Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ Робототехники и комплексной автоматизации

КАФЕДРА Системы автоматизированного проектирования (РК-6)

ОТЧЕТ О ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ

Студент	Никифорова Ирина Андреевна
Группа	РК6-61б
Тип задания	лабораторная работа
Тема лабораторной работы	Сетевое программирование

Студент	_____ *подпись, дата*	**Никифорова И. А.** *фамилия, и.о.*
Преподаватель	_____ *подпись, дата*	**Федорук В.Г.** *фамилия, и.о.*

Оценка _____

Москва, 2019 г.

Оглавление

Задание на лабораторную работу

Разработать сетевой вариант программы моделирования лифтовой системы (см. задание 8 из лаб. работы 1). При этом процессы для БН, ЛП и ЛГ выполняются на отдельных узлах сети (при этом "нажатие" кнопок должно осуществляться на клавиатурах соответствующих узлов).

Описание использованного протокола сетевого взаимодействия

Для сетевого взаимодействия узлов лифтовой системы был использован стандартный протокол TCP.

Описание структуры программы

Для удобства файлы программы были разделены на две части: серверные (блок управления, папка server) и клиентские (лифт, папка client).

Структура директорий изображена на листинге 1. Здесь файлы client/client.c и client/client.h содержат реализацию сетевого взаимодействия на стороне клиента, файлы server/server.h и server/server.c - на стороне сервера. Файл client/elevator.c реализует логику работы лифта. Дополнительный заголовочный файл client/terminal.h содержит макросы для удобства графического вывода в терминал.

Листинг 1. Структура файлов программы. Синий цвет означает директорию, зеленый - исполняемый файл, черный - файл исходного кода.

```
.
├── client
│   ├── client.c
│   ├── client.h
│   ├── elevator.c
│   ├── lift
│   └── terminal.h
├── compile.sh
└── server
    ├── server
    ├── server.c
    └── server.h
```

Исполняемые файлы должны быть запущены каждый в своей консоли или на отдельном сетевом узле в следующем порядке: сначала блок управления (server), после этого лифты (lift) в любом порядке. Все программы должны быть запущены на различных свободных портах. Компиляция выполняется при запуске bash-скрипта (листинг 2).

Листинг 2. Компиляция программ

```
#!/bin/bash

cc client/client.c client/elevator.c -o client/lift -pthread
cc server/server.c -o server/server -pthread
```

Запуск программ описан в листинге 3. Для грузового лифта и для пассажирского запускается одна и та же программа, различаются только аргументы. Сервер по умолчанию запускается на порту 1234, клиент - на заданном порту.

Листинг 3. Пример запуска программ

```
1. запуск лифта: ./lift 1237 5
2. (где 1237 - port, 5 - значение скорости лифта)
3. все файлы, которые нужны в одной папке для компиляции:
4. client.c, client.h, elevator.c
5.
6. запуск: ./server
7. все файлы, которые нужны в одной папке для компиляции:
8. server.c, server.h
9.
10.   сначала нужно запустить сервер, потом клиент
```

Программы общаются по протоколу TCP. Взаимодействие клиентов и сервера изображено на рисунках 1 и 2.

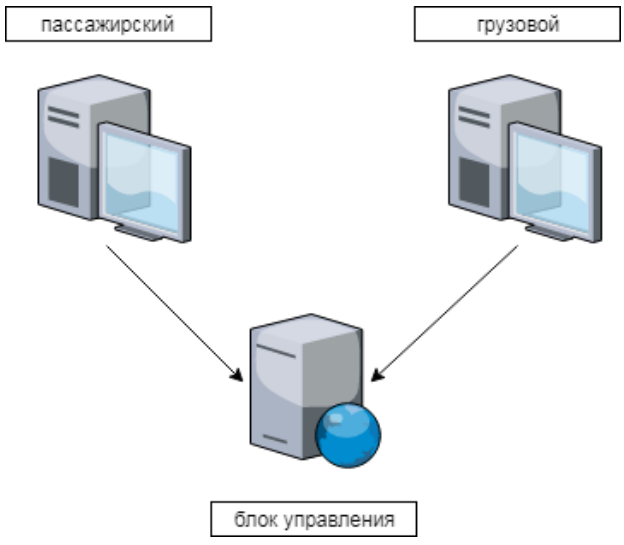

Рис. 1. Клиентские программы запрашивают подключение к управляющему серверу

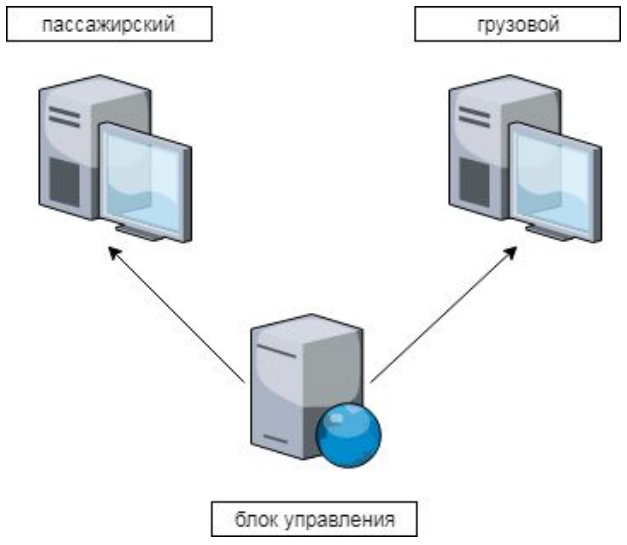

Рис. 2. Блок управления устанавливает соединение и посылает команды клиентским программам

Главный файл исходного кода программы-сервера - server.c. Заголовочный файл для этой программы server.h содержит объявления функций и некоторые константы, такие как номер порта запуска и название сокета в файловой системе и другие.

В начале функции *main* программы server.c происходит привязка сигнала SIGINT, посылаемого при нажатии Ctrl + C, к функции завершения работы сервера *stop_server*. Это необходимо, чтобы при получении сигнала программа освобождала свои ресурсы и только после этого завершала работу. Функция *stop_server* закрывает все открытые сокеты на чтение и на запись и удаляет файл сокета из файловой системы компьютера.

Далее в функции *main* происходит вызов функции *create_socket*, которая открывает сокет программы-сервера. В функции *create_socket* вызываются функции библиотеки socket.h. Сначала, с помощью *socket* создается сокет. Далее, заполняется уже существующая в глобальной области видимости структура *sockaddr*. Для переиспользования адреса сокета вызывается функция *setsockopt*. В конце функции происходит именование сокета с помощью заполненной ранее структуры *sockaddr*. После этого сокет готов к использованию.

Сервер рассчитан на соединение с двумя клиентами. Для того, чтобы одновременно обслуживать два лифта он был написан в многопоточном варианте. Использовалась библиотека pthread.h. Создание потоков и их настройка осуществляется в самой функции main.

Для каждого из потоков была написана своя функция: *lift1* и *lift2*. Они практически идентичны: каждая из них ждет с помощью системного вызова *accept* подключения к первоначально созданному сокету. *Accept* выбирает первое подключение из очереди ждущих для указанного в первом аргументе слушающего сокета. Она также возвращает новый сокет, в который уже можно записывать данные для подключившегося клиента (сам первоначальный сокет

остается неизменным после этого вызова, поэтому можно также ждать подключения на него в другой функции). Так как непонятно, какой из вызовов *accept* сработает первым, неясно какой поток будет отвечать за какой лифт. В контексте данной задачи это и неважно, так как логика работы обоих потоков идентична. Для избежания этой ситуации при различной логике работы лифтов можно вызывать accept в другом потоке (например, в *main*) и далее передавать уже новые сокеты функциям *lift1* и *lift2*.

В этой функции дальше идет отправка номера этажа в нужный лифт (по названию функции) при совпадении номера лифта с исходным в переменной lift_num. После совпадения, lift_num перезаписывается для того, чтобы исходная функция поняла, что соединение было установлено, а этаж передан.

Такая конструкция с lift_num нужна, чтобы в функции *main* управление передавалось каждому лифту по очереди. Это происходит в цикле while: там бесконечно считывается номер этажа и отправляется на следующий лифт.

Функция main завершается ожиданием завершения дочерних потоков.

Не менее важный файл - главный сетевой файл программы-клиента - client.c. Здесь также есть своя функция *create_socket*, абсолютно аналогичная такой же функции в server.c. Тут дополнительно есть функция *set_sockaddr_of_srv*, позволяющая заполнить структуру sockaddr для сервера, и функция *stop_client*, завершающая работу клиента. Применение этих функций будет в файле elevator.c. Они были вынесены в отдельный файл, так как непосредственно связаны с передачей данных по сети. В файле client.h перечислены необходимые константы и объявления описанных выше функций.

Файл, реализующий логику работы лифта - elevator.c. В нем описывается структура, описывающая параметры лифта в каждый конкретный момент - она была названа Elevator.

Сам лифт, над которым выполняют преобразования все функции, в программе называется lift. Для удобства он задан глобально. В начале для него определены исходные параметры.

Для работы со структурой lift были созданы основные функции. Первая из них - *go*. Она отправляет лифт на этаж, переданный ей первым параметром. Вторым параметром она принимает константу INSIDE или OUTSIDE - место, откуда был вызван лифт.

Функция *go* проверяет доступность этажа в данном лифте, определяет в какую сторону и сколько этажей двигаться, а затем двигает лифт в нужном направлении с его скоростью. После прибытия она уменьшает или увеличивает количество человек в лифте в зависимости от второго параметра, переданного в функцию. Задано, что при вызове изнутри, на остановке один человек покинет лифт, а при вызове снаружи - зайдет в лифт.

Для уменьшения или увеличения количества человек в лифте были написаны специальные функции *add_persons* и *remove_persons*.

Программа-лифт должна выполнять сразу три функции: прием вызовов на узле запуска, прием сообщений с сервера и отображение - именно поэтому она была написана в многопоточной форме. Для каждой из задач была написана своя функция, выполняющаяся в отдельном потоке. Создание и управление потоками осуществляется в функции *main* (там также осуществляется проверка аргументов программы и очистка экрана, но ничего более).

Для приема сообщений с сервера в отдельном потоке работает функция *message*. Она осуществляет создание и открытие сокета, а потом, с помощью библиотечной функции *connect* пытается соединиться с сервером. Как только соединение осуществлено, она запускает бесконечный цикл считывания сообщений, где принимает, печатает и выполняет команды с сервера, если лифт сейчас не двигается. Если лифт двигается, то прежде, чем выполнить команду, будет ожидание его остановки.

Чтобы принимать вызовы внутри лифта (те, которые набираются на сетевом узле самого лифта), была написана функция calls. Она также работает в отдельном потоке. Ее задача - считывание нового этажа и отправка на него лифта, как только он будет готов выполнять задачу.

Функция view занимается отображением состояния лифта. Она максимально проста - бесконечно, каждую секунду она отрисовывает состояние лифта, предварительно очистив ранее заполненные части экрана. Для этого используются макросы файла terminal.h - так осуществляется псевдографический вывод.

Описание основных используемых структур данных

Для хранения данных о лифте была использована стандартная структура данных языка СИ - структура. Также, для хранения буфера передачи по сети, была использована стандартная структура данных массив.

Блок-схема программы

Блок-схема логики работы основной функции (*main*) программы-сервера представлена на рис.4.

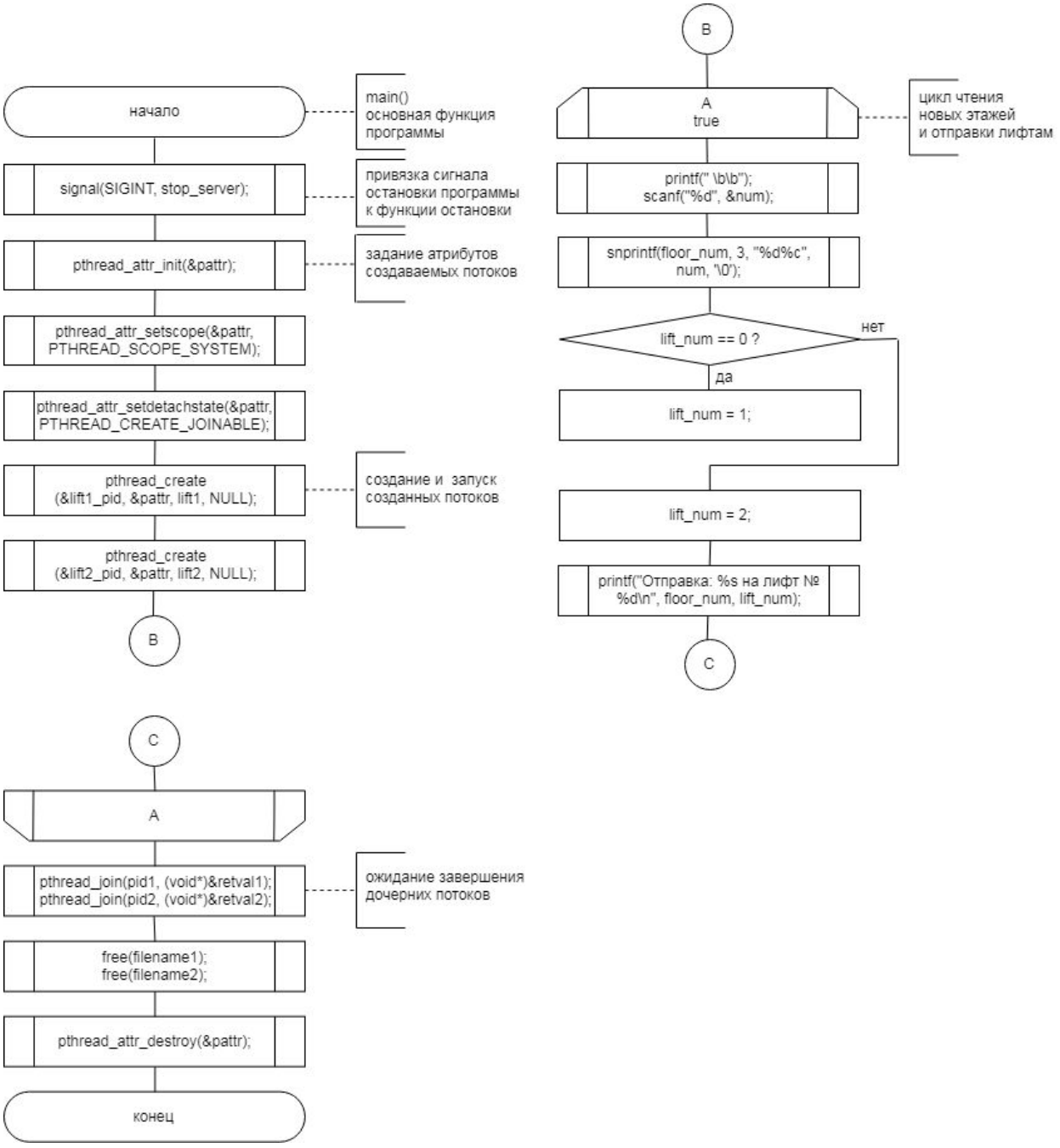

Рис.4. Блок-схема функции *main* программы-сервера.

Для программы-клиента были создана блок-схема функции *message* (рис.5), так как она является основной для данной программы.

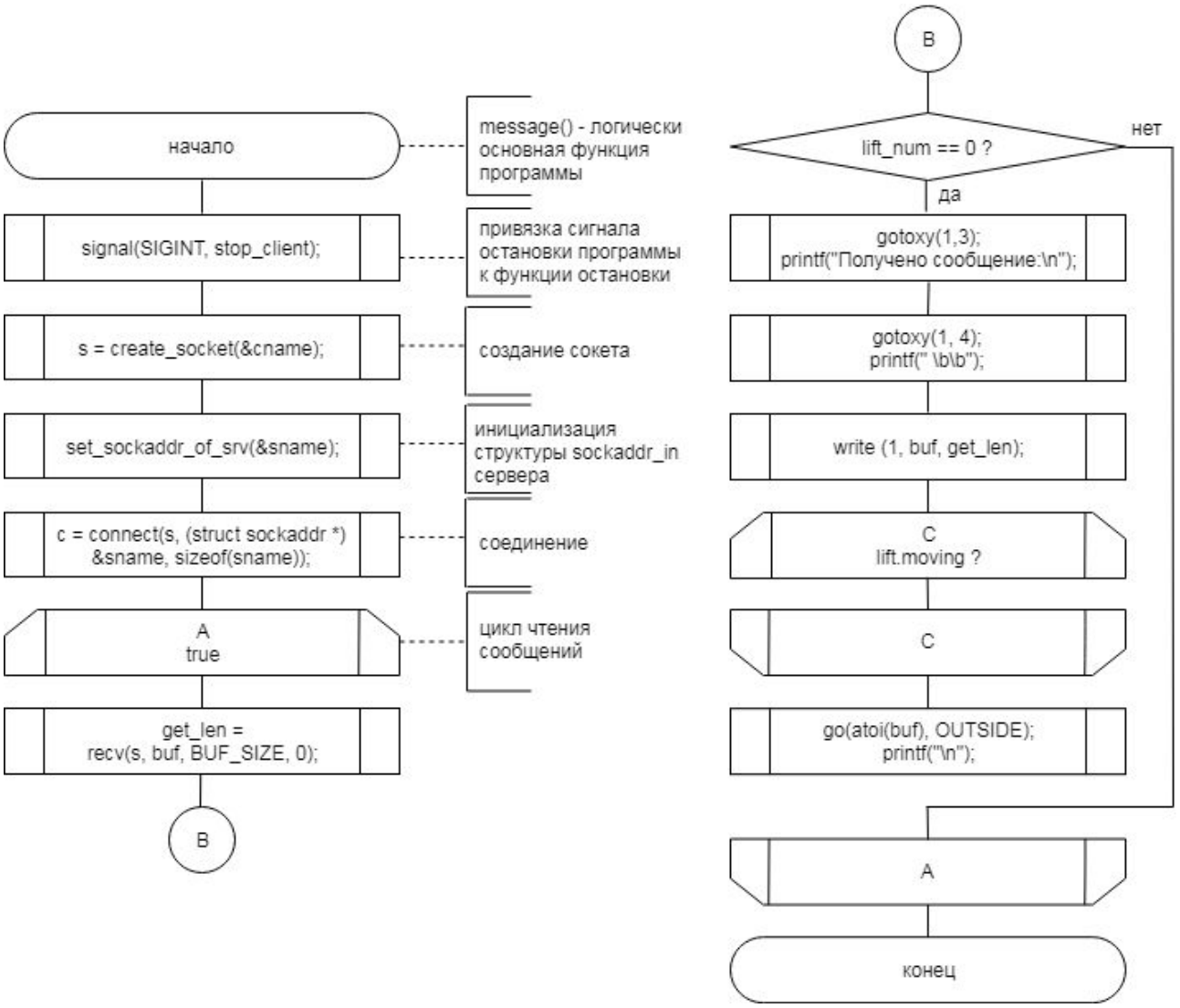

Рис. 5. Блок-схема логически основной функции программы-клиента message

Примеры результатов работы программы

На рисунках 6 - 13 представлены примеры результатов работы программы.

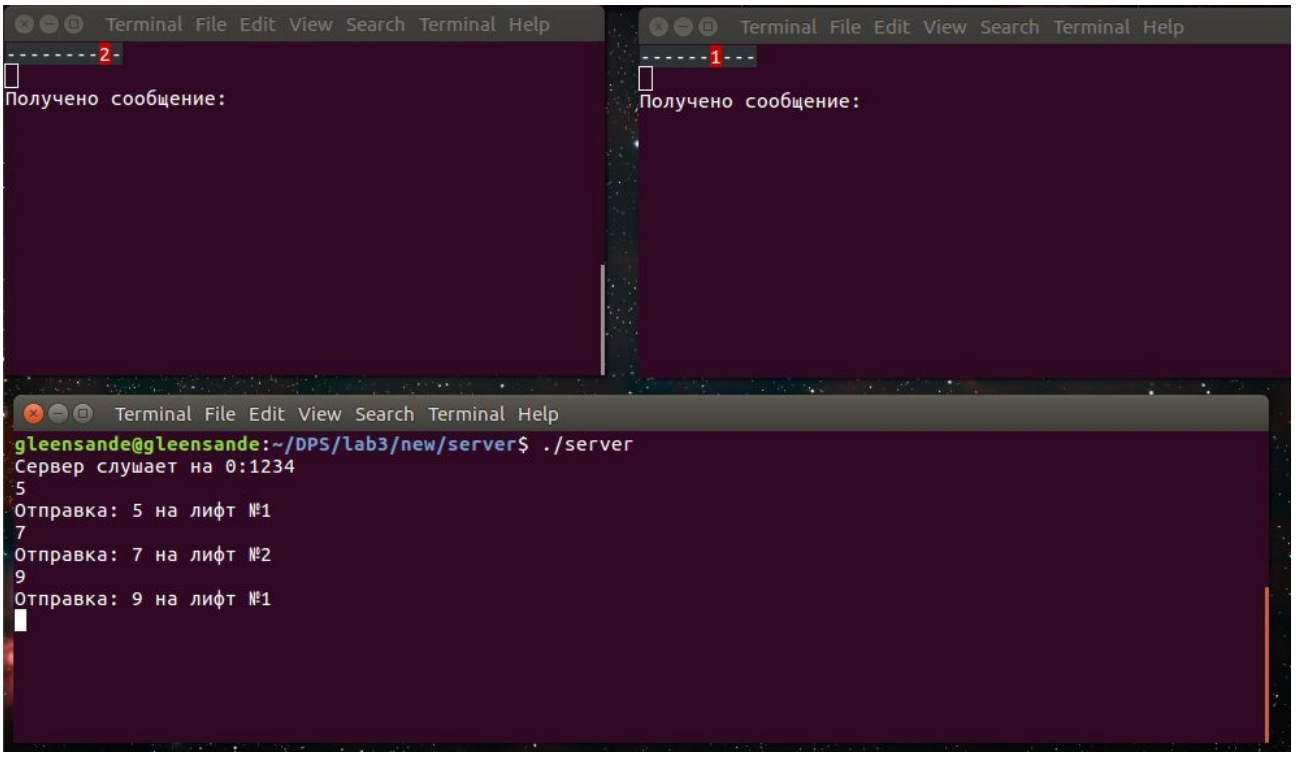

Рис. 6. Количество пассажиров в лифте изменяется. В лифте №1 (левая консоль вверху) находится два пассажира

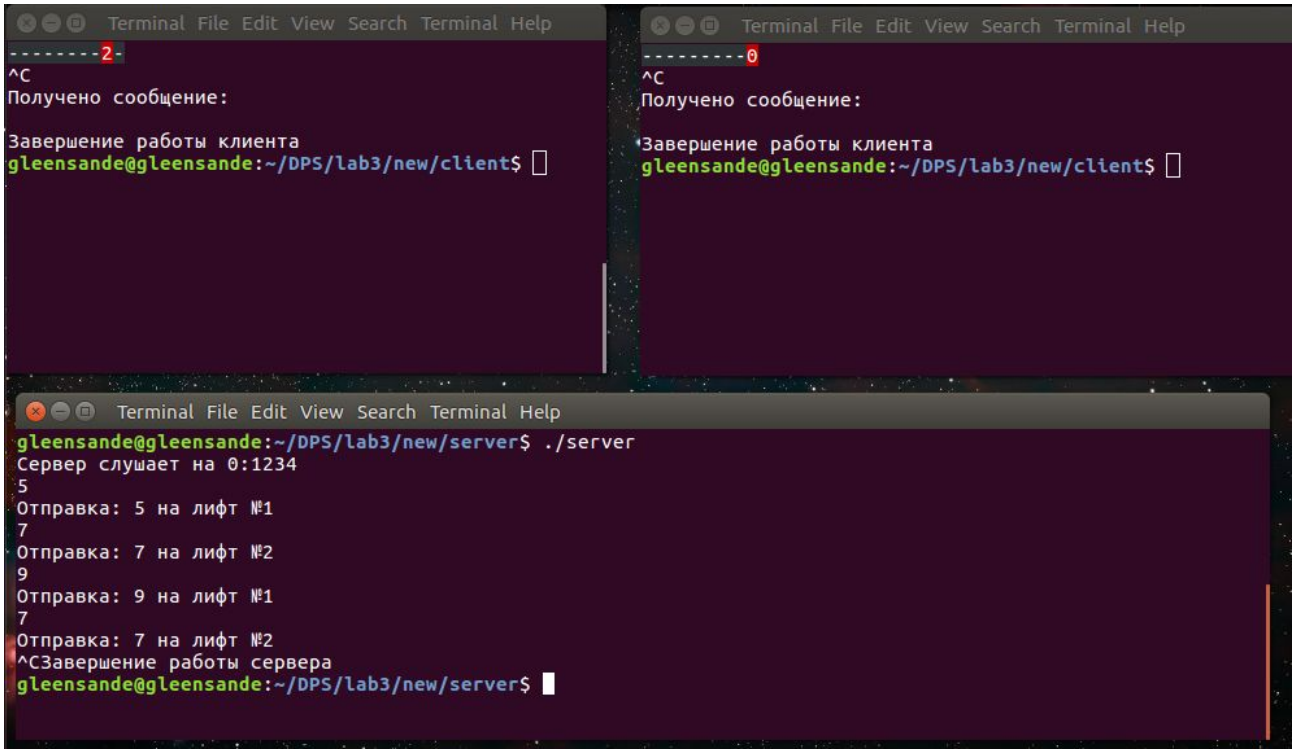

Рис. 7. Завершение работы лифтов и сервера

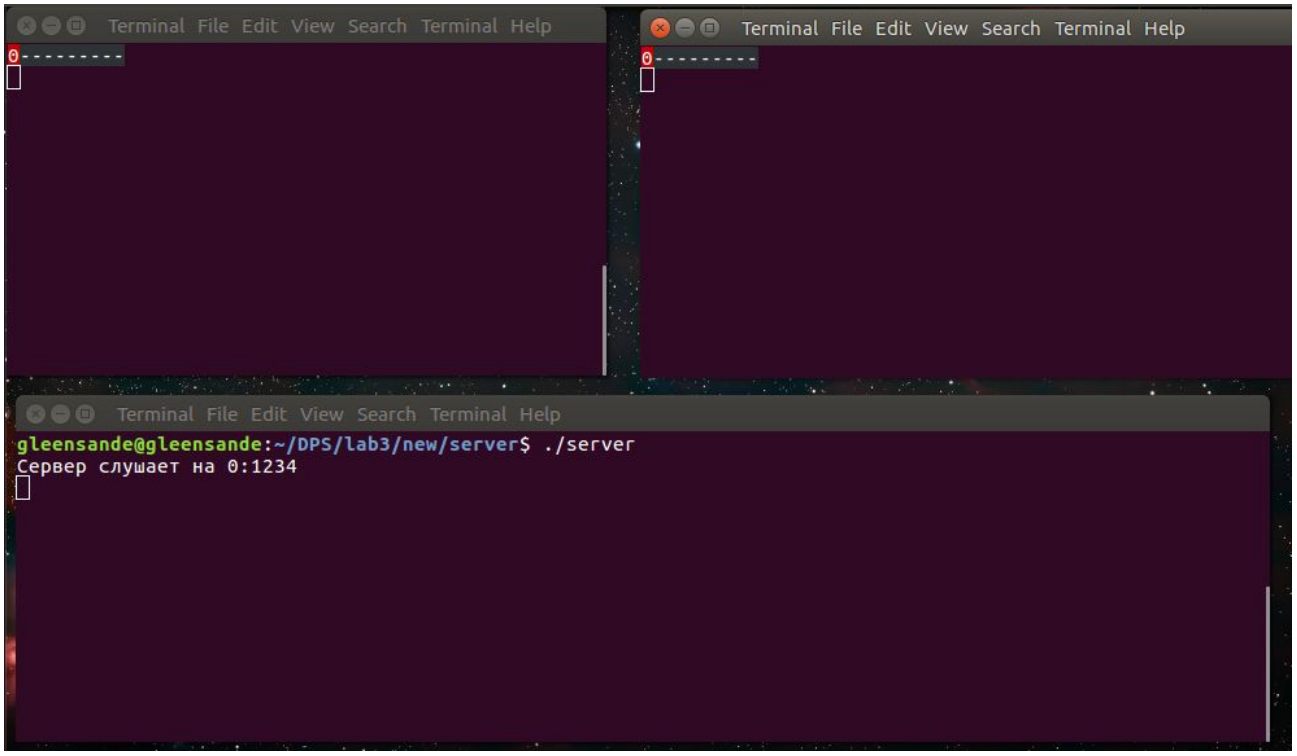

Рис. 8. Запуск клиентов (верхние консоли)

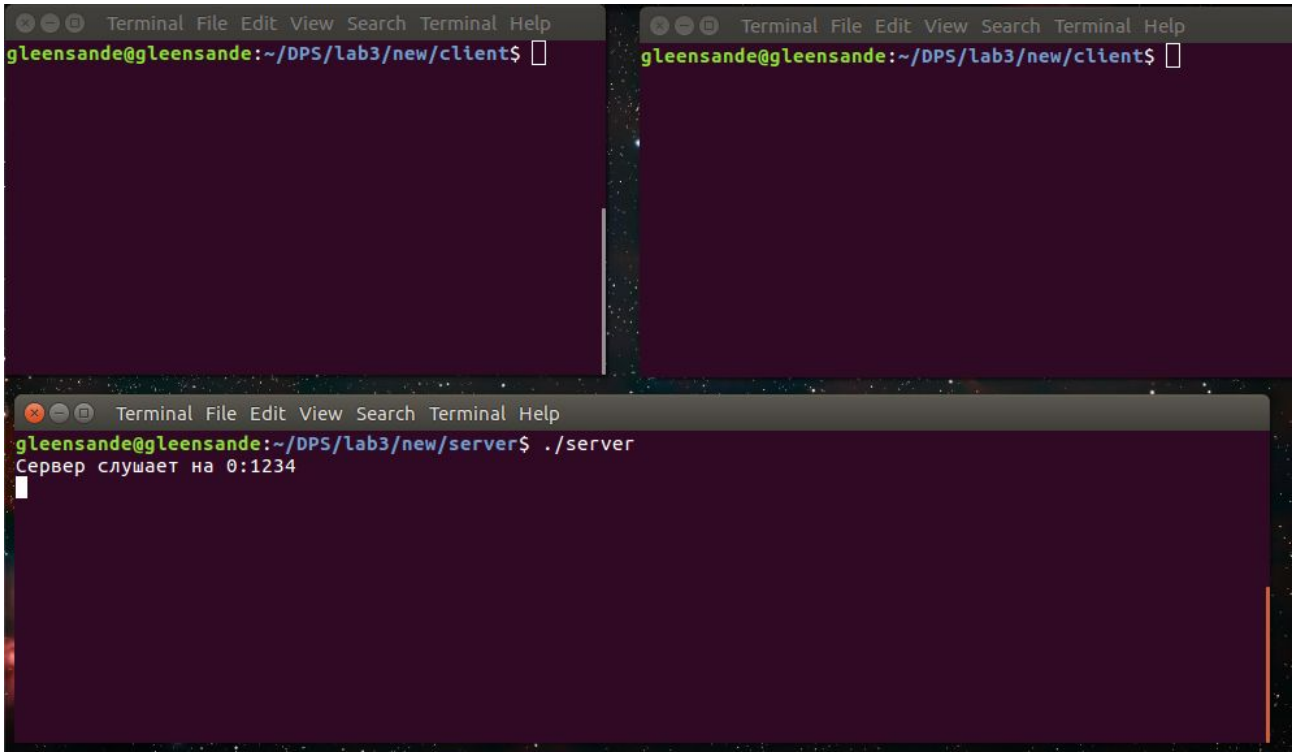

Рис. 9. Запуск сервера (нижняя консоль)

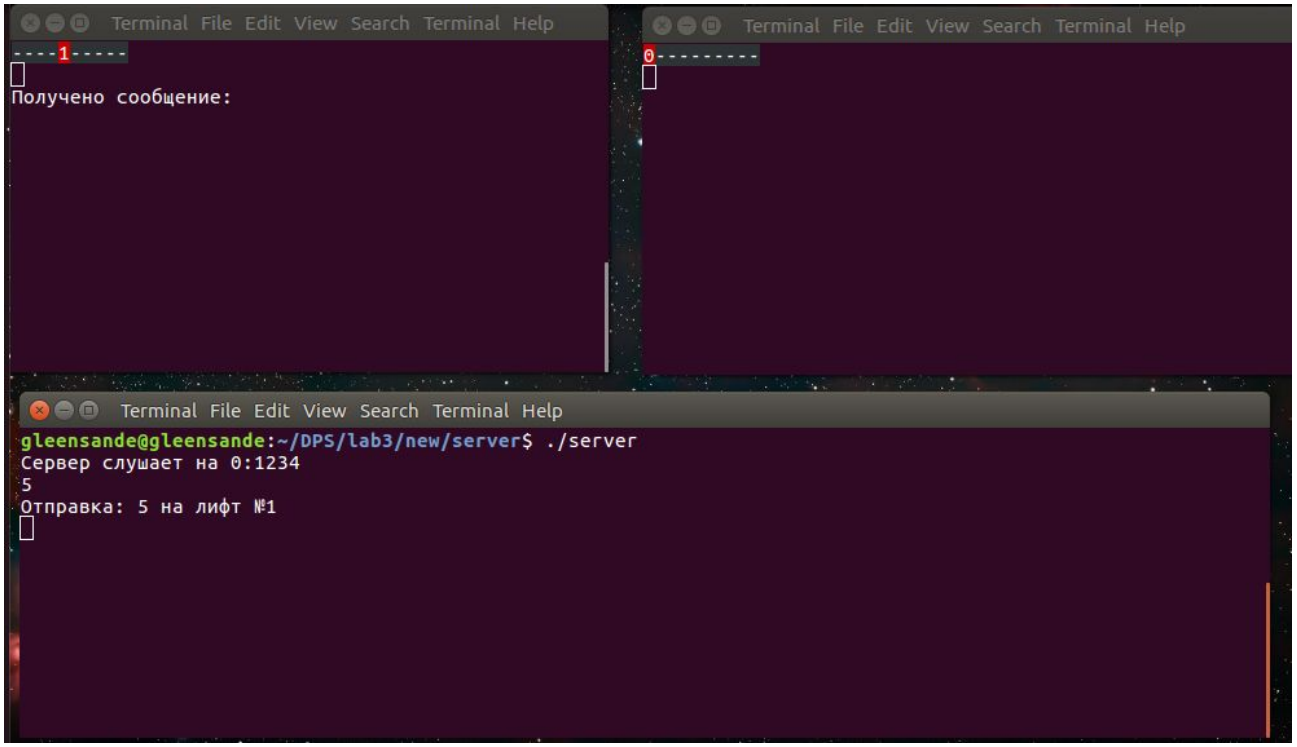

Рис. 10. Отправка лифта №1 (левая верхняя консоль) на этаж пять

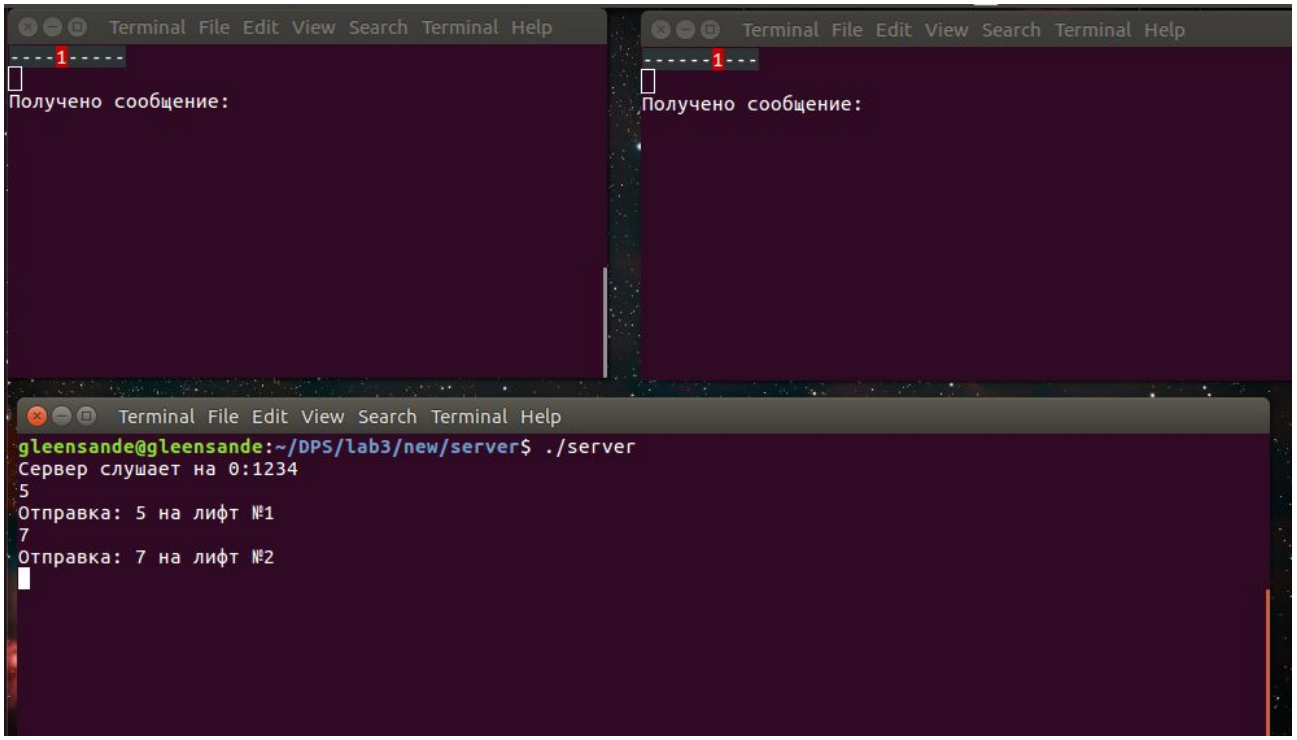

Рис. 11. Отправка лифта №2 (правая верхняя консоль) на этаж семь

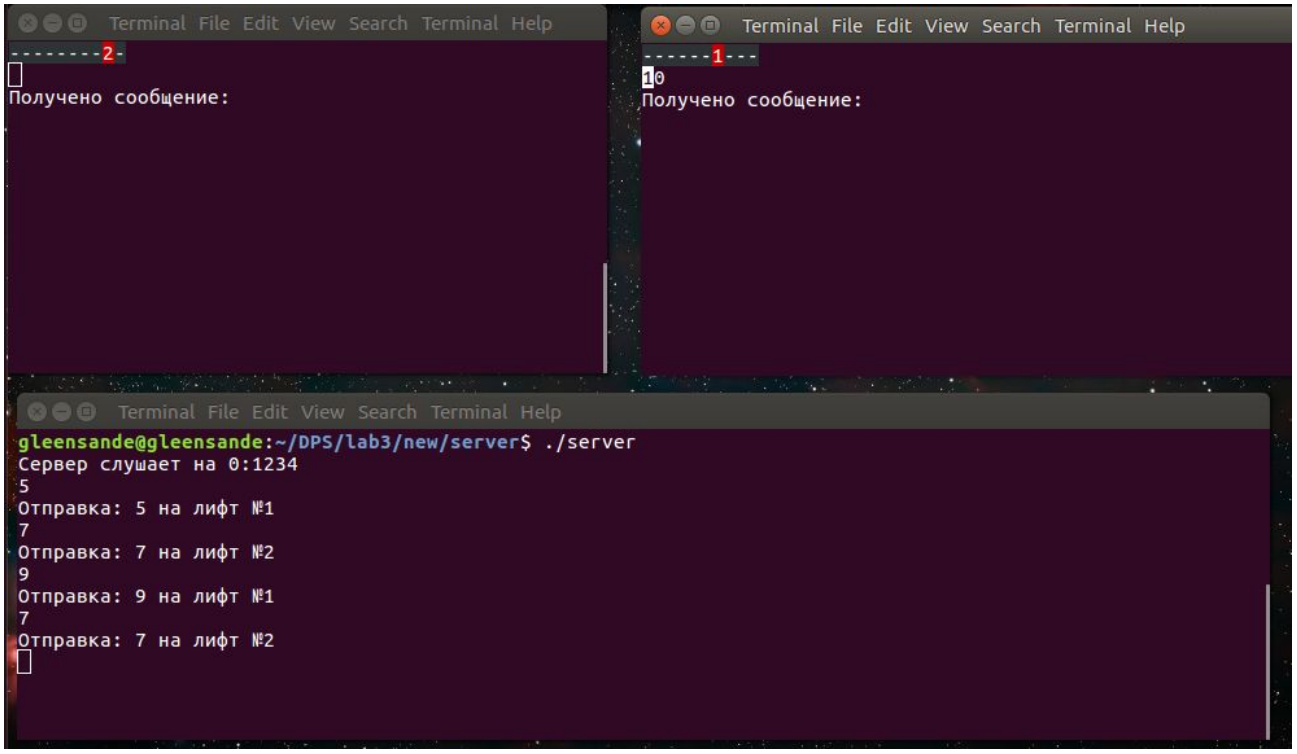

Рис. 12. Вызов лифта №2 (правая верхняя консоль) изнутри лифта на этаж 10, один пассажир

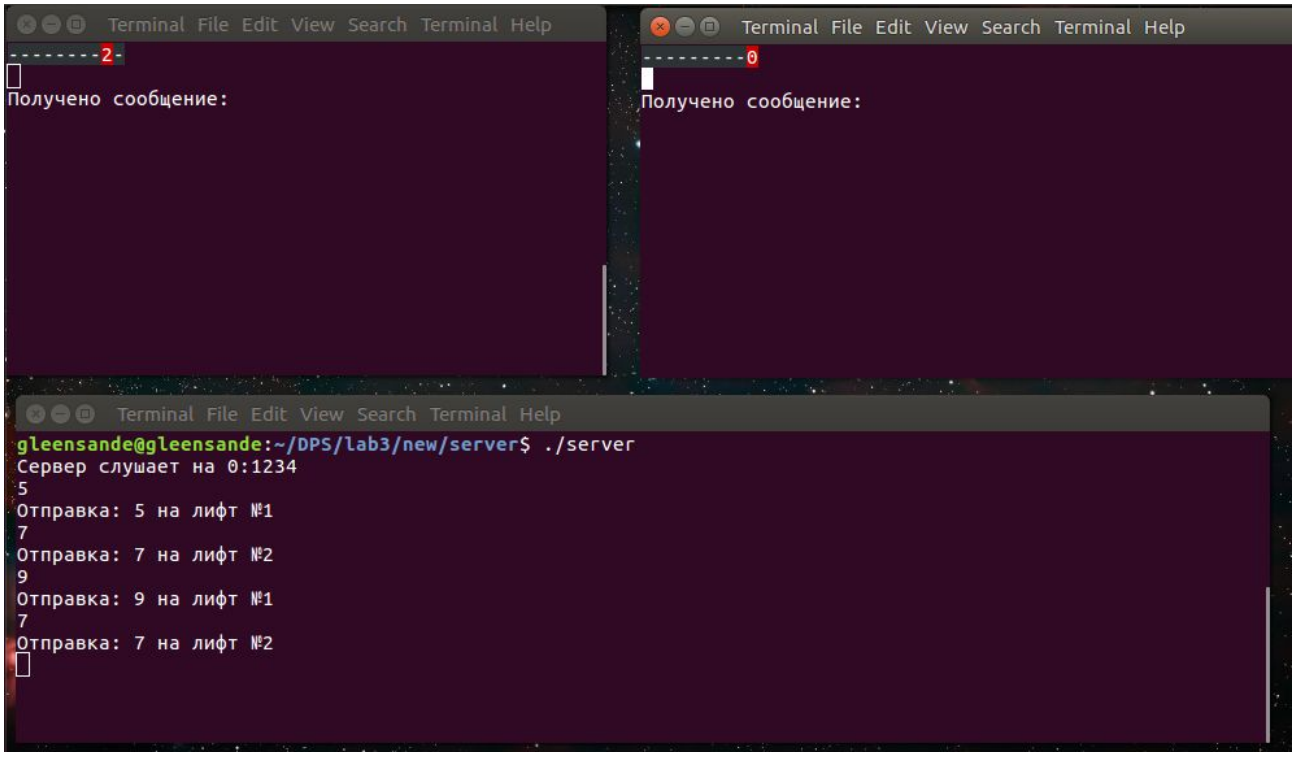

Рис. 13. Лифт №2 (правая верхняя консоль) прибывает на 10 этаж и высаживает пассажира

Текст программы

Листинг 4. Заголовочный файл client.h

```
#define S_HOST "localhost"      // хост удаленного сервера
#define S_PORT 1234             // порт удаленного сервера

#define BOTH 2 // аргумент shutdown для закрытия на чтение и запись

#define BUF_SIZE 64          // буфер для приема сообщения с сервера

int s;                          // сокет программы

int C_PORT;                     // порт, где будет работать клиент

// создание сокета и его именование
int create_socket(struct sockaddr_in* cname);

// инициализация структуры sockaddr_in сервера
void set_sockaddr_of_srv(struct sockaddr_in* sname);

// завершение работы клиента
void stop_client(int n);

#endif  // LIFT_CLIENT_H
```

Листинг 5. Файл client.c

```
#include "client.h"

// создание сокета и его именование
int create_socket(struct sockaddr_in* cname) {
    // открытие сокета
    s = socket(AF_INET, SOCK_STREAM, 0);
    if (s == -1) {
        printf("Не удалось открыть сокет\n");
        return -1;
    }

    // создание и заполнение структуры sockaddr
    memset((char *)cname, '\0', sizeof(*cname));
    cname->sin_family = AF_INET;
    if (C_PORT == 0) {
        C_PORT = 1235;
        cname->sin_port = C_PORT;
    }

    // установление адреса клиента
    cname->sin_addr.s_addr = INADDR_ANY;

    // для переиспользования адреса
    if (setsockopt(
              s,
              SOL_SOCKET,
              SO_REUSEADDR,
              &(int){ 1 },
              sizeof(int)
    ) < 0) {
        printf("Setsockopt(SO_REUSEADDR) не может сработать\n");
    }

    // именование сокета с помощью структуры sockaddr
    int binded = bind(s, (struct sockaddr *)cname, sizeof(*cname));
    if (binded == -1) {
        printf("Не удалось именовать сокет\n");
        printf("err: %s\n", strerror(errno));
        return -1;
    }

    return s;
}

// инициализация структуры sockaddr_in сервера
void set_sockaddr_of_srv(struct sockaddr_in* sname) {
    memset((char *)sname, '\0', sizeof(sname));

    sname->sin_family = AF_INET;
    sname->sin_port = S_PORT;

    struct hostent* hp = gethostbyname(S_HOST);
    memcpy((void*)&(sname->sin_addr), hp->h_addr, hp->h_length);
}

// завершение работы клиента
```

```
void stop_client(int n) {
   printf("\033[%d;%dH", 5, 1);
   printf("Завершение работы клиента\n");
   if (s != 0) {
       shutdown(s, BOTH);
   }
   remove(C_NAME);
   exit(0);
}
```

Листинг 6. Заголовочный файл terminal.h

```
#include <stdio.h>

#ifndef LIFT_TERMINAL_H
#define LIFT_TERMINAL_H

#define ESC "\033"
#define abs(N) (((N) < 0) ? (-(N)) : ((N)))
#define clrscr() printf(ESC "[2J") //clear screen, go (1,1)
#define gotoxy(x, y) printf(ESC "[%d;%dH", y, x);
#define set_display_atrib(color) printf(ESC "[%dm", color)
#define resetcolor() printf(ESC "[0m")

#define B_BLACK 40
#define B_RED 41

#endif  // LIFT_TERMINAL_H
```

Листинг 7. Файл elevator.c

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <stdlib.h>

#include "client.h"
#include "terminal.h"

#define TRUE 1
#define FALSE 0

#define INSIDE 0
#define OUTSIDE 1

int server_go;
// структура состояния и конфигурации лифта
typedef struct {
   double v_max;          // скорость лифта, когда он едет в секундах
    const int floors;        // количество этажей, на которое расчитан
лифт
   const int persons_max; // максимально допустимое количество человек
   int floor;             // текущий этаж пребывания лифта
   char moving;           // двигается ли лифт
   int person_num;        // количество человек внутри
} Elevator;

// лифт и его начальное состояние
Elevator lift = {10.0, 10, 5, 1, FALSE, 0};

// добавляет person_num человек в лифт, если это возможно
char add_persons(int person_num) {
   // если суммарное количество больше допустимого, ошибка
   if ((lift.person_num + person_num) > lift.persons_max)
   {
       printf("Too many persons, sorry\n");
       return FALSE;
   }

   lift.person_num += person_num;
   return TRUE;
}

// удаляет person_num человек из лифта, если это возможно
char remove_persons(int person_num) {
   // если попытка удалить больше, чем есть, ошибка
   if (person_num > lift.person_num)
   {
       printf("Error in num of pearsons to leave, sorry\n");
       return FALSE;
   }

   lift.person_num -= person_num;
   return TRUE;
}

// печатает текущее состояние лифта
```

```
void print_state() {
   gotoxy(1, 1);
   for (int i = 1; i <= lift.floors; i++)
   {
       if (i != lift.floor)
       {
           set_display_atrib(B_BLACK);
           printf("-");
       }
       else
       {
           set_display_atrib(B_RED);
           printf("%d", lift.person_num);
       }
   }
   resetcolor();
   printf("\n");
}

// отправляет лифт на этаж destination
char go(int destination, char from) {
   if (destination < 1 || destination > lift.floors)
   {
       printf("Unreachable destination: %d\n", destination);
       return FALSE;
   }

   // сколько этажей нужно проехать
   int steps = abs(destination - lift.floor);

   // вниз или вверх двигаться
   int step = (destination > lift.floor) ? 1 : -1;

   // движение
   for (int i = 0; i < steps; i++)
   {
       lift.moving = TRUE;
       sleep(10 / lift.v_max);
       lift.floor += step;
   }

   // если лифт вызвали снаружи, то по прибытии 1 человек заходит
   // иначе - 1 выходит
    if (from == INSIDE) {
        remove_persons(1);
    } else if (from == OUTSIDE) {
        add_persons(1);
    }
   lift.moving = FALSE;

   return TRUE;
}

// функция потока вызовов лифта изнутри
void *calls(void *attr) {
   int num;
   while (1)
```

```
    {
        gotoxy(1, 2);
        printf("  \b\b");

        scanf("%d", &num);
        while(lift.moving) {}
        go(num, INSIDE);
    }
}

// функция потока отображения состояния
void *view(void *attr) {
    while (1)
    {
        print_state();
        sleep(1);
    }
}

// функция потока получения команды с сервера и ее выполнения
void *message(void *attr) {
    // на сигнал SIGINT (Ctrl+C)
    // - вызов функции завершения работы клиента
    signal(SIGINT, stop_client);

    // открытие сокета
    struct sockaddr_in cname;
    s = create_socket(&cname);
    if (s == -1) {
        printf("Неудача при открытии сокета\n");
        return (void*)(-1);
    }

    // инициализация структуры sockaddr_in сервера
    struct sockaddr_in sname;
    set_sockaddr_of_srv(&sname);

    // соединение
    int c = connect(s, (struct sockaddr *)&sname, sizeof(sname));
    if (c == -1) {
        printf("Не удается получить доступ к серверу\n");
        return (void*)(-1);
    }

    // цикл бесконечного считывания команд сервера и их выполнения
    while (1) {
        // получение сообщения с сервера
        char buf[BUF_SIZE];
        int get_len = recv(s, buf, BUF_SIZE, 0);
        if (get_len == -1) {
            printf("Сообщение с сервера не может быть получено\n");
            return (void*)(-1);
        }

        // печать полученного сообщения и отправка лифта
        // в пункт назначения, как только он освободится
        if (get_len != 0) {
```

```
            gotoxy(1,3);
            printf("Получено сообщение:\n");
            gotoxy(1, 4);
            printf("  \b\b");

            write (1, buf, get_len);
            while(lift.moving) {}
            go(atoi(buf), OUTSIDE);
            printf("\n");
        }
    }
}

// создание потоков и управление ими
int main(int argc, char* argv[]) {
    // очистка экрана
    clrscr();

    // установление порта по первому аргументу,
    // скорости в соответствии со вторым аргументом программы,
    // если они были поданы, иначе по-умолчанию остается v=1, port=1235
    if (argc == 3) {
        C_PORT = atoi(argv[1]);
        lift.v_max = atof(argv[2]);
    }

    // задание атрибутов потоков
    pthread_attr_t pattr;
    pthread_attr_init(&pattr);
    pthread_attr_setscope(&pattr, PTHREAD_SCOPE_SYSTEM);
    pthread_attr_setdetachstate(&pattr, PTHREAD_CREATE_JOINABLE);

    // создание и запуск потоков
    pthread_t calls_pid, view_pid, message_pid;
    pthread_create(&calls_pid, &pattr, calls, NULL);
    pthread_create(&view_pid, &pattr, view, NULL);
    pthread_create(&message_pid, &pattr, message, NULL);

    // ожидание завершения дочерних потоков
    int retval1 = 0, retval2 = 0;
    pthread_join(calls_pid, (void *)&retval1);
    pthread_join(view_pid, (void *)&retval2);
}
```

Листинг 8. Заголовочный файл server.h

```
// для печати
#include <stdio.h>
#include <unistd.h>

// для tcp и sockaddr_in
#include <netinet/in.h>

// для memset
#include <memory.h>

// сокеты
#include <sys/types.h>
#include <sys/socket.h>

// сигналы + exit
#include <signal.h>
#include <stdlib.h>

#ifndef LIFT_SERVER_H
#define LIFT_SERVER_H

#define S_NAME "/tmp/srv.sock"  // название сокета в файловой системе
#define S_PORT 1234             // порт, где будет работать сервер

#define BOTH 2                  // аргумент shutdown для закрытия
                                // на чтение и на запись

// сокеты программы
int s, new_s1, new_s2;

// структура имени сокета
struct sockaddr_in sname;

// создание сокета и его именование
int create_socket(struct sockaddr_in* sname);

// завершение работы сервера
void stop_server(int n);

#endif  // LIFT_SERVER_H
```

Листинг 9. Файл server.c

```
#include <pthread.h>
#include "server.h"

char floor_num[3];
char lift_num = 0;

// создание сокета и его именование
int create_socket(struct sockaddr_in* sname) {
    // открытие сокета
    s = socket(AF_INET, SOCK_STREAM, 0);
    if (s == -1) {
        printf("Не удалось открыть сокет\n");
        return -1;
    }

    // создание и заполнение структуры sockaddr
    memset ((char *)sname, '\0', sizeof(sname));
    sname->sin_family = AF_INET;
    sname->sin_port = S_PORT;

    // для переиспользования адреса
     if (setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &(int){ 1 }, sizeof(int)) <
0) {
        printf("Setsockopt(SO_REUSEADDR) не может сработать\n");
    }

    // установление адреса сервера
    sname->sin_addr.s_addr = INADDR_ANY;

    // именование сокета с помощью структуры sockaddr
    int binded = bind(s, (struct sockaddr *)sname, sizeof(*sname));
    if (binded == -1) {
        printf("Не удалось именовать сокет\n");
        return -1;
    }

    return s;
}

// завершение работы сервера
void stop_server(int n) {
    printf("Завершение работы сервера\n");
    if (s != 0) {
        shutdown(s, BOTH);
    }
    if (new_s1!= 0) {
        shutdown(new_s1, BOTH);
    }
    if (new_s2!= 0) {
        shutdown(new_s2, BOTH);
    }
    remove(S_NAME);
    exit(0);
}

void* lift1(void* attr) {
```

```
    // принятие соединения от клиента
    int addrlen = 0;
    new_s1 = accept(s, (struct sockaddr *)&sname, &addrlen);
    if (new_s1 == -1) {
        printf("Неудача при попытке принять соединение от клиента\n");
        return (void*)(-1);
    }
     while(1) {
         if (lift_num == 1) {
             lift_num = 3;
            write(new_s1, floor_num, sizeof(floor_num));
         }
     }
}

void* lift2(void* attr) {
    // принятие соединения от клиента
    int addrlen = 0;
    new_s2 = accept(s, (struct sockaddr *)&sname, &addrlen);
    if (new_s2 == -1) {
        printf("Неудача при попытке принять соединение от клиента\n");
        return (void*)(-1);
    }
     while(1) {
         if (lift_num == 2) {
             lift_num = 0;
            write(new_s2, floor_num, sizeof(floor_num));
         }
     }
}

int main() {
    // на сигнал SIGINT (Ctrl+C) - вызов функции завершения работы сервера
    signal(SIGINT, stop_server);

    // открытие сокета
    s = create_socket(&sname);
    if (s == -1) {
        printf("Неудача при открытии сокета\n");
        return -1;
    }

    // прослушивание входящего потока данных
    int l = listen(s, 1);
    if (l == -1) {
        printf("Неудача при попытке слушать\n");
        return -1;
    }
    printf("Сервер слушает на %d:%d\n", sname.sin_addr.s_addr, S_PORT);

    // задание атрибутов потоков
    pthread_attr_t pattr;
    pthread_attr_init(&pattr);
    pthread_attr_setscope(&pattr, PTHREAD_SCOPE_SYSTEM);
    pthread_attr_setdetachstate(&pattr, PTHREAD_CREATE_JOINABLE);

    // создание и запуск потоков
```

```
    pthread_t lift1_pid, lift2_pid;
    pthread_create(&lift1_pid, &pattr, lift1, NULL);
    pthread_create(&lift2_pid, &pattr, lift2, NULL);

    // прослушивание вызовов снаружи лифтов
    int num;
    while (1)
    {
        printf("  \b\b");

        scanf("%d", &num);
        snprintf(floor_num, 3, "%d%c", num, '\0');
         if (lift_num == 0) {
             lift_num = 1;
         } else {
             lift_num = 2;
         }
         printf("Отправка: %s на лифт №%d\n", floor_num, lift_num);
    }

    // ожидание завершения дочерних потоков
    int retval1 = 0, retval2 = 0;
    pthread_join(lift1_pid, (void *)&retval1);
    pthread_join(lift2_pid, (void *)&retval2);
}
```