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

Кафедра программного обеспечения информационных технологий

И. Г. Алексеев, А. П. Занкович

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Лабораторный практикум
для студентов специальности
«Информационные системы и технологии в экономике»
дневной формы обучения

Библиотека БГУИР

Минск БГУИР 2009

УДК 004.451(075.8)
ББК 32.973.26-018.2я73
А47

Р е ц е н з е н т – доцент Института информационных технологий БГУИР, кандидат технических наук В. Н. Мухаметов

Библиотека БГУИР

Алексеев, И. Г.

А47 Операционные системы : лаб. практикум для студ. спец. «Информационные системы и технологии в экономике» днев. формы обуч. / И. Г. Алексеев, А. П. Занкович. – Минск : БГУИР, 2009. – 32 с.
ISBN 978-985-488-336-6

Рассмотрены основные принципы программирования в операционной системе Unix/Linux, методы создания и взаимодействия процессов в операционных системах Windows и Unix/Linux. Содержится описание шести лабораторных работ по курсу «Операционные системы».

УДК 004.451(075.8)
ББК 32.973.26-018.2я73

ISBN 978-985-488-336-6

© Алексеев И. Г., Занкович А. П., 2009
© УО «Белорусский государственный университет информатики и радиоэлектроники», 2009

СОДЕРЖАНИЕ

Краткие теоретические сведения 4

Лабораторная работа №1. Система команд и файловая структура ОС Unix/Linux 5

Лабораторная работа №2. Управление ОС Linux с помощью интерпретатора BASH 11

Лабораторная работа №3. Основные принципы программирования в ОС Unix/Linux 15

Лабораторная работа №4. Процессы и потоки в ОС Unix/Linux 17

Лабораторная работа №5. Процессы и потоки в ОС Windows 22

Лабораторная работа №6. Средства межпроцессного взаимодействия ОС 24

Литература 31

Библиотека БГУИР

КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Одной из основных подсистем операционной системы (ОС), непосредственно влияющей на производительность и функционирование вычислительной машины, является подсистема управления ***процессами и потоками***, которая занимается их созданием, выполнением, уничтожением, поддерживает взаимодействие между ними, а также распределяет процессорное время между несколькими одновременно существующими в системе ***процессами и потоками***.

Современные ОС являются многозадачными. Многозадачность – способ организации вычислительного процесса, при котором одновременно выполняется несколько программ или ***процессов***.

При ***вытесняющей*** многозадачности функции планирования потоков целиком сосредоточены в операционной системе, и каждому потоку для выполнения предоставляется поочередно ограниченный непрерывный период процессорного времени – ***квант***. ***Поток или процесс***, который исчерпал свой ***квант***, переводится в состояние ***готовности*** и ожидает, когда ему будет предоставлен новый ***квант*** процессорного времени, а на выполнение выбирается новый ***поток*** или ***процесс*** из очереди готовых для выполнения.

Выделяемые ***кванты*** времени могут быть одинаковыми или различными для всех ***потоков*** или ***процессов***. Величина ***кванта*** обычно выбирается небольшой (не больше 6–16 миллисекунд), чтобы пользователь не ощущал присутствия в системе одновременно нескольких десятков ***процессов***. Смена активного ***потока*** происходит, если ***поток*** завершился и покинул систему, произошла ошибка, поток перешел в состояние ожидания, исчерпан ***квант*** процессорного времени, отведенный данному ***потоку.***

На рисунке показана упрощенная схема работы системы управления процессами с помощью двух очередей.

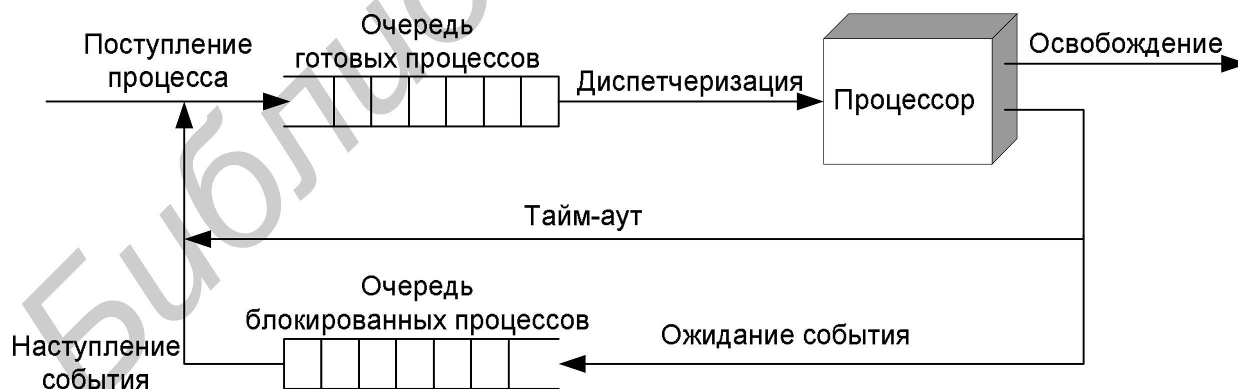

Рис. Реализация системы управления процессами
с помощью двух очередей

Библиотека БГУИР

Лабораторная работа №1

СИСТЕМА КОМАНД И ФАЙЛОВАЯ СТРУКТУРА ОС UNIX/LINUX

Цель работы: изучить команды ОС для работы с файлами, каталогами, дисками, системной датой и временем; текстовый редактор Kate и файловый менеджер Midnight Commander.

Теоретическая часть

Операционная система ***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 нажатии клавиш:
Ctrl + ***Alt*** + ***F1*** *или* ***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

Библиотека БГУИР

Окончание табл. 1

Команда	Аргументы/ключи	Пример	Описание
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 count=1***	Копирование побайтное
ln	Файл1 файл2 ***-l***	***ln файл1 файл2*** ***ln –l файл1 файл2***	Создать жесткую 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*** и другие. Все файловые системы поддерживают ***журналирование***. ***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*** и т. д., 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***.

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

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* – ***flags*** – имеет целочисленный тип и определяет метод доступа. Параметр ***flags*** принимает одно из значений, заданных постоянными в заголовочном файле ***fcnt1.h***. В файле определены три постоянные:

Библиотека БГУИР

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);
```

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 ***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

Библиотека БГУИР

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 и запись. 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 ***Linux*** представлена в табл. 2. Используют следующие сокращения 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

/	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, 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***)

Библиотека БГУИР

Окончание табл. 2

/	Корневой каталог
/tmp	Каталог для временных файлов, хранящих промежуточные данные, необходимых для работы тех или иных программ и удаляющихся 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

Порядок выполнения работы

1. Изучить теоретическую часть лабораторной работы.
2. В консольном режиме, используя команды из табл.1, создать 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***. Используя команду ***man,*** найти в справочной системе 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

Варианты индивидуальных заданий

Во всех заданиях необходимо для чтения или записи файла использовать функции посимвольного ввода – вывода ***fgetc()***, ***fputc( )*** или ***getc()***, ***putc()***. **Должен быть контроль ошибок открытия, закрытия, чтения и записи файла или каталога**. Вывод сообщений об ошибках должен производиться в стандартный поток вывода сообщений об ошибках (***stderr***) в следующем виде: ***имя_модуля: текст_сообщения***. Имя модуля берется из аргументов командной строки.

1. Программа ввода символов с клавиатуры и записи их в файл (имя файла вводится в качестве аргумента при запуске программы). Предусмотреть выход после ввода определенного символа (например ***ctrl-F***).
2. Программа просмотра текстового файла и вывода его содержимого на экран (имя файла передается в качестве аргумента при запуске программы, второй аргумент ***N*** устанавливает вывод по группам строк (по ***N*** строк) или сплошным текстом (***N*** = 0)).

Библиотека БГУИР

3. Программа копирования одного файла в другой, имена файлов передаются в качестве аргументов командной строки при запуске программы. Предусмотреть копирование прав доступа к файлу.

4. Программа подсчета числа отображаемых символов в строках текстового файла. Результаты подсчета записываются во второй текстовый файл (имена файлов передаются в качестве аргументов командной строки при запуске программы). Пример работы программы:
исходный текстовый файл из трех строк:

QWER
REEEt
WEEEEEEERSIIIONN

файл, полученный в результате работы программы:

1. 4
2. 15
3. 16
итого: 3 строки 35 символов

5. Программа подсчета числа слов в текстовом файле. Результаты подсчета записываются во второй текстовый файл (имена файлов передаются в качестве аргументов командной строки при запуске программы). Пример вывода программы для текстового файла:

QWER REEEt
WE E EEE EER SI I IO NN

файл, полученный в результате работы программы:

1. 2 слова
2. 8 слов
итого: 3 строки 10 слов

6. Программа, подсчитывающая количество символов с одинаковыми кодами ASCII в текстовом файле. Результаты подсчета записываются в другой текстовый файл (имена файлов передаются в качестве аргументов командной строки при запуске программы). Пример вывода программы для текстового файла:

QWER REEEt
WE E EEE EER SI I IO NN

файл, полученный в результате работы программы:

1. Q код ASCII 41 =1
2. W код ASCII 42 =2
3. E код ASCII 42 =11
…
итого: 25 символов

Библиотека БГУИР

Лабораторная работа №2

УПРАВЛЕНИЕ ОС LINUX С ПОМОЩЬЮ ИНТЕРПРЕТАТОРА BASH

Цель работы: исследовать 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

Командным интерпретатором используются следующие переменные:

$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 передается 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) 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
       …
       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
    …
```

Библиотека БГУИР

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

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

4. Написать скрипт, компилирующий и запускающий программу (имя исходного файла и ***exe-***файла результата задается пользователем в качестве аргу-

Библиотека БГУИР

ментов 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 задается пользователем в качестве третьего аргумента командной строки). Диапазон (min – max) размеров файлов задается пользователем в качестве первого и второго аргументов 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 подкаталогах (строка и имя каталога задаются пользователем в качестве первого и второго аргументов 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 содержимому файлов в двух каталогах, например ***Dir1*** и ***Dir2***. Пользователь задает имена ***Dir1*** и ***Dir2*** в качестве первого и второго аргументов 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 его подкаталогах все файлы, владельцем которых является заданный пользователь. Имя владельца и каталог задаются пользователем в качестве первого и второго аргументов командной строки. Скрипт выводит результаты в файл (третий аргумент командной строки) в следующем виде: полный путь, имя файла, его размер. На консоль выводится общее число просмотренных файлов.

6. Написать скрипт, находящий в заданном каталоге и его подкаталогах все файлы заданного размера (имя каталога задается пользователем в качестве первого аргумента командной строки). Диапазон (min – max) размеров файлов задается пользователем в качестве второго и третьего аргументов командной строки. Скрипт выводит результаты поиска в файл (четвертый аргумент командной строки) в следующем виде: полный путь, имя файла, его размер. На консоль выводится общее число просмотренных файлов.

7. Написать скрипт, подсчитывающий суммарный размер файлов в заданном каталоге и всех его подкаталогах (имя каталога задается пользователем в ка-

Библиотека БГУИР

честве первого аргумента командной строки). Скрипт выводит результаты подсчета в файл (второй аргумент командной строки) в следующем виде: каталог (полный путь), суммарный размер файлов, число просмотренных файлов.

Лабораторная работа №3

ОСНОВНЫЕ ПРИНЦИПЫ ПРОГРАММИРОВАНИЯ В ОС LINUX

Цель работы: изучить файловую систему ОС ***Linux*** и основных функций для работы с каталогами и файлами.

Теоретическая часть

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 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 функции ***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, используется функция:

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

Библиотека БГУИР

Порядок выполнения работы

1. Изучить теоретическую часть работы.
2. Написать программу вывода на экран содержимого заданного пользователем каталога. Вывести с использованием программы содержимое текущего и корневого каталогов. Предусмотреть контроль ошибок открытия, закрытия, чтения каталога. Вывод сообщений об ошибках должен производиться в стандартный поток вывода сообщений об ошибках (***stderr***) в следующем виде: ***имя_модуля текст_сообщения***.

Варианты индивидуальных заданий

Должен быть контроль ошибок для всех операций с файлами и каталогами.

1. Отсортировать в заданном каталоге (аргумент 1 командной строки) и во всех его подкаталогах файлы по следующим критериям (аргумент 2 командной строки, задается 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 пределах от ***N1*** до ***N2*** (***N1, N2*** задаются в аргументах командной строки), 2 – дата создания находится в 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 нем, суммарный размер файлов, имя файла с наибольшим размером.
6. Написать программу, находящую в заданном каталоге и его подкаталогах все файлы заданного размера (имя каталога задается пользователем в качестве первого аргумента командной строки). Диапазон (min – max) размеров файлов задается пользователем в качестве второго и третьего аргументов командной строки. Программа выводит результаты поиска в файл (четвертый аргумент командной строки) в следующем виде: полный путь, имя файла, его размер. На консоль выводится общее число просмотренных файлов.

Библиотека БГУИР

Лабораторная работа №4

ПРОЦЕССЫ И ПОТОКИ В ОС LINUX

Цель работы: исследовать методы создания процессов в ОС ***Linux***, основные функции создания и управления процессами, обмен данными между процессами.

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 создает 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 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”);
}
```

Библиотека БГУИР

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 любой дочерний процесс, чей идентификатор группы процессов равен абсолютному значению *pid*.

***pid* = –1** означает, что нужно ожидать любой дочерний процесс; функция **wait** ведет себя точно так же.

***pid* = 0** означает, что нужно ожидать любой дочерний процесс, чей идентификатор группы процессов равен таковому у текущего процесса.

pid* > 0** означает, что нужно ожидать дочерний процесс, 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 асинхронный сбор информации о статусе завершившихся порожденных процессов процессом-родителем.

Для перегрузки исполняемой программы можно использовать функции семейства ***exec***. Основное различие между функциями в семействе ***exec*** 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***. Поскольку традиционный процесс в концепции нитей исполнения трактуется как процесс, содержащий единственную ***нить*** исполнения, можно 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***, и передает 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 в следующих случаях:

- 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 завер-

Библиотека БГУИР

шившийся ***thread*** либо при выходе из ассоциированной с ним функции, либо при выполнении функции ***pthread_exit()***. Если пользователя не интересует, что вернула ему нить исполнения, то 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 ***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 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 (для дочерних процессов) 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.
3. В основной программе создать два потока. После этого процесс-отец создает файл, записывает в него строки вида: ***N pid,*** текущее время в формате ***часы: минуты: секунды: миллисекунды*** (где ***N*** – номер выводимой строки) и выводит формируемые строки в левой половине экрана. Количество записываемых в файл строк ***k* = 100**. Оба потока читают строки из файла и выводят их в правой части экрана в виде ***поток id*** потока ***текущее время*** (миллисекунды (мсек)) ***строка из файла***

Варианты индивидуальных заданий

1. Разработать программу по условию п. 3 порядка выполнения работы, но вместо потоков создать 3 процесса, которые осуществляют те же функции, что и в п. 3 (читают строки из файла и выводят их в правой части экрана), в виде ***процесс pid*** процесса ***текущее время*** (миллисекунды (мсек)) ***строка из файла***.

Библиотека БГУИР

2. 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 задает значения ***K***, ***N*** и количество ***n*** членов ряда Тейлора. Для расчета каждого члена ряда Тейлора запускается отдельная нить. Каждая нить выводит на экран свой ***id*** и рассчитанное значение ряда. Головной процесс суммирует все члены ряда Тейлора и полученное значение ***y[i]*** записывает в файл.

3. Написать программу синхронизации двух каталогов, например ***Dir1*** и ***Dir2***. Пользователь задает 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 с использованием функции ***fork()*** в отдельном процессе для каждого копируемого файла. Каждый процесс выводит на экран свой ***pid,*** имя копируемого файла и число скопированных байт. Число запущенных процессов не должно превышать ***N*** (вводится пользователем).

4. Написать программу поиска одинаковых по содержимому файлов в двух каталогов, например ***Dir1*** и ***Dir2***. Пользователь задает 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 не должно превышать ***N*** (вводится пользователем).

5. Написать программу поиска заданной пользователем комбинации из ***m*** байт (***m*** **< 255**) во всех файлах текущего каталога. Пользователь задает 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, число просмотренных байт и результаты поиска. Число запущенных процессов не должно превышать ***N*** (вводится пользователем).

6. 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()***. Предусмотреть контроль ошибок.

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

номер процесса/потока pid ppid текущее время (миллисекунды (мсек)).

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

Библиотека БГУИР

Лабораторная работа №5

ПРОЦЕССЫ И ПОТОКИ В ОС WINDOWS

Цель работы: изучить методы создания процессов и потоков в ОС ***Windows***, основные функции управления процессами и потоками, обмен данными между процессами и потоками.

Теоретическая часть

Для создания процесса в ***Windows*** используется функция

BOOL CreateProcess(
LPCTSTR lpApplicationName, // имя исполняемого модуля
LPTSTR lpCommandLine, // командная строка
LPSECURITY_ATTRIBUTES lpProcessAttributes, // указатель на структуру
LPSECURITY_ATTRIBUTES lpThreadAttributes, // указатель на структуру
BOOL bInheritHandles, // флаг наследования текущего процесса
DWORD dwCreationFlags, // флаги способов создания процесса
LPVOID lpEnvironment, // указатель на блок среды
LPCTSTR lpCurrentDirectory, // текущий диск или каталог
LPSTARTUPINFO lpStartupInfo, // указатель на структуру ***STARTUPINFO***
LPPROCESS_INFORMATION lpProcessInformation // указатель на структуру
);

Основным параметром является имя исполняемого модуля, соответствующее имени программы, запускаемой во вновь созданном процессе. Пример:

```
void main()
{
    STARTUPINFO si;
    ZeroMemory(&si,sizeof(STARTUPINFO));
    PROCESS_INFORMATION pi;
    CreateProcess("c:\\1.exe",null,null,null,false,null,null,nulL,&si,&pi)
```

Для создания потока (нити) в Windows используется функция

HANDLE CreateThread(
LPSECURITY_ATTRIBUTES lpThreadAttributes, // атрибуты защиты
DWORD dwStackSize, // размер стека
LPTHREAD_START_ROUTINE lpStartAddress, // адрес функции потока
LPVOID lpParameter, // параметр, который будет передан функции потока
DWORD dwCreationFlags, // 0 – исполнение начинается немедленно
LPDWORD lpThreadId // идентификатор ***id*** нового потока
);

При каждом вызове этой функции система создает объект ядра (поток) и выделяет память под стек потока из адресного пространства процесса. Новый поток выполняется в контексте того же процесса, что и родительский поток. Он получает доступ ко всем описателям объектов ядра, всей памяти и стекам всех

Библиотека БГУИР

потоков в процессе. За счет этого потоки в рамках одного процесса могут легко взаимодействовать друг с другом.

Для передачи сообщений используется специальная ***win32***-функция:

```
LRESULT SendMessage(  HWND hWnd,   // handle окна
  UINT Msg,                        // посылаемое сообщение
  WPARAM wParam,                   // первый параметр сообщения
  LPARAM lParam                    // второй  параметр сообщения
);
```

Выяснив ***Window handle*** нужного окна, пользователь может посылать ему ***Windows Messages***.

Порядок выполнения работы

1. Написать программу, запускающую в новом процессе программу ***notepad***.
2. Написать программу, запускающую новый поток в текущем процессе. В потоке выполняется функция, выводящая каждые 50 мс на экран следующее сообщение: ***номер_сообщения id_потока текущее_время (сек:мсек)***.
3. Написать программу ***A***, запускающую в новом процессе программу ***B***. Обе программы должны посылать друг другу сообщения – ***Windows message***.

Варианты индивидуальных заданий

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

   номер процесса/потока pid текущее время (миллисекунды (мсек)).

   Время ***t =((номер процесса/потока по дереву)*200*** (миллисекунды (мсек)).
2. Написать программу, выводящую на экран список запущенных процессов и потоков и их атрибуты (***pid*** и др.) в виде таблицы.
3. 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 задает значения ***K***, ***N*** и количество ***n*** членов ряда Тейлора. Для расчета каждого члена ряда Тейлора запускается отдельный поток. Каждый поток выводит на экран свой ***pid*** и рассчитанное значение ряда. Головной процесс суммирует все члены ряда Тейлора и полученное значение ***y*[*i*]** записывает в массив и в файл пользователя.
4. Написать программу синхронизации двух каталогов, например ***Dir1*** и ***Dir2***. Пользователь задает имена ***Dir1*** и ***Dir2***. В результате работы программы файлы, имеющиеся в ***Dir1***, но отсутствующие в ***Dir2***, должны скопироваться в ***Dir2*** вместе. Процедуры копирования должны запускаться в отдельном потоке для каждого копируемого файла. Число запущенных потоков не должно превышать ***N*** (вводится пользователем). Каждый поток выводит на экран свой ***id,*** имя копируемого файла и число скопированных байт.

Библиотека БГУИР

5. Написать программу поиска одинаковых по содержимому файлов в двух каталогах, например ***Dir1*** и ***Dir2***. Пользователь задает 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 потоке выводит на экран свой ***id***, имя файла, число просмотренных байт и результаты сравнения. Число запущенных потоков не должно превышать ***N*** (вводится пользователем).

6. Написать программу поиска заданной пользователем комбинации из ***m*** байт (***m*** < 255) во всех файлах текущего каталога. Пользователь задает 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, число просмотренных байт и результаты поиска. Число запущенных потоков не должно превышать ***N*** (вводится пользователем).

Лабораторная работа №6

СРЕДСТВА МЕЖПРОЦЕССНОГО ВЗАИМОДЕЙСТВИЯ ОС

Цель работы: изучить методы и средства взаимодействия процессов в ОС ***Linux***.

Теоретическая часть

Все процессы в ***Linux*** выполняются в раздельных адресных пространствах. Для организации межпроцессного взаимодействия необходимо использовать специальные методы:

1) общие файлы;
2) общую или разделяемую память;
3) очереди сообщений (***queue***);
4) сигналы (***signal***);
5) каналы (***pipe***);
6) 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*** 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

Библиотека БГУИР

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 процессы. Он состоит из комбинации следующих бит:

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.

2. 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 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 образом для рассматриваемых средств межпроцессного взаимодействия. 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 компиляции программы необходимо подключить библиотеку ***rt.lib*** следующим способом: ***gcc 1.c –o 1.exe -lrt***

3. Очереди сообщений (***queue***) являются более сложным методом связи взаимодействующих процессов по сравнению с программными каналами.

Библиотека БГУИР

С помощью очередей можно из одной или нескольких задач независимым образом посылать сообщения некоторой задаче-приемнику. При этом только процесс-приемник может читать и удалять сообщения из очереди, а процессы-клиенты имеют право лишь помещать в очередь свои сообщения. Очередь работает только в одном направлении, если необходима двухсторонняя связь, следует создать две очереди. Очереди сообщений предоставляют возможность использовать несколько дисциплин обработки сообщений:

- ***FIFO*** – сообщение, записанное первым, будет прочитано первым;
- ***LIFO*** – сообщение, записанное последним, будет прочитано первым;
- приоритетная – сообщения читаются с учетом их приоритетов;
- произвольный доступ – можно читать любое сообщение, а программный канал обеспечивает только дисциплину ***FIFO***.

Для открытия очереди служит функция ***mq_open()***, которая по аналогии с файлами создает описание открытой очереди и ссылающийся на него дескриптор типа ***mqd_t***, возвращаемый в качестве нормального результата.

```
#include <mqueue.h>
mqd_t mq_open ( const char *name, int oflag, ...);
```

При чтении сообщение из очереди не удаляется, т. е. оно может быть прочитано несколько раз. В очередях реально присутствуют не сами сообщения, а только их адреса в памяти и размеры. Эта информация размещается системой в сегменте памяти, доступном для всех задач, общающихся с помощью данной очереди. Каждый процесс, использующий очередь, должен предварительно получить разрешение на использование общего сегмента памяти с помощью системных запросов, потому что очередь – системный механизм, и для работы с ней требуются системные ресурсы и обращение к самой ОС.

4. 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 на 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** – посылающий процесс не является процессом суперпользователя, поэтому сигнал посылается всем процессам в системе, для которых иденти-

Библиотека БГУИР

фикатор пользователя совпадает с эффективным идентификатором пользователя процесса, посылающего сигнал.

pid **= –1** – посылающий процесс является процессом суперпользователя, поэтому 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 –** сигнал посылается всем процессам из группы, идентификатор которой равен абсолютному значению аргумента ***pid*** (если позволяют привилегии).

sig **= 0** – 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);

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); }
```

5. 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.* 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.* Данным каналом могут пользоваться только создавший его процесс и его потомки. Для создания используется вызов:

Библиотека БГУИР

int pipe(int fd[2]);

6. 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.

1. Создание и получение доступа к набору семафоров:

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

Параметр ***key*** 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 массив для указанного ключа не существует, то массив 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;

Библиотека БГУИР

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***

2. 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 создает семафор (***сем1***) и общий файл. Дочерний процесс записывает в файл по одной строке всего **100** строк вида ***номер_строки pid_процесса текущее_время*** (миллисекунды (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 создает неименованный канал. Дочерний процесс записывает в канал 100 строк вида ***номер_строки pid_процесса текущее_время*** (миллисекунды

Библиотека БГУИР

(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 создает 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*** (миллисекунды (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 создает 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*** (миллисекунды (мс)). 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 создает семафор (***сем1***) и разделяемую память. Оба дочерних процесса непрерывно записывают в разделяемую память по **100** строк вида ***номер_строки pid_процесса текущее_время*** (миллисекунды (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.

Библиотека БГУИР

ЛИТЕРАТУРА

Основная

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. : Рус. Ред.; СПб. : Питер, 2005. – 992 с.

4. Олифер, В. Г. Сетевые операционные системы : учеб. / В. Г. Олифер, Н. А. Олифер. – СПб. : Питер, 2001. – 544 с.

Дополнительная

5. Робачевский, А. М. Операционная система Unix / А. М. Робачевский. – Спб. : BHV – Санкт-Петербург, 1997. – 528 с.

6. Нортон, П. Руководство П. Нортона : MS Windows 2000 Professional / П. Нортон. – М. : Рус. Ред., 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.

Библиотека БГУИР

Св. план 2008, поз. 51

Учебное издание

Алексеев Игорь Геннадиевич
Занкович Артем Петрович

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Лабораторный практикум
для студентов специальности
«Информационные системы и технологии в экономике»
дневной формы обучения

Библиотека БГУИР

Редактор Е. Н. Батурчик
Корректор Л. А. Шичко
Компьютерная верстка Л. А. Шичко

Подписано в печать 2.02.2009. Формат 60×84 1/16. Бумага офсетная.
Гарнитура «Таймс». Печать ризографическая. Усл. печ. л. 1,98.
Уч.-изд. л. 1,8. Тираж 100 экз. Заказ 329.

Издатель и полиграфическое исполнение: Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
ЛИ №02330/0056964 от 01.04.2004. ЛП №02330/0131666 от 30.04.2004.
220013, Минск, П. Бровки, 6