

Linux 系统下的多线程编程入门

引言

线程（thread）技术早在 60 年代就被提出，但真正应用多线程到操作系统中去，是在 80 年代中期，solaris 是这方面的佼佼者。传统的 Unix 也支持线程的概念，但是在一个进程（process）中只允许有一个线程，这样多线程就意味着多进程。现在，多线程技术已经被许多操作系统所支持，包括 Windows/NT，当然，也包括 Linux。

为什么有了进程的概念后，还要再引入线程呢？使用多线程到底有哪些好处？什么样的系统应该选用多线程？我们首先必须回答这些问题。

使用多线程的理由之一是和进程相比，它是一种非常"节俭"的多任务操作方式。我们知道，在 Linux 系统下，启动一个新的进程必须分配给它独立的地址空间，建立众多的数据表来维护它的代码段、堆栈段和数据段，这是一种"昂贵"的多任务工作方式。而运行于一个进程中的多个线程，它们彼此之间使用相同的地址空间，共享大部分数据，启动一个线程所花费的空间远远小于启动一个进程所花费的空间，而且，线程间彼此切换所需的时间也远远小于进程间切换所需要的时间。

使用多线程的理由之二是线程间方便的通信机制。对不同进程来说，它们具有独立的数据空间，要进行数据的传递只能通过通信的方式进行，这种方式不仅费时，而且很不方便。线程则不然，由于同一进程下的线程之间共享数据空间，所以一个线程的数据可以直接为其它线程所用，这不仅快捷，而且方便。当然，数据的共享也带来其他一些问题，有的变量不能同时被两个线程所修改，有的子程序中声明为 static 的数据更有可能给多线程程序带来灾难性的打击，这些正是编写多线程程序时最需要注意的地方。

除了以上所说的优点外，不和进程比较，多线程程序作为一种多任务、并发的生活方式，当然有以下的优点：

- 1) 提高应用程序响应。这对图形界面的程序尤其有意义，当一个操作耗时很长时，整个系统都会等待这个操作，此时程序不会响应键盘、鼠标、菜单的操作，而使用多线程技术，将耗时长操作（time consuming）置于一个新的线程，可以避免这种尴尬的情况。
- 2) 使多 CPU 系统更加有效。操作系统会保证当线程数不大于 CPU 数目时，不同的线程运行于不同的 CPU 上。
- 3) 改善程序结构。一个既长又复杂的进程可以考虑分为多个线程，成为几个独立或半独立的运行部分，这样的程序会利于理解和修改。

下面我们先来尝试编写一个简单的多线程程序。

简单的多线程编程

Linux 系统下的多线程遵循 POSIX 线程接口，称为 pthread。编写 Linux 下的多线程程序，需要使用头文件 pthread.h，连接时需要使用库 libpthread.a。顺便说一下，Linux 下 pthread 的实现是通过系统调用 clone（）来实现的。clone（）是 Linux 所特有的系统调用，它的使用方式类似 fork，关于 clone（）的详细情况，有兴趣的读者可以去查看有关文档说明。下面我们展示一个最简单的多线程程序 example1.c。

```
/* example.c */
#include <stdio.h>
#include <pthread.h>
void thread(void)
{
    int i;
    for(i=0;i<3;i++)
        printf("This is a pthread.n");
```

```

}

int main(void)
{
    pthread_t id;
    int i,ret;
    ret=pthread_create(&id,NULL,(void *) thread,NULL);
    if(ret!=0)
    {
        printf("Create pthread error!\n");
        exit(1);
    }
    for(i=0;i<3;i++)
        printf("This is the main process.\n");
    pthread_join(id,NULL);
    return(0);
}

```

我们编译此程序：

```
gcc example1.c -lpthread -o example1
```

运行 example1，我们得到如下结果：

```

This is the main process.
This is a pthread.
This is the main process.
This is the main process.
This is a pthread.
This is a pthread.

```

再次运行，我们可能得到如下结果：

```

This is a pthread.
This is the main process.
This is a pthread.
This is the main process.
This is a pthread.
This is the main process.

```

前后两次结果不一样，这是两个线程争夺 CPU 资源的结果。上面的示例中，我们使用到了两个函数，pthread_create 和 pthread_join，并声明了一个 pthread_t 型的变量。

pthread_t 在头文件/usr/include/bits/pthreadtypes.h 中定义：

```
typedef unsigned long int pthread_t;
```

它是一个线程的标识符。函数 `pthread_create` 用来创建一个线程，它的原型为：

```
extern int pthread_create __P ((pthread_t * __thread, __const pthread_attr_t * __attr, void
*(* __start_routine) (void *), void * __arg));
```

第一个参数为指向线程标识符的指针，第二个参数用来设置线程属性，第三个参数是线程运行函数的起始地址，最后一个参数是运行函数的参数。这里，我们的函数 `thread` 不需要参数，所以最后一个参数设为空指针。第二个参数我们也设为空指针，这样将生成默认属性的线程。对线程属性的设定和修改我们将在下一节阐述。当创建线程成功时，函数返回 0，若不为 0 则说明创建线程失败，常见的错误返回代码为 `EAGAIN` 和 `EINVAL`。前者表示系统限制创建新的线程，例如线程数目过多了；后者表示第二个参数代表的线程属性值非法。创建线程成功后，新创建的线程则运行参数三和参数四确定的函数，原来的线程则继续运行下一行代码。

函数 `pthread_join` 用来等待一个线程的结束。函数原型为：

```
extern int pthread_join __P ((pthread_t __th, void ** __thread_return));
```

第一个参数为被等待的线程标识符，第二个参数为一个用户定义的指针，它可以用来存储被等待线程的返回值。这个函数是一个线程阻塞的函数，调用它的函数将一直等待到被等待的线程结束为止，当函数返回时，被等待线程的资源被收回。一个线程的结束有两种途径，一种是象我们上面的例子一样，函数结束了，调用它的线程也就结束了；另一种方式是通过函数 `pthread_exit` 来实现。它的函数原型为：

```
extern void pthread_exit __P ((void * __retval)) __attribute__ ((__noreturn__));
```

唯一的参数是函数的返回代码，只要 `pthread_join` 中的第二个参数 `thread_return` 不是 `NULL`，这个值将被传递给 `thread_return`。最后要说明的是，一个线程不能被多个线程等待，否则第一个接收到信号的线程成功返回，其余调用 `pthread_join` 的线程则返回错误代码 `ESRCH`。

在这一节里，我们编写了一个最简单的线程，并掌握了最常用的三个函数 `pthread_create`、`pthread_join` 和 `pthread_exit`。下面，我们来了解线程的一些常用属性以及如何设置这些属性。

修改线程的属性

在上一节的例子里，我们用 `pthread_create` 函数创建了一个线程，在这个线程中，我们使用了默认参数，即将该函数的第二个参数设为 `NULL`。的确，对大多数程序来说，使用默认属性就够了，但我们还是有必要来了解一下线程的有关属性。

属性结构为 `pthread_attr_t`，它同样在头文件 `/usr/include/pthread.h` 中定义，喜欢追根问底的人可以自己去看。属性值不能直接设置，须使用相关函数进行操作，初始化的函数为 `pthread_attr_init`，这个函数必须在 `pthread_create` 函数之前调用。属性对象主要包括是否绑定、是否分离、堆栈地址、堆栈大小、优先级。默认的属性为非绑定、非分离、缺省 1M 的堆栈、与父进程同样级别的优先级。

关于线程的绑定，牵涉到另外一个概念：轻进程（LWP: Light Weight Process）。轻进程可以理解为内核线程，它位于用户层和系统层之间。系统对线程资源的分配、对线程的控制是通过轻进程来实现的，一个轻进程可以控制一个或多个线程。默认状况下，启动多少轻进程、哪些轻进程来控制哪些线程是由系统来控制的，这种状况即称为非绑定的。绑定状况下，则顾名思义，即某个线程固定的“绑”在一个轻进程之上。被绑定的线程具有较高的响应速度，这是因为 CPU 时间片的调度是面向轻进程的，绑定的线程可以保证在需要的时候它总有一个轻进程可用。通过设置被绑定的轻进程的优先级和调度级可以使得绑定的线程满足诸如实时反应之类的要求。

设置线程绑定状态的函数为 `pthread_attr_setscope`，它有两个参数，第一个是指向属性结构的指针，第二个是绑定类型，它有两个取值：`PTHREAD_SCOPE_SYSTEM`（绑定的）和 `PTHREAD_SCOPE_PROCESS`（非绑定的）。下面的代码即创建了一个绑定的线程。

```
#include <pthread.h>
pthread_attr_t attr;
pthread_t tid;

/*初始化属性值，均设为默认值*/
pthread_attr_init(&attr);
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

pthread_create(&tid, &attr, (void *) my_function, NULL);
```

线程的分离状态决定一个线程以什么样的方式来终止自己。在上面的例子中，我们采用了线程的默认属性，即为非分离状态，这种情况下，原有的线程等待创建的线程结束。只有当 `pthread_join()` 函数返回时，创建的线程才算终止，才能释放自己占用的系统资源。而分离线程不是这样子的，它没有被其他的线程所等待，自己运行结束了，线程也就终止了，马上释放系统资源。程序员应该根据自己的需要，选择适当的分离状态。设置线程分离状态的函数为 `pthread_attr_setdetachstate (pthread_attr_t *attr, int detachstate)`。第二个参数可选为 `PTHREAD_CREATE_DETACHED`（分离线程）和 `PTHREAD_CREATE_JOINABLE`（非分离线程）。这里要注意的一点是，如果设置一个线程为分离线程，而这个线程运行又非常快，它很可能在 `pthread_create` 函数返回之前就终止了，它终止以后就可能将线程号和系统资源移交给其他的线程使用，这样调用 `pthread_create` 的线程就得到了错误的线程号。要避免这种情况可以采取一定的同步措施，最简单的方法之一是在被创建的线程里调用 `pthread_cond_timewait` 函数，让这个线程等待一会儿，留出足够的时间让函数 `pthread_create` 返回。设置一段等待时间，是在多线程编程里常用的方法。但是注意不要使用诸如 `wait()` 之类的函数，它们是使整个进程睡眠，并不能解决线程同步的问题。

另外一个可能常用的属性是线程的优先级，它存放在结构 `sched_param` 中。用函数 `pthread_attr_getschedparam` 和函数 `pthread_attr_setschedparam` 进行存放，一般说来，我们总是先取优先级，对取得的值修改后再存放回去。下面即是一段简单的例子。

```
#include <pthread.h>
#include <sched.h>
pthread_attr_t attr;
pthread_t tid;
sched_param param;
int newprio=20;

pthread_attr_init(&attr);
pthread_attr_getschedparam(&attr, &param);
param.sched_priority=newprio;
pthread_attr_setschedparam(&attr, &param);
pthread_create(&tid, &attr, (void *)myfunction, myarg);
```

线程的数据处理

和进程相比，线程的最大优点之一是数据的共享性，各个进程共享父进程处沿袭的数据段，可以方便的获得、修改数据。但这也给多线程编程带来了许多问题。我们必须当心有多个不同的进程访问相同的变量。许多函数是不可重入的，即同时不能运行一个函数的多个拷贝（除非使用不同的数据段）。在函数中声明的静态变量常常带来问题，函数的返回值也会有问题。因为如果返回的是函数内部静态声明的空间的地址，则在一个线程调用该函数得到地址后使用该地址指向的数据时，别的线程可能调用此函数并修改了这一段数据。在进程中共享的变量必须用关键字 `volatile` 来定义，这是为了防止编译器在优化时（如 `gcc` 中使用 `-Ox` 参数）改变它们的使用方式。为了保护变量，我们必须使用信号量、互斥等方法来保证我们对变量的正确使用。下面，我们就逐步介绍处理线程数据时的有关知识。

1、线程数据

在单线程的程序里，有两种基本的数据：全局变量和局部变量。但在多线程程序里，还有第三种数据类型：线程数据（TSD: Thread-Specific Data）。它和全局变量很象，在线程内部，各个函数可以象使用全局变量一样调用它，但它对线程外部的其它线程是不可见的。这种数据的必要性是显而易见的。例如我们常见的变量 `errno`，它返回标准的出错信息。它显然不能是一个局部变量，几乎每个函数都应该可以调用它；但它又不能是一个全局变量，否则在 A 线程里输出的很可能是 B 线程的出错信息。要实现诸如此类的变量，我们就必须使用线程数据。我们为每个线程数据创建一个键，它和这个键相关联，在各个线程里，都使用这个键来指代线程数据，但在不同的线程里，这个键代表的的数据是不同的，在同一个线程里，它代表同样的数据内容。

和线程数据相关的函数主要有 4 个：创建一个键；为一个键指定线程数据；从一个键读取线程数据；删除键。

创建键的函数原型为：

```
extern int pthread_key_create __P ((pthread_key_t * __key, void (* __destr_function)
(void *)));
```

第一个参数为指向一个键值的指针，第二个参数指明了一个 `destructor` 函数，如果这个参数不为空，那么当每个线程结束时，系统将调用这个函数来释放绑定在这个键上的内存块。这个函数常和函数 `pthread_once ((pthread_once_t * __once_control, void (* __init_routine) (void)))` 一起使用，为了让这个键只被创建一次。函数 `pthread_once` 声明一个初始化函数，第一次调用 `pthread_once` 时它执行这个函数，以后的调用将被它忽略。

在下面的例子中，我们创建一个键，并将它和某个数据相关联。我们要定义一个函数 `createWindow`，这个函数定义一个图形窗口（数据类型为 `Fl_Window *`，这是图形界面开发工具 `FLTK` 中的数据类型）。由于各个线程都会调用这个函数，所以我们使用线程数据。

```
/* 声明一个键*/
pthread_key_t myWinKey;
/* 函数 createWindow */
void createWindow ( void ) {
    Fl_Window * win;
    static pthread_once_t once= PTHREAD_ONCE_INIT;
    /* 调用函数 createMyKey, 创建键*/
    pthread_once ( & once, createMyKey );
    /* win 指向一个新建立的窗口*/
```

```

win=new Fl_Window( 0, 0, 100, 100, "MyWindow");
/* 对此窗口作一些可能的设置工作，如大小、位置、名称等*/
setWindow(win);
/* 将窗口指针值绑定在键 myWinKey 上*/
pthread_setspecific ( myWinKey, win);
}

/* 函数 createMyKey，创建一个键，并指定了 destructor */
void createMyKey ( void ) {
    pthread_keycreate(&myWinKey, freeWinKey);
}

/* 函数 freeWinKey，释放空间*/
void freeWinKey ( Fl_Window * win){
    delete win;
}

```

这样，在不同的线程中调用函数 `createMyWin`，都可以得到在线程内部均可见的窗口变量，这个变量通过函数 `pthread_getspecific` 得到。在上面的例子中，我们已经使用了函数 `pthread_setspecific` 来将线程数据和一个键绑定在一起。这两个函数的原型如下：

```

extern int pthread_setspecific __P ((pthread_key_t __key, __const void
* __pointer));
extern void *pthread_getspecific __P ((pthread_key_t __key));

```

这两个函数的参数意义和使用方法是显而易见的。要注意的是，用 `pthread_setspecific` 为一个键指定新的线程数据时，必须自己释放原有的线程数据以回收空间。这个过程函数 `pthread_key_delete` 用来删除一个键，这个键占用的内存将被释放，但同样要注意的是，它只释放键占用的内存，并不释放该键关联的线程数据所占用的内存资源，而且它也不会触发函数 `pthread_key_create` 中定义的 `destructor` 函数。线程数据的释放必须在释放键之前完成。

2、互斥锁

互斥锁用来保证一段时间内只有一个线程在执行一段代码。必要性显而易见：假设各个线程向同一个文件顺序写入数据，最后得到的结果一定是灾难性的。

我们先看下面一段代码。这是一个读/写程序，它们公用一个缓冲区，并且我们假定一个缓冲区只能保存一条信息。即缓冲区只有两个状态：有信息或没有信息。

```

void reader_function ( void );
void writer_function ( void );

char buffer;
int buffer_has_item=0;
pthread_mutex_t mutex;
struct timespec delay;
void main ( void ){

```



```

pthread_t reader;
/* 定义延迟时间*/
delay.tv_sec = 2;
delay.tv_nsec = 0;
/* 用默认属性初始化一个互斥锁对象*/
pthread_mutex_init (&mutex, NULL);
pthread_create(&reader, pthread_attr_default, (void *)&reader_function, NULL);
writer_function( );
}

void writer_function (void){
    while(1){
        /* 锁定互斥锁*/
        pthread_mutex_lock (&mutex);
        if (buffer_has_item==0){
            buffer=make_new_item( );
            buffer_has_item=1;
        }
        /* 打开互斥锁*/
        pthread_mutex_unlock(&mutex);
        pthread_delay_np(&delay);
    }
}

void reader_function(void){
    while(1){
        pthread_mutex_lock(&mutex);
        if(buffer_has_item==1){
            consume_item(buffer);
            buffer_has_item=0;
        }
        pthread_mutex_unlock(&mutex);
        pthread_delay_np(&delay);
    }
}
}

```

这里声明了互斥锁变量 `mutex`，结构 `pthread_mutex_t` 为不公开的数据类型，其中包含一个系统分配的属性对象。函数 `pthread_mutex_init` 用来生成一个互斥锁。NULL 参数表明使用默认属性。如果需要声明特定属性的互斥锁，须调用函数 `pthread_mutexattr_init`。函数 `pthread_mutexattr_setpshared` 和函数 `pthread_mutexattr_settype` 用来设置互斥锁属性。前一个函数设置属性 `pshared`，它有两个取值，`PTHREAD_PROCESS_PRIVATE` 和 `PTHREAD_PROCESS_SHARED`。前者用来不同进程中的线程同步，后者用于同步本进程的不同线程。在上面的例子中，我们使用的是默认属性 `PTHREAD_PROCESS_PRIVATE`。后者用来设置互斥锁类型，可选的类型有 `PTHREAD_MUTEX_NORMAL`、

PTHREAD_MUTEX_ERRORCHECK、PTHREAD_MUTEX_RECURSIVE 和 PTHREAD_MUTEX_DEFAULT。它们分别定义了不同的上锁、解锁机制，一般情况下，选用最后一个默认属性。

pthread_mutex_lock 声明开始用互斥锁上锁，此后的代码直至调用 pthread_mutex_unlock 为止，均被上锁，即同一时间只能被一个线程调用执行。当一个线程执行到 pthread_mutex_lock 处时，如果该锁此时被另一个线程使用，那此线程被阻塞，即程序将等待到另一个线程释放此互斥锁。在上面的例子中，我们使用了 pthread_delay_np 函数，让线程睡眠一段时间，就是为了防止一个线程始终占据此函数。

上面的例子非常简单，就不再介绍了，需要提出的是在使用互斥锁的过程中很有可能会出现死锁：两个线程试图同时占用两个资源，并按不同的次序锁定相应的互斥锁，例如两个线程都需要锁定互斥锁 1 和互斥锁 2，a 线程先锁定互斥锁 1，b 线程先锁定互斥锁 2，这时就出现了死锁。此时我们可以使用函数 pthread_mutex_trylock，它是函数 pthread_mutex_lock 的非阻塞版本，当它发现死锁不可避免时，它会返回相应的信息，程序员可以针对死锁做出相应的处理。另外不同的互斥锁类型对死锁的处理不一样，但最主要的还是要程序员自己在程序设计注意这一点。

3、条件变量

前一节中我们讲述了如何使用互斥锁来实现线程间数据的共享和通信，互斥锁一个明显的缺点是它只有两种状态：锁定和非锁定。而条件变量通过允许线程阻塞和等待另一个线程发送信号的方法弥补了互斥锁的不足，它常和互斥锁一起使用。使用时，条件变量被用来阻塞一个线程，当条件不满足时，线程往往解开相应的互斥锁并等待条件发生变化。一旦其它的某个线程改变了条件变量，它将通知相应的条件变量唤醒一个或多个正被此条件变量阻塞的线程。这些线程将重新锁定互斥锁并重新测试条件是否满足。一般说来，条件变量被用来进行线程间的同步。

条件变量的结构为 pthread_cond_t，函数 pthread_cond_init() 被用来初始化一个条件变量。它的原型为：

```
extern int pthread_cond_init __P ((pthread_cond_t * __cond, __const pthread_condattr_t * __cond_attr));
```

其中 cond 是一个指向结构 pthread_cond_t 的指针，cond_attr 是一个指向结构 pthread_condattr_t 的指针。结构 pthread_condattr_t 是条件变量的属性结构，和互斥锁一样我们可以用它来设置条件变量是进程内可用还是进程间可用，默认值是 PTHREAD_PROCESS_PRIVATE，即此条件变量被同一进程内的各个线程使用。注意初始化条件变量只有未被使用时才能重新初始化或被释放。释放一个条件变量的函数为 pthread_cond_destroy (pthread_cond_t cond)。

函数 pthread_cond_wait() 使线程阻塞在一个条件变量上。它的函数原型为：

```
extern int pthread_cond_wait __P ((pthread_cond_t * __cond, pthread_mutex_t * __mutex));
```

线程解开 mutex 指向的锁并被条件变量 cond 阻塞。线程可以被函数 pthread_cond_signal 和函数 pthread_cond_broadcast 唤醒，但是要注意的是，条件变量只是起阻塞和唤醒线程的作用，具体的判断条件还需用户给出，例如一个变量是否为 0 等等，这一点我们从后面的例子中可以看到。线程被唤醒后，它将重新检查判断条件是否满足，如果还不满足，一般说来线程应该仍阻塞在这里，被等待被下一次唤醒。这个过程一般用 while 语句实现。

另一个用来阻塞线程的函数是 pthread_cond_timedwait()，它的原型为：

```
extern int pthread_cond_timedwait __P ((pthread_cond_t * __cond, pthread_mutex_t * __mutex, __const struct timespec * __abstime));
```

它比函数 pthread_cond_wait() 多了一个时间参数，经历 abstime 段时间后，即使条件

变量不满足，阻塞也被解除。

函数 `pthread_cond_signal()` 的原型为：

```
extern int pthread_cond_signal __P((pthread_cond_t * __cond));
```

它用来释放被阻塞在条件变量 `cond` 上的一个线程。多个线程阻塞在此条件变量上时，哪一个线程被唤醒是由线程的调度策略所决定的。要注意的是，必须用保护条件变量的互斥锁来保护这个函数，否则条件满足信号又可能在测试条件和调用 `pthread_cond_wait` 函数之间被发出，从而造成无限制的等待。下面是使用函数 `pthread_cond_wait()` 和函数 `pthread_cond_signal()` 的一个简单的例子。

```
pthread_mutex_t count_lock;
pthread_cond_t count_nonzero;
unsigned count;
decrement_count () {
    pthread_mutex_lock (&count_lock);
    while(count==0)
        pthread_cond_wait( &count_nonzero, &count_lock);
    count=count -1;
    pthread_mutex_unlock (&count_lock);
}

increment_count(){
    pthread_mutex_lock(&count_lock);
    if(count==0)
        pthread_cond_signal(&count_nonzero);
    count=count+1;
    pthread_mutex_unlock(&count_lock);
}
```

`count` 值为 0 时, `decrement` 函数在 `pthread_cond_wait` 处被阻塞, 并打开互斥锁 `count_lock`。此时, 当调用到函数 `increment_count` 时, `pthread_cond_signal()` 函数改变条件变量, 告知 `decrement_count()` 停止阻塞。读者可以试着让两个线程分别运行这两个函数, 看看会出现什么样的结果。

函数 `pthread_cond_broadcast(pthread_cond_t *cond)` 用来唤醒所有被阻塞在条件变量 `cond` 上的线程。这些线程被唤醒后将再次竞争相应的互斥锁, 所以必须小心使用这个函数。

4、信号量

信号量本质上是一个非负的整数计数器, 它被用来控制对公共资源的访问。当公共资源增加时, 调用函数 `sem_post()` 增加信号量。只有当信号量值大于 0 时, 才能使用公共资源, 使用后, 函数 `sem_wait()` 减少信号量。函数 `sem_trywait()` 和函数 `pthread_mutex_trylock()` 起同样的作用, 它是函数 `sem_wait()` 的非阻塞版本。下面我们逐个介绍和信号量有关的一些函数, 它们都在头文件 `/usr/include/semaphore.h` 中定义。

信号量的数据类型为结构 `sem_t`, 它本质上是一个长整型的数。函数 `sem_init()` 用来初始化一个信号量。它的原型为:

```
extern int sem_init __P((sem_t * __sem, int __pshared, unsigned int __value));
```

sem 为指向信号量结构的一个指针；pshared 不为 0 时此信号量在进程间共享，否则只能为当前进程的所有线程共享；value 给出了信号量的初始值。

函数 sem_post(sem_t *sem)用来增加信号量的值。当有线程阻塞在这个信号量上时，调用这个函数会使其中的一个线程不再阻塞，选择机制同样是由线程的调度策略决定的。

函数 sem_wait(sem_t *sem)被用来阻塞当前线程直到信号量 sem 的值大于 0，解除阻塞后将 sem 的值减一，表明公共资源经使用后减少。函数 sem_trywait (sem_t *sem)是函数 sem_wait () 的非阻塞版本，它直接将信号量 sem 的值减一。

函数 sem_destroy(sem_t *sem)用来释放信号量 sem。

下面我们来看一个使用信号量的例子。在这个例子中，一共有 4 个线程，其中两个线程负责从文件读取数据到公共的缓冲区，另两个线程从缓冲区读取数据作不同的处理（加和乘运算）。

```
/* File sem.c */
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#define MAXSTACK 100
int stack[MAXSTACK][2];
int size=0;
sem_t sem;
/* 从文件 1.dat 读取数据，每读一次，信号量加一*/
void ReadData1(void){
    FILE *fp=fopen("1.dat","r");
    while(!feof(fp)){
        fscanf(fp,"%d %d",&stack[size][0],&stack[size][1]);
        sem_post(&sem);
        ++size;
    }
    fclose(fp);
}
/*从文件 2.dat 读取数据*/
void ReadData2(void){
    FILE *fp=fopen("2.dat","r");
    while(!feof(fp)){
        fscanf(fp,"%d %d",&stack[size][0],&stack[size][1]);
        sem_post(&sem);
        ++size;
    }
    fclose(fp);
}
/*阻塞等待缓冲区有数据，读取数据后，释放空间，继续等待*/
void HandleData1(void){
    while(1){
        sem_wait(&sem);
```

```

        printf("Plus:%d+%d=%dn",stack[size][0],stack[size][1],
        stack[size][0]+stack[size][1]);
        --size;
    }
}

void HandleData2(void){
    while(1){
        sem_wait(&sem);
        printf("Multiply:%d*%d=%dn",stack[size][0],stack[size][1],
        stack[size][0]*stack[size][1]);
        --size;
    }
}

int main(void){
    pthread_t t1,t2,t3,t4;
    sem_init(&sem,0,0);
    pthread_create(&t1,NULL,(void *)HandleData1,NULL);
    pthread_create(&t2,NULL,(void *)HandleData2,NULL);
    pthread_create(&t3,NULL,(void *)ReadData1,NULL);
    pthread_create(&t4,NULL,(void *)ReadData2,NULL);
    /* 防止程序过早退出，让它在此无限期等待*/
    pthread_join(t1,NULL);
}

```

在 Linux 下，我们用命令 `gcc -lpthread sem.c -o sem` 生成可执行文件 `sem`。我们事先编辑好数据文件 `1.dat` 和 `2.dat`，假设它们的内容分别为 `1 2 3 4 5 6 7 8 9 10` 和 `-1 -2 -3 -4 -5 -6 -7 -8 -9 -10`，我们运行 `sem`，得到如下的结果：

```

Multiply:-1*-2=2
Plus:-1+-2=-3
Multiply:9*10=90
Plus:-9+-10=-19
Multiply:-7*-8=56
Plus:-5+-6=-11
Multiply:-3*-4=12
Plus:9+10=19
Plus:7+8=15
Plus:5+6=11

```

从中我们可以看出各个线程间的竞争关系。而数值并未按我们原先的顺序显示出来这是由于 `size` 这个数值被各个线程任意修改的缘故。这也往往是多线程编程要注意的问题。

小结

多线程编程是一个很有意思也很有用的技术，使用多线程技术的网络蚂蚁是目前最常用的下载工具之一，使用多线程技术的 `grep` 比单线程的 `grep` 要快上几倍，类似的例子还有很

多。希望大家能用多线程技术写出高效实用的好程序来。

共 2 页。 [912](#)

资深 Linux 程序员的开发经验谈

Spence Murray 是 Linux 开发高手之一，同时长期以来他一直是 UNIX 的坚定支持者。本文介绍的是 Murray 和他在 Codemonks Consulting 的同事在日常的 Linux 开发以及应用服务工作中用到的基本技术: shell 脚本，相信 Linux 的开发人员都会受益于这项有用而且通用的技术。

Spence Murray 是 Codemonks Consulting 的创始人之一，自从 20 世纪 80 年代最早在 SunOS 上编写代码到现在，一直致力于 UNIX/Linux 的开发。从那时起，他曾在 IBM 公司的 AIX、SGI 公司的 Irix 工作，长时间地编写跨平台的 UNIX 代码，包括 HP/UX, Irix, Solaris/SunOS, SCO UNIX, 各种 BSD, MacOS X, 当然，还有 Linux。从图形/视频设备驱动程序到 UI 代码，他什么工作都做过。Murray 编写的跨平台代码包括 X Window System Xserver 代码，以及作为 Netscape Navigator 一部分的核心浏览器代码。

Murray 最经常使用的 Linux 工具是 vi、bash 和 Emacs。“不论我是在写 C、C++、Java、shell 脚本，还是 HTML，大部分的时间我都在这些工具中来回切换”，他说。

Linux 秘密武器

Murray 认为，对一个 Linux 开发人员来说，shell 是一个强大的软件开发工具，无论怎么评价都不过分。“在我做的每一项工作中都要用到 shell 脚本，不论是快速地阅读和修改普通文本还是编写代码”，他说。“它轻便而快捷，它短小的命令使得来回移动代码称为一个迅速而没有痛苦的过程。作为一名编辑，它很快就会成为第二本能”。

对 Murray 来说，Emacs 作为一个开发工具出现的晚了一些。“在 90 年代早期，我尝试使用 Emacs 作为一个 IDE，并很快就转换门厅。Emacs 非常强大，在那些日子里，我会一直开着一个 Emacs 窗口，经常打开几十个源文件，每个都有我编辑的上下文、使用 gdb 的调试会话以及在不同的源目录下运行的 bash 脚本。有很多关于 Emacs 的资料，可以说，这是个可怕的工具...再者，您可以在任何您想要花时间去开发开发的系统上运行 Emacs。

自从 20 世纪 80 年代中期第一次使用 SunOS 支持的 vi 这个简洁的环境以来，Emacs 编辑器已经成为了 Murray 的标准工具。“它在各种流派的 UNIX 上都可以使用，这是我在致力于跨平台的开发工作时选择它的主要原因之一”，他说。

Linux 开发人员：了解您的 shell

Murray 要求您要了解您的 shell。“Bash、tcsh、csh——shell 是您最基本的软件开发工具”，他强调说。“它可以做许多了不起的事情。所有的工作都要依赖于它.....和它的强大功能”。作为说明通用的 shell 脚本功能强大的例子，在参考资料部分中有一个可以下载的文件，其中有一组脚本，用于获得 Red Hat 发行的更新 RPM 软件包并将它们合并到原来的软件包和定制的软件包。下载文件并解压缩后，您可以在 /developerworks/rpm_update_scripts 目录下找到脚本。最终结果是一个包括所有软件包最新版本的目录和一个用于网络安装的升级的 hdlist 文件。

下面的代码片段实现的是对 Red Hat RPM 软件包的自动更新，以创建一个使用最新的 RPM 的可以安装版本。这对任何一个维护公共 Linux 服务器的人来说是一个基本的步骤。就我们而言，我们通常是维护许多公共 Linux 服务器上的大量网络服务。下面是可以自动完成更新最新的安全和功能的过程的部分脚本。

下面的脚本样例证明了普通的 shell 编程技术可以广泛应用于各种系统配置和程序设计应用。脚本使用的是 bourne shell，它是在不同的 UNIX 系统中最为常见的 shell。这样

就可以保证这些非常轻便的代码可以稍加修改或者不加修改地在不同的 UNIX 系统上使用。修改 Red Hat 软件包的规范以应用于其它 Linux 发行版本是很容易的。

freshen.sh 使用指定的 RPM ftp 更新站点上的 RPM 软件包来更新原有的 RPM 列表。执行过滤器来替换更新 RPM 软件包。最后，长长的发行列表根据从更新镜像站点上得到的新 RPM 软件包完成更新。

清单 1. fresh.sh

```
#!/bin/sh
rh_ver=$1
rh_path=$2
update_dir=${rh_path}/RH${rh_ver}-updates
custom_dir=${rh_path}/RH${rh_ver}-custom
install_dir=${rh_path}/RH${rh_ver}-install

# Sanity check for the original directory.

# Create update and install directories if they don't exist

[ -d ${update_dir} ] || mkdir ${update_dir}
[ -d ${install_dir}/RedHat/RPMS ] || mkdir -p ${install_dir}/RedHat/RPMS

# Get latest updates from fresh rpms FTP site

./get_update.sh ${rh_ver} ${update_dir}

# Create/update hardlinks from update, and custom directories
# to the install directory. We assume that original RPMS are already
# hardlinked to the install directory, so all we need to do is filter
# out any replaced by updated packages.

./do-links.sh ${update_dir} ${install_dir}/RedHat/RPMS
[ -d ${custom_dir} ] && ./do-links.sh ${custom_dir}
${install_dir}/RedHat/RPMS

# Filter out all but the latest version of everything.

./filter-rpms.pl $install_dir/RedHat/RPMS

# Rebuild the hard disk lists
/usr/lib/anaconda-runtime/genhdlist ${install_dir}
```

freshen.sh 调用 do-links.sh 和 get_update.sh，分别去设置 RPM 发行版本的源、宿(省略了源 RPM 软件包；硬链接用来设置目的 RPM)和检索更新。

清单 2. do-links.sh

```
#!/bin/sh

src=$1
dest=$2

#for file in $src/*; do
for file in `find $src -name *.rpm -a ! -name *.src.rpm -print`; do
base=`basename $file`;
if test ! -f $dest/$base; then
echo "Linking $file";
ln $file $dest
else
echo "EXISTS: $file";
fi
done
```

清单 3. get_update.sh

```
#!/bin/sh
rh_ver=$1
dest=$2
echo "Retrieving updates for version ${rh_ver} to $dest"
lftp << EOF
open ftp.freshrpms.net

mirror -n pub/redhat/linux/updates/${rh_ver}/en/os/i386 $dest/i386
mirror -n pub/redhat/linux/updates/${rh_ver}/en/os/i486 $dest/i486
mirror -n pub/redhat/linux/updates/${rh_ver}/en/os/i586 $dest/i568
mirror -n pub/redhat/linux/updates/${rh_ver}/en/os/i686 $dest/i686
mirror -n pub/redhat/linux/updates/${rh_ver}/en/os/SRPMS $dest/SRPMS
mirror -n pub/redhat/linux/updates/${rh_ver}/en/os/noarch $dest/noarch
```

Java 和 Linux

在 Codemonks，相当多的开发工作是在 Linux 上用 Java 完成。这两个工具的组合为创建商业级质量的 Web 应用提供了一个平台，Murray 说。“在做这些项目的过程中，我们发现我们要总体上了解客户已有的应用代码”，他回忆说。locks.c (在下载得到的压缩文件中的 /developerworks/locks 目录下) 是一个代码片段，实现的是用于 Java Virtual Machine Profiler Interface (JVMPI) 的读/写锁以及大量的调试代码。

Linux 开发人员的代表

“在情况允许的时候，不要写特定于系统的代码”，Murray 说，而是克服困难去“写好的跨平台的代码”。受雇的 Murray 坚持认为他最大的资本永远是“写具有商业品质的代码，构

建和提供网络服务，定制 OS 或内核，而且完全基于可靠的开放源代码的平台”。

下面是一个代码片段，来自于一个跨平台的定制的 IMAP 服务器，这个服务器由 Linux 和 MacOS X 的开发人员共同开发。代码实现的是一个用来处理字符串的简单的增长缓存。这样避免了缓存溢出的问题(不要忘记那些安全漏洞)，而不必要您每次做某些事情的时候重新分配空间。它是通过维护一个简单的可变长的缓存来实现的，这个缓存可以写满和清空。这个缓存已经被用于一个实验用的 IMAP 服务器，这个服务器是由一个团队紧张工作了一周完成的。

除了一个简单的字符串缓冲区的实现之外，这段代码还实现了一个可变大小的字符串数组。它完成的是一个简单的接口，当您写完一个字符串以后，您可以标记它然后继续写下一个。此外，这样会节约空间分配，并且将比较乱的代码组织到一起。

完整的 IMAP 服务器的代码将在今年某个时间发布。

清单 4. 定制的 IMAP 服务器一部分

```
#ifndef HOED_BUF_H
#define HOED_BUF_H

typedef struct {
    char *str;
    int size;
    int length;
    int str_start;
    int max_size;

    int n_strings;
    int size_strings;
    int *str_posn;
    char **str_set;
} hoed_buf_t;

#if __GNUC__ > 2 || (__GNUC__ == 2 && __GNUC_MINOR__ > 4)
#define PRINTF(f, a) __attribute__((format (printf, f, a)))
#else
#define PRINTF(f,a)
#endif

extern hoed_buf_t *hoed_buf_alloc(int init_size, int max_size);
extern void hoed_buf_free(hoed_buf_t *);
extern void hoed_buf_reset(hoed_buf_t *);
extern void hoed_buf_new_string(hoed_buf_t *);
extern char **hoed_buf_get_set(hoed_buf_t *, int *n_string);

extern char *hoed_buf_put_char(hoed_buf_t *, char toadd);
extern char *hoed_buf_sprintf(hoed_buf_t *, const char *format,...)
```

```
PRINTF(2,3);
extern char *hoed_buf_strcat(hoed_buf_t *, const char *append);
extern char *hoed_buf_cat_sprintf(hoed_buf_t *, const char *format, ...)
    PRINTF(2,3);

#endif /* HOED_BUF_H */
```

称为杀手级的 Linux 应用程序

对 Murray 来说，有两个杀手级的 Linux 应用程序：Emacs 和 Netscape Navigator。“Emacs 或许是给人印象最深而且广为应用的基于 Linux 的应用程序”，他说。“另一个是 Netscape Navigator。有一次，我们要支持 20 多种 UNIX，我在 Linux 上完成了所有的工作”。

他继续说，“有趣的是，基于 Linux 的应用程序可能运行于许多不同风格的 UNIX 系统上，甚至安装了 Cygwin 的 Windows 系统。”

Linux 的未来如何？

当前，Murray 正在进行的 Linux 项目有好几个，从支持电子邮件、消息和共享数据库的分布式办公应用程序到使用标准工具的网络应用程序(标准工具包括：Apache/Tomcat, PHP, PostgreSQL, MySQL, 和 Linux)。Murray 有他自己的公司专门为网络服务和网络应用提供主机服务。

对 Murray 来说，得益于 Linux 强大功能的应用程序的列表在不断地增长。“有很多”，他说。“Oracle, WebSphere, Apache, PostgreSQL, MySQL, Cyrus IMAP... 这个列表很长而且在不断增长。”

对 Murray 来说，Linux 到此为止了。“我们所有的服务器都运行 Linux；不管目标平台如何，我们主要的开发都在 Linux 上进行；我们把 Linux 推荐给用户来运行服务器应用程序”，他说。“Linux 快速发展的步伐，开放源代码组织对它的广泛支持，低廉的开发费用，如果把这些结合在一起，您就知道它是一个难以击败的平台”。

深入浅出 Linux 设备驱动编程之引言

2006-10-16 13:00 作者： 宋宝华 出处： 天极开发 责任编辑： [方舟](#)

相关专题： [Linux 设备驱动程序开发入门](#)

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开发工程师的层次，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：



深入浅出 Linux 设备驱动编程之内核模块

2006-10-17 15:36 作者： 宋宝华 出处： 天极开发 责任编辑： [方舟](#)

相关专题： [Linux 设备驱动程序开发入门](#)

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 and 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。

深入浅出 Linux 设备驱动之字符设备驱动程序

2006-10-18 13:51 作者： 宋宝华 出处： 天极软件 责任编辑： 方舟

相关专题： [Linux 设备驱动程序开发入门](#)

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"globalvar"。对"globalvar"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 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 for device，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。

lseek()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 (*lseek) (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。

深入浅出 **Linux** 设备驱动之字符设备驱动程序

2006-10-18 13:51 作者： 宋宝华 出处： 天极软件 责任编辑： [方舟](#)

设备"globalvar"的驱动程序的这些函数应分别命名为 gbalvar_open、gbalvar_release、gbalvar_read、gbalvar_write、gbalvar_ioctl，因此设备"globalvar"的基本入口点结构变量 gbalvar_fops 赋值如下：

```
struct file_operations gbalvar_fops = {
    read: gbalvar_read,
    write: gbalvar_write,
};
```

上述代码中对 gbalvar_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 gbalvar_read(struct file *, char *, size_t, loff_t*);
static ssize_t gbalvar_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 gbalvar_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
```

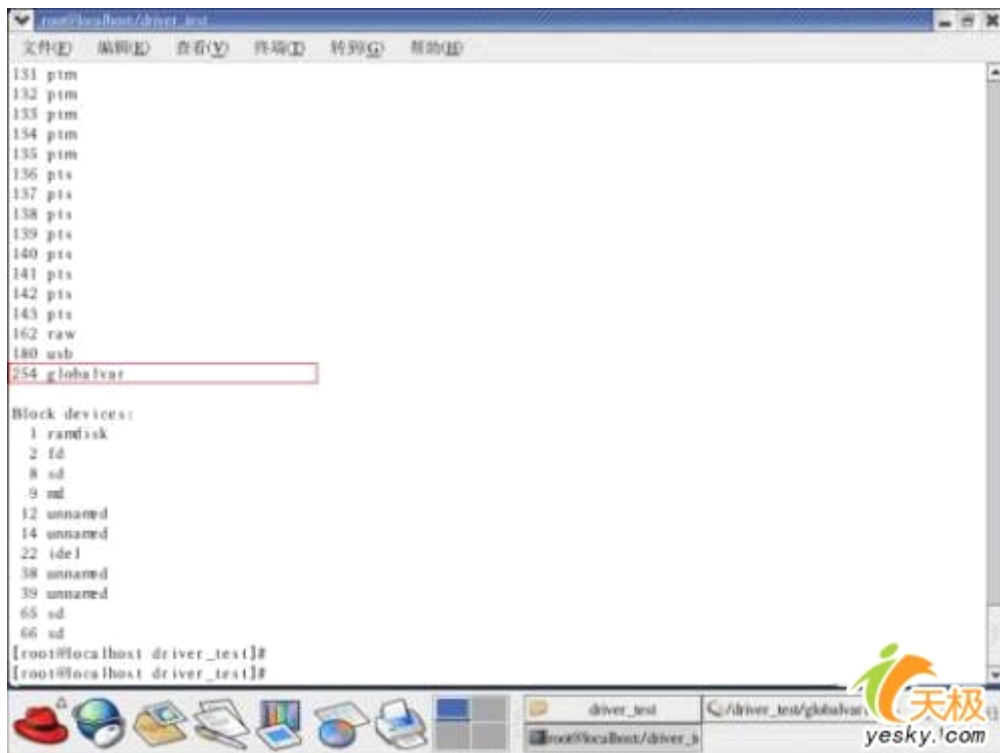
编译代码，运行：

```
insmod 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。

深入浅出 Linux 设备驱动之并发控制

2006-10-20 16:05 作者： 宋宝华 出处： 天极开发 责任编辑： [方舟](#)

相关专题： [Linux 设备驱动程序开发入门](#)

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。

深入浅出 Linux 设备驱动之并发控制

2006-10-20 16:05 作者： 宋宝华 出处： 天极开发 责任编辑： [方舟](#)

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))

```

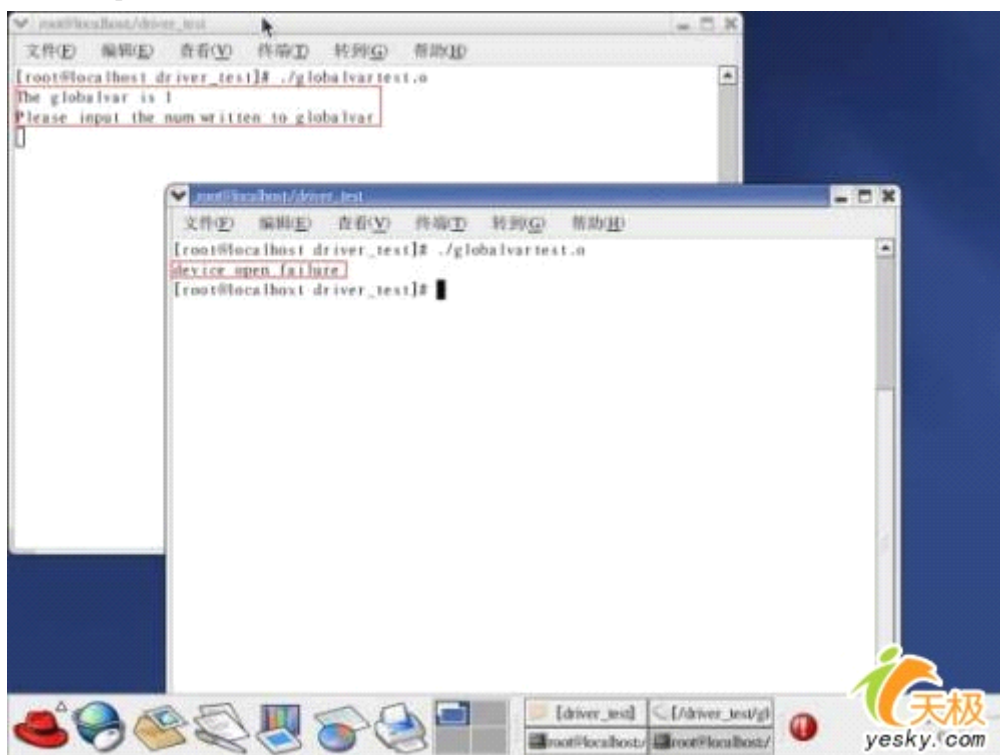
```

{
    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：



2006-10-22 07:00 作者： 宋宝华 出处： 天极软件 责任编辑： [方舟](#)

相关专题： [Linux 设备驱动程序开发入门](#)

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：

```

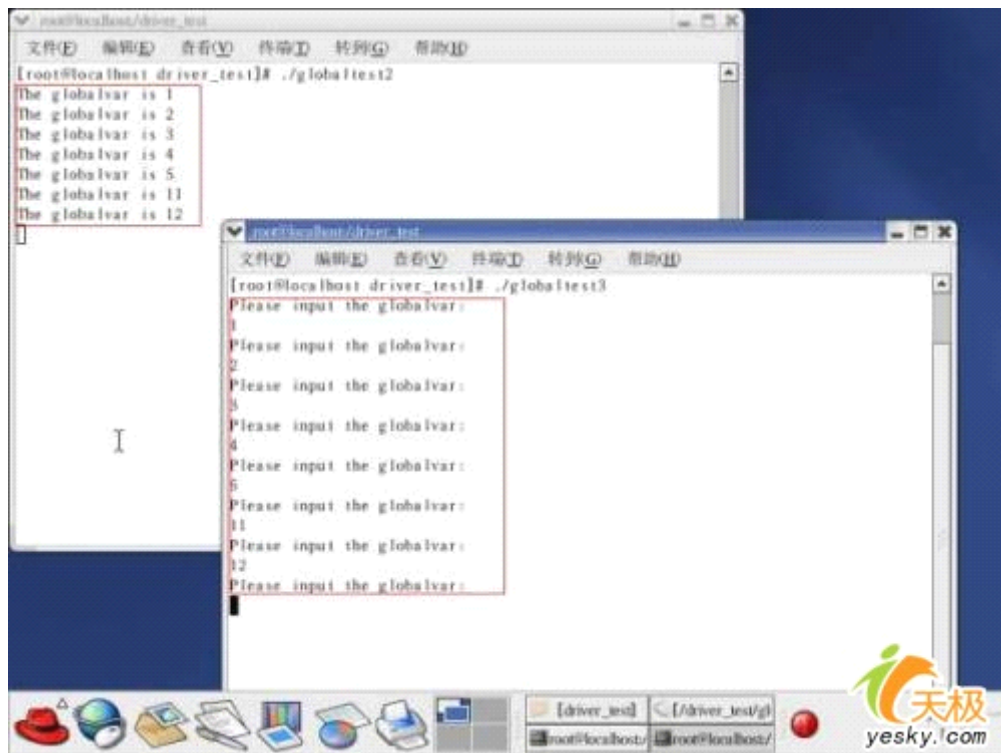
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>
main()
{
    int fd, num;

    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：



The image shows two terminal windows on a Linux desktop. The top window, titled 'root@localhost: driver_test', shows the output of a program that prints the value of a global variable 'globalvar'. The output is: 'The globalvar is 1', 'The globalvar is 2', 'The globalvar is 3', 'The globalvar is 4', 'The globalvar is 5', 'The globalvar is 11', and 'The globalvar is 12'. The bottom window, also titled 'root@localhost: driver_test', shows the input to a program that reads the value of 'globalvar'. The input is: 'Please input the globalvar:', followed by the values '1', '2', '3', '4', '5', '11', and '12'. The desktop background is blue, and the taskbar at the bottom shows various icons and the 'yesky.com' logo.

```
root@localhost: driver_test [root@localhost driver_test]# ./globaltest2
The globalvar is 1
The globalvar is 2
The globalvar is 3
The globalvar is 4
The globalvar is 5
The globalvar is 11
The globalvar is 12

root@localhost: driver_test [root@localhost driver_test]# ./globaltest3
Please input the globalvar:
1
Please input the globalvar:
2
Please input the globalvar:
3
Please input the globalvar:
4
Please input the globalvar:
5
Please input the globalvar:
11
Please input the globalvar:
12
Please input the globalvar:
```

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！

Linux 设备驱动编程之阻塞与非阻塞

2006-10-22 07:00 作者： 宋宝华 出处： 天极软件 责任编辑： [方舟](#)

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，以使用户程序获知是否可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_head_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>

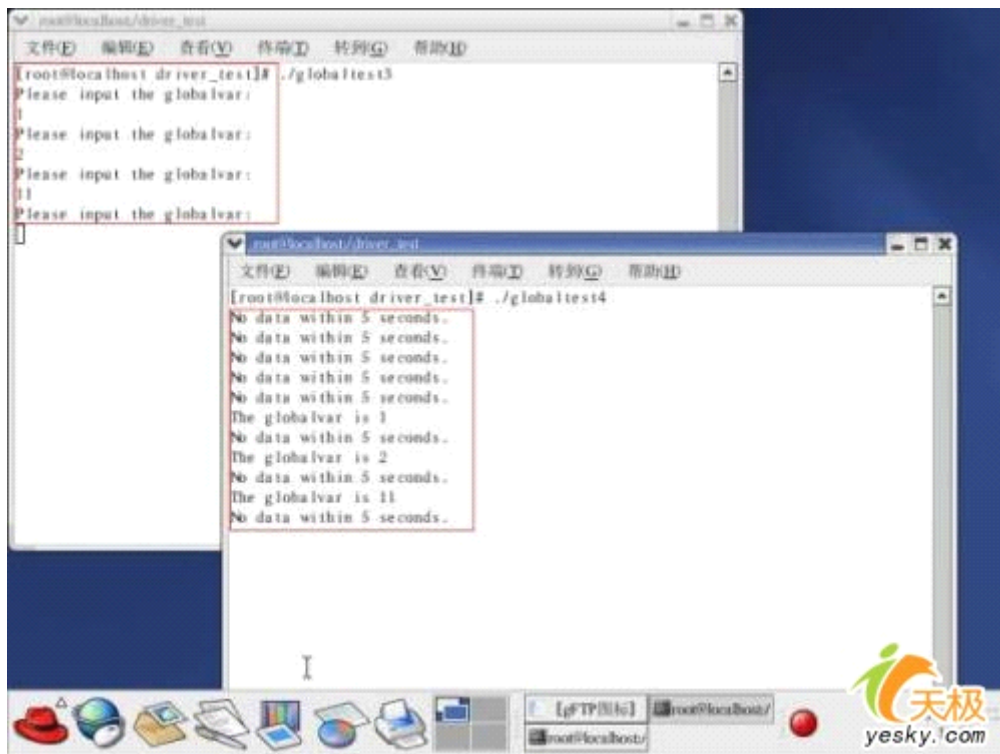
main()
{
    int fd, num;
    fd_set rfd;
    struct timeval tv;

    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(&rfd);
            FD_SET(fd, &rfd);
            //设置超时时间为 5s
            tv.tv_sec = 5;
            tv.tv_usec = 0;
            select(fd + 1, &rfd, NULL, NULL, &tv);
            //数据是否可获得?
            if (FD_ISSET(fd, &rfd))
            {
                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：



Linux 设备驱动编程之异步通知

2006-10-24 13:56 作者： 宋宝华 出处： 天极开发 责任编辑： 方舟

相关专题： [Linux 设备驱动程序开发入门](#)

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);
}

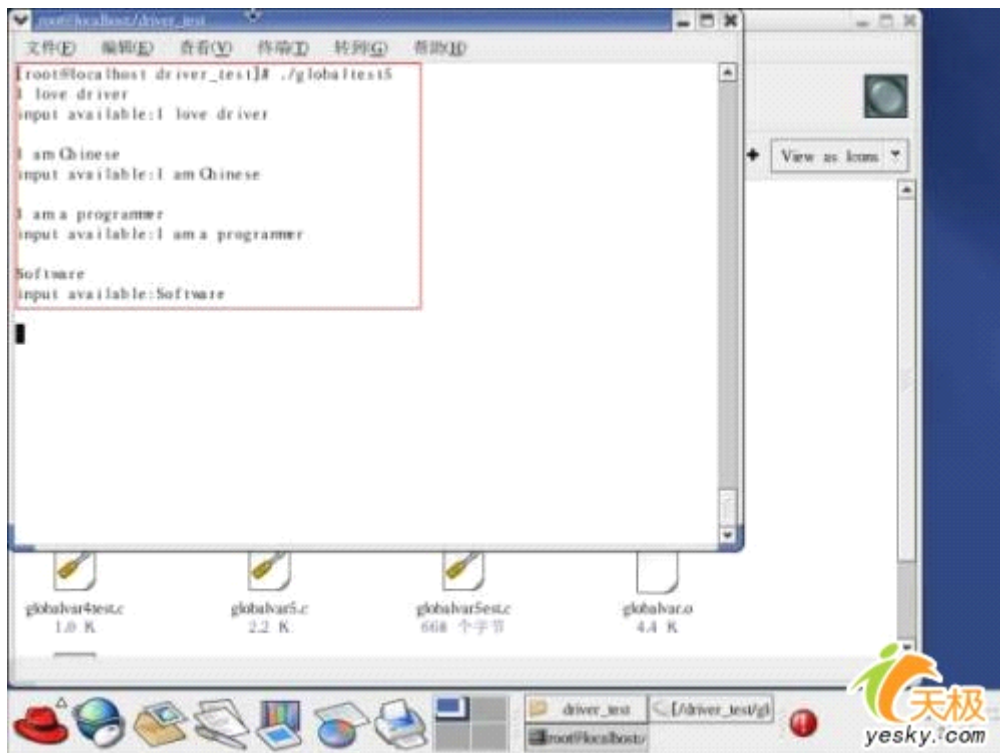
main()
{
    int oflags;

    //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。

Linux 设备驱动编程之中断处理

2006-10-25 13:36 作者： 宋宝华 出处： 天极开发 责任编辑： 方舟

相关专题： [Linux 设备驱动程序开发入门](#)

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_disable(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。

Linux 设备驱动编程之定时器

2006-10-26 13:34 作者： 出处： Linux 责任编辑： [方舟](#)

相关专题： [Linux 设备驱动程序开发入门](#)

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);
}
}
...
}
```

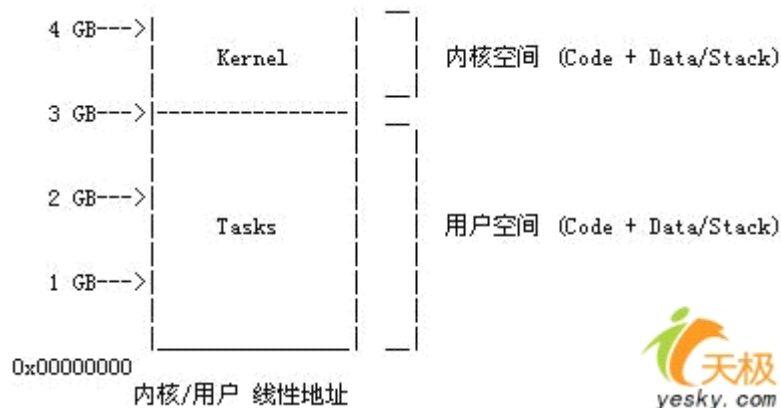
Linux 设备驱动编程之内存与 I/O 操作

2006-10-27 13:35 作者： 宋宝华 出处： 天极开发 责任编辑： [方舟](#)

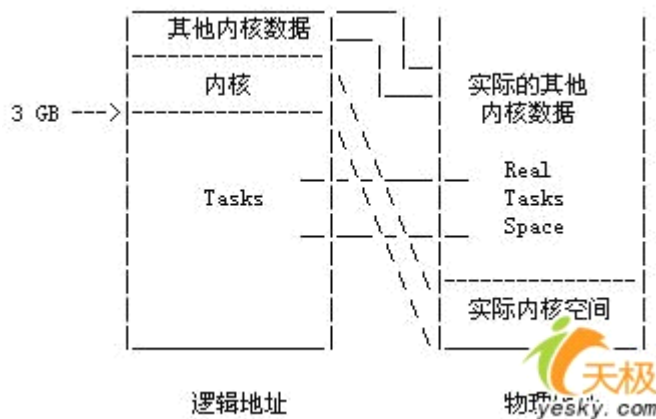
相关专题： [Linux 设备驱动程序开发入门](#)

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。

Linux 设备驱动编程之内存与 I/O 操作

2006-10-27 13:35 作者： 宋宝华 出处： 天极开发 责任编辑： [方舟](#)

（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)BCDHOURL & 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！

Linux 设备驱动编程之内存与 I/O 操作

2006-10-27 13:35 作者： 宋宝华 出处： 天极开发 责任编辑： [方舟](#)

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_t prot);

```

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。

Linux 设备驱动编程之结构化设备驱动程序

2006-10-31 15:45 作者： 宋宝华 出处： 天极开发 责任编辑： [方舟](#)

相关专题： [Linux 设备驱动程序开发入门](#)

阅读本文请首先阅读：《[Linux 设备驱动编程之内存与 I/O 操作](#)》

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 设备驱动的开发者的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），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。

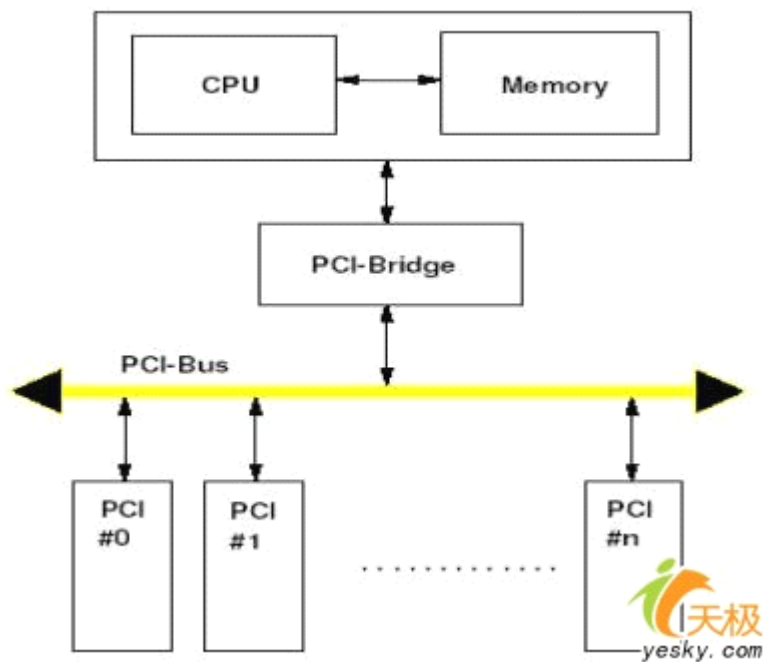
Linux 设备驱动编程之复杂设备驱动

2006-11-01 11:32 作者： 宋宝华 出处： 天极开发 责任编辑： [方舟](#)

相关专题： [Linux 设备驱动程序开发入门](#)

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;
}
printf("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;

Linux 设备驱动编程之复杂设备驱动

2006-11-01 11:32 作者: 宋宝华 出处: 天极开发 责任编辑: [方舟](#)

(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。驱动程序会在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;
}

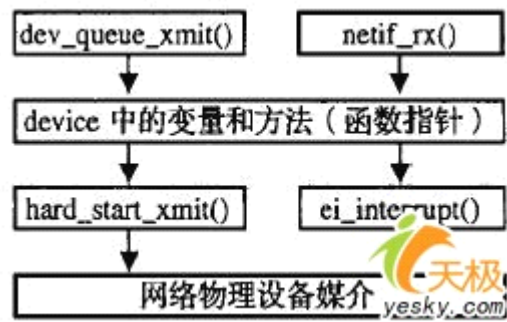
/* announce yourself */
printk(KERN_INFO "usbsample: probe successful for %s (maxp is
%i)\n", sample->name, sample->maxp);

/*
 * 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++;
#ifdef 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;

#ifdef 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。

Linux 设备驱动编程之复杂设备驱动

2006-11-01 11:32 作者： 宋宝华 出处： 天极开发 责任编辑： [方舟](#)

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);
    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;

```

}

Linux 下 Makefile 的 automake 生成全攻略

2004-10-19 09:59 作者： 余涛 出处： 天极网 责任编辑： 方舟

相关专题： [Linux 设备驱动程序开发入门](#)

作为 Linux 下的程序开发人员，大家一定都遇到过 Makefile，用 make 命令来编译自己写的程序确实是很方便。一般情况下，大家都是手工写一个简单 Makefile，如果要想写出一个符合自由软件惯例的 Makefile 就不那么容易了。

在本文中，将给大家介绍如何使用 **autoconf** 和 **automake** 两个工具来帮助我们自动地生成符合自由软件惯例的 Makefile，这样就可以象常见的 GNU 程序一样，只要使用“./configure”，“make”，“make instal”就可以把程序安装到 Linux 系统中去了。这将特别适合想做开放源代码软件的程序开发人员，又或如果你只是自己写些小的 Toy 程序，那么这篇文章对你也会有很大的帮助。

一、Makefile 介绍

Makefile 是用于自动编译和链接的，一个工程有很多文件组成，每一个文件的改变都会导致工程的重新链接，但是不是所有的文件都需要重新编译，Makefile 中纪录有文件的信息，在 make 时会决定在链接的时候需要重新编译哪些文件。

Makefile 的宗旨就是：让编译器知道要编译一个文件需要依赖其他的哪些文件。当那些依赖文件有了改变，编译器会自动的发现最终的生成文件已经过时，而重新编译相应的模块。

Makefile 的基本结构不是很复杂，但当一个程序开发人员开始写 Makefile 时，经常会怀疑自己写的是否符合惯例，而且自己写的 Makefile 经常和自己的开发环境相关联，当系统环境变量或路径发生了变化后，Makefile 可能还要跟着修改。这样就造成了手工书写 Makefile 的诸多问题，automake 恰好能很好地帮助我们解决这些问题。

使用 automake，程序开发人员只需要写一些简单的含有预定义宏的文件，由 autoconf 根据一个宏文件生成 configure，由 automake 根据另一个宏文件生成 Makefile.in，再使用 configure 依据 Makefile.in 来生成一个符合惯例的 Makefile。下面我们将详细介绍 Makefile 的 automake 生成方法。

二、使用的环境

本文所提到的程序是基于 Linux 发行版本：Fedora Core release 1，它包含了我们要用到的 autoconf，automake。

三、从 helloworld 入手

我们从大家最常使用的例子程序 helloworld 开始。

下面的过程如果简单地说来就是：

新建三个文件：

helloworld.c

configure.in

Makefile.am

然后执行：

```
aclocal; autoconf; automake --add-missing; ./configure; make; ./helloworld
```

就可以看到 Makefile 被产生出来，而且可以将 helloworld.c 编译通过。

很简单吧，几条命令就可以做出一个符合惯例的 Makefile，感觉如何呀。

现在开始介绍详细的过程：

1、建目录

在你的工作目录下建一个 helloworld 目录, 我们用它来存放 helloworld 程序及相关文件, 如在/home/my/build 下:

```
$ mkdir helloworld
$ cd helloworld
```

2、 helloworld.c

然后用你自己最喜欢的编辑器写一个 helloworld.c 文件, 如命令: vi helloworld.c。使用下面的代码作为 helloworld.c 的内容。

```
int main(int argc, char** argv)
{
    printf("Hello, Linux World!\n");
    return 0;
}
```

完成后保存退出。

现在在 helloworld 目录下就应该有一个你自己写的 helloworld.c 了。

3、生成 configure

我们使用 autoscan 命令来帮助我们根据目录下的源代码生成一个 configure.in 的模板文件。

命令:

```
$ autoscan
$ ls
configure.scan helloworld.c
```

执行后在 helloworld 目录下会生成一个文件: configure.scan, 我们可以拿它作为 configure.in 的蓝本。

现在将 configure.scan 改名为 configure.in, 并且编辑它, 按下面的内容修改, 去掉无关的语句:

```
=====configure.in      内      容      开      始
=====
# -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.

AC_INIT(helloworld.c)
AM_INIT_AUTOMAKE(helloworld, 1.0)

# Checks for programs.
AC_PROG_CC

# Checks for libraries.

# Checks for header files.

# Checks for typedefs, structures, and compiler characteristics.
```

```
# Checks for library functions.
AC_OUTPUT(Makefile)
=====configure.in      内      容      结      束
=====
```

然后执行命令 `aclocal` 和 `autoconf`，分别会产生 `aclocal.m4` 及 `configure` 两个文件：

```
$ aclocal
$ ls
aclocal.m4 configure.in helloworld.c
$ autoconf
$ ls
aclocal.m4 autom4te.cache configure configure.in helloworld.c
```

大家可以看到 `configure.in` 内容是一些宏定义，这些宏经 `autoconf` 处理后会变成检查系统特性、环境变量、软件必须的参数的 `shell` 脚本。

`autoconf` 是用来生成自动配置软件源代码脚本（`configure`）的工具。`configure` 脚本能独立于 `autoconf` 运行，且在运行的过程中，不需要用户的干预。

要生成 `configure` 文件，你必须告诉 `autoconf` 如何找到你所用的宏。方式是使用 `aclocal` 程序来生成你的 `aclocal.m4`。

`aclocal` 根据 `configure.in` 文件的内容，自动生成 `aclocal.m4` 文件。`aclocal` 是一个 `perl` 脚本程序，它的定义是：“`aclocal - create aclocal.m4 by scanning configure.ac`”。

`autoconf` 从 `configure.in` 这个列举编译软件时所需要各种参数的模板文件中创建 `configure`。

`autoconf` 需要 `GNU m4` 宏处理器来处理 `aclocal.m4`，生成 `configure` 脚本。

`m4` 是一个宏处理器。将输入拷贝到输出，同时将宏展开。宏可以是内嵌的，也可以是用户定义的。除了可以展开宏，`m4` 还有一些内建的函数，用来引用文件，执行命令，整数运算，文本操作，循环等。`m4` 既可以作为编译器的前端，也可以单独作为一个宏处理器。

Linux 下 Makefile 的 automake 生成全攻略

4、新建 Makefile.am

新建 `Makefile.am` 文件，命令：

```
$ vi Makefile.am
```

内容如下：

```
AUTOMAKE_OPTIONS=foreign
bin_PROGRAMS=helloworld
helloworld_SOURCES=helloworld.c
```

`automake` 会根据你写的 `Makefile.am` 来自动生成 `Makefile.in`。

`Makefile.am` 中定义的宏和目标，会指导 `automake` 生成指定的代码。例如，宏 `bin_PROGRAMS` 将导致编译和连接的目标被生成。

5、运行 automake

命令：

```
$ automake --add-missing
configure.in: installing `./install-sh'
configure.in: installing `./mkinstalldirs'
```



```
configure.in: installing `./missing'
Makefile.am: installing `./depcomp'
```

automake 会根据 Makefile.am 文件产生一些文件，包含最重要的 Makefile.in。

6、执行 configure 生成 Makefile

```
$ ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
checking for gcc... gcc
checking for C compiler default output... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ANSI C... none needed
checking for style of include used by make... GNU
checking dependency style of gcc... gcc3
configure: creating ./config.status
config.status: creating Makefile
config.status: executing depfiles commands
$ ls -l Makefile
-rw-rw-r-- 1 yutao yutao 15035 Oct 15 10:40 Makefile
```

你可以看到，此时 Makefile 已经产生出来了。

7、使用 Makefile 编译代码

```
$ make
if gcc -DPACKAGE_NAME="" -DPACKAGE_TARNAME=""
-DPACKAGE_VERSION="" -

DPackage_STRING="" -DPackage_BUGREPORT=""
-DPACKAGE="helloworld" -DVERSION="1.0"

-I. -I. -g -O2 -MT helloworld.o -MD -MP -MF ".deps/helloworld.Tpo" \
-c -o helloworld.o `test -f 'helloworld.c' || echo './'`helloworld.c; \
then mv -f ".deps/helloworld.Tpo" ".deps/helloworld.Po"; \
else rm -f ".deps/helloworld.Tpo"; exit 1; \
fi
gcc -g -O2 -o helloworld helloworld.o
```

运行 helloworld

```
$ ./helloworld
Hello, Linux World!
```

这样 `helloworld` 就编译出来了，你如果按上面的步骤来做的话，应该也会很容易地编译出正确的 `helloworld` 文件。你还可以试着使用一些其他的 `make` 命令，如 `make clean`，`make install`，`make dist`，看看它们会给你什么样的效果。感觉如何？自己也能写出这么专业的 `Makefile`，老板一定会对你刮目相看。

Linux 下 `Makefile` 的 `automake` 生成全攻略

四、深入浅出

针对上面提到的各个命令，我们再做些详细的介绍。

1、`autoscan`

`autoscan` 是用来扫描源代码目录生成 `configure.scan` 文件的。`autoscan` 可以用目录名做为参数，但如果你不使用参数的话，那么 `autoscan` 将认为使用的是当前目录。`autoscan` 将扫描你所指定目录中的源文件，并创建 `configure.scan` 文件。

2、`configure.scan`

`configure.scan` 包含了系统配置的基本选项，里面都是一些宏定义。我们需要将它改名为 `configure.in`

3、`aclocal`

`aclocal` 是一个 perl 脚本程序。`aclocal` 根据 `configure.in` 文件的内容，自动生成 `aclocal.m4` 文件。`aclocal` 的定义是：“`aclocal - create aclocal.m4 by scanning configure.ac`”。

4、`autoconf`

`autoconf` 是用来产生 `configure` 文件的。`configure` 是一个脚本，它能设置源程序来适应各种不同的操作系统平台，并且根据不同的系统来产生合适的 `Makefile`，从而可以使你的源代码能在不同的操作系统平台上被编译出来。

`configure.in` 文件的内容是一些宏，这些宏经过 `autoconf` 处理后会变成检查系统特性、环境变量、软件必须的参数的 shell 脚本。`configure.in` 文件中的宏的顺序并没有规定，但是你必须所有宏的最前面和最后面分别加上 `AC_INIT` 宏和 `AC_OUTPUT` 宏。

在 `configure.in` 中：

#号表示注释，这个宏后面的内容将被忽略。

`AC_INIT(FILE)`

这个宏用来检查源代码所在的路径。

```
AM_INIT_AUTOMAKE(PACKAGE, VERSION)
```

这个宏是必须的，它描述了我们将要生成的软件包的名字及其版本号：`PACKAGE` 是软件包的名字，`VERSION` 是版本号。当你使用 `make dist` 命令时，它会给你生成一个类似 `helloworld-1.0.tar.gz` 的软件发行包，其中就有对应的软件包的名字和版本号。

`AC_PROG_CC`

这个宏将检查系统所用的 C 编译器。

`AC_OUTPUT(FILE)`

这个宏是我们输出的 `Makefile` 的名字。

我们在使用 `automake` 时，实际上还需要用到其他的一些宏，但我们可以用 `aclocal` 来帮我们自动产生。执行 `aclocal` 后我们会得到 `aclocal.m4` 文件。

产生了 `configure.in` 和 `aclocal.m4` 两个宏文件后，我们就可以使用 `autoconf` 来产生 `configure` 文件了。

5、 Makefile.am

Makefile.am 是用来生成 Makefile.in 的，需要你手工书写。Makefile.am 中定义了一些内容：

AUTOMAKE_OPTIONS

这个是 automake 的选项。在执行 automake 时，它会检查目录下是否存在标准 GNU 软件包中应具备的各种文件，例如 AUTHORS、ChangeLog、NEWS 等文件。我们将其设置成 foreign 时，automake 会改用一般软件包的标准来检查。

bin_PROGRAMS

这个是指定我们所要产生的可执行文件的文件名。如果你要产生多个可执行文件，那么在各个名字间用空格隔开。

helloworld_SOURCES

这个是指定产生“helloworld”时所需要的源代码。如果它用到了多个源文件，那么请使用空格符号将它们隔开。比如需要 helloworld.h，helloworld.c 那么请写成 helloworld_SOURCES= helloworld.h helloworld.c。

如果你在 bin_PROGRAMS 定义了多个可执行文件，则对应每个可执行文件都要定义相对的 filename_SOURCES。

6、 automake

我们使用 automake --add-missing 来产生 Makefile.in。

选项--add-missing 的定义是“add missing standard files to package”，它会让 automake 加入一个标准的软件包所必须的一些文件。

我们用 automake 产生出来的 Makefile.in 文件是符合 GNU Makefile 惯例的，接下来我们只要执行 configure 这个 shell 脚本就可以产生合适的 Makefile 文件了。

7、 Makefile

在符合 GNU Makefile 惯例的 Makefile 中，包含了一些基本的预先定义的操作：

make

根据 Makefile 编译源代码，连接，生成目标文件，可执行文件。

make clean

清除上次的 make 命令所产生的 object 文件（后缀为“.o”的文件）及可执行文件。

make install

将编译成功的可执行文件安装到系统目录中，一般为/usr/local/bin 目录。

make dist

产生发布软件包文件（即 distribution package）。这个命令将会将可执行文件及相关文件打包成一个 tar.gz 压缩的文件用来作为发布软件的软件包。

它会在当前目录下生成一个名字类似“PACKAGE-VERSION.tar.gz”的文件。PACKAGE 和 VERSION，是我们在 configure.in 中定义的 AM_INIT_AUTOMAKE(PACKAGE, VERSION)。

make distcheck

生成发布软件包并对其进行测试检查，以确定发布包的正确性。这个操作将自动把压缩包文件解开，然后执行 configure 命令，并且执行 make，来确认编译不出现错误，最后提示你软件包已经准备好，可以发布了。

```
=====
helloworld-1.0.tar.gz is ready for distribution
=====
```

```
make distclean
```

类似 `make clean`，但同时也将 `configure` 生成的文件全部删除掉，包括 `Makefile`。

五、结束语

通过上面的介绍，你应该可以很容易地生成一个你自己的符合 GNU 惯例的 `Makefile` 文件及对应的项目文件。

如果你想写出更复杂的且符合惯例的 `Makefile`，你可以参考一些开放代码的项目中的 `configure.in` 和 `Makefile.am` 文件，比如：嵌入式数据库 `sqlite`，单元测试 `cppunit`。

入门文章：教你学会编写 Linux 设备驱动

内核版本: 2.4.22

阅读此文的目的: 学会编写 Linux 设备驱动。

阅读此文的方法: 阅读以下 2 个文件: `hello.c`，`asdf.c`。

此文假设读者:

已经能用 C 语言编写 Linux 应用程序，

理解"字符设备文件，块设备文件，主设备号，次设备号"，

会写简单的 Shell 脚本和 `Makefile`。

1. "hello.c"

```
-----
/*
 * 这是我们的第一个源文件，
 * 它是一个可以加载的内核模块，
 * 加载时显示"Hello, World!",
 * 卸载时显示"Bye!".
 * 需要说明一点，写内核或内核模块不能用写应用程序时的系统调用或函数库，
 * 因为我们写的就是为应用程序提供系统调用的代码。
 * 内核有专用的函数库，如，，等，
 * 现在还没必要了解得很详细，
 * 这里用到的 printk 的功能类似于 printf。
 * "/usr/src/linux"是你实际的内核源码目录的一个符号链接，
 * 如果没有现在就创建一个，因为下面和以后都会用到。
 * 编译它用"gcc -c -I/usr/src/linux/include hello.c",
 * 如果正常会生成文件 hello.o，
 * 加载它用"insmod hello.o",
 * 只有在文本终端下才能看到输出。
 * 卸载它用"rmmod hello"
 */
/*
 * 小技巧: 在用户目录的.bashrc 里加上一行:
 * alias mkmod='gcc -c -I/usr/src/linux/include'
 * 然后重新登陆 Shell，
 * 以后就可以用"mkmod hello.c"的方式来编译内核模块了。
 */
/* 开始例行公事 */
```

```

#ifndef __KERNEL__
#define __KERNEL__
#endif
#ifndef MODULE
#define MODULE
#endif
#include
#include
MODULE_LICENSE("GPL");
#ifdef CONFIG_SMP
#define __SMP__
#endif
/* 结束例行公事 */
#include/* printk()在这个文件里 */
static int init_module()
{
    printk("Hello, World!\n");
    return 0; /* 如果初始工作失败，就返回非 0 */
}
static void cleanup_module()
{
    printk("Bye!\n");
}

```

2. "asdf.c"

```

/*
 * 这个文件是一个内核模块。
 * 内核模块的编译，加载和卸载在前面已经介绍了。
 * 这个模块的功能是，创建一个字符设备。
 * 这个设备是一块 4096 字节的共享内存。
 * 内核分配的主设备号会在加载模块时显示。
 */
/* 开始例行公事 */
#ifndef __KERNEL__
#define __KERNEL__
#endif
#ifndef MODULE
#define MODULE
#endif
#include
#include
#ifdef CONFIG_SMP
#define __SMP__

```

```

#endif
MODULE_LICENSE("GPL");
/* 结束例行公事 */
#include/* copy_to_user(), copy_from_user */
#include/* struct file_operations, register_chrdev(), ... */
#include/* printk()在这个文件里 */
#include/* 和任务调度有关 */
#include/* u8, u16, u32 ... */
/*
* 关于内核功能库，可以去网上搜索详细资料，
*/
/* 文件被操作时的回调功能 */
static int asdf_open (struct inode *inode, struct file *filp);
static int asdf_release (struct inode *inode, struct file *filp);
static ssize_t asdf_read (struct file *filp, char *buf, size_t count, loff_t *f_pos);
static ssize_t asdf_write (struct file *filp, const char *buf, size_t count, loff_t *f_pos);
static loff_t asdf_lseek (struct file * file, loff_t offset, int orig);
/* 申请主设备号时用的结构，在 linux/fs.h 里定义 */
struct file_operations asdf_fops = {
open: asdf_open,
release: asdf_release,
read: asdf_read,
write: asdf_write,
llseek: asdf_lseek,
};
static int asdf_major; /* 用来保存申请到的主设备号 */
static u8 asdf_body[4096]="asdf_body\n"; /* 设备 */
static int init_module()
{
    printk ("Hi, This' A Simple Device File!\n");
    asdf_major = register_chrdev (0, "A Simple Device File", &asdf_fops); /* 申请字符
设备的主设备号 */
    if (asdf_major < 0) return asdf_major; /* 申请失败就直接返回错误编号 */
    printk ("The major is:%d\n", asdf_major); /* 显示申请到的主设备号 */
    return 0; /* 模块正常初始化 */
}
static void
cleanup_module()
{
    unregister_chrdev(asdf_major, "A Simple Device File"); /* 注销以后，设备就不存在了 */
    printk("A Simple Device has been removed, Bye!\n");
}
/*
* 编译这个模块然后加载它，

```

- * 如果正常，会显示你的设备的主设备号。
- * 现在你的设备就建立好了，我们可以测试一下。
- * 假设你的模块申请到的主设备号是 254，
- * 运行"mknod abc c 254 0"，就建立了我们的设备文件 abc。
- * 可以把它当成一个 4096 字节的内存块来测试一下，
- * 比如"cat abc"， "cp abc image"， "cp image abc"，
- * 或写几个应用程序用它来进行通讯。
- * 介绍一下两个需要注意的事，
- * 一是 printk() 的显示只有在非图形模式的终端下才能看到，
- * 二是加载过的模块最好在不用以后卸载掉。
- * 如果对 Linux 环境的系统调用很陌生，建议先看 APUE 这本书。

```

*/
static int
asdf_open /* open 回调 */
(
    struct inode *inode,
    struct file *filp
){
    printk("^_^: open %s\n ", \
current->comm);
    /*
    * 应用程序的运行环境由内核提供，内核的运行环境由硬件提供。
    * 这里的 current 是一个指向当前进程的指针，
    * 现在没必要了解 current 的细节。
    * 在这里，当前进程正打开这个设备，
    * 返回 0 表示打开成功，内核会给它一个文件描述符。
    * 这里的 comm 是当前进程在 Shell 下的 command 字符串。
    */
    return 0;
}
static int
asdf_release /* close 回调 */
(
    struct inode *inode,
    struct file *filp
){
    printk("^_^: close\n ");
    return 0;
}
static ssize_t
asdf_read /* read 回调 */
(
    struct file *filp,
    char *buf,

```

```

    size_t count,
    loff_t *f_pos
){
    loff_t pos;
    pos = *f_pos; /* 文件的读写位置 */
    if ((pos==4096) || (count>4096)) return 0; /* 判断是否已经到设备尾，或写的长度超过设
设备大小 */
    pos += count;
    if (pos > 4096) {
        count -= (pos - 4096);
        pos = 4096;
    }
    if (copy_to_user(buf, asdf_body+*f_pos, count)) return -EFAULT; /* 把数据写到应用
程序空间 */
    *f_pos = pos; /* 改变文件的读写位置 */
    return count; /* 返回读到的字节数 */
}

static ssize_t
asdf_write /* write 回调，和 read 一一对应 */
(
    struct file *filp,
    const char *buf,
    size_t count,
    loff_t *f_pos
){
    loff_t pos;
    pos = *f_pos;
    if ((pos==4096) || (count>4096)) return 0;
    pos += count;
    if (pos > 4096) {
        count -= (pos - 4096);
        pos = 4096;
    }
    if (copy_from_user(asdf_body+*f_pos, buf, count)) return -EFAULT;
    *f_pos = pos;
    return count;
}

static loff_t
asdf_lseek /* lseek 回调 */
(
    struct file * file,
    loff_t offset,
    int orig
){

```



```

loff_t pos;
pos = file->f_pos;
switch (orig) {
case 0:
pos = offset;
break;
case 1:
pos += offset;
break;
case 2:
pos = 4096+offset;
break;
default:
return -EINVAL;
}
if ((pos>4096) || (pos<0)) {
printk("^_: lseek error %d\n", pos);
return -EINVAL;
}
return file->f_pos = pos;
}

```

Linux 下设备完全驱动之一

时间：2005-09-25 作者：鄢晓烨 来源：赛迪

本讲主要概述 Linux 设备驱动框架、驱动程序的配置文件及常用的加载驱动程序的方法；并且介绍 Red Hat Linux 安装程序是如何加载驱动的，通过了解这个过程，我们可以自己将驱动程序放到引导盘中；安装完系统后，使用 kudzu 自动配置硬件程序。

Linux 设备驱动概述

1. 内核和驱动模块

操作系统是通过各种驱动程序来驾驭硬件设备，它为用户屏蔽了各种各样的设备，驱动硬件是操作系统最基本的功能，并且提供统一的操作方式。正如我们查看屏幕上的文档时，不用去管到底使用 nVIDIA 芯片，还是 ATI 芯片的显卡，只需知道输入命令后，需要的文字就显示在屏幕上。硬件驱动程序是操作系统最基本的组成部分，在 Linux 内核源程序中也占有较高的比例。

Linux 内核中采用可加载的模块化设计（LKMs，Loadable Kernel Modules），一般情况下编译的 Linux 内核是支持可插入式模块的，也就是将最基本的核心代码编译在内核中，其它的代码可以选择是在内核中，或者编译为内核的模块文件。

如果需要某种功能，比如需要访问一个 NTFS 分区，就加载相应的 NTFS 模块。这种设计可以使内核文件不至于太大，但是又可以支持很多的功能，必要时动态地加载。这是一种跟微内核设计不太一样，但却是切实可行的内核设计方案。

我们常见的驱动程序就是作为内核模块动态加载的，比如声卡驱动和网卡驱动等，而 Linux 最基础的驱动，如 CPU、PCI 总线、TCP/IP 协议、APM（高级电源管理）、VFS 等驱动程序则编译在内核文件中。有时也把内核模块就叫做驱动程序，只不过驱动的内容不一定

是硬件罢了，比如 ext3 文件系统的驱动。

理解这一点很重要。因此，加载驱动时就是加载内核模块。下面来看一下有关模块的命令，在加载驱动程序要用到它们：lsmod、modprob、insmod、rmmod、modinfo。

lsmod 列出当前系统中加载的模块，例如：

#lsmod （与 cat /proc/modules 得出的内容是一致的）

Module	Size	Used by	Not tainted
radeon	115364	1	
agpgart	56664	3	
nls_iso8859-1	3516	1	(autoclean)
loop	12120	3	(autoclean)
smbfs	44528	2	(autoclean)
parport_pc	19076	1	(autoclean)
lp	9028	0	(autoclean)
parport	37088	1	(autoclean) [parport_pc lp]
autofs	13364	0	(autoclean) (unused)
ds	8704	2	
yenta_socket	13760	2	
pcmcia_core	57184	0	[ds yenta_socket]
tg3	55112	1	
sg	36940	0	(autoclean)
sr_mod	18104	0	(autoclean)
microcode	4724	0	(autoclean)
ide-scsi	12208	0	
scsi_mod	108968	3	[sg sr_mod ide-scsi]
ide-cd	35680	0	
cdrom	33696	0	[sr_mod ide-cd]
nls_cp936	124988	1	(autoclean)
nls_cp437	5148	1	(autoclean)
vfat	13004	1	(autoclean)
fat	38872	0	(autoclean) [vfat]
keybdev	2976	0	(unused)
mousedev	5524	1	
hid	22212	0	(unused)
input	5888	0	[keybdev mousedev hid]
ehci-hcd	20104	0	(unused)
usb-uhci	26412	0	(unused)
usbcore	79392	1	[hid ehci-hcd usb-uhci]
ext3	91592	2	
jbd	52336	2	[ext3]

上面显示了当前系统中加载的模块，左边数第一列是模块名，第二列是该模块大小，第三列则是该模块使用的数量。

如果后面为 unused，则表示该模块当前没在使用。如果后面有 autoclean，则该模块可以被 rmmod -a 命令自动清洗。rmmod -a 命令会将目前有 autoclean 的模块卸载，如果这时候某个模块未被使用，则将该模块标记为 autoclean。如果在行尾的 [] 括号内有模块名称，则括

号内的模块就依赖于该模块。例如：

```
cdrom          34144    0    [sr_mod ide-cd]
```

其中 `ide-cd` 及 `sr_mod` 模块就依赖于 `cdrom` 模块。

系统的模块文件保存在 `/lib/modules/2.4.XXX/kerne` 目录中，根据分类分别在 `fs`、`net` 等子目录中，他们的互相依存关系则保存在 `/lib/modules/2.4.XXX/modules.dep` 文件中。

需要注意，该文件不仅写入了模块的依存关系，同时内核查找模块也是在这个文件中，使用 `modprobe` 命令，可以智能插入模块，它可以根据模块间依存关系，以及 `/etc/modules.conf` 文件中的内容智能插入模块。比如希望加载 `ide` 的光驱驱动，则可运行下面命令：

```
# modprobe ide-cd
```

此时会发现，`cdrom` 模块也会自动插入。

`insmod` 也是插入模块的命令，但是它不会自动解决依存关系，所以一般加载内核模块时使用的命令为 `modprobe`。

`rmmod` 可以删除模块，但是它只可以删除没有使用的模块。

`Modinfo` 用来查看模块信息，如 `modinfo -d cdrom`，在 Red Hat Linux 系统中，模块的相关命令在 `modutils` 的 RPM 包中。

2. 设备文件

当我们加载了设备驱动模块后，应该怎样访问这些设备呢？Linux 是一种类 Unix 系统，Unix 的一个基本特点是“一切皆为文件”，它抽象了设备的处理，将所有的硬件设备都像普通文件一样看待，也就是说硬件可以跟普通文件一样来打开、关闭和读写。

系统中的设备都用一个设备特殊文件代表，叫做设备文件，设备文件又分为 **Block**（块）型设备文件、**Character**（字符）型设备文件和 **Socket**（网络插件）型设备文件。**Block** 设备文件常常指定哪些需要以块（如 512 字节）的方式写入的设备，比如 **IDE** 硬盘、**SCSI** 硬盘、光驱等。

而 **Character** 型设备文件常指定直接读写，没有缓冲区的设备，比如并口、虚拟控制台等。**Socket**（网络插件）型设备文件指定的是网络设备访问的 **BSD socket** 接口。

```
# ls -l /dev/hda /dev/video0 /dev/log
```

```
brw-rw----    1 root    disk      3,    0 Sep 15  2003 /dev/hda
srw-rw-rw-    1 root    root              0 Jun  3 16:55 /dev/log
crw-----    1 root    root      81,   0 Sep 15  2003 /dev/video0
```

上面显示的是三种设备文件，注意它们最前面的字符，**Block** 型设备为 **b**，**Character** 型设备为 **c**，**Socket** 设备为 **s**。

由此可以看出，设备文件都放在 `/dev` 目录下，比如硬盘就是用 `/dev/hd*` 来表示，`/dev/hda` 表示第一个 IDE 接口的主设备，`/dev/hda1` 表示第一个硬盘上的第一个分区；而 `/dev/hdc` 表示第二个 IDE 接口的主设备。可以使用下面命令：

```
# dd if=/dev/hda of=/root/a.img bs=446 count=1
```

把第一个硬盘上前 446 个字节的 MBR 信息导入到 `a.img` 文件中。

对于 **Block** 和 **Character** 型设备，使用主（Major）和辅（minor）设备编号来描述设备。主设备编号来表示某种驱动程序，同一个设备驱动程序模块所控制的所有设备都有一个共同的主设备编号，而辅设备编号用于区分该控制器下不同的设备，比如，`/dev/hda1`（block 3/1）、`/dev/hda2`（block 3/2）和 `/dev/hda3`（block 3/3）都代表着同一块硬盘的三个分区，他们的主设备号都是 3，辅设备号分别为 1、2、3。这些设备特殊文件用 `mknod` 命令来创建：

```
# mknod harddisk b 3 0
```

我们就在当前位置创建出一个与 `/dev/hda` 一样的、可以访问第一个 IDE 设备主硬盘的

文件，文件名叫做 `harddisk`。

使用下面命令可以查看设备编号：

```
#file /dev/hda
```

```
/dev/hda: block special (3/0)
```

其中 `Block` 代表 `/dev/hda` 是系统的 `Block` 型（块型）设备文件，它的主设备编号为 3，辅设备编号为 0。

```
#ls -l /dev/hda /dev/hdb
```

```
brw-rw---- 1 root disk 3, 0 Sep 15 2003 /dev/hda
```

```
brw-rw---- 1 root disk 3, 64 Sep 15 2003 /dev/hdb
```

使用 `ls -l` 也可以看到设备编号，`/dev/hdb` 代表第一个 IDE 接口的从设备（Slave）也是 `Block` 设备，编号为(3/64),还有另外一种设备文件是 `/dev/tty*`。使用如下命令：

```
#echo "hello tty1" > /dev/tty1
```

将字符串“`hello tty1`”输出到 `/dev/tty1` 代表的第一个虚拟控制台上，此时按“`Alt + F1`”可以看到该字符出现在屏幕上，这个特殊的文件就代表着我们的第一虚拟控制台。

```
# file /dev/tty1
```

```
/dev/tty1: character special (4/1)
```

由上可以看到，它的类型为 `Character` 型（字符型）设备文件，主设备号为 4，辅设备号为 1。同样，`/dev/tty2` 代表着第二个虚拟控制台，是 `Character` 设备，编号为 (4/2)。

当将 `/dev/cdrom` 加载到 `/mnt/cdrom` 中时，只要访问 `/mnt/cdrom` 系统就会自动引入到 `/dev/cdrom` 对应的驱动程序中，访问实际的数据。

有关设备文件的编号可以看内核文档 `/usr/src/linux-2.*/Documentation/devices.txt` 文件（在 `Kernel` 的源文件解包后的 `Documentation` 目录中），其中详细叙述了各种设备文件编号的意义。

3.使用 `/proc` 目录中的文件监视驱动程序的状态

通过设备文件怎样访问到相应的驱动程序呢？它们中间有一个桥梁，那就是 `proc` 文件系统，它一般会被加载到 `/proc` 目录。访问设备文件时，操作系统通常会通过查找 `/proc` 目录下的值，确定由哪些驱动模块来完成任务。如果 `proc` 文件系统没有加载，访问设备文件时就会出现错误。

`Linux` 系统中 `proc` 文件系统是内核虚拟的文件系统，其中所有的文件都是内核中虚拟出来的，各种文件实际上是当前内核在内存中的参数。它就像是专门为访问内核而打开的一扇门，比如访问 `/proc/cpuinfo` 文件，实际上就是访问目前的 `CPU` 的参数，每一次系统启动时系统都会通过 `/etc/fstab` 中设置的信息自动将 `proc` 文件系统加载到 `/proc` 目录下：

```
# grep proc /etc/fstab
```

```
none/proc proc defaults 0 0
```

此外，也可以通过 `mount` 命令手动加载：

```
# mount -t proc none /proc
```

通过 `/proc` 目录下的文件可以访问或更改内核参数，可以通过 `/proc` 目录查询驱动程序的信息。

下面先让我们看一下 `/proc` 目录中的信息：

```
# ls /proc
```

```
1 4725 5032 5100 5248 5292 crypto kcore partitions
```

```
14 4794 5044 5110 5250 5293 devices kmsg pci
```

```
2 4810 5075 5122 5252 5295 dma ksyms self
```

```

3  4820  5079  5132  5254  5345 driver loadavg slabinfo
4      4831  5080  5151  5256  6      execdomains  locks  stat
4316  4910  5081  5160  5258  7      fb          lvm    swaps
4317  4912  5082  5170  5262  70     filesystems mdstat sys
4318  4924  5083  5180  5271  8      fs          meminfo
sysrq-trigger
4319  4950  5084  5189  5287  9      ide          misc  sysvipc
4620  4963  5085  5232  5288  apm      interrupts modules tty
4676  5      5086  5242  5289  bus      iomem      mounts  uptime
4680  5005  5087  5244  5290  cmdline ioports    mtrr    version
4706  5018  5088  5246  5291  cpuinfo  irq        net

```

需要知道的是，这些文件都是实时产生的虚拟文件，访问它们就是访问内存中真实的数据。这些数据是实时变化产生的，可以通过以下命令来查看文件的具体值：

```

# cat /proc/interrupts
CPU0
0:      50662      XT-PIC  timer
1:         3      XT-PIC  keyboard
2:         0      XT-PIC  cascade
5:       618      XT-PIC  ehci-hcd, eth1
8:         1      XT-PIC  rtc
9:         0      XT-PIC  usb-uhci, usb-uhci
11:       50      XT-PIC  usb-uhci, eth0
12:       16      XT-PIC  PS/2 Mouse
14:     8009      XT-PIC  ide0
15:         0      XT-PIC  ide1
NMI:         0
ERR:         0

```

其它文件的含意见表 1 所示。

/proc/sys 目录下的文件一般可以直接更改，相当于直接更改内核的运行参数，例如：

```
# echo 1 > /proc/sys/net/ipv4/ip_forward
```

上面代码可以将内核中的数据包转发功能打开。

另外，Linux 系统中提供一些命令来查询系统的状态，如 free 可以查看目前的内存使用情况，ide_info 可以查看 ide 设备的信息，例如：

```
#ide_info /dev/had
```

类似的命令还有 scsi_info，可以查看 SCSI 设备的信息。这些命令一般也是查询/proc 目录下的文件，并返回结果。

系统初始化过程驱动程序的安装

在 Linux 安装过程中，系统上的硬件会被检测，基于检测到的结果安装程序会决定哪些模块需要在引导时被载入。Red Hat 的安装程序为 anaconda，它提供了自动检测硬件，并且

安装的机制。

但是，如果计算机内的某些硬件没有默认的驱动程序，比如一块 SCSI 卡，我们可以在启动后的 boot 提示符下，输入“linux dd”，在加载完内核后，系统会自动提示插入驱动盘，这时就有机会把该硬件的 Linux 驱动程序装入。

如果在安装系统时，某种硬件总是因为中断冲突（ISA 总线的设备较常见，比如一块 ISA 网卡）没法正常驱动，或者是缺少驱动程序，那么可以在 boot 提示符下输入“linux noprobe”。在这种模式下，安装程序不会自动配置找到的硬件，可以自己来选择现有驱动，配置驱动程序的参数，或者选择用光盘或软盘加载驱动程序。

定制引导盘

系统启动时是如何加载驱动的？下面让我们来看一下 Red Hat 的安装光盘是怎样引导的。当 Linux 安装光盘启动时，加载位于光盘上 isolinux 中的内核文件 vmlinuz，内核运行完毕后，又将 initrd.img 的虚拟文件系统加载到内存中。这个文件为 ext2 文件系统的镜像，经过 gzip 压缩，可以通过以下步骤查看该镜像中的内容：

```
# mount /mnt/cdrom
# mkdir /mnt/imgdir
# gunzip < /mnt/cdrom/isolinux/initrd.img > /ext2img
# mount -t ext2 -o loop /ext2img /mnt/imgdir
# cd /mnt/imgdir
# ls -F
bin@
dev/
etc/
linuxrc@
lost+found/
modules/
proc/
sbin/
tmp/
var/
# cd modules
# ls
module-info
modules.cgz
modules.dep
modules.pcimap
pcitable
```

其中 modules.dep 为模块的注册文件，同时有各种模块的依存关系。modules.cgz 为 cpio 的打包文件，实际的各种驱动模块就在该文件中。我们可以通过以下命令解包：

```
# cpio -idmv < modules.cgz
```

由此可以看到，解包出来的目录 2.4.21-4XXX。进入该目录下的 i386 目录，就可以看到当前启动盘中支持的所以驱动程序：

```
# ls
3c59x.o
3w-xxxx.o
```

```
8139cp.o
8139too.o
8390.o
aacraid.o
acenic.o
aic79xx.o
.....
```

若希望在系统中加入需要的驱动程序，可以相应地修改这些文件，比如在 `modules.dep` 中加入该模块的名字和依存关系，将编译好的驱动模块文件加入 `modules.cgi` 中，这样就可以制定自己的安装光盘。

硬盘上的系统启动过程与上面类似，但是 `initrd` 的镜像文件要更简单些，一般在 `initrd-2.4.XXX.img` 的虚拟文件系统中，只会在 `/lib` 目录下包含 `ext3.o` `jbd.o` `lvm-mod.o` 等少数文件，用来驱动硬盘上的 `ext3` 的文件系统。加载文件系统后，就可以使用 `/lib/modules/2.4.XXX/` 下的 `modules.dep` 文件及 `Kernel` 目录中的各种驱动文件。

自动配置安装

如果安装完 Linux 系统后，又添加了新的硬件，那么系统必须载入正确的驱动程序才可以使用它。在 Red Hat Linux 中，可以使用 `kudzu` 来配置硬件。这是 PnP 设备的检测程序，当系统使用新硬件引导后，运行 `kudzu`（默认会自动运行），如果新硬件被支持，那么它就会被自动检测到。该程序还会为它配置驱动模块，把结果写入到文件 `/etc/sysconfig/hwconf` 中，`kudzu` 可以通过对比这个文件发现新安装的硬件，并进行配置；也可以通过编辑模块配置文件 `/etc/modules.conf` 来手工指定加载模块。

`Kudzu` 服务默认每次启动时都要运行，如果需要缩短启动时间，使用下面命令可以停止系统启动时的 `kudzu` 服务：

```
# chkconfig kudzu off
```

如果要安装新的硬件，可以手动运行 `kudzu` 程序。

```
# kudzu
```

那么 `kudzu` 程序如何认识硬件的呢？可以查看 `/usr/share/hwdata/` 目录下的文件，根据这些文件中的 PnP 信息，`kudzu` 可以识别各种硬件设备。

以上介绍了 Linux 下驱动程序的大体结构、主要的加载方式和相关配置文件，在安装 Linux 时加载驱动程序，并且根据需要定制自己的引导盘，在安装完成后安装新的、即插即用硬件。下一讲开始，我们将学习具体硬件驱动的安装方法。

Linux 下设备完全驱动之二(1)

时间：2005-09-25 作者：郗晓烨 来源：赛迪

前一节 Linux 培训园地：Linux 下设备完全驱动之一，大家看过之后相信一定印象深刻。这一节首先讲述 IDE 硬盘及光驱的设置、IDE 刻录机的使用，以及如何安装 SCSI 硬盘驱动。然后介绍以太网卡驱动模块的加载及网络接口的启动过程，如何调整网卡的参数，Modem、ADSL 和宽带的驱动安装，以及 PPP 连接的设置等。

IDE 硬盘及光驱

1.IDE 设备的驱动过程

操作系统首先是安装在块设备上，没有对块设备的支持系统就无法启动，所以首先介绍常见块设备的安装。硬盘就是最常见的块设备，普通 PC 上的硬盘通常是 IDE 接口的，而服

务器上的硬盘通常是 SCSI 接口的。

一般内核中内置对通用 IDE 控制芯片的支持。下面看一下 IDE 硬盘在内核中的驱动过程，`dmesg` 命令可以看到内核在启动和加载内核模块时的信息：

```
# dmesg | less
```

在 Linux 内核启动过程中，可以发现内核首先驱动初始化 CPU、内存、系统时钟部分，接着加载 PCI 总线的驱动，然后就加载了通用的 IDE 驱动程序：

```
Uniform Multi-Platform E-IDE driver Revision: 7.00beta4-2.4
```

接着初始化 IDE 的控制器，IDE 控制器集成在 Intel 的 ICH4 南桥芯片组中，IDE 控制芯片驱动加载后，进行初始化传输模式：

```
ICH4: chipset revision 1
```

```
ICH4: not 100% native mode: will probe irqs later
```

```
ide0: BM-DMA at 0xbfa0-0xbfa7, BIOS settings: hda:DMA, hdb:pio
```

```
ide1: BM-DMA at 0xbfa8-0xbfaf, BIOS settings: hdc:DMA, hdd:pio
```

该驱动程序会向核心中注册主设备号为 3 的 block 型设备。可以看到，在 IDE 控制器初始化时，占用的 I/O 资源及分配给它的中断号：

```
ide0 at 0x1f0-0x1f7,0x3f6 on irq 14
```

```
ide1 at 0x170-0x177,0x376 on irq 15
```

接着使用 IDE 控制器查找连接在 IDE 接口上的设备，如果检查到硬盘则加载 IDE 硬盘的驱动程序，设置了该硬盘的基本参数，设置传输方式为 UDMA (100)，也就是 ATA100 (100Mb/s 的传输速度)，并且根据这个驱动程序检测硬盘上的分区：

```
hda: attached ide-disk driver.
```

```
hda: host protected area => 1
```

```
hda: 78140160 sectors (40008 MB) w/7898KiB Cache, CHS=4864/255/63,  
UDMA(100)
```

```
ide-floppy driver 0.99.newide
```

```
Partition check:
```

```
hda: hda1 hda2 hda3 hda4 < hda5 hda6 hda7 hda8 hda9 >
```

`/dev/hda` 代表第一个 IDE 接口的主设备，它的设备号为 block (3/0)，而 `/dev/hda1` 是这块硬盘的第一个分区，设备编号是 block (3/1)；`/dev/hdb` 代表第一个 IDE 接口的从设备，设备编号为 block (3/64)。

由此我们可以看到，内核默认可以支持 1~63 个分区，其中第一个逻辑分区的编号肯定为 `/dev/hda5`。但是，在 `/dev` 目录下查找有 `hda1~hda32`，共 32 个分区文件，如果需要更多的分区，就需要使用 `mknod` 命令来创建更多的设备文件。

`/dev/hdc` 是第二个 IDE 接口的主设备；`/dev/hdd` 是第二个 IDE 接口的从设备。

2. 安装、升级常见的 IDE 驱动程序

通用的 IDE 控制器可以通过内核这样加载起来，如果遇到一些较新的芯片组，当前的内核无法完全发挥出新硬件的性能，这时就要向内核中打补丁，例如，2.4.20-8 的内核就无法支持 VIA VT8237 芯片组中的 IDE ATA133 方式，需要向内核中打补丁。

先到 VIA 的网站下载相关补丁，网址为 <http://www.viaarena.com/?PageID=297#ATA>，注意要选择适合当前自己内核的驱动，接下来是升级内核，给内核打补丁。

```
# rpm -ivh kernel-source-<Kernel Version>.i386.rpm
```

安装需要版本的源代码包。把刚才链接中的补丁下载，将这个 patch 文件解开：

```
# tar xzvf VIA IDE ATA133 Patch 8237 ver0.8.gz
```

进入解包出来的目录，将需要的 patch 文件 cp 到 `/usr/src` 目录：


```
# cp <Linux OS>-patch-<Kernel Version> /usr/src
```

Kernel Version 代表内核的版本号，Linux OS 代表不同的 Linux 系统。

```
# cd /usr/src
```

```
# patch -p0 < <Linux OS>-patch-<Kernel Version>
```

将 patch 打入内核中，重新编译内核：

```
# cd /usr/src/linux-<Kernel Version>
```

编辑 Makefile 文件，把 "EXTRAVERSION=" 改成 "EXTRAVERSION=-test"，这是给新的内核命名。

```
# cp /boot/config-XXX .config
```

```
# make menuconfig ( config 或 xconfig 也可以 )
```

确定 "ATA/IDE/MFM/RLL support/IDE,ATA and ATAPI Block devices" 中的 "VIA82CXXX chipset support"被选中。

开始编译内核：

```
# make dep
```

```
# make clean
```

```
# make bzImage
```

```
# make modules
```

```
# make modules_install
```

```
# cp arch/i386/boot/bzImage (或 vmlinuz-test) /boot/vmlinuz-test
```

```
# cp /boot/initrd-< Kernel Version >.img /boot/initrd-test.img
```

编辑 /boot/grub/grub.conf 文件，在最后添加下面三行：

```
title linux-test
```

```
kernel /boot/vmlinuz-test ro root=/dev/hda1
```

```
initrd /boot/initrd-test.img
```

重新启动系统，使用刚刚编译的内核就会发现启动信息中多出一行“linux - test”。留意启动时的信息，就会发现有下面一行信息：

```
"VP_IDE: VIA vt8237 (rev 00) IDE UDMA133 Controller on pci00:0f.1"
```

如果正常启动，各种服务也都没有问题，那么以后就可以用这个新的内核了。可以用 # hdparm -i 命令来调整硬盘的传输方式，检查硬盘目前的传输模式。

```
/dev/hda:
```

```
Model=IC25N040ATCS05-0, FwRev=CS40A63A, SerialNo=CLP429F4HALVPA
```

```
Config={ HardSect NotMFM HdSw>15uSec Fixed DTR>10Mbs }
```

```
RawCHS=16383/16/63, TrkSize=0, SectSize=0, ECCbytes=4
```

```
BuffType=DualPortCache, BuffSize=7898kB, MaxMultSect=16, MultSect=16
```

```
CurCHS=16383/16/63, CurSects=16514064, LBA=yes, LBASects=78140160
```

```
IORDY=on/off, tPIO={min:240,w/IO:120}, tDMA={min:120,rec:120}
```

```
PIO modes: pio0 pio1 pio2 pio3 pio4
```

```
DMA modes: mdma0 mdma1 mdma2
```

```
UDMA modes: udma0 udma1 udma2 udma3 udma4 *udma5 //这里显示所支持的
```

硬盘传输模式

```
AdvancedPM=yes: mode=0x80 (128) WriteCache=enabled
```

```
Drive conforms to: ATA/ATAPI-5 T13 1321D revision 3:
```

* signifies the current active mo

查看一下当前硬盘的工作模式，如果不是 ATA133，则可以灵活使用控制硬盘传输模式的命令：

```
# hdparm -d1 /dev/hda //enable DMA 模式
# hdparm -d0 /dev/hda //disable DMA 模式
# hdparm -X70 /dev/hda //将传输模式切换到 UDMA 6 -ATA133 模式
-X 后数字 16~18 代表 SDMA 0~2, 32~34 代表 MDMA 0~2, 64~70 代表 UDMA 0~6。
```

将最后一行加入到/etc/rc.d/rc.sysinit 或/etc/rc.d/rc.local 文件中，可以让硬盘每次启动都工作在 ATA133 下。

3. 安装 nforce 芯片组的驱动程序

AMD64 平台的 nforce 系列芯片组性能强劲，虽然可以用常规的方法加以驱动，但有可能无法发挥新设备的特性，或者该芯片组的网卡、声卡无法使用。nVIDIA 提供了更方便的 RPM 文件供使用，其中一些是源代码 tar 文件，需要进行编译；有些是 RPM 包，直接进行安装就可以了。

nforce 芯片组在 Linux 下驱动的下载地址为 http://www.nvidia.com/object/linux_nforce_1.0-0275.html，其中包含了内核的补丁、芯片组中对网卡及声卡的驱动程序。下载经过编译的 RPM 包可以直接安装：

```
# rpm -ivh NVIDIA_nforce.athlon.rpm
```

如果使用的不是 SuSE 或 Red Hat Linux 系统，也可以下载带有源代码的 .src.rpm 包经过编译后再安装：

```
# rpm -ivh NVIDIA_nforce.src.rpm
//将驱动程序的源程序文件安装到系统中
# cd /usr/src/redhat/SPECS
# rpmbuild -bb NVIDIA_nforce.specs
//编译源驱动程序，编译成可直接使用的 rpm 文件
# cd /usr/src/redhat/RPM/i386/
//根据具体包的不同，也可能是 i686、noarch 等
# rpm -ivh NVIDIA_nforce.i386.rpm
//程序自动安装包内的驱动程序，并利用 RPM 包中的脚本自动配置
```

光驱的驱动及刻录机的使用

1. 光驱的驱动过程

编译内核时，在 ATA/IDE/MFM/RLL 选单中都会有 IDE/ATAPI CDROM support 的选项，通常所见到的内核都将这个部分编译在了内核中，所以不需设置光驱就可以使用。下面看一下系统启动时光驱是如何驱动的：

```
# dmesg |grep CD
hdc: HL-DT-STCD-RW/DVD-ROM GCC-4240N, ATAPI CD/DVD-ROM drive
Uniform CD-ROM driver Revision: 3.12
```

一般的通用 CD/DVD-ROM 驱动程序就可以将光驱驱动起来，不需要特别的配置。如果遇到的是 SCSI 光驱，则一般的内核也可以驱动，因为在编译内核时，一般会将 SCSI 的内容编译出来。

如果没有 SCSI 的光驱驱动，则可以自己手动编译。make menuconfig（或 xconfig）时，在 SCSI support 选单中只要将 SCSI support 设置为“Y”或“M”（Y 代表该部分编译在内核中，M 表示该部分编译为内核模块）；SCSI CD-ROM support 设置为“Y”或“M”；SCSI generic

support 设置为“Y”或“M”，这样就可以驱动 SCSI 接口的光驱了。

2. CD/DVD 刻录机的驱动及使用

现在 CD 或 DVD 刻录机越来越普及，那么在 Linux 下如何使用内置的 CD 或 DVD 刻录机呢？默认的情况下，系统会将刻录机视作只读的驱动器，只加载普通的 CD/DVD-ROM 驱动，光驱无法写入。

因此，需要用 `ide-scsi` 伪设备驱动程序来驱动刻录机，将普通的 IDE 接口的设备模拟成一个 SCSI 接口的设备，这时才能向其中刻录。我们可以使用三种方式来实现将 IDE 光驱模拟为 SCSI 光驱。假设使用 GRUB 作为 bootloader，刻录机安装在第二个 IDE 接口，是主设备，则它默认应该为 `/dev/hdc`。

(1) 更改 `/boot/grub/grub.conf` 文件在 `kernel /boot/vmlinuz-2.XXX ro root=/dev/hda1` 之后，添加 `hdc=ide-scsi`。

(2) 更改 `/etc/modules.conf` 文件添加下面两行：

```
ide-cd ignore="\hdc\  
            ide-scsi
```

(3) 直接编译内核

不编译 ATA/IDE/MFM/RLL 选单中的 IDE/ATAPI CDROM support 部分，但是要编译对 SCSI 光驱的支持。

上面三种方法都是不希望系统用自带的普通 CD/DVD-ROM 驱动程序去驱动刻录机，希望将刻录机模拟成一个 SCSI 设备，其中第一种方法最简单，成功后就可以通过以下命令来刻录 CD 或 DVD 光盘。

```
# mkisofs -Jv -V examplecd -o example.iso /root/
```

将 `/root/` 目录下的文件做成一个名叫 `example.iso` 的光盘镜像文件，该文件的卷标为 `xamplecd`。

还可通过命令将该文件加载到 `/mnt/iso` 文件夹中，可以自由添加删除镜像中的文件，但要注意不要超过光盘的容量。

```
# mkdir /mnt/iso
```

```
# mount -t iso9660 -o loop example.iso /mnt/iso
```

调整 `/mnt/iso` 文件的内容：

```
# umount /mnt/iso
```

最后使用 `cdrecord` 命令来刻录：

```
# cdrecord -scanbus
```

查看 SCSI 总线中刻录机的配置信息：

```
Cdrecord 2.0 (i686-pc-linux-gnu) Copyright (C) 1995-2002 J?rg Schilling
```

```
Linux sg driver version: 3.1.25
```

```
Using libscg version 'schily-0.7'
```

```
cdrecord: Warning: using unofficial libscg transport code version
```

```
(schily - Red Hat-scsi-linux-sg.c-1.75-RH '@(#)scsi-linux-sg.c
```

```
1.75 02/10/21 Copyright 1997 J. Schilling').
```

```
scsibus0:
```

```
0,0,0      0) 'HL-DT-ST' 'RW/DVD GCC-4240N' 'E112' Removable CD-ROM
```

```
0,1,0      1) *
```

```
0,2,0      2) *
```

```
0,3,0      3) *
```

0,4,0	4) *
0,5,0	5) *
0,6,0	6) *
0,7,0	7) *

可以看到目前的光驱在 SCSI 总线的参数，然后根据参数来输入下面的命令刻录光盘：

```
# cdrecord -v -eject speed=24 dev=0,0,0 example.iso
```

speed=24 是以 24 倍速来刻录光盘，dev=后加上刚才显示的刻录机的 SCSI 参数。

Linux 下设备完全驱动之三(1)

图形化的界面能让我们方便地享用 Linux 的强大功能，而且现在 Linux 的图形化界面已经越来越完善。Linux 中经常使用的图形显示系统是 X—Window，但是由于其与常见的 MS Windows 系统有很大的差异，所以在使用时常会遇到一些问题，比如无法驱动显卡、显示器参数错误、花屏和图形界面无法启动等。

因此，本文将在简介 X-Window 的基础上，介绍如何安装常见显卡和声卡的驱动程序、相关的配置文件，以及参数调整等内容。

X-Window 概述

X—Window 是一套显示系统，包括 Server 端和 Client 端，他们之间使用 X 协议互相通信。X 诞生于 1984 年，在较短的时间内它就发布到了 11 个版本—X11。X11 经过多年的发展，现在已经发布到 X11R6。

X Client 将希望显示的图形发送到 X Server，X Server 将图形显示在显示器上，同时为 X Client 提供鼠标、键盘的输入服务。因为 C/S 结构，可以将 X 的 Server 和 Client 分别运行在两台计算机上，甚至可以安装一些软件，让 Windows 作为 X Server，让 Linux 作为 Client，将 KDE 或 GNOME 等桌面环境显示到 Windows 主机上来。

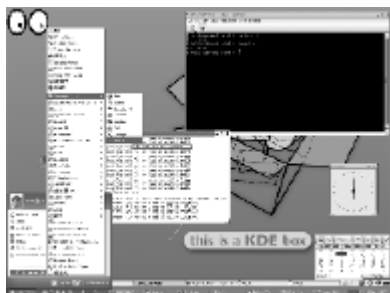


图 1 所示让 Windows 作为 X Server

使用 Xmanager 1.3.9，在 Windows XP 上运行 X Server，然后运行一台 Linux 主机下的 startkde，将 KDE 桌面环境显示在 Windows XP 下。

使用 startx 命令可以启动 X-Window 系统。实际上，是在一台计算机上同时运行 Server 和 Client，在运行 startx 之后，首先启动的是 XFree86，它是 Linux 平台上最常用的 X Server 端；然后，又运行 X 的 Client 程序，如 startkde (KDE 的启动脚本) 或 gnome-session (GNOME 的启动脚本)。

它们利用 X 协议连接本机的 X Server，将图形显示出来。

Linux 上经常使用的 X

Server 程序就是 XFree86，它的任务是驱动显示卡、显示器、鼠标、键盘等设备，为 X Client 提供显示、输入服务等。

XFree86 的主要文件目录如下：

XFree86 文件所在的主要目录为/usr/X11R6；

XFree86 的可执行程序文件目录为/usr/X11R6/bin；

XFree86 自带的驱动程序所在目录为/usr/X11R6/lib/modules/drivers；

X 的配置文件及启动脚本所在目录为/etc/X11；

XFree86 的启动配置文件目录为/etc/X11/XF86Config 或 XF86Config-4，如果这两个文件都存在，XF86Config-4 文件优先。

XFree86 自带了一些设备的驱动，具体位置在/usr/X11R6/lib/modules/drivers，如果机器显卡太新，无法使用 X-Window，比如使用了 Intel 的 855GM 芯片组中集成的显卡无法启动 X，则可以考虑升级 XFree86。

当前 XFree86 成熟的版本是 4.3，最新的版本 4.4 正在完善之中。从以下的链接中可获得 4.3 版的 XFree86 所支持的所有显卡列表 <http://xfree86.linuxforum.net/4.3.0/RELNOTES2.html#3>。

XFree86 官方版本下载网站是 <http://www.xfree86.org/downloads.html>，中国的镜像网站是 <http://xfree86.linuxforum.net/downloads.html>。

也可以从中科红旗网站 http://www.redflag-linux.com/source/download/XFree86_driver4.3.tgz 下载。

下载后安装 tgz 包：

```
# tar zxvf XFree86_driver4.3.tgz
# sh install.sh
```

安装 Intel i865G 芯片组显卡驱动

Intel i865G 是一种比较新的显卡芯片组，它内置了显示模块，但是安装较早的 Linux 版本 X-Window 有可能无法启动。处理这个问题的方法是，一种是直接升级到 XFree86 4.3；另一种是通过 Intel 官方的网站 http://downloadfinder.intel.com/scripts-df/support_intel.asp?iid=HPAGE+header_support_download& 下载驱动程序。

在该链接的下载页面可以选择下载 RPM 包，也可以选择下载 tar.gz 的压缩文档。尽管这两种的安装方式略有不同，但是经过以下三步基本上可以让显卡在 X-Window 中正常使用。

1. 安装驱动程序

(1) 选择下载 RPM 包：

```
# rpm -Uvh --force intelgraphics_20040607_i386.rpm
```

(2) 选择下载的是 tar.gz 压缩包要执行下面代码：

```
# tar xzvf IntelGraphics_060704.tar.gz
# cd dripkg
# ./install.sh
```

这时程序将自动安装，安装程序将更新/usr/X11R6/modules/drivers/i810_drv.o 的驱动程序，使其支持新的 i865G 芯片组中的显卡。

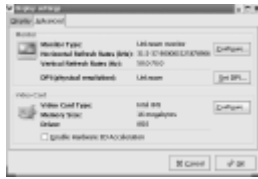


图 2 使用 redhat-config-xfree86

2.选择驱动程序

有三种方法可以选择驱动程序：

(1) 如果是 Red Hat 系统则可以运行下面代码：

```
# redhat-config-xfree86
```

图 2 显示为使用 redhat-config-xfree86。这个命令可以自动侦测显卡、配置分辨率和色深,以及选择 Advanced 页面,然后单击 Video Card 的 Configure 按钮,选择正确的显卡。但是,如果是 Red Hat Linux 8.0 之前版本,则没有这个命令,应该运行下面代码：

```
# Xconfigurator
```

这样程序会一步步提示选择正确的驱动程序。

这两种方法都会将结果写入/etc/X11/XF86Config 配置文件。

(2) 直接更改 XFree86 的配置文件

直接更改 XFree86 的配置文件这种方法在 Linux 系统中比较通用,运行下面命令：

```
# vi /etc/X11/XF86Config
```

或者

```
XF86Config-4
```

在其中找到下面代码,将 Driver 后面改为 i810,表示使用 i810_drv.o 驱动程序：

Section "Device"

```
Identifier "Videocard0"
Driver      "i810"
VendorName  "Videocard vendor"
BoardName   "Intel 865"
EndSection
```

该配置文件中其它需要注意的地方如下：

Section "Monitor"

```
Identifier "Monitor0"
VendorName "Monitor Vendor"
ModelName  "Unknown monitor"
HorizSync  31.5 - 37.9
VertRefresh 50.0 - 70.0
Option      "dpms"
EndSection
```

上面这个部分是设置显示器的类型,不知道自己的显示器叫什么名字没关系,但需要将显示器的垂直刷新率和水平刷新率正确设置,这两个参数可以查看显示器的说明书。以上是 15 英寸显示器最常用的频率,如果这个参数设置错误,显示器就会花屏或黑屏。

Section "Screen"

```
Identifier "Screen0"
Device      "Videocard0"
```

```

Monitor      "Monitor0"
DefaultDepth  16
SubSection "Display"
Depth        16
Modes        "1024x768" "800x600" "640x480"    //默认会使用最高的分辨率
EndSubSection
EndSection

```

这里是设置显示模式的地方，包括屏幕的分辨率和色深，默认使用最高的分辨率。如果想使用较低的分辨率，那么将高分辨率删除就可以了，在这个配置文件中显示器会使用“1024×768”分辨率，色深是 16 位色。有关 XF86Config 文件的详解可以参看它的 man page:

man XF86Config

(3) 让 XFree86 自己生成 XF86Config 文件

运行下面命令:

XFree86 -configure

XFree86 将自动侦测显卡及显示器，在用户的主目录下生成一个名叫 XF86Config.new 的文件。可以用以下命令测试这个文件运行是否正常:

XFree86 -xf86config ~/XF86Config.new

这个代码是指定 X Server 使用 ~/XF86Config.new 作为配置文件。

如果有白色 X 型的光标显示，就可以按“Ctrl+Alt+Backspace”结束 X。然后运行下面的命令，用新的配置文件来替换系统配置文件。

cp ~/XF86Config.new /etc/X11/XF86Config

3. 启动 X-Window

运行 startx，启动 X-Window。这种方式依赖于控制台 (tty)，将 X-Window 作为当前控制台的 Shell 子进程来运行。

另一种方法是运行桌面管理程序 gdm (GNOME 桌面管理程序)、kdm (KDE 的桌面管理程序) 或 xdm (XFree86 的桌面管理程序)。这种方式不依赖于当前的 Shell，即使当前 Shell 关闭，X-Window 一样会继续运行。

以上三步是 Linux 下安装 X-Window 显卡驱动程序的通常步骤，绝大多数显卡都可以经过以上的步骤正常使用，所不同的是下载地址和具体的驱动程序的生成方式可能不一样。

Linux 下设备完全驱动之四(1)

时间: 2005-09-25 作者: 郗晓烨 来源:

赛迪比如，scanner.o 模块对应的就是 USB 扫描仪的驱动;

尽管各种数码设备越来越多，但是 Linux 在对它们的配置上还没有做到像 Windows 一样方便，很多人正是因为心爱的数码相机无法在 Linux 下使用，而不得不保留一个 Windows 系统。所以，掌握如优盘、移动硬盘、数码相机等设备在 Linux 下的驱动方法，对于熟练使用 Linux 而言是必备的技能。

这一讲将会介绍常见 USB、IEEE1394、PCMCIA 等移动设备的驱动概念和安装技巧，为读者更好地配置和使用这些设备提供帮助。

USB 设备驱动概述

USB 是通用串行总线 (Universal Serial Bus)，是在 1994 年由 Intel、NEC、微软和 IBM

等公司共同提出的。USB 的目的在于将众多的接口（串口、并口、PS2 口等），改为通用的标准。它仅仅使用一个 4 针插头作为标准插头，并通过这个标准接头连接各种外设，如鼠标、键盘、游戏手柄、打印机、数码相机等。USB 接口的特点是支持热插拔，支持单接口上接多个设备等。

USB 的优点此处不再赘述，我们主要来看一看 Linux 对 USB 的支持。USB 采用串行方式传输数据，USB 1.1 最大数据传输率为 12Mbps，Linux 内核为 2.4 以上版本都可以支持。

USB 2.0 规范是由 USB 1.1 规范演变而来的，它理论上速度较 1.1 提高了 40 倍，达到了 480Mb/s，但目前常见的 USB 2.0 设备只能达到理论值的一半。Linux 内核 2.4.19 版本开始对 USB 2.0 进行支持。

除了内核的版本要对 USB 接口进行支持之外，还要确定目前的系统是否编译了 USB 的驱动模块。如果让 Linux 系统支持 USB 设备，还需要一些驱动模块。我们都知道内置的驱动程序一般都在 `/lib/modules/2.4XXX/kernel/drivers` 目录中。这个目录中会有 usb 及几个子目录，可以从中找到以下的几个 USB 关键基础模块：

`usbcore.o` 所有 USB 设备都需要的基本驱动模块；

`host/ehci-hcd.o` USB 2.0 设备支持；

`host/usb-uhci.o` Intel VIA 等芯片组 USB 部分的驱动；

`host/usb-ohci.o` iMac、SiS、Ali 等非 Intel 芯片组 USB 部分的驱动；

`storage/usb-storage.o` USB 接口的存储设备，如移动硬盘、U 盘等都会用到；

`hid.o` USB 接口的键盘、鼠标等人机交互设备的基础支持。

在该目录下还有一些具体设备的驱动程序，分别驱动不同设备：

```
# ls -p /lib/modules/2.4.21-4.EL/kernel/drivers/usb/
```

```
                acm.o          CDCEther.o  hpusbcsd.o   microtek.o   rtl8150.o   usbcore.o
wacom.o
                audio.o       dabusb.o    kaweth.o     pegasus.o    scanner.o   usb-midi.o
                brlvgcr.o     hid.o       kbtabs.o     powermate.o  serial/     usbnet.o
                catc.o        host/       mdc800.o     printer.o    storage/    uss720.o
```

`usb-storage.o` USB 对与存储部分的驱动模块；

`scsi_mod.o` 对 SCSI 设备的支持；

`sd_mod.o` 对 SCSI 硬盘支持模块，针对 USB 硬盘；

`sr_mod.o` 对 SCSI 光盘支持模块，针对 USB 光驱；

`sg.o` SCSI 序列的通用支持模块；

`ide-scsi.o` 该模块可以把 IDE 设备模拟成 SCSI 接口。

通过查看 `/lib/modules/2.XXX/modules.dep` 文件，可以查看上述这些模块是否存在。一般情况下这些模块已经被编译，否则需要重新编译内核模块。

在正确地装载了驱动以后，可以通过访问 `/dev/sd?` 设备来访问优盘或移动硬盘。通常情况下，通过访问 `sda1` 来访问移动硬盘或优盘的第一个分区。在 USB 基本驱动存在的情况下插入优盘，就可以看到如下信息：

```
usb.c: USB device 7 (vend/prod 0xea0/0x6803) is not claimed by any active driver.
```

```
Starting timer : 0 0
```

```
Vendor: Netac      Model: OnlyDisk    Rev: 1.11
```

```
Type:   Direct-Access    ANSI SCSI revision: 02
```

```
Starting timer : 0 0
```

```
Attached scsi removable disk sda at scsi0, channel 0, id 0, lun 0
```


SCSI device sda: 32256 512-byte hdwr sectors (17 MB)

sda: Write Protect is off

上面这一段是 usb-storage.o 和 SCSI 驱动在起作用，我们可以看到优盘被认成是 sda，要使用它可以先查看分区表：

```
# fdisk -l /dev/sda
```

Disk /dev/sda: 16 MB, 16515072 bytes

2 heads, 32 sectors/track, 504 cylinders

Units = cylinders of 64 * 512 = 32768 bytes

Device	Boot	Start	End	Blocks	Id	System
/dev/sda1	*	1	503	16080	1	FAT12

```
# mkdir /mnt/usb
```

```
# mount -t msdos /dev/sda1 /mnt/usb
```

 该优盘得类型为 FAT12

之后就可以通过访问/mnt/usb 来访问优盘了。注意，如果要拔掉优盘或移动存储设备，请先 umount 然后再拔出，这样可以保证数据全部被写入，否则系统会出错，数据可能不完整。

```
# umount /mnt/usb
```

USB 数码相机的驱动

数码相机种类繁多，但是在 Linux 下使用数码相机有比较简便的方法。我们知道 USB 接口是数码相机的主要接口，在 Linux 中访问数码相机，通常可以通过下面两种方法。

1. 使用专门软件

Red Hat Linux 中自带的 gtkam 软件是一个提供了数码照相机图形化界面的程序，它支持 100 多种数码相机。gtkam 可以直接与数码照相机相连，允许直接打开、查看、并删除图像。在 Red Hat 9.0 的光盘中，有 gtkam 的 RPM 包：

```
# ls gtkam*
```

gtkam-XXXX.i386.rpm gtkam-gimp-XXXX.i386.rpm

```
# rpm -ivh --aid gtkam*
```

```
# rpm -ivh --aid --force gphoto2-XXXX.i386.rpm
```

gtkam 基于 gphoto，有时无法使用 gtkam 的原因是因为 gphoto 没有安装或模块被覆盖，所以需要重新安装 gphoto。安装完毕后，在 X-Window 中运行 gtkam 就可以看到如图 1 所示界面。

```
# gtkam
```

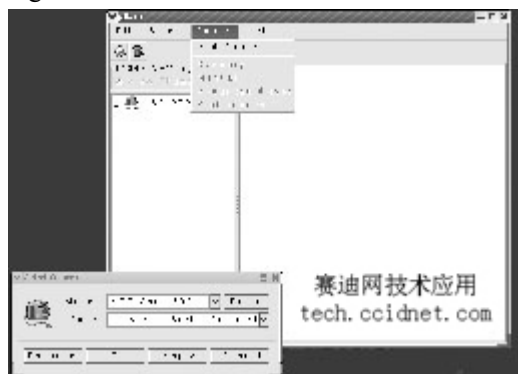


图 1 gtkam 界面

在图 1 中，单击 gtkam 的“camera”→“Add Camera”可以添加数码相机，然后在弹出的窗体中单击“Detect”，测试数码相机的连接类型。最后单击“OK”，就可以看到数码相机中的照片了。

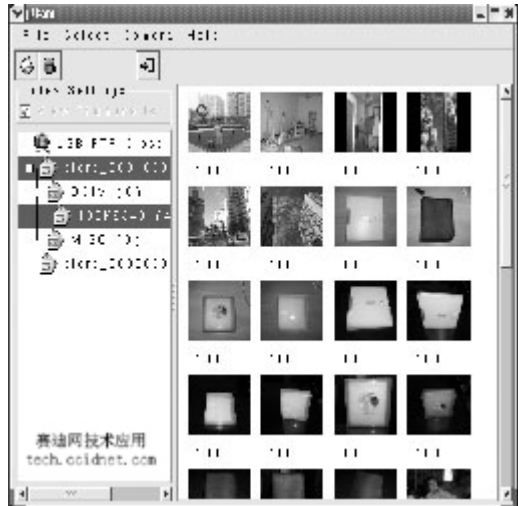


图 2 所示使用 gtkam 连接 Kodak DX6340 相机，看到存储卡中的照片。选择照片，单击保存就可以把照片保存到硬盘上。

我们也可以从 http://sourceforge.net/project/showfiles.php?group_id=8874&release_id=209817 获得 gtkam 的最新版本及源代码，下载后可以使用其中的 install.sh 来进行安装。

2.把数码相机当做是 USB 存储设备还有一种方式就是把数码相机当做是 USB 存储设备，如优盘、读卡器等，这样就可以像访问优盘那样来访问数码相机：

```
# mount -t vfat /dev/sda1 /mnt/usb
```

但是，采用这种方式极有可能遇到不能支持的数码相机，比如笔者使用的 Kodak DX6340 数码相机，插入 USB 口之后出现以下信息：

```
usb.c: USB device 2 (vend/prod 0x40a/0x570)
       is not claimed by any active driver.
```

对这种问题的解决办法不只是可以驱动不支持的数码相机，像不支持的存储设备也都可以使用，比如优盘、读卡器等。

(1) 使用 lsmod 确定 USB 基本驱动模块已经装载，如果没有使用以上的命令装载：

```
# modprobe ehci-hcd; modprobe usb-uhci;
    modprobe usb-storage
    # modprobe ide-scsi; modprobe scsi_mod;
    modprobe sd_mod
```

(2) 使用 cat /proc/bus/usb/devices 得到当前系统 USB 总线上的设备信息，尤其注意 Vendor、ProdID、Product 等信息：

```
C:* #Ifs= 1 Cfg#= 1 Atr=40 MxPwr= 0mA
    I: If#= 0 Alt= 0 #EPs= 1 Cls=09(hub ) Sub=00 Prot=00 Driver=hub
```

E: Ad=81(I) Atr=03(Int.) MxPS= 8 Ivl=255ms
T: Bus=02 Lev=01 Prnt=01 Port=00 Cnt=01 Dev#= 3 Spd=12 MxCh= 0
D: Ver= 2.00 Cls=00(>ifc) Sub=00 Prot=00 MxPS= 8 #Cfgs= 1
P: Vendor=040a ProdID=0570 Rev= 1.00
S: Manufacturer=Eastman Kodak Company
S: Product=KODAK EasyShare DX6340 Zoom Digital Camera
S: SerialNumber=KCKCJ33400274

第八章 设备驱动

操作系统的目的之一就是将系统硬件设备细节从用户视线中隐藏起来。例如虚拟文件系统对各种类型已安装的文件系统提供了统一的视图而屏蔽了具体底层细节。本章将描述 Linux 核心对系统中物理设备的管理。

CPU 并不是系统中唯一的智能设备，每个物理设备都拥有自己的控制器。键盘、鼠标和串行口由一个高级 I/O 芯片统一管理，IDE 控制器控制 IDE 硬盘而 SCSI 控制器控制 SCSI 硬盘等等。每个硬件控制器都有各自的控制和状态寄存器(CSR)并且各不相同。例如 Adaptec 2940 SCSI 控制器的 CSR 与 NCR 810 SCSI 控制器完全不一样。这些 CSR 被用来启动和停止，初始化设备及对设备进行诊断。在 Linux 中管理硬件设备控制器的代码并没有放置在每个应用程序中而是由内核统一管理。这些处理和管理硬件控制器的软件就是设备驱动。Linux 核心设备驱动是一组运行在特权级上的内存驻留底层硬件处理共享库。正是它们负责管理各个设备。

设备驱动的一个基本特征是设备处理的抽象概念。所有硬件设备都被看成普通文件；可以通过和操纵普通文件相同的标准系统调用来打开、关闭、读取和写入设备。系统中每个设备都用一种特殊的设备相关文件来表示(device special file)，例如系统中第一个 IDE 硬盘被表示成/dev/hda。块（磁盘）设备和字符设备的设备相关文件可以通过 mknod 命令来创建，并使用主从设备号来描述此设备。网络设备也用设备相关文件来表示，但 Linux 寻找和初始化网络设备时才建立这种文件。由同一个设备驱动控制的所有设备具有相同的主设备号。从设备号则被用来区分具有相同主设备号且由相同设备驱动控制的不同设备。例如主 IDE 硬盘的每个分区的从设备号都不相同。如/dev/hda2 表示主 IDE 硬盘的主设备号为 3 而从设备号为 2。Linux 通过使用主从设备号将包含在系统调用中的（如将一个文件系统 mount 到一个块设备）设备相关文件映射到设备的设备驱动以及大量系统表格中，如字符设备表，chrdevs。Linux 支持三类硬件设备：字符、块及网络设备。字符设备指那些无需缓冲直接读写的设备，如系统的串口设备/dev/cua0 和/dev/cua1。块设备则仅能以块为单位读写，典型的块大小为 512 或 1024 字节。块设备的存取是通过 buffer cache 来进行并且可以进行随机访问，即不管块位于设备中何处都可以对其进行读写。块设备可以通过其设备相关文件进行访问，但更为平常的访问方法是通过文件系统。只有块设备才能支持可安装文件系统。网络设备可以通过 BSD 套接口访问，我们将在[网络](#)一章中讨论网络子系统。

Linux 核心中虽存在许多不同的设备驱动但它们具有一些共性：

核心代码

设备驱动是核心的一部分，象核心中其它代码一样，出错将导致系统的严重损伤。一个编写奇差的设备驱动甚至能使系统崩溃并导致文件系统的破坏和数据丢失。

核心接口

设备驱动必须为 Linux 核心或者其从属子系统提供一个标准接口。例如终端驱动为 Linux 核

心提供了一个文件 I/O 接口而 SCSI 设备驱动为 SCSI 子系统提供了一个 SCSI 设备接口，同时此子系统为核心提供了文件 I/O 和 buffer cache 接口。

核心机制与服务

设备驱动可以使用标准的核心服务如内存分配、中断发送和等待队列等等。

动态可加载

多数 Linux 设备驱动可以在核心模块发出加载请求时加载，同时在不再使用时卸载。这样核心能有效地利用系统资源。

可配置

Linux 设备驱动可以连接到核心中。当核心被编译时，哪些核心被连入核心是可配置的。

动态性

当系统启动及设备驱动初始化时将查找它所控制的硬件设备。如果某个设备的驱动为一个空过程并不会有什么问题。此时此设备驱动仅仅是一个冗余的程序，它除了会占用少量系统内存外不会对系统造成什么危害。

8.1 轮询与中断

设备被执行某个命令时，如“将读取磁头移动到软盘的第 42 扇区上”，设备驱动可以从轮询方式和中断方式中选择一种以判断设备是否已经完成此命令。

轮询方式意味着需要经常读取设备的状态，一直到设备状态表明请求已经完成为止。如果设备驱动被连接进入核心，这时使用轮询方式将会带来灾难性后果：核心将在此过程中无所事事，直到设备完成此请求。但是轮询设备驱动可以通过使用系统定时器，使核心周期性调用设备驱动中的某个例程来检查设备状态。定时器过程可以检查命令状态及 Linux 软盘驱动的工作情况。使用定时器是轮询方式中最好的一种，但更有效的方法是使用中断。

基于中断的设备驱动会在它所控制的硬件设备需要服务时引发一个硬件中断。如以太网设备驱动从网络上接收到一个以太数据报时都将引起中断。Linux 核心需要将来自硬件设备的中断传递到相应的设备驱动。这个过程由设备驱动向核心注册其使用的中断来协助完成。此中断处理例程的地址和中断号都将被记录下来。在 `/proc/interrupts` 文件中你可以看到设备驱动所对应的中断号及类型：

```
0:      727432   timer
1:      20534   keyboard
2:         0   cascade
3:      79691 + serial
4:      28258 + serial
5:         1   sound blaster
11:     20868 + aic7xxx
13:         1   math error
14:      247 + ide0
15:     170 + ide1
```

对中断资源的请求在驱动初始化时就已经完成。作为 IBM PC 体系结构的遗产，系统中有些中断已经固定。例如软盘控制器总是使用中断 6。其它中断，如 PCI 设备中断，在启动时进行动态分配。设备驱动必须在取得对此中断的所有权之前找到它所控制设备的中断号（IRQ）。Linux 通过支持标准的 PCI BIOS 回调函数来确定系统中 PCI 设备的中断信息，包括其 IRQ 号。

如何将中断发送给 CPU 本身取决于体系结构，但是在多数体系结构中，中断以一种特殊模式发送时还将阻止系统中其它中断的产生。设备驱动在其中断处理过程中作的越少越好，这样 Linux 核心将能很快的处理完中断并返回中断前的状态中。为了在接收中断时完成

大量工作，设备驱动必须能够使用核心的底层处理例程或者任务队列来对以后需要调用的那些例程进行排队。

8.2 直接内存访问 (DMA)

数据量比较少时，使用中断驱动设备驱动程序能顺利地在硬件设备和内存之间交换数据。例如波特率为 9600 的 modem 可以每毫秒传输一个字符。如果硬件设备引起中断和调用设备驱动中断所消耗的中断时延比较大（如 2 毫秒）则系统的综合数据传输率会很低。则 9600 波特率 modem 的数据传输只能利用 0.002% 的 CPU 处理时间。高速设备如硬盘控制器或者以太网设备数据传输率将更高。SCSI 设备的数据传输率可达到每秒 40M 字节。

直接内存存取 (DMA) 是解决此类问题的有效方法。DMA 控制器可以在不受处理器干预的情况下在设备和系统内存之间高速传输数据。PC 机的 ISA DMA 控制器有 8 个 DMA 通道，其中七个可以由设备驱动使用。每个 DMA 通道具有一个 16 位的地址寄存器和一个 16 位的记数寄存器。为了初始化数据传输，设备驱动将设置 DMA 通道地址和记数寄存器以描述数据传输方向以及读写类型。然后通知设备可以在任何时候启动 DMA 操作。传输结束时设备将中断 PC。在传输过程中 CPU 可以转去执行其他任务。

设备驱动使用 DMA 时必须十分小心。首先 DMA 控制器没有任何虚拟内存的概念，它只存取系统中的物理内存。同时用作 DMA 传输缓冲的内存空间必须是连续物理内存块。这意味着不能在进程虚拟地址空间内直接使用 DMA。但是你可以将进程的物理页面加锁以防止在 DMA 操作过程中被交换到交换设备上去。另外 DMA 控制器所存取物理内存有限。DMA 通道地址寄存器代表 DMA 地址的高 16 位而页面寄存器记录的是其余 8 位。所以 DMA 请求被限制到内存最低 16M 字节中。

DMA 通道是非常珍贵的资源，一共才有 7 个并且还不能够在设备驱动间共享。与中断一样，设备驱动必须找到它应该使用那个 DMA 通道。有些设备使用固定的 DMA 通道。例如软盘设备总使用 DMA 通道 2。有时设备的 DMA 通道可以由跳线来设置，许多以太网设备使用这种技术。设计灵活的设备将告诉系统它将使用哪个 DMA 通道，此时设备驱动仅需要从 DMA 通道中选取即可。

Linux 通过 `dma_chan` (每个 DMA 通道一个) 数组来跟踪 DMA 通道的使用情况。`dma_chan` 结构中包含有两个域，一个是指向此 DMA 通道拥有者的指针，另一个指示 DMA 通道是否已经被分配出去。当敲入 `cat /proc/dma` 打印出来的结果就是 `dma_chan` 结构数组。

8.3 内存

设备驱动必须谨慎使用内存。由于它属于核心,所以不能使用虚拟内存。系统接收到中断信号时或调度底层任务队列处理过程时，设备驱动将开始运行，而当前进程会发生改变。设备驱动不能依赖于任何运行的特定进程，即使当前是为该进程工作。与核心的其它部分一样，设备驱动使用数据结构来描述它所控制的设备。这些结构被设备驱动代码以静态方式分配，但会增大核心而引起空间的浪费。多数设备驱动使用核心中非页面内存来存储数据。Linux 为设备驱动提供了一组核心内存分配与回收过程。核心内存以 2 的次幂大小的块来分配。如 512 或 128 字节，此时即使设备驱动的需求小于这个数量也会分配这么多。所以设备驱动的内存分配请求可得到以块大小为边界的内存。这样核心进行空闲块组合更加容易。

请求分配核心内存时 Linux 需要完成许多额外的工作。如果系统中空闲内存数量较少，则可能需要丢弃些物理页面或将其写入交换设备。一般情况下 Linux 将挂起请求者并将此进程放置到等待队列中直到系统中有足够的物理内存为止。不是所有的设备驱动（或者真正的 Linux 核心代码）都会经历这个过程，所以如分配核心内存的请求不能立刻得到满足,则此请求可能会失败。如果设备驱动希望在此内存中进行 DMA，那么它必须将此内存设置为 DMA 使能的。这也是为什么是 Linux 核心而不是设备驱动需要了解系统中的 DMA 使能内存的原因。

8.4 设备驱动与核心的接口

Linux 核心与设备驱动之间必须有一个以标准方式进行互操作的接口。每一类设备驱动：字符设备、块设备 及网络设备都提供了通用接口以便在需要时为核心提供服务。这种通用接口使得核心可以以相同的方式来对待不同的设备及设备驱动。如 SCSI 和 IDE 硬盘的区别很大但 Linux 对它们使用相同的接口。

Linux 动态性很强。每次 Linux 核心启动时如遇到不同的物理设备将需要不同的物理设备驱动。Linux 允许通过配置脚本在核心重建时将设备驱动包含在内。设备驱动在启动初始化时可能会发现系统中根本没有任何硬件需要控制。其它设备驱动可以在必要时作为核心模块动态加载到。为了处理设备驱动的动态属性，设备驱动在初始化时将其注册到核心中去。Linux 维护着已注册设备驱动表作为和设备驱动的接口。这些表中包含支持此类设备例程的指针和相关信息。

8.4.1 字符设备

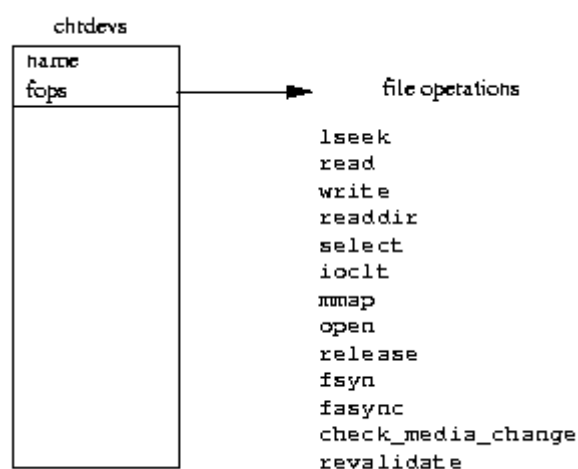


图 8.1 字符设备

字符设备是 Linux 设备中最简单的一种。应用程序可以和存取文件相同的系统调用来打开、读写及关闭它。即使此设备是将 Linux 系统连接到网络中的 PPP 后台进程的 modem 也是如此。字符设备初始化时，它的设备驱动通过在 `device_struct` 结构的 `chrdevs` 数组中添加一个入口来将其注册到 Linux 核心上。设备的主设备标志符用来对此数组进行索引（如对 tty 设备的索引 4）。设备的主设备标志符是固定的。

`chrdevs` 数组每个入口中的 `device_struct` 数据结构包含两个元素；一个指向已注册的设备驱动名称，另一个则是指向一组文件操作指针。它们是位于此字符设备驱动内部的文件操作例程的地址指针，用来处理相关的文件操作如打开、读写与关闭。`/proc/devices` 中字符设备的内容来自 `chrdevs` 数组。

当打开代表字符设备的字符特殊文件时（如 `/dev/cua0`），核心必须作好准备以便调用相应字符设备驱动的文件操作例程。与普通的目录和文件一样，每个字符特殊文件用一个 VFS 节点表示。每个字符特殊文件使用的 VFS inode 和所有设备特殊文件一样，包含着设备的主从标志符。这个 VFS inode 由底层的文件系统来建立（比如 EXT2），其信息来源于设备相关文件名称所在文件系统。

每个 VFS inode 和一组文件操作相关联，它们根据 inode 代表的文件系统对象变化而不同。当创建一个代表字符相关文件的 VFS inode 时，其文件操作被设置为缺省的字符设备操作。

字符设备只有一个文件操作：打开文件操作。当应用打开字符特殊文件时，通用文件打

开操作使用设备的主标志符来索引此 chrdevs 数组，以便得到那些文件操作函数指针。同时建立起描述此字符特殊文件的 file 结构,使其文件操作指针指向此设备驱动中的文件操作指针集合。这样所有应用对它进行的文件操作都被映射到此字符设备的文件操作集合上。

8.4.2 块设备

块设备也支持以文件方式访问。系统对块设备特殊文件提供了非常类似于字符特殊文件的文件操作机制。Linux 在 blkdevs 数组中维护所有已注册的块设备。象 chrdevs 数组一样，blkdevs 也使用设备的主设备号进行索引。其入口也是 device_struct 结构。和字符设备不同的是系统有几类块设备。SCSI 设备是一类而 IDE 设备则是另外一类。它们将以各自类别登记到 Linux 核心中并为核心提供文件操作功能。某类块设备的设备驱动为此类型设备提供了类别相关的接口。如 SCSI 设备驱动必须为 SCSI 子系统提供接口以便 SCSI 子系统能用它来为核心提供对此设备的文件操作。

和普通文件操作接口一样，每个块设备驱动必须为 buffer cache 提供接口。每个块设备驱动将填充其在 blk_dev 数组中的 blk_dev_struct 结构入口。数组的索引值还是此设备的主设备号。这个 blk_dev_struct 结构包含请求过程的地址以及指向请求数据结构链表的指针，每个代表一个从 buffer cache 中来让设备进行数据读写的请求。

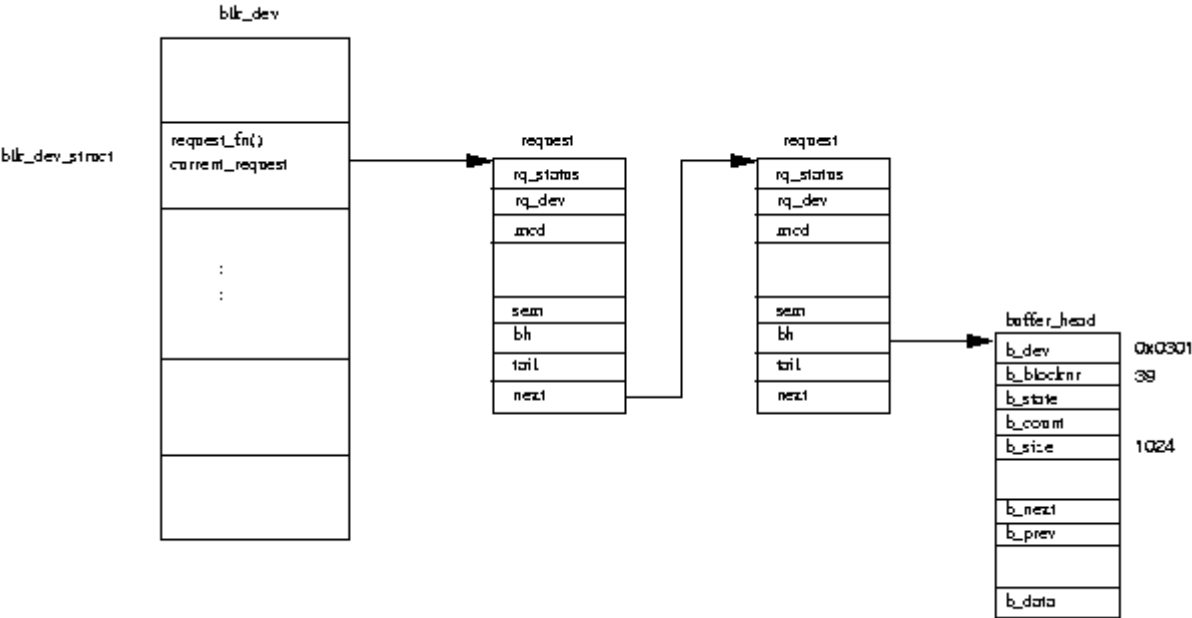


图 8.2 buffer cache 块设备请求

每当 buffer cache 希望从一个已注册设备中读写数据块时,它会将 request 结构添加到其 blk_dev_struct 中。图 8.2 表示每个请求有指向一个或多个 buffer_head 结构的指针，每个请求读写一块数据。如 buffer cache 对 buffer_head 结构上锁，则进程会等待到对此缓冲的块操作完成。每个 request 结构都从静态链表 all_requests 中分配。如果此请求被加入到空请求链表中,则将调用驱动请求函数以启动此请求队列的处理,否则该设备驱动将简单地处理请求链表上的 request。

一旦设备驱动完成了请求则它必须将每个 buffer_head 结构从 request 结构中清除,将它们标记成已更新状态并解锁之。对 buffer_head 的解锁将唤醒所有等待此块操作完成的睡眠进程。如解析文件名称时,EXT2 文件系统必须从包含此文件系统的设备中读取包含下个 EXT2 目录入口的数据块。在 buffer_head 上睡眠的进程在设备驱动被唤醒后将包含此目录入口。request 数据结构被标记成空闲以便被其它块请求使用。

8.5 硬盘

磁盘驱动器提供了一个永久性存储数据的方式，将数据保存在旋转的盘片上。写入数据时磁头将磁化盘片上的一个微粒。这些盘片被连接到一个中轴上并以 3000 到 10,000RPM（每分钟多少转）的恒定速度旋转。而软盘的转速仅为 360RPM。磁盘的读/写磁头负责读写数据，每个盘片的两侧各有一个磁头。磁头读写时并不接触盘片表面而是浮在距表面非常近的空气垫中（百万分之一英寸）。磁头由一个马达驱动在盘片表面移动。所有的磁头被连在一起，它们同时穿过盘片的表面。

盘片的每个表面都被划分成为叫做磁道的狭窄同心圆。0 磁道位于最外面而最大磁道位于最靠近中央主轴。柱面指一组相同磁道号的磁道。所以每个盘片上的第五磁道组成了磁盘的第五柱面。由于柱面号与磁道号相等所以我们经常可以看到以柱面描述的磁盘布局。每个磁道可进一步划分成扇区。它是硬盘数据读写的最小单元同时也是磁盘的块大小。一般的扇区大小为 512 字节并且这个大小可以磁盘制造出来后格式化时设置。

一个磁盘经常被描绘成有多少各柱面、磁头以及扇区。例如系统启动时 Linux 将这样描述一个 IDE 硬盘：

hdb: Conner Peripherals 540MB - CFS540A, 516MB w/64kB Cache, CHS=1050/16/63

这表示此磁盘有 1050 各柱面（磁道），16 个磁头（8 个盘片）且每磁道包含 63 个扇区。这样我们可以通过扇区数、块数以及 512 字节扇区大小计算出磁盘的存储容量为 529200 字节。这个容量和磁盘自身声称的 516M 字节并不相同，这是因为有些扇区被用来存放磁盘分区信息。有些磁盘还能自动寻找坏扇区并重新索引磁盘以正常使用。

物理硬盘可进一步划分成分区。一个分区是一大组为特殊目的而分配的扇区。对磁盘进行分区使得磁盘可以同时被几个操作系统或不同目的使用。许多 Linux 系统具有三个分区：DOS 文件系统分区，EXT2 文件系统分区和交换分区。硬盘分区用分区表来描述；表中每个入口用磁头、扇区及柱面号来表示分区的起始与结束。对于用 DOS 格式化的硬盘有 4 个主分区表。但不一定所有的四个入口都被使用。fdisk 支持 3 中分区类型：主分区、扩展分区及逻辑分区。扩展分区并不是真正的分区，它只不过包含了几个逻辑分区。扩展和逻辑分区用来打破四个主分区的限制。以下是一个包含两个主分区的 fdisk 命令的输出：

Disk /dev/sda: 64 heads, 32 sectors, 510 cylinders

Units = cylinders of 2048 * 512 bytes

Device	Boot	Begin	Start	End	Blocks	Id	System
/dev/sda1		1	1	478	489456	83	Linux native
/dev/sda2		479	479	510	32768	82	Linux swap

Expert command (m for help): p

Disk /dev/sda: 64 heads, 32 sectors, 510 cylinders

Nr	AF	Hd	Sec	Cyl	Hd	Sec	Cyl	Start	Size	ID
1	00	1	1	0	63	32	477	32	978912	83
2	00	0	1	478	63	32	509	978944	65536	82
3	00	0	0	0	0	0	0	0	0	00
4	00	0	0	0	0	0	0	0	0	00

这些内容表明第一个分区从柱面（或者磁道）0，头 1 和扇区 1 开始一直到柱面 477，扇区 22 和头 63 结束。由于每磁道有 32 个扇区且有 64 个读写磁头则此分区在大小上等于柱面数。fdisk 使分区在柱面边界上对齐。它从最外面的柱面 0 开始并向中间扩展 478 个柱面。第二个分区：交换分区从 478 号柱面开始并扩展到磁盘的最内圈。

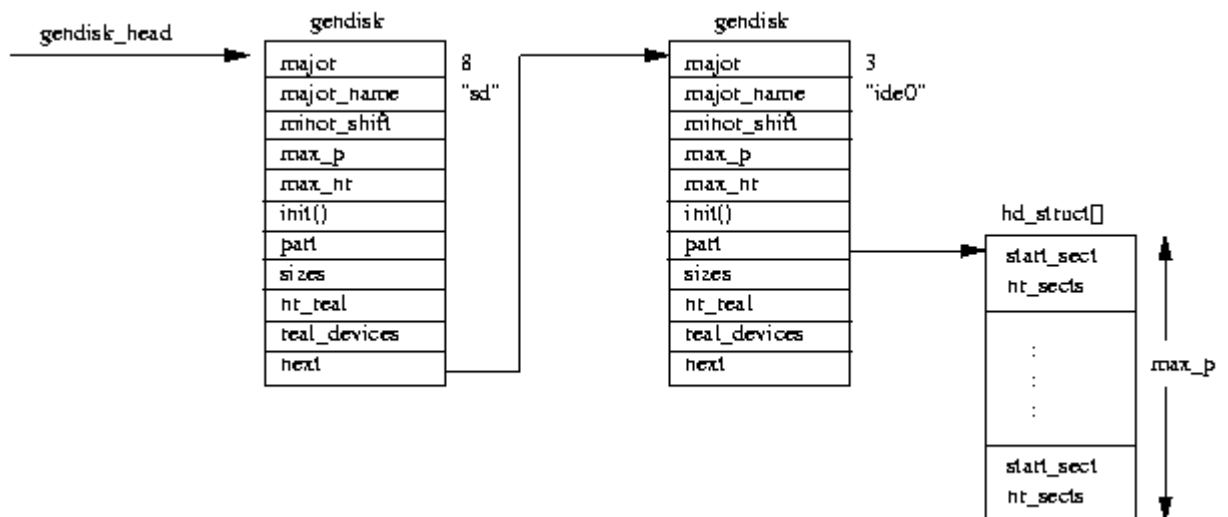


图 8.3 磁盘链表

在初始化过程中 Linux 取得系统中硬盘的拓扑结构映射。它找出有多少中硬盘以及是什么类型。另外 Linux 还要找到每个硬盘的分区方式。所有这些都由 `gendisk_head` 链指针指向的 `gendisk` 结构链表来表示。每个磁盘子系统如 IDE 在初始化时产生表示磁盘结构的 `gendisk` 结构。同时它将注册其文件操作例程并将此入口添加到 `blk_dev` 数据结构中。每个 `gendisk` 结构包含唯一的主设备号，它与块相关设备的主设备号相同。例如 SCSI 磁盘子系统创建了一个主设备号为 8 的 `gendisk` 入口 ("sd")，这也是所有 SCSI 硬盘设备的主设备号。图 8.3 给出了两个 `gendisk` 入口，一个表示 SCSI 磁盘子系统而另一个表示 IDE 磁盘控制器。ide0 表示主 IDE 控制器。

尽管磁盘子系统在其初始化过程中就建立了 `gendisk` 入口，但是只有 Linux 作分区检查时才使用。每个磁盘子系统通过维护一组数据结构将物理硬盘上的分区与某个特殊主从特殊设备互相映射。无论何时通过 `buffer cache` 或文件操作对块设备的读写都将被核心定向到对具有某个特定主设备号的设备文件上（如 `/dev/sda2`）。而从设备号的定位由各自设备驱动或子系统来映射。

8.5.1 IDE 硬盘

Linux 系统上使用得最广泛的硬盘是集成电子磁盘或者 IDE 硬盘。IDE 是一个硬盘接口而不是类似 SCSI 的 I/O 总线接口。每个 IDE 控制器支持两个硬盘，一个为主另一个为从。主从硬盘可以通过盘上的跳线来设置。系统中的第一个 IDE 控制器成为主 IDE 控制器而另一个为从属控制器。IDE 可以以每秒 3.3M 字节的传输率传输数据且最大容量为 538M 字节。EIDE 或增强式 IDE 可以将磁盘容量扩展到 8.6G 字节而数据传输率为 16.6M 字节/秒。由于 IDE 和 EIDE 都比 SCSI 硬盘便宜，所以大多现代 PC 机在包含一个或几个板上 IDE 控制器。

Linux 以其发现控制器的顺序来对 IDE 硬盘进行命名。在主控制器中的主盘为 `/dev/hda` 而从盘为 `/dev/hdb`。`/dev/hdc` 用来表示从属 IDE 控制器中的主盘。IDE 子系统将向 Linux 核心注

册 IDE 控制器而不是 IDE 硬盘。主 IDE 控制器的主标志符为 3 而从属 IDE 控制器的主标志符为 22。如果系统中包含两个 IDE 控制器则 IDE 子系统的入口在 `blk_dev` 和 `blkdevs` 数组的第 2 和第 22 处。IDE 的块设备文件反应了这种编号方式，硬盘 `/dev/hda` 和 `/dev/hdb` 都连接到主 IDE 控制器上，其主标志符为 3。对 IDE 子系统上这些块相关文件的文件或者 `buffer cache` 的操作都通过核心使用主设备标志符作为索引定向到 IDE 子系统上。当发出请求时，此请求由哪个 IDE 硬盘来完成取决于 IDE 子系统。为了作到这一点 IDE 子系统使用从设备编号对应的设备特殊标志符，由它包含的信息来将请求发送到正确的硬盘上。位于主 IDE 控制器上的 IDE 从盘 `/dev/hdb` 的设备标志符为 (3, 64)。而此盘中第一个分区 (`/dev/hdb1`) 的设备标志符为 (3, 65)。

8.5.2 初始化 IDE 子系统

IDE 磁盘与 IBM PC 关系非常密切。在这么多年中这些设备的接口发生了变化。这使得 IDE 子系统的初始化过程比看上去要复杂得多。

Linux 可以支持的最多 IDE 控制器个数为 4。每个控制器用 `ide_hwifs` 数组中的 `ide_hwif_t` 结构来表示。每个 `ide_hwif_t` 结构包含两个 `ide_drive_t` 结构以支持主从 IDE 驱动器。在 IDE 子系统的初始化过程中 Linux 通过访问系统 CMOS 来判断是否有关于硬盘的信息。这种 CMOS 由电池供电所以系统断电时也不会遗失其中的内容。它位于永不停止的系统实时时钟设备中。此 CMOS 内存的位置由系统 BIOS 来设置，它将通知 Linux 系统中有多少个 IDE 控制器与驱动器。Linux 使用这些从 BIOS 中发现的磁盘数据来建立对应此驱动器的 `ide_hwif_t` 结构。许多现代 PC 系统使用 PCI 芯片组如 Intel 82430 VX 芯片组将 PCI EIDE 控制器封装在内。IDE 子系统使用 PCI BIOS 回调函数来定位系统中 PCI (E) IDE 控制器。然后对这些芯片组调用 PCI 特定查询例程。

每次找到一个 IDE 接口或控制器就有建立一个 `ide_hwif_t` 结构来表示控制器和与之相连的硬盘。在操作过程中 IDE 驱动器对 I/O 内存空间中的 IDE 命令寄存器写入命令。主 IDE 控制器的缺省控制和状态寄存器是 `0x1F0 - 0x1F7`。这个地址由早期的 IBM PC 规范设定。IDE 驱动器为每个控制器向 Linux 注册块缓冲 `cache` 和 VFS 节点并将其加入到 `blk_dev` 和 `blkdevs` 数组中。IDE 驱动器需要申请某个中断。一般主 IDE 控制器中断号为 14 而从属 IDE 控制器为 15。然而这些都可以通过命令行选项由核心来重载。IDE 驱动器同时还将 `gendisk` 入口加入到启动时发现的每个 IDE 控制器的 `gendisk` 链表中去。分区检查代码知道每个 IDE 控制器可能包含两个 IDE 硬盘。

8.5.3 SCSI 硬盘

SCSI (小型计算机系统接口) 总线是一种高效的点对点数据总线，它最多可以支持 8 个设备，其中包括多个主设备。每个设备有唯一的标志符并可以通过盘上的跳线来设置。在总线上的两个设备间数据可以以同步或异步方式，在 32 位数据宽度下传输率为 40M 字节来交换数据。SCSI 总线上可以在设备间同时传输数据与状态信息。`initiator` 设备和 `target` 设备间的执行步骤最多可以包括 8 个不同的阶段。你可以从总线上 5 个信号来分辨 SCSI 总线的当前阶段。这 8 个阶段是：

BUS FREE

当前没有设备在控制总线且总线上无事务发生。

ARBITRATION

一个 SCSI 设备试图取得 SCSI 总线的控制权，这时它将其 SCSI 标志符放置到地址引脚上。具有最高 SCSI 标志符编号的设备将获得总线控制权。

SELECTION

当设备通过仲裁成功地取得了对 SCSI 总线的控制权后它必须向它准备发送命令的那个 SCSI 设备发出信号。具体做法是将目标设备的 SCSI 标志符放置在地址引脚上进行声明。

RESELECTION

在一个请求的处理过程中 SCSI 设备可能会断开连接。目标 (target) 设备将再次选择启动设备 (initiator)。不是所有的 SCSI 设备都支持此阶段。

COMMAND

此阶段中 initiator 设备将向 target 设备发送 6、10 或 12 字节命令。

DATA IN, DATA OUT

此阶段中数据将在 initiator 设备和 target 设备间传输。

STATUS

所有命令完毕后将进入此阶段，此时允许 target 设备向 initiator 设备发送状态信息以指示操作成功与否。

MESSAGE IN, MESSAGE OUT

此阶段附加信息将在 initiator 设备和 target 设备间传输。

Linux SCSI 子系统由两个基本部分组成，每个由一个数据结构来表示。

host

一个 SCSI host 即一个硬件设备：SCSI 控制权。NCR 810 PCI SCSI 控制权即一种 SCSI host。在 Linux 系统中可以存在相同类型的多个 SCSI 控制权，每个由一个单独的 SCSI host 来表示。这意味着一个 SCSI 设备驱动可以控制多个控制权实例。SCSI host 总是 SCSI 命令的 initiator 设备。

Device

虽然 SCSI 支持多种类型设备如磁带机、CD-ROM 等等，但最常见的 SCSI 设备是 SCSI 磁盘。SCSI 设备总是 SCSI 命令的 target。这些设备必须区别对待，例如象 CD-ROM 或者磁带机这种可移动设备，Linux 必须检测介质是否已经移动。不同的磁盘类型有不同的主设备号，这样 Linux 可以将块设备请求发送到正确的 SCSI 设备。

初始化 SCSI 子系统

SCSI 子系统的初始化非常复杂，它必须反映处 SCSI 总线及其设备的动态性。Linux 在启动时初始化 SCSI 子系统。如果它找到一个 SCSI 控制器 (即 SCSI hosts) 则会扫描此 SCSI 总线来找出总线上的所有设备。然后初始化这些设备并通过普通文件和 buffer cache 块设备操作使 Linux 核心的其它部分能使用这些设备。初始化过程分成四个阶段：

首先 Linux 将找出在系统核心连接时被连入核心的哪种类型的 SCSI 主机适配器或控制器有硬件需要控制。每个核心中的 SCSI host 在 builtin_scsi_hosts 数组中有一个 Scsi_Host_Template 入口。而 Scsi_Host_Template 结构中包含执行特定 SCSI host 操作，如检测连到此 SCSI host 的 SCSI 设备的例程的入口指针。这些例程在 SCSI 子系统进行自我配置时使用同时它们还是支持此 host 类型的 SCSI 设备驱动的一部分。每个被检测的 SCSI host，即与真正 SCSI 设备连接的 host 将其自身的 Scsi_Host_Template 结构添加到活动 SCSI hosts 的 scsi_hosts 结构链表中去。每个被检测 host 类型的实例用一个 scsi_hostlist 链表中的 Scsi_Host 结构来表示。例如一个包含两个 NCR810 PCI SCSI 控制器的系统的链表中将有两个 Scsi_Host 入口，每个控制器对应一个。每个 Scsi_Host 指向一个代表器设备驱动的 Scsi_Host_Template。

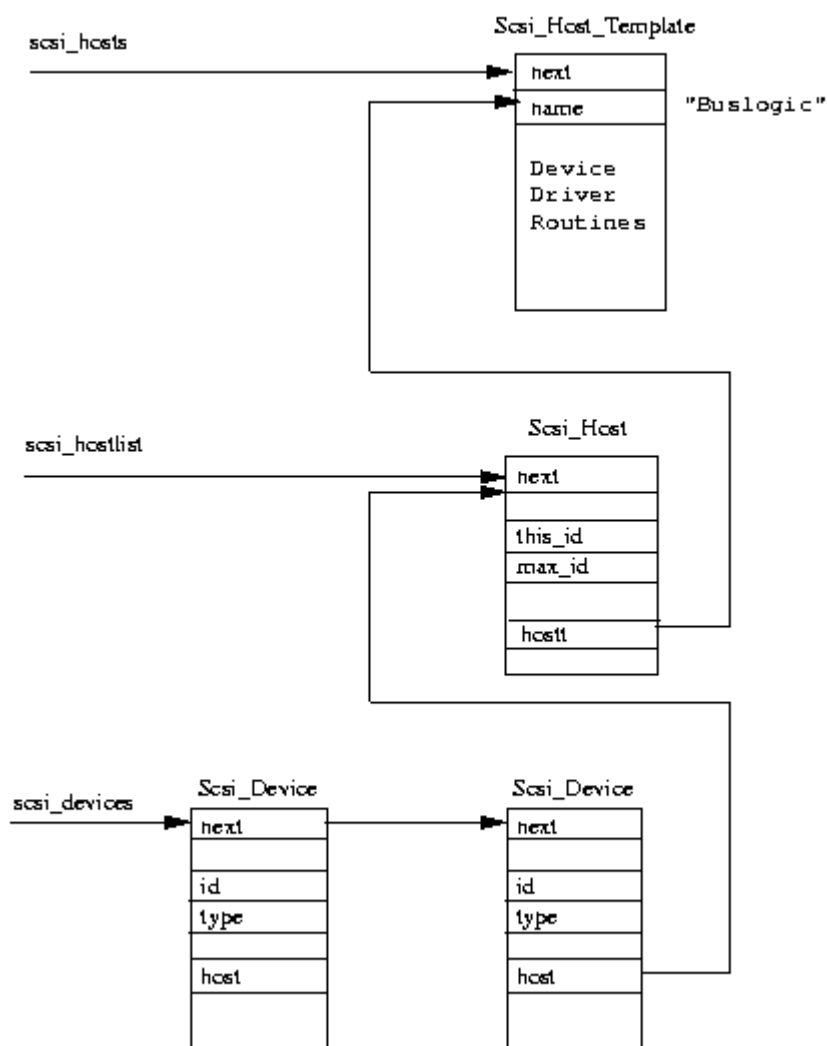


图 8.4 SCSI 数据结构

现在每个 SCSI host 已经找到，SCSI 子系统必须找出哪些 SCSI 设备连接哪个 host 的总线。SCSI 设备的编号是 从 0 到 7，对于一条 SCSI 总线上连接的各个设备，其设备编号或 SCSI 标志符是唯一的。SCSI 标志符可以通过设备上的跳线来设置。SCSI 初始化代码通过在 SCSI 总线上发送一个 TEST_UNIT_READY 命令来找出每个 SCSI 设备。当设备作出相应时其标志符通过一个 ENQUIRY 命令来读取。Linux 将从中得到生产厂商的名称和设备模式以及修订版本号。SCSI 命令由一个 Scsi_Cmdnd 结构来表示同时这些命令通过调用 Scsi_Host_Template 结构中的设备驱动例程传递到此 SCSI host 的设备驱动中。被找到的每个 SCSI 设备用一个 Scsi_Device 结构来表示，每个指向其父 Scsi_Host 结构。所有这些 Scsi_Device 结构被添加到 scsi_device 链表中。图 8.4 给出了这些主要数据结构间的关系。一共有四种 SCSI 设备类型：磁盘，磁带机，CD-ROM 和普通 SCSI 设备。每种类型的 SCSI 设备以不同的主块设备类型单独登记到核心中。如果有多个类型的 SCSI 设备存在则它们只登记自身。每个 SCSI 设备类型，如 SCSI 磁盘维护着其自身的设备列表。它使用这些表将核心块操作（file 或者 buffer cache）定向到正确的设备驱动或 SCSI host 上。每种 SCSI 设备类型用一个 Scsi_Device_Template 结构来表示。此结构中包含此类型 SCSI 设备的信息以及执行各种任务的例程的入口地址。换句话说，如果 SCSI 子系统希望连接一个 SCSI 磁

盘设备它将调用 SCSI 磁盘类型连接例程。如果有多个该种类型的 SCSI 设备被检测到则此 Scsi_Type_Template 结构将被添加到 scsi_devicelist 链表中。

SCSI 子系统的最后一个阶段是为每个已登记的 Scsi_Device_Template 结构调用 finish 函数。对于 SCSI 磁盘类型 设备它将驱动所有 SCSI 磁盘并记录其磁盘布局。同时还将添加一个表示所有连接在一起的 SCSI 磁盘的 gendisk 结构，如图 8.3。

发送块设备请求

一旦 SCSI 子系统初始化完成这些 SCSI 设备就可以使用了。每个活动的 SCSI 设备类型将其自身登记到核心以便 Linux 正确定向块设备请求。这些请求可以是通过 blk_dev 的 buffer cache 请求也可以是通过 blkdevs 的文件 操作。以一个包含多个 EXT2 文件系统分区的 SCSI 磁盘驱动器为例，当安装其中一个 EXT2 分区时系统是怎样将 核心缓冲请求定向到正确的 SCSI 磁盘的呢？

每个对 SCSI 磁盘分区的块读写请求将导致一个新的 request 结构被添加到对应此 SCSI 磁盘的 blk_dev 数组中的 current_request 链表中。如果此 request 正在被处理则 buffer cache 无需作任何工作；否则它必须通知 SCSI 磁盘子系统去处理它的请求队列。系统中每个 SCSI 磁盘用一个 Scsi_Disk 结构来表示。例如/dev/sdb1 的主设备 号为 8 而从设备号为 17；这样产生一个索引值 1。每个 Scsi_Disk 结构包含一个指向表示此设备的 Scsi_Device 结构。这样反过来又指向拥有它的 Scsi_Host 结果。这个来自 buffer cache 的 request 结构将被转换成一个描 叙 SCSI 命令的 Scsi_Cmd 结构，这个 SCSI 命令将发送到此 SCSI 设备同时被排入表示此设备的 Scsi_Host 结构。一旦有适当的数据块需要读写，这些请求将被独立的 SCSI 设备驱动来处理。

8.6 网络设备

网络设备，即 Linux 的网络子系统，是一个发送与接收数据包的实体。它一般是一个象以太网卡的物理设备。有些网络设备如 loopback 设备仅仅是一个用来向自身发送数据的软件。每个网络设备都用一个 device 结构来 表示。网络设备驱动在核心启动初始化网络时将 这些受控设备登记到 Linux 中。device 数据结构中包含有有关设备的信息以及用来支持各种网络协议的函数地址指针。这些函数主要用来使用网络设备传输数据。设备使用标准网络支持机制来将接收到的数据传递到适当的协议层。所有传输与接收到的网络数据用一个 sk_buff 结构来表示，这些灵活的数据结构使得网络协议头可以更容易的添加与删除。网络协议层如何使用网络设备以及 如何使用 sk_buff 来交换数据将在网络一章中详细描叙。本章只讨论 device 数据结构及如何发现与初始化网络。

device 数据结构包含以下有关网络设备的信息：

Name

与使用 mknod 命令创建的块设备特殊文件与字符设备特殊文件不同，网络设备特殊文件仅在于系统网络设备发现与初始化时建立。它们使用标准的命名方法，每个名字代表一种类型的设备。多个相同类型设备将从 0 开始记数。这样以太网设备被命名为/dev/eth0，/dev/eth1,/dev/eth2 等等。一些常见的网络设备如下：

/dev/ethN 以太网设备

/dev/slN SLIP 设备

/dev/pppN PPP 设备

/dev/lo Loopback 设备

Bus Information

这些信息被设备驱动用来控制设备。irq 号表示设备使用的中断号。base address 指任何设备

在 I/O 内存中的控制与状态寄存器地址。DMA 通道指此网络设备使用的 DMA 通道号。所有这些设备在初始化时设置。

Interface Flags

它们描述了网络设备的属性与功能：

IFF_UP	接口已经建立并运行
IFF_BROADCAST	设备中的广播地址有效
IFF_DEBUG	设备调试被使能
IFF_LOOPBACK	这是一个 loopback 设备
IFF_POINTTOPOINT	这是点到点连接 (SLIP 和 PPP)
IFF_NOTRAILERS	无网络追踪者
IFF_RUNNING	资源已被分配
IFF_NOARP	不支持 ARP 协议
IFF_PROMISC	设备处于混乱的接收模式，无论包地址怎样它都将接收
IFF_ALLMULTI	接收所有的 IP 多播帧
IFF_MULTICAST	可以接收 IP 多播帧

Protocol Information

每个设备描述它可以被网络协议层如何使用：

mtu

指不包括任何链路层头在内的，网络可传送的最大包大小。这个值被协议层用来选择适当大小的包进行发送。

Family

这个 family 域表示设备支持的协议族。所有 Linux 网络设备的族是 AF_INET，互联网地址族。

Type

这个硬件接口类型描述网络设备连接的介质类型。Linux 网络设备可以支持多种不同类型的介质。包括以太网、X.25，令牌环，Slip，PPP 和 Apple LocalTalk。

Addresses

结构中包含大量网络设备相关的地址，包括 IP 地址。

Packet Queue

指网络设备上等待传输的 sk_buff 包队列。

Support Functions

每个设备支持一组标准的例程，它们被协议层作为设备链路层的接口而调用。如传输建立和帧传输例程以及添加标准帧头以及收集统计数据的例程。这些统计数据可以使用 ifconfig 命令来观察。

8.6.1 初始化网络设备

网络设备驱动可以象其它 Linux 设备驱动一样建立到 Linux 核心中来。每个潜在的网络设备由一个被 dev_base 链表指针指向的网络设备链表内部的 device 结构表示。当网络层需要某个特定工作执行时。它将调用大量网络服务例程中的一个，这些例程的地址被保存在 device 结构内部。初始化时每个 device 结构仅包含一个初始化或者检测例程的地址。

对于网络设备驱动有两个问题需要解决。首先是不是每个连接到核心中的网络设备驱动

都有设备要控制。其次虽然底层的设备驱动迥然不同，但系统中的以太网设备总是命名为 /dev/eth0 和 /dev/eth1。混淆网络设备这个问题很容易解决。当每个网络设备的初始化例程被调用时，将得到一个指示是否存在当前控制器实例的状态信息。如果驱动找不到任何设备，它那个由 dev_base 指向的 device 链表将被删除。如果驱动找到了设备 则它将用设备相关信息以及网络设备驱动中支撑函数的地址指针来填充此 device 数据结构。

第二个问题，即为以太网设备动态分配标准名称 /dev/ethN 设备特殊文件的工作的解决方法十分巧妙。在设备链表中有 8 个标准入口；从 eth0 到 eth7。它们使用相同的初始化例程，此初始化过程将依次尝试这些被建立到 核心中的以太网设备驱动直到找到一个设备。当驱动找到其以太网设备时它将填充对应的 ethN 设备结构。同时 此网络设备驱动初始化其控制的物理硬件并找出使用的 IRQ 号以及 DMA 通道等信息。如果驱动找到了此网络设备的多个实例它将建立多个 /dev/ethN device 数据结构。一旦所有 8 个标准 /dev/ethN 被分配完毕则不会在检测 其它的以太网设备。

linux 设备驱动读书笔记

机制:提供什么能力

策略:如何使用这些能力

在编写驱动时，程序员应当编写内核代码来存取硬件，但是不能强加特别的策略给用户，因为不同的用户有不同的需求。驱动应当做到使硬件可用，将所有关于如何使用硬件的事情留给应用程序

编写驱动需要注意的地方:

必须注意并发/重入的问题

内核空间和用户空间不能直接操作,必须通过特别的函数(copy_from_user/copy_to_user)来操作。内核线程只有一个非常小的堆栈;它可能小到一个 4096 字节的页。**驱动模块的函数必须与内核函数共享这个堆栈**。因此，声明一个巨大的自动变量不是一个好主意;如果需要大的结构，应当在调用时动态分配。

以双下划线(__)开始的函数通常是一个低层的接口组件，应当小心使用。本质上讲，双下划线告诉程序员:" 如果你调用这个函数，确信你知道你在做什么."

内核代码不能做浮点算术

每个进程的系统栈空间分配的大小为 2 个连续的物理页面(通常来讲是 8K),而 task_struct 占了大约 1K(在栈的底部)，所以系统空间非常有限,**在中断/软中断/驱动程序中不允许嵌套太深或使用大量局部变量**

编写缺少进程上下文的函数需要注意:

不允许存取用户空间。因为没有进程上下文，没有和任何特定进程相关联的到用户空间的途径。

current 指针在原子态没有意义，并且不能使用因为相关的代码没有和已被中断的进程的联系。

不能进行睡眠或者调度。原子代码不能调用 schedule 或者某种 wait_event，也不能调用任何其他可能睡眠的函数。例如，调用 kmalloc(..., GFP_KERNEL) 是违犯规则的。旗标也必须不能使用因为它们可能睡眠。

重要的数据结构的重要成员

struct task_struct {(得到当前进程 task_struct 结构指针的宏为: current)

volatile long state: 进程状态. -1 unrunnable; 0 runnable; >0 stopped

mm_segment_t addr_limit: 线程地址空间: 0-0xBFFFFFFF for user-thead; 0-0xFFFFFFFF for

kernel-thread

struct mm_struct *mm: 虚存管理与映射相关信息,是整个用户空间的抽象

unsigned long sleep_time:

pid_t pid: 进程 pid

uid_t uid,euid,suid,fsuid:

gid_t gid,egid,sgid,fsgid:

gid_t groups[NGROUPS]:

kernel_cap_t cap_effective, cap_inheritable, cap_permitted:

int keep_capabilities:1:

struct user_struct *user:

char comm[16]: 命令名称. 由当前进程执行的程序文件的基本名称(截短到 15 个字符, 如果需要)

struct tty_struct *tty:

unsigned int locks:

struct rlimit rlim[RLIM_NLIMITS]: 当前进程各种资源分配的限制, 如 current->rlim[RLIMIT_STACK]是对用户空间堆栈大小的限制

struct files_struct *files: 打开的文件

};

struct **file_operations**{

struct module *owner;是一个指向拥有这个结构的模块的指针. 这个成员用来当模块在被使用时阻止其被卸载. 一般初始化为: THIS_MODULE

loff_t (*llseek) (struct file *, loff_t, int);用作改变文件中的当前读/写位置, 并且新位置作为(正的)返回值.

ssize_t (*read) (struct file *, char *, size_t, loff_t *);从设备中获取数据. 空指针导致 read 系统调用返回-EINVAL("Invalid argument"). 非负返回值代表了成功读取的字节数

ssize_t (*write) (struct file *, const char *, size_t, loff_t *);发送数据给设备. 空指针导致 write 系统调用返回-EINVAL. 非负返回值代表成功写的字节数.

unsigned int (*poll) (struct file *, struct poll_table_struct *);3 个系统调用的后端: poll, epoll, 和 select. 都用作查询对一个或多个文件描述符的读或写是否会阻塞. poll 方法应当返回一个位掩码指示是否非阻塞的读或写是可能的. 如果一个驱动的 poll 方法为 NULL, 设备假定为不阻塞地可读可写.

int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);提供了发出设备特定命令的方法. 注意:有几个 ioctl 命令被内核识别而不会调用此方法.

int (*mmap) (struct file *, struct vm_area_struct *);请求将设备内存映射到进程的地址空间. 如果这个方法是 NULL,系统调用返回 -ENODEV.

int (*open) (struct inode *, struct file *);open 一个设备文件. 如果这个项是 NULL, 设备打开一直成功

int (*release) (struct inode *, struct file *);在文件结构被释放时引用这个操作. 即在最后一个打开设备文件的文件描述符关闭时调用(而不是每次 close 时都调用)

int (*fsync) (struct file *, struct dentry *, int datasync);fsync 系统调用的后端, 用户调用来刷新任何挂着的数据. 如果这个指针是 NULL, 系统调用返回 -EINVAL.

int (*fasync) (int, struct file *, int);通知设备它的 FASYNC 标志(异步通知)的改变. 这个成员可以是 NULL 如果驱动不支持异步通知.

ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);包含多个内存区的单

个读操作; 如果为 NULL, read 方法被调用(可能多于一次).

ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *); 包含多个内存区的单个写操作; 如果为 NULL, write 方法被调用(可能多于一次).

```
}
```

```
struct file{
```

struct dentry *f_dentry; 关联到文件的目录入口(dentry)结构. 设备驱动不需要关心, 除了作为 filp->f_dentry->d_inode 存取 inode 结构.

struct file_operations *f_op; 和文件关联的操作. 可改变之, 并在返回后新方法会起作用. 例如, 关联到主编号 1 (/dev/null, /dev/zero...) 的 open 根据打开的次编号来更新 filp->f_op

unsigned int f_flags; 文件标志, 例如 O_RDONLY, O_NONBLOCK, 和 O_SYNC. 驱动应当检查 O_NONBLOCK 标志来看是否是请求非阻塞操作

mode_t f_mode; 文件模式确定文件是可读的或者是可写的(或者都是), 通过位 FMODE_READ 和 FMODE_WRITE. 检查是有内核做的, 所以驱动里不需要再次检查

loff_t f_pos; 当前读写位置. 驱动可以读这个值, 但是正常地不应该改变它; 读和写应当使用它们的最后一个参数来更新一个位置. 一个例外是在 llseek 方法中, 它的目的就是改变文件位置.

```
struct fown_struct f_owner;
```

```
unsigned int f_uid, f_gid;
```

```
void *private_data; 可自由使用或者忽略它.
```

```
}
```

```
struct inode{
```

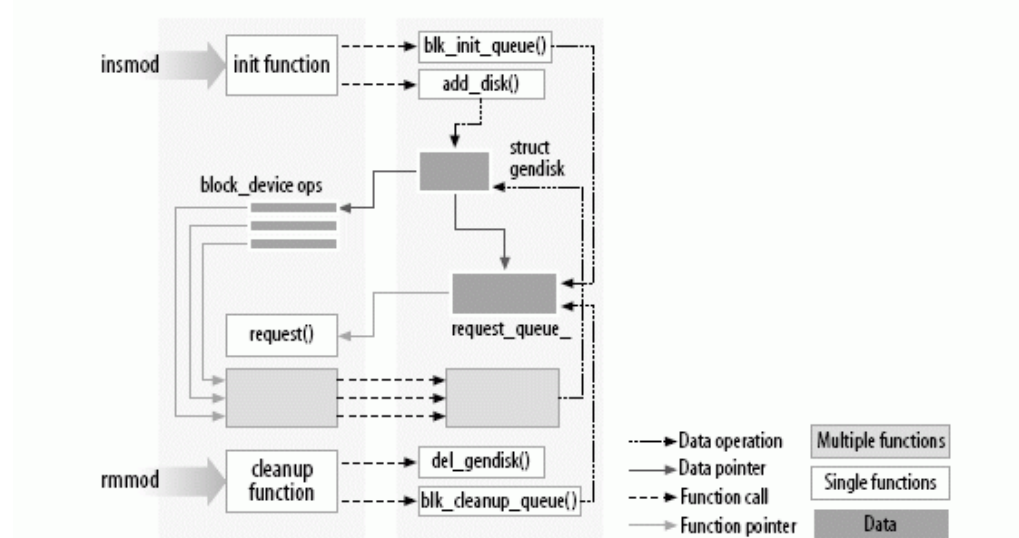
kdev_t i_rdev; 对于代表设备文件的节点, 这个成员包含实际的设备编号. 需要这两个函数来操作: unsigned int iminor(struct inode *inode)/unsigned int imajor(struct inode *inode)

struct char_device *i_cdev; 内核的内部结构, 代表字符设备. 当节点是一个字符设备文件时, 这个成员包含一个指针, 指向这个结构

```
}
```

建立和运行模块

下图展示了函数调用和函数指针在模块中如何使用来增加新功能到一个运行中的内核.



编译和加载

内核版本的问题

linux/version.h 中有下面的宏定义:

UTS_RELEASE

这个宏定义扩展成字符串, 描述了这个内核树的版本. 例如, "2.6.10".

LINUX_VERSION_CODE

这个宏定义扩展成内核版本的二进制形式, 版本号发行号的每个部分用一个字节表示. 例如, 2.6.10 的编码是 132618 (就是, 0x02060a). [4]有了这个信息, 你可以(几乎是)容易地决定你在处理的内核版本.

KERNEL_VERSION(major,minor,release)

这个宏定义用来建立一个整型版本编码, 从组成一个版本号的单个数字. 例如, KERNEL_VERSION(2.6.10) 扩展成 132618. 这个宏定义非常有用, 当你需要比较当前版本和一个已知的检查点.

特殊 GNU make 变量名

obj-m := module.o #最终模块名

module-objs := file1.o file2.o #最终模块用到的 obj 列表

make 命令中的 "M=" 选项使 makefile 在试图建立模块目标(obj-m 变量中指定的)前, 回到你的模块源码目录

Makefile 示例:

```
# If KERNELRELEASE is defined, we've been invoked from the
```

```
# kernel build system and can use its language.
```

```
ifneq ($(KERNELRELEASE),)
```

```
    obj-m := hello.o
```

```
# Otherwise we were called directly from the command
```

```
# line; invoke the kernel build system.
```

```
else
```

```
    KERNELDIR ?= /lib/modules/$(shell uname -r)/build
```

```
    PWD := $(shell pwd)
```

```
default:
```

```
$(MAKE) -C $(KERNELDIR) M=$(PWD) modules
```

```
endif
```

加载卸载:

insmod:仅加载指定的模块)

modprobe:加载指定的模块及其相关模块. 它查看要加载的模块, 看是否它引用了当前内核没有定义的符号. 如果未定义的 symbols, modprobe 在模块搜索路径(/etc/modprobe.conf)中寻找并加载其他定义了 symbols 的模块

rmmod: 从内核中去除指定模块

lsmod: 打印内核中当前加载的模块的列表. 通过读取/proc/modules 或/sys/module 的 sysfs 虚拟文件系统工作

错误信息:

unresolved symbols: 加载是找不到 symbols. 可能是使用了未定义的 symbols; 也可能是需要使用 modprobe 试一下

-1 Invalid module format:编译模块用的内核源代码版本与当前运行的内核的版本不匹配

调试技术

内核的 config 配置

kernel hacking 菜单

CONFIG_DEBUG_KERNEL:

这个选项只是使其他调试选项可用；它应当打开，但是它自己不激活任何的特性。

CONFIG_DEBUG_SLAB:

这个重要的选项打开了内核内存分配函数的几类检查。激活这些检查，就可能探测到一些内存覆盖和遗漏初始化的错误。被分配的每一个字节在递交给调用者之前都设成 0xa5，随后在释放时被设成 0x6b。内核还会在每个分配的内存对象的前后放置特别的守护值；如果这些值曾被改动，内核知道有人已覆盖了一个内存分配区。

CONFIG_DEBUG_PAGEALLOC:

满的页在释放时被从内核地址空间去除(full pages are removed from the kernel address space when freed)(?)。这个选项会显著拖慢系统，但是它也能快速指出某些类型的内存损坏错误。

CONFIG_DEBUG_SPINLOCK

激活这个选项，内核捕捉对未初始化的自旋锁的操作，以及各种其他的错误(例如 2 次解锁同一个锁)。

CONFIG_DEBUG_SPINLOCK_SLEEP

这个选项激活对持有自旋锁时进入睡眠的检查。实际上，如果你调用一个可能会睡眠的函数，它就发出警告，即便这个有疑问的调用没有睡眠。

CONFIG_INIT_DEBUG

用 __init (或者 __initdata) 标志的项在系统初始化或者模块加载后都被丢弃。这个选项激活了对代码的检查，这些代码试图在初始化完成后存取初始化时内存。

CONFIG_DEBUG_INFO

这个选项使得内核在建立时包含完整的调试信息。如果你想使用 gdb 调试内核，你将需要这些信息。如果你打算使用 gdb，你还要激活 CONFIG_FRAME_POINTER。

CONFIG_MAGIC_SYSRQ

激活"魔术 SysRq"键。

CONFIG_DEBUG_STACKOVERFLOW

CONFIG_DEBUG_STACK_USAGE

这些选项能帮助跟踪内核堆栈溢出。堆栈溢出的确证是一个 oops 输出，但是没有任何形式的合理的回溯。第 1 个选项给内核增加了明确的溢出检查；第 2 个使得内核监测堆栈使用并作一些统计，这些统计可以用魔术 SysRq 键得到。

CONFIG_KALLSYMS

这个选项(在"General setup/Standard features"下)使得内核符号信息建在内核中；缺省是激活的。符号选项用在调试上下文中；没有它，一个 oops 列表只能以 16 进制格式给你一个内核回溯，这不是很有用。

CONFIG_IKCONFIG

CONFIG_IKCONFIG_PROC

这些选项(在"General setup"菜单)使得完整的内核配置状态被建立到内核中，可以通过 /proc 来使其可用。大部分内核开发者知道他们使用的哪个配置，并不需要这些选项(会使得内核更大)。但是如果你试着调试由其他人建立的内核中的问题，它们可能有用。

Power management/ACPI 菜单

CONFIG_ACPI_DEBUG

这个选项打开详细的 ACPI (Advanced Configuration and Power Interface) 调试信息，如果你怀疑一个问题和 ACPI 相关可能会用到

Device drivers 菜单

CONFIG_DEBUG_DRIVER

打开了驱动核心的调试信息, 可用以追踪底层支持代码的问题.

Device drivers/SCSI device support 菜单

CONFIG_SCSI_CONSTANTS

建立详细的 SCSI 错误消息的信息. 如果你在使用 SCSI 驱动, 你可能需要这个选项.

Device drivers/Input device support 菜单

CONFIG_INPUT_EVBUG

如果你使用一个输入设备的驱动, 这个选项可能会有用. 然而要小心这个选项的安全性的隐含意义: 它记录了你键入的任何东西, 包括你的密码.

Profiling support 菜单

CONFIG_PROFILING

剖析通常用在系统性能调整, 但是在追踪一些内核挂起和相关问题上也有用.

strace 命令

显示所有的用户空间程序发出的系统调用. 并以符号形式显示调用的参数和返回值. 当一个系统调用失败, 错误的符号值(例如, ENOMEM)和对应的字串(Out of memory) 都会显示.

-t: 来显示每个系统调用执行的时间

-T: 来显示调用中花费的时间

-e: 来限制被跟踪调用的类型

-o: 来重定向输出到一个文件. 缺省地, strace 打印调用信息到 stderr.

GDB 调试

调试内核:

`gdb /usr/src/linux/vmlinux /proc/kcore`

第一个参数是非压缩的 ELF 内核可执行文件的名子, 不是 zImage 或者 bzImage

第二个参数是核心文件的名子.

注意事项:

无法检查 module 的相关内容

不能修改内核数据, 不能单步, 不能设置断点

读到的是内核即时映象, 内核仍在运行, 所以有些数据可能会与即时值不匹配 --刷新映象: `core-file /proc/kcore`

调试模块(内核版本 2.6.7 以上)

Linux 可加载模块是 ELF 格式的可执行映象; ELF 被分成几个 sections. 其中有 3 个典型的 sections 与调试会话相关:

.text

这个节包含有模块的可执行代码. 调试器必须知道在哪里以便能够给出回溯或者设置断点.

.bss

在编译时不初始化的任何变量在 .bss 中

.data

在编译时需要初始化的任何变量在 .data 里.

为了 gdb 能够调试可加载模块需要通知调试器一个给定模块的各个 sections 加载在哪里. 这个信息在 `/sys/module/module_name/sections` 下. 包含名字为 .text, .bss, .data 等文件; 每个文件的内容是那个 section 的基地址.

gdb 的 add-symbol-file 命令用来加载模块相关信息

`add-symbol-file 模块名 text 所在的基地址 -s .bss bss 所在基地址`

址 -s .data data 所在基地址
 add-symbol-file ../scull.ko 0xd0832000 -s .bss 0xd0837100 -s .data 0xd0836be0

KDB 调试

KDB 是来自 oss.sgi.com 的一个非官方补丁. 应用 KDB 时不应该运行任何程序, 特别的, 不能打开网络. 一般地以单用户模式启动系统

进入 KDB:

Pause(或者 Break) 键启动调试器

一个内核 oops(异常?) 发生时

命中一个断点时

命令:

bp function_name

在下次内核进入 function_name 时停止

bt

打印出调用回溯中每个函数的参数

mds variable_name

mds address

查看变量/内存数据

mm address value

将 value 赋给 address 所指向的内存

.....

内核中的数据类型

不同体系结构下各个类型的大小

arch	Size:	char	short	int	long	ptr	long-long	u8	u16	u32	u64
i386		1	2	4	4	4	8	1	2	4	8
alpha		1	2	4	8	8	8	1	2	4	8
armv4l		1	2	4	4	4	8	1	2	4	8
ia64		1	2	4	8	8	8	1	2	4	8
m68k		1	2	4	4	4	8	1	2	4	8
mips		1	2	4	4	4	8	1	2	4	8
ppc		1	2	4	4	4	8	1	2	4	8
sparc		1	2	4	4	4	8	1	2	4	8
sparc64		1	2	4	4	4	8	1	2	4	8
x86_64		1	2	4	8	8	8	1	2	4	8

应当安排有明确类型大小的数据类型, 如 u8, u16, ... uint8_t, uint16_t, ...

接口特定的类型, 请参考原文

其他移植性问题:

Tick: HZ

页大小: PAGE_SIZE

页偏移: PAGE_SHIFT

字节序:

条件编译

```
#include <asm/byteorder.h>
```

```
#ifdef __BIG_ENDIAN
```

```
.....
```

```
#endif
```

```
#ifdef __LITTLE_ENDIAN
```

```
.....
```

```
#endif
```

转换

```
#include <linux/byteorder/big_endian.h>
```

```
#include <linux/byteorder/little_endian.h>
```

```
u32 cpu_to_le32 (u32);
```

```
u32 le32_to_cpu (u32);
```

```
cpu_to_le16/le16_to_cpu/cpu_to_le64/....
```

```
cpus_to_le16/le16_to_cpus/cpus_to_le64/....
```

带's'后缀的是有符号版

数据对齐:

存取不对齐的数据应当使用下列宏

```
#include <asm/unaligned.h>
```

```
get_unaligned(ptr);
```

```
put_unaligned(val, ptr);
```

指针返回值:

有时候内核函数会返回已编码的指针值来指示错误, 这类返回值是否有效的测试应当使用下列宏

void *ERR_PTR(long error); 将错误码转换成指针形式

long IS_ERR(const void *ptr); 判断一个返回值是否有效

long PTR_ERR(const void *ptr); 提取返回的错误码, 在提取前需要判断返回值是否有效

链表:

鼓励使用内核自带的 struct list_head 结构来构造双向链表

```
#include <linux/list.h>
```

```
struct list_head { struct list_head *next, *prev; };
```

```
LIST_HEAD(struct list_head);
```

编译时初始化

```
INIT_LIST_HEAD(struct list_head*)
```

运行时初始化

```
list_add(struct list_head *new, struct list_head *head);
```

在 head 后链入 new. 常用来构造 FILO

```
list_add_tail(struct list_head *new, struct list_head *head);
```

在 head 前链入 new, 常用来构造 FIFO

```
list_del(struct list_head *entry);
```

把 entry 从链表中脱链

```
list_del_init(struct list_head *entry);
```

把 entry 从链表中脱链并重新初始化 entry

```
list_move(struct list_head *entry, struct list_head *head);
```

将 entry 移动到 head 后

`list_move_tail(struct list_head *entry, struct list_head *head);`

将 entry 移动到 head 前

`list_empty(struct list_head *head);`

判断链表是否为空

`list_splice(struct list_head *list, struct list_head *head);`

从链表 head 后断开, 将 list 链表链接进去

`list_entry(struct list_head *ptr, type_of_struct, field_name);`

从 list_head 地址得到包含 list_head 的结构体的开始地址.

类似的宏为 `container_of(pointer, container_type, container_field);` 从结构体成员地址得到结构体指针

`list_for_each(struct list_head *cursor, struct list_head *list)`

这个宏创建一个 for 循环, 执行一次, cursor 指向链表中的下个入口项

`list_for_each_prev(struct list_head *cursor, struct list_head *list)`

这个版本反向遍历链表.

`list_for_each_safe(struct list_head *cursor, struct list_head *next, struct list_head *list)`

如果循环可能删除链表中的项, 使用这个版本.

`list_for_each_entry(type *cursor, struct list_head *list, member)`

直接得到包含 list_head 的结构体的地址

`list_for_each_entry_safe(type *cursor, type *next, struct list_head *list, member)`

如果循环可能删除链表中的项, 使用这个版本

举例:

```
struct list_head todo_list;
```

```
...
```

```
void todo_add_entry(struct todo_struct *new)
```

```
{
    struct list_head *ptr;
    struct todo_struct *entry;

    list_for_each(ptr, &todo_list)
    {
        entry = list_entry(ptr, struct todo_struct, list);
        if (entry->priority < new->priority) {
            list_add_tail(&new->list, ptr);
            return;
        }
    }
    list_add_tail(&new->list, &todo_struct)
}
```

模块特殊宏/函数(注意大小写)

`module_init(initialization_function)`: 声明模块初始化函数

`module_exit(cleanup_function)`: 声明模块注销函数

`EXPORT_SYMBOL(name)`: 声明符号在模块外可用

`EXPORT_SYMBOL_GPL(name)`: 声明符号仅对使用 GPL 许可的模块可用.

`MODULE_LICENSE("GPL")`: 声明模块许可

`MODULE_AUTHOR`: 声明谁编写了模块

`MODULE_DESCRIPTION`: 一个人可读的关于模块做什么的声明

`MODULE_VERSION`: 一个代码修订版本号; 看 `<linux/module.h>` 的注释以便知道创建版本字符串使用的惯例

`MODULE_ALIAS`: 模块为人所知的另一个名子

`MODULE_DEVICE_TABLE`: 来告知用户空间, 模块支持那些设备

`module_param(name, type, perm)`: 声明模块加载时允许设置的参数 (2.6.11 之前版本中为 `MODULE_PARM`)

`module_param_array(name,type,num,perm)`: 声明模块加载时允许设置的数组参数

`name`: 是你的参数(数组)的名子

`type`: 是数组元素的类型

`bool/invbool`: 一个布尔型 (`true` 或者 `false`) 值(相关的变量应当是 `int` 类型). `invbool` 类型颠倒了值, 所以真值变成 `false`, 反之亦然.

`charp`: 一个字符指针值. 需要为其分配内存(`charp`, NOT `char`)

`int/long/short/uint/ulong/ushort`: 基本的变长整型值. 以 `u` 开头的是无符号值.

`num`: 一个整型变量

`perm`: 通常的权限值, 在 `<linux/stat.h>` 中定义. `S_IRUGO`: 可以被所有人读取, 但是不能改变; `S_IRUGO|S_IWUSR`: 允许 `root` 改变参数. 注意, 如果一个参数被 `sysfs` 修改, 模块看到的参数值也改变了, 但模块不会有任何通知

示例:

在模块中声明如下:

```
static char *whom = "world";
```

```
static int howmany = 1;
```

```
module_param(howmany, int, S_IRUGO);
```

```
module_param(whom, charp, S_IRUGO);
```

调用时如下:

```
insmod module_name howmany=10 whom="Mom"
```

模块初始化函数原型为

```
static int __init function(void);
```

大部分注册函数以 `register_` 做前缀

`__init/__initdata`: 给定的函数/数据只是在初始化使用. 模块加载后丢掉这个初始化函数, 使它的内存可做其他用途.

`__devinit/__devinitdata`: 内核没有配置支持 `hotplug` 设备时等同于 `__init/__initdata`.

模块注销函数原型为

```
static void __exit function(void);
```

`__exit/__exitdata`: 如果模块直接建立在内核里, 或者如果内核配置成不允许模块卸载, 标识

为 `__exit` 的函数被简单地丢弃

`container_of(pointer, container_type, container_field)`; 通过一个结构体成员的地址得到结构体的地址

比如:

```
struct test {int a; int b; int c; int e;} test_t;  
&test_t == container_of(&(test_t.c), struct test, c)  
__setup("test=", test_setup);
```

这个宏将 `test_setup` 这个函数放在特定的 section 中.

在执行 `init/main.c::checksetup()` 时会去 `kernel boot commandline` 中寻找字符串 `"test=xxx"`: 如果有找到, 就用 `"xxx"` 作为参数调用 `test_setup`; 否则不运行

在 `insmod` 中如果参数里带有 `"test=xxx"` 也会运行

`void *kmalloc(size_t size, int flags)`; 试图分配 `size` 字节的内存; 返回值是指向那个内存的指针或者如果分配失败为 `NULL`. `flags` 参数用来描述内存应当如何分配

申请的空间大小限制: 大概为 128K

`void kfree(void *ptr)`; 分配的内存应当用 `kfree` 来释放. 传递一个 `NULL` 指针给 `kfree` 是合法的.

`get_free_page` 申请的 `page` 数限制: $2^{\text{MAX_ORDER}}$, 2 的 `MAX_ORDER` 次方个 `page`. 通常 `MAX_ORDER=10`, 也就是最多 $2^{10}=1024$ 个 `page`, 4Mbyte

`int access_ok(int type, const void *addr, unsigned long size)`: 验证用户空间有效性

`type`: `VERIFY_READ/VERIFY_WRITE`. 如果需要验证读写许可, 则只要 `VERIFY_WRITE`

`addr`: 一个用户空间地址,

`size`: 需要验证的大小.

返回值: 1 是成功(存取没问题); 0 是失败(存取有问题). 如果它返回 0, 驱动应当返回 `-EFAULT`

`put_user(datum, ptr)`

`__put_user(datum, ptr)`

写 `datum` 到用户空间. 它们相对(`copy_to_user`)快. 传送的数据大小依赖 `ptr` 参数的类型.

比如: `ptr` 是一个 `char` 指针就传送一个字节

`put_user` 检查用户空间确保能写. 在成功时返回 0, 并且在错误时返回 `-EFAULT`.

`__put_user` 进行更少的检查(它不调用 `access_ok`),

驱动应当调用 `put_user` 来节省几个周期; 或者拷贝几个项时, 在第一次数据传送之前调用 `access_ok` 一次, 之后使用 `__put_user`

`get_user(local, ptr)`

`__get_user(local, ptr)`

从用户空间读单个数据, 获取的值存储于本地变量 `local`;

如果使用上述四个函数时, 发现一个来自编译器的奇怪消息, 例如 `"conversion to non-scalar`

type requested". 必须使用 `copy_to_user` 或者 `copy_from_user`.

`unsigned long copy_to_user(void __user *to, const void *from, unsigned long count);` 拷贝一整段数据到用户地址空间. 任何存取用户空间的函数必须是可重入的. 此函数可能导致睡眠
`unsigned long copy_from_user(void *to, const void __user *from, unsigned long count);` 从用户地址空间拷贝一整段数据. 任何存取用户空间的函数必须是可重入的. 此函数可能导致睡眠
这两个函数的作用不仅限于拷贝数据到和从用户空间: 它们还检查用户空间指针是否有效. 如果指针无效, 不进行拷贝; 如果在拷贝中遇到一个无效地址, 只拷贝有效部分的数据. 在第二种情况下, 返回值是未拷贝的数据数. 驱动应当查看返回值, 并且如果它不是 0, 就返回 `-EFAULT` 给用户.

`int capable(int capability);`

在进行一个特权操作之前, 一个设备驱动应当检查调用进程有合适的能力

`capability` 取值有以下这些:

`CAP_DAC_OVERRIDE`

这个能力来推翻在文件和目录上的存取的限制(数据存取控制, 或者 DAC).

`CAP_NET_ADMIN`

进行网络管理任务的能力, 包括那些能够影响网络接口的.

`CAP_SYS_MODULE`

加载或去除内核模块的能力.

`CAP_SYS_RAWIO`

进行 "raw" I/O 操作的能力. 例子包括存取设备端口或者直接和 USB 设备通讯.

`CAP_SYS_ADMIN`

一个捕获-全部的能力, 提供对许多系统管理操作的存取.

`CAP_SYS_TTY_CONFIG`

进行 tty 配置任务的能力.

`CAP_SYS_ADMIN`

在任务缺乏一个更加特定的能力时, 可以选这个来测试

`int printk(const char *fmt, ...);` 向 console(而不是虚拟终端)打印一条消息, 并通过附加不同的记录级别或者优先级在消息上对消息的严重程度分类. 没有指定优先级的 `printk` 语句缺省是 `DEFAULT_MESSAGE_LOGLEVEL`, 在 `kernel/printk.c` 里指定作为一个整数. 在 2.6.10 内核中, `DEFAULT_MESSAGE_LOGLEVEL` 是 `KERN_WARNING`, 但这个值在不同的内核中可能不一样.

按消息的严重程度从高到低为:

`KERN_EMERG`

用于紧急消息, 常常是那些崩溃前的消息.

`KERN_ALERT`

需要立刻动作的情形.

`KERN_CRIT`

严重情况, 常常与严重的硬件或者软件失效有关.

`KERN_ERR`

用来报告错误情况; 设备驱动常常使用 `KERN_ERR` 来报告硬件故障.

KERN_WARNING

有问题的情况的警告, 这些情况自己不会引起系统的严重问题.

KERN_NOTICE

正常情况, 但是仍然值得注意. 在这个级别一些安全相关的情况会报告.

KERN_INFO

信息型消息. 在这个级别, 很多驱动在启动时打印它们发现的硬件的信息.

KERN_DEBUG

用作调试消息.

使用举例:

```
printk(KERN_INFO "hello, world\n");
```

//注意:消息优先级与正文内容之间没有逗号

`int printk_ratelimit(void);` 在你认为打印一个可能会常常重复的消息之前调用来避免重复输出很多相同的调试信息. 如果这个函数返回非零值, 继续打印你的消息, 否则跳过打印.

使用举例

```
if (printk_ratelimit())
    printk(KERN_NOTICE "The printer is still on fire\n");
```

```
int print_dev_t(char *buffer, dev_t dev);
```

```
char *format_dev_t(char *buffer, dev_t dev);
```

从一个驱动打印消息, 你会想打印与感兴趣的硬件相关联的设备号. 两个宏定义都将设备号编码进给定的缓冲区; 唯一的区别是 `print_dev_t` 返回打印的字符数, 而 `format_dev_t` 返回缓存区

```
void set_current_state(int new_state);
```

设置当前进程的运行状态

`new_state:`

`TASK_INTERRUPTIBLE/TASK_RUNNING/TASK_INTERRUPTIBLE/TASK_UNINTERRUPTIBLE/...`

在新代码中不鼓励使用下面这种方式

```
current->state = TASK_INTERRUPTIBLE
```

```
int in_interrupt(void)
```

如果处理器当前在中断上下文(包括软中断和硬中断)运行就返回非零

```
int in_atomic(void)
```

若调度被禁止(即当前状态是原子态, 包括硬中断, 软件中断以及持有自旋锁时), 返回值是非零. 在持有自旋锁这种情况, `current` 可能是有效的, 但是禁止存取用户空间, 因为它能导致调度发生.

无论何时使用 `in_interrupt()`, 应当真正考虑是否 `in_atomic` 是你实际想要的

主次设备号:

主编号标识设备驱动; 次编号被内核用来决定引用哪个设备

dev_t 类型(在 <linux/types.h>中定义)用来标识设备编号 -- 同时包括主次部分

MAJOR(dev_t dev): 从 dev_t 中取得主设备号

MINOR(dev_t dev): 从 dev_t 中取得次设备号

MKDEV(int major, int minor): 讲主次设备号转换成 dev_t

int register_chrdev_region(dev_t first, unsigned int count, char *name): 获取一个或多个设备编号来使用

first 是你要分配的起始设备编号. first 的次编号部分常常是 0

count 是你请求的连续设备编号的总数

name 是应当连接到这个编号范围的设备的名字; 它会出现在 /proc/devices 和 sysfs 中

int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char *name);动态分配一个主编号

dev 是一个只输出的参数, 它在函数成功完成时持有你的分配范围的第一个数.

firstminor 应当是请求的第一个要用的次编号; 它常常是 0.

count 是你请求的连续设备编号的总数

name 是应当连接到这个编号范围的设备的名字; 它会出现在 /proc/devices 和 sysfs 中

void unregister_chrdev_region(dev_t first, unsigned int count); 释放设备编号

first 是你要分配的起始设备编号. first 的次编号部分常常是 0

count 是你请求的连续设备编号的总数

设备注册:

struct cdev *cdev_alloc(void);为 struct cdev 申请内存空间

void cdev_init(struct cdev *cdev, struct file_operations *fops);初始化 struct cdev 结构. 其成员 owner 应当设置为 THIS_MODULE

cdev 是需要初始化的 struct cdev 结构

fops: 是关联到这个驱动的方法集合(read/write 等)

int cdev_add(struct cdev *dev, dev_t num, unsigned int count);将设备注册到内核

dev 是 struct cdev 结构

num 是这个设备响应的第一个设备号

count 是应当关联到设备的设备号的数目. 常常 count 是 1, 但是有多个设备号对应于一个特定的设备的情形.

void cdev_del(struct cdev *dev);将设备注销

设备注册的老方法:

int register_chrdev(unsigned int major, const char *name, struct file_operations *fops);

major 是感兴趣的主编号

name 是驱动的名字(出现在 /proc/devices)

fops 是缺省的 file_operations 结构.

int unregister_chrdev(unsigned int major, const char *name);

major 和 name 必须和传递给 register_chrdev 的相同, 否则调用会失败.

设备节点:

devfs_handle_t devfs_register (devfs_handle_t dir,

```

    const char *name,
    unsigned int flags,
    unsigned int major, unsigned int minor,
    umode_t mode,
    void *ops, void *info);创建设备节点

```

dir:需要创建的设备文件所在目录,默认为/dev

name: 需要创建的设备文件名

flags: 通常取 DEVFS_FL_DEFAULT

major: 主设备号

minor: 次设备号

mode: 此设备文件的读写权限

ops: 此设备的 file_operations 结构

info:

```

#define DEV_ID      ((void*)123456)
#define DEV_NAME    "XXXXXXXXXXXXXXXXXX"
#define DEV_MAJOR   200
#define DEV_IRQ     IRQ_XXXX
#define DEV_IRQ_MODE SA_SHIRQ

```

...

```

    //regist char device
    #if LINUX_VERSION_CODE < KERNEL_VERSION(2,6,0)
        ret = register_chrdev(DEV_MAJOR, DEV_NAME, &fops);
    #else
        cdev_init(&dev_char, &fops);
        dev_char.owner = THIS_MODULE;
        dev_char.ops = &fops;
        ret = cdev_add(&dev_char, MKDEV(DEV_MAJOR, 0), 1);
    #endif
    if (ret < 0)
        goto __mod_init_err1;
    //make devfs
    #if LINUX_VERSION_CODE < KERNEL_VERSION(2,6,0)
        devfs_handle = devfs_register(NULL, DEV_NAME, DEVFS_FL_DEFAULT,
                                     DEV_MAJOR, 0, S_IFCHR | S_IRUSR | S_IWUSR, &fops, NULL);
        if (NULL == devfs_handle)
        {
            ret = -1;
            goto __mod_init_err2;
        }
    #else
        dev_class = class_create(THIS_MODULE, DEV_NAME);
        if(IS_ERR(dev_class))

```

```

    {
        ret = PTR_ERR(dev_class);
        goto __mod_init_err2;
    }
    class_device_create(dev_class, MKDEV(DEV_MAJOR, 0), NULL, DEV_NAME);
    ret = devfs_mk_cdev(MKDEV(DEV_MAJOR, 0), S_IFCHR | S_IRUGO | S_IWUSR,
DEV_NAME);
    if(ret)
        goto __mod_init_err3;
#endif

.....

#ifdef LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,0)
__mod_init_err3:
    class_device_destroy(dev_class, MKDEV(DEV_MAJOR, 0));
    class_destroy(dev_class);
#endif
__mod_init_err2:
#ifdef LINUX_VERSION_CODE < KERNEL_VERSION(2,6,0)
    unregister_chrdev(DEV_MAJOR, DEV_NAME);
#else
    cdev_del(&dev_char);
#endif
__mod_init_err1:
    free_irq(DEV_IRQ, DEV_ID);

#endif//end of "ifndef INPUT_DEVICE"

__mod_init_err0:
    return ret;

```

file_operations 函数:

ssize_t read(struct file *filp, char __user *buf, size_t count, loff_t *f_pos)

1. 通常应当更新 *offp 中的文件位置来表示在系统调用成功完成后当前的文件位置.
2. 如果"没有数据, 但是可能后来到达", 在这种情况下, read 系统调用应当阻塞.
3. 返回值

如果等于传递给 read 系统调用的 count 参数, 请求的字节数已经被传送

如果是正数, 但是小于 count, 只有部分数据被传送.

如果值为 0, 到达了文件末尾(没有读取数据).

如果值为负值表示有一个错误. 这个值指出了什么错误, 根据 <linux/errno.h>. 出错的典型返回值包括 -EINTR(被打断的系统调用) 或者 -EFAULT(坏地址).

如果一些数据成功传送接着发生错误, 返回值必须是成功传送的字节数. 在函数下一次调用前错误不会报告. 这要求驱动记住错误已经发生, 以便可以在以后返回错误状态.

ssize_t write(struct file *filp, const char __user *buf, size_t count, loff_t *f_pos)

1. 通常应当更新 *offp 中的文件位置来表示在系统调用成功完成后当前的文件位置.
2. 返回值:

如果值等于 count, 要求的字节数已被传送

如果正值, 但是小于 count, 只有部分数据被传送

如果值为 0, 什么没有写. 这个结果不是一个错误

一个负值表示发生一个错误

如果一些数据成功传送接着发生错误, 返回值必须是成功传送的字节数. 在函数下一次调用前错误不会报告. 这要求驱动记住错误已经发生, 以便可以在以后返回错误状态.

ssize_t (*readv) (struct file *filp, const struct iovec *iov, unsigned long count, loff_t *ppos);

ssize_t (*writev) (struct file *filp, const struct iovec *iov, unsigned long count, loff_t *ppos);

struct iovec

```
{
    void __user *iov_base;
    __kernel_size_t iov_len;
};
```

每个 iovec 描述了一块要传送的数据; 它开始于 iov_base (在用户空间)并且有 iov_len 字节长. count 参数告诉有多少 iovec 结构.

若未定义此二函数. 内核使用 read 和 write 来模拟它们,

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

inode 和 filp 指针是对对应应用程序传递的文件描述符 fd 的值, 和传递给 open 方法的相同参数.

cmd 参数从用户那里不改变地传下来,

arg 是可选的参数, 无论是一个整数还是指针(按照惯例应该用指针), 均以 unsigned long 的形式传递进来

返回值:

-EINVAL("Invalid argument"): 命令参数不是一个有效的 POSIX 标准(通常会返回这个)

-ENOTTY: 一个不合适的 ioctl 命令. 这个错误码被 C 库解释为"设备的不适当的 ioctl"(inappropriate ioctl for device)

ioctl 的 cmd 参数应当是系统唯一的, 这是出于阻止向错误的设备发出其可识别但具体内容无法解析的命令的考虑

cmd 参数由这几部分组成: type, number, direction, size

type

魔数. 为整个驱动选择一个数(参考 ioctl-number.txt). 这个成员是 8 位宽(_IOC_TYPEBITS).

number

序(顺序)号. 它是 8 位(_IOC_NRBITS)宽.

direction

数据传送的方向(如果这个特殊的命令涉及数据传送).

数据传送方向是以应用程序的观点来看待的方向

_IOC_NONE: 没有数据传输

_IOC_READ: 从系统到用户空间

_IOC_WRITE: 从用户空间到系统

_IOC_READ|_IOC_WRITE: 数据在 2 个方向被传送

size

用户数据的大小. 这个成员的宽度(_IOC_SIZEBITS)是依赖体系的. 通常是 13 或者 14 位.

命令号相关操作:

_IO(type,nr): 创建没有参数的命令

_IOR(type, nre, datatype): 创建从驱动中读数据的命令

_IOW(type,nr,datatype): 创建写数据的命令

_IOWR(type,nr,datatype): 创建双向传送的命令

_IOC_TYPE(cmd):得到 magic number

_IOC_NR(cmd):得到顺序号

_IOC_DIR(cmd): 得到传送方向

_IOC_SIZE(cmd): 得到参数大小

预定义命令(会被内核自动识别而不会调用驱动中定义的 ioctl)分为 3 类:

1. 可对任何文件发出的(常规, 设备, FIFO, 或者 socket). 这类的 magic number 以'T'开头
2. 只对常规文件发出的那些.
3. 对文件系统类型特殊的那些.

驱动开发只需注意第一类命令,以及以下这些

FIOCLEX

设置 close-on-exec 标志(File IOctl Close on EXec).

FIONCLEX

清除 close-no-exec 标志(File IOctl Not CLoSe on EXec).

FIOASYNC

设置或者复位异步通知. 注意直到 Linux 2.2.4 版本的内核不正确地使用这个命令来修改 O_SYNC 标志. 因为两个动作都可通过 fcntl 来完成, 没有人真正使用 FIOASYNC 命令, 它在这里出现只是为了完整性.

FIOQSIZE

返回一个文件或者目录的大小; 当用作一个设备文件, 返回一个 ENOTTY 错误.

FIONBIO(File IOctl Non-Blocking I/O)

修改在 `filp->f_flags` 中的 `O_NONBLOCK` 标志. 给这个系统调用的第 3 个参数用作指示是否这个标志被置位或者清除. 注意常用的改变这个标志的方法是使用 `fcntl` 系统调用, 使用 `F_SETFL` 命令.

`unsigned int (*poll) (struct file *filp, poll_table *wait);`

`filp`: 文件描述符指针

`wait`: 用于 `poll_wait` 函数

返回值: 可能不必阻塞就立刻进行的操作

`void poll_wait(struct file *, wait_queue_head_t *, poll_table *);`

驱动通过调用函数 `poll_wait` 增加一个等待队列到 `poll_table` 结构. 这个等待队列是驱动定义并处理的, 进程唤醒方式(只唤醒其中一个还是唤醒所有等待的进程)也由驱动(read/write)来决定

返回的位掩码:

POLLIN

设备可不阻塞地读

POLLRDNORM

可以读"正常"数据. 一个可读的设备返回(`POLLIN|POLLRDNORM`).

POLLRDBAND

可从设备中读取带外数据. 当前只用在 Linux 内核的一个地方(DECnet 代码), 并且通常对设备驱动不可用.

POLLPRI

可不阻塞地读取高优先级数据(带外). 这个位使 `select` 报告在文件上遇到一个异常情况, 因为 `select` 报告带外数据作为一个异常情况.

POLLHUP

当读这个设备的进程见到文件尾, 驱动必须设置 `POLLUP(hang-up)`. 一个调用 `select` 的进程被告知设备是可读的, 如同 `select` 功能所规定的.

POLLERR

一个错误情况已在设备上发生. 当调用 `poll`, 设备被报告为可读可写, 因为读写都返回一个错误码而不阻塞.

POLLOUT

设备可被写入而不阻塞.

POLLWRNORM

这个位和 `POLLOUT` 有相同的含义, 并且有时它确实是相同的数. 一个可写的设备返回(`POLLOUT|POLLWRNORM`).

POLLWRBAND

同 `POLLRDBAND`, 这个位意思是带有零优先级的数据可写入设备. 只有 `poll` 的数据报实现使用这个位, 因为一个数据报看传送带外数据.

例子:

```
static unsigned int scull_p_poll(struct file *filp, poll_table *wait)
{
    struct scull_pipe *dev = filp->private_data;
    unsigned int mask = 0;
    /*
     * The buffer is circular; it is considered full
```

```

    * if "wp" is right behind "rp" and empty if the
    * two are equal.
    */
    down(&dev->sem);
    poll_wait(filp, &dev->inq, wait);
    poll_wait(filp, &dev->outq, wait);
    if (dev->rp != dev->wp)
        mask |= POLLIN | POLLRDNORM; /* readable */
    if (spacefree(dev))
        mask |= POLLOUT | POLLWRNORM; /* writable */
    up(&dev->sem);
    return mask;
}

```

这个代码简单地增加了 2 个 `scullpipe` 等待队列到 `poll_table`, 接着设置正确的掩码位, 根据数据是否可以读或写。

如果在进程 A 得到 `poll/select/epoll` 通知后, 另外一个进程 B 将数据读走/将缓冲区填满, 这时候进程 A 进来进行读写操作, 会如何?

```
int (*fasync) (int fd, struct file *filp, int mode);
```

异步通知. (常常假定异步能力只对 `socket` 和 `tty` 可用)

从用户的角度看异步通知的设置过程:

1. 指定一个进程作为文件的拥有者: `fcntl` 系统调用发出 `F_SETOWN` 命令, 进程 ID 被保存在 `filp->f_owner` 给以后使用
 2. 在设备中设置 `FASYNC` 标志: `fcntl` 系统调用发出 `F_SETFL` 命令
- 这样设置后设备有 新数据到达/缓冲有空间 的时候就会发送一个 `SIGIO` 信号到 `filp->f_owner` 中的进程(如果值为负值则发给整个进程组).

举例:

```

signal(SIGIO, &input_handler); /* dummy sample; sigaction() is better */
fcntl(STDIN_FILENO, F_SETOWN, getpid());
oflags = fcntl(STDIN_FILENO, F_GETFL); //get original setting
fcntl(STDIN_FILENO, F_SETFL, oflags | FASYNC);

```

从内核的角度看用户设置过程

1. 当发出 `F_SETOWN`, 一个值被赋值给 `filp->f_owner`.
2. 当发出 `F_SETFL` 来打开 `FASYNC`, 驱动的 `fasync` 方法被调用. 无论何时 `filp->f_flags` 中的 `FASYNC` 的值有改变, 都会调用驱动中的 `fasync` 方法(这个标志在文件打开时缺省地未设置).
3. 当数据到达, 向所有的注册异步通知的进程发出一个 `SIGIO` 信号.

从驱动的角度看内核响应过程:

1. 内核的第一步与驱动无关
2. 内核的第二步驱动应当用下列函数响应:

```
int fasync_helper(int fd, struct file *filp, int mode, struct fasync_struct **fa);
```

当 `FASYNC` 标志因一个打开文件而改变, 这个函数用来从相关的进程列表中添加或删除入口项. 它的所有参数除了最后一个, 都直接来自 `fasync` 方法.

`mode`: 0: 去除入口项; 其他: 添加入口项

fa: 是由驱动提供的一个 struct fasync_struct 结构. (*fa)在第一次使用之前应该初始化成 NULL, 不然可能会出错. 从函数返回的时候会被分配一块内存, 在 mode=0 时 free 掉. 所以添加与去除入口项必须配对使用

```
void kill_fasync(struct fasync_struct **fa, int sig, int band);
```

数据到达时通知相关的进程.

fa: 与 fasync_help 里的 fa 同

sig: 被传递的信号(常常是 SIGIO)

band: 异步状况. 在网络代码里可用来发送"紧急"或者带外数据

POLL_IN: 有新数据到达. 等同于 POLLIN|POLLRDNORM.

POLL_OUT: 有空间可供写入

举例: (注意: 若设备允许多次打开, 每个打开的 filp 需要有独立的 async_queue)

```
struct fasync_struct *async_queue = NULL;
```

```
static int fasync(int fd, struct file *filp, int mode)
```

```
{
    return fasync_helper(fd, filp, mode, &async_queue);
}
```

当数据到达, 用下面的语句来通知异步读者.

```
if (async_queue)
```

```
    kill_fasync(&async_queue, SIGIO, POLL_IN);
```

在 release 方法中应该调用

```
/* remove this filp from the asynchronously notified filp's */
```

```
fasync(-1, filp, 0);
```

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

如果未定义 llseek 方法, 内核缺省通过修改 filp->f_pos 来实现移位

如果需要禁止 lseek 操作, 需要在 open 中调用

```
int nonseekable_open(struct inode *inode; struct file *filp);
```

```
并把 file_operations::llseek 设为 no_llseek(loff_t no_llseek(struct file *file, loff_t offset, int whence))
```

举例:

```
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;
}

```

```
void *mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset)
```

mmap 操作: 将设备驱动里的一段内存映射到用户空间. 通过在 current->mm 中增加具有物理地址->虚拟地址映射关系的 pmd, pte 项来实现

设备驱动的内存区间(被映射区)的大小必须是 PAGE_SIZE 的整数倍.

设备驱动的内存区间(被映射区)的起始地址必须位于 PAGE_SIZE 整数倍的物理地址.

若用户空间所要求的 size 不是 PAGE_SIZE 的整数倍, 内核会自动将其扩大成整数倍

mmap 的设备驱动中的原型为

```
int (*mmap)(struct file *filp, struct vm_area_struct *vma);
```

filp: 设备驱动对应的文件描述符指针

vma: 最终用户空间得到的 struct vm_area_struct. 驱动所看到的这个参数已经被内核填充了大量数据, 驱动所需要做的就是将其地址区域建立合适的页表 (PMD: 中间目录描述表; PTE: 页表项). 若有需要可能还需要更新 struct vm_area_struct::vm_ops. vm_ops 是 struct vm_operations_struct 结构, 其定义如下

```

struct vm_operations_struct {
    void (*open)(struct vm_area_struct * area);
    void (*close)(struct vm_area_struct * area);
    struct page * (*nopage)(struct vm_area_struct * area, unsigned long address, int unused);
};

```

open: 在新增一个对 mmap 所映射区间的引用(比如说 fork)时会调用. 注意: 在 mmap 时(即第一次引用)不会自动调用此回调函数, 所以若有需要则应手工调用

close: 在撤销一个引用时会调用

nopage: 在映射区间发生缺页异常或做 mremap(重新映射)时会调用. 注意: 若 vm_ops 中实现了此函数, 那么 mmap 实现会有所不同, 详细见下

mmap 的实现:

修改页表: 通常会用下列两个函数实现

```
int remap_pfn_range(struct vm_area_struct *vma, unsigned long virt_addr, unsigned long pfn,
unsigned long size, pgprot_t prot);
```

```
int io_remap_page_range(struct vm_area_struct *vma, unsigned long virt_addr, unsigned long
phys_addr, unsigned long size, pgprot_t prot);
```

vma

用户区间的(需要被操作的)vma

virt_addr

用户虚拟地址. 这个函数建立页表为这个虚拟地址范围从 virt_addr 到 virt_addr_size.

pfn

页帧号, 也就是设备驱动中需要被映射出去的内存区间的物理地址.

这个页帧号简单地是物理地址右移 PAGE_SHIFT 位.

对大部分使用, VMA 结构的 vm_pgoff 成员正好包含你需要的值.

这个函数影响物理地址从 (pfn<<PAGE_SHIFT) 到 (pfn<<PAGE_SHIFT)+size.

size

需要被重新映射的区的大小, 以字节为单位.

prot

给新 VMA 要求的访问权限(Protection). 驱动可(并且应当)使用在 vma->vm_page_prot 中找到的值.

remap_pfn_range 用在 pfn 指向实际的系统 RAM 的情况下.

它只能访问保留页(内存管理不起作用的页)和超出物理内存的物理地址.

所以不能映射 get_free_page 得到的空间.

但 ioremap 函数返回的虚拟地址比较特殊, 所以可以用 remap_pfn_range 来映射

io_remap_page_range 用在 phys_addr 指向 I/O 内存时.

实际上, 这 2 个函数除了在 SPARCcpu 上, 每个体系上都是一致的. 并且在大部分情况下被使用看到 remap_pfn_range.

例子:

```
static int simple_remap_mmap(struct file *filp, struct vm_area_struct *vma)
{
    if (remap_pfn_range(vma, vma->vm_start, vma->vm_pgoff, vma->vm_end - vma->vm_start,
        vma->vm_page_prot))
        return -EAGAIN;
    vma->vm_ops = &simple_remap_vm_ops;
    simple_vma_open(vma);
    return 0;
}
```

nopage 实现: nopage 只需要返回引起异常的虚拟地址所对应的 struct page 的结构即可, 内核会自动将其挂入 current->mm 中去. 与 remap_pfn_range 不同, 它可以映射任何空间. 内核会自动调用 vm_ops::nopage 回调函数来实现 mmap, 所以 mmap 本身的实现比较简单, 只需要将 vm_ops 挂到 vma 中去. 典型的看起来象下面这样:

```
static int simple_nopage_mmap(struct file *filp, struct vm_area_struct *vma)
{
    unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;
    if (offset >= __pa(high_memory) || (filp->f_flags & O_SYNC))
        vma->vm_flags |= VM_IO;
    vma->vm_flags |= VM_RESERVED;
    vma->vm_ops = &simple_nopage_vm_ops;
    simple_vma_open(vma);
    return 0;
}
```

nopage 回调函数的实现例子:

```
struct page *simple_vma_nopage(struct vm_area_struct *vma, unsigned long address, int *type)
{
    struct page *pageptr;
```

```

unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;
unsigned long physaddr = address - vma->vm_start + offset;
unsigned long pageframe = physaddr >> PAGE_SHIFT;

if (!pfn_valid(pageframe))
    return NOPAGE_SIGBUS;
pageptr = pfn_to_page(pageframe);
get_page(pageptr);
if (type)
    *type = VM_FAULT_MINOR;
return pageptr;
}

```

这里,

`get_page` 是增加此页面的引用计数,必须实现

`type` 是返回错误类型, 对设备驱动来说, `VM_FAULT_MINOR` 是唯一正确的值

如果由于某些原因, 不能返回一个正常的页(即请求的地址超出驱动的内存区), 可以返回 `NOPAGE_SIGBUS` 指示错误; 也可以返回 `NOPAGE_OOM` 来指示由于资源限制导致的失败.

注意, `PCI` 内存被映射在最高的系统内存之上, 并且在系统内存中没有这些地址的入口, 所以没有对应 `struct page` 来返回指针, `nopage` 不能在这些情况下使用 -- 必须使用 `remap_pfn_range` 代替.

如果 `nopage` 方法被留置为 `NULL`, 处理页出错的内核代码映射零页到出错的虚拟地址.

零页是一个写时拷贝的页, 任何引用零页的进程都看到一个填满 0 的页. 如果进程写到这个页, 内核将一个实际的页挂到进程中去.

因此, 如果一个进程通过调用 `mremap` 扩展一个映射的页, 并且驱动还没有实现 `nopage`, 那么进程将不会因为一个段错误而是因为一个零填充的内存结束

引用设备内存不应当被处理器缓存

防止被缓存的方法可以参考 `driver/video/fbmem.c->fb_mmap` 的做法, 比如其中提及的 `arm` 体系防止空间被缓存的做法如下

```

.....
#ifdef __arm__
    vma->vm_page_prot = pgprot_noncached(vma->vm_page_prot);
    /* This is an IO map - tell maydump to skip this VMA */
    vma->vm_flags |= VM_IO;
#endif
.....

```

并发与竞争

锁使用的规则

不允许一个持锁者第 2 次请求锁

非 static 函数必须明确在函数内部加锁(而不应该留给外部调用前处理); 静态函数可自行处理

当多个锁必须同时获得时,应当以同一顺序申请

当本地与内核的锁必须同时获得时,先申请本地的锁

当 mutex 与 spin lock 必须同时获得时, 先申请 mutex(若先申请 spin lock, 那么另一个想要申请 spin lock 的进程会一直自旋耗用大量资源)

semaphore

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

初始化一个 semaphore, 并将其赋值为 val

mutex:

semaphore 的特殊形式, val 仅允许在 1 和 0 之间变动

```
DECLARE_MUTEX(name);
```

定义并初始化一个未上锁的 mutex

```
DECLARE_MUTEX_LOCKED(name);
```

定义并初始化一个已上锁的 mutex

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

初始化一个未上锁的 mutex

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

初始化一个已上锁的 mutex

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

递减 semaphore 值, 如果必要就让当前进程睡眠等待直到 semaphore 可用

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

同 down, 但是操作是可中断的.

它允许一个在等待一个 semaphore 的用户空间进程被用户中断. 作为一个通用的规则, 不应该使用不可中断的操作, 除非实在是没有选择. 不可中断操作将创建不可杀死的进程.

如果操作是可中断的, 函数返回一个非零值, 并且调用者不持有 semaphore. 正确的使用 down_interruptible 需要一直检查返回值并且针对性地响应.

举例:

```
DECLARE_MUTEX(mutex);
```

...

```
if (down_interruptible(&mutex))
```

```
return -ERESTARTSYS;
```

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

如果在调用 down_trylock 时 semaphore 不可用, 它将立刻返回一个非零值.

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

释放 semaphore

```
void init_rwsem(struct rw_semaphore *sem);
```

```

void down_read(struct rw_semaphore *sem);
int down_read_trylock(struct rw_semaphore *sem);
void up_read(struct rw_semaphore *sem);
void down_write(struct rw_semaphore *sem);
int down_write_trylock(struct rw_semaphore *sem);
void up_write(struct rw_semaphore *sem);
void downgrade_write(struct rw_semaphore *sem);

```

这一系列函数基本与 mutex 对应版本同。区别是，可以有多个进程同时拥有读锁，但仅允许一个进程拥有写锁

另外一个特性：如果有进程尝试写锁定后(即使没有拥有写锁),所有其他尝试读锁定的进程都将等待,直到写锁定解除。

completion

```
DECLARE_COMPLETION(name)
```

定义并初始化一个 completion

```
INIT_COMPLETION(struct completion c);
```

重新初始化一个 completion, 主要是用在被唤醒的进程重新进入等待前的初始化

```
void init_completion(struct completion *c)
```

初始化一个 completion

```
void wait_for_completion(struct completion *c)
```

进行一个不可打断的等待

```
void complete(struct completion *c)
```

唤醒一个等待的进程

```
void complete_call(struct completion *c)
```

唤醒所有等待的进程

```
void complete_and_exit(struct completion *c, long retval)
```

在内核线程 A 收到退出命令后,通知另一个内核线程 B 退出, 并等待 B 退出完成; B 退出完成后调用 complete 通知 A, 如果这种情况下用的是 completion 机制而 A 最后等待 complete 的时候调用的是这个函数,那么,A 一收到 B 退出的通知就会结束整个线程

spin lock

spin lock 是一个互斥设备, 只能有 2 个值:"上锁"和"解锁".

内核抢占(高优先级的进程抢占低优先级的进程)在持有 spin lock 期间被禁止

```
spinlock_t my_lock = SPIN_LOCK_UNLOCKED
```

编译时初始化

```
void spin_lock_init(spinlock_t *lock)
```

运行时初始化

```
void spin_lock(spinlock_t *lock)
```

```
void spin_lock_irqsave(spinlock_t *lock, unsigned long flags)
```

```
void spin_lock_irq(spinlock_t *lock)
```

```
void spin_lock_bh(spinlock_t *lock)
```

加锁

`spin_lock_irqsave` 在获得锁之前在当前处理器禁止中断, 之前的中断状态会保存在 `flags` 里. 按说 `flags` 按值传递的, 如何保存 `irq` 状态呢? 因为 `spin_lock_irqsave` 不是函数而是宏. ^_^
`spin_lock_irq` 如果可以确定没有其他地方禁止中断(因为对应的 `unlock` 函数会打开中断), 可以使用这个函数

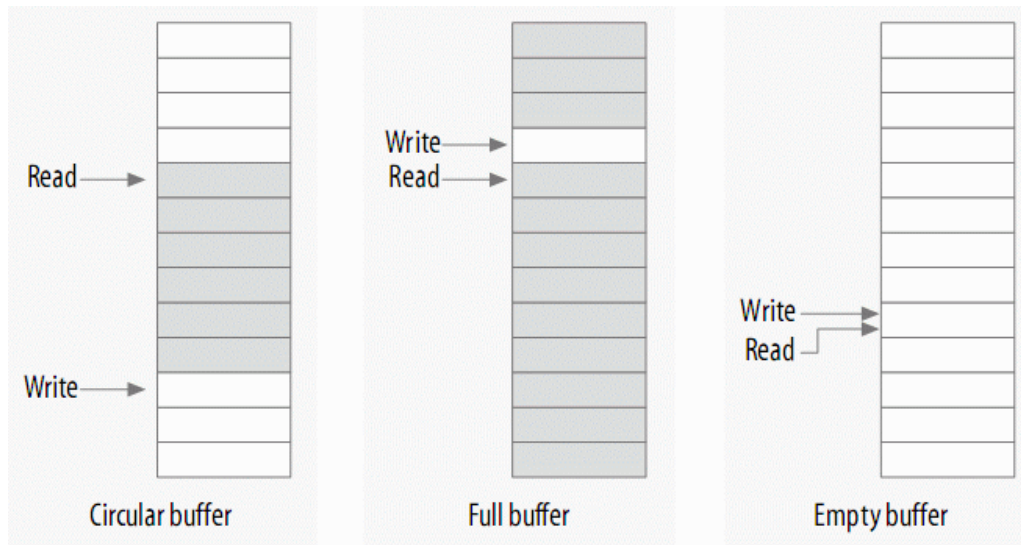
`spin_lock_bh` 在获取锁之前禁止软中断.

之所以需要引入 `irq/soft irq` 开关支持, 是因为如果在线程内拥有锁, 这时候有中断进来, 而中断也要拥有锁才能工作, 就导致了死锁

```
void spin_unlock(spinlock_t *lock)
void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags)
void spin_unlock_irq(spinlock_t *lock)
void spin_unlock_bh(spinlock_t *lock)
与加锁对应的各个版本的解锁
int spin_trylock(spinlock_t *lock)
int spin_trylock_bh(spinlock_t *lock)
非阻塞操作. 没有禁止中断的"try"版本
rwlock_t my_rwlock = RW_LOCK_UNLOCKED
rwlock_init(&my_rwlock)
void read_lock(rwlock_t *lock)
void read_lock_irqsave(rwlock_t *lock, unsigned long flags)
void read_lock_irq(rwlock_t *lock)
void read_lock_bh(rwlock_t *lock)
void read_unlock(rwlock_t *lock)
void read_unlock_irqrestore(rwlock_t *lock, unsigned long flags)
void read_unlock_irq(rwlock_t *lock)
void read_unlock_bh(rwlock_t *lock)
void write_lock(rwlock_t *lock)
void write_lock_irqsave(rwlock_t *lock, unsigned long flags)
void write_lock_irq(rwlock_t *lock)
void write_lock_bh(rwlock_t *lock)
int write_trylock(rwlock_t *lock)
void write_unlock(rwlock_t *lock)
void write_unlock_irqrestore(rwlock_t *lock, unsigned long flags)
void write_unlock_irq(rwlock_t *lock)
void write_unlock_bh(rwlock_t *lock)
读写锁版本的 spin lock
```

用其他方式避免使用锁

环形缓冲



原子变量: `atomic_t`. 所有原子量操作必须使用如下函数

`atomic_t v = ATOMIC_INIT(0)`: 声明并初始化 `atomic` 变量

`void atomic_set(atomic_t *v, int i)`: 设置值

`int atomic_read(atomic_t *v)`: 读值

`void atomic_add(int i, atomic_t *v)`: 值加 `i`

`void atomic_sub(int i, atomic_t *v)`: 值减 `i`

`void atomic_inc(atomic_t *v)`: 值+1

`void atomic_dec(atomic_t *v)`: 值-1

`int atomic_inc_and_test(atomic_t *v)`: 值+1, 并测试值是否为 0, 若为 0 返回真, 否则返回假

`int atomic_dec_and_test(atomic_t *v)`: 值-1, 并测试值是否为 0, 若为 0 返回真, 否则返回假

`int atomic_sub_and_test(int i, atomic_t *v)`: 值-`i`, 并测试值是否为 0, 若为 0 返回真, 否则返回假

`int atomic_add_negative(int i, atomic_t *v)`: 值+`i`, 并测试值是否为负, 若为负返回真, 否则返回假

`int atomic_add_return(int i, atomic_t *v)`: 值+`i`, 并返回值

`int atomic_sub_return(int i, atomic_t *v)`: 值-`i`, 并返回值

`int atomic_inc_return(atomic_t *v)`: 值+1, 并返回值

`int atomic_dec_return(atomic_t *v)`: 值-1, 并返回值

位操作:

位操作依赖于体系结构. `nr` 参数(描述要操作哪个位)常常定义为 `int`, 也可能是 `unsigned long`; 要修改的地址常常是一个 `unsigned long` 指针, 但是几个体系使用 `void *` 代替.

`void set_bit(nr, void *addr)`: 在 `addr` 指向的数据项中将第 `nr` 位设置为 1

`void clear_bit(nr, void *addr)`: 在 `addr` 指向的数据项中将第 `nr` 位设置为 0

`void change_bit(nr, void *addr)`: 对 `addr` 指向的数据项中的第 `nr` 位取反

`test_bit(nr, void *addr)`: 返回 `addr` 指向的数据项中的第 `nr` 位

`int test_and_set_bit(nr, void *addr)`: 在 `addr` 指向的数据项中将第 `nr` 位设置为 1, 并返回设置前的值

`int test_and_clear_bit(nr, void *addr)`: 在 `addr` 指向的数据项中将第 `nr` 位设置为 0, 并返回设置前的值

`int test_and_change_bit(nr, void *addr)`: 对 `addr` 指向的数据项中的第 `nr` 位取反, 并返回取

反前的值

Mark: 大部分现代代码不使用位操作,而是使用自选锁

seqlock: 2.6 内核提供的,

```
seqlock_t lock1 = SEQLOCK_UNLOCKED;
```

```
seqlock_t lock2;
```

```
seqlock_init(&lock2);
```

读存取通过在进入临界区入口获取一个(无符号的)整数序列来工作. 在退出时, 那个序列值与当前值比较; 如果不匹配, 读存取必须重试. 结果是, 读者代码象下面的形式:

```
unsigned int seq;
```

```
do {
```

```
    seq = read_seqbegin(&the_lock);
```

```
    /* Do what you need to do */
```

```
} while read_seqretry(&the_lock, seq);
```

这个类型的锁常常用在保护某种简单计算, 需要多个一致的值. 如果这个计算最后的测试表明发生了一个并发的写, 结果被简单地丢弃并且重新计算.

如果你的 seqlock 可能从一个中断处理里存取, 你应当使用 IRQ 安全的版本来代替:

```
unsigned int read_seqbegin_irqsave(seqlock_t *lock, unsigned long flags);
```

```
int read_seqretry_irqrestore(seqlock_t *lock, unsigned int seq, unsigned long flags);
```

写者必须获取一个排他锁来进入由一个 seqlock 保护的临界区. 为此, 调用:

```
void write_seqlock(seqlock_t *lock);
```

写锁由一个自旋锁实现, 因此所有的通常的限制都适用. 调用:

```
void write_sequnlock(seqlock_t *lock);
```

来释放锁. 因为自旋锁用来控制写存取, 所有通常的变体都可用:

```
void write_seqlock_irqsave(seqlock_t *lock, unsigned long flags);
```

```
void write_seqlock_irq(seqlock_t *lock);
```

```
void write_seqlock_bh(seqlock_t *lock);
```

```
void write_sequnlock_irqrestore(seqlock_t *lock, unsigned long flags);
```

```
void write_sequnlock_irq(seqlock_t *lock);
```

```
void write_sequnlock_bh(seqlock_t *lock);
```

还有一个 write_tryseqlock 在它能够获得锁时返回非零.

读-拷贝-更新(Read-Copy-Update)

针对很多读但是较少写的情况. 所有操作通过指针.

读: 直接操作

写: 将数据读出来, 更新数据, 将原指针指向新数据(这一步需要另外做原子操作的保护)

作为在真实世界中使用 RCU 的例子, 考虑一下网络路由表. 每个外出的报文需要请求检查路由表来决定应当使用哪个接口. 这个检查是快速的, 并且, 一旦内核发现了目标接口, 它不再需要路由表入口项. RCU 允许路由查找在没有锁的情况下进行, 具有相当多的性能好处. 内核中的 Startmode 无线 IP 驱动也使用 RCU 来跟踪它的设备列表.

在读这一边, 使用一个 RCU- 保护的数据结构的代码应当用 rcu_read_lock 和 rcu_read_unlock 调用将它的引用包含起来.

RCU 代码往往是象这样:

```
struct my_stuff *stuff;
```

```
rcu_read_lock();
```

```
stuff = find_the_stuff(args...);
```

```
do_something_with(stuff);
```

```
rcu_read_unlock();
```

`rcu_read_lock` 调用是很快: 它禁止内核抢占但是没有等待任何东西. 在读"锁"被持有时执行的代码必须是原子的. 在对 `rcu_read_unlock` 调用后, 没有使用对被保护的资源的引用. 需要改变被保护的结构的代码必须进行几个步骤: 分配一个新结构, 如果需要就从旧的拷贝数据; 接着替换读代码所看到的指针; 释放旧版本数据(内存).

在其他处理器上运行的代码可能仍然有对旧数据的一个引用, 因此不能立刻释放. 相反, 写代码必须等待直到它知道没有这样的引用存在了. 因为所有持有对这个数据结构引用的代码必须(规则规定)是原子的, 我们知道一旦系统中的每个处理器已经被调度了至少一次, 所有的引用必须消失. 这就是 RCU 所做的; 它留下了一个等待直到所有处理器已经调度的回调; 那个回调接下来被运行来进行清理工作.

改变一个 RCU-保护的数据结构的代码必须通过分配一个 `struct rcu_head` 来获得它的清理回调, 尽管不需要以任何方式初始化这个结构. 通常, 那个结构被简单地嵌入在 RCU 所保护的大的资源里面. 在改变资源完成后, 应当调用:

```
void call_rcu(struct rcu_head *head, void (*func)(void *arg), void *arg);
```

给定的 `func` 在安全的时候调用来释放资源

全部 RCU 接口比我们已见的要更加复杂; 它包括, 例如, 辅助函数来使用被保护的链表. 详细内容见相关的头文件

阻塞 I/O

运行在原子上下文时不能睡眠. 持有一个自旋锁, `seqlock`, RCU 锁或中断已关闭时不能睡眠. 但在持有一个旗标时睡眠是合法的

不能对醒后的系统状态做任何的假设, 并且必须检查来确保你在等待的条件已经满足

确保有其他进程会做唤醒动作

明确的非阻塞 I/O 由 `filp->f_flags` 中的 `O_NONBLOCK/O_NDELAY` 标志来指示. 只有 `read`, `write`, 和 `open` 文件操作受到非阻塞标志影响

下列情况下应该实现阻塞

如果一个进程调用 `read` 但是没有数据可用(尚未), 这个进程必须阻塞. 这个进程在有数据达到时被立刻唤醒, 并且那个数据被返回给调用者, 即便小于在给方法的 `count` 参数中请求的数量.

如果一个进程调用 `write` 并且在缓冲中没有空间, 这个进程必须阻塞, 并且它必须在一个与用作 `read` 的不同的等待队列中. 当一些数据被写入硬件设备, 并且在输出缓冲中的空间变空闲, 这个进程被唤醒并且写调用成功, 尽管数据可能只被部分写入如果在缓冲中没有空间给被请求的 `count` 字节.

若需要在 `open` 中实现对 `O_NONBLOCK` 的支持, 可以返回 `-EAGAIN`

```
DECLARE_WAIT_QUEUE_HEAD(name);
```

定义并初始化一个等待队列

```
init_waitqueue_head(wait_queue_head_t *name);
```

初始化一个等待队列

```
wait_event(queue, condition)
```

`wait_event_interruptible(queue, condition)`

`wait_event_timeout(queue, condition, timeout)`

`wait_event_interruptible_timeout(queue, condition, timeout)`

`queue`: 是要用的等待队列头. 注意它是"通过值"传递的, 而不是指针

`condition`: 需要检查的条件, 只有这个条件为真时才会返回. 注意条件可能被任意次地求值, 因此它不应当有任何边界效应(side effects, 按我的理解就是当外界条件未改变时, 每次计算得到的结果应该相同)

`timeout`: 超时值. 表示要等待的 jiffies 数, 是一个相对时间值. 如果这个进程被其他事件唤醒, 它返回以 jiffies 表示的剩余超时值

上述 4 个函数带 `interruptible` 的是可被中断的

`void wake_up(wait_queue_head_t *queue);`

`void wake_up_interruptible(wait_queue_head_t *queue);`

`wake_up` 唤醒所有的在给定队列上等待的进程. `wake_up_interruptible` 限制只唤醒因 `wait_event_interruptible/wait_event_interruptible_timeout` 睡眠的进程

`wake_up_nr(wait_queue_head_t *queue, int nr);`

`wake_up_interruptible_nr(wait_queue_head_t *queue, int nr);`

类似 `wake_up`, 但它们能够唤醒 `nr` 个互斥等待者, 而不只是一个. 注意: 0 被解释为请求所有的互斥等待者都被唤醒

`wake_up_all(wait_queue_head_t *queue);`

`wake_up_interruptible_all(wait_queue_head_t *queue);`

唤醒所有的进程, 不管它们是否进行互斥等待(尽管可中断的类型仍然跳过在做不可中断等待的进程)

`wake_up_interruptible_sync(wait_queue_head_t *queue);`

正常地, 一个被唤醒的进程可能抢占当前进程, 并且在 `wake_up` 返回之前被调度到处理器. 换句话说, 调用 `wake_up` 可能不是原子的. 如果调用 `wake_up` 的进程运行在原子上下文(它可能持有一个自旋锁, 例如, 或者是一个中断处理), 这个重调度不会发生. 但是, 如果你需要明确要求不要被调度出处理器在那时, 你可以使用 `wake_up_interruptible` 的"同步"变体. 这个函数唤醒其他进程, 但不会让新唤醒的进程抢占 CPU

6.2.5.1. 一个进程如何睡眠

放弃处理器是最后一步, 但是要首先做一件事: 你必须先检查你在睡眠的条件. 做这个检查失败会引入一个竞争条件; 如果你忙于上面的这个过程并且有其他的线程刚刚试图唤醒你, 如果这个条件变为真会发生什么? 你可能错过唤醒并且睡眠超过你预想的时间. 因此, 在睡眠的代码下面, 典型地你会见到下面的代码:

```
if (!condition)
```

```
    schedule();
```

通过在设置了进程状态后检查我们的条件, 我们涵盖了所有的可能的事件进展. 如果我们在等待的条件已经在设置进程状态之前到来, 我们在这个检查中注意到并且不真正地睡眠. 如果之后发生了唤醒, 进程被置为可运行的不管是否我们已真正进入睡眠.

如果在 `if` 判断之后, `schedule` 之前, 有其他进程试图唤醒当前进程, 那么当前进程就会被置为可运行的(但可能会到下次调度才会再次运行), 所以这个过程是安全的

手动睡眠

```
DEFINE_WAIT(my_wait);
```

定义并初始化一个等待队列入口项

```
init_wait(wait_queue_t*);
```

初始化等待队列入口项

```
void prepare_to_wait(wait_queue_head_t *queue, wait_queue_t *wait, int state);
```

添加你的等待队列入口项到队列, 并且设置进程状态.

queue: 等待队列头

wait: 等待队列入口项

state: 进程的新状态; TASK_INTERRUPTIBLE/TASK_UNINTERRUPTIBLE

调用 prepare_to_wait 之后, 需再次检查确认需要等待, 便可调用 schedule 释放 CPU

在 schedule 返回之后需要调用下面的函数做一些清理动作

```
void finish_wait(wait_queue_head_t *queue, wait_queue_t *wait);
```

在此之后需要再次检查是否需要再次等待(条件已经满足还是被其他等待的进程占用?)

互斥等待

当一个等待队列入口有 WQ_FLAG_EXCLUSIVE 标志置位, 它被添加到等待队列的尾部. 没有这个标志的入口项添加到开始.

wake_up 调用在唤醒第一个有 WQ_FLAG_EXCLUSIVE 标志的进程后停止(即进行互斥等待的进程一次只以顺序的方式唤醒一个). 但内核仍然每次唤醒所有的非互斥等待者.

用函数

```
void prepare_to_wait_exclusive(wait_queue_head_t *queue, wait_queue_t *wait, int state);
```

代替手动睡眠中的 prepare_to_wait 即可实现互斥等待

老版本的函数

```
void sleep_on(wait_queue_head_t *queue);
```

```
void interruptible_sleep_on(wait_queue_head_t *queue);
```

这些函数无法避免竞争(在条件判断和 sleep_on 之间会存在竞争). 见上面"6.2.5.1. 一个进程如何睡眠"的分析

时间

HZ: 1 秒产生的 tick. 是一个体系依赖的值. 内核范围此值通常为 100 或 1000, 而对应用程序, 此值始终是 100

jiffies_64: 从系统开机到现在经过的 tick, 64bit 宽

jiffies: jiffies_64 的低有效位, 通常为 unsigned long. 编程时需要考虑 jiffies 隔一段时间会环绕(重新变成 0)的问题

```
u64 get_jiffies_64(void);
```

得到 64 位的 jiffies 值

```
int time_after(unsigned long a, unsigned long b);
```

若 a 在 b 之后, 返回真

```
int time_before(unsigned long a, unsigned long b);
```

若 a 在 b 之前, 返回真

```
int time_after_eq(unsigned long a, unsigned long b);
```

若 a 在 b 之后或 a 与 b 相等, 返回真

```
int time_before_eq(unsigned long a, unsigned long b);
```

若 a 在 b 之前或 a 与 b 相等, 返回真

```
unsigned long timespec_to_jiffies(struct timespec *value);
```

```
void jiffies_to_timespec(unsigned long jiffies, struct timespec *value);
```

unsigned long timeval_to_jiffies(struct timeval *value);
void jiffies_to_timeval(unsigned long jiffies, struct timeval *value);
struct timeval 和 struct timespec 与 jiffies 之间的转换

rdtsc(low32,high32);

rdtscl(low32);

rdtscll(var64);

read tsc/read tsc low/read tsc long long: 读取短延时(很容易回绕), 高精度的时间值. 不是所有平台都支持

cycles_t get_cycles(void);

类似 rdtsc, 但每个平台都提供. 若 CPU 为提供相应功能则此函数一直返回 0

注意:

它们在一个 SMP 系统中不能跨处理器同步. 为保证得到一个一致的值, 你应当为查询这个计数器的代码禁止抢占

短延时(很容易回绕), 高精度

不是所有平台都支持

unsigned long mktime (unsigned int year, unsigned int mon, unsigned int day, unsigned int hour, unsigned int min, unsigned int sec);

转变一个墙上时钟时间到一个 jiffies 值

void do_gettimeofday(struct timeval *tv);

获取当前时间

struct timespec current_kernel_time(void);

获取当前时间. 注意: 返回值是个结构体不是结构体指针

延时

长延时之忙等待(cpu 密集型进程会有动态降低的优先级)

while (time_before(jiffies, j1))

 cpu_relax();

让出 CPU(无法确定什么时候重新得到 CPU, 即无法确定什么时候循环结束, 也即无法保证延时精度)

while (time_before(jiffies, j1)) {

 schedule();

}

long wait_event_timeout(wait_queue_head_t q, condition, long timeout);

long wait_event_interruptible_timeout(wait_queue_head_t q, condition, long timeout);

timeout 表示要等待的 jiffies 数, 是相对时间值, 不是一个绝对时间值. 如果超时到, 这些函数返回 0; 如果这个进程被其他事件唤醒, 它返回以 jiffies 表示的剩余超时值.

set_current_state(TASK_INTERRUPTIBLE);

signed long schedule_timeout(signed long timeout);

不需要等待特定事件的延时

注意: 上述几个函数在超时事件到与被调度之间有一定的延时

短延时之忙等待

void ndelay(unsigned long nsecs);

void udelay(unsigned long usecs);

```
void mdelay(unsigned long msecs);
```

这几个函数是体系特定的. 每个体系都实现 `udelay`, 但是其他的函数可能或者不可能定义
获得的延时至少是要求的值, 但可能更多

```
void msleep(unsigned int millisecs);
```

```
unsigned long msleep_interruptible(unsigned int millisecs);
```

```
void ssleep(unsigned int seconds)
```

使调用进程进入睡眠给定的毫秒数.

如果进程被提早唤醒(`msleep_interruptible`), 返回值是剩余的毫秒数

内核定时器

定时器是基于中断的(通常来讲是通过软中断), 所以会缺少进程上下文.

定时器

Timer 运行在非进程上下文中. 通常只能提供 tick 为单位的精度

```
struct timer_list {
```

```
/* ... */
```

```
unsigned long expires;
```

```
void (*function)(unsigned long);
```

```
unsigned long data;
```

```
};
```

expires: 超时时间

function: 时间到时需要回调的函数

data: function 的参数

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

初始化一个 timer

```
struct timer_list TIMER_INITIALIZER(_function, _expires, _data);
```

定义并初始化一个 timer

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

将 timer 加入调度. 每次这个 timer 被调度后, 就会从调度链表中去掉, 所以如果需要反复运行, 需要重新 `add_timer` 或者用 `mod_timer`

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

将 timer 从调度链表中去除

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

更新一个定时器的超时时间(并将其重新加入到调度链表中去). `mod_timer` 也可代替 `add_timer` 用于激活 timer

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

同 `del_timer` 一样工作, 并且还保证当它返回时, 定时器函数不在任何 CPU 上运行. 这个函数应当在大部分情况下比 `del_timer` 更优先使用. 如果它在非原子上下文被调用可能导致睡眠, 在其他情况下会忙等待. 在持有锁时要十分小心调用. 另外: 如果一个 timer 重新激活自己, 可能会导致此函数一直等待下去(解决方法是在重新激活自己前加入必要的条件限制)

```
int timer_pending(const struct timer_list * timer);
```

返回真或假来指示是否定时器当前是否正被调度运行

调度

Tasklet: 在软中断上下文中运行, 所以必须是原子的

如果系统不在重载下, 可能立刻运行; 否则, 也不会晚于下一个 tick. 通常, 系统在退出 irq 之前会检查有没有 softirq 需要运行, 一般来说 Tasklet 会在这个时候被调度

一个 tasklet 可能和其他 tasklet 并发执行, 但是对它自己是严格地串行的 同样的 tasklet 在一个时刻只能运行在一个 CPU(通常为调度它的 CPU)上

```
struct tasklet_struct {  
    /* ... */  
    void (*func)(unsigned long);  
    unsigned long data;  
};
```

func: 需要运行的函数

data: func 的参数

```
void tasklet_init(struct tasklet_struct *t, void (*func)(unsigned long), unsigned long data);
```

初始化一个 tasklet 结构

```
DECLARE_TASKLET(name, func, data);
```

定义并初始化一个 tasklet 结构

```
DECLARE_TASKLET_DISABLED(name, func, data);
```

定义并初始化一个被禁止调度的 tasklet 结构

```
void tasklet_disable(struct tasklet_struct *t);
```

禁止一个 tasklet.

注意:

这个 tasklet 仍然可能被调度, 但是直到它被使能之后才会执行.

如果这个 tasklet 正在运行, tasklet_disable 忙等待直到这个 tasklet 退出

```
void tasklet_disable_nosync(struct tasklet_struct *t);
```

同 tasklet_disable, 但如果这个 tasklet 正在运行, 则不等待其退出

```
void tasklet_enable(struct tasklet_struct *t);
```

使能一个之前被禁止的 tasklet.

注意:

tasklet_enable 必须匹配 tasklet_disable, 因为内核跟踪每个 tasklet 的"禁止次数".

```
void tasklet_schedule(struct tasklet_struct *t);
```

调度 tasklet 执行.

如果一个 tasklet 在运行前被再次调度(tasklet_schedule), 它只运行一次.

如果它在运行中被调度, 它在完成此次运行后会再次运行; 这保证了在其他事件被处理当中发生的事件收到应有的注意. (这个做法也允许一个 tasklet 重新调度它自己)

```
void tasklet_hi_schedule(struct tasklet_struct *t);
```

调度 tasklet 在更高优先级执行.

```
void tasklet_kill(struct tasklet_struct *t);
```

确保了 this tasklet 不会被再次调度运行.

如果这个 tasklet 正在运行, 这个函数等待直到它执行完毕.

如果这个 tasklet 重新调度它自己, 可能会导致 tasklet_kill 一直等待(所以需要在一个 tasklet 重新调度自己前需要加入一些条件限制).

工作队列:

每个工作队列有一个或多个专用的进程("内核线程")

工作队列就在这个特殊的内核上下文中运行,所以可以是非原子的,但不能存取用户空间

注意:缺省队列对所有驱动程序来说都是可用的;但是只有经过 GP 许可的驱动程序可以用自定义的工作队列

```
DECLARE_WORK(name, void (*function)(void *), void *data);
```

定义并初始化一个工作队列

```
INIT_WORK(struct work_struct *work, void (*function)(void *), void *data);
```

初始化一个工作队列

```
PREPARE_WORK(struct work_struct *work, void (*function)(void *), void *data);
```

同 INIT_WORK, 但不把它链接到工作队列中

```
struct workqueue_struct *create_workqueue(const char *name);
```

这个函数在每个处理器上都创建一个专用的内核线程

```
struct workqueue_struct *create_singlethread_workqueue(const char *name);
```

同 create_workqueue, 但只在一个 cpu 上创建线程

```
int queue_work(struct workqueue_struct *queue, struct work_struct *work);
```

添加工作到给定的队列

```
int queue_delayed_work(struct workqueue_struct *queue, struct work_struct *work, unsigned long delay);
```

添加工作 work 到给定的队列 queue, 但这个工作会延迟 delay 个 jiffies 才会执行

这两个函数如果返回非 0 意味着队列中已经有工作 work 存在了

```
int cancel_delayed_work(struct work_struct *work);
```

取消一个已经加入队列但未运行的工作

若此函数返回 0 表示工作正在运行(可能在其他 cpu 上). 这种情况下, 工作会继续,但不会被再次添加到工作队列中去

若需要确保指定的工作没有运行,需要在这个函数后跟随下列函数:

```
void flush_workqueue(struct workqueue_struct *queue);
```

清空队列中的所有工作

```
void destroy_workqueue(struct workqueue_struct *queue);
```

销毁一个工作队列

内核共享队列:

```
int schedule_work(struct work_struct *work);
```

向内核缺省工作队列(共享队列)中添加一个任务

```
int schedule_delayed_work(struct work_struct *work, unsigned long delay);
```

向内核缺省工作队列(共享队列)中添加一个任务并延迟执行

```
int cancel_delayed_work(struct work_struct *work);
```

```
void flush_scheduled_work(void);
```

共享队列中的相应版本

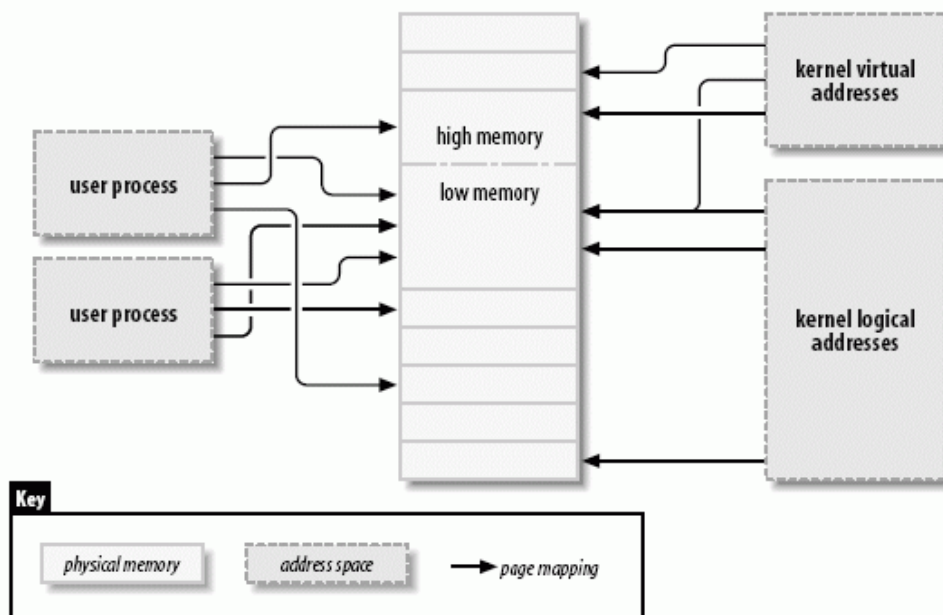
内存分配:

Linux 内核最少 3 个内存区: DMA-capable memory, low memory, high memory.

DMA-capable memory: x86 平台中, DMA 区用在 RAM 的前 16MB, 因为传统的 ISA 设备只能在这个区域内做 DMA 操作; PCI 设备没有这个限制.

Low memory: 因为所有可用的内存最初都是被映射到内核空间再进行分配的, 而内核空间的大小有限(通常被配置为 1G), 所以实际能够利用的内存大小也是有限的, 这部分内存就是 low memory. 内核的很多数据结构都必须放在 low memory 中

High memory: 由于 low memory 大小有限, 在配置了大量内存的主机里, 大于 low memory 部分的空间只能通过明确的虚拟映射映射进来. 这部分大于 low memory 部分的内存就叫 High memory. High memory 没有逻辑地址. 打开内核的 High memory 支持会导致性能下降



User virtual addresses: 用户(进程)虚拟地址. 这是被用户程序见到的常规地址. 用户地址在长度上是 32 位或者 64 位, 依赖底层的硬件结构. 每个进程有它自己的虚拟地址空间.

Physical addresses: 物理地址. 在处理器和系统内存之间使用的地址. 物理地址是 32 或者 64 位的量. 32 位系统在某些情况下可使用更大的物理地址.

Bus addresses: 总线地址. 在外设和内存之间使用的地址. 通常, 它们和处理器使用的物理地址相同; 但一些体系可提供一个 I/O 内存管理单元(IOMMU), 它在总线和主内存之间重映射地址. 一个 IOMMU 可用多种方法使事情简单(例如, 使散布在内存中的缓冲对设备看来是连续的)

Kernel logical addresses: 内核逻辑地址. 这些组成了正常的内核地址空间. 这些地址映射了部分(也许全部)主存并且常常被当作物理内存来对待.

在大部分的体系上, 逻辑地址和它们的相关物理地址只差一个常量偏移. 逻辑地址使用硬件的本地指针大小, 因此, 可能不能在配置大量内存(大于 4G)的 32 位系统上寻址所有的物理内存.

逻辑地址常常存储于 `unsigned long` 或者 `void *` 类型的变量中.

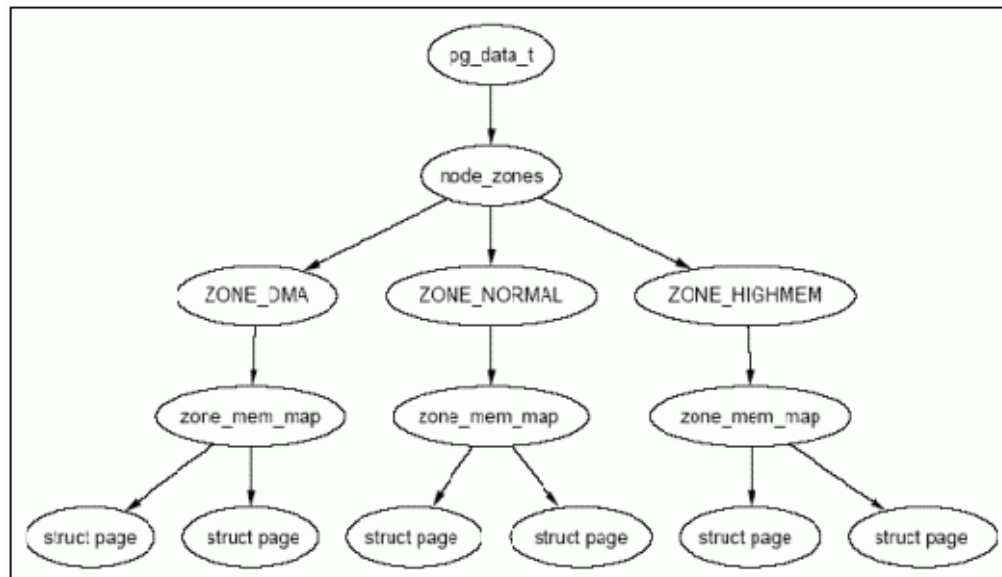
从 `kmalloc` 返回的内存有内核逻辑地址.

Kernel virtual addresses: 内核虚拟地址. 类似于逻辑地址, 它们都是从内核空间地址到物理地址的映射. 内核虚拟地址不必有逻辑地址空间具备的线性的, 一对一到物理地址的映射

所有的逻辑地址都是内核虚拟地址, 但是许多内核虚拟地址不是逻辑地址.
例如, `vmalloc` 分配的内存有虚拟地址(但没有直接物理映射). `kmap` 函数也返回虚拟地址.
虚拟地址常常存储于指针变量.

内存相关的数据结构:

从物理系统供给内存的角度看, 各个数据结构总体关系如下:



一个线性的存储区被称为节点(node)

```

typedef struct pglist_data {
    zone_t node_zones[MAX_NR_ZONES];           该节点的 zone 类型, 一般包括
                                                ZONE_HIGHMEM、ZONE_NORMAL 和 ZONE_DMA 三类
    zonelist_t node_zonelists[GFP_ZONEMASK+1]; 分配时内存时 zone 的排序。由
    free_area_init_core()通过 page_alloc.c::build_zonelists()设置 zone 的顺序
    int nr_zones;                               该节点的 zone 个数, 可以从 1 到 3(即上面
    的 ZONE_HIGHMEM、ZONE_NORMAL 和 ZONE_DMA), 但不是所有的节点都需要有 3
    个 zone
    struct page *node_mem_map;                  当前节点的 struct page 数组, 可能为全局
    mem_map 中的某个位置
    unsigned long *valid_addr_bitmap;           节点内存空洞的位图
    struct bootmem_data *bdata;
    unsigned long node_start_paddr;             该节点的起始物理地址
    unsigned long node_start_mapnr;             全局 mem_map 中的页偏移
    unsigned long node_size;                    该 zone 内的页框总数
    int node_id;                                该节点的 ID, 全系统节点 ID 从 0 开始. 系统中所有
    节点都维护在 pgdat_list 列表中
    struct pglist_data *node_next;
}pg_data_t;
  
```

节点中的内存被分为多块(通常为 DMA memory, low memory, high memory), 这样的块被称为 zone.

通常有这样的划分: ZONE_DMA: 0 -- 16MB ; ZONE_NORMAL: 16MB - 896MB ;

ZONE_HIGHMEM: 896MB --

```
typedef struct zone_struct {  
    /*Commonly accessed fields*/  
    spinlock_t      lock;                操作此结构时需要得到的自旋锁  
    unsigned long    free_pages;          剩余的空闲页总数  
    unsigned long    pages_min, pages_low, pages_high;  zone 中空闲页的阈值, 详细见下  
    int              need_balance;        告诉 kswapd 需要对该 zone 的页进行交换  
    /** free areas of different sizes*/  
    free_area_t free_area[MAX_ORDER];     根据连续页大小分组的空闲页链表组, 连续  
    页大小分为 2^0, 2^1, 2^2, ... 2^MAX_ORDER 个连续页  
    /** Discontig memory support fields.*/  
    struct pglist_data *zone_pgdat;       父管理结构, 见上  
    struct page      *zone_mem_map;       当前 zone 的 mem_map, 即全局 mem_map 中  
    该 zone 所引用的第一页位置  
    unsigned long    zone_start_paddr;    zone 开始的物理地址(包括此地址)  
    unsigned long    zone_start_mapnr;    在全局 mem_map 中的索引 (或下标)  
    /** rarely used fields:*/  
    char             *name;               zone 名字, “DMA”, “Normal”或“HighMem”  
    unsigned long     size;               zone 的大小, 以页为单位  
} zone_t;
```

当 free_pages 达到 pages_min 时, buddy 分配器将采用同步方式进行 kswapd 的工作;

当 free_pages 达到 pages_low 时, kswapd 被 buddy 分配器唤醒, 开始释放页;

当 free_pages 达到 pages_high 时, kswapd 将被唤醒, 此时 kswapd 不会考虑如何平衡该 zone, 直到有 pages_high 空闲页为止。

```
typedef struct page {  
    struct list_head list;               通过此结构挂到空闲队列/干净缓冲队列/脏缓冲  
    队列等  
    struct address_space *mapping;       /* The inode (or ...) we belong to. */  
    unsigned long index;                PFN, page frame number, 在页索引数组中的  
    index  
    struct page *next_hash;             指向页高速缓存哈希表中下一个共享的页  
    atomic_t count;                    页引用计数  
    unsigned long flags;               一套描述页状态的一套位标志. 这些包括  
    PG_locked, 它指示该页在内存中已被加锁, 以及 PG_reserved, 它防止内存管理系统使用该  
    页.  
    struct list_head lru;               /* Pageout list, eg. active_list; protected by  
    pagemap_lru_lock !! */  
    wait_queue_head_t wait;            等待这一页的页队列  
    struct page **pprev_hash;          与 next_hash 相对应  
    struct buffer_head * buffers;      把缓冲区映射到一个磁盘块  
    void *virtual;                    被映射成的虚拟地址(Kernel virtual address,  
    NULL if not kmapped, ie. highmem)  
    struct zone_struct *zone;          属于哪个管理区, 其结构见上  
} mem_map_t;
```

从系统对内存的需求的角度来看, 又有如下结构

```
struct vm_area_struct {
    struct mm_struct * vm_mm;    指向父 mm(用户进程 mm)
    unsigned long vm_start;      当前 area 开始的地址
    unsigned long vm_end;        当前 area 结束的地址
    struct vm_area_struct *vm_next; 指向同一个 mm 中的下一个 area
    pgprot_t vm_page_prot;       当前 area 的存取权限
    unsigned long vm_flags;       描述这个区的一套标志. 详细见下
    rb_node_t vm_rb;
    struct vm_area_struct *vm_next_share;
    struct vm_area_struct **vm_pprev_share;
    struct vm_operations_struct * vm_ops; 一套内核可用来操作此区间的函数, 详细见 mmap

    unsigned long vm_pgoff;       若与文件关联, 则为在文件中的偏移, 以 PAGE_SIZE 为单
    位
    struct file * vm_file;        如果当前 area 与一个文件关联, 则指向文件的 struct file 结构
    unsigned long vm_raend;
    void * vm_private_data;
};
```

vm_flags: 对驱动编写者最感兴趣的标志是 VM_IO 和 VM_RESERVED.

VM_IO 标志一个 VMA 作为内存映射的 I/O 区. VM_IO 标志阻止这个区被包含在进程核转储中.

VM_RESERVED 告知内存管理系统不要试图交换出这个 VMA; 它应当在大部分设备映射中设置.

物理地址与页的关系:

页大小: PAGE_SIZE

物理地址(PA)-->页帧号(PFN, 此页在页索引数组中的 index): PA 右移 PAGE_SHIFT 位得到 PFN

```
struct page *virt_to_page(void *kaddr);
```

内核逻辑地址转换成 struct page 指针. 注意: 需要一个逻辑地址, 不能使用来自 vmalloc 的内存或者 high memory

```
struct page *pfn_to_page(int pfn);
```

页帧号(PFN, page frame number)转换成 struct page 指针

```
void *page_address(struct page *page);
```

返回这个页的内核虚拟地址, 如果这样一个地址存在.

对于高内存, 那个地址仅当这个页已被映射才存在. 大部分情况下, 使用 kmap 的一个版本而不是 page_address.

```
#include <linux/highmem.h>
```

```
void *kmap(struct page *page);
```

```
void kunmap(struct page *page);
```

kmap 为系统中的任何页返回一个内核虚拟地址.

对于低内存页, 它只返回页的逻辑地址;

对于高内存, kmap 在内核地址空间的一个专用部分中创建一个特殊的映射.

使用 `kmap` 创建的映射应当一直使用 `kunmap` 来释放. 映射的总数是有限的, 所以不再使用时需要及时 `unmap`

`kmap` 调用维护一个计数器, 因此如果 2 个或多个函数都在同一个页上调用 `kmap`, 操作也是正常的

注意: 当没有映射可用时, `kmap` 可能睡眠.

```
#include <linux/highmem.h>
```

```
#include <asm/kmap_types.h>
```

```
void *kmap_atomic(struct page *page, enum km_type type);
```

```
void kunmap_atomic(void *addr, enum km_type type);
```

`kmap_atomic` 是 `kmap` 的一种高性能形式. 每个体系都给原子的 `kmaps` 维护一个槽(专用的页表项); `kmap_atomic` 的调用者必须在 `type` 参数中指定哪个槽. 对驱动有意义的唯一插口是 `KM_USER0` 和 `KM_USER1`(对于直接来自用户空间的调用运行的代码), 以及 `KM_IRQ0` 和 `KM_IRQ1`(对于中断处理).

注意带 `atomic` 的 `kmaps` 必须是原子的

```
void *kmallocc(size_t size, int flags);
```

在内核里分配一块内存

`size`: 分配的块的大小. 内核只能分配某些预定义的, 固定大小的字节数组. `kmallocc` 能够处理的最小分配是 32 或者 64 字节. `kmallocc` 能够分配上限 128 KB

`flags`: 分配的标志. "GFP"是 `_get_free_page` 的缩写

`GFP_KERNEL`:表示在进程上下文中调用. 可能会导致睡眠, 所以所在函数必须是可重入的且不能在原子上下文中使用

`GFP_ATOMIC`:用来从中断处理和进程上下文之外的其他代码中分配内存.

`GFP_USER`:用来为用户空间页来分配内存; 它可能睡眠.

`GFP_HIGHUSER`:如同 `GFP_USER`, 但是从高端内存分配

`GFP_NOIO`

`GFP_NOFS`

同 `GFP_KERNEL`, 但是它们有更多限制:

`GFP_NOFS`:不允许进行任何文件系统调用;

`GFP_NOIO`:不允许任何 I/O 初始化.

它们主要地用在文件系统和虚拟内存代码, 那里允许一个分配睡眠, 但是不允许递归的文件系统调用

上面的分配标志可以与下列标志相或来作为参数, 下面的标志改变分配区域:

`__GFP_DMA`:这个标志要求分配在能够 DMA 的内存区. 确切的含义是平台依赖的.

`__GFP_HIGHMEM`:这个标志指示分配的内存可以位于高端内存. 确切的含义是平台依赖的.

`__GFP_COLD`:

这个标志请求一个"冷缓冲"页. (因为正常地, 内存分配器尽力返回"热缓冲"的页.)

它对分配页作 DMA 读是有用的, 因为此时在处理器缓冲中出现的数据是无用的.

`__GFP_`

如何编写 Linux 设备驱动程序

2007-10-29 周一, 10:48

有很多朋友关心驱动, 但更菜的菜鸟居多:) 总结了一下 把 I/O 驱动改成个更简单的 LED 驱动吧

做的工作非常简单，就是让连在 GPC0-GPC2 上的 LED 顺序闪 10 下
目的就是演示一下驱动过程。

一 先补充一下基础知识 懂的朋友就不用看了

嵌入式驱动的概念

设备驱动程序是操作系统内核和机器硬件之间的接口，设备驱动程序为应用程序屏蔽了硬件的细节，这样在应用程序看来，硬件设备只是一个设备文件，应用程序可以像操作普通文件一样对硬件设备进行操作。设备驱动程序是内核的一部分，它主要完成的功能有：对设备进行初始化和释放；把数据从内核传送到硬件和从硬件读取数据；读取应用程序传送给设备文件的数据、回送应用程序请求的数据以及检测和处理设备出现的错误。

Linux 将设备分为最基本的两大类：一类是字符设备，另一类是块设备。字符设备和块设备的主要区别是：在对字符设备发出读/写请求时，实际的硬件 I/O 一般就紧接着发生了。字符设备以单个字节为单位进行顺序读写操作，通常不使用缓冲技术；块设备则是以固定大小的数据块进行存储和读写的，如硬盘、软盘等，并利用一块系统内存作为缓冲区。为提高效率，系统对于块设备的读写提供了缓存机制，由于涉及缓冲区管理、调度和同步等问题，实现起来比字符设备复杂得多。LCD 是以字符设备方式加以访问和管理的，Linux 把显示驱动看做字符设备，把要显示的数据一字节一字节地送往 LCD 驱动器。

Linux 的设备管理是和文件系统紧密结合的，各种设备都以文件的形式存放在 /dev 目录下，称为设备文件。应用程序可以打开、关闭和读写这些设备文件，完成对设备的操作，就像操作普通的数据文件一样。为了管理这些设备，系统为设备编了号，每个设备号又分为主设备号和次设备号。主设备号用来区分不同种类的设备，而次设备号用来区分同一类型的多个设备。对于常用设备，Linux 有约定俗成的编号，如硬盘的主设备号是 3。Linux 为所有的设备文件都提供了统一的操作函数接口，方法是使用数据结构 struct file_operations。这个数据结构中包括许多操作函数的指针，如 open()、close()、read()和 write()等，但由于外设的种类较多，操作方式各不相同。Struct file_operations 结构体中的成员为一系列的接口函数，如用于读/写的 read/write 函数和用于控制的 ioctl 等。打开一个文件就是调用这个文件 file_operations 中的 open 操作。不同类型的文件有不同的 file_operations 成员函数，如普通的磁盘数据文件，接口函数完成磁盘数据块读写操作；而对于各种设备文件，则最终调用各自驱动程序中的 I/O 函数进行具体设备的操作。这样，应用程序根本不必考虑操作的是设备还是普通文件，可一律当作文件处理，具有非常清晰统一的 I/O 接口。所以 file_operations 是文件层次的 I/O 接口。

二 开始写了

采用了在代码里加注释的方法，同时把几个文件上传了一下，喜欢的朋友可以下载当作模板。
每个文件以==隔开

一共需要写 3 个文件，1 个驱动头文件，1 个驱动文件，一个驱动测试用程序文件
分别是 test.h, test.c 和 ledtest.c

简单说说驱动都做什么，怎么做

- 1 系统加载驱动
- 2 应用程序里打开设备（文件）
- 3 应用程序对设备操作
- 4 应用程序关闭设备（文件）
- 5 系统关闭设备

应用程序如何对设备操作？

记得 C 语言里怎么写文件吗？这里很相象的。对于一般的字符设备（还有块设备，网络设备等等）主要有 3 个函数（还有很多，可以看）llseek read: write: ioctl: 这里只用 ioctl:控制函数，当然也可以使用读写函数操作 IO 口，但 ioctl:似乎更适合。

具体实现可以看 ledtest.c 文件了。

test.c 中主要有几个函数 分别负责初始化和清除，打开和关闭。以及 ioctl 对串口寄存器写一些数据。

初始化和清除，打开和关闭函数里都各有一句主要句，已经分别作了注释。只要记住就好了。

对寄存器操作就不单独说了，需要看 44B0 数据手册了。好了 剩下的看代码吧。

```
=====
===== test.h =====
=====

/*****Copyright
(c) *****/
** FREE
**
**-----File Info-----
** File Name: config.h
** Last modified Date: 2006-9-9
** Last Version: 1.0
** Descriptions: User Configurable File
**
**-----
** Created By: ZLG CHENMINGJI
** Created date: 2006-9-9
** Version: 1.0
** Descriptions: First version
**
**-----
** Modified by:MAMAJINCO
** Modified date:2006-9-9
** Version:1.0
** Descriptions:在此忠心感谢 ZLG 的模版 我的高质量编程意识起源于此
**
*****/
//防止重复包含此文件而设置的宏
#ifndef __CONFIG_H
#define __CONFIG_H

//包含必要的头文件
#include <linux/config.h>
```

```

#include <linux/module.h> //模块必须包含的头文件
#include <linux/kernel.h> /* printk()函数，用于向内核输出信息 */
#include <linux/fs.h> /* 很重要 其中包含了如 file_opration 等结构 此结构用于文件层接口
*/
#include <linux/errno.h> /* 错误代码*/
#include <asm/uaccess.h>
#include <linux/types.h>
#include <linux/mm.h>
#include <asm/arch/s3c44b0x.h>

/*****
/* 应用程序配置 */
*****/
//以下根据需要改动
//定义主设备号 设备名称
#define LED_MAJOR_NR 231 //231~239 240~255 都可以
#define DEVICE_NAME "led" /* name for messaging */
#define SET_LED_OFF 0
#define SET_LED_ON 1
#endif
/***** End Of File
*****/

=====END=====

=====test.c=====

*****/

*****Copyright (c)*****

** FREE
**
**-----File Info-----
** File Name: test.c
** Last modified Date: 2006-9-9
** Last Version: 1.0
** Descriptions: User Configurable File
**
**-----
** Created By: ZLG CHENMINGJI
** Created date: 2006-9-9
** Version: 1.0
** Descriptions: First version
**
**-----
** Modified by:MAMAJINCO
** Modified date:2006-9-9

```

```

** Version:1.0
** Descriptions:在此忠心感谢 ZLG 的模版 我的高质量编程意识起源于此
**

*****/

#include "test.h"//包含驱动头文件

/*****

function announce
*****/

//以下是关键函数的声明
static int led_open(struct inode *inode, struct file *filp);
//打开设备时用的 linux 把设备当作文件管理 设备最好在用的时候再打开 尽量不要提前
static int led_release(struct inode *inode, struct file *filp);
static int led_ioctl(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long param);
//控制函数 这里用于控制 LED 亮与灭
int led_init(void);//注册时用的 注意 模块注册的越早越好
void led_cleanup(void);//卸载时用的

/*****

** "全局和静态变量在这里定义"
** global variables and static variables define here
*****/

static struct file_operations LED_fops = /* 前面基础部分提到的重要结构体了*/
{
owner: THIS_MODULE,
#if 0/*注意: #if 0-#endif 里的代码是不编译的, 但这里为了驱动模板讲解上的完整性仍然加上了*/
llseek: gpio_llseek,
read: gpio_read,
write: gpio_write,
#endif
ioctl: led_ioctl,//控制函数
open: led_open, //打开函数, 打开文件时做初始化的
release: led_release,//释放函数, 关闭时调用
};

/*****

** Function name: led_open
** Descriptions: open device
** Input:inode: information of device
** filp: pointer of file
** Output 0: OK
** other: not OK
** Created by: Chenmingji
** Created Date: 2006-9-9

```

```

**-----
** Modified by:mamajinco
** Modified Date: 2006-9-9
**-----
*****/
static int led_open(struct inode *inode, struct file *filp)
{
/*初始化放在 OPEN 里*/
*(volatile unsigned *)S3C44B0X_PCONC) &= 0xfffffc0;
*(volatile unsigned *)S3C44B0X_PCONC) |= 0xfffffd5; /*GPIO C 口 0~2 设置为输出*/

*(volatile unsigned *)S3C44B0X_PUPC) &= 0xfffffc0; /*GPIO C 口 0~2 设置为上拉*/

MOD_INC_USE_COUNT;
return 0;
}

*****/
** Function name: led_release
** Descriptions: release device
** Input:inode: information of device
** filp: pointer of file
** Output 0: OK
** other: not OK
** Created by: Chenmingji
** Created Date: 2006-9-9
**-----
** Modified by: mamajinco
** Modified Date: 2006-9-9
**-----
*****/
static int led_release(struct inode *inode, struct file *filp)
{
MOD_DEC_USE_COUNT;
return(0);
}

/*****
** Function name: led_ioctl
** Descriptions: IO control function
** Input:inode: information of device
** filp: pointer of file
** cmd: command
** arg: additive parameter

```

```

** Output 0: OK
** other: not OK
** Created by: Chenmingji
** Created Date: 2006-9-9
**-----
** Modified by: mamajinco
** Modified Date: 2006-9-9
**-----
*****/

static int led_ioctl(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg)
{
if( arg > 2 )//判断 IO 是否属于 0-2
return -1;

switch(cmd)
{
case 0://把 ARG 传来的 IO 口编号转换成寄存器数据，把相应 IO 口置低
(*(volatile unsigned *)S3C44B0X_PDATC) |= 0x1 << arg;

break;

case 1://把 ARG 传来的 IO 口编号转换成寄存器数据，把相应 IO 口置高
(*(volatile unsigned *)S3C44B0X_PDATC) &= 0x0 << arg;
break;

default:
return -1;
break;
}
return 0;
}

/*****
** Function name: led_init
** Descriptions: init driver
** Input:none
** Output 0: OK
** other: not OK
** Created by: Chenmingji
** Created Date: 2006-9-9
**-----
** Modified by: mamajinco
** Modified Date: 2006-9-9
**-----

```

```

*****/
int led_init(void)
{
int result;

result = register_chrdev(231,"led", &LED_fops);
/*关键语句 用于注册
** 注意!这是传统的注册方法 在 2.4 以上的 linux 版本中 加入了 devfs 设备文件系统 使注册更容易 但为了与大部分资料相同 大家看的方便 这里仍然使用老方法*/
if (result < 0) //用于异常检测
{
printk(KERN_ERR DEVICE_NAME ": Unable to get major %d\n", LED_MAJOR_NR ); //printk
用于向内核输出信息
return(result);
}

printk(KERN_INFO DEVICE_NAME ": init OK\n");

return(0);
}

/*****
** Function name: led_cleanup
** Descriptions: exit driver
** Input:none
** Output none
** Created by: Chenmingji
** Created Date: 2006-9-9
**-----
** Modified by: mamajinco
** Modified Date: 2006-9-9
**-----
*****/

void led_cleanup(void)
{
unregister_chrdev(231, "led"); //与 register_chrdev 配对使用 用于清楚驱动
}

/*****
** End Of File
*****/

```

```

=====END=====
=====
=====ledtest.c=====
=====

/*****Copyright (c)*****/
** FREE
**
**-----File Info-----
** File Name: led.c
** Last modified Date: 2006-9-9
** Last Version: 1.0
** Descriptions: User Configurable File
**
**-----
** Created By: ZLG CHENMINGJI
** Created date: 2006-9-9
** Version: 1.0
** Descriptions: First version
**
**-----
** Modified by:MAMAJINCO
** Modified date:2006-9-9
** Version:1.0
** Descriptions:在此忠心感谢 ZLG 的模版 我的高质量编程意识起源于此
**
*****/

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

void delay(int delay)//延时用函数
{
int i;
for(;delay>0;delay--)
{
for(i=0 ; i < 5000 ; i ++);

}
}

```

```

int main()
{
int fd1;
int j;

fd1= open("/dev/led", O_RDWR);/*打开设备，就象打开文件一样简单*/

if(fd1 == -1)/*异常处理*/
{
printf ( "file can not be open" );
return -1;
}

for (j =0 ;j< 10 ;j ++)*重复 10 次*/
{
ioctl(fd1 , 1 , 0);/*GPC0 上 LED 亮*/
delay(1000);
ioctl(fd1 , 0 , 0);/*GPC0 上 LED 灭*/
ioctl(fd1 , 1 , 1);/*GPC1 上 LED 亮*/
delay(1000);
ioctl(fd1 , 0 , 1);/*GPC1 上 LED 灭*/
ioctl(fd1 , 1 , 2);/*GPC2 上 LED 亮*/
delay(1000);
ioctl(fd1 , 0 , 2);/*GPC2 上 LED 灭*/
delay(1000);
}

close (fd1);/*关闭设备（文件）*/
return 0;

}

```

=====END=====

三 驱动编译进内核

编译的中对于菜鸟来说需要需要注意几点

1 被打错字，包括上面的函数中也是！

就算各位扔砖头我也得说，因为编译进内核是很费时间的~~而且最重要的是对于菜鸟来说 make 的错误提示都是一道关，绝对不要自己给自己设置障碍！我们团队里就常有兄弟姐妹出现这样的错误，怎么看怎么对，尤其是从书上抄下来的命令和字符，l 和 I 还有 I 你怎么分？最后一个是大写的 i :)

2 不要用中文文件名 包括 ABC（复件）

要不然 MAKE 出错

3 在各个现成的文件里修改的时候按照原有的格式修改 要不然菜鸟很难保证不犯低级错误

让我想起了 IBM 的规律总结测试题：6 13 7 14 8 下一个数字是什么？

好了 开始修改！

```
=====START=====
uClinux-dist/linux-2.4.x/drivers/char/Makefile
-----
obj-$(CONFIG_C5471_WDT) += wdt_c5471.o 之后加
obj-$(CONFIG_TEST) += led.o
=====END=====

=====START=====
uClinux-dist/linux-2.4.x/drivers/char/Config.in
-----
if [ "$CONFIG_CPU_S3C44B0X" = "y" ]; then
bool 'Samsung S3C44B0X serial ports support' CONFIG_SERIAL_S3C44B0X 之后加
bool 'Test LED Driver' CONFIG_TEST
=====END=====

=====START=====
uClinux-dist/linux-2.4.x/drivers/char/mem.c
-----
开头的地方扎堆加
#ifdef CONFIG_LEDTEST
extern void led_init(void);
#endif

int __init chr_dev_init(void)之后加
#ifdef CONFIG_TEST
led_init();
#endif
=====END=====

=====START=====
uClinux-dist/vendors/Samsung/44B0/Makefile
-----
ttype,c,3,12 ttyd,c,3,13 ttype,c,3,14 ttyf,c,3,15\之后加
\
led,c,231,0 \
=====END=====
```

四 把程序编译进内核

没什么说的了，和过去写的简单的程序一样加 但这里再重复一次

```
=====START=====
```

uClinux-dist/user/Makefile

```
-----  
扎堆加个下面  
dir_$(CONFIG_USER_LEDTEST) += LEDtest  
=====END=====
```

uClinux-dist/config/Configure.help

```
-----  
扎堆加个下面  
CONFIG_USER_LEDTEST  
Test the LED driver  
=====END=====
```

uClinux-dist/config/Configure.in

```
-----  
#####  
mainmenu_option next_comment  
comment 'LED driver test PG'  
  
bool 'LEDtest' CONFIG_USER_LEDTEST  
endmenu  
  
#####  
=====END=====
```

五 编译 烧写.....省略 200 字 想看的看我写的 helloworld 编译笔记吧

六 下面的操作在板子上执行

```
1 cd /dev  
2 ls  
看见里面有个 LED 了吧?  
3 cd /proc  
4 cat devices  
看见驱动列表吧?  
led 231 也应该在里面  
5 LEDtest  
在任何地方执行这个语句 就可以  
之后看 GPIO 的 C 口电平吧:)
```

Linux 设备驱动程序设计实例

Linux 系统中，设备驱动程序是操作系统内核的重要组成部分，在 与硬件设备之间建立了标准的抽象接口。通过这个接口，用户可以像处理普通文件一样，对硬件设备进行打开(open)、关闭(close)、读写(read/write)等操作。通过分析和设计设备驱动程序，可以深入理解 Linux 系统和进行系统开发。本文通过一个简单的例子来说明设备驱动程序的设计。

1、 程序清单

```

//          MyDev.c          2000 年 2 月 7 日编写
#ifndef __KERNEL__
#  define __KERNEL__      //按内核模块编译
#endif
#ifndef MODULE
#  define MODULE          //设备驱动程序模块编译
#endif
#define DEVICE_NAME "MyDev"
#define OPENSPP 1
#define CLOSESPK 2
//必要的头文件
#include <linux/module.h> //同 kernel.h,最基本的内核模块头文件
#include <linux/kernel.h> //同 module.h,最基本的内核模块头文件
#include <linux/sched.h> //这里包含了进行正确性检查的宏
#include <linux/fs.h> //文件系统所必需的头文件
#include <asm/uaccess.h> //这里包含了内核空间与用户空间进行数据交换时的
函数宏
#include <asm/io.h> //I/O 访问
int my_major=0; //主设备号
static int Device_Open=0;
static char Message[]="This is from device driver";
char *Message_Ptr;
int my_open(struct inode *inode, struct file *file)
{
    //每当应用程序用 open 打开设备时，此函数被调用
    printk("\ndevice_open(%p,%p)\n", inode, file);
    if (Device_Open)
        return -EBUSY; //同时只能由一个应用程序打开
    Device_Open++;
    MOD_INC_USE_COUNT; //设备打开期间禁止卸载
    return 0;
}
static void my_release(struct inode *inode, struct file *file)
{
    //每当应用程序用 close 关闭设备时，此函数被调用
    printk("\ndevice_release(%p,%p)\n", inode, file);
    Device_Open--;
    MOD_DEC_USE_COUNT; //引用计数减 1
}
ssize_t my_read (struct file *f,char *buf,int size,loff_t off)
{
    //每当应用程序用 read 访问设备时，此函数被调用
    int bytes_read=0;
#ifdef DEBUG
    printk("\nmy_read is called. User buffer is %p,size is %d\n",buf,size);
#endif
    if (verify_area(VERIFY_WRITE,buf,size)==-EFAULT)

```

```

return -EFAULT;
    Message_Ptr=Message;
    while(size && *Message_Ptr)
    {
        if(put_user(*(Message_Ptr++),buf++))    //写数据到用户空间
            return -EINVAL;
        size--;
        bytes_read++;
    }
    return bytes_read;
}

ssize_t my_write (struct file *f,const char *buf, int size,loff_t off)
{//每当应用程序用 write 访问设备时，此函数被调用
    int i;
    unsigned char uc;
#ifdef DEBUG
        printk("\nmy_write is called. User buffer is %p,size is %d\n",buf,size);
#endif
    if (verify_area(VERIFY_WRITE,buf,size)==-EFAULT)
        return -EFAULT;
        printk("\nData below is from user program:\n");
        for (i=0;i<size;i++)
            if(!get_user(uc,buf++)) //从用户空间读数据
                printk("%02x ",uc);

        return size;
}

int my_ioctl(struct inode *inode,struct file *f,unsigned int arg1,
unsigned int arg2)
{//每当应用程序用 ioctl 访问设备时，此函数被调用
#ifdef DEBUG
        printk("\nmy_ioctl is called. Parameter is %p,size is %d\n",arg1);
#endif
    switch (arg1)
    {
        case OPENSPK:
            printk("\nNow,open PC's speaker.\n");
            outb(inb(0x61)|3,0x61); //打开计算机的扬声器
            break;
        case CLOSESPK:
            printk("\nNow,close PC's speaker.");
            outb(inb(0x61)&0xfc,0x61); //关闭计算机的扬声器
            break;
    }
}

```

```

struct file_operations my_fops = {
    NULL,          /* lseek */
    my_read,
    my_write,
    NULL,
    NULL,
    my_ioctl,
    NULL,
    my_open,
    my_release,
    /* nothing more, fill with NULLs */
};

int init_module(void)
{
    //每当装配设备驱动程序时，系统自动调用此函数
    int result;
    result = register_chrdev(my_major, DEVICE_NAME, &my_fops);
    if (result < 0) return result;
    if (my_major == 0)
        my_major = result;
    printk("\nRegister Ok. major-number=%d\n", result);
    return 0;
}

void cleanup_module(void)
{
    //每当卸载设备驱动程序时，系统自动调用此函数
    printk("\nnunload\n");
    unregister_chrdev(my_major, DEVICE_NAME);
}

```

2、设备 驱动程序设计

Linux 设备分为字符设备、块设备和网络设备。字符设备是不需要缓冲而直接读写的设备，如串口、键盘、鼠标等，本例就是字符设备驱动程序；块设备的访问通常需要缓冲来支持，以数据块为单位来读写，如磁盘设备等；网络设备是通过套接字来访问的特殊设备。

1) 设备驱动程序和内核与应用程序的接口

无论哪种类型的设备，Linux 都是通过在内核中维护特殊的设备控制块来与设备驱动程序接口的。在字符设备和块设备的控制块中，有一个重要的数据结构 `file_operations`，该结构中包含了驱动程序提供给应用程序访问硬件设备的各种方法，其定义如下（参见 `fs.h`）：

```

struct file_operations {
    loff_t (*llseek) (struct file *, loff_t, int);          //响应应用程序中 lseek 调
    用的函数指针
    ssize_t (*read) (struct file *, char *, size_t, loff_t *);
    //响应应用程序中 read 调用的函数指针

```

```

        ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
//响应应用程序中 write 调用的函数指针
        int (*readdir) (struct file *, void *, filldir_t); //响应应用程序中
readdir 调用的函数指针
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
//响应应用程序中 select 调用的函数指针
        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
//响应应用程序中 ioctl 调用的函数指针
        int (*mmap) (struct file *, struct vm_area_struct *);
//响应应用程序中 mmap 调用的函数指针
        int (*open) (struct inode *, struct file *); //响应应用程序中 open 调
用的函数指针
        int (*flush) (struct file *);
        int (*release) (struct inode *, struct file *); //响应应用程序中 close
调用的函数指针
        int (*fsync) (struct file *, struct dentry *);
        int (*fasync) (int, struct file *, int);
        int (*check_media_change) (kdev_t dev);
        int (*revalidate) (kdev_t dev);
        int (*lock) (struct file *, int, struct file_lock *);
};

```

多数情况下，只需为上面结构中的少数方法编写服务函数，其他均设为 NULL 即可。每一个可装配的设备驱动程序都必须有 `init_module` 和 `cleanup_module` 两个函数，装载和卸载设备时内核自动调用这两个函数。在 `init_module` 中，除了可以对硬件设备进行检查和初始化外，还必须调用 `register_*` 函数将设备登记到系统中。本例中是通过 `register_chrdev` 来登记的，如果是块设备或网络设备则应该用 `register_blkdev` 和 `register_netdev` 来登记。`Register_chrdev` 的主要功能是将设备名和结构 `file_operations` 登记到系统的设备控制块中。

2) 与应用程序的数据交换

由于设备驱动程序工作在内核存储空间，不能简单地用 "="、"`memcpy`"等方法与应用程序交换数据。在头文件 `uaccess.h` 中定义了方法 `put_user(x, ptr)` 和 `get_user(x, ptr)`，用于内核空间与用户空间的数据交换。值 `x` 的类型根据指针 `ptr` 的类型确定，请参见源代码中的 `my_read` 与 `my_write` 函数。

3) 与硬件设备的接口

Linux 中为设备驱动程序访问 I/O 端口、硬件中断和 DMA 提供了简便方法，相应的头文件分别为 `io.h`、`irq.h`、`dma.h`。由于篇幅限制，本例中只涉及到 I/O 端口访问。

Linux 提供的 I/O 端口访问方法主要有：`inb()`、`inw()`、`outb()`、`outw()`、`inb_p()`、`inw_p()`、`outb_p()`、`outw_p()`等。要注意的是，设备驱动程序在使用端口前，应该先用 `check_region()` 检查该端口的占用情况，如果指定的端口可用，则再用 `request_region()` 向系统登记。说明 `check_region()`、`request_region()` 的头文件是 `ioport.h`。

4) 内存分配

设备驱动程序作为内核的一部分，不能使用虚拟内存，必须利用内核提供的 `kmalloc()` 与 `kfree()` 来申请和释放内核存储空间。`Kmalloc()` 带两个参数，第一个要申请的是内存数量，在早期的版本中，这个数量必须是 2 的整数幂，如 128、256。关于 `kmalloc()` 与 `kfree()` 的用法，可参考内核源程序中的 `malloc.h` 与 `slab.c` 程序。

3、程序的编译和访问

本例在 Linux 2.2.x.x 中可以成功编译和运行。先用下面的命令行进行编译：

```
gcc -Wall -O2 -c MyDev.c
```

该命令行中参数-Wall 告诉编译程序显示警告信息；参数-O2 是关于代码优化的设置，注意内核模块必须优化；参数-c 规定只进行编译和汇编，不进行连接。

正确编译后，形成 MyDev.o 文件。可以输入命令 insmod MyDev.o 来装载此程序。如果装配成功，则显示信息：Register Ok.major-number=xx。利用命令 lsmod 可以看到该模块被装配到系统中。

为了访问该模块，应该用命令 mknode 来创建设备文件。下面的应用程序可以对创建的设备文件进行访问。

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#define DEVICE_NAME MyDev
#define OPENS PK 1
#define CLOSESPK 2
char buf[128];
int main(){
    int f=open(DEVICE_NAME,O_RDWR);
    if (f== -1) return 1;
    printf("\nHit enter key to read device...");
    read(f,buf,128); printf(buf);
    printf("\nHit enter key to write device ...");
    write(f,"test",4);
    printf("\nHit enter key to open PC's speaker...");
    ioctl(f,OPENS PK);
    printf("\nHit enter key to close PC's speaker...");
    ioctl(f,CLOSESPK);
    close(f);
}
```