**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №5**

**по дисциплине «Операционные системы»**

Тема: Межпроцессорное взаимодействие

|  |  |  |
| --- | --- | --- |
| Студент гр. 1303 |  | Беззубов Д.В. |
| Преподаватель |  | Душутина Е.В. |

Санкт-Петербург

2023

## **Цель работы.**

* 1. Выполнение работы

1. Рассмотрим работу ненадёжных сигналов. Составим программу, которая обрабатывает пользовательские сигналы SIGUSR1 и SIGUSR2, реагирует по умолчанию на SIGINT и игнорирует SIGCHLD. Породим процесс-копию и уйдем в ожидание сигналов. Внутри программы с помощью kill отправлен сигнал SIGUSR1. Отправим 2 сигнала SIGUSR2. Первый сигнал обработается нашим обработчиком, а второй, после восстановления обработчика, сработает по умолчанию Содержание программы в листинге 1. Пример работы на рисунке 1.

Листинг 1. Код программы sigExam.c

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

#include <stdlib.h>

static void **sigHandler**(int sig) {

**printf**("Catched signal %s\n",sig == SIGUSR1 ? "SIGUSR1": "SIGUSR2");

**printf**("Parent = %d\n",**getppid**());

*// востанавливаем старую диспозицию*

**signal**(sig,SIG\_DFL);

}

int **main**() {

**printf**("\nFather started: pid = %i,ppid = %i\n",**getpid**(),**getppid**());

**signal**(SIGUSR1,sigHandler);

**signal**(SIGUSR2,sigHandler);

**signal**(SIGINT,SIG\_DFL);

**signal**(SIGCHLD,SIG\_IGN);

    int forkRes = **fork**();

    if(forkRes == 0) {

*// программа-потомок*

**printf**("\nSon started: pid = %i,ppid = %i\n",**getpid**(),**getppid**());

*// отправляем сигналы родителю*

        if(**kill**(**getppid**(),SIGUSR1) != 0) {

**printf**("Error while sending SIGUSR1\n");

**exit**(1);

        }

**printf**("Successfully sent SIGUSR1\n");

        return 0;

    }

*//  // программа-родитель*

**wait**(NULL);

*// ждем сигналов*

    for(;;){

**pause**();

    }

    return 0;

}

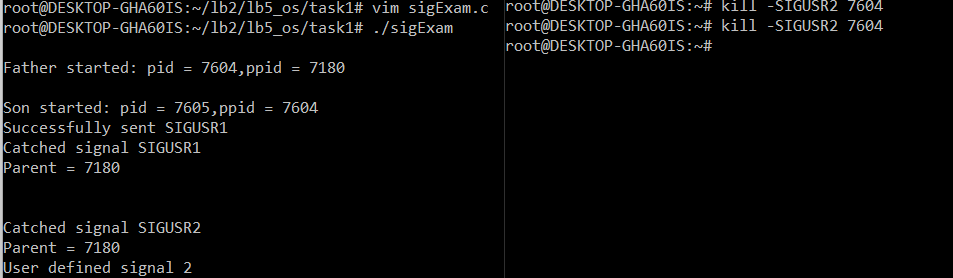


Рисунок 1 – результат работы программы sigExam

Повторим эксперимент для другого сигнала. Возьмём сигнал под номером 35, создадим вторую немного модифицированную программу. Вместо SIGUSR1 перехватываем и обрабатываем сигнал под номером 35 (по умолчанию он завершает программу), при этом сигнал SIGUSR1 будем игнорировать. Из теоретических данных знаем, что процессы изолированы друг от друга, в том числе у них свои диспозиции сигналов. Проверим это: запустим программу, представленную в листинге 1, и модифицированную. И там и там отправим процессу сигнал под номером 45. Результат на рисунке 2.

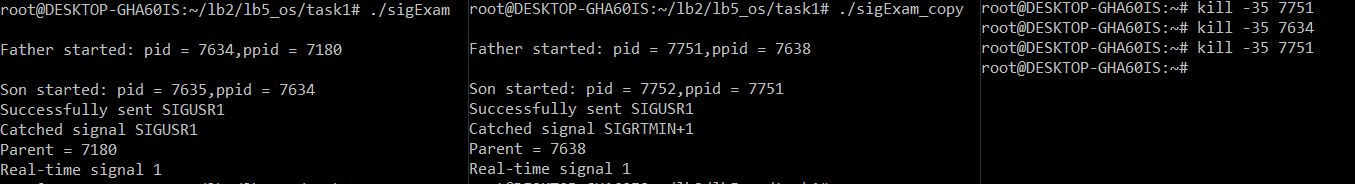


Рисунок 2 – результат отправки сигнала 35 процессам стандартной и модифицированной программ

Как видно из рисунка, программа 1 завершилась после отправки сигнала под номером 35, вторая же его перехватила и обработала, кроме того, на сигнал программы-потомка отреагировала только первая программа, вторая его проигнорировала.

Рассмотрим ещё один эксперимент: напишем программу 3 и программу 4 таким образом: в программе 3 узнаем pid процесса, запишем его в файл и перейдём в ожидание сигналов (при этом написан обработчик для SIGUSR2). Запустим её, тем временем в программе 4 откроется файл, куда раннее записали pid процесса программы 3, соберётся строка и произведётся системный вызов, в результате которого программе 3 пошлётся сигнал SIGUSR2. Содержание программы 3 в листинге 2, программы 4 – в листинге 3. Результат работы на рисунке 3.

Листинг 2. Программа, записывающая свой pid в файл

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

#include <stdlib.h>

static void **sigHandler**(int sig) {

**printf**("Catched signal %s\n",sig == SIGUSR1 ? "SIGUSR1": "SIGUSR2");

**printf**("Parent = %d\n",**getppid**());

*// востанавливаем старую диспозицию*

**signal**(sig,SIG\_DFL);

}

int **main**() {

**signal**(SIGUSR2, sigHandler);

    FILE\* file;

    file = **fopen**("pid\_file.txt", "w");

    char str[100];

    if (!file){

**perror**("Can not open file");

**exit**(1);

    }

    int father\_pid = **getpid**();

**sprintf**(str, "%d", father\_pid);

**printf**("\nFather started with pid = %d\n", father\_pid);

**fprintf**(file, str);

**fclose**(file);

*//  // программа-родитель*

**wait**(NULL);

*// ждем сигналов*

    for(;;){

**pause**();

    }

    return 0;

}

Листинг 3. Программа, отправляющая сигнал процессу из файла

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

#include <stdlib.h>

#include <string.h>

char\* **concat**(const char \*s1, const char \*s2)

{

    char \*result = **malloc**(**strlen**(s1) + **strlen**(s2) + 1); *// +1 for the null-terminator*

**strcpy**(result, s1);

**strcat**(result, s2);

    return result;

}

int **main**() {

    FILE\* file;

    file = **fopen**("pid\_file.txt", "r");

    char str[100];

    if (!file){

**perror**("Can not open file");

**exit**(1);

    }

**fgets**(str, 100, file);

    char\* s = **concat**("kill -12 ", str);

**printf**("%s\n", s);

**system**(s);

**fclose**(file);

    return 0;

}

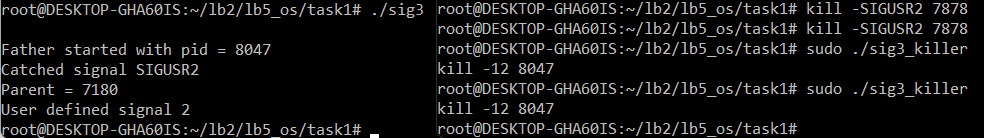


Рисунок 3 – результат отправки сигнала из неродственной программы

Таким образом, повторили эксперимент для процессов, порождаемых в разных файлах. Как и ожидалось, сигнал обрабатывается точно так же, как и в случае с родственными процессами (т. е. сначала сигнал обрабатывается нашим sigHandler, потом восстанавливается диспозиция и программа завершается при второй посылке сигнала).

Рассмотрим обработку сигналов потоком одного процесса. Для этого используем команду sigwait(sigset\_t, sig). Она прекращает исполнение потока, пока один из сигналов из списка sigset\_t не становится в ожидании (или не послан напрямую), и возвращает номер сигнала в sig. Обработаем SIGUSR1 и SIGUSR2 так: на посыл SIGUSR1 остановим поток, а SIGUSR2 проигнорируем. Содержание программы в листинге 4, результат работы на рисунке 4.

Листинг 4.

#include <stdio.h>

#include <stdlib.h>

#include <signal.h>

#include <pthread.h>

#include <unistd.h>

#define **\_GNU\_SOURCE**

pthread\_t thread1;

sigset\_t set;

int thread\_id;

void \***thread\_func**(void \*args) {

    thread\_id = **gettid**();

**printf**("Thread started. %d\n", thread\_id);

    while(1) {

        int signum;

**sigwait**(&set, &signum);

**printf**("Received signal %d\n", signum);

        if (signum == SIGUSR1){

            break;

        }

    }

**sleep**(2);

**printf**("ending thread!\n");

**pthread\_exit**(NULL);

}

void **sig\_handler**(int signum){}

int **main**() {

**signal**(SIGUSR1, sig\_handler);

**signal**(SIGUSR2, SIG\_IGN);

**sigemptyset**(&set);

**sigaddset**(&set, SIGTERM);

**sigaddset**(&set, SIGINT);

**sigaddset**(&set, SIGUSR1);

    if(**pthread\_create**(&thread1, NULL, thread\_func, NULL)) {

**printf**("Error creating thread\n");

        return 1;

    }

**printf**("Main thread started. PID: %d\n", **getpid**());

    int i = 0;

**pthread\_join**(thread1, NULL);

**printf**("main is ending\n");

    return 0;

}

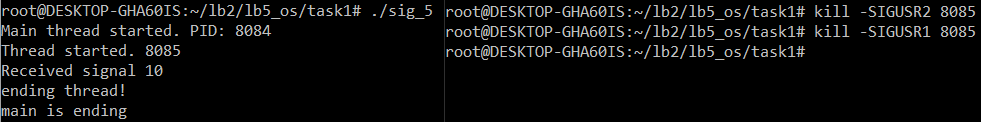


Рисунок 4 – результат обработки сигнала потоком программы

Программа работает, как и ожидалось: сигналы обрабатываются согласно определенными нами обработчиками, а на посыл SIGUSR1 (сигнал 10) поток завершает свою работу.

Так как потоки хранятся в разных адресных пространствах, нет возможности напрямую завершить выполнения потока из другой программы. Но это можно обойти посредством посылки сигнала из одной программы во вторую. Модифицируем программы так: программа из листинга 5 создаёт поток и ожидает его завершения, а программа 6 отправит потоку сигнал SIGUSR1 (в результате которого поток и после него процесс завершатся). Результат работы на рисунке 5.

Листинг 5. Программа, создающая поток и сохраняющая его id в файл

#include <stdio.h>

#include <stdlib.h>

#include <signal.h>

#include <pthread.h>

#include <unistd.h>

#define **\_GNU\_SOURCE**

pthread\_t thread1;

sigset\_t set;

int thread\_id;

void \***thread\_func**(void \*args) {

    thread\_id = **gettid**();

**printf**("Thread started. %d\n", thread\_id);

    while(1) {

        int signum;

**sigwait**(&set, &signum);

**printf**("Received signal %d\n", signum);

        if (signum == SIGUSR1){

            break;

        }

    }

**sleep**(2);

**printf**("ending thread!\n");

**pthread\_exit**(NULL);

}

void **sig\_handler**(int signum){}

int **main**() {

**signal**(SIGUSR1, sig\_handler);

**signal**(SIGUSR2, SIG\_IGN);

**sigemptyset**(&set);

**sigaddset**(&set, SIGTERM);

**sigaddset**(&set, SIGINT);

**sigaddset**(&set, SIGUSR1);

    if(**pthread\_create**(&thread1, NULL, thread\_func, NULL)) {

**printf**("Error creating thread\n");

        return 1;

    }

    FILE\* fp;

    fp = **fopen**("tid\_file.txt", "w");

    char str[100];

    if (fp == NULL){

**perror**("FILE");

**exit**(1);

    }

**sprintf**(str, "%d", thread\_id);

**fprintf**(fp, str);

**fclose**(fp);

**printf**("Waiting for signals to thread 1 from other source...\n");

**pthread\_join**(thread1, NULL);

**printf**("main is ending\n");

    return 0;

}

Листинг 6. Программа отправляющая сигнал потоку

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

#include <stdlib.h>

#include <string.h>

char\* **concat**(const char \*s1, const char \*s2){

    char\* result = **malloc**(**strlen**(s1) + **strlen**(s2) + 1);

**strcpy**(result, s1);

**strcat**(result, s2);

    return result;

}

int **main**()

{

    FILE\* fp;

    fp = **fopen**("tid\_file.txt", "r");

    char str[100];

    if (fp == NULL){

**perror**("FILE");

**exit**(1);

    }

**fgets**(str, 100, fp);

*//printf("%s\n", str);*

    char\* s = **concat**("kill -10 ", str); *//10 == SIGUSR1*

*// printf("string: [%s]\n", s);*

**printf**("Отправляем сигнал!\n");

**system**(s);

**fclose**(fp);

**free**(s);

    return 0;

}

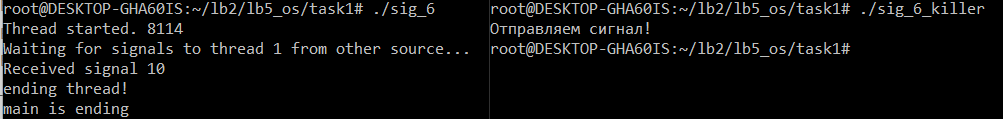


Рисунок 5 – результат завершения потока с помощью посторонней программы

Программа отработала, как и ожидалось: поток принял сигнал, обработал его (завершился) и после него завершился основной процесс.

1. Создадим программу, позволяющую продемонстрировать возможность отложенной обработки (временного блокирования) сигнала SIGINT. Для решения задачи используем структуру sigaction. Блокировку реализуем, вызвав «засыпание» процесса из обработчика пользовательских сигналов. С рабочего терминала отправим процессу sigact сигнал SIGUSR1, а затем SIGINT. Содержимое программы в листинге. Результат работы на рисунке 6.

Листинг 7

#include <stdio.h>

#include <signal.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <unistd.h>

#include <stdlib.h>

#include <fcntl.h>

void (\***mysig**(int sig,void (\*hnd)(int)))(int) {

*// надежная обработка сигналов*

    struct sigaction act,oldact;

    act.sa\_handler = hnd;

**sigemptyset**(&act.sa\_mask);

**sigaddset**(&act.sa\_mask,SIGINT);

    act.sa\_flags = 0;

    if(**sigaction**(sig,&act,0) < 0)

        return SIG\_ERR;

    return act.sa\_handler;

}

void **hndUSR1**(int sig) {

    if(sig != SIGUSR1) {

**printf**("Catched bad signal %d\n",sig);

        return;

    }

**printf**("SIGUSR1 catched\n");

**sleep**(60);

}

int **main**() {

**mysig**(SIGUSR1,hndUSR1);

    for(;;) {

**pause**();

    }

    return 0;

}

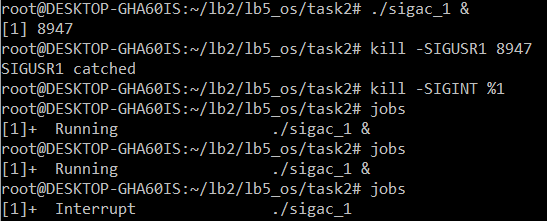


Рисунок 6 – результат отложенной обработки сигнала

В результате сигнал SIGUSR1 принят корректно. После получения SIGINT программа работала ещё минуту, и только после этого завершилась. В этом отличие надежной обработки сигналов от ненадежной: есть возможность отложить прием некоторых других сигналов. Отложенные таким образом сигналы записываются в маску PENDING и обрабатываются после завершения обработки сигналов, которые отложили обработку. Механизм ненадёжных сигналов не позволяет откладывать обработку других сигналов (можно лишь установить игнорирование некоторых сигналов на время обработки.

Изменим обработчик сигнала так, чтобы из него производилась отправка сигнала SIGINT с помощью функции kill(). Проанализируем наличие и очередность обработки сигналов. Содержимое программы в листинге 8, результат работы на рисунке 7.

Листинг 8

#include <stdio.h>

#include <signal.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <unistd.h>

#include <stdlib.h>

#include <fcntl.h>

void (\***mysig**(int sig,void (\*hnd)(int)))(int) {

*// надежная обработка сигналов*

    struct sigaction act,oldact;

    act.sa\_handler = hnd;

**sigemptyset**(&act.sa\_mask);

**sigaddset**(&act.sa\_mask,SIGINT);

    act.sa\_flags = 0;

    if(**sigaction**(sig,&act,0) < 0)

        return SIG\_ERR;

    return act.sa\_handler;

}

void **hndUSR1**(int sig) {

    if(sig != SIGUSR1) {

**printf**("Catched bad signal %d\n",sig);

        return;

    }

**printf**("SIGUSR1 catched. Send SIGINT\n");

**kill**(**getpid**(), SIGINT);

**sleep**(10);

}

int **main**() {

**mysig**(SIGUSR1,hndUSR1);

    for(;;) {

**pause**();

    }

    return 0;

}

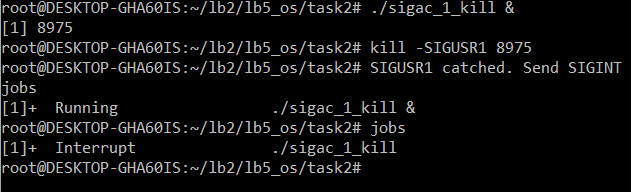


Рисунок 7 – результат работы программы из листинга 8

В данном случае нам не пришлось самостоятельно отправлять сигнал SIGINT, при этом обработка сигнала произошла аналогичным образом с предыдущей программой. При этом обработка полученного сигнала начинается только после завершения текущего обработчика.

1. Попробуем проверить возможность организации очереди для различных типов сигналов (обычных и реального времени). При этом увеличим вложенность обработчиков. Из стандартных будем использовать SIGUSR1 и SIGUSR2, и первые три сигнала реального времени. Программа устроена так: в головной функции в цикле 10 раз процессу отправляется сигнал SIGRTMIN и SIGUSR1 по очереди. В обработчике сигналов SIGRTMIN отправляет сигнал SIGUSR2, который, в свою очередь, отправляет сигнал SIGRTMIN+2. SIGUSR1 же в обработчике вызывает сигнал SIGRTMIN+1. Содержание программы в листинге 9, результат работы на рисунке 8.

Листинг 9. Программа для организации очереди сигналов

 #include <stdio.h>

 #include <stdlib.h>

 #include <signal.h>

 #include <unistd.h>

 static int reg\_flag = 0;

 static int rt\_flag = 0;

 static int signal\_counter = 0;

  const char\* **switch\_int\_to\_sigstr**(int sig){

  if (sig == SIGRTMIN){

  return "SIGRTMIN";

  }

  else if (sig == SIGRTMIN+1){

  return "SIGRTMIN+1";

  }

  else if (sig == SIGRTMIN+2){

  return "SIGRTMIN+2";

  }

  else if (sig == SIGUSR1){

  return "SIGUSR1";

  }

  else if (sig == SIGUSR2){

  return "SIGUSR2";

  }

  return "UNKNOWN";

  }

  void **handler**(const int sig, siginfo\_t\* si, void\* ucontext) {

**printf**("Received signal %s with value %d.\n", **switch\_int\_to\_sigstr**(sig), si->si\_value.sival\_int);

  int pid = **getpid**();

  if (sig == SIGRTMIN){

**sigqueue**(pid, SIGUSR2, (union sigval){.sival\_int = signal\_counter++});

  }

  else if (sig == SIGUSR1){

**sigqueue**(pid, SIGRTMIN+1, (union sigval){.sival\_int = signal\_counter++});

  }

  else if (sig == SIGUSR2){

**sigqueue**(pid, SIGRTMIN+2, (union sigval){.sival\_int = signal\_counter++});

  }

  }

  void (\***mysig**(int sig, void (\*hnd)(int, siginfo\_t\*, void\*)))(int, siginfo\_t\*, void\*) {

*// надежная обработка сигналов*

  struct sigaction act, oldact;

  act.sa\_sigaction = hnd;

**sigemptyset**(&act.sa\_mask);

**sigaddset**(&act.sa\_mask, SIGINT);

  act.sa\_flags = SA\_SIGINFO;

  if (**sigaction**(sig, &act, &oldact) < 0) {

**perror**("sigaction");

**exit**(-1);

  }

  return oldact.sa\_sigaction;

  }

  int **main**(void)

  {

**mysig**(SIGUSR1, handler);

**mysig**(SIGUSR2, handler);

**mysig**(SIGRTMIN, handler);

**mysig**(SIGRTMIN+1, handler);

**mysig**(SIGRTMIN+2, handler);

  int pid = **getpid**();

  int signal;

  for (int i = 0; i< 10;i++){

  signal = (i%2 == 0) ? SIGRTMIN : SIGUSR1;

**sigqueue**(pid, signal, (union sigval){.sival\_int = signal\_counter++});

  }

**sleep**(5); *// Даем время обработчикам сигналов выполнить свою работу*

**printf**("Завершение программы.\n");

  return EXIT\_SUCCESS;

  }

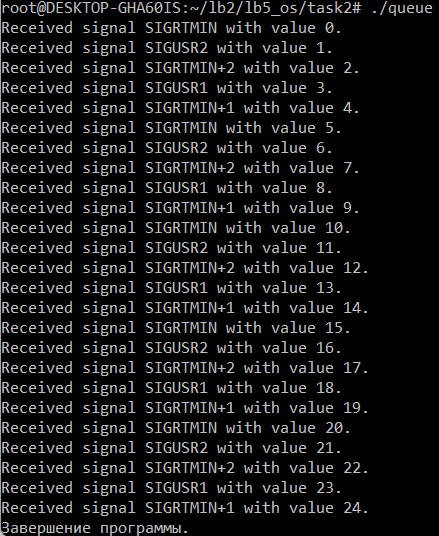


Рисунок 9 – результат организации очереди сигналов

Как видно из рисунка 8, очередь цепочка вложенных вызовов разных типов сигналов сохраняется, следовательно такую очередь организовать можно.

Экспериментально подтвердим, что обработка равноприоритетных сигналов реального времени происходит в порядке FIFO. Для этого напишем программу, в которой обработчик сигналов будет выводить номер сигнала, его порядковый номер (будем использовать SIGRTMIN). В головной процедуре программа ожидает сигналов извне. Для посылки сигналов напишем скрипт, который 10 раз отправит сигнал SIGRTMIN. Содержимое программы в листинге 11, результат работы на рисунке 10.

Листинг 11

#include <signal.h>

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

int received\_signals[10];

int received\_signals\_value[10];

int received\_signals\_count = 0;

void **real\_time\_handler**(int sig\_number, siginfo\_t \* info,

                       void \* arg **\_\_attribute\_\_** ((unused)))

{

    received\_signals[received\_signals\_count] = sig\_number - SIGRTMIN;

    received\_signals\_value[received\_signals\_count] = info->si\_value.sival\_int;

    ++received\_signals\_count;

}

void **send\_real\_time\_signal**(int sig\_number, int value)

{

    union sigval sig\_value;

**printf**("Sending signal SIRTMIN+%d, value %d\n", sig\_number, value);

    sig\_value.sival\_int = value;

    if (**sigqueue**(**getpid**(), sig\_number + SIGRTMIN, sig\_value) < 0) {

**perror**("sigqueue");

**exit**(EXIT\_FAILURE);

    }

}

int **main**()

{

    struct sigaction action;

    sigset\_t set;

    int i;

*// Handler setup*

    action.sa\_sigaction = real\_time\_handler;

**sigemptyset**(&action.sa\_mask);

    action.sa\_flags = SA\_SIGINFO;

**sigaddset**(&action.sa\_mask, SIGRTMIN + 1);

**sigaddset**(&action.sa\_mask, SIGRTMIN + 2);

**sigaddset**(&action.sa\_mask, SIGRTMIN + 3);

*// Block all signals*

**sigprocmask**(SIG\_BLOCK, &action.sa\_mask, NULL);

    if ((**sigaction**(SIGRTMIN + 1, &action, NULL) < 0)

     || (**sigaction**(SIGRTMIN + 2, &action, NULL) < 0)

     || (**sigaction**(SIGRTMIN + 3, &action, NULL) < 0)) {

**perror**("sigaction");

**exit**(EXIT\_FAILURE);

    }

**send\_real\_time\_signal**(1, 0);

**send\_real\_time\_signal**(2, 1);

**send\_real\_time\_signal**(3, 2);

**send\_real\_time\_signal**(1, 3);

**send\_real\_time\_signal**(2, 4);

**send\_real\_time\_signal**(3, 5);

**send\_real\_time\_signal**(3, 6);

**send\_real\_time\_signal**(2, 7);

**send\_real\_time\_signal**(1, 8);

**send\_real\_time\_signal**(3, 9);

*// Unblock all signals*

*// To make sure we're handling all signals before resuming*

**sleep**(1);

**sigfillset**(&set);

**sigprocmask**(SIG\_UNBLOCK, &set, NULL);

*// Display results*

    for (i = 0; i < received\_signals\_count; ++i) {

**printf**("Received signal SIGRTMIN+%d, value %d\n",

               received\_signals[i], received\_signals\_value[i]);

    }

    return EXIT\_SUCCESS;

}

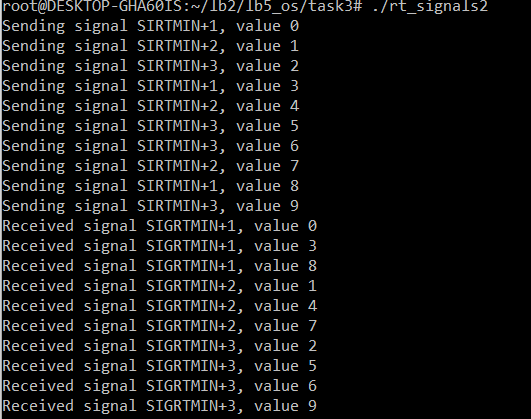


Рисунок 10 – Приоритетность сигналов реального времени

Т.к. одинаковые сигналы реального времени обрабатываются в порядке FIFO, будем посылать разные RT сигналы. Ожидаем, что обработка сигналов произойдет от меньших номеров к большим. Как видно из рисунка, результат совпал с теоретическими предположениями.

4. Неименованные каналы

4.1. Организуем программу (файл pipe.c) так, чтобы процесс-родитель создавал неименованный канал, создавал потомка, закрывал канал на запись и записывал в произвольный текстовый файл считываемую из канала информацию. В функции процесса-потомка будет входить считывание данных из файла и запись их в канал. (Функционирование осуществляется через стандартные потоки ввода/вывода , как было показано выше).

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#define **DEF\_F\_R** "from.txt"

#define **DEF\_F\_W** "to.txt"

int **main**(int argc, char\*\* argv) {

    char fileToRead[32];

    char fileToWrite[32];

    if (argc < 3) {

**printf**("Using default fileNames '%s','%s'\n", DEF\_F\_R, DEF\_F\_W);

**strcpy**(fileToRead, DEF\_F\_R);

**strcpy**(fileToWrite, DEF\_F\_W);

    } else {

**strcpy**(fileToRead, argv[1]);

**strcpy**(fileToWrite, argv[2]);

    }

    int filedes[2];

    if (**pipe**(filedes) < 0) {

**printf**("Father: can't create pipe\n");

**exit**(1);

    }

**printf**("pipe is successfully created\n");

    if (**fork**() == 0) {

*// процесс сын*

*// закрывает пайп для чтения*

**close**(filedes[0]);

        FILE\* f = **fopen**(fileToRead, "r");

        if (!f) {

**printf**("Son: cant open file %s\n", fileToRead);

**exit**(1);

        }

        char buf[100];

        int res;

        while (!**feof**(f)) {

*// читаем данные из файла*

            res = **fread**(buf, sizeof(char), 100, f);

**write**(filedes[1], buf, res); *// пишем их в пайп*

        }

**fclose**(f);

**close**(filedes[1]);

        return 0;

    }

*// процесс отец*

*// закрывает пайп для записи*

**close**(filedes[1]);

    FILE\* f = **fopen**(fileToWrite, "w");

    if (!f) {

**printf**("Father: cant open file %s\n", fileToWrite);

**exit**(1);

    }

    char buf[100];

    int res;

    while (1) {

**bzero**(buf, 100);

        res = **read**(filedes[0], buf, 100);

        if (!res)

            break;

**printf**("Read from pipe: %s\n", buf);

**fwrite**(buf, sizeof(char), res, f);

    }

**fclose**(f);

**close**(filedes[0]);

    return 0;

}

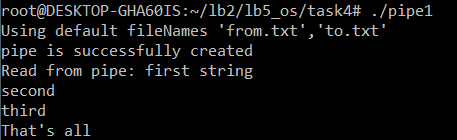


Рисунок 12 – результат запуска программы

Как видно из рисунка 12, содержимое файла from.txt переписалось в файл to.txt с использованием неименованного канала. Поскольку процесс-родитель только читает из канала, то дескриптор для записи filedes[1] он закрывает, аналогично процесс-сын закрывает дескриптор для чтения filedes[0].

Именованные каналы в Unix функционируют подобно неименованным — они позволяют передавать данные только в одну сторону. Однако в отличие от неименованных каналов каждому каналу FIFO сопоставляется полное имя в файловой системе, что позволяет двум неродственным процессам обратиться к одному и тому же FIFO. Эти каналы работают как очереди. После создания канал FIFO должен быть открыт на чтение или запись с помощью либо функции open, либо одной из стандартных функций открытия файлов из библиотеки вводавывода (например, fopen). FIFO может быть открыт либо только на чтение, либо только на запись. Нельзя открывать канал на чтение и запись одновременно, поскольку именованные каналы могут быть только односторонними.

Создадим клиент-серверное приложение, демонстрирующее дуплексную (двунаправленную) передачу информации двумя однонаправленными именованными каналами между клиентом и сервером. В файле server.c создадим 2 именованных канала с помощью mknod(), аргументы которого: имя файла FIFO в файловой системе; флаги владения, прав доступа (установим открытые для всех права доступа на чтение и на запись S\_IFIFO | 0666). Откроем один канал на запись (chan1), другой – на чтение (chan2) и запустим серверную часть программы. В серверной части программы: запишем имя файла в канал 1 (для записи) функцией write(); прочитаем данные из канала 2 и выведем на экран. В файле client.c запрограммируем функции: открытия каналов для чтения (chan1) и записи (chan2). Из первого канала читается имя файла, во второй канал пишется его содержимое. Результат работы на рисунке 13.

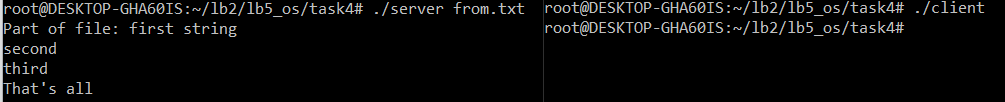


Рисунок 13 – результат запуска сервера и клиента

Клиент и сервер отработали как и ожидалось – сервер открыл каналы и записал туда первую строчку из открываемого текстового файла, затем клиент ее считал и записал во второй канал. Откуда сервер ее взял и вывел.

Напишем скрипт, который запустит несколько серверов и каналов.

*#!/bin/bash*

**gcc** server.c -o server

**gcc** client.c -o client

for i in {**1..3}**

do

**gnome-terminal** -- bash -c "./server testFile$i.txt channel${i}1 channel${i}2; exec bash" &

**sleep** 2

**gnome-terminal** -- bash -c "./client channel${i}1 channel${i}2; exec bash" &

**sleep** 1

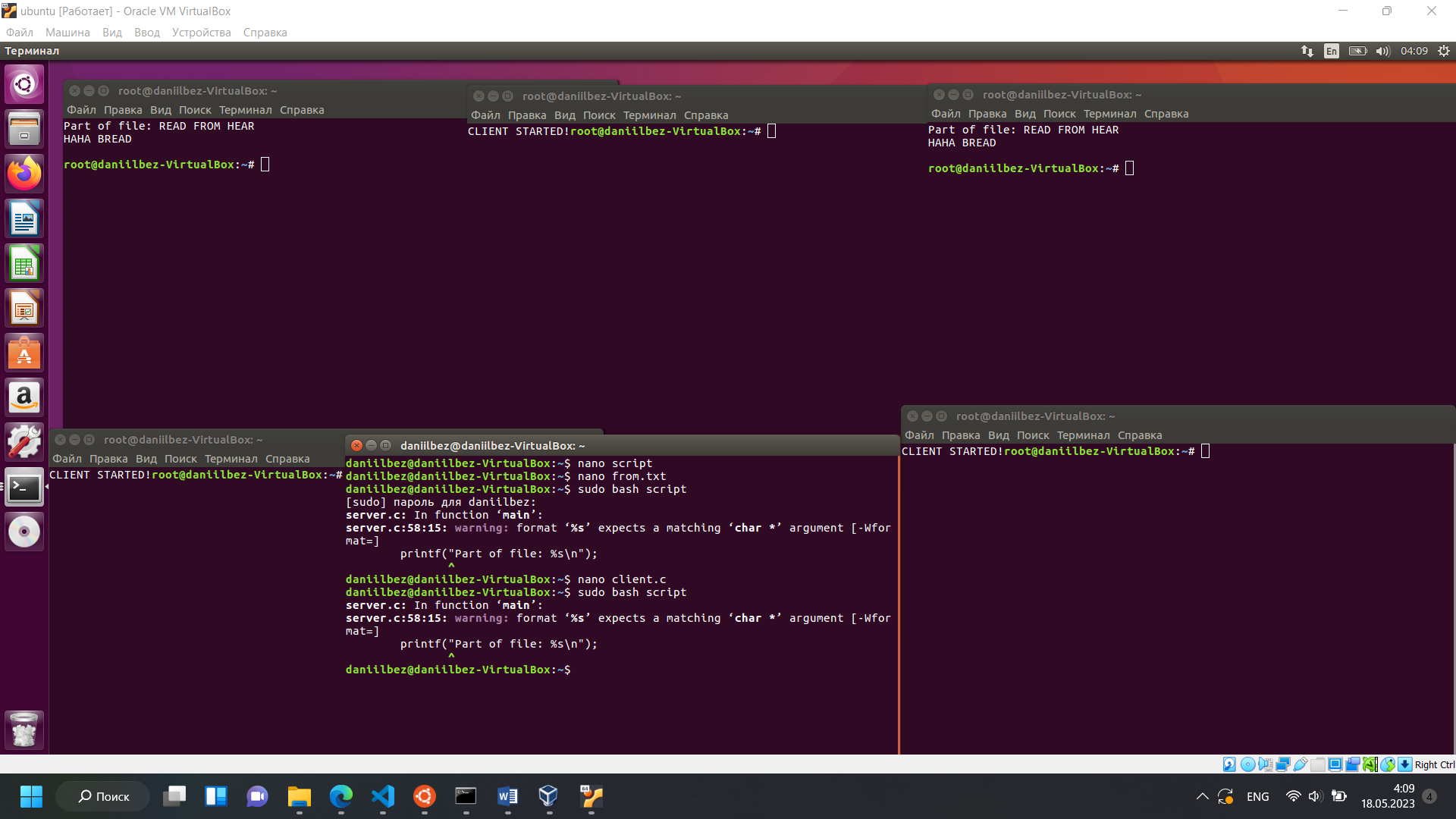


Рисунок 13 – Результат запуска нескольких серверов и клиентов

Несмотря на то, что именованные каналы являются отдельным типом файлов и могут быть видимы разными процессами даже в распределенной файловой системе, использование FIFO для взаимодействия удаленных процессов и обмена информацией между ними невозможно. Так как и в этом случае для передачи данных задействовано ядро. Создаваемый файл служит для получения данных о расположении FIFO в адресном пространстве ядра и его состоянии. Продемонстрируем это на примере. Изменим ранее использованную программу так, чтобы сервер, перед тем как читать данные из канала, ожидал ввода пользователя. Исходный код клиента оставим неизменным.

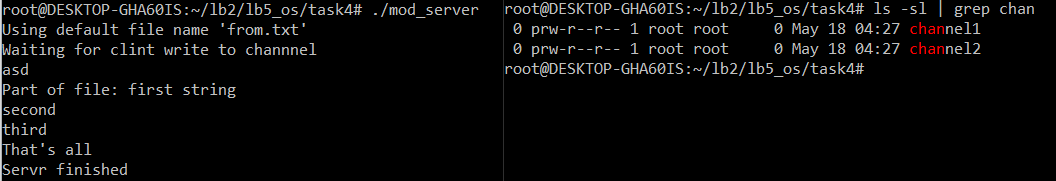


Рисунок 14 – иллюстрация создания файлов-каналов

Размер файла канала не изменяется, несмотря на записанные данные, это свидетельствует о том, что файл используется не как хранилище пересылаемых данных, а только для получения информации системой о них. Сами данные проходят через ядро ОС.

На неименованные каналы и каналы FIFO системой накладываются всего два ограничения: OPEN\_MAX — максимальное количество дескрипторов, которые могут быть одновременно открыты некоторым процессом (POSIX устанавливает для этой величины ограничение снизу); PIPE\_BUF — максимальное количество данных, для которого гарантируется атомарность операции записи (POSIX требует по менее 512 байт). Значение OPEN\_MAX можно узнать, вызвав функцию sysconf, его можно изменить из интерпретатора команд или из процесса. Значение PIPE\_BUF обычно определено в заголовочном файле. Для FIFO с точки зрения стандарта POSIX оно представляет собой переменную (ее значение можно получить в момент выполнения программы), зависимую от полного имени файла, поскольку разные имена могут относиться к разным файловым системам, и эти файловые системы могут иметь различные характеристики.

Для того, чтобы узнать максимальный размер пайпа, можно воспользоваться утилитой ulimit с флагом -а

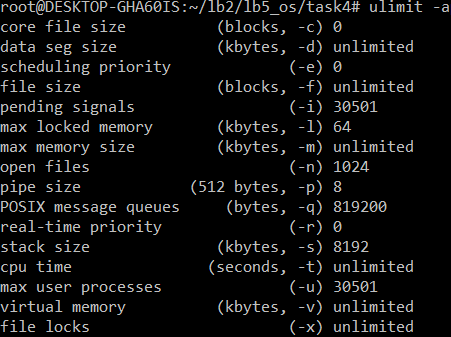


Рисунок 15 – Лимиты, установленные в системе

1. Очереди сообщений

Создадим клиент-серверное приложение, демонстрирующее передачу информации между процессами посредством очередей сообщений. Аналогично предыдущему разделу программа включает 2 файла: серверный и клиентский. В общем случае одновременно могут работать несколько клиентов. Серверный файл содержит: - подключение библиотек (см. листинг ниже) - обработчик сигнала SIGINT (с восстановлением диспозиции и удалением очереди сообщений системным вызовом msgctl() для корректного завершения сервера при получении сигнала SIGINT); - основную программу со следующей структурой:

Листинг клиента

#include <stdio.h>

#include <stdlib.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <sys/types.h>

#include <signal.h>

#include <string.h>

#define **DEF\_KEY\_FILE** "key"

typedef struct {

 long type;

 char buf[100];

} Message;

int queue;

int **main**(int argc, char\*\* argv) {

 char keyFile[100];

**bzero**(keyFile,100);

 if(argc < 2) {

**printf**("Using default key file %s\n",DEF\_KEY\_FILE);

**strcpy**(keyFile,DEF\_KEY\_FILE);

 }

 else

**strcpy**(keyFile,argv[1]);

 key\_t key;

 key = **ftok**(keyFile,'Q');

 if(key == -1) {

**printf**("no got key for key file %s and id 'Q'\n",keyFile);

**exit**(1);

 }

 queue = **msgget**(key,0);

 if (queue < 0) {

**printf**("Can't create queue\n");

**exit**(4);

 }

*// основной цикл работы программы*

 Message mes;

 int res;

 for(;;) {

**bzero**(mes.buf,100);

*// читаем сообщение с консоли*

**fgets**(mes.buf,100,stdin);

 mes.buf[**strlen**(mes.buf) - 1] = '\0';

*// шлем его серверу*

 mes.type = 1L;

 res = **msgsnd**(queue,(void\*)&mes,sizeof(Message),0);

 if(res != 0) {

**printf**("Error while sending msg\n");

**exit**(1);

 }

*// получаем ответ, что все хорошо*

 res = **msgrcv**(queue,&mes,sizeof(Message),2L,0);

 if(res < 0) {

**printf**("Error while recving msg\n");

**exit**(1);

 }

**printf**("Server's response: %s\n",mes.buf);

 }

 return 0;

}

Листинг сервера

#include <stdio.h>

#include <stdlib.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <sys/types.h>

#include <signal.h>

#include <unistd.h>

#include <strings.h>

#include <string.h>

#define **DEF\_KEY\_FILE** "key"

typedef struct {

 long type;

 char buf[100];

} Message;

int queue;

void **intHandler**(int sig) {

**signal**(sig,SIG\_DFL);

 if(**msgctl**(queue,IPC\_RMID,0) < 0) {

**printf**("Can't delete queue\n");

**exit**(1);

 }

}

int **main**(int argc, char\*\* argv) {

 char keyFile[100];

**bzero**(keyFile,100);

 if(argc < 2) {

**printf**("Using default key file %s\n",DEF\_KEY\_FILE);

**strcpy**(keyFile,DEF\_KEY\_FILE);

 }

 else

**strcpy**(keyFile,argv[1]);

 key\_t key;

 key = **ftok**(keyFile,'Q');

 if(key == -1) {

**printf**("no got key for the key file %s and id 'Q'\n",keyFile);

**exit**(1);

 }

 queue = **msgget**(key,IPC\_CREAT | 0666);

 if (queue < 0) {

**printf**("Can't create queue\n");

**exit**(4);

 }

*// до этого момента вызывали exit(), а не kill, т.к. очередь*

*// еще не была создана*

**signal**(SIGINT,intHandler);

*// основной цикл работы сервера*

 Message mes;

 int res;

 for(;;) {

**bzero**(mes.buf,100);

*// получаем первое сообщение с типом 1*

 res = **msgrcv**(queue,&mes,sizeof(Message),1L,0);

 if(res < 0) {

**printf**("Error while recving msg\n");

**kill**(**getpid**(),SIGINT);

 }

**printf**("Client's request: %s\n",mes.buf);

*// шлем клиенту сообщение с типом 2, что все ок*

 mes.type = 2L;

**bzero**(mes.buf,100);

**strcpy**(mes.buf,"OK");

 res = **msgsnd**(queue,(void\*)&mes,sizeof(Message),0);

 if(res != 0) {

**printf**("error while sending msg\n");

**kill**(**getpid**(),SIGINT);

 }

 }

 return 0;

}

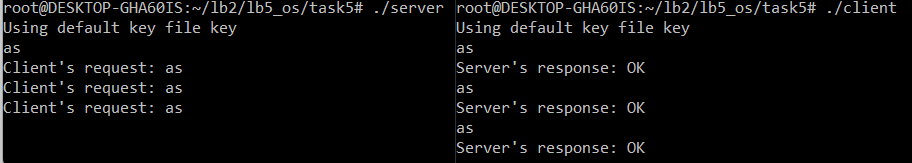


Рисунок 16 – запуск сервера и клиента

Server: Сервер получает ключ, по имени файла. С помощью ключа и идентификатора = 'Q' получает очередь сообщений и ждет сообщений с типом 1 от клиентов.

Client: получение уникального ключа для получения доступа к очереди, получение доступа к созданной сервером очереди. Чтение потока ввода ( сообщения типа 1), посылка строки в очередь.

Server: чтение сообщения типа 1, вывод сообщения на экран и отправка сообщения типа 2.

Client: чтение первого сообщения с типом 2 и вывод его на экран ( OK ). Были написан скрипты, создающие множество серверов и клиентов/множество клиентов для одного сервера.

**Максимальные и минимальные значения констант** можно выяснить различными способами, в частности, просматривая соответствующие файлы каталога /proc/sys/kernel. Наиболее простой способ – воспользоваться утилитой ipcs с ключом -l

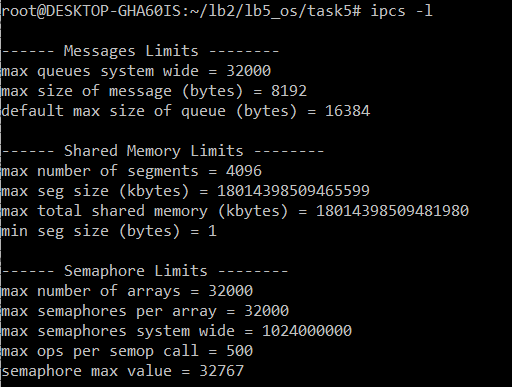


Рисунок 17 – Значения констант в системе

Семафоры

Есть один процесс, выполняющий запись в разделяемую память и один процесс, выполняющий чтение из нее. Под чтением понимается извлечение данных из памяти. Программа должна обеспечить невозможность повторного чтения одних и тех же данных и невозможность перезаписи данных, т. е. новой записи, до тех пор, пока читатель не прочитает предыдущую. В таком варианте задания для синхронизации процессов достаточно двух семафоров.

Покажем, почему не достаточно одного на примере. Так как мы используем один семафор, то алгоритм работы читателя и писателя может быть только таким – захват семафора, выполнение действия (чтение / запись), освобождение семафора. Теперь допустим, что читатель прочитал данные, освободил семафор и еще не до конца использовал квант процессорного времени. Тогда он перейдет на новую итерацию, снова захватит только что освобожденный семафор и снова прочитает данные – ошибка. Теперь покажем, почему достаточно двух семафоров. Придадим одному из них смысл «запись разрешена», т.е. читатель предыдущие данные уже использовал; второму – «чтение разрешено», т.е. писатель уже сгенерировал новые данные, которые нужно прочитать.

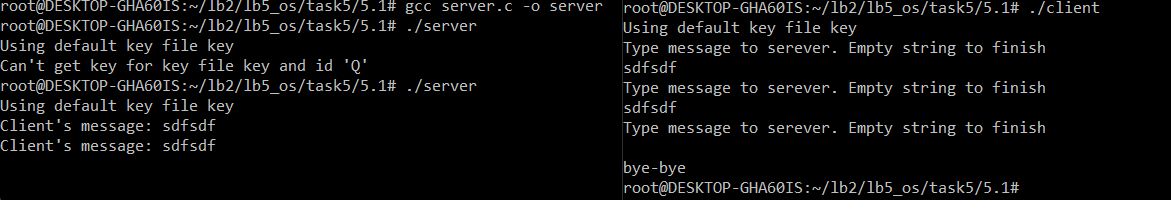


Рисунок 18 – Пример применения семафора

Если в изначальной задаче использовать только один семафор для контроля над доступом к общей памяти, то могут возникнуть следующие проблемы:

**Перезапись данных до их чтения:** Если процесс записи в общую память сможет записать новые данные до того, как процесс чтения успеет прочитать предыдущие данные, то данные могут быть потеряны. Это происходит, когда семафор разрешает процессу записи доступ к общей памяти до того, как процесс чтения успел прочитать предыдущие данные.

**Повторное чтение одних и тех же данных:** Если процесс чтения сможет прочитать данные из общей памяти до того, как процесс записи успеет записать новые данные, то процесс чтения может повторно прочитать одни и те же данные. Это происходит, когда семафор разрешает процессу чтения доступ к общей памяти до того, как процесс записи успел записать новые данные. Эти проблемы возникают из-за отсутствия надлежащего контроля над тем, когда и в каком порядке процессы чтения и записи получают доступ к общей памяти.

Для надлежащего контроля обычно требуется использование двух семафоров: одного для контроля доступа процесса записи, и второго - для контроля доступа процесса чтения. Семафоры, как инструменты синхронизации, используются для контроля доступа к общим ресурсам, в частности, для обеспечения исключающего доступа.

Вот несколько примеров ситуаций, в которых может быть достаточно одного семафора:

**Очередь заданий**: Возьмем для примера сервер, который обрабатывает входящие запросы от клиентов. Все эти запросы помещаются в общую очередь. В то же время, у сервера есть несколько рабочих потоков (worker threads), которые забирают задачи из этой очереди и обрабатывают их. Здесь важно обеспечить, чтобы в одно и то же время только один поток мог взять задачу из очереди, чтобы не возникало конфликтов или ошибок. В этом случае можно использовать один семафор для синхронизации доступа к очереди.

**Доступ к общему файлу или ресурсу**: Представьте, что у вас есть несколько потоков или процессов, которые хотят записать данные в один и тот же файл. Если они начнут делать это одновременно, это может привести к проблемам. В этом случае можно использовать семафор, чтобы гарантировать, что только один процесс или поток может записывать в файл в любой момент времени.

**Обновление общих данных**: Предположим, у вас есть общий счетчик, который используется несколькими потоками. Если несколько потоков попытаются увеличить счетчик одновременно, это может привести к "гонкам" (race conditions) и некорректному результату. Один семафор может быть использован для того, чтобы гарантировать, что только один поток обновляет счетчик в любой момент времени.

В каждом из этих примеров один семафор используется для контроля доступа к общему ресурсу и предотвращения проблем, связанных с одновременным доступом.

* 1. Реализовано добавление множества клиентов и серверов для двух семафоров.

Листинг скрипта

*#! /bin/bash*

num\_readers=2

num\_writers=2

for (( i=0*; i<$num\_readers; i++ ));*

do

    gnome-terminal -- ./server

done

for (( i=0*; i<$num\_writers; i++));*

do

    gnome-terminal -- ./client

done

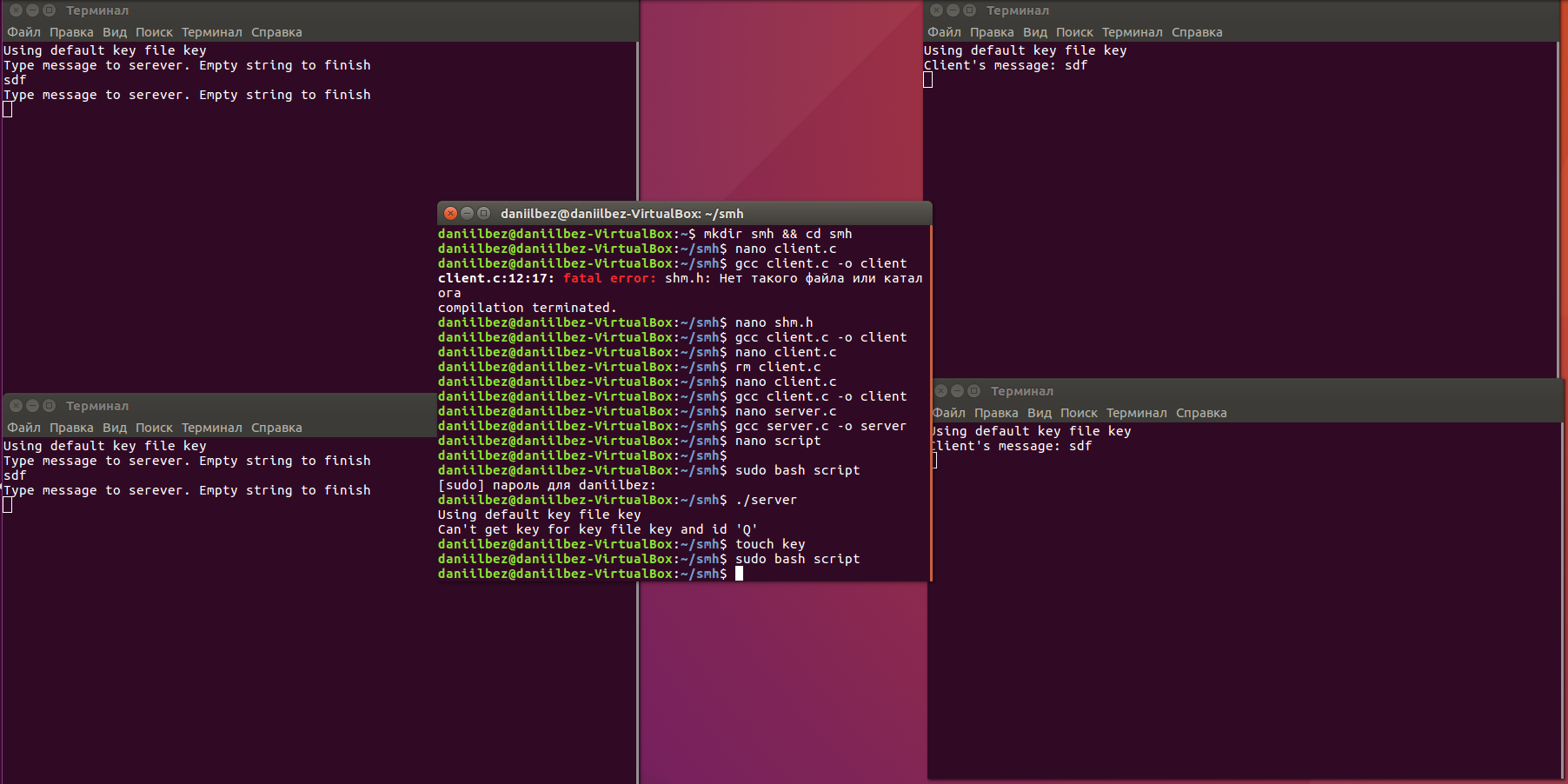


Рисунок 19 – Результат запуска скрипта

1. Cокеты

Пример использования сокета – эхо сервер. Рассмотрим пример программы – сервер прослушивает заданный порт, при запросе нового соединения создаётся новый поток для его обработки. Работа с клиентом организована как бесконечный цикл, в котором выполняется приём сообщения от клиента, вывод его на экран и пересылка обратно клиенту. Клиентская программа после установления соединения с сервером также в бесконечном цикле выполняет чтение ввода пользователя, пересылку серверу, получение работы. Для взаимодействия используются TCP сокеты, это значит, что между сервером и клиентом устанавливается логическое соединение, при этом при получении данных из сокета с помощью вызова recv, есть вероятность получить сразу несколько сообщений, или не полностью прочитать сообщение. Поэтому для установления взаимной однозначности между отосланными и принятыми данными используются функции recvFix и sendFix. Принцип их работы следующий: функция sendFix перед посылкой собственно данных посылает «заголовок» - количество байт в посылке. Функция recvFix вначале принимает этот «заголовок», и вторым вызовом recv считывает переданное количество байт. Считать ровно то, количество байт, которое указанно в аргументе функции recv, позволяет флаг MSG\_WAITALL. Если его не использовать и данных в буфере недостаточно, то будет прочитано меньшее количество. Протестируем предложенное приложение. Для этого немного модифицируем его (каждый клиент при соединении отправит свой порядковый номер серверу). Далее напишем bash-скрипт, создающий N число клиентов. Содержание программ в листинге 18. Результат работы на рисунке 18-19.

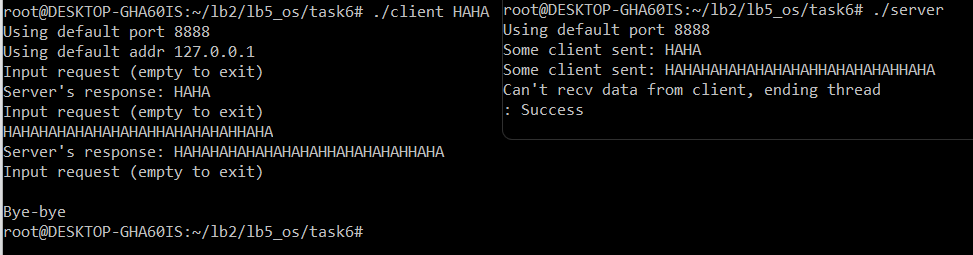


Рисунок 20 – запуск сервера на одном компьютере

Приведём пример использования программы в локальной сети. Использованы виртуальные машины VirtualBox. На них настроены интерфейсы с IP-адресами 15.0.1.3/24 и 15.0.1.4/24 соответственно. Результат на рисунке 20.

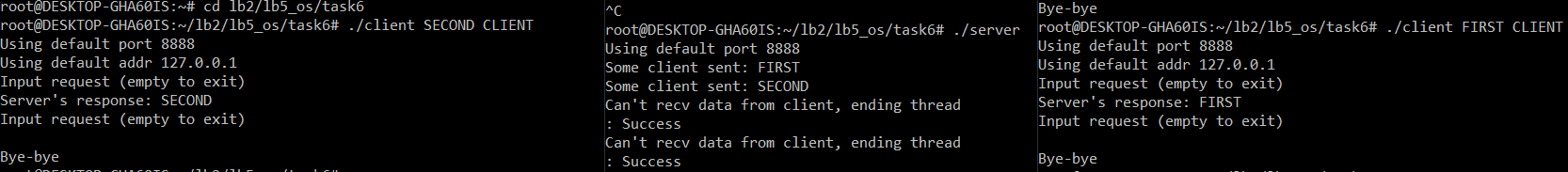


Рисунок 21 – запуск двух клиентов и одного сервера

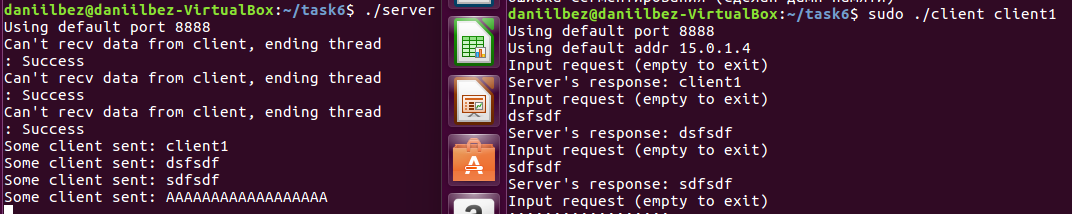


Рисунок 22 – Запуск клиента и сервера на разных машинах

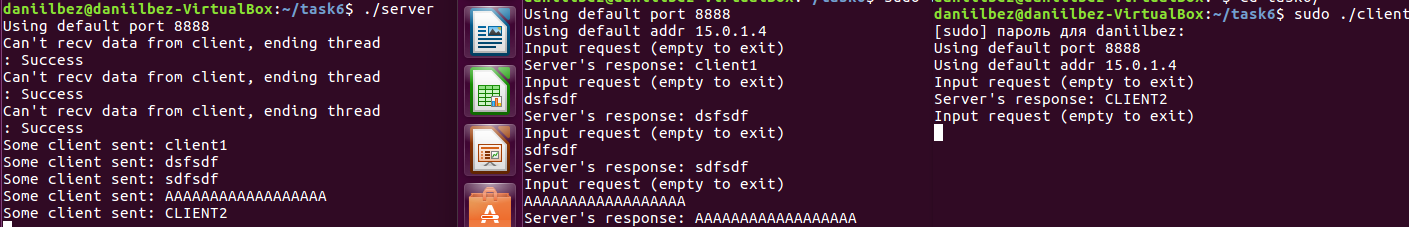


Рисунок 23 – Запуск нескольких клиентов и одного сервера

Чтобы выполнить аналогичное взаимодействие на основе UDP, при создании сокета нужно заменить SOCK\_STREAM на SOCK\_DGRAM, функции recv() и send() на recvfrom() и sendto() соответственно. Рассмотрим пример работы программы на рисунке 19.

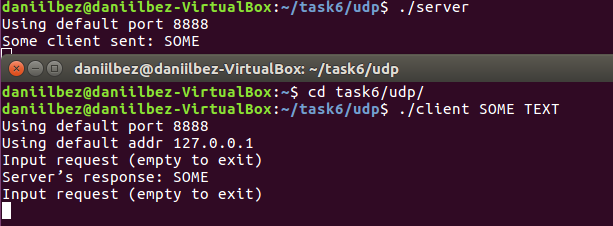


Рисунок 24 – Запуск UDP-сервера и клиента на одной машине

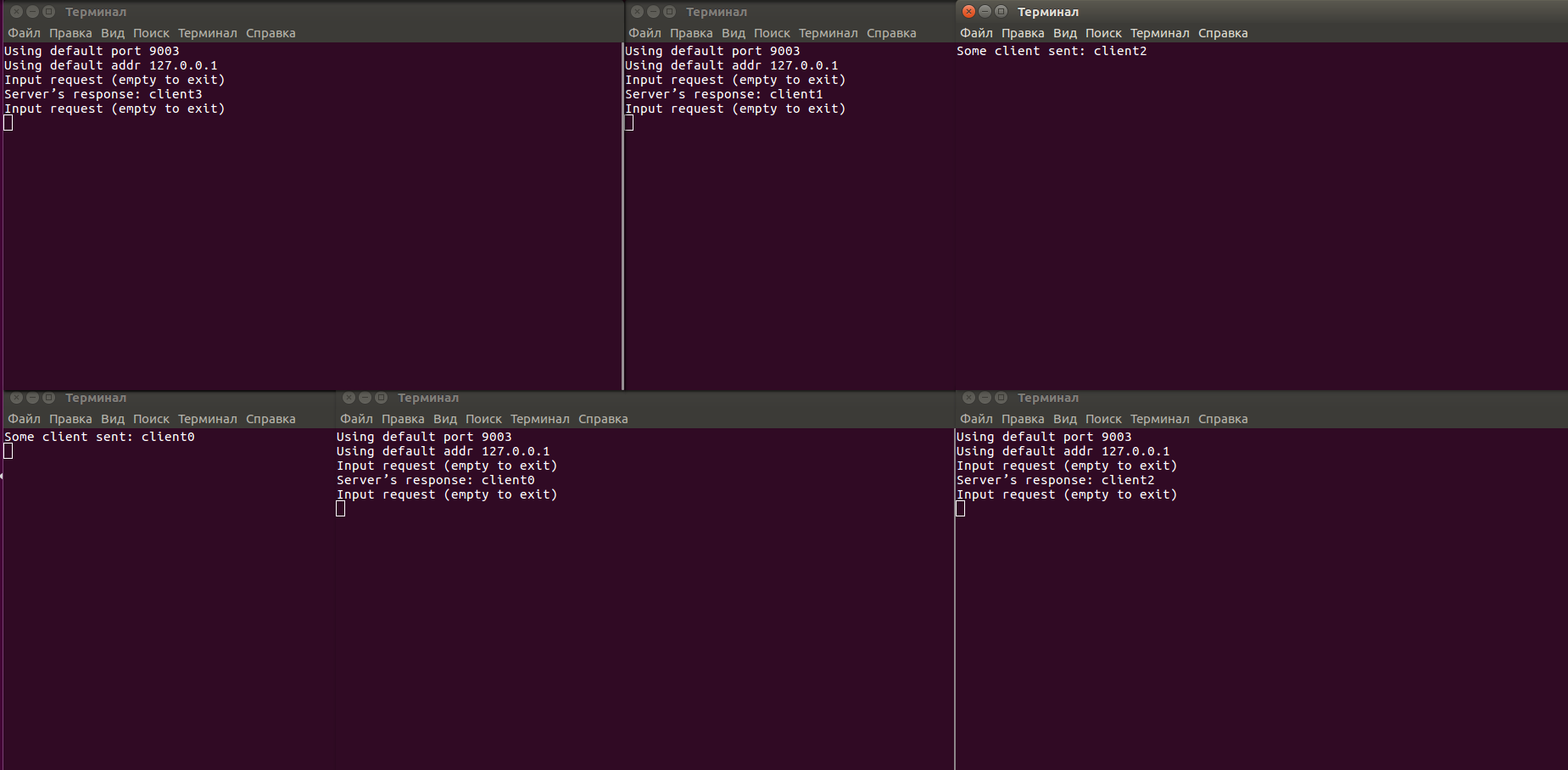


Рисунок 25 – запуск нескольких серверов и нескольких клиентов с помощью скрипта на одной машине

Попробуем запустить на одной машине 100 клиентов и 1 сервер.

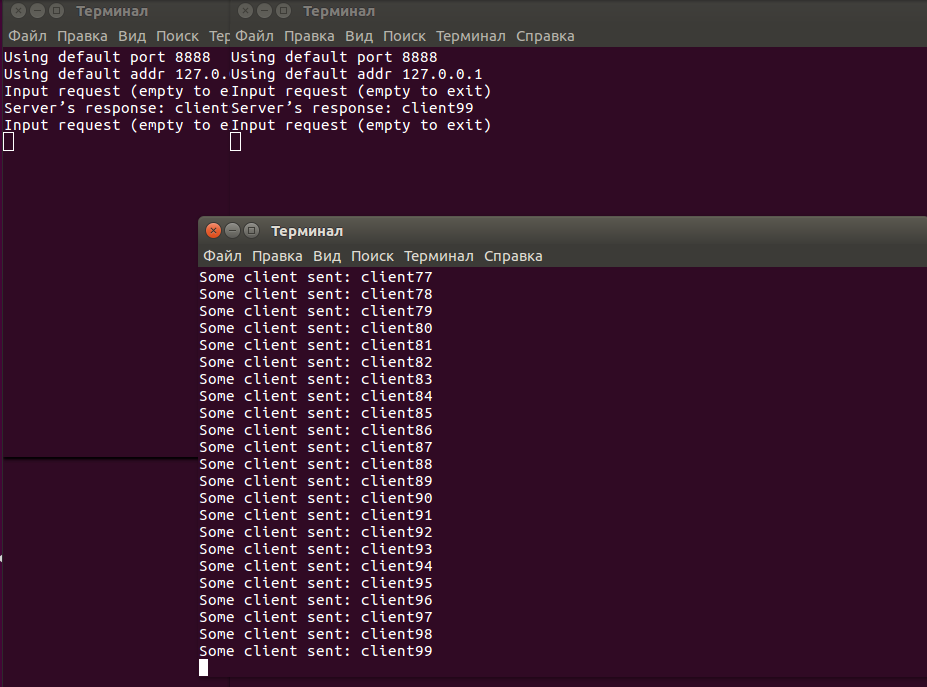


Рисунок 22 – запуск 100 клиентов и 1 сервера

С помощью скрипта запустим 100 клиентов, каждый из которых при запуске будет отправлять на сервер запрос со своим номером – если все клиенты запустятся корректно, то в консоли сервера мы увидим 100 записей. Как видно на рисунке – клиенты запустились корректно.

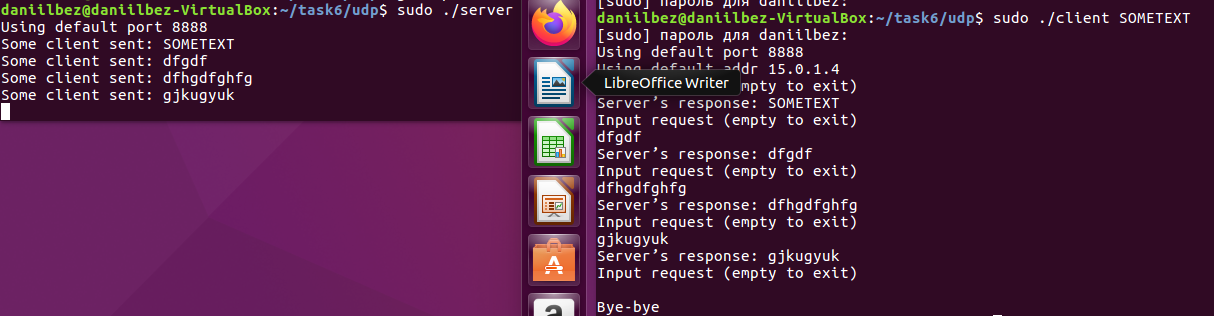
Также на двух машинах была настроена одна сеть и запущены клиент и сервер c адресами 15.0.1.3/24 и 15.0.1.4/24 соответственно, обменивающиеся данными через UDP.

Рисунок 26 – запуск UDP-сервера и клиента на разных машинах

Можно заключить, что между очередями сообщений и сокетами имеются отличия: в первом случае передача данных осуществляется в формате структурированных сообщений, а во втором – в виде байтов. Кроме того, сокеты поддерживают различные протоколы (TCP или UDP), в то время как очереди сообщений используют единый механизм передачи.

Вывод

В результате выполнения работы мы расширили свои знания о механизмах межпроцессного взаимодействия в операционной системе Linux. Мы изучили различные методы передачи данных между процессами, такие как сигналы, каналы, очереди сообщений и сокеты. Кроме того, мы изучили различия между надежными и ненадежными сигналами, а также сигналами реального времени.

В процессе работы мы также углубились в изучение сокетов, провели несколько экспериментов и успешно реализовали обмен сообщениями с помощью сокетов между двумя машинами в одной подсети, используя как протокол TCP, так и UDP. Кроме того, мы освоили практику работы с сигналами, в том числе отправку сигналов от неродственных процессов. Эти знания и навыки могут быть полезны в дальнейшей работе с операционными системами и сетевыми приложениями

**Список источников.**

1. Geeks for Geeks –<https://www.geeksforgeeks.org/>

2. Linux.org –<https://www.linux.org.ru/forum/general/>

3. Linux Documentation – linux.die.net/man

4. Душутина Е.В. Системное программное обеспечение. Межпроцессные взаимодействия в операционных системах: учеб. пособие /Е.В. Душутина. – СПб.: Изд-во Политехн.ун-та, 2016. – 180 с. (ISBN 978-5-7422-5401-0)

5. А.М. Робачевский Операционная система UNIX: учебник / А.М. Робачевский [и др.]. – 2-е изд., перераб.и доп. – СПб. : Изд-во БХВ-Петербург. 2010. – 648 с.