**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №5-6**

**по дисциплине «Операционные системы»**

**Тема**: IPC

|  |  |  |
| --- | --- | --- |
| Студент гр. 1384 |  | Усачева Д.В. |
| Преподаватель |  | Душутина Е. В. |

Санкт-Петербург

2023

## Цель работы.

Изучить порождение и запуск процессов, взаимодействие родственных процессов и управление процессами посредством сигналов.

**Выполнение работы.**

Задание 1.

Создана программа, позволяющая 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.

Листинг 1 – Код программы 1.c (взят из методических указаний):

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

#include <stdlib.h>

#include <sys/wait.h>

static void sigHandler(int sig) {

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

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

// Restore the old disposition

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) {

// Child process

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

// Send signals to parent

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

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

exit(1);

}

printf("Successfully sent SIGUSR1\n");

return 0;

}

// Parent process

wait(NULL); // Wait for child to finish

for (;;) {

pause(); // Wait for signals

}

return 0;

}

./1

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

Листинг 2 – Вывод программы:

./1 &

[1] 267

Father started: pid = 267, ppid = 249

Son started: pid = 268, ppid = 267

Successfully sent SIGUSR1

Caught signal SIGUSR1

Parent = 249

sudo kill -SIGCHLD 267

dari@DESKTOP-7CDTIKD:~/4\_sem/os/5/1$ sudo kill -SIGUSR2 267

Caught signal SIGUSR2

Parent = 249

dari@DESKTOP-7CDTIKD:~/4\_sem/os/5/1$ sudo kill -SIGINT 267

[1]+ Interrupt

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

Листинг 3 – Вывод программы:

Father started: pid = 275, ppid = 249

Son started: pid = 276, ppid = 275

Successfully sent SIGUSR1

Caught signal SIGUSR1

Parent = 249

sudo kill -SIGUSR2 275

Caught signal SIGUSR2

Parent = 249

dari@DESKTOP-7CDTIKD:~/4\_sem/os/5/1$ sudo kill -SIGUSR2 275

[1]+ User defined signal 2 ./1

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.

Теперь проведем эксперимент для нескольких процессов, первый из них отправляет пользовательский сигнал SIGUSR1, второй отправляет SIGUSR2, третий SIGCHLD.

Листинг 4 – Код программы father.c:

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

#include <stdlib.h>

#include <sys/wait.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(SIGCHLD, SIG\_IGN);

signal(SIGUSR2, sigHandler);

int pid1, pid2, pid3;

if((pid1 = fork()) == 0) {

execl("son2", "son2", NULL);

}

if((pid2 = fork()) == 0) {

execl("son1", "son1", NULL);

}

if((pid3 = fork()) == 0) {

execl("son3", "son3", NULL);

}

wait(NULL);

for(;;){

pause();

}

return 0;

}

Листинг 5 – Код программы son1.c:

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

#include <stdlib.h>

int main(){

printf("\nSon1 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;

}

Листинг 6 –– Код программы son2.c:

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

#include <stdlib.h>

int main(){

printf("\nSon2 started: pid = %i,ppid = %i\n",getpid(),getppid());

// отправляем сигналы родителю

if(kill(getppid(),SIGUSR2) != 0) {

printf("Error while sending SIGUSR2\n");

exit(1);

}

printf("Successfully sent SIGUSR2\n");

return 0;

}

Листинг 7– Код программы son3.c:

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

#include <stdlib.h>

int main(){

printf("\nSon3 started: pid = %i,ppid = %i\n",getpid(),getppid());

// отправляем сигналы родителю

if(kill(getppid(),SIGCHLD) != 0) {

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

exit(1);

}

printf("Successfully sent SIGCHLD\n");

return 0;

}

Листинг 8 – Результат выполнения программы:

sudo ./father &

[2] 703

dari@DESKTOP-7CDTIKD:~/4\_sem/os/5/1$

Father started: pid = 704,ppid = 703

Son1 started: pid = 706,ppid = 704

Successfully sent SIGUSR1

Son2 started: pid = 705,ppid = 704

Successfully sent SIGUSR2

Catched signal SIGUSR1

Catched signal SIGUSR2

Parent = 703

Parent = 703

l SIGUSR1

Parent = 703

Son3 started: pid = 707,ppid = 704

Successfully sent SIGCHLD

sudo kill -SIGUSR1 704

[2]- User defined signal 1 sudo ./father

Теперь для эксперимента напишем программу, в которой 2 потока отлавливают пользовательские сигналы.

Листинг 9 – Код программы threads.c:

#include <pthread.h>

#include <signal.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/syscall.h>

void\* thread1\_function(void\* arg);

void\* thread2\_function(void\* arg);

int main()

{

sigset\_t signal\_set;

sigemptyset(&signal\_set);

sigaddset(&signal\_set, SIGUSR1);

sigaddset(&signal\_set, SIGUSR2);

pthread\_sigmask(SIG\_BLOCK, &signal\_set, NULL);

// Создание потоков

pthread\_t thread1, thread2;

pthread\_create(&thread1, NULL, thread1\_function, &signal\_set);

pthread\_create(&thread2, NULL, thread2\_function, &signal\_set);

// Ожидание завершения потоков

pthread\_join(thread1, NULL);

pthread\_join(thread2, NULL);

return 0;

}

void\* thread1\_function(void\* arg)

{

int pid = getpid();

int tid = syscall(SYS\_gettid);

printf("Thread\_1 is started: tid = %d pid = %d\n", tid, pid);

int signal\_number;

while(1) {

sigwait(&arg, &signal\_number);

if (signal\_number == SIGUSR1) {

printf("Caught SIGUSR1 signal in Thread 1!\n");

} else if (signal\_number == SIGUSR2) {

printf("Caught SIGUSR2 signal in Thread 1!\n");

}

}

return NULL;

}

void\* thread2\_function(void\* arg)

{

int pid = getpid();

int tid = syscall(SYS\_gettid);

printf("Thread\_2 is started: tid = %d pid = %d\n", tid, pid);

int signal\_number;

while(1) {

sigwait(&arg, &signal\_number);

if (signal\_number == SIGUSR1) {

printf("Caught SIGUSR1 signal in Thread 2!\n");

} else if (signal\_number == SIGUSR2) {

printf("Caught SIGUSR2 signal in Thread 2!\n");

}

}

return NULL;

}

Листинг 10 – Результат выполнения программы:

sudo ./threads &

[2] 917

dari@DESKTOP-7CDTIKD:~/4\_sem/os/5/1/1\_1$ Thread\_1 is started: tid = 919 pid = 918

Thread\_2 is started: tid = 920 pid = 918

sudo kill -SIGUSR2 920

Caught SIGUSR2 signal in Thread 2!

dari@DESKTOP-7CDTIKD:~/4\_sem/os/5/1/1\_1$ sudo kill -SIGUSR2 919

Caught SIGUSR2 signal in Thread 1!

dari@DESKTOP-7CDTIKD:~/4\_sem/os/5/1/1\_1$ sudo kill -SIGUSR2 920

Caught SIGUSR2 signal in Thread 2!

dari@DESKTOP-7CDTIKD:~/4\_sem/os/5/1/1\_1$ sudo kill -SIGUSR2 919

Caught SIGUSR2 signal in Thread 1!

Теперь проведем эксперимент, в котором каждый порожденный процесс имеет два потока, каждый из которых отлавливает пользовательские сигналы.

Листинг 11 – Код программы father.c:

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

#include <stdlib.h>

#include <sys/wait.h>

int main() {

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

int pid1, pid2, pid3;

if((pid1 = fork()) == 0) {

execl("son2", "son2", NULL);

}

if((pid2 = fork()) == 0) {

execl("son1", "son1", NULL);

}

wait(NULL);

for(;;){

pause();

}

return 0;

}

Листинг 12 – Код программы son1.c:

#include <pthread.h>

#include <signal.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/syscall.h>

void\* thread1\_function(void\* arg);

void\* thread2\_function(void\* arg);

void signal\_handler(int signal\_number)

{

// Обработка сигналов в зависимости от их типа

switch (signal\_number) {

case SIGUSR1:

printf("Caught SIGUSR1 signal in Son 1!\n");

break;

case SIGUSR2:

printf("Caught SIGUSR2 signal in Son 1!\n");

break;

default:

printf("Caught unknown signal in Son 1!\n");

break;

}

}

int main()

{

printf("\nSon1 started: pid = %i,ppid = %i\n",getpid(),getppid());

signal(SIGUSR1, signal\_handler);

signal(SIGUSR2, signal\_handler);

pthread\_t thread1, thread2;

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

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

// Ожидание завершения потоков

pthread\_join(thread1, NULL);

pthread\_join(thread2, NULL);

return 0;

}

void\* thread1\_function(void\* arg)

{

int pid = getpid();

int tid = syscall(SYS\_gettid);

sleep(2);

printf("Son 1 Thread\_1 is started: tid = %d pid = %d\n", tid, pid);

int signal\_number;

while(1);

return NULL;

}

void\* thread2\_function(void\* arg)

{

int pid = getpid();

int tid = syscall(SYS\_gettid);

sleep(2);

printf("Son 1 Thread\_2 is started: tid = %d pid = %d\n", tid, pid);

while(1);

return NULL;

}

Листинг 13 – Код программы son2.c:

#include <pthread.h>

#include <signal.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/syscall.h>

void\* thread1\_function(void\* arg);

void\* thread2\_function(void\* arg);

void signal\_handler(int signal\_number)

{

// Обработка сигналов в зависимости от их типа

switch (signal\_number) {

case SIGUSR1:

printf("Caught SIGUSR1 signal in Son 2!\n");

break;

case SIGUSR2:

printf("Caught SIGUSR2 signal in Son 2!\n");

break;

default:

printf("Caught unknown signal in Son 2!\n");

break;

}

}

int main()

{

printf("\nSon2 started: pid = %i,ppid = %i\n",getpid(),getppid());

signal(SIGUSR1, signal\_handler);

signal(SIGUSR2, signal\_handler);

pthread\_t thread1, thread2;

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

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

// Ожидание завершения потоков

pthread\_join(thread1, NULL);

pthread\_join(thread2, NULL);

return 0;

}

void\* thread1\_function(void\* arg)

{

int pid = getpid();

int tid = syscall(SYS\_gettid);

sleep(2);

printf("Son 2 Thread\_1 is started: tid = %d pid = %d\n", tid, pid);

int signal\_number;

while(1);

return NULL;

}

void\* thread2\_function(void\* arg)

{

int pid = getpid();

int tid = syscall(SYS\_gettid);

sleep(2);

printf("Son 2 Thread\_2 is started: tid = %d pid = %d\n", tid, pid);

while(1);

return NULL;

}

Листинг 14 – Результат выполнения программы:

sudo ./father &

[2] 1102

dari@DESKTOP-7CDTIKD:~/4\_sem/os/5/1/1\_1/2$

Father started: pid = 1103,ppid = 1102

Son2 started: pid = 1104,ppid = 1103

Son1 started: pid = 1105,ppid = 1103

Son 2 Thread\_2 is started: tid = 1107 pid = 1104

Son 2 Thread\_1 is started: tid = 1106 pid = 1104

Son 1 Thread\_2 is started: tid = 1109 pid = 1105

Son 1 Thread\_1 is started: tid = 1108 pid = 1105

sudo kill -SIGUSR1 1107

Caught SIGUSR1 signal in Son 2!

dari@DESKTOP-7CDTIKD:~/4\_sem/os/5/1/1\_1/2$ sudo kill -SIGUSR1 1105

Caught SIGUSR1 signal in Son 1!

dari@DESKTOP-7CDTIKD:~/4\_sem/os/5/1/1\_1/2$ sudo kill -SIGUSR1 1104

Caught SIGUSR1 signal in Son 2!

dari@DESKTOP-7CDTIKD:~/4\_sem/os/5/1/1\_1/2$ sudo kill -SIGUSR1 1106

Caught SIGUSR1 signal in Son 2!

Создана программа, позволяющая 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.

Листинг 15 – Код программы 2.c (взят из методических указаний):

#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;

}

Листинг 16 – Результат выполнения программы:

dari@DESKTOP-7CDTIKD:~/4\_sem/os/5/1$ ./2 &

[1] 1447

dari@DESKTOP-7CDTIKD:~/4\_sem/os/5/1$ sudo kill -SIGUSR1 1447

SIGUSR1 catched

dari@DESKTOP-7CDTIKD:~/4\_sem/os/5/1$ sudo kill -SIGINT 1447

dari@DESKTOP-7CDTIKD:~/4\_sem/os/5/1$ jobs -l

[1]+ 1447 Running ./2 &

dari@DESKTOP-7CDTIKD:~/4\_sem/os/5/1$ jobs -l

[1]+ 1447 Interrupt ./2

dari@DESKTOP-7CDTIKD:~/4\_sem/os/5/1$ jobs –l

Данный эксперимент демонстрирует преимущество надёжных сигналов – возможность отложенной обработки.

Был проведен эксперимент, позволяющий определить возможность организации очереди для различных типов сигналов, обычных и реального времени, (более двух сигналов, для этого увеличьте «вложенность» вызовов обработчиков);

Листинг 17 – Код программы 3.c:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <signal.h>

#define NUM\_SIGNALS 4

void handle\_signal(int signo, siginfo\_t \*info, void \*context) {

printf("Received signal %d with value %d\n", signo, info->si\_value.sival\_int);

}

int main() {

int i;

struct sigaction sa;

union sigval value;

// Set up signal handler

sa.sa\_sigaction = handle\_signal;

sa.sa\_flags = SA\_SIGINFO;

sigemptyset(&sa.sa\_mask);

for (i = SIGRTMIN; i <= SIGRTMAX; i++) {

sigaddset(&sa.sa\_mask, i);

}

for (i = 1; i <= NUM\_SIGNALS; i++) {

sigaction(SIGINT + i, &sa, NULL);

sigaction(SIGRTMIN + i, &sa, NULL);

}

// Send signals

for (i = 1; i <= NUM\_SIGNALS; i++) {

value.sival\_int = i;

printf("Sending signal %d with value %d\n", SIGINT + i,value.sival\_int);

if (sigqueue(getpid(), SIGINT + i, value) != 0) {

perror("sigqueue");

exit(EXIT\_FAILURE);

}

printf("Sending signal %d with value %d\n", SIGRTMIN + i,value.sival\_int);

if (sigqueue(getpid(), SIGRTMIN + i, value) != 0) {

perror("sigqueue");

exit(EXIT\_FAILURE);

}

}

// Wait for signals to be processed

sleep(1);

return 0;

}

Листинг 18 – Результат выполнения программы:

./3

Sending signal 3 with value 1

Received signal 3 with value 1

Sending signal 35 with value 1

Received signal 35 with value 1

Sending signal 4 with value 2

Received signal 4 with value 2

Sending signal 36 with value 2

Received signal 36 with value 2

Sending signal 5 with value 3

Received signal 5 with value 3

Sending signal 37 with value 3

Received signal 37 with value 3

Sending signal 6 with value 4

Received signal 6 with value 4

Sending signal 38 with value 4

Received signal 38 with value 4

Проведенный эксперимент включает отправку в общей сложности 8 сигналов (4 обычных сигнала и 4 сигнала реального времени) одному и тому же процессу с разными значениями, присвоенными каждому сигналу.

Обработчик этих сигналов просто выводит номер сигнала и присоединенное значение. Ожидается, что сигналы будут обрабатываться в порядке FIFO, независимо от того, являются ли они обычными сигналами или сигналами реального времени.

Это связано с тем, что сигналы отправляются в один и тот же процесс и должны ставиться в очередь в порядке их получения. Эксперимент подтверждает это ожидание. Это демонстрирует, что сигналы разных типов могут ставиться в очередь и обрабатываться в согласованном и предсказуемом порядке, если они отправляются в один и тот же процесс.

Подтвердим наличие приоритетов сигналов реального времени:

Листинг 18.1– Код программы 3\_3.c:

#include <stdio.h>

#include <stdlib.h>

#include <signal.h>

#include <unistd.h>

#define NUM\_SIGNALS 5

void handle\_rt\_signal(int sig, siginfo\_t\* info, void\* context)

{

printf("Handling real-time signal %d\n", sig);

usleep(100000);

}

void handle\_normal\_signal(int sig)

{

printf("Handling normal signal %d\n", sig);

if (sig == SIGINT)

{

signal(sig, SIG\_DFL);

}

usleep(100000);

}

int main()

{

printf("Father pid = %d ppid = %d\n", getpid(), getppid());

struct sigaction sa\_rt;

sa\_rt.sa\_sigaction = handle\_rt\_signal;

sa\_rt.sa\_flags = SA\_SIGINFO;

sigemptyset(&sa\_rt.sa\_mask);

for (int i = SIGRTMIN; i <= SIGRTMAX; i++)

{

sigaction(i, &sa\_rt, NULL);

}

// Register normal signals

for (int i = 1; i <= NUM\_SIGNALS; i++)

{

signal(i, handle\_normal\_signal);

}

// Send signals

if (fork() == 0)

{

printf("Son pid = %d ppid = %d\n", getpid(), getppid());

printf("Son sending signals to father...\n");

for (int i = 1; i <= NUM\_SIGNALS; i++)

{

printf("Sending normal signal %d\n", i);

kill(getppid(), i);

}

for (int i = SIGRTMAX; i >= SIGRTMIN; --i)

{

printf("Sending real-time signal %d\n", i);

kill(getppid(), i);

}

printf("Signals sent. Sleeping for 5 seconds to allow handling...\

n");

sleep(3);

return 0;

}

for (;;)

{

pause();

}

return 0;

}

Листинг 18.2 – Результат выполнения программы:

./3\_3

Father pid = 6321 ppid = 6297

Son pid = 6322 ppid = 6321

Son sending signals to father...

Sending normal signal 1

Sending normal signal 2

Sending normal signal 3

Sending normal signal 4

Sending normal signal 5

Sending real-time signal 64

Sending real-time signal 63

Sending real-time signal 62

Sending real-time signal 61

Sending real-time signal 60

Sending real-time signal 59

Handling real-time signal 64

Sending real-time signal 58

Handling real-time signal 59

Sending real-time signal 57

Handling real-time signal 58

Sending real-time signal 56

Handling real-time signal 57

Sending real-time signal 55

Handling real-time signal 56

Sending real-time signal 54

Sending real-time signal 53

Handling real-time signal 54

Sending real-time signal 52

Sending real-time signal 51

Sending real-time signal 50

Sending real-time signal 49

Sending real-time signal 48

Sending real-time signal 47

Handling real-time signal 54

Handling real-time signal 52

Sending real-time signal 46

Sending real-time signal 45

Handling real-time signal 54

Handling real-time signal 52

Handling real-time signal 46

Sending real-time signal 44

Handling real-time signal 54

Handling real-time signal 52

Handling real-time signal 46

Handling real-time signal 45

Sending real-time signal 43

Handling real-time signal 44

Sending real-time signal 42

Handling real-time signal 44

Handling real-time signal 43

Sending real-time signal 41

Sending real-time signal 40

Handling real-time signal 44

Handling real-time signal 43

Handling real-time signal 41

Sending real-time signal 39

Sending real-time signal 38

Sending real-time signal 37

Sending real-time signal 36

Handling real-time signal 44

Handling real-time signal 43

Handling real-time signal 41

Handling real-time signal 39

Sending real-time signal 35

Handling real-time signal 44

Handling real-time signal 43

Handling real-time signal 41

Handling real-time signal 39

Handling real-time signal 36

Sending real-time signal 34

Handling real-time signal 44

Handling real-time signal 43

Handling real-time signal 41

Handling real-time signal 39

Handling real-time signal 36

Handling real-time signal 35

Handling real-time signal 44

Handling real-time signal 43

Handling real-time signal 41

Handling real-time signal 39

Handling real-time signal 36

Handling real-time signal 35

Handling real-time signal 34

Handling real-time signal 38

Handling real-time signal 37

Handling real-time signal 40

Handling real-time signal 42

Handling real-time signal 47

Handling real-time signal 51

Handling real-time signal 50

Handling real-time signal 49

Handling real-time signal 48

Handling real-time signal 53

Handling real-time signal 55

Handling real-time signal 63

Handling real-time signal 61

Handling real-time signal 60

Handling real-time signal 62

Handling normal signal 5

Handling normal signal 4

Handling normal signal 3

Handling normal signal 2

Signals sent. Sleeping for 5 seconds to allow handling...nHandling normal signal 1

Это подтверждает, что сигналы реального времени имеют более высокий приоритет, чем обычные сигналы, и что они обрабатываются в первую очередь, даже если отправляются позже.

Задание 2.

Различают два типа каналов анонимные (иначе их называют «программные» или «неименованные») и именованные. Они по-разному реализованы, но доступ к ним организуется одинаково с помощью обычных функций read и write (унифицированный подход по типу файловой модели). Одним из свойств программных каналов и FIFO является то, что данные по ним передаются в виде потоков байтов (аналогично соединению TCP). Деление этого потока на самостоятельные записи полностью предоставляется приложению (в отличие, например, от очередей сообщений, которые автоматически расставляют границы между записями, аналогично тому, как это делается в дейтаграммах UDP).

Программные (неименованные) каналы – однонаправленные, используются только для связи родственных процессов, в принципе могут использоваться и неродственными процессами, если предоставить им возможность передавать друг другу дескрипторы (т.к. имен они не имеют). Неименованный канал создается посредством системного вызова pipe(2), который возвращает 2 файловых дескриптора filedes[1] для записи в канал и filedes[0] для чтения из канала:

Организуем программу (файл 2.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 данные из файла и запись их в канал.

В текстовом файле from.txt находится: first string second third That's all. В итоге выполнения программы эта строка должна вывестись в файл и в терминал.

Код программы и результат выполнения представлен ниже.

Листинг 19 – Код программы 2.c (взят из методических указаний):

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.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

}

fclose(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;

}

Листинг 20 – Результат выполнения программы:

./2

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

pipe is successfully created

Read from pipe: first string

second

third

That's all

dari@DESKTOP-7CDTIKD:~/4\_sem/os/5/2$ cat to.txt

first string

second

third

That's all

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

Задание 3.

Именованные каналы в Unix функционируют подобно неименованным — они позволяют передавать данные только в одну сторону. Однако в отличие от неименованных каналов каждому каналу FIFO сопоставляется полное имя в файловой системе, что позволяет двум неродственным процессам обратиться к одному и тому же FIFO. Аббревиатура FIFO расшифровывается как «first in, first out» — «первым вошел, первым вышел», то есть эти каналы работают как очереди.

После создания канал FIFO должен быть открыт на чтение или запись с помощью либо функции open, либо одной из стандартных функций открытия файлов из библиотеки ввода-вывода (например, fopen). FIFO может быть открыт либо только на чтение, либо только на запись. Нельзя открывать канал на чтение и запись одновременно, поскольку именованные каналы могут быть только односторонними.

Необходимо 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.

Листинг 21 – Код программы server.c (взят из методических указаний):

#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", buf);

}

close(chan1);

close(chan2);

unlink("channel1");

unlink("channel2");

return 0;

}

Листинг 22 – Код программы client.c (взят из методических указаний):

#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 writing\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:

Листинг 23 – Результат выполнения программы:

./server

Using default file name 'testFile.txt'

Part of file: big fluffy cat

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

Листинг 24– Код программы server.c (взят из методических указаний):

#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", buf);

}

close(chan1);

close(chan2);

unlink("channel1");

unlink("channel2");

printf("Server finished\n");

return 0;

}

Листинг 25 – Результат выполнения программы:

./server2

Using default file name 'testFile.txt'

Waiting for clint write to channel

Part of file: big fluffy cat

Server finished

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.

Задание 4.

Необходимо создать клиент-серверное приложение, демонстрирующее передачу информации между процессами посредством очередей сообщений.

Листинг 26– Код программы server.c (взят из методических указаний):

#include <stdio.h>

#include <stdlib.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <sys/types.h>

#include <signal.h>

#include <unistd.h>

#include <strings.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;

}

Листинг 27– Код программы client.c (взят из методических указаний):

#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;

}

Листинг 28 – Результат выполнения программы:

./server

Using default key file key

Client's request: Hi!!!

Client's request: OMG! Hi

./client

Using default key file key

Hi!!!

Server's response: OK

./client

Using default key file key

OMG! Hi

Server's response: OK

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.

Задание 5.

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

Листинг 29 – Результат выполнения:

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

Рассмотрим несколько вариантов постановки задачи синхронизации доступа к разделяемой памяти.

Есть один процесс, выполняющий запись в разделяемую память и один процесс, выполняющий чтение из нее. Под чтением понимается извлечение данных из памяти. Программа должна обеспечить невозможность повторного чтения одних и тех же данных и невозможность перезаписи данных, т. е. новой записи, до тех пор, пока читатель не прочитает предыдущую. В таком варианте задания для синхронизации процессов достаточно двух семафоров.

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

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

Код программы-читателя и программы-писателя представлен ниже.

Листинг 30– Код программы client.c (взят из методических указаний):

#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 <strings.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);

}

}

Листинг 31– Код программы server.c (взят из методических указаний):

#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 <strings.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);

}

}

}

Листинг 32– Результат выполнения программы:

./server

Using default key file key

Client's message: t\_t

Client's message: :)))

./client

Using default key file key

Type message to serever. Empty string to finish

t\_t

Type message to serever. Empty string to finish

bye-bye

./client

Using default key file key

Type message to serever. Empty string to finish

:)))

Type message to serever. Empty string to finish

bye-bye

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.

Для выполнения данного задания был написан скрипт, который запускает определенное количество читателей и писателей:

Листинг 33 – Скрипт для варианта 2:

#!/bin/bash

# Количество читателей и писателей

num\_readers=2

num\_writers=4

# Запуск читателей

for ((i=0; i<$num\_readers; i++)); do

gnome-terminal -- ./server

done

# Запуск писателей

for ((i=0; i<$num\_writers; i++)); do

gnome-terminal -- ./client

done

Все читатели и писатели будут находиться в одном адресном пространстве, в итоге выполнения можно увидеть конкуренцию читателей за семафор.

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.

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. К примеру, в нулевую ячейку сначала первый поток-писатель записал 0, потом он же записал 5, а потом второй поток - писатель записал также 0. Проверяем, что все данные были прочитаны.

Листинг 34– Код программы client.c (взят из методических указаний):

#include <stdio.h>

#include <stdlib.h>

#include <sys/ipc.h>

#include <sys/sem.h>

#include <sys/shm.h>

#include <strings.h>

#include <string.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 < 5; ++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);

}

Листинг 35– Код программы server.c (взят из методических указаний):

//#pragma once

#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 <strings.h>

#include <string.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);

}

// 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 < 10; ++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);

}

}

}

Листинг 36– Результат выполнения программы:

./client

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

./client

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

./server

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 4 from cell 4

Remove 3 from cell 3

Remove 2 from cell 2

Remove 1 from cell 1

Remove 0 from cell 0

Задание 6.

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.

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.

Листинг 37– Код программы client.c (взят из методических указаний):

#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) {

// 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);

}

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;

}

Листинг 38– Код программы server.c (взят из методических указаний):

#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) {

// 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);

}

// 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;

}

Листинг 39– Результат выполнения программы:

./server

Using default port 8888

Some client sent: why&

Some client sent: ????

./client

Using default port 8888

Using default addr 127.0.0.1

Input request (empty to exit)

why&

Server's response: why&

Input request (empty to exit)

./client

Using default port 8888

Using default addr 127.0.0.1

Input request (empty to exit)

????

Server's response: ????

Input request (empty to exit)

Аналогичный эксперимент, но используя два разных компьютера можно выполнить, если в качестве аргумента клиенту передавать IP-адрес сервера.

Для множества клиентов TCP подходит не лучшим образом в случае, когда сообщения короткие, ведь UDP гораздо быстрее, и шанс потери данных низок, но с помощью TCP можно осуществлять полный контроль над трафиком. Для клиентов до 1000 потерь в обоих случаях не наблюдалось.

Листинг 40– Код программы UDPServer.c:

UDPServer.c

#include <stdio.h>

#include <sys/socket.h>

#include <arpa/inet.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include "Utility.h"

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

{

int sock; /\* Socket \*/

struct sockaddr\_in echoServAddr; /\* Local address \*/

struct sockaddr\_in echoClntAddr; /\* Client address \*/

unsigned int cliAddrLen; /\* Length of incoming message \*/

char echoBuffer[BUF\_SIZE]; /\* Buffer for echo string \*/

unsigned short echoServPort; /\* Server port \*/

int recvMsgSize; /\* Size of received message \*/

if (argc == 2)

echoServPort = atoi(argv[1]); /\* First arg: local port \*/

else

echoServPort = DEFAULT\_PORT;

/\* Create socket \*/

if ((sock = socket(PF\_INET, SOCK\_DGRAM, IPPROTO\_UDP)) < 0)

Error("socket() failed");

memset(&echoServAddr, 0, sizeof(echoServAddr)); /\* Zero out structure \*/

echoServAddr.sin\_family = AF\_INET; /\* Internet address family \*/

echoServAddr.sin\_addr.s\_addr = htonl(INADDR\_ANY); /\* Any incoming interface \*/

echoServAddr.sin\_port = htons(echoServPort); /\* Local port \*/

/\* Bind to the local address \*/

if (bind(sock, (struct sockaddr \*) &echoServAddr, sizeof(echoServAddr)) < 0)

Error("bind() failed");

for (;;) /\* Run forever \*/

{

/\* Set the size of the in-out parameter \*/

cliAddrLen = sizeof(echoClntAddr);

/\* Block until receive message from a client \*/

if ((recvMsgSize = recvfrom(sock, echoBuffer, BUF\_SIZE, 0,

(struct sockaddr \*) &echoClntAddr, &cliAddrLen)) < 0)

Error("recvfrom() failed");

printf("Message: \"%s\", from some client: %s\n", echoBuffer,

inet\_ntoa(echoClntAddr.sin\_addr));

/\* Send received datagram back to the client \*/

if (sendto(sock, echoBuffer, recvMsgSize, 0,

(struct sockaddr \*) &echoClntAddr, sizeof(echoClntAddr)) != recvMsgSize)

Error("sendto() sent a different number of bytes than expected");

}

/\* NOT REACHED \*/

}

Листинг 41– Код программы UDPClient.c :

#include <stdio.h>

#include <sys/socket.h>

#include <arpa/inet.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include "Utility.h"

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

{

int sock; /\* Socket descriptor \*/

struct sockaddr\_in echoServAddr; /\* Echo server address \*/

struct sockaddr\_in fromAddr; /\* Source address of echo \*/

unsigned short echoServPort; /\* Echo server port \*/

unsigned int fromSize; /\* In-out of address size for recvfrom() \*/

char \*servIP; /\* IP address of server \*/

char sendString[BUF\_SIZE]; /\* String to send to echo server \*/

char reciveString[BUF\_SIZE]; /\* Buffer for receiving echoed string \*/

if ((argc < 2) || (argc > 3)) /\* Test for correct number of arguments \*/

{

fprintf(stderr,"Usage: %s <Server IP> <Echo Port>\n", argv[0]);

exit(1);

}

servIP = argv[1]; /\* First arg: server IP address (dotted quad) \*/

echoServPort = atoi(argv[2]); /\* Second arg: string to echo \*/

/\* Create UDP socket \*/

if ((sock = socket(PF\_INET, SOCK\_DGRAM, IPPROTO\_UDP)) < 0)

Error("socket creature is failed");

memset(&echoServAddr, 0, sizeof(echoServAddr)); /\* Zero out structure \*/

echoServAddr.sin\_family = AF\_INET; /\* Internet addr family \*/

echoServAddr.sin\_addr.s\_addr = inet\_addr(servIP); /\* Server IP address \*/

echoServAddr.sin\_port = htons(echoServPort); /\* Server port \*/

for(;;)

{

printf("Input request (empty to exit)\n");

bzero(sendString, BUF\_SIZE);

fgets(sendString, BUF\_SIZE, stdin);

sendString[strlen(sendString) - 1] = '\0';

if(strlen(sendString) == 0) {

printf("Bye-bye\n");

break;

}

if (sendto(sock, sendString, BUF\_SIZE, 0, (struct sockaddr \*)

&echoServAddr, sizeof(echoServAddr)) != BUF\_SIZE)

Error("sendto() sent a different number of bytes than expected\n");

fromSize = sizeof(fromAddr);

if ((recvfrom(sock, reciveString, BUF\_SIZE, 0,

(struct sockaddr \*) &fromAddr, &fromSize)) != BUF\_SIZE)

Error("recvfrom() failed");

if(fromAddr.sin\_addr.s\_addr != echoServAddr.sin\_addr.s\_addr)

{

printf("Error: received a packet from unknown source.\n");

}

printf("Received: %s len=%d\n", reciveString, BUF\_SIZE);

}

close(sock);

exit(0);

}

**Вывод**

В ходе данной работы было изучено межпроцессное взаимодействие — IPC. 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. В Unix поддерживается два вида сигналов: надёжные и ненадёжные. Ненадежные сигналы более просты в использовании, в тоже время надежные сигналы позволяют отложить прием других сигналов до окончания обработки текущего. 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.

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. Кроме того, каналы работают в полудуплексном режиме, т.е. передают данные только в одну сторону. Каналы FIFO представляют собой вид IPC, который может использоваться только в пределах одного узла. Хотя FIFO и обладают именами в файловой системе, они могут применяться только в локальных файловых системах.

Сообщения являются мощным средством межпроцессного обмена данными. Время доставки сообщения сравнимо с временем доставки сигнала, однако сообщение несёт гораздо больше информации, чем сигнал. С помощью сообщений гораздо проще организовать асинхронный обмен данными между процессами, чем с помощью каналов. 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). Сокеты являются средством IPC, которое можно использовать не только между процессами на одном компьютере, но и в сетевом режиме.