Linux 内核源代码漫游

Alessandro Rubini 著, rubini@pop.systemy.it
赵炯 译，gohigh@sh163.net (www.plinux.org)

本章试图以顺序的方式来解释 Linux 源代码，以帮助读者对源代码的体系结构以及很多相关的 unix 特性的实现有一个很好的理解。目标是帮助对 Linux 不甚了解的有经验的 C 程序员对整个 Linux 的设计有所了解。这也就是为什么内核漫游的入点选择为内核本身的启始点:系统引导(启动)。

这份材料需要对 C 语言以及对 Unix 的概念和 PC 机的结构有很好的了解，然而本章中并没有出现任何的 C 代码，而是直接参考（指向）实际的代码的。有关内核设计的最佳篇幅是在本手册的其它章节中，而本章仍趋向于是一个非正式的概述。

本章中所参阅的任何文件的路径名都是指主源代码目录树，通常是/usr/src/linux。

NEW 这里所给出的大多数信息都是取之于 Linux 发行版 1.0 的源代码。虽然如此，有时也会提供对后期版本的参考。这篇漫游中开头有 NEW 图标的任何小节都是强调 1.0 版本后对内核的新的改动。如果没有这样的小节存在，则表示直到版本 1.0.9-1.1.76，没有作过改动。

MORE有时候本章中会有象这样的小节，这是指向正确的代码以对刚讨论过的主题取得更多信息的指示符。当然，这里是指*源代码*。

引导（启动）系统

当 PC 的电源打开后，80x86 结构的 CPU 将自动进入实模式，并从地址 0xFFFF0 开始自动执行程序代码，这个地址通常是 ROM-BIOS 中的地址。PC 机的 BIOS 将执行某些系统的检测，在物理地址 0 处开始初始化中断向量。此后，它将可启动设备的第一个扇区读入内存地址 0x7C00 处，并跳转到这个地方。启动设备通常是软驱或是硬盘。这里的叙述是非常简单的，但这已经足够理解内核初始化的工作过程了。

Linux 的最最前面部分是用 8086 汇编语言编写的(boot/bootsect.S)，它将由 BIOS 读入到内存 0x7C00 处，当它被执行时就会把自己移到绝对地址 0x90000 处，并将启动设备(boot/setup.S)的下 2kB 字节的代码读入内存 0x90200 处，而内核的其它部分则被读入到地址 0x10000 处。在系统加载期间将显示信息"Loading..."。然后控制权将传递给 boot/Setup.S 中的代码，这是另一个实模式汇编语言程序。

启动部分识别主机的某些特性以及 vga 卡的类型。如果需要，它会要求用户为控制台选择显示模式。然后将整个系统从地址 0x10000 移至 0x1000 处，进入保护模式并跳转至系统的余下部分（在 0x1000 处）。

下一步是内核的解压缩。0x1000 处的代码来自于 zBoot/head.S，它初始化寄存器并调用 decompress_kernel(),它们依次是由 zBoot/inflate.c、zBoot/unzip.c 和 zBoot/misc.c 组成。被解压的数据存放到了地址 0x10000 处(1 兆)，这也是为什么 Linux 不能运行于少于 2 兆内存的主要原因。[在 1 兆内存中解压内核的工作已经完成，见 Memory Savers--ED]

MORE将内核封装在一个 gzip 文件中的工作是由 zBoot 目录中的 Makefile 以及工具完成的。它们是值得一看的有趣的文件。

NEW 内核发行版 1.1.75 将 boot 和 zBoot 目录下移到了 arch/i386/boot 中了，这个改动意味着对不同的体系结构允许真正的内核建造，不过我将仍然只讲解有关 i386 的信息。

解压过的代码是从地址 0x10100 处开始执行的**[这里我可能忘记了具体的物理地址了，**

因为我对相应的代码不是很熟]，在那里，所有 32 比特的设置启动被完成: IDT、GDT 以及 LDT 被加载，处理器和协处理器也已确认，分页工作也设置好了；最终调用 start_kernel 子程序。上述操作的源代码是在 boot/head.S 中的，这可能是整个内核中最有诀窍的代码了。

注意如果在前述任何一步中出了错，计算机就会死锁。在操作系统还没有完全运转之前是处理不了出错的。

start_kernel()是位于 init/main.c 中的，并且没有任何返回结果。从现在起的任何代码都是用 C 语言编制的，除了中断管理和系统调用的入/出代码（当然，还有大多数的宏都嵌入了汇编代码）。

让轮子转动起来

在处理了所有错综复杂的问题之后，start_kernel()初始化了内核的所有部分，尤其是:

- 设置内存边界和调用 paging_init()；
- 初始化中断、IRQ 通道和调度；
- 分析（解析）命令行；
- 如果需要，就分配一个数据缓冲区(profiling buffer)以及其它一些小部分；
- 校正延迟循环（计算“BogoMips”数)；
- 检查中断 16 是否能与协处理器工作。

最后，为了生成初始进程，内核准备好了移至 move_to_user_mode()，它的代码也是在同一个源代码文件中的。然后，所谓的空闲任务，进程号 0 就进入无限的空闲循环中运行。

接着初始进程（init process）尝试着运行/etc/init、/bin/init 或者/sbin/init。

如果它们没有一个运行成功的，就会去执行代码“/bin/sh /etc/rc”并且在第一个终端上生成一个根命令解释程序（root shell）。这段代码回溯至 Linux 0.01，当时操作系统只有一个内核，并且没有登录进程。

在从一个标准的地方（让我们假定我们有）用 exec()执行了 init 初始化程序之后，内核就对程序的执行没有了直接的控制。从现在起它的规则是提供对系统调用的处理，以及为异步事件服务(比如硬件中断等)。多任务的环境已经建立，从现在起是 init 程序通过 fork()派生出的系统进程和登录进程来管理多用户的访问了。

由于内核是负责提供服务的，这个漫游文章将通过观察这些服务（“系统调用”）以及通过提供基本数据结构的原理和代码的组织结构继续讨论下去。

内核是如何看见一个进程的

从内核的观点来看，一个进程只是进程表中的一个条目而已。

而进程表以及各个内存管理表和缓冲存储器则是系统中最为重要的数据结构。进程表中的各个单项是 task_struct 结构，是定义在 include/linux/sched.h 中的非常大的数据结构。在 task_struct 中保留着从低层到高层的信息，范围从某些硬件寄存器的拷贝到进程工作目录的 inode 信息。

进程表既是一个数组和双链表，也是一个树结构。它的物理实现是一个静态的指针数组，它的长度是定义在 include/linux/tasks.h 中的常量 NR_TASKS，并且每个结构都位于一个保留内存页中。这个列表结构是通过指针 next_task 和 pre_task 构成的，而树结构则是非常复杂的并且我们在此将不加以讨论。你可能希望改动 NR_TASKS 的默认值 128，但你要保证所有源文件中相关的适当文件都要被重新编译过。

在启动引导过程结束后，内核将总是代表某个进程而工作，并且全局变量 current --- 一个指向某个 task_struct 条目的指针 --- 被用于记录正在运行的进程。current 仅能通

过在 kernel/sched.c 中的调度程序来改变。然而，由于所有的进程都必须访问它，所以使用了宏 for_each_task。当系统负荷很轻时，它要比数组的顺序扫描快得多。

进程总是运行于“用户模式”或“内核模式”。用户程序的主体是运行于用户模式而其中的系统调用则运行于内核模式中。在这两种执行模式中进程所用的堆栈是不一样的 -- 常规的堆栈段用于用户模式，而一个固定大小的堆栈（一页，由该进程所有）则用于内核模式。内核堆栈页是从不交换出去的，因为每当一个系统调用进入时它就必须存在着。

内核中的系统调用（system calls）是作为 C 语言函数存在的，它们的‘正规’名称是以‘sys_’开头的。例如一个名为 burnout 的系统调用将调用内核函数 sys_burnout()。

MORE 系统调用机制在本手册的第三章中进行了讨论。观看在 include/linux/sched.h 中的 for_each_task 和 SET_LINKS 能够帮助理解进程表中的列表和树结构。

创建和结束进程

unix 系统是通过 fork()系统调用创建一个进程的，而进程的终止是通过 exit()或收到一个信号来完成的。它们的 Linux 实现位于 kernel/fork.c 和 kernel/exit.c 中。 派生出一个进程是很容易的，所以 fork.c 程序很短并易于理解。它的主要任务是为新的进程填写数据结构。除了填写各个字段以外，相关的步骤有：

- 取得一个空闲内存页面来保存 task_struct
- 找到一个空闲的进程槽（find_empty_process()）
- 为内存堆栈页 kernel_stack_page 取得另一个空闲的内存页面
- 将父辈的 LDT 拷贝到子进程
- 复制父进程的 mmap 信息

sys_fork() 同样也管理文件描述符和 inode。

NEW 1.0 的内核也对线程提供某些不够完善的支持，所以 fork()系统调用对此也给出了某些示意。内核的线程是主流内核以外的过程产品。

从一个进程中退出是比较有窍门的，因为父进程必须被通告有关任何子进程的退出。而且，一个进程可以由另外一个进程使用 kill()而退出（这些是 Unix 的特性），所以除了 sys_exit()之外，sys_kill()以及 sys_wait()的各种特性也存在于 exit.c 之中了。

这里不对 exit.c 的代码加以讨论---因为它一点也不令人感兴趣。为了以一致的状态退出系统，它涉及到许多细节。而 POSIX 标准对于信号则是要求相当严格的，所以这里必须对其加以叙述。

执行程序

在调用了 fork()之后，就有同一个程序的两个拷贝在运行了，通常一个程序使用 exec()执行另一个程序。exec()系统调用必须定位该执行文件的二进制映像，加载并执行它。词语‘加载’并不一定意味着“将二进制映像拷贝进内存”，因为 Linux 支持按需加载。 exec()的 Linux 实现支持不同的二进制格式。这是通过 linux_binfmt 结构来达到的，其中内嵌了两个指向函数的指针--一个是用于加载可执行文件的，另一个用于加载库函数，每种二进制格式都实现有这两个函数。共享库的加载是在 exec()同一个源程序中实现的，但我们只讨论 exec()本身。 Unix 系统提供了六种 exec()函数。除了一个以外，所有都是以库函数的形式实现的，并且，Linux 内核是单独实现 sys_execve()调用的。它执行一个非常简单的任务：加载可执行文件的头部，并试着去执行它。如果头两个字节是“#!”，那么就会解析该可执行文件的第一行并调用一个解释器来执行它，否则的话，就会顺序地试用各个注册过的二进制格式。 Linux 本身的格式是由 fs/exec.c 直接支持的，并且相关的函数是

load_aout_binary 和 load_aout_library。对于二进制，函数将加载一个“a.out”可执行文件并以使用 mmap()加载磁盘文件或调用 read_exec()而结束。前一种方法使用了 Linux 的按需加载机理，在程序被访问时使用出错加载方式（fault-in）加载程序页面，而后一种方式是在主机文件系统不支持内存映像时（例如“msdos”文件系统）使用的。

NEW 新近的 1.1 内核内嵌了一个修订的 msdos 文件系统，它支持 mmap()。而且 linux_binfmt 结构已是一个链表而不是一个数组了，以允许以一个内核模块的方式加载一个新的二进制格式。最后，结构的本身也已经被扩展成能够访问与格式相关的核心转储程序了。

访问文件系统

众所周知，文件系统是 Unix 系统中最为基本的资源了，它如此的基本和普遍存在以至于它需要一个更为便利的名字--我将忠于标准的称呼简单地称之为“fs”。

我将假设读者早已知道基本的 Unix 文件系统的原理--访问(权限)许可、i 节点(inode)、超级块、加载(mount)和卸载(umount)文件系统。这些概念在标准的 Unix 文献中由比我聪明的作者给出了很好的解释，所以我就不重复他们的工作并且我将只专注于有关 Linux 方面的问题。

早期的 Unix 通常只支持一个文件系统（fs）类型，它的代码散布于整个内核中，现今的实现是在内核和 fs 之间使用一个标准的接口，以便于在不同的体系结构中进行数据的交换。Linux 本身提供了一个标准层以在内核和每种 fs 模块之间传递数据。这个接口层称为 VFS，即“虚拟文件系统”（"virtual filesystem"）。

因而文件系统的代码被分割成了两层：上层是关于内核表格的管理和数据结构的，而低层是由与各文件系统相关的函数集构成的，并且是由 VFS 数据结构进行调用的。

所有与文件系统独立的资料都位于 fs/*.c 文件中。它们涉及如下的问题：

- 管理缓冲寄存器（buffer.c）；
- 对 fcntl()和 ioctl()系统调用作出响应（fcntl.c 和 ioctl.c）；
- 在 inode 和缓冲区上映射管道和 fifo（fifo.c，pipe.c）；
- 管理文件 - 和 inode - 表（file_table.c，inode.c）；
- 锁定和解锁文件和记录（lock.c）；
- 将名称映射到 inode（namei.c，open.c）；
- 实现错综复杂的 select()函数（select.c）；
- 提供信息（stat.c）；
- 加载和卸载文件系统（super.c）；
- 使用 exec()执行可执行程序以及转储核心程序（exec.c）；
- 加载各种二进制格式（bin_fmt*.c，如上面所述）。

而 VFS 接口则由一组相对比较高层次的操作组成，并从与文件系统独立的代码中调用而实际上是由每种文件系统类型执行的。最为相关的数据结构是 inode_operations 和 file_operations，尽管它们不是独自存在的：同样存在着其它一些数据结构。它们都定义在 include/linux/fs.h 文件中。

到实际文件系统的内核入口点是数据结构 file_system_type。file_system_types 的一个数组包含在 fs/filesystems.c 中，并且每当发出了一个加载(mount)命令时都会引用它。然后，相应 fs 类型的函数 read_super 就负责填写结构 super_block 的一个项，而该项又内嵌了结构 super_struct 和结构 type_sb_info。前者为当前的 fs 类型提供了指向一般 fs 操作的指针，而后者对相应 fs 类型内嵌了特定的信息。

NEW 文件系统类型数组已经转换成了一个链表，以允许用内核模块的形式加载新的 fs 类型。函数(un-)register_filesystem 代码包含在 fs/super.c 中。

一个文件系统类型的快速剖析

一个文件系统类型的任务是执行用于映射相应高层 VFS 操作到物理介质（磁盘、网络等等）的低层任务。VFS 接口有足够的灵活性来支持传统的 Unix 文件系统和外来的象 msdos 和 umsdos 文件系统类型。

每一个 fs 类型除了它自己的源代码目录以外，是由下列各项组成的：

- file_systems[]数组中的一个条目（项） （fs/filesystems.c）；
- 超级块（superblock）的 include 文件（include/linux/type_fs_sb.h）；
- i 节点（inode）的 include 文件（include/linux/type_fs_i.h）；
- 普通自己专用的 include 文件（include/linux/type_fs.h）；
- include/linux.fs.h 中的两行#include，以及在结构 super_block 和 inode 中的条目。

对于特定 fs 类型自己的目录，包含有所有的实际代码、inode 和数据的管理程序。

MORE本手册中有关 procfs 的章节，揭示了所有有关那种 fs 类型的低层代码和 VFS 接口。在阅读过那个章节之后，fs/procfs 中的源代码就显得非常容易理解了。

现在我们来观察 VFS 机制的内部工作情况，并以 minix 文件系统的代码作为一个实际例子。我选择 minix 类型是因为它比较短小但却是完整的；而且，Linux 中的所有其它的 fs 类型都衍生于它。在最近 Linux 安装中的事实上的标准文件系统类型 ext2，要比它复杂得多，对 ext2 这个文件系统的探索就留给聪明的读者作为一个练习了。

当一个 minix-fs 被加载后，minix_read_super 就会把从被加载的设备中读取的数据添入 super_block 数据结构中。此时，该结构中的 s_op 域将保留有一个指向 minix_sops 的指针，该指针将被一般文件系统代码用于分派超级块的操作。

在全局系统树结构中链接新加载的 fs 依赖于下列各数据项(假设 sb 是超级块数据结构，而 dir_i 是指向加载点的 inode 的指针）：

- sb->s_mounted 指向被加载文件系统的根目录 i 节点（MINIX_ROOT_INO）；
- dir_i->i_mount 保存有 sb->s_mounted;
- sb->s_covered 保存有 dir_i

卸载操作将最终通过 do_umount 来执行，而它会依次调用 minix_put_super。

每当访问一个文件时，minix_read_inode 就会开始执行；它会使用 minix_inode 各字段中的数据填写系统范围的 inode 数据结构。inode->i_op 字段是依照 inode->i_mode 来填写的，它将负责该文件的任何其它操作。上述 minix 函数的代码可以从 fs/minix/inode.c 中找到。

inode_operations 数据结构是用于把 inode 操作分派给特定 fs 类型的内核函数；该数据结构的第一项是一个指向 file_operations 项的指针，它等同于数据管理的 i_op。minix 文件系统类型允许有 inode 操作集中的三种方式（用于目录、文件和符号链接）和文件操作集中的两种（符号链接不需要文件操作）。

目录操作(仅 minix_readdir)位于 fs/minix/dir.c 中；文件操作(读 read 和写 write)位于 fs/minix/file.c 中而符号操作（读取并跟随着链）位于 fs/minix/symlink.c。

minix 源代码目录中的其余部分用于实现以下任务：

- bitmap.c 用于管理 i 节点与块的分配和释放（而 ext2 文件系统却有两个不同的代码文件）；

- fsynk.c 用于 fsync()系统调用--它管理直接、间接和双重间接块（我假定你是知道这些术语的，因为这是 Unix 的普通知识）；
- namei.c 内嵌有所有与名字有关的 i 节点的操作，比如象节点的创建和消除、重命名和链接；
- truncate.c 执行文件的截断操作。

控制台驱动程序（console driver）

作为大多数 Linux 系统上的主要 I/O 设备，控制台驱动程序是应该受到某些关注的。有关控制台和其它字符驱动程序的源代码可以在 drivers/char 中找到，当我们指称文件时，我们将使用这个特定的目录。

控制台的初始化是由 tty_io.c 中的 tty_init()函数来执行的。这个函数仅仅涉及取得每个设备集的主设备号并调用每个设备集的 init 函数。而 con_init()则是与控制台相关的函数，并存在于 console.c 中。

NEW 在内核 1.1 的开发中，控制台的初始化已经有了很大的变化。console_init()已经从 tty_init()中脱离出来了，并且是由../../main.c 直接调用的。现在虚拟控制台是动态分配的，其代码也已有了很大的变化。所以我将跳过初始化、分配等等的详细讨论。

文件操作是如何分派给控制台的

这一节是相当底层的讨论，你可以放心地跳过本节。

毫无疑问，Unix 设备是通过文件系统来访问的。本节将详细描述从设备文件到实际控制台函数的所有步骤，而且，以下的信息是从内核的 1.1.73 源代码中抽取来的，它与 1.0 的代码可能少许有点不同。

当打开一个设备 i 节点时，在../../fs/devices.c 中的 chrdev_open()函数（或者是 blkdev_open()，但我只专注于字符设备）将被执行。这个函数是通过数据结构 def_chr_fops 取得的，而它又是被 chrdev_inode_operations 引用的，是被所有文件系统类型使用的（见前面有关文件系统的部分）。

chrdev_open 通过在当前操作中替换具体设备的 file_operations 表并且调用特定的 open()函数来管理指定的设备操作的。具体设备的表结构是保存在数组 chrdevs[]中的，并由主设备号作为索引，位于同一个../../devices.c 中。

如果该设备是一个 tty 类型的（我们不是只关注控制台吗？），我们就来讨论 tty 的设备驱动程序，它们的函数在 tty_io.c 之中，由 tty_fops 作为索引。这样，tty_open()就会调用 init_dev()，而 init_dev()就会根据次设备号为设备分配任何所需的数据结构。

次设备号也用于检索已经使用 tty_register_driver()注册登记过的设备的实际驱动程序。而且，该驱动程序仍是另一个用于分派计算的数据结构，正如 file_ops 一样；它是与设备的写操作和控制有关的。最后一个用于管理 tty 的数据结构是线路规程，这将在后面叙述。控制台（以及任何其它的 tty 设备）的线路规程是由 initialize_tty_struct()设置的，并由 init_dev 调用的。

在这一节中我们所涉及的所有事情都是与设备无关的，仅有与特定控制台相关的是 console.c，在 con_init()操作期间已经注册了自己的驱动程序。相反，线路规程是与设备无关的。

MORE The tty_driver 数据结构在<linux/tty_driver.h>中有着完整的描述。

NEW 上述信息是从 1.1.73 源代码中取得的。它是有可能与你的内核有所不同的（“如信息有所变动将不另行通知”）。

控制台写操作

当往一个控制台设备进行写操作时，就会调用 con_write 函数。这个函数管理所有控制字符和换码字符序列，这些字符给应用程序提供全部的屏幕管理操作。所实现的换码序列是 vt102 终端的；这意味着当你使用 telnet 连接到一台非 Linux 主机时，你的环境变量应该有 TERM=vt102；然而，对于本地操作最佳的选择是设置 TERM=console，因为 Linux 控制台提供了一个 vt102 功能的超集。

因而，con_write()主要是由转换语句组成的，用于处理每一次一个字符的有限长状态自动换码序列的解释。在正常方式下，所打印的字符是使用当前属性直接写到显示内存中的。在 console.c 中，数据结构 vc 的所有域使用宏都是可访问的，所以（例如）任何对 attr 的引用，只要 currcons 是所指的控制台的号码，确实是引证了数据结构 vc_cons[currcons] 中的域。

NEW 实际上，新内核中的 vc_cons 已不再是一个数据结构数组了，现在它是指针的数组，其内容是用 kmalloc()操作的。宏的使用大大地简化了代码修改的工作，因为许多代码都不需要被重写。

控制台内存到屏幕内存的实际映射和非映射是由函数 set_scrmem()（它把控制台缓冲区中的数据拷贝到显示内存中）和 get_srcmem()（它把数据拷贝回控制台缓冲区中）执行的。为了减少数据传输的次数，当前控制台的私有缓冲区是物理地映射到实际显示 RAM 上的。这意味着 console.c 中的 get-和 set-_scrmem()是静态的，并且仅在一个控制台转换期间才被调用。

控制台读操作

控制台读操作是由线路规程来完成的。Linux 中默认的（也是唯一的）线路规程被称为 tty_ldisc_N_TTY。线路规程也就是“通过一线路约束输入”。它是另一个函数表（我们已习惯了这种方法，不是吗？），它是有关于设备读操作的。在 termios 标志的帮助下，线路规程也即是从 tty 上控制输入的规程：未处理过的数据、cbreak 和计划的方式；select()；ioctl()等等。

线路规程中的读（read）函数称为 read_chan()，它读取 tty 的缓冲区而不管数据是从哪里来的。原因是通过一个 tty 来到的字符是由异步硬件中断管理的。

MORE 线路规程 N_TTY 也同样在 tty_io.c 中，尽管以后出的内核都使用一个不同的 n_tty.c 源程序。

控制台输入的最底层是键盘管理的一部分，因此它是在 keyboard.c 的 keyboard_interrupt()中处理的。

键盘管理

键盘管理简直是一场噩梦。它限于文件 keyboard.c 中，里面充满了表示不同厂家键盘的各个键码的十六进制数。

我将不对 keyboard.c 进行深入讨论，因为其中没有与内核研究者有关的相关信息。

MORE 对于那些对 Linux 的键盘编程确实感兴趣的人，最好的方法是从 keyboard.c 的最后一行往回看起。最底层的细节是在该文件的上半部分。

转换当前控制台

当前控制台是通过使用函数 change_console()来转换的，它位于 tty_io.c 中由 keyboard.c 和 vt.c 调用（前者响应按键的控制台转换，后者是当一个程序通过引用一个 ioctl()调用时转换控制台）。

实际的转换过程是分两步来执行的，函数 complete_change_console()处理其中的第二部分。转换的分裂意味着在一个与控制着我们正在离开的 tty 的进程的可能的握手以后完成任务。如果控制台不在进程控制之下，change_console()就会自己调用 complete_change_console()。进程需要足够的能力来成功地完成从图形到文本控制台或从文本到图形控制台的转换，并且 X 服务器（例如）是其图形控制台的控制进程。

选择机制

“选择（selection）”是 Linux 文本控制台的剪切（cut）与粘贴（paste）功能。这个技巧主要是由用户级的进程来处理的，它可以用 selection 或 gpm 的具体例子说明。用户级的程序在控制台上使用 ioctl()通知内核来加亮显示屏幕的一个区域。然后，被选择的文本被拷贝到一个选择缓冲区。该缓冲区是 console.c 中的一个静态实体。粘贴文本操作是通过“手工地”将字符放入 tty 输入队列中完成的。整个选择机制是通过#ifdef 受到保护的，所以用户在内核配置期间可以禁用它以节省几千字节的内存。

选择是一个非常低级的功能，因而它工作是任何其它内核活动所看不见的。这意味着许多的#ifdef 只是屏幕在以任何方式作修改之前简单地移动加亮部分。

NEW 新内核特性改善了选择的代码，鼠标指针的加亮可以与被选择的文本独立（内核 1.1.23 或更高）。而且，从 1.1.73 版起，被选择的文本使用了动态的缓冲区而不是静态的了，使得内核小了 4KB。

使用 ioctl()操作设备

ioctl()系统调用是用户进程控制设备文件行为的入口点。Ioctl 管理是从../../fs/ioctl.c 中产生的,实际上 sys_ioctl()就是在这个 ioctl.c 中的。标准的 ioctl 请求就是在那里执行的，其它与文件相关的请求是由 file_ioctl()处理的（在同一个源文件中），而其它任何请求都分派给特定设备的 ioctl()函数。

控制台设备的 ioctl 资料是位于 vt.c 中的，因为控制台驱动程序要将 ioctl 请求分派给 vt_ioctl()。

NEW 上述信息是关于内核 1.1.7x 的。1.0 内核是没有“驱动程序”表的,而且 vt_ioctl()是直接由 file_operations()表指向的。

Ioctl 的资料确实是相当让人混淆的。有些请求是与设备相关的，而有些却是与线路规程相关的。我将试图对 1.0 和 1.1.7x 内核之间发生的任何事概要总结一下。

1.1.7x 系列内核有如下的特性：tty_ioctl.c 只实现了线路规程请求（也就是 n_tty_ioctl()，这是唯一在 n_tty.c 外面的 n_tty 函数），而 file_operations 字段指向 tty_io.c 中的 tty_ioctl()。如果请求号没有被 tty_ioctl()解析出来，它就会被传到 tty->driver.ioctl 或者，如果它失败时，就到 tty->ldisc.ioctl。控制台的与驱动程序相关的资料可以从 vt.c 中找到，而线路规程方面的资料则在 tty_ioctl.c 中。

在 1.0 内核中，tty_ioctl()是在 tty_ioctl.c 中的并有一般 tty 的 file_operations 所指向。未被解析出的请求将用与 1.1.7x 相似的方法被传送到特定的 ioctl 函数或到线路规程代码去。

注意，在这两种情况中，TIOCLINUX 请求是在与设备无关的代码中的，这暗示着控制台选择操作可以通过 ioctl 对任何 tty 进行操作来设置（set_selection()总是在控制台前台

上操作的），而这是一个安全上的漏洞。这也是转移到一个更新的内核的很好理由，在新内核中，通过仅允许超级用户来处理选择弥补了这个漏洞。

有很多请求可以被发给控制台设备，而知道它们的最好方法是浏览源程序文件 vt.c。