Linux 操作系统内核实验指导

编著 张鸿烈

2008-2009

目 录

第一章、Linux 操作系统内核实验环境

1.1 内核编程的特点

相对于用户空间的应用程序，内核开发有很大的不同。最重要的差异包括以下几点：

- 内核编程时不能访问 C 库。
- 内核编程时必须使用 GNU C。
- 内核编程时缺乏像用户空间那样的内存保护机制。
- 内核编程时浮点数很难使用。
- 内核只有一个很小的定长堆栈。
- 内核支持异步终端、抢占和 SMP，因此内核编程时必须时刻注意同步和并发。
- 内核编程要考虑可移植性。

1.1.1 使用 GNU C 编写和开发内核程序

Linux 内核使用 C 语言编写的，但 Linux 内核开发总是要用到一些 gcc 提供的 C 语言扩充部分。（gcc 是 GNU 编译器的集合，它包含了可以用于编译操作系统内核的 C 语言的扩展）

内核开发者使用的 C 语言涵盖了 ISO C99 和 GUN C 扩展特性，与标准 C 有区别,主要的扩展有：

1.内联函数（inline）

内联函数在它所调用的位置上展开。从而消除了函数调用和返回的开销。由于编译器会把内联函数在调用函数的代码处展开，所以也便于代码的优化。不过篇幅较大的程序段做成内联函数会使程序变长，通常把那些执行时间要求短且函数代码也比较短的函数定义成内联函数。

内联函数必须在调用之前定义好，通常在头文件或 C 文件开始的地方定义好内联函数。内联函数使用 static inline 关键字限制，以便编译时不会为内联函数单独建立一个函数体。例如：

static inline void dog(unsigned long tail_size)

2.内联汇编

Gcc 编译器支持 在 C 函数中嵌入汇编指令。当然，在内核编程时，只有知道对应的体系结构，才能使用这个功能。

Linux 的内核混合使用了 C 和汇编语言。在接近体系结构的底层或对执行时间要求严格的地方，一般嵌入汇编语言，而内核其他部分的大部分代码是用 C 语言写的。

3. 分支优化

对于条件选择语句，gcc 内建了一条优化指令。在一个条件经常出现，或者很少出现时，编译可以根据这条指令对条件分支选择进行优化。内核把这条指令封装成了宏：likely()和 unlikely()。例如：

if(foo){

```
/*……*/
}
```

如果这个选择的条件大多数情况都为假，则可以改进为：

```
if(unlikely(foo)){
/*……*/
}
```

反之，这个选择的条件大多数情况都为真，则可以改进为：

```
if(likely(foo)){
/*……*/
}
```

1.1.2 不能使用标准的 C 函数库

与用户空间的应用程序不同，内核不能链接使用标准的 C 函数库，主要的原因在于对于内核来说完整的 C 库太大了。但大部分常用的 C 库函数在内核中都已经实现了。比如操作字符串的函数组就位于内核文件 lib/string.c 中。只要包含<linux/string.h>，就可以使用它们。注意，内核程序中包含的头文件是指内核代码树中的内核头文件，不是指的开发应用程序时的外部头文件。内核代码是无法调用外部库函数的。

在内核中实现的库函数最著名的当数函数 printk()。它是库函数 printf()的内核版本。Printk()和printf()有基本相同的用法和功能。一个显著的区别是printk()可以带有一个优先级标志，syslog 程序会根据这个优先标志决定在什么地方显示这条系统消息。例如：

```
Printk(KERN_ERR “this is an error\n”
```

1.1.3 没有内存保护机制

如果一个用户程序试图进行一次非法的内存访问，内核会发现这个错误并结束整个用户进程。但内核自己非法访问了内存，那后果就很难控制了。内核中发生的内存错误会导致 oops 这是内核中出现的最常见的一类错误。在内核中，不应该去非法内存访问，引用空指针指类的操作，否则系统将会死掉，并没有任何错误提示。

此外，内核中的内存都不分页，也就是说每用掉一个字节，物理内存就减少一个字节。

1.1.4 不要轻易在内核中使用浮点数

在用户空间的进程进行浮点操作的时候，内核会完成从整数到浮点数操作的模式转换。在执行浮点指令时倒地会做些什么，则因体系结构的不同，内核的选择也不同。

和用户空间不同，内核并不能完美的支持浮点操作。在内核中使用浮点数时，除了要人工保存和恢复浮点寄存器外还有一些琐碎的事情要做。为了避免麻烦通常不在内核中使用浮点数。

1.1.5 容量小而且长度固定的内核栈

用户空间的程序可以从栈上分配大量的空间来存放变量，甚至用栈存放巨大的数据结构或者数组都没问题。之所以能这样做是因为应用程序是非常驻内存的，它们可以动态的申请和释放所有可用的内存空间。

内核要求使用固定常驻的内存空间，因此要求尽量少的占用常驻内存，而尽量多的留出内存提供给用户程序使用。因此内核栈的长度是固定大小的，不可动态增长的，32 位机的内核栈是 8KB。64 位机的内核栈是 16KB。

1.1.6 要求同步和并发的编程方法

内核很用以引起竞争条件。和单线程的用户空间程序不同，内核的许多特性要求能够并发的访问能散居据结构，这就要求有同步机制保证不出现竞争条件，特别是：

- Linux 内核支持多处理器的并发处理。所以如果没有适当的保护，在两个或两个以上的处理器上运行的代码很可能会同时访问共享的同一个资源。
- 中断是异步到来的，完全不顾及当前正在执行的代码。如果不加以适当的保护，中断完全有可能在代码正在访问共享资源时到来，从而使中断处理程序改变当前执行代码正在访问的共享资源，造成错误的结果。
- Linux 内核可以抢占。所以如果不加以适当的保护，内核中一段长在执行的代码可能会被另外一段代码抢占，从而有可能导致极端代码同时访问相同的资源。

常用的解决竞争问题的办法有自旋锁和信号量。

1.1.7 可移植性的重要性

尽管用户空间的应用能够程序可以不考虑可移植问题，然而 Linux 内核确实一个可移植的操作系统，并且要一直保持这种特性，也就是说，大部分 C 代码应该与体系结构无关，以便在多种不同体系结构的计算机上都能够编译和执行。例如要考虑保持字节序、64 位对齐、不假定字长和页面长度等一系列标准。

1.2 编译内核的方法

内核原代码组织为一个完整的树形结构，压缩为 tar 格式在网上以名称 Linux-x.xx.tar.gz 发布并可以自由下载（参考网址 http://www.kernel.org）。Linux2.6 以上的版本在其中配备了一套新的编译工具，使编译内核的工作比以前更加容易和自动化。

下载内核源代码包后用 tar 命令解压：

tar xzfv linux-x.xx.tar.gz

将会在当前目录下生成内核源代码目录：

linux-x.xx

进入该目录就可展开对该内核的编译和开发了。

内核源代码的阅读和参考可利用网站：http://lxr.linux.no，该网站提供 linux 各种版本的交叉引用。

1.2.1 内核编译参数的配置

在编译内核之前，首先你必须配置它。由于内核提供了数不胜数的功能，支持了难以计数的硬件，因而有许多东西需要配置。可以配置的各种选项通过带有 CONFIG 前缀的标识符来表示，比如 CONFIG_PREEMPT 代表内核抢占功能是否开启。这些配置项有二选一的，例如 CONFIG_PREEMPT 就是一个二选一的；也有三选一的，例如一些外设的驱动程序就是三选一的。二选一的选项就是 yes/no ，三选一的选项是 yes/no/module。选 module 意味着该配置项被选定了，但编译的时候这部分功能的实现代码是以模块（一种可以动态安装的独立代码段）的形式生成。

内核提供了各种不同的工具方便内核选项的配置。最简单的单也是最不直观的一种是字符界面下的命令行配置工具，启动命令是：

make config

该工具会以会话方式逐个遍历所有的配置项，要求用户每个都要选择 yes/no/module。这个过程需要耗费很长的时间，除非你的计算机没有图形界面。

另一个配置工具是利用 ncurse 库编制的菜单图形界面，它需要系统安装 ncurses 有关的软件包 。启动命令是：

make menuconfig

或者是用基于 X11 的图形工具，它需要安装 Qt3 有关的软件包 。启动命令是：

make xconfig

或者是用基于 X11 的图形工具，它需要安装 GTK+有关的软件包 。启动命令是：

make gconfig

这几种工具将所有的配置项分门别类以树形菜单放置，比如”processor fetures”和”network device”,你可以按类浏览察看或者修改内核配置。

如果你要用旧的配置文件重新配置内核时，可以使用命令：

make oldconfig

该命令会验证和更新内核到默认的配置。

这些配置选项被存放在内核代码树根目录下的.config 文件中，也可以通过直接修改该文件来配置内核选项。

当源文件被多次修改过，文件的依赖关系混乱时，可以使用命令：

make mrproper

该命令确保源代码目录下没有不正确的.o 文件，以及文件间依赖关系的正确性。

1.2.2 内核映像和模块文件的生成

在进行了内核的配置和验证后，你就可以通过 make 命令编译和生成新的内核文件了。这是一个比较长的过程，花费的时间要取决于你的计算机执行速度的快慢。如果

你不希望在编译过程中在显示器上查看错误和警告信息，而是希望在文件中查看错误和警告信息，你可以使用重定向命令将这些信息存入文件而不在显示器上显示。比如：

make > list_file

1.2.3 内核的安装和启动

在内核编译好之后，你还需要安装它。如何安装和你使用的系统以及启动引导工具有关，与您使用的 linux 系统的发行版本有关。例如：

在 Fedora 系统中你可通过命令：

1) 安装内核模块

make install_moudules

**注意如果系统没建对换分区，可能会中止新内核模块安装过程，解决的办法是增加 swap 分区文件。操作步骤如下：*

#dd if=/dev/zero of=swapfile bs=1024 count=102400

#mkswap swapfile

#swapon swapfile

#swapon -s

将新编译好的内核模块建立在/lib/module/目录下。然后使用命令：

2) 安装内核

make install

将内核映像文件 vmlinuz.img，系统引导文件 initrd.img（注意 initrd.img 的生成与 linux 系统的发行版的实现有关）和符号表文件 system.map 建立到/boot 目录中，在 /boot/grub/grub.conf 文件中填写好新内核的启动菜单项，并且保留以前的系统启动菜单项。这样，你可以有选择的选择你要启动的系统，特别是当你新开发的内核系统出现问题时，你可以通过重新选择以前的系统来找出问题，启动菜单的说明如下：

```
# grub.conf generated by anaconda
# Note that you do not have to rerun grub after making changes to this file
# NOTICE:  You have a /boot partition.  This means that
#          all kernel and initrd paths are relative to /boot/, eg.
#          root (hd0,0)
#          kernel /vmlinuz-version ro root=/dev/VolGroup00/LogVol00
#          initrd /initrd-version.img
#boot=/dev/sda
default=0          # 默认启动项。当前为启动菜单的第一项
timeout=5          # 默认启动等待选择时间。当前为 5 秒
splashimage=(hd0,0)/grub/splash.xpm.gz #启动菜单背景图文件
hiddenmenu         # 启动时隐藏启动菜单，仅当有按键才显示启动菜单
title Fedora (2.6.24.4) # 第一启动项名称
```

root (hd0,0) # 第一启动项位置

kernel /vmlinuz-2.6.24.4 ro root=/dev/VolGroup00/LogVol00 rhgb quiet #内核文件名及启动选项

initrd /initrd-2.6.24.4.img # 初始化文件名

title Fedora (2.6.23.17-88.fc7) # 第二启动项名称

root (hd0,0) #第二启动项位置

kernel /vmlinuz-2.6.23.17-88.fc7 ro root=/dev/VolGroup00/LogVol00 rhgb quiet #内核文件名及启动选项

initrd /initrd-2.6.23.17-88.fc7.img # 初始化文件名

3）**用新内核重新启动系统**

sudo reboot

在 Ubuntu 系统中内核的编译和安装:

1) 首先下载和安装内核开发包

sudo apt-get install build-essential kernel-package libncurses5-dev fakeroot

2) 下载和安装 xconfig 图形配置界面需要的图形工具包

sudo aptitude install libqt3-mt-dev libqt3-compat-headers libqt3-mt

3) 把内核源代码解压到/usr/src 目录下

sudo tar xzvf linux-2.6.xx.tar.gz -c /usr/src

cd /usr/src/linux-x.x.xx

4) 配置内核编译参数

sudo make xconfig

5) 编译生成内核和内核初始化解压程序。完成后会在/usr/src/linux-x.x.xx 下生成内核和内核头文件的 deb 安装包

sudo make-kpkg --rootcmd fakeroot --initrd kernel_image kernel_headers

6) 把内核的 deb 安装包安装到系统中，并重新配置/boot/grub/grub.conf 启动文件

cd /usr/src

sudo dpkg -i linux-headers-2.6.xx.Custom_i386.deb

sudo dpkg -i linux-image-2.6.xx.Custom_i386.deb

7) 用新内核重新启动系统

sudo reboot

1.3 内核的调试技术

内核级的程序开发和调试具有远大于于用户级程序开发的艰难和挑战，这是因为内核程序中的一个错误往往可令整个系统完全崩溃而没有任何挽救的措施。排除内核中错误的能力很大程度上取决于你的调试经验和对于系统的了解程度。

如果你能使一个错误重复出现则可以比较容易的找出和排除错误。但如果不能使错误重复出现则只能通过抽象出问题，分析出错原因来排除错误了。

内核、用户程序和硬件之间的交互非常微妙和复杂。一个竞争条件往往只在一个特定的环境、特定的配置和特定的硬件条件下才暴露出来。设计不佳的程序代码可能在

一些系统上表现的相当不错，而在另外的系统上却无法正常运行。这就要求对新设计的内核应当用尽量多的不同软硬件环境调试和测试它。

1.3.1 内核调试配置选项

为了方便调试和测试内核，内核代码树中有一些配置选项是针对内核调试的，它们都在配置编辑器的 kernel hacking 菜单项中，他们都依赖于 CONFIGDEBUG_KERNEL。当你开发内核时作为练习，不妨打开所有这些配置选项。常用的选项有：

Slab layer debugging() slab 层调试选项
High-memory Debugging 高端内存调试选项
I/O mapping debugging I/O 映射调试选项
Spin-lock debugging 自选锁调试选项
Sleep-inside-spinlock checking 自选锁内睡眠调试选项
Stack-overflow checking 栈溢出检查选项
其中的 Sleep-inside-spinlock checking 很有用。

1.3.2 内核中的打印函数 printk()

将内核的工作情况显示出来的最简单的方法就是使用内核格式化打印函数 prink()。在内核启动后的任何地方，任何时候都可以调用它输出要显示的格式化信息。可以在中断的上下文中使用，可以在持有锁时使用可以在多处理器上使用，而且调用者连锁都不必使用。 当然在系统启动过程中，终端还没有初始化之前 printk()不能使用

1.3.2.1 记录等级

Printk 函数和 printf 函数的用法和功能几乎相同，一个最主要的区别是可以在 printk()函数参数的开头指定一个记录等级。内核用这个指定的级别与当前终端的记录等级 console_loglevel 比较，如果指定的级别比当前终端的记录等级低则打印输出信息，否则不打印。可以指定的级别共有 8 级，在文件<linu/linux.h>文件中有它们的宏定义，它们是：

记录等级宏定义名	说明
0KERN_EMERG	一个紧急情况
1KERN_ALERT	一个需要立即注意的错误
2KERN_CRIT	一个临界情况
3KERN_ERR	一个错误
4KERN_WARNING	一个警告
5KERN_NOTICE	一个可能需要注意的错误
6KERN_INFO	一条非正式的消息
7KERN_DEBUG	一条调试信息

如果没有指定一个记录等级，printf() 函数会采用默认的级别

DEFAULT_MESSAGE_LOGLEVEL，它一般为 KERN_WARNING。你可以根据自己的要求指定一个等级来改变这个默认值。例如：

Printk(KERN_DEBUG “This is a debug notice\n”);

Printk(KERN_ERR “This is a error info\n”);

怎样指定一条信息的输出级别取决于你对信息输出的要求。一种选择是保持你的终端的默认记录等级不变，给所有调试信息加 KERN_CRIT 或更低的等级。反之，也可以给所有调试信息加 KERN_DEBUG 等级，而调整终端的默认记录等级。

1.3.2.2 记录缓冲区

内核消息都保存在一个为 LOG_BUF_LEN 大小的缓冲区中。该缓冲区大小可以在编译是通过 CONFIG_LOG_BUF_SHIFT 进行调整。在单处理器上其默认值为 16KB。因此内核在同一时间只能保持 16KB 的内核消息。该记录缓冲区是一个环形队列，即如果缓冲中消息已经放满，而又接收到新的消息，则旧的消息就会被新的消息覆盖。

使用环形队列的主要好处在于同步问题非常容易解决，所以即使在终端上下文中也可以方便的使用 printk()，此外，它使记录的维护也更容易。如果有大量的信息同时产生，新消息只要覆盖掉旧消息即可。在某个问题产生大量的消息时，记录只会覆盖掉它本身，而不会因为失控而消耗掉大量内存。但环形缓冲的唯一缺点是可能会丢失掉消息。

1.3.2.3 syslogd 和 klogd

在标准 Linux 系统上，用户空间的守护进程 klogd 从记录缓冲区中获取消息，再通过 syslogd

守护进程将它们保存到系统日志文件中。Klogd 程序可以从/proc/kmsg 文件中也可以通过 syslog()系统调用读取这些消息，默认情况下，它选择读取/proc 文件方式。不管哪种方式，直到所有新的内核消息可供读出，klogd 都会阻塞。在被环形之后，它会读取新的内核消息并进行处理。默认方式下，处理例程就是把消息传送给 syslogd 守护进程。

Syslogd 守护进程把它接收到的消息加进一个文件中，该文件默认的是/var/log/message。你也可以通过/etc/syslog.conf 配置文件重新指定这个文件的名字。

在载入 klogd 时，可以通过指定-c 标志来改变终端的记录等级。

1.3.4 oops 机制

 oops 是内核告知用户有错误发生的最常应用的方式。由于内核是整个系统的管理者，所以它不能采取像在用户空间出现错误使用的那些简单的手段，因为它很难自行修复或把自己杀死。内核只能发布 opps。这个过程包括向终端上输出错误信息，输出寄存器中保存的信息并输出可供跟踪的回溯线索。内核中出现的问题很难处理，所以内核往往要经历严峻的考验才能发送出 oops 和靠他自己文成一些清理工作。通常，发送完 oops 之后，内核会处于一个不稳定的状态。

举例来说，oops 发生的时候内核可能正在处理重要的事情。他可能持有一把锁或正在和硬件设备交互，内核必须从目前的上下文环境中退出并尝试恢复对系统的控制。多数情况这种尝试都会失败，因为如果 oops 在内核中断上下文时发生，内核根本无法继续，它会在陷入混乱，混乱的结果就是死机。如果 oops 在 idle 进程(pid 为 0)或 init 进程（pid 为 1）时发生，结果同样是系统陷入混乱，因为内核缺了两个重要的进程根本无法工作。不过，要是 oops 在其他进程运行时发生，内核就会杀死该进程并尝试继续执行。

Oops 的产生有很多可能的原因，其中包括内存访问越界或者是执行了非法指令等。

引发 bug 并打印信息

一些内核调用可以用来方便标记 bug，提供断言并输出信息。最长用的 两个是 BUG()和 BUG_ON()。当被掉用的时候，它们会引发 oops，导致栈的回溯和错误信息的打印。为什么这些声明会导致 oops 跟硬件的体系结构是相关的。大部分体系结构把 BUG()和 BUG_ON()定义成某中非法操作，这样自然会产生需要的 oops。你可以把这些调用当作断言使用，想要断言某种情况不该发生：

```
if(bad_thing)
    BUG();
```

或者使用更好的形式：

```
BUG_ON(bad_thing);
```

以下是一个由 BUG_ON 引发的 oops 输出信息：

```
Message from syslogd@ at Fri Sep 12 15:18:12 2008 ...
localhost kernel: ------------[ cut here ]------------
Message from syslogd@ at Fri Sep 12 15:18:12 2008 ...
localhost kernel: invalid opcode: 0000 [#1] SMP 段错误
[honglie@localhost ex6]$
Message from syslogd@ at Fri Sep 12 15:18:12 2008 ...
localhost kernel: Process open (pid: 2955, ti=dfb15000 task=c9104e90 task.ti=dfb15000)
Message from syslogd@ at Fri Sep 12 15:18:12 2008 ...
localhost kernel: Stack: 00000000 00000000 dfb15000 c0403f92 00000000 bff13438
0804845f 00000000
Message from syslogd@ at Fri Sep 12 15:18:12 2008 ...
localhost kernel: 4daf01e4 bff13448 ffffffda 0000007b 0000007b 00000000
00000145 ffffe410
Message from syslogd@ at Fri Sep 12 15:18:12 2008 ...
localhost kernel: 00000073 00000246 bff133dc 0000007b 00000000 00000000
Message from syslogd@ at Fri Sep 12 15:18:12 2008 ...
localhost kernel: Call Trace:
Message from syslogd@ at Fri Sep 12 15:18:12 2008 ...
localhost kernel: [<c0404f4c>] show_trace_log_lvl+0x1a/0x2f
Message from syslogd@ at Fri Sep 12 15:18:12 2008 ...
localhost kernel: [<c0404ffc>] show_stack_log_lvl+0x9b/0xa3
Message from syslogd@ at Fri Sep 12 15:18:12 2008 ...
localhost kernel: [<c04050ab>] show_registers+0xa7/0x178
Message from syslogd@ at Fri Sep 12 15:18:13 2008 ...
localhost kernel: [<c0405292>] die+0x116/0x1f7
Message from syslogd@ at Fri Sep 12 15:18:13 2008 ...
localhost kernel: [<c060bb80>] do_trap+0x8a/0xa3
Message from syslogd@ at Fri Sep 12 15:18:13 2008 ...
localhost kernel: [<c0405632>] do_invalid_op+0x88/0x92
Message from syslogd@ at Fri Sep 12 15:18:13 2008 ...
```

localhost kernel: [<c060b952>] error_code+0x72/0x78
Message from syslogd@ at Fri Sep 12 15:18:13 2008 ...
localhost kernel: [<c0403f92>] syscall_call+0x7/0xb
Message from syslogd@ at Fri Sep 12 15:18:13 2008 ...
localhost kernel: =======================
Message from syslogd@ at Fri Sep 12 15:18:13 2008 ...
localhost kernel: Code: 8b 45 0c 89 55 f8 8d 55 f4 e8 a0 61 0b 00 83 f8 01 19 c0 f7 d0 83 e0 f2 59 5b 5b 5d c3 55 89 e5 53 83 ec 04 8b 5d 08 85 db 75 04 <0f> 0b eb fe 8d 55 f8 89 d8 e8 0a dc ff ff 5a 85 c0 b8 ff ff ff
Message from syslogd@ at Fri Sep 12 15:18:13 2008 ...

可以用 panic()引发更严重的错误。panic()不但会打印错误消息，而且还会挂起整个系统。显然，你只应该在极端恶劣的情况下使用它：

```
if(terrible_thing)
    panic(“foo is %ld\n”,foo);
```

有时候，你只是需要在终端上打印一下栈的回溯信息来帮助你测试。此时可以使用 dump_stack()。它只在终端上打印寄存器上下文和函数的跟踪线索：

```
if(!debug_check){
    printk(KERN_DEBUG “provide some information...\n”);
    dump_stack();
}
```

可以使用 dmesg 命令查看到内核输出的信息：

```
provide some information...

Pid: 3046, comm: open Not tainted 2.6.24.4 #18
 [<c0404f4c>] show_trace_log_lvl+0x1a/0x2f
 [<c040574f>] show_trace+0x12/0x14
 [<c0405a45>] dump_stack+0x6c/0x72
 [<c0432625>] sys_myevent_open+0x1b/0xae
 [<c0403f92>] syscall_call+0x7/0xb
 =======================
```

第二章 Linux 内核实验内容

2.1 proc 文件系统实验

2.1.1 Proc 文件系统简介

通过编写与 proc 虚拟文件系统的接口程序快速的熟悉内核的一些重要特征。
在此首先说明一下 proc 文件系统和他在系统调试中所起的作用.
使用 proc 文件系统主要的优点在于：
1.应用程序获取系统内核数据无需切换到系统态，从而增加了系统的安全性。例如 ps 命令就是通过 proc 获取进程信息的。
2.当内核版本升级时，须读取内核信息的应用程序无需改动，从而改进了应用程序的兼容性。
3.应用程序还可以通过 proc 跟踪和调试内核。
4.应用程序可以通过 proc 直接改变内核参数，无需重新编译内核就可以改变或优化内核行为。
/proc 是一个虚拟文件系统，他以文件系统的形式向用户提供系统当前的运行状态。这就为用户应用程序获取系统内部信息提供了一个的安全、方便的界面。
proc 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。用户和应用程序可以通过 proc 得到系统的信息，并可以改变内核的某些参数。由于系统的信息，如进程，是动态改变的，所以用户或应用程序读取 proc 文件时，proc 文件系统是动态从系统内核读出所需信息并提交的。

2.1.2 pro 文件系统的目录结构

pro 文件系统的目录结构如下：

目录名称	目录内容
apm	高级电源管理信息
cmdline	内核命令行
Cpuinfo	关于 Cpu 信息
Devices	可以用到的设备（块设备/字符设备）
Dma	Used DMS channels
Filesystems	支持的文件系统
Interrupts	中断的使用
Ioports	I/O 端口的使用
Kcore	内核核心印象
Kmsg	内核消息
Ksyms	内核符号表
Loadavg	负载均衡
Locks	内核锁

Meminfo 内存信息
Misc Miscellaneous
Modules 加载模块列表
Mounts 加载的文件系统
Partitions 系统识别的分区表
Rtc Real time clock
Slabinfo Slab pool info
Stat 全面统计状态表 s
Swaps 对换空间的利用情况
Version 内核版本
Uptime 系统正常运行时间

并不是所有这些目录在你的系统中都有，这取决于你的内核配置和装载的模块。另外，在/proc 下还有三个很重要的目录：
net，scsi 和 sys。Sys 目录是可写的，可以通过它来访问或修改内核的参数（见下一部分），而 net 和 scsi 则依赖于内核配置。例如，如果系统不支持 scsi，则 scsi 目录不存在。
除了以上介绍的这些，还有的是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对应的一个目录
在/proc 下，以进程的 PID 号为目录名，它们是读取进程信息的接口。而 self 目录则是读取进程本身的信息接口，是一个 link。Proc 文件系统的名字就是由之而起。

2.1.3 Proc 文件系统中的进程目录

进程目录的结构如下：
目录名称 目录内容
Cmdline 命令行参数
Environ 环境变量值
Fd 一个包含所有文件描述符的目录
Mem 进程的内存被利用情况
Stat 进程状态
Status Process status in human readable form
Cwd 当前工作目录的链接
Exe Link to the executable of this process
Maps 内存印象
Statm 进程内存状态信息
Root 链接此进程的 root 目录
用户如果要查看系统信息，可以用 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
```

有关 proc 文件系统详细的结构和其中各个目录的意义，可以通过 man proc 命令联机查阅。

2.1.4 proc 文件系统的实验程序样例

```
/*************************************************************************
**
                          main.c  -  description
    begin                : 月/日/年
    copyright            : (C) 2003 by XXX
    Function             : 观察 linux 内核行为
/*************************************************************************
**/
#include <stdio.h>
#include <sys/time.h>
#define LB_SIZE 80
enum TYPE{STANDARD,SHORT,LONG};
FILE *thisProcFile;
  //Proc 打开文件指针
struct timeval now;
  //系统时间日期
enum TYPE reportType;
 //观察报告类型
char repTypeName[16];
char *lineBuf;       //proc 文件读出行缓冲
int interval;        //系统负荷监测时间间隔
int duration;        //系统负荷监测时段
int iteration;
char c1,c2;          //字符处理单元

void sampleLoadAvg(){ //观察系统负荷
     int i=0;
        //打开负荷文件
        ...
```

```
        //读出、处理读出行，如去除前导空格和无用空格
        ...
        //将读出行分出不同字段,按照字段的不同含义处理为可阅读格式
        ...
        //打印处理好的信息内容
        …
        ...
    fclose(thisProcFile);

}

void sampleTime(){//观察系统启动时间
    long uptime,idletime;
    int  day,hour,minute,second;
    int  i,j;
    char temp[80];
     i=j=0;

//打开计时文件
        …
        //读出、处理读出行，如去除前导空格和无用空格
        ...
        //将读出行分出不同字段,按照字段的不同含义处理为可阅读格式
        ...
        //打印处理好的信息内容
        …
        ...
        //将启动时间的秒数转换为长整数

        ...
        //转换成日时钟秒

        ...
        //将启动时间的空闲秒数转换为长整数
        ...
        //转换成日时钟秒
        ...
        //打印处理好的信息内容
            ...
}

int main(int argc,char *argv[])
{
     lineBuf = (char *)malloc(LB_SIZE+1);
```

```
  reportType = STANDARD;
  strcpy(repTypeName,"Standard");

  if(argc >1){
     sscanf(argv[1],"%c%c",&c1,&c2);//取命令行选择符
if(c1!='-'){        //提示本程序命令参数的用法
   exit(1);
}
if(c2 == 'b'){//观察部分 B
      printf("******PART B **********\n");
      //打开内存信息文件
        …
      //读出文件全部的内容
      …
     //处理并用方便阅读的格式显示
      ...
      fclose(thisProcFile);

     //观察系统启动时间
      sampleTime();
     }
    else if(c2=='c'){//观察部分 C
       printf("******PART C**********\n");
      //打开系统状态信息文件
        …
      //读出文件全部的内容
      …
      //处理并用方便阅读的格式显示
      ...
      fclose(thisProcFile);
    }

else if(c2 == 'd'){ //观察部分 D
      printf("******PART D **********\n");
  if(argc<4){
     printf("usage:observer [-b] [-c][-d int dur]\n");
     exit(1);
  }
  reportType = LONG;
  strcpy(repTypeName,"Long");

  //用命令行参数指定的时间段和时间间隔连续的
  //读出系统负荷文件的内容用方便阅读的格式显示
  ...
```

```
    }
    else{//观察部分 B
     printf("******PART A **********\n");
     reportType = SHORT;
     strcpy(repTypeName,"Short");
     //取出并显示系统当前时间
     ...
     //读出并显示机器名
     ...
     fclose(thisProcFile);

     //读出并显示全部 CPU 信息

      ...
     fclose(thisProcFile);

      //读出并显示系统版本信息
      ...
     fclose(thisProcFile);
   }
}
```

2.1.4 实验问题

问题 A:

提取

1、cpu 类型

2、内核版本

问题 B:

1、启动以来经历的时间，以 dd:hh:mm:ss 报告

问题 C:

1、cpu 执行用户态、系统态、空闲态所用时间

2、多少次磁盘请求

3、多少次上下文切换

4、启动了多少次进程

实验问题 D:

1、内存总量

2、可用内存

3、系统平均负荷

可完善以上样例程序完成问题 A-D 提出的问题。给出一个分析报告。

2.2 Shell 命令解释系统设计实验

2.2.1 设计自己的 shell 系统得意义

本实验是练习怎样编写与内核的接口程序。最经典的接口界面程序当属 Shell 命令解释程序，在 Unix 世界中有众多的 Shell 命令解释程序，它们各有自己的特色和特长。自己编写一个 Shell 命令解释程序不但可以定制特定的界面功能也可以深入了解和挖掘内核的各种技术。

Shell 命令解释程序中元字符的处理是 shell 中一个强大的功能，利用对各种元字符的不同解释可以充分挖掘出内核的各种强大的潜能。例如实验教材中提到的利用"&"符号启动并发的后台进程，利用"<"、">"符号启动I/O重定向，利用"|"启动管道读写等等。我们也可以实验实现一些其他的元字符功能，如"*"通配符，";"连接符等等。

2.2.2 linx 中管道的类型

关于管道，可有两种实现形式，即无名管道和有名管道。无名管道使用内存缓冲实现管道机制；有名管道使用管道文件实现管道机制，注意在程序执行之前先用 shell 命令建立两个命名管道，例如:

```
mkfifo fifiA -m 0666
mkfifo fifoB -m 0666
```

或用程序执行系统调用建立两个命名管道:

```
mkfifo("fifoA",0666);
mkfifo("fifoB",0666);
```

(0666 表示这两个管道可读可写)

2.2.3 一个简易的 shell 解释系统样例

以下是一个简单的 Shell 命令解释程序，仅供大家参考，大家可根据自己的思路和要求编制功能更加完备和丰富的 Shell。

```
/*************************************************************************
**
                        myshell.c  -  description
                           -------------------
    copyright            : (C) 2002 by XXX
    Function             : 我的shell命令,练习&,<,>,|控制符控制进程并发、通
```

```
信、I/O
/*************************************************************************
**/
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>

#define  LS 80

int main(){
    char *lineBuf,*tempBuf,*filename;  //shell 命令行处理缓冲区
    char **argv1,**argv2;              //指向命令行可变参数串的指针
    int fd[2];                         //管道句柄
    pid_t pid1,pid2;                   //管道两端的进程号
     int argc,status,in_arg;               //其他临时工作单元
    int i,j,k;
    int f_out,f_in;
    char is_pip,is_dir,is_bkg;


    //分配命令行缓冲区
    ...
    //分配命令行参数缓区
    ...
    printf("$"); //显示 shell 准备好提示

     //从标准输入读一行
     ...
     //循环处理命令行参数分割符并统计出参数个数
    ...
    ...
    //分配参数串指针空间
    ...

     //从命令行中分割各个参数为单独的串
    ...
    ...

    //判断参数串中输入了那些 shell 控制符号

        //有管道符吗?
           //分出管道两端命令名
           ...
```

```
        //建立管道
      ...
  //有 I/0 重定向符?
      //分出重定向文件名
       ...
  //父进程建立子进程 1
   …
  //如果为父进程
    //有后台执行符?
       //无,执行则等待子进程
       …
       //有,则立即 shell 准备好提示符
          printf("$");
    //有管道,管道写入文件结束符
           …
          close(fd[1]);//关闭管道输出
//如果为子进程 1
  if(is_pip=='|'){ //有管道
     //关闭子进程 1 读管道
      ...
     //进程 1 标准输出转向写管道
      ...
     close(fd[1]);
  }
  if(is_dir=='>'){ //如果有输出重定向,则打开该文件
      //以写方式打开重定向文件
      ...
      //新文件描述符代替标准输出
      ...
  }
  if(is_dir=='<'){ //如果有输入重定向,则打开该文件
      //以只读方式打开重定向文件
      ...
      //新文件描述符代替标准输入
      ...
  }

  //装入并执行子进程 1 新命令
  ...

if(is_pip=='|'){//有管道,
  //父进程建立子进程 2
  //如果为父进程
      //父进程等待子进程执行结束;
```

```
            ...
        //如果为子进程
            //关闭子进程 2 写管道
            //进程 2 标准输出转向读管道
            close(fd[0]);
            //装入并执行子进程 2 新命令
            …
}
/**************************END**************************************/
```

2.2.4 实验问题

问题 A:

实现一个能处理前后台运行命令的 shell

问题 B:

实现一个带有管道功能的 shell

问题 C:

实现一个能处理 I/O 重定向的 shell

问题 D:

实现一个能在一行上处理多条命令的 shell

将问题 A-D 集中到一个 shell 解析程序中。

2.3 内核的定时机制实验

2.3.1 内核定时机制的功能和作用

本实验是练习怎样编写调用内核的时间测量功能为应用程序测量和精确定时。通过这个实验我们可以进一步理解 Linux 内核的定时机制及其数据结构以及怎样从用户空间去访问内核空间的时间数据。

2.3.2 系统时间的获取和内核定时机制

从用户空间去获取系统时间数据需要以下基本代码：

```
#include <sys/time>
…
struct timeval{
    long tv_sec; //从 1970-1-1 12：到现在经过的秒数
    long tv_usec;//从从上 1 秒到现在经过的微秒数
} theTime;

…
gettimeofday(&theTime,NULL); //获取系统时间的系统调用
…
```

每个进程使用的各种定时器需要 Linux 的内部定时器。使用内部定时器可以跟踪记录 3 种不同类型的定时机制，它们反映了不同时间的划分，这些定时机制有各自不同的作用和应用。

它们是 ：

- ITIMER_REAL: 反映进程经过的实际时间，这种定时器到时后发 SIGALRM 信号。它与 struct task_struct 结构中的 it_real_value 和 it_real_incr 字段有关。
- ITIMER_VIRTUAL: 反映进程经过的虚拟时间，只有相关进程正在执行时该时间才会增加。这种定时器到时后发 SIGVTALRM 信号。与 struct task_struct 结构中的 it_virt_value 和 it_virt_incr 字段有关
- ITIMER_PROF:反映进程经过的虚拟时间加上内核为相关进程执行工作的时间之和。这种定时器到时后发 SIGPROF 信号。与 struct task_struct 结构中的 it_prof_value 和 it_prof_incr 字段有关。

每个定时器需要周期性的设定一个初始时间值，之后递减计数到 0 后引发定时中断，产生超时信号通知对应进程定时器时间到，然后定时器重新从设置的初始值再次开始递减计数。

三种定时器都使用 setitimer()系统调用进行初始化：

```
#include <sys/time.h>
…
```

```
setitimer(
        int timerType ,//定时器类型
        const struct itimerval *value, //定时器初始的和当前的秒数和毫秒数
        struct itimerval *oldValue
       )

struct itimerval{
    struct timeval it_it_interval; //下一次定时初值。若为 0 定时器停止
    struct timeval it_value  //定时器当前值
} ;
```

三种定时器都使用 getitimer()系统调用获取定时器当前值：

```
#include <sys/time.h>
…
getitimer(
        int timerType ,//定时器类型
        const struct itimerval *value, //定时器当前的秒数和毫秒数
       )
```

各类定时器到时的信号处理函数可以使用系统调用函数指定：

```
sighandler_t signal(int signum,  //信号类型
                    sighandler_t handler //信号处理函数名
                    );
```

2.3.3 利用内核的定时机制测试应用程序的例子

```
/*************************************************************************
**
                          mytimer.c  -  description
                             -------------------
    copyright            : (C) 2002 by XXX
    Function             : 测试并发进程执行中的各种时间。
                           给定 3 个菲波纳奇项数值可选在 36-45 之间
/*************************************************************************
**/

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/time.h>

static void psig_real(void);//父进程的 3 个定时中断处理函数原型
static void psig_virtual(void);
```

```
static void psig_prof(void);

static void c1sig_real(void);//子进程 1 的 3 个定时中断处理函数原型
static void c1sig_virtual(void);
static void c1sig_prof(void);

static void c2sig_real(void);//子进程 2 的 3 个定时中断处理函数原型
static void c2sig_virtual(void);
static void c2sig_prof(void);

long unsigned int fibonnacci(unsigned int n);

//记录 3 种定时的秒数的变量
static long p_real_secs=0,c1_real_secs=0,c2_real_secs=0;
static long p_virtual_secs=0,c1_virtual_secs=0,c2_virtual_secs=0;
static long p_prof_secs=0,c1_prof_secs=0,c2_prof_secs=0;

//记录 3 种定时的毫秒秒数的结构变量
static struct itimerval p_realt,c1_realt,c2_realt;
static struct itimerval p_virtt,c1_virtt,c2_virtt;
static struct itimerval p_proft,c1_proft,c2_proft;

int main(int argc,char **argv)
{
   long unsigned fib=0;
   int pid1,pid2;
   unsigned int fibarg;
   int status;
   int i;

   if(argc < 3) {
     printf("Usage: testsig arg1 arg2 arg3\n");
     return 1;
   }

   //父进程设置 3 种定时处理入口
   signal(SIGALRM,psig_real);
   signal(SIGVTALRM,psig_virtual);
   signal(SIGPROF,psig_prof);

   //初始化父进程 3 种时间定时器
   p_realt.it_interval.tv_sec = 9;
   p_realt.it_interval.tv_usec = 999999;
   p_realt.it_value.tv_sec = 9;
```

```
p_realt.it_value.tv_usec = 999999;
setitimer(ITIMER_REAL,&p_realt,NULL);

........

pid1 = fork();
if(pid1==0){
  //子进程 1 设置 3 种定时处理入口
  …
  //初始化子进程 1 的 3 种时间定时器

  ......

  //子进程 1 开始计算 fib
  fib = fibonacci(atoi(argv[1]));

  //打印子进程 1 所花费的 3 种时间值
  getitimer(ITIMER_REAL,&c1_realt);
  printf("Child1 fib=%ld\nChild1 Real Time=%ldSec:%ldMsec\n",
         fib,c1_real_secs+ 9 -c1_realt.it_value.tv_sec,
         999999 - c1_realt.it_value.tv_usec/1000);
    ......

}
else if((pid2=fork()) == 0){
  //子进程 2 设置 3 种定时中段入口
 ...
 //初始化子进程 2 的 3 种时间定时器
  ......

  //子进程 2 开始计算 fib
  …
  //打印子进程 1 所花费的 3 种时间值
  ......

}
else  {
  //父进程开始计算 fib
  ...
  //从内核中取出并打印父进程所花费的 3 种时间值
  ......

  //等待子进程结束
  waitpid(0,&status,0);
```

```
      waitpid(0,&status,0);
   }
}

//父进程的 3 个定时中断处理函数
static void psig_real(){
   p_real_secs += 10;
}
   ......


//子进程 1 的 3 个定时中断处理函数
static void c1sig_real(){
 ...
}
......

//子进程 2 的 3 个定时中断处理函数
static void c2sig_real(){
...
}
......

//fib 的递归计算函数
long unsigned int fibonacci(unsigned int n)
{
    ...
}

/************************END*****************************************/
```

2.3.4 实验问题

问题 A

使用 ITIMER_REAL 型定时器实现一个 gettimeofday()，将它设置为每秒产生一个信号，并计算已经经过的秒数。

问题 B

使用以上实现的 gettimeofday()实现一个精确到微秒级的“壁钟”。

问题 C

实现以上一个父进程和两个子进程并发递归计算不同项数的 fibonacci 序列的程序，分析每个进程三种类型定时器测出的时间关系。

2.4 动态模块设计实验

2.4.1 内核动态模块的的功能和作用

Linux 模块是一些可以独立于内核单独编译的内核函数和数据类型集合，是可增删的内核部分。模块在内核启动时装载称为静态装载，在内核已经运行时装载称为动态装载。模块可以扩充内核所期望的任何功能，但通常用于实现设备驱动程序，

2.4.2 模块最基本的框架

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>

static ini  __init 模块名_init(void)
{
    /*
* 安装模块的初始化工作 ……
*/
return 0;
}

Static void  __exit 模块名_exit(void)
{
    /*
* 卸载模块前的清理工作 ……
*/

}

module_init(模块名_init);
module_exit(模块名_exit);
```

2.4.3 动态模块的编译

Makefile 文件的约定
预定义的宏：
   obj-m := 模块名.o
预定义的命令：
   make -C 内核源代码的安装路径 M=动态模块源代码路径 modules

将生成动态安装的模块文件：模块名.ko

例如：

```
obj-m := hello.o
all:
    make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) modules
clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) clean
```

如果有多个目标模块：

```
obj-m := hello.o
hello-objs := a.o b.o
```

2.4.4 与动态模块有关的 Shell 命令

列出已安装的动态模块名：

```
lsmod
```

安装一个动态模块：

```
insmod 模块名.ko
```

卸载一个动态模块：

```
rmmod 模块名.ko
```

安装依赖模块：

```
modprobe
```

显示模块信息：

```
modinfo 模块名.ko
```

2.4.5 模块安装时携带的可选参数

```
#include <linux/moduleparam.h>
module_param(变量名,类型,权限)
```

支持的类型：

```
Byte.short,ushort,int,uint,long,ulong,bool,charp
```

例如在 hello.c 中加入：

```
#include <linux/moduleparam.h>
static int test;
module_param(test,int,0644);
```

用法 ：

```
insmod hello.ko test=10
```

许可和文档有关的宏

```
MODULE_LICENSE(“GPL”);
MODULE_DESCRIPTION(“xxxxxxxx”);
MODULE_AUTHOR(“xxxxxxx”);
```

2.4.6 在模块中使用内核的/proc 接口

建立一个 proc 目录文件：

```
struct proc_dir_entry *
proc_mkdir(const char *,          /*要创建的 proc 目录名*/
            struct proc_dir_entry *      /*上级目录，NULL 代表/proc*/
            );
```

建立一个 proc 普通文件：

```
struct proc_dir_entry *
create_proc_entry(const char *name,      /*要创建的 proc 文件名*/
                  mode_t mode,           /*proc 文件的权限*/
                  struct proc_dir_entry *parent/*上级目录，NULL 代表/proc*/
                              );
```

删除一个 proc 文件：

```
void
remove_proc_entry(const char *name, /*要删除的 proc 文件名*/
                struct proc_dir_entry *parent   /*上级目录，NULL 代表/proc*/
                );
```

proc 入口数据结构：

```
struct proc_dir_entry {
    unsigned int low_ino;                       /*目录入口的 inod 节点号*/
    unsigned short namelen;                         /*节点名长度*/
    const char *name;                           /*节点名*/
    mode_t mode;                                    /*节点类型和权限*/
    nlink_t nlink;                              /*节点的连接数*/
    uid_t uid;                                  /*拥有该节点的用户 uid*/
    gid_t gid;                                  /*拥有该节点的组 gid*/
    loff_t size;                              /*节点的大小，除非限制长度否则为 0*/
    struct inode_operations * proc_iops;            /*对该节点的操作*/
    const struct file_operations * proc_fops;       /*对该文件的操作*/
    get_info_t *get_info;                   /*如果定义，则当有读操作时调用*/
    struct module *owner;                       /*拥有该节点的模块*/
```

```
    struct proc_dir_entry *next, *parent, *subdir;/*节点间的关系，初始为 NULL*/
    void *data;                                   /*节点对应文件中内容*/
    read_proc_t *read_proc;                       /*读操作函数*/
    write_proc_t *write_proc;                     /*写操作函数*/
    atomic_t count;                               /* 节点引用计数 */
    int deleted;                                  /* 节点删除标志 */
    void *set;
  };
```

其中几个重要的字段：
owner=THIS_MODULE;
data＝指向文件数据区（自己定义）
read_proc=指向你的读操作处理函数（自己定义）
write_proc=指向你的写操作处理函数（自己定义）

读写操作函数的参数：

```
int (read_proc_t)(char *page,                 /*存读出数据的地址*/
                  char **start,               /*内核中不用*/
                  off_t off,                  /*读出数据在 page 中的偏量*/
                  int count,                  /*读出数据字节数*/
                  int *eof,                   /*文件尾标志*/
                  void *data);                /*文件数据的地址*/

int (write_proc_t)(struct file *file,         /*通常不用*/
                  const char __user *buffer,  /* 用 户 空 间 数 据 地 址 ， 需 要 用
copy_from_user 写入*/
                  unsigned long count,        /*从用户空间写入的数据长度*/
                   void *data);               /*写入文件的地址*/
```

2.4.7 利用动态模块创建 Proc 文件的样例

```
/**************************************************************************
                          main.c  -  description
    begin                : 一 4 月   4 21:01:11 CST 2003
    copyright            : (C) 2003 by 张鸿烈
    Function             : 编写创建 proc 文件系统的模块,该程序创建在/proc 目录下
                         : 创建 mydir 目录,在 mydir 目录下创建保存当前系统时间
                         : jiffies 值的文件 myfile,
**************************************************************************/
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/proc_fs.h>
#include <linux/jiffies.h>
```

```
#include <asm/uaccess.h>
#include <linux/moduleparam.h>
#define MODULE_NAME "Myproc"
#define MYDATA_LEN    16

//放用户空间传入的数据
struct my_proc_data{
   char value[MYDATA_LEN];
};
struct my_proc_data mydata;

//proc 结构变量
static struct proc_dir_entry *example_dir;
static struct proc_dir_entry *date_file;
static int    param;
module_param(param, int , 0644);

//读文件 myfile 的读驱动函数
static int proc_read(char *page, char **start,off_t off,int count,int *eof,void *data ){
     int len=0 ;
     struct my_proc_data *mydatap = (struct my_proc_data *) data;
     len+=sprintf(page,"%s%ld\n",mydatap->value,jiffies);
     return len;
}

//写文件 myfile 的写驱动函数
static int proc_write(struct file *file,const char *buffer,unsigned long count,void *data){
     int len ;
     struct my_proc_data *mydatap = (struct my_proc_data *) data;
     if(count>MYDATA_LEN)
          len = MYDATA_LEN;
     else
          len = count;
     if(copy_from_user(mydatap->value,buffer,len)){
          return -EFAULT;
     }
     mydatap->value[len-1] = '\0';
     return len;
}

//装入模块
int init_module(void)
{
   //创建 proc/myfile 目录
```

```
    example_dir = (struct proc_dir_entry * )proc_mkdir("mydir",0);
    if(example_dir == 0){
        printk("mkdir fail\n");
          return -1;
    }

  //创建/proc/mydir/myfile 文件
  date_file = (struct proc_dir_entry * )create_proc_entry("myfile",0666,example_dir);
  if(date_file == 0){
          remove_proc_entry("myfile",0);
          printk("mkfile fail\n");
          return -ENOMEM;
    }
    strcpy(mydata.value,"Ticks=");
    date_file->data=&mydata;
    date_file->read_proc=&proc_read;
    date_file->write_proc=&proc_write;
    date_file->owner=THIS_MODULE;
    return 0;
}

//卸载模块
void cleanup_module(void)
{
        remove_proc_entry("myfile",example_dir);
        remove_proc_entry("mydir",NULL);
        printk("Goodbye.\n");
}

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Test");
MODULE_AUTHOR("xxx");
```

2.4.8 实验问题

问题 A

分析实验以上模块，编写一个测试该模块的用户程序。比较该模块读取的的时间和用 gettimeofday()读取的时间的精度。

问题 B

实现一个模块用它遍历当前进程的父进程和任务队列，并将遍历的结果输出到一个 proc 文件中（遍历可以从 current 当前进程开始，父进程遍历到初始化进程，遍历任务队列可以利用 for_each_process 宏）。

2.5 新系统调用设计实验

系统调用是内核提供给用户应用程序使用的内核函数名，这些函数提供了内核为用户应用程序所提供的系统服务功能。这些函数在用户应用程序中的书写格式与用户自定义函数形式上没有什么不同，但这些函数的编写和调用过程却与自定义函数有着很大的不同。它们需要事先在内核中安排好入口和函数体，当调用时会引发系统软中断根据对应的存根函数查中断表进入请求的内核函数。

2.5.1 在内核源代码中引入用户自定义系统调用入口

在内核中加入新的系统调用需要遵照系统规定的统一规范，一般步骤为：

1、在采用的系统内核源代码中找到与系统架构相对应的系统调用表文件如：syscall_table.S，在中添加新的系统调用函数名，要求声明的格式为：

long sys_系统调用函数名

例如：

.long sys_mySyscall

2、在采用的系统内核源代码中找到与系统架构相对应的系统调用号文件如：unistd.h，在其中添加新的系统调用号的宏定义，要求声明的格式为：

#define _NR_ 系统调用函数名 系统调用最后一个编号

例如：

#define _NR_mySyscall 333

3、选择一个编有系统调用函数的内核文件将新加入的系统调用函数编写进去，例如：

许多系统调用在 kernel/sys.c 中，我们也可在其中编写一个新的系统调用函数，假设该函数带有两个参数，第一个参数为一个 struct timeval 结构的指针，可通过该指针取回系统的秒和毫秒值，第二参数为 struct timespec 结构的指针，可通过该指针取回系统的秒和纳秒值 。该函数返回 jiffies 的值。

```
asmlinkage long  sys_mySyscall( struct timeval *  t_time,
                                struct timespec * x_time)
{
 /*
 *This is my Syscall funtion
 * get  timeval and  timespec  while return jiffies
 */

}
```

重新编译内核，用新内核重启系统。

2.5.2 在用户空间访问新加入的系统调用

1. 建立测试新系统调用的应用程序

```
#include <stdio.h>
#include <errno.h>
#include <sys/time.h>
#include <linux/unistd.h> //不同内核版本可能位置不同
//系统调用入口汇编软中断宏，
//2.6.X 系统之前定义的方法
//_syscall2(long,my_syscall,struct timespec*,x_time,struct timeval*,theTime)

//2.6.X 系统之后定义的方法
long my_syscall(struct timeval *v_time,struct timespec *s_time){
    return syscall(__NR_my_syscall,v_time,s_time);
}

int main()
{
    //编写调用新的系统调用的程序
    //打印出各种时间值,将它们转换成日期和时、分、秒、微、秒、纳秒显示
     …
}
```

2、编译测试用的应用程序,例如该程序名为 testcall.c：

```
gcc -D__KERNEL__ testcall.c -o testcall
```

3、执行 testcall 检测看是否能打印出各种系统时间值。

2.5.3 实验问题

问题 A

分析实现以上系统调用，编写一个测试该系统调用的用户程序。比较该系统调用读取的时间和用 gettimeofday()读取的时间的精度。

问题 B

编写一个能统计某时间段中缺页中断次数的程序。由于每次缺页都要进入缺页中断函数 do_page_fault 一次所以可以认为执行该函数的次数就是缺页发生的次数。可以定义一个全局变量作为缺页计数器，并在 do_page_faul 函数中每次加 1。统计程序可以利用一个能返回缺页计数器和系统时间值的系统调用，在某时间段开始和结束点调用两次系统调用，其差值即为该时间段中产生的缺页中断次数。

2.6 构造新内核同步机制实验

2.6.1 同步机制设计的总体思路

要设计一组新的内核同步原语，它们具有如下的功能：
能够使多个进程阻塞在某一特定的事件上，直到另一进程完成这一事件释放相关资源，给内核发送特定消息然后由内核唤醒这些被阻塞的进程。如果没有进程阻塞在这个事件上，则消息被忽略。
可以编写 4 个系统调用来实现这些功能要求：
1、生成一个事件的系统调用函数：

int myevent_open(int eventNum);

生成一个事件，返回该事件的 ID，如果参数为 0，表示是一个新的事件，否则就是一个存在的事件。
2、将进程阻塞到一个事件的系统调用函数：

int myevent_wait(int eventNum);

进程阻塞到 eventNum 事件,直到该事件完成才被唤醒。
3、唤醒等某事件进程的系统调用函数：

int myevent_signal(int eventNum);

唤醒所有等 eventNum 事件的进程，如果队列为空，则忽略。
4、撤销一个事件的系统调用函数：

int myevent_close(int eventNum);

撤销一个参数代表的事件，成功返回 0。

2.6.2 设计事件的数据结构和系统调用函数

进程阻塞需要一个等待队列，所有等待同一事件号的进程都挂接到该队列。可以考虑定义如下的一个队列结点的数据结构：

```
typedef struct __myevent{
    int eventNum;           // 事件号
    wait_queue_head_t *p;   // 系统等待队列首指针
    struct __myevent *next; // 队列链指针
}myevent_t;
```

然后说明两个全局的指针变量：

```
myevent_t * lpmyevent_head = NULL ; // 链头指针
myevent_t * lpmyevent_end = NULL ;  // 链尾指针
```

说明一个事件查找函数：

```
myevent_t * scheventNum(int eventNum, myevent_t **prev)
{
    myevent_t *tmp = lpmyevent_head;
```

```
    *prev = NULL;
    while(tmp){
        if(tmp->eventNum == eventNum)
            return tmp;
        *prev = tmp;
        tmp = tmp->next;
    }
    return NULL;
}
```

建立或查找事件的系统调用，它返回建立的信事件的编号：

```
asmlinkage int sys_myevent_open(int eventNum)
{
  myevent_t *new;
  myevent_t *prev;
  if(eventNum)
    if(!scheventNum( eventNum, &prev))
        return -1;
    else
        return eventNum;
  else{
    new =  (myevent_t *) kmalloc(sizeof(myevent_t),GFP_KERNEL);
    new->p = (wait_queue_head_t *) kmalloc(sizeof(wait_queue_head_t),GFP_KERNEL);
    new->next = NULL;
    new->p->task_list.next = &new->p->task_list;
    new->p->task_list.prev = &new->p->task_list;
    if(!lpmyevent_head){
        new->eventNum = 2; //从 2 开始按偶数递增事件号
        lpmyevent_head = lpmyevent_end = new;
        return new->eventNum;
    }
    else{ //事件队列不为空，按偶数递增一个事件号
        new->eventNum = lpmyevent_end->eventNum + 2;
        lpmyevent_end->next = new;
        lpmyevent_end = new;
    }
    return new->eventNum;
  }
  return 0;
}
```

等待事件的系统调用(内核 2.6.XX)：

```
asmlinkage int sys_myevent_wait(int eventNum)
{
```

```
  myevent_t *tmp;
  myevent_t *prev = NULL;

  if((tmp = scheventNum( eventNum, &prev)) != NULL){
   DEFINE_WAIT(wait);  //初始化一个 wait_queue_head
   prepare_to_wait(tmp->p,&wait,TASK_INTERRUPTIBLE);  //当前进程进入阻塞队列
    schedule();   //重新调度
   finish_wait(tmp->p,&wait);  //进程被唤醒从阻塞队列退出
   return eventNum;
    }
  return -1;
}
```

唤醒等待事件进程的系统调用：

```
asmlinkage int sys_myevent_signal(int eventNum)
{
    myevent_t *tmp = NULL;
    myevent_t *prev = NULL;

    if(!(tmp = scheventNum(eventNum,&prev)))
        return 0;
    wake_up(tmp->p); //唤醒等该事件的进程
    return 1;
}
```

撤销某个事件的系统调用：

```
asmlinkage int sys_myevent_close(int eventNum)
{
    myevent_t *prev=NULL;
    myevent_t *releaseItem;

    if(releaseItem = scheventNum(eventNUm,&prev)){
      if( releaseItem == lpmyevent_end)
        lpmyevent_end = prev;
      else if(releaseItem == lpmyevent_head)
        lpmyevent_head =  lpmyevent_head->next;
      else
        prev->next = releaseNum->next;

      sys_myevent_signal(eventNum);
      if(releaseNum){
        kfree(releaseNum);
        return releaseNum;
      }
```

```
    }
    return 0;
}
```

重新编译内核，用新内核重启系统。

2.6.3 测试设计的同步机制

建立一个事件的测试程序：

```
#include <linux/unistd.h>
#include <stdio.h>
#include <stdlib.h>
int myevent_open(int flag){
    return syscall(__NR_myevent_open,flag);
 }
int main(int argc, char ** argv)
{
    int i;
    if(argc != 2)
        return -1;
    i = myevent_open(atoi(argv[1]));
        printf("%d\n",i);
    return 0 ;
}
```

让进程阻塞在一个事件的测试程序：

```
#include <linux/unistd.h>
#include <stdio.h>
#include <stdlib.h>
int myevent_wait(int flag){
    return syscall(__NR_myevent_wait,flag);
}

int main(int argc, char ** argv)
{
    int i;
    if(argc != 2)
        return -1;
    i = myevent_wait(atoi(argv[1]));
    printf("%d\n",i);
    return 0 ;
}
```

唤醒阻塞进程的测试程序：

```
#include <linux/unistd.h>
#include <stdio.h>
#include <stdlib.h>

int myevent_wait(int flag){
    return syscall(__NR_myevent_signal,flag);
 }

int main(int argc, char ** argv)
{
    int i;
    if(argc != 2)
        return -1;
    i = myevent_signal(atoi(argv[1]));
    printf("%d\n",i);
    return 0 ;
}
```

撤销一个事件的测试程序：

```
#include <linux/unistd.h>
#include <stdio.h>
#include <stdlib.h>
 int myevent_close(int flag){
    return syscall(__NR_myevent_close,flag);
 }
int main(int argc, char ** argv)
{
    int i;
    if(argc != 2)
        return -1;
    i = myevent_close(atoi(argv[1]));
        printf("%d\n",i);
    return 0 ;
}
```

假定将以上四个测试程序分别编译为：open,wait,signal,close

运行：

#open 0

建立一个事件编号为 2 的事件，在接着运行几个：

```
#wait 2
......
#wait 2
```

#ps aux
将会看到以上几个运行 wait 的进程进入阻塞状态。接着运行：
#signal 2
#ps aux
将会看到以上几个运行 wait 的进程被唤醒。
若在 wait 后运行：
#close 2
运行 wait 的进程被唤醒,同时事件 2 被撤销。

2.6.4 实验问题

实验问题 A

分析以上进程同步机制的功能和工作过程，通过重新定义 struct __myevent 结构来模拟一个信号量同步机制，并编写测试程序检测它的功能。

实验问题 B

在 2.6 节实验问题 B 中我们加入了一个缺页计数器每当调用缺页处理函数则对其加一，并利用自定义系统调用返回该值。考虑一个并发控制方案利用读/写者自旋锁控制对于缺页计数器的访问。可利用模块读 proc 文件获取缺页计数器的值。编一个 shell 批处理命令并发启动多个读命令检测它的功能。

2.7 字符设备驱动程序实验

2.7.1 设备编号的内部表示

内核中使用定义在<linux/types.h>中的 dev_t 类型变量保存设备编号，在 2.6 中 dev_t 是一个 32 位整数，12 位用来表示主设备号，20 位用来表示次设备号。每种设备都对应一个唯一的设备号。

分配和释放设备号
2.6 中可以用静态或动态方法分配设备号。例如动态分配一个字符设备的函数：

```
int alloc_chrdev_region(
   dev_t *dev,                //保存获取的设备号
   unsigned int firstminor, //第一个次设备号，通常为 0
   unsigned int count,        //请求的连续的次设备数
   char *name);               //关联的设备名
```

设备卸载时应该释放这些设备号：

```
void unregister_chrdev_region(
     dev_t first,             //要释放的设备号
     unsigned int count);   //数量
```

程序中可用定义在<linux/kdev_t.h>的宏：
获得主设备号：MAJOR(dev_t dev)
获得次设备号：MINOR(dev_t dev)
主设备号和次设备号转换为 dev_t 类型：MKDEV(int major,int minor)

2.7.2 加载并建立设备文件

系统已将部分主设备号静态分配给了大多数常见设备（可见内核源码树中 Documentation/devices.txt 文件）。这些主设备号包括动态加载的主设备号可以从 /proc/devices 虚拟文件中动态的读到。
对于动态加载的设备可以通过输入或编写 shell 脚本动态的获取系统主设备号来建立设备文件。
与设备驱动有关的重要数据结构
建立和注册了设备编号为的是将设备驱动的一些重要数据结构连接到这些编号上，大部分基本驱动程序都涉及到定义在<linux/fs.h>中的三个重要的内核数据结构：
file_operations
file
inode
（具体内容请对照查阅您使用的内核版本源代码）

file_operations 结构（指向它的指针常用 fops 表示）中包含了一组函数指针（方法），主

要用来实现系统调用，例如 open,read 等等。每个打开的文件和一组函数关联。对于特定设备中不支持的操作，该结构中对应指针可置为 NULL。一种标准化的结构初始化方法可以不编写 NULL 字段，仅声明设备可支持操作的函数名。例如对于只读设备可以声明为：

```
struct file_operations my_fops = {
    .owner = THIS_MODULE,
    .open = my_open,
    .read = my_read,
    .release = my_release,
};
```

open 方法的主要工作：

- 检查设备特定错误
- 如果首次打开，则进行初始化
- 如有必要，更新 fops 指针
- 分配并填写置于 filp->private_data 里的数据结构

release 方法的主要工作：

- 释放由 open 分配的保存在 filp->private_data 中的所有内容。
- 在最后一次关闭操作时关闭设备。

Read 方法主要的工作是从内核空间拷贝数据到用户空间，要使用函数：

```
unsigned long copy_to_user(void __user *to, //用户空间指针
                   const void *from, //内核空间指针
                   unsigned long count);
```

0 成功，大于 0 为还需要拷贝的内存数量，小于 0 表示有错。

write 方法主要的工作是从用户空间拷贝数据到内核空间，要使用函数：

```
unsigned long copy_from_user(void *to,          //内核空间指针
                   const void __user *from, //用户空间指针
                   unsigned long count);
```

0 成功，大于 0 为还需要拷贝的内存数量，小于 0 表示有错。

file 结构(指向它的指针常用 filp 表示)代表一个打开的文件(注意它与用户空间中的 FILE 结构无任何关联)，每执行一次 open 就建立一个(并传递给所有 fops 中声明了的操作函数)，直到所有打开的文件都关闭。

inode 结构在内核中包含了一个文件大量的内部信息，多个 filp 可以同时指向单一的一个 inode。对于字符设备驱动主要关注的两个字段：

```
struct inode{
  dev_t i_rdev;     //设备号
  struct cdev *i_cdev; //struct cdev 是字符设备内部结构，当某类字符设备有多个
                    //子设备时 i_cdev 是队列中的头指针。
  ......
```

```
}
```

考虑版本兼容性不要直接从中获取设备号，应使用两个宏来获取：

```
unsigned int iminor(struct inode *inode);
unsigned int imajor(struct inode *inode);
```

2.7.3 字符设备的注册

每个特定的字符设备驱动的数据结构都需要嵌入一个 struct cdev 结构（定义在 linux/cdev.h 中） 。

动态获取一个独立的 cdev:

```
struct cdev  *my_cdev = cdev_alloc();
my_cdev->ops = &my_fops;
my_cdev->owner =  THIS_MODULE;
```

初始化分配到的 cdev:

```
void cdev_init(struct cdev *cdev, struct file_operation *fops);
```

添加特定设备的 cdev 到内核设备链中：

```
int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
```

从内核中删除：

```
void cdev_del(struct cdev *dev);
```

2.7.4 设备中断处理

设备驱动程序模块在使用中断前要先请求一个中断通道号(IRQ)，注册一个中断处理例程。然后在用完后要释放该通道。申请和释放中断通道号的函数声明在头文件<linux/sched.h>中。

请求中断号函数：

```
int request_irq(unsigned int irq,  //中断号
                irqreturn_t ( * handler) //要安装的中断处理函数的指针
                            (int, //中断号
                            void *, //用于识别共享中断中的设备号
                            struct pt_reg *), //
                unsigned long flag, //中断管理位掩码,有以下定义：
                                    //SA_INTERRUPT 快速中断处理
                                    //SA_SHIRQ 中断共享
                                    //SA_SAMPLE_RANDOM 能贡献熵池随机数
                const char *dev_name, //中断拥有者字符串
                void *dev_id);      //用于识别共享中断中的设备号
```

当成功请求到中断通道时， request_irq 返回 0，返回负值表示错误码，返回-EBUSY 表示已有另一驱动程序占用了要请求的通道。

释放中断通道的函数：

```
void free_irq(unsigned int irq,void *dev_id);
```

在声明了一个中断请求函数后别忘了编写中断处理例程，即 request_irq 中的（* handler)所指向的函数，中断处理函数通常要读写设备端口数据或唤醒在该设备上等待的进程。

2.7.5 模拟字符设备的例题

一个简化了的读写内存操作的字符设备驱动的例子 scull（类似于一个虚拟 FIFO 设备驱动。）。主要简化了内存读写的功能，因此该例子仅能实现固定长度的设备读写。但通过它可以看出设备驱动程序最基本的内部工作过程。同学们可以通过以下程序具体实现一下这个虚拟的字符设备驱动。实验一下怎样测试和使用它，它的工作情况又是怎样的？进一步通过它分析一下 dev_t,struct file_operations, struct file, struct inode,struct cdev 以及一个特定的字符设备数据结构它们各自的作用及其之间的联系和实现的方法、步骤。

头文件 scull.h:

```
#ifndef _SCULL_H_
#define _SCULL_H_

#include <linux/ioctl.h> /* needed for the _IOW etc stuff used later */

/*
 * Macros to help debugging
 */

#undef PDEBUG             /* undef it, just in case */
#ifdef SCULL_DEBUG
#  ifdef __KERNEL__
     /* This one if debugging is on, and kernel space */
#    define PDEBUG(fmt, args...) printk( KERN_DEBUG "scull: " fmt, ## args)
#  else
     /* This one for user space */
#    define PDEBUG(fmt, args...) fprintf(stderr, fmt, ## args)
#  endif
#else
#  define PDEBUG(fmt, args...) /* not debugging: nothing */
#endif

#undef PDEBUGG
#define PDEBUGG(fmt, args...) /* nothing: it's a placeholder */

#ifndef SCULL_MAJOR
#define SCULL_MAJOR 0   /* dynamic major by default */
```

```
#endif

#ifndef SCULL_NR_DEVS
#define SCULL_NR_DEVS 4    /* scull0 through scull3 */
#endif

#ifndef SCULL_P_NR_DEVS
#define SCULL_P_NR_DEVS 4  /* scullpipe0 through scullpipe3 */
#endif

/*
 * The bare device is a variable-length region of memory.
 * Use a linked list of indirect blocks.
 *
 * "scull_dev->data" points to an array of pointers, each
 * pointer refers to a memory area of SCULL_QUANTUM bytes.
 *
 * The array (quantum-set) is SCULL_QSET long.
 */
#ifndef SCULL_QUANTUM
#define SCULL_QUANTUM 4000
#endif

#ifndef SCULL_QSET
#define SCULL_QSET    1000
#endif

/*
 * The pipe device is a simple circular buffer. Here its default size
 */
#ifndef SCULL_P_BUFFER
#define SCULL_P_BUFFER 4000
#endif

/*
 * Representation of scull quantum sets.
 */
struct scull_qset {
    void **data;
    struct scull_qset *next;
};

#define SCULL_DATA_SIZE         16
```

```
struct scull_dev {
    char data[SCULL_DATA_SIZE]; /* Pointer to first quantum set */
    unsigned long size;         /* amount of data stored here */
    struct semaphore sem;       /* mutual exclusion semaphore     */
    struct cdev cdev;       /* Char device structure        */
};

/*
 * Split minors in two parts
 */
#define TYPE(minor) (((minor) >> 4) & 0xf)  /* high nibble */
#define NUM(minor)  ((minor) & 0xf)     /* low  nibble */


/*
 * The different configurable parameters
 */
extern int scull_major;     /* main.c */
extern int scull_nr_devs;
extern int scull_quantum;
extern int scull_qset;

extern int scull_p_buffer;  /* pipe.c */


/*
 * Prototypes for shared functions
 */

int     scull_p_init(dev_t dev);
void    scull_p_cleanup(void);
int     scull_access_init(dev_t dev);
void    scull_access_cleanup(void);

int     scull_trim(struct scull_dev *dev);

ssize_t scull_read(struct file *filp, char __user *buf, size_t count,
                   loff_t *f_pos);
ssize_t scull_write(struct file *filp, const char __user *buf, size_t count,
                    loff_t *f_pos);
loff_t  scull_llseek(struct file *filp, loff_t off, int whence);
int     scull_ioctl(struct inode *inode, struct file *filp,
                    unsigned int cmd, unsigned long arg);
/*
```

```
 * Ioctl definitions
 */

/* Use 'k' as magic number */
#define SCULL_IOC_MAGIC  'k'
/* Please use a different 8-bit number in your code */

#define SCULL_IOCRESET    _IO(SCULL_IOC_MAGIC, 0)

/*
 * S means "Set" through a ptr,
 * T means "Tell" directly with the argument value
 * G means "Get": reply by setting through a pointer
 * Q means "Query": response is on the return value
 * X means "eXchange": switch G and S atomically
 * H means "sHift": switch T and Q atomically
 */
#define SCULL_IOCSQUANTUM _IOW(SCULL_IOC_MAGIC,  1, int)
#define SCULL_IOCSQSET    _IOW(SCULL_IOC_MAGIC,  2, int)
#define SCULL_IOCTQUANTUM _IO(SCULL_IOC_MAGIC,   3)
#define SCULL_IOCTQSET    _IO(SCULL_IOC_MAGIC,   4)
#define SCULL_IOCGQUANTUM _IOR(SCULL_IOC_MAGIC,  5, int)
#define SCULL_IOCGQSET    _IOR(SCULL_IOC_MAGIC,  6, int)
#define SCULL_IOCQQUANTUM _IO(SCULL_IOC_MAGIC,   7)
#define SCULL_IOCQQSET    _IO(SCULL_IOC_MAGIC,   8)
#define SCULL_IOCXQUANTUM _IOWR(SCULL_IOC_MAGIC, 9, int)
#define SCULL_IOCXQSET    _IOWR(SCULL_IOC_MAGIC,10, int)
#define SCULL_IOCHQUANTUM _IO(SCULL_IOC_MAGIC,  11)
#define SCULL_IOCHQSET    _IO(SCULL_IOC_MAGIC,  12)

/*
 * The other entities only have "Tell" and "Query", because they're
 * not printed in the book, and there's no need to have all six.
 * (The previous stuff was only there to show different ways to do it.
 */
#define SCULL_P_IOCTSIZE _IO(SCULL_IOC_MAGIC,   13)
#define SCULL_P_IOCQSIZE _IO(SCULL_IOC_MAGIC,   14)
/* ... more to come */

#define SCULL_IOC_MAXNR 14

#endif /* _SCULL_H_ */

/*****************************************************************************
```

```
**/
```

主程序 main.c 文件

```
#include <linux/config.h>
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/init.h>

#include <linux/kernel.h>   /* printk() */
#include <linux/slab.h> /* kmalloc() */
#include <linux/fs.h>       /* everything... */
#include <linux/errno.h>    /* error codes */
#include <linux/types.h>    /* size_t */
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <linux/cdev.h>

#include <asm/uaccess.h>    /* copy_*_user */

#include "scull.h"      /* local definitions */

/*
 * Our parameters which can be set at load time.
 */

int scull_major =   SCULL_MAJOR;
int scull_minor =   0;
int scull_nr_devs = SCULL_NR_DEVS;  /* number of bare scull devices */

module_param(scull_major, int, S_IRUGO);
module_param(scull_minor, int, S_IRUGO);
module_param(scull_nr_devs, int, S_IRUGO);

MODULE_AUTHOR("Alessandro Rubini, Jonathan Corbet");
MODULE_LICENSE("Dual BSD/GPL");

struct scull_dev *scull_devices;    /* allocated in scull_init_module */


/*
 * Empty out the scull device; must be called with the device
 * semaphore held.
 */
```

```
int scull_trim(struct scull_dev *dev)
{
    memset(dev->data, 0, SCULL_DATA_SIZE);
    dev->size = 0;
    return 0;
}

/*
 * Open and close
 */

int scull_open(struct inode *inode, struct file *filp)
{
    struct scull_dev *dev; /* device information */

    dev = container_of(inode->i_cdev, struct scull_dev, cdev);
    filp->private_data = dev; /* for other methods */

    /* now trim to 0 the length of the device if open was write-only */
    if ( (filp->f_flags & O_ACCMODE) == O_WRONLY) {
        if (down_interruptible(&dev->sem))
            return -ERESTARTSYS;
        scull_trim(dev); /* ignore errors */
        up(&dev->sem);
    }
    return 0;          /* success */
}

int scull_release(struct inode *inode, struct file *filp)
{
    return 0;
}

/*
 * Data management: read and write
 */

ssize_t scull_read(struct file *filp, char __user *buf, size_t count,
                loff_t *f_pos)
{
    struct scull_dev *dev = filp->private_data;
    ssize_t retval = 0;

    if (down_interruptible(&dev->sem))
```

```
        return -ERESTARTSYS;
    if (*f_pos >= dev->size)
        goto out;

    if (*f_pos + count > dev->size)
        count = dev->size - *f_pos;

    if (copy_to_user(buf, dev->data + *f_pos, count)) {
        retval = -EFAULT;
        goto out;
    }
    *f_pos += count;
    retval = count;

out:
    up(&dev->sem);
    return retval;
}

ssize_t scull_write(struct file *filp, const char __user *buf, size_t count,
                loff_t *f_pos)
{
    struct scull_dev *dev = filp->private_data;
    ssize_t retval = -ENOMEM; /* value used in "goto out" statements */

    if (down_interruptible(&dev->sem))
        return -ERESTARTSYS;

    if (*f_pos >= SCULL_DATA_SIZE) {
        retval = -ENOSPC;
        goto out;
    }
    if (*f_pos + count > SCULL_DATA_SIZE)
        count = SCULL_DATA_SIZE - *f_pos;

    if (copy_from_user(dev->data + *f_pos, buf, count)) {
        retval = -EFAULT;
        goto out;
    }
    *f_pos += count;
    retval = count;
    dev->size = *f_pos;

  out:
```

```
    up(&dev->sem);
    return retval;
}

loff_t scull_llseek(struct file *filp, loff_t off, int whence)
{
    struct scull_dev *dev = filp->private_data;
    loff_t newpos;

    switch(whence) {
        case 0: /* SEEK_SET */
            newpos = off;
            break;
        case 1: /* SEEK_CUR */
            newpos = filp->f_pos + off;
            break;
        case 2: /* SEEK_END */
            newpos = dev->size + off;
            break;
        default: /* can't happen */
            return -EINVAL;
    }
    if (newpos < 0)
        return -EINVAL;
    filp->f_pos = newpos;
    return newpos;
}

struct file_operations scull_fops = {
    .owner =    THIS_MODULE,
    .llseek =   scull_llseek,
    .read =     scull_read,
    .write =    scull_write,
    .open =     scull_open,
    .release =  scull_release,
};

/*
 * Finally, the module stuff
 */

/*
 * The cleanup function is used to handle initialization failures as well.
 * Thefore, it must be careful to work correctly even if some of the items
```

```
 * have not been initialized
 */
void scull_cleanup_module(void)
{
    int i;
    dev_t devno = MKDEV(scull_major, scull_minor);

    /* Get rid of our char dev entries */
    if (scull_devices) {
        for (i = 0; i < scull_nr_devs; i++) {
            scull_trim(scull_devices + i);
            cdev_del(&scull_devices[i].cdev);
        }
        kfree(scull_devices);
    }

    /* cleanup_module is never called if registering failed */
    unregister_chrdev_region(devno, scull_nr_devs);
}


/*
 * Set up the char_dev structure for this device.
 */
static void scull_setup_cdev(struct scull_dev *dev, int index)
{
    int err, devno = MKDEV(scull_major, scull_minor + index);

    cdev_init(&dev->cdev, &scull_fops);
    dev->cdev.owner = THIS_MODULE;
    dev->cdev.ops = &scull_fops;
    err = cdev_add (&dev->cdev, devno, 1);
    /* Fail gracefully if need be */
    if (err)
        printk(KERN_NOTICE "Error %d adding scull%d", err, index);
}


int scull_init_module(void)
{
    int result, i;
    dev_t dev = 0;

/*
```

```
 * Get a range of minor numbers to work with, asking for a dynamic
 * major unless directed otherwise at load time.
 */
    if (scull_major) {
        dev = MKDEV(scull_major, scull_minor);
        result = register_chrdev_region(dev, scull_nr_devs, "scull");
    } else {
        result = alloc_chrdev_region(&dev, scull_minor, scull_nr_devs,
                "scull");
        scull_major = MAJOR(dev);
    }
    if (result < 0) {
        printk(KERN_WARNING "scull: can't get major %d\n", scull_major);
        return result;
    }

        /*
     * allocate the devices -- we can't have them static, as the number
     * can be specified at load time
     */
    scull_devices = kmalloc(scull_nr_devs * sizeof(struct scull_dev), GFP_KERNEL);
    if (!scull_devices) {
        result = -ENOMEM;
        goto fail;  /* Make this more graceful */
    }
    memset(scull_devices, 0, scull_nr_devs * sizeof(struct scull_dev));

        /* Initialize each device. */
    for (i = 0; i < scull_nr_devs; i++) {
        init_MUTEX(&scull_devices[i].sem);
        scull_setup_cdev(&scull_devices[i], i);
    }

    return 0; /* succeed */

  fail:
    scull_cleanup_module();
    return result;
}

module_init(scull_init_module);
module_exit(scull_cleanup_module);

/*****************************************************************************
```

```
*/
```

Makefile 文件

```
scull-objs := main.o
obj-m := scull.o
all:
    make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) modules
clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) clean

/*****************************************************************************
/
```

加载设备驱动模块和安装设备文件的脚本程序文件 load

```
#!/bin/sh
# $Id: scull_load,v 1.4 2004/11/03 06:19:49 rubini Exp $
module="scull"
device="scull"
mode="664"

# invoke insmod with all arguments we got
# and use a pathname, as insmod doesn't look in . by default
/sbin/insmod ./$module.ko $* || exit 1

# retrieve major number
major=$(awk "\$2==\"$module\" {print \$1}" /proc/devices)

# Remove stale nodes and replace them, then give gid and perms
# Usually the script is shorter, it's scull that has several devices in it.

rm -f /dev/${device}[0-3]
mknod /dev/${device}0 c $major 0
mknod /dev/${device}1 c $major 1
mknod /dev/${device}2 c $major 2
mknod /dev/${device}3 c $major 3
ln -sf ${device}0 /dev/${device}
chmod $mode  /dev/${device}[0-3]

/*****************************************************************************/
```

卸载设备驱动模块和删除设备文件的脚本程序文件 unload

```
#!/bin/sh
module="scull"
device="scull"
# invoke rmmod with all arguments we got
/sbin/rmmod $module $* || exit 1
# Remove stale nodes
rm -f /dev/${device} /dev/${device}[0-3]
```

2.7.6 实验问题

问题 A

分析调试以上 scull 设备驱动，并用字符设备有关的系统命令测试其功能，进而解析字符设备 I/O 驱动的构造和基本工作过程。

问题 B

在驱动程序内部，阻塞在 read 调用的进程在数据到达时被唤醒；通常硬件会发一个中断来通知这个事件，然后作为中断程序处理的一部分驱动程序会唤醒等待进程。当没有硬件或中断处理程序时（如以上 scull 字符设备模拟程序）我们可以使用一个缓冲区和另一个进程写进程来产生数据并唤醒读取进程；类似的，阻塞在缓冲区 write 调用上的写进程也可以有另一读进程唤醒。这就是实现类管道设备的一种特殊设备驱动。请将以上 scull 设备驱动改造为这种管道类设备驱动程序。

2.8 文件系统实验

Lnux 文件系统使用虚拟文件系统 VFS 作为内核文件子系统。可以安装多种不同形式的文件系统在其中共存并协同工作。VFS 对用户提供了统一的文件访问接口，用户无需考虑具体的文件系统和实际的物理介质形式即可在多种不同的文件系统和物理介质间用同一种操作方法方便的交换数据。即 linux 的 VFS 使用高级的抽象方法巧妙的隐藏了多种文件系统和物理设备间的差异。

文件系统的抽象层

VFS 之所以能够连接各种各样的文件系统是因为它把各种形式的文件系统都抽象为 Unix 文件系统的四个对象：安装点、目录项、索引 i 节点、文件(普通，目录，设备)及其操作：

2.8.1 安装点对象

linux 文件系统是一个树形分层结构，文件系统有一个根节点各种文件系统在根节点之下被安装在某个特定的安装点上，该安装点在全局层次结构中被称为命名空间。一个已经安装的文件系统的安装点由超级块对象代表。超级块对象存储着特定文件系统的信息，通常对应于存放特定文件系统的文件控制块的特定磁盘扇区。超级块对象由 super_block 结构体表示：

```
struct super_block {
        ...
        const struct super_operations    *s_op;
        …
}
```

文件系统安装时，内核调用 alloc_super()函数，创建并初始化超级块并将其信息填充到内存超级块对象中。

超级块结构中的 s_op 域指向了一个超级块操作函数表结构体。

```
struct super_operations {
        struct inode *(*alloc_inode)(struct super_block *sb);
        …
}
```

超级块操作执行文件系统和 i 节点的低层操作，其中一些函数根据不同的文件系统是可选的，可以将不需要的操作函数指针置为 NULL。

2.8.2 索引 i 节点对象（活动 i 节点）

索引 i 节点对象包含了内核要操作的文件的全部控制信息，对应着打开文件的 i 节点表。即使没有 i 节点的文件系统其文件控制信息在 VSF 系统中也都转化为 i 节点对象。i 节点对象由 inode 结构表示：

```
struct inode {
        const struct inode_operations      *i_op;
        …
}
```

同超级块对象的处理类似，索引 i 节点操作也对应一组 i 节点操作表 inode_operations,

```
struct inode_operations {
        int (*create) (struct inode *,struct dentry *,int, struct nameidata *);
          …
}
```

用户的系统系统调用 create()和 open()就是调用了 inode_operations 中的 create 操作为对应目录项建立的 i 节点。

2.8.3 目录项对象

linux 中目录属于文件对象，但每个目录项属于目录项对象。通过目录项对象及其操作可以方便快速的进行路径名的查找、解析。

目录项对象代表了文件路径名的各个部分，目录文件名和普通文件名都属于目录项对象。目录项对象在文件访问过程中使用，所以没有对应的磁盘数据结构，数据不需要写回磁盘。目录项对象目录项对象有三种状态：

- 被使用：对应一个有效的 i 节点，并且正在使用。
- 未被使用： 对应一个有效的 i 节点，但当前未用。
- 负状态：无对应 i 节点，已备以后再用。

目录项对象由 dentry 结构体表示：

```
struct dentry {
     ...
     struct dentry_operations *d_op;
     ...
}
```

其中的 struct dentry_operations 结构说明了目录项对象所有的操作方法：

```
struct dentry_operations {
     …
     int (*d_compare) (struct dentry *, struct qstr *, struct qstr *);
     …
}
```

例如 d_compare()操作比较两文件名字符串是否相等，相等返回 0。

2.8.4 文件对象

文件对象表示已经打开的文件，它是直接连系用户应用程序的内核对象。它由用户进程的 open 系统调用建立，由 close 系统调用撤销。由于多个进程可以同时以不同方式打开和操作同一个文件，所以一个打开的文件可以同时对应多个文件对象，但多个进程通过它都指向同一目录对象进而指向同一个 i 节点对象。像目录对象一样文件对象不需要保存磁盘数据。

文件对象由 file 结构表示：

```
struct file {
          …
          struct file_operations      *f_op;
          …
}
```

其中的 struct file_operations 结构说明了文件对象所有的操作方法，这些正是我们所熟悉的文件系统调用的内核表示：

```
struct file_operations {
        int (*open) (struct inode *, struct file *);
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
          ...
}
```

2.8.5 VFS 文件系统其他辅助数据结构

用来描述各种文件系统特定的功能和行为的文件系统类型结构：

```
struct file_system_type {
          ...
        int (*get_sb) (struct file_system_type *, int,
                       const char *, void *, struct vfsmount *);
          ...
};
```

get_sb()函数从磁盘上读取超级块，并在文件系统安装时组装超级块对象。其他字段用于描述特定文件系统的属性。每种文件系统不管其有多少安装实例或还未安装到系统，系统都只有唯一一个 struct file_system_type 结构。

当某个文件系统被安装时，将有一个文件系统安装点数据结构 vfsmount 被创建。它记录着该文件系统的父文件系统、安装点目录、该文件的根目录、超级块、子文件系统、设备文件名等信息:

```
struct vfsmount {
        struct list_head mnt_hash;
        struct vfsmount *mnt_parent;       /* fs we are mounted on */
        struct dentry *mnt_mountpoint;    /* dentry of mountpoint */
        struct dentry *mnt_root;           /* root of the mounted tree */
        struct super_block *mnt_sb;        /* pointer to superblock */
        struct list_head mnt_mounts;       /* list of children, anchored here */
        struct list_head mnt_child;    /* and going through their mnt_child */
        int mnt_flags;
          …
};
```

通过 vfsmount 所维护的各种链表我们就可以跟踪和摸清各种文件系统复杂的关系。

2.8.6 与进程相关的文件系统数据结构

系统中每一个进程都有一组与其相关的打开文件，如：执行文件，要访问的数据文件、当前工作目录、安装点等。有三个数据结构将 VFS 和进程相关的文件联系到一起，进程可以通过它们访问到与自己有关的文件环境。

第一个数据结构是 files_struct 结构，主要用于描述进程打开文件对象。进程描述表

task_struct 中的 files 指针指向该结构：

```
struct files_struct {
        …
        struct file ** fd;
        struct file * fd_array[NR_OPEN_DEFAULT];
 };
```

f_array 指针数组指向一组已打开文件对象。fd 初始指向 f_array，当打开的文件个数多于 NR_OPEN_DEFAULT 的值时，内核会分配新的 struct file 数组并将 fd 指向新数组。

第二个数据结构是 fs_struct 结构，主要用于描述进程文件系统信息。进程描述表 task_struct 中的 fs 指针指向该结构：

```
struct fs_struct {
        atomic_t count;
        rwlock_t lock;
        int umask;
       struct path root, pwd;
};
```

其中的 struct path 结构描述了该进程的安装点、根目录项和当前目录项对象：

```
struct path {
        struct vfsmount *mnt;
        struct dentry *dentry;
};
```

通过 fs 我们可以找到与该进程有关的目录项、i 节点、安装点以及超级块信息。

第三个数据结构是 namespace 结构，主要为了进程可以共享同样的命名空间。进程描述表 task_struct 中的 namespace 指针指向该结构：

```
struct mnt_namespace {
        atomic_t                count;
        struct vfsmount *       root;
        struct list_head        list;
        wait_queue_head_t poll;
        int event;
  };
```

其中的 list 域是链接已安装文件系统的双向链表，由它可以遍历到全体安装文件的命名空间。

2.8.7 从当前进程访问内核 VFS 文件系统的例子

通过以上对 VFS 文件系统的简要分析，可以知道我们可以从当前进程探测和操纵 VSF 文件系统，以下代码说明了怎样获取当前进程所处的文件系统的块长和文件系统类型：

```
asmlinkage int sys_get_files_info(char * filesystem_type, unsigned long blk_size){
    struct fs_struct *fs ;

    struct vfsmount *mnt ;
    struct super_block *mnt_sb ;
    struct file_system_type *s_type;
```

```
    read_lock(&current->fs->lock);
            fs = current->fs;
            mnt = (&fs->pwd)->mnt;
            mnt_sb = mnt-> mnt_sb ;
            s_type = mnt_sb -> s_type;
            printk("PWD Filesystem Type is : %s\n",s_type->name);
            filesystem_type    = s_type->name;
            printk("PWD= %ld\n",mnt_sb->s_blocksize);
            blk_size = mnt_sb->s_blocksize);
        read_unlock(&current->fs->lock);
        return 0;
}
```

当在用户主文件夹下执行调用该函数的程序时会报告：
PWD Filesystem Type is : ext3
PWD Filesystem Blocksize = 4096

当我们插入一个使用 windows 系统的 FAT32 格式的 U 盘，在 U 盘目录下执行调用该函数的程序时会报告：
PWD Filesystem Type is : vfat
PWD Filesystem Blocksize = 512

2.8.8 实验问题

问题 A

编写一个 get_FAT_boot 函数，通过系统调用或动态模块调用它可以提取和显示出 FAT 文件系统盘的引导扇区信息。这些信息的格式定义在内核文件<include/linux/msdos_fs.h>的 fat_boot_sector 结构体中。函数可通过系统调用或动态模块调用。

问题 B

编写一个 get_FAT_dir 函数，通过系统调用或动态模块调用它可以返回 FAT 文件系统的当前目录表，从中找出和统计空闲的目录项（文件名以 0X00 打头的为从未使用过目录项，以 0XE5 打头的为已删除的目录项）,将这些空闲的目录项集中调整到目录表的前部。这些信息的格式定义在内核文件<include/linux/msdos_fs.h>的 msdos_dir_entry 结构体中。

2.9 块设备驱动程序实验

（本节讨论基于 linux-2.6.24 内核源代码）

2.9.1 块设备的接口和注册

一个块设备驱动程序主要通过传输固定大小的数据块(内核缺省定义为 1024 字节，对应着 512 字节的 2 段物理扇区）。因此块设备的驱动与字符设备的驱动有着很大的不同，要求有特定任务的接口。

块设备特定的分配和释放设备号函数同注册一个字符设备的过程一样，块设备工作的第一步也是首先请求系统分配一个主设备号。执行该任务的函数是在<linux/fs.h>声明的 register_blkdev 函数：

```
int register_blkdev(
   unsigned int major,          //如果为 0，则有内核分配并返回一个设备号
   const char *name);           //关联的设备名
```

设备卸载时应该释放注销这个设备号：

```
int unregister__blkdev(
    unsigned int major,     //要释放的设备号
    const char *name);  //关联的设备名
```

2.9.2 块设备的建立

register_blkdev 函数仅能获取一个设备号，但不能建立一个独立的磁盘，必须通过建立一个称为 gendisk（在<linux/genhd.h>中声明)的数据结构，注册一个磁盘。

```
struct gendisk {
    int major;          /* 主设备号 */
    int first_minor;        /* 第一个次设备号，一般为 0*/
    int minors;                 /* 次设备数（不是分区数），至少有一个*/
    char disk_name[32];     /* 主设备名 */
    struct hd_struct **part;    /* [次设备的分区索引] */
    struct block_device_operations fops; /* 块设备的操作函数入口*/
    struct request_queue queue;  /* 内核管理块设备 I/O 请求的队列结构*/
    void private_data;       /* 为驱动程序提供的访问内部数据的指针*/
    sector_t capacity;      /* 该设备具有的扇区数，sector_t 为 64 位宽*/
    int flags;          /* 驱动程序状态标志 */
    /* 一下字段本实验暂不涉及，不再说明 */
    ……
};
```

2.9.3 块设备操作

块设备通过在<linux/fs.h>中声明的 block_device_operations 结构提供对于块设备施加的操作：

```
struct block_device_operations {

    /* 打开、关闭设备。如定义旋转盘片，锁仓门，弹出盘片等*/
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);

    /* 设备控制。如获取磁盘物理参数等动作*/
    int (*ioctl) (struct inode *, struct file *, unsigned, unsigned long);
    long (*unlocked_ioctl) (struct file *, unsigned, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned, unsigned long);
    int (*direct_access) (struct block_device *, sector_t, unsigned long *);

    /* 如果该设备支持可移动介质，通过它们检查介质是否被更换，需刷新 */
    int (*media_changed) (struct gendisk *);
    int (*revalidate_disk) (struct gendisk *);
    int (*getgeo)(struct block_device *, struct hd_geometry *);

       /* 哪个驱动程序在管理该设备 */
    struct module *owner;
};
```

注意，该结构中没有负责读写的操作函数，块设备 I/O 要求高效的传输，因此都统一由内核提供的 request 机制来完成。

2.9.4 块设备的请求处理

1．请求结构体

块设备驱动程序的核心是请求函数，该函数成功调用后将构造出一个 request 结构的实例(定义在文件<linux/blkdev.h>中),之后的 I/O 处理函数都将围绕该结构的数据进行工作。request 结构涉及的信息很多,以下仅就本实验涉及的字段加以说明:

```
struct request {
        struct list_head queuelist;   /* 请求链表*/
        struct list_head donelist;
        struct request_queue *q;

        sector_t sector;            /* 传输的开始扇区索引号 */
        unsigned long nr_sectors; /* 需要传输的剩余扇区数 */
        unsigned int current_nr_sectors;
```

```
        /* 当前段中需要传输的剩余扇区数 */
        char *buffer;                /* 接收数据的缓冲区 */
         unsigned int cmd_flags;
           /* 数据传输方向 */
        /* 应通过宏 rq_data_dir(struct queue q) 操作 cmd_flags,*/
         /* 返回 0 表示从设备读,*/
        /* 返回非 0 表示向设备写*/

         struct bio *bio;          /* 请求的 bio 结构的链表头*/
         struct bio *biotail;      /* 请求的 bio 结构的链表尾*/

         void *elevator_private;   /* 用于 I/O 调度的指针*/
         void *elevator_private2;

         struct gendisk *rq_disk;  /* 对应的磁盘设备 */
         unsigned long start_time;

          .....

}
```

隐藏在 request 机制背后的底层机制是一个称为 bio 的机制，它把不连续的内存页映射为连续的 I/O 缓冲区。它将系统中所有 request 用链表管理起来并将所有 request 映射到物理内存页中(包括 DMA 传输区),一些高性能块设备可能需要 DMA 传输,还有一些经常进行随机访问的块设备(如 flash 内存,RAM 盘等)使用请求队列并不能提高性能。对于这些设备最好不用请求队列而是直接使用 bio 操作内存物理页进行数据传输。

一个 bio 对应一次 I/O 请求，但通过调度算法可将连续的 bio 合并成一个请求，即一个请求可以包含多个 bio。Bio 由 bio 结构体定义：

```
struct bio {
        sector_t                bi_sector;      /*要传输的第一扇区地址*/
        struct bio              *bi_next;       /* 下一个 bio */
        struct block_device     *bi_bdev;
        unsigned long           bi_flags;       /*状态、命令等 */
        unsigned long           bi_rw;          /*低位表示 READ/WRITE, 高位优先级*/
        unsigned short          bi_vcnt;        /*有多少*bio_vec's /
        unsigned short          bi_idx;         /* 当前 bvl_vec */

        /* 不相邻的物理段数目 */
        unsigned short          bi_phys_segments;

        /* 物理的和 DMA 合并后不相邻的物理段数目 */
        unsigned short          bi_hw_segments;
        unsigned int            bi_size;        /*字节为单位的传输数据长度 */

        /* 为明了最大 hw 尺寸，考虑本 bio 第一和最后一个虚拟可合并段的大小 */
```

```
        unsigned int            bi_hw_front_size;
        unsigned int            bi_hw_back_size;

        unsigned int            bi_max_vecs;     /*最大 bvl_vecs 数 */
        struct bio_vec          *bi_io_vec;      /* 实际的 vec 列表*/
        bio_end_io_t            *bi_end_io;
        atomic_t                bi_cnt;          /* pin count */
        void                    *bi_private;
        bio_destructor_t        *bi_destructor; /* destructor */
};
```

一个块设备的所有请求连成一个请求队列，并捆定到该设备。请求队列跟踪等候块 I/O 请求，提供使用多个 I/O 调度器（电梯算法）。请求队列由 request_queue 结构体定义(在文件<linux/blkdev.h>中)：

```
struct request_queue
{
        ------
        /* 保护队列结构体的自旋锁  */
        spinlock_t              __queue_lock;
        spinlock_t              *queue_lock;

        struct kobject kobj;    /* 队列内核对象*/

        /*队列的设置 */
        unsigned long           nr_requests;        /* 最大请求数 */
        unsigned int            nr_congestion_on;
        unsigned int            nr_congestion_off;
        unsigned int            nr_batching;

        unsigned int            max_sectors;        /*最大扇区数*/
        unsigned int            max_hw_sectors;     /*最大合并扇数*/
        unsigned short          max_phys_segments;  /*最大物理段数*/
        unsigned short          max_hw_segments;    /*最大合并段数*/
        unsigned short          hardsect_size;      /*物理扇长度*/
        unsigned int            max_segment_size;   /*段最大长度*/

        unsigned long           seg_boundary_mask; /*段边界掩码*/
        unsigned int            dma_alignment;      /*DMA 内存对齐*/
        ……
};
```

完成请求的队列初始化的函数 blk_init_queue(定义在文件<linux/blkdev.h>中)带有一个请求函数,和提供一个在请求队列上的自旋锁。

```
request_queue_t *       //请求队列初始化函数返回指向成功初始化后的请求队列指针
blk_init_queue(request_fn_proc *, //设备驱动程序提供的请求函数
```

spinlock_t *); //自旋锁保证请求函数在上下文中是一个原子操作

要从请求队列中获取一个未完成的 I/O 请求可以使用定义在<linux/elevator.h>中的函数:

struct request * elv_next_request(struct request_queue *q);

该函数从队列中返回第一个未完成的 request 请求;当没有请求要处理时返回 NULL,但不删除队列中的 request。

在块设备请求队列中可能存在一些非文件读写的数据,例如请求一些设备的控制和诊断的信息,判断一个请求是否是文件数据请求,可以使用定义在<linux/blhdev.h>中的宏:

blk_fs_request(rq)

返回非 0 表示是一个文件数据请求,返回 0 表示不是一个文件数据请求。

某些非文件读写的数据可以使用定义在<linux/blhdev.h>中的函数 end_request 确定是否实际传输它:

void end_request(struct request *req, int uptodate);

uptodate 为 0,表示忽略这些数据;1 表示传输这些数据请求。

request 机制采用电梯算法响应并发到达的 I/O 请求,对于处理大量磁盘连续读写的数据访问是非常高效的。

2.9.5 一个简化了的 RAM 盘块设备驱动 sbull

Sbull 程序仅采用请求队列方式操作数据访问。但通过它可以看出块设备驱动程序最基本的内部工作过程，和怎样实现了一个 RAM 盘的设备驱动。通过测试和使用这个块设备的工作可进一步分析一下有关块设备驱动有关的数据结构和操作方法,了解怎样在内核中编写一个特定的块设备驱动的基本方法和过程。

头文件 sbull.h:

```
#include <linux/ioctl.h>
/* Multiqueue only works on 2.4 */
#ifdef SBULL_MULTIQUEUE
#     warning "Multiqueue only works on 2.4 kernels"
#endif
/*
 * Macros to help debugging
 */
#undef PDEBUG                 /* undef it, just in case */
#ifdef SBULL_DEBUG
#   ifdef __KERNEL__
      /* This one if debugging is on, and kernel space */
#     define PDEBUG(fmt, args...) printk( KERN_DEBUG "sbull: " fmt, ## args)
#   else
      /* This one for user space */
#     define PDEBUG(fmt, args...) fprintf(stderr, fmt, ## args)
```

```
#   endif
#else
#   define PDEBUG(fmt, args...) /* not debugging: nothing */
#endif
#undef PDEBUGG
#define PDEBUGG(fmt, args...) /* nothing: it's a placeholder */
#define SBULL_MAJOR 0            /* dynamic major by default */
#define SBULL_DEVS 2            /* two disks */
#define SBULL_RAHEAD 2           /* two sectors */
#define SBULL_SIZE 2048         /* two megs each */
                                        -4-
#define SBULL_BLKSIZE 1024      /* 1k blocks */
#define SBULL_HARDSECT 512      /* 2.2 and 2.4 can used different values */
#define SBULLR_MAJOR 0           /* Dynamic major for raw device */
/*
  * The sbull device is removable: if it is left closed for more than
  * half a minute, it is removed. Thus use a usage count and a
  * kernel timer
  */
typedef struct Sbull_Dev {
     int size;
     int usage;
     struct timer_list timer;
     spinlock_t lock;
     u8 *data;
#ifdef SBULL_MULTIQUEUE
     request_queue_t *queue;
     int busy;
#endif
}
/*************************************************************************/
```

主程序 sbull.c

```
/*
  * Sample disk driver, from the beginning.
  */
#include <linux/config.h>
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/init.h>
#include <linux/sched.h>
#include <linux/kernel.h>     /* printk() */
#include <linux/slab.h>       /* kmalloc() */
#include <linux/fs.h>         /* everything... */
#include <linux/errno.h>      /* error codes */
```

```
#include <linux/timer.h>
#include <linux/types.h>      /* size_t */

#include <linux/fcntl.h>    /* O_ACCMODE */
#include <linux/hdreg.h>    /* HDIO_GETGEO */
#include <linux/kdev_t.h>
#include <linux/vmalloc.h>
#include <linux/genhd.h>
#include <linux/blkdev.h>
#include <linux/buffer_head.h>      /* invalidate_bdev */
#include <linux/bio.h>
MODULE_LICENSE("Dual BSD/GPL");
static int sbull_major = 0;
module_param(sbull_major, int, 0);
static int hardsect_size = 512;
module_param(hardsect_size, int, 0);
static int nsectors = 102400;       /* How big the drive is */
module_param(nsectors, int, 0);
static int ndevices = 4;
module_param(ndevices, int, 0);
/*
 * The different "request modes" we can use.
 */
enum {
       RM_SIMPLE   = 0,      /* The extra-simple request function */
       RM_FULL     = 1,      /* The full-blown version */
       RM_NOQUEUE = 2,       /* Use make_request */
};
static int request_mode = RM_SIMPLE;
module_param(request_mode, int, 0);
/*
 * Minor number and partition management.
 */
#define SBULL_MINORS        16
#define MINOR_SHIFT 4
#define DEVNUM(kdevnum)     (MINOR(kdev_t_to_nr(kdevnum)) >> MINOR_SHIFT
/*
 * We can tweak our hardware sector size, but the kernel talks to us
 * in terms of small sectors, always.
 */
#define KERNEL_SECTOR_SIZE          512

/*
  * After this much idle time, the driver will simulate a media change.
```

```
  */
#define INVALIDATE_DELAY 30*HZ
/*
  * The internal representation of our device.
  */
struct sbull_dev {
          int size;                       /* Device size in sectors */
          u8 *data;                        /* The data array */
          short users;                    /* How many users */
          short media_change;              /* Flag a media change? */
          spinlock_t lock;                 /* For mutual exclusion */
          struct request_queue *queue;    /* The device request queue */
          struct gendisk *gd;             /* The gendisk structure */
          struct timer_list timer;        /* For simulated media changes */
};
static struct sbull_dev *Devices = NULL;
/*
  * Handle an I/O request.
  */
static void sbull_transfer(struct sbull_dev *dev, unsigned long sector,
               unsigned long nsect, char *buffer, int write)
{
        unsigned long offset = sector*KERNEL_SECTOR_SIZE;
        unsigned long nbytes = nsect*KERNEL_SECTOR_SIZE;
        if ((offset + nbytes) > dev->size) {
               printk (KERN_NOTICE "Beyond-end write (%ld %ld)\n", offset, nbytes);
               return;
        }
        if (write)
               memcpy(dev->data + offset, buffer, nbytes);
        else
               memcpy(buffer, dev->data + offset, nbytes);
}
/*
  * The simple form of the request function.
  */
static void sbull_request(request_queue_t *q)
{
        struct request *req;
        while ((req = elv_next_request(q)) != NULL) {
               struct sbull_dev *dev = req->rq_disk->private_data;
               if (! blk_fs_request(req)) {
                       printk (KERN_NOTICE "Skip non-fs request\n");
                       end_request(req, 0);
```

```
                        continue;
                }
        //      printk (KERN_NOTICE "Req dev %d dir %ld sec %ld, nr %d f %lx\n",
        //                      dev - Devices, rq_data_dir(req),
        //                      req->sector, req->current_nr_sectors,
        //                      req->flags);
                sbull_transfer(dev, req->sector, req->current_nr_sectors,
                                req->buffer, rq_data_dir(req));
                end_request(req, 1);
        }
}
/*
  * Transfer a single BIO.
  */
static int sbull_xfer_bio(struct sbull_dev *dev, struct bio *bio)
{
        int i;
        struct bio_vec *bvec;
        sector_t sector = bio->bi_sector;
        /* Do each segment independently. */
        bio_for_each_segment(bvec, bio, i) {
                char *buffer = __bio_kmap_atomic(bio, i, KM_USER0);
                sbull_transfer(dev, sector, bio_cur_sectors(bio),
                                buffer, bio_data_dir(bio) == WRITE);
                sector += bio_cur_sectors(bio);
                __bio_kunmap_atomic(bio, KM_USER0);
        }
        return 0; /* Always "succeed" */
}

/*
  * Transfer a full request.
  */
static int sbull_xfer_request(struct sbull_dev *dev, struct request *req)
{
        struct bio *bio;
        int nsect = 0;
        rq_for_each_bio(bio, req) {
                sbull_xfer_bio(dev, bio);
                nsect += bio->bi_size/KERNEL_SECTOR_SIZE;
        }
        return nsect;
}
/*
```

```
  * Smarter request function that "handles clustering".
  */
static void sbull_full_request(request_queue_t *q)
{
        struct request *req;
        int sectors_xferred;
        struct sbull_dev *dev = q->queuedata;
        while ((req = elv_next_request(q)) != NULL) {
               if (! blk_fs_request(req)) {
                       printk (KERN_NOTICE "Skip non-fs request\n");
                       end_request(req, 0);
                       continue;
               }
               sectors_xferred = sbull_xfer_request(dev, req);
               if (! end_that_request_first(req, 1, sectors_xferred)) {
                       blkdev_dequeue_request(req);
                       end_that_request_last(req,1);
               }
        }
}
/*
  * The direct make request version.
  */
static int sbull_make_request(request_queue_t *q, struct bio *bio)
{
        struct sbull_dev *dev = q->queuedata;
        int status;
        status = sbull_xfer_bio(dev, bio);
        bio_endio(bio, bio->bi_size, status);
        return 0;
}
/*
  * Open and close.
  */
static int sbull_open(struct inode *inode, struct file *filp)
{
        struct sbull_dev *dev = inode->i_bdev->bd_disk->private_data;
        del_timer_sync(&dev->timer);
        filp->private_data = dev;
        spin_lock(&dev->lock);
        if (! dev->users)
                check_disk_change(inode->i_bdev);
        dev->users++;
        spin_unlock(&dev->lock);
```

```
        return 0;
}
static int sbull_release(struct inode *inode, struct file *filp)
{
        struct sbull_dev *dev = inode->i_bdev->bd_disk->private_data;
        spin_lock(&dev->lock);
        dev->users--;
        if (!dev->users) {
                dev->timer.expires = jiffies + INVALIDATE_DELAY;
                add_timer(&dev->timer);
        }
        spin_unlock(&dev->lock);

        return 0;
}
/*
  * Look for a (simulated) media change.
  */
int sbull_media_changed(struct gendisk *gd)
{
        struct sbull_dev *dev = gd->private_data;
        return dev->media_change;
}
/*
  * Revalidate.   WE DO NOT TAKE THE LOCK HERE, for fear of deadlocking
  * with open.   That needs to be reevaluated.
  */
int sbull_revalidate(struct gendisk *gd)
{
        struct sbull_dev *dev = gd->private_data;
        if (dev->media_change) {
                dev->media_change = 0;
                memset (dev->data, 0, dev->size);
        }
        return 0;
}
/*
  * The "invalidate" function runs out of the device timer; it sets
  * a flag to simulate the removal of the media.
  */
void sbull_invalidate(unsigned long ldev)
{
        struct sbull_dev *dev = (struct sbull_dev *) ldev;
        spin_lock(&dev->lock);
```

```
        if (dev->users || !dev->data)
                printk (KERN_WARNING "sbull: timer sanity check failed\n");
        else
                dev->media_change = 1;
        spin_unlock(&dev->lock);

}
/*
  * The ioctl() implementation
  */
int sbull_ioctl (struct inode *inode, struct file *filp,
                     unsigned int cmd, unsigned long arg)
{
        long size;
        struct hd_geometry geo;
        struct sbull_dev *dev = filp->private_data;
        switch(cmd) {
             case HDIO_GETGEO:
                /*
                 * Get geometry: since we are a virtual device, we have to make
                 * up something plausible.   So we claim 16 sectors, four heads,
                 * and calculate the corresponding number of cylinders.    We set
the
                 * start of data at sector four.
                 */
                size = dev->size*(hardsect_size/KERNEL_SECTOR_SIZE);
                geo.cylinders = (size & ~0x3f) >> 6;
                geo.heads = 4;
                geo.sectors = 16;
                geo.start = 4;
                if (copy_to_user((void __user *) arg, &geo, sizeof(geo)))
                        return -EFAULT;
                return 0;
        }
        return -ENOTTY; /* unknown command */
}
/*
  * The device operations structure.
  */
static struct block_device_operations sbull_ops = {
        .owner              = THIS_MODULE,
        .open               = sbull_open,
        .release            = sbull_release,
        .media_changed      = sbull_media_changed,
```

```
        .revalidate_disk = sbull_revalidate,
        .ioctl             = sbull_ioctl
};
/*
  * Set up our internal device.
  */
static void setup_device(struct sbull_dev *dev, int which)
{
        /*
          * Get some memory.
          */
        memset (dev, 0, sizeof (struct sbull_dev));
        dev->size = nsectors*hardsect_size;
        dev->data = vmalloc(dev->size);
        if (dev->data == NULL) {
                printk (KERN_NOTICE "vmalloc failure.\n");
                return;
        }
        spin_lock_init(&dev->lock);
        /*
          * The timer which "invalidates" the device.
          */
        init_timer(&dev->timer);
        dev->timer.data = (unsigned long) dev;
        dev->timer.function = sbull_invalidate;
        /*
          * The I/O queue, depending on whether we are using our own
          * make_request function or not.
          */
        switch (request_mode) {
             case RM_NOQUEUE:
                dev->queue = blk_alloc_queue(GFP_KERNEL);
                if (dev->queue == NULL)
                        goto out_vfree;
                blk_queue_make_request(dev->queue, sbull_make_request);
                break;
             case RM_FULL:
               dev->queue = blk_init_queue(sbull_full_request, &dev->lock);
               if (dev->queue == NULL)
                      goto out_vfree;
               break;
            default:
               printk(KERN_NOTICE "Bad request mode %d, using simple\n",
request_mode);
```

```
                /* fall into.. */
            case RM_SIMPLE:
                dev->queue = blk_init_queue(sbull_request, &dev->lock);
                if (dev->queue == NULL)
                        goto out_vfree;
                break;
        }
        blk_queue_hardsect_size(dev->queue, hardsect_size);
        dev->queue->queuedata = dev;
        /*
          * And the gendisk structure.
          */
        dev->gd = alloc_disk(SBULL_MINORS);
        if (! dev->gd) {
                printk (KERN_NOTICE "alloc_disk failure\n");
                goto out_vfree;
        }
        dev->gd->major = sbull_major;
        dev->gd->first_minor = which*SBULL_MINORS;
        dev->gd->fops = &sbull_ops;
        dev->gd->queue = dev->queue;
        dev->gd->private_data = dev;
        snprintf (dev->gd->disk_name, 32, "sbull%c", which + 'a');
        set_capacity(dev->gd, nsectors*(hardsect_size/KERNEL_SECTOR_SIZE));
        add_disk(dev->gd);
        return;
  out_vfree:
        if (dev->data)
                vfree(dev->data);
}
static int __init sbull_init(void)
{
        int i;
        /*
          * Get registered.
          */
        sbull_major = register_blkdev(sbull_major, "sbull");
        if (sbull_major <= 0) {
                printk(KERN_WARNING "sbull: unable to get major number\n");
                return -EBUSY;
        }
        /*
          * Allocate the device array, and initialize each one.
          */
```

```
        Devices = kmalloc(ndevices*sizeof (struct sbull_dev), GFP_KERNEL);
        if (Devices == NULL)
                goto out_unregister;
        for (i = 0; i < ndevices; i++)
                setup_device(Devices + i, i);
        return 0;
  out_unregister:
        unregister_blkdev(sbull_major, "sbd");
        return -ENOMEM;
}
static void sbull_exit(void)
{
        int i;
        for (i = 0; i < ndevices; i++) {
                struct sbull_dev *dev = Devices + i;
                del_timer_sync(&dev->timer);
                if (dev->gd) {
                        del_gendisk(dev->gd);
                        put_disk(dev->gd);
                }
                if (dev->queue) {
                        if (request_mode == RM_NOQUEUE)
                                blk_put_queue(dev->queue);
                        else
                                blk_cleanup_queue(dev->queue);
                 }
                 if (dev->data)
                         vfree(dev->data);
         }
         unregister_blkdev(sbull_major, "sbull");
         kfree(Devices);
}
module_init(sbull_init);
module_exit(sbull_exit);

/*************************************************************************/
```

加载块设备驱动,安装块设备文件节点的脚本程序 sbull_load

```
#!/bin/sh
function make_minors {
     let part=1
     while (($part < $minors)); do
        let minor=$part+$2
        mknod $1$part b $major $minor
```

```
        let part=$part+1
    done
}
# FIXME: This isn't handling minors (partitions) at all.
module="sbull"
device="sbull"
mode="664"
chardevice="sbullr"
minors=16
# Group: since distributions do it differently, look for wheel or use staff
if grep '^staff:' /etc/group > /dev/null; then
    group="staff"
else
    group="wheel"
fi
# invoke insmod with all arguments we got
# and use a pathname, as newer modutils don't look in . by default
/sbin/insmod -f ./$module.ko $* || exit 1
major=`cat /proc/devices | awk "\\$2==\"$module\" {print \\$1}"`
# Remove stale nodes and replace them, then give gid and perms
rm -f /dev/${device}[a-d]* /dev/${device}
mknod /dev/${device}a b $major 0
make_minors /dev/${device}a 0
mknod /dev/${device}b b $major 16
make_minors /dev/${device}b 16
mknod /dev/${device}c b $major 32
make_minors /dev/${device}c 32
mknod /dev/${device}d b $major 48
make_minors /dev/${device}d 48
ln -sf ${device}a /dev/${device}
chgrp $group /dev/${device}[a-d]*
chmod $mode    /dev/${device}[a-d]*
/*****************************************************************************/
```

卸载块设备驱动和块设备文件节点的脚本程序 sbull_unload

```
#!/bin/sh
module="sbull"
device="sbull"
# invoke rmmod with all arguments we got
/sbin/rmmod $module $* || exit 1
# Remove stale nodes
rm -f /dev/${device}[a-d]* /dev/${device}
/*****************************************************************************/
```

块设备驱动的 Makefile 文件

```
# Comment/uncomment the following line to disable/enable debugging
#DEBUG = y
# Add your debugging flag (or not) to CFLAGS
ifeq ($(DEBUG),y)
   DEBFLAGS = -O -g -DSBULL_DEBUG # "-O" is needed to expand inlines
else
   DEBFLAGS = -O2
endif
CFLAGS += $(DEBFLAGS)
CFLAGS += -I..
ifneq ($(KERNELRELEASE),)
# call from kernel build system
obj-m := sbull.o
else
KERNELDIR ?= /lib/modules/$(shell uname -r)/build
PWD       := $(shell pwd)
default:
        $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif
clean:
        rm -rf *.o *~ core .depend .*.cmd *.ko *.mod.c .tmp_versions
depend .depend dep:
        $(CC) $(CFLAGS) -M *.c > .depend
ifeq (.depend,$(wildcard .depend))
include .depend
endif
```

2.9.6 实验问题

问题 A

分析调试以上 sbull 设备驱动，并用块设备有关的系统命令测试其功能，进而解析块设备 I/O 驱动的构造和基本工作过程。

问题 B

对于随机访问的块设备(如 flash 内存,RAM 盘等)使用请求队列并不能提高性能。对于这些设备最好不用请求队列而是直接使用 bio 操作内存物理页进行数据传输。分析本节 sbull 设备驱动程序，看它是否提供了随机访问方式，如果提供了这种方式，请编程测试和分析各种请求方式的性能有何不同。你能在驱动程序中一些关键点上加入定时器采集到不同方式中进行大数据量传输时花费的微妙级的时间吗？

参考教材:

[1] Linux 内核设计与实现 RoberLove 著 陈莉君 等译
[2] 深入理解 Linux 内核 Daniel P. Bovet 著 陈莉君 等译
[3] Linux 设备驱动程序 Jonathan Corbet, 著 魏永明 等译
[4] Linux 操作系统内核实习 Gary Nutt 著 潘登 等译