# МИНОБРНАУКИ РОССИИ

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

# «ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

**Кафедра МОЭВМ**

# ОТЧЕТ

**по лабораторной работе №5**

# по дисциплине «Операционные системы» Тема: IPC.

|  |  |
| --- | --- |
| Студент гр. 1384 | Мамин Р. А. |
| Преподаватель | Душутина Е.В. |

Санкт-Петербург 2023

# Цель работы.

Цель работы заключается в изучении основных концепций порождения и запуска процессов в операционной системе, а также изучении взаимодействия родственных процессов и методов управления ими с помощью сигналов. В рамках работы необходимо ознакомиться с механизмом порождения процессов в ОС, изучить способы управления процессами, такие как приостановка, продолжение, остановка, завершение и изменение приоритета выполнения процессов. Также следует изучить механизм передачи информации между родственными процессами через каналы и обмен сообщениями с помощью сигналов. Для достижения цели необходимо изучить соответствующие системные вызовы и функции в языке программирования Си, а также разработать и отладить программы, демонстрирующие порождение и управление процессами, обмен сообщениями и сигналами между ними.

# Характеристики операционной системы.

Операционная система: *Ubuntu 20.04 LTS*

Ядро: *Linux 5.15.0-69-generic*

* 1. **Ненадёжные сигналы**
     1. Создана программа, позволяющая изменить диспозицию сигналов, а именно, установить: - обработчик пользовательских сигналов SIGSEGV и SIGTRAP; - реакцию по умолчанию на сигнал SIGCHLD; Породить процесс- копию и уйти в ожидание сигналов. Обработчик сигналов должен содержать восстановление диспозиции и оповещение на экране о полученном сигнале и идентификаторе родительского процесса. Процесс-потомок, получив идентификатор родительского процесса, должен отправить процессу-отцу сигнал SIGTRAP и извещение об удачной или неудачной отправке указанного сигнала. Остальные сигналы можно сгенерировать из командной строки.

*1\_1.c*

#include <signal.h> #include <stdio.h> #include <sched.h> #include <sys/mman.h> #include <fcntl.h> #include <stdlib.h> #include <unistd.h> #include <sys/types.h> #include <sys/wait.h>

*static void* sigsegv\_handler( *int* signo );

*static void* sigtrap\_handler(*int* signo) { signal(SIGTRAP, sigtrap\_handler);

printf("Signal SIGTRAP is handled in father!\n");

}

*int* main(){

pid\_t child\_pid; pid\_t pid = getpid();

printf( "Father process is started! "); printf( "PID: %d\n", pid );

*/\* Установка диспозиций сигналов \*/*

signal( SIGCHLD, SIG\_DFL);

signal( SIGTRAP, sigtrap\_handler); signal( SIGSEGV, sigsegv\_handler);

*if*((child\_pid = fork()) == 0 ){ execl("son", "son", NULL);

}

*else* {

*while*(1) pause();

}

*return* 0;

}

*static void* sigsegv\_handler(*int* signo) { signal(SIGSEGV, sigsegv\_handler); printf("Signal SIGSEGV is handled!\n");

}

*son.c*

#include <signal.h> #include <stdio.h> #include <sched.h> #include <sys/mman.h> #include <fcntl.h> #include <stdlib.h> #include <unistd.h> #include <sys/types.h> #include <sys/wait.h>

*int* main() {

pid\_t child\_pid = getpid(); pid\_t ppid = getppid();

printf("Child process is started!\n"); printf("PPID: %d, PID: %d\n", ppid, child\_pid);

kill(ppid, SIGTRAP);

printf("Signal SIGTRAP is sended by son!\n");

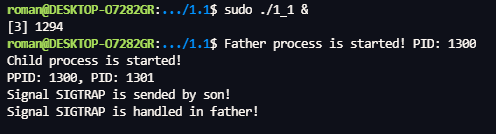
*while*(1) {sleep(2);}

*return* 0;

}

Программа от приведённой в примере отличается тем, что процесс порождается в отдельном файле через *execl()* и обрабатывает другие сигналы.

Результат работы программы:



Отправим сигнал из терминала:



Cигналы успешно обрабатываются для процессов в разных файлах.

Теперь проделаем аналогичную работу с потоками. В следующей программе один поток порождает другой и отправляет ему сигналы. Также предусмотрена возможность отправки сигналов из терминала.

*1\_2.c*

#include <stdio.h> #include <stdlib.h> #include <unistd.h> #include <pthread.h> #include <signal.h> #include <sys/syscall.h>

*volatile* sig\_atomic\_t running = 1;

*void* signal\_handler(*int* signum) { pid\_t tid = syscall(SYS\_gettid);

printf("Поток с TID %d получил сигнал: %d\n", tid, signum);

*if* (signum == SIGINT) { running = 0;

}

}

*void* \*child\_thread(*void* \*arg) {

*// Установка обработчика сигналов для потока*

signal(SIGUSR1, signal\_handler); signal(SIGUSR2, signal\_handler); signal(SIGINT, signal\_handler); signal(SIGTERM, signal\_handler);

pid\_t tid = syscall(SYS\_gettid);

printf("Дочерний поток с TID %d запущен\n", tid);

*// Цикл выполняется, пока переменная running равна 1 while* (running) {

sleep(1);

}

printf("Дочерний поток с TID %d завершен\n", tid);

*return* NULL;

}

*int* main() {

pthread\_t thread\_id;

*int* status;

*// Создание потока*

status = pthread\_create(&thread\_id, NULL, child\_thread, NULL);

*if* (status != 0) { perror("pthread\_create"); exit(EXIT\_FAILURE);

}

*// Установка обработчика сигналов для главного потока*

signal(SIGUSR1, signal\_handler); signal(SIGUSR2, signal\_handler); signal(SIGINT, signal\_handler); signal(SIGTERM, signal\_handler);

pid\_t tid = syscall(SYS\_gettid);

printf("Главный поток с TID %d запущен\n", tid); sleep(3);

pthread\_kill(thread\_id, SIGUSR1); pthread\_kill(thread\_id, SIGUSR2);

*// Цикл выполняется, пока переменная running равна 1 while* (running) {

sleep(1);

}

printf("Главный поток с TID %d завершен\n", tid);

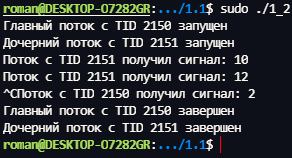
*// Ожидание завершения дочернего потока*

pthread\_join(thread\_id, NULL);

*return* 0;

}

Результат работы программы:



Теперь разделим потоки на отдельные файлы:

*child.c*

#include <stdio.h> #include <stdlib.h> #include <unistd.h> #include <pthread.h> #include <signal.h> #include <sys/syscall.h>

*volatile* sig\_atomic\_t running = 1;

*void* signal\_handler(*int* signum) { pid\_t tid = syscall(SYS\_gettid);

printf("Поток с TID %d получил сигнал: %d\n", tid, signum);

*if* (signum == SIGINT) { running = 0;

}

}

*void* \*child\_thread(*void* \*arg) {

*// Установка обработчика сигналов для потока*

signal(SIGUSR1, signal\_handler); signal(SIGUSR2, signal\_handler); signal(SIGINT, signal\_handler); signal(SIGTERM, signal\_handler);

pid\_t tid = syscall(SYS\_gettid);

printf("Дочерний поток с TID %d запущен\n", tid);

*// Цикл выполняется, пока переменная running равна 1 while* (running) {

sleep(1);

}

printf("Дочерний поток с TID %d завершен\n", tid);

*return* NULL;

}

pthread\_t create\_child\_thread() { pthread\_t thread\_id;

*int* status;

*// Создание потока*

status = pthread\_create(&thread\_id, NULL, child\_thread, NULL);

*if* (status != 0) { perror("pthread\_create"); *return* 0;

}

*return* thread\_id;

}

*Main.c*

#include <stdio.h> #include <stdlib.h> #include <unistd.h> #include <pthread.h> #include <signal.h> #include <sys/syscall.h>

*extern* pthread\_t create\_child\_thread();

*extern volatile* sig\_atomic\_t running;

*extern void* signal\_handler(*int* signum);

*int* main() {

pthread\_t thread\_id;

*int* status;

*// Создание потока*

thread\_id = create\_child\_thread();

*if* (thread\_id == 0) { perror("pthread\_create"); exit(EXIT\_FAILURE);

}

*// Установка обработчика сигналов для главного потока*

signal(SIGUSR1, signal\_handler); signal(SIGUSR2, signal\_handler); signal(SIGINT, signal\_handler); signal(SIGTERM, signal\_handler);

pid\_t tid = syscall(SYS\_gettid);

printf("Главный поток с TID %d запущен\n", tid); sleep(3);

pthread\_kill(thread\_id, SIGUSR1); pthread\_kill(thread\_id, SIGUSR2);

*// Цикл выполняется, пока переменная running равна 1 while* (running) {

sleep(1);

}

printf("Главный поток с TID %d завершен\n", tid);

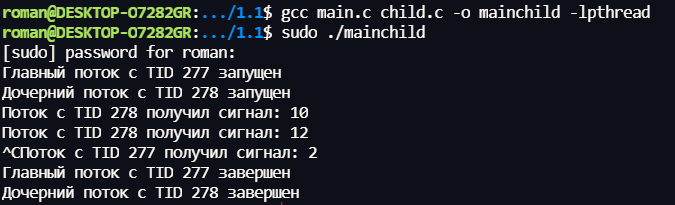
*// Ожидание завершения дочернего потока*

pthread\_join(thread\_id, NULL);

*return* 0;

}

Результат работы программы:



# 2 Надёжные сигналы

* + - 1. Создана программа, позволяющая продемонстрировать возможность отложенной обработки (временного блокирования) сигнала (например, SIGINT).

Вся необходимая для управления сигналами информация передается через указатель на структуру sigaction. Блокировку реализуем, вызвав "засыпание" процесса на одну минуту из обработчика пользовательских сигналов. В основной программе установим диспозицию этих сигналов. С рабочего терминала отправим процессу sigact сигнал SIGUSR1 или SIGUSR2, а затем сигнал SIGINT.

Код программы 2.c:

|  |  |  |
| --- | --- | --- |
| #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) { | | |

}

mysig(SIGUSR1, hndUSR1);

*for* (;;) { pause();

}

*return* 0;

}

*int* main() {

*if* (sig != SIGUSR1) {

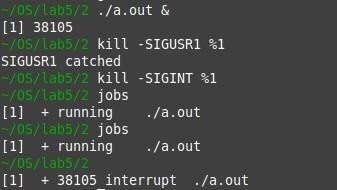
printf("Catched bad signal %d\n", sig);

*return*;

}

printf("SIGUSR1 catched\n"); sleep(60);

}



Чтобы иметь возможность отправить сигналы с терминала следует запустить программу в фоновом режиме. По результатам сигнал SIGUSR1 принят корректно, но после посылки сигнала SIGINT программа продолжала выполняться еще 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 завершенияобработки сигналов, которые отложили обработку. Механизм ненадёжных сигналов не позволяет откладывать обработку других сигналов (можно лишь установить игнорирование некоторых сигналов на время обработки).

2.2. Изменим обработчик сигнала так, чтобы из него производилась отправка другого сигнала. Пусть из обработчика сигнала SIGUSR1 функцией kill() генерируется сигнал SIGINT. Проанализируем наличие и очередность обработки сигналов.

*2\_2.с*

*if*(sig != SIGUSR1) {

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

*return*;

}

printf("SIGUSR1 catched, sending SIGINT\n"); kill(getpid(),SIGINT);

sleep(10);

#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) {

}

*int* main() { mysig(SIGUSR1,hndUSR1); *for*(;;) {

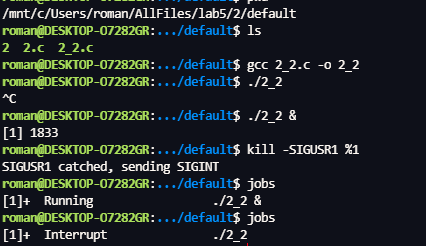
pause();

}

*return* 0;

}

Вывод программы:



Преимущества надежных сигналов по сравнению с ненадежным:

Диспозиция устанавливается только один раз, и диспозицию не нужно устанавливать каждый раз в функции-обработчике.

При установке диспозиции, можно указать какие сигналы будут блокироваться на время обработки конкретного сигнала, это значительно увеличивает надежность обработки сигнала.

Напишем программу, которая наглядно демонстрирует разницу обработки надёжных и ненадёжных сигналов.

*Demo.c*

printf("\nНадежный сигнал (SIGINT) получен.\n");

#include <stdio.h> #include <stdlib.h> #include <signal.h> #include <unistd.h>

*volatile int* reliable\_signal\_count = 0; *volatile int* total\_reliable\_signals\_sent = 0; *volatile int* unreliable\_signal\_count = 0;

*volatile int* total\_unreliable\_signals\_sent = 0;

*void* sigint\_handler(*int* signal\_number) {

printf("Отправлено надежных сигналов (SIGUSR1): %d\n", total\_reliable\_signals\_sent); printf("Обработано надежных сигналов (SIGUSR1): %d\n", reliable\_signal\_count); printf("Потеряно надежных сигналов (SIGUSR1): %d\n",

total\_reliable\_signals\_sent - reliable\_signal\_count);

printf("\nОтправлено ненадежных сигналов (SIGALRM): %d\n", total\_unreliable\_signals\_sent); printf("Обработано ненадежных сигналов (SIGALRM): %d\n", unreliable\_signal\_count); printf("Потеряно ненадежных сигналов (SIGALRM): %d\n",

total\_unreliable\_signals\_sent - unreliable\_signal\_count); exit(0);

}

*void* sigusr1\_handler(*int* signal\_number) { reliable\_signal\_count++;

sleep(2); *// Искусственная задержка для имитации долгой обработки сигнала*

}

*void* sigalrm\_handler(*int* signal\_number) { unreliable\_signal\_count++;

sleep(2); *// Искусственная задержка для имитации долгой обработки сигнала*

signal(SIGALRM, sigalrm\_handler); *// Восстанавливаем обработчик сигнала после вызова*

}

*int* main() {

*struct* sigaction sa\_int, sa\_usr1;

sa\_int.sa\_handler = sigint\_handler; sigemptyset(&sa\_int.sa\_mask); sa\_int.sa\_flags = 0;

sa\_usr1.sa\_handler = sigusr1\_handler; sigemptyset(&sa\_usr1.sa\_mask); sa\_usr1.sa\_flags = 0;

sigaction(SIGINT, &sa\_int, NULL); sigaction(SIGUSR1, &sa\_usr1, NULL); signal(SIGALRM, sigalrm\_handler);

printf("Демонстрация работы сигналов:\n"); printf("Процесс ID: %d\n", getpid());

printf("Надежный сигнал (SIGINT): остановка программы и вывод результатов\n"); printf("Надежный сигнал (SIGUSR1): инкремент счетчика без потерь\n"); printf("Ненадежный сигнал (SIGALRM): инкремент счетчика с возможны\n"); printf("Ожидание сигналов...\n");

sigset\_t mask, oldmask; sigemptyset(&mask); sigaddset(&mask, SIGUSR1); sigaddset(&mask, SIGALRM); *while* (1) {

sigprocmask(SIG\_BLOCK, &mask, &oldmask);

kill(getpid(), SIGUSR1); *// Генерируем надежный сигнал SIGUSR1 для текущего процесса*

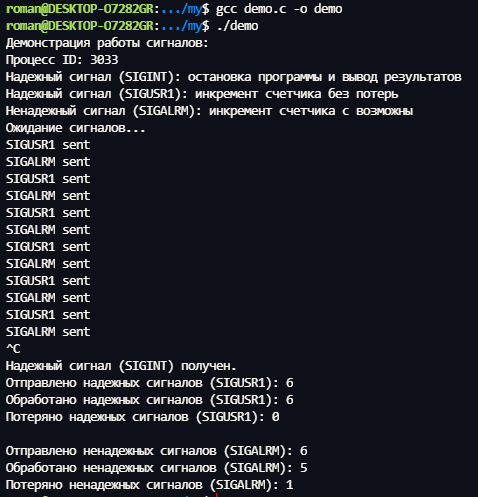
printf("SIGUSR1 sent\n"); total\_reliable\_signals\_sent++;

kill(getpid(), SIGALRM); *// Генерируем ненадежный сигнал SIGALRM для текущего процесса*

printf("SIGALRM sent\n"); total\_unreliable\_signals\_sent++;

|  |  |  |
| --- | --- | --- |
|  | | sigsuspend(&oldmask); *// Ожидание сигналов*  sigprocmask(SIG\_UNBLOCK, &mask, NULL); |
|  | }  *return* 0; | |
| } | | |

В данной программе в бесконечном цикле отправляются надёжные и ненадёжные сигналы и в конце программы выводится статистика отправленных и обработанных сигналов.



Видим, что ненадёжные сигналы теряются в отличие от надёжных.

# Сигналы реального времени

* 1. – 3.2. Был проведен эксперимент, позволяющий определить возможность организации очереди для различных типов сигналов, обычных и реального времени, (более двух сигналов, для этого увеличьте «вложенность» вызовов обработчиков);

Также экспериментально было подтверждено, что обработка равноприоритетных сигналов реального времени происходит в порядке FIFO;

*3\_1.с*

*// Подключение необходимых заголовочных файлов*

#include <stdio.h> #include <stdlib.h> #include <unistd.h> #include <signal.h>

*// Определение количества сигналов для отправки*

#define NUM\_SIGNALS 4

*// Обработчик сигналов*

*void* handle\_signal(*int* signo, siginfo\_t \*info, *void* \*context) {

*// Вывод информации о полученном сигнале*

printf("Received signal %d with value %d\n", signo, info->si\_value.sival\_int);

}

*int* main() {

*int* i;

*struct* sigaction sa; *// Структура для настройки обработчика сигнала*

*union* sigval value; *// Объединение для хранения значения сигнала*

*// Настройка обработчика сигналов* sa.sa\_sigaction = handle\_signal; sa.sa\_flags = SA\_SIGINFO; sigemptyset(&sa.sa\_mask);

*for* (i = SIGRTMIN; i <= SIGRTMAX; i++) {

sigaddset(&sa.sa\_mask, i);

}

*for* (i = 1; i <= NUM\_SIGNALS; i++) {

sigaction(SIGINT + i, &sa, NULL); sigaction(SIGRTMIN + i, &sa, NULL);

}

*// Отправка сигналов*

*for* (i = 1; i <= NUM\_SIGNALS; i++) {

value.sival\_int = i;

printf("Sending normal signal %d with value %d\n", SIGINT + i, value.sival\_int);

*if* (sigqueue(getpid(), SIGINT + i, value) != 0) { perror("sigqueue");

exit(EXIT\_FAILURE);

}

printf("Sending real-time signal %d with value %d\n", SIGRTMIN + i, value.sival\_int);

*if* (sigqueue(getpid(), SIGRTMIN + i, value) != 0) { perror("sigqueue");

exit(EXIT\_FAILURE);

}

}

*// Ожидание обработки сигналов*

sleep(1);

*return* 0;

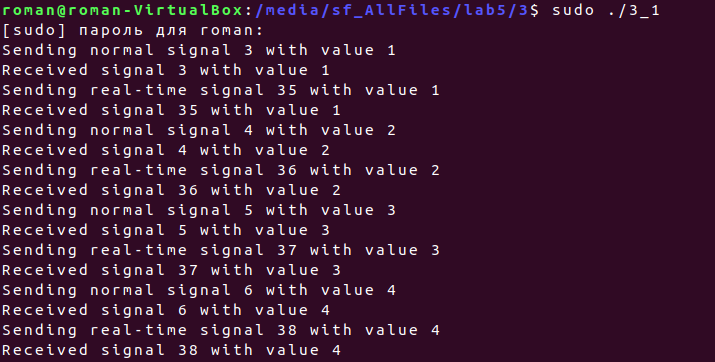
}

Проведенный эксперимент включает отправку в общей сложности 8

сигналов (4 обычных сигнала и 4 сигнала реального времени) одному и тому же процессу с разными значениями, присвоенными каждому сигналу. Обработчик этих сигналов просто выводит номер сигнала и присоединенное значение.

Ожидается, что сигналы будут обрабатываться в порядке FIFO, независимо от того, являются ли они обычными сигналами или сигналами реального времени. Это связано с тем, что сигналы отправляются в один и тот же процесс и должны ставиться в очередь в порядке их получения.

Эксперимент подтверждает это ожидание, так как сигналы обрабатываются в порядке их отправки. В частности, сначала обрабатываются обычные сигналы в том порядке, в котором они были отправлены (т. е. 1, 2, 3, 4), а затем сигналы реального времени в том порядке,в котором они были отправлены (т. е. 1, 2, 3, 4).

Это демонстрирует, что сигналы разных типов могут ставиться в очередь и обрабатываться в согласованном и предсказуемом порядке, если они отправляются в один и тот же процесс. Кроме того, он подтверждает, что сигналы реального времени с одинаковым приоритетом обрабатываются в порядке FIFO, как и обычные сигналы.

3.3. Подтвердим наличие приоритетов сигналов реального времени

*3\_2.с*

#include <stdio.h> #include <stdlib.h> #include <signal.h> #include <unistd.h>

#define NUM\_SIGNALS 5

*// Обработчик сигналов реального времени*

*void* handle\_rt\_signal(*int* sig, siginfo\_t\* info, *void*\* context) { printf("Handling real-time signal %d\n", sig); usleep(1000000); *// Sleep for 1 second*

}

*// Обработчик обычных сигналов*

*void* handle\_normal\_signal(*int* sig) { printf("Handling normal signal %d\n", sig);

}

*int* main() {

*// Перенаправление вывода в файл*

freopen("output.txt", "w", stdout);

*// Регистрация обработчика сигналов реального времени*

*struct* sigaction sa\_rt; sa\_rt.sa\_sigaction = handle\_rt\_signal; sa\_rt.sa\_flags = SA\_SIGINFO; sigemptyset(&sa\_rt.sa\_mask);

*for* (*int* i = SIGRTMIN; i <= SIGRTMAX; i++) {

sigaction(i, &sa\_rt, NULL);

}

*// Регистрация обработчика обычных сигналов*

*for* (*int* i = 1; i <= NUM\_SIGNALS; i++) { signal(i, handle\_normal\_signal);

}

*// Отправка сигналов*

printf("Sending signals...\n");

*for* (*int* i = 1; i <= NUM\_SIGNALS; i++) { printf("Sending normal signal %d\n", i); kill(getpid(), i);

}

*for* (*int* i = SIGRTMIN; i <= SIGRTMAX; i++) {

printf("Sending real-time signal %d\n", i); kill(getpid(), i);

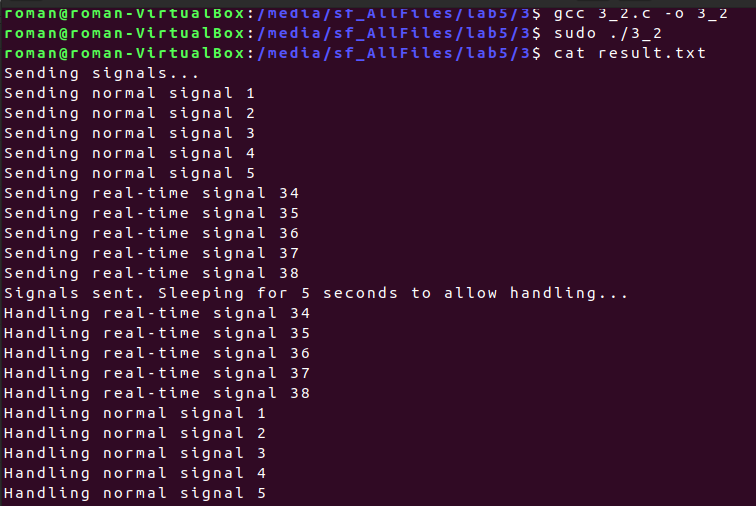
}

*// Ожидание обработки сигналов (засыпание на 5 секунд)* printf("Signals sent. Sleeping for 5 seconds to allow handling...\n"); sleep(5);

*return* 0;

}

Результат работы программы:



Это подтверждает, что сигналы реального времени имеют более высокий приоритет, чем обычные сигналы, и что они обрабатываются в первую очередь, даже если отправляются позже.

# 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).

Код программы:

#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);

}

}

*return* 0;

}

fclose(f); close(filedes[0]);

printf("Read from pipe: %s\n", buf); fwrite(buf, *sizeof*(*char*), res, f);

*if* (!res)

*break*;

*while* (1) {

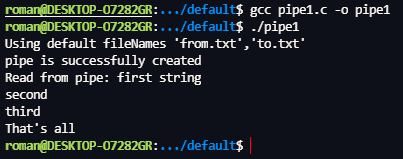
bzero(buf, 100);

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

*char* buf[100];

*int* res;

Вывод:



Содержимое файла from.txt успешно переписалось в изначально пустой файл to.txt с использованием неименованного канала.

Так как процесс-родитель только читает из канала, то дескриптор для записи (filedes[1]) он закрывает, аналогично процесс-сын в начале работы закрывает дескриптор для чтения из канала (filedes[0]).

Процесс-потомок читает строки из файла, записывает в канал, далее процесс родитель читает из канала строки и записывает их в файл. При этом процесс-потомок закрывает дескриптор канала на чтение. А процесс-родитель закрывает дескриптор канала на запись. Тем самым соблюдается передача данных в одну сторону.

4.2.

Код server.c:

#include <stdio.h> #include <stdlib.h> #include <unistd.h> #include <string.h> #include <sys/types.h> #include <sys/stat.h> #include <fcntl.h>

#define DEF\_FILENAME "testFile.txt"

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

*char* fileName[30];

*if*(argc < 2) {

printf("Using default file name '%s'\n",DEF\_FILENAME); strcpy(fileName,DEF\_FILENAME);

}

*else*

strcpy(fileName,argv[1]);

*// создаем два канала*

*int* res = mknod("channel1",S\_IFIFO | 0666,0);

*if*(res) {

printf("Can't create first channel\n"); exit(1);

}

res = mknod("channel2",S\_IFIFO | 0666,0);

*if*(res) {

printf("Can't create second channel\n"); exit(1);

}

*// открываем первый канал для записи int* chan1 = open("channel1",O\_WRONLY); *if*(chan1 == -1) {

printf("Can't open channel for writing\n"); exit(0);

}

*// открываем второй канал для чтения int* chan2 = open("channel2",O\_RDONLY); *if*(chan2 == -1) {

printf("Can't open channe2 for reading\n"); exit(0);

}

*// пишем имя файла в первый канал*

write(chan1,fileName,strlen(fileName));

*// читаем содержимое файла из второго канала*

*char* buf [100];

*for*(;;) {

bzero(buf,100);

res = read(chan2,buf,100);

*if*(res <= 0)

*break*;

printf("Part of file: %s\n", buf);

}

close(chan1); close(chan2); unlink("channel1");

}

unlink("channel2");

*return* 0;

Код client.c:

#include <stdio.h> #include <stdlib.h> #include <unistd.h> #include <string.h> #include <sys/types.h> #include <sys/stat.h> #include <fcntl.h>

*int* main() {

*// каналы сервер уже создал, открываем их*

*int* chan1 = open("channel1", O\_RDONLY);

*if* (chan1 == -1) {

printf("Can't open channel1 for reading\n"); exit(0);

}

*int* chan2 = open("channel2", O\_WRONLY);

*if* (chan2 == -1) {

printf("Can't open channel2 for writing\n"); exit(0);

}

*// читаем имя файла из первого канала*

*char* fileName[100]; bzero(fileName, 100);

*int* res = read(chan1, fileName, 100);

*if* (res <= 0) {

printf("Can't read fileName from channel1\n"); exit(0);

}

*// открываем файл на чтение*

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

*if* (!f) {

printf("Can't open file %s\n", fileName); exit(0);

}

*// читаем из файла и пишем во второй канал*

*char* buf[100];

*while* (!feof(f)) {

*// читаем данные из файла*

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

*// пишем их в канал*

write(chan2, buf, res);

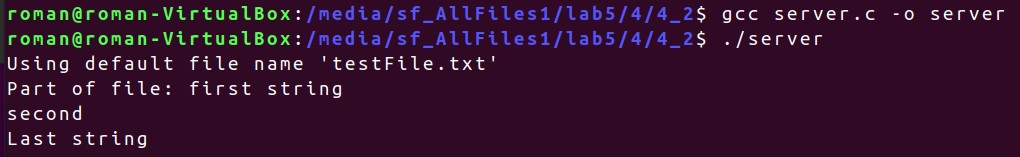
}

fclose(f); close(chan1); close(chan2);

*return* 0;

}

Вывод:



\



Сервер создает два канала, записывает в один из них имя файла и ждёт данные от клиента. Каналы создаются в рабочей папке сервера, и использовать их может любой процесс, а не только дочерний по отношению к серверу. Клиент после запуска также открывает уже созданные каналы, считывает имя файла и отсылает серверу его содержимое, используя второй канал. После завершения передачи, сервер уничтожает каналы с помощью функции unlink().

Server: создает 2 именованных канала — первый на запись, второй — на чтение. Записывает имя файла в первый канал. Client: открывает первый канал

— на чтение, второй — на запись. Читает имя файла из первого канала, открывает файл и записывает содержимое во второй канал. Server: чтение строк из второго канала и вывод на экран.

Написан скрипт, создающий множество клиентов и серверов:

*script.sh*

*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" &

#!/bin/bash

*gcc* server.c -o server

*gcc* client.c -o client

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

*do*

printf("Can't create second channel\n"); exit(1);

}

res = mknod("channel2",S\_IFIFO | 0666,0);

*if*(res) {

printf("Can't create first channel\n"); exit(1);

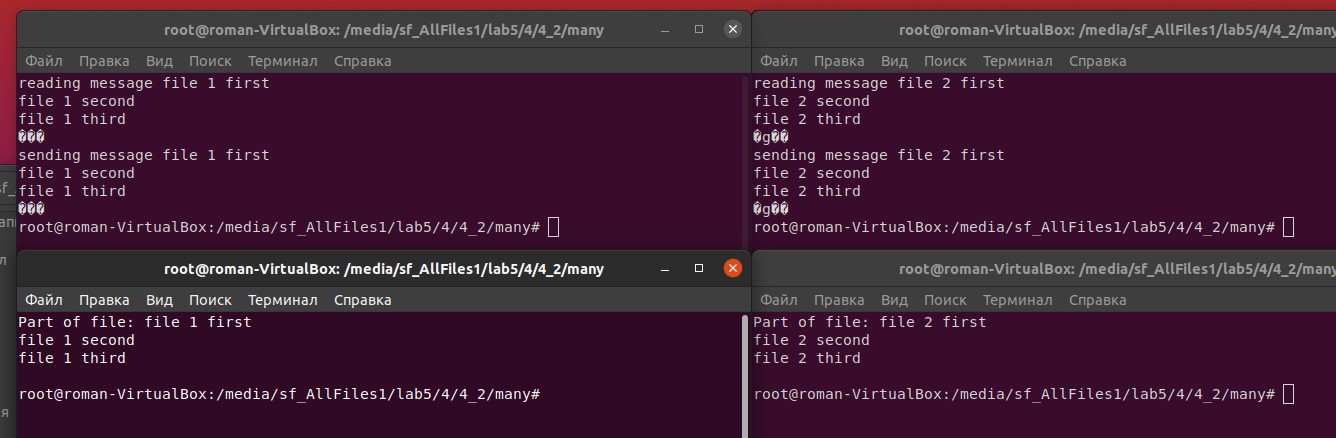
*int* res = mknod("channel1",S\_IFIFO | 0666,0);

*if*(res) {

*sleep* 1

*done*

Результат работы:



4.3. Был изменен server.c. Код server.c:

strcpy(fileName,argv[1]);

*// создаем два канала*

}

*else*

printf("Using default file name '%s'\n",DEF\_FILENAME); strcpy(fileName,DEF\_FILENAME);

*char* fileName[30];

*if*(argc < 2) {

#include <stdio.h> #include <stdlib.h> #include <unistd.h> #include <string.h> #include <sys/types.h> #include <sys/stat.h> #include <fcntl.h>

#define DEF\_FILENAME "testFile.txt"

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

*// открываем второй канал для чтения*

}

printf("Can't open channel for writing\n"); exit(0);

*int* chan1 = open("channel1",O\_WRONLY);

*if*(chan1 == -1) {

*// открываем первый канал для записи*

}

write(chan1,fileName,strlen(fileName));

*// читаем содержимое файла из второго канала*

*// пишем имя файла в первый канал*

*int* chan2 = open("channel2",O\_RDONLY);

*if*(chan2 == -1) {

printf("Can't open channe2 for reading\n"); exit(0);

}

}

*char* buf [100];

printf("Waiting for clint write to channnel\n"); getchar();

*for*(;;) {

bzero(buf,100);

res = read(chan2,buf,100);

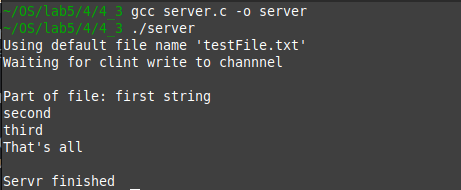
*if*(res <= 0)

*break*;

printf("Part of file: %s\n", buf);

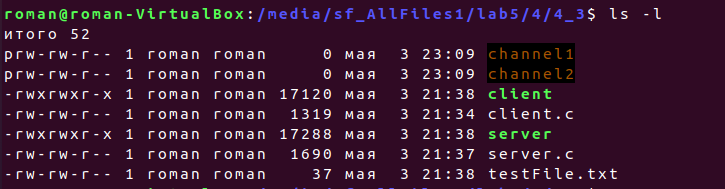
}

close(chan1); close(chan2); unlink("channel1"); unlink("channel2"); printf("Servr finished\n"); *return* 0;

Вывод:



Размер файловой системы



Видим, что размер каналов не меняется, так как они работают через ядро системы машины, а не хранят в себе информацию.

# 5.0 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: Код client.c:

#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;

}

Код server.c:

#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);

}

}

*return* 0;

}

printf("error while sending msg\n"); kill(getpid(),SIGINT);

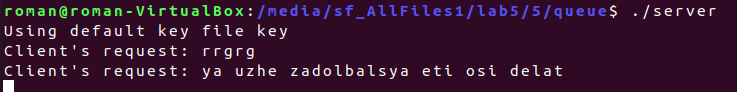
mes.type = 2L; bzero(mes.buf,100); strcpy(mes.buf,"OK");

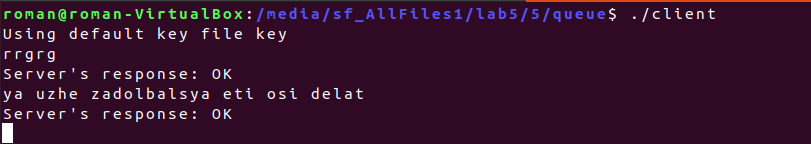
res = msgsnd(queue,(*void*\*)&mes,*sizeof*(Message),0);

*if*(res != 0) {

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

Вывод:





Описание работы сервера: Сервер получает ключ, по имени файла. С помощью ключа и идентификатора = 'Q' получает очередь сообщений и ждет

сообщений с типом 1 от клиентов. При получении сообщения сервер выводитего на экран и отсылает обратное сообщение с типом 2, содержащее фразу

«ОК».

Описание работы клиента: Клиент получает ту же очередь, что и сервер и ждет ввода пользователя. Считав ввод, он шлет сообщение с типом 1, содержащее считанные данные и ожидает от сервера подтверждения о принятии.

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. Чтениепотока 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.

*Script\_many.sh*

key=$i

# Запуск сервера

*gnome-terminal* -- bash -c "./server $key; exec bash"

*sleep* 1

# Запуск клиента

*gnome-terminal* -- bash -c "./client $key $i; exec bash"

*sleep* 1

*done*

*ny.sh*

#!/bin/bash

*gcc* -o server server.c

*gcc* -o client client.c

# Создание 3 пар сервер-клиент

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

*do*

*Script.sh*

#!/bin/bash

*gnome-terminal* -- ./client 9000 $i &

*echo* "Started client $i connecting to queue"

*done*

# Запускаем сервер с очередью 9000 *gnome-terminal* -- ./server 9000 & *echo* "Started server on queue"

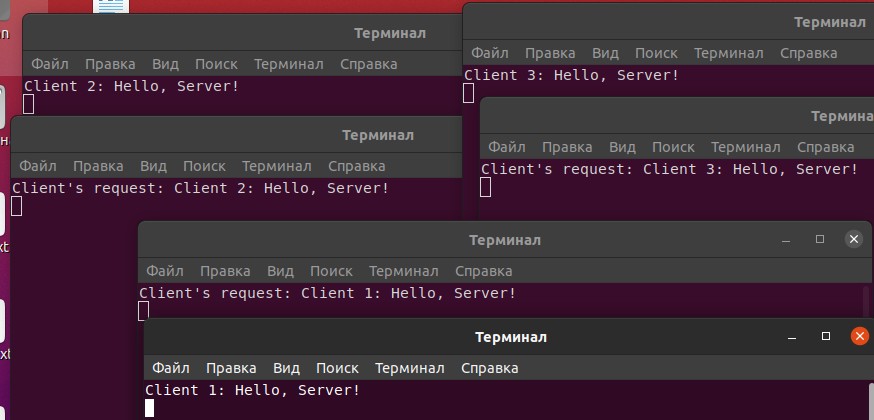
# Даем серверу время на запуск

*sleep* 2

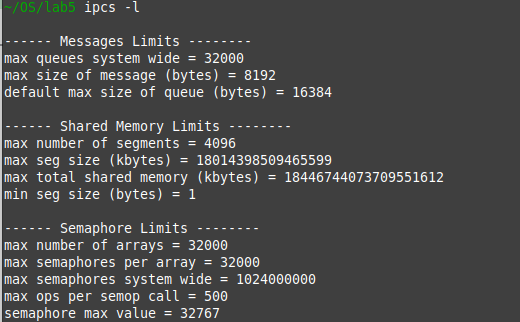
# Запускаем 5 клиентов, каждый подключается к серверу

*for* i *in* {0..4}; *do*

Результат работы:



1. **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.

# 5.1. 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 невозможностьповторного 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 можетбыть 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 одномуиз них смысл «запись разрешена», т.е. читатель предыдущие данные уже использовал; второму – «чтение разрешено», т.е. писатель уже сгенерировал новые данные, которые нужно прочитать.

*server.c*

#include <stdio.h> #include <stdlib.h> #include <signal.h> #include <unistd.h> #include <sys/types.h> #include <sys/ipc.h> #include <sys/sem.h> #include <sys/shm.h> #include <sys/time.h> #include <strings.h> #include <string.h> #include "shm.h"

Message\* p\_msg;

*int* shmemory;

*int* semaphore;

*void* intHandler(*int* sig) {

*//отключаем разделяемую память*

*if*(shmdt(p\_msg) < 0) {

printf("Error while detaching shm\n"); exit(1);

}

*//удаляем shm и семафоры*

*if*(shmctl(shmemory, IPC\_RMID, 0) < 0) { printf("Error while deleting shm\n"); exit(1);

}

*if*(semctl(semaphore, 0, IPC\_RMID) < 0) { printf("Error while deleting semaphore\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;

*//будем использовать 1 и тот же ключ для семафора и для shm*

*if*((key = ftok(keyFile, 'Q')) < 0) {

printf("Can't get key for key file %s and id 'Q'\n",keyFile); exit(1);

}

*//создаем shm*

*if*((shmemory = shmget(key, *sizeof*(Message), IPC\_CREAT | 0666)) < 0) { printf("Can't create shm\n");

exit(1);

}

*//присоединяем shm в наше адресное пространство*

*if*((p\_msg = (Message\*)shmat(shmemory, 0, 0)) < 0) { printf("Error while attaching shm\n");

exit(1);

}

*// устанавливаем обработчик сигнала*

signal(SIGINT, intHandler);

*//создаем группу из 2 семафоров*

*//1 - показывает, что можно читать*

*//2 - показывает, что можно писать*

*if*((semaphore = semget(key, 2, IPC\_CREAT | 0666)) < 0) { printf("Error while creating semaphore\n"); kill(getpid(),SIGINT);

}

*// устнавливаем 2 семафор в 1, т.е. можно писать*

*if*(semop(semaphore, setWriteEna, 1) < 0) {

*for*(;;) {

*// ждем пока клиент начнет работу*

*// основной цикл работы*

}

printf("execution complete\n"); kill(getpid(),SIGINT);

}

}

*if*(semop(semaphore, readEna, 1) < 0) {

printf("execution complete\n"); kill(getpid(),SIGINT);

}

*//читаем сообщение от клиента*

printf("Client's message: %s", p\_msg->buf);

*// говорим клиенту, что можно снова писать*

*if*(semop(semaphore, setWriteEna, 1) < 0) { printf("execution complete\n"); kill(getpid(),SIGINT);

}

*client.c*

|  |  |  |
| --- | --- | --- |
| #include <stdio.h> #include <stdlib.h> #include <signal.h> #include <unistd.h> #include <sys/types.h> #include <sys/ipc.h> #include <sys/sem.h> #include <sys/shm.h> #include <sys/time.h> #include <strings.h> #include <string.h> #include "shm.h"  *int* main(*int* argc, *char*\*\* argv) { | | |
|  | Message\* p\_msg; *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;  *int* shmemory;  *int* semaphore; | |
| *//будем использовать 1 и тот же ключ для семафора и для shm* | | |
|  | *if*((key = ftok(keyFile, 'Q')) < 0) { | |
|  | printf("Can't get key for key file %s and id 'Q'\n",keyFile);  exit(1); |
|  |
| }  *//создаем shm*  *if*((shmemory = shmget(key, *sizeof*(Message), 0666)) < 0) { | |
|  | printf("Can't create shm\n"); exit(1); |
|  |
| } | |

}

}

printf("Error while detaching shm\n"); exit(1);

}

*//отключение от области разделяемой памяти*

*if*(shmdt(p\_msg) < 0) {

}

printf("Can't execute a operation\n"); exit(11);

}

*//запись сообщения в разделяемую память*

sprintf(p\_msg->buf,"%s", buf);

*//говорим серверу, что он может читать*

*if*(semop(semaphore, setReadEna, 1) < 0) {

printf("Can't execute a operation\n"); exit(1);

}

*//хотим отправить сообщение*

*if*(semop(semaphore, writeEna, 1) < 0) {

printf("bye-bye\n"); exit(0);

bzero(buf,100);

printf("Type message to serever. Empty string to finish\n"); fgets(buf,100,stdin);

*if*(strlen(buf) == 1 && buf[0] == '\n') {

}

*char* buf[100];

*for*(;;) {

printf("Error while creating semaphore\n"); exit(1);

}

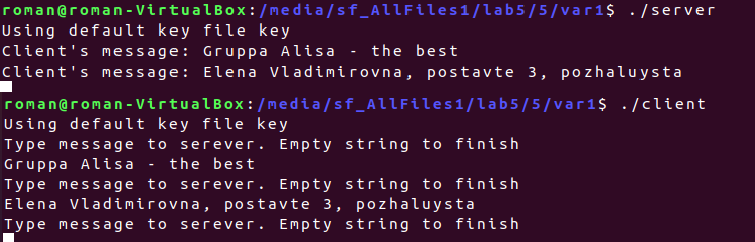
*if*((semaphore = semget(key, 2, 0666)) < 0) {

printf("Error while attaching shm\n"); exit(1);

*//присоединяем shm в наше адресное пространство*

*if*((p\_msg = (Message\*)shmat(shmemory, 0, 0)) < 0) {

Результат работы программы:



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.

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

*script.sh*

#!/bin/bash

# Количество читателей и писателей

num\_readers=2 num\_writers=2

# Запуск читателей

*for* ((i=0; i*<*$num\_readers; i++)); *do gnome-terminal* -- ./server

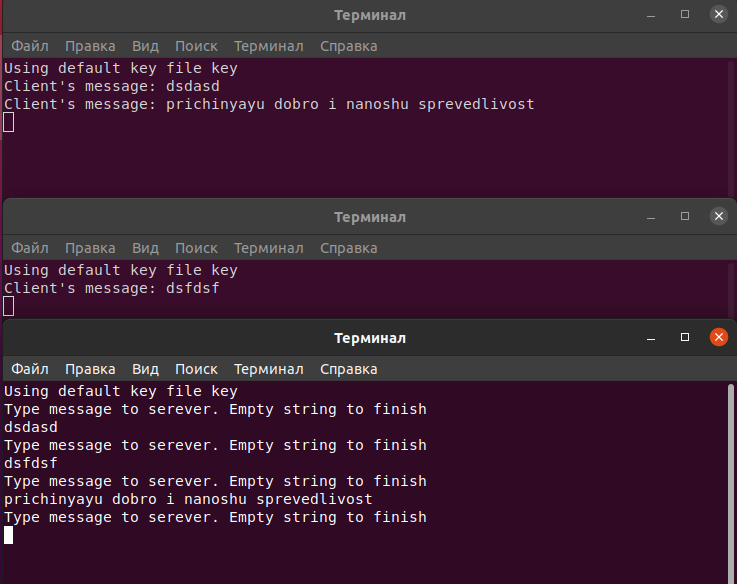
*done*

# Запуск писателей

*for* ((i=0; i*<*$num\_writers; i++)); *do gnome-terminal* -- ./client

*done*

Результат работы:



# 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.

*server.c*

|  |  |  |
| --- | --- | --- |
| #include <sys/types.h> #include <sys/socket.h> #include <netinet/in.h> #include <arpa/inet.h> #include <stdio.h> #include <stdlib.h> #include <string.h> #include <unistd.h>  #define DEF\_PORT 8888  #define DEF\_IP "127.0.0.1"  *int* main(*int* argc, *char* \*\*argv) { | | |
|  | *char* \*addr; | |
| *int* port; | |
| *char* \*readbuf; | |
| printf("Using default port %d\n", DEF\_PORT); | |
| port = DEF\_PORT; | |
| printf("Using default addr %s\n", DEF\_IP); | |
| addr = DEF\_IP; | |
| *// создаем сокет* | |
| *struct* sockaddr\_in peer; | |
| peer.sin\_family = AF\_INET; | |
| peer.sin\_port = htons(port); | |
| peer.sin\_addr.s\_addr = inet\_addr(addr); | |
| *int* sock = socket(AF\_INET, SOCK\_STREAM, 0); | |
| *if* (sock < 0) { | |
|  | perror("Can't create socket\n"); |
|  | exit(1); |
| } | |
| *// присоединяемся к серверу* | |
| *int* res = connect(sock, (*struct* sockaddr \*)&peer, *sizeof*(peer)); | |
| *if* (res) { | |

perror("Can't connect to server:"); exit(1);

}

*// основной цикл программы*

*char* buf[100]; *int* first\_msg = 1; *for* (;;) {

printf("Input request (empty to exit)\n");

*if* (first\_msg == 0){ bzero(buf, 100);

fgets(buf, 100, stdin); buf[strlen(buf) - 1] = '\0';

}

*else*{

strcpy(buf, argv[1]); buf[strlen(buf)] = '\0'; first\_msg = 0;

}

*if* (strlen(buf) == 0) { printf("Bye-bye\n"); *return* 0;

}

res = sendFix(sock, buf, 0);

*if* (res <= 0) {

perror("Error while sending:"); exit(1);

}

bzero(buf, 100);

res = readFix(sock, buf, 100, 0);

*if* (res <= 0) {

perror("Error while receiving:"); exit(1);

}

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

}

*return* 0;

}

*int* readFix(*int* sock, *char* \*buf, *int* bufSize, *int* flags) {

*// читаем "заголовок" - сколько байт составляет наше сообщение*

*unsigned* msgLength = 0;

*int* res = recv(sock, &msgLength, *sizeof*(*unsigned*), flags | MSG\_WAITALL);

*if* (res <= 0) *return* res;

*if* (res > bufSize) {

printf("Recieved more data, then we can store, exiting\n"); exit(1);

}

*// читаем само сообщение*

*return* recv(sock, buf, msgLength, flags | MSG\_WAITALL);

}

*int* sendFix(*int* sock, *char* \*buf, *int* flags) {

*// число байт в сообщении*

*unsigned* msgLength = strlen(buf);

*int* res = send(sock, &msgLength, *sizeof*(*unsigned*), flags);

*if* (res <= 0) *return* res;

send(sock, buf, msgLength, flags);

}

*Client.c*

#include <sys/types.h> #include <sys/socket.h> #include <netinet/in.h> #include <arpa/inet.h> #include <stdio.h> #include <stdlib.h> #include <string.h> #include <unistd.h>

#define DEF\_PORT 8888

#define DEF\_IP "127.0.0.1"

*// обработка одного клиента*

*void* \*clientHandler(*void* \*args)

{

*int* sock = (*int*)args;

*char* buf[100];

*int* res = 0;

*for* (;;)

{

bzero(buf, 100);

res = readFix(sock, buf, 100, 0);

*if* (res <= 0)

{

perror("Can't recv data from client, ending thread\n"); pthread\_exit(NULL);

}

printf("Some client sent: %s\n", buf); res = sendFix(sock, buf, 0);

*if* (res <= 0)

{

perror("send call failed"); pthread\_exit(NULL);

}

}

}

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

{

*int* port = 0;

*if* (argc < 2)

{

printf("Using default port %d\n", DEF\_PORT); port = DEF\_PORT;

}

*else*

port = atoi(argv[1]);

*struct* sockaddr\_in listenerInfo; listenerInfo.sin\_family = AF\_INET; listenerInfo.sin\_port = htons(port); listenerInfo.sin\_addr.s\_addr = htonl(INADDR\_ANY);

*int* listener = socket(AF\_INET, SOCK\_STREAM, 0);

*if* (listener < 0)

{

perror("Can't create socket to listen: ");

exit(1);

}

*int* res = bind(listener, (*struct* sockaddr \*)&listenerInfo, *sizeof*(listenerInfo)); *if* (res < 0)

{

perror("Can't bind socket"); exit(1);

}

*// слушаем входящие соединения*

res = listen(listener, 5);

*if* (res)

{

perror("Error while listening:"); exit(1);

}

*// основной цикл работы*

*for* (;;)

{

*int* client = accept(listener, NULL, NULL); pthread\_t thrd;

res = pthread\_create(&thrd, NULL, clientHandler, (*void* \*)(client));

*if* (res)

{

printf("Error while creating new thread\n");

}

}

*return* 0;

}

*int* readFix(*int* sock, *char* \*buf, *int* bufSize, *int* flags)

{

*// читаем "заголовок" - сколько байт составляет наше сообщение*

*unsigned* msgLength = 0;

*int* res = recv(sock, &msgLength, *sizeof*(*unsigned*), flags | MSG\_WAITALL);

*if* (res <= 0)

*return* res;

*if* (res > bufSize)

{

printf("Recieved more data, then we can store, exiting\n"); exit(1);

}

*// читаем само сообщение*

*return* recv(sock, buf, msgLength, flags | MSG\_WAITALL);

}

*int* sendFix(*int* sock, *char* \*buf, *int* flags)

{

*// шлем число байт в сообщении*

*unsigned* msgLength = strlen(buf);

*int* res = send(sock, &msgLength, *sizeof*(*unsigned*), flags);

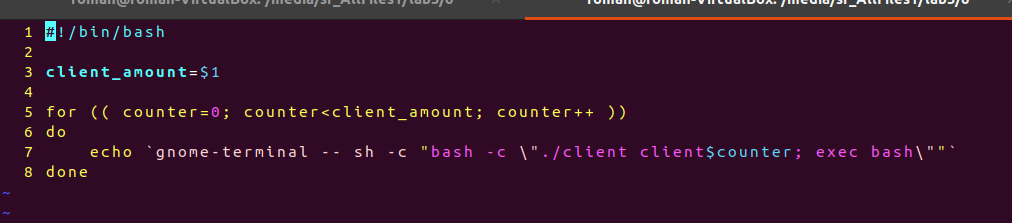
*if* (res <= 0)

*return* res;

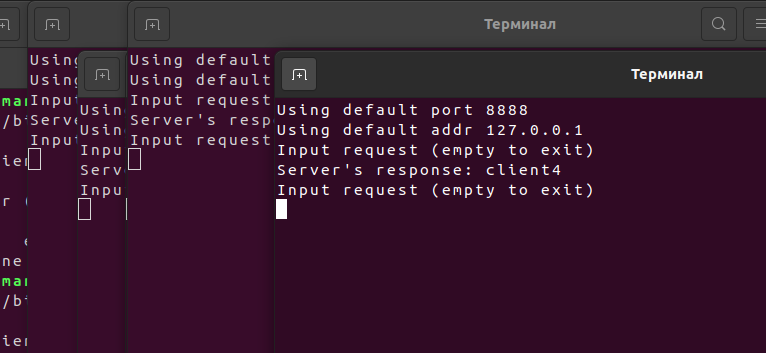
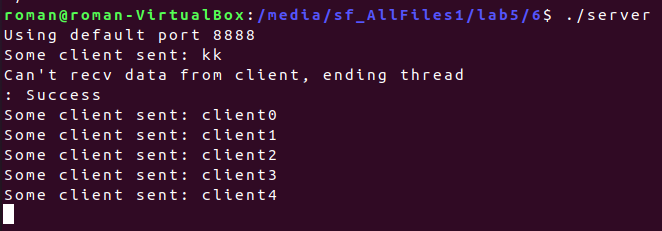
send(sock, buf, msgLength, flags);

}

*script.sh*



*Результат работы программы:*



Можно заметить, что программа работает без ошибок. В файле server.c создается сокет, который привязывается к определенному порту, после чего начинает прослушивать входящие соединения. Для каждого клиента, который подключается, создается отдельный поток, который принимает сообщения от клиента, выводит их на экран и отправляет обратно тот же текст. Функции

sendFix и readFix используются для передачи сообщений с учетом правил протокола TCP, включая размер сообщения в заголовок. В клиентском коде (client.c) также создается сокет, который подключается к серверу, указав IP- адрес и порт. Затем в цикле пользователь вводит сообщения, которые отправляются на сервер с помощью функции send, и в ответ приходит сообщение от сервера, которое выводится на экран.

Для того чтобы реализовать подобное взаимодействие с использованием протокола UDP, необходимо изменить тип сокета с SOCK\_STREAM на SOCK\_DGRAM, а также заменить функции recv() и send() на recvfrom() и sendto() соответственно.

Былы написан скрипты, создающие множество серверов для множества клиентов.

*Script1.sh*

port=*$*((9000 + i))

*gnome-terminal* -- ./server $port &

*echo* "Started server $i on port $port"

#!/bin/bash

# Запускаем серверы на портах 9000-9004

*for* i *in* {0..4}; *do*

port=*$*((9000 + i))

*gnome-terminal* -- ./client 127.0.0.1 $port "Hello, Server $i!" &

*echo* "Started client $i connecting to port $port"

*done*

*done*

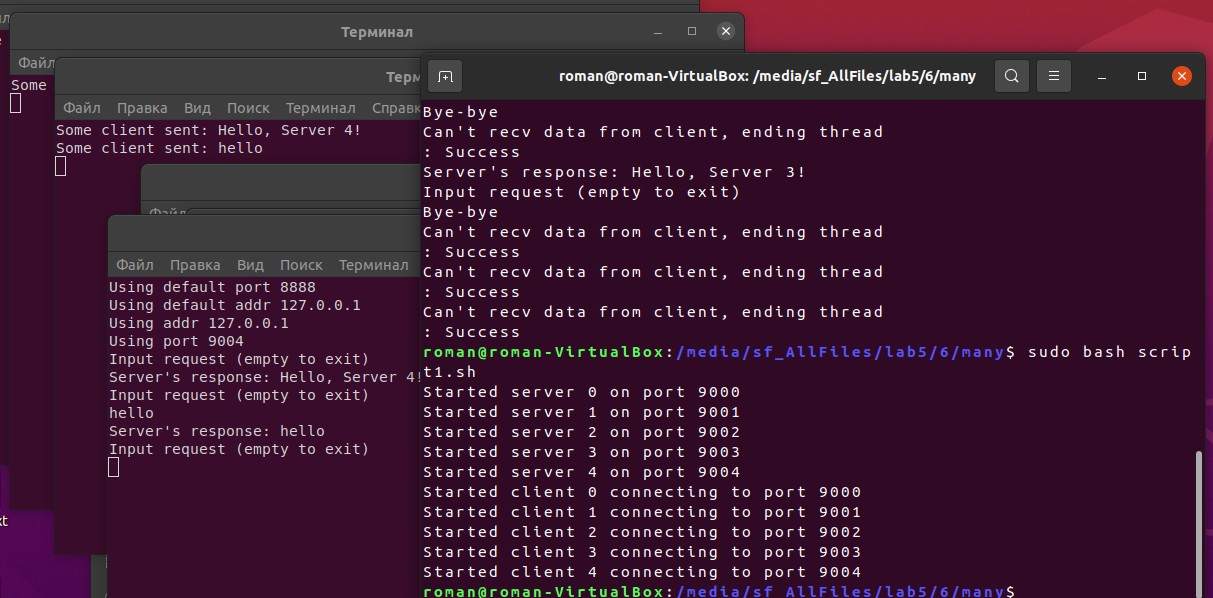
# Даем серверам время на запуск

*sleep* 2

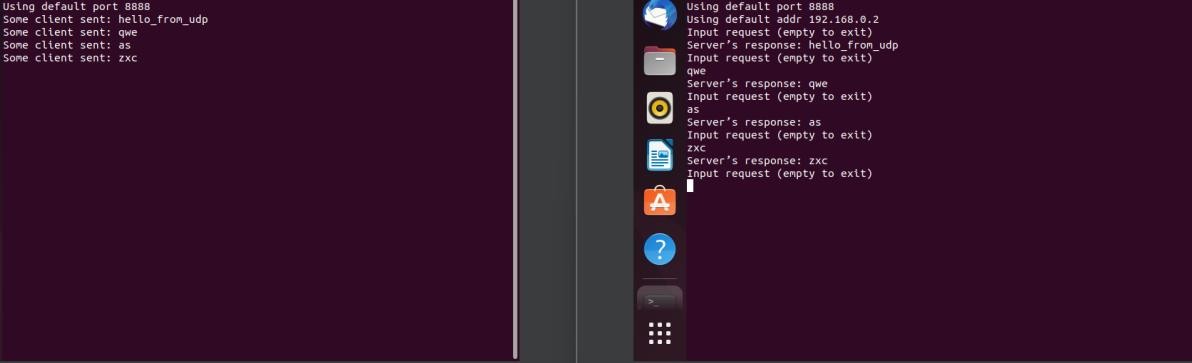
# Запускаем клиентов, каждый подключается к своему серверу

*for* i *in* {0..4}; *do*

Результат работы:



Также на двух машинах была настроена одна сеть и запущены клиент и сервер c адресами 192.168.0.1/24 и 192.168.0.2/24 соответственно, обменивающиеся данными через *UDP.*



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.

# Выводы.

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 литературы.

* W. Richard Stevens, Stephen A. Rago. UNIX Network Programming, Volume 2: Interprocess Communications (Second Edition). Prentice Hall, 1999.
* Michael Kerrisk. The Linux Programming Interface: A Linux and UNIX System Programming Handbook. No Starch Press, 2010.
* Douglas Comer. Internetworking with TCP/IP, Volume III: Client-Server Programming and Applications, Linux/Posix Sockets Version (Third Edition). Prentice Hall, 2000.
* Brian W. Kernighan, Rob Pike. The Unix Programming Environment (Second Edition). Prentice Hall, 1984.
* Andrew S. Tanenbaum, Herbert Bos. Modern Operating Systems (Fourth Edition). Pearson Education, 2015.
* Maurice J. Bach. The Design of the Unix Operating System. Prentice Hall, 1986.
* William Stallings. Operating Systems: Internals and Design Principles (Ninth Edition). Pearson Education, 2018.