第三章 字符驱动

这一章的目标是编写一个完整的字符设备驱动。我们选择开发字符驱动，因为这类驱动适合大多数简单的硬件设备。相比于块驱动或网络驱动（我们会在后面的章节中讨论），字符驱动更容易理解。我们最终的目标是编写一个模块化的字符驱动，但在这一章，我们不会讨论模块化的问题。

在整个章节中，我们会展示从一个真实设备驱动：scull（简单字符实用程序加载地方）中提取的代码片段。scull是一个字符驱动，它对内存区域进行操作，就像它是一个设备一样。在这一章中，由于scull的这种特性，我们将“设备”和“scull使用的内存区域”这两个词互换使用。

scull的优点是它不依赖于硬件。scull只是对一些从内核分配的内存进行操作。任何人都可以编译和运行scull，而且scull可以在Linux运行的计算机架构上移植。另一方面，这个设备除了演示内核和字符驱动之间的接口，并允许用户运行一些测试外，没有做任何“有用”的事情。

3.1scull的设计

驱动编写的第一步是定义驱动将为用户程序提供的功能（机制）。由于我们的“设备”是计算机内存的一部分，我们可以自由地对其进行操作。它可以是顺序或随机访问设备，可以是一个设备，也可以是多个设备，等等。

为了使scull作为编写真实设备的真实驱动的模板变得有用，我们将向你展示如何在计算机内存之上实现几种设备抽象，每种都有不同的特性。

scull源代码实现了以下设备。模块实现的每种设备都被称为一种类型.

- `scull0到scull3` 四个设备，每个设备都由一个既全局又持久的内存区域组成。全局意味着如果设备被多次打开，设备内的数据将被所有打开它的文件描述符共享。持久意味着如果设备被关闭并重新打开，数据不会丢失。这个设备很有趣，因为它可以使用常规命令，如cp、cat和shell I/O重定向等进行访问和测试。

- `scullpipe0到scullpipe3` 四个FIFO（先进先出）设备，它们像管道一样工作。一个进程读取另一个进程写入的内容。如果多个进程读取同一个设备，它们会争夺数据。scullpipe的内部将展示如何实现阻塞和非阻塞的读写，而不必求助于中断。虽然真实的驱动程序使用硬件中断与设备同步，但阻塞和非阻塞操作的主题是一个重要的问题，它与中断处理（在第10章中介绍）是分开的。

- `scullsingle scullpriv sculluid scullwuid` 这些设备类似于scull0，但在允许打开时有一些限制。第一个（scullsingle）一次只允许一个进程使用驱动，而scullpriv对每个虚拟控制台（或X终端会话）是私有的，因为每个控制台/终端上的进程获得的内存区域是不同的。sculluid和scullwuid可以被多次打开，但一次只能由一个用户打开；前者在另一个用户锁定设备时返回“设备忙”错误，而后者实现了阻塞打开。这些scull的变体似乎在混淆策略和机制，但它们值得一看，因为一些真实的设备需要这种类型的管理。

每个scull设备都展示了驱动的不同特性，并提出了不同的难题。本章介绍了scull0到scull3的内部；更高级的设备在第6章中介绍。scullpipe在“A Blocking I/O Example”部分中描述，其他设备在“Access Control on a Device File”部分中描述。

3.2主次编号

字符设备通过文件系统中的名称进行访问。这些名称被称为特殊文件或设备文件，或简单地称为文件系统树的节点；它们通常位于/dev目录中。字符驱动的特殊文件在`ls -l`的输出的第一列中由“c”标识。块设备也出现在/dev中，但它们由“b”标识。本章的重点是字符设备，但以下许多信息也适用于块设备。

如果你执行ls -l命令，你会在设备文件条目中看到两个数字（由逗号分隔），这些条目位于最后修改日期之前，通常显示文件长度的地方。这些数字是特定设备的主设备号和次设备号。下面的列表显示了一些在典型系统上出现的设备。它们的主设备号是1、4、7和10，而次设备号是1、3、5、64、65和129。

SHELL

```
crw-rw-rw- 1 root  root  1,  3 Apr 11  2002 null
crw------- 1 root  root  10, 1 Apr 11  2002 psaux
crw------- 1 root  root  4,  1 Oct 28 03:04 tty1
crw-rw-rw- 1 root  tty   4, 64 Apr 11  2002 ttys0
crw-rw---- 1 root  uucp  4, 65 Apr 11  2002 ttyS1
crw--w---- 1 vcsa  tty   7,  1 Apr 11  2002 vcs1
crw--w---- 1 vcsa  tty   7,129 Apr 11  2002 vcsa1
crw-rw-rw- 1 root  root  1,  5 Apr 11  2002 zero
```

传统上，主设备号用于标识与设备关联的驱动程序。例如，/dev/null和/dev/zero都由驱动程序1管理，而虚拟控制台和串行终端由驱动程序4管理；同样，vcs1和vcsa1设备都由驱动程序7管理。现代的Linux内核允许多个驱动程序共享主设备号，但你看到的大多数设备仍然遵循一主一驱动的原则。

次设备号被内核用来确定正在引用的确切设备。根据你的驱动程序的编写方式（我们将在下面看到），你可以从内核直接获取到你的设备的指针，或者你可以自己使用次设备号作为本地设备数组的索引。无论哪种方式，内核本身几乎不知道次设备号的任何信息，除了它们引用了由你的驱动程序实现的设备这一事实。

3.2.1. 设备编号的内部表示

在内核中，dev_t类型（在<linux/types.h>中定义）用于保存设备号——包括主设备号和次设备号。从内核的2.6.0版本开始，dev_t是一个32位的量，其中12位用于主设备号，20位用于次设备号。当然，你的代码永远不应该对设备号的内部组织做任何假设；相反，它应该使用在<linux/kdev_t.h>中找到的一组宏。要获取dev_t的主部分或次部分，使用：

```c
MAJOR(dev_t dev);

MINOR(dev_t dev);
```

如果你有主设备号和次设备号，并需要将它们转换为dev_t，使用：

```c
MKDEV(int major, int minor);
```

注意，2.6内核可以容纳大量的设备，而之前的内核版本只能容纳255个主设备号和255个次设备号。人们认为更宽的范围将在相当长的一段时间内足够使用，但计算领域充满了这种性质的错误假设。所以你应该预料到dev_t的格式在未来可能会再次改变；然而，如果你仔细编写你的驱动程序，这些变化将不会成为问题。

3.2.2. 分配和释放设备编号

当设置字符设备时，你的驱动程序需要做的第一件事就是获取一个或多个设备号来使用。这个任务需要的函数是`register_chrdev_region`，它在<linux/fs.h>中声明：

```c
int register_chrdev_region(dev_t first, unsigned int count,
char *name);
```

这里，first是你希望分配的范围的开始设备号。first的次设备号部分通常是0，但没有这样的要求。count是你请求的连续设备号的总数。注意，如果count很大，你请求的范围可能会溢出到下一个主设备号；但只要你请求的号码范围是可用的，一切都会正常工作。最后，name是应该与这个号码范围关联的设备的名称；它将出现在/proc/devices和sysfs中。

与大多数内核函数一样，如果分配成功执行，register_chrdev_region的返回值将为0。如果出现错误，将返回一个负的错误代码，你将无法访问请求的区域。

如果你提前知道你想要的确切设备号，`register_chrdev_region`就可以胜任。然而，你通常不会知道你的设备将使用哪些主设备号；Linux内核开发社区一直在努力转向使用动态分配的设备号。内核会很乐意为你动态分配一个主设备号，但你必须通过使用不同的函数来请求这个分配：

```c
int alloc_chrdev_region(dev_t *dev, unsigned int firstminor,
unsigned int count, char *name);
```

在这个函数中，dev是一个只输出的参数，成功完成后，它将保存你分配范围中的第一个号码。firstminor应该是请求使用的第一个次设备号；它通常是0。count和name参数的工作方式与给register_chrdev_region的参数相同。

无论你如何分配你的设备号，当它们不再使用时，你应该释放它们。设备号通过以下方式释放：

```c
void unregister_chrdev_region(dev_t first, unsigned int
count);
```

通常在你的模块的清理函数中调用`unregister_chrdev_region`。

上述函数为你的驱动程序的使用分配设备号，但它们并没有告诉内核你实际上会如何使用这些号码。在用户空间程序可以访问这些设备号之前，你的驱动程序需要将它们连接到实现设备操作的内部函数。我们将在稍后描述如何完成这个连接，但首先有一些必要的离题要处理。

3.2.3. 主编号的动态分配

一些主设备号被静态地分配给最常见的设备。这些设备的列表可以在内核源代码树的Documentation/devices.txt中找到。然而，静态号码已经被分配给你的新驱动程序的可能

性很小，而且新的号码不再被分配。所以，作为一个驱动程序编写者，你有一个选择：你可以简单地选择一个看起来未被使用的号码，或者你可以以动态的方式分配主号码。选择一个号码可能只有在你是驱动程序的唯一用户时才会有效；一旦你的驱动程序被更广泛地部署，随机选择的主号码将导致冲突和麻烦。因此，对于新的驱动程序，我们强烈建议你使用动态分配来获取你的主设备号，而不是从当前空闲的号码中随机选择一个。换句话说，你的驱动程序几乎肯定应该使用alloc_chrdev_region而不是register_chrdev_region。

- `alloc_chrdev_region`和`register_chrdev_region`都是用于在Linux内核中注册设备号的函数，但它们的使用方式和目的有所不同。

- `register_chrdev_region`函数用于注册一个已知的设备号范围。它的原型如下：

```c
int register_chrdev_region(dev_t first, unsigned int count,
char *name);
```

- 其中，`first`是你希望分配的设备号范围的开始设备号，`count`是你请求的连续设备号的总数，`name`是应该与这个号码范围关联的设备的名称。

- `alloc_chrdev_region`函数用于动态分配一个设备号范围。它的原型如下：

```c
int alloc_chrdev_region(dev_t *dev, unsigned int firstminor,
unsigned int count, char *name);
```

- 其中，`dev`是一个输出参数，成功返回后，它将保存你分配范围中的第一个设备号。`firstminor`是请求使用的第一个次设备号，通常为0。`count`是请求的设备号的数量，`name`是设备的名称。

- 这两个函数的主要区别在于，`register_chrdev_region`需要你提前知道你想要的设备号，而`alloc_chrdev_region`则会为你动态分配一个设备号。因此，如果你不知道哪个设备号可用，或者你不想硬编码设备号，那么你应该使用`alloc_chrdev_region`。

- 无论你使用哪个函数，当你不再需要设备号时，你都应该使用`unregister_chrdev_region`函数来释放它们。

动态分配的缺点是你不能提前创建设备节点，因为分配给你的模块的主号码会变化。对于驱动程序的正常使用，这几乎不是问题，因为一旦号码被分配，你可以从/proc/devices中读取它。

因此，要加载使用动态主号码的驱动程序，insmod的调用可以被一个简单的脚本替换，该脚本在调用insmod后，读取/proc/devices以创建特殊文件。

一个典型的/proc/devices文件看起来像下面这样：

```shell
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 ttyS
  6 lp
  7 vcs
 10 misc
 13 input
 14 sound
 21 sg
180 usb
Block devices:
  2 fd
  8 sd
 11 sr
 65 sd
 66 sd
```

下面这个脚本是用于加载已经被分配了动态设备号的模块的。它使用awk工具从/proc/devices中获取信息，然后在/dev中创建设备文件。

以下是scull_load脚本，它是scull分发包的一部分。驱动程序的用户可以从系统的rc.local文件中调用这样的脚本，或者在需要模块时手动调用它。

SHELL

```shell
#!/bin/sh

module="scull"

device="scull"

mode="664"

# 调用insmod加载模块,如果加载失败则退出

/sbin/insmod ./$module.ko $* || exit 1

# 删除旧的设备节点

rm -f /dev/${device}[0-3]

# 从/proc/devices获取主设备号

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

# 创建设备节点

mknod /dev/${device}0 c $major 0

mknod /dev/${device}1 c $major 1

mknod /dev/${device}2 c $major 2

mknod /dev/${device}3 c $major 3

# 设置适当的组和权限
#首先，检查系统中是否存在"staff"组，如果不存在，则使用"wheel"组。然后，
#使用`chgrp`命令更改设备节点的组，使用`chmod`命令更改设备节点的权限。

group="staff"

grep -q '^staff:' /etc/group || group="wheel"

chgrp $group /dev/${device}[0-3]
```

```
chmod $mode  /dev/${device}[0-3]
```

这个脚本可以通过重新定义变量和调整mknod行来适应另一个驱动程序。上面显示的脚本创建了四个设备，因为四是scull源代码中的默认值。

这段脚本的最后几行可能看起来有些难懂：为什么要改变设备的组和模式呢？原因是这个脚本必须由超级用户运行，所以新创建的特殊文件是由root拥有的。权限位默认只允许root有写入权限，而任何人都可以获得读取权限。通常，一个设备节点需要不同的访问策略，所以以某种方式改变访问权限是必要的。我们脚本中的默认设置是给一组用户访问权限，但你的需求可能会有所不同。在第6章的“设备文件的访问控制”一节中，sculluid的代码演示了驱动程序如何实施其自己的设备访问授权类型。还有一个scull_unload脚本可以清理/dev目录并移除模块。

作为加载和卸载的替代方案，你可以写一个init脚本，准备放在你的发行版用于这些脚本的目录中。作为scull源的一部分，我们提供了一个相当完整和可配置的init脚本示例，叫做scull.init；它接受常规的参数`start`、`stop`和`restart`并执行`scull_load`和`scull_unload`的角色。如果反复创建和销毁/dev节点听起来过于繁琐，有一个有用的解决方法。如果你只是加载和卸载一个驱动程序，你可以在第一次用你的脚本创建特殊文件后，只使用rmmod和insmod：动态数字不是随机的，你可以依赖同一个数字每次都会被选择，如果你不加载任何其他（动态）模块的话。避免冗长的脚本在开发过程中是有用的。但显然，这个技巧不能扩展到一次处理多个驱动程序。

- `/sbin/insmod ./$module.ko $* || exit 1`
- `$*`是一个特殊的shell变量，它代表了所有的命令行参数。
- 当你运行这个脚本并传递一些参数时，`$*`会被替换为你传递的所有参数。例如，如果你运行`./scull_load arg1 arg2 arg3`，那么在脚本中，`$*`就会被替换为`arg1 arg2 arg3`。
- 如果`insmod`命令失败（返回非零值），`||`操作符将导致脚本执行`exit 1`命令，这将立即终止脚本并返回错误代码1.
- `rm -f /dev/${device}[0-3]`
- `/dev/${device}[0-3]`会匹配到`/dev/scull0`，`/dev/scull1`，`/dev/scull2`和`/dev/scull3`这四个文件名。`[0-3]`是一个shell的文件名扩展（globbing）特性.

- Shell的文件名扩展（globbing）是一种用于匹配文件名的功能。它允许你使用特殊的模式来匹配一个或多个文件名，而不需要明确指定完整的文件名。以下是一些常用的文件名扩展模式：

1. `*`：匹配任何数量的任何字符。例如，`*.txt`会匹配所有以`.txt`结尾的文件。
2. `?`：匹配任何单个字符。例如，`?.txt`会匹配所有只有一个字符并以`.txt`结尾的文件，如`a.txt`，`b.txt`等。
3. `[...]`：匹配方括号中的任何一个字符。例如，`[abc].txt`会匹配`a.txt`，`b.txt`和`c.txt`。
4. `[!...]`或`[^...]`：匹配方括号中没有列出的任何字符。例如，`[!abc].txt`或`[^abc].txt`会匹配除`a.txt`，`b.txt`和`c.txt`之外的任何文件。
5. `{...}`：匹配大括号中的任何一个字符串（这些字符串由逗号分隔）。例如，`{a,b,c}.txt`会匹配`a.txt`，`b.txt`和`c.txt`。

- `major=$(awk "$2=="$module" {print $1}" /proc/devices)`

- 这行命令使用了`awk`，一个强大的文本处理工具，从`/proc/devices`文件中提取主设备号。

- `awk`命令的参数是一个脚本，这个脚本定义了`awk`应该如何处理输入的每一行。在这个例子中，脚本是`"$2=="$module" {print $1}"`。

- `$2=="$module"`：这是一个条件表达式，它检查每一行的第二个字段（字段之间由空格或制表符分隔）是否等于变量`$module`的值。在这个脚本中，`$module`的值是"scull"。注意，这里的`$2`是`awk`的语法，表示当前行的第二个字段，而`$module`是shell变量。

- `{print $1}`：这是一个动作，当上述条件满足时，它会执行。这个动作打印每一行的第一个字段，也就是主设备号。

- 整个命令的作用是：从`/proc/devices`文件中找到第二个字段等于"scull"的行，然后打印出这些行的第一个字段（主设备号）。

- `major=$(awk "$2=="$module" {print $1}" /proc/devices)`这行命令将`awk`的输出赋值给shell变量`major`。因此，`major`变量将包含"scull"模块的主设备号。

- 在Shell脚本中，`$(命令)`是命令替换的一种形式。它的作用是执行括号中的命令，并将其输出替换到原位置。

- 例如，如果你有一个命令`echo $(date)`，Shell首先会执行括号中的`date`命令，然后将`date`命令的输出替换到原位置，最后执行`echo`命令。所以，`echo $(date)`的

效果等同于直接在命令行中输入 `date` 命令并回车.

- `grep -q '^staff:' /etc/group || group="wheel"`
- `grep` 命令用于在文本中搜索匹配的字符串。在这个例子中，`grep` 命令搜索 `/etc/group` 文件中以"staff:"开头的行。`-q` 选项告诉 `grep` 在找到匹配的行后立即退出，而不输出任何内容。
- `mknod` 是一个Unix和Linux系统中的命令，用于创建字符设备文件或块设备文件。`mknod` 命令的基本语法是：

```c
mknod [选项]... 名称 类型 [主设备号 次设备号]
```

- **名称**：要创建的设备文件的名称。
- **类型**：设备的类型，可以是 `b`（块设备）、`c` 或 `u`（字符设备）、`p`（命名管道）。
- **主设备号** 和 **次设备号**：设备的主设备号和次设备号。这两个参数只对字符设备和块设备有效。
- 例如，`mknod /dev/mydevice c 10 7` 命令会创建一个名为 `/dev/mydevice` 的字符设备文件，其主设备号是10，次设备号是7。
- 需要注意的是，`mknod` 命令通常需要超级用户权限才能执行。在现代的Linux系统中，设备文件通常由udev或devfs这样的设备管理器自动创建，而不是手动使用 `mknod` 命令创建。

在我们看来，分配主要编号的最佳方式是默认使用动态分配，同时留给自己在加载时或甚至在编译时指定主要编号的选项。scull的实现方式就是这样；它使用一个全局变量，scull_major，来保存选定的编号（还有一个scull_minor用于次要编号）。这个变量被初始化为SCULL_MAJOR，这在scull.h中定义。在分发的源代码中，SCULL_MAJOR的默认值是0，意味着“使用动态分配”。用户可以接受默认值，或者通过在编译前修改宏或在insmod命令行上指定一个scull_major的值来选择一个特定的主要编号。最后，通过使用scull_load脚本，用户可以在scull_load的命令行上向insmod传递参数。

这是我们用在 scull 的源码中获取主编号的代码:

```c
if (scull_major) {
 dev = MKDEV(scull_major, scull_minor);
 result = register_chrdev_region(dev, scull_nr_devs,
"scull");
} else {
 result = alloc_chrdev_region(&dev, scull_minor,
scull_nr_devs, "scull");
 scull_major = MAJOR(dev);
}
if (result < 0) {
 printk(KERN_WARNING "scull: can't get major %d\n",
scull_major);
 return result;
}
```

本书使用的几乎所有例子驱动使用类似的代码来分配它们的主编号.

3.3. 一些重要数据结构

你可以想象，设备号注册只是驱动程序代码必须执行的许多任务中的第一个。我们很快就会看到其他重要的驱动程序组件，但首先需要偏离一下话题。大多数基本的驱动程序操作涉及到三个重要的内核数据结构，分别是file_operations、file和inode。要做任何有趣的事情，基本上需要熟悉这些结构，所以在我们深入了解如何实现基本的驱动程序操作的细节之前，我们现在将快速地看一下它们每一个。

3.3.1. 文件操作file_operations

- 文件操作（file operations）是Linux内核中的一个重要概念，它定义了一组函数指针，这些函数指针对应了在文件或设备上可以执行的各种操作，如打开、读取、写入、关闭等。这些操作是通过一个名为`file_operations`的结构体来定义的。

- 当你在编写一个设备驱动程序时，你需要定义一个`file_operations`结构体，来告诉内核你的驱动程序支持哪些操作。例如，如果你的驱动程序支持读取设备，你需要提供一个`read`函数，并将其地址赋给`file_operations`结构体的`read`成员。

- 你应该掌握的程度取决于你的需求。如果你只是想了解Linux内核的基本工作原理，那么理解`file_operations`结构体的基本概念和作用就足够了。但如果你打算编写设备驱动程序，那么你就需要深入理解每个文件操作的具体含义，以及如何实现它们。

- 在实际编程中，你可能不需要实现`file_operations`结构体中的所有操作。例如，许多设备并不支持映射（mmap）操作，因此你可以将`mmap`成员设置为NULL。你应该根据你的设备的具体功能和需求来决定需要实现哪些操作。

到目前为止，我们已经为我们的使用保留了一些设备号，但我们还没有将任何驱动程序的操作连接到这些号码。file_operations结构就是字符驱动程序设置这种连接的方式。这个结构在<linux/fs.h>中定义，是一组函数指针的集合。每个打开的文件（在内部用一个我们稍后将要检查的file结构表示）都与它自己的一组函数相关联（通过包含一个指向file_operations结构的f_op字段）。这些操作主要负责实现系统调用，因此，被命名为open、read等。我们可以把文件看作是一个“对象”，操作它的函数看作是它的“方法”，使用面向对象编程的术语来表示由对象声明的对自身的操作。这是我们在Linux内核中看到的面向对象编程的第一个迹象，我们将在后面的章节中看到更多。

通常，一个file_operations结构或一个指向它的指针被称为fops（或者其变体）。结构中的每个字段必须指向驱动程序中实现特定操作的函数，或者对于不支持的操作被留空。当指定一个NULL指针时，内核的确切行为对于每个函数都是不同的，正如本节后面的列表所示。

下面的列表介绍了一个应用程序可以在设备上调用的所有操作。我们试图保持列表的简洁，以便它可以被用作参考，只是简单地总结每个操作和当使用NULL指针时默认的内核行为。

在你阅读file_operations方法的列表时，你会注意到许多参数包含了__user字符串。这个注解是一种文档形式，指出一个指针是一个用户空间地址，不能直接解引用。对于正常的编译，__user没有任何效果，但它可以被外部检查软件用来找出用户空间地址的误用。

在描述了一些其他重要的数据结构后，本章的其余部分解释了最重要的操作的角色，并提供了提示、警告和真实的代码示例。我们推迟讨论更复杂的操作到后面的章节，因为我们还没有准备好深入研究诸如内存管理、阻塞操作和异步通知等主题。

```
struct module *owner
```

- 第一个file_operations字段根本不是一个操作；它是一个指向“拥有”结构的模块的指针。这个字段被用来防止模块在其操作正在使用时被卸载。几乎所有的时间，它都被简单地初始化为THIS_MODULE，这是在<linux/module.h>中定义的宏。

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

- llseek方法用于改变文件中当前的读/写位置，新的位置作为一个（正的）返回值返回。loff_t参数是一个“长偏移”，即使在32位平台上也至少有64位宽。错误通过负的返回值表示。如果这个函数指针是NULL，seek调用将以可能无法预测的方式修改文件结构中的位置计数器（在“文件结构”一节中描述）。

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

- 用于从设备中检索数据。这个位置的空指针会导致read系统调用失败，返回-EINVAL（“无效的参数”）。非负的返回值表示成功读取的字节数（返回值是一个“有符号大小”类型，通常是目标平台的本地整数类型）。

```
ssize_t (*aio_read)(struct kiocb *, char __user *, size_t, loff_t *);
```

- 启动一个异步读取操作——一个可能在函数返回之前还未完成的读取操作。如果这个方法是NULL，所有的操作都将由read（同步地）处理。

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

- 向设备发送数据。如果为NULL，程序调用write系统调用时返回-EINVAL。如果返回值为非负，表示成功写入的字节数。

```
ssize_t (*aio_write)(struct kiocb *, const char __user *, size_t, loff_t *);
```

- 在设备上启动一个异步写操作。

```
int (*readdir) (struct file *, void *, filldir_t);
```

- 对于设备文件，这个字段应该为NULL；它用于读取目录，只对文件系统有用。

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

- poll方法是三个系统调用的后端：poll、epoll和select，所有这些都用于查询对一个或多个文件描述符的读或写是否会阻塞。poll方法应返回一个位掩码，指示是否可以进行非阻塞读或写，可能还向内核提供信息，可以用来将调用进程置于休眠状态，直到I/O变得可能。如果驱动程序将其poll方法设置为NULL，设备被假定为可读写且不会阻塞。

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

- ioctl系统调用提供了一种发出设备特定命令的方式（例如格式化软盘的一条轨道，这既不是读也不是写）。此外，内核会识别一些不需要参考fops表的ioctl命令。如果设备没有提供ioctl方法，系统调用会为任何未预定义的请求返回错误（-ENOTTY，“设备没有这样的ioctl”）。

```
int (*mmap) (struct file *, struct vm_area_struct *);
```

- mmap用于请求将设备内存映射到进程的地址空间。如果此方法为NULL，mmap系统调用返回-ENODEV。

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

- 虽然这总是在设备文件上执行的第一个操作，但驱动程序不需要声明相应的方法。如果此项为NULL，打开设备总是成功的，但你的驱动程序不会收到通知。

```
int (*flush) (struct file *);
```

- 当进程关闭其对设备的文件描述符的副本时，会调用flush操作；它应执行（并等待）设备上的任何未完成的操作。这不应与用户程序请求的fsync操作混淆。目前，很少有驱动程序使用flush；例如，SCSI磁带驱动程序使用它，以确保所有写入的数据在设备关闭之前都到达磁带。如果flush为NULL，内核简单地忽略用户应用程序的请求。

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

- 当文件结构被释放时，会调用此操作。像open一样，release也可以为NULL。

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

- 此方法是fsync系统调用的后端，用户调用它来刷新任何待处理的数据。如果此指针为NULL，系统调用返回-EINVAL。

```
int (*aio_fsync)(struct kiocb *, int);
```

- 这是fsync方法的异步版本。

```
int (*fasync) (int, struct file *, int);
```

- 此操作用于通知设备其FASYNC标志发生了变化。异步通知是一个高级主题，在第6章中有描述。如果驱动程序不支持异步通知，此字段可以为NULL。

```
int (*lock) (struct file *, int, struct file_lock *);
```

- lock方法用于实现文件锁定；锁定是常规文件的必不可少的特性，但几乎从未由设备驱动程序实现。

```
ssize_t (*readv) (struct file *, const struct iovec *, unsigned
long, loff_t *);
ssize_t (*writev) (struct file *, const struct iovec *, unsigned
long, loff_t *);
```

- 这些方法实现了散列/聚集读写操作。应用程序偶尔需要进行涉及多个内存区域的单个读或写操作；这些系统调用允许它们这样做，而不强制对数据进行额外的复制操作。如果这些函数指针被设置为NULL，那么将调用read和write方法（可能不止一次）。

```
ssize_t (*sendfile)(struct file *, loff_t *, size_t, read_actor_t,
void *);
```

- 此方法实现了sendfile系统调用的读取部分，该调用将数据从一个文件描述符移动到另一个文件描述符，复制的次数最少。例如，需要通过网络连接发送文件内容的web服务器会使用它。设备驱动程序通常将sendfile设置为NULL。

```
ssize_t (*sendpage) (struct file *, struct page *, int, size_t,
loff_t *, int);
```

- sendpage是sendfile的另一半；内核调用它，一次发送一页数据到对应的文件。设备驱动程序通常不实现sendpage。

```
unsigned long (*get_unmapped_area)(struct file *, unsigned long,
unsigned long, unsigned long, unsigned long);
```

- 此方法的目的是在进程的地址空间中找到一个适合映射底层设备上的内存段的位置。这个任务通常由内存管理代码执行；此方法存在是为了让驱动程序能够强制执行特定设备可能有的任何对齐要求。大多数驱动程序可以将此方法设置为NULL。

```
int (*check_flags)(int)
```

- 此方法允许模块检查传递给fcntl(F_SETFL...)调用的标志。

```
int (*dir_notify)(struct file *, unsigned long);
```

- 当应用程序使用fcntl请求目录更改通知时，会调用此方法。它只对文件系统有用；驱动程序不需要实现dir_notify。

scull 设备驱动只实现最重要的设备方法. 它的 file_operations 结构体如下:

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

采用C语言中的标签结构体初始化语法的优点在于，它使得驱动程序在结构体定义发生变化时更具有可移植性，而且可以说，它使代码更紧凑、更易读。

标签初始化允许重新排序结构体成员；在某些情况下，通过将指向频繁访问的成员的指针放在同一硬件缓存行中，实现了显著的性能提升。

例如，你可以这样初始化一个结构体：

```c
struct file_operations fops = {

    .read = device_read,

    .write = device_write,

    .open = device_open,

    .release = device_release

};
```

在这个例子中，`file_operations`是一个已经定义好的结构体类型，它定义在Linux内核中，用于描述文件操作。`.read`、`.write`、`.open`和`.release`就是标签，`device_read`、`device_write`、`device_open`和`device_release`是对应的函数指针。这种初始化方式的好处是，即使`file_operations`结构体的定义发生了变化，只要这些成员的名称没有变，你的代码就不需要修改。`fops`是一个`file_operations`类型的变

量。你可以将你的设备支持的操作赋给 `fops` 的相应成员。例如，如果你的设备支持读取操作，你可以提供一个函数 `device_read`，然后将其地址赋给 `fops.read`

3.3.2. 文件结构file Structure

在设备驱动程序中，定义在<linux/fs.h>中的 `struct file` 是第二重要的数据结构。注意，文件与用户空间程序的FILE指针无关。FILE是在C库中定义的，永远不会出现在内核代码中。另一方面，struct file是一个内核结构，永远不会出现在用户程序中。

file结构代表一个打开的文件。（它并不特定于设备驱动程序；系统中的每个打开的文件都有一个关联的struct file在内核空间。）它由内核在打开时创建，并传递给任何操作文件的函数，直到最后关闭。在所有文件实例都关闭后，内核会释放数据结构。

在内核源代码中，指向struct file的指针通常被称为 `file` 或 `filp`（“file pointer”）。我们将一致地称指针为filp，以防止与结构本身的歧义。因此，file指的是结构，filp指的是指向结构的指针。

struct file的最重要字段在这里显示。就像在前一节中一样，首次阅读时可以跳过列表。然而，稍后在本章中，当我们面对一些真正的C代码时，我们将更详细地讨论这些字段。

```
mode_t f_mode;
```

- f_mode是一个mode_t类型的字段，它通过FMODE_READ和FMODE_WRITE位标识文件是可读的、可写的还是两者都是。你可能想在你的open或ioctl函数中检查此字段的读/写权限，但你不需要检查read和write的权限，因为内核在调用你的方法之前会检查。如果文件没有被打开以进行那种类型的访问，尝试读取或写入将被拒绝，而驱动程序甚至不知道这一点。

```
loff_t f_pos;
```

- f_pos是一个loff_t类型的字段，表示当前的读取或写入位置。loff_t在所有平台上都是64位值（在gcc术语中是long long）。如果驱动程序需要知道文件的当前位置，可以读取此值，但通常不应改变它；read和write应使用它们接收的最后一个参数更新位置，而不是直接操作filp→f_pos。这个规则的唯一例外是在llseek方法中，其目的是改变文件位置。

```
unsigned int f_flags;
```

- f_flags是一个unsigned int类型的字段，表示文件标志，如O_RDONLY、O_NONBLOCK和O_SYNC。驱动程序应检查O_NONBLOCK标志，看是否请求了非阻塞操作（我们在第6章的“阻塞和非阻塞操作”一节中讨论了非阻塞I/O）；其他标志很少使用。特别是，应使用f_mode而不是f_flags检查读/写权限。所有的标志都在头文件<linux/fcntl.h>中定义。

```
struct file_operations *f_op;
```

- f_op是一个指向file_operations结构的指针，表示与文件关联的操作。内核在实现open的一部分时分配指针，然后在需要分派任何操作时读取它。filp→f_op中的值永远不会被内核保存以供以后参考；这意味着你可以改变与你的文件关联的文件操作，新的方法在你返回给调用者后将生效。例如，与主设备号1关联的open代码（/dev/null、/dev/zero等）根据正在打开的次设备号替换filp→f_op中的操作。这种做法允许在同一个主设备号下实现多种行为，而不需要在每个系统调用时引入开销。替换文件操作的能力是内核对面向对象编程中“方法覆盖”的等价物。

```
void *private_data;
```

- private_data是一个void *类型的字段，open系统调用在调用驱动程序的open方法之前将此指针设置为NULL。你可以自由地使用这个字段，或者忽略它；你可以使用这个字段指向分配的数据，但是你必须记住在文件结构被内核销毁之前在release方法中释放那个内存。private_data是一个保存系统调用之间状态信息的有用资源，我们的大多数示例模块都使用它。

```
struct dentry *f_dentry;
```

- f_dentry是一个指向dentry结构的指针，表示与文件关联的目录条目（dentry）结构。设备驱动程序编写者通常不需要关心dentry结构，除了访问inode结构作为filp->f_dentry→d_inode。

实际的结构还有一些其他的字段，但对设备驱动程序来说并不有用。我们可以安全地忽略这些字段，因为驱动程序从不创建file结构；他们只访问在其他地方创建的结构。

3.3.3. inode 结构

inode结构是Linux文件系统中的一个重要概念，它在内核中用于表示文件。每个文件在文件系统中都有一个与之关联的inode，它包含了大量关于文件的信息，如文件大小、文件类型（普通文件、目录、设备文件等）、文件的所有者、文件的权限、文件的创建时间、最后访问时间等.

inode结构与file结构不同，file结构表示的是一个打开的文件描述符，而inode结构表示的是文件本身。对于一个文件，可能有多个file结构与之关联（当多个进程打开同一个文件时），但这些file结构都指向同一个inode结构。

对于设备驱动程序来说，inode结构中主要关注两个字段:

```
dev_t i_rdev;
```

- i_rdev是一个dev_t类型的字段，对于表示设备文件的inode，这个字段包含实际的设备号。

```
struct cdev *i_cdev;
```

- i_cdev是一个指向cdev结构的指针，cdev是内核的内部结构，代表字符设备；当inode指向一个字符设备文件时，这个字段包含一个指向内核内部表示字符设备的cdev结构的指针。

在2.5开发系列的过程中，i_rdev的类型发生了变化，导致许多驱动程序出现问题。为了鼓励更多的可移植编程，内核开发人员添加了两个宏，可以用来从inode获取主设备号和次设备号：

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

`iminor`是一个宏，接收一个指向`inode`的指针，返回该`inode`的次设备号。

`imajor`也是一个宏，接收一个指向`inode`的指针，返回该`inode`的主设备号。

为了不被下一次改变所捕获，应该使用这些宏，而不是直接操作`i_rdev`。

3.4. 字符设备注册

如我们之前提到的，内核使用类型为struct cdev的结构来内部表示字符设备。在内核调用你的设备的操作之前，你必须分配并注册一个或多个这样的结构。为了做到这一点，你的代码应该包含<linux/cdev.h>，在这里定义了结构和其相关的辅助函数。

有两种分配和初始化这些结构的方法。如果你希望在运行时获取一个独立的cdev结构，你可以使用如下代码：

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

然而，你可能希望将cdev结构嵌入到你自己的设备特定结构中，这就是scull所做的。在这种情况下，你应该用以下代码初始化你已经分配的结构：

```c
void cdev_init(struct cdev *cdev, struct file_operations
*fops);
```

无论哪种方式，都有一个你需要初始化的struct cdev字段。就像file_operations结构一样，struct cdev有一个owner字段，应该设置为THIS_MODULE。

一旦cdev结构设置好了，最后一步是通过调用以下函数告诉内核：

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

这里，dev是cdev结构，num是此设备响应的第一个设备号，count是应与设备关联的设备号的数量。通常count是一，但在某些情况下，让一个特定的设备对应多个设备号是有意义的。例如，考虑SCSI磁带驱动器，它允许用户空间通过为每个物理设备分配多个次要号来选择操作模式（如密度）。

在使用cdev_add时，有几个重要的事情需要记住。首先，这个调用可能会失败。如果它返回一个负的错误代码，你的设备就没有被添加到系统中。然而，它几乎总是成功的，这就引出了另一个问题：一旦cdev_add返回，你的设备就是"活跃"的，它的操作可以被内核调用。你不应该在你的驱动程序完全准备好处理设备操作之前调用cdev_add。

要从系统中移除一个字符设备，调用：

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

显然，你在将其传递给cdev_del后，不应该访问cdev结构。

3.4.1. scull 中的设备注册

在内部，scull用类型为struct scull_dev的结构表示每个设备。这个结构定义如下：

```c
struct scull_dev {
    struct scull_qset *data;  /* 指向第一个量子集的指针 */
    int quantum;              /* 当前量子大小 */
    int qset;                 /* 当前数组大小 */
    unsigned long size;       /* 这里存储的数据量 */
    unsigned int access_key;  /* 由sculluid和scullpriv使用 */
    struct semaphore sem;     /* 互斥信号量 */
    struct cdev cdev;         /* 字符设备结构 */
};
```

我们在讨论到这个结构的各个字段时会进行讨论，但现在，我们要注意cdev，这是一个struct cdev，它将我们的设备接口到内核。这个结构必须按照上述描述进行初始化并添加到系统中；处理这个任务的scull代码是：

```c
static void scull_setup_cdev(struct scull_dev *dev, int
index)

{

    int err, devno = MKDEV(scull_major, scull_minor +
index);

    cdev_init(&dev->cdev, &scull_fops);

    dev->cdev.owner = THIS_MODULE;

    dev->cdev.ops = &scull_fops;

    err = cdev_add (&dev->cdev, devno, 1);

    /* 如果需要，优雅地失败 */

    if (err)

    printk(KERN_NOTICE "Error %d adding scull%d", err,
index);

}
```

由于cdev结构嵌入在struct scull_dev中，必须调用cdev_init来初始化该结构。

3.4.2. 老方法

如果你深入研究2.6内核中的大量驱动程序代码，你可能会注意到相当多的字符驱动程序并没有使用我们刚刚描述的cdev接口。你看到的是还没有升级到2.6接口的旧代码。由于这些代码本身就能工作，所以这种升级可能还需要很长时间。为了完整性，我们描述了旧的字符设备注册接口，但新的代码不应该使用它；这种机制可能会在未来的内核中消失。

注册字符设备驱动程序的经典方法是使用：

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

这里，major是感兴趣的主要号码，name是驱动程序的名称（它出现在/proc/devices中），fops是默认的file_operations结构。调用register_chrdev为给定的主要号码注册0-255的次要号码，并为每个号码设置一个默认的cdev结构。使用这个接口的驱动程序必须准备好处理所有256个次要号码的打开调用（无论它们是否对应于真实的设备），并且它们不能使用大于255的主要或次要号码。

如果你使用register_chrdev，从系统中移除你的设备的正确函数是：

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

major和name必须与传递给register_chrdev的相同，否则调用将失败。

3.5. open 和 release

到此我们已经快速浏览了这些成员, 我们开始在真实的 scull 函数中使用它们.

3.5.1. open 方法

open方法是为驱动程序提供的，用于做任何准备后续操作的初始化。在大多数驱动程序中，open应执行以下任务：

- 检查设备特定的错误（如设备未准备好或类似的硬件问题）
- 如果是第一次打开设备，则初始化设备
- 必要时更新 `f_op` 指针
- 分配并填充要放入 `filp->private_data` 的任何数据结构

然而，通常首要的任务是识别正在打开的设备。记住，open方法的原型是：

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

inode参数以其i_cdev字段的形式包含我们需要的信息，该字段包含我们之前设置的cdev结构。唯一的问题是我们通常不希望得到cdev结构本身，而是希望得到包含该cdev结构的scull_dev结构。C语言允许程序员玩各种技巧来进行这种转换；然而，编程这种技巧容易出错，并导致其他人难以阅读和理解的代码。幸运的是，在这种情况下，内核黑客已经为我们做了这些棘手的事情，以container_of宏的形式，定义在<linux/kernel.h>中：

```c
container_of(pointer, container_type, container_field);
```

这个宏接受一个指向名为`container_field`的字段的指针，该字段在类型为`container_type`的结构中，并返回一个指向包含结构的指针。在`scull_open`中，这个宏用于找到适当的设备结构：

```c
struct scull_dev *dev; /* 设备信息 */

dev = container_of(inode->i_cdev, struct scull_dev, cdev);

filp->private_data = dev; /* 用于其他方法 */
```

一旦找到scull_dev结构，scull就将一个指针存储在file结构的private_data字段中，以便在未来更容易地访问。

识别正在打开的设备的另一种方法是查看存储在inode结构中的次要号码。如果你使用register_chrdev注册你的设备，你必须使用这种技术。确保使用iminor从inode结构获取次要号码，并确保它对应于你的驱动程序实际准备处理的设备。

scull_open的代码（稍微简化）是：

```c
int scull_open(struct inode *inode, struct file *filp)

{

    struct scull_dev *dev; /* 设备信息 */

    dev = container_of(inode->i_cdev, struct scull_dev,
cdev);

    filp->private_data = dev; /* 用于其他方法 */

    /* 如果打开是只写的，现在将设备的长度修剪为0 */

    if ( (filp->f_flags & O_ACCMODE) == O_WRONLY) {

        scull_trim(dev); /* 忽略错误 */

    }

    return 0;          /* 成功 */

}
```

代码看起来相当稀疏，因为当调用open时，它不做任何特定的设备处理。它不需要，因为scull设备在设计上是全局的和持久的。具体来说，没有像“首次打开时初始化设备”这样的操作，因为我们不为sculls保持打开计数。

对设备执行的唯一真正操作是：当设备打开用于写入时，将其截断到长度为0。这是因为，按设计，用较短的文件覆盖scull设备会导致设备数据区域变短。这类似于打开一个常规文件用于写入会将其截断到零长度。如果设备打开用于读取，该操作不做任何事情。

我们将在后面看到当我们查看其他scull个性的代码时，真正的初始化是如何工作的。

- `container_of`宏是一个在Linux内核编程中常用的宏，它用于获取包含某个成员的结构体的指针。
- 这个宏的定义如下：

```c
#define container_of(ptr, type, member) ({                  \
    const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
    (type *)( (char *)__mptr - offsetof(type,member) );})
```

- 这个宏接受三个参数：一个指针 `ptr`，一个类型 `type`，和一个成员名 `member`。`ptr` 是指向 `type` 类型中 `member` 成员的指针。`container_of` 宏通过这个成员的指针来获取包含这个成员的整个结构体的指针。
- 例如，如果你有一个结构体 `struct my_struct`，它有一个成员 `int my_field`，并且你有一个指向 `my_field` 的指针 `int *p`。你可以使用 `container_of(p, struct my_struct, my_field)` 来获取一个指向整个 `my_struct` 结构体的指针。
- 这个宏在内核编程中非常有用，因为它允许你从结构体的一个部分（例如，一个回调函数或一个数据成员）轻松地获取到整个结构体。

3.5.2. release 方法

release方法的角色是open的反向操作。有时你会发现方法实现被称为device_close而不是device_release。无论哪种方式，设备方法都应执行以下任务：

- 释放open在filp->private_data中分配的任何东西(释放在 `open` 方法中分配的任何资源)
- 如果这是最后一次关闭设备文件，那么还需要关闭设备。

scull的基本形式没有硬件需要关闭，所以需要的代码最小：

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

你可能会想知道当设备文件被关闭的次数多于它被打开的次数时会发生什么。毕竟，dup和fork系统调用在不调用open的情况下创建了打开文件的副本；然后在程序终止时关闭那些副本。例如，大多数程序不打开他们的stdin文件（或设备），但是所有的程序最终都会关闭它。驱动程序如何知道一个打开的设备文件何时真正被关闭呢?

答案很简单：并非每个close系统调用都会导致调用release方法。只有那些实际释放设备数据结构的调用才会调用该方法——因此它的名字。内核保持一个文件结构被使用的次数的计数器。fork和dup都不创建新的文件结构（只有open做到了）；他们只是在现有结构中增加计数器。当文件结构的计数器降到0时，也就是结构被销毁时，close系统调用只执行release方法。release方法和close系统调用之间的这种关系保证了你的驱动程序对每个open只看到一个release调用。

注意，每次应用程序调用close时，都会调用flush方法。然而，很少有驱动程序实现flush，因为通常在关闭时除非涉及到release，否则没有什么要执行的。

如你所想，即使应用程序在没有明确关闭其打开的文件的情况下终止，前面的讨论也适用：内核在进程退出时自动关闭任何文件，通过内部使用close系统调用。

- 为什么设备文件可能会被关闭的次数多于它被打开的次数?这是因为 `dup` 和 `fork` 系统调用可以在不调用 `open` 的情况下创建打开文件的副本，然后在程序终止时关闭这些副本。

3.6. scull 的内存使用

在介绍读写操作之前，我们最好先看看scull是如何以及为什么进行内存分配的。"如何"是为了深入理解代码，"为什么"则展示了驱动程序编写者需要做出的选择，尽管scull作为设备并不典型。 本节只讨论scull中的内存分配策略，并未展示编写真实驱动程序所需的硬件管理技巧。这些技巧将在第9章和第10章中介绍。因此，如果你对理解面向内存的scull驱动程序的内部工作原理不感兴趣，你可以跳过这一节。

scull使用的内存区域，也称为设备，长度可变。你写的越多，它就越长；通过用较短的文件覆盖设备来进行修剪。 scull驱动程序引入了两个用于在Linux内核中管理内存的核心函数。这些函数在<linux/slab.h>中定义，分别是：

```c
void *kmalloc(size_t size, int flags);
void kfree(void *ptr);
```

调用kmalloc试图分配size字节的内存；返回值是指向该内存的指针，如果分配失败则返回NULL。flags参数用于描述应如何分配内存；我们将在第8章详细研究这些标志。现在，我

们总是使用GFP_KERNEL。应使用kfree释放分配的内存。你绝对不能将从kmalloc获取的以外的任何东西传递给kfree。然而，将NULL指针传递给kfree是合法的。

kmalloc并不是分配大量内存的最有效方式（参见第8章），所以scull选择的实现方式并不特别聪明。一个聪明的实现的源代码会更难阅读，而这一节的目标是展示读和写，而不是内存管理。这就是为什么代码只使用kmalloc和kfree，而不采用分配整个页面的方式，尽管那种方式会更有效。

另一方面，我们不想限制“设备”区域的大小，这既有哲学原因，也有实际原因。从哲学角度看，对被管理的数据项设置任意限制总是一个坏主意。从实际角度看，scull可以用来暂时消耗你的系统内存，以便在低内存条件下运行测试。运行这样的测试可能会帮助你理解系统的内部工作。你可以使用命令cp /dev/zero /dev/scull0来用scull消耗所有的实际RAM，你也可以使用dd工具来选择复制到scull设备的数据量。

在scull中，每个设备都是一个指针的链表，每个指针都指向一个scull_qset结构。默认情况下，每个这样的结构最多可以通过中间指针数组引用四百万字节。发布的源代码使用一个包含1000个指针的数组，这些指针指向4000字节的区域。我们将每个内存区域称为一个量子，将数组（或其长度）称为一个量子集。scull设备及其内存区域在图3-1中显示。

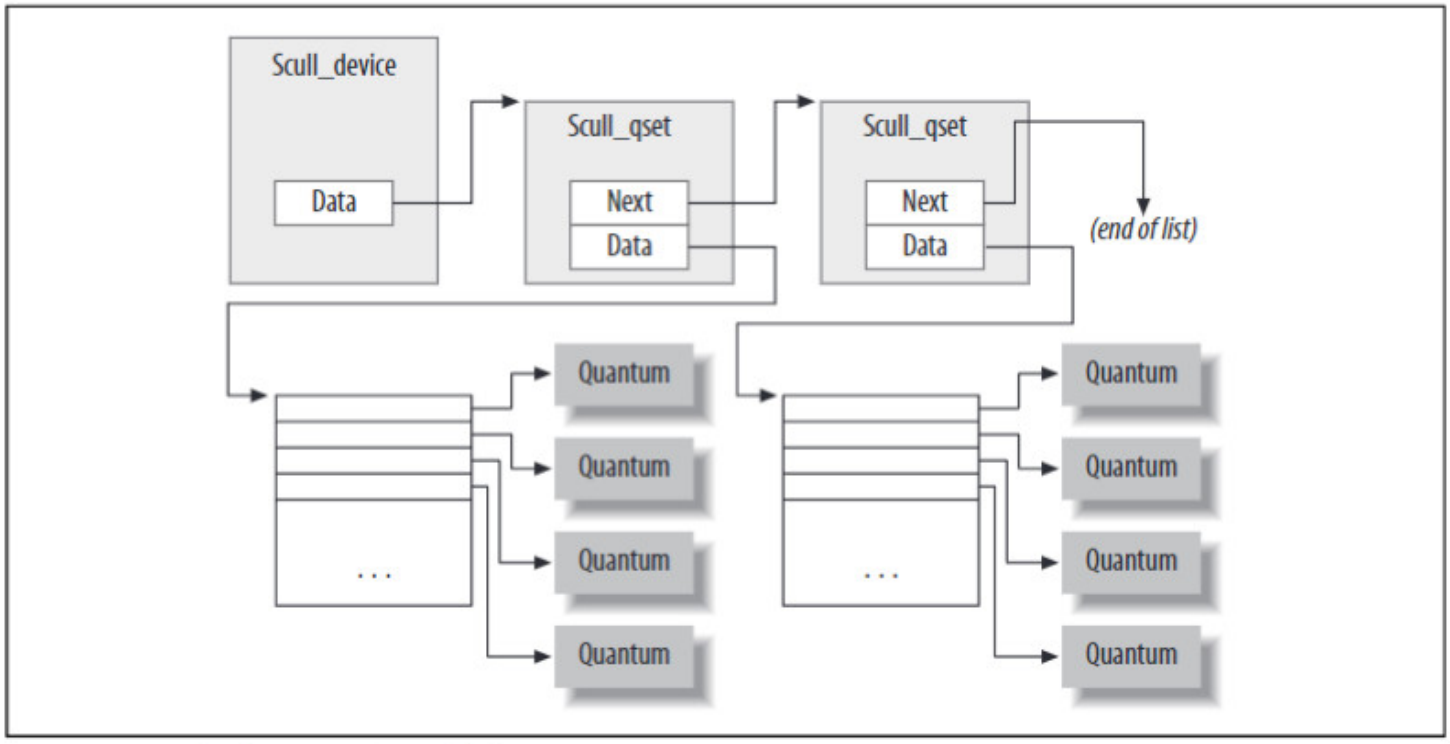

Figure 3-1. The layout of a scull device

选择的数字是这样的，即在scull中写入一个字节会消耗8000或12000字节的内存：4000字节用于量子，4000或8000字节用于量子集（取决于在目标平台上指针是用32位还是64位表示）。然而，如果你写入大量的数据，链表的开销并不太大。每四兆字节的数据只有一个列表元素，设备的最大大小受计算机的内存大小限制。

选择适当的量子和量子集的值是一个策略问题，而不是机制问题，最优的大小取决于设备的使用方式。因此，scull驱动程序不应强制使用任何特定的量子和量子集大小。在scull中，用

户可以通过几种方式改变这些值：在编译时改变scull.h中的宏SCULL_QUANTUM和SCULL_QSET，或者在模块加载时设置整数值scull_quantum和scull_qset，或者在运行时使用ioctl改变当前和默认值。

使用宏和整数值来允许在编译时和加载时配置，这让人想起了主要号码是如何选择的。我们对驱动程序中任何与策略相关或任意的值都使用这种技术。

剩下的唯一问题是如何选择默认的数字。在这个特定的情况下，问题是找到最好的平衡，即半填充的量子和量子集导致的内存浪费，以及如果量子和集合小，就会出现分配、释放和指针链接的开销。此外，还应考虑kmalloc的内部设计。（我们现在不会追究这一点；kmalloc的内部结构将在第8章中探讨。）默认数字的选择来自于这样的假设，即在测试scull时，可能会写入大量的数据，尽管设备的正常使用最可能只传输几千字节的数据。

- "量子"是指分配给scull设备的每个内存区域的大小。在scull的实现中，每个量子默认是4000字节。这意味着，即使你只写入一个字节的数据，scull设备也会分配一个4000字节的量子来存储这个字节

- "量子集"是指指向量子的指针数组的长度。在scull的实现中，每个量子集默认包含1000个指针，这些指针指向4000字节的区域。每个指针都需要占用一定的内存空间，这个空间的大小取决于目标平台上指针的表示方式。如果指针是32位的，那么每个指针就需要4字节的内存空间；如果指针是64位的，那么每个指针就需要8字节的内存空间。因此，一个量子集会消耗4000字节（如果指针是32位的）或8000字节（如果指针是64位的）的内存。

我们已经看到了代表我们设备的内部结构scull_dev。该结构的量子和qset字段分别保存设备的量子和量子集大小。然而，实际的数据是由一个不同的结构跟踪的，我们称之为struct scull_qset：

```c
struct scull_qset {
    void **data;
    struct scull_qset *next;
};
```

下一个代码片段实际展示了如何使用struct scull_dev和struct scull_qset来保存数据。函数scull_trim负责释放整个数据区域，当文件被打开用于写入时，scull_open会调用它。它只是遍历列表，并释放找到的任何量子和量子集。

```c
int scull_trim(struct scull_dev *dev)

{

    struct scull_qset *next, *dptr;

    int qset = dev->qset;   /* "dev" is not-null */

    int i;

    for (dptr = dev->data; dptr; dptr = next) { /* all the
list items */

        if (dptr->data) {

            for (i = 0; i < qset; i++)

                kfree(dptr->data[i]);

            kfree(dptr->data);

            dptr->data = NULL;

        }

        next = dptr->next;

        kfree(dptr);

    }

    dev->size = 0;

    dev->quantum = scull_quantum;

    dev->qset = scull_qset;

    dev->data = NULL;

    return 0;
```

```
}
```

scull_trim也在模块清理函数中使用，以将scull使用的内存返回给系统。

3.7. 读和写

read和write方法都执行相似的任务，即从应用程序代码复制数据和向应用程序代码复制数据。因此，它们的原型非常相似，值得同时介绍：

```c
ssize_t read(struct file *filp, char __user *buff,
    size_t count, loff_t *offp);
ssize_t write(struct file *filp, const char __user *buff,
    size_t count, loff_t *offp);
```

对于这两种方法，filp是文件指针，count是请求的数据传输的大小。buff参数指向持有要写入的数据的用户缓冲区，或者新读取的数据应放置的空缓冲区。最后，offp是一个指向“长偏移类型”对象的指针，该对象指示用户正在访问的文件位置。返回值是一个“有符号大小类型”；其用途将在后面讨论。

让我们重复一下，read和write方法的buff参数是一个用户空间指针。因此，它不能被内核代码直接解引用。这个限制有几个原因：

- 根据你的驱动程序运行的架构以及内核如何配置，用户空间指针在内核模式下可能根本无效。可能没有该地址的映射，或者它可能指向其他的，随机的数据。
- 即使指针在内核空间中意味着相同的事情，用户空间内存是分页的，当系统调用被执行时，相关的内存可能不在RAM中。试图直接引用用户空间的内存可能会产生页面错误，这是内核代码不允许做的。结果会是一个“oops”，这将导致执行系统调用的进程死亡。
- 问题中的指针是由用户程序提供的，可能是有bug的或恶意的。如果你的驱动程序盲目地解引用一个用户提供的指针，它提供了一个开放的门，允许用户空间程序访问或覆盖系统中的任何地方的内存。如果你不希望负责破坏你的用户系统的安全，你不能直接解引用一个用户空间的指针。

显然，你的驱动程序必须能够访问用户空间缓冲区才能完成其工作。然而，为了安全，这种访问必须始终由特殊的，内核提供的函数执行。我们在这里介绍一些这样的函数（它们在<asm/uaccess.h>中定义），其余的在第6章的“使用ioctl参数”部分中介绍；它们使用一些特殊的，依赖于架构的魔法来确保内核和用户空间之间的数据传输以安全和正确的方式进行。

scull中的read和write的代码需要将整个数据段复制到用户地址空间或从用户地址空间复制出来。这种能力由以下内核函数提供，它们复制一个任意的字节数组，并位于大多数read和write实现的核心：

```c
unsigned long copy_to_user(void __user *to,
                           const void *from,
                           unsigned long count);
unsigned long copy_from_user(void *to,
                             const void __user *from,
                             unsigned long count);
```

虽然这些函数的行为就像普通的memcpy函数，但在从内核代码访问用户空间时必须使用一些额外的注意事项。正在被访问的用户页面可能当前不在内存中，虚拟内存子系统可以在页面被转移到位的同时让进程睡眠。例如，当页面必须从交换空间中检索时，就会发生这种情况。对驱动程序编写者的最终结果是，任何访问用户空间的函数必须是可重入的，必须能够与其他驱动程序函数并发执行，并且，特别是，必须处于可以合法睡眠的位置。我们将在第5章中回到这个主题。

这两个函数的作用不仅限于复制数据到用户空间和从用户空间复制数据：它们还检查用户空间指针是否有效。如果指针无效，不执行复制；如果在复制过程中遇到无效地址，只复制部分数据。在这两种情况下，返回值是还需要复制的内存量。scull代码查找这个错误返回，如果它不是0，就向用户返回-EFAULT。

用户空间访问和无效用户空间指针的主题有些高级，将在第6章中讨论。然而，值得注意的是，如果你不需要检查用户空间指针，你可以调用copy_to_user和copy_from_user。例如，如果你知道你已经检查过参数，这是有用的。然而，要小心；如果你实际上没有检查你传递给这些函数的用户空间指针，那么你可以创建内核崩溃和/或安全漏洞。

就实际的设备方法而言，read方法的任务是将数据从设备复制到用户空间（使用copy_to_user），而write方法必须将数据从用户空间复制到设备（使用copy_from_user）。每个read或write系统调用请求传输特定数量的字节，但驱动程序可以自由地传输更少的数据——读取和写入的确切规则略有不同，将在本章后面描述。

无论方法传输的数据量是多少，它们通常应该更新`* offp`处的文件位置，以表示系统调用成功完成后的当前文件位置。当适当的时候，内核会将文件位置的变化传播回文件结构。然而，pread和pwrite系统调用有不同的语义；它们从给定的文件偏移操作，并不改变任何其他系统调用看到的文件位置。这些调用传入一个指向用户提供的位置的指针，并丢弃你的驱动程序做出的更改。

图3-2表示了一个典型的read实现如何使用它的参数。

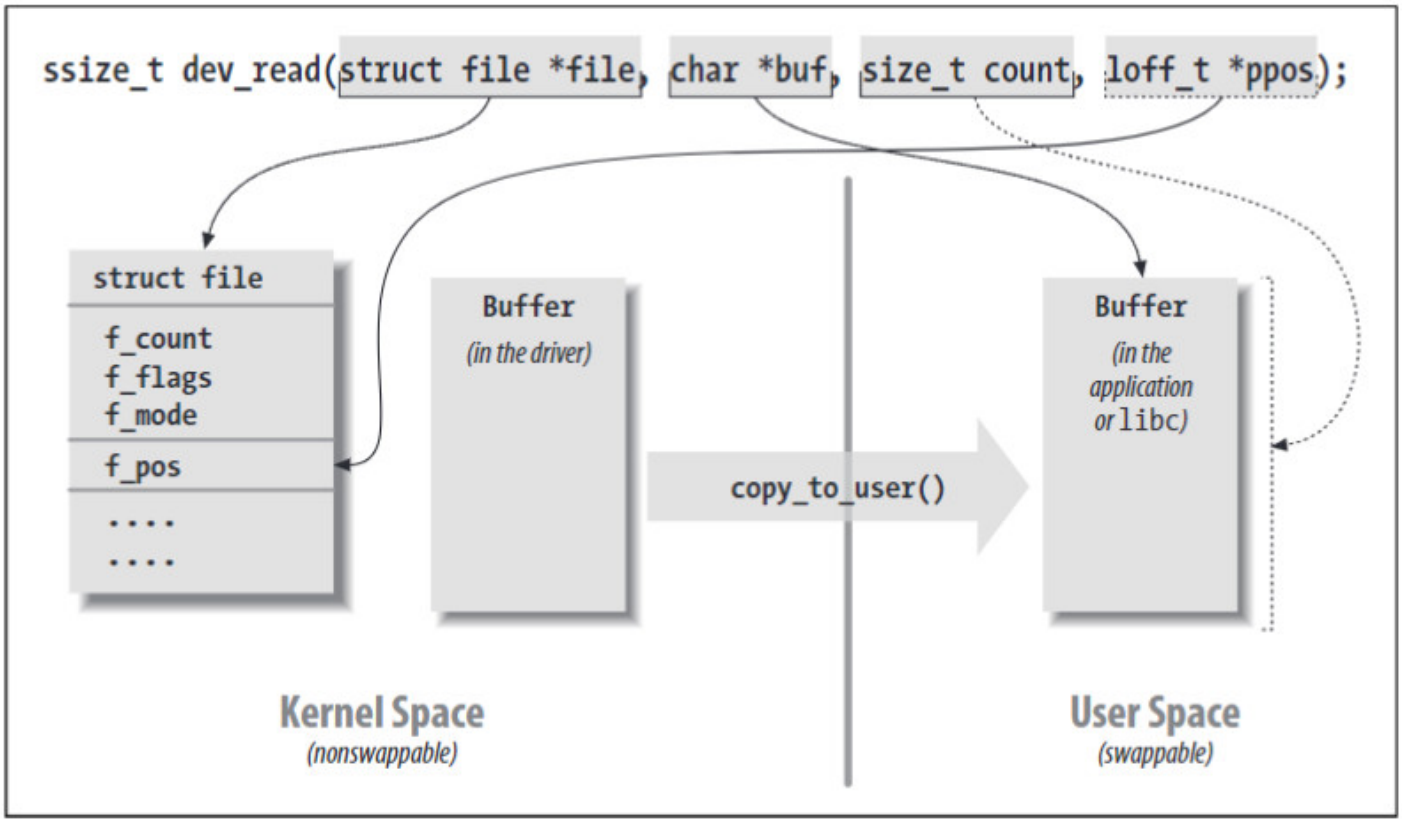

Figure 3-2. The arguments to read

如果发生错误，read和write方法都会返回一个负值。相反，大于或等于0的返回值告诉调用程序已成功传输了多少字节。如果一些数据正确地传输了，然后发生了错误，返回值必须是成功传输的字节的计数，错误直到下次调用函数时才会被报告。当然，实现这个约定需要你的驱动程序记住错误已经发生，以便它可以在将来返回错误状态。

虽然内核函数返回一个负数来表示错误，而这个数的值表示发生了什么类型的错误（如第2章中介绍的），但在用户空间运行的程序总是看到-1作为错误返回值。他们需要访问errno变量来找出发生了什么。用户空间的行为是由POSIX标准规定的，但该标准并没有对内核如何内部操作做出要求.

- `errno`是一个在C和C++中定义的全局变量，用于存储系统调用和某些库函数在错误时的错误代码。当这些函数执行成功时，`errno`通常不会被改变，但如果发生错误，

`errno` 会被设置为一个特定的错误代码。

- 这些错误代码是预定义的整数常量，每个都对应一个特定的错误情况。例如，`EPERM`（1）表示操作不被允许，`ENOENT`（2）表示没有找到文件或目录，等等。这些错误代码在 `errno.h` 头文件中定义。

- 在调用可能设置 `errno` 的函数后，程序员通常会检查 `errno` 是否改变，以确定是否发生了错误。如果 `errno` 被设置，那么可以使用 `perror` 或 `strerror` 函数来获取和打印错误消息。

3.7.1. read 方法

Linux系统中read系统调用的返回值的解释：

- 如果返回值等于传递给read系统调用的count参数，那么请求的字节数已经被传输。这是最理想的情况。
- 如果返回值是正数，但小于count，那么只有部分数据被传输。这可能因设备的不同而有不同的原因。最常见的情况是，应用程序会重试read操作。例如，如果你使用fread函数进行读取，那么库函数会重新发出系统调用，直到完成请求的数据传输。
- 如果返回值是0，那么已经到达文件的末尾（并且没有读取到数据）。
- 如果返回值是负数，那么就发生了错误。返回值会根据<linux/errno.h>来指定错误的类型。在错误发生时返回的典型值包括-EINTR（中断的系统调用）或-EFAULT（错误的地址）。

上述列表中缺少的一种情况是“没有数据，但可能会在稍后到达”。在这种情况下，read系统调用应该阻塞。我们将在第6章中处理阻塞输入。

scull代码利用了这些规则。特别是，它利用了部分读取规则。每次调用scull_read只处理一个数据量，而不实现循环来收集所有数据；这使得代码更短，更易于阅读。如果读取程序真的需要更多的数据，它会重复调用。如果使用标准I/O库（即，fread）来读取设备，应用程序甚至不会注意到数据传输的量化。

如果当前的读取位置大于设备大小，scull的read方法返回0，表示没有可用的数据（换句话说，我们已经到达了文件的末尾）。这种情况可能发生在进程A正在读取设备，而进程B打开它进行写入，从而将设备截断到长度为0。进程A突然发现自己已经超过了文件的末尾，下一次read调用返回0。

这是 read 的代码(忽略对 down_interruptible 的调用, 我们在下一章中讨论它们):

```c
//一个read方法，它从scull设备中读取数据
ssize_t scull_read(struct file *filp, char __user *buf,
size_t count, loff_t *f_pos)
{
        struct scull_dev *dev = filp->private_data;
        struct scull_qset *dptr; /* the first listitem */
        int quantum = dev->quantum, qset = dev->qset;
        int itemsize = quantum * qset; /* how many bytes in
the listitem */
        int item, s_pos, q_pos, rest;
        ssize_t retval = 0;
                //获取设备的信息和信号量。如果获取信号量时被中断，函
数返回-ERESTARTSYS
        if (down_interruptible(&dev->sem))
                return -ERESTARTSYS;
        //检查文件位置是否超过设备大小。如果超过，就直接跳到`out`标
签，释放信号量并返回。
        if (*f_pos >= dev->size)
                goto out;
        //如果读取的数据加上文件位置超过设备大小，就调整读取的字节数，
使其不超过设备大小。
        if (*f_pos + count > dev->size)
                count = dev->size - *f_pos;

        /* find listitem, qset index, and offset in the
quantum */
        //计算要读取的数据在设备中的位置，包括列表项、qset索引和量子中
的偏移量。
        item = (long)*f_pos / itemsize;
        rest = (long)*f_pos % itemsize;
        s_pos = rest / quantum;
        q_pos = rest % quantum;

        /* follow the list up to the right position (defined
elsewhere) */
        //跟踪到正确的位置。如果找不到数据，就跳到`out`标签，释放信号
量并返回。
        dptr = scull_follow(dev, item);
        if (dptr == NULL || !dptr->data || ! dptr-
>data[s_pos])
```

```
                goto out; /* don't fill holes */
                //如果要读取的字节数超过量子的剩余部分，就调整读取的
字节数，使其不超过量子的剩余部分。
        /* read only up to the end of this quantum */
        if (count > quantum - q_pos)
                count = quantum - q_pos;
                //将数据从内核空间复制到用户空间。如果复制失败，就设
置返回值为-EFAULT，然后跳到out标签。
        if (copy_to_user(buf, dptr->data[s_pos] + q_pos,
count))
        {
                retval = -EFAULT;
                goto out;

        }
        //更新文件位置，并设置返回值为已读取的字节数
        *f_pos += count;
        retval = count;
//在out标签处，释放信号量并返回
out:
        up(&dev->sem);
        return retval;
}
```

- `filp`：一个指向`struct file`的指针，表示打开的文件描述符。
- `buf`：一个用户空间指针，用于存储从设备读取的数据。
- `count`：要读取的字节数。
- `f_pos`：一个指向文件位置的指针，表示从哪里开始读取。
- 列表项（listitem）：scull设备将数据存储在一个链表中，链表中的每个元素被称为一个列表项。每个列表项都包含一个指向一个qset的指针。
- qset：qset是一个指针数组，每个指针都指向一个量子。qset的大小是可配置的，由dev→qset决定。
- 量子（quantum）：量子是实际存储数据的内存区域。每个量子的大小也是可配置的，由dev→quantum决定。

在读取数据时，我们需要确定数据在这个结构中的位置。这就涉及到以下三个步骤：

1. 确定数据在哪个列表项中。这是通过将文件位置除以一个列表项中的总字节数（即量子大小乘以qset大小）得到的。
2. 确定数据在列表项的哪个qset中。这是通过将文件位置对一个列表项中的总字节数取余，然后再除以量子大小得到的。
3. 确定数据在量子的哪个位置。这是通过将文件位置对一个列表项中的总字节数取余，然后再对量子大小取余得到的。

3.7.2. write 方法

关于Linux系统中write系统调用的返回值的解释：

- 如果返回值等于count，那么请求的字节数已经被传输。
- 如果返回值是正数，但小于count，那么只有部分数据被传输。程序很可能会重试写入剩余的数据。
- 如果返回值是0，那么没有写入任何数据。这个结果不是错误，没有理由返回错误代码。再次，标准库会重试调用write。我们将在第6章中，引入阻塞写入时，详细讨论这种情况的具体含义。
- 如果返回值是负数，那么就发生了错误；和read一样，有效的错误值是在<linux/errno.h>中定义的。

不幸的是，可能仍然有一些行为不当的程序，在执行部分传输时发出错误消息并中止。这是因为一些程序员习惯于看到write调用要么失败，要么完全成功，这实际上是大多数时候发生的情况，设备也应该支持这一点。scull实现中的这个限制可以被修复，但我们并不想过度复杂化代码。

scull的write代码每次处理一个量子，就像read方法一样。

```c
ssize_t scull_write(struct file *filp, const char __user
*buf, size_t count, loff_t *f_pos)
{
        struct scull_dev *dev = filp->private_data;
        struct scull_qset *dptr;
        int quantum = dev->quantum, qset = dev->qset;
        int itemsize = quantum * qset;
        int item, s_pos, q_pos, rest;
        ssize_t retval = -ENOMEM; /* value used in "goto
out" statements */
        if (down_interruptible(&dev->sem))
                return -ERESTARTSYS;

        /* find listitem, qset index and offset in the
quantum */
        item = (long)*f_pos / itemsize;
        rest = (long)*f_pos % itemsize;
        s_pos = rest / quantum;
        q_pos = rest % quantum;
        /* follow the list up to the right position */
        dptr = scull_follow(dev, item);
        if (dptr == NULL)
                goto out;
        if (!dptr->data)
        {
                dptr->data = kmalloc(qset * sizeof(char *),
GFP_KERNEL);
                if (!dptr->data)
                        goto out;
                memset(dptr->data, 0, qset * sizeof(char
*));
        }
        if (!dptr->data[s_pos])
        {
                dptr->data[s_pos] = kmalloc(quantum,
GFP_KERNEL);
                if (!dptr->data[s_pos])

                        goto out;
        }
```

```
        /* write only up to the end of this quantum */
        if (count > quantum - q_pos)

                count = quantum - q_pos;
        if (copy_from_user(dptr->data[s_pos]+q_pos, buf,
count))
        {
                retval = -EFAULT;
                goto out;

        }
        *f_pos += count;
        retval = count;

        /* update the size */
        if (dev->size < *f_pos)
                dev->size = *f_pos;

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

}
```

函数的参数解释如下：

- `filp`：一个指向`struct file`的指针，表示打开的文件描述符。
- `buf`：一个用户空间指针，包含要写入设备的数据。
- `count`：要写入的字节数。
- `f_pos`：一个指向文件位置的指针，表示从哪里开始写入。

函数的主要步骤如下：

1. 首先，获取设备的信息和信号量。如果获取信号量时被中断，函数返回-ERESTARTSYS。
2. 计算要写入的数据在设备中的位置，包括列表项、qset索引和量子中的偏移量。
3. 跟踪到正确的位置。如果找不到列表项，就跳到`out`标签，释放信号量并返回。
4. 如果列表项的数据指针为空，就为其分配内存，并初始化为0。如果分配内存失败，就跳到`out`标签。
5. 如果qset的数据指针为空，就为其分配内存。如果分配内存失败，就跳到`out`标签。

6. 如果要写入的字节数超过量子的剩余部分，就调整写入的字节数，使其不超过量子的剩余部分。
7. 将数据从用户空间复制到内核空间。如果复制失败，就设置返回值为-EFAULT，然后跳到`out`标签。
8. 更新文件位置，并设置返回值为已写入的字节数。
9. 如果设备的大小小于文件位置，就更新设备的大小。
10. 在`out`标签处，释放信号量并返回。

这个函数的主要目的是将数据从用户空间写入到设备中。在写入数据时，它会处理各种边界条件，例如找不到列表项、分配内存失败、复制数据失败等。

3.7.3. readv 和 writev

Unix系统长期以来支持两个名为readv和writev的系统调用。这两个“向量”版本的read和write接受一个结构体数组，每个结构体都包含一个指向缓冲区的指针和一个长度值。readv调用会按顺序读取每个缓冲区指定的数量。而writev则会将每个缓冲区的内容聚集在一起，并作为一个单独的写操作输出。

如果你的驱动程序没有提供处理向量操作的方法，那么readv和writev会通过多次调用你的read和write方法来实现。然而，在许多情况下，直接实现readv和writev会更有效率。

向量操作的原型如下：

```c
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);
```

这里，filp和ppos参数与read和write的相同。iovec结构体在<linux/uio.h>中定义，如下所示：

```c
struct iovec
{
    void _ _user *iov_base;
    __kernel_size_t iov_len;
};
```

每个iovec描述了要传输的数据块；它从用户空间的iov_base开始，长度为iov_len字节。count参数告诉方法有多少个iovec结构体。这些结构体由应用程序创建，但内核会在调用驱动程序之前将它们复制到内核空间。

向量操作的最简单实现可能是一个直接的循环，只是将每个iovec的地址和长度传递给驱动程序的read或write函数。然而，通常情况下，为了实现有效和正确的行为，驱动程序需要做一些更智能的事情。例如，对磁带驱动程序的writev应该将所有iovec结构体的内容作为磁带上的一个单独记录写入。

然而，许多驱动程序并不能从自己实现这些方法中获得任何好处。因此，scull省略了它们。内核用read和write模拟它们，最终结果是一样的。

3.8使用新设备

一旦你掌握了刚刚描述的四种方法，驱动程序就可以编译和测试了；它会保留你写入的任何数据，直到你用新数据覆盖它。该设备就像一个数据缓冲区，其长度仅受可用的实际RAM的限制。你可以尝试使用cp，dd和输入/输出重定向来测试驱动程序。

你可以使用free命令来查看根据写入scull的数据量，可用内存的大小是如何缩小和扩大的。

为了更自信地一次读写一个量子，你可以在驱动程序的适当位置添加一个printk，并观察当应用程序读取或写入大量数据时会发生什么。或者，使用strace工具来监视程序发出的系统调用以及它们的返回值。追踪cp或ls -l > /dev/scull0显示量子化的读取和写入。监视（和调试）技术将在第4章中详细介绍.

3.9 参考

```c
#include <linux/types.h>
//dev_t是内核中用来表示设备号的类型。

int MAJOR(dev_t dev);

int MINOR(dev_t dev);

//这些宏从设备号中提取主要和次要的数字。

dev_t MKDEV(unsigned int major, unsigned int minor);

//这个宏从主要和次要的数字中构建一个dev_t数据项。

#include <linux/fs.h>

//“filesystem”头文件是编写设备驱动程序所需的头文件。许多重要的函数和数据结构在这里声明。

int register_chrdev_region(dev_t first, unsigned int count,
char *name);

int alloc_chrdev_region(dev_t *dev, unsigned int firstminor,
unsigned int count, char *name);

void unregister_chrdev_region(dev_t first, unsigned int
count);

//这些函数允许驱动程序分配和释放设备号的范围。当预先知道所需的主要数字时，应使用register_chrdev_region；对于动态分配，应使用alloc_chrdev_region。

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

//这是旧的（pre-2.6）字符设备注册例程。它在2.6内核中被模拟，但不应用于新的代码。如果主要数字不是0，那么它将不变地使用；否则，将为此设备分配一个动态数字。

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

```
//这个函数撤销了用register_chrdev做的注册。主要和名称字符串必须包含用于注册驱动程序的相同值。

struct file_operations;

struct file;

struct inode;

//这三个重要的数据结构被大多数设备驱动程序使用。file_operations结构体保存了字符驱动程序的方法；struct file表示一个打开的文件，struct inode表示磁盘上的文件。

#include <linux/cdev.h>

struct cdev *cdev_alloc(void);

void cdev_init(struct cdev *dev, struct file_operations *fops);

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

void cdev_del(struct cdev *dev);

//这些函数用于管理cdev结构，它在内核中表示字符设备。

#include <linux/kernel.h>

container_of(pointer, type, field);

//这是一个方便的宏，可以用来从一个指向其中包含的其他结构的指针获取一个指向结构的指针。

#include <asm/uaccess.h>

//这个包含文件声明了内核代码用来将数据移动到用户空间和从用户空间移动数据的函数。

unsigned long copy_from_user (void *to, const void *from, unsigned long
```

```
  count);

unsigned long copy_to_user (void *to, const void *from,
unsigned long count);

//这些函数用于在用户空间和内核空间之间复制数据。
```