**«Санкт-Петербургский Государственный Политехнический Университет»**

Кафедра компьютерных систем и программных технологий

**Отчёт по лабораторной работе №4**

Тема: «Средства межпроцессного взаимодействия (IPC) в ОС LINUX»

Дисциплина: «Операционные системы»

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Выполнил студент гр.3530901/70203 | \_\_\_\_\_\_\_\_\_\_\_ | | | Иванов И.Д. |
| Преподав­атель, к.т.н., доц. | \_\_\_\_\_\_\_\_\_\_\_ | | | Душутина Е. В. |
|  |  | “\_\_\_”\_\_\_\_\_\_\_\_\_\_\_\_ 2019 г. | | |
|  |  | |  | |

**1. Цель работы**

Изучение основных принципов межпроцессорных взаимодействий в ОС LINUX.

**2. Программа работы**

1.1. Составьте программу, 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 полученном (удачно или неудачно) сигнале и идентификаторе родительского процесса. Процесс-потомок, получив идентификатор родительского процесса, должен отправить процессу-отцу сигнал SIGUSR1 и извещение об удачной или неудачной отправке указанного сигнала. Остальные сигналы можно сгенерировать из командной строки

1.2 Повторите эксперимент для других сигналов и процессов, порождаемых в разных файлах; а также для потоков одного и разных процессов.

2.1. Создать программу, позволяющую продемонстрировать возможность отложенной обработки (временного блокирования) сигнала (например, SIGINT).

2.2. Изменить обработчик сигнала так, чтобы из него производилась отправка другого сигнала.

3.1. Проведите эксперимент, позволяющий определить возможность организации

очереди для различных типов сигналов, обычных и реального времени, (более двух

сигналов, для этого увеличьте «вложенность» вызовов обработчиков).

3.2. Экспериментально подтвердите, что обработка равноприоритетных сигналов реального времени происходит в порядке FIFO.

3.3. Опытным путем подтвердите наличие приоритетов сигналов реального времени.

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

5.1. Создать клиент-серверное приложение, демонстрирующее дуплексную (двунаправленную) передачу информации двумя однонаправленными именованными каналами между клиентом и сервером.

5.2. Изменить программу из пункта 5.1. так, чтобы сервер, перед тем как читать данные из канала, ожидал пользовательского ввода.

6.1. Создать клиент-серверное приложение, демонстрирующее передачу информации между процессами посредством очередей сообщений.

7. Рассмотреть несколько вариантов постановки задачи синхронизации доступа к разделяемой памяти

8.1. Рассмотреть пример использования сокетов – простейший эхо-сервер.

8.2. Модифицировать, если необходимо, предложенное приложение и реализовать обмен сервера с множеством клиентов. Количество клиентов: 10, 100, 1000. Выполнить аналогичное взаимодействие на основе UDP.

**Система, на которой производится работа:**

asdf@BigAwesomeTurtle:/home/asdf/lab\_2$uname -a

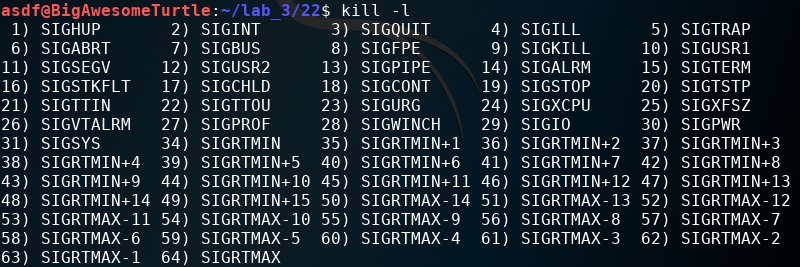
Linux BigAwesomeTurtle 5.2.0-kali2-686-pae #1 SMP Debian 5.2.9-2kali1 (2019-08-22) i686 GNU/Linux

Дистрибутив – Kali Linux

**3. Выполнение работы**

**1.1. Составьте программу, позволяющую изменить диспозицию сигналов.**

Вывод списка сигналов:



**Текст программы:**

#include <stdio.h>  
#include <signal.h>  
#include <unistd.h>  
#include <stdlib.h>  
  
static void sigHandler(int sig) {  
printf("Catched signal %s\n",sig == SIGUSR1 ? "SIGUSR1" : "SIGUSR2"); printf("Parent = %d\n",getppid());  
//востанавливаем старую диспозицию  
signal(sig,SIG\_DFL);  
}  
  
int main() {  
printf("\nFather started: pid = %i,ppid = %i\n",getpid(),getppid());  
signal(SIGUSR1,sigHandler);  
signal(SIGUSR2,sigHandler);  
signal(SIGINT,SIG\_DFL);  
signal(SIGCHLD,SIG\_IGN);  
int forkRes = fork();  
if(forkRes == 0) {  
//программа сын  
printf("\nSon started: pid = %i,ppid = %i\n",getpid(),getppid());  
//шлем сигналы отцу  
if(kill(getppid(),SIGUSR1) != 0) {  
printf("Error while sending SIGUSR1\n");  
exit(1);  
}  
printf("Successfully sent SIGUSR1\n"); return 0;  
}  
//программа отец  
wait(NULL);  
//ждем сигналов  
for(;;){  
pause();  
}  
return 0;  
}

**Результат выполнения программы:**

asdf@BigAwesomeTurtle:~/lab\_4/1\_1$ ./1\_1.out

Father started: pid = 2696,ppid = 1340

Son started: pid = 2697,ppid = 2696

Successfully sent SIGUSR1

Catched signal SIGUSR1

Parent = 1340

Процесс-потомок отправил сигнал SIGUSR1, а процесс-отец его успешно принял. Отправим еще 3 сигнала процессу-отцу: SIGCHLD, SIGUSR2, SIGINT:

asdf@BigAwesomeTurtle:~/lab\_4/1\_1$ kill -SIGUSR2 2696  
asdf@BigAwesomeTurtle:~/lab\_4/1\_1$ kill -SIGCHLD 2696  
asdf@BigAwesomeTurtle:~/lab\_4/1\_1$ kill -SIGINT 2696

Результат:

Catched signal SIGUSR2  
Parent = 1340  
  
asdf@BigAwesomeTurtle:~/lab\_4/1\_1$

Сигнал SIGUSR2 также был «пойман», на сигнал SIGCHLD не последовало никакой реакции (так как он был проигнорирован), и сигнал SIGINT привел к завершению работы.

Запустим программу еще раз и дважды отправим ей сигнал SIGUSR2:

asdf@BigAwesomeTurtle:~/lab\_4/1\_1$ ./1\_1.out  
  
Father started: pid = 2713,ppid = 1340  
  
Son started: pid = 2714,ppid = 2713  
Successfully sent SIGUSR1  
Catched signal SIGUSR1  
Parent = 1340  
Catched signal SIGUSR2  
Parent = 1340  
User defined signal 2

В результате первый сигнал был «пойман», второй обработался по умолчанию. Это происходит потому, что в обработчике прерываний после первого приема сигнала происходит восстановление диспозиции сигналов. Аналогичная ситуация была бы при двукратной отправке процессу сигнала SIGUSR1

**1.2. Повторите эксперимент для других сигналов и процессов, порождаемых в разных файлах; а также для потоков одного и разных процессов.**

**Содержимое father.c:**

#include <stdio.h>  
#include <stdlib.h>  
#include <signal.h>  
int main(){  
int pid, ppid, status, a, b, c;  
pid = getpid();  
ppid = getppid();  
printf("\n\nFATHER PARAM: pid=%i ppid=%i\n", pid, ppid);  
if ((a = fork()) == 0)  
execl("son1.out", "son1.out", NULL);  
if ((b = fork()) == 0)  
execl("son2.out", "son2.out", NULL);  
if ((c = fork()) == 0)  
execl("son3.out", "son3.out", NULL);  
sleep(6);  
kill(a, SIGINT);  
kill(b, SIGCHLD);  
kill(c, SIGUSR1);  
system("ps xf > file.txt");  
//wait(&status);  
return 0;  
}

**Содержимое son1.c:**

#include <stdio.h>  
#include <signal.h>  
#include <stdlib.h>  
int main(){  
int pid,ppid;  
signal(SIGINT, SIG\_DFL);  
pid=getpid();  
ppid=getppid();  
printf("\n\nSON1 PARAMS: pid=%i ppid=%i\n\n",pid,ppid);  
sleep(10);  
printf("We will not see this, because of SIGINT interruption");  
return 0; //статус завершения 0  
}

**Содержимое son2.c:**

#include <stdio.h>  
#include <stdlib.h>  
#include <signal.h>  
int main(){  
int pid,ppid;  
signal(SIGCHLD, SIG\_IGN);  
pid=getpid();  
ppid=getppid();  
printf("\n\nSON2 PARAMS: pid=%i ppid=%i\n\n",pid,ppid);  
sleep(10);  
return 0; // статус завершения 0  
}

**Содержимое son3.c:**

#include <stdio.h>  
#include <stdlib.h>  
#include <signal.h>  
int catch (int a) {  
printf("Son3 catch SIGUSR1\n");  
return 0;  
}  
  
int main(){  
int pid,ppid;signal(SIGUSR1, catch);  
pid=getpid();  
ppid=getppid();  
printf("\n\nSON3 PARAMS: pid=%i ppid=%i\n\n",pid,ppid);  
sleep(10);  
return 0; // статус завершения 0  
}

**Результат выполнения программы:**

asdf@BigAwesomeTurtle:~/lab\_4/1\_2$ ./father.out  
  
FATHER PARAM: pid=3092 ppid=2706  
  
SON2 PARAMS: pid=3094 ppid=3092  
  
SON1 PARAMS: pid=3093 ppid=3092  
  
SON3 PARAMS: pid=3095 ppid=3092

Son3 catch SIGUSR1

В данном случае процесс father порождает три процесса-потомка, находящихся в других файлах. Первому потомку посылается сигнал SIGINT, который завершает его работу, так как обрабатывается по-умолчанию (строка “We will not see this, because of SIGINT unterruption” не выведется). Второму потомку посылается сигнал SIGCHLD, который остаётся проигнорированным. Третьему потомку посылается сигнал SIFUSR1, который он успешно ловит и обрабатывает.

**Эксперимент для потоков:**

**Текст программы:**

#include <signal.h>  
#include <sys/types.h>  
#include "pthread.h"  
#include <stdio.h>  
pthread\_t t1;  
void handler(int sig){  
printf("Catched signal %s\n",sig == SIGUSR1 ? "SIGUSR1": "SIGUSR2");  
printf("Parent = %d\n",getppid());  
signal(SIGINT, SIG\_DFL);  
}  
  
void\* thread1(void\* ptr){

signal(SIGUSR1,handler);  
signal(SIGUSR2,handler);  
int count =0;  
while (1){  
printf("Thread\_1 итерация %i\n", count);  
count++;  
sleep(1);  
}  
}  
  
int main(){  
pthread\_create(&t1, NULL, thread1, NULL);  
pthread\_join(t1, NULL);  
pthread\_kill(t1,SIGUSR2);  
while(1){  
printf("Father thread \n");  
sleep(1);  
}  
return 0;  
}

**Результат выполнения программы:**

Запустим программу и отправим 3 сигнала процессу-отцу: SIGUSR1, SIGCHLD, SIGINT:

asdf@BigAwesomeTurtle:~/lab\_4/1\_2\_2$ kill -SIGUSR1 3246  
asdf@BigAwesomeTurtle:~/lab\_4/1\_2\_2$ kill -SIGCHLD 3246  
asdf@BigAwesomeTurtle:~/lab\_4/1\_2\_2$ kill -SIGINT 3246

Результат:

asdf@BigAwesomeTurtle:~/lab\_4/1\_2\_2$ ./1\_2.out  
Thread\_1 итерация 0  
Thread\_1 итерация 1  
Thread\_1 итерация 2  
Thread\_1 итерация 3  
Thread\_1 итерация 4  
Thread\_1 итерация 5  
Thread\_1 итерация 6  
Thread\_1 итерация 7  
Catched signal SIGUSR1  
Parent = 2706  
Thread\_1 итерация 8  
Thread\_1 итерация 9  
Thread\_1 итерация 10  
Thread\_1 итерация 11  
Thread\_1 итерация 12  
Thread\_1 итерация 13  
Thread\_1 итерация 14  
Thread\_1 итерация 15  
Thread\_1 итерация 16  
Thread\_1 итерация 17  
Thread\_1 итерация 18  
Thread\_1 итерация 19  
Thread\_1 итерация 20

Сигнал SIGUSR1 был успешно пойман и обработан, SIGCHLD проигнорирован, а SIGINT завершил работу процесса, так как был обработан по-умолчанию.

**2.1. Создать программу, позволяющую продемонстрировать возможность отложенной обработки (временного блокирования) сигнала (например, SIGINT).**

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

**Текст программы:**

#include <stdio.h>

#include <signal.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <unistd.h>

#include <stdlib.h>

#include <fcntl.h>

void (int sig,void (\*hnd(int)))(int) {

//надежная обработка сигналов

struct sigaction act,oldact;

act.sa\_handler = hnd;

sigemptyset(&act.sa\_mask);

sigaddset(&act.sa\_mask,SIGINT);

act.sa\_flags = 0;

if(sigaction(sig,&act,0) < 0)

return SIG\_ERR;

return act.sa\_handler;

}

void hndUSR1(int sig) {

if(sig != SIGUSR1) {

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

return;

}

printf("SIGUSR1 catched\n");

sleep(60);

}

int main() {

mysig(SIGUSR1,hndUSR1);

for(;;) {

pause();

}

return 0;

}

**Результат выполнения программы:**

asdf@BigAwesomeTurtle:~/lab\_4/2\_1$ ./2\_1.out &  
[3] 3362  
asdf@BigAwesomeTurtle:~/lab\_4/2\_1$ kill -SIGUSR1 3362  
SIGUSR1 catched  
asdf@BigAwesomeTurtle:~/lab\_4/2\_1$ kill -SIGINT 3362  
asdf@BigAwesomeTurtle:~/lab\_4/2\_1$ jobs  
[1]+ Running ./2\_1.out &  
asdf@BigAwesomeTurtle:~/lab\_4/2\_1$ jobs  
[1]+ Interrupt ./2\_1.out

Чтобы иметь возможность отправить сигналы с терминала, запустим программу в фоновом режиме. По результатам сигнал SIGUSR1 принят корректно, но после посылки сигнала SIGINT программа продолжала выполняться еще минуту, и только после этого завершилась. В этом отличие надежной обработки сигналов от ненадежной: есть возможность отложить прием некоторых других сигналов. Отложенные таким образом сигналы записываются в маску PENDING и обрабатываются после завершения обработки сигналов, которые отложили обработку. Механизм ненадёжных сигналов не позволяет откладывать обработку других сигналов (можно лишь установить игнорирование некоторых сигналов на время обработки).

**2.2.** **Изменить 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.

**Текст программы:**

#include <stdio.h>

#include <signal.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <unistd.h>

#include <stdlib.h>

#include <fcntl.h>

void (int sig,void (\*hnd(int)))(int) {

// надежная обработка сигналов

struct sigaction act,oldact;

act.sa\_handler = hnd;

sigemptyset(&act.sa\_mask);

sigaddset(&act.sa\_mask,SIGINT);

act.sa\_flags = 0;

if(sigaction(sig,&act,0) < 0)

return SIG\_ERR;

return act.sa\_handler;

}

void hndUSR1(int sig) {

if(sig != SIGUSR1) {

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

return;

}

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

kill(getpid(),SIGINT);

sleep(10);

}

int main() {

mysig(SIGUSR1,hndUSR1);

for(;;) {

pause();

}

return 0;

}

**Результат выполнения программы:**

asdf@BigAwesomeTurtle:~/lab\_4/2\_2$ ./2\_2.out &

[3] 3435

asdf@BigAwesomeTurtle:~/lab\_4/2\_2$ kill -SIGUSR1 3435

SIGUSR1 catched, sending SIGINT

asdf@BigAwesomeTurtle:~/lab\_4/2\_2$ jobs

[1]+ Running ./2\_2.out &

asdf@BigAwesomeTurtle:~/lab\_4/2\_2$ jobs

[1]+ Interrupt ./2\_2.out

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

**3.1-3.3 Проведите эксперимент, позволяющий определить возможность организации очереди для различных типов сигналов, обычных и реального времени, (более двух сигналов, для этого увеличьте «вложенность» вызовов обработчиков);** **Экспериментально подтвердите, что обработка равноприоритетных сигналов реального времени происходит в порядке FIFO;** **Опытным путем подтвердите наличие приоритетов сигналов реального времени.**

**Текст программы:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/signal.h>

int rec\_sig[20]; // массив сигналов

unsigned int nfree\_elem = 0; // номер элемента

void sighandler(int signo) { // обработчик

rec\_sig[nfree\_elem++] = signo; // сохраняем номер принятого сигнала

return;

}

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

sigset\_t mask;

struct sigaction act;

int i;

memset(&act, 0, sizeof(act));

sigemptyset(&mask); // иниц. набора сигналов пустми значениями

//добавляем в маску сигналы

sigaddset(&mask, SIGRTMIN);

sigaddset(&mask, SIGRTMIN + 1);

sigaddset(&mask, SIGRTMIN + 2);

sigaddset(&mask, SIGUSR1);

sigaddset(&mask, SIGUSR1);

sigaddset(&mask, SIGUSR2);

sigaddset(&mask, SIGCHLD);

//устанавливаем обработчик для сигнала

act.sa\_handler = sighandler;

act.sa\_mask = mask;

sigaction(SIGRTMIN, &act, NULL);

sigaction(SIGRTMIN + 1, &act, NULL);

sigaction(SIGRTMIN + 2, &act, NULL);

sigaction(SIGUSR1, &act, NULL);

sigaction(SIGUSR2, &act, NULL);

sigaction(SIGCHLD, &act, NULL);

//блокируем обработчик сигналов

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

//посылаем сигналы самому себе в случайном порядке

raise(SIGUSR2);

raise(SIGUSR2);

raise(SIGRTMIN + 2);

raise(SIGRTMIN + 1);

raise(SIGUSR1);

raise(SIGCHLD);

raise(SIGUSR1);

raise(SIGRTMIN);

raise(SIGRTMIN + 1);

raise(SIGRTMIN);

raise(SIGUSR1);

raise(SIGCHLD);

raise(SIGUSR1);

raise(SIGRTMIN + 1);

raise(SIGRTMIN + 2);

raise(SIGCHLD);

raise(SIGRTMIN);

raise(SIGRTMIN);

raise(SIGCHLD);

raise(SIGUSR1);

//разблокируем сигналы, что вызовет их немедленную отправку

sigprocmask(SIG\_UNBLOCK, &mask, NULL);

//выводим сигналы в порядке их получения

for(i=0;i<nfree\_elem;++i){

if (rec\_sig[i] == SIGUSR1) {

printf("We get SIGUSR1\n");

} else if (rec\_sig[i] == SIGCHLD) {

printf("We get SIGCHLD\n");

} else if (rec\_sig[i] == SIGRTMIN) {

printf("We get SIGRTMIN\n");

} else if (rec\_sig[i] == SIGRTMIN + 1) {

printf("We get SIGRTMIN + 1\n");

} else if (rec\_sig[i] == SIGRTMIN + 2) {

printf("We get SIGRTMIN + 2\n");

} else if (rec\_sig[i] == SIGUSR2) {

printf("We get SIGUSR2\n");

}

}

return 0;

}

**Результат выполнения программы:**

asdf@BigAwesomeTurtle:~/lab\_4/3\_1$ ./3\_1.out

We get SIGUSR1

We get SIGUSR2

We get SIGCHLD

We get SIGRTMIN

We get SIGRTMIN

We get SIGRTMIN

We get SIGRTMIN

We get SIGRTMIN + 1

We get SIGRTMIN + 1

We get SIGRTMIN + 1

We get SIGRTMIN + 2

We get SIGRTMIN + 2

В данном случае мы устанавливаем обработчики для сигналов SIGRTMIN, SIGRTMIN+1, SIGRTMIN+2, SIGUSR1, SIGUSR2 и SIGCHLD, после чего блокируем обработчик сигналов (необходимо для их одновременного прихода), посылаем себе множество сигналов, разблокируем обработчик и смотрим, в каком порядке и количестве обрабатываются сигналы.

Было послано 5 сигналов SIGUSR1, 2 сигнала SIGUSR2, 4 сигнала SIGCHLD, 4 сигнала SIGRTMIN, 3 сигнала SIGRTMIN + 1 и 2 сигнала SIGRTMIN + 2.

В результате видно, что были обработаны все посланные сигналы реального времени и по-одному посланному обычному сигналу каждого типа. Сигналы реального времени обрабатываются в соответствии с приоритетами (сигнал с меньшим номером имеет больший приоритет), а обработка равноприоритетных сигналов реального времени происходит в порядке FIFO.

**4.1.** **Организовать программу так, чтобы процесс-родитель создавал неименованный канал, создавал потомка, закрывал канал на запись и записывал в произвольный текстовый файл считываемую из канала информацию.**

**Текст программы:**

#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
  
#define DEF\_F\_R "from.txt"  
#define DEF\_F\_W "to.txt"  
  
int main(int argc, char\*\* argv) {  
char fileToRead[32];  
char fileToWrite[32];  
if(argc < 3 ){  
printf("Using default fileNames '%s','%s'\n",DEF\_F\_R,DEF\_F\_W);  
strcpy(fileToRead,DEF\_F\_R);  
strcpy(fileToWrite,DEF\_F\_W);  
}  
else {  
strcpy(fileToRead,argv[1]);  
strcpy(fileToWrite,argv[2]);  
}  
int filedes[2];  
if(pipe(filedes) < 0) {  
printf("Father: can't create pipe\n");  
exit(1);  
}  
printf("pipe is successfully created\n");  
if(fork() == 0) {  
// процесс сын  
// закрывает пайп для чтения  
close(filedes[0]);  
FILE\* f = fopen(fileToRead,"r");  
if (!f) {  
printf("Son: cant open file %s\n",fileToRead);  
exit(1);  
}  
char buf[100];  
int res;  
while(!feof(f)) {  
// читаем данные из файла  
res = fread(buf,sizeof(char),100,f);  
write(filedes[1],buf,res); // пишем их в пайп  
}  
close(f);  
close(filedes[1]);  
return 0;  
}  
// процесс отец  
// закрывает пайп для записи  
close(filedes[1]);  
FILE \*f = fopen(fileToWrite,"w");  
if (!f) {  
printf("Father: cant open file %s\n",fileToWrite);  
exit(1);  
}  
char buf[100];  
int res;  
while(1) {  
bzero(buf,100);  
res = read(filedes[0],buf,100);  
if(!res)  
break;  
printf("Read from pipe: %s\n",buf);  
fwrite(buf,sizeof(char),res,f);  
}  
fclose(f);  
close(filedes[0]);  
return 0;  
}

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

**Результат выполнеия программы:**

asdf@BigAwesomeTurtle:~/lab\_4/4\_1$ ./pipe.out  
Using default fileNames 'from.txt','to.txt'  
pipe is successfully created  
Read from pipe: First  
second  
Third?  
The end.

asdf@BigAwesomeTurtle:~/lab\_4/4\_1$

**Содержимое файла from.txt:**

First  
second  
Third?  
The end.

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

**Содержимое файла to.txt после запуска программы:**

First  
second  
Third?  
The end.

**5.1. Создать клиент-серверное приложение, демонстрирующее дуплексную (двунаправленную) передачу информации двумя однонаправленными именованными каналами между клиентом и сервером.**

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, права 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 канал читается имя файла, во второй канал пишется его

содержимое.

**Содержимой файла server.c:**

#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
#include <string.h>  
#include <sys/types.h>  
#include <sys/stat.h>  
#include <fcntl.h>  
  
#define DEF\_FILENAME "testFile.txt"  
  
int main(int argc, char\*\* argv) {  
char fileName[30];  
if(argc < 2) {  
printf("Using default file name '%s'\n",DEF\_FILENAME);  
strcpy(fileName,DEF\_FILENAME);  
}  
else  
strcpy(fileName,argv[1]);  
// создаем два канала  
int res = mknod("channel1",S\_IFIFO | 0666,0);  
if(res) {  
printf("Can't create first channel\n");  
exit(1);  
}  
res = mknod("channel2",S\_IFIFO | 0666,0);  
if(res) {  
printf("Can't create second channel\n");  
exit(1);  
}  
// открываем первый канал для записи  
int chan1 = open("channel1",O\_WRONLY);  
if(chan1 == -1) {  
printf("Can't open channel for writing\n");  
exit(0);  
}  
// открываем второй канал для чтения  
int chan2 = open("channel2",O\_RDONLY);  
if(chan2 == -1) {  
printf("Can't open channe2 for reading\n");  
exit(0);  
}  
// пишем имя файла в первый канал  
write(chan1,fileName,strlen(fileName));  
// читаем содержимое файла из второго канала  
char buf [100];  
for(;;) {  
bzero(buf,100);  
res = read(chan2,buf,100);  
if(res <= 0)  
break;  
printf("Part of file: %s\n");  
}  
close(chan1);  
close(chan2);  
unlink("channel1");  
unlink("channel2");  
return 0;  
}

**Содержимой файла client.c:**

#include<stdio.h>  
#include<stdlib.h>  
#include<unistd.h>  
#include<string.h>  
#include<sys/types.h>  
#include<sys/stat.h>  
#include<fcntl.h>  
  
int main() {  
// каналы сервер уже создал, открываем их  
int chan1 = open("channel1", O\_RDONLY);  
if(chan1 == -1) {  
printf("Can't open channel1 for reading\n");  
exit(0);  
}  
int chan2 = open("channel2", O\_WRONLY);  
if(chan2 == -1) {  
printf("Can't open channel2 for reading\n");  
exit(0);  
}  
// читаем имя файла из первого канала  
char fileName[100];  
bzero(fileName,100);  
int res = read(chan1,fileName,100);  
if(res <= 0) {  
printf("Can't read fileName from channel1\n");  
exit(0);  
}  
// открываем файл на чтение  
FILE \*f = fopen(fileName,"r");  
if(!f) {  
printf("Can't open file %s\n",fileName);  
exit(0);  
}  
// читаем из файла и пишем во второй канал  
char buf[100];  
while(!feof(f)) {  
// читаем данные из файла  
res = fread(buf,sizeof(char),100,f);  
// пишем их в канал  
write(chan2,buf,res);  
}  
fclose(f);  
close(chan1);  
close(chan2);  
return 0;  
}

Запустим программу server из одного терминала и client из другого:

**Результат работы сервера:**

asdf@BigAwesomeTurtle:~/lab\_4/5\_1$ ./server.out  
Using default file name 'testFile.txt'  
Part of file: First  
second  
Third?  
The end.  
asdf@BigAwesomeTurtle:~/lab\_4/5\_1$

**Результат работы клиента:**

asdf@BigAwesomeTurtle:~/lab\_4/5\_1$ ./client.out

**Содержимое файла testFile.txt:**

First  
second  
Third?  
The end.

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

**5.2. Изменить программу из пункта 5.1. так, чтобы сервер, перед тем как читать данные из канала, ожидал пользовательского ввода.**

Текст программы клиента оставим неизменным.

**Текст программы сервера:**

#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
#include <string.h>  
#include <sys/types.h>  
#include <sys/stat.h>  
#include <fcntl.h>  
  
#define DEF\_FILENAME "testFile.txt"  
  
int main(int argc, char \*\*argv){  
char fileName[30];  
if (argc < 2){  
printf("Using default file name '%s'\n", DEF\_FILENAME);  
strcpy(fileName, DEF\_FILENAME);  
}  
else  
strcpy(fileName, argv[1]);  
// создаем два канала  
int res = mknod("channel1", S\_IFIFO | 0666, 0);  
if (res){  
printf("Can't create first channel\n");  
exit(1);  
}  
res = mknod("channel2", S\_IFIFO | 0666, 0);  
if (res){  
printf("Can't create second channel\n");  
exit(1);  
}  
// открываем первый канал для записи  
int chan1 = open("channel1", O\_WRONLY);  
if (chan1 == -1){  
printf("Can't open channel for writing\n");  
exit(0);  
}  
// открываем второй канал для чтения  
int chan2 = open("channel2", O\_RDONLY);  
if (chan2 == -1){  
printf("Can't open channe2 for reading\n");  
exit(0);  
}  
// пишем имя файла в первый канал  
write(chan1, fileName, strlen(fileName));  
// читаем содержимое файла из второго канала  
char buf[100];  
printf("Waiting for clint write to channnel\n");  
getchar();  
for (;;){  
bzero(buf, 100);  
res = read(chan2, buf, 100);  
if (res <= 0)  
break;  
printf("Part of file: %s\n");  
}  
close(chan1);  
close(chan2);  
unlink("channel1");  
unlink("channel2");  
printf("Server finished\n");  
return 0;  
}

Запустим клиент и сервер. Пока сервер ожидает ввода, посмотрим размер файла канала.

Сервер:

asdf@BigAwesomeTurtle:~/lab\_4/5\_2$ ./server.out  
Using default file name 'testFile.txt'  
Waiting for clint write to channnel

Клиент:

asdf@BigAwesomeTurtle:~/lab\_4/5\_2$ ./client.out

Client finished  
asdf@BigAwesomeTurtle:~/lab\_4/5\_2$

Размер файла каналов:

asdf@BigAwesomeTurtle:~/lab\_4/5\_2$ ls -sl | grep chan  
0 prw-r--r— 1 asdf asdf 0 Dec 8 21:52 channel1  
0 prw-r--r— 1 asdf asdf 0 Dec 8 21:52 channel2

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. Позволим серверу выполняться дальше, нажав на enter, и убедимся, что данные получены:

Waiting for clint write to channnel  
  
Part of file: First  
second  
Third?  
The end.  
Server finished  
asdf@BigAwesomeTurtle:~/lab\_4/5\_2$

​

**6.1. Создать клиент-серверное приложение, демонстрирующее передачу**

**информации между процессами посредством очередей сообщений.**

Аналогично предыдущему разделу программа включает 2 файла: серверный и

клиентский. В общем случае одновременно могут работать несколько клиентов.

Серверный файл содержит:

- подключение библиотек (см. листинг ниже)

- обработчик сигнала SIGINT (с восстановлением диспозиции и удалением очереди

сообщений системным вызовом msgctl() для корректного завершения сервера при

получении сигнала SIGINT);

- основную программу.

В основной программе создается очередь сообщений, используя системный вызов msgget(key, PERM | IPC\_CREAT), организовывается цикл ожидания сообщения и его чтение.

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().

**Содержимое файла server.c:**

#include <stdio.h>  
#include <stdlib.h>  
#include <sys/ipc.h>  
#include <sys/msg.h>  
#include <sys/types.h>  
#include <signal.h>  
#include <string.h>  
  
#define DEF\_KEY\_FILE "key"  
  
typedef struct{  
long type;  
char buf[100];  
} Message;  
  
int queue;  
  
void intHandler(int sig){  
signal(sig, SIG\_DFL);  
if (msgctl(queue, IPC\_RMID, 0) < 0){  
printf("Can't delete queue\n");  
exit(1);  
}  
}  
  
int main(int argc, char \*\*argv){  
char keyFile[100];  
bzero(keyFile, 100);  
if (argc < 2){  
printf("Using default key file %s\n", DEF\_KEY\_FILE);  
strcpy(keyFile, DEF\_KEY\_FILE);  
}  
else  
strcpy(keyFile, argv[1]);  
key\_t key;  
key = ftok(keyFile, 'Q');  
if (key == -1){  
printf("Can't get key for the key file %s and id 'Q'\n", keyFile);  
exit(1);  
}  
queue = msgget(key, IPC\_CREAT | 0666);  
if (queue < 0){  
printf("Can't create queue\n");  
exit(4);  
}  
// до этого момента вызывали exit(), а не kill, т.к. очередь  
// еще не была создана  
signal(SIGINT, intHandler);  
// основной цикл работы сервера  
Message mes;  
int res;  
for (;;){  
bzero(mes.buf, 100);  
// получаем первое сообщение с типом 1  
res = msgrcv(queue, &mes, sizeof(Message), 1L, 0);  
if (res < 0){  
printf("Error while recving msg\n");  
kill(getpid(), SIGINT);  
}  
printf("Client's request: %s\n", mes.buf);  
// шлем клиенту сообщение с типом 2, что все ок  
mes.type = 2L;  
bzero(mes.buf, 100);  
strcpy(mes.buf, "OK");  
res = msgsnd(queue, (void \*)&mes, sizeof(Message), 0);  
if (res != 0){  
printf("error while sending msg\n");  
kill(getpid(), SIGINT);  
}  
}  
return 0;  
}

**Содержимое файла client.c:**

#include <stdio.h>  
#include <stdlib.h>  
#include <sys/ipc.h>  
#include <sys/msg.h>  
#include <sys/types.h>  
#include <signal.h>  
#include <string.h>  
  
#define DEF\_KEY\_FILE "key"  
  
typedef struct{  
long type;  
char buf[100];  
} Message;  
  
int queue;  
  
int main(int argc, char \*\*argv){  
char keyFile[100];  
bzero(keyFile, 100);  
if (argc < 2){  
printf("Using default key file %s\n", DEF\_KEY\_FILE);  
strcpy(keyFile, DEF\_KEY\_FILE);  
}  
else  
strcpy(keyFile, argv[1]);  
key\_t key;  
key = ftok(keyFile, 'Q');  
if (key == -1){  
printf("Can't get key for key file %s and id 'Q'\n", keyFile);  
exit(1);  
}  
queue = msgget(key, 0);  
if (queue < 0){  
printf("Can't create queue\n");  
exit(4);  
}  
// основной цикл работы программы  
Message mes;  
int res;  
for (;;){  
bzero(mes.buf, 100);  
// читаем сообщение с консоли  
fgets(mes.buf, 100, stdin);  
mes.buf[strlen(mes.buf) - 1] = '\0';  
// шлем его серверу  
mes.type = 1L;  
res = msgsnd(queue, (void \*)&mes, sizeof(Message), 0);  
if (res != 0){  
printf("Error while sending msg\n");  
exit(1);  
}  
// получаем ответ, что все хорошо  
res = msgrcv(queue, &mes, sizeof(Message), 2L, 0);  
if (res < 0){  
printf("Error while recving msg\n");  
exit(1);  
}  
printf("Server's response: %s\n", mes.buf);  
}  
return 0;  
}

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 отсылает обратное

сообщение с типом 2, содержащее фразу «ОК».

Описание работы клиента: Клиент получает ту же очередь, что и сервер и ждет

ввода пользователя. Считав ввод, он шлет сообщение с типом 1, содержащее считанные

данные и ожидает от сервера подтверждения о принятии.

Запустим два процесса client и серверный процесс c разных терминалов.

**Результат выполнения программы:**

Сервер:

asdf@BigAwesomeTurtle:~/lab\_4/6\_1$ ./server.out  
Using default key file key  
Client's request: first client  
Client's request: I'm second client

Первый клиент:

asdf@BigAwesomeTurtle:~/lab\_4/6\_1$ ./client.out  
Using default key file key  
first client  
Server's response: OK

Второй клиент:

asdf@BigAwesomeTurtle:~/lab\_4/6\_1$ ./client.out  
Using default key file key  
I'm second client  
Server's response: OK

**Количественные ограничения средств IPC**

Максимальные и минимальные значения констант можно выяснить различными способами, в частности, просматривая соответствующие файлы каталога /proc/sys/kernel. Наиболее простой способ – воспользоваться утилитой ipcs с ключом -l.

Пример выполнения команды ipcs:

asdf@BigAwesomeTurtle:~/lab\_4$ 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) = 4177919  
max total shared memory (kbytes) = 17112760316  
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

**7. Рассмотреть несколько вариантов постановки задачи синхронизации доступа к**

**разделяемой памяти**

**7.1. Вариант 1.** Есть один процесс, выполняющий запись в разделяемую память и один процесс, выполняющий чтение из нее. Под чтением понимается извлечение данных из памяти. Программа должна обеспечить невозможность повторного чтения одних и тех же данных и невозможность перезаписи данных, т.е. новой записи, до тех пор, пока читатель не прочитает предыдущую.

В таком варианте задания для синхронизации процессов достаточно двух семафоров.

Покажем, почему не достаточно одного на примере.

Так как мы используем один семафор, то алгоритм работы читателя и писателя может быть только таким – захват семафора, выполнение действия (чтение / запись), освобождение семафора. Теперь допустим, что читатель прочитал данные, освободил семафор и еще не до конца использовал квант процессорного времени. Тогда он перейдет на новую итерацию, снова захватит только что освобожденный семафор и снова прочитает данные – ошибка.

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

Тогда алгоритм работы читателя и писателя будет выглядеть так:

****

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.

При таком алгоритме работы, оба процесса после выполнения своей задачи и освобождения одного из семафоров, будут ждать освобождения другого семафора, которое произведет другой процесс, но только после выполнения своей работы. Таким образом повторное чтение, или повторная запись стала невозможной.

**Текст программы читателя:**

#include <stdio.h>

#include <stdlib.h>

#include <signal.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

#include <sys/shm.h>

#include <sys/time.h>

#include <string.h>

#include "shm.h"

Message\* p\_msg;

int shmemory;

int semaphore;

void intHandler(int sig) {

//отключаем разделяемую память

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

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

exit(1);

}

//удаляем shm и семафоры

if(shmctl(shmemory, IPC\_RMID, 0) < 0) {

printf("Error while deleting shm\n");

exit(1);

}

if(semctl(semaphore, 0, IPC\_RMID) < 0) {

printf("Error while deleting semaphore\n");

exit(1);

}

}

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

char keyFile[100];

bzero(keyFile,100);

if(argc < 2) {

printf("Using default key file %s\n",DEF\_KEY\_FILE);

strcpy(keyFile,DEF\_KEY\_FILE);

}

else

strcpy(keyFile,argv[1]);

key\_t key;

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

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

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

exit(1);

}

//создаем shm

if((shmemory = shmget(key, sizeof(Message), IPC\_CREAT | 0666)) < 0) {

printf("Can't create shm\n");

exit(1);

}

//присоединяем shm в наше адресное пространство

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

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

exit(1);

}

// устанавливаем обработчик сигнала

signal(SIGINT, intHandler);

//создаем группу из 2 семафоров

//1 - показывает, что можно читать

//2 - показывает, что можно писать

if((semaphore = semget(key, 2, IPC\_CREAT | 0666)) < 0) {

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

kill(getpid(),SIGINT);

}

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

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

printf("execution complete\n");

kill(getpid(),SIGINT);

}

// основной цикл работы

for(;;) {

// ждем пока клиент начнет работу

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

printf("execution complete\n");

kill(getpid(),SIGINT);

}

//читаем сообщение от клиента

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

// говорим клиенту, что можно снова писать

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

printf("execution complete\n");

kill(getpid(),SIGINT);

}

}

}

**Текст программы писателя:**

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

#include <sys/shm.h>

#include <string.h>

#include "shm.h"

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

Message\* p\_msg;

char keyFile[100];

bzero(keyFile,100);

if(argc < 2) {

printf("Using default key file %s\n",DEF\_KEY\_FILE);

strcpy(keyFile,DEF\_KEY\_FILE);

}

else

strcpy(keyFile,argv[1]);

key\_t key;

int shmemory;

int semaphore;

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

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

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

exit(1);

}

//создаем shm

if((shmemory = shmget(key, sizeof(Message), 0666)) < 0) {

printf("Can't create shm\n");

exit(1);

}

//присоединяем shm в наше адресное пространство

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

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

exit(1);

}

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

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

exit(1);

}

char buf[100];

for(;;) {

bzero(buf,100);

printf("Type message to serever. Empty string to finish\n");

fgets(buf,100,stdin);

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

printf("bye-bye\n");

exit(0);

}

//хотим отправить сообщение

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

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

exit(1);

}

//запись сообщения в разделяемую память

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

//говорим серверу, что он может читать

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

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

exit(11);

}

}

//отключение от области разделяемой памяти

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

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

exit(1);

}

}

**Содержимое файла shm.h:**

#define DEF\_KEY\_FILE "key"

typedef struct {

long type;

char buf[100];

} Message;

static struct sembuf readEna[1] = {0,-1,0};

static struct sembuf writeEna[1] = {1,-1,0};

static struct sembuf setReadEna[1] = {0,1,0};

static struct sembuf setWriteEna[1] = {1,1,0};

**Результат выполнения программы:**

Поток-писатель 1:

asdf@BigAwesomeTurtle:~/lab\_4/7\_1$ ./writer.out

Using default key file key

Type message to serever. Empty string to finish

Hello

Type message to serever. Empty string to finish

I'm

Type message to serever. Empty string to finish

reader 1

Type message to serever. Empty string to finish

bye-bye

asdf@BigAwesomeTurtle:~/lab\_4/7\_1$

Поток-писатель 2:

asdf@BigAwesomeTurtle:~/lab\_4/7\_1$ ./writer.out

Using default key file key

Type message to serever. Empty string to finish

Hi

Type message to serever. Empty string to finish

Reader 2 is here

Type message to serever. Empty string to finish

bye-bye

asdf@BigAwesomeTurtle:~/lab\_4/7\_1$

Поток-читатель:

asdf@BigAwesomeTurtle:~/lab\_4/7\_1$ ./reader.out

Using default key file key

Client's message: Hello

Client's message: I'm

Client's message: reader 1

Client's message: Hi

Client's message: Reader 2 is here

^Cexecution complete

Все сообщения от клиента сервером прочитаны.

**7.2. Вариант 2.** К условиям предыдущей задачи добавляется условие корректной работы нескольких читателей и нескольких писателей одновременно. Как и в предыдущем варианте под чтением понимается извлечение данных из памяти, т. е. одну порцию данных может прочитать только один читатель.

Легко понять, что это условие не приводит к необходимости использования дополнительных средств синхронизации. Теперь вместо одного процесса, за каждый семафор будут конкурировать несколько. Но повторная запись или чтение все также невозможно. Так как, чтобы очередной процесс-писатель отработал, нужно освобождение семафора, которое выполняется из процесса-читателя, и наоборот.

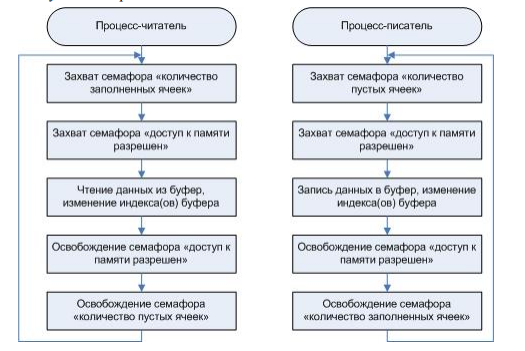
**7.3. Вариант 3.** К условиям предыдущей задачи добавляется наличие не единичного буфера, а буфера некоторого размера. Тип буфера (очередь, стек, кольцевой буфер) не имеет значения.

Двух семафоров по прежнему достаточно, но это приведет к вырождению буфера, так как все процессы будут работать только с одной ячейкой.

Так как размер буфера не равен единице, то больше нет необходимости в чередовании операций чтения и записи, допустима ситуация нескольких записей подряд, и после этого нескольких чтений. Нужно только следить, чтобы не было записи в уже заполненный буфер и не было чтения из пустого буфера. Для этого выберем другие типы семафора и придадим им другую семантику. Возьмем два считающих семафора. Максимальное значение обоих – размер буфера. Первый инициализируется нулем и имеет смысл «количество заполненных ячеек», второй инициализируется N, где N – размер буфера и имеет смысл «количество пустых ячеек». Процессы-читатели перед своей работой захватывают семафор «количество заполненных ячеек», т.е. ждут появления хотя бы одной порции данных, а после чтения освобождают семафор «количество пустых ячеек». Процессы- писатели перед записью захватывают семафор «количество пустых ячеек», т.е. ждут появления хотя бы одной пустой ячейки для записи, а после записи освобождают семафор «количество полных ячеек». Таким образом, решается проблема чтения из пустого буфера и запись в полный.

Так как семафоры не бинарные, захватить их может сразу несколько процессов, т.е. несколько процессов попадут в секцию записи или чтения. В этом случае, если операция записи или чтения не атомарная (а зачастую так оно и есть), может произойти нарушение нормальной работы программы, к примеру, несколько процессов-писателей попытаются произвести запись в одну и ту же ячейку буфера, или несколько читателей выполнят чтение одной и той же ячейки. Таким образом, операции записи-чтения становятся критическими секциями, доступ к которым также необходимо синхронизировать. Для этого будет достаточно еще одного бинарного семафора, имеющего смысл «доступ к памяти разрешен». Оба типа процессов должны захватывать его при попытке взаимодействия с памятью и освобождать после.

В итоге, алгоритм работы процессов-писателей, и процессов-читателей выглядит следующим образом:

****

В качестве разделяемого ресурса используется массив, находящийся в разделяемой памяти. Ячейка памяти, находящаяся за последним элементом массива, интерпретируется как индекс последнего записанного элемента.

**Текст программы читателя:**

#include <stdio.h>

#include <stdlib.h>

#include <signal.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

#include <sys/shm.h>

#include <sys/time.h>

#include "shm.h"

#include <strings.h>

#include <string.h>

int\* buf;

int shmemory;

int semaphore;

void intHandler(int sig) {

shmdt(buf);

shmctl(shmemory, IPC\_RMID, 0);

semctl(semaphore, 0, IPC\_RMID);

}

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

char keyFile[100];

bzero(keyFile,100);

if(argc < 2) {

printf("Using default key file %s\n", DEF\_KEY\_FILE);

strcpy(keyFile, DEF\_KEY\_FILE);

}

else

strcpy(keyFile,argv[1]);

key\_t key;

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

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

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

exit(1);

}

//создаем shm

if(( shmemory = shmget (key,(BUF\_SIZE+1)int,IPC\_CREAT|0666))< 0){

printf("Can't create shm\n");

exit(1);

}

//присоединяем shm в наше адресное пространство

if((buf = (int\*)shmat(shmemory, 0, 0)) < 0) {

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

exit(1);

}

// устанавливаем обработчик сигнала

signal(SIGINT, intHandler);

//создаем группу из 3 семафоров

//1 - число свободных ячеек

//2 - число занятых ячеек

// 3 работа с памятью

if((semaphore = semget(key, 3, IPC\_CREAT | 0666)) < 0) {

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

kill(getpid(),SIGINT);

}

// устанавливаем индекс в -1,

//первый записывающий клиент установит его в ноль

buf[BUF\_SIZE] = -1;

// инициализируем массив -1

int j = 0;

for(j = 0; j < BUF\_SIZE; ++j) {

buf[j] = -1;

}

//устнавливаем 1 семафор в число свободных ячеек,т.е. можно писать

if(semop(semaphore, setFree, 1) < 0) {

printf("execution complete\n");

kill(getpid(),SIGINT);

}

// говорим, что память свободна

if(semop(semaphore, mem\_unlock, 1) < 0) {

printf("execution complete\n");

kill(getpid(),SIGINT);

}

printf("Press enter to start working\n");

getchar();

// основной цикл работы

int i = 0;

for(i = 0; i < 15; ++i) {

// ждем, пока будет хоть одна непустая ячейка

if(semop(semaphore, waitNotEmpty, 1) < 0) {

printf("execution complete\n");

kill(getpid(),SIGINT);

}

// ждем возможности поработать с памятью

if(semop(semaphore, mem\_lock, 1) < 0) {

printf("execution complete\n");

kill(getpid(),SIGINT);

}

//читаем сообщение от клиента

int res = buf[buf[BUF\_SIZE]];

buf[BUF\_SIZE] = buf[BUF\_SIZE] - 1;

printf("Remove %d from cell %d\n", res,buf[BUF\_SIZE]+1);

// освобождаем память

if(semop(semaphore, mem\_unlock, 1) < 0) {

printf("execution complete\n");

kill(getpid(),SIGINT);

}

// увеличиваем число пустых ячеек

if(semop(semaphore, releaseEmpty, 1) < 0) {

printf("execution complete\n");

kill(getpid(),SIGINT);

}

}

}

**Текст программы писателя:**

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

#include <sys/shm.h>

#include "shm.h"

#include <string.h>

int\* buf;

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

char keyFile[100];

bzero(keyFile,100);

if(argc < 2) {

strcpy(keyFile,DEF\_KEY\_FILE);

}

else

strcpy(keyFile,argv[1]);

key\_t key;

int shmemory;

int semaphore;

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

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

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

exit(1);

}

//создаем shm

if((shmemory = shmget(key, (BUF\_SIZE+1)int, 0666)) < 0){

printf("Can't create shm\n");

exit(1);

}

//присоединяем shm в наше адресное пространство

if((buf = (int\*)shmat(shmemory, 0, 0)) < 0) {

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

exit(1);

}

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

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

exit(1);

}

printf("Press enter to start working\n");

getchar();

int send = 0;

char tb[10];

int i = 0;

for(i = 0; i < 10;++i) {

//ждем, пока будет хоть одна свободная ячейка

if(semop(semaphore, waitNotFull, 1) < 0) {

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

exit(1);

}

// ждем доступа к разделяемой памяти

if(semop(semaphore, mem\_lock, 1) < 0) {

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

exit(1);

}

printf("Add %d to cell %d\n",send,buf[BUF\_SIZE]+1);

++buf[BUF\_SIZE];

buf[buf[BUF\_SIZE]] = send++;

//освобождаем доступ к памяти

if(semop(semaphore, mem\_unlock, 1) < 0) {

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

exit(11);

}

//увеличиваем число занятых ячеек

if(semop(semaphore, releaseFull, 1) < 0) {

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

exit(11);

}

}

//отключение от области разделяемой памяти

shmdt(buf);

}

**Текст программы shm.h:**

#define DEF\_KEY\_FILE "key"

#define BUF\_SIZE 100

typedef struct {

long type;

char buf[100];

} Message;

static struct sembuf setFree[1] = {0,BUF\_SIZE,0};

static struct sembuf waitNotEmpty[1] = {1,-1,0};

static struct sembuf releaseEmpty[1] = {0,1,0};

static struct sembuf releaseFull[1] = {1,1,0};

static struct sembuf waitNotFull[1] = {0,-1,0};

static struct sembuf mem\_lock[1] = {2,-1,0};

static struct sembuf mem\_unlock[1] = {2,1,0};

**Результат выполнения программы:**

Процесс-писатель 1:

asdf@BigAwesomeTurtle:~/lab\_4/7\_2$ ./writer.out

Press enter to start working

Add 0 to cell 0

Add 1 to cell 1

Add 2 to cell 2

Add 3 to cell 3

Add 4 to cell 4

Add 5 to cell 0

Add 6 to cell 1

Add 7 to cell 2

Add 8 to cell 3

Add 9 to cell 4

Процесс-писатель 2:

asdf@BigAwesomeTurtle:~/lab\_4/7\_2$ ./writer.out

Press enter to start working

Add 0 to cell 0

Add 1 to cell 1

Add 2 to cell 2

Add 3 to cell 3

Add 4 to cell 4

Add 5 to cell 0

Add 6 to cell 1

Add 7 to cell 2

Add 8 to cell 3

Add 9 to cell 4

Процесс-читатель:

asdf@BigAwesomeTurtle:~/lab\_4/7\_2$ ./reader.out

Using default key file key

Press enter to start working

Remove 4 from cell 4

Remove 3 from cell 3

Remove 2 from cell 2

Remove 1 from cell 1

Remove 0 from cell 0

Remove 9 from cell 4

Remove 8 from cell 3

Remove 7 from cell 2

Remove 6 from cell 1

Remove 5 from cell 0

Remove 4 from cell 4

Remove 3 from cell 3

Remove 2 from cell 2

Remove 1 from cell 1

Remove 0 from cell 0

Процессы-писатели записывают по 10 чисел в массив, процесс-читатель считывает первые 15 из записанных. По результатам, синхронизация работает корректно, выхода за пределы массива нет, записанные данные не затираются до их прочтения, все данные были прочитаны.

**8.1.** **Рассмотреть пример использования сокетов – простейший эхо-сервер.**

Сервер прослушивает заданный порт. При приходе нового соединения создается

новый поток для его обработки. Работа с клиентом организована как бесконечный цикл, в

котором выполняется прием 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 его серверу, и

получение ответа. Если была введена пустая строка, клиент завершается.

**Содержимое файла server.c:**

#include <sys/types.h>  
#include <sys/socket.h>  
#include <netinet/in.h>  
#include <arpa/inet.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <unistd.h>  
  
#define DEF\_PORT 8888  
#define DEF\_IP "127.0.0.1"  
  
// обработка одного клиента  
void\* clientHandler(void\* args) {  
int sock = (int)args;  
char buf[100];  
int res = 0;  
for(;;) {  
bzero(buf,100);  
res = readFix(sock, buf,100, 0);  
if ( res <= 0 ) {  
perror( "Can't recv data from client, ending thread\n" );  
pthread\_exit(NULL);  
}  
printf( "Some client sent: %s\n",buf);  
res = sendFix(sock,buf,0);  
if ( res <= 0 ) {  
perror( "send call failed" );  
pthread\_exit(NULL);  
}  
}  
}  
  
int main( int argc, char\*\* argv) {  
int port = 0;  
if(argc < 2) {  
printf("Using default port %d\n",DEF\_PORT);  
port = DEF\_PORT;  
}  
else  
port = atoi(argv[1]);  
struct sockaddr\_in listenerInfo;  
listenerInfo.sin\_family = AF\_INET;  
listenerInfo.sin\_port = htons( port );  
listenerInfo.sin\_addr.s\_addr = htonl( INADDR\_ANY );  
int listener = socket(AF\_INET, SOCK\_STREAM, 0 );  
if ( listener < 0 ) {  
perror( "Can't create socket to listen: " );  
exit(1);  
}  
int res = bind(listener,(struct sockaddr \*)&listenerInfo, sizeof(listenerInfo));  
if ( res < 0 ) {  
perror( "Can't bind socket" );  
exit( 1 );  
}  
// слушаем входящие соединения  
res = listen(listener,5);  
if (res) {  
perror("Error while listening:");  
exit(1);  
}  
// основной цикл работы  
for(;;) {  
int client = accept(listener, NULL, NULL );  
pthread\_t thrd;  
res = pthread\_create(&thrd, NULL, clientHandler, (void\*)(client));  
if (res){  
printf("Error while creating new thread\n");  
}  
}  
return 0;  
}  
  
int readFix(int sock, char\* buf, int bufSize, int flags) {  
// читаем "заголовок" - сколько байт составляет наше сообщение  
unsigned msgLength = 0;  
int res=recv(sock,&msgLength,sizeof(unsigned),flags|MSG\_WAITALL );  
if (res <= 0)return res;  
if(res > bufSize) {  
printf("Recieved more data, then we can store, exiting\n");  
exit(1);  
}  
// читаем само сообщение  
return recv(sock, buf, msgLength, flags | MSG\_WAITALL);  
}  
  
int sendFix(int sock, char\* buf, int flags) {  
// шлем число байт в сообщении  
unsigned msgLength = strlen(buf);  
int res = send(sock, &msgLength, sizeof(unsigned), flags );  
if (res <= 0)  
return res;  
send(sock, buf, msgLength, flags);  
}

**Содержимое файла client.c:**

#include <sys/types.h>  
#include <sys/socket.h>  
#include <netinet/in.h>  
#include <arpa/inet.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <unistd.h>  
  
#define DEF\_PORT 8888  
#define DEF\_IP "127.0.0.1"  
  
int main( int argc, char\*\* argv) {  
char\* addr;  
int port;  
char\* readbuf;  
if(argc <3) {  
printf("Using default port %d\n",DEF\_PORT);  
port = DEF\_PORT;  
}  
else  
port = atoi(argv[2]);  
if(argc < 2) {  
printf("Using default addr %s\n",DEF\_IP);  
addr = DEF\_IP;  
}  
else  
addr = argv[1];  
// создаем сокет  
struct sockaddr\_in peer;  
peer.sin\_family = AF\_INET;  
peer.sin\_port = htons( port );  
peer.sin\_addr.s\_addr = inet\_addr( addr );  
int sock = socket( AF\_INET, SOCK\_STREAM, 0 );  
if ( sock < 0 ){  
perror( "Can't create socket\n" );  
exit( 1 );  
}  
// присоединяемся к серверу  
int res = connect( sock, ( struct sockaddr \* )&peer, sizeof(peer ) );  
if (res) {  
perror( "Can't connect to server:" );  
exit( 1 );  
}  
// основной цикл программы  
char buf[100];  
for(;;) {  
printf("Input request (empty to exit)\n");  
bzero(buf,100);  
fgets(buf, 100, stdin);  
buf[strlen(buf)-1] = '\0';  
if(strlen(buf) == 0) {  
printf("Bye-bye\n");  
return 0;  
}  
res = sendFix(sock, buf,0);  
if ( res <= 0 ) {  
perror( "Error while sending:" );  
exit( 1 );  
}  
bzero(buf,100);  
res = readFix(sock, buf, 100, 0);  
if ( res <= 0 ) {  
perror( "Error while receiving:" );  
exit(1);  
}  
printf("Server's response: %s\n",buf);  
}  
return 0;  
}  
  
int readFix(int sock, char\* buf, int bufSize, int flags) {  
// читаем "заголовок" - сколько байт составляет наше сообщение  
unsigned msgLength = 0;  
int res=recv(sock,&msgLength,sizeof(unsigned),flags|MSG\_WAITALL);  
if (res <= 0)return res;  
if(res > bufSize) {  
printf("Recieved more data, then we can store, exiting\n");  
exit(1);  
}  
// читаем само сообщение  
return recv(sock, buf, msgLength, flags | MSG\_WAITALL);  
}  
  
int sendFix(int sock, char\* buf, int flags) {  
// число байт в сообщении  
unsigned msgLength = strlen(buf);  
int res = send(sock, &msgLength, sizeof(unsigned), flags );  
if (res <= 0)  
return res;  
send(sock, buf, msgLength, flags);  
}

Для взаимодействия используются TCP сокеты, это значит, что между сервером и клиентом устанавливается логическое соединение, при этом при получении данных из сокета с помощью вызова recv, есть вероятность получить сразу несколько сообщений, или не полностью прочитать сообщение. Поэтому для установления взаимной однозначности между отосланными и принятыми данными используются функции recvFix и sendFix. Принцип их работы следующий: функция sendFix перед посылкой собственно данных посылает «заголовок» - количество байт в посылке. Функция recvFix вначале принимает этот «заголовок», и вторым вызовом recv считывает переданное количество байт. Считать ровно то, количество байт, которое указанно в аргументе функции recv, позволяет флаг MSG\_WAITALL. Если его не использовать и данных в буфере недостаточно, то будет прочитано меньшее количество.

**Результат выполнения программы:**

Клиент:

asdf@BigAwesomeTurtle:~/lab\_4/8\_1$ ./client.out  
Using default port 8888  
Using default addr 127.0.0.1  
Input request (empty to exit)  
Hello  
Server's response: Hello  
Input request (empty to exit)  
something else  
Server's response: something else  
Input request (empty to exit)  
goodbye  
Server's response: goodbye  
Input request (empty to exit)  
  
Bye-bye

Сервер:

asdf@BigAwesomeTurtle:~/lab\_4/8\_1$ ./server.out  
Using default port 8888  
Some client sent: Hello  
Some client sent: something else  
Some client sent: goodbye  
Can't recv data from client, ending thread  
: Success

**Ограничения количества подключений TCP**

Для демонстрации ограничений используем следующий скирпт:

./server.out &

for value in $(seq 1 $1)

do

./client.out &

Done

Для 10 клиентов подключение прошло успешно. Система в норме.

Для 100 клиентов подключение также прошло успешно, но на подключение всех клиентов понадобилось значительное время.

Для 1000 клиентов произошло зависание системы.

**8.2. Клиент-сервер на основе UDP**

Реализация эхо сервера. Для взаимодействия используются UDP сокеты, это значит, что между сервером и клиентом не устанавливается логическое соединение.

**Содержимое файла server.c:**

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <string.h>

#include <netdb.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#define BUFSIZE 1024

void error(char \*msg) {

perror(msg);

exit(1);

}

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

int sockfd; /\* socket \*/

int portno = 5005; /\* port to listen on \*/

int clientlen; /\* byte size of client's address \*/

struct sockaddr\_in serveraddr; /\* server's addr \*/

struct sockaddr\_in clientaddr; /\* client addr \*/

struct hostent \*hostp; /\* client host info \*/

char buf[BUFSIZE]; /\* message buf \*/

char \*hostaddrp; /\* dotted decimal host addr string \*/

int optval; /\* flag value for setsockopt \*/

int n; /\* message byte size \*/

/\*

\* socket: create the parent socket

\*/

sockfd = socket(AF\_INET, SOCK\_DGRAM, 0);

if (sockfd < 0)

error("ERROR opening socket");

/\* setsockopt: Handy debugging trick that lets

\* us rerun the server immediately after we kill it;

\* otherwise we have to wait about 20 secs.

\* Eliminates "ERROR on binding: Address already in use" error.

\*/

optval = 1;

setsockopt(sockfd, SOL\_SOCKET, SO\_REUSEADDR,

(const void \*)&optval , sizeof(int));

/\*

\* build the server's Internet address

\*/

bzero((char \*) &serveraddr, sizeof(serveraddr));

serveraddr.sin\_family = AF\_INET;

serveraddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

serveraddr.sin\_port = htons((unsigned short)portno);

/\*

\* bind: associate the parent socket with a port

\*/

if (bind(sockfd, (struct sockaddr \*) &serveraddr,

sizeof(serveraddr)) < 0)

error("ERROR on binding");

/\*

\* main loop: wait for a datagram, then echo it

\*/

clientlen = sizeof(clientaddr);

while (1) {

/\*

\* recvfrom: receive a UDP datagram from a client

\*/

bzero(buf, BUFSIZE);

n = recvfrom(sockfd, buf, BUFSIZE, 0,

(struct sockaddr \*) &clientaddr, &clientlen);

if (n < 0)

error("ERROR in recvfrom");

/\*

\* gethostbyaddr: determine who sent the datagram

\*/

hostp = gethostbyaddr((const char \*)&clientaddr.sin\_addr.s\_addr,

sizeof(clientaddr.sin\_addr.s\_addr), AF\_INET);

if (hostp == NULL)

error("ERROR on gethostbyaddr");

hostaddrp = inet\_ntoa(clientaddr.sin\_addr);

if (hostaddrp == NULL)

error("ERROR on inet\_ntoa\n");

printf("server received: %s\n", buf);

/\*

\* sendto: echo the input back to the client

\*/

n = sendto(sockfd, buf, strlen(buf), 0,

(struct sockaddr \*) &clientaddr, clientlen);

if (n < 0)

error("ERROR in sendto");

}

}

**Содержимое файла client.c:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <netdb.h>

#define BUFSIZE 1024

void error(char \*msg) {

perror(msg);

exit(0);

}

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

int sockfd, portno = 5005, n;

int serverlen;

struct sockaddr\_in serveraddr;

struct hostent \*server;

char hostname[] = "localhost";

char buf[BUFSIZE];

/\* socket: create the socket \*/

sockfd = socket(AF\_INET, SOCK\_DGRAM, 0);

if (sockfd < 0)

error("ERROR opening socket");

/\* gethostbyname: get the server's DNS entry \*/

server = gethostbyname(hostname);

if (server == NULL) {

fprintf(stderr,"ERROR, no such host as %s\n", hostname);

exit(0);

}

/\* build the server's Internet address \*/

bzero((char \*) &serveraddr, sizeof(serveraddr));

serveraddr.sin\_family = AF\_INET;

bcopy((char \*)server->h\_addr,

(char \*)&serveraddr.sin\_addr.s\_addr, server->h\_length);

serveraddr.sin\_port = htons(portno);

while(1) {

/\* get a message from the user \*/

bzero(buf, BUFSIZE);

printf("Please enter msg: ");

fgets(buf, BUFSIZE, stdin);

/\* send the message to the server \*/

serverlen = sizeof(serveraddr);

n = sendto(sockfd, buf, strlen(buf), 0, &serveraddr, serverlen);

if (n < 0)

error("ERROR in sendto");

/\* print the server's reply \*/

n = recvfrom(sockfd, buf, strlen(buf), 0, &serveraddr, &serverlen);

if (n < 0)

error("ERROR in recvfrom");

printf("Echo from server: %s", buf);

}

return 0;

}

**Результат выполнения программы:**

Клиент 1:

asdf@BigAwesomeTurtle:~/lab\_4/8\_2$ ./client.out  
Please enter msg: Hi  
Echo from server: Hi  
Please enter msg: client 1  
Echo from server: client 1  
Please enter msg: is here  
Echo from server: is here

Клиент 2:

asdf@BigAwesomeTurtle:~/lab\_4/8\_2$ ./client.out  
Please enter msg: Hello  
Echo from server: Hello  
Please enter msg: 1  
Echo from server: 1  
Please enter msg: 2  
Echo from server: 2

Сервер:

asdf@BigAwesomeTurtle:~/lab\_4/8\_2$ ./server.out  
server received: Hi  
server received: client 1  
server received: is here  
server received: Hello  
server received: 1  
server received: 2

**Ограничения количества подключений UDP**

Подключение 10 и 100 клиентов проходит успешно, при попытке подключения 1000 клиентов оперативная память компьютера истощается, и система перестаёт реагировать.

Сокеты являются средством IPC, которое можно использовать не только между процессами на одном компьютере, но и в сетевом режиме.

**4. Вывод**

В процессе выполнения работы были рассмотрены основные механизмы межпроцессного взаимодействия в ОС Unix: сигналы, очереди сообщений, именованные и неименованные каналы, сокеты, разделяемая память и семафоры