Санкт-Петербургский государственный политехнический университет

Факультет технической кибернетики

Кафедра компьютерных систем и программных технологий

Отчёт по лабораторной работе №4

«Средства межпроцессного взаимодействия в ОС LINUX»

Работу выполнил студент группы № 4081/12

Дорофеев Юрий Владимирович

Работу принял преподаватель _____

Малышев Игорь Алексеевич

г. Санкт-Петербург

2012

План работы

1. Цель работы

Изучить следующие средства межпроцессного взаимодействия (IPC) в ОС LINUX:

1) надежные и ненадежные сигналы;
2) именованные и неименованные каналы;
3) очереди сообщений;
4) семафоры и разделяемая память.

2. Программа работы

2.1. “Ненадежные” сигналы

Составим программу *sig_father.c*, позволяющую изменить диспозицию сигналов, а именно, установить:

- обработчик пользовательских сигналов *SIGUSR1* и *SIGUSR2*;
- реакцию по умолчанию на сигнал *SIGINT*;
- игнорирование сигнала *SIGCHLD*;
- породить процесс-копию и уйти в ожидание сигналов.

Обработчик сигналов должен содержать восстановление диспозиции и оповещение на экране о (удачно или неудачно) полученном сигнале и идентификаторе родительского процесса.

Процесс-потомок, получив идентификатор родительского процесса, посылает процессу-отцу сигнал *SIGUSR1* и извещает об удачной или неудачной отправке указанного сигнала. Остальные сигналы генерируются из командной строки.

sig_father.c :

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

static void sigusr1_hndlr(int signo)
{
      printf("Get signal SIGUSR1\n");
      printf("PPID = %d\n", getppid());
      signal(SIGUSR1, SIG_DFL);
      return;
}

static void sigusr2_hndlr(int signo)
{
      printf("Get signal SIGUSR2\n");
      printf("PPID = %d\n", getppid());
      signal(SIGUSR2, SIG_DFL);
      return;
}
int main(void)
{
      int pid, ppid, status;
      int pid_son;
      pid=getpid();
```

```
            ppid=getppid();
            printf("\n FATHER PARAM: pid=%i  ppid=%i \n", pid, ppid);

            signal(SIGUSR1, sigusr1_hndlr);
            signal(SIGUSR2, sigusr2_hndlr);
            signal(SIGINT, SIG_DFL);
            signal(SIGCHLD, SIG_IGN);

            if((pid_son = fork())==0) execl("sig_son.out","sig_son.out",
NULL);

            wait(&status);
            while (1) pause();
            return 0;
      }
```

sig_son.c :

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

      int main(void)
      {
            int pid, ppid;

            pid=getpid();
            ppid=getppid();
            printf("\n SON PARAM: pid=%i  ppid=%i \n", pid, ppid);

            sleep(1);

            if ((kill(ppid, SIGUSR1)) != 0)
            {
                  printf("SIGUSR1 error\n");
                  exit(1);
            }

            return 0;
      }
```

Откомпилируем обе программы:

```
      gcc -o sig_father sig_father.c;
      gcc -o sig_son sig_son.c
```

Запустите на выполнение исполняемый модуль *./sig_father*:

```
            ./sig_father.out
            FATHER PARAM: pid=2271  ppid=1630
            SON PARAM: pid=2272  ppid=2271
            Get signal SIGUSR1
            PPID = 1630
```

Программа *sig_son* посылает сигнал *USR1*, процесс *sig_father* его принимает и обрабатывает ("Get signal SIGUSR1"). Далее с другого терминала пошлем следующие сигналы:

```
kill -s SIGCHLD 2258
kill -s SIGUSR2 2258
kill -s SIGINT 2258
```

Сигнал *SIGCHLD* игнорируется и в командной строке первого терминала ничего не выводится. При передаче сигнала *SIGUSR2*, процесс *sig_father* обрабатывает его и выводит на консоль следующее:

```
Lab4/1$ ./sig_father.out
FATHER PARAM: pid=2258  ppid=1668
SON PARAM: pid=2259  ppid=2258
Get signal SIGUSR1
Get signal SIGUSR2
PPID = 1630
```

Сигнал *SIGINT* вызывает реакцию по умолчанию, а следовательно завершает процесс *sig_father*. Появляется приглашение в shell.

Повторим отправку сигнала *SIG_USR2*:

```
User defined signal 2
```

Это вызвано тем, что в обработчике происходит восстановление диспозиции сигнала, т.е. при повторном принятии сигнала *SIG_USR2* процесс будет реагировать на него по умолчанию (аналогично для сигнала *SIG_USR1*).

2.2. “Надежные” сигналы

2.2.1. Составьте программу *sigact2.c*, позволяющую заблокировать сигнал *SIGINT*. Блокировку реализуем вызвав "засыпание" процесса на одну минуту из обработчика пользовательских сигналов:

sigact2.c :

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

struct sigaction act;
static void sigusr1_hndlr(int signo)
{
      printf("Get signal SIGUSR1\n");
      sleep(60);
}

static void sigusr2_hndlr(int signo)
{
      printf("Get signal SIGUSR2\n");
```

```
            sleep(60);
      }

      void (*mysig (int sig, void (*handler) (int))) (int) // ”надежная”
обработка сигналов
      {
            act.sa_handler = handler;     // установка обработчика сигнала sig
            sigemptyset(&act.sa_mask);    // обнуление маски
            sigaddset(&act.sa_mask, SIGINT);// блокировка сигнала SIGINT
            act.sa_flags = 0;
            if(sigaction(sig, &act, 0) < 0)
                  return (SIG_ERR);
            return (act.sa_handler);
      }

      int main(void)
      {
            int pid, ppid;
            pid=getpid();
            ppid=getppid();
            printf("\n FATHER PARAM: pid=%i  ppid=%i \n", pid, ppid);

            mysig(SIGUSR1, sigusr1_hndlr);
            mysig(SIGUSR2, sigusr2_hndlr);
            mysig(SIGINT, SIG_DFL);

            while (1) pause();
            return 0;
      }
```

С рабочего терминала отправим процессу sigact2 сигнал SIGUSR2, а затем сигнал SIGINT:

```
            kill -s SIGUSR2 2368
            kill -s SIGINT 2368
```

Получаем:

```
            FATHER PARAM: pid=2368  ppid=1668
            Get signal SIGUSR2
```

Процесс *sigact2* обрабатывает сигнал *SIGUSR2* и "засыпает" на 60 секунд. Таким образом, процесс завершается сигналом *SIGINT* только по прошествию 60 секунд.

2.2.2. Изменим обработчик так, чтобы отправка сигнала *SIGINT* производилась из обработчика сигнала *SIG_USR1*

sigact3.c :

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

      struct sigaction act, oldact;
```

```
      static void sigusr1_hndlr(int signo)
      {
            printf("Get signal SIGUSR1\n");
            sleep(2);
            if (kill(getpid(), SIGINT ))
            {
                  printf("Couldn't sent SIGINT\n");
                  exit(1);
            }
            else  printf("SIGINT send\n");
            sleep(10);
      }

      void (*mysig (int sig, void (*handler) (int))) (int) // ”надежная”
обработка сигналов
      {
            act.sa_handler = handler;     // установка обработчика сигнала sig
            sigemptyset(&act.sa_mask);    // обнуление маски
            sigaddset(&act.sa_mask, SIGINT);// блокировка сигнала SIGINT
            act.sa_flags = 0;

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

      int main(void)
      {
            int pid, ppid;
            pid=getpid();
            ppid=getppid();
            printf("\n FATHER PARAM: pid=%i  ppid=%i \n", pid, ppid);

            mysig(SIGUSR1, sigusr1_hndlr);

            while (1) pause();
            return 0;
      }
```

При приеме сигнала *SIGUSR1*, происходит посылка сигнала *SIGINT*. Но процесс на него не реагирует (не завершается):

```
            ~/Lab4/2$ ./sigact3.out
            FATHER PARAM: pid=2432  ppid=1630
            Get signal SIGUSR1
            SIGINT send
```

В этом и есть отличие надежных сигналов от ненадежных: выполнение обработки не может быть прервано сигналом SIGINT.

2.3. Неименованные каналы

Создадим неименованный канал в файле *pipe.c* посредством системного вызова *pipe(2),* который возвращает 2 файловых дескриптора *filedes[0]* для записи в канал и для чтения из канала *filedes[1]*. Доступ к дескрипторам канала может получить как процесс, вызвавший *pipe()* , так и его дочерние процессы.

pipe.c :

```
      #include <stdio.h>
      #include <unistd.h>
      #include <stdlib.h>
      #include <string.h>

      int main(int argc, char *argv[])
      {
            int filedes[2];
            int err;
            char buf[256];      //данные из файла
            char buf_pipe[256]; //данные из канала

            if (err=pipe(filedes) < 0)
            {
                  printf("Father : can't create pipe\n");
                  exit(-1);
            }
            printf("Father : pipe is created\n");

            if (!fork())
            {
                  sleep(2);
                  FILE* input = fopen("in.txt", "r");
                  if (input<=0)
                  {
                        printf("Son can't open <in.txt>\n");
                        exit(-1);
                  }
                  else
                  {
                        fscanf(input, "%s", buf);
                        printf("Son read <in.txt>: %s\n", buf);
                        write(filedes[1], buf, strlen(buf));    // запись в
pipe
                        printf("Son write in pipe\n");
                        fclose(input);
                        close(filedes[1]);
                  }
            return 0;
            }
            else
            {
                  read(filedes[0], buf_pipe, 18);      // чтение из pipe
                  close(filedes[0]);
                  printf("Father read pipe: %s\n", buf_pipe);
                  FILE* output= fopen("out.txt", "w");
                  if (output<=0)
                  {
                        printf("Father can't open <out.txt>\n");
                        exit(-1);
                  }
                  else
```

```
                {
                        fprintf(output, "%s", buf_pipe);
                        fclose(output);
                }
        }
}
```

В результате запуска процесса *pipe.out* происходят следующие действия:

- создается неименованный канал родительским процессом;
- запускается процесс-потомок, который открывает файл *in.txt*, считывает данные из него и записывает их в неименованный канал;
- чтение данных из неименованного канала и запись в файл *out.txt.*

```
~/Lab4/3$ ./pipe.out
Father : pipe is created
Son read <in.txt>: My_name_is_Yury
Son write in pipe
Father read pipe: My_name_is_Yury
```

2.4. Именованные каналы

В файле *server.c*:

В основной программе создадим 2 именованных канала, используя системный вызов *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 server(*fdr, fdw*); закроем файловые дескрипторы.

В серверной части программы запишем имя файла в канал 1 (для записи) функцией *write()*; прочитаем данные из канала 2 в зарезервированный ранее буфер.

В файле *client.c*:

В основной программе запрограммируем функции:

- открытия каналов для чтения (*chan1*) и записи (*chan2*);
- инициации клиентской части программы *client(fdr, fdw)*;
- закрытия файловых дескрипторов;
- удаления каналов системным вызовом *unlink(chan n).*

В клиентской части программы - чтение имени файла из канала 1, сообщение серверу о начале записи, открытие канала 2 и запись в него данных, прочитанных из файла.

server.c :

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

```
void server(int chan1, int chan2)
{
	char buff[17];
	//запись имени файла в канал
	write(chan1,"infile", 17);
	printf("SERVER is RUN\n");
	//чтение из канала данных в буфер
	read(chan2, buff, 17);
	printf("SERVER: %s recieved\n", buff);
	printf ("\nSERVER: Data transfer ends\n\n");
}

int main(void)
{
	int chan1, chan2;

	//создание именованных каналов
	mknod("chan1", S_IFIFO | 0666, 0);
	mknod("chan2", S_IFIFO | 0666, 0);

	//открытие каналов на запись и на чтение
	chan1=open("chan1", O_WRONLY);
	chan2=open("chan2", O_RDONLY);

	//запуск серверной части
	server(chan1,chan2);

	close(chan1);
	close(chan2);

	return 0;
}
```

client.c :

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

void client(int chan1, int chan2)
{
	char buff[18], filename[18];
	int infile, count;
	printf("CLIENT is RUN\n");
	//считывание имени файла
	read(chan1, filename, 20);
	printf("CLIENT: Transfering from a file: %s\n", filename);
	//открытие файла на чтение
	infile = open(filename, O_RDONLY);
	//чтение данных из файла в буфер
	count = read(infile, buff, 18);
	buff[count] = 0;
	printf ("CLIENT: Sending %s to server\n",buff);
	//запись в канал данных
	write(chan2, buff, 18);
	printf("CLIENT: data transfer complete.\n\n");
	close(infile);
}
```

```
int main(void)
{
	int chan1, chan2;

	//открытие каналов на запись и на чтение
	chan1=open("chan1", O_RDONLY);
	chan2=open("chan2", O_WRONLY);

	client(chan1,chan2);

	close(chan1);
	close(chan2);

	unlink("chan1");
	unlink("chan2");

	return 0;
}
```

Откомпилируем программы и запустим на выполнение исполняемые модули *./server* и *./client*:

```
~/Lab4/4$ ./server.out &
[1] 2832
~/Lab4/4$ ./client.out
CLIENT is RUN
SERVER is RUN
CLIENT: Transfering from a file: infile
CLIENT: Sending My_name_is_Yury to server
SERVER: My_name_is_Yury recieved
SERVER: Data transfer ends
CLIENT: data transfer complete.
[1]+  Done                    ./server.out
```

Сервер отправляет клиенту имя файла (*infile*) и принимает данные от него (*My_name_is_Yury*). Клиент выводит имя файла полученное от сервера, считывает данные из этого файла и отправляет их серверу. Затем клиент реализует удаление каналов.

2.5. Очереди сообщений

Серверный файл содержит:

- обработчик сигнала *SIGINT* (с восстановлением диспозиции и удалением очереди сообщений системным вызовом *msgctl()* для корректного завершения сервера при получении сигнала *SIGINT*);
- основную программу:

```
void main(void)
{
 Message msg_rcv; //принимаемое сообщение
 Message msg_snd; //посылаемое сообщение
 key_t key;  //ключ, необходимый для создания очереди
 int length, n;
 signal(SIGINT, sig_hndlr);
 //получение ключа
```

```
 if((key = ftok("/home/your_path/test.txt", 'A')) < 0)
 {
  printf("Server : can't receive a key\n");
  exit(-1);
 }
...
```

Далее создаётся очередь сообщений (msgget(key, PERM | IPC_CREAT)).
Организовывается цикл ожидания сообщения и его чтение.
Сервер в цикле читает сообщения из очереди (тип = 1) и посылает на каждое сообщение ответ клиенту (тип = 2). В случае возникновения любых ошибок функцией *kill()* инициируйте посылку сигнала *SIGINT*.
В файле *client.c* аналогично серверному коду получим ключ, затем доступ к очереди сообщений, отправим сообщение серверу (тип 1). Клиент ожидает сообщение, а затем читает его (тип 2).

server.c :

```
#include <stdio.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <stdlib.h>

int   msgid = -1;
typedef struct
{
      long  mtype;
      char  mtext[100];
}
Msg;

void sig_hndlr(int sig)
{
      signal(SIGINT, SIG_DFL);
      printf("\nSIGINT sended\n");
      msgctl(msgid, IPC_RMID,0);          //удаление очереди сообщений
}

int main()
{
      Msg   msg_rcv;                //структура принимаемого сообщения
      Msg   msg_snd;                //структура отправляемого сообщения
      key_t key;                    //ключ, необходимый для создания
очереди
      int   num;
      char buff[6];
      int sendID;

      signal(SIGINT, sig_hndlr);
      printf("SERVER: Start\n");

      //подключение ключа
      if ((key = ftok("test.txt", 'A')) < 0)
      {
            printf("SERVER: can't receive a key\n");
            exit(1);
      }
```

```
            //создание очереди сообщений (возвращает идентификатор сообщения
ассоциирующийся со значением ключа)
            msgid = msgget(key, 0666 | IPC_CREAT);
            if ( msgid < 0 )
            {
                  printf("SERVER: can't create message queue\n");
                  exit(1);
            }

            msg_rcv.mtype = 1;      //тип сообщения на чтение
            msg_snd.mtype = 2;      //тип сообщения на запись

            while (1)
            {
                  //прием сообщения
                  num=msgrcv(msgid, &msg_rcv, sizeof(msg_rcv), msg_rcv.mtype,
0);
                  if (num>0)
                  {
                        printf("SERVER: Message received: %s\n",
msg_rcv.mtext);
                        //записываем текст сообщения
                        sprintf(msg_snd.mtext, "Hello, client!");
                        printf("SERVER: Sending message %s\n", msg_snd.mtext);
                        //отправка сообщения
                        if (msgsnd(msgid, &msg_snd, sizeof(msg_snd), 0))
                        {
                              printf("SERVER: error sending message.\n");
                              kill(getpid(), SIGINT);
                        }
                  }
                  else if (num==-1)
                  {
                        printf("SERVER: error receiving message.\n");
                        kill(getpid(), SIGINT);
                  }
            }
      }
```

client.c :

```
      #include <stdio.h>
      #include <sys/types.h>
      #include <sys/ipc.h>
      #include <sys/msg.h>
      #include <stdlib.h>

      typedef struct
      {
            long  mtype;
            char  mtext[100];
      }
      Msg;

      int main(void)
      {
            Msg   msg_rcv;                //структура принимаемого сообщения
            Msg   msg_snd;                //структура отправляемого сообщения
            key_t key;                    //ключ, необходимый для создания
очереди
            int   num=0;
```

```
            int   msgid;

            printf("CLIENT: Start\n");
            //подключение ключа
            if ((key = ftok("test.txt", 'A')) < 0)
            {
                  printf("CLIENT: can't receive a key.\n");
                  exit(1);
            }
            //создание очереди сообщений (возвращает идетификатор сообщения
ассоциирующийся со значением ключа)
            msgid = msgget(key, 0);
            if ( msgid < 0 )
            {
                  printf("CLIENT: can't create queue.\n");
                  exit(1);
            }

            msg_rcv.mtype = 2;        //тип сообщения на чтение
            msg_snd.mtype = 1;        //тип сообщения на запись

            //записываем текст сообщения
            sprintf(msg_snd.mtext, "Hey, server!");
            //отправка сообщения
            if (msgsnd(msgid, &msg_snd, sizeof(msg_snd), 0))
            {
                  printf("CLIENT: error sending message.\n");
                  exit(1);
            }
            printf("CLIENT: Sending message: %s\n", msg_snd.mtext);
            while (num==0)
            {
                  //получение сообщения
                  num=msgrcv(msgid, &msg_rcv, sizeof(msg_rcv), msg_rcv.mtype,
0);
                  if (num<0)
                  {
                        printf("CLIENT: error receiving message.\n");
                        exit(1);
                  }
            }

            printf("CLIENT: Received message: %s\n", msg_rcv.mtext);
            printf("CLIENT: Stop\n");

            return 0;
      }
```

Запустим на выполнение обе программы с разных терминалов:

```
            ~/Lab4/5$ ./server.out
            SERVER: Start
            SERVER: Message received: Hey, server!
            SERVER: Sending message Hello, client!
            ^C
            SIGINT sended
            SERVER: error receiving message.
```

```
            ~/Lab4/5$ ./client.out
            CLIENT: Start
            CLIENT: Sending message: Hey, server!
```

```
CLIENT: Received message: Hello, client!
CLIENT: Stop
```

Наблюдаем обмен сообщениями между клиентом и сервером. После приема сообщения от сервера клиент завершает свою работу, а сервер продолжает ждать сообщения. При поступлении сигнала *SIG_INT*, сервер уведомляет нас о нем и завершает свою работу.

Запустим одновременно несколько клиентов:

```
~/Lab4/5$ ./server.out
SERVER: Start
SERVER: Message received: Hey, server! I am client_0
SERVER: Sending message Hello, client!
SERVER: Message received: Hey, server! I am client_1
SERVER: Sending message Hello, client!
```

```
~/Lab4/5$ ./client.out | ./client1.out
CLIENT_0: Start
CLIENT_0: Sending message: Hey, server! I am client_0
CLIENT_0: Received message: Hello, client!
CLIENT_0: Stop
CLIENT_1: Start
CLIENT_1: Sending message: Hey, server! I am client_1
CLIENT_1: Received message: Hello, client!
CLIENT_1: Stop
```

Видно, что сервер отвечает на сообщения в порядке очереди: сначала получает сообщение от *client_0* и отвечает на него, затем читает сообщение *client_1* и так же отвечает на него. Реализуется очередь сообщений.

2.6. Семафоры и разделяемая память

Последовательность действий:

- откомпилировать программы (gcc -o server server.c; gcc -o client client.c)
- запустить на выполнение исполняемый модуль ./server
- запустить на выполнение исполняемый модуль ./client
- проверить их работоспособность, откорректировать, если это необходимо
- послать процессу server сигнал SIGINT

Одновременно могут работать несколько клиентов. Сервер ожидает начала работы какого-либо клиента, после чего ждет освобождения разделяемой памяти, блокирует ее и читает сообщение. Затем сервер пишет сообщение в разделяемую память и освобождает ресурс. Сервер корректно завершает свою работу при получении сигнала *SIGINT* (он удаляет созданные ранее семафоры и область разделяемой памяти, затем завершается сам).

shmem.h :

```
#define MAXBUFF 80 //максимальная длина сообщения в разделяемой памяти
#define PERM 0666  //права доступа к разделяемой памяти
```

```
typedef struct mem //структура данных в разделяемой памяти
{
 int segment;
 char buff[MAXBUFF];
}Message;

//ожидание начала выполнения клиента
static struct sembuf proc_wait[1] = {1, -1, 0};
//уведомление сервера о том, что клиент начал работу
static struct sembuf proc_start[1] = {1, 1, 0};
//блокирование разделяемой памяти
static struct sembuf mem_lock[2] = {0, 0, 0, 0, 1, 0};
//освобождение ресурса
static struct sembuf mem_unlock[1] = {0, -1, 0};
```

server.c :

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <stdlib.h>
#include "shmem.h"

Message* msgptr;
int shmid, semid; //идентификаторы сегмента и набора семафоров

void hndlr(int sig) //обработчик сигнала SIGINT
{
	signal(SIGINT, hndlr);
	//отключение от области разделяемой памяти
	if(shmdt(msgptr) < 0)
	{
		printf("Server : error\n");
		exit(-1);
	}
	//удаление созданных объектов
	if(shmctl(shmid, IPC_RMID, 0) < 0)
	{
		printf("Server : can't delete area\n");
		exit(-1);
	}
	printf("Server : area is deleted\n");
	if(semctl(semid, 0, IPC_RMID) < 0)
	{
		printf("Server : can't delete semaphore\n");
		exit(-1);
	}
	printf("Server : semaphores are deleted\n");
}

void main(void)
{
	key_t key;
	signal(SIGINT, hndlr);
	//получение ключа как для семафора так и для разделяемой памяти
	if((key = ftok("test.txt", 'A')) < 0)
	{
```

```
                printf("Server : can't get a key\n");
                exit(-1);
        }
        //создание области разделяемой памяти
        if((shmid = shmget(key, sizeof(Message), PERM | IPC_CREAT)) < 0)
        {
                printf("Server : can't create an area\n");
                exit(-1);
        }
        printf("Server : area is created\n");
        //присоединение области
        if((msgptr = (Message*)shmat(shmid, 0, 0)) < 0)
        {
                printf("Server : error of joining\n");
                exit(-1);
        }
        printf("Server : area is joined\n");
        //создание группы из 2 семафоров
        //1 - для синхронизации работы с разделяемой памятью
        //2 - для синхронизации выполнения процессов
        if((semid = semget(key, 2, PERM | IPC_CREAT)) < 0)
        {
                printf("Server : can't create a semaphore\n");
                exit(-1);
        }
        printf("Server : semaphores are created\n");

        while(1)
        {
                //ожидание начала работы клиента
                if(semop(semid, &proc_wait[0], 1) < 0)
                {
                        printf("Server : execution complete\n");
                        exit(-1);
                }
                //ожидание завершения работы клиента с разделяемой памятью
                if(semop(semid, &mem_lock[0], 2) < 0)
                {
                        printf("Server : can't execute a operation\n");
                        exit(-1);
                }
                //вывод сообщения, записанного клиентом в разделяемую память
                printf("Server : read message\n%s", msgptr->buff);
                //запись сообщения в разделяемую память
                sprintf(msgptr->buff,"Message from server with PID = %d\n",
getpid());
                //освобождение ресурса
                if(semop(semid, &mem_unlock[0], 1) < 0)
                {
                        printf("Server : can't execute a operation\n");
                        exit(-1);
                }
        }
}
```

client.c

```
#include <stdio.h>
#include <sys/types.h>
```

```
	#include <sys/ipc.h>
	#include <sys/sem.h>
	#include <sys/shm.h>
	#include <stdlib.h>
	#include "shmem.h"

	void main(void)
	{
		Message* msgptr;
		key_t key;
		int shmid, semid;
		//получение ключа
		if((key = ftok("test.txt", 'A')) < 0)
		{
			printf("Client : can't get a key\n");
			exit(-1);
		}
		//получение доступа к разделяемой памяти
		if((shmid = shmget(key, sizeof(Message), 0)) < 0)
		{
			printf("Client : access denied\n");
			exit(-1);
		}
		//присоединение разделяемой памяти
		if((msgptr = (Message*)shmat(shmid, 0, 0)) < 0)
		{
			printf("Client : error of joining\n");
			exit(-1);
		}
		//получение доступа к семафору
		if((semid = semget(key, 2, PERM)) < 0)
		{
			printf("Client : access denied\n");
			exit(-1);
		}
		//блокировка разделяемой памяти
		if(semop(semid, &mem_lock[0], 2) < 0)
		{
			printf("Client : can't execute a operation\n");
			exit(-1);
		}
		//уведомление сервера о начале работы
		if(semop(semid, &proc_start[0], 1) < 0)
		{
			printf("Client : can't execute a operation\n");
			exit(-1);
		}
		//запись сообщения в разделяемую память
		sprintf(msgptr->buff,"Message from client with PID = %d\n",
getpid());
		//освобождение ресурса
		if(semop(semid, &mem_unlock[0], 1) < 0)
		{
			printf("Client : can't execute a operation\n");
			exit(-1);
		}
		//ожидание завершения работы сервера с разделяемой памятью
		if(semop(semid, &mem_lock[0], 2) < 0)
		{
			printf("Client : can't execute a operation\n");
			exit(-1);
		}
		//чтение сообщения из разделяемой памяти
```

```
            printf("Client : read message\n%s", msgptr->buff);
            //освобождение разделяемой памяти
            if(semop(semid, &mem_unlock[0], 1) < 0)
            {
                  printf("Client : can't execute a operation\n");
                  exit(-1);
            }
            //отключение от области разделяемой памяти
            if(shmdt(msgptr) < 0)
            {
                  printf("Client : error\n");
                  exit(-1);
            }
      }
```

Запусти сервер и 2 клиента:

```
            ~/Lab4/6_sem$ ./server
            Server : area is created
            Server : area is joined
            Server : semaphores are created
            Server : read message
            Message from client with PID = 2155
            Server : read message
            Message from client_1 with PID = 2156
```

```
            ~/Lab4/6_sem$ ./client
            Client : read message
            Message from server with PID = 2154
            ~/Lab4/6_sem$ ./client1
            Client_1 : read message
            Message from server with PID = 2154
```

Произведем посылку сигнала *SIGINT* серверу:

```
            ~/Lab4/6_sem$ kill -s SIGINT 2154
            Server : area is deleted
            Server : semaphores are deleted
            Server : execution complete
```

Семафоры обеспечивают удобную синхронизацию доступа к разделяемым ресурсам. Разделяемая память обеспечивает совместный доступ к данным. Одновременно могут работать несколько клиентов. Алгоритм работы представлен на рисунке 1.

Сервер ожидает начала работы какого-либо клиента, затем ожидает завершения его работы (освобождения разделяемой памяти), блокирует разделяемую память и читает сообщение, записанное клиентом. Затем сервер записывает сообщение в разделяемую память и освобождает ресурс. Клиент блокирует разделяемую память, читает сообщение от сервера, освобождает ресурс и отключается от области разделяемой памяти. Сервер корректно завершает свою работу при получении сигнала *SIGINT* (он удаляет созданные ранее семафоры и область разделяемой памяти, затем завершается сам).

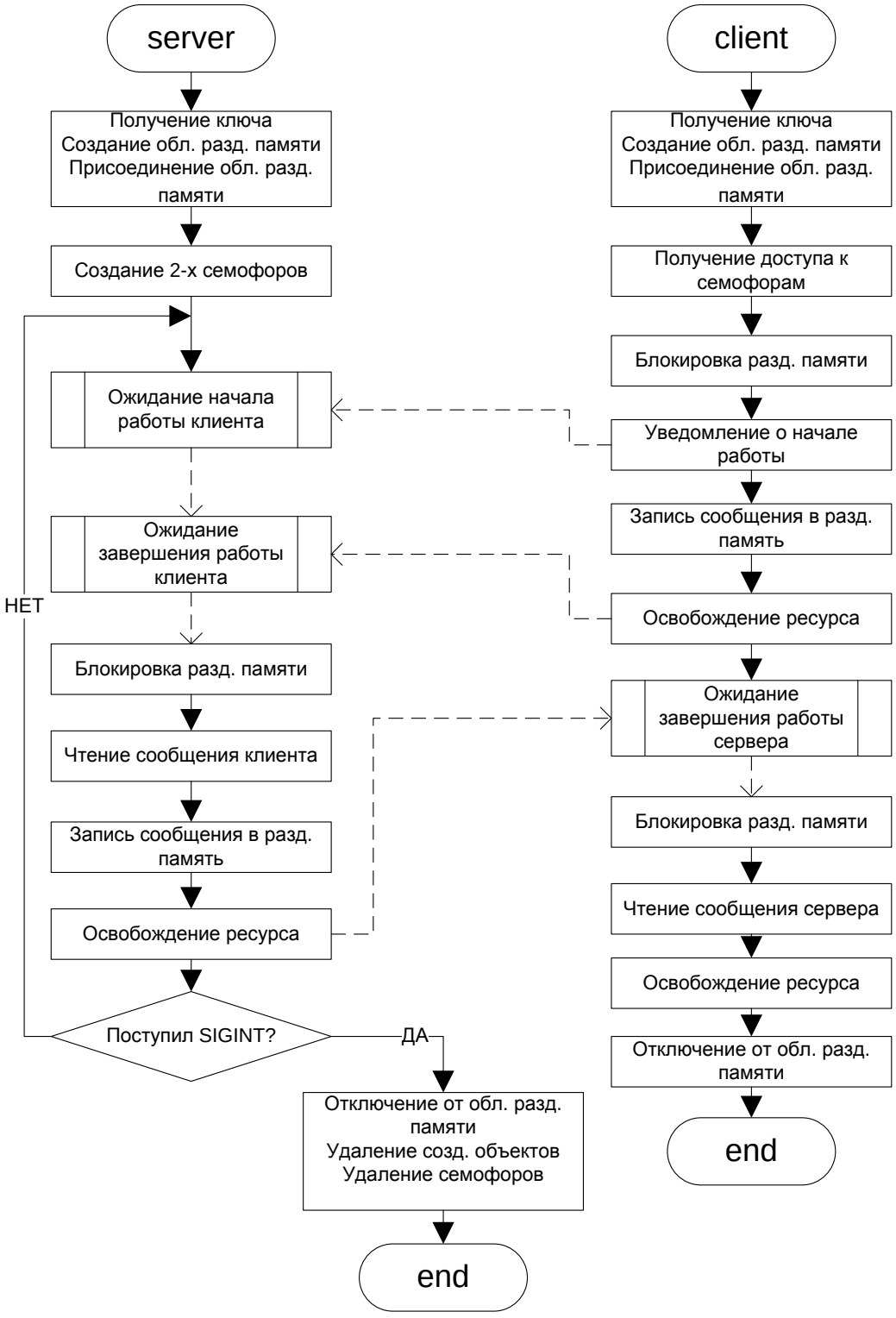

Рис. 1. Алгоритм работы программы

3. Выводы

Сигналы позволяют передавать уведомления между процессами (через ядро) или между ядром и процессом. Сигналы делятся на надежные и ненадежные. Надежные сигналы сохраняют обработчик сигнала, после его обработки, в отличии от ненадежных, которые после обработки сигнала, устанавливают реакцию на сигнал по умолчанию.

Именованные каналы - один из методов межпроцессного взаимодействия. Традиционный канал — «безымянен», потому что существует анонимно и только во время выполнения процесса. Именованный канал — существует в системе и после завершения процесса. Он должен быть «отсоединён» или удалён когда уже не используется.

Очередь сообщений является разделяемым системным ресурсом. Каждая очередь имеет свой ключ и права доступа, определяющие какие операции способны выполнять процессы, использующие очередь сообщений. Для доступа к очереди сообщений существуют операции чтения и записи в очередь. Размер сообщения и его тип могут быть разными для разных процессов. У каждого процесса должен быть свой тип принимаемого сообщения, для того что бы можно было точно определить для кого это сообщение отправлено.

Разделяемая память представляет собой наиболее быстрый способ межпроцессорного взаимодействия. Основная трудность в организации доступа к разделяемой памяти – это синхронизация процессов, работающих с ней. Средствами синхронизации доступа служат семафоры. Семафоры размещаются в адресном пространстве ядра и являются счетчиками, управляющими доступом к общим ресурсам. Чаще всего они используются как блокирующий механизм, не позволяющий одному процессу захватить занятый ресурс. Над семафорами определены три типа операций:

- увеличение значения семафора на N;
- блокировка процесса, вызывающего операцию, до тех пор, пока значение семафора не станет равно нулю;
- блокировка процесса до тех пор, пока значение семафора не станет больше заданного в операции значения N.