**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

**ОТЧЕТ**

**по лабораторной работе №5-6**

**по дисциплине по дисциплине «Операционные системы и среды»**

**Тема: «IPC»**

Санкт-Петербург

2023

**Цель работы**

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

**Выполнение работы**

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

Разработана программа на языке Си, которая из процесса-потомка отлавливает сигнал SIGUSR1, выводит соответствующее сообщение и восстанавливает системный обработчик, а родительский процесс отлавливает и выводит соответствующее сообщение для сигналов SIGUSR1 и SIGUSR2, а также восстанавливает их системные обработчики, для сигнала SIGINT установлен системный обработчик, а сигнал SIGCHLD игнорируется.

Код программы представлен ниже в листинге 1, работа данной программы представлена в листинге 2.

Листинг 1 – Код программы 1.c

#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;

}

Листинг 2 – Выполнение программы 1.с

sudo ./1 &

[1] 4816

text@text-VirtualBox:~/56/1$

Father started: pid = 4817, ppid = 4816

Son started: pid = 4818, ppid = 4817

Successfully sent SIGUSR1

Caught signal SIGUSR1

Parent = 4816

sudo kill -SIGCHLD 4817

text@text-VirtualBox:~/56/1$ sudo kill -SIGUSR2 4817

Caught signal SIGUSR2

Parent = 4816

text@text-VirtualBox:~/56/1$ sudo kill -SIGINT 4817

[1]+ Прерывание sudo ./1

Как видно, сигнал SIGCHLD игнорируется, сигнал прерывания обрабатывается по умолчанию, а пользовательский сигнал сначала отлавливался, после первого раза он обрабатывался по умолчанию.

Теперь проведем эксперимент для нескольких процессов, первый из них отправляет пользовательский сигнал SIGUSR1, второй отправляет SIGUSR2, третий SIGCHLD. Код программ представлен ниже листинге 3, результат представлен в листинге 4.

Листинг 3 – Код программ father.c, son1.c, son2.c, son3.c

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;

}

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;

}

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;

}

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;

}

Результат программы представлен ниже в листинге 4.

Листинг 4 – Результат выполнения программы

sudo ./father &

[2] 5873

text@text-VirtualBox:~/56/1/1.1$

Father started: pid = 5874,ppid = 5873

Son2 started: pid = 5875,ppid = 5874

Son1 started: pid = 5876,ppid = 5874

Successfully sent SIGUSR2

Catched signal SIGUSR2

Parent = 5873

Catched signal SIGUSR1

Parent = 5873

Successfully sent SIGUSR1

Son3 started: pid = 5877,ppid = 5874

Successfully sent SIGCHLD

sudo kill -SIGUSR1 5874

[2]+ Определяемый пользователем сигнал 1 sudo ./father

text@text-VirtualBox:~/56/1/1.1$ sudo kill -SIGUSR1 5874

kill: (5874): Нет такого процесса

Теперь для эксперимента напишем программу, в которой 2 потока отлавливают пользовательские сигналы, Код программы и результат ее выполнения представлен ниже

Листинг 5 – 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 6 – Результат выполнения программы

sudo ./threads &

[1] 9401

text@text-VirtualBox:~/56/1/1.1/1.1.2$ Thread\_1 is started: tid = 9403 pid = 9402

Thread\_2 is started: tid = 9404 pid = 9402

sudo kill -SIGUSR2 9404

Caught SIGUSR2 signal in Thread 2!

text@text-VirtualBox:~/56/1/1.1/1.1.2$ sudo kill -SIGUSR2 9403

Caught SIGUSR2 signal in Thread 1!

text@text-VirtualBox:~/56/1/1.1/1.1.2$ sudo kill -SIGUSR2 9404

Caught SIGUSR2 signal in Thread 2!

text@text-VirtualBox:~/56/1/1.1/1.1.2$ sudo kill -SIGUSR2 9402

Caught SIGUSR2 signal in Thread 1!

text@text-VirtualBox:~/56/1/1.1/1.1.2$ sudo kill 9402

[1]+ Завершено sudo ./threads

Теперь проведем эксперимент, в котором каждый порожденный процесс имеет два потока, каждый из которых отлавливает пользовательские сигналы. Код программ представлен ниже в листинге 7.

Листинг 7 – Код программ

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;

}

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;

}

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 8 – Результат работы программы

sudo ./father &

[1] 9615

text@text-VirtualBox:~/56/1/1.1/1.1.3$

Father started: pid = 9616,ppid = 9615

Son1 started: pid = 9618,ppid = 9616

Son2 started: pid = 9617,ppid = 9616

Son 2 Thread\_2 is started: tid = 9622 pid = 9617

Son 1 Thread\_1 is started: tid = 9620 pid = 9618

Son 2 Thread\_1 is started: tid = 9619 pid = 9617

Son 1 Thread\_2 is started: tid = 9621 pid = 9618

sudo kill -SIGUSR1 9622

Caught SIGUSR1 signal in Son 2!

text@text-VirtualBox:~/56/1/1.1/1.1.3$ sudo kill -SIGUSR1 9620

text@text-VirtualBox:~/56/1/1.1/1.1.3$ Caught SIGUSR1 signal in Son 1!

sudo kill -SIGUSR1 9619

Caught SIGUSR1 signal in Son 2!

text@text-VirtualBox:~/56/1/1.1/1.1.3$ sudo kill -SIGUSR1 9621

Caught SIGUSR1 signal in Son 1!

1.2. Разработана программа, которая обрабатывает сигнал SIGUSR и совершает отложенную обработку (спустя минуту) сигнала SIGINT.

Листинг 9 – Код программы

#include <stdio.h>

#include <signal.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <unistd.h>

#include <stdlib.h>

#include <fcntl.h>

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

// надежная обработка сигналов

struct sigaction act,oldact;

act.sa\_handler = hnd;

sigemptyset(&act.sa\_mask);

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

act.sa\_flags = 0;

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

return SIG\_ERR;

return act.sa\_handler;

}

void hndUSR1(int sig) {

if(sig != SIGUSR1) {

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

return;

}

printf("SIGUSR1 catched\n");

sleep(60);

}

int main() {

mysig(SIGUSR1,hndUSR1);

for(;;) {

pause();

}

return 0;

}

Листинг 10 – Результат работы программы

sudo ./1\_2 &

[2] 10549

text@text-VirtualBox:~/56/1/1.2/1.2.1$ sudo kill -SIGUSR1 10549

text@text-VirtualBox:~/56/1/1.2/1.2.1$ SIGUSR1 catched

sudo kill -SIGINT 10549

text@text-VirtualBox:~/56/1/1.2/1.2.1$ jobs -l

[2]+ 10549 Запущен sudo ./1\_2 &

text@text-VirtualBox:~/56/1/1.2/1.2.1$ jobs -l

[2]+ 10549 Прерывание sudo ./1\_2

text@text-VirtualBox:~/56/1/1.2/1.2.1$ jobs -l

text@text-VirtualBox:~/56/1/1.2/1.2.1$

Данный эксперимент демонстрирует преимущество надёжных сигналов – возможность отложенной обработки. Также изменён обработчик сигналов таким образом, что из одного обработчика сигнала (SIGUSR1) посылается другой сигнал (SIGINT).

Листинг 11 – Код программы

#include <stdio.h>

#include <signal.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <unistd.h>

#include <stdlib.h>

#include <fcntl.h>

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

// надежная обработка сигналов

struct sigaction act,oldact;

act.sa\_handler = hnd;

sigemptyset(&act.sa\_mask);

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

act.sa\_flags = 0;

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

return SIG\_ERR;

return act.sa\_handler;

}

void hndUSR1(int sig) {

if(sig != SIGUSR1) {

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

return;

}

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

kill(getpid(),SIGINT);

sleep(10);

}

int main() {

mysig(SIGUSR1,hndUSR1);

for(;;) {

pause();

}

return 0;

}

Листинг 12 – Результат выполнения программы

sudo ./1\_2 &

[1] 10861

text@text-VirtualBox:~/56/1/1.2/1.2.2$ sudo kill -SIGUSR1 10861

SIGUSR1 catched, sending SIGINT

text@text-VirtualBox:~/56/1/1.2/1.2.2$ jobs

[1]+ Запущен sudo ./1\_2 &

text@text-VirtualBox:~/56/1/1.2/1.2.2$ jobs -l

[1]+ 10861 Прерывание sudo ./1\_2

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

Листинг 13 – Код программы

#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 14 – Результат выполнения

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, так как сигналы обрабатываются в порядке их отправки. В частности, сначала обрабатываются обычные сигналы в том порядке, в котором они были отправлены (т. е. 1, 2, 3, 4), а затем сигналы реального времени в том порядке, в котором они были отправлены (т. е. 1, 2, 3, 4). 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. Кроме того, он подтверждает, что сигналы реального времени с одинаковым приоритетом обрабатываются в порядке FIFO, как и обычные сигналы.

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

В текстовом файле from.txt находится 1 строка: Hello, Robert. В итоге выполнения программы эта строка должна вывестись в файл и в терминал.

Код программы и результат выполнения представлен ниже.

Листинг 15 – Код программы

#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 16 – Результат выполнения программы

sudo ./pipe

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

pipe is successfully created

Read from pipe: Hello, Robert

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.

ТЗ: 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. Ниже представлены коды программ и результат их выполнения представлены ниже.

Листинг 17 – Код программ

server.c

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#define DEF\_FILENAME "testFile.txt"

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

char fileName[30];

if(argc < 2) {

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

strcpy(fileName,DEF\_FILENAME);

}

else

strcpy(fileName,argv[1]);

// создаем два канала

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

if(res) {

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

exit(1);

}

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

if(res) {

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

exit(1);

}

// открываем первый канал для записи

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

if(chan1 == -1) {

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

exit(0);

}

// открываем второй канал для чтения

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

if(chan2 == -1) {

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

exit(0);

}

// пишем имя файла в первый канал

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

// читаем содержимое файла из второго канала

char buf [100];

for(;;) {

bzero(buf,100);

res = read(chan2,buf,100);

if(res <= 0)

break;

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

}

close(chan1);

close(chan2);

unlink("channel1");

unlink("channel2");

return 0;

}

client.c

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

int main() {

// каналы сервер уже создал, открываем их

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

if (chan1 == -1) {

printf("Can't open channel1 for reading\n");

exit(0);

}

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

if (chan2 == -1) {

printf("Can't open channel2 for 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;

}

Листинг 18 – Результат выполнения программ

sudo ./server

Using default file name 'testFile.txt'

Part of file: Hello, Robert!

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

Листинг 19 – Измененный код программы

#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("Servr finished\n");

return 0;

}

Листинг 20 – Результат выполнения программы

Using default file name 'testFile.txt'

Waiting for client write to channel

q

Part of file: Hello, Robert!

Server finished

4. Листинг 21 – Коды программ

server.c

#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 receiving 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;

}

client.c

#include <stdio.h>

#include <stdlib.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <sys/types.h>

#include <signal.h>

#include <string.h>

#define DEF\_KEY\_FILE "key"

typedef struct {

long type;

char buf[100];

} Message;

int queue;

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

char keyFile[100];

bzero(keyFile,100);

if(argc < 2) {

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

strcpy(keyFile,DEF\_KEY\_FILE);

}

else

strcpy(keyFile,argv[1]);

key\_t key;

key = ftok(keyFile,'Q');

if(key == -1) {

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

exit(1);

}

queue = msgget(key,0);

if (queue < 0) {

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

exit(4);

}

// основной цикл работы программы

Message mes;

int res;

for(;;) {

bzero(mes.buf,100);

// читаем сообщение с консоли

fgets(mes.buf,100,stdin);

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

// шлем его серверу

mes.type = 1L;

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

if(res != 0) {

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

exit(1);

}

// получаем ответ, что все хорошо

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

if(res < 0) {

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

exit(1);

}

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

}

return 0;

}

Листинг 22 – Результат выполнения с двумя клиентами

server

Using default key file key

Client's request: hellloooooooooooo

Client's request: hello you to

client 1

Using default key file key

hellloooooooooooo

Server's response: OK

client2

Using default key file key

hello you to

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.

5.

**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. Многие сетевые приложения построены на о