**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МОЭВМ**

отчет

**по лабораторной работе №5-6**

**по дисциплине «Операционные системы»**

Тема: «Cредства межпроцессного взаимодействия

в UNIX-подобных ОС»

|  |  |  |
| --- | --- | --- |
| Студент гр. 1384 |  | Кондратенко К. Е. |
| Преподаватель |  | Душутина Е. В. |

Санкт-Петербург

2023

**Цель работы.**

--

**Задание.**

Задание 1. Описание сигналов в unix.

Задание 1.1. Ненадежный сигнал

Задание 1.2. Надежный сигнал

Задание 1.3. Сигналы POSIX реального времени

Задание 2. Неименнованые каналы.

Задание 3. Именнованые каналы.

Задание 4. Очереди сообщений

Задание 5. Семафоры и разделяемая память

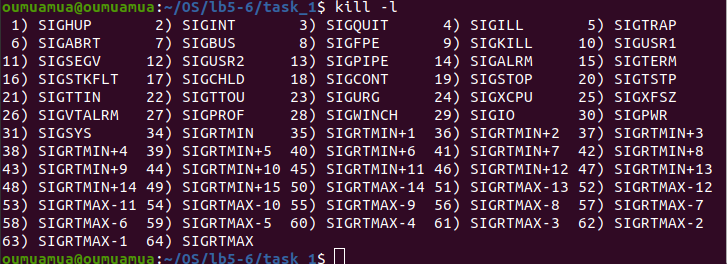
Задание 6. Сокеты

**Ход работы.**

**Задание 1. Описание сигналов в Unix.**

Ознакомимся с полным перечнем сигналов с помощью команды

kill -l:



Видно, что количество сигналов соответсвует разрядности системы (64 битная). Производить системные вызовы можно из терминала, путь ввода команды «kill <pid>» или из программы, подключив иблиотеку «#include <signal.h>» и введя команду «int kill(pid\_t pid, int sig);». Сигналы можно вызыывать вручную, как описано выше, также сигналы могут генерироваться нажатием комбинации клавишь в терминале, а еще ОС может генерировать сигнал, например при выход за границы массива.

Сигналы можно игнорировать, обрабатывать самим или же давать им действовать по умолчанию. Эти дийсвтия называются диспозицией сигнала.

Порожденный вызовом fork() процесс наследует диспозицию сигналов от своего родителя. Однако при вызове exec() диспозиция всех перехватываемых сигналов будет установлена ядром на действие по умолчанию.

В ОС поддерживается ряд функций, позволяющих управлять диспозицией сигналов. Наиболее простой в использовании является функция signal(). Она позволяет устанавливать и изменять диспозицию сигнала.

#include <signal.h>

void (\*signal (int sig,

void (\*disp)(int)))(int);

Аргумент sig определяет сигнал, диспозицию которого нужно изменить. Аргумент disp определяет новую диспозицию сигнала.

Более гибкое управление сигналами предоставляет функция sigaction():

int sigaction( int sig, const struct sigaction \* act, struct sigaction \* oact );

Данная функция позволяет вызывающему процессу получить информацию или установить (или и то и другое) действие, соответствующее какому-либо сигналу или группе сигналов.

Выбор функции управления сигналами определяет свойства сигнала как средства IPC: signal()обеспечивает так называемую ненадежную передачу сигнала, тогда как sigaction() гарантирует надежную передачу. Последнее означает, что если при возникновении сигнала система занята обработкой другого сигнала (назовем его «текущим»), то возникший сигнал не будет потерян, а его обработка будет отложена до окончания текущего обработчика.

**Задание 1.1. Ненадежный сигнал**

Создать программу, позволяющую изменить диспозицию сигналов, а именно, установить:

- обработчик пользовательских сигналов SIGUSR1 и SIGUSR2;

- реакцию по умолчанию на сигнал SIGINT;

- игнорирование сигнала SIGCHLD;

Для выполнения поставленной задачи была разработана программа signal\_task1.c на языке си.

Разработанный программный код:

#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-потомок

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

// отправляем сигналы родителю

if(kill(getppid(),SIGUSR1) != 0)

{

printf("Error while sending SIGUSR1\n");

exit(1);

}

printf("Successfully sent SIGUSR1\n");

return 0;

}

// программа-родитель

wait(NULL);

// ждем сигналов

for(;;)

{

pause();

}

return 0;

}

Скомипилируем и запустим программу:  
oumuamua@oumuamua:~/OS/lb5-6/task\_1$ ./st\_1

Father started: pid = 5025,ppid = 2452

Son started: pid = 5026,ppid = 5025

Successfully sent SIGUSR1

Catched signal SIGUSR1

Parent = 2452

Откроем новый терминал и введем команды:

oumuamua@oumuamua:~/OS/lb5-6/task\_1$ kill -SIGUSR2 5025

oumuamua@oumuamua:~/OS/lb5-6/task\_1$ kill -SIGCHLD 5025

oumuamua@oumuamua:~/OS/lb5-6/task\_1$ kill -SIGINT 5025

Сообщение в старом терминале:

Catched signal SIGUSR2

Parent = 2452

oumuamua@oumuamua:~/OS/lb5-6/task\_1$

Видно, что сигнал SIGUSR2 был пойман и обработна, сигнал SIGCHLD был прогнорирован, а SIGINT завершил процесс.

Запустим программу еще раз.

oumuamua@oumuamua:~/OS/lb5-6/task\_1$ ./st\_1

Father started: pid = 5194,ppid = 2452

Son started: pid = 5195,ppid = 5194

Successfully sent SIGUSR1

Catched signal SIGUSR1

Parent = 2452

Откроем второй терминал и введем дважды команду «kill -SIGUSR2 5194»

Сообщения в старом терминале:

Catched signal SIGUSR2

Parent = 2452

User defined signal 2

oumuamua@oumuamua:~/OS/lb5-6/task\_1$

Видно, что первый SIGUSR2 был пойман и обработна, после чего был восстановлен стандартный обработчик и получен второй сигнал SIGUSR2, который завершил программу.

**Задание 1.2. Надежный сигнал.**

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.

Для выполнения данного задания разработана программа signal\_task1.2.c на языке си.

Код 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 (\*mysig(int sig,void (\*hnd)(int)))(int)

{

// надежная обработка сигналов

struct sigaction act, oldact;

act.sa\_handler = hnd;

sigemptyset(&act.sa\_mask);

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

act.sa\_flags = 0;

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

return SIG\_ERR;

return act.sa\_handler;

}

void hndUSR1(int sig)

{

if(sig != SIGUSR1)

{

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

return;

}

printf("SIGUSR1 catched\n");

sleep(60);

}

int main()

{

mysig(SIGUSR1,hndUSR1);

for(;;)

{

pause();

}

return 0;

}

Пример работы программы:

oumuamua@oumuamua:~/OS/lb5-6/task\_1/sig1.2$ ./sigact &

[1] 5821

oumuamua@oumuamua:~/OS/lb5-6/task\_1/sig1.2$ SIGUSR1 catched

oumuamua@oumuamua:~/OS/lb5-6/task\_1/sig1.2$ jobs

[1]+ Running ./sigact &

oumuamua@oumuamua:~/OS/lb5-6/task\_1/sig1.2$ kill -SIGINT %1

oumuamua@oumuamua:~/OS/lb5-6/task\_1/sig1.2$ jobs

[1]+ Running ./sigact &

oumuamua@oumuamua:~/OS/lb5-6/task\_1/sig1.2$ jobs

[1]+ Interrupt ./sigact

oumuamua@oumuamua:~/OS/lb5-6/task\_1/sig1.2$ jobs

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).

Изменим обработчик и запишем полученную программу в файл signal\_task\_1.2.1.c, скомпилируем с помощью команды «gcc -w -o sigact\_2 signal\_task\_1.2.1.c»

Пример запуска программы:

oumuamua@oumuamua:~/OS/lb5-6/task\_1/sig1.2$ ./sigact\_2 &

[1] 6735

oumuamua@oumuamua:~/OS/lb5-6/task\_1/sig1.2$ kill -SIGUSR1 %1

oumuamua@oumuamua:~/OS/lb5-6/task\_1/sig1.2$ SIGUSR1 catched, sending SIGINT

oumuamua@oumuamua:~/OS/lb5-6/task\_1/sig1.2$ jobs

[1]+ Running ./sigact\_2 &

oumuamua@oumuamua:~/OS/lb5-6/task\_1/sig1.2$ jobs

[1]+ Interrupt ./sigact\_2

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).

**Задание 1.3. Сигналы POSIX реального времени**

Некоторые реализации POSIX ОС могут обрабатывать все сигналы как сигналы реального времени, но для UNIX-подобных ОС это не является обязательным. Если мы хотим, чтобы сигналы гарантированно обрабатывались как сигналы реального времени, мы должны:

• использовать сигналы с номерами в диапазоне от SIGRTMIN до

SIGRTMAX

• должны указать флаг SA\_SIGINFO при вызове sigaction() с установкой

обработчика сигнала

14• обработчик сигнала реального времени, устанавливаемый с флагом

SA\_SIGINFO, объявляется как:

void func(int signo, siginfo\_t \*info, void \*context); где

signo— номер сигнала,

siginfo\_t — структура, определяемая как

typedef struct {

int si\_signo; /\* то же, что и signo \*/

int si\_code; /\* SI\_{USER,QUEUE,TIMER,ASYNCIO,MESGQ} \*/

union sigval si\_value; /\* целое или указатель от отправителя \*/

} siginfo\_t;

на что указывает context — зависит от реализации.

Таким образом, сигналы реального времени несут больше

информации, чем прочие сигналы (при отправке сигнала, не

обрабатываемого как сигнал реального времени, единственным

аргументом обработчика является номер сигнала).

• SIGRTMIN и SIGRTMAX – это еще и макросы (вызывающие sysconf),

которые позволяют изменять сами эти значения.

«Характеристики сигналов реального времени» означает следующее:

• Сигналы помещаются в очередь.

• Если сигнал будет порожден несколько раз, он будет несколько раз

получен адресатом. Более того, повторения одного и того же сигнала

доставляются в порядке очереди (FIFO). Если же сигналы в очередь не

помещаются, неоднократно порожденный сигнал будет получен лишь

один раз.

• Когда в очередь помещается множество неблокируемых сигналов в

диапазоне SIGRTMIN—SIGRTMAX, сигналы с меньшими номерами

доставляются раньше сигналов с большими номерами. То есть сигнал с

номером SIGRTMIN имеет «больший приоритет», чем сигнал с номером

SIGRTMIN+1, и т.д.

**Задание 2. Неименнованные каналы**

Организуем программу (файл pipe.c) так, чтобы процесс-родитель создавал неименованный канал, создавал потомка, закрывал канал на запись и записывал в произвольный текстовый файл считываемую из канала информацию. В функции процесса-потомка будет входить считывание данных из файла и запись их в канал.

Исходный код 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;

}

Примеры вызова — создадим и заполним файл from.txt, выведем это содержание, создадим пустой файл to.txt, скомпилируем программу и запустим исполняемый файл.  
 oumuamua@oumuamua:~/OS/lb5-6/task\_2$ nano from.txt

oumuamua@oumuamua:~/OS/lb5-6/task\_2$ cat from.txt

string [1]

string [2]

string [3]

EOF

oumuamua@oumuamua:~/OS/lb5-6/task\_2$ touch to.txt

oumuamua@oumuamua:~/OS/lb5-6/task\_2$ gcc -w -o pipe pipe.c

oumuamua@oumuamua:~/OS/lb5-6/task\_2$ ./pipe

Using default fileNames 'from.txt','to.txt'

pipe is successfully created

Read from pipe: string [1]

string [2]

string [3]

EOF

Выведем содержимое файла to.txt

oumuamua@oumuamua:~/OS/lb5-6/task\_2$ cat to.txt

string [1]

string [2]

string [3]

EOF

Видно, что изначально пустой файл успешно заполнен.

Программа успешно отработало, задание выполнено.

Главное применение неименованных каналов в ОС Unix – реализация конвейеров команд в интерпретаторах командной строки.

Интерпретатор создает процессы и каналы между ними так, что открытый для чтения конец каждого канала будет подключен к стандартному потоку ввода, а открытый на запись — к стандартному потоку вывода.

**Задание 3. Именнованный каналы.**

Именнованные каналы работают почти также, как и неименнованные, однако каждому канало соответсвует имя в файловой системе — это позваляет неродственным процессам работать с одним каналом.

FIFO — очередь. Каналы могут быть открыты либо на чтение, либо на запись.

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, прав доступа (установим открытые для всех права доступа на чтение и на запись S\_IFIFO | 0666). Откроем один канал на запись (chan1), другой — на чтение (chan2) и запустим серверную часть программы. В серверной части программы: запишем имя файла в канал 1 (для записи) функцией write(); прочитаем данные из канала 2 и выведем на экран.

В файле client.c запрограммируем функции: открытия каналов для чтения (chan1) и записи (chan2). Из первого канал читается имя файла, во второй канал пишется его содержимое.

Код программы server.c:

oumuamua@oumuamua:~/OS/lb5-6/task\_3$ cat server.c

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#define DEF\_FILENAME "testFile.txt"

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

{

char fileName[30];

if(argc < 2)

{

printf("Using default file name '%s'\n",DEF\_FILENAME);

strcpy(fileName,DEF\_FILENAME);

}

else

strcpy(fileName,argv[1]);

// создаем два канала

int res = mknod("channel1", S\_IFIFO | 0666,0);

if(res)

{

printf("Can't create first channel\n");

exit(1);

}

res = mknod("channel2", S\_IFIFO | 0666,0);

if(res)

{

printf("Can't create second channel\n");

exit(1);

}

// открываем первый канал для записи

int chan1 = open("channel1",O\_WRONLY);

if(chan1 == -1)

{

printf("Can't open channel for writing\n");

exit(0);

}

// открываем второй канал для чтения

int chan2 = open("channel2",O\_RDONLY);

if(chan2 == -1)

{

printf("Can't open channe2 for reading\n");

exit(0);

}

// пишем имя файла в первый канал

write(chan1,fileName,strlen(fileName));

// читаем содержимое файла из второго канала

char buf [100];

for(;;)

{

bzero(buf,100);

res = read(chan2,buf,100);

if(res <= 0)

break;

printf("Part of file: %s\n");

}

close(chan1);

close(chan2);

unlink("channel1");

unlink("channel2");

return 0;

}

Код программы client.c

oumuamua@oumuamua:~/OS/lb5-6/task\_3$ cat 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;

}

Результат работы при запуски сревара в одном терминале и клиента в другом:

Using default file name 'testFile.txt'

Part of file: string [1]

string [2]

string [3]

EOF

Содержимое файла testFile.txt

oumuamua@oumuamua:~/OS/lb5-6/task\_3$ cat testFile.txt

string [1]

string [2]

string [3]

EOF

Логика работы: 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().

Программа создает каналы. Они служит для получения данных о расположении FIFO в адресном пространстве ядра и его состоянии, а не являются буффером хранения информации. Чтобы проверить это изменим сервер так, чтобы он ожидал подтверждение пользователя, чтобы мы могли проверить — меняется ли размерность канала при создании и при передачи информации.

Проверим это, запустив измененный сервер и клиент. Сервер просит подтвердить работу, в это время откроем еще один терминал и вызовем команду:

oumuamua@oumuamua:~/OS/lb5-6/task\_3/3.1$ ls -l | grep "channel"

prw-rw-r-- 1 oumuamua oumuamua 0 апр 27 18:08 channel1

prw-rw-r-- 1 oumuamua oumuamua 0 апр 27 18:08 channel2

Теперь подтвердим дейсвтия для сервера и посмотрим размерность каналов после.

oumuamua@oumuamua:~/OS/lb5-6/task\_3/3.1$ ls -l | grep "channel"

prw-rw-r-- 1 oumuamua oumuamua 0 апр 27 18:12 channel1

prw-rw-r-- 1 oumuamua oumuamua 0 апр 27 18:12 channel2

При этом сервер отработал в шататном режиме:

oumuamua@oumuamua:~/OS/lb5-6/task\_3/3.1$ ./server

Using default file name 'testFile.txt'

Waiting for clint write to channnel

Part of file: string [1]

string [2]

string [3]

EOF

Waiting after clint write to channnel

Размер каналов не меняется. Это свидетльствует о том, что каналы используются не для хранения пересылаемых данных, а для предоставление системе информации об этих данных.

**Ограничения**:

На неименованные каналы и каналы FIFO системой накладываются всего два ограничения: OPEN\_MAX — максимальное количество дескрипторов, которые могут быть одновременно открыты некоторым процессом (POSIX устанавливает для этой величины ограничение снизу);

PIPE\_BUF — максимальное количество данных, для которого гарантируется атомарность операции записи (POSIX требует по менее 512 байт).

**Задание 4. Очереди сообщений**

Очередь сообщений гарантирует сохранение размерности сообщение — это гарантирует то, что сообщения не смешаются. Процесс, который читает сообщения, может читать не всю очередь, а только необходимые сообщения.

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.

Исходный код программы server.c

oumuamua@oumuamua:~/OS/lb5-6/task\_4$ cat 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("no got key for the key file %s and id 'Q'\n",keyFile);

exit(1);

}

queue = msgget(key,IPC\_CREAT | 0666);

if (queue < 0)

{

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

exit(4);

}

// до этого момента вызывали exit(), а не kill, т.к. очередь

// еще не была создана

signal(SIGINT,intHandler);

// основной цикл работы сервера

Message mes;

int res;

for(;;)

{

bzero(mes.buf,100);

// получаем первое сообщение с типом 1

res = msgrcv(queue,&mes,sizeof(Message),1L,0);

if(res < 0)

{

printf("Error while recving msg\n");

kill(getpid(),SIGINT);

}

printf("Client's request: %s\n",mes.buf);

// шлем клиенту сообщение с типом 2, что все ок

mes.type = 2L;

bzero(mes.buf,100);

strcpy(mes.buf,"OK");

res = msgsnd(queue,(void\*)&mes,sizeof(Message),0);

if(res != 0)

{

printf("error while sending msg\n");

kill(getpid(),SIGINT);

}

}

return 0;

}  
  
 Исходный код программы client.c

oumuamua@oumuamua:~/OS/lb5-6/task\_4$ cat client.c

#include <stdio.h>

#include <stdlib.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <sys/types.h>

#include <signal.h>

#include <string.h>

#define DEF\_KEY\_FILE "key"

typedef struct

{

long type;

char buf[100];

} Message;

int queue;

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

{

char keyFile[100];

bzero(keyFile,100);

if(argc < 2)

{

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

strcpy(keyFile,DEF\_KEY\_FILE);

}

else

strcpy(keyFile,argv[1]);

key\_t key;

key = ftok(keyFile,'Q');

if(key == -1)

{

printf("no got key for key file %s and id 'Q'\n",keyFile);

exit(1);

}

queue = msgget(key,0);

if (queue < 0)

{

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

exit(4);

}

// основной цикл работы программы

Message mes;

int res;

for(;;)

{

bzero(mes.buf,100);

// читаем сообщение с консоли

fgets(mes.buf,100,stdin);

mes.buf[strlen(mes.buf) - 1] = '\0';

// шлем его серверу

mes.type = 1L;

res = msgsnd(queue,(void\*)&mes,sizeof(Message),0);

if(res != 0)

{

printf("Error while sending msg\n");

exit(1);

}

// получаем ответ, что все хорошо

res = msgrcv(queue,&mes,sizeof(Message),2L,0);

if(res < 0)

{

printf("Error while recving msg\n");

exit(1);

}

printf("Server's response: %s\n",mes.buf);

}

return 0;

}

Пример работы:

Вызов сервера:

oumuamua@oumuamua:~/OS/lb5-6/task\_4$ ./server

Using default key file key

Client's request: hello1

Client's request: hello 2

Вызов первого клиента:

oumuamua@oumuamua:~/OS/lb5-6/task\_4$ ./client

Using default key file key

hello1

Server's response: OK

Вызов второго клиента:

oumuamua@oumuamua:~/OS/lb5-6/task\_4$ ./client

Using default key file key

hello 2

Server's response: OK

Таким образом нам удалось организовать очередь сообщений. Доходят успешно.

С помощью команды ipcs -l узнаем максимальные размер сообщения и очереди.

oumuamua@oumuamua:~/OS/lb5-6/task\_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) = 18014398509465599

max total shared memory (kbytes) = 18014398509481980

min seg size (bytes) = 1

------ Semaphore Limits --------

max number of arrays = 32000

max semaphores per array = 32000

max semaphores system wide = 1024000000

max ops per semop call = 500

semaphore max value = 32767

Как видно из вызова: максимальный раз сообщения 8192 байт, максимальный размер очереди 32000, максимальный раз очереди по умолчания 16384.

**Задание 5. Семафоры и разделяемая память**

Семафоры нужны для обеспечение транзакционнсти (синхронизации) доступа к разделяемой памяти — пока ее использует на запись один процесс, другой не имеет права переписывать ее и наоборот. Такая система позволяет избежать коллизий записи-чтения.

Задача: 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.

Код программы читателя :  
 Код программы писателя:

Резаультат работы первого писателя:  
 oumuamua@oumuamua:~/OS/lb5-6/task\_5/t5.1$ ./client

Using default key file key

Type message to serever. Empty string to finish

f1

Type message to serever. Empty string to finish

f1.1

Type message to serever. Empty string to finish

f1.2

Type message to serever. Empty string to finish

f1 - done

Результат работы второго писателя:

oumuamua@oumuamua:~/OS/lb5-6/task\_5/t5.1$ ./client

Using default key file key

Type message to serever. Empty string to finish

f2

Type message to serever. Empty string to finish

f2.2

Type message to serever. Empty string to finish

f2.3

Type message to serever. Empty string to finish

f2 - done

Результат работы читателя:

oumuamua@oumuamua:~/OS/lb5-6/task\_5/t5.1$ ./server

Using default key file key

Client's message: f1

Client's message: f1.1

Client's message: f2

Client's message: f2.2

Client's message: f2.3

Client's message: f1.2

Client's message: f1 - done

Client's message: f2 - done

Сообщения доходят, при том, что мы писали сообщения в оба терминала поочередно - все работает, благодаря системы семафоров.

Теперь реализуем задачу, похожую на предыдущую, но буфер сделаем неединичным. Необходимо переписать логика так. Чтобы семафоры контролировали количество заполненных и незаполненных ячеек, иначе будет выраждение буфера, если мы после каждой публикации будем считывать данные.

Код программы читателя:

Код программы писателя:

**Задание 6. Сокеты**