

**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: Межпроцессорное взаимодействие**

Студент гр. 1304

\_\_\_\_\_

Кривоченко Д.И.

Преподаватель

\_\_\_\_\_

Душутин Е.В.

Санкт-Петербург

2023

## Цель работы.

## Ход работы.

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

Листинг 1 – Программа с обработкой сигналов

|    |  |
|----|--|
|    | File: 1.1_ex.c   |
| 1  | #include <stdio.h>   |
| 2  | #include <signal.h>  |
| 3  | #include <unistd.h>  |
| 4  | #include <stdlib.h>  |
| 5  | static void sigHandler(int sig)  |
| 6  | {  |
| 7  | printf("Caught signal %s\n", sig == SIGUSR1 ? "SIGUSR1" : "SIGUSR2");  |
| 8  | printf("Parent = %d\n", getppid());                                    |
| 9  | // восстанавливаем старую диспозицию                                   |
| 10 | signal(sig, SIG_DFL);  |
| 11 | }  |
| 12 | int main()   |
| 13 | {  |
| 14 | printf("\nFather started: pid = %i,ppid = %i\n", getpid(), getppid()); |
| 15 | signal(SIGUSR1, sigHandler);   |
| 16 | signal(SIGUSR2, sigHandler);   |
| 17 | signal(SIGINT, SIG_DFL);   |
| 18 | signal(SIGCHLD, SIG_IGN);  |
| 19 | int forkRes = fork();  |
| 20 | if (forkRes == 0)  |
| 21 | {  |
| 22 | // программа-потомок   |
| 23 | printf("\nSon started: pid = %i,ppid = %i\n", getpid(), getppid());    |
| 24 | // отправляем сигналы родителю   |
| 25 | if (kill(getppid(), SIGUSR1) != 0)                                     |
| 26 | {  |
| 27 | printf("Error while sending SIGUSR1\n");                               |
| 28 | exit(1);   |
| 29 | }  |
| 30 | printf("Successfully sent SIGUSR1\n");                                 |
| 31 | return 0;  |
| 32 | }  |
| 33 | // программа-родитель  |
| 34 | wait(NULL);  |
| 35 | // ждем сигналов   |
| 36 | for (;;)   |
| 37 | {  |
| 38 | pause();   |
| 39 | }  |
| 40 | return 0;  |
| 41 | }  |

|   |   |
|---|---|
| <pre>golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter1\$ ./1.1_ex Father started: pid = 486,ppid = 279  Son started: pid = 487,ppid = 486 Successfully sent SIGUSR1 Caught signal SIGUSR1 Parent = 279 Caught signal SIGUSR2 Parent = 279 User defined signal 2 golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter1\$</pre> | <pre>golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter1\$ kill -SIGUSR2 486 golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter1\$ kill -SIGUSR2 486 golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter1\$</pre> |
|---|---|

*Рисунок 1 - результат работы программы из листинга 1*

Как и ожидалось, после восстановления обработчика сигналов вызван обработчик по умолчанию.

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

*Листинг 2 - Модифицированная программа с обработкой сигналов*

|    |   |
|----|---|
|    | File: 1.1_ex_copy.c   |
| 1  | //КОПИЯ 1.1ex, ПРИ ЭТОМ НЕ ВОССТАНАВЛИВАЕТСЯ ДИСПОЗИЦИЯ И НЕ ОБРА |
| 2  | БАТЫВАЕТСЯ SIGUSR1  |
| 3  | #include <stdio.h>  |
| 4  | #include <signal.h>   |
| 5  | #include <unistd.h>   |
| 6  | #include <stdlib.h>   |
| 7  | static void sigHandler(int sig)                                   |
| 8  | {   |
| 9  | printf("Caught signal %s\n", sig == SIGUSR1 ? "SIGUSR1" : "SIG    |
| 10 | USR2");   |
| 11 | printf("Parent = %d\n", getppid());                               |
| 12 | // восстанавливаем старую диспозицию                              |
| 13 | signal(sig, SIG_DFL);   |
| 14 | }   |
| 15 | int main()  |
| 16 | {   |
| 17 | printf("\nFather started: pid = %i,ppid = %i\n", getpid(), getp   |
| 18 | pid());   |
| 19 | signal(45, sigHandler);   |
| 20 | signal(SIGUSR2, sigHandler);                                      |
| 21 | signal(SIGINT, SIG_DFL);  |
| 22 | signal(SIGCHLD, SIG_IGN);   |
| 23 | int forkRes = fork();   |
| 24 | if (forkRes == 0)   |
| 25 | {   |
| 26 | // программа-потомок  |
| 27 | printf("\nSon started: pid = %i,ppid = %i\n", getpid(), getpp     |
| 28 | id());  |
| 29 | // отправляем сигналы родителю                                    |
| 30 | if (kill(getppid(), SIGUSR1) != 0)                                |
|    | {   |
|    | printf("Error while sending SIGUSR1\n");                          |
|    | exit(1);  |

```

31     }
32     printf("Successfully sent SIGUSR1\n");
33     return 0;
34 }
35 // программа-родитель
36 wait(NULL);
37 // ждем сигналов
38 for (;;)
39 {
40     pause();
41 }
42 return 0;

```

|  |   |   |
|--|---|---|
| <pre> ./1.1_ex Father started: pid = 594,ppid = 279 Son started: pid = 595,ppid = 594 Successfully sent SIGUSR1 Caught signal SIGUSR1 Parent = 279 Real-time signal 11 golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter1\$ </pre> | <pre> ./1.1_ex_copy Father started: pid = 683,ppid = 309 Son started: pid = 684,ppid = 683 Successfully sent SIGUSR1 Caught signal SIGUSR2 Parent = 309 Real-time signal 11 golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter1\$ </pre> | <pre> golvs1noob@DESKTOP-2IJSUSC:~\$ kill -45 594 golvs1noob@DESKTOP-2IJSUSC:~\$ kill -45 683 golvs1noob@DESKTOP-2IJSUSC:~\$ kill -45 683 golvs1noob@DESKTOP-2IJSUSC:~\$ </pre> |
|--|---|---|

Рисунок 2 – результат работы программы из листинга 2

Как и ожидалось, ввиду разной диспозиции сигналов, программа из листинга 1 не обрабатывала сигнал 45 написанным обработчиком, а программа 2 – обрабатывала.

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

Листинг 3 – содержание программы, ожидаемой сигнал

|    |   |
|----|---|
|    | File: 1.1_demo.c  |
| 1  | #include <stdio.h>  |
| 2  | #include <signal.h>   |
| 3  | #include <unistd.h>   |
| 4  | #include <stdlib.h>   |
| 5  | static void sigHandler(int sig)                                       |
| 6  | {   |
| 7  | printf("Caught signal %s\n", sig == SIGUSR1 ? "SIGUSR1" : "SIGUSR2"); |
| 8  | printf("%d", getppid());  |
| 9  | // восстанавливаем старую диспозицию                                  |
| 10 | signal(sig, SIG_DFL);   |
| 11 | }   |
| 12 | int main()  |
| 13 | {   |
| 14 | signal(SIGUSR2, sigHandler);  |
| 15 | FILE* fp;   |
| 16 | fp = fopen("pid_file.txt", "w");                                      |
| 17 | char str[100];  |
| 18 | if (fp == NULL){  |
| 19 | perror("FILE");   |
| 20 | exit(1);  |
| 21 | }   |

```

22 | int father_pid = getpid();
23 | sprintf(str, "%d", father_pid);
24 | printf("\nFather started: pid = %i\n", father_pid);
25 | //putw(father_pid, fp);
26 | fprintf(fp, str);
27 | fclose(fp);
28 | // ждем сигналов
29 | for (;;)
30 | {
31 |     pause();
32 | }
33 |
34 | return 0;
35 | }

```

Листинг 4 - содержание программы, посылающей сигнал программе 3

|    | File: 1.1.demo_copy.c   |
|----|---|
| 1  | #include <stdio.h>  |
| 2  | #include <signal.h>   |
| 3  | #include <unistd.h>   |
| 4  | #include <stdlib.h>   |
| 5  | #include <string.h>   |
| 6  | static void sigHandler(int sig)                                       |
| 7  | {   |
| 8  | printf("Caught signal %s\n", sig == SIGUSR1 ? "SIGUSR1" : "SIGUSR2"); |
| 9  | printf("%d", getppid());  |
| 10 | // восстанавливаем старую диспозицию                                  |
| 11 | signal(sig, SIG_DFL);   |
| 12 | }   |
| 13 | char* concat(const char *s1, const char *s2){                         |
| 14 | char* result = malloc(strlen(s1) + strlen(s2) + 1);                   |
| 15 | strcpy(result, s1);   |
| 16 | strcat(result, s2);   |
| 17 | return result;  |
| 18 | }   |
| 19 | int main()  |
| 20 | {   |
| 21 | FILE* fp;   |
| 22 | fp = fopen("pid_file.txt", "r");                                      |
| 23 | char str[100];  |
| 24 | char buf_2[100];  |
| 25 | if (fp == NULL){  |
| 26 | perror("FILE");   |
| 27 | exit(1);  |
| 28 | }   |
| 29 | fgets(str, 100, fp);  |
| 30 | //printf("%s\n", str);  |
| 31 | char* s = concat("kill -SIGUSR2 ", str);                              |
| 32 | system(s);  |
| 33 | fclose(fp);   |
| 34 | free(s);  |
| 35 | // ждем сигналов  |
| 36 |   |
| 37 | return 0;   |
| 38 | }   |

|   |   |
|---|---|
| <pre> golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter1\$ clear golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter1\$ ./1.1_demo Father started: pid = 2068 Caught signal SIGUSR2 User defined signal 2 golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter1\$ </pre> | <pre> golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter1\$ ./1.1.demo_copy golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter1\$ ./1.1.demo_copy golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter1\$ </pre> |
|---|---|

Рисунок 3 - демонстрация обработки сигналов неродственных процессов

Таким образом, повторили эксперимент для процессов, порождаемых в разных файлах. Как и ожидалось, сигнал обрабатывается точно так же, как и в случае с родственными процессами (т. е. сначала сигнал обрабатывается нашим

sighandler, потом восстанавливается диспозиция и программа завершается при второй послылке сигнала).

Рассмотрим обработку сигналов потоком одного процесса. Для этого используем команду `sigwait(sigset_t, sig)`. Она прекращает исполнение потока, пока один из сигналов из списка `sigset_t` не становится в ожидании (или не послан напрямую), и возвращает номер сигнала в `sig`. Обработаем SIGUSR1 и SIGUSR2 так: на посыл SIGUSR1 остановим поток, а SIGUSR2 проигнорируем. Содержание программы в листинге 5, результат работы на рисунке 4.

Листинг 5 – программа с обработкой сигнала потоком

```
File: 1.1.thread1.c
1 | #include <stdio.h>
2 | #include <stdlib.h>
3 | #include <signal.h>
4 | #include <pthread.h>
5 | #include <unistd.h>
6 | #define _GNU_SOURCE
7 |
8 | pthread_t thread1;
9 | sigset_t set;
10 | int thread_id;
11 | void *thread_func(void *args) {
12 |     thread_id = gettid();
13 |     printf("Thread started. %d\n", thread_id);
14 |     while(1) {
15 |         int signum;
16 |         sigwait(&set, &signum);
17 |         printf("Received signal %d\n", signum);
18 |         if (signum == SIGUSR1){
19 |             break;
20 |         }
21 |     }
22 |     sleep(2);
23 |     printf("ending thread!\n");
24 |     pthread_exit(NULL);
25 | }
26 |
27 |
28 | void sig_handler(int signum){
29 | }
30 |
31 |
32 | int main() {
33 |
34 |     signal(SIGUSR1, sig_handler);
35 |     signal(SIGUSR2, SIG_IGN);
36 |     sigemptyset(&set);
37 |     sigaddset(&set, SIGTERM);
38 |     sigaddset(&set, SIGINT);
39 |     sigaddset(&set, SIGUSR1);
40 |     if(pthread_create(&thread1, NULL, thread_func, NULL)) {
41 |         printf("Error creating thread\n");
42 |         return 1;
43 |     }
44 |     printf("Main thread started. PID: %d\n", getpid());
45 |     int i = 0;
46 |     pthread_join(thread1, NULL);
47 |     printf("main is ending\n");
48 |     return 0;
49 | }
```

```

Main thread started. PID: 904. Waiting 3s before ending thread
Thread started. 905
Received signal 10
ending thread!
main is ending

```

Рисунок 4 – результат работы программы из листинга 5

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

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 из листинга 6 создаёт поток и ожидает его завершения, а программа 7 отправит потоку сигнал SIGUSR1 (в результате которого поток и после него процесс завершатся). Результат работы на рисунке 5.

Листинг 6 – программа, порождающая поток

```

File: 1.1.thread_creator.c
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <signal.h>
4  #include <pthread.h>
5  #include <unistd.h>
6  #define _GNU_SOURCE
7
8  pthread_t thread1;
9  sigset_t set;
10 int thread_id;
11 void *thread_func(void *args) {
12     thread_id = gettid();
13     printf("Thread started. %d\n", thread_id);
14     while(1) {
15         int signum;
16         sigwait(&set, &signum);
17         printf("Received signal %d\n", signum);
18         if (signum == SIGUSR1){
19             break;
20         }
21     }
22     sleep(2);
23     printf("ending thread!\n");
24     pthread_exit(NULL);
25 }
26
27
28 void sig_handler(int signum){
29 }
30
31
32 int main() {
33
34     signal(SIGUSR1, sig_handler);
35     signal(SIGUSR2, SIG_IGN);
36     sigemptyset(&set);

```

```

37     sigaddset(&set, SIGTERM);
38     sigaddset(&set, SIGINT);
39     sigaddset(&set, SIGUSR1);
40     if(pthread_create(&thread1, NULL, thread_func, NULL)) {
41         printf("Error creating thread\n");
42         return 1;
43     }
44
45     FILE* fp;
46     fp = fopen("tid_file.txt", "w");
47     char str[100];
48     if (fp == NULL){
49         perror("FILE");
50         exit(1);
51     }
52     sprintf(str, "%d", thread_id);
53     fprintf(fp, str);
54     fclose(fp);
55
56     printf("Waiting for signals to thread 1 from other source...\n");
57
58     pthread_join(thread1, NULL);
59     printf("main is ending\n");
60     return 0;
61 }

```

Листинг 7 - программа, отправляющая сигнал потоку

```

File: 1.1.signal_sender.c
1  #include <stdio.h>
2  #include <signal.h>
3  #include <unistd.h>
4  #include <stdlib.h>
5  #include <string.h>
6  // static void sigHandler(int sig)
7  // {
8  //     printf("Caught signal %s\n", sig == SIGUSR1 ? "SIGUSR1" : "SIGUSR2");
9  //     printf("%d", getpid());
10 // }
11
12 char* concat(const char *s1, const char *s2){
13     char* result = malloc(strlen(s1) + strlen(s2) + 1);
14     strcpy(result, s1);
15     strcat(result, s2);
16     return result;
17 }
18
19 int main()
20 {
21     FILE* fp;
22     fp = fopen("tid_file.txt", "r");
23     char str[100];
24     if (fp == NULL){
25         perror("FILE");
26         exit(1);
27     }
28     fgets(str, 100, fp);
29     //printf("%s\n", str);
30     char* s = concat("kill -10 ", str); //10 == SIGUSR1
31     // printf("string: [%s]\n", s);
32     printf("Отправляем сигнал!\n");
33     system(s);
34     fclose(fp);
35     free(s);
36     // ждем сигналов
37
38     return 0;
39 }

```

|   |  |
|---|--|
| golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter1\$ ./1.1.thread_creator | golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter1\$ batcat tid_file.txt |
| Thread started. 1363  | File: tid_file.txt   |
| Waiting for signals to thread 1 from other source ...             |  |
| Received signal 10  | 1 1363   |
| ending thread!  |  |
| main is ending  | golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter1\$ ./1.1.signal_sender |
| golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter1\$                      | Отправляем сигнал!   |
|   | golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter1\$                     |

Рисунок 5 – результат работы программы из листингов 6-7

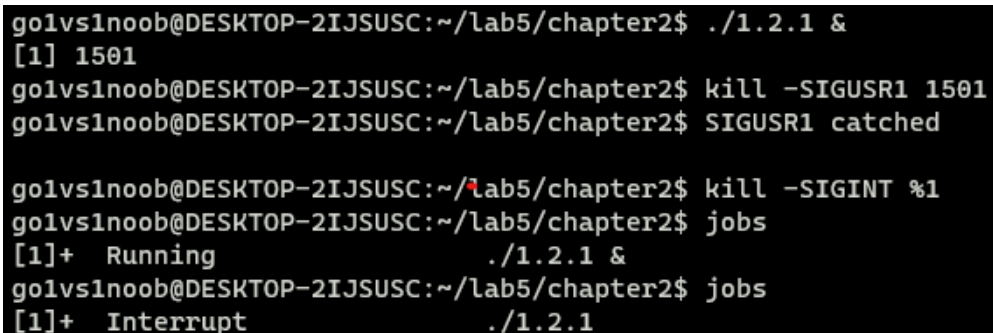


Программа отработала, как и ожидалось: поток принял сигнал, обработал его (завершился) и после него завершился основной процесс.

2. Создадим программу, позволяющую продемонстрировать возможность отложенной обработки (временного блокирования) сигнала SIGINT. Для решения задачи используем структуру sigaction. Блокировку реализуем, вызвав «засыпание» процесса из обработчика пользовательских сигналов. С рабочего терминала отправим процессу sigact сигнал SIGUSR1, а затем SIGINT. Содержимое программы в листинге 8. Результат работы на рисунке 6.

Листинг 8 – программа, демонстрирующая отложенную обработку сигнала

```
File: 1.2.1.c
1  #include <stdio.h>
2  #include <signal.h>
3  #include <sys/types.h>
4  #include <sys/stat.h>
5  #include <unistd.h>
6  #include <stdlib.h>
7  #include <fcntl.h>
8  void (*mysig(int sig, void (*hnd)(int)))(int)
9  {
10     // надежная обработка сигналов
11     struct sigaction act, oldact;
12     act.sa_handler = hnd;
13     sigemptyset(&act.sa_mask);
14     sigaddset(&act.sa_mask, SIGINT);
15     act.sa_flags = 0;
16     if (sigaction(sig, &act, 0) < 0)
17         return SIG_ERR;
18     return act.sa_handler;
19 }
20 void hndUSR1(int sig)
21 {
22     if (sig != SIGUSR1)
23     {
24         printf("Caught bad signal %d\n", sig);
25         return;
26     }
27     printf("SIGUSR1 caught\n");
28     sleep(10);
29 }
30 int main()
31 {
32     mysig(SIGUSR1, hndUSR1);
33     for (;;)
34     {
35         pause();
36     }
37     return 0;
38 }
```



```
golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter2$ ./1.2.1 &
[1] 1501
golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter2$ kill -SIGUSR1 1501
golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter2$ SIGUSR1 caught

golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter2$ kill -SIGINT %1
golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter2$ jobs
[1]+  Running                  ./1.2.1 &
golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter2$ jobs
[1]+  Interrupt                 ./1.2.1
```

Рисунок 6 – результат работы программы из листинга 8

В результате сигнал SIGUSR1 принят корректно, но после отсылки SIGINT программа работала ещё 10 секунд, и только после этого завершилась. В этом отличие надежной обработки сигналов от ненадежной: есть возможность отложить прием некоторых других сигналов. Отложенные таким образом сигналы записываются в маску PENDING и обрабатываются после завершения обработки сигналов, которые отложили обработку. Механизм ненадёжных сигналов не позволяет откладывать обработку других сигналов (можно лишь установить игнорирование некоторых сигналов на время обработки).

Изменим обработчик сигнала так, чтобы из него производилась отправка другого сигнала. Пусть из обработчика сигнала SIGUSR1 функцией kill() генерируется сигнал SIGINT. Проанализируем наличие и очередность обработки сигналов. Содержимое программы в листинге 9, результат работы на рисунке 7.

Листинг 9 – программа, демонстрирующая отложенную обработку сигнала, вызванного через другой сигнал

```
| File: 1.2.2.c
1  #include <stdio.h>
2  #include <signal.h>
3  #include <sys/types.h>
4  #include <sys/stat.h>
5  #include <unistd.h>
6  #include <stdlib.h>
7  #include <fcntl.h>
8  void (*mysig(int sig, void (*hnd)(int)))(int)
9  {
10     // надежная обработка сигналов
11     struct sigaction act, oldact;
12     act.sa_handler = hnd;
13     sigemptyset(&act.sa_mask);
14     sigaddset(&act.sa_mask, SIGINT);
15     act.sa_flags = 0;
16     if (sigaction(sig, &act, 0) < 0)
17         return SIG_ERR;
18     return act.sa_handler;
19 }
20 void hndUSR1(int sig)
21 {
22     if (sig != SIGUSR1)
23     {
24         printf("Caught bad signal %d\n", sig);
25         return;
26     }
27     printf("SIGUSR1 caught, sending SIGINT\n");
28     kill(getpid(), SIGINT);
29     sleep(10);
30 }
31 int main()
32 {
33     mysig(SIGUSR1, hndUSR1);
34     for (;;)
35     {
36         pause();
37     }
38     return 0;
39 }
```

```
golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter2$ ./1.2.2 &
[1] 1639
golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter2$ kill -SIGUSR1 1639
SIGUSR1 caught, sending SIGINT
golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter2$ jobs
[1]+  Running                  ./1.2.2 &
golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter2$ jobs
[1]+  Interrupt                 ./1.2.2
```

Рисунок 7 – результат работы программы из листинга 9

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

3. 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. Содержание программы в листинге 10, результат работы на рисунке 8.

Листинг 10 – программа, демонстрирующая организацию очереди сигналов.

```
File: 1.c
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <signal.h>
4  #include <unistd.h>
5
6  static int reg_flag = 0;
7  static int rt_flag = 0;
8  static int signal_counter = 0;
9
10
11 const char* switch_int_to_sigstr(int sig){
12     if (sig == SIGRTMIN){
13         return "SIGRTMIN";
14     }
15     else if (sig == SIGRTMIN+1){
16         return "SIGRTMIN+1";
17     }
18     else if (sig == SIGRTMIN+2){
19         return "SIGRTMIN+2";
20     }
21     else if (sig == SIGUSR1){
22         return "SIGUSR1";
23     }
24     else if (sig == SIGUSR2){
25         return "SIGUSR2";
26     }
27     return "UNKNOWN";
```

```

28 | }
29 |
30 | void handler(const int sig, siginfo_t* si, void* ucontext) {
31 |     printf("Received signal %s with value %d.\n", switch_int_to_sigstr(sig), si-
>si_value.sival_int);
32 |     int pid = getpid();
33 |     if (sig == SIGRTMIN){
34 |         sigqueue(pid, SIGUSR2, (union sigval){.sival_int = signal_counter++});
35 |     }
36 |     else if (sig == SIGUSR1){
37 |         sigqueue(pid, SIGRTMIN+1, (union sigval){.sival_int = signal_counter++});
38 |     }
39 |     else if (sig == SIGUSR2){
40 |         sigqueue(pid, SIGRTMIN+2, (union sigval){.sival_int = signal_counter++});
41 |     }
42 | }
43 |
44 | void (*mysig(int sig, void (*hnd)(int, siginfo_t*, void*)))(int, siginfo_t*, void*) {
45 |     // надежная обработка сигналов
46 |     struct sigaction act, oldact;
47 |     act.sa_sigaction = hnd;
48 |     sigemptyset(&act.sa_mask);
49 |     sigaddset(&act.sa_mask, SIGINT);
50 |     act.sa_flags = SA_SIGINFO;
51 |     if (sigaction(sig, &act, &oldact) < 0) {
52 |         perror("sigaction");
53 |         exit(-1);
54 |     }
55 |     return oldact.sa_sigaction;
56 | }
57 |
58 |
59 | int main(void)
60 | {
61 |
62 |     mysig(SIGUSR1, handler);
63 |     mysig(SIGUSR2, handler);
64 |     mysig(SIGRTMIN, handler);
65 |     mysig(SIGRTMIN+1, handler);
66 |     mysig(SIGRTMIN+2, handler);
67 |
68 |     int pid = getpid();
69 |     int signal;
70 |     for (int i = 0; i < 10; i++){
71 |         signal = (i%2 == 0) ? SIGRTMIN : SIGUSR1;
72 |         sigqueue(pid, signal, (union sigval){.sival_int = signal_counter++});
73 |     }
74 |
75 |
76 |     sleep(5); // Даем время обработчикам сигналов выполнить свою работу
77 |     printf("Завершение программы.\n");
78 |     return EXIT_SUCCESS;
79 | }

```

---

```

golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter3$ ./1
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 8 – результат работы программы из листинга 10*

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

Экспериментально подтвердим, что обработка равноприоритетных сигналов реального времени происходит в порядке FIFO. Для этого напишем программу, в которой обработчик сигналов будет выводить номер сигнала, его порядковый номер (будем использовать SIGRTMIN). В головной процедуре программа ожидает сигналов извне. Для отправки сигналов напишем скрипт, который 10 раз отправит сигнал SIGRTMIN. Содержимое программы и скрипта в листинге 11, результат работы на рисунке 9.

*Листинг 11 – программа, демонстрирующая обработку равноприоритетных сигналов и скрипт, отправляющий сигналы.*

---

```

File: 2_console.c

```

---

```

1 | #include <signal.h>
2 | #include <stdio.h>
3 | #include <unistd.h>
4 | #include <stdlib.h>

```

```

5 |
6 | #define AMOUNT_OF_SIGNALS 10
7 |
8 | static volatile int counter = 0;
9 | int sig_iter = 0;
10 |
11 | void handler(int sig, siginfo_t* si, void* ucontext) {
12 |     printf("Received signal %d with value %d.\n", sig, si->si_value.sival_int);
13 |     sleep(1);
14 | }
15 |
16 | void (*mysig(int sig, void (*hnd)(int, siginfo_t*, void*)))(int, siginfo_t*, void*) {
17 |     // надежная обработка сигналов
18 |     struct sigaction act, oldact;
19 |     act.sa_sigaction = hnd;
20 |     sigemptyset(&act.sa_mask);
21 |     sigaddset(&act.sa_mask, SIGINT);
22 |     act.sa_flags = SA_SIGINFO;
23 |     if (sigaction(sig, &act, &oldact) < 0) {
24 |         perror("sigaction");
25 |         exit(-1);
26 |     }
27 |     return oldact.sa_sigaction;
28 | }
29 |
30 | int main() {
31 |     printf("PID: %d\n", getpid());
32 |     mysig(SIGUSR1, handler);
33 |     mysig(SIGRTMIN, handler);
34 |     for (;;) {
35 |         pause();
36 |     }
37 |     return 0;
38 | }

```

---

File: script\_2.sh

---

```

1 | #!/bin/bash
2 |
3 | signal_val=34
4 | pid=$1
5 | val_counter=0
6 | val_counter_sigusr1=0
7 |
8 |
9 | for (( counter=0; counter<10; counter++ ))
10 | do
11 |     echo `sudo kill -34 $pid -q $val_counter`
12 |     ((val_counter=val_counter+1))
13 | done

```

---

|  |   |
|--|---|
| <pre>golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter3\$ ./2_console PID: 384 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.</pre> | <pre>golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter3\$ clear golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter3\$ bash script_2.sh 384</pre> |
|--|---|

Рисунок 9 – результат работы программы из листинга 11

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

Опытным путём подтвердим наличие приоритета сигналов реального времени над простыми сигналами. Для этого используем программу из листинга 11, но изменим скрипт, посылающий сигналы. Сначала отправим два простых сигнала SIGUSR1, а потом – 10 сигналов реального времени. Содержание скрипта в листинге 12, результат работы – на рисунке 10.

Листинг 12 – изменённый скрипт для посылки сигналов.

|    |   |
|----|---|
|    | File: script.sh                             |
| 1  | #!/bin/bash                                 |
| 2  |   |
| 3  | signal_val=34                               |
| 4  | pid=\$1                                     |
| 5  | val_counter=0                               |
| 6  |   |
| 7  | echo `sudo kill -10 \$pid -q 1`             |
| 8  | echo `sudo kill -10 \$pid -q 2`             |
| 9  |   |
| 10 | for (( counter=0; counter<10; counter++ ))  |
| 11 | do  |
| 12 | echo `sudo kill -34 \$pid -q \$val_counter` |
| 13 | ((val_counter=val_counter+1))               |
| 14 | done  |

|   |   |
|---|---|
| <pre>golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter3\$ ./2_console PID: 1005 Received signal 10 with value 1. 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 2.</pre> | <pre>golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter3\$ bash script.sh 1005</pre> |
|---|---|

Рисунок 9 – результат работы программы из листинга 12

Как видно из рисунка, 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.

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 (`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. Содержимое программы в листинге 13. Результат работы на рисунке 10.

Листинг 13 – программа для записи в файл с помощью `pipe()`

|    | File: pipe.c  |
|----|---|
| 1  | <code>#include &lt;stdio.h&gt;</code>   |
| 2  | <code>#include &lt;stdlib.h&gt;</code>  |
| 3  | <code>#include &lt;unistd.h&gt;</code>  |
| 4  | <code>#include &lt;string.h&gt;</code>  |
| 5  | <code>#define DEF_F_R "from.txt"</code>   |
| 6  | <code>#define DEF_F_W "to.txt"</code>   |
| 7  | <code>int main(int argc, char **argv)</code>  |
| 8  | <code>{</code>  |
| 9  | <code>    char fileToRead[32];</code>   |
| 10 | <code>    char fileToWrite[32];</code>  |
| 11 | <code>    if (argc &lt; 3)</code>   |
| 12 | <code>    {</code>  |
| 13 | <code>        printf("Using default fileNames '%s','%s'\n", DEF_F_R, DEF_F_W);</code> |
| 14 | <code>        strcpy(fileToRead, DEF_F_R);</code>                                     |
| 15 | <code>        strcpy(fileToWrite, DEF_F_W);</code>                                    |
| 16 | <code>    }</code>  |
| 17 | <code>    else</code>   |
| 18 | <code>    {</code>  |
| 19 | <code>        strcpy(fileToRead, argv[1]);</code>                                     |
| 20 | <code>        strcpy(fileToWrite, argv[2]);</code>                                    |
| 21 | <code>    }</code>  |
| 22 | <code>    int filedes[2];</code>  |
| 23 | <code>    if (pipe(filedes) &lt; 0)</code>  |
| 24 | <code>    {</code>  |
| 25 | <code>        printf("Father: can't create pipe\n");</code>                           |
| 26 | <code>        exit(1);</code>   |
| 27 | <code>    }</code>  |
| 28 | <code>    printf("pipe is successfully created\n");</code>                            |
| 29 | <code>    if (fork() == 0)</code>   |
| 30 | <code>    {</code>  |
| 31 | <code>        // процесс сын</code>   |
| 32 | <code>        // закрывает пайп для чтения</code>                                     |
| 33 | <code>        close(filedes[0]);</code>   |
| 34 | <code>        FILE *f = fopen(fileToRead, "r");</code>                                |
| 35 | <code>        if (!f)</code>  |



```

36     {
37         printf("Son: cant open file %s\n", fileToRead);
38         exit(1);
39     }
40     char buf[100];
41     int res;
42     while (!feof(f))
43     {
44         // читаем данные из файла
45         res = fread(buf, sizeof(char), 100, f);
46         write(filedes[1], buf, res); // пишем их в пайп
47     }
48     close(f);
49     close(filedes[1]);
50     return 0;
51 }
52 // процесс отец
53 // закрывает пайп для записи
54 close(filedes[1]);
55 FILE *f = fopen(fileToWrite, "w");
56 if (!f)
57 {
58     printf("Father: cant open file %s\n", fileToWrite);
59     exit(1);
60 }
61 char buf[100];
62 int res;
63 while (1)
64 {
65     bzero(buf, 100);
66     res = read(filedes[0], buf, 100);
67     if (!res)
68         break;
69     printf("Read from pipe: %s\n", buf);
70     fwrite(buf, sizeof(char), res, f);
71 }
72 fclose(f);
73 close(filedes[0]);
74 return 0;
75 }

```

```

golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter3$ ./pipe
Using default fileNames 'from.txt','to.txt'
pipe is successfully created
Read from pipe: str1
str2
str3
Hello world!

golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter3$ cat to.txt
str1
str2
str3
Hello world!

golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter3$ cat from.txt
str1
str2
str3
Hello world!

```

Рисунок 10 – результат работы программы из листинга 13

Как видно из рисунка 10, содержимое файла from.txt переписалось в файл to.txt с использованием неименованного канала. Поскольку процесс-родитель только

читает из канала, то дескриптор для записи `filedes[1]` он закрывает, аналогично процесс-сын закрывает дескриптор для чтения `filedes[0]`.

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

Создадим клиент-серверное приложение, демонстрирующее дуплексную (двунаправленную) передачу информации двумя однонаправленными именованными каналами между клиентом и сервером. В файле `server.c` создадим 2 именованных канала с помощью `mknod()`, аргументы которого: имя файла FIFO в файловой системе; флаги владения, прав доступа (установим открытые для всех права доступа на чтение и на запись `S_IFIFO | 0666`). Откроем один канал на запись (`chan1`), другой — на чтение (`chan2`) и запустим серверную часть программы. В серверной части программы: запишем имя файла в канал 1 (для записи) функцией `write()`; прочитаем данные из канала 2 и выведем на экран. В файле `client.c` запрограммируем функции: открытия каналов для чтения (`chan1`) и записи (`chan2`). Из первого канала читается имя файла, во второй канал пишется его содержимое. Содержимое программ в листинге 14. Результат работы на рисунке 11.

Листинг 14 – серверная и клиентские части клиент-серверного приложения

|   | File: server.c                                   |
|---|--|
| 1 | <code>#include &lt;stdio.h&gt;</code>            |
| 2 | <code>#include &lt;stdlib.h&gt;</code>           |
| 3 | <code>#include &lt;unistd.h&gt;</code>           |
| 4 | <code>#include &lt;string.h&gt;</code>           |
| 5 | <code>#include &lt;sys/types.h&gt;</code>        |
| 6 | <code>#include &lt;sys/stat.h&gt;</code>         |
| 7 | <code>#include &lt;fcntl.h&gt;</code>            |
| 8 | <code>#define DEF_FILENAME "testFile.txt"</code> |
| 9 | <code>int main(int argc, char **argv)</code>     |

```

10 | {
11 |     char fileName[30];
12 |     if (argc < 2)
13 |     {
14 |         printf("Using default file name '%s'\n", DEF_FILENAME);
15 |         strcpy(fileName, DEF_FILENAME);
16 |     }
17 |     else
18 |         strcpy(fileName, argv[1]);
19 |     // создаем два канала
20 |     int res = mknod("channel1", S_IFIFO | 0666, 0);
21 |     if (res)
22 |     {
23 |         printf("Can't create first channel\n");
24 |         exit(1);
25 |     }
26 |     res = mknod("channel2", S_IFIFO | 0666, 0);
27 |     if (res)
28 |     {
29 |         printf("Can't create second channel\n");
30 |         exit(1);
31 |     }
32 |     // открываем первый канал для записи
33 |     int chan1 = open("channel1", O_WRONLY);
34 |     if (chan1 == -1)
35 |     {
36 |         printf("Can't open channel for writing\n");
37 |         exit(0);
38 |     }
39 |     // открываем второй канал для чтения
40 |     int chan2 = open("channel2", O_RDONLY);
41 |     if (chan2 == -1)
42 |     {
43 |         printf("Can't open channel2 for reading\n");
44 |         exit(0);
45 |     }
46 |     // пишем имя файла в первый канал
47 |     write(chan1, fileName, strlen(fileName));
48 |     // читаем содержимое файла из второго канала
49 |     char buf[100];
50 |     for (;;)
51 |     {
52 |         bzero(buf, 100);
53 |         res = read(chan2, buf, 100);
54 |         if (res <= 0)
55 |             break;
56 |         printf("Part of file: %s\n");
57 |     }
58 |     close(chan1);
59 |     close(chan2);
60 |     unlink("channel1");
61 |     unlink("channel2");
62 |     return 0;
63 | }

```

---

File: client.c

---

```

1 | #include <stdio.h>
2 | #include <stdlib.h>
3 | #include <unistd.h>
4 | #include <string.h>
5 | #include <sys/types.h>
6 | #include <sys/stat.h>
7 | #include <fcntl.h>
8 |
9 | int main()
10 | {
11 |     // каналы сервер уже создал, открываем их
12 |     int chan1 = open("channel1", O_RDONLY);
13 |     if (chan1 == -1)
14 |     {
15 |         printf("Can't open channel1 for reading\n");
16 |         exit(0);
17 |     }
18 |     int chan2 = open("channel2", O_WRONLY);
19 |     if (chan2 == -1)
20 |     {
21 |         printf("Can't open channel2 for reading\n");
22 |         exit(0);

```

```

23     }
24     // читаем имя файла из первого канала
25     char fileName[100];
26     bzero(fileName, 100);
27     int res = read(chan1, fileName, 100);
28     if (res <= 0)
29     {
30         printf("Can't read fileName from channel1\n");
31         exit(0);
32     }
33     // открываем файл на чтение
34     FILE *f = fopen(fileName, "r");
35     if (!f)
36     {
37         printf("Can't open file %s\n", fileName);
38         exit(0);
39     }
40     // читаем из файла и пишем во второй канал
41     char buf[100];
42     while (!feof(f))
43     {
44         // читаем данные из файла
45         res = fread(buf, sizeof(char), 100, f);
46         // пишем их в канал
47         write(chan2, buf, res);
48     }
49     fclose(f);
50     close(chan1);
51     close(chan2);
52     return 0;
53 }

```

|   |   |
|---|---|
| <pre> golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter3\$ ./server Using default file name 'testFile.txt' Part of file: str1 str2 str3 hello world! </pre> | <pre> golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter3\$ ./client golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter3\$ </pre> |
|---|---|

Рисунок 11 – результат работы программы из листинга 14

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()`.

Несмотря на то, что именованные каналы являются отдельным типом файлов и могут быть видимы разными процессами даже в распределенной файловой системе, использование FIFO для взаимодействия удаленных процессов и обмена информацией между ними невозможно. Так как и в этом случае для передачи данных задействовано ядро. Создаваемый файл служит для получения данных о расположении FIFO в адресном пространстве ядра и его состоянии. Продемонстрируем это на примере. Изменим ранее использованную

программу так, чтобы сервер, перед тем как читать данные из канала, ожидал ввода пользователя. Исходный код клиента оставим неизменным. Новый код сервера в листинге 15. Результат работы на рисунке 12.

Листинг 15 - изменённый файл сервера

|    | File: server_edited.c                                   |
|----|---|
| 1  | #include <stdio.h>                                      |
| 2  | #include <stdlib.h>                                     |
| 3  | #include <unistd.h>                                     |
| 4  | #include <string.h>                                     |
| 5  | #include <sys/types.h>                                  |
| 6  | #include <sys/stat.h>                                   |
| 7  | #include <fcntl.h>                                      |
| 8  | #define DEF_FILENAME "testFile.txt"                     |
| 9  | int main(int argc, char **argv)                         |
| 10 | {   |
| 11 | char fileName[30];                                      |
| 12 | if (argc < 2)   |
| 13 | {   |
| 14 | printf("Using default file name '%s'\n", DEF_FILENAME); |
| 15 | strcpy(fileName, DEF_FILENAME);                         |
| 16 | }   |
| 17 | else  |
| 18 | strcpy(fileName, argv[1]);                              |
| 19 | // создаем два канала                                   |
| 20 | int res = mknod("channel1", S_IFIFO   0666, 0);         |
| 21 | if (res)  |
| 22 | {   |
| 23 | printf("Can't create first channel\n");                 |
| 24 | exit(1);  |
| 25 | }   |
| 26 | res = mknod("channel2", S_IFIFO   0666, 0);             |
| 27 | if (res)  |
| 28 | {   |
| 29 | printf("Can't create second channel\n");                |
| 30 | exit(1);  |
| 31 | }   |
| 32 | // открываем первый канал для записи                    |
| 33 | int chan1 = open("channel1", O_WRONLY);                 |
| 34 | if (chan1 == -1)  |
| 35 | {   |
| 36 | printf("Can't open channel for writing\n");             |
| 37 | exit(0);  |
| 38 | }   |
| 39 | // открываем второй канал для чтения                    |
| 40 | int chan2 = open("channel2", O_RDONLY);                 |
| 41 | if (chan2 == -1)  |
| 42 | {   |
| 43 | printf("Can't open channe2 for reading\n");             |
| 44 | exit(0);  |
| 45 | }   |
| 46 | // пишем имя файла в первый канал                       |
| 47 | write(chan1, fileName, strlen(fileName));               |
| 48 | // читаем содержимое файла из второго канала            |
| 49 | char buf[100];  |
| 50 | printf("Waiting for clint write to channnel\n");        |
| 51 | getchar();  |
| 52 | for (;;)  |
| 53 | {   |
| 54 | bzero(buf, 100);  |
| 55 | res = read(chan2, buf, 100);                            |
| 56 | if (res <= 0)   |
| 57 | break;  |
| 58 | printf("Part of file: %s\n");                           |
| 59 | }   |
| 60 | close(chan1);   |
| 61 | close(chan2);   |
| 62 | unlink("channel1");                                     |
| 63 | unlink("channel2");                                     |
| 64 | printf("Servr finished\n");                             |
| 65 | return 0;   |
| 66 | }   |

```

golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter3$ ./server_edited
Using default file name 'testFile.txt'
Waiting for clint write to channnel

Part of file: str1
str2
str3
hello world!
Servr finished

golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter3$ ./client
golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter3$

golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter3$ ls -sl | grep chan
0 prw-r--r-- 1 golvs1noob golvs1noob 0 Apr 26 14:11 channel1
0 prw-r--r-- 1 golvs1noob golvs1noob 0 Apr 26 14:11 channel2

```

Рисунок 12 – результат работы программы с сервером из листинга 15

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

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

```

golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter6$ gcc dummy.c
golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter6$ ./a.out
max amount of files that process can have open: 200809
PIPE_BUF: 4096

```

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 неименованных 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 неименованном 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.

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()`. Исходный код в листинге 16, результат работы на рисунке 13.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/ipc.h>
4  #include <sys/msg.h>
5  #include <sys/types.h>
6  #include <signal.h>
7  #define DEF_KEY_FILE "key"
8  typedef struct
9  {
10     long type;
11     char buf[100];
12 } Message;
13 int queue;
14 void intHandler(int sig)
15 {
16     signal(sig, SIG_DFL);
17     if (msgctl(queue, IPC_RMID, 0) < 0)
18     {
19         printf("Can't delete queue\n");
20         exit(1);
21     }
22 }
23 int main(int argc, char **argv)
24 {
25     char keyFile[100];
26     bzero(keyFile, 100);
27     if (argc < 2)
28     {
29         printf("Using default key file %s\n", DEF_KEY_FILE);
30         strcpy(keyFile, DEF_KEY_FILE);
31     }
32     else
33         strcpy(keyFile, argv[1]);
34     key_t key;
35     key = ftok(keyFile, 'Q');
36     if (key == -1)
37     {
38         printf("no got key for the key file %s and id 'Q'\n", keyFile);
39         exit(1);
40     }
41     queue = msgget(key, IPC_CREAT | 0666);
42     if (queue < 0)
43     {
44         printf("Can't create queue\n");
45         exit(4);
46     }
47     // до этого момента вызывали exit(), а не kill, т.к. очередь
48     // еще не была создана
49     signal(SIGINT, intHandler);
50     // основной цикл работы сервера
51     Message mes;
52     int res;
53     for (;;)
54     {
55         bzero(mes.buf, 100);
56         // получаем первое сообщение с типом 1
57         res = msgrcv(queue, &mes, sizeof(Message), 1L, 0);
58         if (res < 0)
59         {
60             printf("Error while recving msg\n");
61             kill(getpid(), SIGINT);
62         }
63         printf("Client's request: %s\n", mes.buf);
64         // шлем клиенту сообщение с типом 2, что все ок
65         mes.type = 2L;
66         bzero(mes.buf, 100);
67         strcpy(mes.buf, "OK");
68         res = msgsnd(queue, (void *)&mes, sizeof(Message), 0);
69         if (res != 0)
70         {
71             printf("error while sending msg\n");
72             kill(getpid(), SIGINT);
73         }
74     }
75     return 0;
76 }

```



```

File: named_channels_client.c
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/ipc.h>
4  #include <sys/msg.h>
5  #include <sys/types.h>
6  #include <signal.h>
7  #define DEF_KEY_FILE "key"
8  typedef struct
9  {
10     long type;
11     char buf[100];
12 } Message;
13 int queue;
14 int main(int argc, char **argv)
15 {
16     char keyFile[100];
17     bzero(keyFile, 100);
18     if (argc < 2)
19     {
20         printf("Using default key file %s\n", DEF_KEY_FILE);
21         strcpy(keyFile, DEF_KEY_FILE);
22     }
23     else
24         strcpy(keyFile, argv[1]);
25     key_t key;
26     key = ftok(keyFile, 'Q');
27     if (key == -1)
28     {
29         printf("no got key for key file %s and id 'Q'\n", keyFile);
30         exit(1);
31     }
32     queue = msgget(key, 0);
33     if (queue < 0)
34     {
35         printf("Can't create queue\n");
36         exit(4);
37     }
38     // основной цикл работы программы
39     Message mes;
40     int res;
41     for (;;)
42     {
43         bzero(mes.buf, 100);
44         // читаем сообщение с консоли
45         fgets(mes.buf, 100, stdin);
46         mes.buf[strlen(mes.buf) - 1] = '\0';
47         // шлем его серверу
48         mes.type = 1L;
49         res = msgsnd(queue, (void *)&mes, sizeof(Message), 0);
50         if (res != 0)
51         {
52             printf("Error while sending msg\n");
53             exit(1);
54         }
55         // получаем ответ, что все хорошо
56         res = msgrcv(queue, &mes, sizeof(Message), 2L, 0);
57         if (res < 0)
58         {
59             printf("Error while recving msg\n");
60             exit(1);
61         }
62         printf("Server's response: %s\n", mes.buf);
63     }
64     return 0;
65 }

```

|  |   |
|--|---|
| <pre>./named_channels_client Using default key file key hello from client 1 Server's response: OK hello! Server's response: OK</pre>     | <pre>golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter3\$ ./named_channels_server Using default key file key Client's request: hello from client 2! Client's request: hello!!! Client's request: hello from client 1 Client's request: hello!</pre> |
| <pre>./named_channels_client2 Using default key file key hello from client 2! Server's response: OK hello!!! Server's response: OK</pre> |   |

Рисунок 13 – результат работы программы с сервером из листинга 16

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 «OK».

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.

Максимальные и минимальные значения констант выясним с помощью утилиты `ipcs -l`. Результат на рисунке 14.

```
golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter3$ 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
```

Из рисунка 14 видно, что, например, размер одного сообщения не может быть больше 8192 байт, а очередь может содержать не более 32000 сообщений в один момент времени.

**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 прочитать. 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 прочитать. 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. Содержание программы в листинге 17, результат работы на рисунке 15.

Листинг 17 -

```
File: semaphores_reader.c
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <signal.h>
4  #include <unistd.h>
5  #include <sys/types.h>
6  #include <sys/ipc.h>
7  #include <sys/sem.h>
8  #include <sys/shm.h>
9  #include <sys/time.h>
10 #include "shm.h"
11 Message *p_msg;
12 int shmmemory;
13 int semaphore;
14 void intHandler(int sig)
15 {
16     // отключаем разделяемую память
17     if (shmdt(p_msg) < 0)
18     {
19         printf("Error while detaching shm\n");
20         exit(1);
21     }
22     // удаляем shm и семафоры
23     if (shmctl(shmmemory, IPC_RMID, 0) < 0)
24     {
25         printf("Error while deleting shm\n");
26         exit(1);
27     }
28     if (semctl(semaphore, 0, IPC_RMID) < 0)
29     {
30         printf("Error while deleting semaphore\n");
31         exit(1);
32     }
33 }
34 int main(int argc, char **argv)
35 {
36     char keyFile[100];
37     bzero(keyFile, 100);
38     if (argc < 2)
39     {
40         printf("Using default key file %s\n", DEF_KEY_FILE);
41         strcpy(keyFile, DEF_KEY_FILE);
42     }
43     else
44         strcpy(keyFile, argv[1]);
45     key_t key;
46     // будем использовать 1 и тот же ключ для семафора и для shm
47     if ((key = ftok(keyFile, 'Q')) < 0)
48     {
49         printf("Can't get key for key file %s and id 'Q'\n", keyFile);
50         exit(1);
51     }
52     // создаем shm
53     if ((shmmemory = shmget(key, sizeof(Message), IPC_CREAT | 0666)) < 0)
54     {
55         printf("Can't create shm\n");
56         exit(1);
57     }
58     // присоединяем shm в наше адресное пространство
59     if ((p_msg = (Message *)shmat(shmmemory, 0, 0)) < 0)
60     {
61         printf("Error while attaching shm\n");
62         exit(1);
63     }
64     // устанавливаем обработчик сигнала
65     signal(SIGINT, intHandler);
```

```

66 // создаем группу из 2 семафоров
67 // 1 - показывает, что можно читать
68 // 2 - показывает, что можно писать
69 if ((semaphore = semget(key, 2, IPC_CREAT | 0666)) < 0)
70 {
71     printf("Error while creating semaphore\n");
72     kill(getpid(), SIGINT);
73 }
74 // устанвливаем 2 семафор в 1, т.е. можно писать
75 if (semop(semaphore, setWriteEna, 1) < 0)
76 {
77     printf("execution complete\n");
78     kill(getpid(), SIGINT);
79 }
80 // основной цикл работы
81 for (;;)
82 {
83     // ждем пока клиент начнет работу
84     if (semop(semaphore, readEna, 1) < 0)
85     {
86         printf("execution complete\n");
87         kill(getpid(), SIGINT);
88     }
89     // читаем сообщение от клиента
90     printf("Client's message: %s", p_msg->buf);
91     // говорим клиенту, что можно снова писать
92     if (semop(semaphore, setWriteEna, 1) < 0)
93     {
94         printf("execution complete\n");
95         kill(getpid(), SIGINT);
96     }
97 }
98 }

```

---

File: semaphores\_writer.c

---

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/types.h>
4  #include <sys/ipc.h>
5  #include <sys/sem.h>
6  #include <sys/shm.h>
7  #include "shm.h"
8
9  int main(int argc, char **argv)
10 {
11     Message *p_msg;
12     char keyFile[100];
13     bzero(keyFile, 100);
14     if (argc < 2)
15     {
16         printf("Using default key file %s\n", DEF_KEY_FILE);
17         strcpy(keyFile, DEF_KEY_FILE);
18     }
19     else
20         strcpy(keyFile, argv[1]);
21     key_t key;
22     int shmemory;
23     int semaphore;
24     // будем использовать 1 и тот же ключ для семафора и для shm
25     if ((key = ftok(keyFile, 'Q')) < 0)
26     {
27         printf("Can't get key for key file %s and id 'Q'\n", keyFile);
28         exit(1);
29     }
30     // создаем shm
31     if ((shmemory = shmget(key, sizeof(Message), 0666)) < 0)
32     {
33         printf("Can't create shm\n");
34         exit(1);
35     }
36     // присоединяем shm в наше адресное пространство
37     if ((p_msg = (Message *)shmat(shmemory, 0, 0)) < 0)
38     {
39         printf("Error while attaching shm\n");
40         exit(1);
41     }
42     if ((semaphore = semget(key, 2, 0666)) < 0)
43     {

```

```

44         printf("Error while creating semaphore\n");
45         exit(1);
46     }
47     char buf[100];
48     for (;;)
49     {
50         bzero(buf, 100);
51         printf("Type message to serever. Empty string to finish\n");
52         fgets(buf, 100, stdin);
53         if (strlen(buf) == 1 && buf[0] == '\n')
54         {
55             printf("bye-bye\n");
56             exit(0);
57         }
58         // хотим отправить сообщение
59         if (semop(semaphore, writeEna, 1) < 0)
60         {
61             printf("Can't execute a operation\n");
62             exit(1);
63         }
64         // запись сообщения в разделяемую память
65         sprintf(p_msg->buf, "%s", buf);
66         // говорим серверу, что он может читать
67         if (semop(semaphore, setReadEna, 1) < 0)
68         {
69             printf("Can't execute a operation\n");
70             exit(1);
71         }
72     }
73     // отключение от области разделяемой памяти
74     if (shmdt(p_msg) < 0)
75     {
76         printf("Error while detaching shm\n");
77         exit(1);
78     }
79 }

```

|   |  |
|---|--|
| <pre> root@DESKTOP-2IJSUSC:/home/golvs1noob/lab5/chapter3# ./semaphores_writer Using default key file key Type message to serever. Empty string to finish abc Type message to serever. Empty string to finish  bye-bye root@DESKTOP-2IJSUSC:/home/golvs1noob/lab5/chapter3# </pre>  | <pre> golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter3\$ ./semaphores_reader Using default key file key Client's message: qwe Client's message: abc Client's message: hellooo  ^Cexection complete Error while detaching shm golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter3\$ </pre> |
| <pre> golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter3\$ ./semaphores_writer1 Using default key file key Type message to serever. Empty string to finish qwe Type message to serever. Empty string to finish hellooo Type message to serever. Empty string to finish  bye-bye golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter3\$ </pre> |  |

Рисунок 15 –

Как видно из рисунка 15, все сообщения от клиента сервером прочитаны.

**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).

**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.

**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 TCP сокеты, это значит, что между сервером и клиентом устанавливается логическое соединение, при этом при получении данных из сокета с помощью вызова `recv`, есть вероятность получить сразу несколько сообщений, или не полностью прочитать сообщение. Поэтому для установления взаимной однозначности между отосланными и принятыми данными используются функции `recvFix` и `sendFix`. Принцип их работы следующий: функция `sendFix` перед посылкой собственно данных посылает

«заголовок» - количество байт в посылке. Функция `recvFix` вначале принимает этот «заголовок», и вторым вызовом `recv` считывает переданное количество байт. Считать ровно то, количество байт, которое 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 в листинге 18. Результат работы на рисунке 16-17.

|    | File: server.c   |
|----|--|
| 1  | <code>#include &lt;sys/types.h&gt;</code>  |
| 2  | <code>#include &lt;sys/socket.h&gt;</code>                                       |
| 3  | <code>#include &lt;netinet/in.h&gt;</code>                                       |
| 4  | <code>#include &lt;arpa/inet.h&gt;</code>  |
| 5  | <code>#include &lt;stdio.h&gt;</code>  |
| 6  | <code>#include &lt;stdlib.h&gt;</code>   |
| 7  | <code>#include &lt;string.h&gt;</code>   |
| 8  | <code>#include &lt;unistd.h&gt;</code>   |
| 9  | <code>#define DEF_PORT 8888</code>   |
| 10 | <code>#define DEF_IP "127.0.0.1"</code>  |
| 11 | <code>// обработка одного клиента</code>   |
| 12 |  |
| 13 | <code>void *</code>  |
| 14 | <code>clientHandler(void *args)</code>   |
| 15 | <code>{</code>   |
| 16 | <code>    int sock = (int)args;</code>   |
| 17 | <code>    char buf[100];</code>  |
| 18 | <code>    int res = 0;</code>  |
| 19 | <code>    for (;;) </code>   |
| 20 | <code>    {</code>   |
| 21 | <code>        bzero(buf, 100);</code>  |
| 22 | <code>        res = readFix(sock, buf, 100, 0);</code>                           |
| 23 | <code>        if (res &lt;= 0)</code>  |
| 24 | <code>        {</code>   |
| 25 | <code>            perror("Can't recv data from client, ending thread\n");</code> |
| 26 | <code>            pthread_exit(NULL);</code>                                     |
| 27 | <code>        }</code>   |
| 28 | <code>        printf("Some client sent: %s\n", buf);</code>                      |
| 29 | <code>        res = sendFix(sock, buf, 0);</code>                                |
| 30 | <code>        if (res &lt;= 0)</code>  |
| 31 | <code>        {</code>   |
| 32 | <code>            perror("send call failed");</code>                             |
| 33 | <code>            pthread_exit(NULL);</code>                                     |
| 34 | <code>        }</code>   |
| 35 | <code>    }</code>   |
| 36 | <code>}</code>   |
| 37 | <code>int main(int argc, char **argv)</code>                                     |
| 38 | <code>{</code>   |
| 39 | <code>    int port = 0;</code>   |
| 40 | <code>    if (argc &lt; 2)</code>  |
| 41 | <code>    {</code>   |
| 42 | <code>        printf("Using default port %d\n", DEF_PORT);</code>                |
| 43 | <code>        port = DEF_PORT;</code>  |
| 44 | <code>    }</code>   |
| 45 | <code>    else</code>  |
| 46 | <code>        port = atoi(argv[1]);</code>                                       |
| 47 | <code>    struct sockaddr_in listenerInfo;</code>                                |
| 48 | <code>    listenerInfo.sin_family = AF_INET;</code>                              |
| 49 | <code>    listenerInfo.sin_port = htons(port);</code>                            |
| 50 | <code>    listenerInfo.sin_addr.s_addr = htonl(INADDR_ANY);</code>               |
| 51 | <code>    int listener = socket(AF_INET, SOCK_STREAM, 0);</code>                 |
| 52 | <code>    if (listener &lt; 0)</code>  |



```

53     {
54         perror("Can't create socket to listen: ");
55         exit(1);
56     }
57     int res = bind(listener, (struct sockaddr *)&listenerInfo, sizeof(listenerInfo));
58     if (res < 0)
59     {
60         perror("Can't bind socket");
61         exit(1);
62     }
63     // слушаем входящие соединения
64     res = listen(listener, 5);
65
66     if (res)
67     {
68         perror("Error while listening:");
69         exit(1);
70     }
71     // основной цикл работы
72     for (;;)
73     {
74         int client = accept(listener, NULL, NULL);
75         pthread_t thrd;
76         res = pthread_create(&thrd, NULL, clientHandler, (void *) (client));
77         if (res)
78         {
79             printf("Error while creating new thread\n");
80         }
81     }
82     return 0;
83 }
84 int readFix(int sock, char *buf, int bufSize, int flags)
85 {
86     // читаем "заголовок" - сколько байт составляет наше сообщение
87     unsigned msgLength = 0;
88     int res = recv(sock, &msgLength, sizeof(unsigned), flags | MSG_WAITALL);
89     if (res <= 0)
90         return res;
91     if (res > bufSize)
92     {
93         printf("Recieved more data, then we can store, exiting\n");
94         exit(1);
95     }
96     // читаем само сообщение
97     return recv(sock, buf, msgLength, flags | MSG_WAITALL);
98 }
99 int sendFix(int sock, char *buf, int flags)
100 {
101     // шлем число байт в сообщении
102     unsigned msgLength = strlen(buf);
103     int res = send(sock, &msgLength, sizeof(unsigned), flags);
104     if (res <= 0)
105         return res;
106     send(sock, buf, msgLength, flags);
107 }

```

---

File: client.c

---

```

1  #include <sys/types.h>
2  #include <sys/socket.h>
3  #include <netinet/in.h>
4  #include <arpa/inet.h>
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <string.h>
8  #include <unistd.h>
9  #define DEF_PORT 8888
10 #define DEF_IP "127.0.0.1"
11 int main(int argc, char **argv)
12 {
13     char *addr;
14     int port;
15     char *readbuf;
16     printf("Using default port %d\n", DEF_PORT);
17     port = DEF_PORT;
18     printf("Using default addr %s\n", DEF_IP);
19     addr = DEF_IP;
20     // создаем сокет
21     struct sockaddr_in peer;

```

```

22     peer.sin_family = AF_INET;
23     peer.sin_port = htons(port);
24     peer.sin_addr.s_addr = inet_addr(addr);
25     int sock = socket(AF_INET, SOCK_STREAM, 0);
26     if (sock < 0)
27     {
28         perror("Can't create socket\n");
29         exit(1);
30     }
31     // присоединяемся к серверу
32     int res = connect(sock, (struct sockaddr *)&peer, sizeof(peer));
33     if (res)
34     {
35         perror("Can't connect to server:");
36         exit(1);
37     }
38     // основной цикл программы
39     char buf[100];
40
41     int first_msg = 1;
42
43     for (;;)
44     {
45         printf("Input request (empty to exit)\n");
46         if (first_msg == 0){
47             bzero(buf, 100);
48             fgets(buf, 100, stdin);
49             buf[strlen(buf) - 1] = '\0';
50         }
51         else{
52             strcpy(buf, argv[1]);
53             buf[strlen(buf)] = '\0';
54             first_msg = 0;
55         }
56         if (strlen(buf) == 0)
57         {
58             printf("Bye-bye\n");
59             return 0;
60         }
61         res = sendFix(sock, buf, 0);
62         if (res <= 0)
63         {
64             perror("Error while sending:");
65             exit(1);
66         }
67         bzero(buf, 100);
68         res = readFix(sock, buf, 100, 0);
69         if (res <= 0)
70         {
71             perror("Error while receiving:");
72             exit(1);
73         }
74
75         printf("Server's response: %s\n", buf);
76     }
77     return 0;
78 }
79 int readFix(int sock, char *buf, int bufSize, int flags)
80 {
81     // читаем "заголовок" - сколько байт составляет наше сообщение
82     unsigned msgLength = 0;
83     int res = recv(sock, &msgLength, sizeof(unsigned), flags | MSG_WAITALL);
84     if (res <= 0)
85         return res;
86     if (res > bufSize)
87     {
88         printf("Recieved more data, then we can store, exiting\n");
89         exit(1);
90     }
91     // читаем само сообщение
92     return recv(sock, buf, msgLength, flags | MSG_WAITALL);
93 }
94 int sendFix(int sock, char *buf, int flags)
95 {
96     // число байт в сообщении
97     unsigned msgLength = strlen(buf);
98     int res = send(sock, &msgLength, sizeof(unsigned), flags);
99     if (res <= 0)
100         return res;

```

```

101 |         send(sock, buf, msgLength, flags);
102 |     }

```

File: sock\_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

```

```

Using default port 8888
Using default addr 127.0.0.1
Input request (empty to exit)
Server's response: hello_from_client_1
Input request (empty to exit)
hello!
Server's response: hello!
Input request (empty to exit)
123
Server's response: 123
Input request (empty to exit)

golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter6$ ./client hello_from_client_2
Using default port 8888
Using default addr 127.0.0.1
Input request (empty to exit)
Server's response: hello_from_client_2
Input request (empty to exit)
hello 2!
Server's response: hello 2!
Input request (empty to exit)
qweqw
Server's response: qweqw
Input request (empty to exit)
yay!
Server's response: yay!
Input request (empty to exit)

```

Рисунок 16 – результат работы программы с двумя клиентами

```

Using default port 8888
Using default addr 127.0.0.1
Input request (empty to exit)
Server's response: client999
Input request (empty to exit)
hello from final client
Server's response: hello from final client
Input request (empty to exit)

```

Рисунок 17 – результат работы программы с тысячей клиентов

Как видно из рисунка 17, реализация с множеством клиентов не потребовала изменений.

Приведём пример использования программы в локальной сети. Используются виртуальные машины VirtualBox. На них настроены интерфейсы с IP-адресами 192.168.0.1/24 и 192.168.0.2/24 соответственно. Результат на рисунке 18.

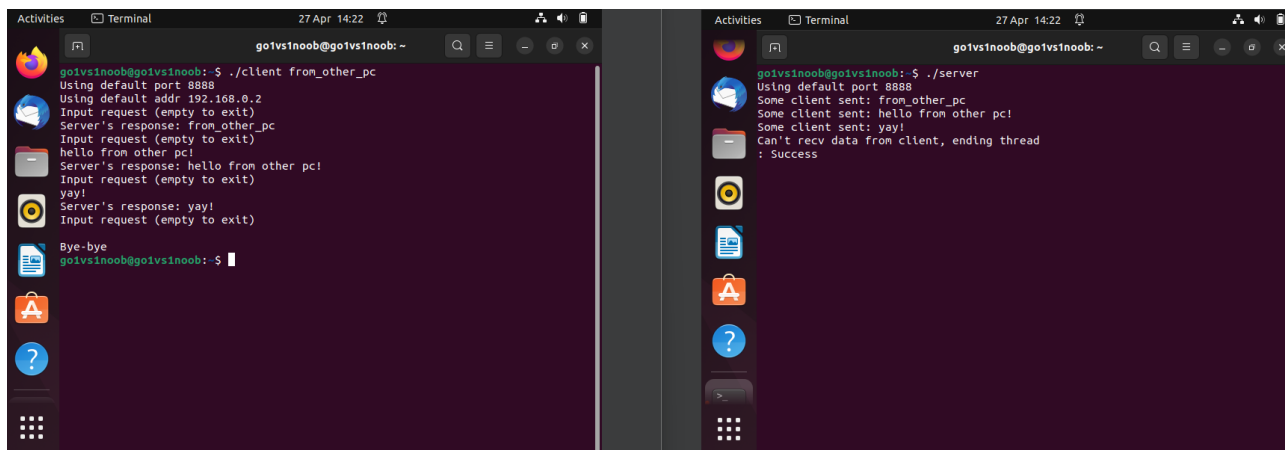


Рисунок 18 – результат работы программы на двух разных машинах в одной подсети

Как видно из рисунка 18, программа работает корректно. Ещё раз подытожим логику программы: серверный код (server.c) создает сокет, привязывает его к определённому порту и начинает прослушивать входящие соединения. Для каждого подключившегося клиента создается отдельный поток, который принимает сообщения от клиента, выводит их на экран и отправляет обратно тот же текст. Функции `sendFix` и `readFix` используются для правильной передачи сообщений, включая размер сообщения в заголовок для соблюдения правил протокола TCP. Клиентский код (client.c) также создает сокет и подключается к серверу, указав IP-адрес и порт. Затем в цикле пользователь вводит сообщения, они отправляются на сервер с помощью функции `send`, и в ответ приходит сообщение от сервера, которое выводится на экран. При вводе пустой строки программа завершается.

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

```

golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter6$ ./udp_server
Using default port 8888
Some client sent: udp1
Some client sent: udp2
Some client sent: hello1
Some client sent: wqe
Some client sent: hello2

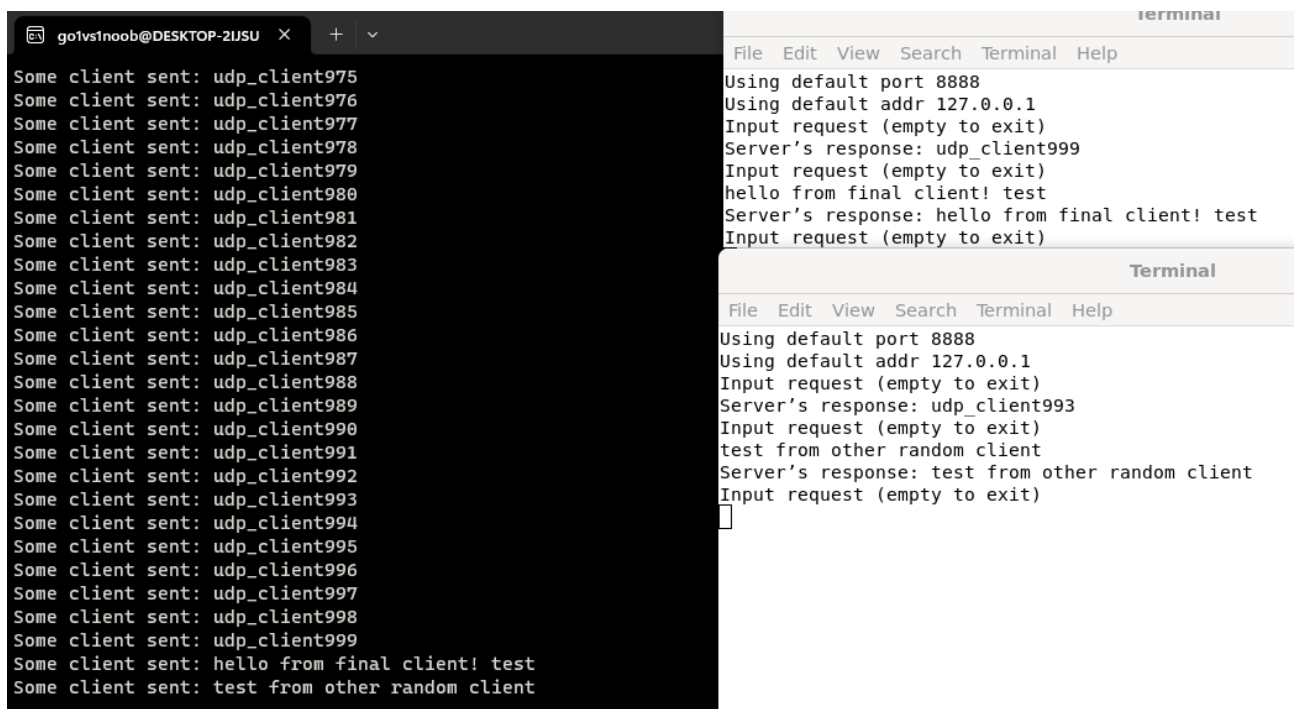
golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter6$ ./udp_client udp2
Using default port 8888
Using default addr 127.0.0.1
Input request (empty to exit)
Server's response: udp2
Input request (empty to exit)
hello1
Server's response: hello1
Input request (empty to exit)
wqe
Server's response: wqe
Input request (empty to exit)

golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter6$ ./udp_client udp1
Using default port 8888
Using default addr 127.0.0.1
Input request (empty to exit)
Server's response: udp1
Input request (empty to exit)
hello2
Server's response: hello2
Input request (empty to exit)

```

Рисунок 19 – результат работы программы с UDP соединением

Таким образом, получилось модифицировать программу, чтобы коммуникация проходила через UDP соединение. Попробуем подключить к серверу 1000 клиентов. Результат на рисунке 20.



```

golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter6$ ./udp_server
Some client sent: udp_client975
Some client sent: udp_client976
Some client sent: udp_client977
Some client sent: udp_client978
Some client sent: udp_client979
Some client sent: udp_client980
Some client sent: udp_client981
Some client sent: udp_client982
Some client sent: udp_client983
Some client sent: udp_client984
Some client sent: udp_client985
Some client sent: udp_client986
Some client sent: udp_client987
Some client sent: udp_client988
Some client sent: udp_client989
Some client sent: udp_client990
Some client sent: udp_client991
Some client sent: udp_client992
Some client sent: udp_client993
Some client sent: udp_client994
Some client sent: udp_client995
Some client sent: udp_client996
Some client sent: udp_client997
Some client sent: udp_client998
Some client sent: udp_client999
Some client sent: hello from final client! test
Some client sent: test from other random client

golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter6$ ./udp_client udp2
Using default port 8888
Using default addr 127.0.0.1
Input request (empty to exit)
Server's response: udp_client999
Input request (empty to exit)
hello from final client! test
Server's response: hello from final client! test
Input request (empty to exit)

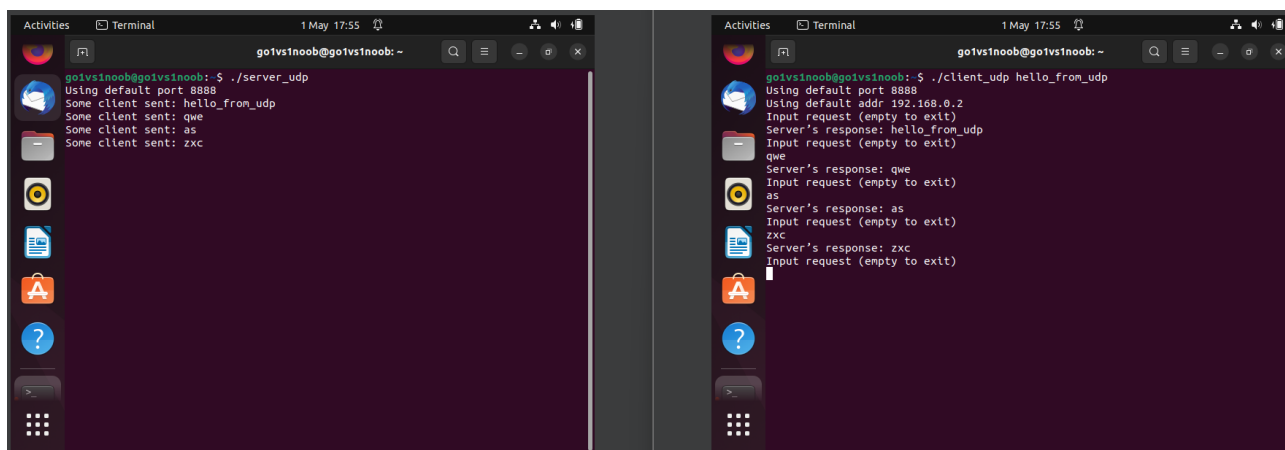
golvs1noob@DESKTOP-2IJSUSC:~/lab5/chapter6$ ./udp_client udp1
Using default port 8888
Using default addr 127.0.0.1
Input request (empty to exit)
Server's response: udp_client993
Input request (empty to exit)
test from other random client
Server's response: test from other random client
Input request (empty to exit)

```

Рисунок 20 – результат работы программы с UDP соединением с 1000 клиентов

Как видно из рисунка 20, программа работает корректно.

Повторим эксперимент с использованием сокета на машинах в локальной сети с протоколом UDP. Использованы виртуальные машины VirtualBox. На них настроены интерфейсы с IP-адресами 192.168.0.1/24 и 192.168.0.2/24 соответственно. Результат на рисунке 21.



*Рисунок 21 – результат работы программы на двух разных машинах в одной подсети*

Можно сделать вывод, что между очередями сообщений и сокетами есть различия: первое – формат передачи данных. Очереди сообщений основаны на передаче структурированных сообщений, а сокеты – на передаче байтов. Кроме того, сокеты могут использовать различные протоколы (TCP или UDP), когда как очереди сообщений используют один и тот же механизм.

### **Вывод.**

В результате выполнения работы закрепили практику работы с сигналами, в том числе посылку сигналов от неродственных процессов. Изучены такие понятия, как надежные/ненадежные сигналы, сигналы реального времени. Помимо сигналов изучены прочие методы передачи данных между процессами: каналы, очереди сообщений, сокеты. С последними проделано несколько экспериментов – в том числе обмен сообщениями с помощью сокетов между двумя машинами в одной подсети с использованием разных протоколов (TCP и UDP).

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Geeks for Geeks – <https://www.geeksforgeeks.org/>
2. Linux.org – <https://www.linux.org.ru/forum/general/>
3. Linux Documentation – [linux.die.net/man](http://linux.die.net/man)