

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*

1.1 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 send by son!\n");

while(1) {sleep(2);}
return 0;
}

```

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

Результат работы программы:

```

roman@DESKTOP-07282GR:~/1.1$ sudo ./1_1 &
[3] 1294
roman@DESKTOP-07282GR:~/1.1$ Father process is started! PID: 1300
Child process is started!
PPID: 1300, PID: 1301
Signal SIGTRAP is send by son!
Signal SIGTRAP is handled in father!

```

Отправим сигнал из терминала:

```

roman@DESKTOP-07282GR:~/1.1$ sudo kill -SIGSEGV 1300
Signal SIGSEGV is handled!

```

Сигналы успешно обрабатываются для процессов в разных файлах.

Теперь сделаем 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:

```
roman@DESKTOP-07282GR:~/1.1$ sudo ./1_2
Главный поток с TID 2150 запущен
Дочерний поток с TID 2151 запущен
Поток с TID 2151 получил сигнал: 10
Поток с TID 2151 получил сигнал: 12
^СПоток с TID 2150 получил сигнал: 2
Главный поток с TID 2150 завершен
Дочерний поток с TID 2151 завершен
roman@DESKTOP-07282GR:~/1.1$
```

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:

```
roman@DESKTOP-07282GR:~/1.1$ gcc main.c child.c -o mainchild -lpthread
roman@DESKTOP-07282GR:~/1.1$ sudo ./mainchild
[sudo] password for roman:
Главный поток с TID 277 запущен
Дочерний поток с TID 278 запущен
Поток с TID 278 получил сигнал: 10
Поток с TID 278 получил сигнал: 12
^Поток с TID 277 получил сигнал: 2
Главный поток с TID 277 завершен
Дочерний поток с TID 278 завершен
```

2 Надёжные сигналы

1.1. 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) {
```



```

    if (sig != SIGUSR1) {
        printf("Caught bad signal %d\n", sig);
        return;
    }
    printf("SIGUSR1 caught\n");
    sleep(60);
}

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

```

```

}

```

```

~/05/Lab5/2 ./a.out &
[1] 38105
~/05/Lab5/2 kill -SIGUSR1 %1
SIGUSR1 caught
~/05/Lab5/2 kill -SIGINT %1
~/05/Lab5/2 jobs
[1]  + running      ./a.out
~/05/Lab5/2 jobs
[1]  + running      ./a.out
~/05/Lab5/2
[1]  + 38105 interrupt ./a.out

```

Чтобы иметь возможность отправить сигналы с терминала следует запустить программу в фоновом режиме. По результатам сигнал SIGUSR1 принят корректно, но после отправки 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 ненадежных 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.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;
}

```

Вывод программы:

```

/mnt/c/Users/roman/AllFiles/lab5/2/default
roman@DESKTOP-07282GR:~/default$ ls
2  2.c  2_2.c
roman@DESKTOP-07282GR:~/default$ gcc 2_2.c -o 2_2
roman@DESKTOP-07282GR:~/default$ ./2_2
^C
roman@DESKTOP-07282GR:~/default$ ./2_2 &
[1] 1833
roman@DESKTOP-07282GR:~/default$ kill -SIGUSR1 %1
SIGUSR1 caught, sending SIGINT
roman@DESKTOP-07282GR:~/default$ jobs
[1]+  Running                  ./2_2 &
roman@DESKTOP-07282GR:~/default$ jobs
[1]+  Interrupt                ./2_2

```

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

```

#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("\nНадёжный сигнал (SIGINT) получен.\n");
}

```

```

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.

```

roman@DESKTOP-07282GR:~/my$ gcc demo.c -o demo
roman@DESKTOP-07282GR:~/my$ ./demo
Демонстрация работы сигналов:
Процесс ID: 3033
Надёжный сигнал (SIGINT): остановка программы и вывод результатов
Надёжный сигнал (SIGUSR1): инкремент счетчика без потерь
Ненадёжный сигнал (SIGALRM): инкремент счетчика с возможны
Ожидание сигналов...
SIGUSR1 sent
SIGALRM sent
SIGUSR1 sent
SIGALRM sent
SIGUSR1 sent
SIGALRM sent
SIGUSR1 sent
SIGALRM sent
SIGUSR1 sent
SIGALRM sent
SIGUSR1 sent
SIGALRM sent
SIGUSR1 sent
SIGALRM sent
^C
Надёжный сигнал (SIGINT) получен.
Отправлено надёжных сигналов (SIGUSR1): 6
Обработано надёжных сигналов (SIGUSR1): 6
Потеряно надёжных сигналов (SIGUSR1): 0

Отправлено ненадёжных сигналов (SIGALRM): 6
Обработано ненадёжных сигналов (SIGALRM): 5
Потеряно ненадёжных сигналов (SIGALRM): 1

```

Видим, что ненадёжные сигналы теряются в отличие от надёжных.

3. Сигналы реального времени

3.1 – 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.c

```
// 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.

```
roman@roman-VirtualBox:/media/sf_AllFiles/lab5/3$ sudo ./3_1
[sudo] пароль для roman:
Sending normal signal 3 with value 1
Received signal 3 with value 1
Sending real-time signal 35 with value 1
Received signal 35 with value 1
Sending normal signal 4 with value 2
Received signal 4 with value 2
Sending real-time signal 36 with value 2
Received signal 36 with value 2
Sending normal signal 5 with value 3
Received signal 5 with value 3
Sending real-time signal 37 with value 3
Received signal 37 with value 3
Sending normal signal 6 with value 4
Received signal 6 with value 4
Sending real-time signal 38 with value 4
Received signal 38 with value 4
```

3.3. Подтвердим наличие приоритетов сигналов реального времени

3 2.c

```
#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:

```
roman@roman-VirtualBox:/media/sf_AllFiles/lab5/3$ gcc 3_2.c -o 3_2
roman@roman-VirtualBox:/media/sf_AllFiles/lab5/3$ sudo ./3_2
roman@roman-VirtualBox:/media/sf_AllFiles/lab5/3$ cat result.txt
Sending signals...
Sending normal signal 1
Sending normal signal 2
Sending normal signal 3
Sending normal signal 4
Sending normal signal 5
Sending real-time signal 34
Sending real-time signal 35
Sending real-time signal 36
Sending real-time signal 37
Sending real-time signal 38
Signals sent. Sleeping for 5 seconds to allow handling...
Handling real-time signal 34
Handling real-time signal 35
Handling real-time signal 36
Handling real-time signal 37
Handling real-time signal 38
Handling normal signal 1
Handling normal signal 2
Handling normal signal 3
Handling normal signal 4
Handling normal signal 5
```

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

4. Неименованные каналы

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

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 filedес[2];

    if (pipe(filedес) < 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(filedес[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(filedес[1], buf, res); // пишем их в пайп
        }

        fclose(f);
        close(filedес[1]);
        return 0;
    }

    // процесс отец
    // закрывает пайп для записи
    close(filedес[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;
}

```

Вывод:

```

roman@DESKTOP-07282GR:~/default$ gcc pipe1.c -o pipe1
roman@DESKTOP-07282GR:~/default$ ./pipe1
Using default fileNames 'from.txt', 'to.txt'
pipe is successfully created
Read from pipe: first string
second
third
That's all
roman@DESKTOP-07282GR:~/default$ |

```

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:

```
roman@roman-VirtualBox:/media/sf_AllFiles1/lab5/4/4_2$ gcc server.c -o server
roman@roman-VirtualBox:/media/sf_AllFiles1/lab5/4/4_2$ ./server
Using default file name 'testFile.txt'
Part of file: first string
second
Last string
```

```
roman@roman-VirtualBox:/media/sf_AllFiles1/lab5/4/4_2$ gcc client.c -o client
roman@roman-VirtualBox:/media/sf_AllFiles1/lab5/4/4_2$ ./client
```

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()`.

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

```
#!/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" &
done
```

```
sleep 1
done
```

Результат работы:

```
root@roman-VirtualBox: /media/sf_AllFiles1/lab5/4/4_2/many
Файл Правка Вид Поиск Терминал Справка
reading message file 1 first
file 1 second
file 1 third
000
sending message file 1 first
file 1 second
file 1 third
000
root@roman-VirtualBox: /media/sf_AllFiles1/lab5/4/4_2/many#

root@roman-VirtualBox: /media/sf_AllFiles1/lab5/4/4_2/many
Файл Правка Вид Поиск Терминал Справка
reading message file 2 first
file 2 second
file 2 third
000
sending message file 2 first
file 2 second
file 2 third
000
root@roman-VirtualBox: /media/sf_AllFiles1/lab5/4/4_2/many#

root@roman-VirtualBox: /media/sf_AllFiles1/lab5/4/4_2/many
Файл Правка Вид Поиск Терминал Справка
Part of file: file 1 first
file 1 second
file 1 third
root@roman-VirtualBox: /media/sf_AllFiles1/lab5/4/4_2/many#

root@roman-VirtualBox: /media/sf_AllFiles1/lab5/4/4_2/many
Файл Правка Вид Поиск Терминал Справка
Part of file: file 2 first
file 2 second
file 2 third
root@roman-VirtualBox: /media/sf_AllFiles1/lab5/4/4_2/many#
```

4.3. Был изменен server.c.

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];
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:

```

~/05/lab5/4/4 3 gcc server.c -o server
~/05/lab5/4/4 3 ./server
Using default file name 'testFile.txt'
Waiting for clint write to channnel

Part of file: first string
second
third
That's all

Servr finished

```



```
~/OS/lab5/4/4_3 ./client  
Client finished
```

Размер файловой системы

```
roman@roman-VirtualBox: /media/sf_AllFiles1/lab5/4/4_3$ ls -l  
итого 52  
rw-rw-r-- 1 roman roman 0 мая 3 23:09 channel1  
rw-rw-r-- 1 roman roman 0 мая 3 23:09 channel2  
-rwxrwxr-x 1 roman roman 17120 мая 3 21:38 client  
-rw-rw-r-- 1 roman roman 1319 мая 3 21:34 client.c  
-rwxrwxr-x 1 roman roman 17288 мая 3 21:38 server  
-rw-rw-r-- 1 roman roman 1690 мая 3 21:37 server.c  
-rw-rw-r-- 1 roman roman 37 мая 3 21:38 testFile.txt
```

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);
    }
}

```

```
// 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;
}
```

Вывод:

```
roman@roman-VirtualBox:/media/sf_AllFiles1/lab5/5/queue$ ./server
Using default key file key
Client's request: rrgrg
Client's request: ya uzhe zadolbalsya eti osi delat
```

```
roman@roman-VirtualBox:/media/sf_AllFiles1/lab5/5/queue$ ./client
Using default key file key
rrgrg
Server's response: OK
ya uzhe zadolbalsya eti osi delat
Server's response: OK
```

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.

Server: Сервер получает ключ, по имени файла. С помощью ключа и идентификатора = 'Q' получает очередь сообщений и ждет сообщений с типом 1 от клиентов. Client: получение уникального ключа для получения доступа к очереди, получение доступа к созданной сервером очереди. Чтение потока ввода (сообщения типа 1), посылка строки в очередь. Server: чтение сообщения типа 1, вывод сообщения на экран и отправка сообщения типа 2. Client: чтение первого сообщения с типом 2 и вывод его на экран (ОК).

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

Script_many.sh

```
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
    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
```

Script.sh

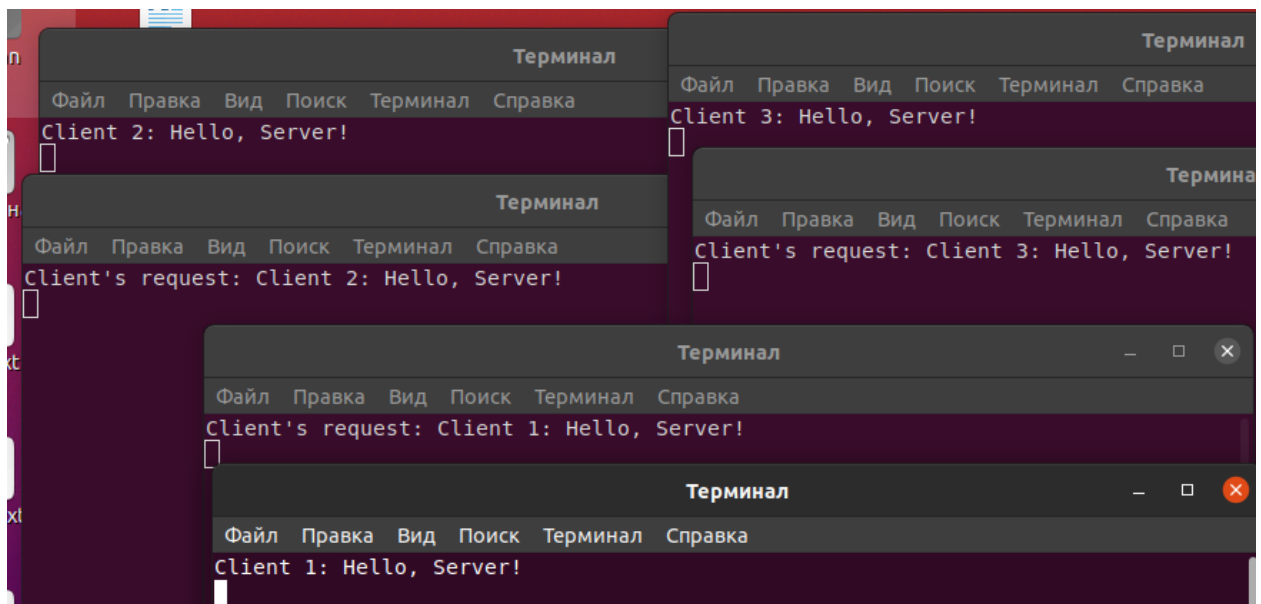
```
#!/bin/bash
```

```
# 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
    gnome-terminal -- ./client 9000 $i &
    echo "Started client $i connecting to queue"
done
```

Результат работы:



5. 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`.

```
~/05/lab5 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) = 18446744073709551612
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
```

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 невозможность повторного чтения одних и тех же данных и невозможность перезаписи данных, т. е. новой записи, до тех пор, пока читатель не прочитает предыдущую.

В таком варианте задания для синхронизации процессов достаточно двух семафоров. Покажем, почему не достаточно одного на примере. Так как мы используем один семафор, то алгоритм работы читателя и писателя может быть только таким – захват семафора, выполнение действия (чтение / запись), освобождение семафора. Теперь допустим, что читатель прочитал данные, освободил семафор и еще не до конца использовал квант процессорного времени. Тогда он перейдет на новую итерацию, снова захватит только что освобожденный семафор и снова прочитает данные – ошибка.

Теперь покажем, почему достаточно двух семафоров. Придадим одному из них смысл «запись разрешена», т.е. читатель предыдущие данные уже использовал; второму – «чтение разрешено», т.е. писатель уже сгенерировал новые данные, которые нужно прочитать.

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);
    }
    //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);
}
}
}
}

```

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);
    }
}

```

```

//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 serever. 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);
}
}

```

Результат работы программы:

```

roman@roman-VirtualBox: /media/sf_AllFiles1/lab5/5/var1$ ./server
Using default key file key
Client's message: Gruppa Alisa - the best
Client's message: Elena Vladimirovna, postavte 3, pozhaluysta
roman@roman-VirtualBox: /media/sf_AllFiles1/lab5/5/var1$ ./client
Using default key file key
Type message to serever. Empty string to finish
Gruppa Alisa - the best
Type message to serever. Empty string to finish
Elena Vladimirovna, postavte 3, pozhaluysta
Type message to serever. Empty string to finish

```

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:

```
Терминал
Файл  Правка  Вид  Поиск  Терминал  Справка
Using default key file key
Client's message: dsdasd
Client's message: prichinyayu dobro i nanoshu sprevedlivost
█

Терминал
Файл  Правка  Вид  Поиск  Терминал  Справка
Using default key file key
Client's message: dsfdsf
█

Терминал
Файл  Правка  Вид  Поиск  Терминал  Справка
Using default key file key
Type message to serever. Empty string to finish
dsdasd
Type message to serever. Empty string to finish
dsfdsf
Type message to serever. Empty string to finish
prichinyayu dobro i nanoshu sprevedlivost
Type message to serever. Empty string to finish
█
```

6. Сокеты

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 отправкой собственно данных посылает «заголовок» - количество байт в отправке. 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.

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;
}

```

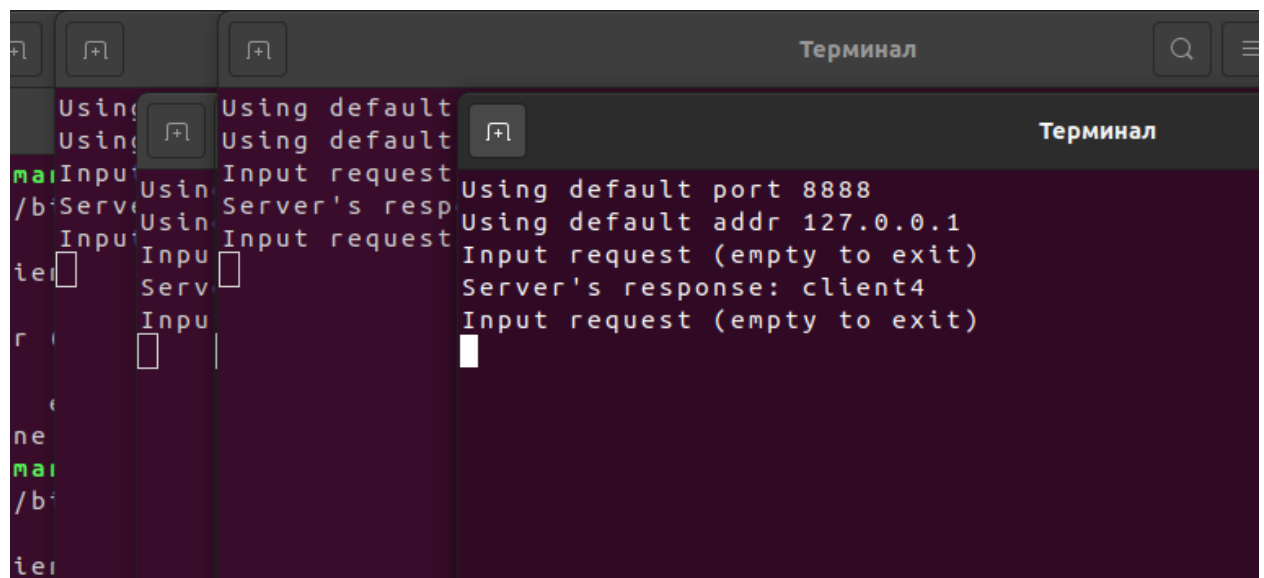
```
send(sock, buf, msgLength, flags);
}
```

script.sh

```
roman@roman-VirtualBox: /media/sf_AllFiles1/lab5/6$ cat script.sh
1 #!/bin/bash
2
3 client_amount=$1
4
5 for (( counter=0; counter<client_amount; counter++ ))
6 do
7     echo `gnome-terminal -- sh -c "bash -c \"./client client$counter; exec bash\""`
8 done
```

Результат работы программы:

```
roman@roman-VirtualBox: /media/sf_AllFiles1/lab5/6$ ./server
Using default port 8888
Some client sent: kk
Can't recv data from client, ending thread
: Success
Some client sent: client0
Some client sent: client1
Some client sent: client2
Some client sent: client3
Some client sent: client4
```



```
Using default port 8888
Using default addr 127.0.0.1
Input request (empty to exit)
Server's response: client4
Input request (empty to exit)
```

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.

Были написаны скрипты, создающие множество серверов для множества клиентов.

Script1.sh

```
#!/bin/bash

# Запускаем серверы на портах 9000-9004
for i in {0..4}; do
    port=$((9000 + i))
    gnome-terminal -- ./server $port &
    echo "Started server $i on port $port"
done

# Даем серверам время на запуск
sleep 2

# Запускаем клиентов, каждый подключается к своему серверу
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
```

Результат работы:

```

Терминал
roman@roman-VirtualBox: /media/sf_AllFiles/lab5/6/many
File Edit View Search Terminal Help
Some client sent: Hello, Server 4!
Some client sent: hello
Bye-bye
Can't recv data from client, ending thread
: Success
Server's response: Hello, Server 3!
Input request (empty to exit)
Bye-bye
Can't recv data from client, ending thread
: Success
Can't recv data from client, ending thread
: Success
Can't recv data from client, ending thread
: Success
roman@roman-VirtualBox:/media/sf_AllFiles/lab5/6/many$ sudo bash scrip
t1.sh
Started server 0 on port 9000
Started server 1 on port 9001
Started server 2 on port 9002
Started server 3 on port 9003
Started server 4 on port 9004
Started client 0 connecting to port 9000
Started client 1 connecting to port 9001
Started client 2 connecting to port 9002
Started client 3 connecting to port 9003
Started client 4 connecting to port 9004
roman@roman-VirtualBox:/media/sf_AllFiles/lab5/6/many$

```

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

```

Using default port 8888
Some client sent: hello_from_udp
Some client sent: qwe
Some client sent: as
Some client sent: zxc
Using default port 8888
Using default addr 192.168.0.2
Input request (empty to exit)
Server's response: hello_from_udp
Input request (empty to exit)
qwe
Server's response: qwe
Input request (empty to exit)
as
Server's response: as
Input request (empty to exit)
zxc
Server's response: zxc
Input request (empty to exit)

```

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.