

Московский Авиационный Институт
(Национальный Исследовательский Университет)
Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

**Курсовой проект по курсу
«Операционные системы»**

Студент: Ткаченко Егор Юрьевич
Группа: М8О-207Б-21
Вариант: 24
Преподаватель: Миронов Евгений Сергеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2022

Содержание

- 1 Репозиторий
- 2 Постановка задачи
- 3 Общие сведения о программе
- 4 Общий метод и алгоритм решения
- 5 Исходный код
- 6 Демонстрация работы программы
- 7 Выводы

Репозиторий

https://github.com/Tnirpps/OS_lab

Постановка задачи

Цель работы

1. Приобретение практических навыков в использовании знаний, полученных в течении курса.
2. Проведение исследования в выбранной предметной области.

Задание

Необходимо спроектировать и реализовать программный прототип в соответствии с выбранным вариантом. Произвести анализ и сделать вывод на основании данных, полученных при работе программного прототипа.

Клиент-серверная система для передачи мгновенных сообщений. Базовый функционал должен быть следующим: клиент может присоединиться к серверу, введя логин; клиент может отправить сообщение другому клиенту по его логину; клиент в реальном времени принимает сообщения от других клиентов; необходимо предусмотреть возможность создания «групповых чатов». Связь между сервером и клиентом должна быть реализована при помощи memory map.

Общие сведения о программе

Программа распределительного узла компилируется из файла main.c, программа клиентского узла компилируется из файла user.c. В программе используется библиотека для работы с shared memory и mutex. В программе используются следующие системные вызовы:

- 1 shm_open() – создаёт/открывает объект общей памяти.
- 2 shm_unlink() – обратная к shm_open().
- 3 ftruncate() – устанавливает файлу заданную длину в байтах.
- 4 close() – закрывает файловый дескриптор.
- 5 pthread_mutex_lock() – блокирует мьютекс.
- 6 pthread_mutex_unlock() – разблокирует мьютекс.
- 7 pthread_cond_wait() – открывает мьютекс и ждёт изменения состояния переменной условия.
- 8 pthread_cond_signal() – посылает сигнал на разблокировку.
- 9

Общий метод и алгоритм решения

Для реализации поставленной задачи необходимо:

1. Реализовать способ общения между процессами
2. Наладить блокировку процессов, чтобы избежать «race condition»
3. Добавить возможность создания групповых чатов.

Исходный код

```
===== main.c =====  
  
#include <fcntl.h>  
#include <pthread.h>  
#include <stdio.h>  
#include <string.h>  
#include <sys/mman.h>  
#include <sys/stat.h>  
#include <sys/types.h>  
#include <unistd.h>  
#include <assert.h>  
#include <malloc.h>  
  
#define COLOR_WHITE_BOLD_UNDERLINED "\e[1;04m"  
#define COLOR_GREEN "\033[1;32m"  
#define COLOR_ORANGE "\033[1;33m"  
#define COLOR_MAGENTA "\033[1;35m"  
#define COLOR_OFF "\e[m"  
  
const char* COLORS_PRINT[6] = {COLOR_OFF, "", COLOR_WHITE_BOLD_UNDERLINED,  
COLOR_ORANGE, COLOR_MAGENTA, COLOR_GREEN};  
  
const char* SHARED_SERVER_NAME = "shared_SeRvEr";  
const char* SHARED_MUTEX_NAME = "shared_mutex";  
const char* SHARED_COND_NAME = "shared_cond";  
const int MAX_U_COUNT = 100;  
  
#define check_ok(VALUE, OK_VAL, MSG) if (VALUE != OK_VAL) { printf("%s", MSG); return 1; }
```

```
#define check_wrong(VALUE, WRONG_VAL, MSG) if (VALUE == WRONG_VAL) { printf("%s",  
MSG); return 1; }
```

```
// TODO: create structure for msg. Now it looks terrible
```

```
const int MSG_LEN_LIMIT = 128;
```

```
const int NICK_MAX_LEN = 8;
```

```
const char SERVER = 1;
```

```
const char EXE = 2;
```

```
const int SHARED_MEMORY_SIZE = 1 + 2 * NICK_MAX_LEN + 1 + MSG_LEN_LIMIT;
```

```
#define LEN 8
```

```
#define COUNT 100
```

```
typedef struct {
```

```
    char name[LEN];
```

```
    int connected [COUNT];
```

```
} chat;
```

```
void print(const char* msg, unsigned color) {
```

```
    if (color >= 6) {
```

```
        printf("%s", msg);
```

```
    } else {
```

```
        printf("%s", COLORS_PRINT[color]);
```

```
        printf("%s", msg);
```

```
        printf("%s", COLORS_PRINT[0]);
```

```
    }
```

```
}
```

```
int authorization(const char* string, char (*users)[NICK_MAX_LEN], int count) {
```

```
    for (int i = 0; i < count; ++i) {
```

```
        int good = 1;
```

```
        int j = 0;
```

```
        while (users[i][j] != '\0' && string[j] != '\0') {
```

```
            if (users[i][j] != string[j]) {
```

```
                good = 0;
```

```
                break;
```

```
            }
```

```
            /// deb
```

```

        if (j >= NICK_MAX_LEN) {
            printf("ERRRRRORO\n");
            return 0;
        }
        j++;
    }
    if (good == 1 && users[i][j] == string[j]) {
        return i + 1;
    }
}
return 0;
}

int is_for_server(const char* shared) {
    return (shared[0] == SERVER);
}

int get_user_index(char (*users)[NICK_MAX_LEN], char* user) {
    for (int i = 0; i < MAX_U_COUNT; ++i) {
        if (strcmp(users[i], user) == 0) {
            return i;
        }
    }
    return -1;
}

//[flag server;dest;poster; option;msg...msg]
void send_msg(const char server, const char* dest, const char* poster, const char option, const char*
msg, char* envelope) {
    memset(envelope, server, sizeof(char));
    memcpy(envelope + 1, dest, NICK_MAX_LEN);
    memcpy(envelope + 1 + NICK_MAX_LEN, poster, NICK_MAX_LEN);
    memset(envelope + 1 + 2 * NICK_MAX_LEN, option, sizeof(char));
    memcpy(envelope + 2 + 2 * NICK_MAX_LEN, msg, MSG_LEN_LIMIT);
}

int server(char (*users)[NICK_MAX_LEN], chat* chats, int count, int c_count) {
    int fd;

```

```

int fdMutex;
int fdCond;
pthread_mutex_t* mutex;
pthread_cond_t* condition;

pthread_mutexattr_t mutex_attribute;
check_ok(pthread_mutexattr_init(&mutex_attribute), 0, "Error initializing mutex attribute!\n")
check_ok(pthread_mutexattr_setpshared(&mutex_attribute, PTHREAD_PROCESS_SHARED), 0,
"Error sharing mutex attribute!\n")

pthread_condattr_t condition_attribute;
check_ok(pthread_condattr_init(&condition_attribute), 0, "Error initializing cond attribute!\n")
check_ok(pthread_condattr_setpshared(&condition_attribute, PTHREAD_PROCESS_SHARED), 0,
"Error sharing cond attribute!\n")

/* Shared file */
fd = shm_open(SHARED_SERVER_NAME, O_RDWR | O_CREAT, S_IRWXU);
check_wrong(fd, -1, "Error creating shared file!\n")
check_ok(ftruncate(fd, SHARED_MEMORY_SIZE), 0, "Error truncating shared file!\n")
/* Shared mutex */
fdMutex = shm_open(SHARED_MUTEX_NAME, O_RDWR | O_CREAT, S_IRWXU);
check_ok(ftruncate(fdMutex, sizeof(pthread_mutex_t)), 0, "Error creating shared mutex file!\n")

mutex = (pthread_mutex_t*) mmap(NULL, sizeof(pthread_mutex_t), PROT_READ | PROT_WRITE,
MAP_SHARED, fdMutex, 0);
check_wrong(mutex, MAP_FAILED, "Error mapping shared mutex!\n")
check_ok(pthread_mutex_init(mutex, &mutex_attribute), 0, "Error initializing mutex!\n")

/* Shared cond */
fdCond = shm_open(SHARED_COND_NAME, O_RDWR | O_CREAT, S_IRWXU);
check_ok(ftruncate(fdCond, sizeof(pthread_cond_t)), 0, "Error creating shared cond file!\n")

condition = (pthread_cond_t*) mmap(NULL, sizeof(pthread_cond_t), PROT_READ | PROT_WRITE,
MAP_SHARED, fdCond, 0);
check_ok(pthread_cond_init(condition, &condition_attribute), 0, "Error initializing cond!\n")

check_ok(pthread_mutexattr_destroy(&mutex_attribute), 0, "Error destroying mutex attribute!\n")
check_ok(pthread_condattr_destroy(&condition_attribute), 0, "Error destroying cond attribute!\n")

```

```

char* sharedFile;

sharedFile = mmap(NULL, SHARED_MEMORY_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, 0);

check_wrong(sharedFile, MAP_FAILED, "Error creating shared file!")

sharedFile[0] = 0;

sharedFile[2 * NICK_MAX_LEN + 1] = 99;

char c;

// options:
// 0 => reg;
// 1 => send msg
// 99 => empty


// flag: = SERVER or = 0 or EXE
// dest = кому отправлять сообщение
// poster = от кого оно пришло
// [flag server;dest;poster;option;msg...msg]
char dest[NICK_MAX_LEN];
char poster[NICK_MAX_LEN];
char* msg = sharedFile + 2 + 2 * NICK_MAX_LEN;
char command;

// while (1) {
while (1) {
    check_ok(pthread_mutex_lock(&mutex), 0, "Error locking mutex on server!\n")
    printf("wait for msg...\n");
    while (is_for_server(sharedFile) != 1) {
        check_ok(pthread_cond_wait(&condition, &mutex), 0, "Error waiting cond on server!\n")
    }
    printf("smth for me\n");
    // init of server
    memcpy(dest, sharedFile + 1, NICK_MAX_LEN);
    memcpy(poster, sharedFile + NICK_MAX_LEN + 1, NICK_MAX_LEN);
    command = sharedFile[2 * NICK_MAX_LEN + 1];

    if (command == 0) {
        int id = authorization(poster, users, count);
        memcpy(sharedFile + 1, poster, NICK_MAX_LEN);
        //      memcpy(sharedFile + NICK_MAX_LEN + 1, ADMIN, NICK_MAX_LEN);

```



```

printf("%s\n", poster);
if (id > 0) {
    // success
    sharedFile[2 * NICK_MAX_LEN + 1] = 1;
} else {
    // fail
    sharedFile[2 * NICK_MAX_LEN + 1] = 0;
}
} else if (command == 1) {
    int id = get_user_index(users, dest);
    if (id == -1) {
        int exist = 0;
        for (int i = 0; i < c_count; ++i) {
            if (strcmp(chats[i].name, dest) == 0) {
                int j = 0;
                while (chats[i].connected[j] != -1) {
                    send_msg(EXE, users[chats[i].connected[j]], poster, command, msg, sharedFile);
                    printf("[%s -> %s] %s", poster, users[chats[i].connected[j]], msg);
                    ++j;
                }
                for (int t = 0; t < count; ++t) {
                    check_ok(pthread_cond_signal(condition), 0, "Error sending signal on server!\n")
                }
                check_ok(pthread_mutex_unlock(mutex), 0, "Error unlocking mutex on server!\n")

                check_ok(pthread_mutex_lock(mutex), 0, "Error locking mutex on server!\n")
                printf("wait for empty....\n");
                // fixed bug if smb send while this loop don't exec
                while (sharedFile[2 * NICK_MAX_LEN + 1] != 99) {
                    check_ok(pthread_cond_wait(condition, mutex), 0, "Error waiting cond on server!\n")
n")
                }
            }
        }
        printf("j = %d\n", j);
        exist += j;
    }
}
if (exist == 0) {
    printf("No such chat\n");
}

```

```

        // we don't send, just clear
        sharedFile[2 * NICK_MAX_LEN + 1] = 99;
    }

} else {
    // user [flag server;dest;poster;option;msg...msg]

    send_msg(0, dest, poster, 1, msg, sharedFile);
    printf("типо отправил сообщение\n");
    printf("[%s -> %s] %s", poster, dest, msg);
}

} else {
    printf("command: %d", command);
}

sharedFile[0] = 0;
for (int i = 0; i < count; ++i) {
    check_ok(pthread_cond_signal(condition), 0, "Error sending signal on server!\n")
}
check_ok(pthread_mutex_unlock(mutex), 0, "Error unlocking mutex on server!\n")
}
check_wrong(munmap(sharedFile, SHARED_MEMORY_SIZE), -1, "Error unmapping fd1!")
check_ok(munmap(mutex, sizeof(pthread_mutex_t)), 0, "Error unmapping mutex!\n")
check_ok(munmap(condition, sizeof(pthread_cond_t)), 0, "Error unmapping cond!\n")
check_wrong(shm_unlink(SHARED_SERVER_NAME), -1, "Error unlinking shared file!\n")
check_wrong(shm_unlink(SHARED_MUTEX_NAME), -1, "Error unlinking shared mutex file!\n")
check_wrong(shm_unlink(SHARED_COND_NAME), -1, "Error unlinking shared cond file!\n")

return 0;
}

void read_name(char* s, const char* obj) {
    printf("Enter %s name (no longer than %d chars): ", obj, NICK_MAX_LEN - 1);
    fgets(s, NICK_MAX_LEN, stdin);
    // if line longer than NICK_MAX_LEN
    printf("Have read: ");
    printf("%s", s);
    if (s[NICK_MAX_LEN - 2] != '\0') {

```

```

    s[NICK_MAX_LEN - 2] = '\n';
    while (getc(stdin) != '\n');
}
// remove \n
for (int i = NICK_MAX_LEN - 1; i >= 0; --i) {
    if (s[i] == '\n') {
        s[i] = '\0';
        break;
    }
}
}

void set_user_to_chat(int id, chat* c) {
    for (int i = 0; i < MAX_U_COUNT; ++i) {
        if (c->connected[i] == -1 || c->connected[i] == id) {
            c->connected[i] = id;
            return;
        }
    }
}

int main() {
    char users[MAX_U_COUNT][NICK_MAX_LEN];
    chat chats[MAX_U_COUNT];
    for (int i = 0; i < MAX_U_COUNT; ++i) {
        memset(users[i], 0, NICK_MAX_LEN);
        memset(chats[i].name, 0, NICK_MAX_LEN);
        for (int j = 0; j < MAX_U_COUNT; ++j) {
            chats[i].connected[j] = -1;
        }
    }
    int u_count = 0;
    int c_count = 0;
    print("Hello, this is the best local messenger for Unix!\n", 0);
    int chose = 0;
    while (chose != -1) {
        print("enter your query:\n", 0);
        print("0: add User   ", 3);

```

```

print("1: create Chat ", 3);
print("2: start server ", 5);
print("-1: exit\n", 4);
char s[NICK_MAX_LEN];
int users_in_chat;
memset(s, 0, NICK_MAX_LEN);
scanf("%d", &chose);
// get "\n"
getc(stdin);
switch (chose) {
    case 0:
        if (u_count == MAX_U_COUNT) {
            printf("cannot create more users\n");
            continue;
        }
        read_name(s, "User");
        if (get_user_index(users, s) == -1) {
            memcpy(users[u_count], s, NICK_MAX_LEN);
            ++u_count;
        } else {
            printf("%s have been created already\n", s);
        }
        break;
    case 1:
        if (c_count == MAX_U_COUNT) {
            printf("cannot create more chats\n");
            continue;
        }
        read_name(s, "Chat");
        if (get_user_index(users, s) != -1) {
            printf("invalid name, %s user exist", s);
            continue;
        }
        /*
        * it is possible to create 2 chats with the same name
        * they will work as one common
        */
        memcpy(chats[c_count].name, s, NICK_MAX_LEN);

```

```

printf("Enter number of users to connect to chat '%s': ", chats[c_count].name);
scanf("%d", &users_in_chat);
// get \n
getc(stdin);
if (users_in_chat > MAX_U_COUNT) {
    printf("We cannot add so many users; the limit is: %d", MAX_U_COUNT);
    memset(chats[c_count].name, 0, NICK_MAX_LEN);
    continue;
}
printf("Input %d existing users (one user in each line)\n", users_in_chat);
for (int j = 0; j < users_in_chat; ++j) {
    read_name(s, "User");
    int id = get_user_index(users, s);
    if (id == -1) {
        printf("all users had to be created, we dont have %s; try another user\n", s);
        j--;
        continue;
    }
    set_user_to_chat(id, &chats[c_count]);
}
printf("\n");
++c_count;
break;
case 2:
    printf("\t\tServer has started.\t\t\n");
    server(users, chats, u_count, c_count);
    break;
case -1:
    for (int i = 0; i < u_count; ++i) {
        printf("%s ,", users[i]);
    }
    print("clear ALL\n", 1);
    break;
default:
    print("no such chose \U0001F63F\n", 0);
}
}
printf("end\n");

```

```

    return 0;
}

===== user.c =====

#include <fcntl.h>
#include <pthread.h>
#include <stdio.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <string.h>

#define COLOR_WHITE_BOLD_UNDERLINED "\e[1;04m"
#define COLOR_GREEN "\033[1;32m"
#define COLOR_ORANGE "\033[1;33m"
#define COLOR_MAGENTA "\033[1;35m"
#define COLOR_OFF "\e[m"

#define check_ok(VALUE, OK_VAL, MSG) if (VALUE != OK_VAL) { printf("%s", MSG); return 1; }
#define check_wrong(VALUE, WRONG_VAL, MSG) if (VALUE == WRONG_VAL) { printf("%s",
MSG); return 1; }

const char* COLORS_PRINT[6] = {COLOR_OFF, "", COLOR_WHITE_BOLD_UNDERLINED,
COLOR_ORANGE, COLOR_MAGENTA, COLOR_GREEN};

const char* SHARED_SERVER_NAME = "shared_SeRvEr";
const char* SHARED_MUTEX_NAME = "shared_mutex";
const char* SHARED_COND_NAME = "shared_cond";
const int MAX_U_COUNT = 100;

const int MSG_LEN_LIMIT = 128;
const int NICK_MAX_LEN = 8;
const char SERVER = 1;
const char EXE = 2;
const int SHARED_MEMORY_SIZE = 1 + 2 * NICK_MAX_LEN + 1 + MSG_LEN_LIMIT;

typedef struct {
    char* me;

```

```

char* shared;
pthread_mutex_t* mutex;
pthread_cond_t* cond;
} Token;

void print(const char* msg, unsigned color) {
    if (color >= 6) {
        printf("%s", msg);
    } else {
        printf("%s", COLORS_PRINT[color]);
        printf("%s", msg);
        printf("%s", COLORS_PRINT[0]);
    }
}

int is_for_me(const char* shared, const char* me) {
    for (int i = 0; i < NICK_MAX_LEN; ++i) {
        if (shared[i] != me[i]) {
            return 0;
        }
    }
    return 1;
}

void* pooling(void* arg) {
    Token* token = ((Token*) arg);
    printf("start pooling\n");
    while (1) {
        pthread_mutex_lock(token->mutex);
        while (is_for_me(token->shared + 1, token->me) != 1) {
            pthread_cond_wait(token->cond, token->mutex);
        }
        printf("[%s] %s", token->shared + NICK_MAX_LEN + 1, token->shared + 2 * NICK_MAX_LEN
+ 2);
        memset(token->shared, 0, SHARED_MEMORY_SIZE);
        token->shared[2 * NICK_MAX_LEN + 1] = 99; // make it empty
        pthread_cond_signal(token->cond);
    }
}

```

```

        pthread_mutex_unlock(token->mutex);
    }
    return NULL;
}

void start_pooling(const Token* token) {
    pthread_t th;
    if (pthread_create(&th, NULL, &pooling, (void*)token) != 0) {
        printf("cannot create thread\n");
        return;
    }
}

int parse(char* dest, char* buff) {
    if (buff[0] != '[') return -1;
    int i = 1;
    for (; buff[i] != ']'; ++i) {
        if (i > NICK_MAX_LEN) return -1;
        dest[i - 1] = buff[i];
    }
    for (int j = i; j <= NICK_MAX_LEN; ++j) {
        dest[j - 1] = 0;
    }
    return (i + 2);
}

int main(int argc, char** argv) {
    /* Shared file */
    int fd = shm_open(SHARED_SERVER_NAME, O_RDWR, S_IRWXU);
    check_wrong(fd, -1, "Error opening shared file in child process!\n")
    char* sharedFile = mmap(NULL, SHARED_MEMORY_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, 0);
    check_wrong(sharedFile, MAP_FAILED, "Error mapping shared file in child process!\n")
    /* Shared mutex */
    int fdMutex = shm_open(SHARED_MUTEX_NAME, O_RDWR, S_IRWXU);
    check_wrong(fdMutex, -1, "Error opening shared mutex file in child process!\n")

```



```

pthread_mutex_t* mutex = mmap(NULL, sizeof(pthread_mutex_t), PROT_READ | PROT_WRITE,
MAP_SHARED, fdMutex, 0);
check_wrong(mutex, MAP_FAILED, "Error mapping shared mutex file in child process!\n")
/* Shared cond */
int fdCond = shm_open(SHARED_COND_NAME, O_RDWR, S_IRWXU);
check_wrong(fdCond, -1, "Error opening shared cond file in child process!\n")
pthread_cond_t* condition = mmap(NULL, sizeof(pthread_cond_t), PROT_READ | PROT_WRITE,
MAP_SHARED, fdCond, 0);
check_wrong(condition, MAP_FAILED, "Error mapping shared cond file in child process!\n")

// have to auth
int auth = 0;
char me[NICK_MAX_LEN];
memset(me, 0, NICK_MAX_LEN);
while (auth == 0) {
    printf("Enter User name (no longer than %d chars): ", NICK_MAX_LEN - 1);
    fgets(me, NICK_MAX_LEN, stdin);
    // if line longer than NICK_MAX_LEN
    if (me[NICK_MAX_LEN - 2] != '\0') {
        me[NICK_MAX_LEN - 2] = '\n';
        while (getc(stdin) != '\n');
    }
    // remove \n
    for (int i = NICK_MAX_LEN - 1; i >= 0; --i) {
        if (me[i] == '\n') {
            me[i] = '\0';
            break;
        }
    }
    // send query
    check_ok(pthread_mutex_lock(mutex), 0, "Error locking mutex in child!\n")
    printf("wait for empty....\n");
    while (sharedFile[2 * NICK_MAX_LEN + 1] != 99 || sharedFile[0] == EXE) {
        check_ok(pthread_cond_wait(condition, mutex), 0, "Error waiting cond in child!\n")
    }
    printf("sent\n");
    memset(sharedFile + 1, 0, NICK_MAX_LEN); // can be removed
    memcpy(sharedFile + NICK_MAX_LEN + 1, me, NICK_MAX_LEN);
}

```

```

sharedFile[2 * NICK_MAX_LEN + 1] = 0;
sharedFile[0] = SERVER;
for (int i = 0; i < 10; ++i) {
    check_ok(pthread_cond_signal(condition), 0, "Error sending signal in child!\n")
}
check_ok(pthread_mutex_unlock(mutex), 0, "Error unlocking mutex in child!\n")

check_ok(pthread_mutex_lock(mutex), 0, "Error locking mutex in child!\n")
printf("wait for anwer...\n");
while (is_for_me(sharedFile + 1, me) != 1) {
    check_ok(pthread_cond_wait(condition, mutex), 0, "Error waiting cond in child!\n")
}
auth = sharedFile[2 * NICK_MAX_LEN + 1];
sharedFile[2 * NICK_MAX_LEN + 1] = 99; // make it empty
memset(sharedFile, 0, 2 * NICK_MAX_LEN + 1);
for (int i = 0; i < 10; ++i) {
    check_ok(pthread_cond_signal(condition), 0, "Error sending signal in child!\n")
}
check_ok(pthread_mutex_unlock(mutex), 0, "Error unlocking mutex in child!\n")
if (auth == 1) {
    print("SUCCESS!\n", 5);
    break;
} else {
    /*
    * http://uc.org.ru/node/200
    */
    printf("\033[1;4;31m");
    printf("FAIL=(\n");
    printf("%s", COLOR_OFF);
}
}

Token token = {me, sharedFile, mutex, condition};
start_pooling(&token);
print("Now you can send msg like this:\n  [Vasya01] Hello, how are you??\n  or [!] to exit\n", 1);

// sending msg
char buff [NICK_MAX_LEN + 2 + MSG_LEN_LIMIT];
char dest[NICK_MAX_LEN];

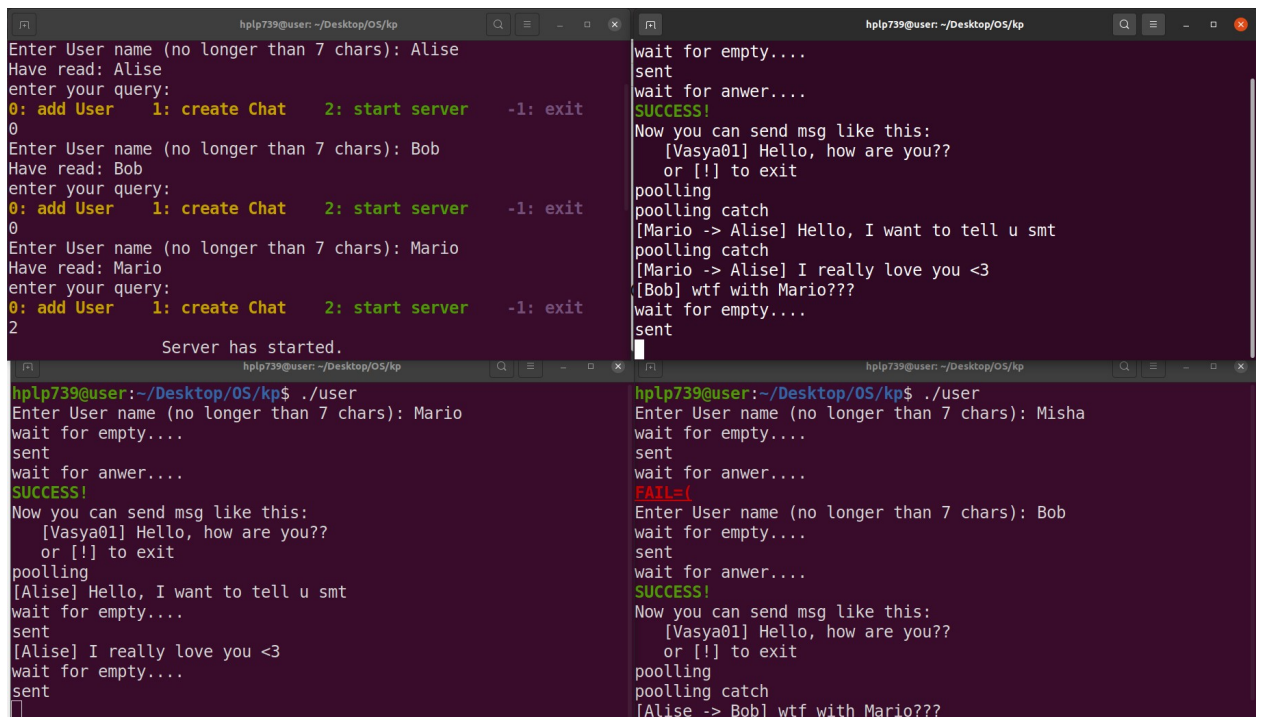
```

```

char poster[NICK_MAX_LEN];
// buff = "[NICK] msg" so + 2
while (fgets(buff, NICK_MAX_LEN + 2 + MSG_LEN_LIMIT, stdin)) {
    if (buff[1] == '!') {
        break;
    }
    int start_msg = parse(dest, buff);
    if (start_msg == -1) {
        print("cannot parse your query\n", 3);
        continue;
    }
    check_ok(pthread_mutex_lock(mutex), 0, "Error locking mutex in child!\n")
    printf("wait for empty....\n");
    while (sharedFile[2 * NICK_MAX_LEN + 1] != 99 || sharedFile[0] == EXE) {
        check_ok(pthread_cond_wait(condition, mutex), 0, "Error waiting cond in child!\n")
    }
    printf("sent\n");
    memcpy(sharedFile + 1, dest, NICK_MAX_LEN);
    memcpy(sharedFile + NICK_MAX_LEN + 1, me, NICK_MAX_LEN);
    memcpy(sharedFile + 2 * NICK_MAX_LEN + 2, buff + start_msg, MSG_LEN_LIMIT);
    sharedFile[2 * NICK_MAX_LEN + 1] = 1;
    sharedFile[0] = SERVER;
    for (int i = 0; i < 10; ++i) {
        check_ok(pthread_cond_signal(condition), 0, "Error sending signal in child!\n")
    }
    check_ok(pthread_mutex_unlock(mutex), 0, "Error unlocking mutex in child!\n")
    memset(buff, 0, NICK_MAX_LEN + 2 + MSG_LEN_LIMIT);
}
}

```

Демонстрация работы программы



```
hplp739@user: ~/Desktop/OS/kp
Enter User name (no longer than 7 chars): Alise
Have read: Alise
enter your query:
0: add User    1: create Chat    2: start server    -1: exit
0
Enter User name (no longer than 7 chars): Bob
Have read: Bob
enter your query:
0: add User    1: create Chat    2: start server    -1: exit
0
Enter User name (no longer than 7 chars): Mario
Have read: Mario
enter your query:
0: add User    1: create Chat    2: start server    -1: exit
2
Server has started.

hplp739@user: ~/Desktop/OS/kp
wait for empty...
sent
wait for answer...
SUCCESS!
Now you can send msg like this:
[Vasya01] Hello, how are you??
or [!] to exit
pooling
pooling catch
[Mario -> Alise] Hello, I want to tell u smt
pooling catch
[Mario -> Alise] I really love you <3
[Bob] wtf with Mario???
wait for empty...
sent

hplp739@user: ~/Desktop/OS/kp$ ./user
Enter User name (no longer than 7 chars): Mario
wait for empty...
sent
wait for answer...
SUCCESS!
Now you can send msg like this:
[Vasya01] Hello, how are you??
or [!] to exit
pooling
[Alise] Hello, I want to tell u smt
wait for empty...
sent
[Alise] I really love you <3
wait for empty...
sent

hplp739@user: ~/Desktop/OS/kp$ ./user
Enter User name (no longer than 7 chars): Misha
wait for empty...
sent
wait for answer...
FAIL=1
Enter User name (no longer than 7 chars): Bob
wait for empty...
sent
wait for answer...
SUCCESS!
Now you can send msg like this:
[Vasya01] Hello, how are you??
or [!] to exit
pooling
pooling catch
[Alise -> Bob] wtf with Mario???
```

Выводы

Составлена и отлажена программа на языке C, реализующая клиент-серверную систему «мгновенных сообщений». Общение между пользователем и сервером осуществляется при помощи методу map. В системе присутствует возможность создания групповых чатов.