目　录

深入浅出 Linux 设备驱动编程

宋宝华 21cnbao@21cn.com

1.引言

目前，Linux 软件工程师大致可分为两个层次：

（1）Linux 应用软件工程师（Application Software Engineer）：主要利用 C 库函数和 Linux API 进行应用软件的编写；

（2）Linux 固件工程师（Firmware Engineer）：主要进行 Bootloader、Linux 的移植及 Linux 设备驱动程序的设计。

一般而言，固件工程师的要求要高于应用软件工程师的层次，而其中的 Linux 设备驱动编程又是 Linux 程序设计中比较复杂的部分，究其原因，主要包括如下几个方面：

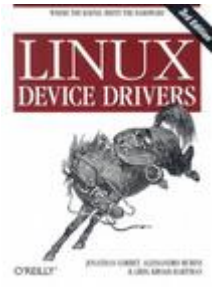
（1）设备驱动属于 Linux 内核的部分，编写 Linux 设备驱动需要有一定的 Linux 操作系统内核基础；

（2）编写 Linux 设备驱动需要对硬件的原理有相当的了解，大多数情况下我们是针对一个特定的嵌入式硬件平台编写驱动的；

（3）Linux 设备驱动中广泛涉及到多进程并发的同步、互斥等控制，容易出现 bug；

（4）由于属于内核的一部分，Linux 设备驱动的调试也相当复杂。

目前，市面上的 Linux 设备驱动程序参考书籍非常稀缺，少有的经典是由 Linux 社区的三位领导者 Jonathan Corbet、Alessandro Rubini、Greg Kroah-Hartman 编写的《Linux Device Drivers》（目前该书已经出版到第 3 版，中文译本由中国电力出版社出版）。该书将 Linux 设备驱动编写技术进行了较系统的展现，但是该书所列举实例的背景过于复杂，使得读者需要将过多的精力投放于对例子背景的理解上，很难完全集中精力于 Linux 驱动程序本身。往往需要将此书翻来覆去地研读许多遍，才能有较深的体会。



（《Linux Device Drivers》中英文版封面）

本文将仍然秉承《Linux Device Drivers》一书以实例为主的风格，但是实例的背景将非常简单，以求使读者能将集中精力于 Linux 设备驱动本身，理解 Linux 内核模块、Linux 设备驱动的结构、Linux 设备驱动中的并发控制等内容。另外，与《Linux Device Drivers》所不同的是，针对设备驱动的实例，本文还给出了用户态的程序来访问该设备，展现设备驱动的运行情况及用户态和内核态的交互。相信阅读完本文将为您领悟《Linux Device Drivers》一书中的内容打下很好的基础。

本文中的例程除引用的以外皆由笔者亲自调试通过，主要基于的内核版本为 Linux 2.4，例子要在其他内核上运行只需要做少量的修改。

构建本文例程运行平台的一个较好方法是：在 Windows 平台上安装 VMWare 虚拟机，并在 VMWare 虚拟机上安装 Red Hat。注意安装的过程中应该选中“开发工具”和“内核开发”二项（如果本文的例程要在特定的嵌入式系统中运行，还应安装相应的交叉编译器，并

包含相应的 Linux 源代码），如下图：

2.Linux 内核模块

Linux 设备驱动属于内核的一部分，Linux 内核的一个模块可以以两种方式被编译和加载：

（1）直接编译进 Linux 内核，随同 Linux 启动时加载；

（2）编译成一个可加载和删除的模块，使用 insmod 加载（modprobe 和 insmod 命令类似，但依赖于相关的配置文件），rmmod 删除。这种方式控制了内核的大小，而模块一旦被插入内核，它就和内核其他部分一样。

下面我们给出一个内核模块的例子：

```
#include <linux/module.h>    //所有模块都需要的头文件
#include <linux/init.h>          // init&exit 相关宏
MODULE_LICENSE("GPL");

static int __init hello_init (void)
{
   printk("Hello module init\n");
   return 0;
}

static void __exit hello_exit (void)
{
   printk("Hello module exit\n");
```

```
}

module_init(hello_init);
module_exit(hello_exit);
```

分析上述程序，发现一个 Linux 内核模块需包含模块初始化和模块卸载函数，前者在 insmod 的时候运行，后者在 rmmod 的时候运行。初始化与卸载函数必须在宏 module_init 和 module_exit 使用前定义，否则会出现编译错误。

程序中的 MODULE_LICENSE("GPL")用于声明模块的许可证。

如果要把上述程序编译为一个运行时加载和删除的模块，则编译命令为：

```
gcc –D__KERNEL__ -DMODULE –DLINUX –I /usr/local/src/linux2.4/include -c –o hello.o hello.c
```

由此可见，Linux 内核模块的编译需要给 gcc 指示–D__KERNEL__ -DMODULE –DLINUX 参数。-I 选项跟着 Linux 内核源代码中 Include 目录的路径。

下列命令将可加载 hello 模块：

```
insmod ./hello.o
```

下列命令完成相反过程：

```
rmmod hello
```

如果要将其直接编译入 Linux 内核，则需要将源代码文件拷贝入 Linux 内核源代码的相应路径里，并修改 Makefile。

我们有必要补充一下 Linux 内核编程的一些基本知识：

内存

在 Linux 内核模式下，我们不能使用用户态的 malloc()和 free()函数申请和释放内存。进行内核编程时，最常用的内存申请和释放函数为在 include/linux/kernel.h 文件中声明的 kmalloc()和 kfree()，其原型为：

```
void *kmalloc(unsigned int len, int priority);
void kfree(void *__ptr);
```

kmalloc 的 priority 参数通常设置为 GFP_KERNEL，如果在中断服务程序里申请内存则要用 GFP_ATOMIC 参数，因为使用 GFP_KERNEL 参数可能会引起睡眠，不能用于非进程上下文中（在中断中是不允许睡眠的）。

由于内核态和用户态使用不同的内存定义，所以二者之间不能直接访问对方的内存。而应该使用 Linux 中的用户和内核态内存交互函数（这些函数在 include/asm/uaccess.h 中被声明）：

```
unsigned long copy_from_user(void *to, const void *from, unsigned long n);
unsigned long copy_to_user (void * to, void * from, unsigned long len);
```

copy_from_user、copy_to_user 函数返回不能被复制的字节数，因此，如果完全复制成功，返回值为 0。

include/asm/uaccess.h 中定义的 put_user 和 get_user 用于内核空间和用户空间的单值交互（如 char、int、long）。

这里给出的仅仅是关于内核中内存管理的皮毛，关于 Linux 内存管理的更多细节知识，我们会在本文第 9 节《内存与 I/O 操作》进行更加深入地介绍。

输出

在内核编程中，我们不能使用用户态 C 库函数中的 printf()函数输出信息，而只能使用 printk()。但是，内核中 printk()函数的设计目的并不是为了和用户交流，它实际上是内核的一种日志机制，用来记录下日志信息或者给出警告提示。

每个 printk 都会有个优先级，内核一共有 8 个优先级，它们都有对应的宏定义。如果未指定优先级，内核会选择默认的优先级 DEFAULT_MESSAGE_LOGLEVEL。如果优先级数字比 int console_loglevel 变量小的话，消息就会打印到控制台上。如果 syslogd 和 klogd 守护进程在运行的话，则不管是否向控制台输出，消息都会被追加进/var/log/messages 文件。klogd 只处理内核消息，syslogd 处理其他系统消息，比如应用程序。

模块参数

2.4 内核下，include/linux/module.h 中定义的宏 MODULE_PARM(var,type) 用于向模块传递命令行参数。var 为接受参数值的变量名，type 为采取如下格式的字符串[min[-max]]{b,h,i,l,s}。min 及 max 用于表示当参数为数组类型时，允许输入的数组元素的个数范围；b：byte；h：short；i：int；l：long；s：string。

在装载内核模块时，用户可以向模块传递一些参数：

```
insmod modname var=value
```

如果用户未指定参数，var 将使用模块内定义的缺省值。

3.字符设备驱动程序

Linux 下的设备驱动程序被组织为一组完成不同任务的函数的集合，通过这些函数使得 Windows 的设备操作犹如文件一般。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，如 open ()、close ()、read ()、write () 等。

Linux 主要将设备分为二类：字符设备和块设备。字符设备是指设备发送和接收数据以字符的形式进行；而块设备则以整个数据缓冲区的形式进行。字符设备的驱动相对比较简单。

下面我们来假设一个非常简单的虚拟字符设备：这个设备中只有一个 4 个字节的全局变量 int global_var，而这个设备的名字叫做“gobalvar”。对“gobalvar”设备的读写等操作即是对其中全局变量 global_var 的操作。

驱动程序是内核的一部分，因此我们需要给其添加模块初始化函数，该函数用来完成对所控设备的初始化工作，并调用 register_chrdev() 函数注册字符设备：

```
static int __init gobalvar_init(void)
{
  if (register_chrdev(MAJOR_NUM, " gobalvar ", &gobalvar_fops))
     {
     //…注册失败
     }
  else
     {
     //…注册成功
     }
}
```

其中，register_chrdev 函数中的参数 MAJOR_NUM 为主设备号,“gobalvar”为设备名，gobalvar_fops 为包含基本函数入口点的结构体，类型为 file_operations。当 gobalvar 模块被加载时，gobalvar_init 被执行，它将调用内核函数 register_chrdev，把驱动程序的基本入口点指针存放在内核的字符设备地址表中，在用户进程对该设备执行系统调用时提供入口地址。

与模块初始化函数对应的就是模块卸载函数，需要调用 register_chrdev()的“反函数”unregister_chrdev()：

```
static void __exit gobalvar_exit(void)
```

```
{
  if (unregister_chrdev(MAJOR_NUM, " gobalvar "))
    {
    //…卸载失败
    }
  else
    {
    //…卸载成功
    }
}
```

随着内核不断增加新的功能，file_operations 结构体已逐渐变得越来越大，但是大多数的驱动程序只是利用了其中的一部分。对于字符设备来说，要提供的主要入口有：open ()、release ()、read ()、write ()、ioctl ()、llseek()、poll()等。

open()函数 对设备特殊文件进行 open()系统调用时，将调用驱动程序的 open () 函数：

```
int (*open)(struct inode * ,struct file *);
```

其中参数 inode 为设备特殊文件的 inode (索引结点) 结构的指针，参数 file 是指向这一设备的文件结构的指针。open()的主要任务是确定硬件处在就绪状态、验证次设备号的合法性(次设备号可以用 MINOR(inode-> i - rdev) 取得)、控制使用设备的进程数、根据执行情况返回状态码(0 表示成功，负数表示存在错误) 等；

release()函数 当最后一个打开设备的用户进程执行 close ()系统调用时，内核将调用驱动程序的 release () 函数：

```
void (*release) (struct inode * ,struct file *) ;
```

release 函数的主要任务是清理未结束的输入/输出操作、释放资源、用户自定义排他标志的复位等。

read()函数 当对设备特殊文件进行 read() 系统调用时，将调用驱动程序 read() 函数：

```
ssize_t (*read) (struct file *, char *, size_t, loff_t *);
```

用来从设备中读取数据。当该函数指针被赋为 NULL 值时，将导致 read 系统调用出错并返回-EINVAL（“Invalid argument，非法参数”）。函数返回非负值表示成功读取的字节数（返回值为“signed size”数据类型，通常就是目标平台上的固有整数类型）。

globalvar_read 函数中内核空间与用户空间的内存交互需要借助第 2 节所介绍的函数：

```
static ssize_t globalvar_read(struct file *filp, char *buf, size_t len, loff_t *off)
{
  …
  copy_to_user(buf, &global_var, sizeof(int));
  …
}
```

write( ) 函数 当设备特殊文件进行 write () 系统调用时，将调用驱动程序的 write () 函数：

```
ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
```

向设备发送数据。如果没有这个函数，write 系统调用会向调用程序返回一个-EINVAL。如果返回值非负，则表示成功写入的字节数。

globalvar_write 函数中内核空间与用户空间的内存交互需要借助第 2 节所介绍的函数：

```
static ssize_t globalvar_write(struct file *filp, const char *buf, size_t len, loff_t
  *off)
```

```
{
  …
  copy_from_user(&global_var, buf, sizeof(int));
  …
}
```

ioctl() 函数 该函数是特殊的控制函数，可以通过它向设备传递控制信息或从设备取得状态信息，函数原型为：

```
int (*ioctl) (struct inode * ,struct file * ,unsigned int ,unsigned long);
```

unsigned int 参数为设备驱动程序要执行的命令的代码，由用户自定义，unsigned long 参数为相应的命令提供参数，类型可以是整型、指针等。如果设备不提供 ioctl 入口点，则对于任何内核未预先定义的请求，ioctl 系统调用将返回错误（-ENOTTY，“No such ioctl fordevice，该设备无此 ioctl 命令”）。如果该设备方法返回一个非负值，那么该值会被返回给调用程序以表示调用成功。

llseek()函数 该函数用来修改文件的当前读写位置，并将新位置作为（正的）返回值返回，原型为：

```
loff_t (*llseek) (struct file *, loff_t, int);
```

poll()函数 poll 方法是 poll 和 select 这两个系统调用的后端实现，用来查询设备是否可读或可写，或是否处于某种特殊状态，原型为：

```
unsigned int (*poll) (struct file *, struct poll_table_struct *);
```

我们将在“设备的阻塞与非阻塞操作”一节对该函数进行更深入的介绍。

设备“gobalvar”的驱动程序的这些函数应分别命名为 gobalvar_open、gobalvar_ release、gobalvar_read、gobalvar_write、gobalvar_ioctl，因此设备“gobalvar”的基本入口点结构变量 gobalvar_fops 赋值如下：

```
struct file_operations gobalvar_fops = {
read: gobalvar_read,
write: gobalvar_write,
};
```

上述代码中对 gobalvar_fops 的初始化方法并不是标准 C 所支持的，属于 GNU 扩展语法。

完整的 globalvar.c 文件源代码如下：

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <asm/uaccess.h>

MODULE_LICENSE("GPL");

#define MAJOR_NUM    254     //主设备号

static ssize_t globalvar_read(struct file *, char *, size_t, loff_t*);
static ssize_t globalvar_write(struct file *, const char *, size_t, loff_t*);

//初始化字符设备驱动的 file_operations 结构体
struct file_operations globalvar_fops =
```

```
{
    read: globalvar_read, write: globalvar_write,
};
static int global_var = 0;      //“globalvar”设备的全局变量

static int __init globalvar_init(void)
{
    int ret;

    //注册设备驱动
    ret = register_chrdev(MAJOR_NUM, "globalvar", &globalvar_fops);
    if (ret)
    {
        printk("globalvar register failure");
    }
    else
    {
        printk("globalvar register success");
    }
    return ret;
}

static void __exit globalvar_exit(void)
{
    int ret;

    //注销设备驱动
    ret = unregister_chrdev(MAJOR_NUM, "globalvar");
    if (ret)
    {
        printk("globalvar unregister failure");
    }
    else
    {
        printk("globalvar unregister success");
    }
}

static ssize_t globalvar_read(struct file *filp, char *buf, size_t len, loff_t *off)
{
    //将 global_var 从内核空间复制到用户空间
    if (copy_to_user(buf, &global_var, sizeof(int)))
    {
        return    - EFAULT;
```

```
    }
    return sizeof(int);
}

static ssize_t globalvar_write(struct file *filp, const char *buf, size_t len, loff_t
    *off)
{
    //将用户空间的数据复制到内核空间的 global_var
    if (copy_from_user(&global_var, buf, sizeof(int)))
    {
        return    - EFAULT;
    }
    return sizeof(int);
}

module_init(globalvar_init);
module_exit(globalvar_exit);
```

运行

```
gcc –D__KERNEL__ -DMODULE –DLINUX –I /usr/local/src/linux2.4/include -c –o
globalvar.o globalvar.c
```

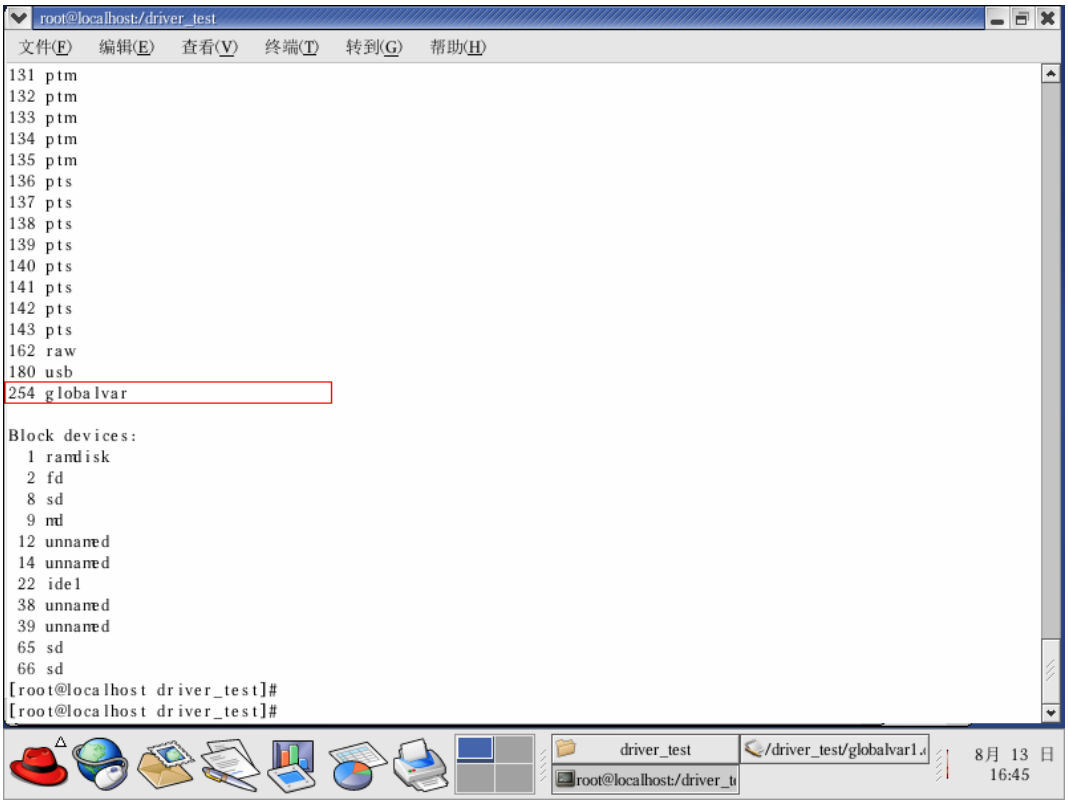
编译代码，运行

```
inmod globalvar.o
```

加载 globalvar 模块，再运行

```
cat /proc/devices
```

发现其中多出了“254 globalvar”一行，如下图：



接着我们可以运行：

```
mknod /dev/globalvar c 254 0
```

创建设备节点，用户进程通过/dev/globalvar 这个路径就可以访问到这个全局变量虚拟设备了。我们写一个用户态的程序 globalvartest.c 来验证上述设备：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>
main()
{
  int fd, num;
  //打开“/dev/globalvar”
  fd = open("/dev/globalvar", O_RDWR, S_IRUSR | S_IWUSR);
  if (fd != -1 )
  {
    //初次读 globalvar
     read(fd, &num, sizeof(int));
     printf("The globalvar is %d\n", num);

   //写 globalvar
     printf("Please input the num written to globalvar\n");
     scanf("%d", &num);
     write(fd, &num, sizeof(int));

   //再次读 globalvar
     read(fd, &num, sizeof(int));
     printf("The globalvar is %d\n", num);

     //关闭“/dev/globalvar”
     close(fd);
  }
  else
  {
     printf("Device open failure\n");
  }
}
```

编译上述文件：

```
gcc –o globalvartest.o globalvartest.c
```

运行

```
./globalvartest.o
```

可以发现“globalvar”设备可以正确的读写。

4.设备驱动中的并发控制

在驱动程序中，当多个线程同时访问相同的资源时（驱动程序中的全局变量是一种典型的共享资源），可能会引发“竞态”，因此我们必须对共享资源进行并发控制。Linux 内核中解决并发控制的最常用方法是自旋锁与信号量（绝大多数时候作为互斥锁使用）。

自旋锁与信号量“类似而不类”，类似说的是它们功能上的相似性，“不类”指代它们在本质和实现机理上完全不一样，不属于一类。

自旋锁不会引起调用者睡眠，如果自旋锁已经被别的执行单元保持，调用者就一直循环查看是否该自旋锁的保持者已经释放了锁，“自旋”就是“在原地打转”。而信号量则引起调用者睡眠，它把进程从运行队列上拖出去，除非获得锁。这就是它们的“不类”。

但是，无论是信号量，还是自旋锁，在任何时刻，最多只能有一个保持者，即在任何时刻最多只能有一个执行单元获得锁。这就是它们的“类似”。

鉴于自旋锁与信号量的上述特点，一般而言，自旋锁适合于保持时间非常短的情况，它可以在任何上下文使用；信号量适合于保持时间较长的情况，会只能在进程上下文使用。如果被保护的共享资源只在进程上下文访问，则可以以信号量来保护该共享资源，如果对共享资源的访问时间非常短，自旋锁也是好的选择。但是，如果被保护的共享资源需要在中断上下文访问（包括底半部即中断处理句柄和顶半部即软中断），就必须使用自旋锁。

与信号量相关的 API 主要有：

定义信号量

```
struct semaphore sem;
```

初始化信号量

```
void sema_init (struct semaphore *sem, int val);
```

该函数初始化信号量，并设置信号量 sem 的值为 val

```
void init_MUTEX (struct semaphore *sem);
```

该函数用于初始化一个互斥锁，即它把信号量 sem 的值设置为 1，等同于 sema_init (struct semaphore *sem, 1)；

```
void init_MUTEX_LOCKED (struct semaphore *sem);
```

该函数也用于初始化一个互斥锁，但它把信号量 sem 的值设置为 0，等同于 sema_init (struct semaphore *sem, 0)；

获得信号量

```
void down(struct semaphore * sem);
```

该函数用于获得信号量 sem，它会导致睡眠，因此不能在中断上下文使用；

```
int down_interruptible(struct semaphore * sem);
```

该函数功能与 down 类似，不同之处为，down 不能被信号打断，但 down_interruptible 能被信号打断；

```
int down_trylock(struct semaphore * sem);
```

该函数尝试获得信号量 sem，如果能够立刻获得，它就获得该信号量并返回 0，否则，返回非 0 值。它不会导致调用者睡眠，可以在中断上下文使用。

释放信号量

```
void up(struct semaphore * sem);
```

该函数释放信号量 sem，唤醒等待者。

与自旋锁相关的 API 主要有：

定义自旋锁

```
spinlock_t spin;
```

初始化自旋锁

spin_lock_init(lock)

该宏用于动态初始化自旋锁 lock

获得自旋锁

spin_lock(lock)

该宏用于获得自旋锁 lock，如果能够立即获得锁，它就马上返回，否则，它将自旋在那里，直到该自旋锁的保持者释放；

spin_trylock(lock)

该宏尝试获得自旋锁 lock，如果能立即获得锁，它获得锁并返回真，否则立即返回假，实际上不再“在原地打转”；

释放自旋锁

spin_unlock(lock)

该宏释放自旋锁 lock，它与 spin_trylock 或 spin_lock 配对使用；

除此之外，还有一组自旋锁使用于中断情况下的 API。

下面进入对并发控制的实战。首先，在 globalvar 的驱动程序中，我们可以通过信号量来控制对 int global_var 的并发访问，下面给出源代码：

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <asm/semaphore.h>

MODULE_LICENSE("GPL");

#define MAJOR_NUM    254

static ssize_t globalvar_read(struct file *, char *, size_t, loff_t*);
static ssize_t globalvar_write(struct file *, const char *, size_t, loff_t*);

struct file_operations globalvar_fops =
{
   read: globalvar_read, write: globalvar_write,
};
static int global_var = 0;
static struct semaphore sem;

static int __init globalvar_init(void)
{
   int ret;
   ret = register_chrdev(MAJOR_NUM, "globalvar", &globalvar_fops);
   if (ret)
   {
      printk("globalvar register failure");
   }
```

```
    else
    {
        printk("globalvar register success");
        init_MUTEX(&sem);
    }
    return ret;
}

static void __exit globalvar_exit(void)
{
    int ret;
    ret = unregister_chrdev(MAJOR_NUM, "globalvar");
    if (ret)
    {
        printk("globalvar unregister failure");
    }
    else
    {
        printk("globalvar unregister success");
    }
}

static ssize_t globalvar_read(struct file *filp, char *buf, size_t len, loff_t *off)
{
    //获得信号量
    if (down_interruptible(&sem))
    {
        return  - ERESTARTSYS;
    }

    //将 global_var 从内核空间复制到用户空间
    if (copy_to_user(buf, &global_var, sizeof(int)))
    {
        up(&sem);
        return  - EFAULT;
    }

    //释放信号量
    up(&sem);

    return sizeof(int);
}

ssize_t globalvar_write(struct file *filp, const char *buf, size_t len, loff_t
```

```
    *off)
{
    //获得信号量
    if (down_interruptible(&sem))
    {
        return  - ERESTARTSYS;

    }

    //将用户空间的数据复制到内核空间的 global_var
    if (copy_from_user(&global_var, buf, sizeof(int)))
    {
        up(&sem);
        return  - EFAULT;
    }

    //释放信号量
    up(&sem);

    return sizeof(int);
}

module_init(globalvar_init);
module_exit(globalvar_exit);
```

接下来，我们给 globalvar 的驱动程序增加 open()和 release()函数，并在其中借助自旋锁来保护对全局变量 int globalvar_count（记录打开设备的进程数）的访问来实现设备只能被一个进程打开（必须确保 globalvar_count 最多只能为 1）：

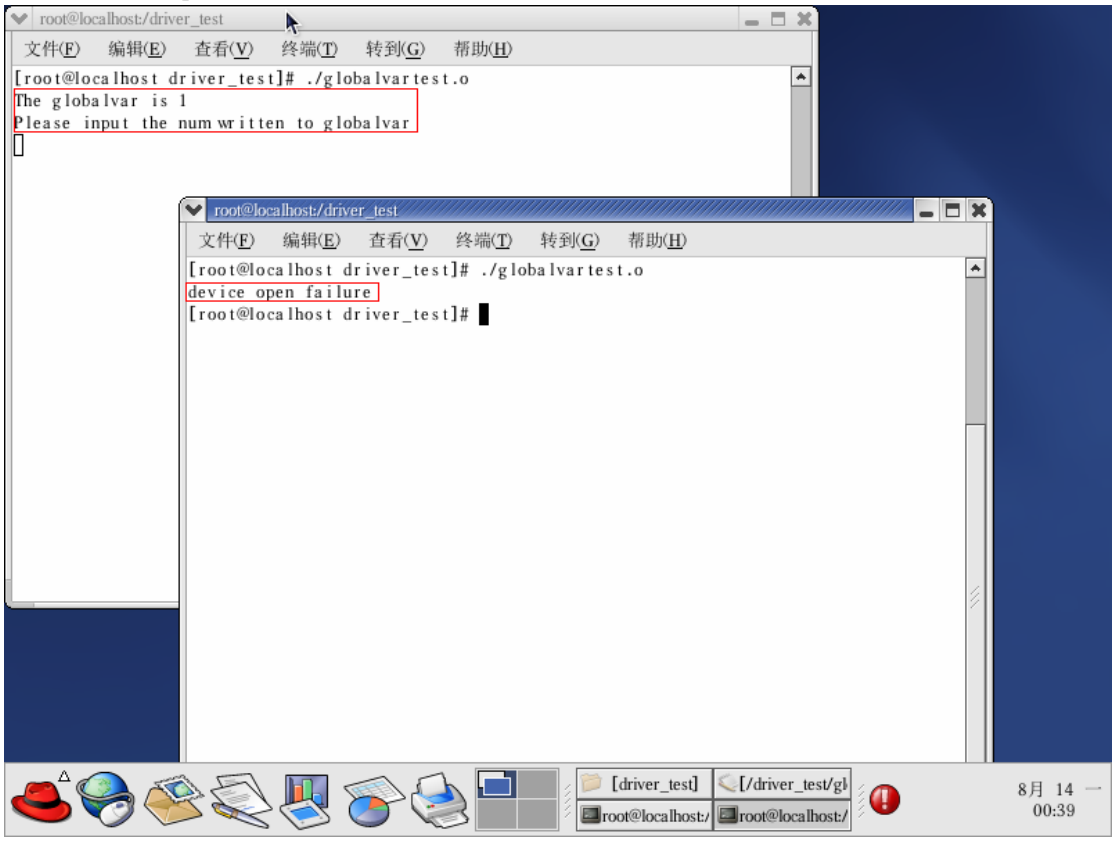
```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <asm/semaphore.h>

MODULE_LICENSE("GPL");

#define MAJOR_NUM  254

static ssize_t globalvar_read(struct file *, char *, size_t, loff_t*);
static ssize_t globalvar_write(struct file *, const char *, size_t, loff_t*);
static int globalvar_open(struct inode *inode, struct file *filp);
static int globalvar_release(struct inode *inode, struct file *filp);

struct file_operations globalvar_fops =
{
```

```
    read: globalvar_read, write: globalvar_write, open: globalvar_open, release:
        globalvar_release,
};

static int global_var = 0;
static int globalvar_count = 0;
static struct semaphore sem;
static spinlock_t spin = SPIN_LOCK_UNLOCKED;


static int __init globalvar_init(void)
{
    int ret;
    ret = register_chrdev(MAJOR_NUM, "globalvar", &globalvar_fops);
    if (ret)
    {
        printk("globalvar register failure");
    }
    else
    {
        printk("globalvar register success");
        init_MUTEX(&sem);
    }
    return ret;
}

static void __exit globalvar_exit(void)
{
    int ret;
    ret = unregister_chrdev(MAJOR_NUM, "globalvar");
    if (ret)
    {
        printk("globalvar unregister failure");
    }
    else
    {
        printk("globalvar unregister success");
    }
}

static int globalvar_open(struct inode *inode, struct file *filp)
{

    //获得自选锁
```

```
    spin_lock(&spin);

    //临界资源访问
    if (globalvar_count)
    {
        spin_unlock(&spin);
        return    - EBUSY;
    }
    globalvar_count++;

    //释放自选锁
    spin_unlock(&spin);

    return 0;
}

static int globalvar_release(struct inode *inode, struct file *filp)
{
    globalvar_count--;
    return 0;
}

static ssize_t globalvar_read(struct file *filp, char *buf, size_t len, loff_t
    *off)
{
    if (down_interruptible(&sem))
    {
        return    - ERESTARTSYS;
    }
    if (copy_to_user(buf, &global_var, sizeof(int)))
    {
        up(&sem);
        return    - EFAULT;
    }
    up(&sem);
    return sizeof(int);
}

static ssize_t globalvar_write(struct file *filp, const char *buf, size_t len,
    loff_t *off)
{
    if (down_interruptible(&sem))
    {
        return    - ERESTARTSYS;
```

```
    }
    if (copy_from_user(&global_var, buf, sizeof(int)))
    {
        up(&sem);
        return    - EFAULT;
    }
    up(&sem);
    return sizeof(int);
}

module_init(globalvar_init);
module_exit(globalvar_exit);
```

为了上述驱动程序的效果，我们启动两个进程分别打开/dev/globalvar。在两个终端中调用./globalvartest.o 测试程序，当一个进程打开/dev/globalvar 后，另外一个进程将打开失败，输出“device open failure”，如下图：



5.设备的阻塞与非阻塞操作

阻塞操作是指，在执行设备操作时，若不能获得资源，则进程挂起直到满足可操作的条件再进行操作。非阻塞操作的进程在不能进行设备操作时，并不挂起。被挂起的进程进入 sleep 状态，被从调度器的运行队列移走，直到等待的条件被满足。

在 Linux 驱动程序中，我们可以使用等待队列（wait queue）来实现阻塞操作。wait queue 很早就作为一个基本的功能单位出现在 Linux 内核里了，它以队列为基础数据结构，与进程

调度机制紧密结合，能够用于实现核心的异步事件通知机制。等待队列可以用来同步对系统资源的访问，上节中所讲述 Linux 信号量在内核中也是由等待队列来实现的。

下面我们重新定义设备“globalvar”，它可以被多个进程打开，但是每次只有当一个进程写入了一个数据之后本进程或其它进程才可以读取该数据，否则一直阻塞。

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <linux/wait.h>
#include <asm/semaphore.h>

MODULE_LICENSE("GPL");

#define MAJOR_NUM    254

static ssize_t globalvar_read(struct file *, char *, size_t, loff_t*);
static ssize_t globalvar_write(struct file *, const char *, size_t, loff_t*);

struct file_operations globalvar_fops =
{
   read: globalvar_read, write: globalvar_write,
};

static int global_var = 0;
static struct semaphore sem;
static wait_queue_head_t outq;
static int flag = 0;

static int __init globalvar_init(void)
{
   int ret;
   ret = register_chrdev(MAJOR_NUM, "globalvar", &globalvar_fops);
   if (ret)
   {
      printk("globalvar register failure");
   }
   else
   {
      printk("globalvar register success");
      init_MUTEX(&sem);
      init_waitqueue_head(&outq);
   }
   return ret;
}
```

```
static void __exit globalvar_exit(void)
{
    int ret;
    ret = unregister_chrdev(MAJOR_NUM, "globalvar");
    if (ret)
    {
        printk("globalvar unregister failure");
    }
    else
    {
        printk("globalvar unregister success");
    }
}


static ssize_t globalvar_read(struct file *filp, char *buf, size_t len, loff_t
    *off)
{
    //等待数据可获得
    if (wait_event_interruptible(outq, flag != 0))
    {
        return    - ERESTARTSYS;
    }

    if (down_interruptible(&sem))
    {
        return    - ERESTARTSYS;
    }

    flag = 0;
    if (copy_to_user(buf, &global_var, sizeof(int)))
    {
        up(&sem);
        return    - EFAULT;
    }

    up(&sem);

    return sizeof(int);
}

static ssize_t globalvar_write(struct file *filp, const char *buf, size_t len,
    loff_t *off)
```

```
{
    if (down_interruptible(&sem))
    {
        return  - ERESTARTSYS;
    }
    if (copy_from_user(&global_var, buf, sizeof(int)))
    {
        up(&sem);
        return  - EFAULT;
    }
    up(&sem);
    flag = 1;
    //通知数据可获得
    wake_up_interruptible(&outq);

    return sizeof(int);
}

module_init(globalvar_init);
module_exit(globalvar_exit);
```

编写两个用户态的程序来测试，第一个用于阻塞地读/dev/globalvar，另一个用于写/dev/globalvar。只有当后一个对/dev/globalvar 进行了输入之后，前者的 read 才能返回。

读的程序为：

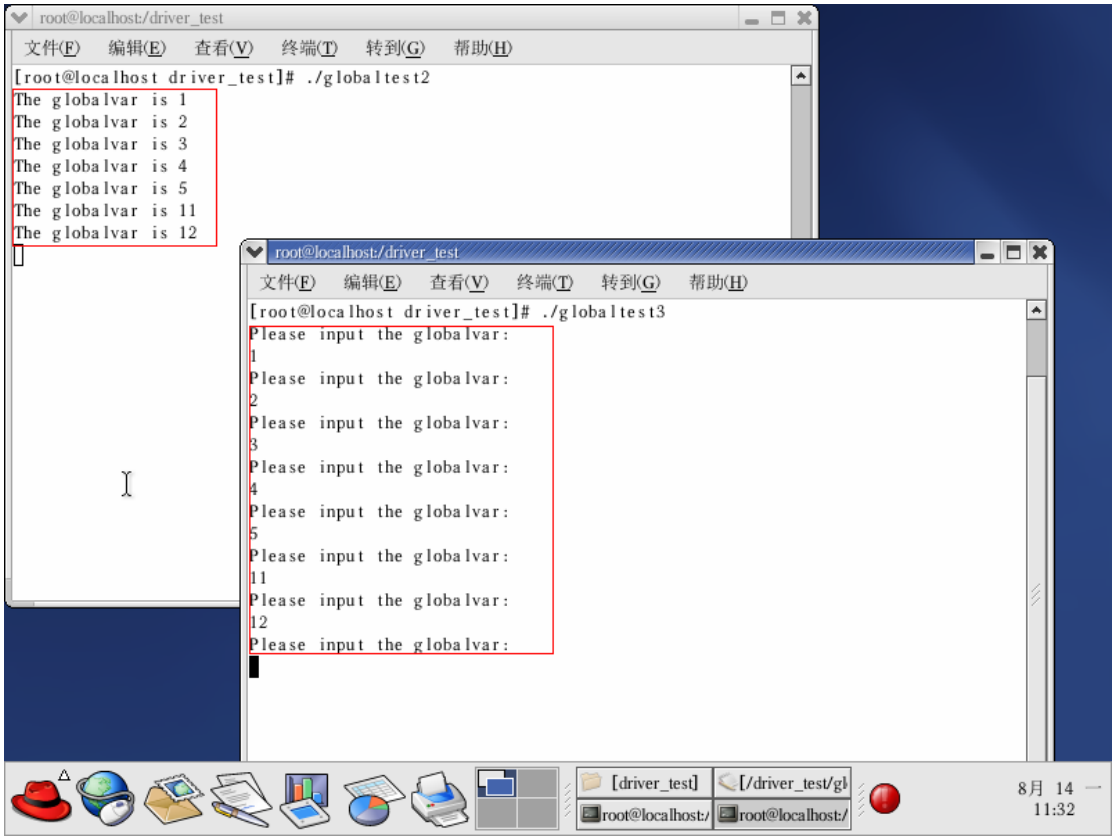
```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>
main()
{
    int fd, num;

    fd = open("/dev/globalvar", O_RDWR, S_IRUSR | S_IWUSR);
    if (fd !=  - 1)
    {
        while (1)
        {
            read(fd, &num, sizeof(int)); //程序将阻塞在此语句，除非有针对 globalvar 的输入
            printf("The globalvar is %d\n", num);

            //如果输入是 0，则退出
            if (num == 0)
            {
                close(fd);
                break;
```

```
        }
    }
  }
  else
  {
    printf("device open failure\n");
  }
}
```

写的程序为：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>
main()
{
  int fd, num;

  fd = open("/dev/globalvar", O_RDWR, S_IRUSR | S_IWUSR);
  if (fd !=  - 1)
  {
    while (1)
    {
      printf("Please input the globalvar:\n");
      scanf("%d", &num);
      write(fd, &num, sizeof(int));

      //如果输入 0，退出
      if (num == 0)
      {
        close(fd);
        break;
      }
    }
  }
  else
  {
    printf("device open failure\n");
  }
}
```

打开两个终端，分别运行上述两个应用程序，发现当在第二个终端中没有输入数据时，第一个终端没有输出（阻塞），每当我们在第二个终端中给 globalvar 输入一个值，第一个终端就会输出这个值，如下图：



关于上述例程，我们补充说一点，如果将驱动程序中的 read 函数改为：

```
static ssize_t globalvar_read(struct file *filp, char *buf, size_t len, loff_t
    *off)
{
    //获取信号量：可能阻塞
    if (down_interruptible(&sem))
    {
        return  - ERESTARTSYS;
    }

    //等待数据可获得：可能阻塞
    if (wait_event_interruptible(outq, flag != 0))
    {
        return  - ERESTARTSYS;
    }
    flag = 0;

    //临界资源访问
    if (copy_to_user(buf, &global_var, sizeof(int)))
    {
        up(&sem);
        return  - EFAULT;
    }
```

```
    //释放信号量
    up(&sem);

    return sizeof(int);
}
```

即交换 wait_event_interruptible(outq, flag != 0)和 down_interruptible(&sem)的顺序，这个驱动程序将变得不可运行。实际上，当两个可能要阻塞的事件同时出现时，即两个 wait_event 或 down 摆在一起的时候，将变得非常危险，死锁的可能性很大，这个时候我们要特别留意它们的出现顺序。当然，我们应该尽可能地避免这种情况的发生！

+还有一个与设备阻塞与非阻塞访问息息相关的论题，即 select 和 poll，select 和 poll 的本质一样，前者在 BSD Unix 中引入，后者在 System V 中引入。poll 和 select 用于查询设备的状态，以便用户程序获知是否能对设备进行非阻塞的访问，它们都需要设备驱动程序中的 poll 函数支持。

驱动程序中 poll 函数中最主要用到的一个 API 是 poll_wait，其原型如下：

```
void poll_wait(struct file *filp, wait_queue_heat_t *queue, poll_table * wait);
```

poll_wait 函数所做的工作是把当前进程添加到 wait 参数指定的等待列表（poll_table）中。下面我们给 globalvar 的驱动添加一个 poll 函数：

```
static unsigned int globalvar_poll(struct file *filp, poll_table *wait)
{
    unsigned int mask = 0;

    poll_wait(filp, &outq, wait);

    //数据是否可获得？
    if (flag != 0)
    {
        mask |= POLLIN | POLLRDNORM; //标示数据可获得
    }

    return mask;
}
```

需要说明的是，poll_wait 函数并不阻塞，程序中 poll_wait(filp, &outq, wait)这句话的意思并不是说一直等待 outq 信号量可获得，真正的阻塞动作是上层的 select/poll 函数中完成的。select/poll 会在一个循环中对每个需要监听的设备调用它们自己的 poll 支持函数以使得当前进程被加入各个设备的等待列表。若当前没有任何被监听的设备就绪，则内核进行调度（调用 schedule）让出 cpu 进入阻塞状态，schedule 返回时将再次循环检测是否有操作可以进行，如此反复；否则，若有任意一个设备就绪，select/poll 都立即返回。

我们编写一个用户态应用程序来测试改写后的驱动。程序中要用到 BSD Unix 中引入的 select 函数，其原型为：

```
int select(int numfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
```

其中 readfds、writefds、exceptfds 分别是被 select()监视的读、写和异常处理的文件描述符集合，numfds 的值是需要检查的号码最高的文件描述符加 1。timeout 参数是一个指向 struct timeval 类型的指针，它可以使 select()在等待 timeout 时间后若没有文件描述符准备好则返回。

struct timeval 数据结构为：

```
struct timeval
{
    int tv_sec; /* seconds */
    int tv_usec; /* microseconds */
};
```

除此之外，我们还将使用下列 API：

FD_ZERO(fd_set *set)－－清除一个文件描述符集；

FD_SET(int fd,fd_set *set)－－将一个文件描述符加入文件描述符集中；

FD_CLR(int fd,fd_set *set)－－将一个文件描述符从文件描述符集中清除；

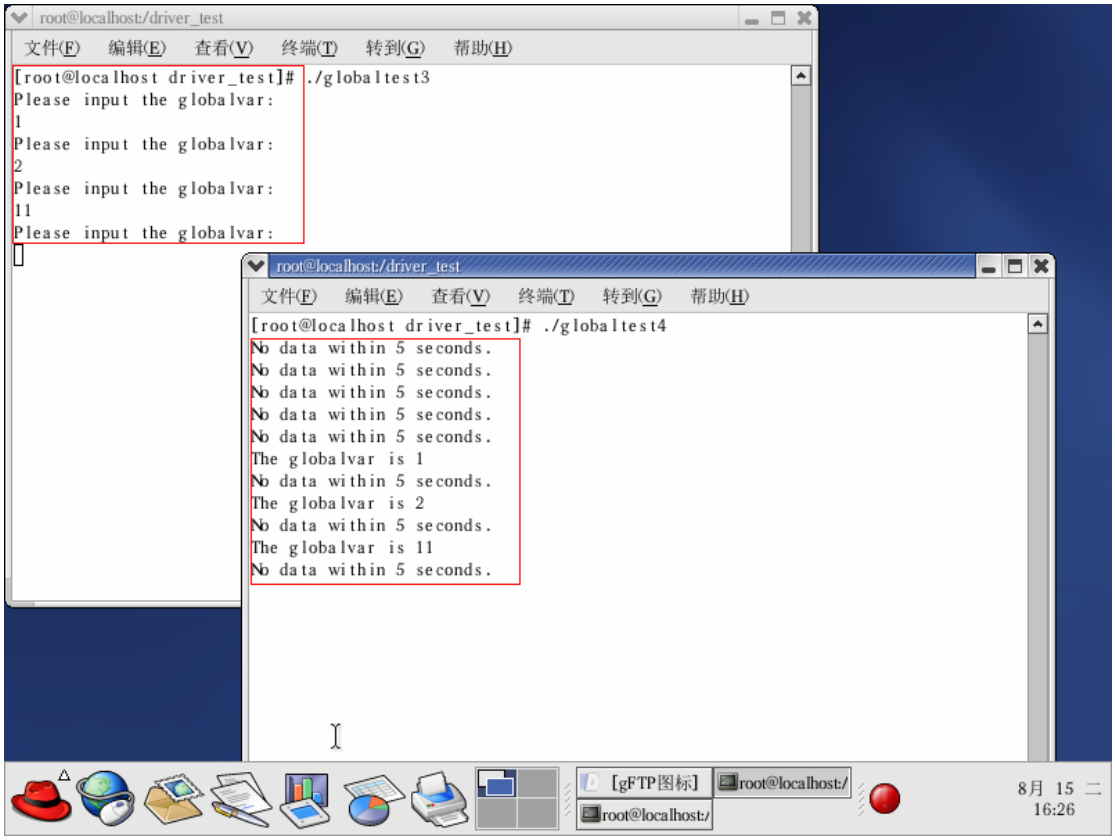
FD_ISSET(int fd,fd_set *set)－－判断文件描述符是否被置位。

下面的用户态测试程序等待/dev/globalvar 可读，但是设置了 5 秒的等待超时，若超过 5 秒仍然没有数据可读，则输出“No data within 5 seconds”：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

main()
{
  int fd, num;
  fd_set rfds;
  struct timeval tv;

  fd = open("/dev/globalvar", O_RDWR, S_IRUSR | S_IWUSR);
  if (fd !=  - 1)
  {
    while (1)
    {
      //查看 globalvar 是否有输入
      FD_ZERO(&rfds);
      FD_SET(fd, &rfds);
      //设置超时时间为 5s
      tv.tv_sec = 5;
      tv.tv_usec = 0;
      select(fd + 1, &rfds, NULL, NULL, &tv);

      //数据是否可获得？
      if (FD_ISSET(fd, &rfds))
      {
        read(fd, &num, sizeof(int));
```

```
            printf("The globalvar is %d\n", num);

            //输入为 0，退出
            if (num == 0)
            {
                close(fd);
                break;
            }
        }
        else
            printf("No data within 5 seconds.\n");
    }
  }
  else
  {
    printf("device open failure\n");
  }
}
```

开两个终端，分别运行程序：一个对 globalvar 进行写，一个用上述程序对 globalvar 进行读。当我们在写终端给 globalvar 输入一个值后，读终端立即就能输出该值，当我们连续 5 秒没有输入时，“No data within 5 seconds”在读终端被输出，如下图：



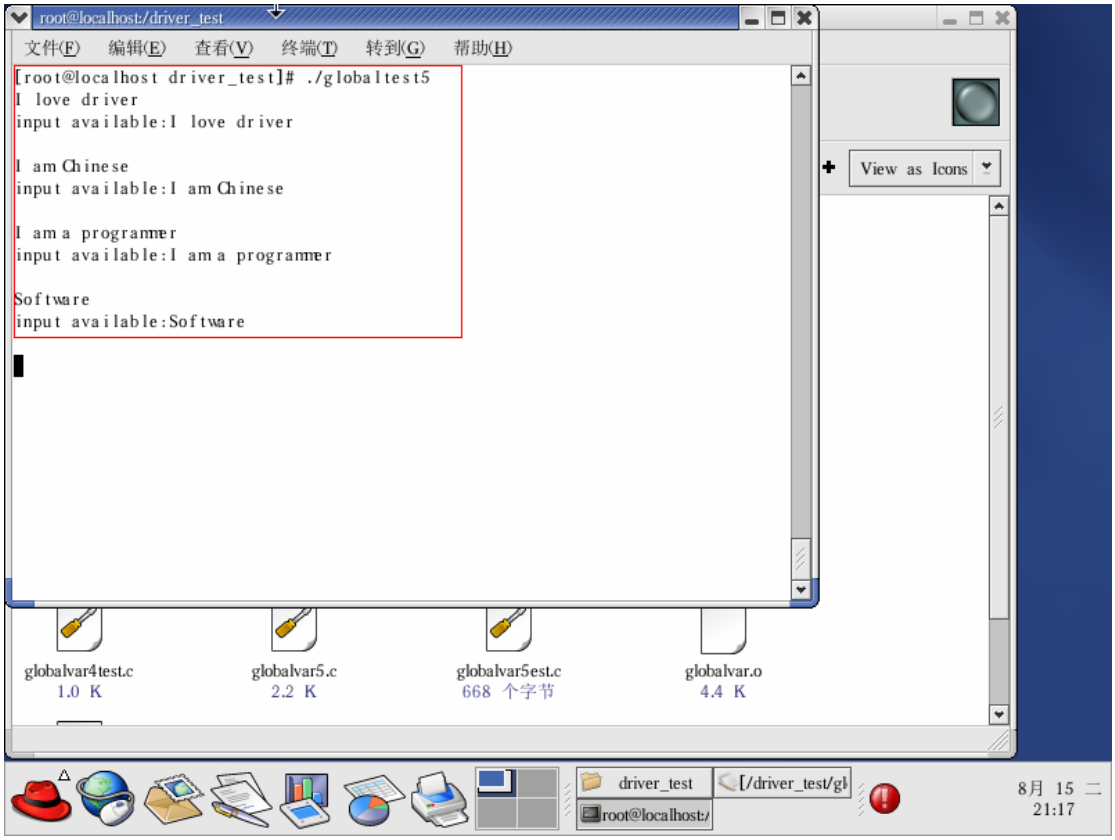
6.设备驱动中的异步通知

结合阻塞与非阻塞访问、poll 函数可以较好地解决设备的读写，但是如果有了异步通知就更方便了。异步通知的意思是：一旦设备就绪，则主动通知应用程序，这样应用程序根本就不需要查询设备状态，这一点非常类似于硬件上“中断”地概念，比较准确的称谓是“信号驱动(SIGIO)的异步 I/O”。

我们先来看一个使用信号驱动的例子，它通过 signal(SIGIO, input_handler)对 STDIN_FILENO 启动信号机制，输入可获得时 input_handler 被调用，其源代码如下：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>
#include <signal.h>
#include <unistd.h>
#define MAX_LEN 100
void input_handler(int num)
{
   char data[MAX_LEN];
   int len;

   //读取并输出 STDIN_FILENO 上的输入
   len = read(STDIN_FILENO, &data, MAX_LEN);
   data[len] = 0;
   printf("input available:%s\n", data);
}

main()
{
   int oflags;

   //启动信号驱动机制
   signal(SIGIO, input_handler);
   fcntl(STDIN_FILENO, F_SETOWN, getpid());
   oflags = fcntl(STDIN_FILENO, F_GETFL);
   fcntl(STDIN_FILENO, F_SETFL, oflags | FASYNC);

   //最后进入一个死循环，程序什么都不干了，只有信号能激发 input_handler 的运行
   //如果程序中没有这个死循环，会立即执行完毕
   while (1);
}
```

程序的运行效果如下图：



为了使设备支持该机制，我们需要在驱动程序中实现 fasync()函数，并在 write()函数中当数据被写入时，调用 kill_fasync()函数激发一个信号，此部分工作留给读者来完成。

7.设备驱动中的中断处理

与 Linux 设备驱动中中断处理相关的首先是申请与释放 IRQ 的 API request_irq()和 free_irq()，request_irq()的原型为：

```
int request_irq(unsigned int irq,
                void (*handler)(int irq, void *dev_id, struct pt_regs *regs),
                unsigned long irqflags,
                const char * devname,
                void *dev_id);
```

irq 是要申请的硬件中断号；

handler 是向系统登记的中断处理函数，是一个回调函数，中断发生时，系统调用这个函数，dev_id 参数将被传递；

irqflags 是中断处理的属性，若设置 SA_INTERRUPT，标明中断处理程序是快速处理程序，快速处理程序被调用时屏蔽所有中断，慢速处理程序不屏蔽；若设置 SA_SHIRQ，则多个设备共享中断，dev_id 在中断共享时会用到，一般设置为这个设备的 device 结构本身或者 NULL。

free_irq()的原型为：

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

另外，与 Linux 中断息息相关的一个重要概念是 Linux 中断分为两个半部：上半部（tophalf）和下半部(bottom half)。上半部的功能是“登记中断”，当一个中断发生时，它进行相应地硬件读写后就把中断例程的下半部挂到该设备的下半部执行队列中去。因此，上半

部执行的速度就会很快，可以服务更多的中断请求。但是，仅有“登记中断”是远远不够的，因为中断的事件可能很复杂。因此，Linux 引入了一个下半部，来完成中断事件的绝大多数使命。下半部和上半部最大的不同是下半部是可中断的，而上半部是不可中断的，下半部几乎做了中断处理程序所有的事情，而且可以被新的中断打断！下半部则相对来说并不是非常紧急的，通常还是比较耗时的，因此由系统自行安排运行时机，不在中断服务上下文中执行。

Linux 实现下半部的机制主要有 tasklet 和工作队列。

tasklet 基于 Linux softirq，其使用相当简单，我们只需要定义 tasklet 及其处理函数并将二者关联：

```
void my_tasklet_func(unsigned long); //定义一个处理函数：
DECLARE_TASKLET(my_tasklet,my_tasklet_func,data); // 定义一个 tasklet 结构 my_tasklet，与 my_tasklet_func(data)函数相关联
```

然后，在需要调度 tasklet 的时候引用一个简单的 API 就能使系统在适当的时候进行调度运行：

```
tasklet_schedule(&my_tasklet);
```

此外，Linux 还提供了另外一些其它的控制 tasklet 调度与运行的 API：

```
DECLARE_TASKLET_DISABLED(name,function,data); //与 DECLARE_TASKLET 类似，但等待 tasklet 被使能
tasklet_enable(struct tasklet_struct *); //使能 tasklet
tasklet_disble(struct tasklet_struct *); //禁用 tasklet
tasklet_init(struct tasklet_struct *,void (*func)(unsigned long),unsigned long); //类似 DECLARE_TASKLET()
tasklet_kill(struct tasklet_struct *); // 清除指定 tasklet 的可调度位，即不允许调度该 tasklet
```

我们先来看一个 tasklet 的运行实例，这个实例没有任何实际意义，仅仅为了演示。它的功能是：在 globalvar 被写入一次后，就调度一个 tasklet，函数中输出“tasklet is executing”：

```
#include <linux/interrupt.h>
…

//定义与绑定 tasklet 函数
void test_tasklet_action(unsigned long t);
DECLARE_TASKLET(test_tasklet, test_tasklet_action, 0);

void test_tasklet_action(unsigned long t)
{
  printk("tasklet is executing\n");
}

…

ssize_t globalvar_write(struct file *filp, const char *buf, size_t len, loff_t
  *off)
{
  …
  if (copy_from_user(&global_var, buf, sizeof(int)))
  {
```

```
        return    - EFAULT;
    }

    //调度 tasklet 执行
    tasklet_schedule(&test_tasklet);
    return sizeof(int);
}
```

由于中断与真实的硬件息息相关，脱离硬件而空谈中断是毫无意义的，我们还是来举一个简单的例子。这个例子来源于 SAMSUNG S3C2410 嵌入式系统实例，看看其中实时钟的驱动中与中断相关的部分：

```
static struct fasync_struct *rtc_async_queue;
static int __init rtc_init(void)
{
    misc_register(&rtc_dev);
    create_proc_read_entry("driver/rtc", 0, 0, rtc_read_proc, NULL);

    #if RTC_IRQ
        if (rtc_has_irq == 0)
            goto no_irq2;

        init_timer(&rtc_irq_timer);
        rtc_irq_timer.function = rtc_dropped_irq;
        spin_lock_irq(&rtc_lock);
        /* Initialize periodic freq. to CMOS reset default, which is 1024Hz */
        CMOS_WRITE(((CMOS_READ(RTC_FREQ_SELECT)      &0xF0)      |      0x06),
RTC_FREQ_SELECT);
        spin_unlock_irq(&rtc_lock);
        rtc_freq = 1024;
        no_irq2:
    #endif

    printk(KERN_INFO "Real Time Clock Driver v" RTC_VERSION "\n");

    return 0;
}

static void __exit rtc_exit(void)
{
    remove_proc_entry("driver/rtc", NULL);
    misc_deregister(&rtc_dev);

    release_region(RTC_PORT(0), RTC_IO_EXTENT);
    if (rtc_has_irq)
        free_irq(RTC_IRQ, NULL);
```

```
}
static void rtc_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    /*
     *      Can be an alarm interrupt, update complete interrupt,
     *      or a periodic interrupt. We store the status in the
     *      low byte and the number of interrupts received since
     *      the last read in the remainder of rtc_irq_data.
     */

    spin_lock(&rtc_lock);
    rtc_irq_data += 0x100;
    rtc_irq_data &= ~0xff;
    rtc_irq_data |= (CMOS_READ(RTC_INTR_FLAGS) &0xF0);

    if (rtc_status &RTC_TIMER_ON)
        mod_timer(&rtc_irq_timer, jiffies + HZ / rtc_freq + 2 * HZ / 100);

    spin_unlock(&rtc_lock);

    /* Now do the rest of the actions */
    wake_up_interruptible(&rtc_wait);

    kill_fasync(&rtc_async_queue, SIGIO, POLL_IN);
}

static int rtc_fasync (int fd, struct file *filp, int on)
{
        return fasync_helper (fd, filp, on, &rtc_async_queue);
}

static void rtc_dropped_irq(unsigned long data)
{
    unsigned long freq;

    spin_lock_irq(&rtc_lock);

    /* Just in case someone disabled the timer from behind our back... */
    if (rtc_status &RTC_TIMER_ON)
        mod_timer(&rtc_irq_timer, jiffies + HZ / rtc_freq + 2 * HZ / 100);

    rtc_irq_data += ((rtc_freq / HZ) << 8);
    rtc_irq_data &= ~0xff;
    rtc_irq_data |= (CMOS_READ(RTC_INTR_FLAGS) &0xF0); /* restart */
```

```
    freq = rtc_freq;

    spin_unlock_irq(&rtc_lock);

    printk(KERN_WARNING "rtc: lost some interrupts at %ldHz.\n", freq);

    /* Now we have new data */
    wake_up_interruptible(&rtc_wait);

    kill_fasync(&rtc_async_queue, SIGIO, POLL_IN);
}
```

RTC 中断发生后，激发了一个异步信号，因此本驱动程序提供了对第 6 节异步信号的支持。并不是每个中断都需要一个下半部，如果本身要处理的事情并不复杂，可能只有一个上半部，本例中的 RTC 驱动就是如此。

8.定时器

Linux 内核中定义了一个 timer_list 结构，我们在驱动程序中可以利用之：

```
struct timer_list {
    struct list_head list;
    unsigned long expires;   //定时器到期时间
    unsigned long data;   //作为参数被传入定时器处理函数
    void (*function)(unsigned long);
};
```

下面是关于 timer 的 API 函数：

增加定时器

```
void add_timer(struct timer_list * timer);
```

删除定时器

```
int del_timer(struct timer_list * timer);
```

修改定时器的 expire

```
int mod_timer(struct timer_list *timer, unsigned long expires);
```

使用定时器的一般流程为：

（1）timer、编写 function；

（2）为 timer 的 expires、data、function 赋值；

（3）调用 add_timer 将 timer 加入列表；

（4）在定时器到期时，function 被执行；

（5）在程序中涉及 timer 控制的地方适当地调用 del_timer、mod_timer 删除 timer 或修改 timer 的 expires。

我们可以参考 drivers\char\keyboard.c 中键盘的驱动中关于 timer 的部分：

```
…
#include <linux/timer.h>
…
static struct timer_list key_autorepeat_timer =
```

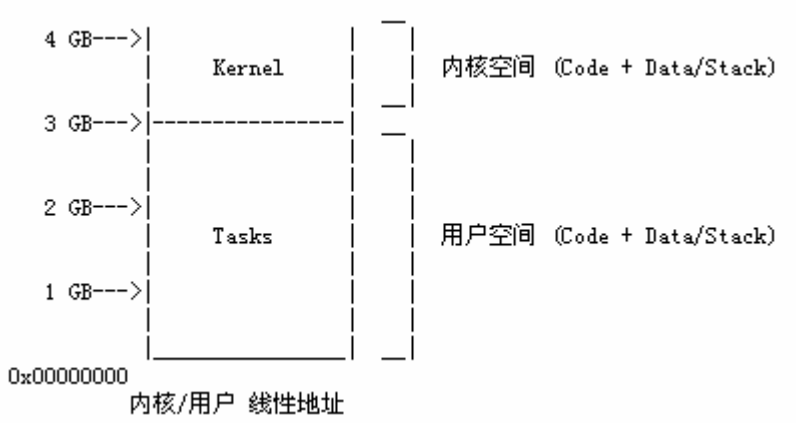
```
{
        function: key_callback
};

static void
kbd_processkeycode(unsigned char keycode, char up_flag, int autorepeat)
{
        char raw_mode = (kbd->kbdmode == VC_RAW);

        if (up_flag) {
                rep = 0;
                if(!test_and_clear_bit(keycode, key_down))
                        up_flag = kbd_unexpected_up(keycode);
        } else {
                rep = test_and_set_bit(keycode, key_down);
                /* If the keyboard autorepeated for us, ignore it.
                 * We do our own autorepeat processing.
                 */
                if (rep && !autorepeat)
                        return;
        }

        if (kbd_repeatkeycode == keycode || !up_flag || raw_mode) {
                kbd_repeatkeycode = -1;
                del_timer(&key_autorepeat_timer);
        }
…
/*
         * Calculate the next time when we have to do some autorepeat
         * processing.    Note that we do not do autorepeat processing
         * while in raw mode but we do do autorepeat processing in
         * medium raw mode.
         */
        if (!up_flag && !raw_mode) {
                kbd_repeatkeycode = keycode;
                if (vc_kbd_mode(kbd, VC_REPEAT)) {
                        if (rep)
                                key_autorepeat_timer.expires = jiffies + kbd_repeatinterval;
                        else
                                key_autorepeat_timer.expires = jiffies + kbd_repeattimeout;
                        add_timer(&key_autorepeat_timer);
                }
        }
…
```

}

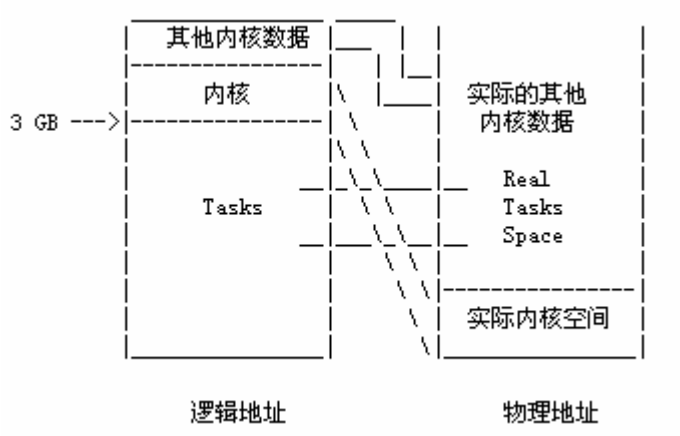
9.内存与 I/O 操作

对于提供了 MMU（存储管理器，辅助操作系统进行内存管理，提供虚实地址转换等硬件支持）的处理器而言，Linux 提供了复杂的存储管理系统，使得进程所能访问的内存达到 4GB。

进程的 4GB 内存空间被人为的分为两个部分——用户空间与内核空间。用户空间地址分布从 0 到 3GB(PAGE_OFFSET，在 0x86 中它等于 0xC0000000)，3GB 到 4GB 为内核空间，如下图：



内核空间中，从 3G 到 vmalloc_start 这段地址是物理内存映射区域（该区域中包含了内核镜像、物理页框表 mem_map 等等），比如我们使用的 VMware 虚拟系统内存是 160M，那么 3G～3G+160M 这片内存就应该映射物理内存。在物理内存映射区之后，就是 vmalloc 区域。对于 160M 的系统而言，vmalloc_start 位置应在 3G+160M 附近（在物理内存映射区与 vmalloc_start 期间还存在一个 8M 的 gap 来防止跃界），vmalloc_end 的位置接近 4G(最后位置系统会保留一片 128k 大小的区域用于专用页面映射)，如下图：



kmalloc 和 get_free_page 申请的内存位于物理内存映射区域，而且在物理上也是连续的，它们与真实的物理地址只有一个固定的偏移，因此存在较简单的转换关系，virt_to_phys() 可以实现内核虚拟地址转化为物理地址：

```
#define __pa(x)    ((unsigned long)(x)-PAGE_OFFSET)
extern inline unsigned long virt_to_phys(volatile void * address)
{
```

```
    return __pa(address);
}
```

上面转换过程是将虚拟地址减去 3G（PAGE_OFFSET=0XC000000）。

与之对应的函数为 phys_to_virt()，将内核物理地址转化为虚拟地址：

```
#define __va(x)   ((void *)((unsigned long)(x)+PAGE_OFFSET))
extern inline void * phys_to_virt(unsigned long address)
{
    return __va(address);
}
```

virt_to_phys()和 phys_to_virt()都定义在 include\asm-i386\io.h 中。

而 vmalloc 申请的内存则位于 vmalloc_start～vmalloc_end 之间，与物理地址没有简单的转换关系，虽然在逻辑上它们也是连续的，但是在物理上它们不要求连续。

我们用下面的程序来演示 kmalloc、get_free_page 和 vmalloc 的区别：

```
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/vmalloc.h>
MODULE_LICENSE("GPL");

unsigned char *pagemem;
unsigned char *kmallocmem;
unsigned char *vmallocmem;

int __init mem_module_init(void)
{
    //最好每次内存申请都检查申请是否成功
    //下面这段仅仅作为演示的代码没有检查
    pagemem = (unsigned char*)get_free_page(0);
    printk("<1>pagemem addr=%x", pagemem);

    kmallocmem = (unsigned char*)kmalloc(100, 0);
    printk("<1>kmallocmem addr=%x", kmallocmem);

    vmallocmem = (unsigned char*)vmalloc(1000000);
    printk("<1>vmallocmem addr=%x", vmallocmem);

    return 0;
}

void __exit mem_module_exit(void)
{
    free_page(pagemem);
    kfree(kmallocmem);
    vfree(vmallocmem);
}
```

```
module_init(mem_module_init);
module_exit(mem_module_exit);
```

我们的系统上有 160MB 的内存空间，运行一次上述程序，发现 pagemem 的地址在 0xc7997000（约 3G+121M）、kmallocmem 地址在 0xc9bc1380（约 3G+155M）、vmallocmem 的地址在 0xcabeb000（约 3G+171M）处，符合前文所述的内存布局。

接下来，我们讨论 Linux 设备驱动究竟怎样访问外设的 I/O 端口（寄存器）。

几乎每一种外设都是通过读写设备上的寄存器来进行的，通常包括控制寄存器、状态寄存器和数据寄存器三大类，外设的寄存器通常被连续地编址。根据 CPU 体系结构的不同，CPU 对 IO 端口的编址方式有两种：

（1）I/O 映射方式（I/O-mapped）

典型地，如 X86 处理器为外设专门实现了一个单独的地址空间，称为“I/O 地址空间”或者“I/O 端口空间”，CPU 通过专门的 I/O 指令（如 X86 的 IN 和 OUT 指令）来访问这一空间中的地址单元。

（2）内存映射方式（Memory-mapped）

RISC 指令系统的 CPU（如 ARM、PowerPC 等）通常只实现一个物理地址空间，外设 I/O 端口成为内存的一部分。此时，CPU 可以象访问一个内存单元那样访问外设 I/O 端口，而不需要设立专门的外设 I/O 指令。

但是，这两者在硬件实现上的差异对于软件来说是完全透明的，驱动程序开发人员可以将内存映射方式的 I/O 端口和外设内存统一看作是“I/O 内存”资源。

一般来说，在系统运行时，外设的 I/O 内存资源的物理地址是已知的，由硬件的设计决定。但是 CPU 通常并没有为这些已知的外设 I/O 内存资源的物理地址预定义虚拟地址范围，驱动程序并不能直接通过物理地址访问 I/O 内存资源，而必须将它们映射到核心虚地址空间内（通过页表），然后才能根据映射所得到的核心虚地址范围，通过访内指令访问这些 I/O 内存资源。Linux 在 io.h 头文件中声明了函数 ioremap（），用来将 I/O 内存资源的物理地址映射到核心虚地址空间（3GB－4GB）中，原型如下：

```
void * ioremap(unsigned long phys_addr, unsigned long size, unsigned long flags);
```

iounmap 函数用于取消 ioremap（）所做的映射，原型如下：

```
void iounmap(void * addr);
```

这两个函数都是实现在 mm/ioremap.c 文件中。

在将 I/O 内存资源的物理地址映射成核心虚地址后，理论上讲我们就可以象读写 RAM 那样直接读写 I/O 内存资源了。为了保证驱动程序的跨平台的可移植性，我们应该使用 Linux 中特定的函数来访问 I/O 内存资源，而不应该通过指向核心虚地址的指针来访问。如在 x86 平台上，读写 I/O 的函数如下所示：

```
#define readb(addr) (*(volatile unsigned char *) __io_virt(addr))
#define readw(addr) (*(volatile unsigned short *) __io_virt(addr))
#define readl(addr) (*(volatile unsigned int *) __io_virt(addr))

#define writeb(b,addr) (*(volatile unsigned char *) __io_virt(addr) = (b))
#define writew(b,addr) (*(volatile unsigned short *) __io_virt(addr) = (b))
#define writel(b,addr) (*(volatile unsigned int *) __io_virt(addr) = (b))

#define memset_io(a,b,c)      memset(__io_virt(a),(b),(c))
#define memcpy_fromio(a,b,c) memcpy((a),__io_virt(b),(c))
```

```
#define memcpy_toio(a,b,c)   memcpy(__io_virt(a),(b),(c))
```

最后，我们要特别强调驱动程序中 mmap 函数的实现方法。用 mmap 映射一个设备，意味着使用户空间的一段地址关联到设备内存上，这使得只要程序在分配的地址范围内进行读取或者写入，实际上就是对设备的访问。

笔者在 Linux 源代码中进行包含“ioremap”文本的搜索，发现真正出现的 ioremap 的地方相当少。所以笔者追根索源地寻找 I/O 操作的物理地址转换到虚拟地址的真实所在，发现 Linux 有替代 ioremap 的语句，但是这个转换过程却是不可或缺的。

譬如我们再次摘取 S3C2410 这个 ARM 芯片 RTC（实时钟）驱动中的一小段：

```
static void get_rtc_time(int alm, struct rtc_time *rtc_tm)
{
    spin_lock_irq(&rtc_lock);
    if (alm == 1) {
        rtc_tm->tm_year = (unsigned char)ALMYEAR & Msk_RTCYEAR;
        rtc_tm->tm_mon = (unsigned char)ALMMON & Msk_RTCMON;
        rtc_tm->tm_mday = (unsigned char)ALMDAY & Msk_RTCDAY;
        rtc_tm->tm_hour = (unsigned char)ALMHOUR & Msk_RTCHOUR;
        rtc_tm->tm_min = (unsigned char)ALMMIN & Msk_RTCMIN;
        rtc_tm->tm_sec = (unsigned char)ALMSEC & Msk_RTCSEC;
    }
    else {
 read_rtc_bcd_time:
        rtc_tm->tm_year = (unsigned char)BCDYEAR & Msk_RTCYEAR;
        rtc_tm->tm_mon = (unsigned char)BCDMON & Msk_RTCMON;
        rtc_tm->tm_mday = (unsigned char)BCDDAY & Msk_RTCDAY;
        rtc_tm->tm_hour = (unsigned char)BCDHOUR & Msk_RTCHOUR;
        rtc_tm->tm_min = (unsigned char)BCDMIN & Msk_RTCMIN;
        rtc_tm->tm_sec = (unsigned char)BCDSEC & Msk_RTCSEC;

        if (rtc_tm->tm_sec == 0) {
            /* Re-read all BCD registers in case of BCDSEC is 0.
                See RTC section at the manual for more info. */
            goto read_rtc_bcd_time;
        }
    }
    spin_unlock_irq(&rtc_lock);

    BCD_TO_BIN(rtc_tm->tm_year);
    BCD_TO_BIN(rtc_tm->tm_mon);
    BCD_TO_BIN(rtc_tm->tm_mday);
    BCD_TO_BIN(rtc_tm->tm_hour);
    BCD_TO_BIN(rtc_tm->tm_min);
    BCD_TO_BIN(rtc_tm->tm_sec);

    /* The epoch of tm_year is 1900 */
```

```
    rtc_tm->tm_year += RTC_LEAP_YEAR - 1900;

    /* tm_mon starts at 0, but rtc month starts at 1 */
    rtc_tm->tm_mon--;
}
```

I/O 操作似乎就是对 ALMYEAR、ALMMON、ALMDAY 定义的寄存器进行操作，那这些宏究竟定义为什么呢？

```
#define ALMDAY              bRTC(0x60)
#define ALMMON              bRTC(0x64)
#define ALMYEAR             bRTC(0x68)
```

其中借助了宏 bRTC，这个宏定义为：

```
#define bRTC(Nb)        __REG(0x57000000 + (Nb))
```

其中又借助了宏__REG，而__REG 又定义为：

```
# define __REG(x)       io_p2v(x)
```

最后的 io_p2v 才是真正“玩”虚拟地址和物理地址转换的地方：

```
#define io_p2v(x) ((x) | 0xa0000000)
```

与__REG 对应的有个__PREG：

```
# define __PREG(x)      io_v2p(x)
```

与 io_p2v 对应的有个 io_v2p：

```
#define io_v2p(x) ((x) & ~0xa0000000)
```

可见有没有出现 ioremap 是次要的，关键问题是有无虚拟地址和物理地址的转换！

下面的程序在启动的时候保留一段内存，然后使用 ioremap 将它映射到内核虚拟空间，同时又用 remap_page_range 映射到用户虚拟空间，这样一来，内核和用户都能访问。如果在内核虚拟地址将这段内存初始化串"abcd"，那么在用户虚拟地址能够读出来：

```
/************mmap_ioremap.c**************/
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/errno.h>
#include <linux/mm.h>
#include <linux/wrapper.h>    /* for mem_map_(un)reserve */
#include <asm/io.h>                 /* for virt_to_phys */
#include <linux/slab.h>     /* for kmalloc and kfree */

MODULE_PARM(mem_start, "i");
MODULE_PARM(mem_size, "i");

static int mem_start = 101, mem_size = 10;
static char *reserve_virt_addr;
static int major;

int mmapdrv_open(struct inode *inode, struct file *file);
int mmapdrv_release(struct inode *inode, struct file *file);
int mmapdrv_mmap(struct file *file, struct vm_area_struct *vma);
```

```
static struct file_operations mmapdrv_fops =
{
    owner: THIS_MODULE, mmap: mmapdrv_mmap, open: mmapdrv_open, release:
        mmapdrv_release,
};

int init_module(void)
{
    if ((major = register_chrdev(0, "mmapdrv", &mmapdrv_fops)) < 0)
    {
        printk("mmapdrv: unable to register character device\n");
        return ( - EIO);
    }
    printk("mmap device major = %d\n", major);

    printk("high memory physical address 0x%ldM\n", virt_to_phys(high_memory) /
        1024 / 1024);

    reserve_virt_addr = ioremap(mem_start *1024 * 1024, mem_size *1024 * 1024);
    printk("reserve_virt_addr = 0x%lx\n", (unsigned long)reserve_virt_addr);
    if (reserve_virt_addr)
    {
        int i;
        for (i = 0; i < mem_size *1024 * 1024; i += 4)
        {
            reserve_virt_addr[i] = 'a';
            reserve_virt_addr[i + 1] = 'b';
            reserve_virt_addr[i + 2] = 'c';
            reserve_virt_addr[i + 3] = 'd';
        }
    }
    else
    {
        unregister_chrdev(major, "mmapdrv");
        return    - ENODEV;
    }

    return 0;
}

/* remove the module */
void cleanup_module(void)
{
    if (reserve_virt_addr)
```

```
    iounmap(reserve_virt_addr);

  unregister_chrdev(major, "mmapdrv");

  return ;
}

int mmapdrv_open(struct inode *inode, struct file *file)
{
  MOD_INC_USE_COUNT;
  return (0);
}

int mmapdrv_release(struct inode *inode, struct file *file)
{
  MOD_DEC_USE_COUNT;
  return (0);
}

int mmapdrv_mmap(struct file *file, struct vm_area_struct *vma)
{
  unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;
  unsigned long size = vma->vm_end - vma->vm_start;

  if (size > mem_size *1024 * 1024)
  {
    printk("size too big\n");
    return ( - ENXIO);
  }

  offset = offset + mem_start * 1024 * 1024;

  /* we do not want to have this area swapped out, lock it */
  vma->vm_flags |= VM_LOCKED;
  if (remap_page_range(vma, vma->vm_start, offset, size, PAGE_SHARED))
  {
    printk("remap page range failed\n");
    return    - ENXIO;
  }

  return (0);
}
```

remap_page_range 函数的功能是构造用于映射一段物理地址的新页表，实现了内核空间与用户空间的映射，其原型如下：

int remap_page_range(vma_area_struct *vma, unsigned long from, unsigned long to, unsigned long size, pgprot_tprot);

使用 mmap 最典型的例子是显示卡的驱动，将显存空间直接从内核映射到用户空间将可提供显存的读写效率。

10.结构化设备驱动程序

在 1~9 节关于设备驱动的例子中，我们没有考虑设备驱动程序的结构组织问题。实际上，Linux 设备驱动的开发者习惯于一套约定俗成的数据结构组织方法和程序框架。

设备结构体

Linux 设备驱动程序的编写者喜欢把与某设备相关的所有内容定义为一个设备结构体，其中包括设备驱动涉及的硬件资源、全局软件资源、控制（自旋锁、互斥锁、等待队列、定时器等），在涉及设备的操作时，仅仅操作这个结构体就可以了。

对于“globalvar”设备，这个结构体就是：

```
struct globalvar_dev
{
  int global_var = 0;
  struct semaphore sem;
  wait_queue_head_t outq;
  int flag = 0;
};
```

open()和 release()

一般来说，较规范的 open( )通常需要完成下列工作：

1. 检查设备相关错误，如设备尚未准备好等；
2. 如果是第一次打开，则初始化硬件设备；
3. 识别次设备号，如果有必要则更新读写操作的当前位置指针 f_ops；
4. 分配和填写要放在 file->private_data 里的数据结构；
5. 使用计数增 1。

release( )的作用正好与 open( )相反，通常要完成下列工作：

1. 使用计数减 1；
2. 释放在 file->private_data 中分配的内存；
3. 如果使用计算为 0，则关闭设备。

我们使用 LDD2 中 scull_u 的例子：

```
int scull_u_open(struct inode *inode, struct file *filp)
{
    Scull_Dev *dev = &scull_u_device; /* device information */
    int num = NUM(inode->i_rdev);

    if (!filp->private_data && num > 0)
        return -ENODEV; /* not devfs: allow 1 device only */
    spin_lock(&scull_u_lock);
    if (scull_u_count &&
        (scull_u_owner != current->uid) &&   /* allow user */
        (scull_u_owner != current->euid) && /* allow whoever did su */
```

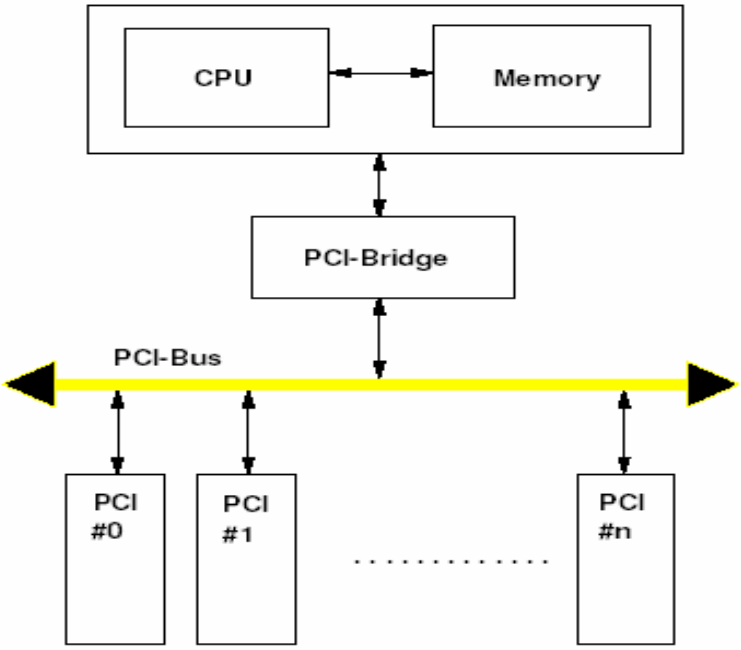
```
                                !capable(CAP_DAC_OVERRIDE)) { /* still allow root */
                spin_unlock(&scull_u_lock);
                return -EBUSY;      /* -EPERM would confuse the user */
        }

        if (scull_u_count == 0)
                scull_u_owner = current->uid; /* grab it */

        scull_u_count++;
        spin_unlock(&scull_u_lock);

        /* then, everything else is copied from the bare scull device */

        if ( (filp->f_flags & O_ACCMODE) == O_WRONLY)
                scull_trim(dev);
        if (!filp->private_data)
                filp->private_data = dev;
        MOD_INC_USE_COUNT;
        return 0;                   /* success */
}

int scull_u_release(struct inode *inode, struct file *filp)
{
        scull_u_count--; /* nothing else */
        MOD_DEC_USE_COUNT;
        return 0;
}
```

上面所述为一般意义上的设计规范，应该说是 option（可选的）而非强制的。

11.复杂设备驱动

这里所说的复杂设备驱动涉及到 PCI、USB、网络设备、块设备等（严格意义而言，这些设备在概念上并不并列，例如与块设备并列的是字符设备，而 PCI、USB 设备等都可能属于字符设备），这些设备的驱动中又涉及到一些与特定设备类型相关的较为复杂的数据结构和程序结构。本文将不对这些设备驱动的细节进行过多的介绍，仅仅进行轻描淡写的叙述。

PCI 是 The Peripheral Component Interconnect –Bus 的缩写，CPU 使用 PCI 桥 chipset 与 PCI 设备通信，PCI 桥 chipset 处理了 PCI 子系统与内存子系统间的所有数据交互，PCI 设备完全被从内存子系统分离出来。下图呈现了 PCI 子系统的原理：



每个 PCI 设备都有一个 256 字节的设备配置块，其中前 64 字节作为设备的 ID 和基本配置信息，Linux 中提供了一组函数来处理 PCI 配置块。在 PCI 设备能得以使用前，Linux 驱动程序需要从 PCI 设备配置块中的信息决定设备的特定参数，进行相关设置以便能正确操作该 PCI 设备。

一般的 PCI 设备初始化函数处理流程为：

（1）检查内核是否支持 PCI-Bios；

（2）检查设备是否存在，获得设备的配置信息；

1～2 这两步的例子如下：

```
int pcidata_read_proc(char *buf, char **start, off_t offset, int len, int *eof,
   void *data)
{
   int i, pos = 0;
   int bus, devfn;

   if (!pcibios_present())
      return sprintf(buf, "No PCI bios present\n");

   /*
    * This code is derived from "drivers/pci/pci.c". This means that
    * the GPL applies to this source file and credit is due to the
    * original authors (Drew Eckhardt, Frederic Potter, David
    * Mosberger-Tang)
    */
   for (bus = 0; !bus; bus++)
   {
       /* only bus 0 :-) */
      for (devfn = 0; devfn < 0x100 && pos < PAGE_SIZE / 2; devfn++)
```

```
        {
            struct pci_dev *dev = NULL;

            dev = pci_find_slot(bus, devfn);
            if (!dev)
                continue;

            /* Ok, we've found a device, copy its cfg space to the buffer*/
            for (i = 0; i < 256; i += sizeof(u32), pos += sizeof(u32))
                pci_read_config_dword(dev, i, (u32*)(buf + pos));
            pci_release_device(dev); /* 2.0 compatibility */
        }
    }
    *eof = 1;
    return pos;
}
```

其中使用的 pci_find_slot()函数定义为：

```
struct pci_dev *pci_find_slot (unsigned int bus,
                               unsigned int devfn)
{
    struct pci_dev *pptr = kmalloc(sizeof(*pptr), GFP_KERNEL);
    int index = 0;
    unsigned short vendor;
    int ret;

    if (!pptr) return NULL;
    pptr->index = index; /* 0 */
    ret = pcibios_read_config_word(bus, devfn, PCI_VENDOR_ID, &vendor);
    if (ret /* == PCIBIOS_DEVICE_NOT_FOUND or whatever error */
    || vendor==0xffff || vendor==0x0000) {
        kfree(pptr); return NULL;
    }
    printk("ok (%i, %i %x)\n", bus, devfn, vendor);
    /* fill other fields */
    pptr->bus = bus;
    pptr->devfn = devfn;
    pcibios_read_config_word(pptr->bus, pptr->devfn,
                             PCI_VENDOR_ID, &pptr->vendor);
    pcibios_read_config_word(pptr->bus, pptr->devfn,
                             PCI_DEVICE_ID, &pptr->device);
    return pptr;
}
```

（3）根据设备的配置信息申请 I/O 空间及 IRQ 资源；

（4）注册设备。

USB 设备的驱动主要处理 probe（探测）、disconnect（断开）函数及 usb_device_id（设备信息）数据结构，如：

```
static struct usb_device_id sample_id_table[] =
{
   {
      USB_INTERFACE_INFO(3, 1, 1), driver_info: (unsigned long)"keyboard"
   } ,
   {
      USB_INTERFACE_INFO(3, 1, 2), driver_info: (unsigned long)"mouse"
   }
   ,
   {
      0,    /* no more matches */
   }
};

static struct usb_driver sample_usb_driver =
{
   name: "sample", probe: sample_probe, disconnect: sample_disconnect, id_table:
      sample_id_table,
};
```

当一个 USB 设备从系统拔掉后，设备驱动程序的 disconnect 函数会自动被调用，在执行了 disconnect 函数后，所有为 USB 设备分配的数据结构，内存空间都会被释放：

```
static void sample_disconnect(struct usb_device *udev, void *clientdata)
{
      /* the clientdata is the sample_device we passed originally */
      struct sample_device *sample = clientdata;

      /* remove the URB, remove the input device, free memory */
      usb_unlink_urb(&sample->urb);
      kfree(sample);
      printk(KERN_INFO "sample: USB %s disconnected\n", sample->name);

      /*
        * here you might MOD_DEC_USE_COUNT, but only if you increment
        * the count in sample_probe() below
        */
      return;
}
```

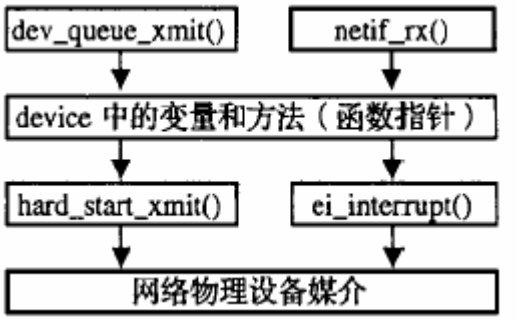
当驱动程序向子系统注册后，插入一个新的 USB 设备后总是要自动进入 probe 函数。驱动程序会为这个新加入系统的设备向内部的数据结构建立一个新的实例。通常情况下，probe 函数执行一些功能来检测新加入的 USB 设备硬件中的生产厂商和产品定义以及设备所属的类或子类定义是否与驱动程序相符，若相符，再比较接口的数目与本驱动程序支持设备的接口数目是否相符。一般在 probe 函数中也会解析 USB 设备的说明，从而确认新加入

的 USB 设备会使用这个驱动程序：

```
static void *sample_probe(struct usb_device *udev, unsigned int ifnum,
                          const struct usb_device_id *id)
{
    /*
     * The probe procedure is pretty standard. Device matching has already
     * been performed based on the id_table structure (defined later)
     */
    struct usb_interface *iface;
    struct usb_interface_descriptor *interface;
    struct usb_endpoint_descriptor *endpoint;
    struct sample_device *sample;

    printk(KERN_INFO "usbsample: probe called for %s device\n",
        (char *)id->driver_info /* "mouse" or "keyboard" */ );

    iface = &udev->actconfig->interface[ifnum];
    interface = &iface->altsetting[iface->act_altsetting];

    if (interface->bNumEndpoints != 1) return NULL;

    endpoint = interface->endpoint + 0;
    if (!(endpoint->bEndpointAddress & 0x80)) return NULL;
    if ((endpoint->bmAttributes & 3) != 3) return NULL;

    usb_set_protocol(udev, interface->bInterfaceNumber, 0);
    usb_set_idle(udev, interface->bInterfaceNumber, 0, 0);

    /* allocate and zero a new data structure for the new device */
    sample = kmalloc(sizeof(struct sample_device), GFP_KERNEL);
    if (!sample) return NULL; /* failure */
    memset(sample, 0, sizeof(*sample));
    sample->name = (char *)id->driver_info;

    /* fill the URB data structure using the FILL_INT_URB macro */
    {
    int pipe = usb_rcvintpipe(udev, endpoint->bEndpointAddress);
    int maxp = usb_maxpacket(udev, pipe, usb_pipeout(pipe));

    if (maxp > 8) maxp = 8; sample->maxp = maxp; /* remember for later */
    FILL_INT_URB(&sample->urb, udev, pipe, sample->data, maxp,
                          sample_irq, sample, endpoint->bInterval);
    }
```

```
        /* register the URB within the USB subsystem */
        if (usb_submit_urb(&sample->urb)) {
        kfree(sample);
        return NULL;
        }
        /* announce yourself */
        printk(KERN_INFO "usbsample: probe successful for %s (maxp is %i)\n",
             sample->name, sample->maxp);

        /*
          * here you might MOD_INC_USE_COUNT; if you do, you'll need to unplug
          * the device or the devices before being able to unload the module
          */

        /* and return the new structure */
        return sample;
}
```

在网络设备驱动的编写中，我们特别关心的就是数据的收、发及中断。网络设备驱动程序的层次如下：



网络设备接收到报文后将其传入上层：

```
/*
 * Receive a packet: retrieve, encapsulate and pass over to upper levels
 */
void snull_rx(struct net_device *dev, int len, unsigned char *buf)
{
        struct sk_buff *skb;
        struct snull_priv *priv = (struct snull_priv *) dev->priv;

        /*
          * The packet has been retrieved from the transmission
          * medium. Build an skb around it, so upper layers can handle it
          */
        skb = dev_alloc_skb(len+2);
        if (!skb) {
                printk("snull rx: low on mem - packet dropped\n");
                priv->stats.rx_dropped++;
```

```
            return;
        }
        skb_reserve(skb, 2); /* align IP on 16B boundary */
        memcpy(skb_put(skb, len), buf, len);

        /* Write metadata, and then pass to the receive level */
        skb->dev = dev;
        skb->protocol = eth_type_trans(skb, dev);
        skb->ip_summed = CHECKSUM_UNNECESSARY; /* don't check it */
        priv->stats.rx_packets++;
#ifndef LINUX_20
        priv->stats.rx_bytes += len;
#endif
        netif_rx(skb);
        return;
}
```

在中断到来时接收报文信息：

```
void snull_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
        int statusword;
        struct snull_priv *priv;
        /*
         * As usual, check the "device" pointer for shared handlers.
         * Then assign "struct device *dev"
         */
        struct net_device *dev = (struct net_device *)dev_id;
        /* ... and check with hw if it's really ours */

        if (!dev /*paranoid*/ ) return;

        /* Lock the device */
        priv = (struct snull_priv *) dev->priv;
        spin_lock(&priv->lock);

        /* retrieve statusword: real netdevices use I/O instructions */
        statusword = priv->status;
        if (statusword & SNULL_RX_INTR) {
                /* send it to snull_rx for handling */
                snull_rx(dev, priv->rx_packetlen, priv->rx_packetdata);
        }
        if (statusword & SNULL_TX_INTR) {
                /* a transmission is over: free the skb */
                priv->stats.tx_packets++;
                priv->stats.tx_bytes += priv->tx_packetlen;
```

```
            dev_kfree_skb(priv->skb);
        }

        /* Unlock the device and we are done */
        spin_unlock(&priv->lock);
        return;
}
```

而发送报文则分为两个层次，一个层次是内核调用，一个层次完成真正的硬件上的发送：

```
/*
 * Transmit a packet (called by the kernel)
 */
int snull_tx(struct sk_buff *skb, struct net_device *dev)
{
        int len;
        char *data;
        struct snull_priv *priv = (struct snull_priv *) dev->priv;

#ifndef LINUX_24
        if (dev->tbusy || skb == NULL) {
                PDEBUG("tint for %p, tbusy %ld, skb %p\n", dev, dev->tbusy, skb);
                snull_tx_timeout (dev);
                if (skb == NULL)
                        return 0;
        }
#endif

        len = skb->len < ETH_ZLEN ? ETH_ZLEN : skb->len;
        data = skb->data;
        dev->trans_start = jiffies; /* save the timestamp */

        /* Remember the skb, so we can free it at interrupt time */
        priv->skb = skb;

        /* actual deliver of data is device-specific, and not shown here */
        snull_hw_tx(data, len, dev);

        return 0; /* Our simple device can not fail */
}

/*
 * Transmit a packet (low level interface)
 */
void snull_hw_tx(char *buf, int len, struct net_device *dev)
{
```

```
    /*
     * This function deals with hw details. This interface loops
     * back the packet to the other snull interface (if any).
     * In other words, this function implements the snull behaviour,
     * while all other procedures are rather device-independent
     */
    struct iphdr *ih;
    struct net_device *dest;
    struct snull_priv *priv;
    u32 *saddr, *daddr;

    /* I am paranoid. Ain't I? */
    if (len < sizeof(struct ethhdr) + sizeof(struct iphdr)) {
            printk("snull: Hmm... packet too short (%i octets)\n",
                    len);
            return;
    }

    if (0) { /* enable this conditional to look at the data */
            int i;
            PDEBUG("len is %i\n" KERN_DEBUG "data:",len);
            for (i=14 ; i<len; i++)
                    printk(" %02x",buf[i]&0xff);
            printk("\n");
    }
    /*
     * Ethhdr is 14 bytes, but the kernel arranges for iphdr
     * to be aligned (i.e., ethhdr is unaligned)
     */
    ih = (struct iphdr *)(buf+sizeof(struct ethhdr));
    saddr = &ih->saddr;
    daddr = &ih->daddr;

    ((u8 *)saddr)[2] ^= 1; /* change the third octet (class C) */
    ((u8 *)daddr)[2] ^= 1;

    ih->check = 0;          /* and rebuild the checksum (ip needs it) */
    ih->check = ip_fast_csum((unsigned char *)ih,ih->ihl);

    if (dev == snull_devs)
            PDEBUGG("%08x:%05i --> %08x:%05i\n",
                    ntohl(ih->saddr),ntohs(((struct tcphdr *)(ih+1))->source),
                    ntohl(ih->daddr),ntohs(((struct tcphdr *)(ih+1))->dest));
    else
```

```
            PDEBUGG("%08x:%05i <-- %08x:%05i\n",
                    ntohl(ih->daddr),ntohs(((struct tcphdr *)(ih+1))->dest),
                    ntohl(ih->saddr),ntohs(((struct tcphdr *)(ih+1))->source));

    /*
     * Ok, now the packet is ready for transmission: first simulate a
     * receive interrupt on the twin device, then  a
     * transmission-done on the transmitting device
     */
    dest = snull_devs + (dev==snull_devs ? 1 : 0);
    priv = (struct snull_priv *) dest->priv;
    priv->status = SNULL_RX_INTR;
    priv->rx_packetlen = len;
    priv->rx_packetdata = buf;
    snull_interrupt(0, dest, NULL);

    priv = (struct snull_priv *) dev->priv;
    priv->status = SNULL_TX_INTR;
    priv->tx_packetlen = len;
    priv->tx_packetdata = buf;
    if (lockup && ((priv->stats.tx_packets + 1) % lockup) == 0) {
        /* Simulate a dropped transmit interrupt */
        netif_stop_queue(dev);
        PDEBUG("Simulate lockup at %ld, txp %ld\n", jiffies,
                (unsigned long) priv->stats.tx_packets);
    }
    else
        snull_interrupt(0, dev, NULL);
}
```

块设备也以与字符设备 register_chrdev、unregister_ chrdev 函数类似的方法进行设备的注册与释放。但是，register_chrdev 使用一个向 file_operations 结构的指针，而 register_blkdev 则使用 block_device_operations 结构的指针，其中定义的 open、release 和 ioctl 方法和字符设备的对应方法相同，但未定义 read 或者 write 操作。这是因为，所有涉及到块设备的 I/O 通常由系统进行缓冲处理。

块驱动程序最终必须提供完成实际块 I/O 操作的机制，在 Linux 中，用于这些 I/O 操作的方法称为“request（请求）”。在块设备的注册过程中，需要初始化 request 队列，这一动作通过 blk_init_queue 来完成，blk_init_queue 函数建立队列，并将该驱动程序的 request 函数关联到队列。在模块的清除阶段，应调用 blk_cleanup_queue 函数。看看 mtdblock 的例子：

```
static void handle_mtdblock_request(void)
{
    struct request *req;
    struct mtdblk_dev *mtdblk;
    unsigned int res;
```

```
    for (;;) {
        INIT_REQUEST;
        req = CURRENT;
        spin_unlock_irq(QUEUE_LOCK(QUEUE));
        mtdblk = mtdblks[minor(req->rq_dev)];
        res = 0;

        if (minor(req->rq_dev) >= MAX_MTD_DEVICES)
            panic("%s : minor out of bound", __FUNCTION__);

        if (!IS_REQ_CMD(req))
            goto end_req;

        if ((req->sector + req->current_nr_sectors) > (mtdblk->mtd->size >> 9))
            goto end_req;

        // Handle the request
        switch (rq_data_dir(req))
        {
            int err;

            case READ:
            down(&mtdblk->cache_sem);
            err = do_cached_read (mtdblk, req->sector << 9,
                        req->current_nr_sectors << 9,
                        req->buffer);
            up(&mtdblk->cache_sem);
            if (!err)
                res = 1;
            break;

            case WRITE:
            // Read only device
            if ( !(mtdblk->mtd->flags & MTD_WRITEABLE) )
                break;

            // Do the write
            down(&mtdblk->cache_sem);
            err = do_cached_write (mtdblk, req->sector << 9,
                        req->current_nr_sectors << 9,
                        req->buffer);
            up(&mtdblk->cache_sem);
            if (!err)
                res = 1;
```

```
                    break;
            }

end_req:
            spin_lock_irq(QUEUE_LOCK(QUEUE));
            end_request(res);
        }
}

int __init init_mtdblock(void)
{
        int i;

        spin_lock_init(&mtdblks_lock);
        /* this lock is used just in kernels >= 2.5.x */
        spin_lock_init(&mtdblock_lock);

#ifdef CONFIG_DEVFS_FS
        if (devfs_register_blkdev(MTD_BLOCK_MAJOR, DEVICE_NAME, &mtd_fops))
        {
            printk(KERN_NOTICE "Can't allocate major number %d for Memory Technology
Devices.\n",
                    MTD_BLOCK_MAJOR);
            return -EAGAIN;
        }

        devfs_dir_handle = devfs_mk_dir(NULL, DEVICE_NAME, NULL);
        register_mtd_user(&notifier);
#else
        if (register_blkdev(MAJOR_NR,DEVICE_NAME,&mtd_fops)) {
            printk(KERN_NOTICE "Can't allocate major number %d for Memory Technology
Devices.\n",
                        MTD_BLOCK_MAJOR);
            return -EAGAIN;
        }
#endif

        /* We fill it in at open() time. */
        for (i=0; i< MAX_MTD_DEVICES; i++) {
            mtd_sizes[i] = 0;
            mtd_blksizes[i] = BLOCK_SIZE;
        }
        init_waitqueue_head(&thr_wq);
        /* Allow the block size to default to BLOCK_SIZE. */
```

```
    blksize_size[MAJOR_NR] = mtd_blksizes;
    blk_size[MAJOR_NR] = mtd_sizes;

    BLK_INIT_QUEUE(BLK_DEFAULT_QUEUE(MAJOR_NR), &mtdblock_request,
&mtdblock_lock);

    kernel_thread (mtdblock_thread, NULL,
CLONE_FS|CLONE_FILES|CLONE_SIGHAND);
    return 0;
}

static void __exit cleanup_mtdblock(void)
{
    leaving = 1;
    wake_up(&thr_wq);
    down(&thread_sem);
#ifdef CONFIG_DEVFS_FS
    unregister_mtd_user(&notifier);
    devfs_unregister(devfs_dir_handle);
    devfs_unregister_blkdev(MTD_BLOCK_MAJOR, DEVICE_NAME);
#else
    unregister_blkdev(MAJOR_NR,DEVICE_NAME);
#endif
    blk_cleanup_queue(BLK_DEFAULT_QUEUE(MAJOR_NR));
    blksize_size[MAJOR_NR] = NULL;
    blk_size[MAJOR_NR] = NULL;
}
```

12.总结

至此，我们可以对 Linux 设备驱动程序的编写作一总结。驱动的编写涉及如下主题：

（1）内核模块、驱动程序的结构；

（2）驱动程序中的并发控制；

（3）驱动程序中的中断处理；

（4）内核模块、驱动程序的结构；

（5）驱动程序中的定时器；

（6）驱动程序中的 I/O 与内存访问；

（7）驱动程序与用户程序的通信。

实际内容相当错综复杂，掌握起来有相当地难度。而本质上，这些内容仅分为两类：（1）设备的访问；（2）设备访问的控制。前者是目的，而为了达到访问的目的，又需要借助并发控制等辅助手段。