第 5 章 字符设备驱动程序的扩展操作

在关于字符设备驱动程序的那一章中，我们构建了一个完整的设备驱动程序，从中用户可以读也可以写。但实际一个驱动程序通常会提供比同步 read 和 write 更多的功能。现在如果出了什么毛病，我已经配备了调试工具，我们可以大胆的实验并实现新操作。

通过补充设备读写操作的功能之一就是控制硬件，最常用的通过设备驱动程序完成控制动作的方法就是实现 ioctl 方法。另一种方法是检查写到设备中的数据流，使用特殊序列做为控制命令。尽管有时也使用后者，但应该尽量避免这样使用。不过稍后我们还是会在本章的“非 ioctl 设备控制”一节中介绍这项技术。

正如我在前一章中所猜想的，ioctl 系统调用为驱动程序执行“命令”提供了一个设备相关的入口点。与 read 和其他方法不同，ioctl 是设备相关的，它允许应用程序访问被驱动硬件的特殊功能－－配置设备以及进入或退出操作模式。这些“控制操作”通常无法通过 read/write 文件操作完成。例如，你向串口写的所有数据都通过串口发送出去了，你无法通过写设备改变波特率。这就是 ioctl 所要做的：控制 I/O 通道。

实际设备（与 scull 不同）的另一个重要功能是，读或写的数据需要同其他硬件交互，需要某些同步机制。阻塞型 I/O 和异步触发的概念将满足这些需求，本章将通过一个改写的 scull 设备介绍这些内容。驱动程序利用不同进程间的交互产生异步事件。与最初的 scull 相同，你无需特殊硬件来测试驱动程序是否可以工作。直到第 8 章“硬件管理”我才会真正去与硬件打交道。

ioctl

在用户空间内调用 ioctl 函数的原型大致如下：

（代码）

由于使用了一连串的“.”的缘故，该原型在 Unix 系统调用列表之中非常突出，这些点代表可变数目参数。但是在实际系统中，系统调用实际上不会有可变数目个参数。因为用户程序只能通过第 2 章“编写和运行模块”的“用户空间和内核空间”一节中介绍的硬件“门”才能访问内核，系统调用必须有精确定义的参数个数。因此，ioctl 的第 3 个参数事实上只是一个可选参数，这里用点只是为了在编译时防止编译器进行类型检查。第 3 个参数的具体情况与要完成的控制命令（第 2 个参数）有关。某些命令不需要参数，某些需要一个整数做参数，而某些则需要一个指针做参数。使用指针通常是可以用来向 ioctl 传递任意数目数据；设备可以从用户空间接收任意大小的数据。

系统调用的参数根据方法的声明传递给驱动程序方法：

（代码）

inode 和 filp 指针是根据应用程序传递的文件描述符 fd 计算而得的，与 read 和 write 的用法一致。参数 cmd 不经修改地传递给驱动程序，可选的 arg 参数无论是指针还是整数值，它都以 unsigned long 的形式传递给驱动程序。如果调用程序没有传递第 3 个参数，驱动程序所接收的 arg 没有任何意义。

由于附加参数的类型检查被关闭了，如果非法参数传递给 ioctl，编译器无法向你报警，程序员在运行前是无法注意这个错误的。这是我所见到的 ioctl 语义方面的唯一一个问题。

如你所想，大多数 ioctl 实现都包括一个 switch 语句来根据 cmd 参数选择正确的操作。不同

的命令对应不同的数值，为了简化代码我们通常会使用符号名代替数值。这些符号名都是在预处理中赋值的。不同的驱动程序通常会在它们的头文件中声明这些符号；scull 就在 scull.h 中声明了这些符号。

选择 ioctl 命令

在编写 ioctl 代码之前，你需要选择对应不同命令的命令号。遗憾的是，简单地从 1 开始选择号码是不能奏效的。

为了防止对错误的设备使用正确的命令，命令号应该在系统范围内是唯一的。这种失配并不是不很容易发生，程序可能发现自己正在对象 FIFO 和 kmouse 这类非串口输入流修改波特率。如果每一个 ioctl 命令都是唯一的，应用程序就会获得一个 EINVAL 错误，而不是无意间成功地完成了操作。

为了达到唯一性的目的，每一个命令号都应该由多个位字段组成。Linux 的第一版使用了一个 16 位整数：高 8 位是与设备相关的“幻”数，低 8 位是一个序列号码，在设备内是唯一的。这是因为，用 Linus 的话说，他有点“无头绪”，后来才接收了一个更好的位字段分割方案。遗憾的是，很少有驱动程序使用新的约定，这就挫伤了程序员使用新约定的热情。在我的源码中，为了发掘这种约定都提供了那些功能，同时防止被其他开发人员当成异教徒而禁止，我使用了新的定义命令的方法。

为了给我的驱动程序选择 ioctl 号，你应该首先看看 include/asm/ioctl.h 和 Documentation/ioctl-number.txt 这两个文件。头文件定义了位字段：类型（幻数），基数，传送方向，参数的尺寸等等。ioctl-number.txt 文件中罗列了在内核中使用的幻数。这个文件的新版本（2.0 以及后继内核）也给出了为什么应该使用这个约定的原因。

很不幸，在 1.2.x 中发行的头文件没有给出切分 ioctl 位字段宏的全集。如果你需要象我的 scull 一样使用这种新方法，同时还要保持向后兼容性，你使用 scull/sysdep.h 中的若干代码行，我在那里给出了解决问题的文档的代码。

现在已经不赞成使用的选择 ioctl 号码的旧方法非常简单：选择一个 8 位幻数，比如“k”（十六进制为 0x6b），然后加上一个基数，就象这样：

（代码）

如果应用程序和驱动程序都使用了相同的号码，你只要在驱动程序里实现 switch 语句就可以了。但是，这种在传统 Unix 中有基础的定义 ioctl 号码的方法，不应该再在新约定中使用。这里我介绍就方法只是想给你看看一个 ioctl 号码大致是个什么样子的。

新的定义号码的方法使用了 4 个位字段，它们有如下意义。下面我所介绍的新符号都定义在<linux/ioctl.h>中。

类型

幻数。选择一个号码，并在整个驱动程序中使用这个号码。这个字段有 8 位宽（_IOC_TYPEBITS）。

号码

基（序列）数。它也是 8 位宽（_IOC_NRBITS）。

方向

如果该命令有数据传输，它定义数据传输的方向。可以使用的值有，_IOC_NONE（没有数据传输），_IOC_READ，_IOC_WRITE 和_IOC_READ | _IOC_WRITE（双向传输数据）。数据传输是从应用程序的角度看的；IOC_READ 意味着从设备中读数据，驱动程序必须向用户空间写数据。注意，该字段是一个位屏蔽码，因此可以用逻辑 AND 操作从中分解出_IOC_READ 和_IOC_WRITE。

尺寸
所涉及的数据大小。这个字段的宽度与体系结构有关，当前的范围从 8 位到 14 位不等。你可以在宏_IOC_SIZEBITS 中找到某种体系结构的具体数值。不过，如果你想要你的驱动程序可移植，你只能认为最大尺寸可达 255 个字节。系统并不强制你使用这个字段。如果你需要更大尺度的数据传输，你可以忽略这个字段。下面我们将介绍如何使用这个字段。
包含在<linux/ioctl.h>之中的头文件<asm/ioctl.h>定义了可以用来构造命令号码的宏：_IO(type,nr)，_IOR(type,nr,size)，_IOW(type,nr,size)和 IOWR(type,nr,size)。每一个宏都对应一种可能的数据传输方向，其他字段通过参数传递。头文件还定义了解码宏：_IOC_DIR(nr)，_IOC_TYPE(nr)，_IOC_NR(nr)和_IOC_SIZE(nr)。我不打算详细介绍这些宏，头文件里的定义已经足够清楚了，本节稍后会给出样例。
这里是 scull 中如果定义 ioctl 命令的。特别地，这些命令设置并获取驱动程序的配置参数。在标准的宏定义中，要传送的数据项的尺寸有数据项自身的实例代表，而不是 sizeof(item)，这是因为 sizeof 是宏扩展后的一部分。
（代码）
最后一条命令，HARDRESET，用来将模块使用计数器复位为 0，这样就可以在计数器发生错误时就可以卸载模块了。实际的源码定义了从 IOCHQSET 到 HARDRESET 间的所有命令，但这里没有列出。
我选择用两种方法实现整数参数传递——通过指针和显式数值，尽管根据已有的约定，ioctl 应该使用指针完成数据交换。同样，这两种方法还用于返回整数：通过指针和设置返回值。如果返回值是正的，这就可以工作；对与任何一个系统调用的返回值，正值是受保护的（如我们在 read 和 write 所见到的），而负值则被认为是一个错误值，用其设置用户空间中的 errno 变量。
“交换”和“移位”操作并不专用于scull设备。我实现“交换”操作是为了给出“方向”字段的所有可能值，而“移位”操作则将“告知”和“查询”操作组合在一起。某些时候是需要原子性*测试兼设置这类操作的——特别是当应用程序需要加锁和解锁时。
显式的命令基数没有什么特殊意义。它只是用来区分命令的。事实上，由于 ioctl 号码的“方向”为会有所不同，你甚至可以在读命令和写命令中使用同一个基数。我选择除了在声明中使用基数外，其他地方都不使用它，这样我就不必为符号值赋值了。这也是为什么显式的号码出现在上面的定义中。我只是向你介绍一种使用命令号码的方法，你可以自由地采用不同的方法使用它。
当前，参数 cmd 的值内核并没有使用，而且以后也不可能使用。因此，如果你想偷懒，你可以省去上面那些复杂的声明，而直接显式地使用一组 16 位数值。但另一方面，如果你这样做了，你就无法从使用位字段中受益了。头文件<linux/kd.h>就是这种旧风格方法的例子，但是它们并不是因为偷懒才这样做的。修改这个文件需要重新编译许多应用程序。

返回值

ioctl 的实现通常就是根据命令号码的一个 switch 语句。但是，当命令号码不能匹配任何一个合法操作时，default 选择使用是什么？这个问题是很有争议性的。大多数内核函数返回-EINVAL（“非法参数”），这是由于命令参数确实不是一个合法的参数，这样做是合适的。然而，POSIX 标准上说，如果调用了一个不合适的 ioctl 命令，应该返回-ENOTTY。对应的

* 当一段程序代码总是被当做一条指令被执行，且不可能在期间发生其他操作，我们就称这段代码是原子性的。

消息是“不是终端”——这不是用户所期望的。你不得不决定是严格依从标准还是一般常识。我们将本章的后面介绍为什么依从 POSIX 标准需要返回 ENOTTY。

预定义命令

尽管 ioctl 系统调用大部分都用于操作设备，但还有一些命令是由内核识别的。注意，这些命令是在你自己的文件操作前调用的，所以如果你选择了和它们相同的命令号码，你将无法接收到那个命令的请求，而且由于 ioctl 命令的不唯一性，应用程序会请求一些未可知的请求。

预定义命令分为 3 组：用于任何文件（普通，设备，FIFO 和套接字文件）的，仅用于普通文件的以及和文件系统相关的；最后一组命令只能在宿主机文件系统上执行（间 chattr 命令）。设备驱动程序编写者仅对第 1 组感兴趣就可以了，它们的幻数是“T”。分析其他组的工作将留做读者的练习；ext2_ioctl 是其中最有意思的函数（尽管比你想象的要容易得多），它实现了只追加标志和不可变标志。

下列 ioctl 命令对任何文件都是预定义的：

FIOCLEX

设置 exec 时关闭标志（File IOctl Close on EXec）。

FIONCLEX

清除 exec 时关闭标志。

FIOASYNC

设置或复位文件的同步写。Linux 中没有实现同步写；但这个调用存在，这样请求同步写的应用程序就可以编译和运行了。如果你不知道同步写是怎么回事，你也不用费神去了解它了：你不需要它。

FIONBIO

“File IOctl Nonblocking I/O（文件 ioctl 非阻塞型 I/O）”（稍后在“阻塞型和非阻塞型操作”一节中介绍）。该调用修改 filp->f_flags 中的 O_NONBLOCK 标志。传递给系统调用的第 3 个参数用来表明该标志是设置还是清除。我们将在本章后面谈到它的作用。注意，fcntl 系统调用使用 F_SETFL 命令也可以修改这个标志。

列表中的最后一项引入了一个新系统调用 fcntl，它看起来和 ioctl 很象。事实上，fcntl 调用与 ioctl 非常相似，它也有一个命令参数和额外（可选的）一个参数。它和 ioctl 分开主要是由于历史原因：当 Unix 开发人员面对“控制”I/O 操作的问题时，他们决定文件和设备应该是不同的。那时，唯一的设备是终端，这也就解释了为什么-ENOTTY 是标准的非法 ioctl 命令的返回值。这个问题是是否保持向后兼容性的老问题。

使用 ioctl 参数

我们需要讲解的最后一点是，在分析 scull 驱动程序的 ioctl 代码前，首先弄明白如何使用那个额外的参数。如果它是一个整数就非常简单了：可以直接使用它。如果它是一个指针，就必须注意一些问题了。

当用一个指针引用用户空间时，我们首先要确保它指向了合法的用户空间，并且对应页面当前恰在映射中。如果内核代码试图访问范围之外的地址，处理器就会发出一个异常。内核代码中的异常将由上至 2.0.x 的内核转换为 oops 消息。设备驱动程序应该通过验证将要访问的用户地址空间的合法性来防止这种失效的发生，如果地址是非法的应该返回一个错误码。

Linux 2.1 中引入新功能之一就是内核代码的异常处理。遗憾的是，正确的实现需要驱动程序-内核接口的较大改动。本章给出的非法只适用于旧内核，从 1.2.13 到 2.0.x。新接口将在第 17 章“近期发展”的“处理内核空间失效”一节中介绍，那里给出的例子通过某些预处理宏将使支持的内核扩展到 2.1.43。

内核 1.x.y 和 2.0.x 的地址验证是通过函数 verify_area 实现的，它的原型定义在<linux/mm.h>中：

（代码）

第一个参数应该是 VERIFY_READ 或 VERIFY_WRITE，这取决于你要在内存区上完成读还是写操作。ptr 参数是一个用户空间地址，extent 是一个字节计数。例如，如果 ioctl 需要从用户空间读一个整数，extent 就是 sizeof(int)。如果在指定的地址上进行读和写操作，使用 VERIFY_WRITE，它是 VERIFY_READ 的超集。

验证读只检查地址是否是合法的：除此之外，验证写要好检查只读和 copy-on-write 页面。copy-on-write 页面一个共享可写页面，它还没有被任何共享进程写过；当你验证写时，verify_area 完成“复制兼完成可写配置”操作。很有意思的是，这里无需检查页面是否“在”内存中，这是由于合法页面将由失效函数正确地进行处理，甚至从内核代码中调用也可以。我们已经在第 3 章“字符设备”的“Scull 的内存使用”一节中看到内核代码可以成功地完成页面失效处理。

象大多数函数一样，verify_area 返回一个整数值：0 意味着成功，负值代表一个错误，应该将这个错误返回给调用者。

scull 源码在 switch 之前分析 ioctl 号码的各个位字段：

（代码）

在调用 verify_area 之后，再有驱动程序完成真正的数据传送。除了 memcpy_tofs 和 memcpy_fromfs 函数外，程序员还可以使用两个专为常用数据尺寸（1，2 和 4 个字节，在以及 64 位平台上的 8 个字节）优化的函数。这些函数定义在<asm/segment.h>中。

put_user(datum, ptr)

实际上它是一个最终调用__put_user 的宏；编译时将其扩展为一条机器指令。驱动程序应该尽可能使用 put_user，而不是 memcpy_tofs。由于在宏表达式中不进行类型检查，你可以传递给 put_user 任何类型的数据指针，不过它应该是一个用户空间地址。数据传输的尺寸依赖于 ptr 参数的类型，这是在编译时通过特殊的 gcc 伪函数实现的，这里没有介绍的必要。结果，如果 ptr 是一个字符指针，就传递 1 个字节，依此类推分别有 2，4 和 8 个字节。如果被指引的数据不是所支持的尺寸，被编译的代码就会调用函数 bad_user_access_length。如果这些编译代码是一个模块，由于这个符号没有开放，模块就不能加载了。

get_user(ptr)

这个宏用来从用户空间获取一个数据。除了数据传输的方向不同外，它与 put_user 是一样的。当 insmod 不能解析符号时，bad_user_access_length 的又臭又长的名字可以当作一个很有意义的错误信息。这样，开发人员就可以在向大众分布模块前加载和测试模块，他会很快找到并修改错误。相反，如果使用了不正确尺寸的 put_user 和 get_user 直接编译到了内核中，bad_user_access_length 就会导致系统 panic。尽管对于尺寸错误的数据传输来说，oops 比其系统 panic 要友好得多，但还是选择了较为激进的方法来尽力杜绝这种错误。

scull 的 ioctl 实现只传送设备的可配置参数，其代码非常简单，罗列如下：

（代码）

还有 6 项是操作 scull_qset 的。这些操作 scull_quantum 的一样，为了节省空间，没有在上面的例子中列出。

从调用者的角度看（即从用户空间），传递和接收参数的 6 种方法如下所示：

（代码）
如果你需要写一个可以在Linux 1.2里运行的模块，get_user和put_user会是非常棘手的函数，因为它们直到内核 1.3 才引入到系统中。在切换到类型依赖宏之前，程序员使用一些称为get_user_byte 等等的函数。旧的宏只在内核 1.3 中定义了，在 2.0 内核中，只有你事先使用了#define WE_REALLY_WANT_TO_USE_A_BROKEN_INTERFACE 时才能使用旧的宏。不过为了可移植性，为旧内核定义 put_user 是一种更好的解决方法，于是为了驱动程序可以在旧内核中良好运行，scull/sydep.h 包含了这些宏的定义。

非 ioctl 设备控制

有时通过向设备自身发送写序列能够更好地完成对设备的控制。例如，这一技术使用在控制台驱动程序中，它称为“escape 序列”，用来控制光标移动，改变默认颜色，或是完成某些配置任务。用这种方法实现设备控制的好处是，用户仅用写数据就可以完成对设备的控制，无需使用（有时是写）完成设备配置的程序。
例如，程序 setterm 通过打印 escape 序列完成对控制台（或其他终端）的配置。这种方法的优点是可以远程控制设备。由于可以简单地重定向数据流完成配置工作，控制程序可以运行在另外一台不同的计算机上，而不一定非要在被控设备的计算机上。你已经在终端上使用了这项技术，但这项技术可以更通用一些。
“通过打印控制”的缺点是，它给设备增加了策略限制；例如，只有你确认控制序列不会出现在正常写到设备的数据中时，这项技术才是可用的。对于终端来说，这只能说是部分正确。尽管文本显示只意味着显示ASCII字符，但有时控制字符也会出现在正在打印的数据中，因此会影响控制台的配置。例如，当你对二进制文件进行grep时可能会发生这样的情况；分解出的字符行可能什么都包含，最后经常会造成控制台的字体错误[*]。
写控制特别适合这样的设备，不传输数据，仅相应命令，如机器人设备。
例如，我所写的驱动程序之一是控制一个在两个轴上的摄像头的移动。在这个驱动程序中，“设备”是一对旧的步进马达，它既不能读也不能写。向步进马达“发送数据流”多少没有多大意义。在这种情况下，驱动程序将所写的数据解释为 ASCII 命令，并将请求转换为脉冲，实现对步进马达的控制。命令可以是任何象“向左移动 14 步”，“达到位置 100，43”或“降低默认速度”之类的字串。驱动程序仅将/dev 中的设备节点当作为应用程序设立的命令通道。对该设备直接控制的优点是，你可以使用 cat 来移动摄像头，而无需写并编译发出 ioctl 调用的特殊代码。
当编写“面向命令的”驱动程序时，没有理由要实现 ioctl 方法。为解释器多实现一条命令对于实现和使用来说，都更容易。
好奇的读者可以看看由 O’Reilly FTP 站点提供的源码 stepper 目录中的 stepper 驱动程序；由于我认为代码没有太大的意义（而且质量也不是太高），这里没有包含它。

阻塞型 I/O

read 的一个问题是，当尚未有数据可读，而又没有到文件尾时如何处理。
默认的回答是，“我们必须睡眠等待数据。”本节将介绍进程如何睡眠，如何唤醒，以及一个应用程序如何在不阻塞 read 调用的情况下，查看是否有数据。对于写来说也可以适用同样

[*] Ctrl-N设置替代字体，它有图形字符组成，因此你的外壳的输入来说是不友好的；如果你遇见了这样的问题，回应一个Ctrl-O字符来恢复主字体。

的方法。
通常，在我向你介绍真实的代码前，我将解释若干概念。

睡眠和唤醒

当进程等待事件（可以是输入数据，子进程的终止或是其他什么）时，它需要进入睡眠状态以便其他进程可以使用计算资源。你可以调用如下函数之一让进程进入睡眠状态：
（代码）
然后用如下函数之一唤醒进程：
（代码）
在前面的函数中，wait_queue 指针的指针用来代表事件；我们将在“等待队列”一节中详细讨论这个数据结构。从现在开始，唤醒进程需要使用进程睡眠时使用的同一个队列。因此，你需要为每一个可能阻塞进程的事件对应一个等待队列。如果你管理 4 个设备，你需要为阻塞读预备 4 个等待队列，为阻塞写再预备 4 个。存放这些队列的最佳位置是与每个设备相关的硬件数据结构（在我们的例子中就是 Scull_Dev）。
但“可中断”调用和普通调用有什么区别呢？
sleep_on 不能信号取消，但 interruptible_sleep_on 可以。其实，仅在内核的临界区才调用 sleep_on；例如，当等待从磁盘上读取交换页面时。没有这些页面进程就无法继续运行，用信号打断这个操作是没有任何意义的。然而，在所谓“长系统调用”，如 read，中要使用 interruptible_sleep_on。当进程正等待键盘输入时，用一个信号将进程杀死是很有意义的。
类似地，wake_up 唤醒睡在队列上的任何一个进程，而 wake_up_interruptible 仅唤醒可中断进程。
做为一个驱动程序编写人员，由于进程仅在 read 或 write 期间才睡眠在驱动程序代码上，你应该调用 interruptible_sleep_on 和 wake_up_interruptible。不过，事实上由于没有“不可中断”的进程在你的队列上睡眠，你也可以调用 wake_up。但是，出于源代码一致性的考虑，最好不这样做。（此外，wake_up 比它的搭档来说要稍微慢一点。）

编写可重入的代码

当进程睡眠后，驱动程序仍然活着，而且可以由另一个进程调用。让我们一控制台驱动程序为例。当一个应用在 tty1 上等待键盘输入，用户切换到 tty2 上并派生了一个新的外壳。现在，两个外壳都在控制台驱动程序中等待键盘输入，但它们睡在不同的队列上：一个睡在与 tty1 相关的队列上，一个睡在与 tty2 相关的队列上。每个进程都阻塞在 interruptible_sleep_on 函数中，但驱动程序让可以继续接收和响应其他 tty 的请求。
可以通过编写“可重入代码”轻松地处理这种情况。可重入代码是不在全局变量中保留状态信息的代码，因此能够管理交织在一起的调用，而不会将它们混淆起来。如果所有的状态信息都与进程有关，就不会发生相互干扰。
如果需要状态信息，既可以在驱动程序函数的局部变量中保存（每个进程都有不同的堆栈来保存局部变量），也可以保存在访问文件用的filp中的private_data中。由于同一个filp可能在两个进程间共享（通常是父子进程），最好使用局部变量*。
如果你需要保存大规模的状态信息，你可以将指针保存在局部变量中，并用 kmalloc 获取实

* 注意，内核栈无法存储太大的数据项。在这种情况下，我建议为大数据分配内存，并将这些空间的地址保存在局部变量中。

际存储空间。此时，你千万别忘了 kfree 这些数据，因为当你在内核空间工作时，没有“在进程终止时释放所有资源”的说法。
你需要将所有调用了 sleep_on（或是 schedule）的函数写成可重入的，并且包括所有在这个函数调用轨迹中的所有函数。如果 sample_read 调用了 sample_getdata，后者可能会阻塞，由于调用它们的进程睡眠后无法阻止另一个进程调用这些函数，sample_read 和 sample_gendata 都必须是可重入的。此外，任何在用户空间和内核空间复制数据的函数也必须是可重入的，这是因为访问用户空间可能会产生页面失效，当内核处理失效页面时，进程可以会进入睡眠状态。

等待队列

我听见你在问的下一个问题是，“我到底如何使用等待队列呢？”
等待队列很容易使用，尽管它的设计很是微妙，但你不需要直到它的内部细节。处理等待队列的最佳方式就是依照如下操作：

- 声明一个 struct wait_queue *变量。你需要为每一个可以让进程睡眠的事件预备这样一个变量。这就是我建议你放在描述硬件特性数据结构中的数据项。
- 将该变量的指针做为参数传递给不同的 sleep_on 和 wake_up 函数。

这相当容易。例如，让我们想象一下，当进程读你的设备时，你要让这个进程睡眠，然后在某人向设备写数据后唤醒这个进程。下面的代码就可以完成这些工作：
（代码）
该设备的这段代码就是例子程序中的 sleepy，象往常一样，可以用 cat 或输入/输出重定向等方法测试它。
上面列出的两个操作是你唯一操作在等待队列上的两个操作。不过，我知道某些读者对它的内部结构感兴趣，但通过源码掌握它的内部结构很困难。如果你不对更多的细节感兴趣，你可以跳过下一节，你不会损失什么的。注意，我谈论的是“当前”实现（版本 2.0.x），但没有什么规定限制内核开发人员必须依照那样的实现。如果出现了更好的实现，内核很容易就会使用新的，由于驱动程序编写人员只能通过那两个合法操作使用等待队列，对他们来说没有什么坏的影响。
当前 struct wait_queue 额实现使用了两个字段：一个指向 struct task_struct 结构（等待进程）的指针，和一个指向 struct wait_queue（链表中的下一个结构）的指针。等待队列是循环链表，最后一个结构指向第一个结构。
该设计的引入注目的特点是，驱动程序编写人员从来不声明或使用这个结构；他们仅仅传递它的指针或指针的指针。实际的结构是存在的，但只在一个地方：在__sleep_on 函数的局部变量中，上面介绍的两个 sleep_on 函数最终会调用这个函数。
这看上去有点奇怪，不过这是一个非常明智的选择，因为无需处理这种结构的分配和释放。进程每次睡在某个队列上，描述其睡眠的数据结构驻留在进程对应的不对换的堆栈页中。
当进程加入或从队列中删除时，实际的操作如图 5-1 所示。
（图 5-1 等待队列的工作示意）

阻塞型和非阻塞型操作

在分析功能完整的 read 和 write 方法前，我们还需要看看另外一个问题，这就是 filp->f_flags 中的 O_NONBLOCK 标志。这个标志定义在<linux/fcntl.h>中，在最近的内核中，这个头文

件由<linux/fs.h>自动包含了。如果你在内核 1.2 中编译你的模块，你需要手动包含 fcntl.h。这个标志的名字取自“打开-非阻塞”，因为这个标志可以在打开时指定（而且，最初只能在打开时指定）。由于进程在等待数据时的正常行为就是睡眠，这个标志默认情况下是复位的。在阻塞型操作的情况下，应该实现下列操作：

- 如果进程调用 read，但（尚）没有数据，进程必须阻塞。当数据到达时，进程被唤醒，并将数据返回给调用者，即便少于方法的 count 参数中所请求的数据量，也是如此。
- 如果进程调用了 write，缓冲区又没有空间，进程也必须阻塞，而且它必须使用与用来实现读的等待队列不同的等待队列。当数据写进设备后，输出缓冲区中空出部分空间，唤醒进程，write 调用成功完成，如果缓冲区中没有请求中 count 个字节，则进程可能只是完成了部分写。

前面的列表的两个语句都假设，有一个输入和输出缓冲区，而且每个设备驱动程序都有一个。输入缓冲区需要用来在数据达到而又没有人读时避免丢失数据，输出缓冲区用来尽可能增强计算机的性能，尽管这样做不是严格必须的。由于如果系统调用不接收数据的话，数据仍然保存在用户空间的缓冲区中，write 中可以丢失数据。

在驱动程序中实现输出缓冲区可以获得一定的性能收益，这主要是通过较少了用户级/内核级转换和上下文切换的数目达到的。如果没有输出缓冲区（假设是一个慢设备），每次系统调用只接收一个或很少几个字节，并且当进程在 write 中睡眠时，另一进程就会运行（有一次上下文切换）。当第一个进程被唤醒后，它恢复运行（又一次上下文切换），write 返回（内核/用户转换），进程还要继续调用系统调用写更多的数据（内核/用户转换）；然后调用再次被阻塞，再次进行整个循环。如果输出缓冲区足够大，write 首次操作时就成功了；数据在中断时被推送给设备，而不必将控制返回用户空间。适合于设备的输出缓冲区的尺寸显然是和设备相关的。

我们没有在 scull 中使用输入缓冲区，这是因为当调用 read 时，数据已经就绪了。类似地，也没有使用输出缓冲区，数据简单地复制到设备对应的内存区中。我们将在第 9 章“中断处理”的“中断驱动的 I/O”一节中介绍缓冲区的使用。

如果设置了 O_NONBLOCK 标志，read 和 write 的行为是不同的。此时，如果进程在没有数据就绪时调用了 read，或者在缓冲区没有空间时调用了 write，系统简单地返回-EAGAIN。如你所料，非阻塞型操作立即返回，允许应用查询数据。当使用 stdio 函数处理非阻塞型文件时，由于你很容易误将非阻塞返回认做是 EOF，应用程序应该非常小心。你必须始终检查 errno。

你也许可以从它的名字猜到，O_NONBLOCK 在 open 方法也可有作用。当 open 调用可能会阻塞很长时间时，就需要 O_NONBLOCK 了；例如，当打开一个 FIFO 文件而又（尚）无写者时，或是访问一个被锁住的磁盘文件时。通常，打开设备成功或失败，无需等待外部事件。但是，有时打开设备需要需要很长时间的初始化，你可以选择打开 O_NONBLOCK，如果设置了标志，在设备开始初始化后，会立即返回一个-EAGAIN（再试一次）。你还可以为支持访问策略选择实现阻塞型 open，方式与文件锁类似。我们稍后将在“替代 EBUSY 的阻塞型打开”一节中看到这样一种实现。

只有 read，write 和 open 文件操作受非阻塞标志的影响。

样例实现：scullpipe

/dev/scullpipe 设备（默认有 4 个设备）是 scull 模块的一部分，用来展示如何实现阻塞型 I/O。在驱动程序内部，阻塞在 read 调用的进程在数据达到时被唤醒；通常会发出一个中断来通知这样一种事件，驱动程序在处理中断时唤醒进程。由于你应该无需任何特殊硬件－－没有

任何中断处理函数，就可以在任何计算机上运行 scull，scull 的目标与传统驱动程序完全不同。我选择的方法是，利用另一个进程产生数据，唤醒读进程；类似地，用读进程唤醒写者。这种实现非常类似与一个 FIFO（或“命名管道”）文件系统节点的实现，设备名就出自此。设备驱动程序使用一个包含两个等待队列和一个缓冲区的设备结构。缓冲区的大小在通常情况下是可以配置的（编译时，加载时和运行时）。
（代码）
read 实现管理阻塞型和非阻塞型数据，如下所示：
（代码）
如你所见，我在代码中留下了 PDEBUG 语句。当你编译驱动程序时，你可以打开消息，这样就可以更容易地看到不同进程间的交互了。
跟在 interruptible_sleep_on 后的 if 语句处理信号处理。这条语句保证对信号恰当和预定的处理过程，它会让内核完成系统调用重启或返回-EINTR（内核在内部处理-ERESTARTSYS，最终返回到用户空间的是-EINTR）。我不想让内核对阻塞信号完成这样的处理，主要时我想忽略这些信号。否则，我们可以返回-ERESTARTSYS 错误给内核，让它完成它的处理工作。我们将在所有的 read 和 write 实现中使用一样的语句进行信号处理。
write 的实现与 read 非常相似。它唯一的“特殊”功能时，它从不完全填充缓冲区，总时留下至少一个字节的空洞。因此，当缓冲区空的时候，wp 和 rp 时相等的；当存在数据时，它们是不等的。
（代码）
正如我所构想的，设备没有实现阻塞型 open，这要比实际的 FIFO 要简单得多。如果你想要看看实际的代码，你可以在内核源码的 fs/pipe.c 中找到那些代码。
要测试 scullpipe 设备的阻塞型操作，你可以在其上运行一些应用，象往常一样，可以使用输入/输出重定义等方法。由于普通程序不执行非阻塞型操作，测试非阻塞活动要麻烦些。misc-progs 源码目录中包含了一个很简单的程序，称为 nbtest，用它来测试非阻塞型操作，该程序罗列如下。它所做的就是使用非阻塞型 I/O 复制它的输入和输出，并在期间稍做延迟。延迟时间可以通过命令行传递，默认情况下时 1 秒钟。
（代码）

Select

在使用非阻塞型 I/O 时，应用程序经常要利用 select 系统调用，当涉及设备文件时，它依赖于一个设备方法。这个系统调用还用来实现不同源输入的多路复用。在下面的讨论中，我假设你知道在用户空间中 select 的语义的用法。注意，内核 2.1.23 引入了 poll 系统调用，因此为了支持这两个系统调用，它改变驱动程序方法的工作方式。
为了保存所有正在等待文件（或设备）的信息，Linux 2.0 的 select 系统调用的实现使用了 select_table 结构。再次提醒你，你无需了解它的内部结构（但不管怎样，我们一会会稍做介绍），而且只允许调用操作该结构的函数。
当 select 方法发现无需阻塞时，它返回 1；当进程应该等待，它应该“几乎”进入睡眠状态。在这种情况下，要在 select_table 结构中加入等待队列，并且返回 0。
仅当选择的文件中没有一个可以接收或返回数据时，进程才真正进入睡眠状态。这一过程发生在 fs/select.c 的 sys_select 中。
写 select 操作的代码要比介绍它要容易得多，现在就可以 scull 中时如何实现的：
（代码）
这里没有代码处理“第 3 种形式的选择”，选择异常。这种形式的选择时通过 mode == SEL_EX

标别的，但大多数时候你都将其编写为默认情况，在其他选择均失败时执行。异常事件的含义与设备有关，所以你可以选择是否在你自己的驱动程序中实现它们。这种功能将只会为专为你的驱动程序设计的程序使用，但那并不它的初衷。在这方面，它与依赖于设备的 ioctl 调用很相似。在实际使用中，select 中异常条件的主要用途是，通知网络连接上带外（加急）数据的达到，但它还用在终端层以及管道/FIFO 实现中（你可以查看 fs/pipe.c 中的 SEL_EX）。不过要注意，其他 Unix 系统的管道和 FIFO 没有实现异常条件。
这里给出的 select 代码缺少对文件尾的支持。当 read 调用达到文件尾时，它应该返回 0，select 必须通过通告设备可读来支持这种行为，这样应用程序就不会永远等待调用 read 了。例如，在实际的 FIFO 中，当所有的写者都关闭了文件时，读者会看到文件尾，而在 scullpipe 中，读者永远也看不到文件尾。设计这种不同行为的原因时，一般将 FIFO 当做两个进程间的通信通道，而 scullpipe 是一个只要至少有一个读者，所有人就都可以输入数据的垃圾筒。此外，也没有必要重新实现内核中已经有了的设备。
象 FIFO 那样实现文件尾意味着要在 read 和读 select 中检查 dev->nwriters，并做相应处理。不过很遗憾，如果读者在写者前打开设备，它马上就看到文件尾了，没有机会等待数据到达。最好的解决这个问题的方法是，实现阻塞型 open，这个任务做为练习留给读者。

与 read 和 write 的交互

select 调用的目的是事先判断是否有 I/O 操作会阻塞。从这个方面说，它时对 read 和 write 的补充。由于 select 可以让驱动程序同时等待多个数据流（但这与这里的情况无关），select 在这方面也时很有用途的。
为了让应用正确工作，正确实现这 3 个调用时非常重要的。尽管下面的规则已经多多少少谈过了一些，我还要在这里再总结一下。

从设备读取数据

如果在输入缓冲区中有数据，即便比所请求的数据少，而且驱动程序可以保证剩下的数据会马上达到，read 调用应该不经过任何可以察觉的延迟立即返回。如果你至少可以返回 1 个字节，而且很方便的话，你总可以返回比请求少的数据（我们在 scull 就是这样做的）。当前内核中总线鼠标的实现在这方面就时错的，某些程序（如 dd）无法正确读取这些设备。
如果输入缓冲区中没有数据，在至少有一个字节可读前 read 必须阻塞，除非设置了 O_NONBLOCK。非阻塞型 read 立即返回-EAGAIN（尽管在这种情况下某些旧的 System V 会返回 0）。在至少有一个字节可读前，select 必须报告设备不可读。只要有数据可读，我们就使用上一条规则。
如果我们到了文件尾，，无论是否有 O_NONBLOCK，read 都应该立即返回 0。select 应该报告说文件可读。

向设备写数据

如果输出缓冲区有空间，write 应该不做任何延迟返回。它可以接收少于请求数目的数据，但是它须接收至少一个字节。在这种情况下，select 应该报告设备可写。
如果输出缓冲区是满的，在空间释放前 write 一直阻塞，除非设置了 O_NONBLOCK 标志。

非阻塞型 write 立即返回，返回值为-EAGAIN（或者在某些条件为 0，如前面旧版本的 System V 所说）。select 应该报告文件不可写。但另一方面，如果设备不能接收任何数据，无论是否设置了 O_NONBLOCK，write 都返回-ENOSPC（“设备无可用空间”）。
如果使用设备的程序需要确保等候在输出队列中的数据真的完成了传送，驱动程序必须提供一个 fsync 方法。例如，可移动设备使用提供 fsync 入口点。千万不要在调用返回前让 write 调用等待数据传送结束。这是因为，需要应用程序都可用 select 检查设备是否时可以写的。如果设备报告可以写，write 调用应该保持一致，不能阻塞。

刷新待处理输出

我们已经看到 write 方法为什么不能满足所有数据输出的需求。通过同名系统调用调用的 fsync 函数弥补了这一空缺。
如果某些应用需要确保数据传送到设备上，设备就必须实现 fsync 方法。无论是否设置了 O_NONBLOCK 标志，fsync 调用应该仅在设备已经完全刷新数据后才能返回，甚至花些时间也要如此。
fsync 方法没有什么不寻常的功能。调用不是时间关键的，所以每个设备驱动程序都可以按照作者的风格实现这个方法。大多数时候，字符设备驱动程序的 fsync 在其 fops 结构都对应一个 NULL 指针。而块设备总是通过调用通用的 block_fsync 函数实现这个方法，block_fsync 刷新设备的所有缓冲块，一直等到 I/O 结束。

所使用的数据结构

内核 2.0 所使用的 select 特殊实现相当高效，而且还有点复杂。如果你不对操作系统的细节感兴趣，你可以直接跳到下一节。
首先，我建议你看一看图 5-2，它展示了调用 select 所涉及的步骤。看看这张图会有助于搞清除下面的讨论。
select 的工作是由函数 select_wait 和 free_wait 完成的，select_wait 是声明在<linux/sched.h>里的内嵌函数，而 free_wait 则是在 fs/select.c 中定义的。它们使用的数据结构时 struct select_table_entry 数组，每一项都是由 struct wait_queue 和 struct wait_queue **组成的。前者是插入到设备等待队列的实际数据结构（当调用 sleep_on 时以局部变量形式存在的数据结构），而后者是在所选条件有一个为真将当前进程从队列中删除时所需要的“句柄”——例如，当选择 scullpipe 进行读操作时，它包含&dev->inq（见“Select”一节中较早的例子）。简而言之，select_wait 将下一个空闲的 select_table_entry 插入到指定的等待队列中。当系统调用返回时，free_wait 利用对应的指针删除自己等待队列中的每一项。
select_table 结构（有一个指向数据的指针和活动数据变量组成）在 do_select 声明为一个局部变量，这一点与__sleep_on 相似。但数组项却保存在另一个页面上，否则它就会造成当前进程的堆栈溢出。
如果你对这些描述理解起来较为困难，看看源码。一旦你理解了实现，你就可以看到它时如此的简洁和高效。

异步触发

尽管大多数时候阻塞型和非阻塞型操作的组合，以及 select 方法可以有效地查询设备，但某些时候用这种技术管理就不够高效了。例如，想我们想象一下，一个在低优先级执行长计算循环的进程，但它需要尽可能快地处理输入的数据。如果输入通道时键盘，你可以想进程发送信号（使用“INTR”字符，一般就是 Ctrl-C），但是这种信号是 tty 层的一部分，在一般字符设备设备中没有。我们所需要的异步触发与此有些不同。此外，任何输入数据都应该产生一个中断，而不仅仅是 Ctrl-C。

为了打开文件的异步触发机制，用户程序必须执行两个步骤。首先，它们指定进程是文件的“属主”。文件属主的用户 ID 保存在 filp->f_owner 中，可以通过 fcntl 系统调用的 F_SETOWN 命令设置这个值。此外，为了确实打开异步触发机制，用户程序还必须通过另一个 fcntl 命令设置设备的 FASYNC 标志。

在完成这两个步骤后，无论何时新数据到达，输入文件都产生一个 SIGIO 信号。信号发送给存放在 filp->f_owner 的进程（如果是负值，则是进程组）。

例如，如下这些行代码打开 stdin 输入文件到当前进程的异步触发机制：

（代码）

源码中的名为 asynctest 的程序就是这样读取 stdin 的程序。它可以用来测试 scullpipe 的异步功能。这个程序类似与 cat，但在文件尾时不会终止；它只对输入反应，而不是在没有输入时反应。

不过要注意，并不是所有的设备都支持异步触发，而且你可以选择支持或不支持。应用通常应该假设仅有套接字和终端才有异步能力。例如，至少对当前内核来说，管道和 FIFO 就不支持异步触发。鼠标提供了异步触发（尽管在 1.2 中没有），这是因为某些程序希望鼠标能够象 tty 那样发送 SIGIO。

对于输入触发来说还存在一个问题。当进程接收到 SIGIO 后，它不知道时哪个文件有新输入了。如果多于一个文件可以异步触发进程，通知有数据待处理，应用程序仍须借助 select 探测到底发生了什么。

从驱动程序的角度看

与我们更相关的一个话题是，设备驱动程序如何实现异步信号。下面的列表从内核角度给出了这种操作的详细过程：

- 当调用 F_SETOWN 时，除了对 filp->f_owner 赋值以外什么也不做。
- 当调用 F_SETFL 打开 FASYNC 标志时，驱动程序的 fasync 方法被调用。无论 FASYNC 的值何时发生变化，该方法都被调用，通知驱动程序该标志的变化，以便驱动程序能够正确地响应。在文件被打开时，这个标志默认时被清 0 的。我们一会就可以看到这个驱动程序方法的标准实现。
- 当数据达到时，想所有注册异步触发的进程发送 SIGIO 信号。

尽管实现的第一步很简单——在驱动程序端没有什么可做的——其他步骤则为了跟踪不同的异步读者，要涉及一个动态数据结构；同时可能有多个读者。然而，这个动态数据结构不依赖与某个特定设备，内核提供了一套合适的通用实现方法，你不必在每个驱动程序都重写一遍代码了。

遗憾的是，内核 1.2 没有包含这个实现。对与旧版本的内核来说，在模块中实现异步触发不是很容易，你得写你自己的数据结构。为了简单，scull 模块不对旧内核提供异步触发机制。

由 Linux 提供的通用实现基于一个数据结构和两个函数（要在上面谈的步骤中调用）。声明相关内容的头文件是<linux/fs.h>－－没什么新的－－数据结构称为 struct fasync_struct。如我们处理等待队列一样，我们需要在设备相关的数据结构中插入这个结构的指针。事实上，我们已经在“样例实现：scullpipe”一节中看到了这个字段。
根据如下原型调用那两个函数：
（代码）
当打开文件的 FASYNC 标志被修改时，调用前者从感兴趣进程列表上增加或删除文件，当数据达到时，则应该调用后者。
这里时 scullpipe 如何实现 fasync 方法的：
（代码）
很清除，所有的工作都是由 fasync_helper 完成的。然后，不可能不在驱动程序的方法中实现这一功能，这是因为助手函数需要访问指向 struct fasync_struct *的正确指针（这里是&dev->fasync_queue），而且只有驱动程序才能提供这些信息。
当数据到达时，必须执行下面的语句异步通知读者。由于 scullpipe 读者的新数据是由另一个调用 write 的进程产生的，这条语句出现 scullpipe 的 write 方法中。
（代码）
似乎我们已经完成了所有的工作，但还忘了一件事。我们必须在文件关闭时调用我们的 fasync 方法，去除活动异步读者链表中去除被关闭的文件。尽管这个调用仅当 filp->f_flags 设置为 FASYNC 才需要，调用这个函数不会有什么危害，而且它时比较普遍的实现。例如，如下代码行就是 scullpipe 的 close 方法的一部分：
（代码）
异步触发机制使用的数据结构与 struct wait_queue 结构非常相似，因为两者都涉及等待事件。不同之处是，前者使用 struct file 代替了 struct task_struct。为了向进程发送信号，通过队列中的 struct file 获取 f_owner 字段。

定位设备

本章的难点都以讲过了，下面我们将浏览一下 lseek 方法，它很有用而且很容易实现。注意，在 2.1.0 中该方法的原型稍有变化，第 17 章的“原型区别”一节中将详细介绍。

lseek 实现

我已经说了，如果设备操作中没有 lseek，内核响应的默认实现是通过修改 filp->f_pos，从文件的起始处和当前位置开始定位。
如果对你的设备来说，相对文件尾定位有意义，你应该提供自己的方法，它看上去就象如下这些代码：
（代码）
这里，唯一的设备相关操作是从设备中获取文件长度。不过为了能够让 lseek 系统调用正确工作，无论何时发生了数据传送，read 和 write 调用都必须相应更新 filp->f_pos；它们也应该使用 f_pos 字段定位要传送的数据。如第 3 章的“读和写”一节中介绍的，scull 的实现包括了这些功能。
scull 处理一个精确定义的数据区，上面的实现对于 scull 来说是有意义的，但大部分设备仅提供了数据流而不是数据区（想想串口和键盘），定位时没有意义的。如果是这种情况，你

并不能从声明 lseek 操作中摆脱出来，因为默认方法时允许定位的。相反，你应该使用如下这段代码：
（代码）
刚刚看到的代码摘自 scullpipe 设备，它时不可定位的；尽管错误码的符号名代表“是管道”，它实际翻译为“非法定位”。由于位置指示器 filp->f_pos 对于非可定位设备时没有意义的，在数据传递过程中 read 或 write 都不必更新这个字段。

设备文件的访问控制

有时提供访问控制对于设备节点的可靠性来说时至关重要的。不仅未授权的用户不允许使用设备（可以通过文件系统的权限位来指定），而且有时只允许一次一个授权用户打开设备。
上面给出的代码中除了文件系统权限位外，没有实现任何访问控制。如果 open 系统调用将请求转发给驱动程序，open 就会成功。这里我将介绍实现某些附加检查的新技术。
这里的问题与 tty 的使用很相似。在 tty 的使用中，当用户登录到系统后，login 进程修改设备节点的属主，防止其他进程侵入 tty 数据流。然而，仅仅为了保证对设备的唯一使用，每次都通过特权程序修改设备属主时不现实的。
出现在本节的设备与基本 scull 设备有类似的功能（即，都实现了一个持久性内存区）；它与 scull 的区别仅仅时访问控制，这些控制是在 open 和 close 操作中实现的。

独享设备

提供访问控制的最残忍的方法就是，每次只允许一个进程打开设备（独享）。我个人不喜欢这种技术，因为它阻碍了用户的灵活性。一个用户可能需要在同一个设备上运行多个进程，一个读状态信息，其他进程写数据。通常有许多程序和外壳脚本可以完成许多任务。换句话说，独享更象时策略而非机制（至少我认为这样）。
除了我讨厌独享外，它对于设备驱动程序来说非常易于实现，这里给出代码。源码摘自一个称为 scullsingle 的设备。
open 调用基于一个全局整数标志拒绝访问：
（代码）
另一方面，close 调用标记设备不在忙。
（代码）
存放打开标志（scull_s_count）最佳场所是设备结构（这里是 Scull_Dev），因为从概念上说，它从属于设备。
然而，scull 驱动程序单独使用一个变量保存打开标志，主要时为了与基本 scull 设备使用相同的设备结构，减少代码的重复。

限制每次只由一个用户访问

更有用的访问控制实现是，如果没有其他人能够控制设备，只允许一个使用有访问权。这种检查时在正常的权限检查后进行的，它可以提供比指定属主和组权限位更严格的访问控制。这种方法与 tty 使用的策略时相同的，但它没有使用一个外部特权程序。
这一功能比起实现独享来要有一定的技巧。此时，需要两个数据项，一个打开计数和设备“属

主”的 UID。再次说明，存放这些数据项的最佳位置是在设备结构内部；但基于和前面 scullsingle 中同样的原因，例子中使用了全局变量。设备的名字时 sculluid。
open 调用在首次打开时授权，但它记下设备的属主。这意味着，用户可以多次打开设备，允许合作进程无缝衔接在一起。与此同时，其他用户不能打开设备，避免了外部干扰。由于这个函数的版本基本和前一个相同，这里仅列出了相关部分：
（代码）
这里虽然代码完成了权限检查，我还是决定返回-EBUSY 而不是-EPERM 来告诉被拒绝的用户正确的原因。返回“权限拒绝”通常都是检查/dev 文件的权限和属主的结果，而“设备忙”则正确地告诉用户，已有进程正在使用设备。
由于 close 仅仅时对使用计数减 1，它的代码这里没有列出。

阻塞型打开替代 EBUSY

当设备不能访问时返回一个错误，这是通常最合理的方式，但有些情况下，你最好让进程等待设备。
例如，如果数据通信通道同时用来定时传送报告（使用 crontab），同时也偶尔根据人们的需要使用，定时报告最好稍微延迟一会儿，而不是因为通道正忙就失败返回。
这是程序员在实际驱动程序时必须做出的选择之一，正确的答案依赖所解决的问题。
正如你所猜到的，可以用返回 EBUSY 替代阻塞型打开。
scullwuid 时 sculluid 的另一个版本，它在 open 中等待设备而不是返回-EBUSY。它和 sculluid 的唯一不同时 open 操作中的部分代码：
（代码）
然后，release 方法要负责唤醒任何等待的进程：
（代码）
阻塞型打开实现的问题在于，对于交互式用户来说这是非常不愉快的，他可能会猜想设备出了什么毛病。交互式用户通常使用 cp 和 tar 之类的命令，它们都没有在 open 调用中加上 O_NONBLOCK 选项。隔壁某些用磁带做备份的人更愿意被告知“设备或资源忙”，而不是坐在一边，猜想为什么今天使用 tar 时硬盘会不声不响呢。
这类问题（相同设备的不兼容策略）最好为每种访问策略实现一个设备节点的方法来解决，这与/dev/ttyS0 和/dev/cua0 用不同方法操作同一个串口，或是/dev/sculluid 和/dev/scullwuid 提供两种不同策略访问同一内存区很相似。

在打开时克隆设备

管理访问控制的另一项技术是，在进程打开设备时创建设备的一个私有副本。
很明显，如果设备没有绑定到某个硬件对象上的话，这样做是有可能的；scull 就是这种“软件”设备的例子。kmouse 模块也使用了这一技术，所以每个虚拟控制台好象都有一个鼠标设备。当设备的副本是由软件驱动程序创建的时，我称我们为“虚拟设备”——就如同“虚拟控制台”都使用同一个物理 tty 设备一样。
虽然这种对访问控制的需要并不常见，这个实现同样很能说明，内核代码可以轻松地修改从应用角度看的外部世界（即，计算机）。事实上，这个话题确实有点怪异，如果你不感兴趣，你可以直接跳到下一章。
软件包 scull 中的/dev/scullpriv 设备节点实现了虚拟设备。scullpriv 实现利用进程控制终端的

次设备号做为访问虚拟设备的键值；这种选择造成一个完成不同的策略。例如，如果使用 uid，就会造成每个用户一个不同的虚拟设备，而使用 pid 则为每个访问设备的进程创建一个新设备。
决定使用控制终端可以方便地通过输入/输出重定向测试设备。
open 方法如下所示。它必须查找一个合适的虚拟设备，可能的话还要创建一个。由于函数的最后一部分是从基本 scull 中复制过来的，我们在前面已经看过了，这里没有列出。
（代码）
close 方法没有什么特别处理。它在最后一次关闭时释放设备，为了简化对设备的测试，我没有维护打开计数。如果设备在最后一个关闭时释放了，除非有一个后台进程至少打开设备一次，否则在写设备后无法再从设备中读到相同的数据。驱动程序样例选择使用较为简单的方法保存数据，所以在下次打开设备时，你可以找到那些数据。当 cleanup_module 被调用时，释放设备。
这里时/dev/scullpriv 的 close 实现，它也结束了本章。
（代码）

快速索引

本章介绍了如下这些符号和头文件：
#include <linux/ioctl.h>
定义了所有用于定义 ioctl 命令宏的头文件。现在它包含在<linux/fs.h>中。Linux 1.2 没有定义本章介绍那些宏；如果需要向后兼容的话，我建议你看一看 scull/sysdep.h，那里为老版本的内核定义相应的符号。
_IOC_NRBITS
_IOC_TYPEBITS
_IOC_SIZEBITS
_IOC_DIRBITS
ioctl 命令不同位字段的可用位数。还有 4 个宏定义了不同的 MASK（掩码），4 个宏定义了不同的 SHIFT（偏移），但它们基本仅用于内部使用。由于_IOC_SIZEBITS 在不同体系结构上的值不同，它时一个需要检查的比较重要的值。
_IOC_NONE
_IOC_READ
_IOC_WRITE
“方向”位字段的可能值。“读”和“写”时不同的位，可以 OR 在一起实现读/写。这些值都是基于 0 的。
_IOC(dir,type,nr,size)
_IO(type,nr)
_IOR(type,nr,size)
_IOW(tyep,nr,size)
_IOWR(type,nr,size)
用于生成 ioctl 命令的宏。
_IOC_DIR(nr)
_IOC_TYPE(nr)
_IOC_NR(nr)
_IOC_SIZE(nr)

用于解码 ioctl 命令的宏。特别地，_IOC_TYPE(nr)可以是_IOC_READ 和_IOC_WRITE 的 OR 组合。

#include <linux/mm.h>

int verify_area(int mode, const void *ptr, unsigned long extent);

这个函数验证指向用户空间的指针是否可用。verify_area 处理页面失效，必须在除 read 和 write 以外的地方访问用户前调用，read 和 write 被调用时缓冲区已经被验证了。非 0 返回值报告错误，应该将其返回给调用者。在内核 2.1 中使用 verify_area 以及如下这些宏和函数的话，见第 17 章的“访问用户空间”一节。

VERIFY_READ

VERIFY_WRITE

verify_area 中 mode 参数的可能值。VERIFY_WRITE 是 VERIFY_READ 的超集。

#include <asm/segment.h>

void put_user(datum, ptr);

usigned long get_user(ptr);

用从用户空间中存取单个数据的宏。传送的字节个数依赖于 sizeof(*ptr)。版本 1.2 中没有这些函数。如果你想在版本 1.2 和 2.0 中同时编译模块的话，你可以看看 scull/sysdep.h。

void put_user_byte(val, ptr);

usigned char get_user_byte(ptr);

这些函数以及它们的_word 和_long 函数都已经废弃了。2.0 和以后的内核在#idef WE_REALLY_WANT_TO_USE_A_BROKE_INTERFACE 里定义了这些函数。强烈要求程序员调用 put_user 和 get_user。

#include <linux/sched.h>

void interruptible_sleep_on(struct wait_queue **q);

void sleep_on(struct wait_queue **q);

调用两者其一就可以让当前进程睡眠。通常，你应该选择 interruptible 形式实现阻塞型读和写。

void wake_up(struct wait_queue **q);

void wake_up_interruptible(struct wait_queue **q);

这些函数唤醒睡在队列 q 上的进程。_interruptible 形式的函数仅唤醒可中断的进程。

void schedule(void);

这个从运行队列里挑选一个可运行的进程运行。所选的进程可以时 current 或其他进程。由于 sleep_on 函数在内部调用了 schedule，你通常不会直接调用这个函数。

void select_wait(struct wait_queue **wait_address, select_table *p);

这个将当前进程放到一个等待队列中，但不立即进行调度。它是设计用来实现设备驱动程序的 select 方法的。2.1.23 中完全修改了 select 的实现；详情可见第 17 章的“poll 方法”一节。

#include <linux/fs.h>

SEL_IN

SEL_OUT

SEL_EX

这些宏的某一个做为 mode 参数传递给设备的 select 方法。

int fasync_helper(struct inode *inode, struct file *filp, int mode, struct fasync_struct **fa);

这个函数时实现 fasync 设备方法的“助手”函数。mode 参数与传递给方法的值相同，而 fa 指向一个设备相关的 fasync_struct *变量。

void kill_fasync(struct fasync_struct *fa, int sig);

如果驱动程序支持异步触发，这个函数用来向注册到 fa 的进程发送信号。