Министерство образования Республики Беларусь

Учреждение образования

Белорусский государственный университет
информатики и радиоэлектроники

Кафедра программного обеспечения
информационных технологий

Алексеев Игорь Геннадиевич,
Бранцевич Петр Юльянович

“ОПЕРАЦИОННЫЕ СИСТЕМЫ И СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ”

Часть 1

учебно-методическое пособие для студентов специальности

«Программное обеспечение информационных технологий»

Минск 2009

УДК 004.04 (075.8)
ББК 32.973 я 73
А47

Рецензент:

А47 **Алексеев И.Г. Учебно-методическое пособие** Операционные системы и системное программирование: для студ. спец. «Программное обеспечение информационных технологий»/ И.Г Алексеев, П.Ю. Бранцевич – Мн.: БГУИР, 2009. – 73 с.
ISBN 985-444-№

В пособии рассмотрены основные команды операционной системы UNIX, предназначенные для работы с файлами и каталогами, а также для создания процессов и организации взаимодействия между ними. Даны структуры лабораторных работ по курсу «ОСиСП»

УДК 004.04 (075.8)
ББК 32.973 я 73

А47

ISBN 985-444-387-6

СОДЕРЖАНИЕ

1. ОСНОВНЫЕ КОМАНДЫ ОС UNIX 4
2. ЛАБОРАТОРНЫЕ РАБОТЫ 7
Лабораторная работа №1 Работа в ОС UNIX/Linux, интерпретатор BASH 7
Лабораторная работа № 2 Работа с файлами и каталогами ОС UNIX 12
Лабораторная работа № 3 Процессы в ОС UNIX/Linux 17
Лабораторная работа № 4 Использование сигналов в ОС UNIX/Linux 21
Лабораторная работа № 5 Использование каналов в ОС UNIX/Linux 26
Лабораторная работа № 6 Потоки в ОС UNIX/Linux 33
Лабораторная работа № 7 Семафоры в ОС UNIX/Linux 36
Лабораторная работа № 8 Использование общей памяти в ОС UNIX/Linux 39

ЛИТЕРАТУРА 42

1. ОСНОВНЫЕ КОМАНДЫ ОС UNIX

Операционная система ОС ***Linux*** создана на основе ОС ***UNIX*** и во многом имеет схожую структуру и систему команд. Пользователь может работать в текстовом режиме с помощью командной строки, или с использованием графического интерфейса ***X Window*** и одного из менеджеров рабочего стола (например, ***KDE*** или ***GNOME***). Причем, одновременно в системе могут работать 7 пользователей (6- в текстовом режиме консоли и 1 – в графическом режиме), переключение между пользователями осуществляется по нажатию клавиш:

Ctrl + ***Alt*** + ***F1*** ***Ctrl*** + ***Alt*** + ***F7***

В табл. 1 приведены основные команды системы

Таблица 1

Команда	Аргументы/ключи	Пример	Описание
dir	каталог	***dir*** ***dir /home***	Выводит на консоль содержимое каталога
ls	***-all*** и другие (см. man)	***ls -all***	Выводит на консоль содержимое каталога
ps	***-a*** ***-x*** и другие (см. man)	***ps -a***	Выводит на консоль список процессов
mkdir	имя каталога	***mkdir stud11***	Создает каталог
rmdir	имя каталога	***rmdir stud11***	Удаляет каталог
rm	файл	***rm myfile1***	Удаляет файл
mv	файл новое_имя	***mv myfile1 myf1***	Переименование файла
cat	файл	***cat 1.txt***	Вывод файла на консоль
cd	имя каталога	***cd home***	Переход по каталогам
grep	(см. man)	***grep "^a" "words.txt"***	Поиск строки в файле
kill	***pid*** процесса	***kill 12045***	Уничтожает процесс
top			Выводит на консоль список процессов
htop			Выводит на консоль полный список запущенных процессов
su			Переход в режим root
chmod	права_доступа файл	***chmod 777 1.txt***	Изменение прав доступа к файлам
mount	устройство каталог	***mount /dev/cdrom /MyCD***	Монтирование устройств
dd	***if***=файл ***of***=файл ***bs=n count=n***	***dd if=/dev/hda1 of=/F.bin bs=512***	Копирование побайтное

		count=1	
ln	файл1 файл2 ***-l***	***ln файл1 файл2*** ***ln –l файл1 файл2***	Создать жёсткую или символическую ссылку на файл
uname	***-a***	***uname –a***	Информация о системе
find	***find*** файл	***find /home a1.txt***	Поиск файлов
man		***man fgetc***	Справка по системе
info		***info fgetc***	Справка по системе

Linux и Windows используют различные файловые системы для хранения и организации доступа к информации на дисках. В Linux используются файловые системы- ***Ext2/Ext3***, ***RaiserFS, FFS*** и другие. Все файловые системы имеют поддержку ***журналирования***. ***Журналируемая*** файловая система сначала записывает изменения, которые она будет проводить в отдельную часть файловой системы (***журнал***) и только потом вносит необходимые изменения в остальную часть файловой системы. После удачного выполнения всех транзакций, записи удаляются из ***журнала***. Это обеспечивает лучшее сохранение целостности системы и уменьшает вероятность потери данных. Следует отметить, что ***Linux*** поддерживает доступ к ***Windows***-разделам.

Файловая система ***Linux*** имеет лишь один корневой каталог, который обозначается косой чертой (***/***). В файловой структуре ***Linux*** нет дисков ***A, B, C, D*** …., а есть только каталоги. В ***Linux*** различаются прописные и строчные буквы в командах, именах файлов и каталогов. В ***Windows*** у каждого файла существует лишь одно имя, в ***Linux*** их может быть много. Это – «***жесткие***» ссылки, которые указывают непосредственно на индексный дескриптор файла. Жесткая ссылка – это один из принципов организации файловой системы ***Linux***.

Структура каталогов ОС ***Linux*** представлена в табл. 1. Есть также несколько полезных сокращений для имен каталогов:

- Одиночная точка (**.**) обозначает текущий рабочий каталог.
- Две точки (**..**) обозначают родительский каталог текущего рабочего.
- Тильда (~) обозначает домашний каталог пользователя (обычно это каталог, который является текущим рабочим при запуске Bash).

Таблица 1

/	Корневой каталог
/bin	Содержит исполняемые файлы самых необходимых для работы системы программ. Каталог ***/bin*** не содержит подкаталогов.
/boot	Здесь находятся само ядро системы (файл ***vmlinuz***-...) и файлы, необходимые для его загрузки.
/dev	Каталог ***/dev*** содержит файлы устройств (драйверы).
/etc	Это каталог конфигурационных файлов, т. е. файлов, содержащих

	информацию о настройках системы (например, настройки программ).
/home	Содержит домашние каталоги пользователей системы.
/lib	Здесь находятся библиотеки (функции, необходимые многим программам).
/media	Содержит подкаталоги, которые используются как точки монтирования для сменных устройств (CD-ROM'ов, floppy-дисков и др.)
/mnt	Данный каталог (или его подкаталоги) может служить точкой монтирования для временно подключаемых файловых систем.
/proc	Содержит файлы с информацией о выполняющихся в системе процессах.
/root	Это домашний каталог администратора системы.
/sbin	Содержит исполняемые программы, как и каталог ***/bin***. Однако использовать программы, находящиеся в этом каталоге может только администратор системы (***root***).
/tmp	Каталог для временных файлов, хранящих промежуточные данные, необходимых для работы тех или иных программ, и удаляющиеся после завершения работы программ.
/usr	Каталог для большинства программ, которые не имеют значения для загрузки системы. Структура этого каталога фактически дублирует структуру корневого каталога.
/var	Содержит данные, которые были получены в процессе работы одних программ и должны быть переданы другим, и файлы журналов со сведениями о работе системы.

2. ЛАБОРАТОРНЫЕ РАБОТЫ

Лабораторная работа №1

УПРАВЛЕНИЕ ОС LINUX, ИНТЕРПРЕТАТОР BASH

Цель работы – изучить основные объекты, команды, типы данных и операторы управления интерпретатора BASH; создать скрипт-файл.

Теоретическая часть

Bash - это ***sh***-совместимый интерпретатор командного языка, выполняющий команды, прочитанные со стандартного входного потока или из файла. ***Скрипт-файл*** – это обычный текстовый файл, содержащий последовательность команд ***bash***, для которого установлены права на выполнение. Пример скрипта, выводящего содержимое текущего каталога на консоль и в файл:

```
#!/bin/bash
dir
dir  > 1.txt
```

Следующие переменные используются командным интерпретатором.

$0,$1,$2,$3…	Значения аргументов командной строки при запуске скрипта. Где ***$0***-имя самого файла скрипта, ***$1***- первый аргумент, ***$2***- второй аргумент, и т.д.
$@	Все аргументы командной строки, каждый в кавычках
$?	Код возврата последней команды

Пример простого скрипта, выводящего на консоль и в файл содержимое каталога, где имя каталога передаётся скрипту в качестве аргументов при запуске:

Запуск скрипта: ***>./mydir /home/stud***

Скрипт:

```
#!/bin/bash
dir $1
dir  $1 > 1.txt
```

Можно создать собственную переменную и присвоить ей значение:

```
A=121
A="121"
let  A=121
let  "A=A+1"
```

Вывод значения на консоль: ***echo $A***

Проверка условия: ***test[expr]***

где ***expr***: а) для строк: $S_1 = S_2$ — S_1 содержит S_2

$S_1 != S_2$ — S_1 не содержит S_2

-n S_1 — если длина $S_1 > 0$

-z S_1 — если длина $S_1 = 0$

б) целые i_1 и i_2

i_1 – ***ge*** i_2

i_1 – ***gt*** i_2

i_1 – ***ie*** i_2

i_1 – ***et*** i_2

i_1 – ***nt*** i_2

в) файлы

-d name_file — является ли файл каталогом

-f name_file — является ли файл обычным файлом

-r name_file — доступен ли файл для чтения

-s name_file — имеет ли файл ненулевую длину

-w name_file — доступен ли файл для записи

-x name_file — является ли файл исполняемым

г) логически операции

!exp — логическое отрицание (не)

exp1 –a exp2 — умножение условий (и)

exp1 –o exp2 — сложение условий (или)

Проверка условия: ***if [expr]***

```
then   com 1
  …
  com   n
(elif expr2
        com1
  …
  com   n
)
else
        com 1
  …
  com   n
fi
```

Если условие ***expr=true*** то команда ***com 1… com n***

Проверка нескольких условий: ***case string1 in***

```
str 1)
      com 1
      …
       com n
      ; ;
str 2)
      com 1
```

```
        …
         com n
        ; ;
str 3)
         com 1
         …
        com n
        ; ;
*)                          // default
        com 1
        …
         com n
        ; ;
esac
```

Функция пользователя: ***fname2 (arg1,arg2...argN)***

```
{
commands
}
```

Организация циклов:

1. ```
   for   var1  in   list
         do
         com1
         …
         com n
         done
   ```

2. ```
   while  exp
          com1
          …
          com n
          end
   ```

3. ```
   until   exp          // аналог do-while
          do
          com1
                …
          com n
          done
   ```

Порядок выполнения работы

1. Изучить теоретическую часть лабораторной работы.
2. В консольном режиме создать, используя команды из табл.1, в ***домашней папке*** подкаталог: ***/номер_группы/ФИО_студента,*** где в дальнейшем будут храниться все файлы студента. Перейти в корневой каталог
   ```

и вывести его содержимое используя команды ***dir*** и ***ls –all*** , проанализировать различия.

3. Проверить действие команд ***ps***, ***ps –x, top, htop***. Найти в справочной системе используя команду ***man*** справку по функциям ***fprintf***, ***fputc*** и команде ***ls***.

4. В текстовом редакторе ***joe*** (вызов: ***joe 1.c***) написать программу ***1.c***, выводящую на экран фразу ***"HELLO SUSE Linux"***. Компилировать полученную программу компилятором ***gcc***: ***gcc 1.c –o 1.exe***. Запустить полученный файл ***1.exe*** на выполнение: ***./1.EXE***

5. Написать скрипт, выводящий на консоль и в файл все аргументы командной строки.

6. Написать скрипт, выводящий в файл (имя файла задаётся пользователем в качестве первого аргумента командной строки) имена всех файлов с заданным расширением (третий аргумент командной строки) из заданного каталога (имя каталога задаётся пользователем в качестве второго аргумента командной строки).

7. Написать скрипт, компилирующий и запускающий программу (имя исходного файла и ***exe-*** файла результата задаётся пользователем в качестве аргументов командной строки). В случае ошибок при компиляции вывести на консоль сообщение об ошибках и не запускать программу на выполнение.

Варианты индивидуальных заданий

1. Написать скрипт для поиска файлов заданного размера в заданном каталоге (имя каталога задаётся пользователем в качестве третьего аргумента командной строки). Диапазон (мин.- мах.) размеров файлов задаётся пользователем в качестве первого и второго аргумента командной строки.

2. Написать скрипт с использованием цикла ***for***, выводящий на консоль размеры и права доступа для всех файлов в заданном каталоге и всех его подкаталогах (имя каталога задается пользователем в качестве первого аргумента командной строки).

3. Написать скрипт для поиска заданной пользователем строки во всех файлах заданного каталога и всех его подкаталогов (строка и имя каталога задаются пользователем в качестве первого и второго аргумента командной строки). На консоль выводятся полный путь и имена файлов, в содержимом которых присутствует заданная строка, и их размер. Если к какому либо каталогу нет доступа, необходимо вывести соответсвующее сообщение и продолжить выполнение.

4. Написать скрипт поиска одинаковых по их содержимому файлов в двух каталогов, например, ***Dir1*** и ***Dir2***. Пользователь задаёт имена ***Dir1*** и ***Dir2*** в качестве первого и второго аргумента командной строки. В результате работы программы файлы, имеющиеся в ***Dir1***, сравниваются с файлами в ***Dir2*** по их содержимому. На экран выводятся число просмотренных файлов и результаты сравнения.

5. Написать скрипт находящий в заданном каталоге и всех его подкаталогах все файлы, владельцем которых является заданный пользователь. Имя владельца и каталог задаются пользователем в качестве первого и второго аргумента командной строки. Скрипт выводит результаты в файл (третий аргумент командной строки) в виде полный путь, имя файла, его размер. На консоль выводится общее число просмотренных файлов.

6. Написать скрипт находящий в заданном каталоге и всех его подкаталогах все файлы заданного размера. Диапазон (мин.- мах.) размеров файлов задаётся пользователем в качестве первого и второго аргумента командной строки. Имя владельца и каталог задаются пользователем в качестве первого и второго аргумента командной строки. Скрипт выводит результаты поиска в файл (четвертый аргумент командной строки) в виде полный путь, имя файла, его размер. На консоль выводится общее число просмотренных файлов.

7. Написать скрипт подсчитавающий суммарный размер файлов в заданном каталоге и всех его подкаталогах (имя каталога задаётся пользователем в качестве аргумента командной строки). Скрипт выводит результаты посдчета в файл (второй аргумент командной строки) в виде католог(полный путь), суммарный размер файлов число просмотренных файлов.

Лабораторная работа №2

РАБОТА С ФАЙЛАМИ И КАТАЛОГАМИ ОС UNIX

Цель работы – изучить основные системные вызовы и функции в ОС UNIX для работы с файлами и каталогами.

Теоретическая часть

Функция ***main: int main(int argc[] , char *argv[ ] [, char *envp[ ]]);***
Данное объявление позволяет удобно передавать аргументы командной строки и переменные окружения. Определение аргументов:

argc - количество аргументов, которые содержатся в ***argv[]*** (всегда больше либо равен 1);
argv - в массиве строки представляют собой параметры из командной строки, введенные пользователем программы. По соглашению, **argv [0] –** это команда, которой была запущена программа, **argv[1]** – первый параметр из командной строки и так далее до argv[argc] – элемент, всегда равный NULL;
envp - это массив строк, которые представляют собой переменные окружения. Массив заканчивается значением ***NULL***.

Для выполнения операций записи и чтения данных в существующем файле его следует открыть при помощи вызова ***open()***.

int open (const char *pathname, int flags, [mode_t mode]);
int fopen (const char *pathname, int flags, [mode_t mode]);

Второй аргумент системного вызова ***open*** - ***flag***s - имеет целочисленный тип и определяет метод доступа. Параметр ***flags*** принимает одно из значений, заданных постоянными в заголовочном файле ***fcnt1.h***. В файле определены три постоянных:

O_RDONLY – открыть файл только для чтения,
O_WRONLY – открыть файл только для записи,
O_RDWR – открыть файл для чтения и записи,
или ***“r”***, ***“w”***, ***“rw”*** для ***fopen().***

Третий параметр ***mode*** устанавливает права доступа к файлу и является необязательным, он используется только вместе с флагом ***O_CREAT***. Пример создания нового файла:

```
# include <sys / types.h>
# include <sys / stat.h>
# include <fcnt1.h>
int Fd1;
FILE *F1;
F1=fopen (“Myfile2.txt”, “w”, 644);
Fd1=open (“Myfile1.txt”, O_CREAT, 644);
```

Системные вызовы ***stat*** и ***fstat*** позволяют процессу определить значения свойств в существующем файле.

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
int stat (const char *pathname, struct stat *buf);
int fstat (int filedes, struct stat *buf);
```

Пример: ***stat("1.exe", &st1);***

Где ***pathname*** – полное имя файла, ***buf*** –структура типа ***stat***. Эта структура после успешного вызова будет содержать связанную с файлом информацию.

Поля структуры stat включает следующие элементы:

```
struct stat {
    dev_t     st_dev;     /* логическое устройство, где находится файл */
    ino_t     st_ino;     /* номер индексного дескриптора */
    mode_t    st_mode;    /* права доступа к файлу */
    nlink_t   st_nlink;   /* количество жестких ссылок на файл */
    uid_t     st_uid;     /* ID пользователя-владельца */
    gid_t     st_gid;     /* ID группы-владельца */
    dev_t     st_rdev;    /* тип устройства */
    off_t     st_size;    /* общий размер в байтах */
    unsigned long st_blksize; /* размер блока ввода-вывода */
    unsigned long st_blocks;  /* число блоков, занимаемых файлом */
    time_t    st_atime;   /* время последнего доступа */
    time_t    st_mtime;   /* время последней модификации */
    time_t    st_ctime;   /* время последнего изменения */
        };
```

Права доступа в ***Linux***. Права доступа к файлам представлены в виде последовательности бит, где каждый бит означает разрешение на запись (***w***), чтение (***r***) или выполнение (***x***). Права доступа записываются для владельца-создателя файла (***owner***); группы, к которой принадлежит владелец–создатель файла (***group***); и всех остальных (***other***). Например, при выводе команды ***dir*** запись типа:

```
-rwx r-x r-w    1.exe
```

означает, что владелец файла ***1.exe*** имеет права на чтение, запись и выполнение, группа имеет права только на чтение и выполнение, все остальные имеют права только на чтение. В восьмеричном виде получится значение ***0754***. В действительности манипулирует файлами не сам пользователь, а запущенный им процесс. Для просмотра прав доступа можно использовать функцию ***stat***.

Для записи прав доступа служит функция ***chmod***:

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod(const char *pathname, mode_t mode);
```

Пример: ***chmod("1.exe", 0777);***

Каталоги в ОС ***Linux*** –это особые файлы. Для открытия или закрытия каталогов существуют вызовы:

```
#include <dirent.h>
DIR *opendir (const char *dirname);
```

int closedir(DIR *dirptr);

Для работы с каталогами существуют системные вызовы:

int mkdir (const char *pathname, mode_t mode) – создание нового каталога,

int rmdir(const char *pathname) – удаление каталога. Первый параметр – имя создаваемого каталога, второй – права доступа:

retval=mkdir("/home/s1/t12/alex",0777);

retval=rmdir("/home/s1/t12/alex");

Заметим, что вызов ***rmdir("/home/s1/t12/alex")*** будет успешен, только если удаляемый каталог пуст, т.е. содержит записи "точка" (.) и "двойная точка" (..).

Для чтения записей каталога существует вызов:

struct dirent *readdir(DIR *dirptr);

Структура dirent такова: ***struct dirent {***

```
long          d_ino;
off_t         d_off;
unsigned short d_reclen;
char          d_name [1];
    };
```

Поле ***d_ino*** - это число, которое уникально для каждого файла в файловой системе. Значением поля ***d_off*** служит смещение данного элемента в реальном каталоге. Поле ***d_name*** есть начало массива символов, задающего имя элемента каталога. Данное имя ограничено нулевым байтом и может содержать не более ***MAXNAMLEN*** символов. Тем самым описываемая структура имеет переменную длину, хранящуюся в поле ***d_reclen***.

Пример вызова:

```
DIR *dp;
struct dirent *d;
 d=readdir(dp);
```

При первом вызове функция ***readdir*** в структуру ***dirent*** будет считана первая запись каталога. После прочтения всего каталога в результате последующих вызовов ***readdir*** будет возвращено значение ***NULL***. Для возврата указателя в начало каталога на первую запись существует вызов:

void rewindir(DIR *dirptr);

Чтобы получить имя текущего рабочего каталога существует функция:

char *getcwd(char *name, size_t size);

Время в ***Linux*** отсчитывается в секундах, прошедшее с начала этой эпохи (***00:00:00 UTC, 1 Января 1970 года***). Для получения системного времени можно использовать следующие функции:

```
#include <sys/time.h>
time_t   time (time_t *tt);
int   gettimeofday(struct timeval *tv, struct timezone *tz);

   struct timeval {
                long    tv_sec;       /* секунды */
```

```
    long    tv_usec;        /* микросекунды */
};
```

Порядок выполнения работы

1. Изучить теоретическую часть лабораторной работы.
2. Написать программу вывода сообщения на экран.
3. Написать программу ввода символов с клавиатуры и записи их в файл (в качестве аргумента при запуске программы вводится имя файла). Для чтения или записи файла использовать ***только*** функции посимвольного ввода-вывода ***getc(),putc()***, ***fgetc(),fputc()***. Предусмотреть выход после ввода определённого символа (например: ***ctrl-F***). Предусмотреть контроль ошибок открытия/закрытия/чтения файла.
4. Написать программу вывода содержимого текстового файла на экран (в качестве аргумента при запуске программы передаётся имя файла, второй аргумент (***N***) устанавливает вывод по группам строк (по ***N*** –строк) или сплошным текстом (***N***=0)). Для вывода очередной группы строк необходимо ожидать нажатия пользователем любой клавиши. Для чтения или записи файла использовать ***только*** функции посимвольного ввода-вывода ***getc(),putc()***, ***fgetc(),fputc()***. Предусмотреть контроль ошибок открытия/закрытия/чтения/записи файла.
5. Написать программу копирования одного файла в другой. В качестве параметров при вызове программы передаются имена первого и второго файлов. Для чтения или записи файла использовать ***только*** функции посимвольного ввода-вывода ***getc(),putc()***, ***fgetc(),fputc()***. Предусмотреть копирование прав доступа к файлу и контроль ошибок открытия/закрытия/чтения/записи файла.
6. Написать программу вывода на экран содержимого текущего и корневого каталогов. Предусмотреть контроль ошибок открытия/закрытия/чтения каталога.

Варианты индивидуальных заданий

1. Отсортировать в заданном каталоге (аргумент 1 командной строки) и во всех его подкаталогах файлы по следующим критериям (аргумент 2 командной строки, задаётся в виде целого числа):1 – по размеру файла, 2 – по имени файла. Записать отсортированные файлы в новый каталог (аргумент 3 командной строки).

2. Найти в заданном каталоге (аргумент 1 командной строки) и всех его подкаталогах заданный файл (аргумент 2 командной строки). Вывести на консоль полный путь к файлу имя файла, его размер, дату создания, права

доступа, номер индексного дескриптора. Вывести также общее количество просмотренных каталогов и файлов.

3. Для заданного каталога (аргумент 1 командной строки) и всех его подкаталогов вывести в заданный файл (аргумент 2 командной строки) и на консоль имена файлов, их размер и дату создания, удовлетворяющих заданным условиям: 1 – размер файла находится в заданных пределах от ***N1*** до ***N2*** (***N1,N2*** задаются в аргументах командной строки), 2 – дата создания находится в заданных пределах от ***M1*** до ***M2*** (***M1***,***M2*** задаются в аргументах командной строки).

4. Найти совпадающие по содержимому файлы в двух заданных каталогах (аргументы 1 и 2 командной строки) и всех их подкаталогах. Вывести на консоль и в файл (аргумент 3 командной строки) их имя, размер, дату создания, права доступа, номер индексного дескриптора.

5. Подсчитать суммарный размер файлов в заданном каталоге (аргумент 1 командной строки) и для каждого его подкаталога отдельно. Вывести на консоль и в файл (аргумент 2 командной строки) название подкаталога, количество файлов в нём, суммарный размер файлов, имя файла с наибольшим размером.

6. Написать программу, находящую в заданном каталоге и всех его подкаталогах все файлы, заданного размера. Имя каталога задаётся пользователем в качестве первого аргумента командной строки. Диапазон (min.- max.) размеров файлов задаётся пользователем в качестве второго и третьего аргументов командной строки. Программа выводит результаты поиска в файл (четвертый аргумент командной строки) в виде полный путь, имя файла, его размер. На консоль выводится общее число просмотренных файлов.

Лабораторная работа №3

ПРОЦЕССЫ В ОС LINUX

Цель работы – изучение вопросов порождения и взаимодействия процессов в ОС LINUX.

Теоретическая часть

В ОС Linux для создания процессов используется системный вызов ***fork():***

```
#include <sys/types.h>
#include <unistd.h>
pid_t  fork (void);
```

В результате успешного вызова ***fork()*** ядро создаёт новый процесс, который является почти точной копией вызывающего процесса. Другими словами, новый процесс выполняет копию той же программы, что и создавший его процесс, при этом все его объекты данных имеют те же самые значения, что и в вызывающем процессе. Созданный процесс называется ***дочерним процессом***, а процесс, осуществивший вызов ***fork()***, называется ***родительским***. После вызова родительский процесс и его вновь созданный потомок выполняются одновременно, при этом оба процесса продолжают выполнение с оператора, который следует сразу же за вызовом ***fork()***. Процессы выполняются в разных адресных пространствах, поэтому прямой доступ к переменным одного процесса из другого процесса невозможен.

Следующая короткая программа более наглядно показывает работу вызова ***fork()*** и использование процесса:

```
#include <stdio.h>
#include <unistd.h>
int main ()
{
   pid_t pid;      /* идентификатор процесса */
   printf ("Пока всего один процесс\n");
   pid = fork ();    /* Создание нового процесса */
  printf ("Уже два процесса\n");
  if (pid = = 0){
            printf ("Это Дочерний процесс его pid=%d\n", getpid());
             printf ("А pid его Родительского процесса=%d\n", getppid());
          }
   else if (pid > 0)
            printf ("Это Родительский процесс pid=%d\n", getpid());
   else
     printf ("Ошибка вызова fork, потомок не создан\n");
}
```

Для корректного завершения дочернего процесса в родительском процессе необходимо использовать функцию ***wait()*** или ***waitpid()***:

pid_t wait(int *status);

pid_t waitpid(pid_t pid, int *status, int options);

Функция ***wait*** приостанавливает выполнение родительского процесса до тех пор, пока дочерний процесс не прекратит выполнение или до появления сигнала, который либо завершает текущий процесс, либо требует вызвать функцию-обработчик. Если дочерний процесс к моменту вызова функции уже завершился (так называемый ***«зомби»***), то функция немедленно возвращается. Системные ресурсы, связанные с дочерним процессом, освобождаются.

Функция ***waitpid ()*** приостанавливает выполнение родительского процесса до тех пор, пока дочерний процесс, указанный в параметре ***pid***, не завершит выполнение, или пока не появится сигнал, который либо завершает родительский процесс, либо требует вызвать функцию-обработчик. Если указанный дочерний процесс к моменту вызова функции уже завершился (так называемый ***«зомби»***), то функция немедленно возвращается. Системные ресурсы, связанные с дочерним процессом, освобождаются. Параметр ***pid*** может принимать несколько значений:

pid< -1 означает, что нужно ждать любого дочернего процесса, чей идентификатор группы процессов равен абсолютному значению ***pid***.

pid= -1 означает ожидать любого дочернего процесса; функция **wait** ведет себя точно так же.

pid = 0 означает ожидать любого дочернего процесса, чей идентификатор группы процессов равен таковому у текущего процесса.

pid> 0 означает ожидать дочернего процесса, чем идентификатор равен ***pid***.

Значение ***options*** создается путем битовой операции ***ИЛИ*** над следующими константами:

WNOHANG - означает вернуть управление немедленно, если ни один дочерний процесс не завершил выполнение.

WUNTRACED - означает возвращать управление также для остановленных дочерних процессов, о чьем статусе еще не было сообщено.

Каждый дочерний процесс при завершении работы посылает своему процессу-родителю специальный сигнал ***SIGCHLD***, на который у всех процессов по умолчанию установлена реакция "игнорировать сигнал". Наличие такого сигнала совместно с системным вызовом ***waitpid()*** позволяет организовать асинхронный сбор информации о статусе завершившихся порожденных процессов процессом-родителем.

Для перегрузки исполняемой программы можно использовать функции семейства ***exec***. Основное отличие между разными функциями в семействе состоит в способе передачи параметров.

int execl(char *pathname, char *arg0, arg1, ..., argn, NULL);
int execle(char *pathname, char *arg0, arg1, ..., argn, NULL, char **envp);
int execlp(char *pathname, char *arg0, arg1, ..., argn, NULL);
int execlpe(char *pathname, char *arg0, arg1, ..., argn, NULL, char **envp);

int execv(char *pathname, char *argv[]);
int execve(char *pathname, char *argv[],char **envp);
int execvp(char *pathname, char *argv[]);
int execvpe(char *pathname, char *argv[],char **envp);

Основное отличие между разными функциями в семействе состоит в способе передачи параметров. Как видно из рис. 1, все эти функции выполняют один системный вызов ***execve.***

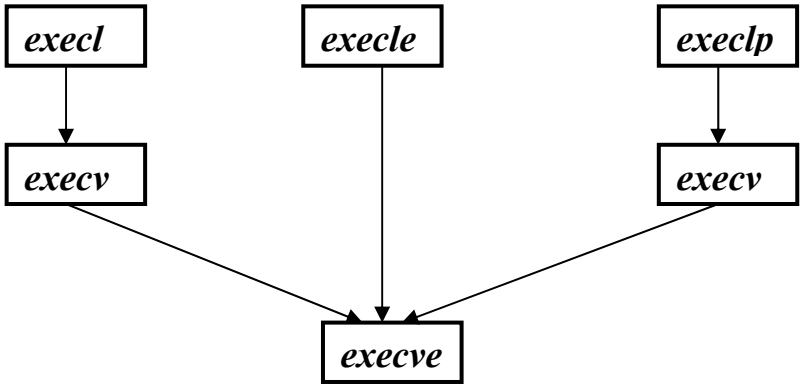


Рис. 1. Дерево семейства вызовов ***exec***

Порядок выполнения работы

1. Изучить теоретическую часть лабораторной работы.

2. Написать программу, создающую два дочерних процесса с использованием двух вызовов ***fork()***. Родительский и два дочерних процесса должны выводить на экран свой ***pid*** и ***pid*** родительского процесса и текущее время в формате: ***часы: минуты: секунды: миллисекунды***. Используя вызов ***system(),*** выполнить команду ***ps -x*** в родительском процессе. Найти свои процессы в списке запущенных процессов.

Варианты индивидуальных заданий

1. Написать программу нахождения массива ***K*** последовательных значений функции ***y[i]=sin(2*PI*i/N) (i=0,1,2…K-1)*** с использованием ряда Тейлора. Пользователь задаёт значения ***K***, ***N*** и количество ***n*** членов ряда Тейлора. Для расчета каждого члена ряда Тейлора запускается отдельный поток. Каждый поток выводит на экран свой ***id*** и рассчитанное значение ряда. Головной процесс суммирует все члены ряда Тейлора, и полученное значение ***y[i]*** записывает в файл.
2. Написать программу синхронизации двух каталогов, например, ***Dir1*** и ***Dir2***. Пользователь задаёт имена ***Dir1*** и ***Dir2***. В результате работы программы файлы, имеющиеся в ***Dir1***, но отсутствующие в ***Dir2***, должны скопироваться в ***Dir2*** вместе с правами доступа. Процедуры копирования

должны запускаться в отдельном процессе для каждого копируемого файла. Каждый процесс выводит на экран свой ***pid,*** имя копируемого файла и число скопированных байт. Число одновременно работающих процессов не должно превышать ***N*** (вводится пользователем).

3. Написать программу поиска одинаковых по их содержимому файлов в двух каталогов, например, ***Dir1*** и ***Dir2***. Пользователь задаёт имена ***Dir1*** и ***Dir2***. В результате работы программы файлы, имеющиеся в ***Dir1***, сравниваются с файлами в ***Dir2*** по их содержимому. Процедуры сравнения должны запускаться в отдельном процессе для каждой пары сравниваемых файлов. Каждый процесс выводит на экран свой ***pid***, имя файла, общее число просмотренных байт и результаты сравнения. Число одновременно работающих процессов не должно превышать ***N*** (вводится пользователем).
4. Написать программу поиска заданной пользователем комбинации из ***m*** байт (***m < 255***) во всех файлах текущего каталога. Пользователь задаёт имя каталога. Главный процесс открывает каталог и запускает для каждого файла каталога отдельный процесс поиска заданной комбинации из ***m*** байт. Каждый процесс выводит на экран свой ***pid***, имя файла, общее число просмотренных байт и результаты поиска. Число одновременно работающих процессов не должно превышать ***N*** (вводится пользователем).
5. Разработать программу «***интерпретатор команд***», которая воспринимает команды, вводимые с клавиатуры, (например, ***ls -l /bin/bash***) и осуществляет их корректное выполнение. Для этого каждая вводимая команда должна выполняться в отдельном процессе с использованием вызова ***exec()***. Предусмотреть контроль ошибок.
6. Создать дерево процессов по индивидуальному заданию. Каждый процесс постоянно, через время ***t*** , выводит на экран следующую информацию:
   номер процесса/потока pid ppid текущее время (мсек).
   Время ***t =((номер процесса/потока по дереву)*200*** (мсек)***.***

Лабораторная работа №4

ИСПОЛЬЗОВАНИЕ СИГНАЛОВ В ОС LINUX

Цель работы – изучение механизма взаимодействия процессов с использованием сигналов.

Теоретическая часть

Сигналы не могут непосредственно переносить информацию, что ограничивает их применимость в качестве общего механизма межпроцессного взаимодействия. Тем не менее, каждому типу сигналов присвоено мнемоническое имя (например, SIGINT), которое указывает, для чего обычно используется сигнал этого типа. Имена сигналов определены в стандартном заголовочном файле <signal.h> при помощи директивы препроцессора #define. Как и следовало ожидать, эти имена соответствуют небольшим положительным целым числам. С точки зрения пользователя получение процессом сигнала выглядит как возникновение прерывания. Процесс прерывает исполнение, и управление передается функции-обработчику сигнала. По окончании обработки сигнала процесс может возобновить регулярное исполнение. Типы сигналов принято задавать специальными символьными константами. Системный вызов ***kill()*** предназначен для передачи сигнала одному или нескольким специфицированным процессам в рамках полномочий пользователя.

#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int signal);

Послать сигнал (не имея полномочий суперпользователя) можно только процессу, у которого эффективный идентификатор пользователя совпадает с эффективным идентификатором пользователя для процесса, посылающего сигнал. Аргумент ***pid*** указывает процесс, которому посылается сигнал, а аргумент ***sig*** – какой сигнал посылается. В зависимости от значения аргументов:

pid > 0 сигнал посылается процессу с идентификатором ***pid***;

pid=0 сигнал посылается всем процессам в группе, к которой принадлежит посылающий процесс;

pid=-1 и посылающий процесс не является процессом суперпользователя, то сигнал посылается всем процессам в системе, для которых идентификатор пользователя совпадает с эффективным идентификатором пользователя процесса, посылающего сигнал.

pid = -1 и посылающий процесс является процессом суперпользователя, то сигнал посылается всем процессам в системе, за исключением системных процессов (обычно всем, кроме процессов ***с pid = 0*** и ***pid = 1***).

pid < 0, но не ***–1***, то сигнал посылается всем процессам из группы, идентификатор которой равен абсолютному значению аргумента ***pid*** (если позволяют привилегии).

если ***sig = 0***, то производится проверка на ошибку, а сигнал не посылается. Это можно использовать для проверки правильности аргумента ***pid*** (есть ли в системе процесс или группа процессов с соответствующим идентификатором).

Системные вызовы для установки собственного обработчика сигналов:

```
#include <signal.h>
void (*signal (int sig,  void (*handler) (int)))(int);
int sigaction(int sig, const struct sigaction *act, struct sigaction *oldact);
```

Структура ***sigaction*** имеет следующий формат:

```
struct sigaction {
   void (*sa_handler)(int);
   void (*sa_sigaction)(int, siginfo_t *, void *);
   sigset_t sa_mask;
   int sa_flags;
   void (*sa_restorer)(void);
```

Системный вызов ***signal*** служит для изменения реакции процесса на какой-либо сигнал. Параметр ***sig*** – это номер сигнала, обработку которого предстоит изменить. Параметр handler описывает новый способ обработки сигнала – это может быть указатель на пользовательскую функцию-обработчик сигнала, специальное значение ***SIG_DFL*** (восстановить реакцию процесса на сигнал ***sig*** по умолчанию) или специальное значение ***SIG_IGN*** (игнорировать поступивший сигнал ***sig***). Системный вызов возвращает указатель на старый способ обработки сигнала, значение которого можно использовать для восстановления старого способа в случае необходимости.

Пример пользовательской обработки сигнала ***SIGUSR1***.

```
void *my_handler(int nsig) {   код  функции-обработчика сигнала }
int main() {
   (void) signal(SIGUSR1, my_handler); }
```

Системный вызов ***sigaction*** используется для изменения действий процесса при получении соответствующего сигнала. Параметр ***sig*** задает номер сигнала и может быть равен любому номеру. Если параметр ***act*** не равен нулю, то новое действие, связянное с сигналом ***sig***, устанавливается соответственно ***act***. Если ***oldact*** не равен нулю, то предыдущее действие записывается в ***oldact***.

Большинство типов сигналов ***UNIX*** предназначены для использования ядром, хотя есть несколько сигналов, которые посылаются от процесса к процессу:

SIGALRM – сигнал таймера (***alarm clock***). Посылается процессу ядром при срабатывании таймера. Каждый процесс может устанавливать не менее трех таймеров. Первый из них измеряет прошедшее реальное время. Этот таймер устанавливается самим процессом при помощи системного вызова ***alarm()***;

SIGCHLD – сигнал останова или завершения дочернего процесса (***child process terminated or stopped***). Если дочерний процесс останавливается или

завершается, то ядро сообщит об этом родительскому процессу, послав ему данный сигнал. По умолчанию родительский процесс игнорирует этот сигнал, поэтому, если в родительском процессе необходимо получать сведения о завершении дочерних процессов, то нужно перехватывать этот сигнал;

SIGHUP – сигнал освобождения линии (***hangup signal***). Посылается ядром всем процессам, подключенным к управляющему терминалу (***control terminal***) при отключении терминала. Он также посылается всем членам сеанса, если завершает работу лидер сеанса (обычно процесс командного интерпретатора), связанного с управляющим терминалом;

SIGINT – сигнал прерывания программы (***interrupt***). Посылается ядром всем процессам сеанса, связанного с терминалом, когда пользователь нажимает клавишу прерывания. Это также обычный способ остановки выполняющейся программы;

SIGKILL – сигнал уничтожения процесса (***kill***). Это довольно специфический сигнал, который посылается от одного процесса к другому и приводит к немедленному прекращению работы получающего сигнал процесса;

SIGPIPE – сигнал о попытке записи в канал или сокет, для которых принимающий процесс уже завершил;

SIGPOLL – сигнал о возникновении одного из опрашиваемых событий (***pollable event***). Этот сигнал генерируется ядром, когда некоторый открытый дескриптор файла становится готовым для ввода или вывода;

SIGPROF – сигнал профилирующего таймера (***profiling time expired***). Как было упомянуто для сигнала ***SIGALRM***, любой процесс может установить не менее трех таймеров. Второй из этих таймеров может использоваться для измерения времени выполнения процесса в пользовательском и системном режимах. Этот сигнал генерируется, когда истекает время, установленное в этом таймере, и поэтому может быть использован средством профилирования программы;

SIGQUIT – сигнал о выходе (***quit***). Очень похожий на сигнал ***SIGINT***, этот сигнал посылается ядром, когда пользователь нажимает клавишу выхода используемого терминала. В отличие от ***SIGINT***, этот сигнал приводит к аварийному завершению и сбросу образа памяти;

SIGSTOP – сигнал останова (***stop executing***). Это сигнал управления заданиями, который останавливает процесс. Его, как и сигнал ***SIGKILL***, нельзя проигнорировать или перехватить;

SIGTERM – программный сигнал завершения (***software termination signal***). Программист может использовать этот сигнал для того, чтобы дать процессу время для «наведения порядка», прежде чем посылать ему сигнал ***SIGKILL***;

SIGTRAP – сигнал трассировочного прерывания (***trace trap***). Это особый сигнал, который в сочетании с системным вызовом ptrace используется отладчиками, такими как ***sdb, adb, gdb***;

SIGTSTP – терминальный сигнал остановки (***terminal stop signal***). Он формируется при нажатии специальной клавиши останова;

SIGTTIN – сигнал о попытке ввода с терминала фоновым процессом (***background process attempting read***). Если процесс выполняется в фоновом режиме и пытается выполнить чтение с управляющего терминала, то ему посылается этот сигнал. Действие сигнала по умолчанию – остановка процесса;

SIGTTOU – сигнал о попытке вывода на терминал фоновым процессом (***background process attempting write***). Аналогичен сигналу ***SIGTTIN***, но генерируется, если фоновый процесс пытается выполнить запись в управляющий терминал. Действие сигнала по умолчанию – остановка процесса;

SIGURG – сигнал о поступлении в буфер сокета срочных данных (***high bandwidth data is available at a socket***). Он сообщает процессу, что по сетевому соединению получены срочные внеочередные данные;

SIGUSR1 и ***SIGUSR2*** – пользовательские сигналы (***user defined signals 1 and 2***). Так же, как и сигнал ***SIGTERM***, эти сигналы никогда не посылаются ядром и могут использоваться для любых целей по выбору пользователя;

SIGVTALRM – сигнал виртуального таймера (***virtual timer expired***). Третий таймер можно установить так, чтобы он измерял время, которое процесс выполняет в пользовательском режиме.

Наборы сигналов определяются при помощи типа ***sigset_t***, который определен в заголовочном файле ***<signal.h>***. Выбрать определенные сигналы можно, начав либо с полного набора сигналов и удалив ненужные сигналы, либо с пустого набора, включив в него нужные. Инициализация пустого и полного набора сигналов выполняется при помощи процедур ***sigemptyset*** и ***sigfillset*** соответственно. После инициализации с наборами сигналов можно оперировать при помощи процедур ***sigaddset*** и ***sigdelset***, соответственно добавляющих и удаляющих указанные вами сигналы.

Описание данных процедур:

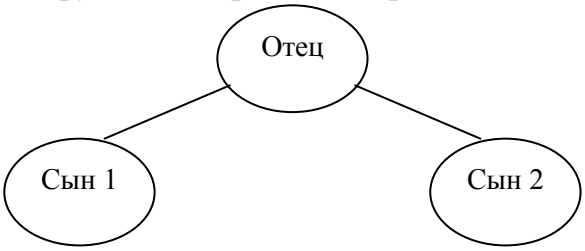
```
#include <signal.h>
/* Инициализация*/
int sigemptyset (sigset_t *set);
int sigfillset (sigset_t *set);
/*Добавление и удаление сигналов*/
int sigaddset (sigset_t *set, int signo);
int sigdelset (sigset_t *set, int signo);
```

Процедуры ***sigemptyset*** и ***sigfillset*** имеют единственный параметр – указатель на переменную типа ***sigset_t***. Вызов ***sigemptyset*** инициализирует набор ***set***, исключив из него все сигналы. И наоборот, вызов ***sigfillset*** инициализирует набор, на который указывает ***set***, включив в него все сигналы. Приложения должны вызывать ***sigemptyset*** или ***sigfillset*** хотя бы один раз для каждой переменной типа ***sigset_t***.

Процедуры ***sigaddset*** и ***sigdelset*** принимают в качестве параметров указатель на инициализированный набор сигналов и номер сигнала, который должен быть добавлен или удален. Второй параметр, ***signo***, может быть символическим именем константы, таким как ***SIGINT***, или настоящим номером сигнала, но в последнем случае программа окажется системно-зависимой.

Порядок выполнения работы

1. Изучить теоретическую часть лабораторной работы.
2. Организовать функционирование процессов следующей структуры:



Процессы определяют свою работу выводом сообщений вида :

N pid ppid текущее время (мсек) (***N*** – текущий номер сообщения) на экран. “Отец” одновременно, посылает сигнал ***SIGUSR1*** “сыновьям”. “Сыновья” получив данный сигнал, посылают в ответ “Отцу” сигнал ***SIGUSR2***. “Отец” получив сигнал ***SIGUSR2***, через время ***t=100*** мсек одновременно, посылает сигнал ***SIGUSR1*** “сыновьям”. И так далее… Написать функции-обработчики сигналов, которые при получении сигнала выводят сообщение о получении сигнала на экран. При получении/посылке сигнала они выводят соответствующее сообщение:

N pid ppid текущее время (мсек) ***сын такой-то get/put SIGUSRm***.

Предусмотреть механизм для определения “Отцом”, от кого из “Сыновей” получен сигнал.

Варианты индивидуальных заданий

.

1. То же что и в пункте 2, но для процессов написать функции-обработчики сигналов от клавиатуры, которые запрашивали бы подтверждение на завершение работы при получении такого сигнала.

2. Организовать функционирование процессов следующей структуры: ***1-2-3-4*** (процесс 1 создаёт процесс 2, процесс 2 создаёт процесс 3, процесс 3 создаёт процесс 4). Далее организовать передачу/приём сигналов в следующей последовательности: ***1-2 (SIGUSR1), 2-3 (SIGUSR1), 3-4 (SIGUSR1), 4-1 (SIGUSR2), 1-2 (SIGUSR2), 2-3 (SIGUSR2), 3-4 (SIGUSR2), 4-1 (SIGUSR1)*** и так далее. Каждый процесс выдерживает паузу ***t=100*** мсек между приёмом и посылкой сигнала и выводит на консоль следующую информацию:

N pid ppid текущее время (мсек) ***процесс_такой-то get/put SIGUSRm***.

3. Организовать функционирование процессов следующей структуры: ***1-(2,3),3-4*** (процесс 1 создаёт процессы 2 и 3, процесс 3 создаёт процесс 4).. Далее организовать передачу/приём сигналов в следующей последовательности: ***1-(2,3) (SIGUSR1), 3-4 (SIGUSR1), 4-1 (SIGUSR2), 1-(2,3) (SIGUSR2), 3-4 (SIGUSR2), 4-1 (SIGUSR1)*** и так далее. Каждый процесс выдерживает паузу ***t=100*** мсек между приёмом и посылкой сигнала и выводит на консоль следующую информацию:

N pid ppid текущее время (мсек) ***процесс_такой-то get/put SIGUSRm***.

Лабораторная работа №5

ИСПОЛЬЗОВАНИЕ КАНАЛОВ В ОС LINUX

Цель работы - изучение механизма взаимодействия процессов с использованием каналов.

Теоретическая часть

Каналы являются одной из самых сильных и характерных особенностей ОС UNIX, доступных даже с уровня командного интерпретатора. Каналы. Программный канал – это файл особого типа (***FIFO***: «первым вошел – первым вышел»). Процессы могут записывать и считывать данные из канала как из обычного файла. Если канал заполнен, процесс записи в канал останавливается до тех пор, пока не появится свободное место, чтобы снова заполнить его данными. С другой стороны, если канал пуст, то читающий процесс останавливается до тех пор, пока пишущий процесс не запишет данные в этот канал. В отличие от обычного файла здесь нет возможности позиционирования по файлу с использованием указателя.

В ОС Linux различают два вида программных каналов:

- Именованный программный канал. Именованный программный канал может служить для общения и синхронизации произвольных процессов, знающих имя данного программного канала и имеющих соответствующие права доступа. Для создания используется вызов:

int mkfifo(const char *filename, mode_t mode);

- Неименованный программный канал. Неименованным программным каналом могут пользоваться только создавший его процесс и его потомки. Для создания используется вызов:

int pipe(int fd[2]);

Переменная ***fd*** является массивом из двух целых чисел, который будет содержать дескрипторы файлов, обозначающие канал. После успешного вызова ***fd [0]*** будет открыт для чтения из канала, а ***fd [1]*** – для записи в канал. В случае неудачи вызов ***pipe*** вернет значение -1. Это может произойти, если в момент вызова произойдет превышение максимально возможного числа дескрипторов файлов, которые могут быть одновременно открыты процессами пользователя (в этом случае переменная ***errno*** будет содержать значение ***EMFILE***), или если произойдет переполнение таблицы открытых файлов в ядре (в этом случае переменная ***errno*** будет содержать значение ***ENFILE***).

После создания канала с ним можно работать просто при помощи вызовов ***read*** и ***write***. Следующий пример демонстрирует это: он создает канал, записывает в него три сообщения, а затем считывает их из канала:

```
#include <unistd.h>
#include <stdio.h>
char *msg1 = "hello, world #1";
main ()
{
  char inbuf [15];
  int p [2], j;
  if (pipe (p) == -1) {
    perror ("Ошибка вызова pipe");
    exit (1);
  }
   write (p[1], msg1, 15);   /*Запись в канал*/
   read (p[0], inbuf, MSGSIZE);  /*Чтение из канала*/
   printf ("%s\n", inbuf);
   exit (0);
}
```

Размеры блоков при записи в канал и чтении из него необязательно должны быть одинаковыми, хотя в нашем примере это и было так. Можно, например, писать в канал блоками по 512 байт, а затем считывать из него по 1 символу, так же как и в случае обычного файла. Тем не менее, использование блоков фиксированного размера дает определенные преимущества.

На рис. 3 показано, как канал соединяет два процесса. Здесь видно, что и в родительском, и в дочернем процессах открыто по два дескриптора файла, позволяя выполнять запись в канал и чтение из него. Поэтому любой из процессов может выполнять запись в файл с дескриптором ***p[1]*** и чтение из файла с дескриптором ***p[0]***. Это создает определенную проблему – каналы предназначены для использования в качестве однонаправленного средства связи. Если оба процесса будут одновременно выполнять чтение из канала и запись в него, то это приведет к путанице.

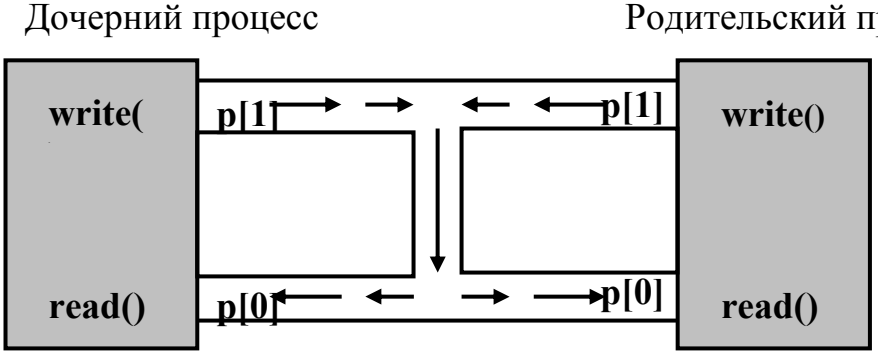


Рис. 3. Работы с каналами.

Чтобы избежать этого, каждый процесс должен выполнять либо чтение из канала, либо запись в него и закрывать дескриптор файла, как только он стал не нужен. Фактически программа должна выполнять это для того, чтобы избежать неприятностей, если посылающий данные процесс закроет дескриптор файла, открытого на запись.

В конечном итоге получится однонаправленный поток данных от дочернего процесса к родительскому. Эта упрощенная ситуация показана на рис. 5.3.

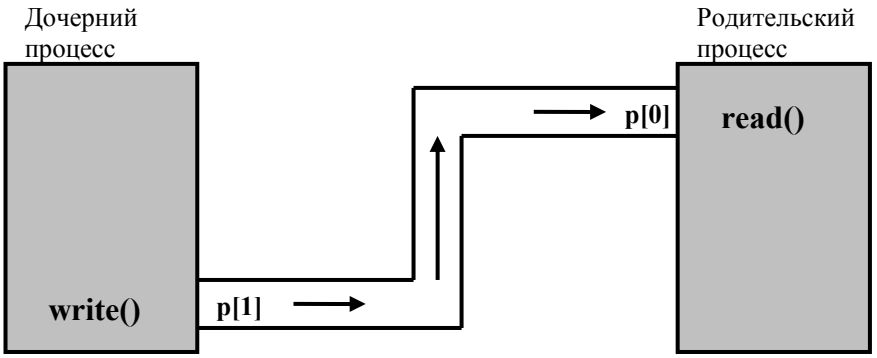


Рис. 4. Третий пример работы с каналами

Для простых приложений применение неблокирующих операций чтения и записи работает прекрасно. Для работы с множеством каналов одновременно существует другое решение, которое заключается в использовании системного вызова ***select***.

Возможна ситуация, когда родительский процесс выступает в качестве серверного процесса и может иметь произвольное число связанных с ним клиентских (дочерних) процессов, как показано на рис. 5.

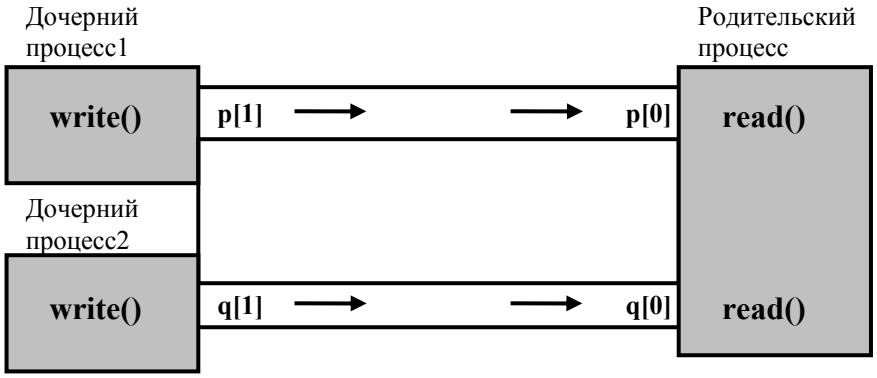


Рис. 5. Клиент/сервер с использованием каналов

В этом случае серверный процесс должен как-то справляться с ситуацией, когда одновременно в нескольких каналах может находиться информация, ожидающая обработки. Кроме того, если ни в одном из каналов нет ожидающих данных, то может иметь смысл приостановить работу серверного процесса до их появления, а не опрашивать постоянно каналы. Если информация поступает более чем по одному каналу, то серверный процесс должен знать обо всех таких каналах для того, чтобы работать с ними в правильном порядке (например, согласно их приоритету).

Это можно сделать при помощи системного вызова ***select*** (существует также аналогичный вызов ***poll***). Системный вызов ***select*** используется не только для каналов, но и для обычных файлов, терминальных устройств, именованных каналов и сокетов. Системный вызов ***select*** показывает, какие дескрипторы файлов из заданных наборов готовы для чтения, записи или ожидают обработки ошибок. Иногда серверный процесс не должен совсем прекращать работу, даже если не происходит никаких событий, поэтому в вызове ***select*** также можно задать предельное время ожидания. Описание данного вызова:

#include <sys/time.h>

int select (int nfds, fd_set *readfds, fd_set *writefds, fd_set *errorfds, struct timeval *timeout);

Первый параметр ***nfds*** задает число дескрипторов файлов, которые могут представлять интерес для сервера. Программист может определять это значение самостоятельно или воспользоваться постоянной ***FD_SETSIZE***, которая определена в файле ***<sys/time.h>***. Значение постоянной равно максимальному числу дескрипторов файлов, которые могут быть использованы вызовом ***select***.

Второй, третий и четвертый параметры вызова являются указателями на битовые маски, в которых каждый бит соответствует дескриптору файла. Если бит включен, то это обозначает интерес к соответствующему дескриптору файла. Набор ***readfds*** определяет дескрипторы, для которых сервер ожидает возможности чтения; набор ***writefds*** – дескрипторы, для которых сервер ожидает возможности выполнить запись; набор ***errorfds*** – дескрипторы, для которых сервер ожидает появление ошибки или исключительной ситуации. Так как работа с битами довольно неприятна и приводит к немобильности программ, существуют абстрактный тип данных ***fd_set***, а также макросы или функции для работы с объектами этого типа:

#include <sys/time.h>

/*Инициализация битовой маски, на которую указывает fdset*/

void FD_ZERO (fd_set *fdset);

/*Установка бита fd в маске, на которую указывает fdset*/

void FD_SET (int fd, fd_set *fdset);

/*Установлен ли бит fd в маске, на которую указывает fdset?*/

int FD_ISSET (int fd, fd_set *fdset);

/*Сбросить бит fd в маске, на которую указывает fdset*/

void FD_GLR (int fd, fd_set *fdset);

Следующий пример демонстрирует, как отслеживать состояние двух открытых дескрипторов файлов:

```
#include <sys/time.h>
#include <sys/types.h>
#include <fcntl.h>
...
int fd1, fd2;
fd_set readset;
fd1 = open (“file1”, O_RDONLY);
fd2 = open (“file2”, O_RDONLY);
FD_ZERO (& readset);
FD_SET (fd1, &readset);
FD_SET (fd2, &readset);
switch (select (5, &readset, NULL, NULL, NULL))
{   /*Обработка ввода*/  }
```

Пятый параметр вызова ***select*** является указателем на следующую структуру timeval:

```
#include <sys/time.h>
struct timeval {
  long tv_sec;          /*Секунды*/
  long tv_usec;         /*и микросекунды*/
};
```

Если указатель является нулевым, как в этом примере, то вызов ***select*** будет заблокирован, пока не произойдет “интересующее” процесс событие. Если в этой структуре задано нулевое время, то вызов завершится немедленно. Если структура содержит ненулевое значение, то возврат из вызова произойдет через заданное время, когда файловые дескрипторы неактивны.

Возвращаемое вызовом ***select*** значение равно -1 в случае ошибки, нулю – после истечения временного интервала или целому числу, равному числу «интересующих» программу дескрипторов файлов.

Порядок выполнения работы

1. Изучить теоретическую часть лабораторной работы.

2. Написать программу, создающую дочерний процесс. Родительский процесс создаёт семафор (***сем1***) и общий файл. Дочерний процесс записывает в файл по одной строке всего ***100*** строк вида: ***номер_строки pid_процесса текущее_время*** (мсек). Родительский процесс читает из файла строки и выводит их на экран в следующем виде: ***pid строка_прочитанная_из_файла***. Семафор ***сем1*** используется процессами для разрешения, кому из процессов получить доступ к файлу.

3. Написать программу, создающую дочерний процесс. Родительский процесс создаёт неименованный канал. Дочерний процесс записывает в канал

100 строк вида: ***номер_строки pid_процесса текущее_время*** (мсек). Родительский процесс читает из канала строки и выводит их на экран в следующем виде: ***pid строка прочитанная из файла***.

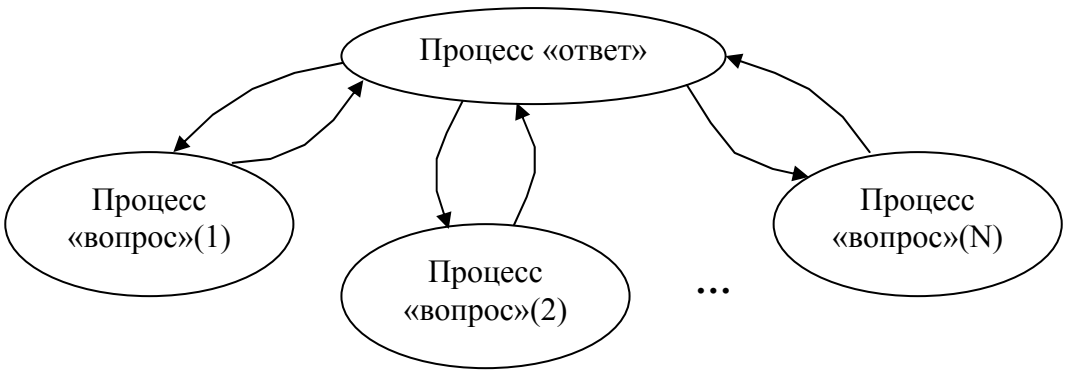
Варианты индивидуальных заданий

1. Создать два дочерних процесса. Родительский процесс создаёт семафор (***сем1***) и общий файл отображенный в память. Оба дочерних процесса непрерывно записывают в файл по ***100*** строк вида: ***номер_строки pid_процесса текущее_время*** (мсек). Всего процессы должны записать ***1000*** строк. Семафор ***сем1*** используется процессами для разрешения кому из процессов получить доступ к файлу. Родительский процесс читает из файла по ***75*** строк и выводит их на экран. Дочерние процессы начинают операции с файлом после получения сигнала ***SIGUSR1*** от родительского процесса.

2. Создать два дочерних процесса. Родительский процесс создаёт семафоры (***сем1***), (***сем2***) и ***2*** неименованных канала (***кан1 и кан2)***. Оба дочерних процесса непрерывно записывают в каналы по ***100*** строк вида: ***номер_строки pid_процесса текущее_время*** (мсек). Всего процессы должны записать ***1000*** строк. Семафоры (***сем1***), (***сем2***) используются процессами для разрешения кому из процессов получить доступ к каналу. Родительский процесс читает из каждого канала по ***75*** строк и выводит их на экран. Дочерние процессы начинают операции с каналами после получения сигнала ***SIGUSR2*** от родительского процесса.

3. Создать два дочерних процесса. Родительский процесс создаёт семафор (***сем1***) и разделяемую память. Оба дочерних процесса непрерывно записывают в разделяемую память по ***100*** строк вида: ***номер_строки pid_процесса текущее_время*** (мсек). Всего процессы должны записать ***1000*** строк. Семафор ***сем1*** используется процессами для разрешения кому из процессов получить доступ к разделяемой памяти. Родительский процесс читает из разделяемой памяти по ***75*** строк и выводит их на экран. Дочерние процессы начинают операции с файлом после получения сигнала ***SIGUSR1*** от родительского процесса.

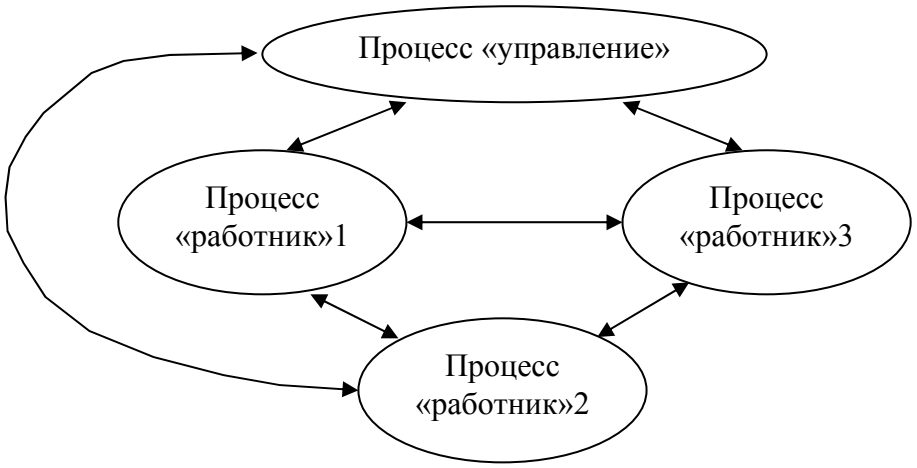
4. Организовать взаимодействие процессов следующей структуры:



Процессы «вопрос»(ы) посылают запросы процессу «ответ» по неименованным каналам и получают по ним ответы. Должны быть

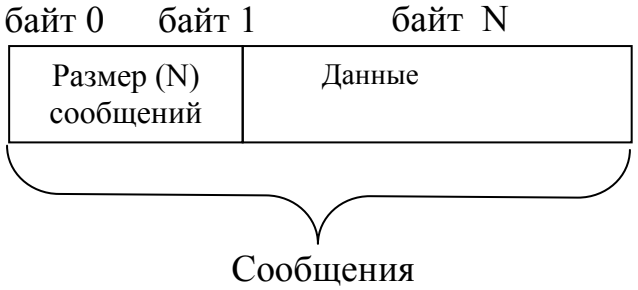
предусмотрены типы ответов, которые инициируют завершение процессов «вопрос», а также должны быть вопросы, которые инициируют порождение новых процессов.

5. Организовать взаимодействие процессов следующей структуры:

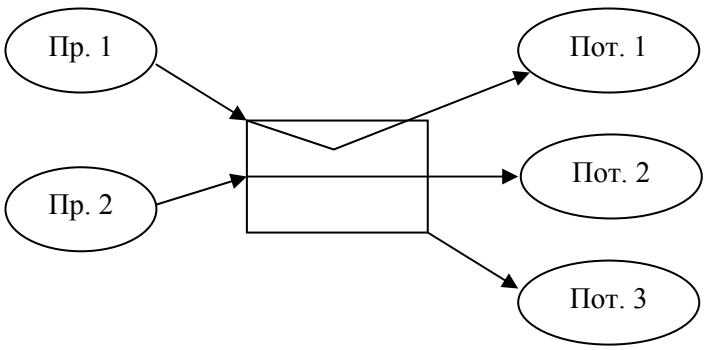


Процессы «работники» по неименованным каналам обмениваются между собой данными. Неименованные каналы существуют также между процессом «Управление» и процессами «работниками». Процесс «Управление» инициирует завершение процессов «работников».

6. Смоделировать посредством неименованного канала работу системы «производители-потребители». Производители посылают сообщения переменной длины, потребители читают эти сообщения. При записи и чтении данных в канал решить задачу взаимного исключения. Формат порции записи:



Структура «производители-потребители»:



Лабораторная работа №6

ПОТОКИ В ОС LINUX

Цель работы – изучение потоков в ОС Linux.

Теоретическая часть

Существует расширенная реализация понятия ***процесс***, когда ***процесс*** представляет собой совокупность выделенных ему ресурсов и набора ***нитей исполнения***. ***Нити (threads)*** или потоки процесса разделяют его программный код, глобальные переменные и системные ресурсы, но каждая ***нить*** имеет собственный программный счетчик, свое содержимое регистров и свой стек. Все глобальные переменные доступны в любой из дочерних нитей. Каждая нить исполнения имеет в системе уникальный номер – идентификатор ***нити***. Поскольку традиционный процесс в концепции нитей исполнения трактуется как процесс, содержащий единственную ***нить*** исполнения, мы можем узнать идентификатор этой ***нити*** и для любого обычного процесса. Для этого используется функция ***pthread_self()***. Нить исполнения, создаваемую при рождении нового процесса, принято называть **начальной** или **главной** нитью исполнения этого процесса. Для создания нитей используется функция ***pthread_create***:

#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
void *(*start_routine)(void*), void *arg);

Функция создает новую нить в которой выполняется функция пользователя ***start_routine*** ,передавая ей в качестве аргумента параметр ***arg***. Если требуется передать более одного параметра, они собираются в структуру, и передается адрес этой структуры. При удачном вызове функция ***pthread_create*** возвращает значение ***0*** и помещает идентификатор новой нити исполнения по адресу, на который указывает параметр ***thread***. В случае ошибки возвращается положительное значение, которое определяет код ошибки, описанный в файле ***<errno.h>***. Значение системной переменной ***errno*** при этом не устанавливается. Параметр ***attr*** служит для задания различных атрибутов создаваемой нити. Функция нити должна иметь заголовок вида:

void * start_routine (void *)

Завершение функции потока происходит если:

- функция нити вызвала функцию ***pthread_exit()***;
- функция нити достигла точки выхода;
- нить была досрочно завершена другой нитью.

Функция ***pthread_join()*** используется для перевода нити в состояние ожидания:

#include <pthread.h>

int pthread_join (pthread_t thread, void **status_addr);

Функция ***pthread_join()*** блокирует работу вызвавшей ее нити исполнения до завершения нити с идентификатором ***thread***. После разблокирования в указатель, расположенный по адресу ***status_addr***, заносится адрес, который вернул завершившийся ***thread*** либо при выходе из ассоциированной с ним функции, либо при выполнении функции ***pthread_exit()***. Если нас не интересует, что вернула нам нить исполнения, в качестве этого параметра можно использовать значение ***NULL***.

Для компиляции программы с нитями необходимо подключить библиотеку ***pthread.lib*** следующим способом:

gcc 1.c –o 1.exe -lpthread

Порядок выполнения работы

1. Изучить теоретическую часть лабораторной работы.

2. Написать программу, создающую два дочерних потока. Родительский процесс и два дочерних потока должны выводить на экран свой ***id*** и ***pid*** родительского процесса и текущее время в формате: ***часы: минуты: секунды: миллисекунды***.

Варианты индивидуальных заданий

1. Написать программу нахождения массива ***K*** последовательных значений функции ***y[i]=sin(2*PI*i/N) (i=0,1,2…K-1)*** с использованием ряда Тейлора. Пользователь задаёт значения ***K***, ***N*** и количество ***n*** членов ряда Тейлора. Для расчета каждого члена ряда Тейлора запускается отдельный поток. Каждый поток выводит на экран свой ***id*** и рассчитанное значение ряда. Головной процесс ожидает завершения работы всех потоков и суммирует все члены ряда Тейлора. Полученное значение ***y[i]*** он записывает в файл результата.

2. Написать программу синхронизации двух каталогов, например, ***Dir1*** и ***Dir2***. Пользователь задаёт имена ***Dir1*** и ***Dir2***. В результате работы программы файлы, имеющиеся в ***Dir1***, но отсутствующие в ***Dir2***, должны скопироваться в ***Dir2*** вместе с правами доступа. Процедуры копирования должны запускаться в отдельном потоке для каждого копируемого файла. Каждый поток выводит на экран свой ***id,*** имя копируемого файла и число скопированных байт. Число одновременно работающих потоков не должно превышать ***N*** (вводится пользователем).

3. Написать программу поиска одинаковых по их содержимому файлов в двух каталогов, например, ***Dir1*** и ***Dir2***. Пользователь задаёт имена ***Dir1*** и ***Dir2***. В результате работы программы файлы, имеющиеся в ***Dir1***, сравниваются с файлами в ***Dir2*** по их содержимому. Процедуры сравнения должны запускаться в отдельном потоке для каждой пары сравниваемых файлов. Каждый поток выводит на экран свой ***id***, имя файла, общее число

просмотренных байт и результаты сравнения. Число одновременно работающих потоков не должно превышать ***N*** (вводится пользователем).

4. Написать программу поиска заданной пользователем комбинации из ***m*** байт (***m < 255***) во всех файлах текущего каталога. Пользователь задаёт имя каталога. Главный процесс открывает каталог и запускает для каждого файла каталога отдельный поток поиска заданной комбинации из ***m*** байт. Каждый поток выводит на экран свой ***id***, имя файла, общее число просмотренных байт и результаты поиска. Число одновременно работающих потоков не должно превышать ***N*** (вводится пользователем).

5. Создать дерево потоков по индивидуальному заданию. Каждый поток постоянно, через время ***t*** , выводит на экран следующую информацию:

номер процесса/потока id ppid текущее время (мсек).

Время ***t =((номер процесса/потока по дереву)*200*** (мсек)**.**

Лабораторная работа №7

СЕМАФОРЫ В ОС UNIX/LINUX

Цель работы – изучение механизма взаимодействия процессов с использованием семафоров.

Теоретическая часть

Семафор – переменная определенного типа, которая доступна параллельным процессам для проведения над ней только двух операций:

- ***A(S, n)*** – увеличить значение семафора ***S*** на величину ***n***;
- ***D(S, n)*** – если значение семафора ***S*** < ***n***, процесс блокируется. Далее ***S*** = ***S - n***;
- ***Z(S)*** – процесс блокируется до тех пор, пока значение семафора ***S*** не станет равным 0.

Семафор играет роль вспомогательного критического ресурса, так как операции ***A*** и ***D*** неделимы при своем выполнении и взаимно исключают друг друга. Семафорный механизм работает по схеме, в которой сначала исследуется состояние критического ресурса, а затем уже осуществляется допуск к критическому ресурсу или отказ от него на некоторое время. Основным достоинством семафорных операций является отсутствие состояния «активного ожидания», что может существенно повысить эффективность работы мультипрограммной вычислительной системы.

Для работы с семафорами имеются следующие системные вызовы:

Создание и получение доступа к набору семафоров:

int semget(key_t key, int nsems, int semflg);

Параметр ***ke***y является ключом для массива семафоров, т.е. фактически его именем. В качестве значения этого параметра может использоваться значение ключа, полученное с помощью функции ***ftok()***, или специальное значение ***IPC_PRIVATE***. Использование значения ***IPC_PRIVATE*** всегда приводит к попытке создания нового массива семафоров с ключом, который не совпадает со значением ключа ни одного из уже существующих массивов и не может быть получен с помощью функции ***ftok()*** ни при одной комбинации ее параметров. Параметр ***nsems*** определяет количество семафоров в создаваемом или уже существующем массиве. В случае если массив с указанным ключом уже имеется, но его размер не совпадает с указанным в параметре ***nsems***, констатируется возникновение ошибки.

Параметр ***semflg*** – флаги – играет роль только при создании нового массива семафоров и определяет права различных пользователей при доступе к массиву, а также необходимость создания нового массива и поведение

системного вызова при попытке создания. Он является некоторой комбинацией (с помощью операции побитовое или – "|") следующих предопределенных значений и восьмеричных прав доступа:
IPC_CREAT — если массива для указанного ключа не существует, он должен быть создан;
IPC_EXCL — применяется совместно с флагом ***IPC_CREAT***. При совместном их использовании и существовании массива с указанным ключом, доступ к массиву не производится и констатируется ошибка, при этом переменная ***errno***, описанная в файле <***errno.h***>, примет значение ***EEXIST***;

0400 — разрешено чтение для пользователя, создавшего массив
0200 — разрешена запись для пользователя, создавшего массив
0040 — разрешено чтение для группы пользователя, создавшего массив
0020— разрешена запись для группы пользователя, создавшего массив
0004— разрешено чтение для всех остальных пользователей
0002— разрешена запись для всех остальных пользователей
Пример: ***semflg= IPC_CREAT | 0022***

Изменение значений семафоров:
int semop(int semid, struct sembuf *sops, int nsops);
Параметр ***semid*** является дескриптором System V IPC для набора семафоров, т. е. значением, которое вернул системный вызов ***semget()*** при создании набора семафоров или при его поиске по ключу. Каждый из ***nsops*** элементов массива, на который указывает параметр ***sops***, определяет операцию, которая должна быть совершена над каким-либо семафором из массива IPC семафоров, и имеет тип структуры:
struct sembuf {
short sem_num; //номер семафора в массиве IPC семафоров (начиная с ***0***);
short sem_op; //выполняемая операция;
short sem_flg; // флаги для выполнения операции.
}
Значение элемента структуры ***sem_op*** определяется следующим образом:

- для выполнения операции **A(S,n)** значение должно быть равно ***n***;
- для выполнения операции ***D(S,n)*** значение должно быть равно ***-n***;
- для выполнения операции ***Z(S)*** значение должно быть равно ***0***.

Семантика системного вызова подразумевает, что все операции будут в реальности выполнены над семафорами только перед успешным возвращением из системного вызова. Если при выполнении операций ***D*** или ***Z*** процесс перешел в состояние ожидания, то он может быть выведен из этого состояния при возникновении следующих форсмажорных ситуаций: массив семафоров был удален из системы; процесс получил сигнал, который должен быть обработан.

Выполнение разнообразных управляющих операций (включая удаление) над набором семафоров:

int semctl(int semid, int semnum, int cmd, union semun arg);

Изначально все семафоры инициируются нулевым значением.

Порядок выполнения работы

1. Изучить теоретическую часть лабораторной работы

2. Написать программу, создающую дочерний процесс. Родительский процесс создаёт семафор (***сем1***) и общий файл. Дочерний процесс записывает в файл по одной строке всего ***100*** строк вида: ***номер_строки pid_процесса текущее_время*** (мсек). Родительский процесс читает из файла строки и выводит их на экран в следующем виде: ***pid строка_прочитанная_из_файла***. Семафор ***сем1*** используется процессами для разрешения, кому из процессов получить доступ к файлу.

Варианты индивидуальных заданий

1. Организовать функционирование процессов следующей структуры: ***1-2-3-4*** (процесс 1 создаёт процесс 2, процесс 2 создаёт процесс 3, процесс 3 создаёт процесс 4). Далее организовать с использованием общего файла передачу/приём следующей информации в последовательности: ***1-2, 2-3, 3-4, 4-1, 1-2, 2-3, 3-4, 4-1*** и так далее. Каждый процесс выдерживает паузу ***t=100*** мсек между приёмом и посылкой информации. Каждый процесс читает из файла переданные ему строки, выводит их на консоль, затем добавляет свои ***M*** (***M*** – номер процесса) строк вида:

M pid ppid текущее время (мсек) ***процесс_такой-то***

и передаёт их далее. Для синхронизации работы использовать семафоры.

2. Организовать функционирование процессов следующей структуры: ***1-(2,3),3-4*** (процесс 1 создаёт процессы 2 и 3, процесс 3 создаёт процесс 4). Далее организовать с использованием общего файла передачу/приём следующей информации в последовательности: ***1-2, 2-3, 3-4, 4-1, 1-2, 2-3, 3-4, 4-1*** и так далее. Каждый процесс читает из файла переданные ему строки, выводит их на консоль, затем добавляет свои ***M*** (***M*** – номер процесса) строк вида:

M pid ppid текущее время (мсек) ***процесс_такой-то***

и передаёт их далее. Для синхронизации работы использовать семафоры.

Лабораторная работа №8

ИСПОЛЬЗОВАНИЕ ОБЩЕЙ ПАМЯТИ В ОС LINUX

Цель работы - изучение механизма взаимодействия процессов с использованием общей памяти.

Теоретическая часть

Использование общей или разделяемой памяти заключается в создании специальной области памяти, позволяющей иметь к ней доступ нескольким процессам. Системные вызовы для работы с разделяемой памятью:

#include <sys/mman.h>

int shm_open (const char *name, int oflag, mode_t mode);

int shm_unlink (const char *name);

Вызов ***shm_open*** создает и открывает новый (или уже существующий) объект разделяемой памяти. При открытии с помощью функции ***shm_open()*** возвращается файловый дескриптор. Имя ***name*** трактуется стандартным для рассматриваемых средств межпроцессного взаимодействия образом. Посредством аргумента ***oflag*** могут указываться флаги ***O_RDONLY, O_RDWR, O_CREAT, O_EXCL*** и/или ***O_TRUNC***. Если объект создается, то режим доступа к нему формируется в соответствии со значением ***mode*** и маской создания файлов процесса. Функция ***shm_unlink*** выполняет обратную операцию, удаляя объект, предварительно созданный с помощью ***shm_open***. После подключения сегмента разделяемой памяти к виртуальной памяти процесса этот процесс может обращаться к соответствующим элементам памяти с использованием обычных машинных команд чтения и записи, не прибегая к использованию дополнительных системных вызовов.

```
int main (void) {
  int fd_shm;  /* Дескриптор объекта в разделяемой памяти*/
  if ((fd_shm = shm_open (“myshered.shm”, O_RDWR | O_CREAT, 0777)) < 0) {
perror ("error create shm");      return (1); }
```

Для ускорения работы следует использовать файлы, отображаемые в памяти при помощи системного вызова ***mmap()***:

```
#include <unistd.h>
#include <sys/mman.h>
void * mmap(void *start, size_t length, int prot , int flags, int fd, off_t offset);
```

Функция ***mmap*** отображает ***length*** байтов, начиная со смещения ***offset*** файла, определенного файловым описателем ***fd***, в память, начиная с адреса ***start***. Последний параметр ***offset*** необязателен, и обычно равен ***0***. Настоящее местоположение отраженных данных возвращается самой функцией ***mmap***, и никогда не бывает равным ***0***. Аргумент ***prot*** описывает желаемый режим защиты памяти (он не должен конфликтовать с режимом открытия файла):

PROT_EXEC данные в памяти могут исполняться;
PROT_READ данные в памяти можно читать;
PROT_WRITE в область можно записывать информацию;
PROT_NONE доступ к этой области памяти запрещен.
Параметр ***flags*** задает тип отражаемого объекта, опции отражения и указывает, принадлежат ли отраженные данные только этому процессу или их могут читать другие. Он состоит из комбинации следующих битов:
MAP_FIXED использование этой опции не рекомендуется;
MAP_SHARED разделить использование этого отражения с другими процессами, отражающими тот же объект. Запись информации в эту область памяти будет эквивалентна записи в файл. Файл может не обновляться до вызова функций ***msync(2)*** или ***munmap(2)***;
MAP_PRIVATE создать неразделяемое отражение с механизмом ***copy-on-write***. Запись в эту область памяти не влияет на файл. Не определено, являются или нет изменения в файле после вызова ***mmap*** видимыми в отраженном диапазоне.

Для компиляции программы необходимо подключить библиотеку ***rt.lib*** следующим способом: ***gcc 1.c –o 1.exe -lrt***

Порядок выполнения работы

1. Изучить теоретическую часть лабораторной работы

2. Написать программу, создающую дочерний процесс. Родительский процесс создаёт семафор (***сем1***) и общую память. Дочерний процесс записывает в память по 70 строк всего ***1000*** строк вида: ***номер_строки pid_процесса текущее_время*** (мсек). Родительский процесс читает из файла по 81 строке и выводит их на экран в следующем виде: ***pid строка_прочитанная_из_файла***. Семафор ***сем1*** используется процессами для разрешения, кому из процессов получить доступ к файлу.

Варианты индивидуальных заданий

3. Организовать функционирование процессов следующей структуры:
1-2-3-4 (процесс 1 создаёт процесс 2, процесс 2 создаёт процесс 3, процесс 3 создаёт процесс 4). Далее организовать с использованием общей памяти передачу/приём следующей информации в следующей последовательности: ***1-2, 2-3, 3-4, 4-1, 1-2, 2-3, 3-4, 4-1*** и так далее. Каждый процесс выдерживает паузу ***t=100*** мсек между приёмом и посылкой информации и передаёт следующую информацию:
N pid ppid текущее время (мсек) ***процесс_такой-то***.
Для синхронизации работы использовать семафоры.

4. Организовать функционирование процессов следующей структуры:
1-2-3-4 (процесс 1 создаёт процесс 2, процесс 2 создаёт процесс 3, процесс 3 создаёт процесс 4). Далее организовать с использованием общей памяти

передачу/приём следующей информации в следующей последовательности: ***1-2, 2-3, 3-4, 4-1, 1-2, 2-3, 3-4, 4-1*** и так далее. Каждый процесс выдерживает паузу ***t=100*** мсек между приёмом и посылкой информации и передаёт следующую информацию:

N pid ppid текущее время (мсек) ***процесс_такой-то***.

Для синхронизации работы использовать сигналы.

5. Организовать функционирование процессов следующей структуры: ***1-(2,3),3-4*** (процесс 1 создаёт процессы 2 и 3, процесс 3 создаёт процесс 4). Далее организовать с использованием общей памяти передачу/приём следующей информации в следующей последовательности: ***1-2, 2-3, 3-4, 4-1, 1-2, 2-3, 3-4, 4-1*** и так далее. Каждый процесс выдерживает паузу ***t=100*** мсек между приёмом и посылкой информации и передаёт следующую информацию:

N pid ppid текущее время (мсек) ***процесс_такой-то***.

Для синхронизации работы использовать сигналы ***SIGUSR1, SIGUSR2***.

6. Организовать функционирование процессов следующей структуры: ***1-(2,3),3-4*** (процесс 1 создаёт процессы 2 и 3, процесс 3 создаёт процесс 4). Далее организовать с использованием общей памяти передачу/приём следующей информации в следующей последовательности: ***1-2, 2-3, 3-4, 4-1, 1-2, 2-3, 3-4, 4-1*** и так далее. Каждый процесс выдерживает паузу ***t=100*** мсек между приёмом и посылкой информации и передаёт следующую информацию:

N pid ppid текущее время (мсек) ***процесс_такой-то***.

Для синхронизации работы использовать семафоры.

7. Создать два дочерних процесса. Родительский процесс создаёт семафор (***сем1***) и разделяемую память. Оба дочерних процесса непрерывно записывают в разделяемую память по *7* строк вида: ***номер_строки pid_процесса текущее_время*** (мсек). Всего процессы должны записать ***1000*** строк. Семафор ***сем1*** используется процессами для разрешения кому из процессов получить доступ к разделяемой памяти. Родительский процесс читает из разделяемой памяти по ***75*** строк и выводит их на экран. Дочерние процессы начинают операции с файлом после получения сигнала ***SIGUSR1*** от родительского процесса.

ЛИТЕРАТУРА

1. Хэвиленд К., Грэй Д., Салама Б. Системное программирование в UNIX. Руководство программиста по разработке ПО. -М.: ДМК Пресс, 2000. - 368 с.
2. Робачевский А.М. Операционная система UNIX. -СПб.: BHV - Санкт-Петербург, 1997. -528 с.
3. Моли Б. Unix/Linux: теория и практика программирования. - М.: КУДИЦ-ОБРАЗ, 2004, -576 с.
4. Роббинс А. Linux: программирование в примерах. - М.: КУДИЦ-ОБРАЗ, 2005, -656 с.
5. Петерсен Р. LINUX: руководство по операционной системе: Пер. с англ. - К.: Издательская группа BHV, 1997. - 688 с.
6. Чан Т. Системное программирование на C++ для UNIX: Пер. с англ. - К.: Издательская группа BHV, 1997. - 592 с.
7. Лав Р. Linux. Системное программирование. – СПб.: Питер, 2008. – 416 с.

Св. План , поз.

Учебное издание

Алексеев Игорь Геннадьевич,
Бранцевич Петр Юльянович

«Операционные системы и системное программирование»
учебно-методическое пособие
для студентов дневной формы обучения по курсу
«Программное обеспечение информационных
технологий»

Редактор
Корректор