

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ
по лабораторным работам №5, 6
по дисциплине «Операционные системы»
Тема: Средства межпроцессного взаимодействия в ОС семейства Unix

Студентка гр. 1381

Рымарь М.И.

Преподаватель

Душутина Е.В.

Санкт-Петербург

2023

Цель работы.

Изучить на примере ОС Linux такие средства межпроцессного взаимодействия, как сигналы, неименованные каналы, именованные каналы, очереди сообщений, семафоры, разделяемая память, сокеты.

Задание.

Изучить работу со следующими средствами межпроцессного взаимодействия:

1. Сигналы
 - 1.1. Ненадежные сигналы
 - 1.2. Надежные сигналы
 - 1.3. Сигналы реального времени
2. Неименованные каналы
3. Именованные каналы
4. Очереди сообщений
5. Семафоры и разделяемая память
6. Сокеты

Выполнение работы.

1.1. Изучим работу ненадёжных сигналов. Для начала напишем программу, которая обрабатывает пользовательские сигналы SIGUSR1 и SIGUSR2, реагирует по умолчанию на сигнал SIGINT, игнорирует сигнал SIGCHLD. Породим процесс-копию и уйдём в ожидание сигнала. В самой программе с помощью kill отправлен сигнал SIGUSR1. Отправим два сигнала SIGUSR2. Первый сигнал обработается написанным обработчиком, второй сработает по умолчанию после восстановления обработчика. Ниже приведён код task1_1.c, результаты работы программы представлены на рисунках 1-2. После восстановления обработчика сигналов, как и ожидалось, был вызван обработчик по умолчанию.

Программа обработки сигнала task1_1.c:

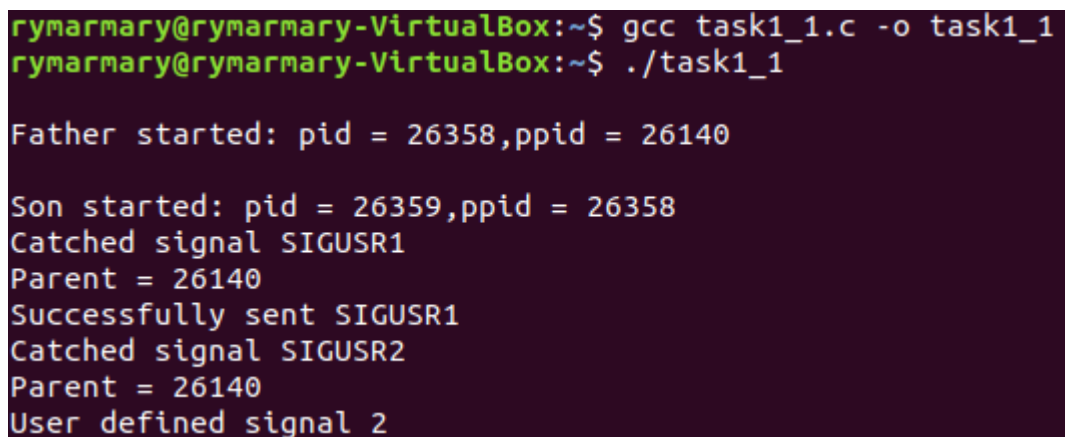
```

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

static void sigHandler(int sig) {
    printf("Caught signal %s\n", sig == SIGUSR1 ? "SIGUSR1" : "SIGUSR2");
    printf("Parent = %d\n", getppid());
    // восстанавливаем 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);
    for (;;) {
        pause();
    }
    return 0;
}

```



```

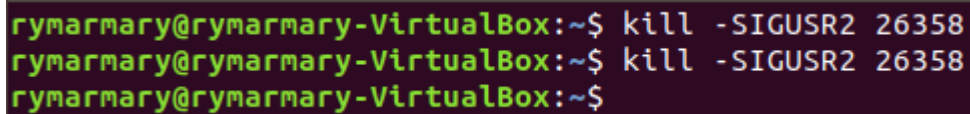
rymarmary@rymarmary-VirtualBox:~$ gcc task1_1.c -o task1_1
rymarmary@rymarmary-VirtualBox:~$ ./task1_1

Father started: pid = 26358,ppid = 26140

Son started: pid = 26359,ppid = 26358
Caught signal SIGUSR1
Parent = 26140
Successfully sent SIGUSR1
Caught signal SIGUSR2
Parent = 26140
User defined signal 2

```

Рисунок 1 – Результат выполнения программы task1_1.c



```
gymarmary@gymarmary-VirtualBox:~$ kill -SIGUSR2 26358
gymarmary@gymarmary-VirtualBox:~$ kill -SIGUSR2 26358
gymarmary@gymarmary-VirtualBox:~$
```

Рисунок 2 – Вызовы сигналов на втором терминале

Повторим эксперимент для другого сигнала. Для примера возьмём сигнал под номером 41, немного модифицируем предыдущую программу следующим образом: вместо SIGUSR1 перехватим и обработаем сигнал под номером 41 (который по умолчанию завершает программу). Зная, что процессы изолированы друг от друга, в том числе у них свои диспозиции сигналов, проверим это, запустив старую task1_1.c и новую версию программы task1_2.c, представленную ниже. И там, и там отправим процессору сигнал с номером 41. Результат приведён на рисунках 3-5. Как и ожидалось, из-за разной диспозиции сигналов программа task1_1.c не обрабатывала сигнал 41, а программа task1_2.c обрабатывала.

Модифицированная программа с обработкой сигнала 41 task1_2.c:

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
#include <wait.h>
#include <sys/types.h>

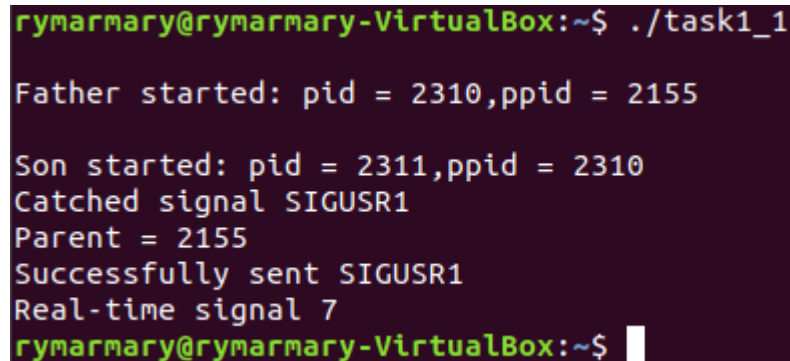
static void sigHandler(int sig) {
    printf("Caught signal %s\n", sig == SIGUSR1 ? "SIGUSR1" : "41");
    printf("Parent = %d\n", getppid());
    // восстанавливаем старую диспозицию
    signal(sig, SIG_DFL);
}

int main() {
    printf("\nFather started: pid = %i, ppid = %i\n", getpid(), getppid());
    signal(41, 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(), SIGUSR2) != 0) {
            printf("Error while sending SIGUSR1\n");
            exit(1);
        }
    }
}
```

```

        printf("Successfully sent SIGUSR2\n");
        return 0;
    }
    // программа родитель
    wait(NULL);
    for (;;) {
        pause();
    }
    return 0;
}

```



```

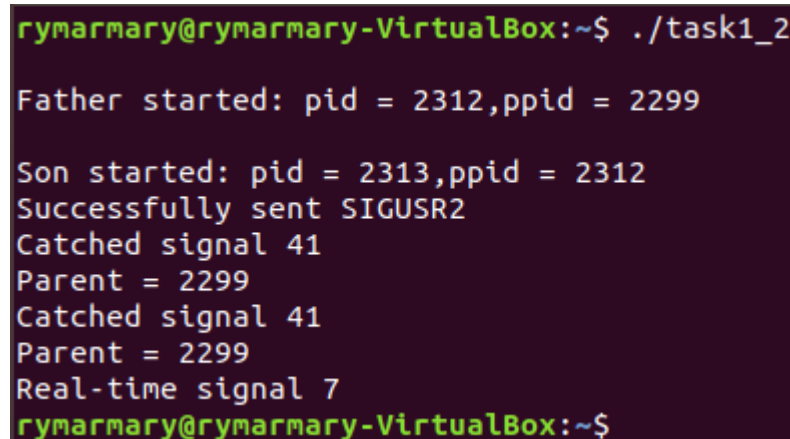
rymary@rymary-VirtualBox:~$ ./task1_1

Father started: pid = 2310,ppid = 2155

Son started: pid = 2311,ppid = 2310
Caught signal SIGUSR1
Parent = 2155
Successfully sent SIGUSR1
Real-time signal 7
rymary@rymary-VirtualBox:~$

```

Рисунок 3 – Работа программы task1_1.c



```

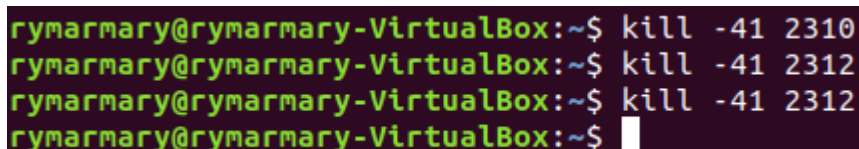
rymary@rymary-VirtualBox:~$ ./task1_2

Father started: pid = 2312,ppid = 2299

Son started: pid = 2313,ppid = 2312
Successfully sent SIGUSR2
Caught signal 41
Parent = 2299
Caught signal 41
Parent = 2299
Real-time signal 7
rymary@rymary-VirtualBox:~$

```

Рисунок 4 – Работа программы task1_2.c



```

rymary@rymary-VirtualBox:~$ kill -41 2310
rymary@rymary-VirtualBox:~$ kill -41 2312
rymary@rymary-VirtualBox:~$ kill -41 2312
rymary@rymary-VirtualBox:~$

```

Рисунок 5 – Вызовы сигналов на дополнительном терминале

Рассмотрим следующий эксперимент: напишем программы task1_3.c и task1_4.c таким образом, что в первой программе узнаём pid процесса, записываем его в файл и переходим в ожидание сигналов (при этом написан обработчик для SIGUSR2). После её запуска откроется файл во второй программе, куда и записали ранее pid процесса прошлой программы. Далее

соберётся строка и произведётся системный вызов, в результате которого в первой программе будет вызван сигнал SIGUSR2. Результат работы представлен на рисунках 6-7. Таким образом, был проведён эксперимент для процессов, порождаемых в разных файлах. Сигнал обрабатывается точно так же, как и в случае с родственными процессами, как и ожидалось. То есть сначала сигнал обрабатывается sighandler, потом восстанавливается диспозиция и программа завершается при второй посылке сигнала.

Программа, ожидающая сигнал, task1_3.c:

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
#include <wait.h>
#include <sys/types.h>
#include <string.h>

static void sigHandler(int sig) {
    printf("Caught signal %s\n", sig == SIGUSR1 ? "SIGUSR1" : "SIGUSR2");
    printf("Parent = %d\n", getppid());
    // восстанавливаем старую диспозицию
    signal(sig, SIG_DFL);
}

int main() {
    signal(SIGUSR2, sigHandler);
    FILE* fp;
    fp = fopen("pid_file.txt", "w");
    char string[100];
    if (!fp) {
        perror("FILE!");
        exit(1);
    }
    int father_pid = getpid();
    sprintf(string, "%d", father_pid);
    printf("\nFather started: pid = %i\n", father_pid);
    fprintf(fp, string);
    fclose(fp);
    // Ждём сигналов
    for (;;) {
        pause();
    }
    return 0;
}
```

Программа, посылающая сигнал, task1_4.c:

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

```

#include <stdlib.h>
#include <wait.h>
#include <sys/types.h>
#include <string.h>

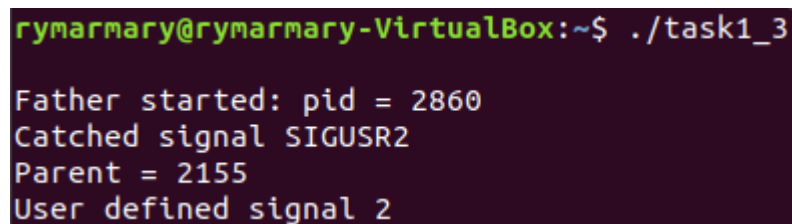
static void sigHandler(int sig) {
    printf("Caught signal %s\n", sig == SIGUSR1 ? "SIGUSR1" : "SIGUSR2");
    printf("Parent = %d\n", getppid());
    // восстанавливаем старую диспозицию
    signal(sig, SIG_DFL);
}

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("pid_file.txt", "r");
    char string[100];
    if (!fp) {
        perror("FILE!");
        exit(1);
    }
    fgets(string, 100, fp);
    char* s = concat("kill -12 ", string);
    system(s);

    fclose(fp);
    free(s);
    return 0;
}

```



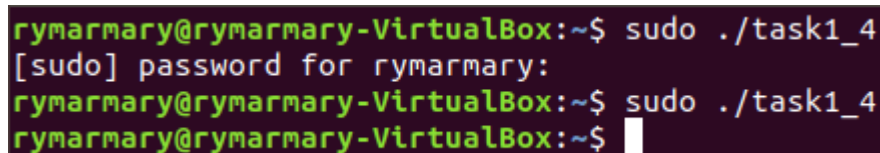
```

rymary@rymary-VirtualBox:~$ ./task1_3

Father started: pid = 2860
Caught signal SIGUSR2
Parent = 2155
User defined signal 2

```

Рисунок 6 – Результат обработки неродственных процессов



```

rymary@rymary-VirtualBox:~$ sudo ./task1_4
[sudo] password for rymary:
rymary@rymary-VirtualBox:~$ sudo ./task1_4
rymary@rymary-VirtualBox:~$ █

```

Рисунок 7 – Дополнительный терминал для работы с task1_3.c и task1_4.c

Рассмотрим следующий эксперимент: обработка 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 станет в

ожидание или не будет послан напрямую. Возвращает номер сигнала в sig. Напишем следующую программу: обработаем SIGUSR1 и SIGUSR2 так, чтобы на посыл SIGUSR1 остановим поток, а SIGUSR2 проигнорируем. Программный код task1_5.c представлен ниже. Результат работы представлен в рисунках 8-9. Как и ожидалось, сигналы обрабатываются согласно определёнными нами обработчиками, а на отправке SIGUSR1 (сигнал 10) поток завершает свою работу.

Программа с обработкой сигнала потоком task1_5.c:

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
#include <wait.h>
#include <sys/types.h>
#include <string.h>
#include <pthread.h>
#include <sys/syscall.h>

#define _GNU_SOURCE

pthread_t thread1;
sigset_t set;
int thread_id;
void *thread_func(void *args) {
    thread_id = syscall(SYS_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) {

}

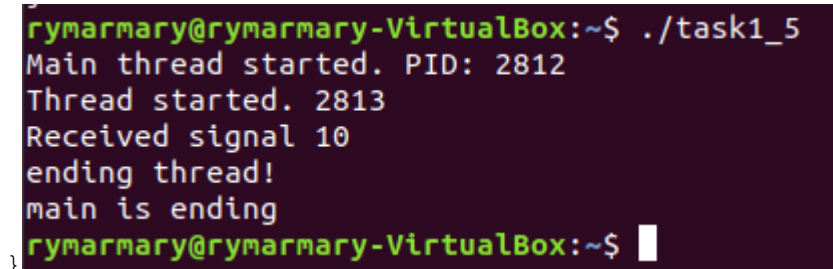
int main() {
    signal(SIGUSR1, sig_handler);
    signal(SIGUSR2, SIG_IGN);
    sigemptyset(&set);
    sigaddset(&set, SIGTERM);
    sigaddset(&set, SIGINT);
    sigaddset(&set, SIGUSR1);
```



```

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;
}

```

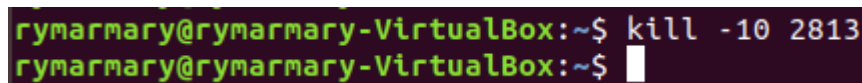


```

rymary@rymary-VirtualBox:~$ ./task1_5
Main thread started. PID: 2812
Thread started. 2813
Received signal 10
ending thread!
main is ending
rymary@rymary-VirtualBox:~$

```

Рисунок 8 – Результат работы task1_5.c



```

rymary@rymary-VirtualBox:~$ kill -10 2813
rymary@rymary-VirtualBox:~$

```

Рисунок 9 – Дополнительный терминал

В отличие от работы с процессами, в случае с потоками нельзя завершить работу одного потока из другой программы с помощью `pthread_kill(pthread_t)`, по причине того, что `pthread_t` хранится в другом адресном пространстве. Это можно обойти посредством посылки сигнала из одной программы во вторую. Модифицируем программы так: программа `task1_6.c` создаёт поток и ожидает его завершения, а программа `task1_7` отправит потоку сигнал `SIGUSR1`, в результате которого поток и после него процесс завершатся. Программный код представлен ниже. Результат работы программы показан на рисунках 10-11.

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

Код программы, создающей поток, `task1_6.c`:

```

#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
#include <wait.h>
#include <sys/types.h>
#include <string.h>
#include <pthread.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 = 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;
}

```

Код программы, отправляющей сигнал, task1_7.c:

```

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

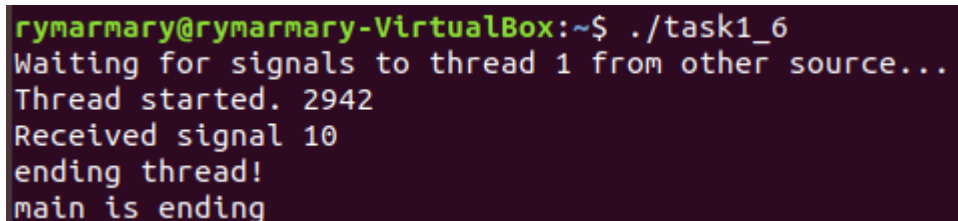
```

```
#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;
}

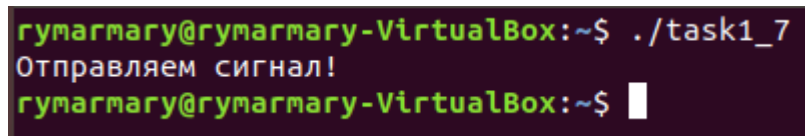
int main() {
    FILE* fp;
    fp = fopen("tid_file.txt", "r");
    char str[100];
    if (fp == NULL){
        perror("FILE");
        exit(1);
    }
    fgets(str, 100, fp);
    char* s = concat("kill -10 ", str); //10 == SIGUSR1
    printf("Отправляем сигнал!\n");
    system(s);
    fclose(fp);
    free(s);
    // ждем сигналов

    return 0;
}
```



```
rymarmary@rymarmary-VirtualBox:~$ ./task1_6
Waiting for signals to thread 1 from other source...
Thread started. 2942
Received signal 10
ending thread!
main is ending
```

Рисунок 10 – Результат работы программы task1_6.c



```
rymarmary@rymarmary-VirtualBox:~$ ./task1_7
Отправляем сигнал!
rymarmary@rymarmary-VirtualBox:~$
```

Рисунок 11 – Дополнительный терминал

1.2. Изучим работу надёжных сигналов. Создадим программу, позволяющую продемонстрировать возможность отложенной обработки (временного блокирования) сигнала SIGINT. Реализуем блокировку, вызвав «засыпание» процесса на одну минуту из обработчика пользовательских сигналов. В основной программе устанавливаем диспозицию этих сигналов. С рабочего терминала отправим процессу sigact сигнал SIGUSR1, а затем SIGINT.

Программный код task1_8.c представлен ниже. Результат работы программы показан на рисунке 12.

По результатам сигнал SIGUSR1 принят корректно, но после послыки 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 завершения обработки сигналов, которые отложили обработку. Механизм ненадёжных сигналов не позволяет откладывать обработку других сигналов (можно лишь установить игнорирование некоторых сигналов на время обработки).

Код программы, демонстрирующей отложенную обработку сигнала, task1_8.c:

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

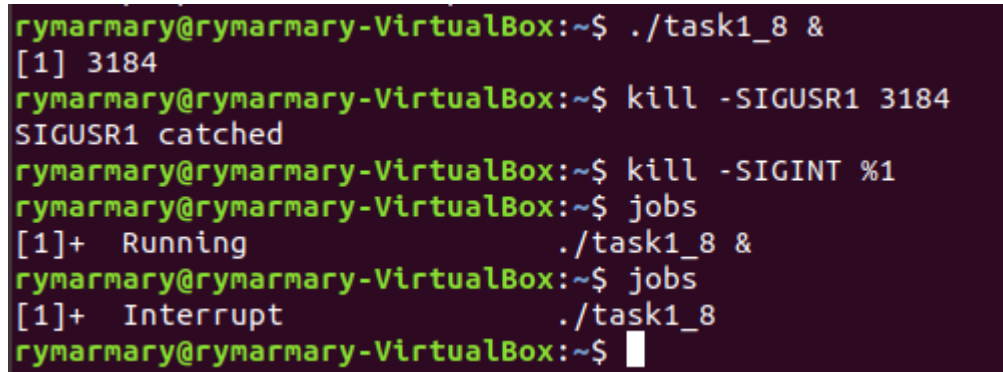
void (*mysig(int sig, void (*hnd)(int)))(int)
{
    // надежная обработка сигналов
    struct sigaction act, oldact;
    act.sa_handler = hnd;
    sigemptyset(&act.sa_mask);
    sigaddset(&act.sa_mask, SIGINT);
    act.sa_flags = 0;
    if (sigaction(sig, &act, 0) < 0)
        return SIG_ERR;
    return act.sa_handler;
}

void hndUSR1(int sig)
{
    if (sig != SIGUSR1)
    {
        printf("Caught bad signal %d\n", sig);
        return;
    }
    printf("SIGUSR1 caught\n");
    sleep(10);
}
```

```

int main()
{
    mysig(SIGUSR1, hndUSR1);
    for (;;)
    {
        pause();
    }
    return 0;
}

```



```

rymarmy@rymarmy-VirtualBox:~$ ./task1_8 &
[1] 3184
rymarmy@rymarmy-VirtualBox:~$ kill -SIGUSR1 3184
SIGUSR1 caught
rymarmy@rymarmy-VirtualBox:~$ kill -SIGINT %1
rymarmy@rymarmy-VirtualBox:~$ jobs
[1]+  Running                  ./task1_8 &
rymarmy@rymarmy-VirtualBox:~$ jobs
[1]+  Interrupt                 ./task1_8
rymarmy@rymarmy-VirtualBox:~$

```

Рисунок 12 – Результат работы программы task1_8.c

Изменим обработчик сигнала таким образом, чтобы из него производилась отправка другого сигнала. Пусть из обработчика сигнала SIGUSR1 функцией kill() генерируется сигнал SIGINT. Проанализируем наличие и очередность обработки сигналов. Программный код task1_9.c представлен ниже. Результат работы программы показан на рисунке 13.

При генерации сигнала (в данном случае SIGINT) из обработчика другого сигнала обработка сгенерированного сигнала задерживается до конца выполнения текущего обработчика (как и в предыдущем эксперименте).

Программа, демонстрирующая отложенную обработку сигнала, вызванного через другой сигнал, task1_9.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);
}

```

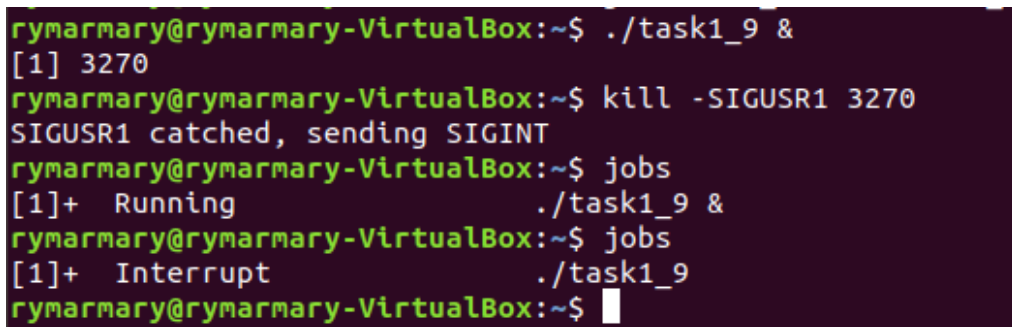
```

    act.sa_flags = 0;
    if (sigaction(sig, &act, 0) < 0)
        return SIG_ERR;
    return act.sa_handler;
}

void hndUSR1(int sig)
{
    if (sig != SIGUSR1)
    {
        printf("Caught bad signal %d\n", sig);
        return;
    }
    printf("SIGUSR1 caught, sending SIGINT\n");
    kill(getpid(), SIGINT);
    sleep(10);
}

int main()
{
    mysig(SIGUSR1, hndUSR1);
    for (;;)
    {
        pause();
    }
    return 0;
}

```



```

rymary@rymary-VirtualBox:~$ ./task1_9 &
[1] 3270
rymary@rymary-VirtualBox:~$ kill -SIGUSR1 3270
SIGUSR1 caught, sending SIGINT
rymary@rymary-VirtualBox:~$ jobs
[1]+  Running                  ./task1_9 &
rymary@rymary-VirtualBox:~$ jobs
[1]+  Interrupt                 ./task1_9
rymary@rymary-VirtualBox:~$

```

Рисунок 13 – Результат работы программы task1_9.c

1.3. Изучим работу сигналов POSIX реального времени. Проведём эксперимент, позволяющий определить возможность организации очереди для различных типов сигналов (обычных и реального времени). При этом увеличим вложенность обработчиков. Из обычных будем использовать сигналы SIGUSR1 и SIGUSR2, а из сигналов реального времени – SIGRTMIN, SIGRTMIN+1, SIGRTMIN+2. 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. Программный код task1_10.c представлен ниже. Результат работы программы показан на рисунке 14.

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

Программа, демонстрирующая организацию очереди сигналов, task1_10.c:

```
#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";
}

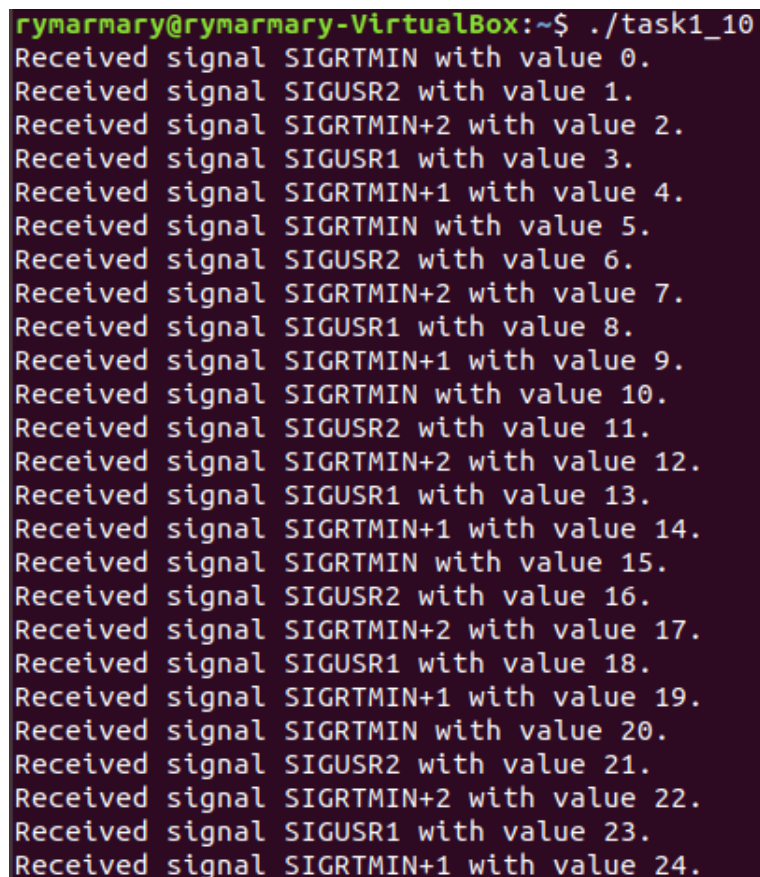
void handler(const int sig, siginfo_t* si, void* ucontext) {
    printf("Received signal %s with value %d.\n",
    switch_int_to_sigstr(sig), si->si_value.sival_int);
    int pid = getpid();
    if (sig == SIGRTMIN){
        sigqueue(pid, SIGUSR2, (union sigval){.sival_int =
signal_counter++});
    }
    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;
}

```



```

rymarmary@rymarmary-VirtualBox:~$ ./task1_10
Received signal SIGRTMIN with value 0.
Received signal SIGUSR2 with value 1.
Received signal SIGRTMIN+2 with value 2.
Received signal SIGUSR1 with value 3.
Received signal SIGRTMIN+1 with value 4.
Received signal SIGRTMIN with value 5.
Received signal SIGUSR2 with value 6.
Received signal SIGRTMIN+2 with value 7.
Received signal SIGUSR1 with value 8.
Received signal SIGRTMIN+1 with value 9.
Received signal SIGRTMIN with value 10.
Received signal SIGUSR2 with value 11.
Received signal SIGRTMIN+2 with value 12.
Received signal SIGUSR1 with value 13.
Received signal SIGRTMIN+1 with value 14.
Received signal SIGRTMIN with value 15.
Received signal SIGUSR2 with value 16.
Received signal SIGRTMIN+2 with value 17.
Received signal SIGUSR1 with value 18.
Received signal SIGRTMIN+1 with value 19.
Received signal SIGRTMIN with value 20.
Received signal SIGUSR2 with value 21.
Received signal SIGRTMIN+2 with value 22.
Received signal SIGUSR1 with value 23.
Received signal SIGRTMIN+1 with value 24.

```

Рисунок 14 – Результат работы программы task1_10.c

Следующим экспериментом 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 отправки сигналов напишем скрипт, который 10 раз отправит сигнал SIGRTMIN. Программный код task1_11.c представлен ниже, как и скрипт script1_11.sh. Результат работы программы показан на рисунках 15-16.

Как и ожидалось, обработка равноприоритетных сигналов реального времени происходит в порядке FIFO.

Программа, демонстрирующая обработку равноприоритетных сигналов, task1_11.c:

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#define AMOUNT_OF_SIGNALS 10
static volatile int counter = 0;
int sig_iter = 0;
int counter = 0;
int count = 0;

void handler(int sig, siginfo_t* si, void* ucontext) {
    printf("Received signal %d with value %d.\n", sig, count++);
    sleep(1);
}

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() {
    printf("PID: %d\n", getpid());
    mysig(SIGUSR1, handler);
    mysig(SIGRTMIN, handler);
    for (;;) {
        pause();
    }
}
```

```

    }
    return 0;
}

```

Скрипт, отправляющий сигналы, script1_11.sh:

```

#!/bin/bash
signal_val=0
pid=$1
val_counter=0
for (( counter=0; counter<10; counter++ ));
do
echo `sudo kill -34 $pid`
((val_counter=val_counter+1))
done

```

```

rymary@rymary-VirtualBox:~$ ./task1_11
PID: 2743
Received signal 34 with value 0.
Received signal 34 with value 1.
Received signal 34 with value 2.
Received signal 34 with value 3.
Received signal 34 with value 4.
Received signal 34 with value 5.
Received signal 34 with value 6.
Received signal 34 with value 7.
Received signal 34 with value 8.
Received signal 34 with value 9.

```

Рисунок 15 – Результат работы task1_11.c

```

rymary@rymary-VirtualBox:~$ sudo bash ./script1_11.sh 2743

```

Рисунок 16 – Дополнительный терминал с запуском скрипта

Опытным путём подтвердим наличие приоритетов сигналов реального времени. Для этого используем прошлую программу, но изменим скрипт, посылающий сигналы. Сначала отправим два простых сигнала SIGUSR1, а потом – 10 сигналов реального времени. Содержание скрипта script1_12.sh представлено ниже. Результат работы программы представлен на рисунке 17-18.

Как видно из результатов, сначала посылается обычный сигнал SIGUSR1, после него в очередь начинают помещаться остальные сигналы. Несмотря на то, что в очередь вторым помещается ещё один сигнал SIGUSR1, сначала обрабатываются сигналы реального времени, помещённые туда позже (SIGRTMIN). Это подтверждает, что сигналы реального времени имеют приоритет над простыми сигналами.

```

#!/bin/bash

```

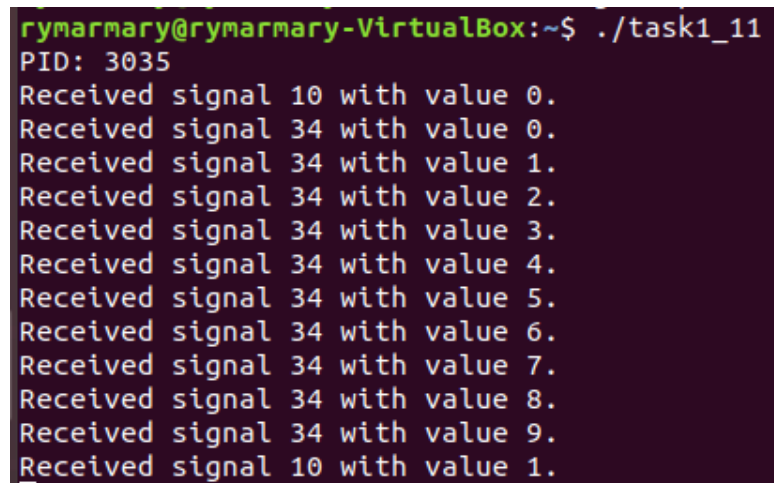
```

signal_val=34
pid=$1
val_counter=0

echo `sudo kill -10 $pid 1`
echo `sudo kill -10 $pid 2`

for (( counter=0; counter<10; counter++ ))
do
    echo `sudo kill -34 $pid $val_counter`
    ((val_counter=val_counter+1))
done

```

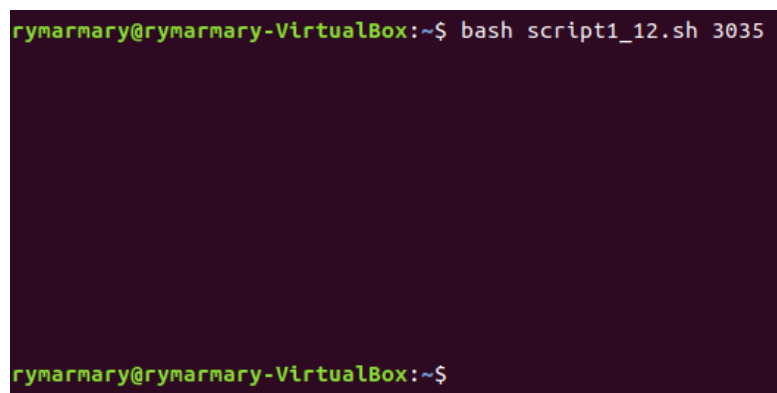


```

rymary@rymary-VirtualBox:~$ ./task1_11
PID: 3035
Received signal 10 with value 0.
Received signal 34 with value 0.
Received signal 34 with value 1.
Received signal 34 with value 2.
Received signal 34 with value 3.
Received signal 34 with value 4.
Received signal 34 with value 5.
Received signal 34 with value 6.
Received signal 34 with value 7.
Received signal 34 with value 8.
Received signal 34 with value 9.
Received signal 10 with value 1.

```

Рисунок 17 – Результат работы программы task1_11



```

rymary@rymary-VirtualBox:~$ bash script1_12.sh 3035

rymary@rymary-VirtualBox:~$

```

Рисунок 18 – Дополнительный терминал

2. Рассмотрим два типа каналов – программные (неименованные) и именованные.

Программные каналы – однонаправленные, используются для связи родственных процессов, но могут использоваться и для неродственных, если предоставить возможность передавать друг другу дескрипторы. Неименованный канал создаётся посредством вызова `pipe()`, который возвращает 2 файловых дескриптора `filedes[1]` для записи в канал `filedes[0]` для чтения из канала.

Организуем программу task2_1.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. Программный код task2_1.c представлен ниже. Результат работы программы показан на рисунке 19.

Как видно из результатов, 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].

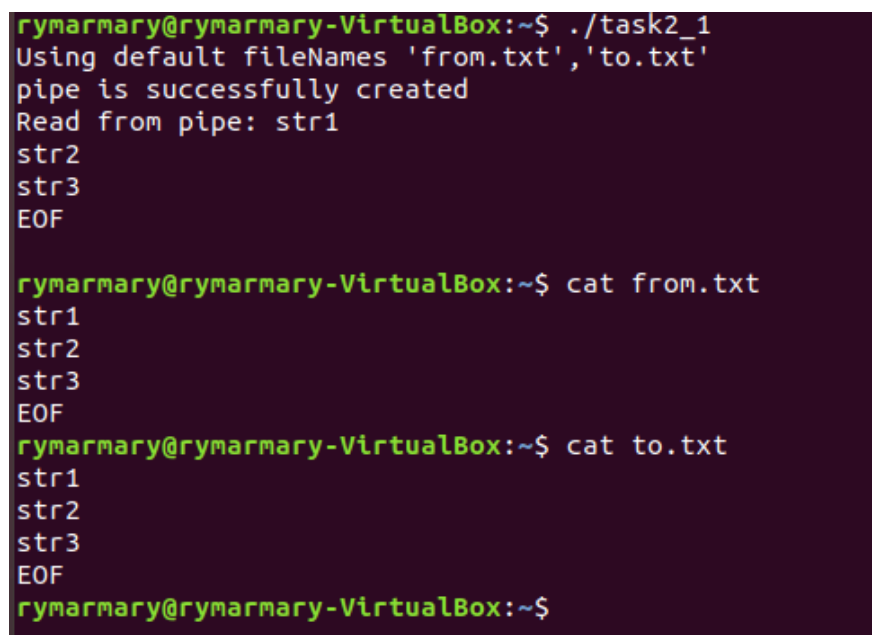
Программа для записи в файл с помощью pipe() task2_1.c:

```
#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
    }
    close(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;
}

```



```

rymarmary@rymarmary-VirtualBox:~$ ./task2_1
Using default fileNames 'from.txt','to.txt'
pipe is successfully created
Read from pipe: str1
str2
str3
EOF

rymarmary@rymarmary-VirtualBox:~$ cat from.txt
str1
str2
str3
EOF

rymarmary@rymarmary-VirtualBox:~$ cat to.txt
str1
str2
str3
EOF

rymarmary@rymarmary-VirtualBox:~$

```

Рисунок 19 – Результат работы task2_1.c

3. 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. Аббревиатура FIFO расшифровывается как «first in, first out» — «первым вошел, первым вышел», то есть 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 ввода-вывода (например, `fdopen`). 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.

Проведём следующий эксперимент: 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. Программный код `server.c` и `client.c` представлен ниже. Результат работы представлен на рисунках 20-21.

Программа работает как ожидалось. 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 отправляет серверу его содержимое, используя второй канал. После завершения передачи, сервер уничтожает каналы с помощью функции `unlink()`.

Код программы `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");
}
close(chan1);
close(chan2);
unlink("channel1");
unlink("channel2");
return 0;
}

```

Код программы 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 reading\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];

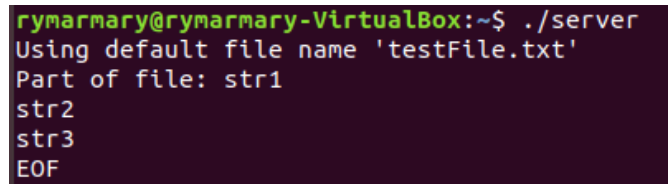
```



```

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;
}

```

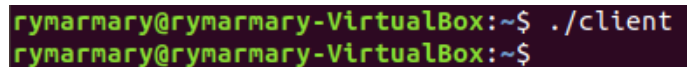


```

rymary@rymary-VirtualBox:~$ ./server
Using default file name 'testFile.txt'
Part of file: str1
str2
str3
EOF

```

Рисунок 20 – Результат работы программы server.c



```

rymary@rymary-VirtualBox:~$ ./client
rymary@rymary-VirtualBox:~$

```

Рисунок 21 – Дополнительный терминал

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: many_clients.sh. На рисунке 22 представлена работа программы.

Скрипт, создающий множество клиентов и серверов, many_clients.sh:

```

#!/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
done

```

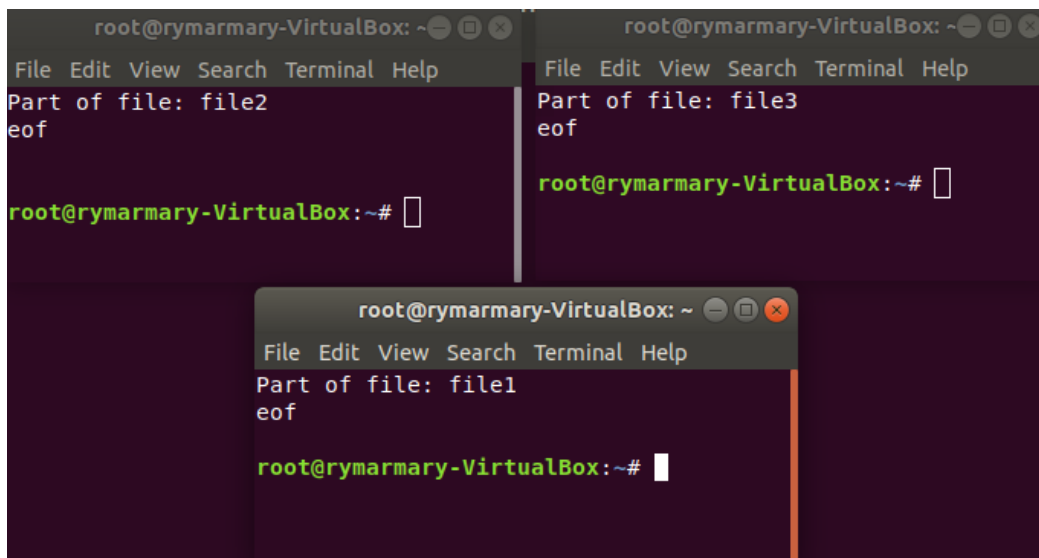


Рисунок 22 – Результат работы с многими клиентами и многими серверами

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.

Продemonстрируем 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. Новый код `server1.c` представлен ниже. Результат работы показан на рисунке 23.

Как результат, 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.

Изменённый код для сервера `server1.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];
    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");
    }
    close(chan1);
    close(chan2);
    unlink("channel1");
}

```

```

unlink("channel2");
printf("Server finished\n");
return 0;
}

```

```

rymarmary@rymarmary-VirtualBox:~$ ls -sl | grep chan
0 prw-r--r-- 1 root      root      0 May 22 19:56 channel1
0 prw-r--r-- 1 root      root      0 May 22 19:56 channel2

```

Рисунок 23 – Размер файла каналов

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. Эти значения указаны на рисунках 24-25.

```

rymarmary@rymarmary-VirtualBox:/$ grep -rn --col OPEN_MAX /usr/include
/usr/include/linux/fs.h:30:#define INR_OPEN_MAX 4096 /* Hard limit for nfile
rlimits */

```

Рисунок 24 – Значение `OPEN_MAX` системы

```

rymarmary@rymarmary-VirtualBox:/$ grep -rn --col PIPE_BUF /usr/include
/usr/include/linux/limits.h:14:#define PIPE_BUF 4096 /* # bytes in a
tomic write to a pipe */
/usr/include/x86_64-linux-gnu/bits/posix1_lim.h:99:#define _POSIX_PIPE_BUF
512

```

Рисунок 25 – Значение `PIPE_BUF` системы

4. Очередь сообщений находится в адресном пространстве ядра и имеет ограниченный размер. В отличие от каналов, которые обладают теми же самыми свойствами, очереди сообщений сохраняют границы сообщений. Это значит, что ядро ОС гарантирует, что сообщение, поставленное в очередь, не смешается с предыдущим или следующим сообщением при чтении из очереди. Очередь

сообщений можно рассматривать как связный список сообщений. Каждое сообщение представляет собой запись, очереди сообщений автоматически расставляют границы между записями, аналогично тому, как это делается в дейтаграммах UDP. Для записи сообщения в очередь не требуется наличия ожидающего его процесса в отличие от именованных каналов и FIFO, в которые нельзя произвести запись, пока не появится считывающий данные процесс. Поэтому процесс может записать в очередь какие-то сообщения, после чего они могут быть получены другим процессом в любое время, даже если первый завершит свою работу. С завершением процесса-источника данные не исчезают (данные, остающиеся в именованном или именованном канале, сбрасываются, после того как все процессы закроют его). Следует заметить, что, к сожалению, не определены системные вызовы, которые позволяют читать сразу из нескольких очередей сообщений, или из очередей сообщений и файловых дескрипторов. Видимо, отчасти и поэтому очереди сообщений широко не используются.

Проведём следующий эксперимент: 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. Сервер в цикле читает сообщения из очереди (тип = 1) функцией `msgrcv()` и посылает на каждое сообщение ответ клиенту (тип = 2) функцией `msgsnd()`. Целесообразно дублировать вывод сообщений на экран для контроля. В случае возникновения любых ошибок функцией `kill()` инициируется посылка сигнала `SIGINT`. Обработчик сигнала выполняет восстановление диспозиции сигналов и удаление очереди сообщений системным вызовом `msgctl()`. В файле `client.c` аналогично серверному коду должен быть получен ключ, затем доступ к очереди сообщений, отправка сообщения серверу (тип 1). Затем организовывается цикл ожидания сообщения клиентом с последующим чтением (тип 2). Таким образом, функции

чтения и отправки сообщения реализуются системными вызовами: `msgrcv()`, `msgsnd()`.

Содержания файлов `client2.c` и `server2.c` представлены ниже, результаты работы программы показаны на рисунках 26-27.

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 выводит его на экран и отправляет 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.

Код программы `server2.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;
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;
}

```

Код программы client2.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;
}

```



```
rymarmary@rymarmary-VirtualBox:~$ ./client2
Using default key file key
hello
Server's response: OK
```

Рисунок 26 – Работа client2.c

```
rymarmary@rymarmary-VirtualBox:~$ ./server2
Using default key file key
Client's request: hello
```

Рисунок 27 – Работа server2.c

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`. Эти значения представлены на рисунке 28.

По результатам видно, что размер одного сообщения не может быть больше 8192 байт, а очередь может содержать не более 32000 сообщений в один момент времени.

```
rymarmary@rymarmary-VirtualBox:~$ ipcs -l

----- Messages Limits -----
max queues system wide = 32000
max size of message (bytes) = 8192
default max size of queue (bytes) = 16384

----- Shared Memory Limits -----
max number of segments = 4096
max seg size (kbytes) = 18014398509465599
max total shared memory (kbytes) = 18014398509481980
min seg size (bytes) = 1

----- Semaphore Limits -----
max number of arrays = 32000
max semaphores per array = 32000
max semaphores system wide = 1024000000
max ops per semop call = 500
semaphore max value = 32767
```

Рисунок 28 – Результат утилиты `ipcs`

Были написаны скрипты, создающие множество серверов и клиентов и множество клиентов для одного сервера.

Скрипт, создающий множество пар серверов-клиентов, `script4_1.sh`:

```
#!/bin/bash
#script_many
```

```

gcc -o server server.c
gcc -o client client.c

# Создание 3 пар сервер-клиент
for i in {1..3}
do
    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

```

Скрипт, создающий множество клиентов для одного сервера, script4_2.sh:

```

#!/bin/bash
#script_many
gcc -o server server.c
gcc -o client client.c

# Создание 3 пар сервер-клиент
for i in {1..3}
do
    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

```

5. Рассмотрим несколько вариантов постановки задачи доступа к разделяемой памяти.

Вариант 1. Пусть 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 недостаточно 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 прочесть. Приведём пример почему достаточно двух семафоров: одному из них придадим смысл «запись разрешена», т.е. читатель предыдущие данные уже использовал; второму – «чтение разрешено», т.е. писатель уже сгенерировал новые данные, которые нужно прочесть.

Оба семафора бинарные и используют стандартные операции, захват семафора – это ожидание освобождения ресурса (установки семафора в 1) и последующий захват ресурса (установки семафора в 0), освобождение ресурса – это установка семафора в 1.

Пару семафоров, использованных таким образом, иногда называют разделенным бинарным семафором, поскольку в любой момент времени только один из них может иметь значение 1.

Содержания программ reader.c и writer.c представлены ниже. Результат работы программ представлен на рисунках 29-30.

Как видно из результатов, все сообщения от клиента сервером прочитаны.

Программный код writer.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.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);
    }
    // присоединяем shm в наше адресное пространство
    if ((p_msg = (Message *)shmat(shmemory, 0, 0)) < 0)
    {
        printf("Error while attaching shm\n");
        exit(1);
    }
    if ((semaphore = semget(key, 2, 0666)) < 0)
    {
        printf("Error while creating semaphore\n");
        exit(1);
    }
    char buf[100];
    for (;;)
    {
        bzero(buf, 100);
        printf("Type message to server. Empty string to finish\n");
        fgets(buf, 100, stdin);
        if (strlen(buf) == 1 && buf[0] == '\n')
        {
            printf("bye-bye\n");
            exit(0);
        }
        // хотим отправить сообщение
        if (semop(semaphore, writeEna, 1) < 0)
        {
            printf("Can't execute a operation\n");
            exit(1);
        }
        // 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(11);
        }
    }
    // отключение от области разделяемой памяти
    if (shmdt(p_msg) < 0)
    {
        printf("Error while detaching shm\n");
        exit(1);
    }
}

```

Программный код reader.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 "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 ((shmmemory = 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(shmmemory, 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);
}
// устанавливаем 2 семафор в 1, т.е. можно писать
if (semop(semaphore, setWriteEna, 1) < 0)
{
    printf("execution complete\n");
    kill(getpid(), SIGINT);
}
// основной цикл работы
for (;;)
{
    // ждем пока клиент начнет работу
    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);
    }
}
}

```

```
rymary@rymary-VirtualBox:~$ ./reader
Using default key file key
Client's message: hey
Client's message: it's 5th task
```

Рисунок 29 – Результат работы reader.c

```
rymary@rymary-VirtualBox:~$ ./writer
Using default key file key
Type message to server. Empty string to finish
hey
Type message to server. Empty string to finish
it's 5th task
Type message to server. Empty string to finish
bye-bye
rymary@rymary-VirtualBox:~$
```

Рисунок 30 – Результат работы writer.c

Вариант 2. К условиям предыдущего варианта добавим условие нескольких читателей и писателей. Это условие выполнится, поскольку повторная запись невозможна (чтобы очередной процесс-писатель отработал нужно освобождение семафора, которое выполняется из процесса-читателя).

Изменили код писателя (writer2.c), сделали ещё двух (writer3.c-writer4.c) писателей. Добавили одного читателя (reader2.c).

Вывод программ демонстрирует конкуренцию за семафоры. Все сообщения писателей были прочитаны или первым, или вторым читателем.

Вариант 3. К условиям предыдущей задачи добавим наличие не единичного буфера, а буфера некоторого размера. Поскольку буфер больше не равен единице, не нужно чередовать чтения и запись, можно записывать несколько записей подряд, делать несколько чтений. Возьмём два считающих семафора. Один из них равен нулю и имеет смысл «количество заполненных ячеек». Перед своей работой процессы-читатели захватывают его, ждут хотя-бы одной порции данных и читают, а после освобождают семафор «количество пустых ячеек». Процессы-писатели перед записью захватывают семафор «количество пустых ячеек» (ждут появление хотя-бы одной пустой ячейки для записи), а после записи освобождают семафор «количество полных ячеек». Так

решается проблема чтение из пустого буфера и запись в полный. Чтобы предотвратить захват несколькими процессами сразу, добавим бинарный семафор «доступ к памяти разрешен». Оба типа процессов должны захватывать его при попытке взаимодействия с памятью и освобождать после. При этом порядок операций освобождения не важен, но порядок захвата может привести к взаимной блокировке процессов.

Во время выполнения цикла for процесс-читатель прочитал 10 чисел первого клиента, 10 чисел второго и затем оставшиеся пять третьего, синхронизация работала корректно. (writer5.c-writer7.c, reader5.c)

```
Press enter to start working

Remove 4 from cell 4
Remove 3 from cell 3
Remove 2 from cell 2
Remove 1 from cell 1
Remove 0 from cell 0
Remove 9 from cell 4
Remove 8 from cell 3
Remove 7 from cell 2
Remove 6 from cell 1
Remove 5 from cell 0
Remove 4 from cell 4
Remove 3 from cell 3
Remove 2 from cell 2
Remove 1 from cell 1
Remove 0 from cell 0
Remove 9 from cell 4
Remove 8 from cell 3
Remove 7 from cell 2
Remove 6 from cell 1
Remove 5 from cell 0
Remove 4 from cell 4
Remove 3 from cell 3
Remove 2 from cell 2
Remove 1 from cell 1
Remove 0 from cell 0
```

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.

Написали скрипт для добавления множества клиентов и серверов для двух семафоров. Скрипт `script5_1.sh` представлен ниже. Результаты работы программы показаны на рисунке 31.

Содержание `script5_1.sh`:

```
#!/bin/bash
# script.sh
# 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
```

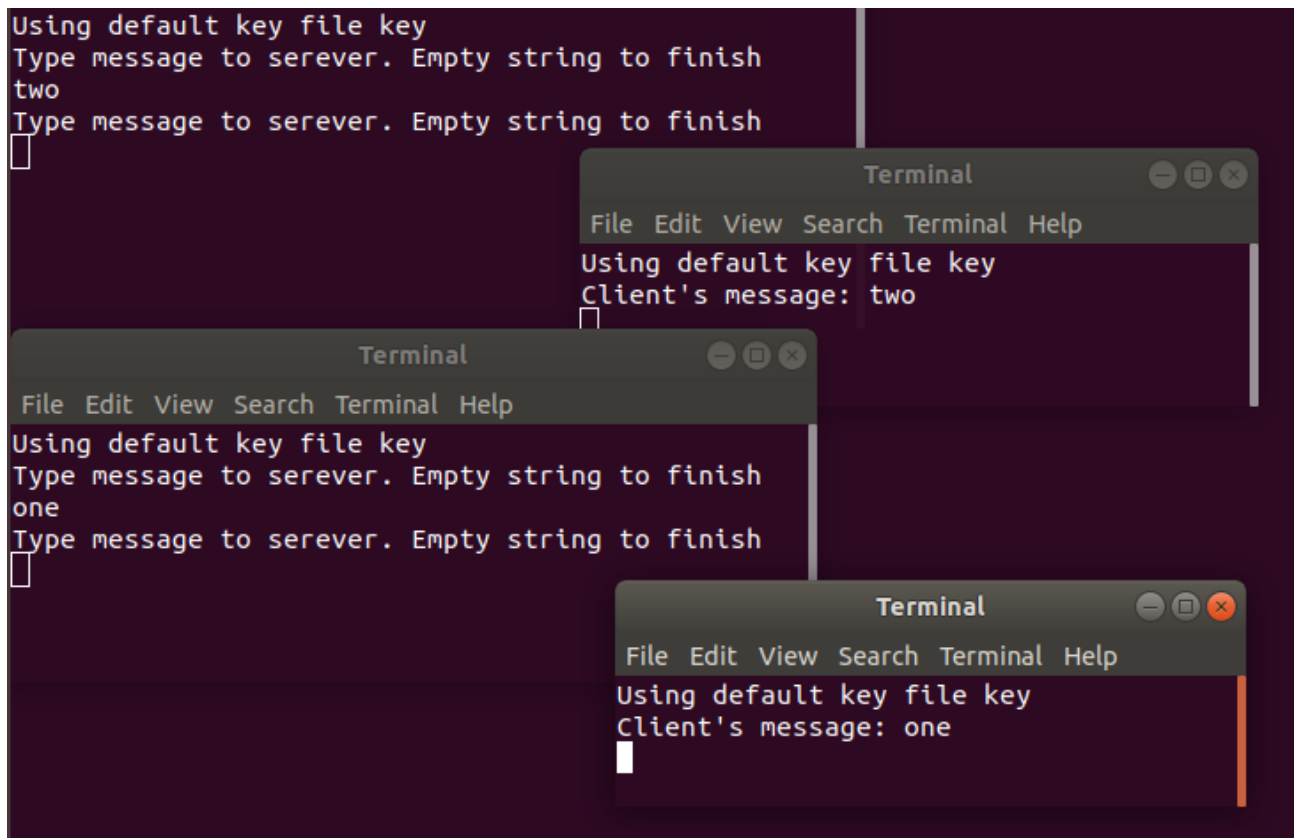


Рисунок 31 – Множество читателей и множество писателей

6. 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 ТСП 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 указано 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 `server6.c` и `client6.c` представлено ниже. Результат работы показан на рисунках 32-33.

Программный код `server6.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>
#include <pthread.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);
}

```

Программный код client6.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);
}

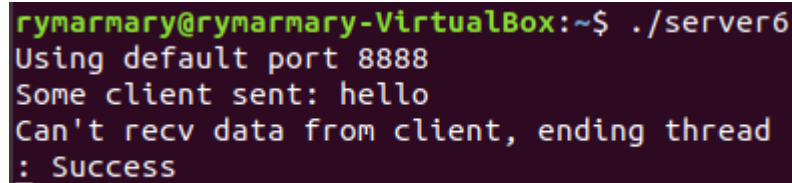
int sendFix(int sock, char *buf, int flags)
{
    // число байт в сообщении
    unsigned msgLength = strlen(buf);

```

```

int res = send(sock, &msgLength, sizeof(unsigned), flags);
if (res <= 0)
    return res;
send(sock, buf, msgLength, flags);
}

```

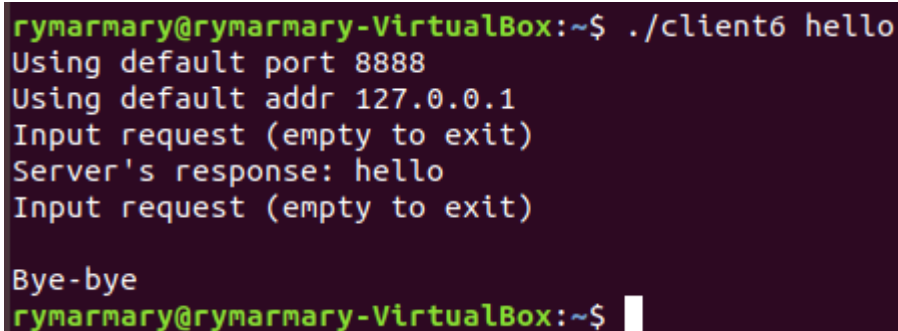


```

rymary@rymary-VirtualBox:~$ ./server6
Using default port 8888
Some client sent: hello
Can't recv data from client, ending thread
: Success

```

Рисунок 32 – Работа программы server6.c



```

rymary@rymary-VirtualBox:~$ ./client6 hello
Using default port 8888
Using default addr 127.0.0.1
Input request (empty to exit)
Server's response: hello
Input request (empty to exit)

Bye-bye
rymary@rymary-VirtualBox:~$

```

Рисунок 33 – Работа программы client6.c

Взаимодействие на основе UDP (udpserver.c, udpclient.c).

При использовании UDP-сокетов сообщения были так же успешно получены и отправлены и сервером, и клиентом по IP-адресу loopback.

При использования большого числа клиентов (около 1000), то по UDP некоторые сообщения могут теряться, значит, UDP менее надёжный протокол. По TCP протоколу все сообщения доходят.

Выводы.

В ходе выполнения лабораторной работы были изучены различные средства межпроцессного взаимодействия в операционной системе Linux.

Были рассмотрены надежные и ненадежные сигналы, сигналы реального времени, неименованные и именованные каналы, очереди сообщений и семафоры. Каждое из этих средств имеет свои особенности и применяется в различных ситуациях.

Надежные и ненадежные сигналы используются для передачи информации между процессами, а сигналы реального времени позволяют обрабатывать события в режиме реального времени.

Неименованные и именованные каналы, а также очереди сообщений обеспечивают передачу данных между процессами, а семафоры позволяют синхронизировать доступ к разделяемой памяти.

При выборе средства межпроцессного взаимодействия необходимо учитывать требования к скорости, надежности и безопасности передачи данных. Операционная система Linux предоставляет широкий выбор инструментов для решения задачи межпроцессного взаимодействия и позволяет выбрать наиболее подходящее средство в зависимости от конкретной задачи.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Барретт Дж. Linux. Карманный справочник. - СПб.: Питер, 2013. - 320 с.
2. 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.
3. А. Таненбаум. Современные операционные системы / А. Таненбаум, Х. Бос. – 4-е изд., перераб. и доп. – СПб.: Изд-во Питер. 2015. – 1120 с.
4. Шотт Б. Командная строка Linux. - СПб.: Питер, 2013. - 416 с.
5. Официальная документация Linux:
<https://www.kernel.org/doc/html/latest/>
6. Руководства и документация по Linux на сайте Linux.org:
<https://www.linux.org/docs/>
7. Руководства и документация по Linux на сайте Ubuntu:
<https://help.ubuntu.com/>