**СОДЕРЖАНИЕ**

ВВЕДЕНИЕ

1. Анализ задания и разработка алгоритма
2. Теоретические сведения
3. Листинг программы
4. Тестирование

ВЫВОДЫ

ЛИТЕРАТУРА

**ВВЕДЕНИЕ**

Целью выполнения работы является закрепление знаний умений и навыков в области взаимодействия модулей, использования системных вызовов и библиотечных функций управления процессами и файлами современных операционных систем для создания системных и пользовательских программ, процедур и функций на примере ОС семейства UNIX/Linux. В ходе выполнения работы студенту необходимо продемонстрировать знания функций, алгоритмов, механизмов управления процессами, разделяемыми ресурсами, файлами, вводом-выводом.

Процесс — понятие, которое определяется по-разному. Это может быть — “упорядоченный набор команд и принадлежащих ему ресурсов”. С точки зрения ОС Unix процесс — это объект, зарегистрированный в специальной таблице процессов.

Телом процесса называется набор команд и данных, которыми оперирует процесс.

Контекст процесса — атрибут, который присутствует практически во всех ОС, в разных ОС он может называться по-разному. Контексты всех процессов размещаются в адресном пространстве ОС и содержат оперативную информацию о состоянии процесса и текущую информацию, связанную с процессом и его запуском.

**1 АНАЛИЗ ЗАДАНИЯ И РАЗРАБОТКА АЛГОРИТМА**

По заданию согласно варианта по списку необходимо организовать копирование содержимого из Файла 1 в остальные файлы (1->2, 1->3, 1->4).

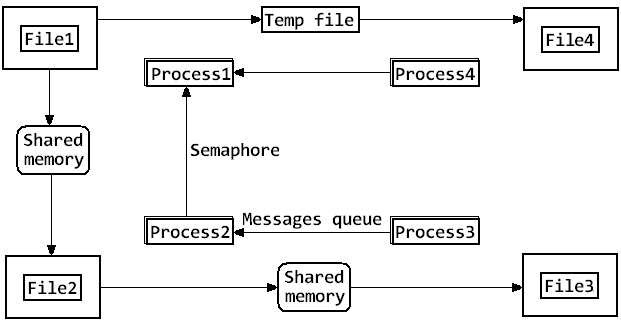
Основные принципы по которым будет создаваться программа:

* Будут созданы 4-е процесса, а именно 1-ый процесс породит 2-ый процесс, 1-ый процесс, в свою очередь породит 3-ий процесс, 1-ый процесс породит 4-тый процесс.
* Каждый процесс будет иметь файл с соответствующими именами – file1, file2, file3, file4.
* Процессы будут обмениваться через разделяемую память и временный файл.
* Обмен содержимым файлов будет происходить по сигналу, семафорам и обмену сообщениями.

Процессы **Process1** и **Process4** обмениваются пользовательскими сигналами, по которым выполняется запись процессом P**rocess1** во временный файл **Temp file**, после чего **Process4** считывает из него данные, удаляет временный файл, затем записывает информацию в **File4,** ждет завершения обмена между процессами **Process2, Process3,** закрывает разделяемую память и уничтожает всю группу процессов.

Процессы **Process1** и **Process2** взаимодействуют с помощью семафоров. **Process1** записывает в разделяемую память содержимое файла **File1**, после этого по семафору **Process2** считывает из памяти данные и пишет в **File2**.

Процессы **Process2** и **Process3** взаимодействуют с помощью очереди сообщений. Когда данные уже записаны процессом **Process2** в **File2**, он отсылает сообщение своему потомку, после чего **Process3** считывает из разделяемой памяти данные, пишет в свой файл **File3,** отсылает сообщение назад и завершается**,** после чего закрывается и его родитель **Process2**.



**Рис.1 Схема взаимодействия процессов**

**2 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

С использованием функций в языке СИ связаны три понятия - определение функции (описание действий, выполняемых функцией), объявление функции (задание формы обращения к функции) и вызов функции.

Определение функции задает тип возвращаемого значения, имя функции, типы и число формальных параметров, а также объявления переменных и операторы, называемые телом функции, и определяющие действие функции. В определении функции также может быть задан класс памяти.

Функция fork:

int fork ( )

Вызов fork приводит к созданию нового процесса (порожденного процесса) - точной копии процесса, сделавшего вызов (родительского процесса). Точнее, порожденный процесс наследует у родительского процесса следующие характеристики:

* Окружение.
* Флаг "закрыть при выполнении вызова exec"
* Способы обработки сигналов (то есть SIG\_DFL, SIG\_IGN, SIG\_HOLD, адреса функций обработки сигналов).
* Разрешение переустанавливать действующий идентификатор пользователя.
* Разрешение переустанавливать действующий идентификатор группы.
* Состояние профилирования (включено/выключено).
* Значение поправки к приоритету.
* Все присоединенные разделяемые сегменты памяти.
* Идентификатор группы процессов.
* Идентификатор группы терминала.
* Текущий рабочий каталог.
* Корневой каталог.
* Маска режима создания файлов.
* Ограничение на размер файла.

Порожденный процесс отличается от родительского процесса следующим:

* Порожденный процесс имеет свой уникальный идентификатор процесса.
* Порожденный процесс имеет иной идентификатор родительского процесса, равный идентификатору процесса, его породившего.
* Порожденный процесс имеет свои собственные копии родительских дескрипторов файлов. Каждый дескриптор файла порожденного процесса разделяет с соответствующим родительским дескриптором файла общий указатель текущей позиции в файле.
* Все semadj значения сбрасываются.
* Порожденный процесс не наследует у родительского процесса признаков удержания в памяти сегмента команд, данных или всего процесса целиком.
* Обнуляются счетчики времени, потраченного для обслуживания этого процесса (tms\_utime, tms\_stime, tms\_cutime, tms\_cstime). Отменяется запрос к будильнику.

**3 ЛИСТИНГ ПРОГРАММЫ**

Программа состоит из главного модуля rgr.c:

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <sys/sem.h>

#include <unistd.h>

#include <signal.h>

#include <fcntl.h>

#include <stdio.h>

#include <errno.h>

#define SHMKEY 5

#define SEMKEY 5

#define K 32

#define Count 4

#define InitVal {1,0,0,0}

#define MSGKEY 5

#define InitT 3

void creat\_mem(void);

void creat\_sem(void);

void prss1(void);

void prss2(void);

void prss3(void);

void prss4(void);

int pid1; int pid2; int pid3; int pid4; int pid; int ppid;

int fd; int st;

extern int p14(int), p41(int);

//mem

int shmid;

int \*pint;

char \*addr;

//sem

int semid;

short initarray[Count] = InitVal;

struct sembuf p, v;

//message:

int prnum;

int msgid;

long nextT;

struct {

long mtype;

int Data;

} Message;

int main(void)

{

remove("file2");

remove("file3");

remove("file4");

creat\_mem();

creat\_sem();

pid1 = getpid();

pid = fork();

if (!pid) prss2();

else prss1();

sleep(2);

wait(&st);

}

void creat\_mem(void)

{

printf("--- func creat\_mem(): memory creating: %dbytes --- pid=%d\n", K, getpid());

shmid = shmget(SHMKEY, 1\*K, 0777|IPC\_CREAT);

addr = shmat(shmid, 0, 0);

pint = (int \*) addr;

}

void creat\_sem(void)

{

printf("--- func creat\_sem(): semaphor creating: --- pid=%d\n", getpid());

semid = semget(SEMKEY, Count, 0777|IPC\_CREAT);

semctl(semid, Count, SETALL, initarray);

p.sem\_op = -1;

p.sem\_flg = SEM\_UNDO;

v.sem\_op = 1;

v.sem\_flg = SEM\_UNDO;

}

void creat\_mesg(void)

{

msgid = msgget(MSGKEY, 0666|IPC\_CREAT);

msgsnd(msgid, (struct msgbuf \*) &Message, 8, 0);

}

void prss1(void)

{

int i;

char buf[32] = " ";

prnum = 1;

p.sem\_num = 0;

v.sem\_num = 1;

ppid = getppid();

printf(" =I= prss%d, pid = %d, parent: %d\n", prnum, pid1, ppid);

pid = fork();

if (!pid) prss4();

else

{

fd = open("file1", O\_RDONLY);

read(fd,buf,strlen(buf));

close(fd);

printf("I: reading from FILE1:\t%s\n",buf);

signal(SIGUSR2, p41);

sleep(1);//ojidanie priema signala ot prssa4

kill(pid1+2,SIGUSR1);

printf("================== prss1: writing to memory\n");

for(i = 0; i <= 31; ++i) pint[i] = buf[i];

semop(semid, &p, 1);

semop(semid, &v, 1);

sleep(2);

wait(&st);

wait(&st);

printf(" =I= \_\_eto konec prssa%d\n", prnum);

}

}

void prss2(void)

{

int i;

char buf\_2[32]=" ";

prnum = 2;

p.sem\_num = 1;

pid2 = getpid();

ppid = getppid();

printf(" =II= prss%d, pid = %d, parent: %d\n", prnum, pid2, ppid);

creat("file2",fd);

pid = fork();

if (!pid) prss3();

else

{

semop(semid, &p, 1);

printf("================== prss%d: file2 editing /Semaphor/\n", prnum);

fd = open("file2", O\_WRONLY);

for(i = 0; i <= 31; ++i) buf\_2[i] = pint[i];

write(fd,buf\_2,strlen(buf\_2));

printf("II: writing to FILE2:\t%s\n",buf\_2);

printf("--- func creat\_mesg(): message creating: --- pid=%d\n", pid2);

Message.mtype = InitT;

Message.Data=3;

creat\_mesg();

printf(" =II= \_\_eto konec prssa%d\n", prnum);

fclose(fd);

}

}

void prss3(void)

{

int i;

char buf\_3[32]=" ";

prnum = 3;

pid3 = getpid();

ppid = getppid();

printf(" =III= prss%d, pid = %d, parent: %d\n", prnum, pid3, ppid);

creat("file3",fd);

msgrcv(msgid, (struct msgbuf \*) (&Message), 8, prnum, 0);

if (Message.Data==3)

{

printf("================== prss%d: file3 editing /Message/\n", prnum);

fd = open("file3", O\_WRONLY);

for(i = 0; i <= 31; ++i) buf\_3[i] = pint[i];

write(fd,buf\_3,strlen(buf\_3));

printf("III: writing to FILE3:\t%s\n",buf\_3);

printf(" =III= \_\_eto konec prssa%d\n", prnum);

fclose(fd);

}

}

void prss4(void)

{

int i;

prnum = 4;

pid4 = getpid();

ppid = getppid();

printf(" =IV= prss%d, pid = %d, parent: %d\n", prnum, pid4, ppid);

creat("file4",fd);

signal(SIGUSR1, p14);

kill(pid1,SIGUSR2);

sleep(1);

printf(" =IV= \_\_eto konec prssa%d\n", prnum);

shmctl(shmid,IPC\_RMID,0);

printf("================== prss4: memory closed\n");

kill(0,SIGKILL);

}

int p14(int signum) //2-oj sig

{

char temp\_buf4[32]=" ";

signal(SIGUSR1, p14);

printf("\*\*\*SIGUSR1\*\*\* : prss 4 (%d) has got a signal from prss 1 (%d)\n",pid4,pid1);

fd = open("temp\_file", O\_RDONLY);

read(fd,temp\_buf4,strlen(temp\_buf4));

close(fd);

creat("file4",fd);

printf("\* \*SIGUSR1\* \* : writing from temp\_file to file4\n");

fd = open("file4", O\_WRONLY);

write(fd,temp\_buf4,strlen(temp\_buf4));

close(fd);

printf("IV: writing to FILE4:\t%s\n",temp\_buf4);

remove("temp\_file");

printf("\* \*SIGUSR1\* \* : temp\_file was removed\n");

printf("\*\*\*SIGUSR1\*\*\* : end\n");

}

int p41(int signum) //1-ij sig

{

char temp\_buf1[32]=" ";

signal(SIGUSR2, p41);

printf("\*\*\*SIGUSR2\*\*\* prss 1 (%d) has got a signal from prss 4 (%d)\n",pid1,pid1+2);

fd = open("file1", O\_RDONLY);

read(fd,temp\_buf1,strlen(temp\_buf1));

close(fd);

creat("temp\_file",fd);

printf("\* \*SIGUSR2\* \* : temp\_file was created\n");

fd = open("temp\_file", O\_WRONLY);

write(fd,temp\_buf1,strlen(temp\_buf1));

close(fd);

printf("\*\*\*SIGUSR2\*\*\* : end\n");

}

**4 ТЕСТИРОВАНИЕ**

Результат выполнения программы в консоли:

yuna@YunieHost:/media/8\_Gb\_hard\_ONPU/LINUX/rgr 28march$ ./rgr

--- func creat\_mem(): memory creating: 32bytes --- pid=6798

--- func creat\_sem(): semaphor creating: --- pid=6798

=II= prss2, pid = 6799, parent: 6798

=I= prss1, pid = 6798, parent: 6655

=III= prss3, pid = 6801, parent: 6799

=IV= prss4, pid = 6800, parent: 6798

I: reading from FILE1: << RGR sPO by yuna 18.05.2008 >>

\*\*\*SIGUSR2\*\*\* prss 1 (6798) has got a signal from prss 4 (6800)

\* \*SIGUSR2\* \* : temp\_file was created

\*\*\*SIGUSR2\*\*\* : end

================== prss1: writing to memory

================== prss2: file2 editing /Semaphor/

II: writing to FILE2: << RGR sPO by yuna 18.05.2008 >>

--- func creat\_mesg(): message creating: --- pid=6799

=II= \_\_eto konec prssa2

\*\*\*SIGUSR1\*\*\* : prss 4 (6800) has got a signal from prss 1 (6798)

================== prss3: file3 editing /Message/

III: writing to FILE3: << RGR sPO by yuna 18.05.2008 >>

=III= \_\_eto konec prssa3

\* \*SIGUSR1\* \* : writing from temp\_file to file4

IV: writing to FILE4: << RGR sPO by yuna 18.05.2008 >>

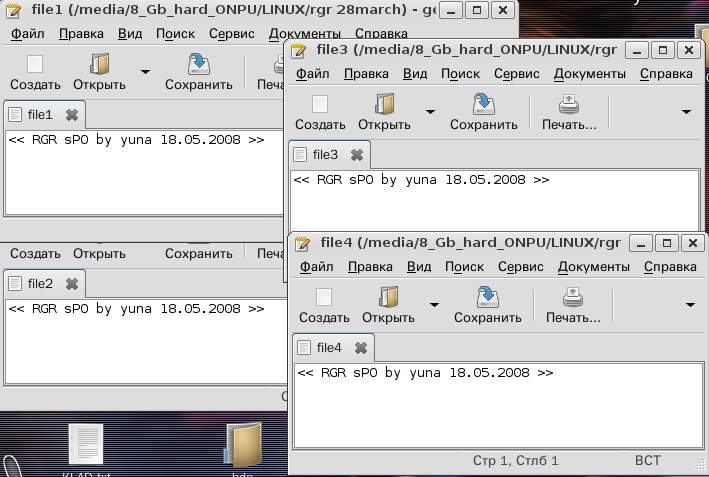
\* \*SIGUSR1\* \* : temp\_file was removed

\*\*\*SIGUSR1\*\*\* : end

=IV= \_\_eto konec prssa4

================== prss4: memory closed

Killed



**Рис.2 Результат работы программы (содержимое из file1 было скопировано в остальные файлы)**

Следовательно, программа работает корректно и поставленная на данную расчетно-графическую работу задача была решена.

**ВЫВОДЫ**

В данной работе частично описана структура системы UNIX, взаимоотношения между процессами, выполняющимися в режиме задачи и в режиме ядра. Процессы выполняются в режиме задачи или в режиме ядра, в котором они пользуются услугами системы благодаря наличию набора обращений к операционной системе.

Архитектура системы поддерживает такой стиль программирования, при котором из небольших программ, выполняющих только отдельные функции, но хорошо, составляются более сложные программы, использующие механизм каналов и переназначение ввода-вывода.

Обращения к операционной системе позволяют процессам производить операции, которые иначе не выполняются. В дополнение к обработке подобных обращений ядро операционной системы осуществляет общие учетные операции, управляет планированием процессов, распределением памяти и защитой процессов в оперативной памяти, обслуживает прерывания, управляет файлами и устройствами и обрабатывает особые ситуации, возникающие в системе.

В функции ядра системы UNIX намеренно не включены многие функции, являющиеся частью других операционных систем, поскольку набор обращений к системе позволяет процессам выполнять все необходимые операции на пользовательском уровне.

**ЛИТЕРАТУРА**

1. Дж. Такет (мл.), С.Барнет. Использование Linux/ Специальное издание.: 5-е изд.: Пер. с англ.: Уч.пос. – М.: Издательский дом «Вильямс», 2000. – 784 с.
2. Максимальная защита Linux. Искусство настройки.: Пер. с англ./ под.ред. Дж.Рея – СПб.: ООО «ДиаСофтЮП», 2002. – 752 с.
3. Браун С. Операционная система UNIX - М.: Мир, 1986 - 463 с.