第六章 高级字符驱动操作

在第三章中，我们构建了一个完整的设备驱动程序，用户可以进行读写操作。但是，真实的设备通常提供的功能不仅仅是同步读写。现在，我们已经配备了调试工具，以防出现问题，同时对并发问题有了深入的理解，这有助于防止问题的发生，因此我们可以安全地创建更高级的驱动程序。

本章将研究一些你需要理解的概念，以编写功能完备的字符设备驱动程序。我们首先实现ioctl系统调用，这是一种常用的设备控制接口。然后，我们将探讨与用户空间同步的各种方式；到本章结束时，你将对如何让进程休眠（并唤醒它们）、实现非阻塞I/O，以及在你的设备可供读写时如何通知用户空间有一个好的理解。我们最后将看看如何在驱动程序中实现几种不同的设备访问策略。

这里讨论的想法将通过修改版本的scull驱动程序进行演示。再次强调，所有的实现都使用内存虚拟设备，所以你可以自己尝试代码，而不需要有任何特定的硬件。到现在为止，你可能已经想要亲自动手实践真实的硬件，但这将要等到第9章。

6.1. ioctl 接口

除了能够读写设备外，大多数驱动程序还需要通过设备驱动程序执行各种类型的硬件控制。大多数设备可以执行超出简单数据传输的操作；用户空间通常需要能够请求，例如，设备锁定其门，弹出其媒体，报告错误信息，改变波特率，或自毁。这些操作通常通过ioctl方法支持，该方法实现了同名的系统调用。

- `ioctl`是一个在Unix和类Unix操作系统中的系统调用，它为设备特定的操作提供了一个接口。这个名字是“输入/输出控制”的缩写。

- 在设备驱动程序中，`ioctl`方法用于实现那些不适合用read和write系统调用来实现的操作。例如，改变设备的通信速率、改变设备的模式、查询设备状态等等。这些操作通常是设备特定的，因此`ioctl`方法的实现通常会根据设备的类型和需要进行控制的硬件特性而变化。

- 在用户空间，`ioctl`系统调用允许用户程序向设备驱动程序发送控制和配置命令。这些命令通常是设备特定的，因此用户程序需要知道设备驱动程序支持的特定`ioctl`命令。

- 用户空间的系统调用是操作系统提供给用户程序的一种接口，用户程序可以通过这种接口请求操作系统提供的服务。这些服务包括文件操作（如打开、读取、写入和关闭文件）、网络通信（如发送和接收数据）、进程管理（如创建和终止进程、等待进程结束）等。

- 系统调用是操作系统内核和用户空间程序之间的主要接口。当用户程序执行系统调用时，它会切换到内核模式，执行内核代码来完成请求的服务，然后返回到用户模式，继续执行用户程序的其余部分。

- 例如，当一个程序想要读取一个文件时，它会执行一个系统调用，请求操作系统打开并读取指定的文件。操作系统会检查请求，如果文件存在并且程序有权限读取它，操作系统会读取文件并将数据返回给程序。

- 系统调用提供了一种安全的方式，使得用户程序可以利用操作系统的功能，而不需要直接访问硬件或其他低级资源。

在用户空间，ioctl系统调用具有以下原型：

```c
int ioctl(int fd, unsigned long cmd, ...);
```

参数	描述
fd	文件描述符
cmd	交互协议，设备驱动将根据 cmd 执行对应操作
...	可变参数 arg，依赖 cmd 指定长度以及类型

ioctl() 函数执行成功时返回 0，失败则返回 -1 并设置全局变量 errorno 值
因此，在用户空间使用 ioctl 时，可以做如下的出错判断以及处理：

```c++
int ret;
ret = ioctl(fd, MYCMD);
if (ret == -1) {
    printf("ioctl: %s\n", strerror(errno));
}
```

在实际应用中，ioctl 最常见的 errorno 值为 ENOTTY（error not a typewriter），顾名思义，即第一个参数 fd 指向的不是一个字符设备，不支持 ioctl 操作，这时候应该检查前面的 open 函数是否出错或者设备路径是否正确
这个原型在Unix系统调用列表中很突出，因为其中的 `...`，通常标记函数具有可变数量的参数。然而，在真实的系统中，系统调用实际上不能有可变数量的参数。系统调用必须有一个明确定义的原型，因为用户程序只能通过硬件“门gates”访问它们。因此，原型中的点不代表可变数量的参数，而代表一个可选的参数，传统上被标识为 `char *argp`。点只是为了在编译期间防止类型检查。第三个参数的实际性质取决于正在发出的特定控制命令（第二个参数）。有些命令不需要参数，有些需要一个整数值，有些需要指向其他数据的指针。使用指针是将任意数据传递给ioctl调用的方式；然后设备能够与用户空间交换任意数量的数据。

ioctl调用的非结构化特性导致它在内核开发者中失去了青睐。每个ioctl命令本质上都是一个单独的，通常未记录的系统调用，没有办法以任何全面的方式审计这些调用。在所有系统上使非结构化的ioctl参数工作也很困难；例如，考虑在32位模式下运行的用户空间进程的64位系统。因此，有强烈的压力通过其他任何方式实现杂项控制操作。可能的替代方案包括将命令嵌入到数据流中（我们将在本章后面讨论这种方法）或使用虚拟文件系统，无论是sysfs还是驱动程序特定的文件系统。（我们将在第14章中看到sysfs。）然而，事实仍然是，对于真正的设备操作，ioctl通常是最简单和最直接的选择.

ioctl驱动方法有一个原型，与用户空间版本略有不同：

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

inode和filp指针是应用程序传递的文件描述符fd对应的值，它们是传递给open方法的相同参数。cmd参数不变地从用户传递，可选的arg参数以无符号长整型的形式传递，无论用户是否以整数或指针的形式给出。如果调用程序没有传递第三个参数，驱动操作接收到的arg值是未定义的。因为在额外参数上禁用了类型检查，所以如果向ioctl传递了无效的参数，编译器无法警告你，任何相关的bug都很难发现。

你可能想象到，大多数ioctl实现都由一个大的switch语句组成，根据cmd参数选择正确的行为。不同的命令有不同的数值，通常给予符号名称以简化编码。符号名称由预处理器定义分配。自定义驱动程序通常在它们的头文件中声明这样的符号；scull.h为scull声明它们。当然，用户程序也必须包含那个头文件，以便访问那些符号。

6.1.1. 选择 ioctl 命令

在编写ioctl的代码之前，你需要选择对应于命令的数字。许多程序员的第一反应是选择一组从0或1开始的小数字。然而，有充分的理由不这样做。ioctl命令的数字在整个系统中应该是唯一的，以防止由于向错误的设备发出正确命令而导致的错误。这种不匹配的情况并不少见，一个程序可能会发现自己试图改变一个非串口输入流（如FIFO或音频设备）的波特率。如果每个ioctl数字都是唯一的，应用程序会得到一个EINVAL错误，而不是成功地做一些无意的事情。

为了帮助程序员创建唯一的ioctl命令代码，这些代码已经被分割成几个位字段。Linux的第一个版本使用16位数字：顶部的八个是与设备相关的“魔术”数字，底部的八个是设备内的唯一序列号。这是因为Linus“一无所知”（他自己的话）；只有后来才想出了更好的位字段划分。不幸的是，相当多的驱动程序仍然使用旧的约定。他们必须这样做：改变命令代码会破坏无数的二进制程序，这是内核开发者不愿意做的事情。

根据Linux内核的约定为你的驱动程序选择ioctl数字，你应该首先检查include/asm/ioctl.h和Documentation/ioctl-number.txt。头文件定义了你将使用的位字段：类型（魔术数字）、序数、传输方向和参数大小。ioctl-number.txt文件列出了整个内核使用的魔术数字，所以你可以选择你自己的魔术数字并避免重叠。文本文件还列出了应该使用约定的原因。

定义ioctl命令数字的推荐方式使用了四个位字段，它们的含义如下。这个列表中引入的新符号在<linux/ioctl.h>中定义。

- 类型（type） 魔术数字。只需选择一个数字（在查阅ioctl-number.txt后）并在驱动程序中使用。这个字段宽8位（`_IOC_TYPEBITS`）。设备类型，占据 8 bit，在一些文献中翻译为 “幻数” 或者 “魔数”，可以为任意 char 型字符，例如‘a’、’b’、’c’ 等等，其主要作用是使 ioctl 命令有唯一的设备标识；

- 序号（number） 序数（顺序）数字。它宽8位(`_IOC_NRBITS`)。命令编号/序数，占据 8 bit，可以为任意 unsigned char 型数据，取值范围 0~255，如果定义了多个 ioctl 命令，通常从 0 开始编号递增；

- 方向（direction） 如果特定命令涉及数据传输，这是数据传输的方向。可能的值有`_IOC_NONE（无数据传输）`、`_IOC_READ`、`_IOC_WRITE`和`_IOC_READ|_IOC_WRITE（数据双向传输）`。数据传输是从应用程序的角度看的；`_IOC_READ`意味着从设备读取，所以驱动程序必须写入用户空间。注意，这个字段是一个位掩码，所以`_IOC_READ`和`_IOC_WRITE`可以使用逻辑AND操作提取。ioctl 命令访问模式（数据传输方向），占据 2 bit，可以为 `_IOC_NONE`、

`_IOC_READ`、`_IOC_WRITE`、`_IOC_READ | _IOC_WRITE`，分别指示了四种访问模式：无数据、读数据、写数据、读写数据；

- 大小（size） 涉及的用户数据的大小。这个字段的宽度取决于架构，但通常是13或14位。你可以在宏 `_IOC_SIZEBITS` 中找到你特定架构的值。使用大小字段并不是必须的——内核不会检查它——但这是个好主意。正确使用这个字段可以帮助检测用户空间的编程错误，并使你能够实现向后兼容性，如果你需要改变相关数据项的大小。然而，如果你需要更大的数据结构，你可以忽略大小字段。我们很快就会看到这个字段是如何使用的。涉及到 ioctl 函数 第三个参数 arg ， 占据 13bit 或者 14bit（体系相关，arm 架构一般为 14 位），指定了 arg 的数据类型及长度，如果在驱动的 ioctl 实现中不检查，通常可以忽略该参数；

在内核中，提供了宏接口以生成上述格式的 ioctl 命令：

```C++
// include/uapi/asm-generic/ioctl.h

#define _IOC(dir,type,nr,size) \
    (((dir)  << _IOC_DIRSHIFT) | \
     ((type) << _IOC_TYPESHIFT) | \
     ((nr)   << _IOC_NRSHIFT) | \
     ((size) << _IOC_SIZESHIFT))
```

被<linux/ioctl.h>包含的头文件<asm/ioctl.h>定义了帮助设置命令数字的宏，如下：**`_IO(type,nr)（对于没有参数的命令）`、`_IOR(type,nr,datatype)（用于从驱动程序读取数据）`、`_IOW(type,nr,datatype)（用于写入数据）`，和 `_IOWR(type,nr,datatype)（用于双向传输）`** 。类型和数字字段作为参数传递，大小字段通过对datatype参数应用sizeof得到。

头文件还定义了一些可能在你的驱动程序中用来解码数字的宏：`_IOC_DIR(nr)`、`_IOC_TYPE(nr)`、`_IOC_NR(nr)`和`_IOC_SIZE(nr)`。我们不会对这些宏进行更多的详细介绍，因为头文件已经很清楚了，稍后在这一部分会展示示例代码。

以下是在scull中定义的一些ioctl命令。特别是，这些命令设置和获取驱动程序的可配置参数。

```c
/* 使用 'k' 作为魔术数字 */
#define SCULL_IOC_MAGIC  'k'
/* 请在你的代码中使用不同的8位数字 */
#define SCULL_IOCRESET    _IO(SCULL_IOC_MAGIC, 0)
/*
 * S 表示通过指针"设置",
 * T 表示直接通过参数值"告知"
 * G 表示"获取": 通过设置指针进行回复
 * Q 表示"查询": 响应在返回值上
 * X 表示"交换": 原子性地切换 G 和 S
 * H 表示"移位": 原子性地切换 T 和 Q
 */
#define SCULL_IOCSQUANTUM _IOW(SCULL_IOC_MAGIC,  1, int)
#define SCULL_IOCSQSET    _IOW(SCULL_IOC_MAGIC,  2, int)
#define SCULL_IOCTQUANTUM _IO(SCULL_IOC_MAGIC,   3)
#define SCULL_IOCTQSET    _IO(SCULL_IOC_MAGIC,   4)
#define SCULL_IOCGQUANTUM _IOR(SCULL_IOC_MAGIC,  5, int)
#define SCULL_IOCGQSET    _IOR(SCULL_IOC_MAGIC,  6, int)
#define SCULL_IOCQQUANTUM _IO(SCULL_IOC_MAGIC,   7)
```

```
#define SCULL_IOCQQSET     _IO(SCULL_IOC_MAGIC,   8)

#define SCULL_IOCXQUANTUM _IOWR(SCULL_IOC_MAGIC, 9, int)

#define SCULL_IOCXQSET     _IOWR(SCULL_IOC_MAGIC,10, int)

#define SCULL_IOCHQUANTUM _IO(SCULL_IOC_MAGIC,  11)

#define SCULL_IOCHQSET     _IO(SCULL_IOC_MAGIC,  12)

#define SCULL_IOC_MAXNR 14
```

通常而言，为了方便会使用宏 _IOC() 衍生的接口来直接定义 ioctl 命令：

C++

```
// include/uapi/asm-generic/ioctl.h

/* used to create numbers */
#define _IO(type,nr)         _IOC(_IOC_NONE,(type),(nr),0)
#define _IOR(type,nr,size)  _IOC(_IOC_READ,(type),(nr),
(_IOC_TYPECHECK(size)))
#define _IOW(type,nr,size)  _IOC(_IOC_WRITE,(type),(nr),
(_IOC_TYPECHECK(size)))
#define _IOWR(type,nr,size) _IOC(_IOC_READ|_IOC_WRITE,
(type),(nr),(_IOC_TYPECHECK(size)))
```

```
_IO:        定义不带参数的 ioctl 命令
_IOW:       定义带写参数的 ioctl 命令 (copy_from_user)
_IOR:       定义带读参数的ioctl命令 (copy_to_user)
_IOWR:      定义带读写参数的 ioctl 命令
```

同时，内核还提供了反向解析 ioctl 命令的宏接口：

```C++
// include/uapi/asm-generic/ioctl.h
/* used to decode ioctl numbers */
#define _IOC_DIR(nr)        (((nr) >> _IOC_DIRSHIFT) &
_IOC_DIRMASK)
#define _IOC_TYPE(nr)       (((nr) >> _IOC_TYPESHIFT) &
_IOC_TYPEMASK)
#define _IOC_NR(nr)     (((nr) >> _IOC_NRSHIFT) &
_IOC_NRMASK)
#define _IOC_SIZE(nr)       (((nr) >> _IOC_SIZESHIFT) &
_IOC_SIZEMASK)
```

实际的源文件定义了一些这里没有显示的额外命令。我们选择实现两种传递整数参数的方式：通过指针和通过显式值（尽管，按照既定的约定，ioctl应该通过指针交换值）。同样，两种方式都用于返回一个整数数字：通过指针或通过设置返回值。只要返回值是正整数，这就能工作；如你现在所知，从任何系统调用返回时，正值是保留的（如我们在read和write中看到的），而负值被认为是错误，并用于在用户空间设置errno。

“交换”和“移位”操作对于scull来说并不特别有用。我们实现了“交换”来展示驱动程序如何将单独的操作组合成一个原子操作，以及“移位”来配对“告知”和“查询”。有时候，需要像这样的原子测试和设置操作，特别是当应用程序需要设置或释放锁时。

命令的显式序数没有特定的含义。它只是用来区分命令。实际上，你甚至可以为读命令和写命令使用相同的序数，因为实际的ioctl数字在“方向”位中是不同的，但是你没有理由这样做。我们选择不在声明之外的任何地方使用命令的序数，所以我们没有为它分配一个符号值。这就是为什么在前面给出的定义中出现了显式数字。这个例子展示了一种使用命令数字的方式，但你可以自由地以不同的方式来做。

除了少数预定义的命令（稍后将讨论）外，ioctl cmd参数的值目前并未被内核使用，而且在未来也很可能不会被使用。因此，如果你感到懒，你可以避免前面显示的复杂声明，并显式声明一组标量数字。另一方面，如果你这样做，你将无法从使用位字段中获益，并且如果你曾经提交你的代码以包含在主线内核中，你将遇到困难。头文件<linux/kd.h>是这种老式方法的一个例子，使用16位标量值来定义ioctl命令。那个源文件依赖于标量数字，因为它使用了当时遵守的约定，而不是出于懒惰。现在改变它会导致无谓的不兼容。

知 linux 内核 - ioctl 函数详解 - 知乎 (zhihu.com)
(https://zhuanlan.zhihu.com/p/553869814)

6.1.2. 返回值

ioctl的实现通常是基于命令号的switch语句。但是，当命令号不匹配有效操作时，应该选择默认选项是什么呢？这个问题存在争议。有几个内核函数返回-EINVAL（“无效的参数”），这是有道理的，因为命令参数确实不是一个有效的参数。然而，POSIX标准规定，如果发出了不适当的ioctl命令，那么应该返回-ENOTTY。这个错误代码被C库解释为“设备的ioctl不适当”，这通常正是程序员需要了解的信息。尽管如此，对于无效的ioctl命令返回-EINVAL仍然非常常见。

6.1.3. 预定义的命令

虽然ioctl系统调用最常用于操作设备，但内核也识别一些命令。请注意，当这些命令应用到你的设备时，它们在调用你自己的文件操作之前就被解码了。因此，如果你为你的ioctl命令选择了相同的编号，你将永远不会看到对该命令的任何请求，而且应用程序会因为ioctl编号之间的冲突而得到意外的结果。

预定义的命令分为三组：

- 可以对任何文件（常规文件、设备、FIFO或套接字）发出的命令
- 只对常规文件发出的命令
- 特定于文件系统类型的命令

最后一组命令由托管文件系统的实现执行（这就是chattr命令的工作方式）。设备驱动程序编写者只对第一组命令感兴趣，其魔术编号是“T”。查看其他组的工作原理留给读者作为练习；ext2_ioctl是一个非常有趣的函数（比人们可能预期的更容易理解），因为它实现了追加只读标志和不可变标志。

以下ioctl命令是为任何文件预定义的，包括设备特殊文件：

- `FIOCLEX` 设置close-on-exec标志（File IOctl CLose on EXec）。设置此标志会导致在调用进程执行新程序时关闭文件描述符。

- `FIONCLEX` 清除close-on-exec标志（File IOctl Not CLos on EXec）。该命令恢复常见的文件行为，撤销上面的FIOCLEX所做的操作。

- `FIOASYNC` 为文件设置或重置异步通知（如本章后面的“异步通知”部分所讨论的）。请注意，直到Linux 2.2.4版本的内核错误地使用此命令来修改O_SYNC标志。由于这两个操作都可以通过fcntl完成，所以实际上没有人使用FIOASYNC命令，这里只是为了完整性而报告。

- `FIOQSIZE` 此命令返回文件或目录的大小；但是，当应用于设备文件时，它会产生ENOTTY错误返回。

- `FIONBIO` “File IOctl Non-Blocking I/O”（在“阻塞和非阻塞操作”部分中描述）。此调用修改filp→f_flags中的O_NONBLOCK标志。系统调用的第三个参数用于指示是否要设置或清除该标志。（我们将在本章后面查看该标志的作用。）请注意，更改此标志的常见方式是使用fcntl系统调用，使用F_SETFL命令。

列表中的最后一项引入了一个新的系统调用，fcntl，它看起来像ioctl。实际上，fcntl调用与ioctl非常相似，因为它获取一个命令参数和一个额外的（可选的）参数。它主要是由于历史原因而与ioctl保持分离：当Unix开发者面临控制I/O操作的问题时，他们决定文件和设备是不同的。当时，唯一具有ioctl实现的设备是ttys，这就解释了为什么-ENOTTY是不正确的ioctl命令的标准回复。事情已经改变，但fcntl仍然是一个独立的系统调用。

6.1.4. 使用 ioctl 参数

在查看scull驱动程序的ioctl代码之前，我们需要讨论的另一点是如何使用额外的参数。如果它是一个整数，那么很容易：可以直接使用它。然而，如果它是一个指针，就必须小心。

当一个指针用于引用用户空间时，我们必须确保用户地址是有效的。试图访问未经验证的用户提供的指针可能会导致行为不正确，内核oops，系统损坏或安全问题。驱动程序有责任对其使用的每一个用户空间地址进行适当的检查，并在地址无效时返回错误。

在第3章中，我们查看了copy_from_user和copy_to_user函数，这些函数可以用来安全地将数据移动到用户空间和从用户空间移动数据。这些函数也可以在ioctl方法中使用，但是ioctl调用通常涉及到可以通过其他方式更有效地操作的小数据项。首先，地址验证（不传输数据）是由函数`access_ok`实现的，该函数在<asm/uaccess.h>中声明：

```
int access_ok(int type, const void *addr, unsigned long size);
```

第一个参数应该是VERIFY_READ或VERIFY_WRITE，取决于要执行的操作是读取**用户空间**内存区域还是写入它。addr参数持有一个用户空间地址，size是字节计数。例如，如果ioctl需要从用户空间读取一个整数值，size是sizeof(int)。如果你需要在给定地址读写，使用VERIFY_WRITE，因为它是VERIFY_READ的超集。

与大多数内核函数不同，access_ok返回一个布尔值：1表示成功（访问是OK的），0表示失败（访问不是OK的）。如果它返回false，驱动程序通常应该返回-EFAULT给调

用者。

关于access_ok有几点有趣的事情需要注意。首先，它并没有完成验证内存访问的全部工作；它只检查内存引用是否在进程可能合理访问的内存区域中。特别是，access_ok确保地址不指向内核空间内存。其次，大多数驱动程序代码实际上不需要调用access_ok。稍后描述的内存访问例程会为你处理这个问题。尽管如此，我们演示了它的使用，以便你可以看到它是如何完成的。

scull源代码利用ioctl号码中的位字段在switch之前检查参数：

```c
int err = 0, tmp;

int retval = 0;

/*

 * 提取类型和数字位字段，不解码

 * 错误的命令：在access_ok()之前返回ENOTTY（不适当的ioctl）

 */

if (_IOC_TYPE(cmd) != SCULL_IOC_MAGIC) return -ENOTTY;

if (_IOC_NR(cmd) > SCULL_IOC_MAXNR) return -ENOTTY;

/*

 * 方向是一个位掩码，VERIFY_WRITE捕获R/W

 * 传输。`direction'是面向用户的，而

 * access_ok是面向内核的，所以"read"和

 * "write"的概念是反转的

 */

if (_IOC_DIR(cmd) & _IOC_READ)

    err = !access_ok(VERIFY_WRITE, (void __user *)arg,
_IOC_SIZE(cmd));

else if (_IOC_DIR(cmd) & _IOC_WRITE)

    err =  !access_ok(VERIFY_READ, (void __user *)arg,
_IOC_SIZE(cmd));
```

```
    if (err) return -EFAULT;
```

调用access_ok后，驱动程序可以安全地执行实际的传输。除了`copy_from_user`和`copy_to_user`函数外，程序员还可以利用一组针对最常用数据大小（一、二、四和八字节）优化的函数。这些函数在下面的列表中描述，并在<asm/uaccess.h>中定义。

`put_user(datum, ptr)` `__put_user(datum, ptr)` 这些宏将数据写入用户空间；它们相对快速，当传输单个值时，应该调用它们而不是copy_to_user。这些宏已经被编写成允许将任何类型的指针传递给put_user，只要它是一个用户空间地址。数据传输的大小取决于ptr参数的类型，并在编译时使用sizeof和typeof编译器内置函数确定。因此，如果ptr是一个char指针，那么传输一个字节，对于两个、四个和可能的八个字节也是如此。

`put_user`检查以确保进程能够写入给定的内存地址。成功时返回0，错误时返回-EFAULT。`__put_user`执行较少的检查（它不调用access_ok），但如果指向的内存不可由用户写入，它仍然可能失败。因此，只有当内存区域已经通过access_ok验证过，`__put_user`才应该被使用。

作为一般规则，当你实现一个read方法，或者当你复制多个项目，因此，在第一次数据传输之前只调用一次access_ok，如上面对ioctl所示，你调用`__put_user`来节省几个周期。

`get_user(local, ptr)` `__get_user(local, ptr)` 这些宏用于从用户空间检索单个数据。它们的行为像`put_user`和`__put_user`，但是数据传输的方向相反。检索到的值存储在本地变量local中；返回值指示操作是否成功。同样，只有当地址已经通过access_ok验证过，`__get_user`才应该被使用。

如果试图使用上述函数之一传输一个不符合特定大小的值，结果通常是编译器的奇怪消息，如“请求转换为非标量类型”。在这种情况下，必须使用`copy_to_user`或`copy_from_user`。

6.1.5. 兼容性和受限操作

设备的访问权限由设备文件的权限控制，驱动程序通常不参与权限检查。然而，有些情况下，任何用户都被授予对设备的读/写权限，但是一些控制操作仍然应该被拒绝。例如，并非所有磁带驱动器的用户都应该能够设置其默认块大小，而被授予磁盘设备读/写

访问权限的用户可能仍然应该被拒绝格式化它的能力。在这些情况下，驱动程序必须进行额外的检查，以确保用户有能力执行请求的操作。

Unix系统传统上将特权操作限制在超级用户账户。这意味着特权是全有或全无的事情——超级用户可以做任何事情，但所有其他用户都受到严格限制。Linux内核提供了一个更灵活的系统，称为capabilities。基于capability的系统抛弃了全有或全无的模式，将特权操作分解为单独的子组。这样，特定的用户（或程序）可以被赋予执行特定的特权操作的能力，而不用放弃执行其他无关操作的能力。内核仅使用capabilities进行权限管理，并导出两个系统调用capget和capset，以允许它们从用户空间管理。

完整的capabilities集可以在<linux/capability.h>中找到。这些是系统所知道的唯一的capabilities；驱动程序作者或系统管理员无法定义新的capabilities，除非修改内核源代码。可能对设备驱动程序编写者感兴趣的那些capabilities的子集包括以下内容：

`CAP_DAC_OVERRIDE` 覆盖文件和目录上的访问限制（数据访问控制，或DAC）的能力。

`CAP_NET_ADMIN` 执行网络管理任务的能力，包括那些影响网络接口的任务。

`CAP_SYS_MODULE` 加载或移除内核模块的能力。

`CAP_SYS_RAWIO` 执行“原始”I/O操作的能力。例子包括访问设备端口或直接与USB设备通信。

`CAP_SYS_ADMIN` 一个全能的capability，提供许多系统管理操作的访问。

`CAP_SYS_TTY_CONFIG` 执行tty配置任务的能力。

在执行特权操作之前，设备驱动程序应检查调用进程是否具有适当的capability；如果没有这样做，可能会导致用户进程执行未经授权的操作，对系统稳定性或安全性产生不良影响。Capability检查是通过capable函数（在<linux/sched.h>中定义）执行的：

```c
int capable(int capability);
```

在scull示例驱动程序中，任何用户都被允许查询量子和量子集大小。然而，只有特权用户才能改变这些值，因为不适当的值可能会严重影响系统性能。在需要时，scull的ioctl实现会如下检查用户的特权级别：

```c
if (! capable (CAP_SYS_ADMIN))

    return -EPERM;
```

在没有更具体的capability用于此任务的情况下，选择了CAP_SYS_ADMIN进行此测试。

6.1.6. ioctl 命令的实现

scull的ioctl实现只传输设备的可配置参数，如下所示：

```c
switch(cmd) {

  case SCULL_IOCRESET:

    scull_quantum = SCULL_QUANTUM;

    scull_qset = SCULL_QSET;

    break;

  case SCULL_IOCSQUANTUM: /* Set: arg points to the value
*/

    if (! capable (CAP_SYS_ADMIN))

        return -EPERM;

    retval = __get_user(scull_quantum, (int __user
*)arg);

    break;

  case SCULL_IOCTQUANTUM: /* Tell: arg is the value */

    if (! capable (CAP_SYS_ADMIN))

        return -EPERM;

    scull_quantum = arg;

    break;

  case SCULL_IOCGQUANTUM: /* Get: arg is pointer to
result */

    retval = __put_user(scull_quantum, (int __user
*)arg);
```

```
    break;

  case SCULL_IOCQQUANTUM: /* Query: return it (it's
positive) */

    return scull_quantum;

  case SCULL_IOCXQUANTUM: /* eXchange: use arg as pointer
*/

    if (! capable (CAP_SYS_ADMIN))

    return -EPERM;

    tmp = scull_quantum;

    retval = __get_user(scull_quantum, (int __user
*)arg);

    if (retval == 0)

        retval = __put_user(tmp, (int __user *)arg);

    break;

  case SCULL_IOCHQUANTUM: /* sHift: like Tell + Query */

    if (! capable (CAP_SYS_ADMIN))

        return -EPERM;

    tmp = scull_quantum;

    scull_quantum = arg;

    return tmp;

  default:  /* redundant, as cmd was checked against
MAXNR */
```

```
    return -ENOTTY;

}

return retval;
```

scull还包括六个作用于scull_qset的条目。这些条目与scull_quantum的条目相同，因此没有必要在此打印。
从调用者（即从用户空间）的角度看，传递和接收参数的六种方式如下：

```c
int quantum;

ioctl(fd,SCULL_IOCSQUANTUM, &quantum);          /* 通过指针设置 */

ioctl(fd,SCULL_IOCTQUANTUM, quantum);           /* 通过值设置 */

ioctl(fd,SCULL_IOCGQUANTUM, &quantum);          /* 通过指针获取 */

quantum = ioctl(fd,SCULL_IOCQQUANTUM);          /* 通过返回值获取 */

ioctl(fd,SCULL_IOCXQUANTUM, &quantum);          /* 通过指针交换 */

quantum = ioctl(fd,SCULL_IOCHQUANTUM, quantum); /* 通过值交换 */
```

当然，正常的驱动程序不会实现这样的混合调用模式。我们在这里这样做只是为了演示事情可以以不同的方式完成。然而，通常，数据交换会一致地执行，要么通过指针，要么通过值，避免两种技术的混合。

6.1.7. 不用 ioctl 的设备控制

有时，通过向设备本身写入控制序列来控制设备会更好。例如，控制台驱动程序就使用了这种技术，其中所谓的转义序列用于移动光标、更改默认颜色或执行其他配置任务。这种方式实现设备控制的好处是，用户只需通过写入数据就可以控制设备，无需使用（或有时编写）专门用于配置设备的程序。当设备可以以这种方式被控制时，发出命令的程序甚至不需要在控制设备的同一系统上运行。

例如，setterm程序通过打印转义序列来对控制台（或其他终端）进行配置。控制程序可以在与被控设备不同的计算机上运行，因为简单的数据流重定向就可以完成配置工作。每次你运行一个远程tty会话时，都会发生这种情况：转义序列在远程打印，但会影响本地tty；这种技术并不仅限于tty。

通过打印进行控制的缺点是，它给设备添加了策略约束；例如，只有当你确定在正常操作期间写入设备的数据中不会出现控制序列时，这种方法才可行。对于tty来说，这只是部分正确。虽然文本显示器只显示ASCII字符，但有时控制字符可能会在写入的数据中滑过，因此，可能会影响控制台的设置。例如，当你将一个二进制文件cat到屏幕上时，可能会发生这种情况；结果可能包含任何内容，你经常会在控制台上看到错误的字体。

对于那些不传输数据，只响应命令的设备，如机器人设备，通过写入进行控制绝对是正确的方式。

例如，你的一位作者为了乐趣编写的一个驱动程序可以在两个轴上移动相机。在这个驱动程序中，“设备”只是一对旧的步进电机，它们不能真正地被读取或写入。对步进电机“发送数据流”的概念几乎没有意义。在这种情况下，驱动程序将被写入的内容解释为ASCII命令，并将请求转换为操纵步进电机的脉冲序列。这个想法在某种程度上类似于你发送给调制解调器的AT命令，以设置通信，主要的区别是用于与调制解调器通信的串行端口也必须传输真实的数据。直接设备控制的优点是，你可以使用cat移动相机，而无需编写和编译特殊的代码来发出ioctl调用。

在编写命令导向的驱动程序时，没有理由实现ioctl方法。在解释器中添加一个额外的命令更容易实现和使用。然而，有时你可能选择反过来做：而不是将write方法变成一个解释器并避免使用ioctl，你可能选择完全避免使用write，并仅使用ioctl命令，同时附带一个特定的命令行工具来向驱动程序发送这些命令。这种方法将复杂性从内核空间移动到用户空间，可能更容易处理，并有助于保持驱动程序的小型化，同时拒绝使用简单的cat或echo命令。

6.2. 阻塞 I/O

在第3章中，我们研究了如何实现驱动程序的read和write方法。然而，我们跳过了一个重要问题：如果驱动程序不能立即满足请求，它应该如何响应？当没有数据可用，但未来预期有更多数据时，可能会调用read。或者，一个进程可能试图写入，但你的设备还没有准备好接受数据，因为你的输出缓冲区已满。调用进程通常不关心这些问题；程序员只是期望调用read或write，并在完成必要的工作后返回调用。因此，在这种情况下，你的驱动程序应该（默认）阻塞进程，让它睡眠，直到请求可以继续。

本节将展示如何让进程睡眠，并在稍后再唤醒它。然而，像往常一样，我们首先需要解释一些概念。

6.2.1. 睡眠的介绍

当一个进程“睡眠”时，它被标记为处于特殊状态，并从调度器的运行队列中移除。直到有什么东西来改变这个状态，这个进程将不会在任何CPU上被调度，因此，它将不会运行。一个睡眠的进程已经被推到系统的一边，等待未来的某个事件发生。

让一个进程睡眠对于一个Linux设备驱动程序来说是一件容易的事情。然而，有几个规则你必须记住，以便以安全的方式编写睡眠代码。

这些规则的第一个是：在你运行在原子上下文中时，永远不要睡眠。我们在第5章介绍了原子操作；一个原子上下文就是一个状态，其中多个步骤必须在没有任何并发访问的情况下执行。这意味着，关于睡眠，你的驱动程序不能在持有自旋锁、序列锁或RCU锁的时候睡眠。如果你禁用了中断，你也不能睡眠。在持有信号量的时候睡眠是合法的，但你应该仔细查看任何这样做的代码。如果代码在持有信号量的时候睡眠，任何等待那个信号量的其他线程也会睡眠。所以，任何在持有信号量的时候发生的睡眠都应该是短暂的，你应该确信，通过持有信号量，你不会阻塞最终会唤醒你的进程。

另一件需要记住的事情是，当你醒来时，你永远不知道你的进程可能已经离开CPU多久，或者在此期间可能发生了什么变化。你通常也不知道是否有其他进程可能因为同样的事件而睡眠；那个进程可能在你之前醒来，并抓取你正在等待的资源。最终的结果是，你在醒来后不能对系统的状态做任何假设，你必须检查你正在等待的条件是否确实成立。

另一个相关的点，当然，是你的进程不能睡眠，除非它确保其他地方的某个人会唤醒它。做唤醒的代码也必须能够找到你的进程才能完成它的工作。确保唤醒发生是通过思考你的代码和知道，对于每个睡眠，究竟是什么事件序列将结束那个睡眠。而让你的睡眠进程能够被找到，是通过一个叫做等待队列的数据结构来实现的。等待队列就像它听起来的那样：一系列的进程，都在等待一个特定的事件。

在Linux中，等待队列是通过一个“等待队列头”来管理的，这是一个类型为`wait_queue_head_t`的结构，定义在<linux/wait.h>中。一个等待队列头可以静态地定义和初始化：

```c
DECLARE_WAIT_QUEUE_HEAD(name);
```

或者动态地如下：

```c
wait_queue_head_t my_queue;

init_waitqueue_head(&my_queue);
```

我们将很快回到等待队列的结构，但我们现在已经知道足够多的信息来首次查看睡眠和唤醒。

6.2.2. 简单睡眠

在Linux内核中，进程在等待某个条件成立时会进入睡眠状态。我们之前提到过，任何进入睡眠状态的进程在唤醒后都必须检查它等待的条件是否真的成立。Linux内核中最简单的睡眠方式是一个叫做wait_event的宏（有几个变体），它将睡眠的细节和检查进程等待的条件结合在一起。wait_event的形式有：

- `wait_event(queue, condition)`
- `wait_event_interruptible(queue, condition)`
- `wait_event_timeout(queue, condition, timeout)`
- `wait_event_interruptible_timeout(queue, condition, timeout)`

在所有的形式中，queue是要使用的等待队列头。注意它是“按值”传递的。condition是一个在宏中睡眠前后评估的任意布尔表达式；直到condition评估为真值，进程才会继续睡眠。注意，condition可能会被评估多次，所以它不应该有任何副作用。

如果你使用wait_event，你的进程会被置入一个不可中断的睡眠状态，我们之前提到过，这通常不是你想要的。更好的选择是wait_event_interruptible，它可以被信号中

断。这个版本返回一个整数值，你应该检查这个值；一个非零值意味着你的睡眠被某种信号中断，你的驱动程序可能应该返回-ERESTARTSYS。

最后的版本（wait_event_timeout和wait_event_interruptible_timeout）等待一个有限的时间；在那个时间段（用jiffies表示，我们将在第7章讨论）过后，无论condition如何评估，宏都会返回一个值为0的结果。

当然，另一半的情况是唤醒。其他的执行线程（可能是一个不同的进程，或者是一个中断处理器）必须为你执行唤醒操作，因为你的进程当然是在睡眠状态。唤醒睡眠进程的基本函数叫做wake_up。它有几种形式（但我们现在只看两种）：

- `void wake_up(wait_queue_head_t *queue);`
- `void wake_up_interruptible(wait_queue_head_t *queue);`

wake_up会唤醒在给定队列上等待的所有进程（尽管情况比这复杂一些，我们稍后会看到）。另一种形式（wake_up_interruptible）只限于执行可中断睡眠的进程。一般来说，这两者是无法区分的（如果你使用的是可中断的睡眠）；实际上，约定是如果你使用wait_event就使用wake_up，如果你使用wait_event_interruptible就使用wake_up_interruptible。

现在我们已经知道了足够多的关于睡眠和唤醒的简单例子。在样本源码中，你可以找到一个叫做sleepy的模块。它实现了一个设备的简单行为：任何试图从设备读取的进程都会被置入睡眠状态。每当有进程写入设备，所有睡眠的进程都会被唤醒。这种行为是通过以下的读和写方法实现的。

```c
static DECLARE_WAIT_QUEUE_HEAD(wq);
static int flag = 0;

ssize_t sleepy_read (struct file *filp, char __user *buf,
size_t count, loff_t *pos)
{
        printk(KERN_DEBUG "process %i (%s) going to
sleep\n",
                current->pid, current->comm);
        wait_event_interruptible(wq, flag != 0);
        flag = 0;
        printk(KERN_DEBUG "awoken %i (%s)\n", current-
>pid, current->comm);
        return 0; /* EOF */
}
ssize_t sleepy_write (struct file *filp, const char
__user *buf, size_t count, loff_t *pos)
{
        printk(KERN_DEBUG "process %i (%s) awakening the
readers...\n",
                current->pid, current->comm);
        flag = 1;
        wake_up_interruptible(&wq);
        return count; /* succeed, to avoid retrial */

}
```

注意在这个例子中使用了标志变量。由于wait_event_interruptible检查的是一个必须变为真的条件，我们使用标志来创建这个条件。如果在调用sleepy_write时有两个进程在等待，会发生什么情况，这是值得思考的。由于sleepy_read一旦唤醒就会将标志重置为0，你可能会认为第二个唤醒的进程会立即回到睡眠状态。在单处理器系统上，这几乎总是会发生。但是，理解为什么你不能依赖这种行为是很重要的。
wake_up_interruptible调用会使两个睡眠的进程都唤醒。他们完全有可能在任何一个有机会重置它之前都注意到标志是非零的。对于这个微不足道的模块，这种竞态条件并不重要。在真实的驱动程序中，这种竞态可能会导致难以诊断的罕见崩溃。如果正确的操

作需要确保只有一个进程看到非零值，那么就必须以原子方式进行测试。我们将在不久的将来看到真实的驱动程序如何处理这种情况。但首先，我们还有另一个主题要讨论

6.2.3. 阻塞和非阻塞操作

在我们查看完整功能的读写方法的实现之前，我们需要讨论的最后一点是决定何时让进程进入睡眠状态。有时候，为了实现正确的Unix语义，即使操作不能完全执行，也需要不阻塞。

也有时候，调用进程会告诉你，无论其I/O是否能够取得任何进展，它都不想阻塞。明确的非阻塞I/O由`filp->f_flags`中的`O_NONBLOCK`标志表示。该标志在<linux/fcntl.h>中定义，该文件会被<linux/fs.h>自动包含。该标志的名字来源于“open-nonblock”，因为它可以在打开时指定（并且最初只能在那里指定）。如果你浏览源代码，你会发现一些对O_NDELAY标志的引用；这是O_NONBLOCK的另一个名字，为了与System V代码兼容而接受。默认情况下，该标志被清除，因为进程等待数据的正常行为就是睡眠。在阻塞操作的情况下，这是默认的，应该实现以下行为以遵循标准语义：

- 如果进程调用read但没有数据（尚）可用，进程必须阻塞。一旦有数据到达，进程就会被唤醒，并将数据返回给调用者，即使返回的数据少于方法的count参数所请求的数量。
- 如果进程调用write并且缓冲区没有空间，进程必须阻塞，并且必须在与用于读取的等待队列不同的等待队列上。当一些数据被写入硬件设备，并且输出缓冲区中变得有空闲空间时，进程被唤醒，write调用成功，尽管如果缓冢区中没有请求的count字节的空间，数据可能只被部分写入。

这两个声明都假设存在输入和输出缓冲区；实际上，几乎每个设备驱动程序都有它们。输入缓冲区是必需的，以避免在没有人读取时丢失到达的数据。相反，写入的数据不能丢失，因为如果系统调用不接受数据字节，它们将保留在用户空间缓冲区中。即便如此，输出缓冲区几乎总是对于从硬件中获取更多性能有用。

在驱动程序中实现输出缓冲区的性能提升来自于上下文切换和用户级/内核级转换的减少。没有输出缓冲区（假设设备慢），每个系统调用只接受一个或几个字符，当一个进程在写入时睡眠，另一个进程运行（这是一个上下文切换）。当第一个进程被唤醒时，它恢复（另一个上下文切换），写入返回（内核/用户转换），并且进程重新迭代系统调用以写入更多数据（用户/内核转换）；调用阻塞并且循环继续。添加输出缓冲区允许驱动程序在每次写入调用时接受更大的数据块，从而相应地提高性能。如果该缓冲区足够大，写入调用在第一次尝试时就会成功——缓冲的数据稍后会被推送到设备——无需将控制权返回到用户空间进行第二次或第三次写入调用。选择适合的输出缓冲区大小显然是设备特定的。

我们在scull中不使用输入缓冲区，因为当读取发出时，数据已经可用。同样，也不使用输出缓冲区，因为数据只是被复制到与设备关联的内存区域。本质上，设备就是一个缓冲区，所以实现额外的缓冲区是多余的。我们将在第10章中看到缓冲区的使用。

如果指定了O_NONBLOCK，read和write的行为就会不同。在这种情况下，如果进程在没有数据可用时调用read，或者在缓冲区没有空间时调用write，调用只会返回-EAGAIN（“再试一次”）。

你可能会预期，非阻塞操作立即返回，允许应用程序轮询数据。在处理非阻塞文件时，应用程序在使用stdio函数时必须小心，因为它们很容易将非阻塞返回误认为是EOF。它们总是需要检查errno。

自然，O_NONBLOCK在open方法中也是有意义的。这发生在调用实际上可以阻塞很长时间的情况下；例如，当打开（用于读取访问）没有（尚）写入者的FIFO，或者访问有待处理锁的磁盘文件。通常，打开设备要么成功，要么失败，无需等待外部事件。然而，有时候，打开设备需要长时间的初始化，你可能选择在你的open方法中支持O_NONBLOCK，如果设置了该标志，在启动设备初始化过程后立即返回-EAGAIN。驱动程序也可以实现阻塞打开以支持类似于文件锁的访问策略。我们将在本章后面的“阻塞打开作为EBUSY的替代方案”一节中看到这样的实现。

一些驱动程序也可能为O_NONBLOCK实现特殊的语义；例如，打开磁带设备通常会阻塞，直到插入磁带。如果用O_NONBLOCK打开磁带驱动器，无论媒体是否存在，打开都会立即成功。

只有read、write和open文件操作受到非阻塞标志的影响。

6.2.4. 一个阻塞 I/O 的例子

最后，我们来看一个实现阻塞I/O的真实驱动方法的例子。这个例子取自scullpipe驱动程序；它是scull的一种特殊形式，实现了类似管道的设备。

在驱动程序中，当数据到达时，被阻塞在read调用中的进程会被唤醒；通常，硬件会发出一个中断信号来表示这样的事件，驱动程序在处理中断的过程中唤醒等待的进程。scullpipe驱动程序的工作方式不同，因此它可以在不需要任何特定硬件或中断处理程序的情况下运行。我们选择使用另一个进程来生成数据并唤醒读取进程；同样，读取进程被用来唤醒等待缓冲区空间变得可用的写入进程。

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。缓冲区的大小可以以通常的方式（在编译时、加载时或运行时）进行配置。

```c
struct scull_pipe {

        wait_queue_head_t inq, outq;       /* 读和写队列
*/

        char *buffer, *end;                /* 缓冲区的开
始，缓冲区的结束 */

        int buffersize;                    /* 用于指针运算
*/

        char *rp, *wp;                     /* 读取的位置，
写入的位置 */

        int nreaders, nwriters;            /* 读/写打开的数
量 */

        struct fasync_struct *async_queue; /* 异步读取器
*/

        struct semaphore sem;              /* 互斥信号量
*/

        struct cdev cdev;                  /* 字符设备结构
*/

};
```

在这个结构中，`inq` 和 `outq` 是等待队列，用于阻塞和唤醒读取和写入进程。`buffer` 和 `end` 是缓冲区的开始和结束。`buffersize` 用于指针运算。`rp` 和 `wp` 是读取和写入的位置。`nreaders` 和 `nwriters` 是读取和写入的打开数量。`async_queue` 是异步读取器。`sem` 是互斥信号量，用于保护这个结构的数据。`cdev` 是字符设备结构，用于注册和注销设备。

这段代码是scull_p_read函数的实现，它管理了阻塞和非阻塞输入。代码如下：

```c
static ssize_t scull_p_read (struct file *filp, char
__user *buf, size_t count, loff_t *f_pos) {

    struct scull_pipe *dev = filp->private_data;

    if (down_interruptible(&dev->sem))

        return -ERESTARTSYS;

    while (dev->rp == dev->wp) { /* 没有数据可读 */

        up(&dev->sem); /* 释放锁 */

        if (filp->f_flags & O_NONBLOCK)

            return -EAGAIN;

        PDEBUG("\"%s\" reading: going to sleep\n",
current->comm);

        if (wait_event_interruptible(dev->inq, (dev->rp
!= dev->wp)))

            return -ERESTARTSYS; /* 信号：告诉文件系统层处理
它 */

        /* 否则循环，但首先重新获取锁 */

        if (down_interruptible(&dev->sem))

            return -ERESTARTSYS;

    }

    /* 好的，数据在那里，返回一些东西 */

    if (dev->wp > dev->rp)
```

```
            count = min(count, (size_t)(dev->wp - dev->rp));
        else /* 写指针已经回绕，返回到dev->end的数据 */
            count = min(count, (size_t)(dev->end - dev->rp));
        if (copy_to_user(buf, dev->rp, count)) {
            up (&dev->sem);
            return -EFAULT;
        }
        dev->rp += count;
        if (dev->rp == dev->end)
            dev->rp = dev->buffer; /* 回绕 */
        up (&dev->sem);
        /* 最后，唤醒任何写入者并返回 */
        wake_up_interruptible(&dev->outq);
        PDEBUG("\"%s\" did read %li bytes\n",current->comm,
    (long)count);
        return count;
    }
```

如你所见，我们在代码中留下了一些PDEBUG语句。当你编译驱动程序时，你可以启用消息，使得更容易跟踪不同进程的交互。

让我们仔细看看 `scull_p_read` 如何处理等待数据的情况。while循环在持有设备信号量的情况下测试缓冲区。如果有数据，我们知道我们可以立即将其返回给用户，无需睡

眠，所以整个循环体都会被跳过。如果缓冲区为空，我们必须睡眠。但在我们可以这样做之前，我们必须放下设备信号量；如果我们在持有它的情况下睡眠，没有任何写入者会有机会唤醒我们。一旦信号量被放下，我们会快速检查用户是否请求了非阻塞I/O，并在是的情况下返回。否则，就是时候调用`wait_event_interruptible`了。

一旦我们通过了那个调用，有些东西唤醒了我们，但我们不知道是什么。一种可能性是进程收到了一个信号。包含`wait_event_interruptible`调用的if语句检查了这种情况。这个语句确保了对信号的正确和预期的反应，这些信号可能是唤醒进程的原因（因为我们处于可中断的睡眠状态）。如果一个信号已经到达并且它没有被进程阻塞，正确的行为是让内核的上层处理事件。为此，驱动程序返回`-ERESTARTSYS`给调用者；这个值被虚拟文件系统（VFS）层内部使用，它要么重新启动系统调用，要么返回`-EINTR`给用户空间。我们使用同样类型的检查来处理每个读和写实现的信号处理。

然而，即使没有信号，我们也还不确定是否有数据可以获取。其他人也可能在等待数据，他们可能会赢得竞争并首先获取数据。所以我们必须再次获取设备信号量；只有这样我们才能再次测试读缓冲区（在while循环中），并真正知道我们可以将缓冲区中的数据返回给用户。所有这些代码的最终结果是，当我们从while循环退出时，我们知道信号量被持有，缓冲区包含我们可以使用的数据。

为了完整性，让我们注意到`scull_p_read`在我们获取设备信号量后的另一个地方也可以睡眠：调用`copy_to_user`。如果scull在复制内核和用户空间的数据时睡眠，它会在持有设备信号量的情况下睡眠。在这种情况下持有信号量是合理的，因为它不会使系统死锁（我们知道内核会执行复制到用户空间并在过程中不试图锁定同一个信号量的操作来唤醒我们），并且因为在驱动程序睡眠时设备内存数组不改变是很重要的。

6.2.5. 高级睡眠

我们已经介绍的功能，许多驱动程序都能满足其睡眠需求。然而，有些情况需要我们更深入地理解Linux等待队列机制的工作原理。复杂的锁定或性能需求可能会迫使驱动程序使用更低级的函数来实现睡眠。在这一部分，我们将深入研究这个更低的级别，以理解当进程睡眠时，实际上发生了什么.

6.2.5.1. 一个进程如何睡眠

如果你查看 `<linux/wait.h>`，你会发现 `wait_queue_head_t` 类型背后的数据结构非常简单；它由一个自旋锁和一个链表组成。添加到该列表的是一个等待队列条目，该条目用 `wait_queue_t` 类型声明。这个结构包含了关于睡眠进程的信息，以及它希望被唤醒的具体方式。

通常，让进程进入睡眠的第一步是分配和初始化一个 `wait_queue_t` 结构，然后将其添加到适当的等待队列中。当一切就绪后，负责唤醒的人将能够找到正确的进程。

下一步是设置进程的状态，标记它正在睡眠。在 `<linux/sched.h>` 中定义了几种任务状态。 `TASK_RUNNING` 表示进程能够运行，尽管它并不一定在任何特定时刻在处理器中执行。有两种状态表示进程正在睡眠：
`TASK_INTERRUPTIBLE` 和 `TASK_UNINTERRUPTIBLE`；它们当然对应于两种类型的睡眠。其他状态通常不会关注驱动程序编写者。

在 2.6 内核中，驱动程序代码通常不需要直接操作进程状态。然而，如果你需要这样做，应使用的调用是：

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

在旧的代码中，你经常会看到这样的代码：

```c
current->state = TASK_INTERRUPTIBLE;
```

但是，不鼓励以这种方式直接更改 current；当数据结构发生变化时，这样的代码容易出错。然而，上述代码确实显示，仅仅改变进程的当前状态并不能使其进入睡眠。通过改变当前状态，你改变了调度器对进程的处理方式，但你还没有放弃处理器。

放弃处理器是最后一步，但在此之前还有一件事要做：你必须先检查你正在等待的条件。如果不进行这个检查，就会引发竞态条件；如果在你进行上述过程时条件成立，而其他线程刚刚试图唤醒你，会发生什么？你可能会完全错过唤醒，睡眠的时间比你预期的要长。因此，在睡眠的代码中，你通常会看到类似以下的内容：

```c
if (!condition)
    schedule( );
```

在设置进程状态后检查我们的条件，我们可以应对所有可能的事件序列。如果我们正在等待的条件在设置进程状态之前就已经出现，我们在这个检查中会注意到，而不会真的睡眠。如果唤醒发生在此之后，无论我们是否已经睡着，进程都会变为可运行状态。

当然，调用 schedule 是调用调度器并放弃 CPU 的方式。每当你调用这个函数，你都在告诉内核考虑哪个进程应该运行，并在必要时切换到那个进程。所以你永远不知道 schedule 会在多久后返回到你的代码。

在 if 测试和可能的调用（以及从）schedule 返回后，还有一些清理工作要做。由于代码不再打算睡眠，它必须确保任务状态被重置为 `TASK_RUNNING`。如果代码刚从 schedule 返回，这一步是不必要的；该函数不会返回，直到进程处于可运行状态。但是，如果因为不再需要睡眠而跳过了对 schedule 的调用，进程状态将是错误的。还需要从等待队列中移除进程，否则可能会被唤醒多次。

6.2.5.2. 手动睡眠

在 Linux 内核的早期版本中，非平凡的睡眠需要程序员手动处理所有上述步骤。这是一个繁琐的过程，涉及大量容易出错的样板代码。如果他们愿意，程序员仍然可以以这种方式编写手动睡眠；`<linux/sched.h>` 包含所有必需的定义，内核源代码中充满了示例。然而，还有一种更简单的方法。

第一步是创建和初始化等待队列条目。这通常是通过这个宏完成的：

```c
DEFINE_WAIT(my_wait);
```

其中 name 是等待队列条目变量的名称。你也可以分两步进行：

```c
wait_queue_t my_wait;

init_wait(&my_wait);
```

但是，通常在实现你的睡眠的循环的顶部放一个 `DEFINE_WAIT` 行更容易。

下一步是将你的等待队列条目添加到队列，并设置进程状态。这两个任务都由这个函数处理：

```c
void prepare_to_wait(wait_queue_head_t *queue,
                     wait_queue_t *wait,
                     int state);
```

这里，queue 和 wait 分别是等待队列头和进程条目。state 是进程的新状态；它应该是 `TASK_INTERRUPTIBLE`（对于可中断的睡眠，这通常是你想要的）或 `TASK_UNINTERRUPTIBLE`（对于不可中断的睡眠）。

在调用 `prepare_to_wait` 之后，进程可以调用 `schedule`——在它确认仍然需要等待之后。一旦 `schedule` 返回，就是清理时间。这个任务也由一个特殊的函数处理：

```c
void finish_wait(wait_queue_head_t *queue, wait_queue_t
*wait);
```

此后，你的代码可以测试其状态，看看是否需要再次等待。

我们远远超过了需要一个例子的时候。之前我们看了 `scullpipe` 的 `read` 方法，它使用 `wait_event`。同一个驱动程序中的 `write` 方法则使用 `prepare_to_wait` 和 `finish_wait` 来等待。通常你不会在单个驱动程序中这样混合方法，但我们这样做是为了能够展示处理睡眠的两种方式。
首先，为了完整性，让我们看一下 `write` 方法本身：

```c
/* How much space is free? */

static int spacefree(struct scull_pipe *dev)

{

    if (dev->rp == dev->wp)

        return dev->buffersize - 1;

    return ((dev->rp + dev->buffersize - dev->wp) % dev-
>buffersize) - 1;

}

static ssize_t scull_p_write(struct file *filp, const
char __user *buf, size_t count,

                loff_t *f_pos)

{

    struct scull_pipe *dev = filp->private_data;

    int result;

    if (down_interruptible(&dev->sem))

        return -ERESTARTSYS;

        /* Make sure there's space to write */

    result = scull_getwritespace(dev, filp);

    if (result)

        return result; /* scull_getwritespace called
up(&dev->sem) */
```

```
    /* ok, space is there, accept something */

    count = min(count, (size_t)spacefree(dev));

    if (dev->wp >= dev->rp)

        count = min(count, (size_t)(dev->end - dev->wp));
/* to end-of-buf */

    else /* the write pointer has wrapped, fill up to rp-
1 */

        count = min(count, (size_t)(dev->rp - dev->wp -
1));

    PDEBUG("Going to accept %li bytes to %p from %p\n",
(long)count, dev->wp, buf);

    if (copy_from_user(dev->wp, buf, count)) {

        up (&dev->sem);

        return -EFAULT;

    }

    dev->wp += count;

    if (dev->wp == dev->end)

        dev->wp = dev->buffer; /* wrapped */

    up(&dev->sem);

    /* finally, awake any reader */

    wake_up_interruptible(&dev->inq);  /* blocked in
read( ) and select( ) */
```

```
    /* and signal asynchronous readers, explained late in
chapter 6 */

    if (dev->async_queue)

        kill_fasync(&dev->async_queue, SIGIO, POLL_IN);

    PDEBUG("\"%s\" did write %li bytes\n",current->comm,
(long)count);

    return count;

}
```

这段代码看起来类似于 `read` 方法，只是我们将睡眠的代码推入了一个名为 `scull_getwritespace` 的单独函数。它的工作是确保缓冲区有新数据的空间，如果需要，可以睡眠，直到该空间可用。一旦空间存在，`scull_p_write` 可以简单地将用户的数据复制到那里，调整指针，并唤醒任何可能在等待读取数据的进程。

处理实际睡眠的代码是：

```c
/* Wait for space for writing; caller must hold device
semaphore.  On

 * error the semaphore will be released before returning.
*/

static int scull_getwritespace(struct scull_pipe *dev,
struct file *filp)

{

    while (spacefree(dev) == 0) { /* full */

        DEFINE_WAIT(wait);

        up(&dev->sem);

        if (filp->f_flags & O_NONBLOCK)

            return -EAGAIN;

            PDEBUG("\"%s\" writing: going to
sleep\n",current->comm);

        prepare_to_wait(&dev->outq, &wait,
TASK_INTERRUPTIBLE);

        if (spacefree(dev) == 0)

            schedule( );

        finish_wait(&dev->outq, &wait);

        if (signal_pending(current))

            return -ERESTARTSYS;
            /* signal: tell the fs layer to handle it */
```

```
            if (down_interruptible(&dev->sem))
                return -ERESTARTSYS;
        }
        return 0;
    }
```

再次注意包含的 `while` 循环。如果没有睡眠就有空间可用，这个函数就简单地返回。否则，它必须放下设备信号量并等待。

代码使用 `DEFINE_WAIT` 来设置一个等待队列条目，并使用 `prepare_to_wait` 来准备实际的睡眠。然后是对缓冲区的必要检查；我们必须处理这样的情况：在我们进入 `while` 循环（并放下信号量）但在我们将自己放入等待队列之前，缓冲区中的空间变得可用。没有这个检查，如果读取进程能够在那段时间内完全清空缓冲区，我们可能会错过唯一的唤醒，永远睡着。确认我们必须睡眠后，我们可以调用 `schedule`。

值得再看一下这种情况：如果在 `if` 语句的测试和调用 `schedule` 之间发生唤醒会发生什么？在这种情况下，一切都好。唤醒将进程状态重置为 `TASK_RUNNING`，并返回 `schedule`——尽管不一定是立即返回。只要测试发生在进程将自己放入等待队列并改变其状态之后，事情就会起作用。

最后，我们调用 `finish_wait`。调用 `signal_pending` 告诉我们是否被信号唤醒；如果是，我们需要返回给用户，让他们稍后再试。否则，我们重新获取信号量，并像往常一样再次测试空闲空间。

6.2.5.3. 互斥等待

我们已经看到，当一个进程在等待队列上调用 `wake_up` 时，所有在该队列上等待的进程都会变为可运行状态。在许多情况下，这是正确的行为。然而，在其他情况下，我们可能提前知道只有一个被唤醒的进程会成功获取所需的资源，其余的进程只能再次睡眠。然而，每一个这样的进程都必须获取处理器，争夺资源（和任何控制锁），并显式地回到睡眠状态。如果等待队列中的进程数量很大，这种“雷鸣般的群体”行为可能会严重降低系统的性能。

为了应对现实世界中的雷鸣般的群体问题，内核开发人员向内核添加了一个“独占等待”选项。独占等待的行为非常像正常的睡眠，但有两个重要的区别：

- 当等待队列条目设置了 `WQ_FLAG_EXCLUSIVE` 标志时，它会被添加到等待队列的末尾。没有该标志的条目则被添加到开始位置。
- 当在等待队列上调用 `wake_up` 时，它在唤醒第一个设置了 `WQ_FLAG_EXCLUSIVE` 标志的进程后就停止了。

最终的结果是，执行独占等待的进程会以有序的方式一个接一个地被唤醒，不会产生雷鸣般的群体。然而，内核仍然每次都会唤醒所有非独占的等待者。

如果满足两个条件，那么在驱动程序中使用独占等待是值得考虑的：你预期对资源有显著的争用，唤醒一个进程就足以在资源变得可用时完全消耗资源。例如，独占等待对 Apache 网络服务器工作得很好；当新的连接进来时，系统上的（通常有很多）Apache 进程中应该只有一个唤醒来处理它。然而，我们在 scullpipe 驱动程序中没有使用独占等待；很少看到读取器争用数据（或写入器争用缓冲区空间），我们无法知道一旦唤醒，一个读取器会消耗所有可用的数据。

将进程放入可中断等待是一个简单的问题，只需调用 `prepare_to_wait_exclusive`：

```c
void prepare_to_wait_exclusive(wait_queue_head_t *queue,
                               wait_queue_t *wait,
                               int state);
```

这个调用，当用于替代 `prepare_to_wait` 时，会在等待队列条目中设置“独占”标志，并将进程添加到等待队列的末尾。注意，没有办法用 `wait_event` 及其变体执行独占等待。

6.2.5.4. 唤醒的细节

我们对唤醒过程的理解比内核内部真正发生的事情要简单。当进程被唤醒时，实际的行为是由等待队列条目中的一个函数控制的。默认的唤醒函数将进程设置为可运行状态，并可能对该进程进行上下文切换，如果它具有更高的优先级。设备驱动程序永远不需要

提供不同的唤醒函数；如果你的确是例外，请查看 <linux/wait.h> 以获取如何操作的信息。

我们还没有看到 wake_up 的所有变体。大多数驱动程序编写者永远不需要其他的，但是，为了完整性，这里是完整的集合：

```c
wake_up(wait_queue_head_t *queue);

wake_up_interruptible(wait_queue_head_t *queue);
```

wake_up 唤醒队列上的每一个进程，除非它在独占等待，并且如果存在，唤醒一个独占等待者。 wake_up_interruptible 做同样的事情，只是它跳过了处于不可中断睡眠的进程。这些函数在返回之前，可能会导致一个或多个被唤醒的进程被调度（尽管如果它们是从原子上下文中调用的，这不会发生）。

```c
wake_up_nr(wait_queue_head_t *queue, int nr);

wake_up_interruptible_nr(wait_queue_head_t *queue, int
nr);
```

这些函数的行为类似于 wake_up，除了它们可以唤醒多达 nr 个独占等待者，而不仅仅是一个。注意，传递 0 被解释为请求唤醒所有的独占等待者，而不是他们中的没有一个。

```c
wake_up_all(wait_queue_head_t *queue);

wake_up_interruptible_all(wait_queue_head_t *queue);
```

这种形式的 wake_up 唤醒所有的进程，无论它们是否正在执行独占等待（尽管可中断的形式仍然跳过执行不可中断等待的进程）。

```c
wake_up_interruptible_sync(wait_queue_head_t *queue);
```

通常，一个被唤醒的进程可能会抢占当前进程，并在 `wake_up` 返回之前被调度到处理器。换句话说，对 `wake_up` 的调用可能不是原子的。如果调用 `wake_up` 的进程在原子上下文中运行（例如，它持有一个自旋锁，或者是一个中断处理程序），这种重新调度就不会发生。通常，这种保护是足够的。然而，如果你需要明确要求在此时不被调度出处理器，你可以使用 `wake_up_interruptible` 的“同步”变体。当调用者即将重新调度时，最常使用此函数，因为先完成剩下的一点工作更有效率。

如果你第一次阅读以上内容时并不完全清楚，不用担心。除了 `wake_up_interruptible`，很少有驱动程序需要调用其他的。

如果你花时间深入研究内核源代码，你可能会遇到我们到目前为止还没有讨论过的两个函数：

```c
void sleep_on(wait_queue_head_t *queue);

void interruptible_sleep_on(wait_queue_head_t *queue);
```

正如你可能预期的，这些函数无条件地将当前进程在给定的队列上睡眠。然而，这些函数强烈不推荐使用，你永远不应该使用它们。如果你思考一下，问题就很明显：`sleep_on` 没有办法防止竞态条件。在你的代码决定必须睡眠和 `sleep_on` 实际执行睡眠之间，总是有一个窗口。在那个窗口期间到达的唤醒会被错过。因此，调用 `sleep_on` 的代码永远不是完全安全的。

当前的计划是在不久的将来从内核中移除 `sleep_on` 及其变体（我们还没有展示的有几种超时形式）。

6.2.6. 测试 scullpipe 驱动

我们已经看到了 `scullpipe` 驱动程序如何实现阻塞 I/O。如果你想尝试一下，这个驱动程序的源代码可以在书中的其他示例中找到。通过打开两个窗口，可以看到阻塞 I/O 的实际操作。第一个窗口可以运行如 `cat /dev/scullpipe` 这样的命令。然后，在

另一个窗口中，将一个文件复制到 `/dev/scullpipe`，你应该能看到该文件的内容出现在第一个窗口中。

测试非阻塞活动更为棘手，因为可用于 shell 的常规程序不执行非阻塞操作。`misc-progs` 源目录包含以下简单的程序，称为 `nbtest`，用于测试非阻塞操作。它所做的就是使用非阻塞 I/O 和延迟重试来将其输入复制到其输出。命令行传递延迟时间，默认为一秒。

```c
int main(int argc, char **argv)

{

    int delay = 1, n, m = 0;

    if (argc > 1)

        delay=atoi(argv[1]);

    fcntl(0, F_SETFL, fcntl(0,F_GETFL) | O_NONBLOCK); /*
stdin */

    fcntl(1, F_SETFL, fcntl(1,F_GETFL) | O_NONBLOCK); /*
stdout */

    while (1) {

        n = read(0, buffer, 4096);

        if (n ⩾ 0)

            m = write(1, buffer, n);

        if ((n < 0 || m < 0) && (errno ≠ EAGAIN))

            break;

        sleep(delay);

    }

    perror(n < 0 ? "stdin" : "stdout");

    exit(1);

}
```

如果你在一个进程跟踪工具（如 `strace`）下运行这个程序，你可以看到每个操作的成功或失败，这取决于当操作尝试时数据是否可用。

6.3. poll 和 select

使用非阻塞 I/O 的应用程序通常也会使用 `poll`， `select` 和 `epoll` 系统调用。 `poll`， `select` 和 `epoll` 的功能基本相同：每个都允许进程确定它是否可以从一个或多个打开的文件中读取或写入，而不会阻塞。这些调用也可以阻塞一个进程，直到给定的一组文件描述符中的任何一个变得可用于读取或写入。因此，它们通常用于必须使用多个输入或输出流的应用程序，而不会在其中任何一个上卡住。同样的功能由多个函数提供，因为两个函数几乎同时由两个不同的组在 Unix 中实现： `select` 在 BSD Unix 中引入，而 `poll` 是 System V 的解决方案。 `epoll` 调用在 2.5.45 中添加，作为使轮询函数扩展到数千个文件描述符的方法。

对任何这些调用的支持都需要设备驱动程序的支持。这种支持（对所有三个调用）是通过驱动程序的 `poll` 方法提供的。此方法具有以下原型：

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

每当用户空间程序执行涉及与驱动程序关联的文件描述符的 `poll`， `select` 或 `epoll` 系统调用时，都会调用驱动程序方法。设备方法负责以下两个步骤：

1. 在一个或多个可能指示 `poll` 状态变化的等待队列上调用 `poll_wait`。如果当前没有文件描述符可用于 I/O，内核会使进程在所有传递给系统调用的文件描述符的等待队列上等待。
2. 返回一个位掩码，描述可以立即执行的操作（如果有的话），而不会阻塞。

这两个操作通常都很简单，并且在一个驱动程序和下一个驱动程序之间看起来非常相似。然而，它们依赖于只有驱动程序才能提供的信息，因此，每个驱动程序必须单独实现。

`poll_table` 结构体，是 `poll` 方法的第二个参数，在内核中用于实现 `poll`， `select` 和 `epoll` 调用；它在 `<linux/poll.h>` 中声明，驱动程序源代码必须包含它。驱动程序编写者不需要了解其内部任何信息，并且必须将其作为一个不透明的对象

使用；它被传递给驱动程序方法，以便驱动程序可以加载可能唤醒进程并改变 `poll` 操作状态的每个等待队列。驱动程序通过调用 `poll_wait` 函数将等待队列添加到 `poll_table` 结构体中：

```c
void poll_wait (struct file *filp, wait_queue_head_t *wait_queue, poll_table *wait);
```

`poll` 方法执行的第二个任务是返回描述哪些操作可以立即完成的位掩码；这也很简单。例如，如果设备有可用的数据，读取将在不睡眠的情况下完成；`poll` 方法应该指示这种状态。有几个标志（通过 `<linux/poll.h>` 定义）用于指示可能的操作：

- `POLLIN`：如果设备可以在不阻塞的情况下读取，必须设置此位。
- `POLLRDNORM`：如果有“正常”数据可供读取，必须设置此位。可读设备返回 `(POLLIN | POLLRDNORM)`。
- `POLLRDBAND`：此位表示设备有带外数据可供读取。它目前只在 Linux 内核的一个地方（DECnet 代码）中使用，并且通常不适用于设备驱动程序。
- `POLLPRI`：高优先级数据（带外）可以在不阻塞的情况下读取。此位会导致 `select` 报告文件上发生了异常条件，因为 `select` 将带外数据报告为异常条件。
- `POLLHUP`：当读取此设备的进程看到文件结束时，驱动程序必须设置 `POLLHUP`（挂起）。调用 `select` 的进程被告知设备是可读的，如 `select` 功能所规定。
- `POLLERR`：设备上发生了错误条件。当调用 `poll` 时，设备被报告为可读和可写，因为 `read` 和 `write` 都在不阻塞的情况下返回错误代码。
- `POLLOUT`：如果设备可以在不阻塞的情况下写入，此位在返回值中设置。
- `POLLWRNORM`：此位与 `POLLOUT` 具有相同的含义，有时它实际上是相同的数字。可写设备返回 `(POLLOUT | POLLWRNORM)`。
- `POLLWRBAND`：像 `POLLRDBAND` 一样，此位表示可以向设备写入非零优先级的数据。只有 `poll` 的数据报实现使用此位，因为数据报可以传输带外数据。

值得重复的是，`POLLRDBAND` 和 `POLLWRBAND` 只对与套接字关联的文件描述符有意义：设备驱动程序通常不会使用这些标志。

`poll` 的描述占用了很多空间，但在实践中使用相对简单。考虑 `scullpipe` 实现的 `poll` 方法：

```c
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;

}
```

此代码只是将两个 scullpipe 等待队列添加到 poll_table，然后根据是否可以读取或写入数据设置适当的掩码位。

如所示的 poll 代码缺少文件结束支持，因为 scullpipe 不支持文件结束条件。对于大多数真实设备，如果没有更多的数据（或将变得）可用，poll 方法应返回 POLLHUP。如果调用者使用了 select 系统调用，文件被报告为可读。无论使用 poll 还是 select，应用程序都知道它可以在不永远等待的情况下调用 read，并且 read 方法返回 0 来表示文件结束。

例如，对于真实的 FIFO，当所有的写入者关闭文件时，读取者会看到文件结束，而在 scullpipe 中，读取者永远不会看到文件结束。行为不同是因为 FIFO 旨在成为两个进程之间的通信通道，而 scullpipe 是一个垃圾桶，只要至少有一个读取者，每个人都可以放入数据。此外，重新实现内核中已经存在的内容没有意义，所以我们选择在我们的示例中实现不同的行为。

像 FIFO 那样实现文件结束的方式将意味着检查 dev→nwriters，无论在 read 还是在 poll 中，如果没有进程为写入打开设备，都会报告文件结束（如刚才描述的）。不幸的是，使用这种实现，如果读取者在写入者之前打开了 scullpipe 设备，它将看到文件结束，而没有机会等待数据。解决这个问题的最佳方法将是像真实的 FIFO 那样在 open 中实现阻塞；这个任务留给读者作为练习。

6.3.1. 与 read 和 write 的交互

poll 和 select 调用的目的是预先确定 I/O 操作是否会阻塞。在这方面，它们补充了 read 和 write。更重要的是，poll 和 select 是有用的，因为它们让应用程序同时等待几个数据流，尽管我们在 scull 示例中没有利用这个特性。

- 正确实现这三个调用对于使应用程序正确工作至关重要：尽管以下规则已经或多或少地被陈述过，我们在这里总结它们：

1. read（或 write）在没有数据（或空间）可用时必须阻塞，除非文件被打开为非阻塞。
2. 如果 read（或 write）在没有数据（或空间）可用时阻塞，那么 poll（或 select）必须报告设备不可读（或不可写）。
3. 如果 poll（或 select）报告设备可读（或可写），那么随后的 read（或 write）必须在不阻塞的情况下返回，至少返回一个字节（或接受一个字节）。

4. 如果设备的状态改变（即，它从不可读变为可读，或者反过来），所有在设备等待队列上睡眠的进程必须被唤醒。这包括正在等待 `read` 或 `write` 的进程，以及正在等待 `poll` 或 `select` 的进程。

- 以上四点基于ai生成,待验证

- `poll`, `select`, `read` 和 `write` 都是操作系统提供的系统调用，用于处理 I/O 操作，但它们的用途和行为有所不同：

1. `read`：此系统调用用于从文件（通常是设备）中读取数据。如果文件中没有可用的数据，`read` 调用可能会阻塞，直到有数据可读。
2. `write`：此系统调用用于将数据写入文件（通常是设备）。如果文件没有足够的空间来接收数据，`write` 调用可能会阻塞，直到有足够的空间。
3. `poll` 和 `select`：这两个系统调用用于查询一个或多个文件（通常是设备）的状态，以确定是否可以进行非阻塞的 `read` 或 `write` 操作。换句话说，它们可以告诉你是否可以立即执行 `read` 或 `write` 调用，而不会阻塞。如果你查询的多个文件中的任何一个都不能进行非阻塞操作，`poll` 和 `select` 可以阻塞，直到至少有一个文件准备好进行 I/O 操作。

- 总的来说，`read` 和 `write` 是用于实际读取和写入数据的系统调用，而 `poll` 和 `select` 是用于在尝试读取或写入之前检查是否会阻塞的系统调用。

6.3.1.1. 从设备中读数据

- 如果输入缓冲区中有数据，那么读取调用应立即返回，即使可用的数据少于应用程序请求的数据，也不会有明显的延迟，即使驱动程序确定剩余的数据很快就会到达。如果出于任何原因这样做方便，你总是可以返回比你被要求的数据少的数据（我们在scull中就是这样做的），只要你返回至少一个字节。在这种情况下，poll应返回`POLLIN|POLLRDNORM`。
- 如果输入缓冲区中没有数据，默认情况下，读取必须阻塞，直到至少有一个字节在那里。另一方面，如果设置了O_NONBLOCK，读取会立即返回，返回值为`-EAGAIN`（尽管System V的一些旧版本在这种情况下返回0）。在这些情况下，poll必须报告设备不可读，直到至少有一个字节到达。一旦缓冲区中有一些数据，我们就回到了前一种情况。
- 如果我们处于文件结束状态，读取应立即返回，返回值为0，与`O_NONBLOCK`无关。在这种情况下，poll应报告POLLHUP。

6.3.1.2. 写入设备

- 如果输出缓冲区有空间，write应该立即返回，不会有延迟。它可能接受的数据少于调用请求的数据，但至少必须接受一个字节。在这种情况下，poll通过返回`POLLOUT|POLLWRNORM`报告设备是可写的。

- 如果输出缓冲区已满，默认情况下write会阻塞，直到有一些空间被释放。如果设置了`O_NONBLOCK`，write会立即返回，返回值为`-EAGAIN`（旧的System V Unix返回0）。在这些情况下，poll应该报告文件不可写。另一方面，如果设备无法接受更多的数据，write返回`-ENOSPC`（“设备上没有剩余空间”），这与`O_NONBLOCK`的设置无关。
- 即使清除了`O_NONBLOCK`，也不要让write调用等待数据传输后再返回。这是因为许多应用程序使用select来找出write是否会阻塞。如果设备被报告为可写，调用就不应该阻塞。如果使用设备的程序想确保它在输出缓冲区中排队的数据实际上被传输，驱动程序必须提供一个fsync方法。例如，可移动设备应该有一个fsync入口点。

虽然这是一套很好的通用规则，但也应该认识到每个设备都是独一无二的，有时候规则必须稍微弯曲一下。例如，记录导向的设备（如磁带驱动器）不能执行部分写入。

6.3.1.3. 刷新挂起的输出Flushing pending output

我们已经看到，单独的write方法并不能满足所有的数据输出需求。fsync函数，通过同名的系统调用来调用，填补了这个空白。这个方法的原型是

```c
int (*fsync) (struct file *file, struct dentry *dentry,
int datasync);
```

如果某个应用程序需要确保数据已经被发送到设备，那么无论是否设置了`O_NONBLOCK`，都必须实现fsync方法。fsync的调用只有在设备完全刷新（即，输出缓冲区为空）后才返回，即使这需要一些时间。datasync参数用于区分fsync和fdatasync系统调用；因此，它只对文件系统代码有兴趣，可以被驱动程序忽略。

fsync方法没有不寻常的特性。调用并不是时间关键的，所以每个设备驱动程序都可以按照作者的喜好来实现它。大多数时候，字符驱动程序在他们的fops中只有一个NULL指针。另一方面，块设备总是用通用的block_fsync来实现方法，这个方法反过来会刷新设备的所有块，等待I/O完成.

6.3.2. 底层的数据结构

poll、select和epoll_ctl系统调用的实际实现相对简单，对于那些对其工作原理感兴趣的人来说；epoll稍微复杂一些，但是基于同样的机制。无论用户应用程序何时调用poll、select或epoll_ctl，内核都会调用系统调用引用的所有文件的poll方法，向每个文

件传递相同的poll_table。poll_table结构只是一个围绕构建实际数据结构的函数的包装器。对于poll和select，该结构是一个包含poll_table_entry结构的内存页的链表。每个poll_table_entry都保存了传递给poll_wait的struct file和wait_queue_head_t指针，以及一个关联的等待队列条目。poll_wait的调用有时也会将进程添加到给定的等待队列中。在poll或select返回之前，内核必须维护整个结构，以便可以从所有这些队列中移除进程。

如果被轮询的驱动程序都没有指示可以进行无阻塞的I/O，那么poll调用只会睡眠，直到它所在的（可能有很多）等待队列之一唤醒它。

在poll的实现中，有趣的是，驱动程序的poll方法可能会被调用，并且poll_table参数为NULL指针。这种情况可能由几个原因引起。如果调用poll的应用程序提供了一个为0的超时值（表示不应该等待），那么就没有理由积累等待队列，系统简单地不会这么做。只要被轮询的任何驱动程序指示I/O是可能的，poll_table指针也会立即设置为NULL。因为内核在那个时候知道不会有等待发生，所以它不会建立等待队列的列表。

当poll调用完成时，poll_table结构被释放，所有以前添加到poll表的等待队列条目（如果有的话）都从表中移除，并从他们的等待队列中移除。

我们试图在图6-1中展示涉及轮询的数据结构；这个图是真实数据结构的简化表示，因为它忽略了poll表的多页性质，并忽视了每个poll_table_entry的文件指针部分。对实际实现感兴趣的读者建议查看<linux/poll.h>和fs/select.c。

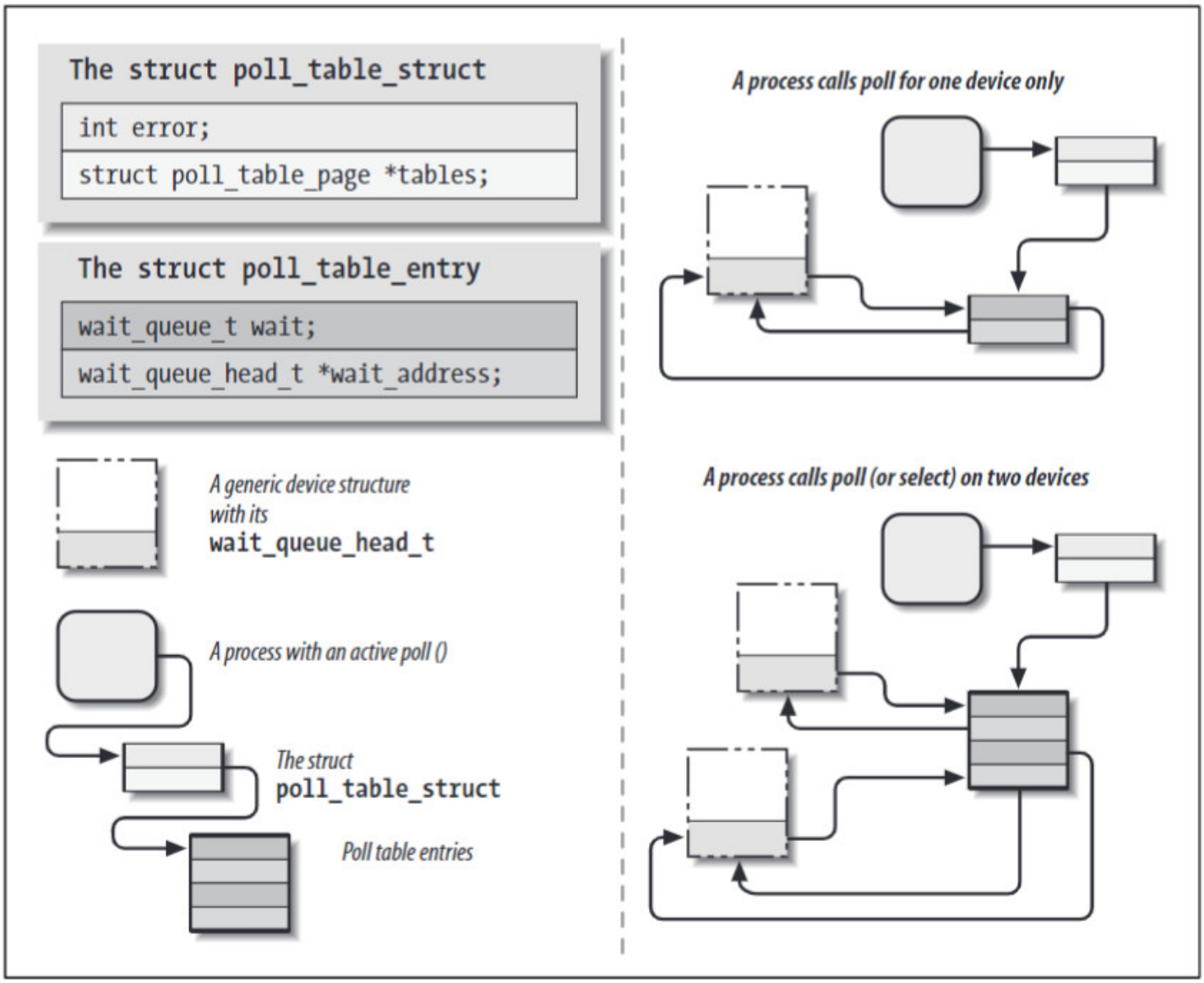

Figure 6-1. The data structures behind poll

在这个时候，我们可以理解新的epoll系统调用背后的动机。在典型的情况下，调用poll或select只涉及到少数的文件描述符，所以设置数据结构的成本很小。然而，有一些应用程序需要处理数千个文件描述符。在这种情况下，每次I/O操作之间建立和拆除这个数据结构变得过于昂贵。epoll系统调用族允许这种应用程序只设置一次内核的内部数据结构，并多次使用它。

6.4. 异步通知

虽然阻塞和非阻塞操作的组合以及select方法大多数时候都足够查询设备，但是有些情况我们迄今为止看到的技术并不能有效地管理。

让我们想象一个进程，它在低优先级下执行一个长时间的计算循环，但需要尽快处理进来的数据。如果这个进程正在响应来自某种数据采集外设的新观察结果，它希望立即知道新数据何时可用。这个应用程序可以被编写成定期调用poll来检查数据，但是，对于许多情况，有更好的方法。通过启用异步通知，这个应用程序可以在数据变得可用时接收一个信号，而不需要关心轮询。

用户程序必须执行两个步骤来启用来自输入文件的异步通知。首先，他们指定一个进程作为文件的“所有者”。当一个进程使用fcntl系统调用执行F_SETOWN命令时，所有者进程的进程ID被保存在filp->f_owner中以供后续使用。这个步骤是必要的，因为内核需要知道通知谁。为了实际启用异步通知，用户程序必须通过F_SETFL fcntl命令在设备中设置FASYNC标志。

在这两个调用执行之后，输入文件可以在新数据到达时请求发送一个SIGIO信号。信号被发送到存储在filp->f_owner中的进程（或进程组，如果值为负）。

例如，用户程序中的以下代码行为stdin输入文件启用异步通知到当前进程：

```c
signal(SIGIO, &input_handler); /* dummy sample;
sigaction() is better */

fcntl(STDIN_FILENO, F_SETOWN, getpid( ));

oflags = fcntl(STDIN_FILENO, F_GETFL);

fcntl(STDIN_FILENO, F_SETFL, oflags | FASYNC);
```

源代码中名为asynctest的程序是一个简单的程序，它像上面那样读取stdin。它可以用来测试scullpipe的异步能力。这个程序类似于cat，但是它不会在文件结束时终止；它只对输入做出响应，而不是对输入的缺失做出响应。

然而，请注意，并非所有的设备都支持异步通知，你可以选择不提供它。应用程序通常假设异步能力只对套接字和ttys可用。

输入通知还有一个剩下的问题。当一个进程接收到一个SIGIO时，它不知道哪个输入文件有新的输入提供。如果有多个文件被启用以异步通知进程有待处理的输入，应用程序仍然必须使用poll或select来找出发生了什么。

6.4.1. 驱动的观点

对我们来说，更相关的话题是设备驱动程序如何实现异步信号。以下列表详细描述了从内核的角度看的操作序列：

1. 当调用F_SETOWN时，除了将一个值赋给filp->f_owner外，什么也不会发生。

2. 当执行F_SETFL以打开FASYNC时，会调用驱动程序的fasync方法。每当filp->f_flags中的FASYNC值改变时，都会调用此方法，以通知驱动程序变化，以便它可以适当地响应。当文件打开时，默认清除该标志。我们将在本节后面查看驱动方法的标准实现。
3. 当数据到达时，必须向所有注册异步通知的进程发送一个SIGIO信号。

虽然实现第一步是微不足道的——驱动程序部分没有什么要做的——但其他步骤涉及到维护一个动态数据结构来跟踪不同的异步读取器；可能有几个。然而，这个动态数据结构并不依赖于特定的设备，内核提供了一个适当的通用实现，这样你就不必在每个驱动程序中重写相同的代码。

Linux提供的通用实现基于一个数据结构和两个函数（这两个函数在前面描述的第二步和第三步中被调用）。声明相关材料的头文件是<linux/fs.h>（这里没有新的东西），数据结构被称为struct fasync_struct。与等待队列一样，我们需要在设备特定的数据结构中插入一个指向该结构的指针。

驱动程序调用的两个函数对应以下原型：

```c
int fasync_helper(int fd, struct file *filp,
        int mode, struct fasync_struct **fa);
void kill_fasync(struct fasync_struct **fa, int sig, int
band);
```

当打开文件的FASYNC标志改变时，fasync_helper被调用以从感兴趣的进程列表中添加或删除条目。除最后一个参数外，所有参数都提供给fasync方法，并可以直接传递。当数据到达时，kill_fasync用于向感兴趣的进程发送信号。它的参数是要发送的信号（通常是SIGIO）和带宽，几乎总是`POLL_IN*`（但在网络代码中可能用于发送“紧急”或带外数据）。

以下是scullpipe如何实现fasync方法的示例：

```c
static int scull_p_fasync(int fd, struct file *filp, int
mode)

{

    struct scull_pipe *dev = filp->private_data;

    return fasync_helper(fd, filp, mode, &dev-
>async_queue);

}
```

很明显，所有的工作都是由fasync_helper完成的。然而，如果没有驱动程序中的方法，就无法实现这个功能，因为辅助函数需要访问正确的`struct fasync_struct *`的指针（这里是`&dev->async_queue`），只有驱动程序才能提供这个信息。

因此，当数据到达时，必须执行以下语句来通知异步读取器。由于scullpipe读取器的新数据是由发出写操作的进程生成的，所以该语句出现在scullpipe的写方法中。

```c
if (dev->async_queue)

    kill_fasync(&dev->async_queue, SIGIO, POLL_IN);
```

注意，一些设备还实现了异步通知，以指示何时可以写入设备；在这种情况下，当然，必须以POLL_OUT模式调用kill_fasync。

我们可能会认为我们已经完成了，但还有一件事情缺失。我们必须在文件关闭时调用我们的fasync方法，以将文件从活动的异步读取器列表中移除。虽然只有在filp->f_flags设置了FASYNC时才需要这个调用，但无论如何调用这个函数都没有关系，这是通常的实现。例如，以下行是scullpipe的release方法的一部分：

```c
/* remove this filp from the asynchronously notified
filp's */

scull_p_fasync(-1, filp, 0);
```

异步通知的底层数据结构几乎与结构struct wait_queue相同，因为两种情况都涉及等待一个事件。不同之处在于，struct file代替了struct task_struct。然后在队列中使用struct file来检索f_owner，以便向进程发送信号。

6.5. 移位一个设备Seeking

6.5.1. llseek 实现

llseek方法实现了lseek和llseek系统调用。我们已经提到，如果设备的操作中缺少llseek方法，那么内核中的默认实现会通过修改filp→f_pos（文件内当前的读/写位置）来执行查找。请注意，为了使lseek系统调用正确工作，read和write方法必须通过使用和更新它们接收到的偏移量参数来配合。

如果查找操作对应于设备上的物理操作，你可能需要提供你自己的llseek方法。一个简单的例子可以在scull驱动程序中看到：

```c
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;

}
```

这里唯一的设备特定操作是从设备中检索文件长度。在scull中，read和write方法按需要配合，如第3章所示。

虽然刚刚展示的实现对于处理明确定义的数据区域的scull来说是有意义的，但大多数设备提供的是数据流，而不是数据区域（只需想想串行端口或键盘），对这些设备进行查找是没有意义的。如果你的设备也是这种情况，你不能仅仅是不声明llseek操作，因为默认方法允许查找。相反，你应该通过在你的open方法中调用nonseekable_open来通知内核你的设备不支持llseek：

```c
int nonseekable_open(struct inode *inode; struct file
*filp);
```

这个调用将给定的filp标记为不可查找的；内核永远不允许对这样的文件进行lseek调用。通过这种方式标记文件，你也可以确保不会通过pread和pwrite系统调用来尝试查找文件。

为了完整性，你还应该在你的file_operations结构中将llseek方法设置为特殊的辅助函数no_llseek，这个函数在<linux/fs.h>中定义。

- `llseek`方法是Linux内核中的一个系统调用，它用于改变文件的当前读/写位置。这个位置通常被称为"文件偏移量"或"文件指针"。`llseek`方法允许你在文件中任意移动这个位置，这对于许多应用程序来说是非常有用的，比如需要读写大文件或需要随机访问文件内容的应用程序。

- 在设备驱动程序中，`llseek`方法通常用于实现设备的查找操作。如果设备支持查找操作，那么你需要在设备的操作中提供你自己的`llseek`方法。如果设备不支持查找操作，你可以通过调用`nonseekable_open`方法来通知内核。

- 这是`llseek`方法的一般形式：

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

其中，第一个参数是一个指向`struct file`的指针，表示要操作的文件；第二个参数是一个`loff_t`类型的值，表示要移动到的新位置；第三个参数是一个整数，表示移动的基准位置，它可以是`SEEK_SET`（从文件开始位置计算偏移量）、`SEEK_CUR`（从当前位置计算偏移量）或`SEEK_END`（从文件结束位置计算偏移量）。

6.6. 在一个设备文件上的存取控制

设备节点的访问控制有时对于其可靠性至关重要。不仅未经授权的用户不应被允许使用设备（这是通过文件系统权限位来强制执行的限制），而且有时只应允许一个授权用户同时打开设备。

这个问题类似于使用终端（tty）。在那种情况下，登录过程会在用户登录系统时更改设备节点的所有权，以防止其他用户干扰或窥探终端的数据流。然而，使用特权程序在每次打开设备时更改设备的所有权，只是为了赋予它唯一的访问权限，这是不切实际的。

到目前为止，我们展示的代码没有实现任何超出文件系统权限位的访问控制。如果open系统调用将请求转发给驱动程序，open就会成功。我们现在介绍一些实现额外检查的技术。

本节中显示的每个设备都具有与基础scull设备相同的行为（也就是说，它实现了一个持久的内存区域），但在访问控制上与scull不同，这是在open和release操作中实现的。

6.6.1. 单 open 设备

提供访问控制的暴力方式是一次只允许一个进程打开设备（单一开放）。这种技术最好避免，因为它会抑制用户的创新。用户可能想在同一设备上运行不同的进程，一个读取状态信息，另一个写入数据。在某些情况下，只要他们可以同时访问设备，用户可以通过运行一些简单的程序通过shell脚本完成很多工作。换句话说，实现单一开放行为相当于创建策略，这可能会妨碍你的用户想要做的事情。

只允许一个进程打开设备有不良的属性，但这也是设备驱动程序实现访问控制最容易的方式，所以在这里展示。源代码是从一个叫做scullsingle的设备中提取的。

scullsingle设备维护了一个名为`scull_s_available`的`atomic_t`变量；该变量被初始化为一个值，表示设备确实可用。open调用递减并测试`scull_s_available`，

并在其他人已经打开设备时拒绝访问：

```c
static atomic_t scull_s_available = ATOMIC_INIT(1);

static int scull_s_open(struct inode *inode, struct file
*filp)

{

    struct scull_dev *dev = &scull_s_device; /* device
information */

    if (! atomic_dec_and_test (&scull_s_available)) {

        atomic_inc(&scull_s_available);

        return -EBUSY; /* already open */

    }

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

    if ( (filp→f_flags & O_ACCMODE) = = O_WRONLY)

        scull_trim(dev);

    filp→private_data = dev;

    return 0;          /* success */

}
```

另一方面，release调用将设备标记为不再忙碌：

```c
static int scull_s_release(struct inode *inode, struct
file *filp)

{

    atomic_inc(&scull_s_available); /* release the device
*/

    return 0;

}
```

通常，我们建议你将open标志scull_s_available放在设备结构（这里是Scull_Dev）内，因为从概念上讲，它属于设备。然而，scull驱动程序使用独立的变量来保存标志，这样它可以使用与裸scull设备相同的设备结构和方法，最小化代码重复。

6.6.2. 一次对一个用户限制存取

比单一开放设备更进一步的步骤是让单一用户在多个进程中打开设备，但一次只允许一个用户打开设备。这种解决方案使得测试设备变得容易，因为用户可以同时从多个进程读写，但假设用户对在多次访问过程中维护数据的完整性负有一定责任。这是通过在open方法中添加检查来实现的；这样的检查在正常权限检查之后进行，只能使访问比由所有者和组权限位指定的更加限制。这与用于tty的访问策略相同，但不需要借助外部特权程序。

这些访问策略比单一开放策略更难实现。在这种情况下，需要两项：一个开放计数和设备的“所有者”的uid。再次，这样的项目最好放在设备结构内；我们的例子使用全局变量，原因如前面对scullsingle的解释。设备的名称是sculluid。

open调用在第一次打开时授予访问权限，但记住了设备的所有者。这意味着用户可以多次打开设备，从而允许合作的进程同时在设备上工作。同时，没有其他用户可以打开它，从而避免外部干扰。由于这个函数版本与前一个几乎相同，所以这里只复制了相关部分：

```c
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);
```

注意，sculluid代码有两个变量（scull_u_owner和scull_u_count）控制对设备的访问，这些变量可能被多个进程同时访问。为了使这些变量安全，我们用一个自旋锁（scull_u_lock）控制对它们的访问。没有这个锁，两个（或更多）进程可能同时测试scull_u_count，并且都可能得出他们有权拥有设备的结论。这里需要一个自旋锁，因为锁持有的时间非常短，而且驱动程序在持有锁的时候不做任何可能睡眠的事情。

我们选择返回 -EBUSY 而不是 -EPERM，尽管代码正在执行权限检查，但这是为了指引被拒绝访问的用户朝正确的方向。对“权限被拒绝”的反应通常是检查 /dev 文件的模式和所有者，而“设备繁忙”正确地建议用户应寻找已经使用设备的进程。 此代码还检查

尝试打开的进程是否有能力覆盖文件访问权限；如果是，即使打开进程不是设备的所有者，也允许打开。在这种情况下，CAP_DAC_OVERRIDE 能力非常适合这项任务。release 方法如下所示：

```c
static int scull_u_release(struct inode *inode, struct
file *filp)

{

    spin_lock(&scull_u_lock);

    scull_u_count--; /* nothing else */

    spin_unlock(&scull_u_lock);

    return 0;

}
```

再次，我们必须在修改计数之前获取锁，以确保我们不会与另一个进程竞争。

6.6.3. 阻塞 open 作为对 EBUSY 的替代

当设备无法访问时，通常返回错误是最明智的做法，但在某些情况下，用户可能更愿意等待设备。

例如，如果一个数据通信通道既用于定期、定时地传输报告（使用 crontab），又用于根据人们的需要进行随意使用，那么比起因为通道当前繁忙而失败，更好的做法是稍微延迟定期操作。

这是程序员在设计设备驱动时必须做出的选择之一，正确的答案取决于要解决的特定问题。

你可能已经猜到，EBUSY 的替代方案是实现阻塞打开。scullwuid 设备是 sculluid 的一个版本，它在打开时等待设备，而不是返回 -EBUSY。它只在打开操作的以下部分与 sculluid 不同：

```c
spin_lock(&scull_w_lock);

while (! scull_w_available( )) {

    spin_unlock(&scull_w_lock);

    if (filp->f_flags & O_NONBLOCK) return -EAGAIN;

    if (wait_event_interruptible (scull_w_wait,
scull_w_available( )))

        return -ERESTARTSYS; /* 告诉文件系统层处理它 */

    spin_lock(&scull_w_lock);

}

if (scull_w_count == 0)

    scull_w_owner = current->uid; /* 抓住它 */

scull_w_count++;

spin_unlock(&scull_w_lock);
```

这个实现再次基于等待队列。如果设备当前不可用，尝试打开它的进程将被放在等待队列上，直到拥有进程关闭设备。

然后，release 方法负责唤醒任何挂起的进程：

```c
static int scull_w_release(struct inode *inode, struct
file *filp)

{

    int temp;

    spin_lock(&scull_w_lock);

    scull_w_count--;

    temp = scull_w_count;

    spin_unlock(&scull_w_lock);

    if (temp == 0)

        wake_up_interruptible_sync(&scull_w_wait); /* 唤
醒其他 uid */

    return 0;

}
```

这是一个调用 wake_up_interruptible_sync 有意义的例子。当我们做唤醒时，我们即将返回到用户空间，这是系统的一个自然调度点。与其在我们做唤醒时可能重新调度，不如直接调用“同步”版本并完成我们的工作。

阻塞打开实现的问题在于，对于交互式用户来说，这真的很不愉快，他们必须不断猜测出了什么问题。交互式用户通常会调用标准命令，如 cp 和 tar，并不能只是在打开调用中添加 O_NONBLOCK。在隔壤房间使用磁带驱动器备份的人，宁愿得到一个简单的“设备或资源繁忙”的消息，而不是被留下猜测为什么今天的硬盘如此安静，而 tar 应该在扫描它。

这种问题（对同一设备需要不同、不兼容的策略）通常最好通过为每个访问策略实现一个设备节点来解决。这种做法的一个例子可以在 Linux 磁带驱动器中找到，它为同一设

备提供了多个设备文件。例如，不同的设备文件将导致驱动器在设备关闭时是否压缩或自动倒带。

6.6.4. 在 open 时复制设备

管理访问控制的另一种技术是根据打开设备的进程创建设备的不同私有副本。

显然，只有当设备不绑定到硬件对象时，这才可能；scull 就是这样一个“软件”设备的例子。/dev/tty 的内部使用了类似的技术，以便给其进程一个不同的“视图”，表示 /dev 入口点代表的是什么。当设备的副本由软件驱动器创建时，我们称之为虚拟设备——就像虚拟控制台使用单个物理 tty 设备一样。

尽管这种访问控制很少需要，但实现可以启示我们，内核代码如何轻易地改变应用程序对周围世界（即计算机）的视角。

/dev/scullpriv 设备节点在 scull 包内实现了虚拟设备。scullpriv 实现使用进程的控制 tty 的设备号作为访问虚拟设备的键。然而，你可以轻易地修改源代码，使用任何整数值作为键；每个选择都会导致不同的策略。例如，使用 uid 会为每个用户创建一个不同的虚拟设备，而使用 pid 键会为每个访问它的进程创建一个新设备。

使用控制终端的决定是为了方便使用 I/O 重定向测试设备：设备由在同一虚拟终端上运行的所有命令共享，并与在另一个终端上运行的命令看到的设备保持分离。

这段代码中的 open 方法主要是在查找正确的虚拟设备，如果没有找到，可能会创建一个。函数的最后部分没有显示，因为它是从我们已经看过的裸 scull 复制过来的。这是一个常见的模式，用于处理设备的打开操作。在这个函数中，首先检查当前进程是否有控制终端，如果没有，函数返回错误。然后，它使用 tty_devnum 函数获取控制终端的设备号，这将作为查找或创建 scull 设备的关键字。最后，它调用 scull_c_lookfor_device 函数来查找或创建设备。如果设备不存在，该函数将创建一个新的设备并添加到设备列表中。

```c
/* 克隆特定的数据结构包括一个关键字段 */

struct scull_listitem {

    struct scull_dev device;

    dev_t key;

    struct list_head list;

};

/* 设备列表，以及保护它的锁 */

static LIST_HEAD(scull_c_list);

static spinlock_t scull_c_lock = SPIN_LOCK_UNLOCKED;

/* 寻找设备或者如果没有则创建一个 */

static struct scull_dev *scull_c_lookfor_device(dev_t
key) {

    struct scull_listitem *lptr;

    list_for_each_entry(lptr, &scull_c_list, list) {

        if (lptr->key == key)

            return &(lptr->device);

    }

    /* 没找到 */

    lptr = kmalloc(sizeof(struct scull_listitem),
GFP_KERNEL);
```

```
    if (!lptr)
        return NULL;
    /* 初始化设备 */
    memset(lptr, 0, sizeof(struct scull_listitem));
    lptr->key = key;
    scull_trim(&(lptr->device));
    /* 初始化它 */
    init_MUTEX(&(lptr->device.sem));
    /* 将它放入列表 */
    list_add(&lptr->list, &scull_c_list);
    return &(lptr->device);
}

static int scull_c_open(struct inode *inode, struct file
*filp) {
    struct scull_dev *dev;
    dev_t key;
    if (!current->signal->tty) {
        PDEBUG("进程 \"%s\" 没有控制终端\n", current-
>comm);
        return -EINVAL;
    }
```

```
        key = tty_devnum(current->signal->tty);
        /* 在列表中寻找一个 scullc 设备 */
        spin_lock(&scull_c_lock);
        dev = scull_c_lookfor_device(key);
        spin_unlock(&scull_c_lock);
        if (!dev)
            return -ENOMEM;
        /* 然后，其他所有的都是从裸设备 scull 复制的 */
    }
```

释放方法并没有做什么特别的事情。通常，它会在最后关闭时释放设备，但我们选择不维护一个打开计数，以简化驱动程序的测试。如果设备在最后关闭时被释放，除非有一个后台进程保持它打开，否则你将无法在写入设备后读取相同的数据。示例驱动程序采取了更简单的方法，保留数据，这样在下次打开时，你会发现它在那里。当调用 scull_cleanup 时，设备会被释放。这段代码优先使用通用的 Linux 链表机制，而不是从头开始重新实现相同的功能。Linux 列表在第 11 章中有讨论。以下是 /dev/scullpriv 的释放实现，这结束了设备方法的讨论。

```c
static int scull_c_release(struct inode *inode, struct
file *filp) {

    /* 没有什么要做的，因为设备是持久的。

    * 一个 `真正的' 克隆设备应该在最后关闭时被释放 */

    return 0;

}
```

6.7. 快速参考

本章介绍了下面的符号和头文件:

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

声明用来定义 ioctl 命令的宏定义. 当前被 <linux/fs.h> 包含.

```
_IOC_NRBITS
_IOC_TYPEBITS
_IOC_SIZEBITS
_IOC_DIRBITS
```

ioctl 命令的不同位段所使用的位数. 还有 4 个宏来指定 MASK 和 4 个指定 SHIFT, 但是它们主要是给内部使用. `_IOC_SIZEBIT` 是一个要检查的重要的值, 因为它跨体系改变.

```
_IOC_NONE
_IOC_READ
_IOC_WRITE
```

"方向"位段可能的值. "read" 和 "write" 是不同的位并且可相或来指定 read/write. 这些值是基于 0 的.

```
_IOC(dir,type,nr,size)
_IO(type,nr)
_IOR(type,nr,size)
_IOW(type,nr,size)
_IOWR(type,nr,size)
```

用来创建 ioclt 命令的宏定义.

```
_IOC_DIR(nr)
_IOC_TYPE(nr)
_IOC_NR(nr)
_IOC_SIZE(nr)
```

用来解码一个命令的宏定义. 特别地, `_IOC_TYPE(nr) 是 _IOC_READ 和 _IOC_WRITE 的 OR 结合.`

```
#include <asm/uaccess.h>
int access_ok(int type, const void *addr, unsigned long
size);
```

检查一个用户空间的指针是可用的. access_ok 返回一个非零值, 如果应当允许存取.

```
VERIFY_READ
VERIFY_WRITE
```

access_ok 中 type 参数的可能取值. VERIFY_WRITE 是 VERIFY_READ 的超集.

```
#include <asm/uaccess.h>
int put_user(datum,ptr);
int get_user(local,ptr);
int __put_user(datum,ptr);
int __get_user(local,ptr);
```

用来存储或获取一个数据到或从用户空间的宏. 传送的字节数依赖 sizeof(*ptr). 常规的版本调用 access_ok , 而常规版本(**put_user 和** get_user) 假定 access_ok 已经被调用了.

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

定义各种 CAP_ 符号, 描述一个用户空间进程可有的能力.

```
int capable(int capability);
```

返回非零值如果进程有给定的能力.

```
#include <linux/wait.h>
typedef struct { /* ... */ } wait_queue_head_t;
void init_waitqueue_head(wait_queue_head_t *queue);
DECLARE_WAIT_QUEUE_HEAD(queue);
```

Linux 等待队列的定义类型. 一个 wait_queue_head_t 必须被明确在运行时使用 init_waitqueue_head 或者编译时使用 DEVLARE_WAIT_QUEUE_HEAD 进行初始化.

```
void wait_event(wait_queue_head_t q, int condition);
int wait_event_interruptible(wait_queue_head_t q, int
condition);
int wait_event_timeout(wait_queue_head_t q, int condition,
int time);
int wait_event_interruptible_timeout(wait_queue_head_t q,
int condition,int time);
```

使进程在给定队列上睡眠, 直到给定条件值为真值.

```
void wake_up(struct wait_queue **q);
void wake_up_interruptible(struct wait_queue **q);
void wake_up_nr(struct wait_queue **q, int nr);
void wake_up_interruptible_nr(struct wait_queue **q, int
nr);
void wake_up_all(struct wait_queue **q);
void wake_up_interruptible_all(struct wait_queue **q);
void wake_up_interruptible_sync(struct wait_queue **q);
```

唤醒在队列 q 上睡眠的进程. `_interruptible` 的形式只唤醒可中断的进程. 正常地, 只有一个互斥等待者被唤醒, 但是这个行为可被 `_nr` 或者 `_all` 形式所改变. `_sync` 版本在返回之前不重新调度 CPU.

```
#include <linux/sched.h>
set_current_state(int state);
```

设置当前进程的执行状态. TASK_RUNNING 意味着它已经运行, 而睡眠状态是 TASK_INTERRUPTIBLE 和 TASK_UNINTERRUPTIBLE.

```
void schedule(void);
```

选择一个可运行的进程从运行队列中. 被选中的进程可是当前进程或者另外一个.

```
typedef struct { /* ... */ } wait_queue_t;
init_waitqueue_entry(wait_queue_t *entry, struct
task_struct *task);
```

wait_queue_t 类型用来放置一个进程到一个等待队列.

```
void prepare_to_wait(wait_queue_head_t *queue,
wait_queue_t *wait, int state);
void prepare_to_wait_exclusive(wait_queue_head_t *queue,
wait_queue_t *wait, int state);
void finish_wait(wait_queue_head_t *queue, wait_queue_t
*wait);
```

帮忙函数, 可用来编码一个手工睡眠.

```
void sleep_on(wiat_queue_head_t *queue);
void interruptible_sleep_on(wiat_queue_head_t *queue);
```

老式的不推荐的函数, 它们无条件地使当前进程睡眠.

```
#include <linux/poll.h>
void poll_wait(struct file *filp, wait_queue_head_t *q,
poll_table *p);
```

将当前进程放入一个等待队列, 不立刻调度. 它被设计来被设备驱动的 poll 方法使用.

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

一个"帮忙者", 来实现 fasync 设备方法. mode 参数是传递给方法的相同的值, 而 fa 指针指向一个设备特定的 fasync_struct *.

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

如果这个驱动支持异步通知, 这个函数可用来发送一个信号到登记在 fa 中的进程.

```
int nonseekable_open(struct inode *inode, struct file
*filp);
loff_t no_llseek(struct file *file, loff_t offset, int
whence);
```

nonseekable_open 应当在任何不支持移位的设备的 open 方法中被调用. 这样的设备应当使用 no_llseek 作为它们的 llseek 方法.