第十六章　核心源码的物理布局

到目前为止，我们从写设备驱动程序的角度讨论了 Linux 核心。但一旦你开始研究核心，你会发现你想“全面理解它”。事实上，你可能整天在浏览源码，搜索源码树，目的只是想搞清楚核心不同部分之间的关系。

这种“沉重的搜索”是我在家里专门设一台计算机的任务之一，并且这是从源码中获取信息的一个有效的办法。然而，在坐在你喜欢的 shell 提示符之前若能得到一些知识基础将会很有帮助。本章基于版本 2.0.x，对 Linux 核心源文件提供一个快速的概览。文件布局在版本之间的改变并不大，尽管我不能保证将来会不会变。因此下面的信息对浏览核心的其它版本应该也很有用，即使它不是权威。

在本章中，每个给出的路径名都是相对于源码的根（通常是*/usr/src/linux*），而没有目录部分的文件名一般假设它居于“当前”目录----即正在讨论的哪个。头文件（当以角括弧的形式命名时----<和>）是相对于源码树的 *include* 目录给出的。我不想介绍 *Documantation* 目录，因为它的作用应该很清楚。

引导核心

看一个程序的一般方法是从执行开始的地方着手。至于 Linux，很难说执行是从那里开始的----它取决于你是如何定义“开始”的。

体系结构相关的开始点是 *start_kernel*，在 *init/main.c* 中。这个函数是从体系结构特定的代码中被调用的，但它并不返回到那里。它掌管着转动轮子，因此可以被认为是“所有函数的母亲”，计算机生命的第一次呼吸。在 *start_kernel* 之前是一片混沌。

在 *start_kernel* 被调用时，处理器已经被初始化了，保护模式（如果有）也被激活了，处理器在最高的优先级执行（通常被称为“管理员模式”），中断被关闭了。*start_kerne*l 函数负责初始化所有的核心数据结构。这个通过调用外部函数来执行子任务，因为每个设置函数都在合适的核心子系统定义。*start_kernel* 也调用 *parse_options*(也在 *init/main.c* 文件中)来对从用户或引导系统的程序处传来的命令行进行解码。

命令行（与 *memory_start*，*memory_end* 一道）用 *setup_arch* 从计算机内存中获取。*setup_arch*，如它的名字提示，是体系结构特定的代码。

init/main.c 中的代码主要由#ifdefs 组成。这是因为初始化是按步发生的，很多步可能被运行或跳过，这依赖于核心的编译时配置。命令行的解释也严重地依赖于条件，因为很多参数只有在被编译的核心含有特定的驱动程序时才有意义。

由 *start_kernel* 调用的初始化函数有两种风格。一些函数没有参数，返回 void；而另一些需要两个 unsigned long 参数，并返回另一个 unsigned long 值。其参数是 memory_start 和 memory_end 的当前值，即未分配的物理内存的边界。返回值是新的 memory_start 值（如你所已知的，核心用 unsigned long 表达内存地址）。这个技术允许子系统在物理内存的开始处分配一个持续的（和连续的）内存区域，如在第七章“把握内存”中“playing dirty”一节中提到的。这种技术的最大的缺点是它只能在引导时使用，对那些需要用于 DMA 的巨大内存区段的模块并不可用。

初始化完成后，*start_kernel* 打印出旗帜字符串，包括 Linux 版本号和编译时间，接着通过调用 *kernel_thread* 派生出(fork)一个 init 进程。

start_kernel 函数接着以任务 0（所谓的“空闲”任务）的形式继续，并调用 *cpu_idle*，它是一个调用 idle 的无限循环。在这一点，SMP(对称多处理器)的工作方式略有不同，但我不打

算讲述这个不同。idle 函数的真正行为是体系结构相关的，对源码的简单搜索可以把你带到可以研究其功能的位置。

引导之前

在前一节，我把 *start_kernel* 看作第一个核心函数。不过，你可能对这点之前发生的事情感兴趣。

在 *start_kernel* 之前运行的代码是低级的，包含汇编码，因此你可能对其细节不感兴趣。不过，我将介绍一下固件（在 PC 世界称为 BIOS）将控制交给 Linux 后计算机中发生了什么。如果你对钻研低级代码没有兴趣，你可以直接跳到“Init 进程”。下面提供了关于 Intel，Alpha，Sparc 引导代码的一些提示，因为这是我能访问的仅有的系统。（如果有人肯捐一些硬件，我将在下一版覆盖更多的平台）。

设置 X86 处理器

个人计算机是基于一个老的设计，后向兼容性一直有很高的优先级。因此，PC 固件还是以一种老方式引导操作系统。一旦引导设备被选择，它的第一个扇区被加载到内存的 0x7C00 处，然后让出控制。

刚加电的处理器处于实模式（也就是说，它象 8086）并只能寻址物理内存的前 640KB。其中一部分已经被固件管理的数据表格占用了。由于核心要比这个大，Linux 的开发者必须找到一个不一般的方法将核心影响加载到内存。结果就是 *zImage*，即核心的压缩映象，它可以被装入底端内存（但愿如此），并在进入保护模式后自解压缩到高端内存。

这样引导扇区发现它面对着五百字节的代码，和半兆字节的空闲内存。引导代码真正做的依赖于系统是如何引导的。引导扇区可以是第一个核心扇区（如果你直接从软盘上引导 *zImage*）或者 *lilo*。如果 Linux 由 *loadlin* 引导，则没有引导扇区什么事，因为在 *loadlin* 运行时，系统已经被引导了。

引导一个 bare-bones *zImage* 核心

如果被引导的系统是软盘上的核心映象，在引导扇区执行的代码是 *arch/i386/boot/boot.S*（一个实模式的汇编文件）。它将自己移到地址 0x90000，从可引导设备上加载另外几个扇区，把它们放在紧挨自己的后面（也就是 0x90200）。接着核心映象的其余部分被加载到地址 0x10000（64KB：固件数据空间之后）。

位于 0x90200 的代码是所谓的“设置”代码（*arch/i386/boot/setup.S* 和 *arch/i386/boot/video.S*），它负责各种硬件的初始化，以及对视频板子的初步检测，以便可以切换到不同的文本模式分辨率。这些任务在实模式中进行（使用 *loadlin* 时则是在 VM86 模式），因此可以使用 BIOS 调用，避免处理硬件特定的细节。

setup.S 接着把整个核心从 0x10000(64KB)移到 0x1000(4KB)；这样在核心代码之前只有一页被浪费了----这页其实并没有真的浪费；它在系统中自有它的用处。代码的这种来回复制是为了摆脱被 BIOS 强加的内存布局，还能不至于覆盖重要的数据。最后 *setup.S* 进入保护模式，跳转到 0x1000。

arch/i386/boot/compressed/head.S(用 *gas* 写成，因为我们已经在保护模式了)设置栈。接着调用 *decompress_kernel*，它把已解开的代码放在地址 0x100000(1M)并跳转到那里。

arch/i386/kernel/head.S 是被解压缩核心的头；它建立最后的处理器设置（与硬件换页有关的寄存器处理）并调用 *start_kernel*。这就是所有需要的----已经完成了。

引导一个 bare-bones *bzImage* 核心

随着越来越多的驱动程序为 Linux 核心开发出来，一个全特征的压缩核心不再能放入低端内存。例如，这种情况对安装核心就可能发生，因为它为了能适应各种配置，塞满了不同的驱动程序。因此，必须设计另一种加载方法。*bzImage* 就是大的 *zImage*，它在不能放入低端内存时也可以被加载。

有几种加载 *bzImage* 的办法，这取决于使用的引导加载程序（boot loader）。核心负责每种情况，现在我打算从原始软盘上是如何引导的。

一个 *bzImage* 核心的引导扇区不能简单地将所有的压缩数据加载到低端内存，所以它必须欺骗（如多数实模式 x86 程序所做的那样）。如果被加载的映象是大的，引导扇区象往常一样加载“设置”扇区，但在主引导循环的每次叠代都有一个“助手”例程被调用。助手例程在 *setup.S* 中定义，因为引导扇区太小无法放下它。这个例程用一个 BIOS 调用将数据从低端移到高端内存，一次移动 64KB，它还要重置目的地址，引导扇区用来从盘上传送下一次的数据。这样，在 *bootsect.S* 中的一般加载例程就不会用尽低端内存。

在核心被加载后，*setup.S* 象往常一样被调用。它除了改变上一个跳转指令的目的地址外，并不做任何特殊的工作。由于我们加载了一个大映象，处理器通过使用一台特殊的机器指令（它允许 386 在实模式段使用 32 位偏移）跳到 0x100000 而不是 0x1000。

解压缩和往常一样工作，但输出不能放在 0x100000（1M），因为压缩的映象已经在那儿了。解压的数据被写到低端的内存直到用尽；接着被写到越过压缩映象的地方。这两个解压的片段通过执行另外的内存移动在 0x100000 处装配起来。但复制例程也居于高端内存，它必须首先将自己复制到低端内存已防止被覆盖；然后它把整个映象移到 0x100000。

到这儿，游戏就结束了。但 kernel/head.S 并没有注意到发生的额外工作，所有事情照常进行。

使用 *lilo*

lilo，Linux 加载程序，居于引导扇区----或者是主引导扇区，或者是磁盘分区的第一个扇区。它使用 BIOS 调用从一个文件系统中加载核心。

这个程序与核心映象面对同样的问题：在机器引导时，仅有半 KB 的代码被装入内存，而且只用几打的指令解码一个文件系统结构也是不可能的。*lilo* 通过在安装时构造一个磁盘映射来解决这个问题。它用这个映射告诉 BIOS 从正确的地方获取每个核心块。这个技术很有效，但你在替换或者重写一个核心映象后必须重新安装 *lilo*----你必须调用 *lilo* 命令，用一个新的核心块表来重新安装引导加载程序。

实际上，*lilo* 扩展了加载机制，它允许用户在引导时选择加载哪个映象。这个选择是通过一个映象的安装定义表来做到的。它用从不同的分区中取出的引导扇区代替它自己的引导扇区来实现。

lilo 比一个 barebone 引导的最大好处（除了能从硬驱直接引导外）是它允许用户象核心传递一个命令行。这个命令行可以在 *lilo* 配置文件中指定，也可以在引导时交互给出。*lilo* 把命令行放在零页（我们将其在 *boot/head.S* 之前保持空闲）的后一半。这一页以后由 *setup_arch*(在 *arch/i386/kernel/setup.c* 中定义)取得。

lilo 的最近版本（18 版本甚至更新）可以加载 *bzImage*，而老的发布是不能的。较新的版本可以用 BIOS 调用将数据加载到高端内存，象 *bootsect.S* 做的那样。

当 *lilo* 完成加载，它跳到 *setup.S*，事情就象我们以前看到的那样继续进行。

使用 *loadlin*

*loadlin*用来将Linux从一个实模式操作系统中引导起来。与*lilo*类似，都是加载数据，传递命令行，跳至*setup.S*。但它有一个优点就是它可以在FAT分区中从一个指定的文件名加载核心，

而不需要一个块的映射。这使得它比较稳定。如果你想加载*bzImage*，你需要*loadlin*的版本1.6 或更新*。有趣的是注意到*loadlin*可能需要玩一些脏活才能加载整个核心，同时又不至于搞乱宿主操作系统。只有在核心的所有部分都被加载了，*loadlin*才能在合适的地址重新装配它，并调用它的入口点。

其它引导方式

还有一些程序可以引导 Linux 核心。其中的两个是 *Etherboot* 和 *syslinux*，当然还有很多。不过我不打算在这里讲述它们，因为它们与我已经讲过的类似，至少与核心相关的部分如此。但要注意，引导一个 Linux 核心并不是象我说的这么简单。要进行大量的检测，版本号经常出现在特别的地方，以抓住用户的错误，并友好的回复。意思是如果发生了什么问题，系统可以在挂起前打印一条信息。局限在 x86 实模式的执行环境下很难完全避免发生错误时挂起，打印一条消息总比什么都没有强。

设置 Alpha 处理器

让一个 Alpha 到达能运行 *start_kernel* 这一点要比 Intel 处理器容易的多，因为在 Alpha 上不需要和实模式或内存限制做斗争。而且，Alpha 工作站通常配有比 PC 好的固件，可以从文件系统装载一个完整的文件。我不想讨论装载一个文件时的实际步骤，因为这个代码没有随 Linux 发布，这样你无法检查它----我也不能，因此就无法谈论它。

milo（迷你加载程序）程序是引导的一般选择。*milo* 比固件要聪明，因为它理解 Linux 和它的文件系统，但又比核心笨，因为它不能运行进程。*milo* 由固件从 FAT 分区执行，可以从 ext2 或 ISO9660 块设备上加载核心。象 *lilo* 和 *loadlin*，*milo* 也向核心传递一个命令行。在 Linux 被加载到内存中正确的虚地址后，*milo* 转向核心，自己消失。

milo 的有些特征依赖于核心源码，因为它需要访问设备，理解文件系统布局。配有驱动程序和文件系统类型，它可以根据文件名从硬盘或 CD-ROM 上取得核心映象。这个设计后面的想法与 *loadlin* 类似，只是 *milo* 使用 Linux 核心的代码，而不是取自别的操作系统环境。

在 Alpha 上引导 Linux 并不总是可用 *milo*。如果你的系统有 SRM 固件，就不能安装 *milo*。相反，你可以使用 arch/alpha/boot 中的原始加载程序。这个加载程序很简单，能从硬盘或软驱中读取一个顺序区域，这与 PC 上 *zImage* 前面的引导扇区所做的工作一样。使用原始加载程序要求核心映象必须（在任何文件系统之外）被复制到磁盘上的连续区域。

如果不考虑系统是如何引导的，控制被传递给 *arch/alpha/kernel/head.S*，但 Linus 说：“没什么需要我们做的了”。源码只是设置几个指针，然后就跳到 *start_kernel*。

设置 Sparc 处理器

Sparc 计算机用一个称为 *silo* 的程序引导 Linux。与 *lilo*，*milo* 命名方法类似，只是用“s”表示 Sparc。引导 Sparc 比 Alpha 要简单一些；它的固件可以访问设备，*silo* 只需要访问 Linux 文件系统，并与用户交互。出于这个目的，*silo* 被链接到 *libext2*，这是支持对未安装分区上的文件进行处理的一个库。

若不使用 *silo*，也可以从软盘或网络上引导计算机。固件可以用 RARP（反向 ARP）和 tftp 协议从以太网上装载一个核心。事实上，我从未用软盘引导过我自己的工作站，因为 Linux 的 Sparc 发布允许通过网络引导来完成系统安装。

对 Sparc 来说，的确没有什么特别的要求。没有实模式，也没有需要复制的内存。一旦核心被加载到 RAM，它便开始执行。

---

* 你可能需要 1.6a或更新来加载 2.1.22 或更新的核心。

Init 进程

由 *start_kernel* 生成的线程派生出 bdflush（源码见 *fs/buffer.c*）和 kswapd（在 *mm/vmscan.c* 中定义），它们因此被赋予进程号 2 和 3。*init* 进程（pid 1）接着进行进一步的初始化，这在之前不可能完成；也就是，它运行与 SMP 相关的函数，如果需要，还有 initrd 引导技巧，以另一个核心线程的形式。在 initrd 结束后，init 线程激活 UMSDOS 文件系统的“伪根(pseudo-root)”。

在完成初始化后，*init* 的实际作用是进入用户空间并执行一个程序（因此变成一个进程）。这样三个 stdio 通道被连到第一个虚拟控制台，核心试图从/etc 执行 *init*。如果失败，它将查看*/bin* 和*/sbin*（在所有最近的发布中，*init* 一定居于此）。如果 *init* 从这三个目录的执行都失败了，进程将会执行*/etc/rc*，如果这个也失败了，它就循环，执行*/bin/sh*。在大多数情况下，函数能运行 *init* 成功；其它选项的目的是为了在 *init* 不能执行时允许系统恢复。

如果核心命令行指定了一条要执行的命令，使用 init=*some_program* 导语，进程 1 就执行指定的命令，而不是调用 *init*。

不管系统是怎么设置的，*init* 最终在用户空间执行，以后的核心操作都是对来自用户程序的系统调用的响应。

kernel 目录

大多数关键的核心功能都是在这个目录实现的。这里最重要的源文件是 *sched.c*，它值得特别对待。

sched.c

正如源文件自己表明的，这是“主核心文件”。它由调度程序和相关操作组成，例如让进程睡眠和唤醒它们，以及核心计时器的管理（见第六章“时间流”中“核心计时器”一节），间隔计时器（它与计帐和性能刻划有关），以及预定义任务队列（见第六章“预定义任务队列”）。

如果你对 Linux 调度程序的实时策略感兴趣，你可以在 schedule 函数及其相关者中找到低级的信息。其中一个相关者是 goodness，它给进程赋优先值，并帮助调度程序选择下一个要运行的进程。

与调度程序控制相关的函数（及系统调用）也在这个文件中定义。这包括设置和取得调度策略及优先级。在除 Alpha 外的其它体系结构上，系统调用 *nice* 也在这个源文件中。

另外，取得和设置用户及组id也在*sched.c*中定义（除了Alpha），同时还有*alarm*调用*。

在 *sched.c* 中还能找到的其它好东西包括 *show_tasks* 和 *show_state* 函数，它们实现了在第四章“调试技巧”中“系统挂起”一节所描述的“魔幻”键中的两个。

进程控制

目录的其它主要部分都是管理进程的。*fork* 和 *exit* 系统调用在两个同名源文件中实现，信号控制在 *signal.c* 中实现。大多数信号处理的调用在 Alpha 中实现的方法是不同的，以保证 Alpha 的移植与 Digital Unix 二进制兼容。

fork 的实现包括 clone 系统调用的代码，*fork.c* 显示了 *clone* 的标志是如何使用的。应该注意 *sys_fork* 并不在 *fork.c* 中定义，因为 Sparc 的实现与其它的版本稍有不同；不过，多数 *sys_fork* 的实现只是调用 *do_fork*，它在 *fork.c* 中定义。提供一个缺省实现（通常叫做 *do_fnct*），而真

* 这个调用在当前的libc版本中不再使用，它通过计时器的方式实现这个函数。

正的系统调用（*sys_fnct*）则在各个移植中声明，这是 Linux 常用的一个技巧，随着新的移植的出现，这个技巧很可能扩展到其它的系统调用。
exit.c 实现 *sys_exit* 和不同的 *wait* 函数，以及信号的实际发送。（*signal.c* 专用于信号处理，而不是发送。）

模块化

文件 *module.c* 和 *ksyms.c* 包含了在第二章“构造和运行模块”中描述的机制。*module.c* 含有被 *insmod* 及相关程序使用的系统调用，*ksyms.c* 声明不属于特定子系统的核心中的公共符号。其它的公共符号由特定核心子系统的初始化函数使用 *register_symtab* 声明。例如，*fs/proc/procfs_syms.c* 为注册新文件声明*/proc* 接口。

其它操作

这个目录中的其余源文件为一些低级操作提供软件接口。*time.c* 从用户程序读写核心时间值，*resource.c* 为 I/O 端口实现请求和释放机制，*dma.c* 为 DMA 通道完成同样的工作。*softirq.c* 处理下半部（见第九章“中断处理”中“下半部”一节），*itimer.c* 定义系统调用来设置和取得间隔计时器值。
想知道核心的消息处理是如何工作的，你可以看 *printk.c*，它显示了在第四章介绍的几个概念的一些细节（也就是说，它包含了 *printk* 和 *sys_syslog* 的代码）。
exec_domain.c 包含了获得与其它风格的 Unix 兼容性所需要的代码，*info.c* 定义了 *sys_info*。*panic.c* 做的工作正如它的名字所示；它还支持在系统不稳定后自动重启动。重新引导发生在由*/proc/sys/kernel/panic* 设置的延迟之后。这个延迟通过对 udelay(1000)的重复调用实现，因为在系统崩溃后，调度程序不再运行，udelay 可以用于不长于 1 毫秒的延迟（见第六章“长延迟”一节）。
sys.c 实现几个系统配置和权限处理函数，如 *uname*，*setsid* 及类似的调用。*sysctl.c* 包含 *sysctl* 调用的实现和在 sysctl 表（系统控制入口点列表）注册及取消注册的入口点。这个文件也提供了按照注册的表访问*/proc/sys* 文件的能力。

mm 目录

在 mm 目录中的文件为 Linux 核心实现内存管理中体系结构无关的部分。这个目录包含换页及内存的分配和释放的函数，还有允许用户进程将内存区间映射到它们地址空间的各种技术。

换页和对换

令人惊奇的是，*swap.c* 并未实现对换算法。相反，它处理核心的命令行选项 swap=和 buff=。这些选项也可以通过 *sysctl* 系统调用或写文件*/proc/sys/vm* 来设置。
sawp_state.c 负责维护对换高速缓存，是这个目录中最难的文件；我不想讨论它的细节，因为很难理解它的设计，除非以前对相关的数据结构和策略已经有了很好的了解。
swapfile.c 实现对换文件和设备的管理。*swapon* 和 *swapoff* 系统调用在这里定义，后者代码非常困难。作为比较，有几个 Unix 系统没有实现 *swapoff*，这样如果不重启动就无法停止向一个设备或文件的对换。*swapfile.c* 还声明了 *get_swap_page*，它从对换池中取得一个空闲页。*vmscan.c* 实现换页策略。*kswapd* 守护进程在这个文件定义，还有扫描内存，运行进程寻找可换出的页的函数。
最后，*page_io.c* 实现了与对换空间之间进行低级数据传送的功能。这个文件管理保证系统

一致性的锁机制，提供同步和异步的 I/O。它还处理与不同设备使用不同块大小相关的问题。（在 Linux 的早期版本，不可能对换到一个 FAT 分区上，因为不支持 512 字节的块。）

分配和释放

再第七章介绍的内存分配技巧都在 *mm* 目录实现。让我们再一次从最常用的函数开始：*kmalloc*。

kmalloc.c 实现内存区域的分配和释放。*kmalloc* 的内存池由一些“桶”组成，每个桶是同样大小的内存区域的列表。*kmalloc.c* 的主要功能是管理每个桶的链表。

当需要新页或有页面被释放时，这个文件利用在 *page_alloc.c* 中定义的函数。页面用 *__get_free_pages* 从空闲内存中取得，这是一个从空闲页列表中取得页面的短函数。如果空闲列表中没有内存可用，就调用 *try_to_free_pages*(*vmscan.c*)。

vmalloc.c 实现了 *vmalloc*，*vremap*，*vfree* 函数。*vmalloc* 返回核心虚拟空间中的连续内存，*vremap* 给出特定物理地址的新的虚地址；它主要用来访问高端内存的 PCI 缓冲。*vfree* 释放内存，如它的名字所示。

其它接口

Linux 内存管理最重要的函数是 *memory.c* 文件的一部分。这些函数一般不能通过系统调用访问，因为它们处理硬件的换页机制。

另一方面，模块的作者的确使用这些函数。*verify_area* 和 *remap_page_range* 在 *memory.c* 中定义。其它有趣的函数是 *do_up_page* 和 *do_no_page*，它们实现核心对次和主页面错的反应。文件中的其余函数处理页表，都非常低级。

内存映射是 *mm* 目录中文件处理的另一个大任务。*filemap.c* 的代码很复杂。它实现常规文件的内存映射，提供支持共享映射的能力。被映射文件通过被映射页的特殊结构 vm_operations 支持，如在十三章“Mmap 和 DMA”中“虚拟内存区域”一节中所描述的。这个源文件页处理异步提前读；注释解释了结构 file 中四个提前读域的含义。这个文件中出现的唯一的系统调用是 *sys_msym*。到内存映射的顶级接口（即 *do_mmap*）出现在 *mmap.c*。

这个文件有定义 *brk* 系统调用开始，它被一个进程用来请求其最高允许的虚地址被增加或减小。*sys_brk* 的代码提供了很多信息，即使你不是内存管理的大师。*mmap.c* 的其余部分集中在 *do_mmap* 和 *do_munmap*。如你所期望的，内存映射通过 filp->f_op 完成，尽管 filp 对 *do_mmap* 可能为 NULL。这是 *brk* 如何分配新的虚拟空间的。它还是依赖于内存映射零页，而不需要特殊代码。

mremap.c 包括 *sys_mremap*。如果你已经搞清楚了 *mmap.c*，这个文件就很容易了。

与内存锁定和解锁相关的四个系统调用在 *mlock.c* 中定义，它是相当简单发源文件。类似第，*mprotect.c* 负责执行 *sys_mprotect*。这些文件在定义上很相似，因为它们都修改了与进程页相关的系统标志。

fs 目录

在我的观点中，这个目录是整个源码树中最有趣的一部分。文件处理是任何 Unix 系统的一个基本活动，所有与文件相关的操作都在这个目录中实现。我不想在这儿描述 *fs* 子目录，因为每种文件系统类型仅仅是把 VFS 层映射到特别的信息布局上。但讲一下 *fs* 目录中多数文件的作用还是很重要的，因为它们携带了大量的信息。

Exec 和二进制格式

Unix 中最重要的系统调用是 *exec*。用户程序可以 *exec* 六种不同的形式，但在只有一个在核心中实现----其它都是隐射到全特征实现 *execve* 上的库函数。
exec 函数用一个已注册二进制格式的表以查找用来加载和执行一个磁盘文件的正确的加载程序。源文件的第一部分定义了 *register_binfmt* 接口。有兴趣的是注意到脚本文件的#!魔幻键被作为二进制格式处理，如 ELF 和另外一些格式（尽管在 1.2 版本中它是 *exec.c* 的一种特殊情况）。*kerneld* 也需要了解一个新的二进制格式是如何按照要求从读取源文件被加载的。每个二进制格式由一个定义了三种操作（加载一个二进制文件，加载一个库，倾倒内核（core dump））的数据结构描述。这个结构在<linux/binfmts.h>中定义。决大多数格式都只支持第一个操作（加载一个文件），不过这个接口已经足够一般化，能够支持任何可预见的新格式的需要。

devices.ch 和 block_dev.c

我们已经用了 *devices.c* 中的代码，因为它负责设备注册和取消注册。它同时也负责缺省的设备 *open* 方法，以及对块设备的 *release* 方法。这些调用为被打开或关闭的设备取得正确的文件操作并将执行分派到正确的方法。对模块自动加载的支持在这个文件中实现。除了打开和关闭设备之外的所有东西都出现在 *drivers/**目录，如 filp->f_op 所指示的。
block_dev.c 包含读写块设备的缺省方法。如你可能记得的，一个块驱动程序并不声明它自己的 I/O 方法，只是它的请求例程。*block_dev.c* 的缺省读写实现了解缓冲高速缓存，除了实际的数据传送它提你做了所有的事情。

VFS:超级块

程序和设备的执行只是 *fs* 目录的一部分。*fs* 的多数文件，以及子目录中的所有文件，都与文件相关的系统调用有关。更特别地：它们实现了所谓的 VFS 机制：虚拟文件系统（或虚拟文件系统切换----这些解释有点互相矛盾）。
概念地说，VFS 是 Linux 文件处理软件的一层。通过利用各种文件系统格式提供的特征，这层提供了到文件的统一接口。在磁盘上布局信息的各种技巧可以通过 VFS 接口用一致的方法访问。实际上，VFS 减少到几个定义“操作”的结构。每个文件系统声明处理超级块、inode、和文件的这些操作。我们在本书中已经使用的 file_operations 结构就是 VFS 接口的一部分。核心通过安装每个文件系统来访问它。*mount* 的一个任务是从磁盘上搜取所谓的“超级块”。超级块是一个文件系统中的主要数据结构。它的名字来自于一个事实，历史上，它曾经是磁盘上的第一个物理块。文件 *super.c* 包括与超级块有关的有趣操作的源码：读取的同步它们、安装和卸装文件系统、在引导时安装根文件系统。
除了这些有趣的（还有点复杂）操作，*super.c* 也返回与文件系统有关的信息，包括由 */proc/mounts* 和*/proc/filesystems* 提供的信息。
函数 *register_filesystem* 和 *unregister_filesystem* 被模块化的文件系统类型使用；它们也在 *super.c* 中定义。文件 *filesystems.c* 是一个#ifdef 语句的短表。依赖于那些选项被编译进核心，对应不同文件系统的各种 *init* 函数被调用。每个文件系统类型的 *init* 函数调用 *register_filesystem*，因此不需要别的条件编译选项。

Inode 和高速缓存技术

VFS 接口的下一块是 inode。每个 inode 由一个由设备号和 inode 号组成的唯一的键值确定。用户程序用文件名去访问文件系统中的结点，核心负责将文件名映射到唯一的键值。为了获得更好的性能，Linux 维护了两个与 inode 键值相关的高速缓存：inode 高速缓存和名字高速

缓存（也叫目录高速缓存）。另外，核心还负责已经熟悉的缓冲高速缓存。
inode 高速缓存是一个哈希表，用于从设备/ inode 号键值查找 inode 结构。高速缓存的实现，以及读写 inode 的例程，都在 *inode.c* 中。这个文件也实现了 inode 结构的锁机制以防止可能的死锁。
名字高速缓存是一个表，它将 inode 号和文件名关联起来。当一个名字被连着用了几次，高速缓存就可以避免重复的目录查询。源文件 *dcache.c* 包含管理高速缓存的软件机制。使用名字高速缓存的多数系统调用和函数是 *namei.c*（表示 name-inode）的一部分，包括 *sys_mkdir*，*sys_symlin*k，*sys_rename*，及类似的调用。
缓冲高速缓存是系统中最大的数据高速缓存，它的实现出现在巨大的文件 *buffer.c* 中。
至于文件，*file_table.c* 负责 file 结构的分配和释放。这包括 *get_empty_filp*，它被 *open*，*pipe*，*socket* 调用。

open.c

fs 中其它源文件中多数负责文件操作----与在驱动程序中需要实现的一样。第一个这样的文件是 *open.c*，它包括了很多系统调用的代码。它也包括 *sys_open*，及它的低级的对应者 *do_open*，还有 *sys_close*。这些系统调用都很直接，被映射到 filp->f_op。
open.c 包含修改 inode 的系统调用：*chown* 和 *chmod*，以及它们的对应者 *fchown* 和 *fchmod*。如果你对安全检查和不可变标志的使用感兴趣，你可以浏览源码，它可以被几乎所有的 Unix 编程者理解。改变 inode 中的次数也被支持----*utime* 和 *utimes* 在这里定义。
chroot，*chdir*，和 *fchdir* 也在 *open.c* 中找到，同时还有其它“改变”函数。
源码中定义的第一个函数是 *statfs* 和 *fstatfs*，它们通过 inode->I_sb->s_op->statfs 被分派文件系统特定的代码。
truncate 和 *access* 调用也出现在这个文件中。后者用进程的实际 uid 和 gid 检查文件的权限，暂时不考虑 fsuid 和 fsgid。

read.c 和 *readdir.c*

正如其名字所示，*read_write.c* 包含读和写，但它也包含了 *lseek* 和 *llseek*（有人猜测这个前导 l 的数目是每十年增加一个）。*lseek* 是使用 off_t(long)的标准调用，而 *llseek* 使用 loff_t(long long)。*llseek* 系统调用映射到 *lseek* 文件操作，它被作为 *lseek* 的超集实现。有趣的是注意到核心 2.1 版将这个方法改名为 *llseek*，以与其实现保持一致。
read 和 *write* 是很简单的函数，因为实际的数据传送是通过 filp->f_op 来分派的；*read_write.c* 也包含了 *readv* 和 *writev* 的代码，它们稍为复杂，因为多块数据的传输必须跨过核心和用户边界。
Linux 不允许你直接读一个目录文件，如果你尝试，将会返回-EISDIR。目录只能用 *readdir* 系统调用来读取，它是文件系统无关的；或使用 *getdents*，即“取得目录项”。用 *getdents* 读一个目录要快一点，因为一个调用可以返回很多目录项，而 *readdir* 一次只能返回一项。不过，*getdents* 只被 libc-5.2 或更新的库所支持。

select.c

select 的完整实现居于 *select.c*，除了 *select_wait* 线入函数。尽管 *select.c* 的代码很有趣，但却很难阅读，因为数据结构太复杂（这在第五章“增强的字符设备驱动程序操作”中“底层数据结构中讨论过”）。不过 *select.c* 倒是阅读核心代码的一个好的起始点，因为它是自包含的；除了一些无关紧要的细节外它不依赖与其它源文件。

Pipe 和 fifos

pipe 和 fifo 通信通道的实现与字符设备驱动程序非常类似。通过为两个通道使用同样的文件操作来避免了代码重复；只有 *open* 方法不一样。除了 *fifo_open* 和 *fifo_init* 外所有的函数都在 *pipe.c* 中定义。由于 fifo 与文件系统结点类似，除了 file_operations 结构，它们还需要一组 inode_operations 与之相关。正确的结构在 *fifo.c* 中定义。

下一个有趣的事情是注意到在 pipe 和 fifo 的实现中，*pipe.c* 定义了两个 file_operation 结构：一个是为通道的可读侧的，一个是为可写侧的。这允许在 *read* 和 *write* 中跳过权限检查。

控制函数

文件的“控制”系统调用在两个根据这个调用命名的文件中：*fcntl.c* 和 *ioctl.c*。前者基本上是自包含的，因为为 *fcntl* 定义的所有的命令都是预定义的。由于其中一些只是 *dup* 的包裹者，因此 *dup* 的实现也在 *fcntl.c* 中。另外，由于 *fcntl* 调用负责异步触发，*kill_fasync* 也可以在此找到。

ioctl.c 包含 *ioctl* 系统调用的外部接口。它是一个短文件，当它收到不认识的命令时，还要依赖于文件操作。

文件锁

Linux 中实现了两类锁接口，*flock* 系统调用和 *fcntl* 命令。后者是 POSIX 兼容的。

文件 *locks.c* 包含了处理两个调用的代码。它也包括对强制锁的支持，它在 2.0.0 之前是一个编译选项，在 2.1.4 被改为一个安装（mount）时选项。

次要文件

fs 的文件还支持磁盘定额（quota）。*dquot.c* 实现了定额机制，而 *noquot.c* 包含空函数；如果定额没有被包含在核心的配置中，它就代替 *dquot.c* 被编译。

最后，*stat.c* 实现了 *stat*、*lstat*、*readlink* 系统调用。在 2.0.x 定义了 *stat* 和 *lstat* 的两个实现，以保持与旧的 x86 库的后向兼容。

网络

Linux 文件体系中的 net 目录是套接字抽象和网络协议的容库；这些特征使用了大量的代码，因为 Linux 支持集中不同的网络协议。每个协议（IP，IPX 等等）都居于它自己的子目录中。Unix 域的套接字被以一种网络协议对待，它们的实现可以在 *unix* 子目录找到。有趣的是注意到 2.0 版的核心只包含了 IPv4，而版本 2.1 包含了相当完整的 IPv6 的支持，新来的标准解决了 IPv4 的编号问题。

Linux 中的网络实现是基于用于设备文件的同样的文件操作。这很自然，因为网络连接（套接字）是用一般文件描述符描述的。在核心中一个套接字是由一个结构 socket(<linux/net.h>) 描述。文件 *socket.c* 是套接字文件操作的容库。它通过结构 proto_ops 分派系统调用到其中的一个网络协议。这个结构被每个网络协议定义以将系统调用映射到低级的数据处理。

net 下的每个目录（除 *bridge*，*core*，*ethernet*）都专用来实现一个网络协议。目录 *bridge* 包含符合 IEEE 规范的以太网桥的优化实现。*core* 中文件实现了通用的网络特征如：设备处理，火墙，选播和异名；这包括独立于底层协议（*core/sock.c*）的套接字缓冲区(*core/skbuff.c*)和套接字操作的处理。最后，*ethernet* 包含通用的以太网函数。

几乎每个 *net* 下的目录都有一个处理系统控制的文件；这样引出的信息可以通过*/proc/sys* 文件树或通过 *sysctl* 系统调用来访问。到 *sysctl* 的核心接口允许对系统控制入口点动态的增加

和去除，在<linux/sysctl.h>中定义。

IPC 和 lib 函数

进程间通信和库函数各有一个小的专用目录。

ipc 目录包含一个名为 *util.c* 的通用文件，以及每种通信方式的一个源文件：*sem.c*，*shm.c*，*msg.c*。*msg.c* 负责消息队列和 *kerneld* 引擎，*kerneld_send*。如果 IPC 没有在编译时打开，*util.c* 引出一些空函数，它们通过返回-ENOSYS 来实现 IPC 相关的系统调用。

库函数就象你在 C 程序中常见的一些工具和变量：*sprintf*，*vsprintf*，errno 整数值，以及被各种<linux/ctype.h>宏使用的_ctype 数组。文件 *string.c* 包含字符串函数的可移植实现，但只有在体系结构特定的代码不包含优化的线入函数时才能编译。如果线入函数在头文件中定义，那么在 *string.c* 中的实现应该用#ifdef 语句排除在外。

lib 中最“有趣”的文件是 *inflate.c*，它是 *gzip* 的“gunzip”部分，是从 *gzip* 本身展开从而允许在引导时使用压缩的 RAMdisk。这个技巧在需要的数据除了压缩，不能在一张软盘上放下时使用。

Drivers

现在对 Linux 的 *drivers* 目录已经没什么可说的了。这个目录中的源文件在整部书里已经被广泛引用；这也使我为什么在讨论源文件树时把它们留在最后。

字符设备、块设备、和网络驱动程序

尽管这些目录中大多数驱动程序是特定于某种特别的硬件的，还是有几个文件在系统设置时起到比较通用的作用。

关于目录 *drivers/char*，实现 N_TTY 行规则的代码就在这儿。N_TTY 是系统 tty 的缺省行规则，它在 *n_tty.c* 中定义。在 drivers/char 中的另一个设备无关的文件是 *misc.c*，它提供对“杂类”设备的支持。一个 “杂类”设备是有一个次设备号的简化的字符设备驱动程序。

这个目录还包含了对 PC 控制台的支持，和其它一些体系结构相关的驱动程序；它实际上包含了一些在其它地方都不合适的杂项的集合。

drivers/block 则要清晰多了。它包含了多数快设备驱动程序的单个文件驱动程序，和全特征的 IDE 驱动程序，它被劈成了多个文件。这个目录的资格文件提供了通用目的的支持；*genhd.c* 处理分区表，*ll_rw_block.c* 负责与物理设备之间数据传送的低级机制。request 结构是 *ll_rw_block.c* 的主要部分。

drivers/net 包含了 PC 网卡驱动程序的长列表，以及几个其它体系结构的驱动程序（例如，*sunlance.c* 是为了多数 Sparc 计算机的接口的）。有些驱动程序比它看起来要复杂一些，这在第十四章“网络驱动程序”中介绍过。例如 *ppp.c* 声明自己的行规则。

在 *drivers/net* 下的通用目的源文件是 *Space.c* 和 *net_init.c*。*Space.c* 主要是包含了一个可用网络设备的表。这个表包含了一个#ifdef 项的长列表，它们在系统引导时被检差以检测和初始化网络设备。*net_init.c* 包含 *ether_setup*，*tr_setup*，和类似的通用目的函数。

SCSI 驱动程序

如在第一章“Linux 核心介绍”中所提到的，Linux 中的 SCSI 驱动程序没有被包含在一般的字符和块设备类中。这是因为 SCSI 接口总线有它自己的标准。因此，将 SCSI 设备和其它驱动程序分开，开发者可以分离和共享公用代码。

drivers/scsi 中多数文件是为特定 SCSI 控制器的低级驱动程序。通用目的的 SCSI 实现在文件 *scsi_*.c* 中定义，还有 *sd.c* 是磁盘支持，*sr.c* 是 CD-ROM 支持，*st.c* 是 SCSI 磁带支持，*sg.c* 是通用 SCSI 支持。最后一个源文件对使用 SCSI 协议的设备定义了通用目的支持。扫描仪和其它通用设备可以由用户程序通过使用*/dev/sg**设备结点来控制。

其它子目录

别的硬件驱动程序由它们自己的子目录。*drivers/cdrom* 包含即不是 IDE 也不是 SCSI 的光驱的驱动程序。它们是常规的块设备驱动程序，有它们自己的主设备号。

drivers/isdn 是（如其名字所示）Linux 的 ISDN 实现；*drivers/sound* 是最常用的 PC 声卡板子的驱动程序集合。*drivers/pci* 包含了一个文件，它包含了所有已知 PCI 设备（销售商/ID 对）的列表。实际的 PCI 功能不在此定义，而是在体系结构特定的目录。

最后，*sbus* 包含了一个 *char* 子目录，和 Sparc 体系结构的控制台代码。随着 2.1 开发的进行，这个目录增长的很快。

体系结构相关性

Linux 核心的 2.0 及更新版具有相当好的跨平台可移植性，也就是说大多数代码可以不加区别地在所有支持的体系结构上运行。到目前为止，我们所看到的所有东西都是与硬件平台完全无关的。

arch 目录树是 Linux 核心中包含平台特定代码的一小部分。每个系统相关的函数都在每个 *arch* 子目录中被复制，因此所有子目录的结构都是类似的。这些子目录中最重要的是 *kernel*，它存放了与主要 *kernel* 源码目录相关的每个系统特定的函数。

有两个汇编源文件总能在 *kernel* 下找到。*head.S* 是在系统引导时执行的启动代码；它有时包含一些例外处理代码。另一个文件是 *entry.S*，它包含了到核心空间的入口点。特别地，每个这样的文件包含其自己体系结构的 sys_call_table；每个体系结构有一个不同的表将系统调用号和函数关联起来。

另一个常常找到的子目录是 *lib*，它包含网络包的优化的校验和例程，有时还包含别的低级操作，如串操作；还有 *mm*，它进行页面错的低级处理（*fault.c*），以及在系统引导时的初始化代码（*init.c*）；还有 *boot*，它包含启动系统的代码。如你可能想象的，*i386/boot* 是 *boot* 子目录中最复杂的部分。

我不打算描述体系结构特定的代码，因为阅读它们并不有趣，而且它们往往充斥着汇编语句。为了理解这些代码，你首先得知道目标体系结构的一些细节。我不论如何也不会觉得阅读体系结构特定的函数有什么乐趣，因为它们是系统中肮脏的部分，处理大量硬件小问题。核心的其它部分才是有趣的。