# 操作系统实验指导

# 

目 录

[操作系统实验指导 I](#_Toc475625507)

[实验一 进程控制 1](#_Toc475625508)

[一、 实验目的 1](#_Toc475625509)

[二、 实验内容 1](#_Toc475625510)

[三、 预备知识 1](#_Toc475625511)

[1. Linux文件编辑 1](#_Toc475625512)

[2. 编译C程序 5](#_Toc475625513)

[3. Linux进程管理命令 8](#_Toc475625514)

[4. 进程控制函数 9](#_Toc475625515)

[5. 管道通信 13](#_Toc475625516)

[四、 实验指导 15](#_Toc475625517)

[实验二 线程同步与通信 16](#_Toc475625518)

[一、 实验目的 16](#_Toc475625519)

[二、 实验内容 16](#_Toc475625520)

[三、 预备知识 16](#_Toc475625521)

[1. 信号灯及其P、V操作 16](#_Toc475625522)

[2. 线程及线程分类 20](#_Toc475625523)

[四、 实验指导 22](#_Toc475625524)

[实验三、共享内存与进程同步 24](#_Toc475625525)

[一、 实验目的 24](#_Toc475625526)

[二、 实验内容 24](#_Toc475625527)

[三、 预备知识 24](#_Toc475625528)

[1. 共享内存 24](#_Toc475625529)

[2. 环形缓冲 28](#_Toc475625530)

[四、 实验指导 28](#_Toc475625531)

[实验四、Linux文件目录 29](#_Toc475625532)

[一、 实验目的 29](#_Toc475625533)

[二、 实验内容 29](#_Toc475625534)

[三、 预备知识 29](#_Toc475625535)

[1. 认识文件 29](#_Toc475625536)

[2. Linux文件属性接口 32](#_Toc475625537)

[3. Linux目录结构接口 33](#_Toc475625538)

[四、 实验指导 34](#_Toc475625539)

# 实验一 进程控制

## 实验目的

1、加深对进程的理解,进一步认识并发执行的实质；

2、分析进程争用资源现象,学习解决进程互斥的方法；

3、掌握Linux进程基本控制；

4、掌握Linux系统中的软中断和管道通信。

## 实验内容

编写程序，演示多进程并发执行和进程软中断、管道通信：

1. 父进程使用系统调用pipe( )建立一个管道，然后使用系统调用fork()创建两个子进程，子进程1和子进程2；
2. 子进程1每隔1秒通过管道向子进程2发送数据：

I send you x times. (x初值为1，每次发送后做加一操作）

子进程2从管道读出信息，并显示在屏幕上。

1. 父进程用系统调用signal()捕捉来自键盘的中断信号（即按Ctrl+C键）；当捕捉到中断信号后，父进程用系统调用Kill()向两个子进程发出信号，子进程捕捉到信号后分别输出下列信息后终止：

Child Process l is Killed by Parent!

Child Process 2 is Killed by Parent!

1. 父进程等待两个子进程终止后，释放管道并输出如下的信息后终止：

Parent Process is Killed!

## 预备知识

### Linux文件编辑

Linux的文本编辑器有很多，比如图形模式的gedit、kwrite、OpenOffice等，文本模式下的vi、vim（vi的增强版本）和nano等。vi或vim是Unix/Linux最基本的文本编辑工具，几乎所有的Unix/Linux发行版本都提供这一编辑器。Vi是全屏幕编辑器，工作在字符模式下，只能编辑字符，不能对字体、段落等进行排版。由于不需要图形界面，它们成了效率很高的文本编辑器。

vim的全称是vi Improved，顾名思义是Unix上流行编辑器vi的模仿和改进版，比vi更容易使用。vi的命令几乎全都可以在vim上使用。Vim的理念是减少使用鼠标、减少敲击键盘、减少手指移动、减少目光移动，相比于vi，vim拥有更多的特性：

* vim 具有程序编辑的能力，可以以字体颜色辨别语法的正确性，方便程序设计；因为程序简单，编辑速度相当快速。
* vim可以当作vi的升级版本，他可以用多种颜色的方式来显示一些特殊的信息。
* vim会依据文件扩展名或者是文件内的开头信息， 判断该文件的内容而自动的执行该程序的语法判断式，再以颜色来显示程序代码与一般信息。
* vim里面加入了很多额外的功能，例如支持正则表达式的搜索、多文件编辑、块复制等等。 这对于在Linux上进行一些配置文件的修改工作是很棒的功能。

vi或vim最大的优势在于，它最常用的命令都是简单的字符，这比起使用复杂的控制组合键要快得多，而且也解放了手指的大量工作，学习使用这些命令的时间很快就能从由此带给你的高效率中得到回报。另外，与Vi不同，Vim也支持在插入模式下使用上下箭头键, 这使初学者可以很容易上手。

基本上vi或vim可以分为三种状态，分别是命令模式、编辑模式和末行模式。不同工作模式下的操作方式有所不同。

（1）命令模式

以vi打开一个文件就直接进入命令模式了(这是默认的模式)。在这个模式中，从键盘上输入的任何字符都被当作编辑命令来解释，而不会显示在屏幕上。如果输入的字符是合法的vi命令，则vi完成相应的动作；否则vi会响铃警告。如可以使用上下左右按键来移动光标，可以使用删除字符或删除整行来处理文件内容，也可以使用复制、粘贴来处理文件数据。

（2）编辑模式

在命令模式中可以进行删除、复制、粘贴等的操作，但是却无法编辑文件的内容，只有当到你按下【i, I, o, O, a, A, r, R】等任何一个字母之后才会进入编辑模式。这时候屏幕的左下方会出现【INSERT或 REPLACE】的字样，此时输入的任何字符都被vi当作文件内容显示在屏幕上。而如果要回到命令模式时， 则必须要按下【Esc】即可退出编辑模式。

（3）末行模式

在命令模式下，按【：】键进入末行模式，此时vi会在屏幕的底部显示“：”符号作为命令行模式的提示符，等待用户输入相关命令。命令执行完毕后，vi自动回到命令模式。

为了实现跨平台操作并兼容不同类型的键盘，vi编辑器中无论是命令还是输入内容都使用字母键。例如按字母键“i”在编辑模式下表述输入“i”字母，而在命令模式下则表示将工作模式转换为编辑模式。

vi的三种工作模式之间的相互转换关系如图 1所示。

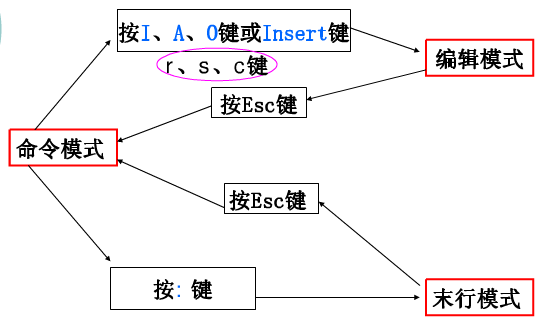


图 1 vi工作模式间的切换

常用的命令操作主要包括：

（1）插入文本命令

只有在编辑模式下才可以插入文本：

i：在光标前

I：在光标所处行行首

a：在光标后

A：在光标所处行行尾

o：在当前行之下新开一行

O：在当前行之上新开一行

Insert：等同于i命令

（2）替换文本命令

r：替换当前字符（一对一）

R：替换当前字符及其后的字符，直至按ESC键（一对一）

s：删除当前字符，并以输入文本替代之（多对一）

S：删除当前行，并以输入文本替代之（替换行）

C：删除当前行，并以输入文本替代之（修改行）

（3）移动光标

在命令模式下，可以利用空格键、退格键、回车键移动光标。此外，输入‘w’以单词为单位向后移动光标；输入‘b’以单词为单位向前移动光标。光标停在下一个或上一个单词的首字母。而在编辑模式下，主要使用键盘上的方向键。

（4）删除文本

在命令模式下删除文本：

x：删除光标处字符

dd：删除整行，包括回车

X：删除光标左边字符

D：删除正行或右边部分

S：删除整行，并进入插入模式

在末行模式下删除文本：

:d （删除当前行）

:nd （删除第n行）

:n1,n2d （删除从n1行开始到n2行为止的所有内容）

（5）复制、剪切和粘贴文本

在编辑模式下只能复制和粘贴，不能剪贴。

在命令模式下：

yy：拷贝当前行文本

P：粘贴文本

dd：删除当前行，执行p，可以粘贴被删除的行

（6）保存、退出

在末行模式下输入命令：

:w （保存文件，:w 文件名）

:q （退出vi，修改过，则不能退出）

:q! （强制退出vi，不保存修改过的文件）

:wq （保存并退出vi，:wq 文件名）

一些常用的编辑软件通常都有恢复功能，即当系统因为某些原因而导致类似当机的情况时，可以利用这个恢复功能将之前未保存的数据找回。vim也有这个功能。当在使用vim编辑时，vim会在与被编辑的文件的目录下，再建立一个名为.filename.swp的文件（执行ls –l命令可以看到）。如果系统因为某些原因断线了，导致编辑的文件还没有保存，这时.filenam.swp 就能发挥救援的功能。

### 编译C程序

Linux的绝大多数应用都用C语言编写，几乎每位Linux程序员面临的首要任务都是灵活运用C编译器。目前Linux下最常用的编译器是GCC。GCC是GNU Compiler Collection的简称，能把高级语言编写的源代码构建成可执行的二进制代码。GCC能支持各种不同的目标体系结构。如它既支持基于宿主的开发，也支持交叉编译。GCC支持的体系结构常见的有X86系列、Arm、PowerPC等。同时GCC还能运行在不同的操作系统上，如Linux、Solaris、Windows等。GCC除了支持C语言外，还支持多种其他语言，能够编译用C、C++和Object C等语言编写的程序，也可以通过不同的前端模块来支持各种语言，如Java、Fortran、Pascal、Modula-3和Ada等，编译、连接成可执行文件。

在使用GCC编译程序时，编译过程被细分为预处理（Pre-Processing）、编译（Cmpiling）、汇编（Assembling）和链接（Linking）四个阶段。预处理阶段主要是在库中寻找头文件，包含到待编译的文件中，编译阶段检查文件的语法，汇编阶段将源代码翻译成机器语言，而在链接阶段则是将所有代码连接成一个可执行程序。

程序员可以根据自己的需要让GCC在编译的任何阶段结束，以便灵活的控制整个编译过程，最常用的有编译模式和编译连接模式两种。一个程序的源代码通常包含在多个源文件之中，这就需要同时编译多个源文件，并将它们连接成一个可执行程序，这时就要采用编译连接模式。在生成可执行程序时，一个程序的源文件无论是一个还是多个，所有被编译和连接的源文件中必须有且仅有一个main函数，因为main函数是该程序的入口点。但在把源文件编译成目标文件时，不需要进行连接，这时main函数不是必需的。

而当调用GCC时，GCC根据文件扩展名（后缀）自动识别文件的类别并调用对应的编译器。GCC遵循的部分后缀约定规则如表 1所示。

表 1 GCC遵循的部分约定规则

|  |  |
| --- | --- |
| **后缀** | **约定规则** |
| .c | C语言源代码文件 |
| .a | 由目标文件构成的档案库文件 |
| .C .cc .cxx | C++源代码文件 |
| .h | 程序包含的头文件 |
| .i | 已经预处理过的C源代码文件 |
| .ii | 已经预处理过的C++源代码文件 |
| .m | Objective-C源代码文件 |
| .o | 编译后的目标文件 |
| .s | 汇编语言源代码文件 |
| .S | 经过预编译的汇编语言源代码文件 |

在使用GCC编译器时，必须给出一系列必要的调用参数和文件名称。GCC最基本的用法是∶gcc [options] [filenames]。

其中options为编译器所需要的编译选项，filenames给出要编译的文件名称。GCC编译器的调用参数大约有100多个，其中多数参数不会用到，最基本、最常用的参数有：

-c，只编译，不连接成为可执行文件。编译器只是根据输入的.c等源代码文件生成.o为后缀的目标文件，通常用于编译不包含主程序的子程序文件。

-o output\_filename，确定输出文件的名称为output\_filename，该名称不能和源文件同名。如果没有该选项，默认生成可执行文件a.out。

-Idirname，指定头文件的查找目录。将dirname指定的目录加入到程序头文件目录列表中，在预编译过程中使用。

-Ldirname，指定库文件的查找目录。将dirname对应的目录加入到程序函数档案库文件的目录列表中，在连接过程中使用。

-lname，在连接过程中，加载名为“libname.a“的函数库（位于系统预设的目录或者由-L选项确定的目录下）。

-Wall，编译文件时发出所有警告信息。

-w：编译文件时不生成任何警告信息。

和其它常用的编译器一样，GCC提供了灵活而强大的代码优化功能，利用它可以生成执行效率更高的代码。GCC还对标准的C和C++语言进行了大量的扩展，提高程序的执行效率，进行代码优化，减轻编程的工作量。常见的调试、优化参数包括：

-g，产生符号调试工具(GNU的gdb)必要的符号信息，在对源代码进行调试时加入该选项。

-O，在编译、连接过程中进行优化处理，从而提高可执行文件的执行效率，但编译、连接的速度就相应地要慢一些。

-O2，比-O更好的进行编译、连接的优化，因此整个编译、连接过程会更慢。

GCC还包含完整的出错检查和警告提示功能，帮助程序员写出更加专业、优美的代码。如当GCC在编译不符合ANSI/ISO C语言标准的源代码时，可以使用“-pedantic”选项在使用了扩展语法的地方产生相应的警告信息。但“-pedantic”选项不能保证被编译程序与ANSI/ISO C标准的完全兼容，它只能帮助程序员发现一些不符合 ANSI/ISO C标准的代码，但不是全部，即离完全兼容的目标越来越近。

除了“-pedantic”之外，GCC还有一些其它编译选项也能产生有用的警告信息。这些选项大多以-W开头，其中最有价值的是“-Wall”选项，使用它能产生尽可能多的警告信息。虽然严格来讲，GCC给出的警告信息不能算作错误，但却很可能成为错误的栖身之所，因此应该尽量避免产生警告信息，使代码始终保持简洁、优美和健壮的特性。

另一个常用的编译选项是“-Werror”，它要求GCC将所有的警告当成错误进行处理。如果编译时带上“-Werror”选项，那么GCC会在所有产生警告的地方停止编译，迫使程序员对自己的代码进行修改。只有当相应的警告信息消除后，才能继续编译过程。

GCC给出的警告信息不仅可以帮助程序员写出更加健壮的程序，而且还是跟踪和调试程序的有力工具。因此，建议在利用GCC编译源代码时，始终带上“-Wall”选项，并尽可能减少警告信息的产生。把它逐渐培养成为一种编程习惯，这将对寻找常见的隐式编程错误很有帮助。

### Linux进程管理命令

Linux系统的Shell作为操作系统的外壳，是用户和Linux内核之间的接口（即为用户提供使用操作系统的接口），它是命令语言、命令解释程序集程序设计语言的统称。当从Shell向Linux传递命令时，内核会做出相应的反应。从用户登录到用户注销的整个期间，用户输入的每个命令都要经过Shell的解释才能执行。

表 2 Shell可执行的用户命令

|  |  |  |
| --- | --- | --- |
| **命令类型** | **功 能** | |
| **内置命令** | **为提高执行效率，部分最常用命令的解释器构筑于Shell内部** | |
| **实用程序** | Linux程序 | **存放在/bin、/sbin目录下Linux自带的程序** |
| 应用程序 | **存放在/usr/bin、/usr/sbin等目录下的应用程序** |
| Shell脚本 | **用Shell语言编写的脚本程序** |
| **用户程序** | **用户编写的其他可执行程序** |

Shell可执行的用户命令可分为两大类：内置命令和实用程序，其中实用程序又分为四类，如表 2所示。

Linux进程管理命令主要有：

* + 1. 进程查看命令
* ps命令：报告进程标识、用户、CPU时间消耗及其他属性

该命令单独使用可以看到前台执行的进程；后台进程可以使用带参数的ps命令（如ps -ax）；提供进程的一次性查看，结果不连续；结果数据很精确，但数据量庞大。

* top命令：显示CPU占用率为前几位的进程

该命令动态显示，输出结果连续，但需要消耗较多的系统资源。

* pstree命令：列出当前的进程，以及它们的树状结构

该命令将当前的执行程序以树状结构显示，弥补ps命令的不足；支持指定特定程序(PID)或使用者(USER)作为显示的起始。

* + 1. 进程终止命令
* kill命令：根据PID向进程发送信号，缺省操作是停止进程

如果进程启动了子进程，只终止父进程，子进程运行中将仍消耗资源成为“僵尸”进程，可用kill -9强制终止退出。

* pkill命令：终止同一进程组内的所有进程。允许指定要终止的进程名称，而非PID
* Killall命令：与pkill应用方法类似，直接杀死运行中的程序

要注意的是，数据库服务器的父进程不能用这些命令杀死（容易产生更多的文件碎片导致数据库崩溃）。

### 进程控制函数

在Linux系统中，有两个基本操作用于创建和修改进程：函数fork( )用来创建一个新的进程，该进程几乎是当前进程的一个完全拷贝；函数族exec( )用来启动另外的程序以取代当前运行的进程。

（1）进程创建

Linux中创建新进程的唯一方法是使用fork( )函数。对于没有接触过Linux操作系统的人来说，fork是最难理解的概念之一：C语言中，if else分支语句只会根据条件进入一个分支执行，但fork( )执行一次却有两个返回值（if、else两个分支的语句都执行了）。例如，图 2是一个最基本的fork( )函数调用程序，该程序执行的结果如图 3所示。

|  |
| --- |
| #include <stdio.h>  void main( ) {  int p1;  while ((p1=fork( )) == -1); // 创建一个进程  if (p1 ==0)  printf(“This is a child process.\n”); //在子进程中  else  printf(“This is a parent process.\n”);  } |

图 2 一个最基本的fork( )程序代码

fork在英文中是“分叉”的意思，表示fork执行后进程将“分叉”，即产生了另一个进程。调用fork( )函数时，从已存在的进程（父进程）中将创建一个新进程（子进程）。子进程是父进程的一个拷贝：子进程和父进程使用相同的代码段；子进程复制父进程的数据与堆栈空间，并继承父进程的用户代码、组代码、环境变量、已打开的文件代码、工作目录和资源限制等。因为子进程几乎是父进程的完全复制，所以父子两个进程会运行同一个程序。

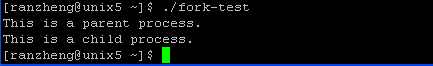


图 3 fork函数调用运行结果

但子进程虽然集成了父进程的一切数据，但它一旦开始运行，就和父进程分开，子进程独自拥有自己的进程号、资源使用和计时器等，与父进程之间不再共享任何数据。如果它们需要交互信息，就只能通过进程间通信的方式来实现。这时，就需要用一种方式来区分它们，系统就是通过函数的返回值来区分父进程和子进程的。对于父进程，fork函数返回子进程的进程号；而对于子进程，fork函数则返回0。因此，可以通过fork函数的返回值可以判定该进程是父进程还是子进程。



图 4 fork程序执行代码解析

通过对图 4的分析，我们可以看到：执行结果中，if、else两条分支语句都执行的原因，实际上是两个进程执行结果的叠加：父进程执行else分支语句（输出“This is a parent process.”），子进程执行if语句（输出“This is a child process.”）。

（2）进程标识符管理

Linux系统使用进程标识符来管理当前系统中的进程，进程的组标识符从父进程继承得到，用于区别进程是否同组。进程的标识符由系统分配，不能被修改；组标识符可以通过相关系统调用修改。

常用的进程标识符管理操作有：

* int getpid( ); 取得当前进程的标识符（进程ID）。许多程序利用该值来建立临时文件，以避免临时文件相同带来的问题。
* int getppid( ); 取得当前进程的父进程ID。
* int getpgrp( ); 取得当前进程的进程组标识符。
* int getpgid(int pid); 将当前进程的进程组标识符改为当前进程的进程ID，使其成为进程组首进程，并返回这一新的进程组标识符。

（3）加载新的进程映像

创建的进程往往希望它能执行新的程序。在Linux中，进程创建与加载新进程映像是分离的操作，当创建了一个进程后，通常将子进程替换成新的进程映像，这时可以利用exec系列的函数来进行。

exec函数族的作用是根据指定的文件名找到可执行文件，并用它来取代调用进程的内容，即在调用进程内部执行一个可执行文件。这个可执行文件既可以是二进制文件，也可以是任何Linux下可执行的脚本文件。在Linux中，并不存在exec( )函数，exec指的是一组函数，一共有6个，分别是：

* int execl(const char \*path, const char \*arg, ...);
* int execlp(const char \*file, const char \*arg, ...);
* int execle(const char \*path, const char \*arg, ..., char \*const envp[]);
* int execv(const char \*path, char \*const argv[]);
* int execvp(const char \*file, char \*const argv[]);
* int execve(const char \*path, char \*const argv[], char \*const envp[]);
* 其中，只有execve是真正意义上的系统调用，其它都是在此基础上经过包装的库函数。它们的区别仅在于执行的参数不一样，相互的关系如图 5所示。



图 5 exec函数族中的函数关系

exec用被执行的程序替换调用它的程序。与fork的区别是：fork创建一个新的进程，产生一个新的PID；exec启动一个新程序，替换原有的进程，因此进程的PID不会改变。exec和fork经常搭配使用。如一个进程希望执行另一个程序，就可以先利用fork函数创建出一个新进程，然后调用任何一个exec函数希望执行的那个程序。

* + - 1. wait/waitpid函数

当子进程退出时，内核会向父进程发送SIGCHLD信号，内核将子进程置为僵尸状态，只保留最小的一些内核数据结构，以便父进程查询子进程的退出状态。父进程查询子进程的退出状态可用wait/waitpid函数：

* pid\_t wait(int \*status);
* pid\_t waitpid(pid\_t pid, int \*status, int options);

wait系统调用会使父进程暂停执行，直到它的一个子进程结束为止，返回值为子进程的PID。waitpid函数用来等待某个特定进程的结束。两个函数都用于等待进程的状态变化，包括正常退出、被信号异常终止、被信号暂停、被信号唤醒继续执行等。在一个子进程终止前，wait使其调用者阻塞，而waitpid有一个选择项，可以使调用者不阻塞。实际上wait函数时waitpid函数的一个特例。

当子进程结束运行时，它与父进程之间的关联还会保持到父进程也正常结束或父进程调用wait/waitpid才终止。这时子进程为僵尸进程，即子进程的数据项不会立刻释放，虽然不再活跃，但子进程还驻留在系统中，因为它的退出码需要保存起来，以备父进程在调用wait/waitpid时使用。如果不想让父进程挂起，可以在父进程中加入语句：signal(SIGCHLD, SIG\_IGN)；表示父进程忽略SIGCHLD信号（子进程退出时向父进程发送）。也可以选择不忽略SIGCHLD信号。

### 管道通信

管道通信是Linux操作系统中最古老的一种进程间通信机制。两个进程利用管道进行通信时，发送信息的进程称为写进程，接收信息的进程称为读进程。管道通信方式的中间介质就是文件（即管道文件），它将写进程和读进程连接在一起，实现两个进程之间的通信。写进程通过写入端（发送端）往管道文件中写入信息；读进程通过读出端（接收端）从管道文件中读取信息。两个进程协调不断地进行写和读，便会构成双方通过管道传递信息的流水线。

管道包括无名管道和有名管道两种，前者用于有亲缘关系的父子进程或兄弟进程间的通信，后者克服了管道没有名字的限制，允许无亲缘关系的任意两个进程间的通信，如图 6所示。



图 6 无名管道和有名管道

1、无名管道

利用系统调用pipe( )可以创建一个无名管道文件，即无名管道：int pipe(int fd[2])，其中，fd[0]为读描述符，fd[1]为写描述符。一个进程在由pipe( )创建管道后，一般再利用fork函数创建子进程，然后通过管道实现父子进程间的通信。一般文件的I/O函数都可以用于管道，如close、read、write等。

无名管道是一种非永久性的管道通信机构。当它访问的进程全部终止时，它也将随之撤消。无名管道只能用在具有家族联系的进程之间，具有以下特点（如所示）：

* 管道是半双工的，数据只能向一个方向流动；需要双方通信时，需要建立两个管道；
* 只能用户父子进程或兄弟进程等具有亲缘关系的进程；
* 管道文件单独构成一种文件系统，并且只存在于内存中，是一个固定大小的缓冲区。在Linux中，该缓冲区的大小为4096字节；
* 一个进程向管道中写的内容被管道另一端的进程读出。写入的内容每次添加在管道缓冲区的末尾，并且每次都从缓冲区的头部读出数据。

2、有名管道

无名管道最大的问题就是它没有名字，只能用于具有亲缘关系的进程间通信。有名管道（也称FIFO）解决了这一问题。FIFO提供一个路径名与之关联，并以FIFO的文件形式存在于文件系统中。这样，进程即使与FIFO的创建进程不存在亲缘关系，只要可以访问该路径，就能彼此通过FIFO相互通信交换数据。FIFO严格遵循先进先出，并且不支持诸如lseek( )等文件定位操作。

在Linux系统中，有名管道可由两种方式创建：命令行方式mknod系统调用和函数mkfifo。管道文件被创建后，就可以使用一般的文件I/O函数如open、close、read和write等来对它进行操作。注意：有名管道比无名管道多了一个打开操作open( )。

有名管道可以长期存在于系统之中，而且提供给任意关系的进程使用，但是使用不当容易导致出错，所以操作系统将命名管道的管理权交由系统来加以控制。

与Linux中文件操作流的标准I/O操作一样，管道操作也支持基于文件流的模式。这种基于文件流的管道主要用来创建一个连接到另一个进程的管道，如执行命令“ls –l | grep test”。由于这类操作很常用，因此标准流管道就将一系列的创建过程合并到一个函数popen()中完成。它完成的工作包括：创建一个管道、fork一个子进程、在子进程中关闭不需要的文件描述符、执行exec函数族调用、执行函数中指定的命令。用popen创建的管道必须使用标准I/O函数进行操作，但不能使用read、write之类不带缓冲的I/O函数与之对应，关闭用popen创建的流管道必须使用函数pclose来关闭，该函数关闭标准I/O流，并等待命令执行结束。

## 实验指导

几个函数程序体框架分别为：

main( ) {

创建无名管道；

设置软中断信号SIGINT；

创建子进程1、2；

等待子进程1、2退出；

关闭管道；

}

父进程信号处理 {

发SIGUSR1给子进程1；

发SIGUSR2给子进程2；

}

SIGUSR1信号处理 {

关闭管道；

显示退出信息；

退出；

}

子进程1 {

设置忽略信号SIGINT；

设置信号SIGUSR1；

while(1) {

发送数据至管道数据；

计数器++；

睡眠1秒；

}

}

子进程2 {

设置忽略信号SIGINT；

设置信号SIGUSR1；

while(1) {

接收管道数据；

显示数据；

}

}

# 实验二 线程同步与通信

## 实验目的

1、掌握Linux下线程的概念；

2、了解Linux线程同步与通信的主要机制；

3、通过信号灯操作实现线程间的同步与互斥。

## 实验内容

通过Linux多线程与信号灯机制，设计并实现计算机线程与I/O线程共享缓冲区的同步与通信。

程序要求：两个线程,共享公共变量a

线程1负责计算(1到100的累加，每次加一个数)

线程2负责打印(输出累加的中间结果)

## 预备知识

### 信号灯及其P、V操作

信号量又称为信号灯，是用来解决进程间同步与互斥问题的一种进程间通信机制。程序对信号量的访问都是原子操作，且只允许对他进行等待（信号量的P操作）和发送（信号量的V操作）操作。信号量最主要的应用就是共享内存方式的进程间通信。

Linux信号量分为POSIX信号量和System V信号量，这一点与共享内存的两种方式比较相似：System V信号量的使用相对复杂，而POSIX信号量则非常简单。

1、POSIX信号量

POSIX信号量分为有名信号量和无名信号量。无名信号量又称为基于内存的信号量，常用于多线程间的同步，也可用于相关进程间的同步。无名信号量用于进行进程间同步时，需要放在进程间的共享内存区中。有名信号量通过IPC名字进行进程间的同步，它的特点是把信号值保存在文件中，这决定了它的用途非常广泛：既可以用于线程，也可以用于相关进程，甚至是不相关的进程。

POSIX信号量有三种操作：

（1）创建/销毁一个信号量；

（2）等待一个信号量（wait，即P操作）；

（3）挂起一个信号量（post，即V操作）。

POSIX有名信号量和无名信号量在使用过程中共享sem\_wait和sem\_post等函数，但因信号量存放位置的不同，两者在信号量的创建和删除上有所不同，有名信号量使用sem\_open代替sem\_init来创建信号量，而在结束时要像关闭文件一样关闭有名信号量。POSIX信号量的函数接口及关系可参见图 7。



图 7 POSIX信号量的操作函数

有名信号量的创建和删除操作的定义为：

# include <semaphore.h>

int sem\_init(sem\_t \*sem, int pshared, unsigned int value);

int sem\_destroy(sem\_t \*sem);

sem\_init用于无名信号量的初始化。对于特定的信号量，必须保证只调用sem\_init进行一次初始化（对于一个已初始化过的信号量调用sem\_init的行为是未定义的）。使用完无名信号量后，调用sem\_destroy销毁该信号量。销毁一个有线程阻塞在其上的信号量的行为也是未定义的。

有名信号量的创建和删除操作的定义为：

# include <semaphore.h>

sem\_t \*sem\_open(const char \*name, int oflag);

sem\_t \*sem\_open(const char \*name, int oflag, mode\_t mode, unsigned int value);

int sem\_close(sem\_t \*sem);

int sem\_unlink(const char \*name);

sem\_open用于创建或打开一个信号量，信号量通过name参数（信号量的名字）进行标识。sem\_close用于关闭打开的信号量。当一个进程终止时，内核对其上仍然打开的所有有名信号量自动执行这个操作。，但sem\_close并没有把信号量从系统中删除（POSIX有名信号量随内核持续），即使当前没有进程打开该信号量，它的值依然存在；直到内核重启或调用sem\_unlink函数删除该信号量（信号量的销毁是在所有进程都关闭信号量时进行）。

有名信号量和无名信号量共享使用信号量的操作函数，分为信号量的P操作、V操作等：

#include <semaphore.h>

int sem\_wait(sem\_t \*sem);

#ifdef \_\_USE\_XOPEN2K

int sem\_timedwait(sem\_t \*sem, const struct timespec \*abs\_timeout);

#endif

int sem\_trywait(sem\_t \*sem);

int sem\_post(sem\_t \*sem);

int set\_getvalue(sem\_t \*sem, int \*sval);

sem\_wait用于获取信号量，并执行信号量的P操作：测试指定信号量的值，如果大于0，就会将它减1并立即返回；如果等于0，调用线程会进入睡眠状态。sem\_trywait和sem\_wait的差别是，当信号量的值等于0时，调用线程不会阻塞，直接返回，并标识EAGAIN错误。sem\_timedwait和sem\_wait的差别是，当信号量的值等于0时，调用线程会限时等待。当等待时间结束时，如果信号量的值,继续为0，则返回错误。

当一个线程使用完某个信号量后，调用sem\_post使该信号量的值加1，如果有等待线程，则会唤醒等待的一个线程（V操作）。sem\_getvalue函数用于查询当前信号量的值。

2、System V信号量

System V信号量是System V进程间通信的组成部分（其他的还有前两节讲到的System V消息队列和System V共享内存）。不同于POSIX信号量，System V信号量在内核中维护。

Linux系统提供了一组System V信号量接口来对信号进行操作，相关信号量操作函数由sys/ipc.h文件引用，信号量的声明则在头文件sys/sem.h中定义。

System V信号量的操作函数主要有3个，其函数原型分别为：

int semget(key\_t key, int num\_sems, int sem\_flgs);

/\*创建一个新信号量或取得一个已有信号量\*/

int semctl(int sem\_id, int sem\_num, int cmd, union semun arg);

/\*删除/初始化信号量\*/

int semop(int sem\_id, struct sembuf \*sops, size\_t nsops);

/\*改变信号量的值，即使用/释放资源使用权\*/

Linux系统中，使用System V信号量通常分为以下4个步骤：

1. 调用semget( )函数创建信号量或获得在系统中已存在的信号量。不同进程通过使用同一个信号量键值来获得同一个信号量。

2. 使用semctl( )函数的SETVAL操作初始化信号量。

3. 调用semop( )函数进行信号量的PV操作，这是实现进程间同步和互斥的核心工作。

4. 如果不需要信号量，则从系统中删除它，此时使用semctl( )函数的IPC\_RMID操作。

信号灯的P、V操作可编码为：

void P(int semid,int index) {

struct sembuf sem;

sem.sem\_num = index;

sem.sem\_op = -1;

sem.sem\_flg = 0; //操作标记：0或IPC\_NOWAIT等

semop(semid,&sem,1); //1:表示执行命令的个数

return;

}

void V(int semid,int index) {

struct sembuf sem;

sem.sem\_num = index;

sem.sem\_op = 1;

sem.sem\_flg = 0;

semop(semid,&sem,1);

return;

}

### 线程及线程分类

根据操作系统的定义，进程是系统资源管理的最小单位，线程是程序执行的最小单位。进程和线程的关系可以用图 8表示，一个进程中至少有一个线程，线程与同属于一个进程的其他线程共享进程所拥有的全部资源。从进程（Process）演化出线程（Thread），最主要的目的是更好的支持多处理器，减少（进程/线程）上下文切换的开销。线程是在共享内存空间中并发的多道执行路径，能更充分的利用内存。线程和进程最大的区别是线程完全共享相同的地址空间，运行在同一地址上。

线程技术早在20世纪60年代就被提出，但真正应用多线程到操作系统是在20世纪80年代中期。在早期的Linux2.2内核中，不存在真正意义上的线程，常用的线程通过fork创建，属于“轻”进程。Linux2.2内核缺省只允许4096个进程/线程同时运行，这与Linux服务器需要服务上千用户相违背。Linux2.4内核消除了线程个数的限制，允许在系统中动态的调整进程数上限。在Linux2.6内核之前，进程是最主要的处理调度单元，系统并不支持内核线程机制。Linux2.6中实现了共享地址空间的进程机制，在1996年第一次获得线程的支持。



图 8 进程和线程的关系

为了改善Linux线程问题，Linux2.6根据新内核机制重新编写了线程库，改善Linux对线程的支持。主要有为两类线程库：

1. 由Red Hat主导的本地化POSIX线程库（Native POSIX Thread Library，简称NTPL），现在已经成为GNU C函数库的一部分，同时也成为Linux线程的标准。它采用的线程模型为1:1的线程模型，即一个内核态线程对应一个用户态线程。当创建一个线程时，在内核中相应创建一个调度实体（即用户态线程），从而最大可能的简化线程的实现。
2. 由IBM开发的新一代POSIX线程库（Next Generation POXIS Threads，简称NGPT），它是基于GNU Pth（GNU Portable Threads）项目而实现的M:N模型（M个用户态线程对应N个核心态线程），通常用户态线程会多于核心态线程。在这种模型中，由线程库本身去处理可能存在的调度，避免了系统调用转到内核态，因此上下文切换会很快。但它增加了线程实现的复杂性，并可能出现优先级反转等问题，而且为了避免出现多个Linux线程标准，该方式已停用。

这里提到了核心态线程和用户态线程，这实际是Linux 线程的两种实现方法，分类的标准主要是线程的调度者在核内还是核外。核心态线程更利于并发使用多处理器的资源，而用户态线程更多考虑的是上下文切换开销。

用户态线程允许多线程的程序运行时不需要特定的内核支持。Unix使用异步I/O机制，即一个进程中的某个线程调用了一个阻塞的系统调用，则该进程被阻塞，进程中的其他所有线程也同时被阻塞。它的缺点是一个进程的多个线程调度中无法发挥多处理器的优势。用户态线程的优点体现在两个方面：（1）某些线程操作的系统消耗大大减少。如属于同一进程的线程之间不需要调用系统调用就可以进程调度切换，将减少额外的消耗，一个进程可以启动上千个线程。（2）用户态线程的实现方式可以被定制或修改，以适应特殊应用的要求；而且用户态线程比核心态线程默认支持更多的线程。

内核态线程在核内以轻量级进程的形式存在，拥有独立的进程表项，而所有的创建、同步、删除等操作都在核外pthread库中进行。它的实现方法允许不同进程中的线程按照同一相对优先调度方法进行调度，这样有利于发挥多处理器的并发优势。

目前，线程主要的实现方法是用户态线程。在Linux Threads中，有一个专门的管理线程处理所有的线程管理工作。当进程第一次创建线程时就会先启动管理线程，后续进程创建线程时，都是该管理线程创建用户线程，并记录轻量级进程号和线程的映射关系。因此，用户线程其实是管理线程的子线程。

 Linux从2.4.17内核开始就包含对Intel P4处理器的超线程的支持。超线程技术（Hyperthreading Technology）是Intel公司的创新技术，就是利用特殊的硬件指令，把两个逻辑内核模拟成两个物理芯片，让单个处理器都能使用线程级并行计算，从而兼容多线程操作系统和软件，并提高处理器的性能。

超线程技术可以使芯片同时进行多线程处理，当在支持多处理器的Linux操作系统之下运行时，同时运行多个不同的软件程序，从而获得更高的运行效果，使用户获得更优异的性能和更短的等待时间。

线程有时也被称为轻量级进程，Linux系统下的多线程遵循POSIX线程接口，称为pthread。Pthread线程库是一套通用的线程库，具有较好的移植性。编写Linux下的多线程程序，需要使用头文件pthread.h，连接时需要使用库libpthread.a。最基本的线程函数主要有一下以下5个函数：

1. int pthread\_create(pthread\_t \* tid, const pthread\_attr\_t \*attr, void \* (\* func)(void \*), void \*arg);   
   创建线程就是确定调用该线程函数的入口点，相当于fork函数。
2. int pthread\_join(pthread\_t tid, void \*\*status);  
   等待tid线程终止，相当于waitpid函数。
3. pthread\_t pthread\_self(void);  
   取得自己的线程ID，相当于getpid函数。
4. int pthread\_detach(pthread\_t tid);  
   将指定的线程脱离，脱离的线程类似于守护进程。一般将自己脱离，方法为：pthread\_detach(pthread\_self( ));
5. void pthread\_exit(void \*status);  
   终止线程的执行。如果该线程未脱离，则其线程ID和退出状态将一直保留到某个线程调用pthread\_join为止。

## 实验指导

下面给出程序结构：

#include 头文件pthread.h、sys/types.h、linux/sem.h等

P、V操作的函数定义：

void P(int semid,int index)

void V(int semid,int index)

信号灯、线程句柄定义：

int semid; pthread\_t p1,p2;

线程执行函数定义：void \*subp1();void \*subp2();

void \*subp1() {

for (......) {

P(…….);

打印;

V(…..);

}

return;

}

主函数：main() {

创建信号灯；

信号灯赋初值；

创建两个线程subp1、subp2;

等待两个线程运行结束；

删除信号灯；

}

# 实验三、共享内存与进程同步

## 实验目的

1、掌握Linux下共享内存的概念与使用方法；

2、掌握环形缓冲的结构与使用方法；

3、掌握Linux下进程同步与通信的主要机制。

## 实验内容

利用多个共享内存（有限空间）构成的环形缓冲，将源文件复制到目标文件，实现两个进程的誊抄。

## 预备知识

### 共享内存

管道、FIFO、消息队列的共同特点是通过内核来进程通信，如图 9(a)所示，向管道、FIFO、消息队列写入数据时需要把数据从进程复制到内核，数据读出时又需要从内核复制数据到进程，即进程间的通信必须借助内核、通过多次数据拷贝来实现。

共享内存是在两个正在运行的进程之间共享和传递数据的一种简单但非常有效的方式，它将同一块内存区映射到共享它的不同进程的地址空间中（如图 9(b)所示），这样进程间的通信就不再需要通过内核在不同的进程间复制，只需对共享的内存区进行操作就可以了，如同malloc( )函数向不同进程返回指向同一个物理内存区域的指针一样，当一个进程改变了这块地址中的内容时，其他进程都会察觉到这个更改。

共享内存是最高效的一种进程间通信方式，因为进程可以直接读写内存，避免了对数据的各种不必要的拷贝；而且进程之间共享使用共享的内存区时，数据内容将一直保存在共享内存中，直到解除映射、通信完毕时才会写回文件，从而达到高效通信的目的。但问题在于，当两个或多个进程使用共享内存进行通信时，系统内核并未对共享内存的访问提供同步机制，容易造成因不同进程同时读写一块共享内存数据而发生混乱，程序员需要依靠某种同步机制（如互斥锁、信号量等）来同步对共享内存的访问。



图 9 管道、FIFO、消息队列和共享内存实现方式的比较

在Linux系统中，每个进程的虚拟内存被分为许多页面，每个进程都会维护一个从内存地址到虚拟内存页面之间的映射关系。尽管每个进程都有自己的内存地址，不同的地址可以同时将同一个内存页面映射到自己的地址空间中，从而达到共享内存的目的。

Linux有两种共享内存机制：POSIX共享内存和System V共享内存。两者都通过tmpfs（一种基于内存的文件系统）实现，但POSIX共享内存通过用户空间挂载的tmpfs文件系统实现，而System V共享内存则由内核本身的tmpfs实现。两种共享内存的区别在于：System V共享内存是持久化的，只要机器不重启或不显式销毁，该共享内存将一直存在，即除非一个进程明确删除该共享内存，否则它始终存在于内存中，直到系统关机；而POSIX共享内存不是持久化的，如果进程关闭，映射将随即失效（事先映射到文件上的情况除外）。

其中，System V共享内存历史悠久，一般Unix系统上都有这套机制；而POSIX共享内存机制接口更加方便易用，一般结合内存映射mmap使用。内存映射机制mmap是POSIX标准的系统调用，有匿名映射和文件映射两种。匿名映射使用进程的虚拟内存空间。文件映射有MAP\_PRIVATE和MAP\_SHARED两种，前者使用COW方式（copy-on-write，写时拷贝技术）把文件映射到当前的进程空间，修改操作不会改动源文件；后者直接把文件映射到当前的进程空间，所有的修改会直接反映到文件的page cache，然后由内核自动同步到映射文件上。

由于接口易用，且可以方便的映射到文件，避免主机宕机造成数据丢失，同时POSIX标准比较通用，Linux操作系统中一般偏向于使用mmap的POSIX共享内存，而非传统的System V的共享内存机制。下面介绍两种共享内存机制的使用。

1、POSIX共享内存

POSIX共享内存机制主要通过内存映射（mmap）机制来实现。mmap( )系统调用使得进程之间通过映射同一个普通文件实现共享内存。普通文件被映射到进程地址空间后，进程可以像访问普通内存一样对文件进行操作，不必再调用read( )、write( )等操作。

在进程间使用POSIX共享内存涉及两个步骤：

（1）指定一个名字参数调用shm\_open( )，创建一个新的共享内存区或打开一个已存在的共享内存区：

int shm\_open(const char \*name, into flag, mode\_t mode);

（2）调用mmap( )把该共享内存区映射到调用进程的地址空间，传递shm\_open的名字参数，之后希望共享该内存区的任何其他进程就可以使用该共享内存了：

void \*mmap(void \*start, size\_t length, int prot, int flags, int fd, off\_t offset);

至此，对共享内存的操作就与普通内存没有区别了，可以使用memcpy、memset等对共享内存进行操作。在共享内存的使用过程中，共享内存区的大小可以通过调用ftruncate进行修改：int ftruncate(int fd, off\_t length)；而当打开一个已存在的共享内存区时，可以调用fstat来获取有关该对象的信息：int stat(const char \*file\_name, struct stat \*buf)。

如需结束对共享内存的使用，执行以下两步：

（1）解除当前进程对这块共享内存的映射：int munmap(void \*start, size\_t length);

（2）从内核清除共享内存：int shm\_unlink(const char \*name);

2、System V共享内存

System V共享内存通过系统调用shmget( )来获得或创建一个IPC共享内存区域，还将在特殊文件系统shm中创建或打开一个同名文件，新建的文件不属于任何一个进程（任何进程够可以访问该共享内存区域）。一般情况下，特殊文件系统shm中的文件不能使用read( )、write( )函数进行访问，但可以直接采用访问内存的方式对其进行访问。

System V共享内存主要有以下几个API，使用时需包括头文件sys/ipc.h和sys/shm.h：

* shmget函数

shmget函数用来创建共享内存，它的原型为：

int shmget(key\_t key, size\_t size, int shmflg);

该函数类似于shm\_open函数或malloc函数，系统按照请求分配（或获取）size大小的内存用作共享内存。Linux系统内核中每个IPC结构都有一个标识符，该标识符是内核由IPC结构的关键字得到，即key值。

* shmat函数

第一次创建完共享内存后，它还不能被任何进程访问，shmat函数的作用就是用来启动对该共享内存的访问，并把共享内存映射到当前进程的地址空间，这样就能方便的访问共享内存了。

shmat函数原型为：void \*shmat(int shm\_id, const void \*shm\_addr, int shmflg);，其功能类似于mmap函数。

* shmdt函数

该函数用于将共享内存从当前进程分离（同munmap函数），即解除共享内存与进程地址空间的映射关系。但分离并不是删除它，只是使该共享内存对当前进程不再可用。函数原型为：int shmdt(void \*shm\_addr);。

* shmctl函数

与shm\_unlink函数一样，实现对共享内存的控制操作：

int shmctl(int shm\_id, int cmd, struct shmid\_ds \*buf);

无论使用哪种共享内存机制来实现进程间通信都必须注意对数据存取的同步，必须确保一个进程读数据时，它所需的的数据已经写好。通常，信号量被用于实现共享数据存取的同步，也可以通过使用shmctl函数设置共享内存的某些标志位如SHM\_LOCK、SHM\_UNLOCK等来实现。

### 环形缓冲

缓冲的目的是为了匹配CPU与设备的速度差异和负荷的不均衡，从而提高处理机与外设的并行程度。

缓冲技术可以用硬件缓冲器来实现，通常容量较小。软件缓冲是应用较广泛的一种缓冲技术，由缓冲区和对缓冲区的管理两部分组成。常用的缓冲技术有： 双缓冲、环形缓冲、缓冲池。

环形缓冲区通常有一个读指针和一个写指针，如图10所示。读指针指向环形缓冲区中可读的数据，写指针指向环形缓冲区中可写的缓冲区。通过移动读指针和写指针就可以实现缓冲区的数据读取和写入。在通常情况下，环形缓冲区的读用户仅仅会影响读指针，而写用户仅仅会影响写指针。如果仅仅有一个读用户和一个写用户，那么不需要添加互斥保护机制就可以保证数据的正确性。如果有多个读写用户访问环形缓冲区，那么必须添加互斥保护机制来确保多个用户互斥访问环形缓冲区。

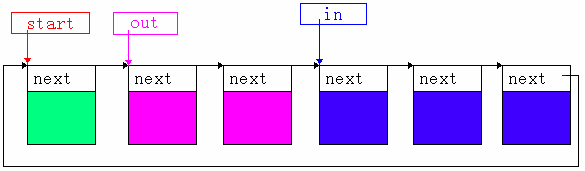


图 10 环形缓冲及读写指针

## 实验指导

主函数可以描述为：

main() {

创建共享内存组；

创建信号灯；

信号灯赋初值；

创建两个进程readbuf、writebuf;

等待两个进程运行结束；

删除信号灯；

删除共享内存组；

}

请自己思考Readbuf负责读、writebuf负责写，如何定义？

# 实验四、Linux文件目录

## 实验目的

1、了解Linux文件系统与目录操作；

2、了解Linux文件系统目录结构；

3、掌握文件和目录的程序设计方法。

## 实验内容

编程实现目录查询功能：

* 功能类似ls -lR；
* 查询指定目录下的文件及子目录信息；
* 显示文件的类型、大小、时间等信息；
* 递归显示子目录中的所有文件信息。

## 预备知识

### 认识文件

Linux系统以文件形式进行管理，即一切皆文件。无论文件是程序、文档、数据库，还是目录，Linux系统都会赋予它相同的结构，即都由索引节点和数据组成。索引节点又称I节点，一般是一个信息记录，包括文件权限、文件主、文件大小、存放位置、建立日期等；数据是文件的实际内容，可以为空，也可以非常大，并且有自己的结构。Linux中以圆点‘．’开头的文件名是隐含文件（dot files），默认方式下使用“ls”命令时不能显示，需要执行带“-a”选项的ls命令查看。

Linux文件类型常见的有：普通文件、目录文件、设备文件、符号链接文件等，可用file命令来识别。

1. 普通文件

普通文件也称常规文件，包含各种长度的字符串。系统对这些文件没有结构化，只是作为有序的字节序列交给应用程序，由应用程序自己组织和解释文件内容。

执行命令“ls –lh”查看某个文件的属性，可以看到类似“-rw-r--r--”的属性显示，其中第一个符号‘-’表示普通文件。这些文件一般由相关的应用程序创建，如图像工具、文档工具、归档工具等，可以用“rm”命令删除。

根据文件内容，普通文件可分为文本文件、数据文件和可执行的二进制程序。文本文件（ASCII）是Linux系统中最多的一种文件，由ASCII字符构成，如txt文本、Shell脚本，可以直接读取文件内容（如“cat”命令）。数据文件（Data File）是具有特定格式的文件，由来自应用程序的数字型和文本型数据构成，如Excel表格和word文档。使用“cat”命令读取数据文件，只会显示乱码，因为它必须由相应的应用程序读取。二进制文件（Binary File）由机器指令和数据构成，Linux中的可执行文件（脚本文件除外）就属于二进制文件，如“cat”命令就是二进制文件。

2. 目录文件

目录文件是一类特殊的文件，利用它可以构成文件系统的分层树型结构。目录文件也包含数据，但与普通文件不同的是，这些数据被系统结构化，由成对的“I节点号/文件名”构成。包括文件名、子目录名及其指针。它是LINUX储存文件名的唯一地方，可用ls列出目录文件。

目录文件的属性特征用‘d’表示。创建目录的命令可以用“mkdir”命令，“cp”命令可以把一个目录复制为另一个目录（“-r”选项）。删除目录用“rm -r”或“rmdir”命令。

3. 设备文件

在Linux系统中，所有设备都被作为一类特殊文件对待，用户像使用普通文件那样操作设备，从而实现设备无关性。设备作为特殊文件也有索引节点，但不同的是它不包含指向磁盘数据块的指针，取而代之的是硬件设备的一个标识符（对应字符设备或块设备）。设备文件一般放在/dev目录下。

有两类特别设备文件：字符设备文件（Character）和块设备文件（Block），分别对应字符设备和块设备两种不同类型的设备驱动器。字符设备是最常用的设备类型，包括一些串行端口的接口设备，如键盘、鼠标等。这类设备的特点是一次性读取，输出不能打断，如鼠标需要滑动到另一个窗口，而不能直接跳到另一个窗口。字符设备文件的属性用‘c’表示。块设备文件的属性为‘d’，对应块设备的输入/输出以块为单位，可以通过设置的缓冲区自动缓存待传送的数据，并实现随机存取，如可以随机的在硬盘的不同区块读写。使用这种接口的设备包括硬盘、光盘等。

设备文件用命令“mknode”创建，用命令“rm”删除。在最新的Linux发行版本中，一般不用自己创建设备文件，因为这些文件和内核相关联。

4. 符号链接文件

符号链接文件是指向同一索引节点的那些目录条目。用“ls”命令来查看时，链接文件的标志用‘l’开头，而文件后面以“->”指向所连接的文件。 可以通过“ln -s sourcefile newfile”的方式创建一个符号链接文件，这与Windows操作系统中的快捷方式相类似。

5. 套接字文件

套接字文件（sockets）通常用于网络数据，属性为‘s’。可以启动一个程序监听客户端的要求，而客户端通过这个socket来进行数据通信。在/var/run目录中经常会出现这种类型的文件。如当启动MySQL服务器时，就会产生一个mysql.sock文件。

Linux是一个多用户系统，每个用户属于一个组，Linux的文件所有权和访问授权与用户id和组密切相关。使用“whoami”命令可以检查当前用户有效的id，执行“groups”命令可以找出当前用户所在的组，“id”命令可以找出用户和组信息。Linux文件的访问权限有三种类型：读（r）、写（w）和执行（x）。写权限包括修改和删除对象的能力。此外，这些权限被分别指定给文件所有者、文件组成员和其他人。

“ls –l”命令用于查看文件属性，第一列信息是一个由10个字符组成的字符串，第一个字母描述文件类型，剩下的9个字母每三个字母为一组，分别表示文件所有者、文件组、其他人对该文件的读、写和执行的权限。‘-’表示相应的权限没有被授予。

不同于Windows，Linux的文件是没有所谓的扩展名的，一个Linux文件能否被执行，与文件权限有关，而与文件名没有任何关系。只要文件权限授予了‘x’属性就可以被执行，但能否执行成功，由文件内容决定，即‘x’只代表文件具有可执行的能力，与执行结果无关。尽管如此，我们仍会遵循Windows文件命名的习惯，希望藉由扩展名来区分文件种类，如将脚本文件命名成\*.sh，C语言头文件仍然定义成\*.h。

### Linux文件属性接口

#include <unistd.h>

#include <sys/stat.h>

#include <sys/types.h>

int fstat(int fildes,struct stat \*buf);

返回文件描述符相关的文件的状态信息

int stat(const char \*path, struct stat \*buf);

通过文件名获取文件信息，并保存在buf所指的结构体stat中

int lstat(const char \*path, struct stat \*buf);

如读取到了符号连接，lstat读取符号连接本身的状态信息，而stat读取的是符号连接指向文件的信息。

struct stat {

unsigned long st\_dev; // 文件所属的设备

unsigned long st\_ino; // 文件相关的inode

unsigned short st\_mode; // 文件的权限信息和类型信息:   
 S\_IFDIR, S\_IFBLK, S\_IFIFO, S\_IFLINK  
 unsigned short st\_nlink; //硬连接的数目  
 unsigned short st\_uid; // 文件所有者的ID   
 unsigned short st\_gid; //文件所有者的组ID   
 unsigned long st\_rdev; //设备类型  
 unsigned long st\_size; //文件大小  
 unsigned long st\_blksize; //块大小  
 unsigned long st\_blocks; //块数  
 unsigned long st\_atime; // 文件最后访问时间   
 unsigned long st\_atime\_nsec;  
 unsigned long st\_mtime; // 最后修改内容的时间   
 unsigned long st\_mtime\_nsec;  
 unsigned long st\_ctime; // 文件最后修改属性的时间  
 unsigned long st\_ctime\_nsec;   
 unsigned long \_\_unused4;   
 unsigned long \_\_unused5;

};

stat结构体几乎保存了所有的文件状态信息。

### Linux目录结构接口

#include <sys/types.h>

#include <dirent.h>

#include <unistd.h>

* opendir()

DIR \*opendir(const char \*name);

通过路径打开一个目录，返回一个DIR结构体指针(目录流)，失败返回NULL；

* readdir()

struct dirent \*readdir(DIR \*)

读取目录中的下一个目录项，没有目录项可以读取时，返回为NULL；

目录项结构：

struct dirent {     
        #ifndef \_\_USE\_FILE\_OFFSET64

\_\_ino\_t d\_ino; //索引节点号

\_\_off\_t d\_off; //在目录文件中的偏移

#else

\_\_ino64\_t d\_ino;

\_\_off64\_t d\_off;

#endif

    unsigned   short   int   d\_reclent;     //文件名的长度

    unsigned   char   d\_type;     //d\_name所指的文件类型     
        char   d\_name[256];     //文件名     
  };

注：需跳过两个目录项“.”和“..”

定义见/usr/include/dirent.h

* chdir()

int chdir(const char \*path);

改变目录,与用户通过cd命令改变目录一样，程序也可以通过chdir来改变目录，这样使得 fopen(),opendir(),这里需要路径的系统调用，可以使用相对于当前目录的相对路径打开文件(目录)。

* closedir()

int closedir(DIR\*)

关闭目录流

## 实验指导

参考程序结构：

#include <unistd.h>

#include <sys/stat.h>

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <dirent.h>

void printdir(char \*dir, int depth){

DIR \*dp;

struct dirent \*entry;

struct stat statbuf;

if ((dp = 打开dir目录) 不成功){

打印出错信息;

返回;

}

将dir设置为当前目录;

while(读到一个目录项){

以该目录项的名字为参数,调用lstat得到该目录项的相关信息;

if(是目录){

if(目录项的名字是”..”或”.”)

跳过该目录项;

打印目录项的深度、目录名等信息

递归调用printdir,打印子目录的信息,其中的depth+4;

} else　打印文件的深度、文件名等信息

}

返回父目录;

关闭目录项;

}

int main(…){

………

}

输入ls -l 应该可以看到如下信息：

drwxr-xr-x 3 killercat killercat 4096 2007-01-11 16:27 Desktop

drwx------ 8 killercat killercat 4096 2007-01-09 14:33 Documents

drwxr-xr-x 2 killercat killercat 4096 2006-11-30 19:27 Downloads

drwx------ 4 killercat killercat 4096 2006-12-16 20:20 References

drwx------ 9 killercat killercat 4096 2007-01-11 13:34 Software

drwxr-xr-x 3 killercat killercat 4096 2006-12-11 16:39 vmware

drwx------ 6 killercat killercat 4096 2007-01-11 13:34 Workspace