRAM, 0);
if (serv < 0) {
perror("Client: error while using socket");
exit(1);
}
struct sockaddr_in serverAddr;
setAddress(hostName, portName, &serverAddr, "udp");
int serverLen = sizeof (serverAddr);
return requestNewSocket(serv, (struct sockaddr *) &serverAddr, &serverLen);
}
int createServer(char* hostName, char* portName) {
// printf("Inside createServer\n");
struct sockaddr_in servInfo;
int servSocket;
setAddress(hostName, portName, &servInfo, "udp");
servSocket = socket(AF_INET, SOCK_DGRAM, 0);
// printf("after setaddr\n");
if (servSocket < 0) {
perror("Server: Error while using socket");
exit(1);
}
if (bind(servSocket, (struct sockaddr*) &servInfo, sizeof (servInfo))) {
```

```
perror("Server: Error while using bind");
exit(1);
}
// printf("returning createServer\n");
return servSocket;
}
void setAddress(char* hostName, char* portName, struct sockaddr_in* info, char*
protocol) {
// printf("Inside setAdres\n");
bzero(info, sizeof (info));
info->sin_family = AF_INET;
if (hostName != NULL) {
if (!inet_aton(hostName, &info->sin_addr)) {
perror("Unknown host ");
exit(1);
}
} else {
info->sin_addr.s_addr = htonl(INADDR_ANY);
}
char* endptr;
// printf("Trying to translate portName: %s\n",portName);
short port = strtol(portName, &endptr, 0);
if (*endptr == '\0')
info->sin_port = htons(port);
else {
perror("Unknown port ");
exit(1);
}
}
/* шлем клиенту все, что хранится в dataList,после каждой посланной строчки
* ожидаем очередного запроса, кроме строчки вида ".", сигнализирующей о конце
передачи
* Если послать не удалось - возвращает -1
*/
int sendResponse(int sock, dataList* dl) {
int res = 0;
command tmpCmd; // используется для приема запроса на очередные данные
(фактически подтверждения принятия предыдущих)
initCommand(&tmpCmd);
int i = 0;
for (i = 0; i < dl->length; ++i) {
res = sendLine(sock, dl->data[i], dl->expectedMsg - 1);// шлем очередную строку
как ответ на предыдущий запрос
if (res == -1) {
return -1;
}
strcpy(dl->lastMsg, dl->data[i]);// устанавливаем новый последний отправленный
ответ
if (strcmp(dl->data[i], ".") != 0) { // если не последняя строчка, то ждем
запроса на новую
if (recvRequest(sock, &tmpCmd, dl) == -1) { // если запрос так получить и не
удалось
return -1;
} else {
continue;
}
} else { // если последняя то выходим
return res;
}
}
return 0;
}
/* получает новый запрос от клиента. Возвращает управление, если пришел запрос
```

```
* с ожидаемым номером или закончился таймаут заданный константой REQUEST_TIMEOUT
* если пришел запрос на 1 меньший, чем ожидалось - шлет предыдущий ответ,
* все остальные запросы игнорирует
*/
int recvRequest(int sock, command* cmd, dataList* prevResponse) {
// переменные, для временного хранения полученного сообщения
char cNumMsg[NUM_MSG_LENGTH]; //номер полученного сообщения
char cMsg[DEFAULT_BUFLEN]; // тело полученного сообщения
bzero(cNumMsg, NUM_MSG_LENGTH);
bzero(cMsg, DEFAULT_BUFLEN);
struct iovec recvBuf[2];
recvBuf[0].iov_base = cNumMsg;
recvBuf[0].iov_len = NUM_MSG_LENGTH;
recvBuf[1].iov_base = cMsg;
recvBuf[1].iov_len = DEFAULT_BUFLEN;
int res = 0;
for (;;) {
int n = waitSocketReady(sock, REQUEST_TIMEOUT);// ждем пока прийдет запрос или
выйдет таймаут
if (n == -1 || n == 0) {
return -1;
}
res = readv(sock, recvBuf, 2); // читаем полученное сообщение
if (res <= 0) { //TODO возможно возвращать лучше не -1 а 0 или -1, чтобы
различать ошибку и конец передачи?
return -1;
}
// получили запрос, сравниваем его номер с ожидаемым
if (atoi(cNumMsg) == prevResponse->expectedMsg) { // если получили тот, который
и ожидали
++(prevResponse->expectedMsg); // ожидаем уже следующий
strcpy(cmd->commandLine, cMsg); // сохраняем полученную команду
return res; // и выходим
}
// сравниваем с номером, на 1 меньше чем ожидаемый
if (atoi(cNumMsg) == prevResponse->expectedMsg - 1) { // если он равен, то шлем
предыдущий ответ и опять ждем запроса
res = sendLine(sock,prevResponse->lastMsg,prevResponse->expectedMsg-1);
if (res == -1) {
return -1;
}
}
// все остальное просто отбрасываем
}
}
/* шлет одну строку клиенту, в качестве ответа на запрос numMsg
*/
int sendLine(int sock, char* line, int numMsg) {
// printf("Inside send line/ Sending '%s' with number '%d'\n",line,numMsg);
char cNumMsg[NUM_MSG_LENGTH];
char cMsg[DEFAULT_BUFLEN];
bzero(cNumMsg, NUM_MSG_LENGTH);
bzero(cMsg, DEFAULT_BUFLEN);
sprintf(cNumMsg, "%d", numMsg);
sprintf(cMsg, "%s", line);
struct iovec sendBuf[2];
sendBuf[0].iov_base = cNumMsg;
```

```
sendBuf[0].iov_len = NUM_MSG_LENGTH;
sendBuf[1].iov_base = cMsg;
sendBuf[1].iov_len = strlen(cMsg) + 1; // +1 т.к. еще символ конца строки
return writev(sock, sendBuf, 2);
}
/* используется клиентом. Шлет одну строку request с номером numMsg и ожидает
ответ
* на эту строку. Если ответ пришел сохраняет его в параметре response и
возвращает
* количество пришедших байт. Если ответа не дождались - возвращает -1
*/
int sendAndRecvOneStr(int sock, char* request, int* numMsg, char* response) {
// куда будем принимать
char receivedMsg[DEFAULT_BUFLEN];// тело принятого сообщения
char receivedNumMsg[NUM_MSG_LENGTH];// номер принятого сообщения в символьном
виде
bzero(receivedMsg, DEFAULT_BUFLEN);
bzero(receivedNumMsg, NUM_MSG_LENGTH);
struct iovec recvbuf[2];
recvbuf[0].iov_base = receivedNumMsg;
recvbuf[0].iov_len = NUM_MSG_LENGTH;
recvbuf[1].iov_base = receivedMsg;
recvbuf[1].iov_len = DEFAULT_BUFLEN;
// что будем слать
char sendMsg[DEFAULT_BUFLEN]; // тело сообщения для отправки
char cNumMsg[NUM_MSG_LENGTH]; // номер сообщения для отправки в символьном виде
bzero(sendMsg, DEFAULT_BUFLEN);
bzero(cNumMsg, NUM_MSG_LENGTH);
sprintf(cNumMsg, "%d", *numMsg);
sprintf(sendMsg, "%s", request);
struct iovec sendbuf[2];
// сначала шлем номер дейтаграммы
sendbuf[0].iov_base = cNumMsg;
sendbuf[0].iov_len = NUM_MSG_LENGTH;
// потом сами данные
sendbuf[1].iov_base = sendMsg;
sendbuf[1].iov_len = strlen(sendMsg);
// номер попытки и таймаут ожидания
int try = 0;
int timeout = 1;
int n = 0;
int needSend = 1;
do {
if (needSend == 1) {
bzero(receivedMsg, DEFAULT_BUFLEN);
n = writev(sock, sendbuf, 2);// если надо шлем наш запрос
}
int isReady = waitSocketReady(sock, timeout); // ждем пока прийдет ответ или
пройдет таймаут
if (isReady == -1) { //критическая ошибка
return -1;
}
if (isReady == 0) { // кончился таймаут а ответа не дождались
needSend = 1; // значит надо послать еще 1 запрос
++try;
++timeout;
continue;
}
// если сюда дошли, значит получили хоть какой-то ответ
n = readv(sock, recvbuf, 2);
if (n <= 0) {
return -1;
}
```

```
// смотрим приняли то, что надо или нет
if (atoi(receivedNumMsg) != *numMsg) { // если не соответствует ожидаемой
needSend = 0; // слать еще раз не будем, просто отбрасываем и ждем нового ответа
try++;
continue;
}
// если приняли то, что хотели
++(*numMsg); // увеличиваем номер сообщения
strcpy(response, receivedMsg); // сохраняем полученное сообщение
return n;
} while (try < REQUEST_TRIES);
return -1;
}
/*используется клиентом. Шлет в общем случае несколько строк запроса из
параметра
* request и принимает в общем случае несколько строк ответа, которые сохраняет в
* параметр response
*/
int sendAndRecv(int sock, dataList* request, dataList* response) {
int i = 0;
int res = 0;
char buf[DEFAULT_BUFLEN];
for (i = 0; i < request->length; ++i) {
bzero(buf, DEFAULT_BUFLEN);
res = sendAndRecvOneStr(sock, request->data[i], &request->expectedMsg, buf); //
шлем очередную строку запроса
if (res == -1) {
return -1;
}
if (strcmp(buf, ".") == 0) { // если сервер говорит об окончании передачи, то
шлем ему следующую строку
continue;
}
addToList(response, buf); // сохраняем первую строку ответа и дочитываем
остальной
for (;;) { // если пришла не точка, значит сервер еще будет досылать ответ. Ждем
его
bzero(buf, DEFAULT_BUFLEN);
res = sendAndRecvOneStr(sock, buf, &request->expectedMsg, buf); // шлем фигню,
так как сервер ее все равно не читает, главное сам факт прихода запроса
if (res == -1) {
return -1;
}
// если сервер говорит об окончании передачи, выходим из внутреннего цикла и
шлем оставшиеся строки запроса
if (strcmp(buf, ".") == 0) {
break;
}
addToList(response, buf);
}
}
}
int waitSocketReady(int sock, int sec) {
//printf("Inside waitSocketReady sock = %d sec = %d\n",sock,sec);
struct timeval tv;
tv.tv_sec = sec;
tv.tv_usec = 0;
fd_set rset;
FD_ZERO(&rset);
FD_SET(sock, &rset);
return (select(sock + 1, &rset, NULL, NULL, &tv));
}
```

```
void printAddrInfo(struct sockaddr_in* addr) {
printf("Addr info: %s:%d\n", inet_ntoa(addr->sin_addr), ntohs(addr->sin_port));
}
int requestNewSocket(int sock, struct sockaddr_in* serverAddr, int*
serverAddrLen) {
int try = 0;
int timeout = 1;
do {
sendto(sock, "Y", 2, 0, (struct sockaddr*) serverAddr, *serverAddrLen);
int isReady = waitSocketReady(sock, timeout);
if (isReady == -1) {
return -1;
}
if (isReady == 0) {
++try;
++timeout;
continue;
}
char buf[2];
int n = recvfrom(sock, buf, 2, 0, (struct sockaddr*) serverAddr, serverAddrLen);
// если какая-то ошибка при подключении (клиентов слишком много)
if (buf[0] == '-') {
printf("%s\n", buf);
exit(1);
}
// коннектимся уже к нашему личному сокету
connect(sock, (struct sockaddr*) serverAddr, *serverAddrLen);
return sock;
} while (try <= 4);
return -1;
}
```

connectionFunction.h

```
#ifndef CONNECTIONFUNCTIONS_H
#define CONNECTIONFUNCTIONS_H
#include <errno.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <pthread.h>
#include <time.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/uio.h>
#include "myTypes.h"
int createClient(char* hostName, char* portName);
int createServer(char* hostName, char* portName);
int requestNewSocket(int serv, struct sockaddr_in* serverAddr, int*
serverAddrLen);
void setAddress(char* hostName, char* portName, struct sockaddr_in* info, char*
protocol);
int sendResponse(int sock, dataList* dl);
int sendPrevResp(int sock, dataList* prevResponse);
int sendLine(int sock, char* line, int numMsg);
int recvRequest(int sock, command* cmd, dataList* prevResponse);
int sendAndRecv(int sock, dataList* request, dataList* response);
int waitSocketReady(int sock, int sec);
```

```
int sendAndRecvOneStr(int sock, char* request, int* numMsg, char* response);
void printAddrInfo(struct sockaddr_in* addr);
#endif /* CONNECTIONFUNCTIONS_H */
```

constants.h

```
#ifndef CONSTANTS_H
#define CONSTANTS_H
#include "stddef.h"
#include "string.h"
#define LINE_DELIM '\n'
#define DEFAULT_PORT "7500"
#define DEFAULT_SERV_ADDR "127.0.0.1"
#define DEFAULT_BUFLEN 512
#define MAX_ARGS 4 // максимальное количество аргументов в команде
#define MAX_ARG_SIZE 256 // максимальная длинна одного аргумента команды
#define NUM_MSG_LENGTH 7 // максимальное количество разрядов в номере сообщения
#define REQUEST_TIMEOUT 1000 // максимальное время между запросами от клиента.
(после этого кикаем)
#define REQUEST_TRIES 4 // сколько раз клиент пытается получить ответ на свой
запрос. (после каждой попытки таймаут увеличивается на 1 секуду)
#define BACKLOG 5
#define MAX_USERS 2
#define USERS_FILE "mail/users.txt"
#define MAX_LETTERS_STRINGS 40
#define HEADERS_DELIM "----"
#define HEADERS_DELIM_LENGTH 4
#endif /* CONSTANTS_H */
```

fileSystemFunctions.c

```
#include "fileSystemFunctions.h"
#include "clientCommandHandlers.h"
#include "dirent.h"
#include "stdio.h"
#include "time.h"
extern pthread_mutex_t file_system_lock;
// возвращает количество писем пользователя. Если не может подсчитать, к примеру
нет папки, возвращает -1
int getLettersCount(char* userName) {
// открываем папку пользователя
char pathToFolder[DEFAULT_BUFLEN];
bzero(pathToFolder, DEFAULT_BUFLEN);
sprintf(pathToFolder, "mail/%s", userName);
DIR* d = opendir(pathToFolder);
if (d == NULL) {
return -1;
}
int lettersCount = 0;
// получаем все записи в папке, и если это текстовый файл, увеличиваем счетчик
http://www.ibm.com/developerworks/ru/library/au-unix-readdir/
struct dirent entry;
struct dirent* result;
readdir_r(d, &entry, &result);
while (result) {
//получаем инфу о файле
if (entry.d_type == DT_REG)
//http://www.delorie.com/gnu/docs/glibc/libc_270.html
lettersCount++;
// получаем инфу о следующем файле
readdir_r(d, &entry, &result);
}
```

```
closedir(d);
return lettersCount;
}
// записывает заголовки письма в стурктуру типа dataList
// ничего не проверяет, т.к. вызывающая функция должна была уже это сделать
void getInfoAboutLetter(user* client, dataList* dl, int letterNumber) {
dataList tempArray;
clearList(&tempArray);
// получаем все письмо во временный массив
getWholeLetter(client, letterNumber, &tempArray);
// проверяем удалось ли прочитать файл
if (tempArray.length == 0) {
addToList(dl, " -ERR: can't read file or file is empty");
return;
}
int i;
char tb[DEFAULT_BUFLEN];
bzero(tb, DEFAULT_BUFLEN);
sprintf(tb, "\tLetter # %d", letterNumber);
addToList(dl, tb);
for (i = 0; i < tempArray.length; ++i) {
// если дошли до конца заголовков то прекращаем(подразумевается что заголовки
есть ВСЕГДА!)
if (strncmp(tempArray.data[i], HEADERS_DELIM, HEADERS_DELIM_LENGTH) == 0) {
break;
}
// иначе копируем в основной результирующий массив
addToList(dl, tempArray.data[i]);
}
}
// ничего не проверяет, т.к. вызывающая функция должна была уже это сделать
void getLetterBody(user* client, int letterNumber, dataList* dl) {
dataList tempArray;
clearList(&tempArray);
// получаем все письмо во временный массив
getWholeLetter(client, letterNumber, &tempArray);
// проверяем удалось ли прочитать файл
if (tempArray.length == 0) {
addToList(dl, " -ERR: can't read file");
return;
}
char tb[DEFAULT_BUFLEN];
bzero(tb, DEFAULT_BUFLEN);
sprintf(tb, "\tLetter # %d", letterNumber);
addToList(dl, tb);
int i;
int isHeader = 1; // флаг, показывающий что сейчас читаются заголовки
for (i = 0; i < tempArray.length; ++i) {
// если наткнулись на конец заголовков
```

```
if (strncmp(tempArray.data[i], HEADERS_DELIM, HEADERS_DELIM_LENGTH) == 0 &&
isHeader == 1) {
isHeader = 0;
continue;
}
if (isHeader == 1)
continue; // пропускаем заголовок
// сюда приходим, если читаем само тело письма
addToList(dl, tempArray.data[i]);
}
}
//возвращает 0, если удалось удалить или -1 если нет
int deleteLetter(char* userName, int letterNumber) {
char fileName[DEFAULT_BUFLEN];
bzero(fileName, DEFAULT_BUFLEN);
sprintf(fileName, "mail/%s/%d", userName, letterNumber);
if (remove(fileName)) {
return -1;
}
pthread_mutex_lock(&file_system_lock);
shiftFiles(userName, letterNumber);
pthread_mutex_unlock(&file_system_lock);
return 0;
}
// возвращает файл в виде массива строк, проверок не выполняет!
// если возникла ошибка, ничего не добавляет в массив, а просто прекращает
выполнение
void getWholeLetter(user* client, int letterNumber, dataList* result) {
// пытаемся открыть файл
char buf[DEFAULT_BUFLEN];
bzero(buf, DEFAULT_BUFLEN);
sprintf(buf, "mail/%s/%d", client->name, letterNumber);
FILE* f = fopen(buf, "r");
if (f == NULL) {
return;
}
// если файл все же открыли читаем построчно и сохраняем в массив
while (!feof(f)) {
bzero(buf, DEFAULT_BUFLEN);
fgets(buf, DEFAULT_BUFLEN, f);
buf[strlen(buf) - 1] = '\0'; // а если ничего не прочитали?
addToList(result, buf);
}
fclose(f);
}
// записывает в массив result первые maxStrings ТЕЛА письма
void getLetterHead(user* client, int letterNumber, int maxStrings, dataList*
result) {
dataList tempArray;
clearList(&tempArray);
// получаем все письмо во временный массив
getWholeLetter(client, letterNumber, &tempArray);
// проверяем удалось ли прочитать файл
if (tempArray.length == 0) {
addToList(result, " -ERR: can't read file");
return;
}
int i;
int curStrings = 0;
int isHeader = 1; // флаг, показывающий что сейчас читаются заголовки
for (i = 0; i < tempArray.length; ++i) {
// если наткнулись на конец заголовков
```

```
if (strncmp(tempArray.data[i], HEADERS_DELIM, HEADERS_DELIM_LENGTH) == 0 &&
isHeader == 1) {
isHeader = 0;
continue;
}
if (isHeader == 1)
continue; // пропускаем заголовок
// сюда приходим, если читаем само тело письма
if (curStrings >= maxStrings) {
break;
} else {
addToList(result, tempArray.data[i]);
curStrings++;
}
}
}
int saveLetter(char* from, char* to, dataList* tmpLetter) {
int i = getFirstFreeNumber(to);
char fileName[DEFAULT_BUFLEN];
bzero(fileName, DEFAULT_BUFLEN);
sprintf(fileName, "mail/%s/%d", to, i);
FILE* f = fopen(fileName, "w");
if (f == NULL)
return -1;
writeHeaders(f, from, to);
for (i = 0; i < tmpLetter->length; ++i) {
fputs(tmpLetter->data[i], f);
fputs("\n", f);
}
fclose(f);
return 0;
}
void writeHeaders(FILE* f, char* from, char* to) {
char buf[DEFAULT_BUFLEN];
bzero(buf, DEFAULT_BUFLEN);
sprintf(buf, "from=%s\n", from);
fputs(buf, f);
bzero(buf, DEFAULT_BUFLEN);
sprintf(buf, "to=%s\n", to);
fputs(buf, f);
bzero(buf, DEFAULT_BUFLEN);
getCurTime(buf);
fputs(buf, f);
bzero(buf, DEFAULT_BUFLEN);
sprintf(buf, "%s\n", HEADERS_DELIM);
fputs(buf, f);
}
void getCurTime(char* buf) {
time_t rawtime;
struct tm * timeinfo;
time(&rawtime);
timeinfo = localtime(&rawtime);
strcpy(buf, (asctime) (timeinfo));
}
int getFirstFreeNumber(char* to) {
```

```
int i = 1;
FILE *f;
char buf[DEFAULT_BUFLEN];
for (;;) {
bzero(buf, DEFAULT_BUFLEN);
sprintf(buf, "mail/%s/%d", to, i);
f = fopen(buf, "r");
if (f == NULL)
return i;
fclose(f);
++i;
}
}
void shiftFiles(char* userName, int letterNumber) {
char newFileName[DEFAULT_BUFLEN];
bzero(newFileName, DEFAULT_BUFLEN);
char oldFileName[DEFAULT_BUFLEN];
bzero(oldFileName, DEFAULT_BUFLEN);
sprintf(newFileName, "mail/%s/%d", userName, letterNumber);
sprintf(oldFileName, "mail/%s/%d", userName, letterNumber + 1);
int i = letterNumber + 1;
while (!rename(oldFileName, newFileName)) {
++i;
bzero(newFileName, DEFAULT_BUFLEN);
bzero(oldFileName, DEFAULT_BUFLEN);
strcpy(newFileName, oldFileName);
sprintf(newFileName, "mail/%s/%d", userName, i);
}
}
```

fileSystemFunctions.h

```
#ifndef FILESYSTEMFUNCTIONS_H
#define FILESYSTEMFUNCTIONS_H
#include "myTypes.h"
#include "constants.h"
#include "stdio.h"
int getLettersCount(char* userName);
void getLetterHead(user* client, int letterNumber, int maxStrings,
dataList* result);
void getInfoAboutLetter(user* client, dataList* dl, int
letterNumber);
void getLetterBody(user* client, int letterNumber, dataList* dl);
void getWholeLetter(user* client, int letterNumber, dataList*
result);
int deleteLetter(char* userName, int letterNumber);
int saveLetter(char* from, char* to, dataList* tmpLetter);
// вспомогательные функции
void writeHeaders(FILE* f, char* from, char* to);
void getCurTime(char* buf);
int getFirstFreeNumber(char* to);
void shiftFiles(char* userName, int letterNumber);
#endif /* FILESYSTEMFUNCTIONS_H */
```

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "connectionFunctions.h"
int main(int argc, char** argv) {
if (argc < 2) {
printf("You should use at least key S or C");
exit(0);
}
// если хотят запустить сервер
if (strncmp(argv[1], "S", 1) == 0) {
//если не задан порт
if (argc == 2) {
printf("Using default port: %s\n", DEFAULT_PORT);
serverMain(DEFAULT_PORT);
} else {
serverMain(argv[2]);
}
// если хотят запустить клиент
} else if (strncmp(argv[1], "C", 1) == 0) {
// если не задано ни адреса ни порта
if (argc == 2) {
printf("Using defalut ip and port: %s: %s\n", DEFAULT_SERV_ADDR, DEFAULT_PORT);
clientMain(DEFAULT_SERV_ADDR, DEFAULT_PORT);
} else if (argc == 3) {
// если задан только адрес
printf("Using defalut port: %s:\n", DEFAULT_PORT);
clientMain(argv[2], DEFAULT_PORT);
} else {
clientMain(argv[2], argv[3]);
}
} else {
// если неизвестный ключ запуска
printf("Illegal first argument '%s'. First argument must be 'S' or 'C'\n",
argv[1]);
exit(0);
}
return 0;
}
```

myTypes.c

```
#include "myTypes.h"
void addToList(dataList* dl, char* string) {
if (dl->length == MAX_LETTERS_STRINGS) {
printf("Error: trying to add string to full list\n");
return;
}
strcpy(dl->data[(dl->length)], string);
++(dl->length);
}
void clearList(dataList* dl) {
dl->length = 0;
int i;
for (i = 0; i < MAX_LETTERS_STRINGS; ++i) {
bzero(dl->data[i], DEFAULT_BUFLEN); // можно оптимизировать
}
}
void printDataList(dataList* dl) {
int i;
```

```
for (i = 0; i < dl->length; ++i) {
printf("\t%s\n", dl->data[i]);
}
}
void initCommand(command* c) {
// printf("Inside init command\n");
bzero(c->commandLine, DEFAULT_BUFLEN);
int i = 0;
// printf("\tgoing to malloc %d arguments\n",MAX_ARGS);
c->arguments = (char**) malloc(MAX_ARGS * sizeof (char*));
for (i = 0; i < MAX_ARGS; ++i) {
c->arguments[i] = (char*) malloc(MAX_ARG_SIZE * sizeof (char));
}
c->argumentsSize = 0;
// printf("Exiting init command\n");
}
void initDataList(dataList* dl) {
// printf("Inside init data list\n");
dl->length = 0;
bzero(dl->lastMsg, DEFAULT_BUFLEN);
dl->expectedMsg = 0;
int i;
for (i = 0; i < MAX_LETTERS_STRINGS; ++i) {
bzero(dl->data[1], DEFAULT_BUFLEN);
}
}
void clearCommand(command* c) {
bzero(c->commandLine, DEFAULT_BUFLEN);
int i;
for (i = 0; i < c->argumentsSize; ++i) {
bzero(c->arguments[i], DEFAULT_BUFLEN);
}
}
void parseCommand(command* c) {
// printf("Inside parse command\n");
if (c->commandLine == NULL || strlen(c->commandLine) == 0) {
printf("Error. Tring to parse empty command\n");
return;
}
c->argumentsSize = 0;
// о функции strtok_r - http://unixcoding.blogspot.ru/2010/10/blog-post_679.html
char *tmpCommand = strdup(c->commandLine);
char *last;
char* token = strtok_r(tmpCommand, " ", &last); // считываем саму команду
strcpy(c->arguments[0], token); // и копируем ее как первый аргумент
++c->argumentsSize;
int i = 0;
for (i = 1; i < MAX_ARGS; ++i) {
token = strtok_r(NULL, " ", &last);
if (token == NULL)
break;
strcpy(c->arguments[i], token);
++c->argumentsSize;
}
free(tmpCommand);
}
```

myTypes.h

```
#ifndef MYTYPES_H
#define MYTYPES_H
#include "constants.h"
typedef struct {
char name[DEFAULT_BUFLEN];
int sock;
int isLogged;
int index;
char *ipAddr[DEFAULT_BUFLEN];
int port;
char* serverIP[DEFAULT_BUFLEN];
int serverPort;
} user;
typedef struct {
int length;
char data[MAX_LETTERS_STRINGS][DEFAULT_BUFLEN];
char lastMsg[DEFAULT_BUFLEN];
int expectedMsg;
} dataList;
typedef struct {
char commandLine[DEFAULT_BUFLEN];
char** arguments;
int argumentsSize;
} command;
typedef int (*servCmdHndlrPtr)(user* client, command* cmd, dataList* response);
typedef int (*consoleCmdHndlrPtr)(command* cmd);
typedef int (*cmdHndlrPtr)(command* c, dataList* dl);
void addToList(dataList* dl, char* string);
void printDataList(dataList* dl);
void clearList(dataList* dl);
void initCommand(command* c);
void clearCommand(command* c);
void parseCommand(command* c);
#endif /* MYTYPES_H */
```

server.c

```
#include "constants.h"
#include "myTypes.h"
#include "serverCommandHandlers.h"
#include "stdio.h"
#include "serverConsoleCommandHandlers.h"
#include "connectionFunctions.h"
#include "serverUtils.h"
int initUserList();
int getFreePlace();
int addClient(struct sockaddr_in* clientAddr, int clientLen);
static void *clientHandler(void *args);
static void *consoleHandler(void *args);
user userlist[MAX_USERS];
pthread_mutex_t userlist_lock;
pthread_mutex_t users_file_lock;
pthread_mutex_t file_system_lock;
int serverMain(char* port) {
pthread_mutex_init(&userlist_lock, NULL);
pthread_mutex_init(&users_file_lock, NULL);
pthread_mutex_init(&file_system_lock, NULL);
initUserList();
// создаем поток, обрабатывающий консольный ввод
pthread_t ConsoleThread;
```

```
if (pthread_create(&ConsoleThread, NULL, consoleHandler, NULL) != 0) {
printf("Can't create console thread\n");
exit(1);
}
// создаем слушающий сокет
int listenSocket = createServer(NULL, port);
char buf[DEFAULT_BUFLEN];
// принимаем подключения
for (;;) {
bzero(buf, DEFAULT_BUFLEN);
// создаем стуктуру, где будем хранить адрес того, кто прислал запрос на
подключение
struct sockaddr_in clientAddr;
bzero(&clientAddr, sizeof (clientAddr));
int clientAddrSize = sizeof (clientAddr);
// получаем запрос на подключение
int n = recvfrom(listenSocket, buf, DEFAULT_BUFLEN, 0, (struct sockaddr*)
&clientAddr, &clientAddrSize);
// проверяем, может мы его уже подключили а это "заблудившийся" запрос
int idx = getClientByIpAndPort(inet_ntoa(clientAddr.sin_addr),
ntohs(clientAddr.sin_port));
if (idx != -1) { // если уже регистрировался у нас, шлем ответ от имени сокета,
выделенного ему
send(userlist[idx].sock, "+", 1, 0); // шлем просто сендом т.к. в функции
addClient выполняем connect
continue;
}
// если это новая заявка на подключение
idx = addClient(&clientAddr, clientAddrSize);
if (idx == -1) { // если нету места, шлем ошибку
bzero(buf, DEFAULT_BUFLEN);
strcpy(buf, "-ERR: no free slots");
sendto(listenSocket, buf, n, 0, (struct sockaddr*) &clientAddr, clientAddrSize);
continue;
}
// если есть место создаем для клиента собственный поток
pthread_t tr1; // это лучше либо где-то хранить, либо вобще не исползовать
if (pthread_create(&tr1, NULL, clientHandler, (void*) idx) != 0) {
printf("cant create clients thread\n");
exit(1); //может не стоит так резко по exit выходить?
}
}
closeAllSockets();
pthread_mutex_destroy(&userlist_lock);
pthread_mutex_destroy(&users_file_lock);
pthread_mutex_destroy(&file_system_lock);
return 0;
}
static void *clientHandler(void *args) {
int index = (int) args;
pthread_mutex_lock(&userlist_lock);
user client = userlist[index]; // don't need mutex,?
pthread_mutex_unlock(&userlist_lock);
// шлем клиенту ответ, используя выделенный для него сокет
send(client.sock, "+", 2, 0);
// структура для хранения принятой команды
command cmd;
initCommand(&cmd);
// структура для хранения ответа пользователю
dataList response;
initDataList(&response);
// указатель на функцию, обработчик текущей команды
servCmdHndlrPtr currentCommand;
```

```
// флаг, сигнализирующий о конце работы
int end = 0;
do {
clearCommand(&cmd); // очищаем прошлую команду
clearList(&response);
int res = recvRequest(client.sock, &cmd, &response);
if (res == -1) {
break;
}
parseCommand(&cmd); // выделяем аргументы команды
currentCommand = selectHandler(cmd.arguments[0]); // получаем указатель на
функцию обработчик комады
if ((*currentCommand)(&client, &cmd, &response) == -1) // выполняем команду
end = 1; // если команда вернула ненулевой код, значит возникли ошибки при
передаче данных
} while (end == 0);
// закрываем сокет клиента и сигнализируем что новый клиент может подключится
closeClientByIndex(index);
return 0;
}
static void *consoleHandler(void *args) {
// делаем stdin неблокирующи
int flags = fcntl(F_GETFL, 0);
flags = flags | O_NONBLOCK;
fcntl(0, F_SETFL, flags);
// создаем структуру, где будем хранить пользовательский ввод
command cmd;
initCommand(&cmd);
consoleCmdHndlrPtr currentCommand; // указатель на текущую команду
printServerMenu();
int end = 0;
do {
clearCommand(&cmd); // очищаем прошлую команду
fgets(cmd.commandLine, DEFAULT_BUFLEN, stdin); // считываем новую команду
cmd.commandLine[strlen(cmd.commandLine) - 1] = '\0'; // fgets считывает вместе с
концом строки, который нам не нужен
if (strlen(cmd.commandLine) == 0)
continue;
parseCommand(&cmd); // выделяем аргументы команды
currentCommand = selectConsoleCommand(cmd.arguments[0]); // получаем указатель
на функцию обработчик комады
if ((*currentCommand)(&cmd) != 0) // выполняем команду
end = 1;
} while (end == 0);
closeAllSockets();
printf("Bye-bye\n");
exit(0);
}
int initUserList() {
int i = 0;
for (i = 0; i < MAX_USERS; i++) {
userlist[i].sock = -1;
userlist[i].port = -1;
userlist[i].serverPort = -1;
userlist[i].index = -1;
userlist[i].isLogged = 0;
bzero(userlist[i].ipAddr, sizeof (userlist[i].ipAddr));
bzero(userlist[i].name, sizeof (userlist[i].name));
bzero(userlist[i].serverIP, sizeof (userlist[i].ipAddr));
}
}
// возвращаем первое свободное место в массиве клиентов
int getFreePlace() {
```

```
// mutex вроде не нужен, это ведь только чтение + менять может только этот поток
int i;
pthread_mutex_lock(&userlist_lock);
for (i = 0; i < MAX_USERS; i++) {
if (userlist[i].sock == -1) {
pthread_mutex_unlock(&userlist_lock);
return i;
}
}
pthread_mutex_unlock(&userlist_lock);
return -1;
}
int addClient(struct sockaddr_in* clientAddr, int clientLen) {
int idx = getFreePlace();
if (idx == -1) {
return -1;
}
userlist[idx].index = idx;
userlist[idx].sock = socket(AF_INET, SOCK_DGRAM, 0);
struct sockaddr_in myAddr;
connect(userlist[idx].sock, (struct sockaddr*) clientAddr, clientLen);
strcpy(userlist[idx].ipAddr, inet_ntoa(clientAddr->sin_addr));
userlist[idx].port = ntohs(clientAddr->sin_port);
return idx;
}
```

serverCommandHandlers.c

```
#include "serverCommandHandlers.h"
#include "myTypes.h"
#include "connectionFunctions.h"
#include "serverUtils.h"
#include "fileSystemFunctions.h"
#include "commands.h"
extern pthread_mutex_t userlist_lock;
extern user userlist[MAX_USERS];
// выбирает обработчик команды, заданной в параметре command
servCmdHndlrPtr selectHandler(char* command) {
if (strcmp(command, LOGIN_CMD) == 0)
return loginCmdHandler;
if (strcmp(command, REG_CMD) == 0)
return regCmdHandler;
if (strcmp(command, INFO_CMD) == 0)
return infoCmdHandler;
if (strcmp(command, COUNT_CMD) == 0)
return countCmdHandler;
if (strcmp(command, BODY_CMD) == 0)
return bodyCmdHandler;
if (strcmp(command, DEL_CMD) == 0)
return delCmdHandler;
if (strcmp(command, FIRST_LINES_CMD) == 0)
return firstLinesCmdHandler;
if (strcmp(command, EXIT_CMD) == 0)
return exitCmdHandler;
if (strcmp(command, LOGOUT_CMD) == 0)
return logoutCmdHandler;
if (strcmp(command, SEND_CMD) == 0)
return sendCmdHandler;
return defaultCmdHandler;
}
/* обработчики команд. Список команд в файле commands.h
```

```
* Выполняют обработку команды заданной параметром request и отсылают результат
пользователю
* данные о котором хранит параметр client. Параметр response используется для
хранения
* промежуточного результата и номера сообщения для посылки клиенту
* Возвращают -1 если связь с клиентом утеряна
*/
int loginCmdHandler(user *client, command* request, dataList* response) {
pthread_mutex_lock(&userlist_lock);
int alreadyWorking = isUserPresentNow(request->arguments[1]);
pthread_mutex_unlock(&userlist_lock);
if (alreadyWorking == 1) {
addToList(response, "-ERR: user with this name is working now");
} else if (client->isLogged == 1) {
addToList(response, "-ERR: logout first");
} else if (checkPass(request->arguments[1], request->arguments[2])) {
addToList(response, "-ERR: wrong user name or password");
} else {
addToList(response, "+OK: login successfull");
userLoggin(client, request->arguments[1]);
}
addToList(response, ".");
return sendResponse(client->sock, response);
}
int regCmdHandler(user *client, command* request, dataList* response) {
if (client->isLogged == 1) {
addToList(response, "-ERR: logout first");
} else if (addUsr(request->arguments[1], request->arguments[2]) != 0) {
addToList(response, "-ERR: user with this name already presents");
} else {
addToList(response, "+OK: register successfull");
userLoggin(client, request->arguments[1]);
}
addToList(response, ".");
return sendResponse(client->sock, response);
}
int countCmdHandler(user *client, command* request, dataList* response) {
if (client->isLogged == 0) {
addToList(response, "-ERR: login first");
} else {
int lettersCount = getLettersCount(client->name);
char tmp[DEFAULT_BUFLEN];
lettersCount == -1 ? sprintf(tmp, "-ERR: can't read directory") : sprintf(tmp,
"+OK: %d", lettersCount);
addToList(response, tmp);
}
addToList(response, ".");
return sendResponse(client->sock, response);
}
int infoCmdHandler(user* client, command* request, dataList* response) {
if (client->isLogged == 0) {
addToList(response, "-ERR: login first");
} else {
//проверяем не пуста ли папка
int numLetters = getLettersCount(client->name);
if (numLetters == 0) {
addToList(response, "+OK: directory is empty");
} else {
// если просят информацию обо всех письмах
if (request->argumentsSize == 1) {
int i;
for (i = 1; i <= numLetters; ++i) { //TODO CHECKKKKK!
getInfoAboutLetter(client, response, i);
```

```
printDataList(response);
if (sendResponse(client->sock, response) == -1) {
return -1;
}
clearList(response);
}
} else { // если просят информацию только об одном письме
// проверяем корректный ли номер письма
if (atoi(request->arguments[1]) > numLetters || atoi(request->arguments[1]) <=
0) { // вторая проверка на всякий случай
addToList(response, "-ERR: no letter with this number");
} else
getInfoAboutLetter(client, response, atoi(request->arguments[1]));
}
}
}
addToList(response, ".");
return sendResponse(client->sock, response);
}
int bodyCmdHandler(user *client, command* request, dataList * response) {
if (client->isLogged == 0) {
addToList(response, "-ERR: login first");
} else {
// проверяем есть ли вобще такое письмо
int letterNumber = atoi(request->arguments[1]);
int lettersCount = getLettersCount(client->name);
if (letterNumber > lettersCount) { // пусть письма нумеруются с 1, главное не
забыть потом
addToList(response, "-ERR: there is no letter with this number");
} else {
// если такое письмо есть, надо его послать
getLetterBody(client, letterNumber, response);
}
}
addToList(response, ".");
return sendResponse(client->sock, response);
}
int delCmdHandler(user *client, command* request, dataList * response) {
if (client->isLogged == 0) {
addToList(response, "-ERR: login first");
} else {
// проверяем есть ли вобще такое письмо
int letterNumber = atoi(request->arguments[1]);
int lettersCount = getLettersCount(client->name);
if (letterNumber > lettersCount) { // пусть письма нумеруются с 1, главное не
забыть потом
addToList(response, "-ERR: there is no letter with this number");
} else { // пытаемся удалить писмьо
int res = deleteLetter(client->name, letterNumber);
addToList(response, res == 0 ? "+OK: letter deleted" : "-ERR: some error while
deleting letter");
}
}
addToList(response, ".");
return sendResponse(client->sock, response);
}
int firstLinesCmdHandler(user *client, command* request, dataList * response) {
if (client->isLogged == 0) { // если не зарегистрирован, шлем ошибку
addToList(response, "-ERR: login first");
} else {
// проверяем есть ли вобще такое письмо
int letterNumber = atoi(request->arguments[1]);
int lettersCount = getLettersCount(client->name);
```

```
if (letterNumber > lettersCount) { // пусть письма нумеруются с 1, главное не
забыть потом
addToList(response, "-ERR: there is no letter with this number");
} else {
// если есть, надо отдать первые M строк этого письма
getLetterHead(client, letterNumber, atoi(request->arguments[2]), response);
}
}
addToList(response, ".");
return sendResponse(client->sock, response);
}
int defaultCmdHandler(user *client, command* request, dataList * response) {
addToList(response, "-ERR: unrecognized command");
addToList(response, ".");
return sendResponse(client->sock, response);
}
int exitCmdHandler(user *client, command* request, dataList * response) {
addToList(response,BYE_STRING);
addToList(response, ".");
sendResponse(client->sock, response);
return -1;
}
int logoutCmdHandler(user *client, command* request, dataList * response) {
if (client->isLogged == 0) {
addToList(response, "-ERR: login first");
} else {
userLoggout(client);
addToList(response, "+OK: you are not logged now");
}
addToList(response, ".");
return sendResponse(client->sock, response);
}
int sendCmdHandler(user *client, command* request, dataList * response) {
if (client->isLogged == 0) { // если не зарегистрирован, шлем ошибку
addToList(response, "-ERR: login first");
addToList(response, ".");
return sendResponse(client->sock, response);
}
// проверяем зарегистрирован ли получател
int res = isUsrRegistred(request->arguments[1]);
if (res != 0) {// если получатель не зарегестрирован у нас
addToList(response, "-ERR: no such recipient");
addToList(response, ".");
return sendResponse(client->sock, response);
}
// шлем подтвержение полученного запроса
addToList(response, ".");
if (sendResponse(client->sock, response) == -1) {
return -1;
}
command tmpCmd;
initCommand(&tmpCmd);
dataList tmpLetter;
initDataList(&tmpLetter);
// принимаем тело письма
for (;;) {
clearList(response);
clearCommand(&tmpCmd);
// получаем очередную строку данных
if (recvRequest(client->sock, &tmpCmd, response) == -1) {
return -1;
} else {
```

```
if (strcmp(tmpCmd.commandLine, ".") == 0) { // если точка значит письмо
кончилось
break;
}
addToList(response, "."); // и шлем подтверждение что мы приняли очередную
строку
if (sendResponse(client->sock, response) == -1) {
return -1;
}
addToList(&tmpLetter, tmpCmd.commandLine); // сохраняем строку во временное
хранилище
}
}
res = saveLetter(client->name, request->arguments[1], &tmpLetter); // сохраняем
пиьсмо уже на диск
res == -1 ? addToList(response, "-ERR: can't save letter") : addToList(response,
"+OK: letter saved");
addToList(response, ".");
return sendResponse(client->sock, response);
}
```

serverCommandHandlers.h

```
#ifndef SERVERCOMMANDHANDLERS_H
#define SERVERCOMMANDHANDLERS_H
#include "myTypes.h"
#include "constants.h"
servCmdHndlrPtr selectHandler(char* command);
int loginCmdHandler(user *client, command* request, dataList* response);
int regCmdHandler(user *client, command* request, dataList* response);
int countCmdHandler(user *client, command* request, dataList* response);
int infoCmdHandler(user *client, command* request, dataList* response);
int bodyCmdHandler(user *client, command* request, dataList* response);
int delCmdHandler(user *client, command* request, dataList* response);
int firstLinesCmdHandler(user *client, command* request, dataList* response);
int defaultCmdHandler(user *client, command* request, dataList* response);
int exitCmdHandler(user *client, command* request, dataList* response);
int logoutCmdHandler(user *client, command* request, dataList* response);
int sendCmdHandler(user *client, command* request, dataList* response);
#endif /* SERVERCOMMANDHANDLERS_H */
```

serverConsoleCommandHandlers.c

```
#include "serverConsoleCommandHandlers.h"
#include "myTypes.h"
#include "commands.h"
extern user userlist[];
extern pthread_mutex_t userlist_lock;
extern pthread_mutex_t users_file_lock;
int consoleShow(command* consoleCommand) {
printf("\tConnected users are:\n");
int i = 0;
user u;
for (i = 0; i < MAX_USERS; ++i) {
pthread_mutex_lock(&userlist_lock);
u = userlist[i];
pthread_mutex_unlock(&userlist_lock);
if (u.sock == -1) {
continue;
}
```

```
printf("User # %d\n", u.index + 1);
printf("IP:PORT = %s:%d\n", u.ipAddr, u.port);
u.isLogged == 1 ? printf("Name: %s\n", u.name) : printf("Not authorized\n");
}
return 0;
}
int consoleKill(command* consoleCommand) {
if (consoleCommand->argumentsSize != 2) {
printf("Need 1 argument\n");
return 0;
}
int sockIndex = atoi(consoleCommand->arguments[1]);
if (sockIndex <= 0 || sockIndex >= MAX_USERS) {
printf("Wrong index\n");
return 0;
}
closeClientByIndex(sockIndex - 1);
return 0;
}
int consoleExit(command* consoleCommand) {
return 1;
}
int consoleMenu(command* consoleCommand) {
printServerMenu();
return 0;
}
int consoleDefault(command* consoleCommand) {
printf("Unknown command\n");
return 0;
}
int closeClientByIndex(int index) {
pthread_mutex_lock(&userlist_lock);
if(userlist[index].sock == -1) {
pthread_mutex_unlock(&userlist_lock);
printf("No user with this index\n");
return -1;
}
close(userlist[index].sock);
userlist[index].sock = -1; // показываем, что место свободно
bzero(userlist[index].ipAddr, sizeof (userlist[index].ipAddr));
userlist[index].port = -1;
userlist[index].index = -1;
pthread_mutex_unlock(&userlist_lock);
}
void printServerMenu() {
printf("Server's menu: \n \n");
printf("|=====Commands:==============|==========================|\n");
printf("| show user list -> %8s \n",SHOW_SERV_CMD);
printf("| kill user -> %8s <N> \n",KICK_SERV_CMD);
printf("| finish work -> %8s \n",EXIT_SERV_CMD);
printf("| show menu -> %8s \n",MENU_SERV_CMD);
printf("|============================|=========================================|
\n");
}
consoleCmdHndlrPtr selectConsoleCommand(char* command) {
if (strcmp(command, SHOW_SERV_CMD) == 0)
return consoleShow;
if (strcmp(command, KICK_SERV_CMD) == 0)
return consoleKill;
if (strcmp(command, EXIT_SERV_CMD) == 0)
return consoleExit;
if (strcmp(command, MENU_SERV_CMD) == 0)
return consoleMenu;
```

```
return consoleDefault;
}
```

serverConsoleCommandHandlers.h

```
#ifndef SERVERCONSOLECOMMANDHANDLERS_H
#define SERVERCONSOLECOMMANDHANDLERS_H
#include <errno.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <pthread.h>
#include <time.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/uio.h>
#include "myTypes.h"
#include "constants.h"
consoleCmdHndlrPtr selectConsoleCommand(char* command);
int consoleShow(command * consoleCommand);
int consoleKill(command * consoleCommand);
int consoleExit(command * consoleCommand);
int consoleMenu(command * consoleCommand);
int consoleDefault(command * consoleCommand);
int closeClientByIndex(int index);
void printServerMenu();
#endif /* SERVERCONSOLECOMMANDHANDLERS_H */
```

serverUtils.c

```
#include "serverUtils.h"
#include "dirent.h"
#include "constants.h"
#include "myTypes.h"
extern user userlist[];
extern pthread_mutex_t userlist_lock;
extern pthread_mutex_t users_file_lock;
void userLoggin(user *client, char* userName) {
// сохраняем имя пользователя, и тот факт что он авторизировался
client->isLogged = 1;
strcpy(client->name, userName);
pthread_mutex_lock(&userlist_lock);
userlist[client->index].isLogged = 1;
strcpy(userlist[client->index].name, userName);
pthread_mutex_unlock(&userlist_lock);
}
void userLoggout(user *client) {
client->isLogged = 0;
bzero(client->name, DEFAULT_BUFLEN);
pthread_mutex_lock(&userlist_lock);
userlist[client->index].isLogged = 0;
bzero(userlist[client->index].name, DEFAULT_BUFLEN);
pthread_mutex_unlock(&userlist_lock);
}
int addUsr(char* userName, char*pass) {
```

```
// добавляет пользователя, если такого еще не существует, создает ему папку
if (isUsrRegistred(userName) == 0) // если такой пользователь уже есть
return -1;
pthread_mutex_lock(&users_file_lock);
FILE *f = fopen(USERS_FILE, "a");
if (f == NULL) {
pthread_mutex_unlock(&users_file_lock);
return -1;
}
// дописываем в файл пользователя
char buf[DEFAULT_BUFLEN];
bzero(buf, DEFAULT_BUFLEN);
sprintf(buf, "%s:%s\n", userName, pass);
fputs(buf, f);
fclose(f);
pthread_mutex_unlock(&users_file_lock);
// создаем папку для почты
bzero(buf, DEFAULT_BUFLEN);
sprintf(buf, "mail/%s", userName);
int res = mkdir(buf, S_IRWXU | S_IRWXG | S_IRWXO);
return res == 0 ? 0 : -1;
}
int isUsrRegistred(char* userName) {
pthread_mutex_lock(&users_file_lock);
FILE *f = fopen(USERS_FILE, "r");
if (f == NULL) {
pthread_mutex_unlock(&users_file_lock);
return -1;
}
// читаем файл построчно и сравниваем с текущим именем пользователя
char buf[DEFAULT_BUFLEN];
bzero(buf, DEFAULT_BUFLEN);
char *last;
while (!feof(f)) {
bzero(buf, DEFAULT_BUFLEN);
fgets(buf, DEFAULT_BUFLEN, f);
// выделяем имя пользователя.
char* curUserName = strtok_r(buf, ":", &last);
if (curUserName == NULL) {
fclose(f);
pthread_mutex_unlock(&users_file_lock);
return -1;
}
if (strcmp(curUserName, userName) == 0) { // и проверяем совпадает ли оно с
переданным
fclose(f);
pthread_mutex_unlock(&users_file_lock);
return 0;
}
}
fclose(f);
pthread_mutex_unlock(&users_file_lock);
return -1;
}
int checkPass(char* userName, char*password) {
pthread_mutex_lock(&users_file_lock);
FILE *f = fopen(USERS_FILE, "r");
if (f == NULL) {
pthread_mutex_unlock(&users_file_lock);
return -1;
}
char curUser[DEFAULT_BUFLEN];
bzero(curUser, DEFAULT_BUFLEN);
```

```
sprintf(curUser, "%s:%s", userName, password);
char buf[DEFAULT_BUFLEN];
// читаем файл построчно и сравниваем с текущим именем пользователя и паролем
while (!feof(f)) {
bzero(buf, DEFAULT_BUFLEN);
fgets(buf, DEFAULT_BUFLEN, f);
if (strlen(buf) == 0) {
fclose(f);
pthread_mutex_unlock(&users_file_lock);
return -1;
}
buf[strlen(buf) - 1] = '\0';
if (strcmp(curUser, buf) == 0) { // и проверяем совпадает ли оно с переданным
fclose(f);
pthread_mutex_unlock(&users_file_lock);
return 0;
}
}
fclose(f);
pthread_mutex_unlock(&users_file_lock);
return -1;
}
void closeAllSockets() {
int i;
for (i = 0; i < MAX_USERS; ++i) {
if (userlist[i].sock == -1)
continue;
closeClientByIndex(i);
}
}
int getClientByIpAndPort(char* ip, int port) {
int i = 0;
for (i = 0; i < MAX_USERS; ++i) {
if (strcmp(userlist[i].ipAddr, ip) == 0 && userlist[i].port == port) {
return i;
}
}
return -1;
}
int isUserPresentNow(char* userName) {
int i;
for(i = 0; i < MAX_USERS; ++i) {
if(userlist[i].sock == -1 || strlen(userlist[i].name) == 0)
continue;
if(strcmp(userlist[i].name,userName) == 0)
return 1;
}
return 0;
}
```

serverUtils.h

```
#ifndef SERVERUTILS_H
#define SERVERUTILS_H
#include <errno.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <pthread.h>
#include <time.h>
```

```
#include <netdb.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/uio.h>
#include "myTypes.h"
void userLoggin(user* client, char* userName);
void userLoggout(user* client);
int addUsr(char* userName, char*pass);
int isUsrRegistred(char* userName);
int checkPass(char* userName, char*password);
void closeAllSockets();
int getClientByIpAndPort(char* ip, int port);
int isUserPresentNow(char* userName);
#endif /* SERVERUTILS_H */
```