# МИНОБРНАУКИ РОССИИ

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

# «ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

**Кафедра МО ЭВМ**

# ОТЧЕТ

**по лабораторной работе №5**

# по дисциплине «Операционные системы»

**Тема: «Средства межпроцессного взаимодействия в ОС семейства Unix»**

| Студентка гр. 1303 |  | Чубан Д.В. |
| --- | --- | --- |
| Преподаватель |  | Душутина Е.В |

Санкт-Петербург 2023

# Цели работы.

1. Изучить средства межпроцессного взаимодействия в ОС семейства Unix.
2. Понять, что из себя представляют: сигналы, анонимные (неименованные) каналы, именованные каналы, очереди сообщений, семафоры, разделяемая память, сокеты. Научиться работать с таким набором средств.

# Выполнение работы.

1) Создать программу, позволяющую изменить диспозицию сигналов. 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.

*Исполняемый код task1.c:*

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

#include <stdlib.h>

#include <wait.h>

static void sigHandler(int sig) {

printf("Catched signal %s\n",sig == SIGUSR1 ? "SIGUSR1": "SIGUSR2");

printf("Parent = %d\n",getppid());

// восстанавливаем 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;

}

*Запустим программу task1.c:*

defrozen@defrozen-ubu:~$ gcc l1.c && ./a.out

Father started: pid = 2413,ppid = 2273

Son started: pid = 2414, ppid = 2413

Catched signal SIGUSR1

Parent = 2273

Successfully sent SIGUSR1

Из результата можно увидеть, что была изменена диспозиция сигналов с помощью signal (), а впоследствии восстановлена старая диспозиция. Также был отправлен сигнал процессу-родителю с помощью команды kill () и выведено сообщение об удачной отправке.

*Далее отправим еще сигналы процессу родителю из командной строки, для этого откроем новое окно терминала и пропишем следующие команды:*

defrozen@defrozen-ubu:~$ kill -SIGUSR2 2413

defrozen@defrozen-ubu:~$ kill -SIGCHLD 2413

defrozen@defrozen-ubu:~$ kill -SIGINT 2413

*На выходе получим следующее:*

Catched signal SIGUSR2

Parent = 2273

То есть сигнал SIGUSR2 был пойман, на следующий сигнал не последовало никакой реакции, так как диспозиция – SIG\_IGN, а третий сигнал завершил работу программы, так как он диспозиция – SIG\_DFL. Если бы дважды был отправлен сигнал SIGUSR2 или SIGUSR1, то программа бы тоже завершилась, так как после первого раза была бы поставлена обработка по умолчанию.

2)Теперьповторим эксперимент для других сигналов и процессов, порождаемых в разных файлах; а также для потоков одного и разных процессов.

*Исполняемый код task2.c:*

#include <stdio.h>

#include <stdlib.h>

#include <signal.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

void handler(int sig) {

printf("Получен сигнал SIGQUIT\n");

signal(SIGQUIT, SIG\_DFL); // Изменяем диспозицию сигнала SIGQUIT на значение по умолчанию

}

int main() {

pid\_t pid;

int status;

// Устанавливаем функцию обработки сигнала для SIGQUIT

signal(SIGQUIT, handler);

// Создаем дочерний процесс с помощью fork

pid = fork();

if (pid == -1) {

printf("Ошибка при создании процесса");

exit(1);

}

if (pid == 0) {

// Дочерний процесс

execl("./file2", "file2", NULL);

exit(0);

} else {

// Родительский процесс

// Отправляем сигнал SIGQUIT

sleep(1);

printf("Отправляем SIGQUIT дочернему процессу...\n");

kill(pid, SIGQUIT);

waitpid(pid, &status, 0);

printf("Дочерний процесс завершен с кодом: %d\n", WEXITSTATUS(status));

}

return 0;

}

*Исполняемый код file2:*

#include <stdio.h>

#include <stdlib.h>

#include <signal.h>

#include <unistd.h>

int main() {

printf("Дочерний процесс начал работу...\n");

// Бесконечный цикл, в котором процесс ждет сигнала

while (1) {

printf("Дочерний процесс ждет сигнала...\n");

sleep(1);

}

return 0;

}

*Запустим программу task2.c:*

defrozen@defrozen-ubu:~$ gcc l2.c && ./a.out

Дочерний процесс начал работу...

Дочерний процесс ждет сигнала...

Отправляем SIGQUIT дочернему процессу…

Дочерний процесс завершен с кодом: 0

# Таким образом, с помощью fork был создан дочерний процесс, который выполнял file2 до тех пор, пока родительский процесс не отправил сигнал SIGQUIT дочернему процессу, после этого диспозиция была восстановлена, а дочерний процесс заершен, после чего был завершен и родительский процесс.

# 3) Теперь изменим диспозицию сигнала для потока:

*Исполняемый код task3.c:*

#include <stdio.h>

#include <stdlib.h>

#include <signal.h>

#include <pthread.h>

#include <unistd.h>

int i = 0;

void handler(int sig) {

printf("Поток %ld получил сигнал %d\n", pthread\_self(), sig);

signal(sig, SIG\_DFL); // изменяем диспозицию сигнала на значение по умолчанию

i++;

}

void \*thread\_func(void \*arg) {

// установка функций обработки сигналов для потока

signal(SIGTERM, handler);

signal(SIGINT, handler);

printf("Поток %ld начал работу...\n", pthread\_self());

// бесконечный цикл, в котором поток ждет сигналов

while (1) {

printf("Поток %ld ждет сигналов...\n", pthread\_self());

sleep(1);

}

return NULL;

}

int main() {

pthread\_t thread1, thread2;

int status1, status2;

pid\_t pid = getpid(); // получить PID процесса

printf("PID процесса: %d\n", pid);

pthread\_t tid = pthread\_self(); // получить TID потока

printf("TID потока: %lu\n", tid); // вывести TID на экран

// создание первого потока

status1 = pthread\_create(&thread1, NULL, thread\_func, NULL);

if (status1) {

printf("Ошибка при создании потока 1\n");

exit(1);

}

// создание второго потока

status2 = pthread\_create(&thread2, NULL, thread\_func, NULL);

if (status2) {

printf("Ошибка при создании потока 2\n");

exit(1);

}

printf("Потоки созданы. Запуск...\n");

// ждем закрытия программы

while (1) {

pthread\_kill(thread1, SIGINT);

sleep(1);

pthread\_kill(thread2, SIGINT);

}

return 0;

}

*Запустим программу task3.c:*

# defrozen@defrozen-ubu:~$ gcc l3.c && ./a.out

# PID процесса: 17043

# TID потока: 140565742778176

# Потоки созданы. Запуск...

# Поток 140565731599936 начал работу...

# Поток 140565731599936 ждет сигналов...

# Поток 140565739992640 получил сигнал 2

# Поток 140565739992640 начал работу...

# Поток 140565739992640 ждет сигналов...

# Поток 140565731599936 получил сигнал 2

# Поток 140565731599936 ждет сигналов...

# Поток 140565739992640 получил сигнал 2

# Поток 140565739992640 ждет сигналов...

# Поток 140565739992640 ждет сигналов...

# Таким образом, была создана программа, изменяющая диспозицию сигнала потоков, изменив диспозицию сигналов дочерних потоков, при получении сигнала потоком-потомком дважды будет обработан сигнал обработчиком, после чего программа завершится(так как будем установлен обработчик по умолчанию).

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

*Исполняемый код task5.c:*

include <stdio.h>

#include <signal.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <unistd.h>

#include <stdlib.h>

#include <fcntl.h>

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

// надежная обработка сигналов

struct sigaction act,oldact;

act.sa\_handler = hnd;

sigemptyset(&act.sa\_mask);

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

act.sa\_flags = 0;

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

return SIG\_ERR;

return act.sa\_handler;

}

void hndUSR1(int sig) {

if(sig != SIGUSR1) {

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

return;

}

printf("SIGUSR1 catched\n");

sleep(60);

}

int main() {

mysig(SIGUSR1,hndUSR1);

for(;;) {

pause();

}

return 0;

}

*Запустим программу task4.c в фоновом режиме:*

defrozen@defrozen-ubu:~$ gcc l5.c -o l5

defrozen@defrozen-ubu:~$ ./l5 &

[1] 17168

defrozen@defrozen-ubu:~$ kill -SIGUSR1 %1

SIGUSR1 catched

defrozen@defrozen-ubu:~$ kill -SIGINT %1

defrozen@defrozen-ubu:~$ jobs

[1]+ Запущен ./l5 &

defrozen@defrozen-ubu:~$ jobs

[1]+ Запущен ./l5 &

defrozen@defrozen-ubu:~$ jobs

[1]+ Прерывание ./l5

Таким образом, в данной программе произошло временное блокирование сигнала SIGINT на 60 секунд, после чего данный сигнал был обработан и программа завершилась.

Теперь изменим обработчик сигнала таким образом, чтобы из него производилась отправка другого сигнала, для этого немного изменим программу.

*Исполняемый код task4\_2.c:*

#include <stdio.h>

#include <signal.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <unistd.h>

#include <stdlib.h>

#include <fcntl.h>

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

// надежная обработка сигналов

struct sigaction act,oldact;

act.sa\_handler = hnd;

sigemptyset(&act.sa\_mask);

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

act.sa\_flags = 0;

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

return SIG\_ERR;

return act.sa\_handler;

}

void hndUSR1(int sig) {

if(sig != SIGUSR1) {

printf("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;

}

*Запустим программу task4\_2.c в фоновом режиме:*

defrozen@defrozen-ubu:~$ ./l5\_2 &

[1] 17305

defrozen@defrozen-ubu:~$ kill -SIGUSR1 %1

defrozen@defrozen-ubu:~$ SIGUSR1 catched, sending SIGINT

jobs

[1]+ Запущен ./l5\_2 &

defrozen@defrozen-ubu:~$ jobs

[1]+ Прерывание ./l5\_2

Таким образом, при генерации сигнала SIGINT из обработчика другого сигнала обработка сгенерированного сигнала задерживается до конца выполнения текущего обработчика, то есть результат такой же как и в предыдущем эксперименте.

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

*Код программы t23.c*:

#include <signal.h>

#include <pthread.h>

#include <stdio.h>

#include <sys/types.h>

#include <linux/unistd.h>

#include <sys/syscall.h>

#include <stdlib.h>

pthread\_t t1, t2;

void hnd(int sig) {

if (sig == SIGUSR1) {

printf("Caught SIGUSR1 %d in Thread 2\n", sig);

signal(sig, hnd);

} else if (sig == SIGUSR2) {

printf("Caught SIGUSR2 %d in Thread 2\n", sig);

signal(sig, hnd);

} else if (sig == SIGRTMIN + 1) {

printf("Caught SIGRTMIN+1 %d in Thread 2\n", sig);

signal(sig, hnd);

} else if (sig == SIGRTMIN + 3) {

printf("Caught SIGRTMIN+3 %d in Thread 2\n", sig);

signal(sig, hnd);

} else if (sig == SIGRTMIN + 5) {

printf("Caught SIGRTMIN+5 %d in Thread 2\n", sig);

signal(sig, hnd);

} else {

printf("Caught unknown signal %d in Thread 2\n", sig);

}

}

void\* thread1() {

int i, count = 0;

int tid, pid;

tid = syscall(SYS\_gettid);

pid = getpid();

printf("Thread\_1 with thread id = %d and pid = %d is started\n", tid, pid);

int n = 5; //можно установить любым

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

count += 1;

printf("Thread\_1: step %d\n", count);

sleep(5); // отправка обычных сигналов из первой нити во вторую

pthread\_kill(t2, SIGUSR2);

pthread\_kill(t2, SIGUSR1);

pthread\_kill(t2, SIGUSR1);

pthread\_kill(t2, SIGRTMIN + 3); // отправка сигналов реального времени

pthread\_kill(t2, SIGRTMIN + 1);

pthread\_kill(t2, SIGRTMIN + 5);

pthread\_kill(t2, SIGRTMIN + 1);

}

}

void\* thread2() {

int i, count = 0;

int tid, pid;

tid = syscall(SYS\_gettid);

pid = getpid();

printf("Thread\_2 with thread id = %d and pid = %d is started\n", tid, pid);

int m = 10; //можно подобрать на свое усмотрение, чтобы все сигналы успевали доставиться

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

count += 1;

sleep(1);

printf("Thread\_2: step %d\n", count);

}

}

int main() {

signal(SIGUSR1, hnd);

signal(SIGUSR2, hnd);

signal(SIGRTMIN + 1, hnd);

signal(SIGRTMIN + 3, hnd);

signal(SIGRTMIN + 5, hnd);

pthread\_create(&t1, NULL, thread1, NULL);

pthread\_create(&t2, NULL, thread2, NULL);

pthread\_join(t1, NULL);

pthread\_join(t2, NULL);

return 0;

}

*Запустим программу t23.c:*

*defrozen@defrozen-ubu:~$ gcc t23.c && ./a.out*

Thread\_1 with thread id = 5582 and pid = 5581 is started

Thread\_1: step 1

Thread\_2 with thread id = 5583 and pid = 5581 is started

Thread\_2: step 1

Thread\_2: step 2

Thread\_2: step 3

Thread\_2: step 4

Thread\_1: step 2

Caught SIGRTMIN+5 39 in Thread 2

Caught SIGRTMIN+3 37 in Thread 2

Caught SIGRTMIN+1 35 in Thread 2

Caught SIGRTMIN+1 35 in Thread 2

Caught SIGUSR2 12 in Thread 2

Caught SIGUSR1 10 in Thread 2

Thread\_2: step 5

Thread\_2: step 6

Thread\_2: step 7

Thread\_2: step 8

Thread\_2: step 9

Thread\_1: step 3

Caught SIGRTMIN+5 39 in Thread 2

Caught SIGRTMIN+3 37 in Thread 2

Caught SIGRTMIN+1 35 in Thread 2

Caught SIGRTMIN+1 35 in Thread 2

Caught SIGUSR2 12 in Thread 2

Caught SIGUSR1 10 in Thread 2

Thread\_2: step 10

Thread\_1: step 4

Thread\_1: step 5

Таким образом, из вывода программы можно наглядно убедиться в том, что SIGRTMIN+5 имеет более высокий приоритет, чем SIGRTMIN+3, SIGRTMIN+3 приоритетнее, чем SIGRTMIN+1, а SIGRTMIN+1 приоритетнее, чем SIGRUSR1.

Программа устанавливает обработчик сигналов handler для сигналов SIGRTMIN и SIGRTMAX с помощью функции sigaction (). То есть сигналы реального времени имеют свой приоритет. Когда в очередь помещается множество неблокируемых сигналов в диапазоне SIGRTMIN—SIGRTMAX, сигналы с меньшими номерами доставляются раньше сигналов с большими номерами.

Теперь проанализируем, какая очередь у обычных сигналов.

*Код программы l6.c*:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <signal.h>

// Массив сигналов, которые будут отправляться

int signals[] = {SIGUSR1, SIGINT, SIGUSR2};

// Счетчики для подсчета количества обработанных сигналов

int counter\_usr1 = 0;

int counter\_usr2 = 0;

// Обработчик сигналов SIGUSR1 и SIGUSR2

void sig\_handler(int signo)

{

if (signo == SIGUSR1) {

printf("SIGUSR1 signal received\n");

counter\_usr1++;

} else if (signo == SIGUSR2) {

printf("SIGUSR2 signal received\n");

counter\_usr2++;

}

else if(signo== SIGINT){

printf("SIGINT signal received\n");

}

}

int main(void)

{

// Установка обработчика сигналов

if (signal(SIGUSR1, sig\_handler) == SIG\_ERR) {

printf("Cannot set signal handler for SIGUSR1\n");

exit(1);

}

if (signal(SIGUSR2, sig\_handler) == SIG\_ERR) {

printf("Cannot set signal handler for SIGUSR2\n");

exit(1);

}

if (signal(SIGINT, sig\_handler) == SIG\_ERR) {

printf("Cannot set signal handler for SIGINT\n");

exit(1);

}

// Отправляем сигналы в порядке FIFO

for (int i = 0; i < 3; i++) {

// printf("Sending signal %d\n", signals[i]);

kill(getpid(), signals[i]);

usleep(100); // задержка для того, чтобы сигналы обрабатывались по одному

}

return 0;

}

*Запустим программу l6.c:*

defrozen@defrozen-ubu:~$ gcc l6.c && ./a.out

SIGUSR1 signal received

SIGINT signal received

SIGUSR2 signal received

Таким образом, обычные сигналы были в очереди по принципу FIFO, то есть первый отосланный сигнал был обработан первым.

Теперь экспериментально подтвердим, что обработка равноприоритетных сигналов реального времени происходит в порядке FIFO.

*Код программы l7.c*:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <signal.h>

#define SIG\_COUNT 4

// Массив сигналов, которые будут отправляться

int signals[SIG\_COUNT];

// Счетчик для подсчета количества обработанных сигналов

int counter = 0;

// Обработчик сигналов SIGRTMIN

void sig\_handler(int signo, siginfo\_t\* info, void\* context)

{

printf("Signal %d received\n", signo);

printf("Signal %d processed\n", \*(int\*)info->si\_value.sival\_ptr);

counter++;

usleep(100); // задержка для того, чтобы следующий сигнал обрабатывался только после того, как предыдущий был полностью обработан

}

int main(void)

{

struct sigaction sig\_action;

// Инициализация сигналов для отправки

for (int i = 0; i < SIG\_COUNT; i++) {

signals[i] = SIGRTMIN;

}

// Инициализация обработчика для sigaction

sigemptyset(&sig\_action.sa\_mask);

sig\_action.sa\_flags = SA\_SIGINFO;

sig\_action.sa\_sigaction = sig\_handler;

// Установка обработчика сигналов

if (sigaction(SIGRTMIN, &sig\_action, NULL) == -1) {

printf("Cannot set signal handler\n");

exit(1);

}

// Отправляем сигналы в порядке FIFO с помощью kill

for (int i = 0; i < SIG\_COUNT; i++) {

printf("Sending signal %d (%d)\n", signals[i], i+1);

kill(getpid(), signals[i]);

usleep(1000); // задержка, чтобы следующий сигнал отправлялся после обработки предыдущего

}

// Проверка порядка обработки сигналов

if (counter == SIG\_COUNT) {

printf("Signals processed in FIFO order\n");

} else {

printf("Signals are not processed in FIFO order\n");

}

return 0;

}

*Запустим программу l7.c:*

defrozen@defrozen-ubu:~$ gcc l7.c && ./a.out

Sending signal 34 (0)

Signal 34 received

Signal 0 processed

Sending signal 34 (1)

Signal 34 received

Signal 1 processed

Sending signal 34 (2)

Signal 34 received

Signal 2 processed

Sending signal 34 (3)

Signal 34 received

Signal 3 processed

Signals processed in FIFO order

Таким образом, доказан тот факт, что сигналы реального времени, имеющие один и тот же приоритет в очереди идут в порядке FIFO. Для этого выводим порядковый номер сигнала перед отправкой и после получения.

Подводя итог, можно сделать вывод о том, что сигналы - самое простое средство IPC, являются достаточно медленными и ресурсоёмкими, не позволяют передавать произвольные данные, служат главным образом для уведомления, обработки нештатных ситуаций и синхронизации.

5.2) Продемонстрируем на примере, в чем разница надежных сигналов от ненадежных:

*Код программы check.c*:

#include <sys/wait.h>

#include<stdio.h>

#include<signal.h>

#include<unistd.h>

void signal\_handler(int signum) {

printf("Received signal %d\n", signum);

}

int main() {

pid\_t pid;

int status;

pid = fork(); // создание дочернего процесса

if (pid < 0) { // проверка на ошибку

printf("Failed to create child process\n");

return 1;

}

else if (pid == 0) { // обработка в дочернем процессе

signal(SIGINT, signal\_handler);

signal(SIGRTMIN, signal\_handler);

printf("Child process is running...\n");

sleep(10); // дочерний процесс ждет 10 секунд

printf("Child process is exiting...\n");

}

else { // обработка в родительском процессе

sleep(2); // родительский процесс ждет 2 секунды

int a=0;

int b=0;

printf("Sending signal %d to the child process...\n", SIGINT);

while(a<5){

kill(pid, SIGINT); // отправка ненадежного сигнала SIGINT

a++;

}

printf("Sending signal %d to the child process...\n", SIGRTMIN);

while(b<5){

kill(pid, SIGRTMIN); // отправка надежного сигнала SIGRTMIN

b++;

}

wait(&status); // ожидание завершения дочернего процесса

printf("Parent process is exiting...\n");

}

return 0;

}

*Результат работы программы:*

defrozen@defrozen-ubu:~$ gcc check.c && ./a.out

Child process is running...

Sending signal 2 to the child process...

Sending signal 34 to the child process...

Received signal 34

Received signal 34

Received signal 34

Received signal 34

Received signal 34

Received signal 2

Child process is exiting...

Parent process is exiting...

Таким образом видим, что так как сигналы присылались практически одновременно, не все ненадежные сигналы были доставлены. Однако все надежные сигналы были доставлены, что наглядно показывает различия данных типов сигналов.

6) Организуем 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.

*Код программы l8.c*:

#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]; // pipefd[0] указывает на конец канала для чтения. pipefd[1] указывает на конец канала для записи.

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;

}

*Запустим программу l8.c:*

*defrozen@defrozen-ubu:~$ gcc l8.c && ./a.out*

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

pipe is successfully created

Read from pipe: firs string

second //содержимое файла from.txt

third

That's all

*Теперь откроем файл to.txt:*

defrozen@defrozen-ubu:~$ cat to.txt

firs string

second

third

That's all

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

Данная программа реализует механизм обмена данными между двумя процессами через неименованный канал (pipe) в операционной системе Linux. программа создает неименованный канал через системный вызов pipe и создает дочерний процесс с помощью функции fork().  
В дочернем процессе (процесс-сын) происходит чтение данных из файла с именем, указанным в качестве первого аргумента программы, и запись их в пайп.  
В родительском процессе (процесс-отец) происходит чтение данных из пайпа и запись их в файл с именем, указанным в качестве второго аргумента программы.  
Вся передача данных происходит блокирующим образом (read и write вызовы), то есть процесс-отец ждет пока в пайпе появятся данные, которые передал процесс-сын, и только после этого их считывает и записывает в файл.

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

*Код программы server7.c*:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <errno.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 && errno!=EEXIST) {

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

exit(1);

}

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

if (res && errno!=EEXIST) {

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

exit(1);

}

// открываем первый канал для записи

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

if (chan1 == -1) {

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

exit(0);

}

// открываем второй канал для чтения

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

if (chan2 == -1) {

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

exit(0);

}

// пишем имя файла в первый канал

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

// читаем содержимое файла из второго канала

char buf[100];

for (;;) {

bzero(buf, 100);

res = read(chan2, buf, 100);

if (res <= 0)

break;

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

}

close(chan1);

close(chan2);

unlink("channel1");

unlink("channel2");

return 0;

}

*Код программы client7.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:*

defrozen@defrozen-ubu:~$ gcc server7.c && ./a.out

Using default file name 'testFile.txt'

Part of file: first string

second

Last string

defrozen@defrozen-ubu:~$ gcc client7.c && ./a.out

*В результате выполнения программы содержимое файла testFile.txt:*

first string

second

Last string

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 ()

8) Изменим ранее использованную программу так, чтобы сервер, перед тем как читать данные из канала, ожидал ввода пользователя. Исходный код клиента оставим неизменным.

*Исполняемый код l8.c:*

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#define DEF\_FILENAME "testFile.txt"

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

char fileName[30];

if (argc < 2) {

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

strcpy(fileName, DEF\_FILENAME);

} else

strcpy(fileName, argv[1]);

// создаем два канала

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

if (res) {

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

exit(1);

}

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

if (res) {

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

exit(1);

}

// открываем первый канал для записи

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

if (chan1 == -1) {

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

exit(0);

}

// открываем второй канал для чтения

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

if (chan2 == -1) {

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

exit(0);

}

// пишем имя файла в первый канал

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

// читаем содержимое файла из второго канала

char buf[100];

printf("Waiting for client write to channel\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*

defrozen@defrozen-ubu:~$ gcc server8.c && ./a.out

Using default file name 'testFile.txt'

Waiting for client write to channel

defrozen@defrozen-ubu:~$ gcc client7.c && ./a.out

defrozen@defrozen-ubu:~$ ls -sl | grep chan

0 prw-rw-r-- 1 dasha dasha 0 мая 2 14:41 channel1

0 prw-rw-r-- 1 dasha dasha 0 мая 2 14:41 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:*

defrozen@defrozen-ubu:~$ gcc server8.c && ./a.out

Using default file name 'testFile.txt'

Waiting for client write to channel

Part of file: first string

second

Last string

Server finished

Таким образом, все данные были получены, после того, как пользователь ввел что-либо в терминал, в нашем случае enter.

Подводя итог, можно сделать вывод о том, что неименованные каналы предназначены только для взаимодействия между родственными процессами, а именованные каналы могут использоваться для взаимодействия между любыми процессами. Именованные и неименованные каналы реализуют запись и чтение по принципу FIFO. Запись и чтение, таким образом, происходит быстро, однако при создании именованного канала затрачивается несколько больше времени.

9) Создать клиент-серверное приложение, демонстрирующее передачу информации между процессами посредством очередей 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).

*Исполняемый код server9.c:*

#include <stdio.h>

#include <stdlib.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <sys/types.h>

#include <signal.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) { // удаление очереди сообщений

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); //создаем новую очередь сообщений, возвращает идентификатор очереди сообщений

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); //записываем сообщение из очереди в буфер msgbuf удаляя сообщение из очереди

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); //отправили сообщение

if(res != 0) {

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

kill(getpid(),SIGINT);

}

}

return 0;

}

*Исполняемый код client9.c:*

#include <stdio.h>

#include <stdlib.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <sys/types.h>

#include <signal.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); //получаем доступ к очереди

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;

}

*Запустим два процесса client и серверный процесс c разных терминалов.*

*Сервер:*

defrozen@defrozen-ubu:~$ gcc server9.c && ./a.out

Using default key file key

Client's request: first client

Client's request: second client

*Первый клиент:*

defrozen@defrozen-ubu:~$ gcc client9.c && ./a.out

Using default key file key

first client

Server's response: OK

*Второй клиент:*

defrozen@defrozen-ubu:~$ gcc client9.c && ./a.out

Using default key file key

second client

Server's response: OK

Введем следующую команду и определим максимальный размер сообщения, а также максимальное количество сообщений в очереди:

defrozen@defrozen-ubu:~$ ipcs -l

------ Лимиты сообщений --------

максимум очередей для всей системы = 32000

максимальный размер сообщения (байты) = 8192

максимальный по умолчанию размер сообщения (байты) = 16384

Подводя итог, сделаем вывод о том, что сообщения являются мощным средством межпроцессного обмена данными. Время доставки сообщения сравнимо с временем доставки сигнала, однако сообщение несёт гораздо больше информации, чем сигнал.

10) Использование семафоров.

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.

*Исполняемый код server10.c:*

#include <stdio.h>

#include <stdlib.h>

#include <signal.h>

#include <string.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"

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 – разделяемую память, возвращает идентификатор общего сегмента памяти

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, чтобы все корректно завершилось

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);

}// возвращает идентификатор набора семафоров

// 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);

}

}

}

*Исполняемый код client10.c:*

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

#include <sys/shm.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};

*Запустим программу server10.c:*

defrozen@defrozen-ubu:~$ gcc server10.c && ./a.out

Using default key file key

Client's message: hi

Client's message: first

Client's message: client

Client's message: hey

Client's message: second

Client's message: client

^Cexecution complete

*Запустим программу client10.c в одном терминале:*

defrozen@defrozen-ubu:~$ gcc client10.c && ./a.out

Using default key file key

Type message to serever. Empty string to finish

hi

Type message to serever. Empty string to finish

first

Type message to serever. Empty string to finish

client

Type message to serever. Empty string to finish

bye-bye

*Запустим программу client10.c в другом терминале:*

defrozen@defrozen-ubu:~$ gcc client10.c && ./a.out

Using default key file key

Type message to serever. Empty string to finish

hey

Type message to serever. Empty string to finish

second

Type message to serever. Empty string to finish

client

Type message to serever. Empty string to finish

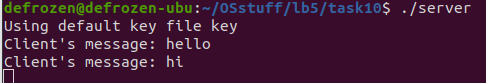
bye-bye

Таким образом были использованы два семафора. Один из них имел смысл - запись разрешена», т.е. читатель предыдущие данные уже использовал; второй – «чтение разрешено», т.е. писатель уже сгенерировал новые данные, которые нужно прочитать. 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.

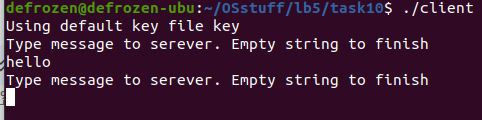
В результате все сообщения от клиента сервером прочитаны.

11) 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.

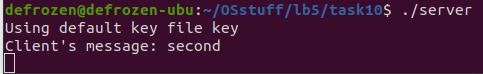
*Запустим несколько клиентов и серверов, выясним, как происходит их работа.*



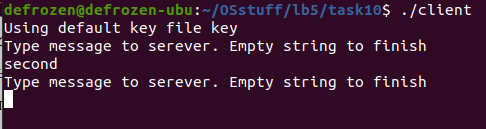
*Первый сервер*



*Первый клиент*



*Второй сервер*



*Второй клиент*

Таким образом, можно заметить, что каждое новое обращение клиента(в каком порядке какой клиент работает не важно) будет приниматься первым и вторым сервером поочередно.

12) К условиям предыдущей задачи добавляется наличие не единичного буфера, а буфера некоторого размера. Тип буфера (очередь, стек, кольцевой буфер) не имеет значения.В качестве разделяемого ресурса используется массив, находящийся в разделяемой памяти.

*Исполняемый код server12.c:*

#include <stdio.h>

#include <stdlib.h>

#include <signal.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

#include <sys/shm.h>

#include <sys/time.h>

#include "shm.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)\*sizeof(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);

}

// устанавливаемобработчиксигнала

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 инекс в -1, //первый записывающий клиентустановит 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);

// освобождаемпамять

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

printf("execution complete\n");

kill(getpid(),SIGINT);

}

// увеличиваемчислопустыхячеек

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

printf("execution complete\n");

kill(getpid(),SIGINT);

}

}

}

*Исполняемый код client12.c:*

#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"

int\* buf;

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;

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)\*sizeof(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);

}

*Запустим программу server12.c:*

*defrozen@defrozen-ubu:~$ gcc server12.c && ./a.out*

UUsing 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

*Запустим программу client12.c:*

defrozen@defrozen-ubu:~$ gcc client12.c && ./a.out

Using default key file key

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

*Запустим еще одну программу client12.c:*

defrozen@defrozen-ubu:~$ gcc client12.c && ./a.out

Using default key file key

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

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

Таким образом, сделаем вывод о том, что семафоры и разделяемая память зачастую используются вместе. Семафоры позволяют синхронизировать доступ к разделяемому ресурсу и гарантировать «взаимное исключение» нескольких процессов при разделении ресурса (пока предыдущий процесс не закончит работу с ресурсом, следующий не начнет ее).

13) Сервер прослушивает заданный порт, при запросе нового соединения, создается новый поток для его обработки. Работа с клиентом должна быть 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 серверу, и получение работы. Если была введена пустая строка, клиент завершается.

*Исполняемый код server13.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>

#include <pthread.h>

#define DEF\_PORT 8888

#define DEF\_IP "127.0.0.1"

int readFix(int sock, char\* buf, int bufSize, int flags);

int sendFix(int sock, char\* buf, int flags);

void\* clientHandler(void\* args);

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;

//информация о сервере, которая будет записана в сокет

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);//создание сокета

if (listener < 0) {

perror("Can't create socket to listen: ");

exit(1);

}

int res = bind(listener, (struct sockaddr \*)&listenerInfo, sizeof(listenerInfo));//привязка сокета к адресу

if (res < 0) {

perror("Can't bind socket");

exit(1);

}

res = listen(listener, 5);//ожидание клиента на сокете, второй арг задает максимальную длину, до которой может расти очередь ожидающих соединений., установим сокет в режим прослушивания

if (res) {

perror("Error while listening:");

exit(1);

}

for (;;) {//ожидает новых клиентов, создает для каждого поток, передает ему дескриптор клиентского сокета

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;

}

void\* clientHandler(void\* args) {//

int sock = (int)args;

char buf[100];

int res = 0;

for (;;) {//ждет поступление данных от клиента, выводит данные на экран и отправляет их обратно клиенту

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 readFix(int sock, char\* buf, int bufSize, int flags) {//возвращает количество считанных байт, считывает байты из сокета, первые 4 – размер сообщения, затем сами данные

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) {//отправляет данные с указанного буфера в сокет, перед эти записав размер отправляемых данных

unsigned msgLength = strlen(buf);

int res = send(sock, &msgLength, sizeof(unsigned), flags);

if (res <= 0)

return res;

send(sock, buf, msgLength, flags);

}

*Исполняемый код client13.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 readFix(int sock, char\* buf, int bufSize, int flags);

int sendFix(int sock, char\* buf, int flags);

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, инициирует соединение на сокете

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 (;;) {//запрашивает у пользователя сообщения для отправки на сервер. С помощью функции sendFix() они отправляются на сервер.

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);

// После отправки клиент ждет ответа от сервера с помощью функции readFix(), которая сначала читает заголовок - количество байт в сообщении, которое ожидается, а затем читает само сообщение.

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("Received more data, than 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) {//отправим сообщение на сервер

// 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);

}

*Запустим программу server13.c:*

defrozen@defrozen-ubu:~$ gcc server13.c && ./a.out

Using default port 8888

Some client sent: hi

Some client sent: this is client

Some client sent: bye

Can't recv data from client, ending thread

: Success

^C

*Запустим программу client13.c:*

defrozen@defrozen-ubu:~$ gcc client13.c && ./a.out

Using default port 8888

Using default addr 127.0.0.1

Input request (empty to exit)

hi

Server's response: hi

Input request (empty to exit)

this is client

Server's response: this is client

Input request (empty to exit)

bye

Server's response: bye

Input request (empty to exit)

Bye-bye

Таким образом, все сообщения были доставлены.

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 readFix и sendFix. Принцип их работы следующий: функция sendFix перед посылкой собственно данных посылает «заголовок» - количество байт в посылке. Функция readFix 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.

Приведем пример использования программы в локальной сети.

*СЕРВЕР:*

defrozen@defrozen-ubu:~$ gcc server13.c && ./a.out

Using default port 8888

Some client sent: hey

Some client sent: how are you

Some client sent: bye

Can't recv data from client, ending thread

: Success

*КЛИЕНТ:*

defrozen@defrozen-ubu:~$ gcc client13.c && ./a.out 10.0.2.15

Using default port 8888

Input request (empty to exit)

hey

Server's response: hey

Input request (empty to exit)

how are you

Server's response: how are you

Input request (empty to exit)

bye

Server's response: bye

Input request (empty to exit)

Bye-bye

Таким образом, сокеты являются средством IPC, которое можно использовать не только между процессами на одном компьютере, но и в сетевом режиме. Проверим это, для этого запустим программу-сервер на одной виртуальной машине, а программу-клиент на другой.

*КЛИЕНТ:*

defrozen@defrozen-ubu:~$ gcc client13.c && ./a.out 192.168.56.101

Using default port 8888

Input request (empty to exit)

hello

Server's response: hello

Input request (empty to exit)

*СЕРВЕР:*

defrozen@defrozen-ubu:~$ gcc server13.c && ./a.out

Using default port 8888

Some client sent: hello

Таким образом, удалось передать сообщение даже в сетевом режиме.

14) Модифицировать, если необходимо, предложенное приложение и реализовать обмен сервера с множеством клиентов. Количество клиентов: 10, 100, 1000. Выполнить аналогичное взаимодействие на основе UDP, сравнить с IPC очереди сообщений.

Код программ клиента и сервера оставим неизменными, однако дополнительно напишем скрипт, который будет генерировать 300 клиентов.

*Исполняемый код script14:*

#!/bin/bash

# loop through 10 times

for i in {1..300}

do

echo -e "hi $i \n" | ./client &

done

*Запустим программу server14.:*

defrozen@defrozen-ubu:~$ gcc server14.c && ./a.out

Using default port 8887

Some client sent: hi 1

Can't recv data from client, ending thread

: Success

Some client sent: hi 101

Some client sent: hi 4

Some client sent: hi 3

Some client sent: hi 10

Some client sent: hi 2

Can't recv data from client, ending thread

: Success

Some client sent: hi 11

Some client sent: hi 113

Some client sent: hi 91

Can't recv data from client, ending thread

*…*

*Запустим скрипт:*

defrozen@defrozen-ubu:~$ ./script14

Using default port 8887

Using default addr 127.0.0.1

Input request (empty to exit)

Server's response: hi 1

Input request (empty to exit)

Bye-bye

Using default port 8887

Using default addr 127.0.0.1

Input request (empty to exit)

Server's response: hi 3

Input request (empty to exit)

Bye-bye

Using default port 8887

Using default addr 127.0.0.1

Input request (empty to exit)

Using default port 8887

Using default addr 127.0.0.1

Input request (empty to exit)

Server's response: hi 12

Input request (empty to exit)

Bye-bye

Using default port 8887

Using default addr 127.0.0.1

Input request (empty to exit)

Server's response: hi 15

Input request (empty to exit)

…….

Error while receiving:: Connection reset by peer

Error while receiving:: Connection reset by peer

Error while receiving:: Connection reset by peer

….

Таким образом, параметр 5 задает максимальный размер очереди ожидания. Если количество запросов на подключение в очереди достигнет этого значения, то любые последующие запросы будут отклонены, до тех пор, пока не освободится хотя бы один сокет. Однако все запросы будут гарантированно обработаны.

**Выводы:**

Подводя итог, можно сделать вывод о том, что в ОС Unix адресные пространства различных процессов изолированы друг от друга. Для взаимодействия процессов используются специальные средства IPC, включающие в себя сигналы, именованные и неименованные каналы, сообщения, сокеты, семафоры и разделяемую память.

**Список использованных источников:**

* <https://elib.spbstu.ru/dl/2/s17-72.pdf/en/view>
* <https://radioprog.ru/post/1411>
* <https://www.opennet.ru/docs/RUS/os_unix/glava_23.html>
* <https://russianblogs.com/article/8398866633/>
* <https://ru.manpages.org/signal/7>