## 第2课 讲稿（文件I/O编程）

1. **Linux系统调用及用户编程接口（API）**

学习Linux编程开发之前，需要先了解Linux系统调用和用户编程接口（API）的概念，在了解了这些之后，会对Linux 以及Linux 的应用编程有更深入的理解。

1. 系统调用

所谓系统调用是指操作系统提供给用户程序调用的一组“特殊接口”，用户程序可以通过这组“特殊接口”来**获得操作系统内核提供的服务**，例如用户可以通过进程控制相关的系统调用来创建进程，实现进程调度、进程管理等。

在这里，为什么用户程序不能直接访问系统内核提供的服务呢？这是由于在Linux中，为了更好地保护内核空间，将程序的运行空间分为**内核空间**和**用户空间**（也就是常称的**内核态**和**用户态**），它们分别运行在不同的级别上，在逻辑上是相互隔离的。因此，用户进程在通常情况下不允许访问内核数据，也无法使用内核函数，它们只能在用户空间操作用户数据，调用用户空间的函数。

但是，在有些情况下，用户空间的进程需要获得一定的系统服务（调用内核空间程序），这时操作系统就必须利用系统提供给用户的“特殊接口”——系统调用规定用户进程进入内核空间的具体位置。进行系统调用时，程序运行空间需要从用户空间进入内核空间，处理完后再返回用户空间。

Linux系统调用部分是非常精简的系统调用（只有250个左右），它继承了UNIX系统调用中最基本和最有用的部分。这些系统调用按照功能逻辑大致可分为进程控制、进程间通信、文件系统控制、系统控制、存储管理、网络管理、socket控制（IP地址和端口控制）、用户管理等几类。

1. 用户编程接口（API）

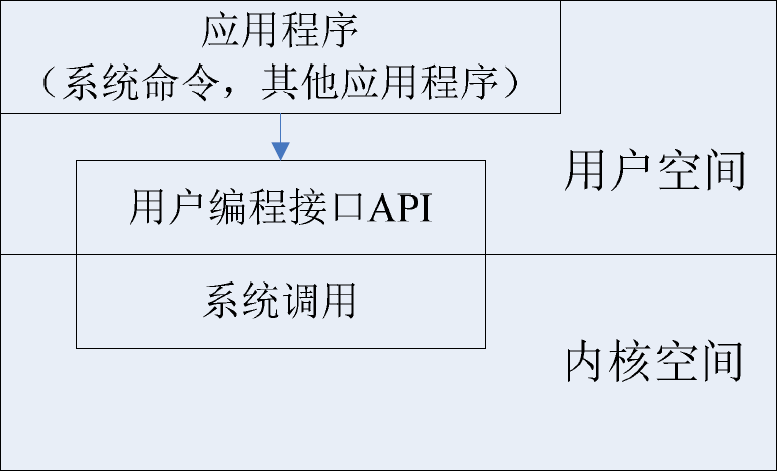
前面讲到的系统调用并不是直接与程序员进行交互的，它仅仅是一个通过软中断机制向内核提交请求，以获取内核服务的接口。在实际使用中程序员调用的通常是用户编程接口——API，也就是API 函数。但并不是所有的函数都一一对应一个系统调用，有时，一个API 函数会需要几个系统调用来共同完成函数的功能，甚至还有一些API 函数不需要调用相应的系统调用（因此它所完成的不是内核提供的服务）。

在Linux 中，用户编程接口（API）遵循了在UNIX 中最流行的应用编程界面标准——POSIX 标准（可移植操作系统接口）。POSIX标准是由IEEE（电气与电子工程师协会）和ISO/IEC（国际电工委员会）共同开发的标准系统。该标准基于当时现有的UNIX 实践和经验，描述了操作系统的系统调用编程接口（实际上就是API），用于保证应用程序可以在源代码一级上在多种操作系统上移植运行。

1. 系统命令

Shell 系统命令与系统调用和用户编程接口（API）之间又是怎样的关系呢？

系统命令相对API 更高了一层，它实际上是一个可执行程序，它的内部引用了用户编程接口（API）来实现相应的功能。它们之间的关系如下图所示：



1. **Linux 中文件及文件描述符概述**

在Linux 中对目录和设备的操作都等同于文件的操作，因此，大大简化了系统对不同设备的处理，提高了效率。**Linux 中的文件主要分为6种：普通文件、目录文件、链接文件、管道文件、套接字文件和设备文件**。

那么，**内核如何区分和引用特定的文件**呢？这里用到了一个重要的概念——**文件描述符**。对于Linux 而言，所有对设备和文件的操作都是使用文件描述符来进行的。文件描述符是一个非负的整数，它是一个**索引值**，并指向在内核中**每个进程打开文件的记录表**。当打开一个现存文件或创建一个新文件时，内核就向进程返回一个文件描述符；当需要读写文件时，也需要把文件描述符作为参数传递给相应的函数。

通常，一个进程启动时，都会打开3 个文件：标准输入、标准输出和标准出错处理。这3 个文件分别对应文件描述符为0、1 和2（也就是宏替换STDIN\_FILENO、STDOUT\_FILENO 和STDERR\_FILENO）。

基于文件描述符的I/O操作是Linux 中最常用的操作之一，基于文件描述符的I/O操作虽然不能移植到类Linux 以外的系统上去（如Windows），但它往往是实现某些I/O操作的惟一途径，如Linux中低级文件操作函数、多路I/O、TCP/IP套接字编程接口（socket接口）等。同时，它们也很好地兼容POSIX标准，因此，可以很方便地移植到任何POSIX平台上。

1. **文件I/O和标准I/O的区别**

文件I/O又称为低级磁盘I/O，遵循POSIX 相关标准，任何兼容POSIX 标准的操作系统上都支持文件I/O。标准I/O被称为高级磁盘I/O，遵循ANSI C相关标准。只要开发环境中有标准C库，标准I/O就可以使用。（Linux中使用的是GLIB C，它是标准C库的超集，不仅包含ANSI C中定义的函数，还包括POSIX标准中定义的函数，因此，Linux 下既可以使用标准I/O，也可以使用文件I/O）。

通过文件I/O 读写文件时，每次操作都会执行相关**系统调用**。这样处理的好处是直接读写实际文件，坏处是频繁的系统调用会增加系统开销。标准I/O可以看成是在文件I/O 的基础上封装了缓冲机制。先读写缓冲区，必要时再访问实际文件，从而减少了系统调用的次数。

文件I/O中用文件描述符表示一个打开的文件，可以访问不同类型的文件，如普通文件、设备文件和管道文件等。而标准I/O中用FILE（流）表示一个打开的文件，通常只用来访问普通文件。

文件I/O不带缓冲，即在用户的进程中对这两个函数不会**自动缓冲，**每次读写操作都要进行一次系统调用，直接对文件进行读写，所以能实时反映文件的内容。而标准I/O在打开文件时会自动缓冲，每次读写时仅对缓冲区进行操作，不对文件操作，退出时才更新文件。

1. **文件I/O 操作**

文件I/O操作的系统调用，主要用到5个函数：open()、read()、write()、lseek()和close()。这些函数的特点是不带缓存，直接对文件（包括设备）进行读写操作。这些函数虽然不是ANSI C的组成部分，但是是POSIX的组成部分。

1. 基本文件操作
2. 函数说明。

* open()函数用于打开或创建文件，在打开或创建文件时可以指定文件的属性及用户的权限等各种参数。
* close()函数用于关闭一个被打开的文件。当一个进程终止时，所有被它打开的文件都由内核自动关闭，很多程序都使用这一功能而不显示地关闭一个文件。
* read()函数用于将从指定的文件描述符中读出的数据放到缓存区中，并返回实际读入的字节数。若返回0，则表示没有数据可读，即已达到文件尾。读操作从文件的当前指针位置开始。当从终端设备文件中读出数据时，通常一次最多读一行。
* write()函数用于向打开的文件写数据，写操作从文件的当前指针位置开始。对磁盘文件进行写操作，若磁盘已满或超出该文件的长度，则write()函数返回失败。
* lseek()函数用于在指定的文件描述符中将文件指针定位到相应的位置。它只能用在可定位（可随机访问）文件操作中。管道、套接字和大部分字符设备文件是不可定位的，所以在这些文件的操作中无法使用lseek()调用。

1. 函数格式。
2. open()函数的语法格式如表1所示：

|  |  |  |
| --- | --- | --- |
| 表1 open()函数的语法格式 | | |
| 所需头文件 | #include <sys/types.h> /\*提供类型pid\_t的定义\*/  #include <sys/stat.h> /\*获取文件属性\*/  #include <fcntl.h> /\*定义open等函数原型\*/ | |
| 函数原型 | int open(const char \*pathname, int flags, int perms) | |
| 函数传入值 | pathname | 被打开的文件名（可包括路径名） |
| flag：文件打开的方式 | O\_RDONLY：以只读方式打开文件 |
| O\_WRONLY：以只写方式打开文件 |
| O\_RDWR：以读写方式打开文件 |
| O\_CREAT：如果该文件不存在，就创建一个新的文件，并用第三个参数为其设置权限 |
| O\_EXCL：如果使用O\_CREAT 时文件存在，则可返回错误消息。  这一参数可测试文件是否存在。此时open是原子操作，防止多个进程同时创建同一个文件 |
| O\_NOCTTY：使用本参数时，若文件为终端，那么该终端不会成为调用open()的那个进程的控制终端 |
| O\_TRUNC：若文件已经存在，那么会删除文件中的全部原有数据，并且设置文件大小为0。 |
| O\_APPEND：以添加方式打开文件，在打开文件的同时，文件指针指向文件的末尾，即将写入的数据添加到文件的末尾 |
| perms | 创建的新文件的存取权限  可以用一组宏定义：S\_I(R/W/X)(USR/GRP/OTH)  其中R/W/X 分别表示读/写/执行权限  USR/GRP/OTH 分别表示文件所有者/文件所属组/其他用户  例如，S\_IRUSR | S\_IWUSR 表示设置文件所有者的可读可写属性。八进制表示法中600也表示同样的权限 |
| 函数返回值 | 成功：返回文件描述符  失败：-1 | |

在open()函数中，flags参数可通过“|”组合构成，但前3个标志常量（O\_RDONLY、O\_WRONLY以及O\_RDWR）不能相互组合。perms是文件的存取权限，既可以用宏定义表示法，也可以用八进制表示法。

1. close()函数的语法格式表2所示。

|  |  |
| --- | --- |
| 表2 close()函数的语法格式 | |
| 所需头文件 | #include <unistd.h> |
| 函数原型 | int close(int fd) |
| 函数输入值 | fd：文件描述符 |
| 函数返回值 | 0：成功  -1：出错 |

1. read()函数的语法格式如表3所示。

|  |  |
| --- | --- |
| 表3 read()函数的语法格式 | |
| 所需头文件 | #include <unistd.h> |
| 函数原型 | ssize\_t read(int fd, void \*buf, size\_t count) |
| 函数传入值 | fd：文件描述符 |
| buf：指定存储器读出数据的缓冲区 |
| count：指定读出的字节数 |
| 函数返回值 | 成功：读到的字节数  0：已到达文件尾  -1：出错 |

在读普通文件时，若读到要求的字节数之前已到达文件的尾部，则返回的字节数会小于希望读出的字节数。

1. write()函数的语法格式如表4所示。

|  |  |
| --- | --- |
| 表4 write()函数的语法格式 | |
| 所需头文件 | #include <unistd.h> |
| 函数原型 | ssize\_t write(int fd, void \*buf, size\_t count) |
| 函数传入值 | fd：文件描述符 |
| buf：指定存储器写入数据的缓冲区 |
| count：指定读出的字节数 |
| 函数返回值 | 成功：已写的字节数  -1：出错 |

在写普通文件时，写操作从文件的当前指针位置开始。

1. lseek()函数的语法格式如表5所示。

|  |  |  |
| --- | --- | --- |
| 表5 lseek()函数的语法格式 | | |
| 所需头文件 | #include <unistd.h>  #include <sys/types.h> | |
| 函数原型 | off\_t lseek(int fd, off\_t offset, int whence) | |
| 函数传入值 | fd：文件描述符 | |
| offset：偏移量，每一读写操作所需要移动的距离，单位是字节，可正可负（向前移，向后移） | |
|  | whence：  当前位置的基点 | SEEK\_SET：当前位置为文件的开头，新位置为偏移量的大小 |
| SEEK\_CUR：当前位置为文件指针的位置，新位置为当前位置加上偏移量 |
| SEEK\_END：当前位置为文件的结尾，新位置为文件的大小加上偏移量的大小 |
| 函数返回值 | 成功：文件的当前位移  -1：出错 | |

1. 函数使用实例。

下面列出文件基本操作的实例，基本功能是从一个文件（源文件）中读取最后5B 数据并拷贝到另一个文件（目标文件）。在实例中源文件是以只读方式打开，目标文件是以只写方式打开（可以是读写方式）。若目标文件不存在，可以创建并设置权限的初始值为0644，即文件所有者可读可写，文件所属组和其他用户只能读。

|  |
| --- |
| **/\* copy\_file.c \*/**  #include <unistd.h>  #include <sys/types.h>  #include <sys/stat.h>  #include <fcntl.h>  #include <stdlib.h>  #include <stdio.h>  #define BUFFER\_SIZE 1 /\* 每次读写缓存大小，单位为字节，影响运行效率\*/  #define SRC\_FILE\_NAME "src\_file" /\* 源文件名\*/  #define DEST\_FILE\_NAME "dest\_file" /\* 目标文件名文件名\*/  #define OFFSET 5 /\* 复制的数据大小，单位为字节，即5个字节\*/  int main()  {  int src\_file, dest\_file;  unsigned char buff[BUFFER\_SIZE];  int real\_read\_len;  /\* 以只读方式打开源文件，若文件不存在则出错\*/  src\_file = **open(SRC\_FILE\_NAME, O\_RDONLY);**  /\* 以只写方式打开目标文件，若此文件不存在则创建该文件, 访问权限值为644 \*/  dest\_file = **open(DEST\_FILE\_NAME,**  **O\_WRONLY|O\_CREAT, S\_IRUSR|S\_IWUSR|S\_IRGRP|S\_IROTH);**  if (src\_file< 0 || dest\_file< 0)  {  printf("Open file error\n");  exit(1);  }  /\* 将源文件的读写指针移到最后5B的起始位置\*/  **lseek(src\_file, -OFFSET, SEEK\_END);**  /\* 读取源文件的最后5B数据并写到目标文件中（指针向前移动OFFSET），每次读写1B \*/  while ((real\_read\_len = read(src\_file, buff, sizeof(buff))) > 0)  {  **write(dest\_file, buff, real\_read\_len);**  }  **close(dest\_file);**  **close(src\_file);**  return 0;  } |

# gcc copy\_file.c -o copy\_file

# vim src\_file //创建源文件，内容为“123456789”

# ./copy\_file //执行程序

# cat dest\_file //查看目标文件，内容应为“6789”，即源文件最后5个字节的内容，因为每个数字占用1个字节位置，再加上结束符，共5个字节。

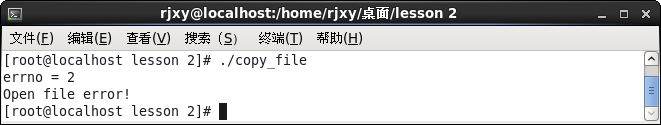
1. 用open()函数打开文件时，如果没有带O\_CREAT选项，则如果该文件不存在时，会提示错误，open()函数的返回值为-1，可以通过errno查看具体的错误信息。

|  |
| --- |
| 【注意】  errno用来记录系统调用时的错误信息，使用时需要添加errno.h头文件。errno的返回值为数字，可以通过《errno代码（中文）.txt》查看数字所对应的具体错误。 |

1. 添加头文件#include <errno.h>，
2. 在src\_file = open(SRC\_FILE\_NAME, O\_RDONLY)后面添加如下代码：

printf("errno = %d\n",errno);

1. 删掉src\_file，再次测试程序，可得到errno的信息，如下图所示。



1. 修改OFFSET的值，可分别获取源文件最后不同OFFSET大小的内容。
2. 将OFFSET修改为2，其他不变；
3. # gcc copy\_file.c -o copy\_file //重新编译
4. # rm dest\_file //删除以前的目标文件，避免影响测试结果，可在打开dest\_file时添加O\_TRUNC，就不需要每次测试时删除dest\_file
5. # ./copy\_file //执行程序，重新生成目标文件
6. # cat dest\_file //查看目标文件，内容应为“9”
7. 若要把源文件第N个字符到结束的所有内容送到目标文件中，需要作如下修改（注意，此时OFFSET值为2）：
8. 将lseek(src\_file, -OFFSET, SEEK\_END)改为lseek(src\_file, **OFFSET, SEEK\_SET**)
9. # gcc copy\_file.c -o copy\_file //重新编译
10. # rm dest\_file //删除以前的目标文件
11. # ./copy\_file //执行程序，重新生成目标文件
12. # cat dest\_file //查看目标文件，内容应为“3456789”，因为此时OFFSET值为2
13. 文件锁
14. fcntl()函数说明。

前面的这5 个基本函数实现了文件的打开、读写等基本操作，本小节将讨论的是，在文件已经共享的情况下如何操作，也就是当多个用户共同使用、操作一个文件时的情况。这时，Linux 通常采用的方法是给文件上锁，来避免共享的资源产生竞争的状态。

文件锁包括建议性锁和强制性锁。建议性锁要求每个上锁文件的进程都要检查是否有锁存在，并且尊重已有的锁。一般情况下，不建议使用建议性锁，因为无法保证每个程序都自动检查是否有锁。强制性锁是由内核执行的锁，当一个文件被上锁进行写入操作的时候，内核将阻止其他任何文件对其进行读写操作。采用强制性锁对性能的影响很大，每次读写操作都必须检查是否有锁存在。

在Linux 中，实现文件上锁的函数有lockf()和fcntl()，其中lockf()用于对文件施加建议性锁，而fcntl()不仅可以施加建议性锁，还可以施加强制锁。同时，fcntl()还能对文件的某一记录上锁，也就是记录锁。

记录锁又可分为读取锁和写入锁，其中读取锁又称为共享锁，它能够使多个进程都能在文件的同一部分建立读取锁。而写入锁又称为互斥锁，在任何时刻只能有一个进程在文件的某个部分上建立写入锁。

当然，在文件的同一部分不能同时建立读取锁和写入锁。

|  |
| --- |
| 【注意】  fcntl()是一个非常通用的函数，它可以对已打开的文件描述符进行各种操作，不仅包括管理文件锁，还包括获得或设置文件描述符和文件描述符标志、文件描述符的复制等很多功能。这里主要介绍建立记录锁的方法。 |

1. fcntl()函数格式。

用于建立记录锁的fcntl()函数格式如表6所示。

|  |  |  |
| --- | --- | --- |
| 表6 fcntl()函数格式 | | |
| 所需头文件 | #include <sys/types.h>  #include <unistd.h>  #include <fcntl.h> | |
| 函数原型 | int fcnt1(int fd, int cmd, struct flock \*lock) | |
| 函数传入值 | fd：文件描述符 | |
| cmd | F\_DUPFD：复制文件描述符 |
| F\_GETFD：获得fd的close-on-exec 标志，若标志未设置，则文件经过exec()函数之后仍保持打开状态 |
| F\_SETFD：设置close-on-exec 标志，该标志由参数arg的FD\_CLOEXEC 位决定 |
| F\_GETFL：得到open设置的标志 |
| F\_SETFL：改变open设置的标志 |
| F\_GETLK：获取当前的文件锁状态，存放在lock中 |
| F\_SETLK：根据lock参数值，设置文件锁 |
| F\_SETLKW：F\_SETLK 的阻塞版本（命令名中的W 表示等待（wait））。在无法获取锁时，会进入睡眠状态；如果可以获取锁或者捕捉到信号则会返回 |
| lock：结构为flock，设置记录锁的具体状态 | |
| 函数返回值 | 0：成功  -1：出错 | |

这里，lock 的结构如下所示：

|  |
| --- |
| struct flock  {  short l\_type;  off\_t l\_start;  short l\_whence;  off\_t l\_len;  pid\_t l\_pid;  } |

lock 结构中每个变量的取值含义如表7所示。

|  |  |
| --- | --- |
| 表7 lock结构中变量的取值含义 | |
| l\_type | F\_RDLCK：读取锁（共享锁） |
| F\_WRLCK：写入锁（排斥锁） |
| F\_UNLCK：解锁 |
| l\_start | 相对位移量（字节） |
| l\_whence：相对位移量的起点（同lseek的whence） | SEEK\_SET：当前位置为文件的开头，新位置为偏移量的大小 |
| SEEK\_CUR：当前位置为文件指针的位置，新位置为当前位置加上偏移量 |
| SEEK\_END：当前位置为文件的结尾，新位置为文件的大小加上偏移量的大小 |
| l\_len | 加锁区域的长度 |

|  |
| --- |
| 【小技巧】  为整个文件加锁，通常的方法是将l\_start设置为0，l\_whence设置为SEEK\_SET，l\_len设置为0。 |

1. fcntl()使用实例

下面首先给出了使用fcntl()函数的文件记录锁功能的代码实现。在该代码中，首先给flock结构体的对应位赋予相应的初值，接着使用两次fcntl()函数，分别用于判断文件是否可以上锁和给相关文件上锁，这里用到的cmd值分别为F\_GETLK 和F\_SETLK（或F\_SETLKW）。

用F\_GETLK 命令判断是否可以进行flock结构所描述的锁操作：若可以进行，则flock结构的l\_type会被设置为F\_UNLCK，其他域不变；若不可行，则l\_pid被设置为拥有文件锁的进程号，其他域不变。

用F\_SETLK 和F\_SETLKW 命令设置flock 结构所描述的锁操作，后者是前者的阻塞版。

文件记录锁功能的源代码如下所示：

|  |
| --- |
| **/\* lock\_set.c \*/**  int **lock\_set**(int fd, int type)  {  **struct flock lock;**  **lock.l\_whence = SEEK\_SET;**  **lock.l\_start = 0;**  **lock.l\_len = 0;**  **lock.l\_type = type; //初始化l\_type**  **lock.l\_pid = -1; //初始化l\_pid**  /\* 判断文件是否可以上锁\*/  **fcntl(fd, F\_GETLK, &lock);**  if (lock.l\_type != F\_UNLCK)  {  /\* 判断文件不能上锁的原因\*/  if (lock.l\_type == **F\_RDLCK**) /\* 该文件已有读取锁\*/  {  printf("Read lock already set by %d\n", lock.l\_pid);  }  else if (lock.l\_type == **F\_WRLCK**) /\* 该文件已有写入锁\*/  {  printf("Write lock already set by %d\n", lock.l\_pid);  }  }  /\* F\_GETLK获取的是文件fd的原始锁定状态，并将这个状态记录在lock结构体中，这个值和初始化的lock.l\_type没有关系。因为lock\_set()函数的第二个参数要求给文件fd添加type类型的锁，所以这里需要再次给lock.l\_type赋值为type，以便后面调用fcntl函数时，可通过F\_SETLKW或者F\_SETLK将lock.l\_type的设定值赋给文件fd，为文件fd加上符合要求的锁\*/  **lock.l\_type = type;**  /\* 根据不同的type 值进行阻塞式上锁或解锁\*/  if ((**fcntl(fd, F\_SETLKW, &lock)**) < 0)  {  printf("Lock failed:type = %d\n", lock.l\_type);  return 1;  }  switch(lock.l\_type)  {  case F\_RDLCK:  {  printf("Read lock set by %d\n", getpid());  }  break;  case F\_WRLCK:  {  printf("Write lock set by %d\n", getpid());  }  break;  case F\_UNLCK:  {  printf("Release lock by %d\n", getpid());  return 1;  }  break;  default:  break;  }/\* end of switch \*/  return 0;  } |

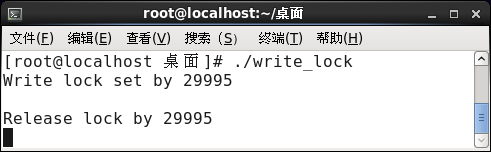
下面的实例是文件写入锁的测试用例，这里首先创建了一个hello文件，之后对其上写入锁，最后释放写入锁，代码如下所示：

|  |
| --- |
| **/\* write\_lock.c \*/**  #include <unistd.h>  #include <sys/types.h>  #include <sys/stat.h>  #include <stdio.h>  #include <stdlib.h>  #include <fcntl.h>  #include "lock\_set.c"  int main(void)  {  int fd;  /\* 首先打开文件\*/  fd = **open("hello",O\_RDWR | O\_CREAT, 0644);**  if(fd< 0)  {  printf("Open file error\n");  exit(1);  }  /\* 给文件上写入锁\*/  **lock\_set(fd, F\_WRLCK);**  getchar();  /\* 给文件解锁\*/  **lock\_set(fd, F\_UNLCK);**  getchar();  close(fd);  exit(0);  } |

# gcc write\_lock.c -o write\_lock //编译

下面是在PC 机上的运行结果。为了使程序有较大的灵活性，我们采用文件上锁后由用户键入一任意键使程序继续运行。建议读者开启两个终端，并且在两个终端上同时运行该程序，以达到多个进程操作一个文件的效果。在这里，我们首先运行终端一，请读者注意终端二中的第一句。

终端一：



终端二：



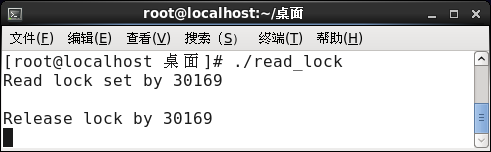
由此可见，写入锁为互斥锁，同一时刻只能有一个写入锁存在。

接下来的程序是文件读取锁的测试用例，原理和上面的程序一样。

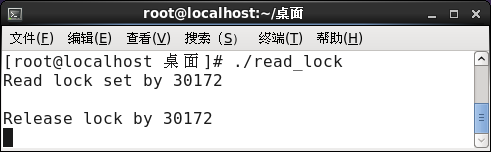
|  |
| --- |
| **/\* fcntl\_read.c \*/**  #include <unistd.h>  #include <sys/types.h>  #include <sys/stat.h>  #include <stdio.h>  #include <stdlib.h>  #include <fcntl.h>  #include "lock\_set.c"  int main(void)  {  int fd;  fd = **open("hello",O\_RDWR | O\_CREAT, 0644);**  if(fd< 0)  {  printf("Open file error\n");  exit(1);  }  /\* 给文件上读取锁\*/  **lock\_set(fd, F\_RDLCK);**  getchar();  /\* 给文件解锁\*/  **lock\_set(fd, F\_UNLCK);**  getchar();  close(fd);  exit(0);  } |

同样开启两个终端，并首先启动终端一上的程序，其运行结果如下所示：

终端一：



终端二：



大家可以将此结果与写入锁的运行结果相比较，可以看出，读取锁为共享锁，当进程30169已设置读取锁后，进程30172仍然可以设置读取锁。

|  |
| --- |
| 【思考】  如果在一个终端上运行设置读取锁的程序，则在另一个终端上运行设置写入锁的程序，会有什么结果呢？ |

【注意】在程序中加上读写锁之后读写锁并没有生效，要使读写锁生效，必须使用-o mand 选项来重新挂载根分区，这才能在文件系统级别使能强制锁功能，同时要修改文件的权限，添加setgid权限，去掉组群的可执行权。

1. # mount -o remount,mand / #重新挂载根分区，并增加mand选项
2. # chmod g+s hello #给文件hello添加setgid权限
3. # chmod g-x hello #给文件hello去掉组群的可执行权