第一章 驱动程序简介

1.1 前言

免费操作系统的众多优点之一，以 Linux 为例，就是它们的内部对所有人开放。操作系统曾经是一个黑暗而神秘的领域，其代码只限于少数程序员，现在任何具备必要技能的人都可以轻松地检查、理解和修改。Linux 有助于民主化操作系统。然而，Linux 内核仍然是一个大型且复杂的代码体，希望成为内核黑客的人需要一个入口，他们可以在不被复杂性压倒的情况下接近代码。通常，设备驱动程序提供了这个入口。

设备驱动程序在 Linux 内核中扮演着特殊的角色。它们是独特的“黑盒”，使特定的硬件响应一个定义良好的内部编程接口；它们完全隐藏了设备工作的细节。用户活动是通过一组标准化的调用进行的，这些调用独立于特定的驱动程序；将这些调用映射到作用于真实硬件的设备特定操作，然后就是设备驱动程序的角色。这种编程接口使得驱动程序可以从内核的其余部分独立构建，并在需要时在运行时“插入”。这种模块化使得 Linux 驱动程序易于编写，以至于现在有数百种驱动程序可用。

对于编写 Linux 设备驱动程序感兴趣的原因有很多。新硬件的出现（和淘汰！）的速度就足以保证驱动程序编写者在可预见的未来都会忙碌。个人可能需要了解驱动程序，以便获得对他们感兴趣的特定设备的访问。硬件供应商，通过为他们的产品提供 Linux 驱动程序，可以将庞大且不断增长的 Linux 用户基础添加到他们的潜在市场。而且，Linux 系统的开源性质意味着，如果驱动程序编写者愿意，驱动程序的源代码可以快速传播到数百万用户。

这本书教你如何编写自己的驱动程序，以及如何在内核的相关部分进行探索。我们采取了一种设备无关的方法；尽可能地，不依赖于任何特定设备，就呈现编程技术和接口。每个驱动程序都是不同的；作为驱动程序编写者，你需要很好地理解你的特定设备。但是，所有驱动程序的大多数原则和基本技术都是相同的。这本书不能教你关于你的设备的知识，但它给你提供了让你的设备工作所需的背景知识。

当你学习编写驱动程序时，你会对 Linux 内核有很多了解；这可能会帮助你理解你的机器是如何工作的，以及为什么事情并不总是像你期望的那样快，或者不完全按照你想要的方式进行。我们逐渐引入新的想法，从非常简单的驱动程序开始，并在其基础上建立；每一个新的概念都伴随着不需要特殊硬件就可以测试的示例代码。

这一章实际上并没有涉及编写代码。然而，我们介绍了一些关于 Linux 内核的背景概念，当我们开始编程时，你会很高兴你知道这些。

1.2 驱动程序的role

作为一名程序员，你可以自己选择你的驱动程序，并在所需的编程时间和结果的灵活性之间选择一个可接受的折衷。虽然说驱动程序是“灵活的”可能看起来有些奇怪，但我们喜欢这个词，因为它强调了设备驱动程序的角色是提供机制，而不是策略。

机制和策略之间的区别是 Unix 设计背后的最好的想法之一。大多数编程问题确实可以分为两部分：“要提供什么功能”（机制mechanism)）和“如何使用这些功能”（策略policy）。如果这两个问题由程序的不同部分，甚至完全不同的程序来解决，那么软件包就更容易开发和适应特定的需求。

例如，Unix 对图形显示的管理分为 X server，它了解硬件并向用户程序提供统一的接口，以及窗口和会话管理器，它们实现特定的策略，而不需要了解硬件。人们可以在不同的硬件上使用相同的窗口管理器，不同的用户可以在同一工作站上运行不同的配置。甚至完全不同的桌面环境，如 KDE 和 GNOME，可以在同一系统上共存。另一个例子是 TCP/IP 网络的分层结构：操作系统提供了套接字抽象(socket abstraction)，它不实施关于要传输的数据的策略，而不同的服务器负责服务（及其相关的策略）。此外，像 ftpd 这样的服务器提供了文件传输机制，而用户可以使用他们喜欢的任何客户端；存在命令行和图形客户端，任何人都可以编写新的用户界面来传输文件。

在驱动程序方面，同样适用机制和策略的分离原则。软盘驱动程序是无策略的——它的角色仅仅是将磁盘显示为连续的数据块数组。系统的更高级别提供策略，例如谁可以访问软盘驱动程序，是否直接访问驱动程序或通过文件系统访问，以及用户是否可以在驱动程序上挂载文件系统。由于不同的环境通常需要以不同的方式使用硬件，所以尽可能无策略是很重要的。

编写驱动程序时，程序员应特别注意这个基本概念：编写内核代码来访问硬件，但不要强制用户采取特定的策略，因为不同的用户有不同的需求。驱动程序应处理使硬件可用的问题，将如何使用硬件的所有问题留给应用程序。因此，如果驱动程序提供对硬件能力的访问而不添加约束，那么它就是灵活的。然而，有时必须做出一些策略决策。例如，一个digital I/O驱动程序可能只提供对硬件的byte-wide访问，以避免处理单个位所需的额外代码。

你也可以从另一个角度看你的驱动程序：它是位于应用程序和实际设备之间的软件层。驱动程序的”特权角色(pvileged role)”允许驱动程序编写者精确地选择设备应该如何出现：不同的驱动程序可以提供不同的能力，即使是对于同一设备。实际的驱动程序设计应该在许多不同的考虑之间取得平衡。例如，单个设备可能被不同的程序同时使用，驱动程序编写者有完全的自由来决定如何处理并发。你可以独立于硬件能力在设备上实现内存映射，或者你可以提供一个用户库来帮助应用程序编写者在可用的原语之上实现新的策略，等等。一个主要的考虑因素是在希望向用户提供尽可能多的选项和你编写驱动程序的时间之间的权衡，以及保持简单以防止错误潜入的需要。

无策略的驱动程序具有一些典型的特性。这些包括支持同步和异步操作，能够被多次打开，能够利用硬件的全部能力，以及缺乏软件层来“简化事情”或提供与策略相关的操作。这种类

型的驱动程序不仅对最终用户能更好的工作，而且事实证明，编写和维护它们也更容易。实际上，无策略是软件设计师的一个常见目标。

许多设备驱动程序实际上是与用户程序一起发布的，以帮助配置和访问目标设备。这些程序可以从简单的实用程序到完整的图形应用程序。例如，tunelp 程序，它调整并行端口打印机驱动程序的操作方式，以及作为 PCMCIA 驱动程序包的一部分的图形化 cardctl 实用程序。通常也会提供一个客户端库，它提供不需要作为驱动程序本身的一部分来实现的功能。

- tunelp 是一个程序，它的功能是调整并行端口打印机驱动程序的操作方式。并行端口打印机驱动程序是用来控制并行端口打印机的软件，tunelp 可以修改这个驱动程序的一些设置，以改变打印机的行为。例如，它可能会改变打印机的数据传输速度或其他相关参数。

- cardctl 是一个图形化的实用程序，它是 PCMCIA 驱动程序包的一部分。PCMCIA 是一种用于笔记本电脑的扩展卡标准，例如网络卡或调制解调器。cardctl 可以用来管理这些卡的设置，例如插入或移除卡，或者改变卡的配置。因为它是图形化的，所以用户可以通过直观的界面来进行这些操作，而不需要使用命令行。

这本书的主题是内核，所以我们尽量不涉及策略问题，也不涉及应用程序或支持库。有时我们会讨论不同的策略以及如何支持它们，但我们不会详细讨论使用设备的程序或它们执行的策略。然而，你应该明白，用户程序是软件包的一个重要部分，即使是无策略的包也会配有配置文件，这些文件会对底层机制应用默认行为.

1.3 划分内核

在 Unix 系统中，有多个并发的进程负责不同的任务。每个进程都会请求系统资源，无论是计算能力、内存、网络连接，还是其他资源。内核是负责处理所有这些请求的大块可执行代码。虽然内核的不同任务之间的区别并不总是清晰明确，但内核的角色可以分为以下几部分：

1. 进程管理 内核负责创建和销毁进程，并处理它们与外界的连接（输入和输出）。不同进程之间的通信（通过信号、管道或进程间通信原语）是整个系统功能的基础，也由内核处理。此外，控制进程如何共享 CPU 的调度器也是进程管理的一部分。更一般地说，内核的进程管理活动在单个 CPU 或少数几个 CPU 上实现了多个进程的抽象。

2. 内存管理 计算机的内存是主要的资源，处理它的策略对系统性能至关重要。内核在有限的可用资源之上为所有进程构建了一个虚拟地址空间。内核的不同部分通过一组函数调用与内存管理子系统交互，这些函数调用从简单的 malloc/free 到更复杂的功能。

3. 文件系统 Unix 系统大量依赖文件系统概念；在 Unix 中，几乎所有的东西都可以被当作文件处理。内核在无结构的硬件之上构建了一个结构化的文件系统，而且整个系统大

量使用了结果文件抽象。此外，Linux 支持多种文件系统类型，也就是在物理介质上组织数据的不同方式。例如，磁盘可能被格式化为 Linux 标准的 ext3 文件系统，常用的 FAT 文件系统或其他几种文件系统。

4. 设备控制 几乎每个系统操作最终都会映射到一个物理设备。除了处理器、内存和少数其他实体外，所有的设备控制操作都是由针对被操作设备的特定代码执行的。这段代码被称为设备驱动程序。内核必须内置每个系统上存在的每个外设的设备驱动程序，从硬盘到键盘和磁带驱动器。这个内核功能的方面是我们在这本书中主要关注的。

5. 网络 操作系统必须管理网络，因为大多数网络操作并不特定于一个进程：传入的数据包是异步事件。必须收集、识别和分派数据包，然后进程才能处理它们。系统负责在程序和网络接口之间传送数据包，并且必须根据它们的网络活动控制程序的执行。此外，所有的路由和地址解析问题都在内核内部实现。

1.4可加载模块

Linux的一个好特性是能够在运行时扩展内核提供的功能集(set of features)。这意味着你可以在系统运行的同时向内核添加功能（也可以移除功能）。

每个可以在运行时添加到内核的代码片段都被称为模块。Linux内核为很多不同类型（或类别）(types (or classes))的模块提供支持，包括但不限于设备驱动程序。每个模块都由对象代码组成（没有链接成一个完整的可执行文件），可以通过insmod程序动态链接到运行的内核，并可以通过rmmod程序解除链接。

图1-1标识了负责特定任务的不同类别的模块——根据模块提供的功能，可以说模块属于特定的类别。图1-1中模块的位置覆盖了最重要的类别，但远未完全，因为Linux中越来越多的功能正在被模块化。

- 在 Linux 系统中，我们可以在系统运行的时候，向内核添加或移除一些功能，这些功能就是通过一种叫做“模块”的代码片段来实现的。你可以把模块想象成一个插件，它可以在需要的时候插入到系统中，也可以在不需要的时候从系统中移除。

- 这些模块有很多种类，包括设备驱动程序等。设备驱动程序就是一种特殊的模块，它让内核能够控制和管理硬件设备。

- 每个模块都是由一些特殊的代码（我们称之为对象代码）组成的，这些代码并没有被链接成一个完整的可执行文件，而是可以在运行时动态地链接到内核中。我们可以使用一个叫做 insmod 的程序来把模块链接到内核，也可以使用一个叫做 rmmod 的程序来把模块从内核中移除。

- 在图1-1中，我们可以看到负责不同任务的各种模块。这些模块根据它们提供的功能被分到了不同的类别中。这个图表并没有包含所有的模块类别，因为 Linux 系统中有越来越

多的功能被模块化，也就是被划分成可以动态添加和移除的模块。

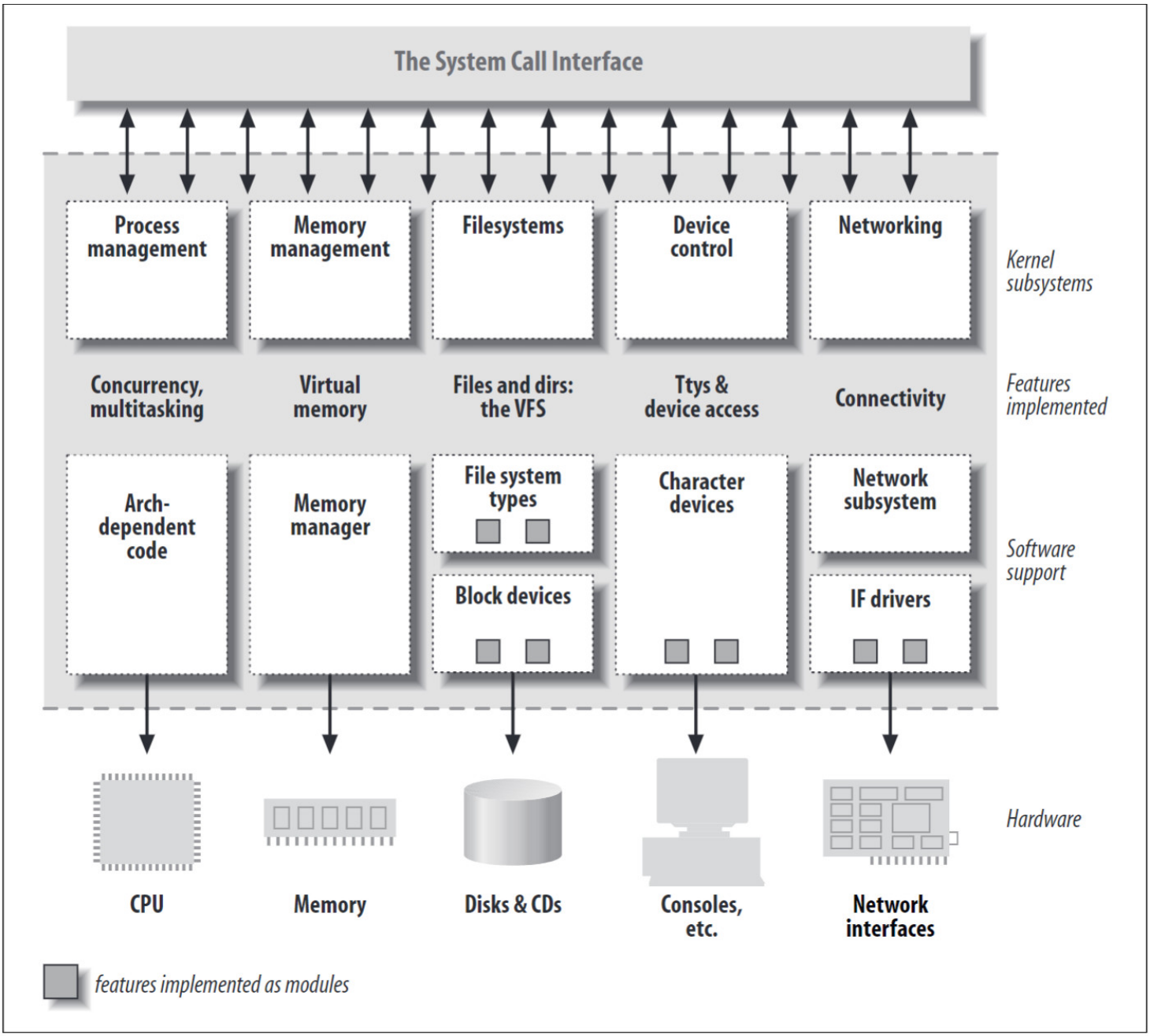

Figure 1-1. A split view of the kernel

1.5设备和模块的分类

Linux 系统对设备的看法主要区分了三种基本的设备类型。每个模块通常实现这三种类型中的一种，因此可以被分类为字符模块、块模块或网络模块(a char module, a block module, or a network module)。这种将模块划分为不同类型或类别的方式并不是固定不变的；程序员可以选择在一个代码块中构建大型模块来实现不同的驱动程序。然而，优秀的程序员通常会为他们实现的每一个新功能创建一个不同的模块，因为分解是可扩展性和可扩展性的关键元素。

字符设备

字符设备是一种可以像字节流（如文件）一样被访问的设备；字符驱动程序负责实现这种行为。这样的驱动程序通常至少实现了打开、关闭、读取和写入的系统调用。文本控制台（/dev/console）和串行端口（/dev/ttyS0等）就是字符设备的例子，因为它们可以很好地

用流抽象来表示。字符设备通过文件系统节点（如/dev/tty1和/dev/lp0）来访问。
字符设备和普通文件的唯一显著区别在于，你可以在普通文件中来回移动，而大多数字符设备只是数据通道，你只能顺序访问它们。然而，也存在一些看起来像数据区域的字符设备，你可以在它们中来回移动；例如，这通常适用于帧捕获器，应用程序可以使用mmap或lseek访问整个获取的图像。

1. `mmap` 函数原型：

```c
void *mmap(void *addr, size_t length, int prot, int flags,
int fd, off_t offset);
```

参数解释：

- `addr`：指定映射区的起始地址，通常设为 NULL，代表让系统自动选择映射区的起始地址。
- `length`：映射区的长度。以字节为单位。
- `prot`：映射区的保护方式。可以为 `PROT_EXEC`、`PROT_READ`、`PROT_WRITE`、`PROT_NONE`。
- `flags`：影响映射区的各种特性。最常用的是 `MAP_SHARED`（对映射区的写入数据会复制回文件内）和 `MAP_PRIVATE`（对映射区的写入操作会产生一个映射文件的复制，即私人的“写入时复制”）。
- `fd`：要映射的文件的文件描述符。
- `offset`：文件映射的偏移量，通常设为0，表示从文件最前方开始对应，offset必须是系统页的整数倍。

2. `lseek` 函数原型：

```c
off_t lseek(int fd, off_t offset, int whence);
```

参数解释：

- `fd`：文件描述符。
- `offset`：偏移量，根据 `whence` 参数的值来解释。
- `whence`：决定了要如何解释 `offset`。可能的值为 `SEEK_SET`（将文件的读写位置设定为 `offset` 字节）、`SEEK_CUR`（将文件的读写位置设定为当前位置加 `offset` 字节）、`SEEK_END`（将文件的读写位置设定为文件长度加 `offset` 字节）。

这两个函数的主要作用：

1. `mmap`：这是一个用于在内存中映射文件的系统调用。它允许程序将一个文件或者其它对象映射到进程的地址空间，实现文件磁盘地址和进程虚拟地址空间中一段内存的一一对应关系。这样，程序就可以像访问普通内存一样对文件进行访问，无需再进行常规的读写操作。
2. `lseek`：这是一个改变文件读写位置的系统调用。它可以改变文件的当前读写位置（也就是文件指针）。这样，当你下一次读取或写入文件时，操作将从新的位置开始。这使得程序可以在文件的任何位置开始读写，而不仅仅是从头开始或者从当前位置继续。

`块设备`

块设备 像字符设备一样，块设备通过 /dev 目录中的文件系统节点进行访问。块设备是可以承载文件系统的设备（例如，磁盘）。在大多数 Unix 系统中，块设备只能处理传输一个或多个完整块的 I/O 操作，这些块通常是 512 字节（或更大的 2 的幂）长度。然而，Linux 允许应用程序像读写字符设备一样读写块设备——它允许一次传输任意数量的字节。因此，块设备和字符设备的区别仅在于内核内部数据管理的方式，以及内核/驱动程序软件接口。像字符设备一样，每个块设备都通过文件系统节点进行访问，它们之间的区别对用户来说是透明的。块驱动程序与字符驱动程序的内核接口完全不同。

`网络接口`

网络接口 任何网络交易都是通过接口进行的，也就是说，一个能够与其他主机交换数据的设备。通常，接口是一个硬件设备，但也可能是一个纯软件设备，比如环回接口。网络接口负责发送和接收数据包，由内核的网络子系统驱动，而不知道单个交易如何映射到实际传输的数据包。许多网络连接（特别是使用 TCP 的连接）是面向流的，但网络设备通常是围绕数据包的传输和接收设计的。网络驱动程序对单个连接一无所知；它只处理数据包。
由于不是面向流的设备，网络接口不容易映射到文件系统中的节点，比如 /dev/tty1。Unix 提供接口访问的方式仍然是为它们分配一个唯一的名字（比如 eth0），但是这个名字在文件系统中没有对应的条目。内核与网络设备驱动程序之间的通信与字符和块驱动程序使用的通信完全不同。内核调用的是与数据包传输相关的函数，而不是读和写。

- 在文件系统中，节点（Node）通常指的是文件或目录。在 Unix 和类 Unix 系统（如 Linux）中，一切都被视为文件，包括硬件设备。这些设备在文件系统中以特殊文件（设备文件）的形式存在，这些特殊文件就是设备的节点。

- 例如，你可能在 /dev 目录下看到了很多设备文件，如 /dev/sda（第一个硬盘）、/dev/tty1（第一个终端）等。这些都是设备的节点，通过这些节点，我们可以像操作普通文件一样操作这些设备。

设备本身在系统中的表现形式可以是分类的依据。比如，设备可以显示为字符设备（如USB串行端口）、块设备（如USB内存卡读卡器）或网络设备（如USB以太网接口）。
驱动模块还可以根据它们支持的设备类型进行分类，比如USB模块、串行模块、SCSI模块等。这些驱动模块会与内核的额外支持函数一起工作，以支持特定类型的设备。例如，每个USB设备都由一个与USB子系统一起工作的USB模块驱动。但是设备本身在系统中会显示为

字符设备（比如USB串行端口）、块设备（比如USB内存卡读卡器）或网络设备（比如USB以太网接口）。

内核开发人员为了避免重复工作和错误，会收集类别广泛的特性，并将它们导出给驱动程序实现者。这样做可以简化和加强驱动程序的编写过程。最近，内核中还添加了其他类别的设备驱动程序，包括FireWire驱动程序和I2C驱动程序，这些都是采取了同样的处理方式。

除了设备驱动程序，内核中的其他硬件和软件功能也是模块化的。一个常见的例子是文件系统。文件系统类型决定了如何在块设备上组织信息，以表示目录和文件的树状结构。这样的实体并不是设备驱动程序，因为没有与信息存放方式明确相关的设备；相反，文件系统类型是软件驱动程序，因为它将低级数据结构映射到高级数据结构。文件系统决定了文件名可以有多长，以及目录条目中存储了关于每个文件的哪些信息。文件系统模块必须实现访问目录和文件的系统调用的最低级别，通过将文件名和路径（以及其他信息，如访问模式）映射到存储在数据块中的数据结构。这样的接口与实际的数据传输（到磁盘或其他介质）完全独立，数据传输是由块设备驱动程序完成的。

如果你想到Unix系统多么依赖于底层的文件系统，你就会意识到这样的软件概念对系统操作至关重要。解码文件系统信息的能力位于内核层次结构的最低级别，并且非常重要；即使你为新的CD-ROM编写了一个块驱动程序，如果你不能在它托管的数据上运行ls或cp，那么它就是无用的。Linux支持文件系统模块的概念，其软件接口声明了可以在文件系统inode、目录、文件和超级块上执行的不同操作。程序员实际上很少需要编写文件系统模块，因为官方内核已经包含了最重要的文件系统类型的代码。