**中南林业科技大学**操作系统**课程实验指导书**

目录

[实验一 握Linux操作系统安装 3](#_Toc139031832)

[实验二 观察Linux行为，使用proc文件系统 23](#_Toc139031833)

[实验四 增加Linux系统调用 29](#_Toc139031834)

[实验五：添加内核模块 34](#_Toc139031835)

[实验6：分析源代码 37](#_Toc139031836)

[实验7显示进程列表 38](#_Toc139031837)

[实验8 添加/proc文件系统 41](#_Toc139031838)

[实验9 设备驱动程序 43](#_Toc139031839)

课程编码： B11020080

课程名称： 操作系统

学 时： 16学时（16学时+1周课程设计）

先修课程：程序设计初步、汇编语言、数据结构、计算机原理。

授课重点: Linux内核编译过程、添加系统调用、实现动态加载的内核模块、实现同步事件原语和虚拟管道设备。

授课难点：Linux的系统调用机制、内核同步互斥函数的使用

阅读章节：《深入理解Linux内核》

《Linux源代码情境分析》

《Linux内核驱动程序》

实验内容：包含与Linux内核相关的七个实验，目的是使学生接触到实际操作系统内核的基本功能机制，包括系统调用机制、内核的同步互斥机制、动态内核模块加载机制、驱动程序接口。其中前5个实验是必需完成的，后2个实验有一定的难度，可选做。每个实验都需要提交实验报告。

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 实验项目名称 | 实验内容 | 学时 | 分组情况 | 实验性质 | 实验要求 | 备注 |
| 1 | 操作系统安装 | 掌握Linux操作系统安装及基本操作 | 2 | 1人1组 | 自主性 | 必做 | 实验部分 |
| 2 | Proc文件系统 | 学习和掌握proc文件系统的功能、工作原理及其应用 | 2 | 1人1组 | 自主性 | 必做 | 实验部分 |
| 3 | 编译内核 | 学习和掌握Linux内核编译过程 | 2 | 1人1组 | 自主性 | 必做 | 实验部分 |
| 4 | 添加系统调用 | 学习和掌握系统调用机制，增加新的系统调用 | 4 | 1人1组 | 自主性 | 必做 | 实验部分 |
| 5 | 编写内核模块 | 学习和掌握模块加载机制，增加新的内核功能 | 4 | 1人1组 | 自主性 | 必做 | 实验部分 |
| 6 | 源码分析 | 学习和掌握分析内核代码的方法 | 2 | 1人1组 | 自主性 | 必做 | 实验部分 |
| 7 | 显示进程列表 | 掌握进程结构及进程管理原理 | 2 | 1人1组 | 自主性 | 必做 | 课设部分 |
| 8 | 创建proc文件 | 创建proc文件 | 2 | 1人1组 | 自主性 | 必做 | 课设部分 |
| 9 | 设备驱动程序设计（1） | 了解Linux设备驱动程序接口，实现块设备的驱动程序RAMDISK | 2 | 1人1组 | 自主性 | 必做 | 课设部分 |
| 10 | 设备驱动程序设计（2） | 通过分析Linux的USB驱动框架程序（usb-skeleton.c）和USB大规模存储设备驱动程序（usb-storage）的源代码，结合前面的RAMDISK驱动程序，设计开发一个U盘驱动程序。 | 4 | 1人1组 | 自主性 | 必做 | 课设部分 |

# 实验一 掌握Linux操作系统安装

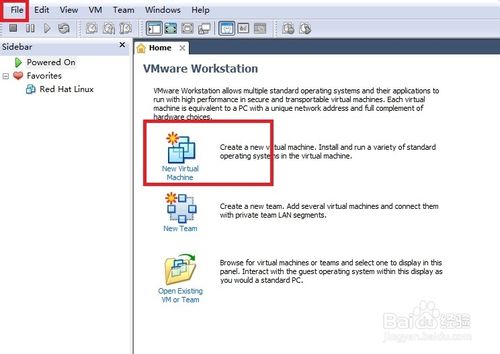
## 实验目的：掌握Linux操作系统安装及基本操作

## 1.1工具与原料

虚拟机软件、linux系统镜像iso文件

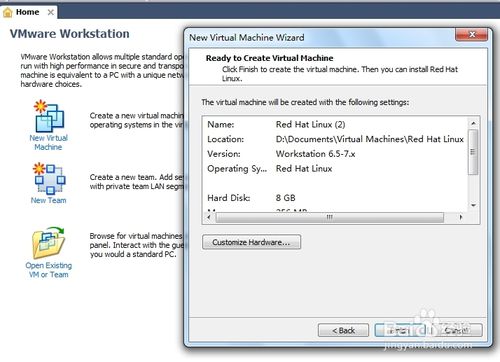
## 1.2．方法与步骤

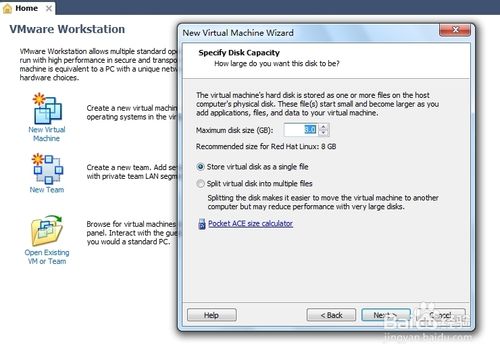
1. 先下载虚拟机软件，再安装好,下载linux系统镜像iso文件,一共有3个，大概有1.8G,新建一个虚拟机，如图片中的红圈所示

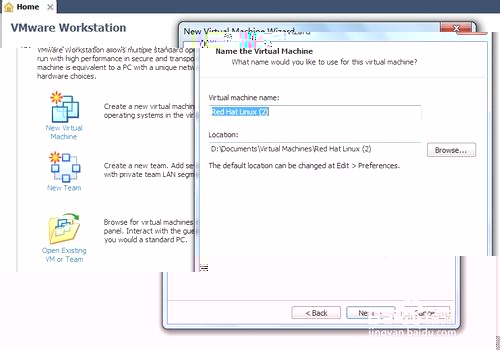
[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=1)

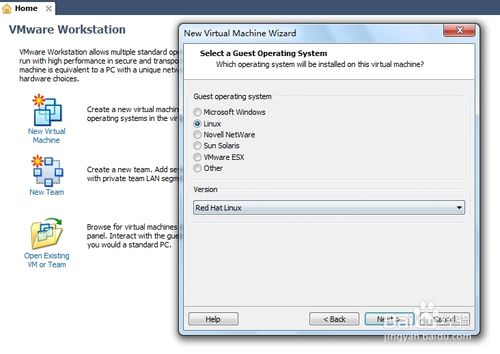
1. 2

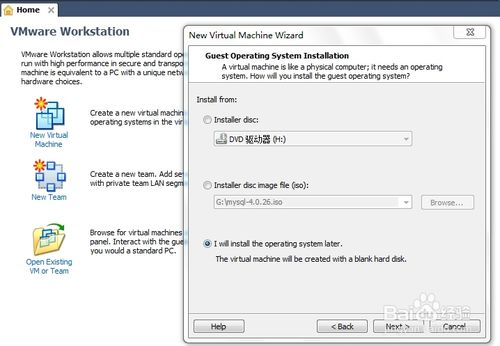
2.安装虚拟机，一切选项可以选择默认，最后点击Finish即可，虚拟机安装完成

[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=2)

[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=3)

[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=4)

[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=5)

[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=6)

[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=7)

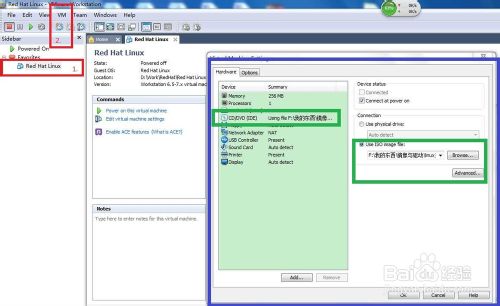
1. 3

好了，各位新，虚拟机安装好了，接下来就开始安装Linux系统了...

1.先鼠标选择左键图红色部分1处

2.再单击图红色部分2处的VM，选择弹出的settings...

3.再弹出的蓝色框中选中绿色的框，将镜像文件选择为下载的shrike\_i386-disc1.iso即可，就可以开始安装Linux系统了。

[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=8)

1. 选择skip，此步为是否检测linux系统,我们选择"skip",跳过检测,并回车,进入图形界面，选择下一步.

[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=9)

[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=10)

1. 语言选择“简体中文”，点击下一步。

键盘类型，选择默认选项，再点击下一步。

鼠标类型，默认，不改变。下一步

[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=11)

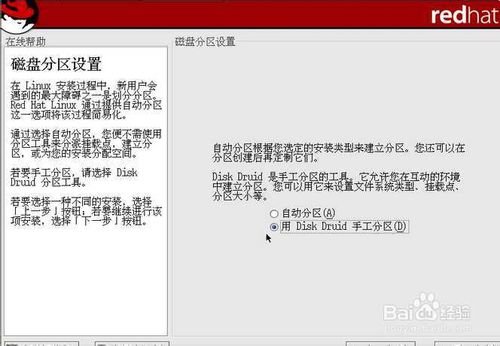
[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=12)

[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=13)

1. 安装类型，由于一般安装linux为了学习，软件开发，服务器。推荐选择"定制".

磁盘分区设置，选择"手工分区".

在弹出的警告框，选择"是" .

[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=14)

[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=15)

[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=16)

1. Linux分区是关键【重要】.

Linux系统一般的分区原则：【适合新手】

1. /boot 分区大小100M

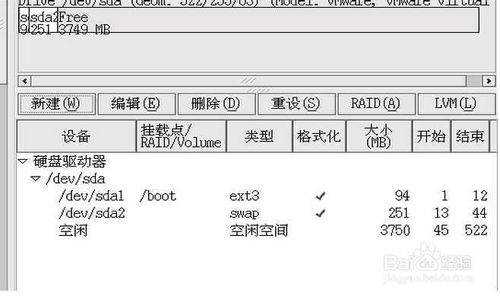
2. swap交换分区一般是你的物理内存2倍

3. / 根分区尽可能大

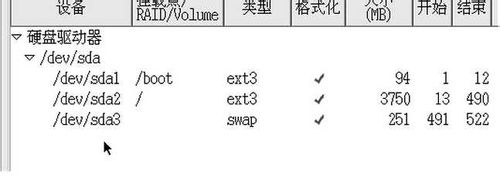
[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=17)

[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=18)

[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=19)

[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=20)

[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=21)

[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=22)

1. 默认选项，点击”下一步“。

防火墙选项，随便选择。

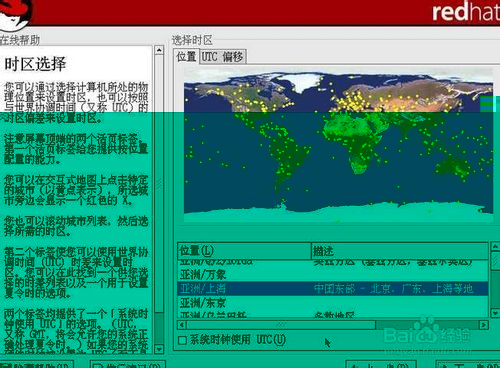
系统默认语言选择中文。

时区选项，选择"亚洲上海"默认即可。

[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=23)

[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=24)

[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=25)

[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=26)

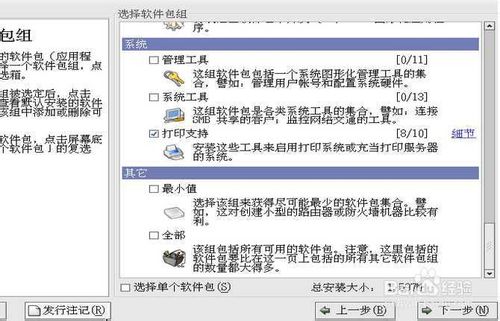
1. 管理员口令设置。即设置管理员密码，密码设置后要记住。本人设置为”24680ok"。

验证设置，默认即可。

[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=27)

[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=28)

1. 软件组件选择，不同用户应该根据自己的情况选择。如果不确定，且电脑硬盘充足，可以全部选择。

[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=29)

1. 安装确认，请点击下一步。

然后在点击”确定“，继续安装，当第一张光盘安装完成后，会弹出提示。

第二张盘安装完成后，会弹出安装第三张盘。

[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=30)

[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=31)

[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=32)

[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=33)

1. 安装完成后，不创建引导盘。

默认选项，下一步。

显示器配置，默认。

[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=34)

[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=35)

[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=36)

1. 默认图形化登录。

安装完成，并退出。

[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=37)

[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=38)

[](http://jingyan.baidu.com/album/4b52d702735fdafc5c774bc5.html?picindex=39)

# 实验二 观察Linux行为，使用proc文件系统

**实验目的**：学习Linux内核、进程、存储和其他资源的一些重要特征。读/proc/stat文件，计算并显示系统CPU占用率和用户态CPU占用率。（编写一个程序使用/proc机制获得以及修改机器的各种资源参数。需要使用fopen()，fscanf()，printf()）

## 2.1介绍

Linux操作系统提供了一套非常有用的在用户态检查内核状态的机制——/proc文件系统。在该文件系统将 进程的地址空间，系统的硬件信息（包括CUP、内存状态以及网卡等各种硬件设备）以及操作系统系统（中断、IO）等内容全部设置成虚拟的Linux文件——它只存在内存当中，而不占用外存空间。它以文件系统的方式为访问系统内核数据的操作提供接口。

/proc 中的每个文件都有一组分配给它的非常特殊的文件许可权，并且每个文件属于特定的用户标识，这里面的文件仅仅包含以下三种权限（除非root用户特别授权）：

1. 只读：任何用户都不能更改该文件；它用于表示系统信息；
2. root 写：如果 /proc 中的某个文件是可写的，则通常只能由 root 用户来写；
3. root 读：有些文件对一般系统用户是不可见的，而只对 root 用户是可见的；
4. 其它：出于各种原因，您可能会看到不同于上面常见的三种许可权的组合。

当然，除了/proc/sys目录以外，其余大部分目录下的文件对于root用户是只读的，该目录下存放的是内核参数，并被设计成为运行时刻修改的。

以下列出了/proc文件系统各个子目录的内容：

|  |  |
| --- | --- |
| 目录名 | 内容描述 |
| apm | 高级电源管理信息 |
| cmdline | 内核命令行 |
| cpuinfo | cup信息 |
| devices | 可用的设备信息 |
| dma | 已经使用的 DMA 管道 |
| filesystems | 系统支持的文件系统 |
| interrupts | 中断信息 |
| ioports | 端口使用信息 |
| kcore | 内核核心映像 |
| kmsg | 内核消息 |
| ksyms | 内核符号表 |
| loadavg | 平均负载 |
| locks | 内核锁 |
| meminfo | 内存信息 |
| misc | misc信息 |
| modules | 内核加载模块列表 |
| mounts | 已加载文件信息 |
| partitions | 系统识别的分区表 |
| rtc | 实时锁 |
| slab | slab 对象池的信息 |
| stat | 全面信息统计状态表 |
| swaps | 交换分区使用情况 |
| version | 内核版本 |
| uptime | 系统正常运行时间 |
| sys | 内核参数 |

并不是所有的目录在你的系统中都有，这取决于你内核的配置和装载的模块。另外，/proc目录下还有两个很重要的目录：net和scsi，均依赖于内核的配置。例如，如果你的系统不支持sici设备则sici目录不存在。

除此之外，你还会发现一些以数字命名的目录，它们是进程目录。系统中当前运行的每一个进程都有一个对应的目录在/proc下，以进程的ID号为目录名，他们就是读取进程信息的接口。每个进程目录的结构如下：

|  |  |
| --- | --- |
| 子目录名称 | 包含内容 |
| cmdline | 该进程的命令行参数 |
| enviroment | 进程换进变量的值 |
| fd | 该进程所打开的文件的描述符 |
| mem | 该进程的内存使用情况 |
| stat | 进程状态 |
| cwd | 进程的当前目录 |
| root | 进程的根目录 |
| maps | 内存映像 |
| statm | 进程内存状态信息 |
| exe | 当前进程的可执行文件（链接） |

如果你要察看系统信息，可以使用cat命令，例如：

# cat /proc/interrupts

CPU0

　 0:　 8728810　　　　 XT-PIC　timer

　 1:　　　 895　　　　 XT-PIC　keyboard

　 2:　　　　 0　　　　 XT-PIC　cascade

　 3:　　 531695　　　　 XT-PIC　aha152x

　 4:　 2014133　　　　 XT-PIC　serial

　 5:　　 44401　　　　 XT-PIC　pcnet\_cs

　 8:　　　　 2　　　　 XT-PIC　rtc

11:　　　　 8　　　　 XT-PIC　i82365

12:　　 182918　　　　 XT-PIC　Mouse

13:　　　　 1　　　　 XT-PIC　fpu PS/2

14:　 1232265　　　　 XT-PIC　ide0

15:　　　　 7　　　　 XT-PIC　ide1

NMI:　　　　　0

前面已经说过，/proc/sys目录不仅提供了内核信息，而且可以通过它修改内核参数。（但是你必须很小心，因为可能会造成系统崩溃。最好是先找一台无关紧要的机子，调试成功后再应用到你的系统上。） 要改变内核的参数，只要用vi编辑或echo参数重定向到文件中即可。下面有一个例子：

# cat /proc/sys/fs/file-max

4096

# echo 8192 > /proc/sys/fs/file-max

# cat /proc/sys/fs/file-max

8192

## 2.2 实验内容

部分A

以超级用户的身份进入/proc目录，在终端上键入ls命令，察看该目录下的内容，同时察看每个文件的读写权限。

选择一台机器，试着回答下列问题：

* CPU的类型和型号。
* 所使用的Linux的版本。
* 从启动到当前时刻经过的时间。
* 内存状态。

部分B

* 编写一个程序，用来获得内核参数（任意的参数均可）。
* 编写一个程序，用来修改内核参数（任意的参数均可）。

## 2.3实验步骤：

1. 使用man命令了解/proc/stat文件的内容和格式：

man proc

2）使用man命令了解下列函数的使用方法：

fopen：

原型：FILE \*fopen( const char \*filename, const char \*mode );

以打开一个流(文件)。filename——文件名，mode——访问许可类型。

fscanf：

原型：int fscanf( FILE \*stream, const char \*format [, argument ]... );

从流中读取格式化的数据。

例子fscanf (fp ,"cpu %ld", &user)是指跳过字符串"cpu "及其前面的字符流，读取一个长整数并存入user中。其中fp为文件流指针。%ld对应长整数，%d对应整数，%s对应字符串。

fclose：

原型：int fclose( FILE \*stream );

关闭流(文件)。

3）用c语言编程

总CPU时间 = 用户占用时间+系统占用时间+优先进程占用时间+闲置进程占用时间

系统CPU占用率 = 系统占用时间 / 总CPU时间

用户态CPU占用率 = 用户占用时间 / 总CPU时间

4）编译和运行

## 2.4 解决方案

本练习完全不涉及到任何的内核编程，而完全使用libC库中的函数。

#include <stdio.h>

#include <sys/types.h>

#include <stdlib.h>

#include <fcntl.h>

...

int main (int argc, char\* argv[])

{

...

int fd=open(argv[1],FLAG,MODE); // 这里，文件作为参数传入。 FLAG和MODE

//的值由打开的功能决定

if(fd!=-1)

{

// 读/写相应的内核参数

...

close(fd);

}

else

{

//做错误处理

}

...

return EXIT\_SUCCESS;

}

# 实验三 Linux内核编译

原来的系统是redhat9.5，内核2.4.20-8，编译的内核2.4.32。

共分为四部分:编译前准备->编译新内核->运行新内核

## 3.1 编译前准备

1)下载一份内核源代码，例如linux-2.6.32.tar.gz，你可在如下地址下载它或者是更新的版本：

http://www.kernel.org/pub/linux/kernel/v2.4/

2) 检查redhat中是否已有模块工具软件module-init-tools（提供depmod [/sbin/depmod]等）：

# rpm –q modutils

如果没有或者版本太低，在如下地址下载最新版本 ：

http://www.kernel.org/pub/linux/kernel/people/rusty/modules/modutils-2.4.21-23.src.rpm

安装：

# rpm -ivh modutils-2.4.21-23.src.rpm

1. 将linux-2.6.32.tar.gz拷贝到目录/usr/src/下，解压源码：

# tar -zxvf linux-2.6.32.tar.gz

生成源码文件子目录/usr/src/linux-2.6.32，进入此目录：

# cd linux-2.6.32

## 3.2 编译新内核

1. 配置内核.

有三种方式配置内核：

# make config 命令行界面

# make menuconfig 字符菜单界面

# make xconfig 图形界面

虽然选择图形界面比较方便，但配置过程很繁琐，可将现有的配置文件拷贝过来使用（/usr/src/linux-2.6.32/.config）。

2) 编译生成新内核:

# cd /usr/src/linux-2.4.32

# make dep 创建代码依赖文件（.depend），每次重新配置后都必须做这一步。

# make bzImage 开始编译系统内核（不包括带M选项的模块），生成的压缩文件bzImage在./arch/i386/boot/下。同时生成未压缩的内核执行文件（vmlinux）和内核符号表（System.map）。

# make modules 开始编译外挂模块。以后重新编译内核时，可省去这一步。

# make modules\_install 将外挂模块放在系统模块安装目录（/lib/modules/2.4.32/）下，以便核心在需要时加载它们。同时在此目录下产生模块依赖文件（modules.dep）。

# make install 将bzImage和System.map拷贝到/boot/下（vmlinuz-2.4.32和System.map-2.4.32），并建立相应的符号链接（vmlinuz和System.map）；生成/dev/initrd映象文件（initrd-2.4.32.img）；在/etc/下的启动配置文件lilo.conf或grub.conf中添加相应项。

3) 运行新内核

# reboot 选择启动新内核：

需要注意的是，如果编译的内核版本号（在Makefile中定义）与正在运行的内核一样，就会覆盖现有内核的文件。为了防止新内核影响原内核，让新内核有一个不同的版本号。最好将编译内核前的虚拟机备份，以便在发生新内核导致系统无法正常运行时使用备份的系统。

4) 启动新内核后，需要重新运行VMware tools的配置程序/usr/bin/vmware-config-tools.pl，以使网络界面正常和共享windows主机的文件夹。

# 实验四 增加Linux系统调用

**实验目的**：学习如何产生一个系统调用以及怎样同过往内核中增加一个新函数从而在内核空间中实现对用户空间的读/写。

## 4．1 介绍

从这个实验开始，我们开始真正的接触Linux内核。Linux系统调用是用于导出可供用户空间程序调用的内核函数的名称，它们不能像普通的libc库中的函数一样调用，而必需通过0x80陷阱进核调用。这样，如果用户创建了一个新的内核函数，就必须在内核陷阱表中创建一个新的表项来引用该函数。

所有的内核函数入口表集中在/user/src/linux/arch/i386/kernel/entrys.S中。该表具有如下所示的形式：

.data

ENTRY(sys\_call\_table) /\* 入口 \*/

.long SYMBOL\_NAME(sys\_ni\_call) /\* 0， 空项 \*/

.long SYMBOL\_NAME(sys\_exit) /\* 1 \*/

.long SYMBOL\_NAME(sys\_fork)

.long SYMBOL\_NAME(sys\_read)

.long SYMBOL\_NAME(sys\_write)

.long SYMBOL\_NAME(sys\_open) /\* 5 \*/

...

.long SYMBOL\_NAME(sys\_signalstack)

.long SYMBOL\_NAME(sys\_sendfile)

...

.long SYMBOL\_NAME(sys\_ni\_call) /\* 空项 \*/

.long SYMBOL\_NAME(sys\_ni\_call)

...

.endr /\* 结束 \*/

表项1包含了exit()系统调用(其实现为sys\_exit()内核函数)，表项2包含了fork()系统调用，以此类推。

系统调用是在sys\_call\_table中定义的，这样当增加一个新的系统调用是，就必须在这个表中增加一个新的表项。编辑该文件，增加自己的系统调用如下：

...

.long SYMBOL\_NAME(sys\_ni\_call) /\* 222 \*/

.long SYMBOL\_NAME(sys\_my\_new\_call) /\*223，用户自定义系统调用\*/

...

注意，linux系统自身保留了221个系统调用。这就意味着，你自己增加的系统调用至少要在第222项以后开始，还要小于256，（内核2.4的有的版本可能多达237个，因此，进建议你选择较为靠后的序号）。另外，这个操作也只有超级用户才有权限完成。

为了使普通的C函数也能够在重新编译内核之后也能调用你的内核函数，最好还要编辑 内核目录下的include/asm/unistd.h文件。

#define \_\_NR\_exit 1

#define \_\_NR\_fork 2

#define \_\_NR\_read 3

#define \_\_NR\_write 4

#define \_\_NR\_open 5

...

#define \_\_NR\_my\_new\_call 223

接下来，就要实现自己的内核函数了。你可以单独在一个文件添加该函数，亦可以在以有的内核源码文件中添加。内核函数的实现体具有以下的形式：

asmlinkage <type> <func\_name> (paramlist){}

其中，asmlinkage关键字指名了函数的参数的传递方式——必须使用堆栈进行。这里涉及到一个问题——int 80 到底做了些什么？事实上，它要完成一系列的动作：

1. 在系统调用表中找出对应于系统调用号的表项；
2. 确定系统调用的参数条目；
3. 将参数从用户地址空间拷贝到u区；
4. 保存当前进程的上下文；
5. 调用内核中的系统调用代码；
6. 返回用户进程。

这就决定了实现内核函数时，并不能直接使用引用型参数——例如：指针型参数或者引用型参数（想一想为什么？），而须使用copy\_from\_user和copy\_to\_user这两个内核API（也可以使用memcpy\_fromfs和memcpy\_tofs）实现参数的读入/写回。

如何在用户态调用内核系统函数呢？原则上只有使用致限引发0x80中断一种方法。例如：当你调在用一个具有两个参数的你同调用sys\_xxx\_call时，若要传入参数param1和param2并且将返回值存放在res中，就要采用如下方法：

long res;

\_\_asm\_\_ volatile (“int $0x80” \

:”=a”(res) \

:”0” (\_\_NR\_sys\_xxx\_call), “b”((long)(param1)),\

“c”((long)(param2)));

return res;

另外，系统定义了syscall0~syscall5这六个宏，分别封装对就有0个到5个参数的内核函数的调用。

最后要注意的是，在编译使用内和函数的C文件时要通知编译器把这些代码当作内和代码而不是普通的用户代码来编译，这可以通过向编译器传递\_\_KERNEL\_\_标志来实现，使用-W编译选项来向装载程序传递all。如下所示：

gcc –Wall –D\_\_KERNEL\_\_ xxx.c

当然，如果你采用重新编译内核的方法，并不需要这样做——只要修改内核源码目录的Makefile，并将你的文件添加至O\_OBJS列表中就可以了。因为Makefile的FLAGS标志已经包含上述编译选项了。

## 4．2 问题陈述

部分A

添加一个新的内核系统调用，具体完成某个你希望实现的功能。

部分B

重新编译内核，使你的系统调用可用。

部分C

编写一个用户态的程序，验证你增加的系统调用。

## 4.3实验步骤：

1. 在arch/i386/kernel/entry.S文件中的系统调用入口表中，找到某个空项：

.long SYMBOL\_NAME(sys\_ni\_syscall)

修改为新加系统调用的入口项：

.long SYMBOL\_NAME(sys\_my\_call) /\* 254——系统调用号 \*/

记住系统调用号（在系统调用入口表中的第几个入口），例如254。

1. 在核心文件kernel/sys.c中添加实现系统调用my\_call()的代码段：

#include <linux/kernel.h>

#include <linux/module.h>

#include <linux/sched.h> /\* struct task\_struct \*/

asmlinkage int sys\_mycall(int number)

　　{

　　return number;

　　}

以下的例子实现了枚据当前所有进程，并且查询每个进程的gid，uid以及pid，并且将所的得结果（进程信息数组，进程数）返回用户空间：

asmlinkage int sys\_mysyscall(struct task\_info\* information, int\* count)

{

int i = 0；

struct task\_struct \*p;

struct task\_info \*info\_head, \*pinfo;

//allocate the space to collect the information

if (!(info\_head = (struct task\_info \*)\

kmalloc(MAX\_N\_TASKS\*sizeof(struct task\_info),GFP\_KERNEL))) {

printk("No memory for this query.\n");

return -1;

};

//get the information of each task

pinfo = info\_head;

p = &init\_task;

do {

//get the information of the task

pinfo -> pid = p -> pid;

pinfo -> uid = p -> uid;

pinfo -> gid = p -> gid;

++ pinfo ;

i ++;

} while ((p = p -> next\_task ) && (p != &init\_task));

/\* put the data to the user space \*/

copy\_to\_user(count, &i, sizeof(int));

copy\_to\_user(information, info\_head, i \* sizeof(struct task\_info));

kfree(pinfo);

return 0;

}

1. 生成和运行新内核

make bzImage

make install

reboot——选择新内核运行

1. 在/usr/include/asm/unistd.h文件中增加：

#define \_\_NR\_get\_cputime 254

1. 编写c程序使用新的系统调用

在应用程序中使用新添加的系统调用mycall。同样为实验目的，我们写了一个简单的例子xtdy.c。

　　/\* xtdy.c \*/

#define \_\_NR\_my\_syscall 250

int errno;

　　#include <linux/unistd.h>

　　\_syscall1(int,mycall,int,ret)

　　main()

　　{

　　printf("%d \n",mycall(100));

　　}

　　编译该程序：

　　# cc -o xtdy xtdy.c

　　执行：

　　# xtdy

　　结果：

　　# 100

　　注意，由于使用了系统调用，编译和执行程序时，用户都应该是超级用户身份。

## 4．4 留下一个干净的环境

修改内核目录中的文件内容。但在完成测试之后，应该记得把环境恢复到最初的状态。如果你在一个环境中工作，但是似乎有什么不对劲，那么可能已经有人污染了这个环境。如果试图在已经污染过环境中工作，那么你的解决方案恐怕就有问题了。因此在能够完成自己的工作之前，一定将工作恢复到原始状态。

# 实验五 添加内核模块

## 实验目的：学习模块，它是Linux特有的一种机制。模块可以用来动态地增加内核的功能。

## 5.1介绍

Linux模块(module)是一些可以作为独立程序来编译（使用适当的标志说明这是内核代码）的函数和数据类型的集合。在装载这些模块时，将它的代码链接到内核中。Linux模块可以在内核启动过程时装载――这称为*静态装载*。它们也可以在内核运行的过程中装载，这称为*动态装载*。

Linux模块可以单独编译成为目标代码，module是一个目标文件。它可以根据需要在系统启动后动态地加载到系统核心之中。当module不再被需要时，可以动态地卸载出系统核心。Linux中大多数设备驱动程序或文件系统都作成的module。超级用户可以通过insmod和rmmod命令显示地将module载入核心或从核心中将它卸载。

核心也可在需要时，请求守护进程(kerneld)装载和卸载module。通过动态地将代码载入核心可以减小核心代码的规模，使核心配置更为灵活。若在调试新核心代码时采用module技术，用户不必每次修改后都需重新编译核心和启动系统。

## 5.2模块的组织结构

一旦Linux module载入核心后，它就成为核心代码的一部分。它与其它核心代码的地位是相同的。module在需要的时可通过*内核符号表*(symbol table)使用核心资源。核心将资源登记在符号表中，当module装载时，核心利用符号表来解决module中资源引用的问题。Linux中允许module堆栈(module stacking)，既一个module可请求其他module为之提供服务。当module装载入系统核心时，系统修改核心中的符号表，将新装载module 提供的资源和符号加到核心符号表中。通过这种通信机制，新载入的module可以访问已装载的module提供的资源。通过/proc/ksyms文件可以查看内核符号表中各类符号及其地址。

若某个module空闲，用户便可将它卸载出核心。在卸载之前，系统释放分配给该module的系统资源，如核心内存，中断等。同时系统将该module提供的符号从内核符号表中删除。

由于module 中代码与核心中其他部分代码的地位是相同的，module的代码错误会导致系统崩溃。而且module一般需要调用核心的资源，所以必须注意module的版本和核心的版本的相配问题。一般在module的装入过程中检查module的版本信息。

模块链接到内核的过程见图5.1。

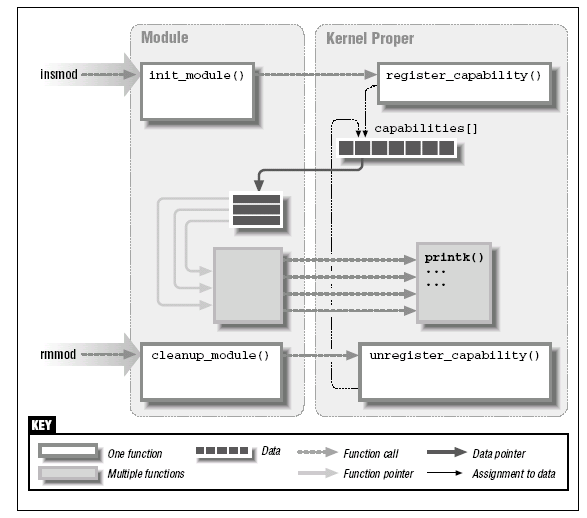


图5.1 将模块链接到内核

因为模块不和函数库连接，所以在源文件中不能包含通常的头文件（为用户态编程而准备）。内核模块只能使用作为内核一部分的函数。和内核相关的的所有内容都在内核源代码（通常位于/usr/src/linux-2.4）的include目录下。

一个内核模块必须至少有两个功能: init\_module 在该模块被插入内核时被调用, cleanup\_module 仅仅在它被清除前调用。 典型的, init\_module 要么在内核里为什么东西登记一个指针，要么用它自己的代码代替内核的某个功能 (通常那个代码做一些事情然后调用原始的功能). cleanup\_module 功能被假定撤消init\_module 做的任何事情, 因此模块可以被安全地卸载。

## 5.3模块的装载与卸载

模块在用户空间中(使用适当的标志)进行编译，结果产生一个可执行格式的文件。如前所述，模块可以引用内核变量，其条件是内核代码已编写好，从而允许变量被另一个模块所引用。简单说来，一个模块可以导出自己的符号，这样其他模块就可以使用它们。

在用insmod命令装载一个模块时(见下面的解决问题部分)，将会发生以下事件：

1. 新模块(通过内核函数create\_module())加入到内核地址空间。
2. 另外一个内核函数get\_kernel\_syms()，通过查找导出的内核符号对模块中的外部符号引用进行解析，导出内核符号之后是一个已经装载到系统中的其他模块的列表。这意味着如果一个模块引用了其他模块中定义的符号，那么定义这个符号的模块就必须在那些引用这个符号的模块装载进系统之前被装载。
3. create\_module()为这个模块分配内存空间。
4. 通过init\_module()系统调用装载模块。该模块定义的符号在此时被导出，供其他可能后来装载的模块使用。
5. Insmod为新装载的模块调用init\_module()函数

当模块卸载时（通过使用rmmod，见下面的“解决问题”部分），调用cleanup\_module()函数。释放模块所使用的空间，同时取消虚拟地址的映象。

**5.4 范例 hello.c**

/\* hello.c

\*

\* "Hello, world" - 内核模块版本.

\*/

/\* 必要的头文件 \*/

/\* 我们在做内核的工作 \*/

#ifndef \_\_KERNEL\_\_

# define \_\_KERNEL\_\_

#include <linux/version.h>

#include <linux/config.h>

#include <linux/kernel.h> /\* 我们进行核心编程\*/

#include <linux/module.h> /\* 我们进行module编程\*/

/\* 处理 CONFIG\_MODVERSIONS \*/

#if CONFIG\_MODVERSIONS==1

#define MODVERSIONS

#endif

/\* 初始化模块 \*/

int init\_module()

{

printk("Hello, world - this is the kernel speaking\n");

/\* 如果我们返回一个非零值, 那就意味着

\* init\_module 初始化失败并且内核模块

\* 不能加载 \*/

return 0;

}

/\* Cleanup - 撤消 init\_module 所做的任何事情 \*/

void cleanup\_module()

{

printk("Short is the life of a kernel module\n");

}

# 实验六 分析源代码

## 6.1 实验目的：

通过阅读源代码，分析研究linux的进程调度策略和算法。要求以源码为依据，回答下面的问题：

* 进程调度队列是如何组织的
* 三种调度类型（SCHED\_FIFO, SCHED\_RR, SCHED\_OHTER）的实现过程
* 优先级是如何定义和动态变化的
* 时间片的赋值？它与优先级的关系？
* 对实时进程和多CPU的支持
* 评价linux的调度策略，提出改进意见。

## 6.2 实验步骤

1. 重点分析内核数据结构task\_struct（在include/linux/sched.h中）和调度函数schedule()（在kernel/sched.c中）
2. 使眨grep命令查找相关变量或函数的引用情况，例如\*

grep “schedule(” \*

可以列出当前目录下所有文件中引用函数schedule()的代码行。

1. 参考有关源码分析资料
2. 以源码为依据，写出关于上述问题的分析报告。

# 实验七 显示进程列表

**练习目的**：编写一个模块，将它作为Linux内核空间的扩展来执行，并报告模块加载时内核的当前进程信息，进一步了解用户空间和内核空间的概念。

## 7.1进程

进程是任何多道程序设计的操作系统中的基本概念。为了管理进程，内核必须对每个进程所做的事情进行清楚的描述。例如，内核必须知道进程的优先级，它是正在CPU上运行还是因某些事件被阻塞，给它分配什么样的地址空间，允许它访问哪个文件等等。这正是进程描述符(process descriptor)的作用进程描述符都是task\_struct 类型结构，它的域包含了与一个进程相关的所有信息。因为进程描述符中存放了那么多信息，所以它是相当复杂的。它本身不仅包含了很多域，而且一些域还包括了指向其他数据结构的指针，依此类推。

为了对给定类型的进程(例如，在可运行状态的所有进程)进行有效的搜索，内核建立了几个进程链表。有一个双向循环链表(参见图3．3)把所有现有的进程联系起来，我们叫它为进程链表(process list)。每个进程的prev\_task和next\_task域用来实现链表。init\_task指向描述符链表的头。另外，宏current总是指向CPU的当前进程。

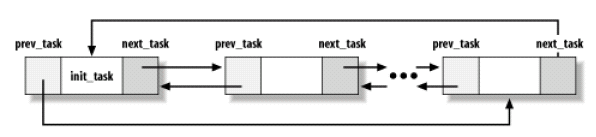


图4.2 进程链表

## 7.2用户空间和内核空间

模块运行在所谓的内核空间（Kernel space）里，而应用程序运行在所谓的用户空间里（User space）中，这个概念是操作系统理论的基础之一。

我们通常将执行模式称作内核空间和用户空间。这两个术语不仅说明两种模式具有不同的优先权等级，而且还说明每个模式都有自己的内存映射，即自己的地址空间。

每当应用程序执行系统系统调用和被中断挂起时，Unix将执行模式从用户空间切换到内核空间，执行系统调用的内核代码运行在进程上下文中，它代表调用进程执行操作，因此能够访问进程地址空间的所有数据；而处理硬件中断的内核代码则和进程是异步的，因而与任何一个特定进程都无关。

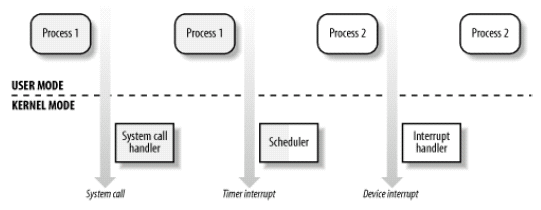


图4.3 用户空间和内核空间切换的示例

## 7.3问题陈述

设计并实现一个模块，遍历进程描述符链表，打印出系统的进程数目、当前进程，并尽量多地打印每个进程的信息。

### 7.3.1部分A

查看系统已加载模块。写一个小的模块打印“hello world”。

### 7.3.2部分B

设计模块遍历进程描述符链表，打印出系统的进程数目、当前进程，并尽量多地打印每个进程的信息。例如：进程PID，进程状态等。

### 7.3.3部分B（可选）

使用模块实现一个系统调用（也实现遍历进程链表的功能），并编写一个用户程序测试。体会利用模块从而带来的不用重新编译内核、调试方便等好处。

## 7.4解决问题

1. 编译内核模块

gcc –Wall –DLINUX –I/usr/src/linux-2.4.32/include –c xxxx.c

一旦设计并编写好模块，必须将其编译成一个适合内核装载的对象文件。需要通知编译程序把这个模块作为内核代码而不是普通的用户代码来编译，这可以通过向编译程序传递一一KERNEL—标志来实现(注意在KERNEL前后各有两个下划线字符)o使用MODULE编译时标志来通知编译程序该文件是一个模块而不是一个普通文件。使用一W编译选项来向装载程序(loader)传递all，使用-c开关通知编译程序在编译完这个文件之后不调用链接程序。

1. 装载模块

任何具有超级用户权限的用户都可以在命令行中装载和卸载模块。命令行/sbin/insmod<module\_name>将执行本练习前面说明的操作来完成整个模块的装载工作。为检查以确保insmod正常工作，在装载模块之前打印/proc/modules，装载之后再打印一次。在使用完模块之后，切记使用/sbin/rmmod命令将其卸载。

insmod xxxx.o

lsmod | grep xxxx

1. 读/proc/procstat文件

cd /proc

cat procstat

1. 卸载模块

rmmod xxxx

lsmod | grep xxxx

## 7.5进程描述符

查看linux/sched.h文件可以得到task\_struct 类型结构定义，试着分析各个域的含义。

下列代码可以遍历进程描述符链表：

struct task\_struct \*p;

p = &init\_task;

do {

……

} while ((p = p -> next\_task ) && (p != &init\_task));

## 7.6更多帮助

有关模块的内容还有很多。本练习主要来自http://www.ibiblio.org/mdw/LDP/lkmpg/的Linux文档工程中出现的由Ori Pomerantz所著的有关模块的文章。

参考书：《Linux设备驱动程序》（第二版）

# 实验八 添加/proc文件系统

## 8.1 实验目的：

通过加载内核模块，为/proc文件系统创建以下内容：

一个名叫proc\_test的子目录。

一个名叫current的文件，只读，读出的内容是读它的进程的情况。

一个名叫current\_too的链接，指向current。

一个名叫hello的文件，可读可写。读出的内容是上次写的内容前面加两句话。

## 8.2 实验步骤：

(1) 请大家互相用U盘copy上面的例子

proc\_test.c

使用U盘的方法：

1. 打开一个终端窗口，插入U盘

2.输入 mount –tvfat /dev/sdb1 /mnt

3.U盘的内容可以在/mnt下读写

4.使用完毕，输入 umount /dev/sdb1

5.取出U盘

(2) 编译这个模块。

gcc -c -I/usr/src/linux-2.4.20-8/include proc\_test.c

(3) 加载这个模块。

insmod proc\_test.o

用lsmod命令检查是否已经加载成功。

如果加载不成功，请找我检查。

* 看看是不是多了一个/proc/proc\_test目录。（输入以下两行命令）

cd /proc

ls

* 看看有没有三个文件，名字叫current，current\_too和hello。

cd /proc/proc\_test

ls

* 看看current文件的内容：

more current

* 看看current\_too文件的内容：

可以看出这个文件的内容和current差不多，其实这个文件是一个链接，指向current。

* 看看hello文件的内容。
* 试试写一下hello这个文件。

打开任意文本编辑器，如gedit，用编辑器打开/proc/proc\_test/hello这个文件，然后清除里面的全部内容，再随便写一句话，保存。

* 再次看看hello文件的内容：

more hello

(4) 卸载模块： rmmod proc\_test

# 实验九 设备驱动程序

在钻研Linux内核的人当中，大多数人都是在写设备驱动程序。尽管由于设备的特殊性，使得每个驱动程序都不一样。但是编写设备驱动程序的许多原则和基本技巧都是一样的，甚至Windows下的设备驱动程序从原理上讲也与之相通。在这一章的练习中，我们将通过分析一个典型的块设备RAM-DISK的驱动程序，学习编写设备驱动程序的一般过程。作为练习，需要将这个RAM-DISK的程序改造成为U盘的驱动程序，并通过它来使用你的U盘。

值得注意的是，在目前常用的Linux内核2.4版本中，设备驱动程序的编写方法发生了比较大的变化，然而目前大部分参考书中仍然以内核2.2甚至2.0版本作为例子。如果本章的内容与其他参考书不符，请留意是否由于内核版本的不同而造成了这些不一致。

## 9.1 Linux下设备驱动程序的基本结构

我们知道，系统调用是操作系统内核和应用程序之间的接口。设备驱动程序是操作系统内核和机器硬件之间的接口，设备驱动程序为应用程序屏蔽了硬件的细节，这样在应用程序看来，硬件设备只是一个设备文件，应用程序可以象操作普通文件一样对硬件设备进行操作。

同样，在Linux系统里，对用户程序来说，设备驱动程序隐藏了设备的具体细节，对各种不同设备提供了一致的接口和虚拟的设备文件，用户程序可以象对其它文件一样对设备文件进行操作。Linux对硬件设备支持两个标准接口：块设备文件和字符设备文件，通过这些设备文件存取的设备称为块设备或字符设备。

块设备接口仅支持面向块的I/O操作，所有I/O操作都通过在内核地址空间中的I/O缓冲区进行，它可以支持几乎任意长度和任意位置上的I/O请求，即提供随机存取的功能。

字符设备接口支持面向字符的I/O操作，它不经过系统的快速缓存，所以它们负责管理自己的缓冲区结构。字符设备接口只支持顺序存取的功能，一般不能进行任意长度的I/O请求，而是限制I/O请求的长度必须是设备要求的基本块长的倍数。

设备由一个主设备号和一个次设备号标识。主设备号唯一标识了设备类型，即设备驱动程序类型，它是块设备表或字符设备表中设备表项的索引。次设备号仅由设备驱动程序解释，一般用于识别在若干可能的硬件设备中，I/O请求所涉及到的那个设备。比如有两个软盘，就可以用从设备号来区分他们。设备文件的的主设备号必须与设备驱动程序在登记时申请的主设备号一致，一些常见的设备如硬盘，其主设备号都是固定的。

最后必须提到的是，在用户进程调用驱动程序时，系统进入核心态。这时不再是抢先式调度。也就是说，系统必须在你的驱动程序的子函数返回后才能进行其他的工作。如果你的驱动程序陷入死循环，那么整个系统也会死机，这时只能重新启动计算机。

设备驱动程序可以分为三个主要组成部分：

(1) 自动配置和初始化子程序，负责检测所要驱动的硬件设备是否存在和是否能正常工作。如果该设备正常，则对这个设备及其相关的、设备驱动程序需要的软件状态进行初始化。这部分驱动程序仅在初始化的时候被调用一次。

(2) 服务于I/O请求的子程序，又称为驱动程序的上半部分。调用这部分是由于系统调用的结果。这部分程序在执行的时候，系统仍认为是和进行调用的进程属于同一个进程，只是由用户态变成了核心态，具有进行此系统调用的用户程序的运行环境，因此可以在其中调用sleep()等与进程运行环境有关的函数。

(3) 中断服务子程序，又称为驱动程序的下半部分。在Linux系统中，并不是直接从中断向量表中调用设备驱动程序的中断服务子程序，而是由Linux系统来接收硬件中断，再由系统调用中断服务子程序。中断可以产生在任何一个进程运行的时候，因此在中断服务程序被调用的时候，不能依赖于任何进程的状态，也就不能调用任何与进程运行环境有关的函数。因为设备驱动程序一般支持同一类型的若干设备，所以一般在系统调用中断服务子程序的时候，都带有一个或多个参数，以标识请求服务的设备。

在系统内部，I/O设备的存取通过一组固定的入口点来进行，这组入口点是由每个设备的设备驱动程序提供的。一般来说，设备驱动程序需要提供如下几个入口点：

(1) open入口点：打开设备准备I/O操作。当应用程序对设备文件进行打开操作，都会调用设备的open入口点。open子程序必须对将要进行的I/O操作做好必要的准备工作，如清除缓冲区等。如果设备是独占的，即同一时刻只能有一个程序访问此设备，则open子程序必须设置一些标志以表示设备处于忙状态。

(2) close入口点：关闭一个设备。当最后一次使用设备终结后，调用close子程序。独占设备必须标记设备可再次使用。

(3) read入口点：一般用于字符设备，从设备上读数据。对于有缓冲区的I/O操作，一般是从缓冲区里读数据。

(4) write入口点：一般用于字符设备，往设备上写数据。对于有缓冲区的I/O操作，一般是把数据写入缓冲区里。

(5) ioctl入口点：执行读、写之外的一些特别定义的操作，如查询设备参数等等。

具体到Linux系统里，设备驱动程序所提供的这组入口点由一个结构来向系统进行说明，此结构定义为：

struct file\_operations {

int (\*lseek)(struct inode \*inode,struct file \*filp, off\_t off,int pos);

int (\*read)(struct inode \*inode,struct file \*filp, char \*buf, int count);

int (\*write)(struct inode \*inode,struct file \*filp, char \*buf,int count);

int (\*readdir)(struct inode \*inode,struct file \*filp, struct dirent \*dirent,int count);

int (\*select)(struct inode \*inode,struct file \*filp, int sel\_type,select\_table \*wait);

int (\*ioctl) (struct inode \*inode,struct file \*filp, unsigned int cmd,unsigned int arg);

int (\*mmap) (void);

int (\*open) (struct inode \*inode, struct file \*filp);

void (\*release) (struct inode \*inode, struct file \*filp);

int (\*fsync) (struct inode \*inode, struct file \*filp);

};

在这个结构里，指出了设备驱动程序所提供的入口点位置，分别是：

(1) lseek：移动文件指针的位置，显然只能用于可以随机存取的设备。

(2) read：进行读操作，参数buf为存放读取结果的缓冲区，count为所要读取的数据长度。返回值为负表示读取操作发生错误，否则返回实际读取的字节数。

(3) write：进行写操作，与read类似。

(4) readdir：取得下一个目录入口点，只有与文件系统相关的设备驱动程序才使用。

(5) select：进行选择操作，如果驱动程序没有提供select入口，select操作将会认为设备已经准备好进行任何的I/O操作。

(6) ioctl：进行读、写以外的其它特殊操作，参数cmd为自定义的的命令。

(7) mmap：用于把设备的内容映射到地址空间，一般只有块设备驱动程序使用。

(8) open：打开设备准备进行I/O操作。返回0表示打开成功，返回负数表示失败。如果驱动程序没有提供open入口，则只要/dev/driver文件存在就认为打开成功。

(9) release：即close操作。

上述入口点在设备驱动程序初始化的时候向系统进行登记，以便系统在适当的时候调用。比如在Linux系统里，通过调用register\_chrdev向系统注册字符型设备驱动程序。register\_chrdev定义为：

int register\_chrdev(unsigned int major, const char \*name, struct file\_operations \*fops);

其中，major是为设备驱动程序向系统申请的主设备号，如果为0则系统为此驱动程序动态地分配一个主设备号。name是设备名。fops就是前面所说的对各个调用的入口点的说明。此函数返回0表示成功。返回-EINVAL表示申请的主设备号非法，一般来说是主设备号大于系统所允许的最大设备号。返回-EBUSY表示所申请的主设备号正在被其它设备驱动程序使用。如果是动态分配主设备号成功，此函数将返回所分配的主设备号。如果register\_chrdev操作成功，设备名就会出现在/proc/devices文件里。

## 9.2 编写Linux设备驱动程序的基本方法

设备驱动程序一般都要申请和使用系统资源，包括内存、时钟、I/O端口等，在这些资源不用的时候，应该释放它们，以利于资源的共享。下面首先简单叙述这些系统资源的使用方法：

(1) 内存

作为系统核心的一部分，设备驱动程序在申请和释放内存时不能调用标准C库的malloc和free，而代之以调用kmalloc和kfree（这两个系统调用的用法可以参考附录五），它们被定义为：

void \* kmalloc(unsigned int len, int priority);

void kfree(void \* obj);

参数len为希望申请的字节数，obj为要释放的内存指针。priority为分配内存操作的优先级，一般用GFP\_KERNEL。

(2) IO端口

与内存不同，使用一个没有申请的I/O端口不会使CPU产生异常，也就不会导致诸如“segmentation fault"一类的错误发生。任何进程都可以访问任何一个I/O端口。此时系统无法保证对I/O端口的操作不会发生冲突，甚至会因此而使系统崩溃。因此，在使用I/O端口前，也应该检查此I/O端口是否已有别的程序在使用，若没有，再把此端口标记为正在使用，在使用完以后释放它。

这样需要用到如下几个函数：

int check\_region(unsigned int from, unsigned int extent);

void request\_region(unsigned int from, unsigned int extent, const char \*name);

void release\_region(unsigned int from, unsigned int extent);

调用这些函数时的参数为：from表示所申请的I/O端口的起始地址；extent为所要申请的从from开始的端口数；name为设备名，将会出现在/proc/ioports文件里。check\_region返回0表示I/O端口空闲，否则表示正在被使用。

在申请了I/O端口之后，就可以采用如下几个函数来访问I/O端口：

inline unsigned int inb(unsigned short port);

inline unsigned int inb\_p(unsigned short port);

inline void outb(char value, unsigned short port);

inline void outb\_p(char value, unsigned short port);

这几个函数在x86体系结构的机器上，被编译成为对应的IN/OUT指令。略有不同的是，其中inb\_p和outb\_p插入了一定的延时以适应某些慢的I/O端口。

(3) 时钟

在设备驱动程序里，可能需要用到计时机制。在Linux系统中，时钟是由系统接管，设备驱动程序可以向系统申请时钟或者定时器。与时钟或定时器有关的系统调用有：

void add\_timer(struct timer\_list \* timer);

int del\_timer(struct timer\_list \* timer);

inline void init\_timer(struct timer\_list \* timer);

其中struct timer\_list的定义为：

struct timer\_list

{

struct timer\_list \*next;

struct timer\_list \*prev;

unsigned long expires;

unsigned long data;

void (\*function)(unsigned long d);

};

expires的含义是，要求系统在这个时间立即执行我们自己定义的function。系统核心有一个全局变量JIFFIES表示当前时间，一般在调用add\_timer时jiffies=JIFFIES+num,表示在num个系统最小时间间隔后执行function。系统最小时间间隔与所用的硬件平台有关，在核心里定义了常数HZ表示一秒内最小时间间隔的数目，则num\*HZ表示num秒。系统计时到预定时间就调用function，并把此子程序从定时队列里删除，因此如果想要每隔一定时间间隔执行一次的话，就必须在function里再一次调用add\_timer。function的参数d即为timer里面的data项。

（4）其他

在设备驱动程序里，还可能会用到如下的一些系统函数：

#define cli() \_\_asm\_\_ \_\_volatile\_\_ ("cli"::)

#define sti() \_\_asm\_\_ \_\_volatile\_\_ ("sti"::)

这两个函数在x86的机器上被编译成为CLI/STI指令，负责打开和关闭中断。

void memcpy\_fromfs(void \* to,const void \* from,unsigned long n);

void memcpy\_tofs(void \* to,const void \* from,unsigned long n);

在用户程序调用read 、write时，因为进程的运行状态由用户态变为核心态，地址空间也变为核心地址空间。而read、write中参数buf是指向用户程序的私有地址空间的，所以不能直接访问，必须通过上述两个系统函数来访问用户程序的私有地址空间。memcpy\_fromfs由用户程序地址空间往核心地址空间复制，memcpy\_tofs则反之。参数to为复制的目的指针，from为源指针，n为要复制的字节数。

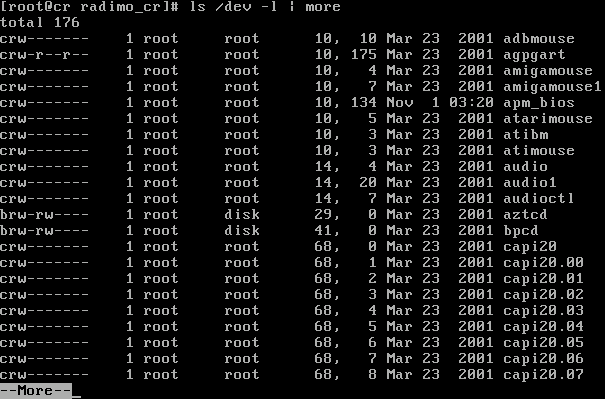
最后，在设备驱动程序里，经常需要调用printk来打印一些调试信息，这个函数的用法与标准C库中的printf类似。不过printk打印的信息通常记录在log文件里，在命令行下面输入dmesg命令可以查看log文件。

在Linux里，一般不直接修改系统内核的源代码，而是把设备驱动程序作为可加载的模块，由系统管理员去加载它，使之成为内核的一部分。也可以由系统管理员把已加载的模块动态的卸载下来。Linux中，模块可以用C语言编写，用gcc编译成目标文件（不进行链接，作为\*.o文件存在），为此需要在gcc命令行里加上-c的参数。在编译时，还应该在gcc的命令行里加上这样的参数：-D\_\_KERNEL\_\_ -DMODULE。由于在不链接时，gcc只允许一个输入文件，因此一个模块的所有部分都必须在一个文件里实现。编译成功的模块，可以用insmod命令加载，用rmmod命令来卸载，并可以用lsmod命令来查看所有已加载的模块及其状态。

编写模块程序的时候，必须提供两个函数，一个是int init\_module(void)，供insmod在加载此模块的时候自动调用，负责进行设备驱动程序的初始化工作。另一个函数是void cleanup\_module (void)，在模块被卸载时调用，负责进行设备驱动程序的清除工作。关于编写模块的知识，可以参考第三章的内容。

用户进程通过位于/dev目录下的设备文件来访问设备。请在你的Linux系统下打开一个终端并输入cd /dev来到/dev目录，然后输入ls –l命令，查看当前计算机上面的所有设备。

下图给出了笔者使用的系统上的一些设备：



通过ls -l输出的第一列中的“c”可以看出，这个设备文件对应的是字符设备。同样还有块设备，它们的第一列是“b”；在设备文件条目的最新修改日期前你会看到两个数（用逗号分隔）。这些数就是相应设备的主设备号和次设备号。如图中设备的主设备号分别是10，14和29等等，而次设备号则分别是10，175，4，7等等。

在成功的向系统注册了设备驱动程序后（调用register\_chrdev成功后），就可以用mknod命令来建立一个设备文件，应用程序需要使用这个设备的时候，只要对此文件进行操作就行了。比如我们输入mknod /dev/radimo b 42 0这个命令，可以在/dev目录下建立一个叫radimo的块设备文件，其主设备号为42，次设备号为0。

## 9.4 RADIMO：一个块设备驱动程序的例子

前面讲过，设备驱动程序所管理的设备包括字符设备和块设备。这一节我们将详细讲述块设备驱动程序的编写方法，字符设备的驱动程序与之相比大同小异，本章不再赘述。

所谓面向块的设备是指数据传输是以块为单位的，典型的块设备包括软盘、硬盘、U盘等，这里硬件的块一般被称作“扇区（Sector）”。而名词“块”常用来指软件上的概念：驱动程序常常使用1KB大小的块，而扇区大小一般为512字节。在这一节，我们将构造一个叫radimo的简单的块设备驱动程序，这个程序的是使用计算机的内存作为硬件设备。换句话说，它是一个RAM-disk的驱动程序，可以从内存中划分出2MB的空间，虚拟出一个磁盘（确切的说，无论是RAM-disk，还是光盘，U盘，都不是使用磁介质进行存储的，因此不能将其称为“磁盘”，但是为了便于理解，我们不妨用“磁盘”作为这些存储设备的统称），可以对这个虚拟磁盘进行格式化，复制文件，查看文件等操作。

Radimo程序的全部源代码可以在<http://www.vrbrothers.com/cr/radimo.zip>下载，并且为了方便阅读，笔者在其中增加了大量的中文注释。为了对设备驱动程序有一个定性的认识，便于下面的讲解，我们先来下载这个程序，并且编译、加载、测试。然后带着对这个程序的基本认识，去学习它的工作原理。

首先我们下载到这个zip包，解开后有三个文件radimo.c，radimo.h和makefile，将这三个文件复制到Linux的某个目录下，并且在终端窗口用cd命令来到该目录下，按照下面的步骤依次执行。在每个步骤执行完毕之后都可以输入dmesg命令，查看系统的log文件。

输入make<回车>，意思是调用gcc编译这个驱动程序。

编译结束后，用ls命令可以看到当前路径下多了一个叫radimo.o的文件，这是编译后得到的设备驱动程序。

输入mknod /dev/radimo b 42 0<回车>，意思是建立一个叫radimo的块设备文件，主设备号为42，次设备号为0。主设备号42一般在Linux系统中是空闲的，可以供我们使用。现在我们有一个叫radimo的设备，可以把它看作一个空白的磁盘。这个设备目前还没有驱动程序。

输入insmod radimo.o<回车>，意思是让操作系统加载这个设备驱动程序。

输入lsmod<回车>，可以看到目前操作系统中已经加载的模块，其中应该包括我们的radimo。

输入mke2fs /dev/radimo<回车>，意思是在radimo这个空白磁盘上建立一个EXT2文件系统。相当于Windows下的格式化磁盘――不用担心，这里格式化的是一个虚拟盘，不会对我们的硬盘造成任何影响。

输入mount /dev/radimo /mnt<回车>，意思是把这个磁盘挂接在文件系统的/mnt目录下。

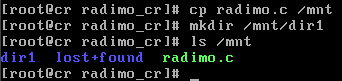
如果以上步骤都顺利执行完毕，那么恭喜你，现在你的/mnt目录就是一个容量2MB的虚拟磁盘，可以象操作其他盘一样建立子目录，复制文件等等。只是这个盘是虚拟的，在我们的驱动程序卸载后，里面的内容就会被清除。

输入df<回车>，意思是查看系统中目前挂接的磁盘（更确切的说法，应该是目前挂接的文件系统，但是“磁盘”的说法比较容易理解，下同）。这时应该可以看到至少两个磁盘，一个就是我们的硬盘，另一个就是刚才挂接的radimo虚拟盘。另外，如果你的系统中还挂接了软盘，光盘，U盘等等，也应该在这里有所体现。

输入cp radimo.c /mnt<回车>，意思是把radimo.c这个文件复制到虚拟盘上面，当然这里也可以是其他大小不超过2MB的文件。

输入mkdir /mnt/dir1<回车>，意思是在虚拟盘上面建立一个叫dir1的子目录，当然这个子目录的名字可以改成其他你喜欢的名字。

输入ls /mnt<回车>，可以看到我们刚才复制的文件，以及建立的子目录。如下图所示：



输入umount /dev/radimo<回车>，意思是从系统中卸载这个磁盘。这时再用ls命令，已经不能看到上面复制的文件和建立的子目录。

输入rmmod radimo<回车>，意思是卸载radimo这个驱动程序。卸载后再用lsmod命令，加载模块列表里面已经没有radimo这一项，系统恢复到了最初的状态。我们的实验到此告一段落。

通过上面这个有趣的实验，我们看到了radimo这个设备驱动程序的作用。但实际上radimo这个程序非常简单，它的代码不过区区200多行，其中最关键部分的代码不到100行。下面，我们用打开radimo.c文件，一起来分析这个驱动程序，从而揭开radimo的神秘面纱，了解块设备驱动程序编写的一般过程。

由于块设备驱动程序的绝大部分是设备无关的，核心的开发者通过把大部分相同的代码放在一个头文件<linux/blk.h>中，来简化驱动程序的代码。因此，每个块设备驱动程序都必须包含这个头文件。实际上，blk.h相当不寻常，比如它定义了几个基于符号MAJOR\_NR的变量，因此MAJOR\_NR必须由驱动程序在它包含这个头文件之前声明。有兴趣的读者可以看看Linux内核源代码中的blk.h，你会发现有好几个设备相关的变量是按照MAJOR\_NR的值声明的，也就是说MAJOR\_NR应该提前知道，驱动程序必须在包含blk.h之前定义MAJOR\_NR和其它几个符号。

下面描述了一些必须提前定义的，在<linux/blk.h>中使用的符号：

MAJOR\_NR：这个符号用来访问一些数组，特别是blk\_dev和blksize\_size。

DEVICE\_NAME：这个符号定义了被生成的设备名。

DEVICE\_NR(kdev\_t device)：这个符号用来从kdev\_t设备号中抽取物理设备的序号。这个宏的值可以是MINOR(device)或别的表达式。这要依据给设备或分区分配次设备号的常规方式而定。对同一个物理设备上的所有分区，这个宏应返回同一个设备号——也就是说，DEVICE\_NR表达的是磁盘号，而不是分区号。

DEVICE\_NO\_RANDOM：如果一个设备不能对随机设备贡献显著的熵值，则应该定义这个符号。如果你不明白什么是随机设备的熵值，定义上这个符号总是没错的。

DEVICE\_OFF(kdev\_t device)：系统将在结束一个读写请求的时候调用这个宏，如果你的驱动程序需要关闭驱动器的马达（如软盘的驱动程序），就把它设置成为你的关闭马达的代码。而我们的radimo或者是以后要写的U盘驱动程序都不需要操作马达，所以设置为空。

DEVICE\_ON(kdev\_t device)：作用和DEVICE\_OFF类似，设置为空就可以了。

前面已经学习过，设备驱动程序首先需要向系统注册，使系统能够正确识别它。显然，当本模块被加载的时候去注册，应该是最合适的。因此，我们在驱动程序的模块初始化函数（即init\_module函数）当中执行注册设备以及初始化其他工作变量的任务。在前面的实验中，当输入insmod命令的时候，这个函数将由操作系统进行调用。

对一个块设备驱动程序进行注册和取消注册的函数分别是：

int register\_blkdev(unsigned int major, const char\*name, struct file\_operations \*fops)； int unregister\_blkdev(unsigned int major, const char\*name)；

其参数的含义与前面讲过的字符设备驱动程序注册函数一样。在我们的radimo程序中，是这样进行注册的：

/\* 【重要】向系统注册块设备 \*/

res = register\_blkdev(RADIMO\_MAJOR, "radimo", &radimo\_fops);

if (res) {

MSG(RADIMO\_ERROR, "couldn't register block device\n");

return res;

}

其中RADIMO\_MAJOR是预先定义的常数42，因为在一般的Linux系统中42这个主设备号是空闲的，可以供我们利用。register\_blkdev的fops参数与前面讲过的略有不同：

static struct block\_device\_operations radimo\_fops = {

open: radimo\_open,

release: radimo\_release,

ioctl: radimo\_ioctl,

};

其中只有open、release和ioctl三个函数，而read，write以及fsync等操作并不需要我们来设计，采用系统中的通用函数就可以了。因为通用的读写操作使用数据缓冲，从而获得比较高的效率。块设备驱动程序可以被缓冲是因为它们的数据服从于计算机的文件层次结构，任何应用程序都无法直接访问。不过，当缓冲的高速缓存不能满足一个读请求或当一个待处理的写操作要刷新到物理磁盘上时，还是需要调用驱动程序来进行真正的数据传送。因此，我们必须要写一个额外的函数radimo\_request来处理实际数据传送的请求，并把这个函数也在系统中注册。

这个函数的原型是：

static int radimo\_request(request\_queue\_t \*request\_queue, int rw, struct buffer\_head \*sbh)；

其中的参数在后面详细讲述。为了把这个函数在系统中注册，我们使用下面这行语句就可以了：

blk\_queue\_make\_request(BLK\_DEFAULT\_QUEUE(RADIMO\_MAJOR), &radimo\_request);

除此之外，还必须将我们的设备驱动程序的一些重要参数告诉操作系统，下面几行代码就是在做这样的工作，其中hardsect\_size，blksize\_size和blk\_size是操作系统维护的三个全局数组，里面保存了所有块设备的扇区大小，软件块大小和总容量大小。

/\* 在系统中注册块的大小、存储容量等参数 \*/

hardsect\_size[RADIMO\_MAJOR] = &radimo\_hard;

blksize\_size[RADIMO\_MAJOR] = &radimo\_soft;

blk\_size[RADIMO\_MAJOR] = &radimo\_size;

好了，到目前为止，我们已经成功的将一个块设备在系统中进行了注册。整个注册的过程以及注册后的系统状况可以由下图表示：（注意图中的blk\_init\_queue函数，和我们调用的blk\_queue\_make\_request函数功能比较类似）



除此之外，如果有其他需要初始化的变量，也应该在模块的初始化函数中一并完成。在radimo程序中，需要申请2MB的存储空间，作为虚拟磁盘的存储区域，下面的代码就是完成这项工作的。vmalloc的用法和前面讲的kmalloc类似，是在系统中申请一块内存，具体的区别，请参考有关资料。

radimo\_storage = (char \*) vmalloc(1024\*radimo\_size);

另外，radimo中还初始化了一个定时器，虽然这个定时器其实没有起任何作用，但是作为例子程序，我们保留了这样一个框架，以后如果要使用定时器的时候，照此办理就可以了。

当模块被卸载的时候，也就是在前面的实验中输入rmmod命令的时候，系统将调用我们设计的cleanup\_module函数来做清理工作，这个函数的实现基本上是init\_module函数的逆过程，本章不再赘述。

在初始化的过程中，我们将radimo\_open，radimo\_release和radimo\_ioctl这三个函数在系统中进行了注册。下面分别阐述这三个函数的调用时机和所做的工作。

radimo\_open是在打开设备时使用的瀂当一个用户态的程序需要使用radimo设备时，首先必须进行open系统调用，open系统调用再来调用我们的radimo\_open函数。具体的说，在前面的实验中，当我们用mke2fs命令建立文件系统（相当于格式化磁盘）的时候，或者用mount命令挂接文件系统的时候，都会调用radimo\_open。我们的radimo\_open函数实现比较简单，唯一值得一提的就是调用了MOD\_INC\_USE\_COUNT这个宏，它的作用是把当前模块的引用计数加一。引用计数是什么呢？是在系统内部维护的、和当前模块相对应的一个整数。在用insmod命令加载模块的时候，这个数被置为零。每次打开设备进行操作的时候，引用计数增加一；每次关闭设备的时候，引用计数减少一。同时，Linux系统规定，只有当引用计数等于零的时候，才允许使用rmmod命令卸载模块。这样，如果一个radimo设备还在使用，就不能卸载驱动程序，避免了在设备运转中“驱动程序突然丢失”的意外故障。

radimo\_release是在关闭设备时使用的。当用户态的程序使用设备完毕之后，需要进行close系统调用，由后者再来调用我们的radimo\_release函数。比如前面的实验中，当mke2fs命令格式化磁盘完毕的时候，或者当我们使用umount命令卸载文件系统的时候，都会调用radimo\_release。你一定猜到了，在radimo\_release中必须进行的一项工作是调用MOD\_DEC\_USE\_COUNT减少本模块的引用计数，事实上这也是我们的radimo\_release函数执行的唯一的任务。

radimo\_ioctl是在用户态的程序对本设备进行ioctl系统调用的时候被执行的，ioctl的含义是IO ConTroL，也就是IO设备的控制。用户程序通常和设备驱动程序约定一些控制命令，然后通过调用ioctl向设备驱动程序发命令。比如我们约定命令BLKGETSIZE的含义是获得设备容量大小，那么当用户程序调用ioctl并把BLKGETSIZE当作参数传递的时候，设备驱动程序就马上知道，用户想获得设备容量，于是可以将容量立即告诉用户。

在用户空间内调用ioctl函数的原型大致如下：

int ioctl(int fd, int cmd, …)；

ioctl的第三个参数事实上只是一个可选参数，这里用省略号只是为了在编译时防止编译器进行类型检查。第三个参数的具体情况与要完成的控制命令（也就是第二个参数）有关。某些命令不需要参数，某些需要一个整数做参数，而某些则需要一个指针做参数。使用指针通常是可以用来向ioctl传递任意数目数据；设备可以从用户空间接收任意大小的数据。

下面我们分别叙述块设备驱动程序经常要处理的命令，它们在<linux/fs.h>中被声明。

BLKGETSIZE：获取当前设备的大小，以扇区数表示。由系统调用传递的数值arg是一个指向long数值的指针，用来将大小拷贝到一个用户空间的变量中。这个ioctl命令可以被mke2fs命令用来获知产生的文件系统的大小。

BLKFLSBUF：字面上的意思是“刷新缓冲区”。当我们写磁盘等设备的时候，实际上要写的数据被保存在磁盘缓冲区里面，用这个命令通知设备驱动程序，现在应该把缓冲区里面的数据实际的写入设备当中了。

BLKRAGET：取得设备当前预读值。当前数值应该用在参数arg中传递给ioctl的指针，写进一个long类型的用户空间变量。如果你不明白什么是预读值，可以在你的驱动程序中暂时照搬radimo中的代码。

BLKRASET：设置预读值。用户进程在arg中传递这个新值。

BLKROSET，BLKROGET：这两个命令用来改变和检查设备的只读标志。

BLKSSZGET：得到设备驱动程序中一个块的大小。

在一个设备驱动程序中，不一定非要实现所有这些命令，也不一定仅仅实现上述这些命令。正如前面讲到的，ioctl的命令是用户程序和设备驱动程序之间的约定。上面几个命令只是在大多数用户程序（如mke2fs命令就是一个典型的用户程序）中使用的比较多而已。

前面我们提到过，当用户发出读写块设备的命令时，实际上操作系统并没有马上调用驱动程序进行IO操作。因为对于块设备来说，操作系统提供了缓存（Cache）的功能，它首先从缓存中寻找有没有用户要读的内容，或者是先把用户要写的内容放在缓冲里面。只有当缓存中没有用户要读的内容，或者必须将数据实际写入设备的时候，才会向设备驱动程序发出IO请求。因此，在设备驱动程序中，必须响应操作系统发出的请求。在radimo驱动程序中，我们编写了radimo\_request这个函数来响应IO请求，并且已经在驱动程序初始化的时候把这个函数注册到了系统中。

需要特别注意的是，在笔者撰写本章内容时，流行的Linux内核版本是2.4.20。而目前市面上大部分参考书，特别是译著，由于上市时间周期的关系，仍然采用2.2甚至更早的2.0内核版本作为例子。在内核2.4以上，响应IO请求的函数原型和编写方法都发生了很大变化，因此，如果发现旧的参考书和本章内容不一致，请以我们讲述的内容为准。同时，笔者无法预知在今后的Linux内核版本中，这个函数还会不会继续变化。要得到最新的消息，请自行参考内核源代码。

radimo\_request的原型如下所示：（适用于内核版本2.4）

static int radimo\_request(request\_queue\_t \*request\_queue, int rw, struct buffer\_head \*sbh)；

这个函数共有三个参数和一个返回值，我们来详细介绍：

第一个参数描述了一个IO请求队列，对我们的程序来说并没有太大使用价值，因此我们仅仅保留它的定义，不再过多考虑这个参数。有兴趣的读者请自行参考内核源码中blkdev.h对这个结构的定义。

第二个参数是一个整数，这个整数只可能取三个值：READ、READA或者WRITE（这三个值都是在内核中预定义的常整数），分别表示要进行的是读操作、预读操作（简单起见，可以和读操作等同对待）还是写操作。

第三个参数是一个结构指针，描述了IO操作可能用到的参数。这个结构的定义在内核源码的fs.h中。它包含了大量的参数，而我们比较关心的只是其中的一小部分。具体来说，在radimo中所用到的，只有b\_size，b\_rsector这两个成员变量；宏bh\_kmap(sbh)以及成员函数b\_end\_io。变量b\_size和b\_rsector分别描述了需要读写的字节数，需要读写的扇区编号；宏bh\_kmap(sbh)描述了一个缓冲区，应该从这个缓冲区里读，或者往这个缓冲区里写；成员函数b\_end\_io则必须在读写成功后被驱动程序调用，通知操作系统做一些善后工作。

现在，我们的radimo\_request接收了这三个参数，之后应该做什么呢？显然，应该计算出具体要读写的设备地址（如硬盘的磁道、扇区号等等），然后进行设备IO，把从设备中取出的数据写入缓冲区（读操作），或者相反，把缓冲区的数据送入设备（写操作）。

对于radimo驱动程序来说，我们是用一块连续的内存来保存实际数据的。那么，要读写的设备地址仅仅是针对这块内存首地址的一个偏移，换言之，就是要知道应该在这块内存的什么位置开始读写。计算的方法也相当简单，如下所示：

offset = sbh->b\_rsector \* radimo\_hard;

total = (unsigned long)sbh->b\_size; //计算需要访问的地址和大小

这两行程序的意思是，从成员变量b\_rsector中取得要读写的扇区号，乘以每个扇区的字节数，得到读写的开始地址，并保存在offset变量中；从成员变量b\_size取出要读写的字节数，保存在total变量中。

下面要做的工作是检查读写是否越界，比如我们的radimo设备只有2MB的容量，那么读写2MB以外的地址显然是非法的。检查读写是否越界的代码非常简单，并且越界的情况几乎不可能发生，但是请注意这段代码决不能省略。事实上，处理这种异常情况的代码是评价一个程序是否健壮的重要标准，在今后的学习中也应该随时加以注意。

现在一切准备就绪，我们要开始进行设备IO了。由于radimo驱动程序使用一块内存作为存储设备，因此它的IO也非常简单，只需要把这块内存中的一部分数据复制到缓冲区里面就可以了，如下所示：

if (rw == READ || rw==READA) { // 如果是读或者预读操作，从虚拟盘的内存中复制数据到缓冲区中

memcpy(bh\_kmap(sbh), radimo\_storage+offset, total);

} else if (rw == WRITE) { // 如果是写操作，从缓冲区中复制数据到虚拟盘的内存中

memcpy(radimo\_storage+offset, bh\_kmap(sbh), total);

} else { /\* can't happen \*/

MSG(RADIMO\_ERROR, "cmd == %d is invalid\n", rw);

}

对于memcpy函数，用C语言写过程序的读者一定不会陌生，它的作用是复制一定数量的数据，不熟悉C语言的读者请自行参考C标准函数库手册。由于我们是在驱动程序和内核之间复制数据，双方都运行在核态下，因此用memcpy就能够满足要求。否则，如果驱动程序需要和用户内存空间交换数据，就必须使用前面讲过的memcpy\_fromfs等函数。

IO操作结束之后，必须调用sbh->b\_end\_io成员函数，通知操作系统读写已经结束。调用这个成员函数所做的事情，请有兴趣的读者自行参考Linux内核源代码。

到此为止，我们已经用一个块设备驱动程序radimo作为例子，讲解了块设备驱动程序设计的框架和一些技巧。由于篇幅所限，本章所讲的知识非常浅显，基本上仅限于实验所需的内容。如果读者有兴趣了解更多，请查阅本章的参考资料。

## 9.5实验内容

仔细阅读上一节中radimo的源代码，将radimo驱动程序改写成为一个U盘驱动程序。在检查时，你需要当场独立完成以下操作：

(1) 用lsmod命令查询系统中是否已经加载usb-storage的模块，这是Linux系统自带的U盘驱动程序。如果已经加载，则用rmmod命令卸载这个模块。

(2) 编译改装后的radimo程序，要求能编译通过，允许出现编译警告。

(3) 用insmod命令加载改装后的radimo程序。

(4) 插入U盘。要求在用dmesg查看log文件时，能够看到改装的radimo程序发现了U盘。允许你的程序只认识一种或者有限的几种U盘，不要求能够认出市面上所有U盘。

(5) 能够在log文件中显示出U盘的容量。

(6) 用mknod命令建立设备文件，并用mount命令挂接这个U盘的文件系统到/mnt。你的U盘可以预先用Windows格式化好。

(7) 用df命令应该能够看到已经挂接好的U盘。

(8) 用ls /mnt命令应该能够看到U盘中的内容。用cp命令能够将U盘中的任意文件复制到硬盘上。

(9) 用umount命令和rmmod命令清理现场。

(10) 在以上各步骤中，随时回答老师提出的问题。不要求回答完全准确，但基本原理必须能够阐述清楚，可以当场查资料。

如果你还能够完成下面的操作，则会有更多的奖励分数：

1. 能够读出U盘主引导扇区的内容，显示在log文件里，并简要解释其含义。
2. 能够在U盘被中途拔出的情况下正确识别，并进行相应的错误处理，不至于死机。
3. 能够把硬盘上的文件复制到U盘上，并在Windows系统下面插入U盘，显示出这个文件，证明复制成功。

## 9.6 解决问题

### 9.6.1 USB设备简述

USB 是目前在打印机，数字存储设备，输入/输出设备，数码像机，MP3播放器等其他周边设备中得到广泛应用的连接方式。USB设备具有使用方便，速度快，连接灵活，即插即用，总线供电等优点。基于USB接口的大容量存储设备（USB Mass Storage）主要有：USB移动硬盘，USB 外置光驱，USB 外置软驱，USB闪存盘（U盘）等。由于USB大容量存储设备的方便与快捷，它们很快得到用户的认可。

虽然USB大容量存储设备中有着不同的解决方法，但通常使用一个带有USB接口引擎的MPU（又可称为USB微控制器）用于处理主机发送的命令以及对存储设备进行操作。我们比较常见的大容量存储设备是基于闪存设备（Flash Device）的，如本实验中需要使用的U盘就是一种典型的闪存设备。

USB组织定义了大容量存储设备的类规范，这个类规范包括四个独立的子类规范，即： 1. USB Mass Storage Class Control/Bulk/Interrupt (CBI) Transport

2. USB Mass Storage Class Bulk-Only Transport

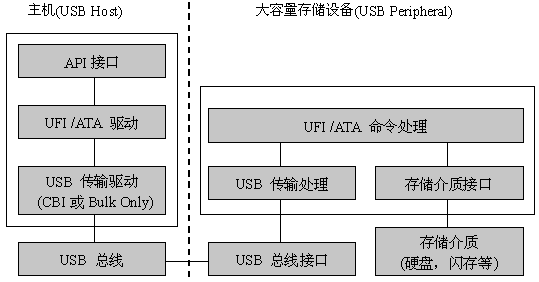
3. USB Mass Storage Class ATA Command Block

4. USB Mass Storage Class UFI Command Specification。

前两个子规范定义了数据/命令/状态在USB上的传输方法。Bulk-Only传输规范仅仅使用Bulk端点传送数据/命令/状态，CBI传输规范则使用Control/Bulk/Interrupt三种类型的端点进行数据/命令/状态传送。后两个子规范则定义了存储介质的操作命令。UFI命令规范是针对USB移动存储而制定的，实际上UFI命令格式是基于SFF-8070i和SCSI-2规范，总共定义了19个12字节长度的操作命令。具体命令格式以及含义，请查阅SCSI规范。

下图是USB大容量存储设备软件结构示意图。虚线左边部分是主机的驱动程序结构。最顶层是应用程序接口（API），用于提供给用户访问存储设备的方法；中间的UFI/ATA驱动层将应用程序的访问转换成UFI或ATA命令/数据格式，与外部存储设备之间按照子规范3或4的定义进行命令/状态/数据的交换；最底层则是USB传输驱动，负责将上层的UFI/ATA数据发送到USB总线上以及接收从存储设备返回的状态/数据。虚线右边部分是大容量存储设备的固件（Firmware）功能结构。在USB总线接口上面是USB传输处理层，它与主机之间按照子规范1或2的定义进行通讯，将主机的命令/数据传递到UFI/ATA 命令处理层并将其状态/数据返回到主机，它还需要检查并处理数据传输过程中的错误。UFI /ATA 命令处理层负责对主机的UFI /ATA 命令进行处理，并将结果返回给主机。存储介质接口提供与不同存储介质连接的方法，负责将接收到的UFI /ATA 命令/数据转换成具体的物理信号发送到存储介质，并从存储介质获取状态/数据。

在具体实现上，存储设备可以选择支持两种传输规范（CBI或Bulk-Only）或者只支持其中的一种。实际上，Bulk-Only传输规范是一种更常用的方法。市面上大多数USB存储设备都是基于Bulk-Only传输规范和UFI命令规范设计的。



### 9.6.2 Linux下的USB设备驱动程序简介

在Linux kernel源码目录中driver/usb/usb-skeleton.c为我们提供了一个最基础的USB驱动程序。我们称为USB骨架。通过它我们仅需要修改一些部分，就可以完成一个USB设备的驱动程序。

Linux USB 驱动程序需要做的第一件事情就是在Linux USB 子系统里注册，并提供一些相关信息，例如这个驱动程序支持哪种设备，当被支持的设备从系统插入或拔出时，会有哪些动作，所有这些信息都传送到USB 子系统中。注册通常在驱动程序的初始化函数里完成，需要调用usb\_register函数。当要从系统卸载驱动程序时，需要注销USB子系统，即需要调用usb\_unregister 函数。

当USB设备插入时，为了使Linux中PCI、USB等设备热插拔系统自动装载驱动程序，需要创建一个MODULE\_DEVICE\_TABLE。USB\_DEVICE宏利用厂商ID和产品ID为我们提供了一个设备的唯一标识。当系统插入一个ID匹配的USB设备到USB总线时，驱动程序中probe 函数也就会被调用。usb\_device 结构指针、接口号和接口ID都会被传递到函数中。驱动程序需要确认插入的设备是否可以被接受，如果不接受，或者在初始化的过程中发生任何错误，probe函数返回一个NULL值。否则返回一个含有设备驱动程序状态的指针。

如果设备从usb总线拔掉，设备指针会调用disconnect 函数。驱动程序就需要清除那些被分配了的所有私有数据、关闭USB设备，并且注销自己。值得注意的是，USB设备可以在任何时间从系统中取走，即使程序目前正在访问它。一个优秀的USB驱动程序要能够很好地处理解决此问题，切断任何当前的读写，同时通知用户空间程序USB设备已经被取走。

当需要对USB设备进行读写的时候，Linux系统提供的usb\_bulk\_msg函数是非常有用的。通常我们调用usb\_bulk\_msg函数并传递一个缓冲区地址，用来缓冲和放置驱动收到的数据，若没有收到数据，就失败并返回一个错误信息。

对于一款不明型号的USB设备，我们可以通过cat /proc/bus/usb/devices命令得到当前系统探测到的USB总线上的设备信息。它包括Vendor、ProdID、Product等。下面是一款杂牌U盘插入后的信息片断：

T: Bus=01 Lev=01 Prnt=01 Port=01 Cnt=02 Dev#= 5 Spd=12 MxCh= 0

D: Ver= 1.10 Cls=00(>ifc ) Sub=00 Prot=00 MxPS=8 #Cfgs= 1

P: Vendor=07c4 ProdID=a400 Rev= 1.13

S: Manufacturer=USB

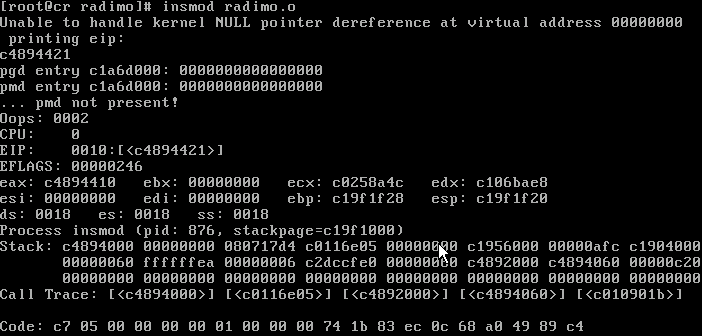
S: Product=Mass Storage

我们从中可以看到这个U盘的Vendor和ProdID，这两项信息在编写驱动程序的时候是非常有用的。

### 9.6.3 设备驱动程序的调试方法

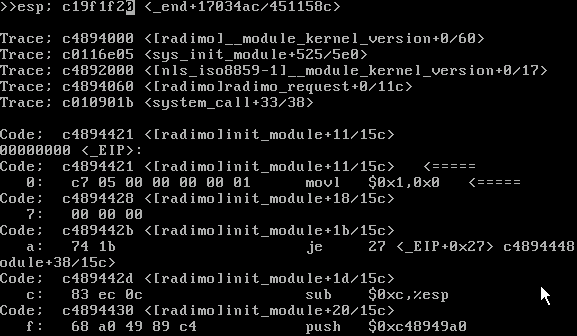
设备驱动程序和普通应用程序一样，在编写的过程中也会出现漏洞和程序错误，也就是我们常说的“BUG”。但是设备驱动程序的调试工作和普通的程序调试相比，难度要大得多。因为设备驱动程序在系统核心态下面运行，我们基本上不可能象对待普通程序那样，对内核代码采用单步执行的方式进行调试。调试驱动程序的最佳手段永远是printk，也就是在怀疑有问题的区域有策略的增加一些printk语句，然后执行到那个地方，通过printk打印在log文件里的信息，推测可能出现的故障。

另外，Oops追查方法也是我们必须掌握的一项基本调试技能。因为设备驱动程序里面的很多BUG都是以内存访问越界或者访问NULL指针的形式出现的，和普通的应用程序里面的内存段冲突错误非常相似。在这种情况下，系统会同时在屏幕上和log文件里记录如下所示的信息，称为Oops信息：



Oops显示了故障时的处理器状态，模块和CPU寄存器内容，页描述符表的位置，以及其他似乎不能理解的信息。这些消息的最大问题就是它们是十六进制数值，对于程序员来说没什么意义，需要将它们解析为相应的符号。

可以使用ksymoops工具帮助开发人员将Oops消息中的数值地址解析为内核符号，这个工具可以在<ftp://ftp.cn.kernel.org/pub/linux/utils/kernel/ksymoops>下载。ksymoops从标准输入获得Oops消息，并从命令行得到内核符号表的文件名。符号表文件通常就是/usr/src/linux/System.map或者/boot/System.map。ksymoops程序以更可读的方式打印调用轨迹和程序代码，而不是最原始的Oops消息。如下图所示：



从上图中显示的信息可以看出，发生错误的代码（图中用<= = =符号指出）是在radimo的init\_module函数中，因为一条mov [0], 1的汇编指令而发生了内存访问错误。同时我们还看到了调用轨迹，也就是说，radimo\_init是由\_\_module\_kernel\_version函数调用的，后者又是由sys\_init\_module函数调用的……从内核源代码中可以找到这些函数。以上所有的信息都对我们判断错误的来源、以及解决问题有所帮助。

总的来说，在Linux环境中调试设备驱动程序有一定的难度。但是一旦遇到问题，也不用惊惶失措，因为问题总是难免的。在Windows下调试设备驱动程序，有SoftIce这样强大的工具，在Linux下虽然没有这样优秀的工具支持，相应的文档也比Windows下的要少。但是Linux内核比Windows2000或者WindowsXP都更稳定，即使在内核态的驱动程序出现了致命错误，一般也不会马上死机，我们可以借此机会，获得Oops等宝贵的信息。另外，在Linux下调试设备驱动程序，我们还有一个最大的宝库，就是Linux内核源代码，这是全世界无数高手共同智慧的结晶。在调试设备驱动程序时发现的每一个错误，都是我们学习内核源码的一个契机，可以由此顺藤摸瓜，理解内核的工作机制，进而和操作系统的基本原理融会贯通。在这个过程中，你的收获之大，决不仅仅是消除了一个BUG那么简单，那将是一种全身心的、难以名状的愉悦。

最后，列出撰写本章内容的一些参考资料，读者在完成本章习题时，其中的一些资料将是至关重要的。

[1] Linux Device Drivers, 2nd Edition. ALESSANDRO RUBINI

[2] Linux设备驱动程序（第二版） （资料[1]的中译本）

[3] Linux程序设计（第二版） Neil Matthew 著，杨晓云 等译

[4] Universal Serial Bus Mass Storage Class Specification Overview

[5] Universal Serial Bus Specification, Revision 1.1

[6] Universal Serial Bus Mass Storage Class Bulk-Only Transport, Revision 1.0

[7] American National Standards for Information Systems――SCSI Primary Commands-2

[8] Software Driver of SmartMedia(TM), Ver 3.0 SAMSUNE Electronics

[9] USB 大容量存储设备的开发. Cypress公司 黄建明

[10] Linux下的硬件驱动——USB设备 联想软件设计中心 赵明

[11] PC机硬盘及通用分区结构 <http://www.yesky.com>

[12] <http://www.driverdevelop.com>