Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра программного обеспечения
информационных технологий

И. Г. Алексеев

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Методическое пособие
для студентов специальности1-40 01 02
«Информационные системы и технологии в экономике»
заочной формы обучения

Минск БГУИР 2011

УДК 004.451(076)
ББК 32.973.26-018.2я73
А47

Р е ц е н з е н т:
доцент Института информационных технологий
учреждения образования
«Белорусский государственный университет информатики и радиоэлектроники»,
кандидат технических наук
В. Н. Мухаметов

Алексеев, И. Г.

А47 Операционные системы : метод. пособие для студ. спец. 1-40 01 02 «Информационные системы и технологии в экономике» заоч. формы обуч. / И. Г. Алексеев. – Минск : БГУИР, 2011. – 32 с.

ISBN 978-985-488-715-9.

В пособии представлены одна контрольная и две лабораторные работы по курсу «Операционные системы». Для каждой из работ приведены индивидуальные задания для их выполнения.

УДК 004.451(076)
ББК 32.973.26-018.2я73

ISBN 978-985-488-715-9

СОДЕРЖАНИЕ

Лабораторная работа №1

СИСТЕМА КОМАНД, ФАЙЛОВАЯ СТРУКТУРА, РАБОТА С ФАЙЛАМИ И УПРАВЛЕНИЕ ОС UNIX/LINUX С ПОМОЩЬЮ ИНТЕРПРЕТАТОРА BASH

Цель работы – изучение команд ОС для работы с файлами, каталогами, дисками, системной датой и временем; текстового редактора и файлового менеджера; исследование основных объектов, команд, типов данных и команд управления интерпретатора ***bash***.

Теоретические сведения

Операционная система ОС ***Linux*** имеет развитую структуру и систему команд. Пользователь может работать либо в текстовом режиме с помощью командной строки, или с использованием графического интерфейса и одного из менеджеров рабочего стола (например ***KDE*** или ***GNOME***). Одновременно в системе могут работать несколько пользователей. В табл. 1 приведены основные команды системы (интерпретатора ***bash***).

Таблица 1

Основные команды системы

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 посылкой сигнала
top			Выводит на консоль список процессов
htop			Выводит на консоль полный список запущенных процессов
su	пароль		Переход в режим ***root***
chmod	права_доступа файл	***chmod 777 1.txt***	Изменение прав доступа к файлам

Окончание табл. 1

Команда	Аргументы/ключи	Пример	Описание
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 count=1***	Копирование побайтное
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

Unix/Linux и ***Windows*** используют различные файловые системы для хранения и организации доступа к информации на дисках. В ***Unix/Linux*** используются файловые системы ***Ext2/Ext3/Ext4***, ***RaiserFS*** и др. Все современные 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 ***Unix/Linux*** поддерживает доступ к ***Windows***-разделам.

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

Корневой каталог /

Корневой каталог / является основой любой файловой системы ***Unix/Linux***. Все остальные каталоги и файлы располагаются в рамках структуры (дерева), порождённой корневым каталогом, независимо от их физического местонахождения.

/bin

В этом каталоге находятся часто употребляемые команды и утилиты системы общего пользования. Сюда входят все базовые команды, доступные даже в том случае, если была примонтирована только корневая файловая система. Примерами таких утилит являются ***ls, cp, sh*** и т. п.

/boot

Каталог содержит всё необходимое для процесса загрузки операционной системы: программу-загрузчик, образ ядра операционной системы и т. п.

/dev

Каталог содержит специальные файлы устройств, при помощи которых осуществляется доступ к периферийным устройствам. Наличие такого каталога не означает, что специальные файлы устройств нельзя создавать в другом месте, просто достаточно удобно иметь один каталог для всех файлов такого типа.

/etc

В этом каталоге находятся системные конфигурационные файлы. В качестве примеров можно привести файлы ***/etc/fstab***, содержащий список монтируемых файловых систем, и ***/etc/resolv.conf***, задающий правила составления локальных ***DNS***-запросов. Среди наиболее важных файлов – скрипты инициализации и останова системы. В системах, наследующих особенности ***Unix/Linux***, для них отведены каталоги ***c /etc/rc0.d*** по ***/etc/rc6.d*** и общий для всех файл описания – ***/etc/inittab***.

/home (необязательно)

Каталог содержит домашние каталоги пользователей. Его наличие в корневом каталоге необязательно, а содержимое зависит от особенностей конкретной ***Unix/Linux*** –подобной операционной системы.

/lib

Каталог для статических и динамических библиотек, необходимых для запуска программ, находящихся в каталогах ***/bin*** и ***/sbin***.

/media

Стандартный каталог для временного монтирования файловых систем – например, гибких и флэш-дисков, компакт-дисков и т. п.

/opt

Каталог для дополнительного программного обеспечения, устанавливаемого в системе. Обычно в этот каталог устанавливаются программы, не входящие в основной дистрибутив.

/root (необязательно)

Домашний каталог суперпользователя. Его наличие в корневом каталоге необязательно.

/sbin

В этом каталоге находятся команды и утилиты для системного администратора. Примерами таких команд являются ***route, halt, init*** и т. п. Для аналогичных целей применяются каталоги ***/usr/sbin и /usr/local/sbin***.

/usr

Этот каталог повторяет структуру корневого каталога – содержит каталоги ***/usr/bin, /usr/lib, /usr/sbin***, служащие для аналогичных целей.

/usr/include

Содержит заголовочные файлы языка С для всевозможных библиотек, расположенных в системе.

/usr/local

Ещё один уровень, в котором можно полностью воспроизвести структуру данных, содержащуюся в корневом каталоге. Обычно этот каталог служит для размещения программ, установленных администратором в дополнение к стандартной поставке операционной системы.

/usr/share

Хранит неизменяющиеся данные для установленных программ. Особый интерес представляет каталог ***/usr/share/doc***, в который добавляется документация ко всем установленным программам.

/var, /tmp

Используются для хранения временных данных системных ***(/var)*** и пользовательских ***(/tmp)*** процессов. Каталог ***/var*** обычно содержит часто изменяемые системные файлы, например, в каталоге ***/var/log*** размещаются системные журналы.

Устройства

ОС ***Unix/Linux*** изолирует приложения от аппаратной части вычислительной системы. Она предоставляет единый интерфейс различных устройств системы в виде специальных файлов устройств. Специальный файл связывает прикладное приложение с драйвером устройства. Каждый специальный файл соответствует какому-либо физическому устройству (диск, устройство печати, терминал).

Вся работа приложения с устройством происходит через специальный файл, а соответствующий ему драйвер обеспечивает выполнение операций I/O в соответствии с конкретным протоколом обмена данными с устройством. Существуют файлы блочных устройств и файлы символьных устройств. Файлы блочных устройств обеспечивают интерфейс устройствам, обмен данными с которыми происходит большими фрагментами – блоками. При этом ядро ОС обеспечивает нужную буферизацию.

Файлы символьных устройств используются для доступа к устройствам, драйверы которых обеспечивают собственную буферизацию и побайтную передачу данных (накопители на магнитной ленте, терминалы). Одно и то же устройство может иметь как блочный, так и символьный интерфейсы. Названия специальных файлов устройств зависят от конкретной версии ***Unix/Linux***, но при этом присутствуют некоторые правила названия таких устройств:

cktldmsn,

где ***k*** – номер контроллера;

l – номер устройства;

m – номер раздела;

n – логический номер устройства.

/dev/cdn – CD–ROM;

/dev/sda, /dev/sdb, /dev/sdc, ...– HDD1, HDD2, HDD3, HDD4,…;

/dev/sda1 – раздел 1 HDD1;

/dev/sda2 – раздел 2 HDD1;

/dev/sda3 – раздел 3 HDD1;

…
/dev/ttypn – подчиненный псевдотерминал;
/dev/console – системная консоль;
/dev/tty – терминальная линия управляющего терминала для данного процесса;
/dev/mem – физическое ОЗУ;
/dev/kmem – виртуальная память ядра;
/dev/null – нулевое устройство, весь вывод на него пропадает, а при попытке ввода из этого устройства возвращается 0 байт информации;
/dev/zero – нулевое устройство, весь вывод на него уничтожается, а ввод приводит к получению последовательности нулей.
При выводе каталога **ls –l** можно определить, какое это специальное устройство и какой номер драйвера используется при его работе.

Права доступа к файлу

В операционной системе ***Unix/Linux*** существует три базовых класса доступа к файлу, в каждом из которых установлены права доступа к файлу. Эти классы следующие.

1. Класс владельца-пользователя файла.
2. Класс членов группы, являющейся владельцем файлов.
3. Класс остальных пользователей, кроме суперпользователя.

Поддерживается три типа прав доступа для каждого класса: ***r*** на чтение; ***w*** на запись; ***x*** на выполнение. Если выполнить команду ***ls –l***, то на экран выводится полный список файлов для каталога, который является текущим

– rwx rwx rwx 1.stud1 gr1 … f1.dat
– r–x r– x – – – 2.stud2 gr1 … f2.dat

тип файла категории пользователя

Первый символ определяет тип файла:
d – каталог, ***c*** – специальное символьное устройство, ***p*** – поименованный ***FIFO*** файл, ***-*** – обычный файл, ***b*** – блоковое устройство.

Права доступа могут быть изменены только владельцем файла или суперпользователем. Для этого используется команда

chmod | ***u*** ***g*** ***o*** ***a*** | ***+*** ***–*** ***=*** | ***r*** ***w*** ***x*** | ***f₁ f₂ f₃***

chmod a + w t.txt
chmod g+x–w r.out

Значение прав доступа различно для разных типов файлов. Например, чтобы просмотреть содержимое файла с помощью команды ***cat***, пользователь должен иметь право на чтение. Для редактирования файла и его изменения должно быть право на запись. Для запуска программы на выполнение должно быть право на выполнение.

Права на доступ к каталогам имеют специфические особенности. К примеру, право чтения каталога позволяет получить лишь имена файлов, находящихся в данном каталоге. Для получения дополнительной информации, например получения команды ***ls –l***, требуется обращение к индексным дескрипторам файла, что требует наличия прав выполнения для каталога. Право выполнения каталога требуется и для команды ***cd***. Права чтения и выполнения для каталогов действуют независимо. Комбинацией этих прав можно добиться создания «***темных каталогов***», файлы которых доступны только в том случае, если пользователь знает заранее их имена, т. к. получение списка файлов в таких каталогах запрещено. Этот подход может быть использован для построения некоторых справочных систем, когда отдельным пользователям сообщаются имена разделов, с которыми они могут работать, а остальные файлы им недоступны.

Создание и удаление файлов в каталоге требует изменения его содержимого и, следовательно, права на запись в этот каталог. При этом не учитываются права доступа для самого файла, т.е. для удаления файла из каталога не требуется каких-либо прав доступа к файлу, нужно иметь лишь право на доступ для каталога, в котором содержится файл.

Дополнительные атрибуты файлов по управлению правами доступа

Существует несколько дополнительных атрибутов, изменяющих стандартное выполнение различных операций. Это относится как к обычным файлам, так и к каталогам.

t; s; s; l

t – сохранить образ выполняемого файла после завершения;

s – установить пользовательский идентификатор процесса при выполнении;

s – установить групповой идентификатор процесса при выполнении;

1 – установить обязательное блокирование файлов при выполнении

Для установления атрибутов используется та же команда: ***chmod u+s f1***.

Атрибут ***t***: после завершения выполнения задачи ее образ (код и данные) остается в памяти, что приводит к тому, что последующие запуски программы занимают намного меньше времени.

Атрибуты ***SUID*** и ***SGID*** позволяют изменить права пользователя при запуске на выполнение файла, имеющего эти атрибуты. Обычно запускаемая программа имеет права доступа к системным ресурсам на основе прав доступа пользователя, запустившего программу. Установка флагов ***SUID*** или ***SGID*** изменяет эти правила, назначая права доступа владельцев исполняемых файлов. То есть если владельцем-пользователем является суперпользователь, то неограниченные права доступа к системным ресурсам получает и пользователь, запустивший этот файл.

Атрибут ***1*** используется для устранения конфликтов, когда одновременно несколько задач работают с одним и тем же файлом.

Для каталогов устанавливаются только два дополнительных параметра: ***t*** и ***s***. Установка ***t*** для каталога позволяет установить дополнительную защиту файлов, находящихся в каталоге. Из такого каталога пользователь может уда-

лить только те файлы, которыми он владеет или на которые он имеет право доступа на запись даже при наличии прав на запись в каталог. Это используется при работе с каталогом временных файлов ***TMP***, который открыт на запись для всех пользователей, но в котором нежелательно удаление каталогов файлов других пользователей.

Атрибут **SGID** тоже имеет специальное значение для каталогов. При установке этого атрибута для каталога вновь созданные файлы этого каталога будут наследовать владельца-группу по владельцу-группе каталога.

Для выполнения операций записи и чтения данных в существующем файле его следует открыть:

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

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

Второй аргумент ***flags*** функции ***open()*** имеет целочисленный тип и определяет метод доступа и принимает одно из значений, заданных константами в заголовочном файле ***fcnt1.h***. В файле определены три постоянные:

O_RDONLY – открыть файл только для чтения;

O_WRONLY – открыть файл только для записи;

O_RDWR – открыть файл для чтения и записи;

Для ***fopen()***– ***“r”***, ***“w”***, ***“rw”.***

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);
```

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

Поля структуры stat включают 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 ***Unix/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.***

Пример: ***stat("1.exe", &st1);***

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);***

Структура каталогов ОС ***Unix/Linux*** представлена в табл. 2. 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 2

Каталог	Описание
/	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

Окончание табл. 2

Каталог	Описание
/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

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,S2,$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 файла-скрипта ***mydir.sh***, выводящего на консоль и в файл содержимое каталога, где имя каталога передаётся скрипту в качестве первого аргумента командной строки:

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

Для запуска скрипта необходимо изменить его права доступа, разрешив выполнение: ***>chmod 777 mydir.sh***

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

Можно создать собственную переменную и присвоить ей значение:

A=121 или ***A="121"*** или ***let A=121*** или ***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) логические операции

!exp — логическое отрицание (не),
exp1 –a exp2 — умножение условий (и),
exp1 –o exp2 — сложение условий (или).

Проверка условия: ***if [expr] ;***

then com 1 — Если условие ***expr=true,*** то команда ***com 1… com n***
…
com n
(elif expr2
com1
…
com n
)
else

```
        com 1
…
com   n
fi
```

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

Варианты индивидуальных заданий

Во всех заданиях должен быть контроль ошибок **(**если к какому-либо каталогу нет доступа, необходимо вывести соответствующее сообщение и продолжить выполнение). Вывод сообщений об ошибках должен производиться в стандартный поток вывода сообщений об ошибках (***stderr***) в следующем виде: ***Имя_модуля: текст_сообщения***. Пример: ***./1.exe : error open file: 1.txt.*** Имя модуля, имя файла берутся из аргументов командной строки. Номер варианта индивидуального задания ***K*** равен числу букв ***N1*** вашей фамилии, умноженному на число букв ***N2*** Вашего имени (по паспорту), умноженному на число букв ***N3*** вашего отчества по модулю 7: ***k= (N1*N2*N3) mod 7***

0. Написать скрипт для поиска файлов заданного размера в заданном каталоге и во всех его подкаталогах (имя каталога задаётся пользователем в качестве третьего аргумента командной строки). Диапазон (мин.– макс.) размеров файлов задаётся пользователем в качестве первого и второго аргумента командной строки. Проверить работу программы для каталога ***/usr и*** диапазона (мин.– макс.) ***1000 1010.***

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

2. Написать скрипт для поиска заданной пользователем строки во всех файлах заданного каталога и во всех его подкаталогах (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 соответствующее сообщение и продолжить выполнение. Проверить работу программы для каталога ***/usr*** и строки «***stdio.h»***.

3. Написать скрипт поиска одинаковых по их содержимому файлов в двух каталогах, 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. Проверить работу программы для каталога ***/usr (Dir1)***и любого каталога в каталоге ***/home (Dir2).***

4. Написать скрипт, находящий в заданном каталоге и во 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. Проверить работу программы для каталога ***/usr*** пользователь ***root.***

5. Написать скрипт, находящий в заданном каталоге и во всех его подкаталогах все файлы, заданного размера в заданном каталоге (имя каталога задаётся пользователем в качестве первого аргумента командной строки). Диапазон (мин.– макс.) размеров файлов задаётся пользователем в качестве второго и третьего аргументов 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. Проверить работу программы для каталога ***/usr*** и диапазона (мин.– макс.) ***1000 1010.***

6. Написать скрипт, подсчитывающий суммарный размер файлов в заданном каталоге и во всех его подкаталогах (имя каталога задаётся пользователем в качестве аргумента командной строки). Скрипт выводит результаты подсчёта в файл (второй аргумент командной строки) в виде: каталог (полный путь), суммарный размер файлов, число просмотренных файлов. Проверить работу программы для каталога ***/usr.***

Лабораторная работа №2

РАБОТА С КАТАЛОГАМИ, ОСНОВНЫЕ ПРИНЦИПЫ ПРОГРАММИРОВАНИЯ ПРОЦЕССОВ И ПОТОКОВ В ОС *UNIX/LINUX*

Цель работы – изучение файловой системы ОС ***Unix/Linux*** и основных функций для работы с каталогами и файлами; исследование методов создания процессов, основных функций создания и управления процессами, обмена данными между процессами.

Теоретические сведения

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

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

Для чтения записей каталога существует вызов:

```
struct dirent *readdir(DIR *dirptr);
```

Структура dirent такова:

```
struct dirent {
        long      d_ino;  // индексный дескриптор файла
        off_t     d_off; //смещение данного элемента в реальном каталоге
```

```
unsigned short d_reclen;// длина структуры
char        d_name [1]; // имя элемента каталога
};
```

Поле ***d_name*** есть начало массива символов, задающего имя элемента каталога. Данное имя ограничено нулевым байтом и может содержать не более ***MAXNAMLEN*** символов.

Пример вызова:

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

При первом вызове функции ***readdir()*** в структуру ***d*** будет считана первая запись каталога. После прочтения последней записи каталога будет возвращено значение ***NULL***. Для возврата указателя в начало каталога на первую запись существует вызов:

```
void rewindir(DIR *dirptr);
```

Для получения текущего рабочего каталога (пути) существует функция

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

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

```
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 ожидание любого дочернего процесса, чей идентификатор группы процессов равен абсолютному значению ***pid;***

pid=–1 ожидание любого дочернего процесса; функция **wait** ведет себя точно так же;

pid =0 ожидание любого дочернего процесса, чей идентификатор группы процессов равен идентификатору текущего процесса;

pid>0 ожидание дочернего процесса, чей 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 асин-

хронный сбор информации о статусе завершившихся порожденных процессов процессом-родителем.

Для перегрузки исполняемой программы можно использовать функции семейства ***exec***. Основное различие 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);

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***. Нить исполнения, создаваемую при рождении нового процесса, принято называть **начальной** или **главной** нитью исполнения этого процесса. Для создания дополнительных 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

Время в ***Unix/Linux*** отсчитывается в секундах, прошедшее с начала этой эпохи (***00:00:00 UTC, 1 Января 1970 года***). Для работы с системным временем можно использовать следующие функции:

```
#include <sys/time.h>
time_t   time (time_t *tt); //текущее время в секундах с 01.01.1970
struct tm *  localtime(time_t *tt)
int   gettimeofday(struct timeval *tv, struct timezone *tz);
struct timeval {
                        long    tv_sec;       /* секунды */
                        long    tv_usec;      /* микросекунды */
                      };
struct tm {
   int tm_sec;        /* seconds */
   int tm_min;        /* minutes */
   int tm_hour;       /* hours */
   int tm_mday;       /* day of the month */
   int tm_mon;        /* month */
   int tm_year;       /* year */
   int tm_wday;       /* day of the week */
   int tm_yday;       /* day in the year */
   int tm_isdst;      /* daylight saving time */
};
```

Варианты индивидуальных заданий

Номер ***индивидуального*** варианта ***K*** равен числу букв вашей фамилии (***N1***), умноженному на число букв по паспорту вашего имени (***N2***), умноженному на число букв Вашего отчества (***N3)*** по модулю ***23***:

K= (N1*N2*N3) mod 23

В каждой программе должен быть контроль ошибок для всех операций с файлами и каталогами.

0. 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). Процедуры копирования должны запускаться в отдельном процессе для каждого копируемого файла с использованием функций ***read()*** и ***write()***. Каждый процесс выводит на экран свой ***pid,*** полный путь, имя копируемого файла и число скопированных байт. Число запущенных процессов в любой момент времени не должно превышать ***N*** (вводится пользователем). Проверить работу программы для каталога ***/usr/include N=6***.

1. Написать программу синхронизации 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***, но отсутствующие в ***Dir2***, должны скопироваться в ***Dir2*** вместе с правами доступа. Процедуры копирования должны запускаться в отдельном процессе для каждого копируемого файла с использованием функций ***read()*** и ***write()***. Каждый процесс выводит на экран свой ***pid,*** полный путь, имя копируемого файла и число скопированных байт. Число запущенных процессов в любой момент времени не должно превышать ***N*** (вводится пользователем). Проверить работу программы для каталога ***/usr/include/ и любого другого каталога в /home/ N=6***.

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. Процедура поиска для каждого подкаталога должна запускаться в отдельном процессе. Каждый процесс выводит на экран свой ***pid,*** полный путь, имя и размер просмотренного файла, общее число просмотренных файлов в подкаталоге. Число запущенных процессов в любой момент времени не должно превышать ***N*** (вводится пользователем). Проверить работу программы для каталога ***/usr*** найти файл ***stdio.h N=6***.

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). Процедура поиска для каждого подкаталога должна запускаться в отдельном процессе. Каждый процесс выводит на экран свой ***pid,*** полный путь, имя и размер просмотренного файла, общее число просмотренных файлов в подкаталоге. Число запущенных процессов в любой момент времени не должно превышать ***N*** (вводится пользователем). Проверить работу программы для каталога ***/usr/ размер 31000 31500 дата с 01.01.1970 по текущую дату N=6***.

4. 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. Процедура просмотра для каждого подкаталога должна запускаться в отдель-

ном процессе. Каждый процесс выводит на экран свой ***pid,*** полный путь, имя и размер просмотренного файла, общее число просмотренных файлов в подкаталоге. Число запущенных процессов в любой момент времени не должно превышать ***N*** (вводится пользователем). Проверить работу программы для каталога ***/usr N=6***.

5. Написать программу, находящую в заданном каталоге и всех его подкаталогах все исполняемые файлы. Диапазон (мин. – макс.) размеров файлов задаётся пользователем в качестве первого и второго аргумента командной строки. Имя каталога задаётся пользователем в качестве третьего аргумента 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. Процедура поиска для каждого подкаталога должна запускаться в отдельном процессе. Каждый процесс выводит на экран свой ***pid,*** полный путь, имя и размер просмотренного файла, общее число просмотренных файлов в подкаталоге. Число запущенных процессов в любой момент времени не должно превышать ***N*** (вводится пользователем). Проверить работу программы для каталога ***/usr/*** и размера ***31000 31500 N=6***.

6. Написать программу нахождения массива значений функции ***y[i]=sin(2*PI*i/N) i=[0,N–1]*** с использованием ряда Тейлора. Пользователь задаёт значения ***N*** и количество ***n*** членов ряда Тейлора. Для расчета каждого члена ряда Тейлора запускается отдельный процесс и его результат (член ряда) записывается в файл. Каждый процесс выводит на экран свой ***id*** и рассчитанное значение ряда. Головной процесс суммирует все члены ряда Тейлора и полученное значение ***y[i]*** записывает в файл. Проверить работу программы для значений ***N,n =[64,5]*** и ***N,n =[32768,7]***.

7. 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 с использованием функции ***fork()*** в отдельном процессе для каждой пары сравниваемых файлов. Каждый процесс выводит на экран свой ***pid***, имя файла, число просмотренных байт и результаты сравнения. Число запущенных процессов в любой момент времени не должно превышать ***N*** (вводится пользователем). Проверить работу программы для каталога ***/usr/include/ и любого другого каталога в /home N=6***.

8. Написать программу поиска заданной пользователем комбинации из ***m*** байт (***m < 255***) во всех файлах текущего каталога. Пользователь задаёт в качестве аргументов командной строки имя каталога, строку поиска, файл результата. Главный процесс открывает каталог и запускает для каждого файла каталога отдельный процесс поиска заданной комбинации из ***m*** байт. Каждый процесс выводит на экран и в файл результата свой ***pid***, полный путь и имя файла, число просмотренных в данном файле байт и результаты поиска (всё в одной строке!). Результаты поиска (только найденные файлы) по предыдущему формату записываются в выходной файл. Число запущенных процессов в любой момент

времени не должно превышать ***N*** (вводится пользователем). Проверить работу программы для каталога ***/usr/include/ и строки "stdio.h"***

9. Тоже что и п. 8, но включая подкаталоги. Проверить работу программы для каталога: ***/usr/*** размер ***31000 31500 N=6.***.

10. Разработать программу «интерпретатор команд», которая воспринимает команды, вводимые с клавиатуры, и осуществляет их корректное выполнение. Для этого каждая вводимая команда должна выполняться в отдельно запускаемом процессе с использованием вызова ***exec()***. Нельзя использовать вызов любого готового интерпретатора из своей программы или вызов ***system***(). Для проверки работы выполнить команду: ***ls –l > 1.txt***. Предусмотреть контроль ошибок и команду выхода из программы.

11. То же, что и в п. 1, но вместо процессов использовать потоки.
12. То же, что и в п. 2, но вместо процессов использовать потоки.
13. То же, что и в п. 3, но вместо процессов использовать потоки.
14. То же, что и в п. 4, но вместо процессов использовать потоки.
15. То же, что и в п. 5, но вместо процессов использовать потоки.
16. То же, что и в п. 6, но вместо процессов использовать потоки.
17. То же, что и в п. 7, но вместо процессов использовать потоки.
18. То же, что и в п. 8, но вместо процессов использовать потоки.
19. То же, что и в п. 9, но вместо процессов использовать потоки.

Контрольная работа

СРЕДСТВА МЕЖПРОЦЕССНОГО ВЗАИМОДЕЙСТВИЯ (СИГНАЛЫ, КАНАЛЫ, РАЗДЕЛЯЕМАЯ ПАМЯТЬ, СЕМАФОРЫ)

Цель работы – изучить методы и средства взаимодействия процессов с использованием сигналов, каналов, разделяемой памяти и семафоров в ОС ***Unix/Linux*.**

Теоретические сведения

Все процессы в ***Unix/Linux*** выполняются в раздельных адресных пространствах и для организации межпроцессного взаимодействия необходимо использовать специальные средства:

1) общие файлы;
2) сигналы (***signal***);
3) каналы (***pipe***);
4) общую или разделяемую память;
5) семафоры.

1. При использовании общих файлов оба процесса открывают один и тот же файл, с помощью которого и обмениваются информацией. 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*** байт, 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*** или ***munmap*;**

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.

2. Сигналы. 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;

если ***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;

если ***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***);

если ***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).

Для того чтобы послать сигнал одновременно нескольким процессам, их необходимо объединить в группу с помощью, например, функций **getpgrp()** или ***setpgid().***

int setpgrp(pid_t pid, pid_t pgid);

int setpgid(pid_t pid, pid_t pgid);

Организация новой группы процессов выполняется системным вызовом ***getpgrp(),*** а получение собственного идентификатора группы процессов – системным вызовом ***getpgrp().*** Функция ***setpgid***() присваивает идентификатор группы процессов ***pgid*** тому процессу, который был определен ***pid***. Если значение ***pid*** равно нулю, то процессу присваивается идентификатор текущего процесса. Если значение ***pgid*** равно нулю, то используется идентификатор процесса, указанный ***pid***. Если ***setpgid*** используется для перевода процесса из одной группы в другую, то обе группы должны быть частью одной сессии. В этом случае ***pgid*** указывает на существующую группу процессов, с которой должен ассоциироваться процесс, а идентификатор сессии этой группы должен соответствовать идентификатору сессии присоединяющегося процесса. ***getpgid*** возвращает идентификатор группы процессов, к которой принадлежит процесс, указанный ***pid***. Если значение ***pid*** равно нулю, то используется идентификатор текущего процесса. Вызов ***setpgrp()*** эквивалентен ***setpgid(0,0)***.
Аналогично значение ***getpgrp()*** эквивалентно ***getpgid(0)***.

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 установки обработчика сигнала для процесса. 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, спе-

циальное 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.***

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.

3. Каналы являются одной из самых сильных и характерных особенностей ОС ***Unix/Linux***, доступных даже с уровня командного интерпретатора. Они позволяют легко соединять между собой произвольные последовательности команд. После создания канала с ним можно работать просто при помощи вызовов ***read*** и ***write***. Каналы обращаются с данными в порядке «первый вошел – первым вышел» ***(FIFO)***. Размеры блоков при записи в канал и чтении из него необязательно должны быть одинаковыми. 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.

В ОС ***Unix/Linux*** различают два вида программных каналов: именованный программный канал и неименованный 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]);

4. Использование 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 shmget(key_t key, int size, int shmflg);

Системный вызов ***shmget*** предназначен для выполнения операции доступа к сегменту разделяемой памяти и, в случае ее успешного завершения, возвращает дескриптор ***System V IPC*** для этого сегмента (целое неотрицательное число, однозначно характеризующее сегмент внутри вычислительной системы и использующееся в дальнейшем для других операций с ним). Параметр ***key*** является ключом ***System V IPC*** для сегмента, т. е. фактически его именем из пространства имен ***System V IPC***. В качестве значения этого параметра может быть использовано значение ключа, полученное с помощью функции ***ftok()***, или специальное значение ***IPC_PRIVATE***. Использование значения ***IPC_PRIVATE*** всегда приводит к попытке создания нового сегмента разделяемой памяти с ключом, который не совпадает со значением ключа ни одного из уже существующих сегментов и не может быть получен с помощью функции ***ftok()*** ни при одной комбинации ее параметров. Параметр ***size*** определяет размер создаваемого или уже существующего сегмента в байтах. В случае если сегмент с указанным ключом уже существует, но его размер не совпадает с указанным в параметре ***size***, констатируется возникновение ошибки. Параметр ***shmflg*** – флаги – играет роль только при создании нового сегмента разделяемой памяти и определяет права различных пользователей при доступе к сегменту, а также необходимость создания нового сегмента и поведение системного вызова при попытке создания. Он является некоторой комбинацией (с помощью операции побитовое или – "|") предопределенных значений и восьмеричных прав доступа.

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

5. 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;

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***.

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.

Варианты индивидуальных заданий

Номер варианта индивидуального задания равен номеру вашего паспорта ***N1*** (только номер без серии), умноженному на число букв ***N2*** вашего имени (по паспорту) по модулю ***16***: ***(N1*N2) mod 19***

0. Создать дерево процессов **1–>2 2–>(3,4) 4–>5 3–>6 6–>7 7–>8.** Это означает, что исходный процесс ***0*** создаёт дочерний процесс ***1,*** который, в свою очередь, создаёт дочерний процесс ***2,*** который, в свою очередь, создаёт дочерние процессы ***3, 4*** и т. д. Процессы непрерывно обмениваются сигналами в следующем порядке: ***1–>2 SIGUSR1 2–>(3,4) SIGUSR2 4–>5 SIGUSR1 3–>6 SIGUSR1 6–>7 SIGUSR1 7–>8 SIGUSR2 8–>1 SIGUSR2***. Запись вида ***2–>(3,4) SIGUSR2*** означает, что процесс ***2*** посылает дочерним процессам ***3,4*** одновременно (т. е. за один вызов ***kill()***) сигнал ***SIGUSR2***. После получения ***101***-го по счету сигнала ***SIGUSR*** родительский процесс посылает сыновьям сигнал ***SIGTERM*** и ожидает завершения всех сыновей, после чего завершается**.** Сыновья, получив сигнал ***SIGTERM,*** завершают работу с выводом на консоль сообщения вида ***pid ppid*** завершил работу после ***X***-го сигнала ***SIGUSR1 и Y***-го сигнала ***SIGUSR2***

где ***X, Y –*** количество посланных за все время работы данным сыном сигналов ***SIGUSR1 и SIGUSR2***

Каждый процесс во время работы выводит на консоль информацию в следующем виде:

N pid ppid послал/получил USR1/USR2 текущее время (мкс)

где ***N –*** номер сына.

1. То же, что и в п. 0 для дерева ***1–>(2,3,4) 2–>(5,6) 6–>7 7–>8*** и последовательности обмена сигналами ***1–>(2,3,4) SIGUSR1 2–>(5,6) SIGUSR2 6–>7 SIGUSR1 7–>8 SIGUSR1 8–>1 SIGUSR2***

2. То же, что и в п. 0 для дерева ***1–>(2,3,4,5) 2–>6 3–>7 4–>8*** и последовательности обмена сигналами ***1–>(2,3,4,5) SIGUSR1 5–>(6,7,8) SIGUSR1 8–>1 SIGUSR1***

3. То же, что и в п. 0 для дерева ***1–>(2,3) 2–>(4,5) 5–>6 6–>(7,8)*** и последовательности обмена сигналами ***1–>(2,3) SIGUSR1 2–>(4,5) SIGUSR1 5–>6 SIGUSR1 6–>(7,8) SIGUSR1 8–>1 SIGUSR1***

4. То же, что и в п. 0 для дерева ***1–>(2,3,4,5) 5–>(6,7,8)*** и последовательности обмена сигналами ***1–>(2,3,4) SIGUSR1 2–>(5,6) SIGUSR2 6–>7 SIGUSR1 7–>8 SIGUSR1 8–>1 SIGUSR2***

5. То же, что и в п. 0 для дерева ***1–>(2,3) 3–>4 4–>(5,6,7) 7–>8*** и последовательности обмена сигналами ***1–>(1,2,3,4,5) SIGUSR1 5–>(6,7,8) SIGUSR1 8–>1 SIGUSR1***

6. То же, что и в п. 0 для дерева ***1–>2 2–>(3,4) 4–>5 3–>6 6–>7 7–>8*** и последовательности обмена сигналами ***1–>(2,3,4) SIGUSR1 2–>(5,6) SIGUSR2 6–>7 SIGUSR1 7–>8 SIGUSR1 8–>1 SIGUSR2***

7. То же, что и в п. 0 для дерева ***1–>(2,3,4,5,6) 6–>(7,8)*** и последовательности обмена сигналами ***1–>(2,3) SIGUSR1 2–>(4,5) SIGUSR1 5–>6 SIGUSR1 6–>(7,8) SIGUSR1 8–>1 SIGUSR1***

8. То же, что и в п. 0 для дерева ***1–>2 2–>(3,4,5) 4–>6 3–>7 5–>8*** и последовательности обмена сигналами ***1–>(2,3,4,5) SIGUSR2 2–>6 SIGUSR1 3–>7 SIGUSR1 4–>8 SIGUSR1 8–>1 SIGUSR1***

9. То же, что и в п. 0 для дерева ***1–>2 2–>3 3–>(4,5,6) 6–>7 4–>8*** и последовательности обмена сигналами ***1–>(2,3) SIGUSR1 3–>4 SIGUSR2 4–>(5,6,7) SIGUSR1 7–>8 SIGUSR1 8–>1 SIGUSR2***

10. То же, что и в п.0 для дерева ***1–>(2,3) 3–>4 4–>(5,6) 6–7 7–>8*** и последовательности обмена сигналами ***1–>(2,3) SIGUSR1 3–>4 SIGUSR2 4–>(5,6,7) SIGUSR1 7–>8 SIGUSR1 8–>1 SIGUSR2***

11. То же, что и в п. 0 для дерева ***1–>2 2–>(3,4,5) 4–>6 3–>7 5–>8*** и последовательности обмена сигналами ***1–>(2,3) SIGUSR1 3–>4 SIGUSR2 4–>(5,6,7) SIGUSR1 7–>8 SIGUSR1 8–>1 SIGUSR2***

12. То же, что и в п. 0 для дерева ***1–>(2,3,4,5,6,7) 2,3,4,5,6,7–>8*** и последовательности обмена сигналами ***1–>(2,3) SIGUSR1 3–>4 SIGUSR2 4–>(5,6,7) SIGUSR1 7–>8 SIGUSR1 8–>1 SIGUSR2***

13. То же, что и в п. 0 для дерева ***1–>2 2–>(3,4,5) 4–>6 3–>7 5–>8*** и последовательности обмена сигналами ***1–>(2,3) SIGUSR1 3–>4 SIGUSR2 4–>(5,6,7) SIGUSR1 7–>8 SIGUSR1 8–>1 SIGUSR2***

14. То же, что и в п. 0 для дерева ***1–>2 2–>3 3–>(4,5,6) 6–>7 4–>8*** и последовательности обмена сигналами ***1–>(2,3) SIGUSR1 3–>4 SIGUSR2 4–>(5,6,7) SIGUSR1 7–>8 SIGUSR1 8–>1 SIGUSR2***

15. То же, что и в п. 0 для дерева ***1–>(2,3,4,5) 2–>(6,7) 7–>8*** и последовательности обмена сигналами ***1–>(2,3) SIGUSR1 3–>4 SIGUSR2 4–>(5,6,7) SIGUSR1 7–>8 SIGUSR1 8–>1 SIGUSR2***.

16. Создать два дочерних процесса. Родительский процесс создаёт семафор (***сем1***) и ***2*** неименованных канала (***кан1 и кан2)***. Оба дочерних процесса непрерывно записывают в каналы по ***110*** строк вида ***номер_строки pid_процесса текущее_время*** (мкс). Родительский процесс читает из каждого канала по ***75*** строк и выводит их на экран. Всего дочерние процессы должны записать по ***1010*** строк. Семафор (***сем1***) используется процессами для разрешения, кому из процессов получить доступ к каналу. Дочерние процессы начинают работу по-

сле получения сигнала ***SIGUSR2*** от родительского процесса. По завершении работы они посылают сигнал ***SIGUSR1*** родительскому процессу.

17. Создать два дочерних процесса. Родительский процесс создаёт семафор (***сем1***) и разделяемую память. Оба дочерних процесса непрерывно записывают в разделяемую память по ***75*** строк вида ***номер_строки pid_процесса текущее_время*** (мкс). 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 работу после получения сигнала ***SIGUSR2*** от родительского процесса. По завершении работы они посылают сигнал ***SIGUSR1*** родительскому процессу.

18. Создать два дочерних процесса. Родительский процесс создаёт семафор (***сем1***) и общий файл. Дочерние процессы записывают в файл по 3 строки за раз всего ***1100*** строк вида ***номер_строки pid_процесса текущее_время*** (мкс). Родительский процесс читает из файла по 3 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. Дочерние процессы начинают работу после получения сигнала ***SIGUSR2*** от родительского процесса. По завершении работы они посылают сигнал ***SIGUSR1*** родительскому процессу.

19. Написать программу, создающую 2 дочерних процесса. Родительский процесс создаёт 2 неименованных канала. Дочерние процессы записывают в канал по 100 строк вида ***номер_строки pid_процесса текущее_время*** (мкс). Родительский процесс читает из канала по 75 строк и выводит их на экран в следующем виде: ***pid строка прочитанная из файла***. Дочерние процессы начинают работу после получения сигнала ***SIGUSR2*** от родительского процесса. По завершении работы они посылают сигнал ***SIGUSR1*** родительскому процессу.

20. 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*** (мкс). 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 работу после получения сигнала ***SIGUSR2*** от родительского процесса. По завершении работы они посылают сигнал ***SIGUSR1*** родительскому процессу.

ЛИТЕРАТУРА

1. Таненбаум, Э. Современные операционные системы / Э. Таненбаум.– 2-е изд. – СПб. : Питер, 2002. – 1040 с.
2. Столингс, В. Операционные системы / В. Столингс. – 3-е изд. – М. : Вильямс, 2002. – 848 с.
3. Руссинович, М. Внутреннее устройство Microsoft Windows : Windows Server 2003, Windows XP и Windows 2000 / М. Руссинович, Д. Соломон. ; пер. с англ. – 4-е изд. – М.: Русская Редакция; СПб. : Питер, 2005. – 992 с.
4. Олифер, В. Г. Сетевые операционные системы: учебник / В. Г. Олифер, Н. А. Олифер. – СПб. : Питер, 2001. – 544 с.
5. Робачевский, А. М. Операционная система Unix/Linux / А. М. Робачевский. – СПб. : BHV – Санкт-Петербург, 1997. – 528 с.
6. Нортон, П. Руководство П. Нортона : MS Windows 2000 Professional / П. Нортон. – М. : Русская Редакция, 2001. – 480 с.
7. Буза, М. К. Операционная среда Windows и ее приложения / М. К. Буза. – Минск : Выш. шк., 1997. – 341 с.
8. Валединский, В. Д. Информатика. Словарь компьютерных терминов / В. Д. Валединский. – М. : Аквариум, 1997. – 226 с.
9. Гордеев, А. В. Системное программное обеспечение / А. В. Гордеев, А. Ю. Молчанов. – СПб. : Питер, 2003. – 736 с.

Св. план 2011, поз. 49

Учебное издание

Алексеев Игорь Геннадиевич

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Методическое пособие
для студентов специальности 1-40 01 02
«Информационные системы и технологии в экономике»
заочной формы обучения

Редактор Н. В. Гриневич
Корректор Е. Н. Батурчик
Компьютерная верстка Ю. Ч. Клочкевич

Подписано в печать 22.09.2011. Формат 60x84 1/16. Бумага офсетная.
Гарнитура «Таймс». Отпечатано на ризографе. Усл. печ. л. 2,09.
Уч.-изд. л. 2,0. Тираж 100 экз. Заказ 126.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
ЛИ №02330/0494371 от 16.03.2009. ЛП №02330/0494175 от 03.04.2009.
220013, Минск, П. Бровки, 6