第十三章　Linux 启动系统

操作系统的启动过程既让人好奇，又让人费解。我们知道，没有操作系统的计算机是无法使用的，那么是谁把操作系统装入到内存，是操作系统自己吗？这显然是一个先有鸡还是先有蛋的问题，幸好，固化在 ROM（PC 机）中的 BIOS 帮了大忙，可以说，BIOS 是整个启动过程的先锋。实际上，尽管一台计算机的启动过程仅仅是昙花一现，但它并不简单。本章将讨论基于 i386 平台的操作系统的启动过程，其相关的保护模式的知识请参见第二章。

13.1　初始化流程

每一个操作系统都要有自己的初始化程序，Linux 也不例外。那么，怎样初始化？我们首先看一下初始化的流程。

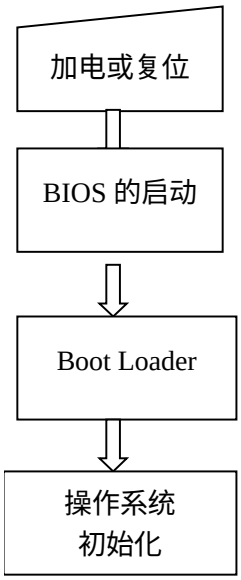

图 13.1　初始化流程

图 13.1 中的加电或复位这一项代表操作者按下电源开关或复位按钮那一瞬间计算机完成的工作。BIOS 的启动是紧跟其后的基于硬件的操作，它的主要作用就是完成硬件的初始化，稍后还要对 BIOS 进行详细的描述。BIOS 启动完成后，Boot Loader 将读操作系统代码，然后由操作系统来完成初始化剩下的所有工作。

13.1.1 系统加电或复位

当一台装有 Intel 386 CPU 的计算机系统的电源开关或复位按钮被按下时，通常所说的冷启动过程就开始了。中央处理器进入复位状态，它将内存中所有的数据清零，并对内存进行校验，如果没有错误，CS 寄存器中将置入 FFFF[0]，IP 寄存器中将置入 0000[0]，其实，这个 CS:IP 组合指向的是 BIOS 的入口，它将作为处理器运行的第一条指令。系统就是通过这个方法进入 BIOS 启动过程的。

13.1.2 BIOS 启动

BIOS 的全名是基本输入输出系统（Basic Input Output System）。它的主要任务是提供 CPU 所需的启动指令。刚才提到了，计算机进入复位状态后，内存被自动清零，CPU 此时是无法获得指令的。计算机的设计者们当然考虑到了这一点，因此，他们预先编好了供系统启动使用的启动程序，把它们存放在 ROM 中，并安排它到一个固定的位置，即 FFFF:0000，CPU 就从 BIOS 中获得了启动所需的指令集。该指令集除了完成硬件的启动过程以外，还要将软盘或硬盘上的有关启动的系统软件调入内存。

让我们看一看 BIOS 中启动程序的主要任务:首先是上电自检（POST Power-On Self Test），然后是对系统内的硬件设备进行监测和连接，并把测试所得的数据存放到 BIOS 数据区，以便操作系统在启动时或启动后使用，最后，BIOS 将从软盘或硬盘上读入 Boot Loader，到底是从软盘还是从硬盘启动要看 BIOS 的设置，如果是从硬盘启动，BIOS 将读入该盘的零柱面零磁道上的 1 扇区（MBR），这个扇区上就存放着 Boot Loader，该扇区的最后一个字存放着系统标志，如果该标志的值为 AA55，BIOS 在完成硬件监测后会把控制权交给 Boot Loader。

除了启动程序以外，BIOS 还提供一组中断以便对硬件设备的访问。我们知道，当键盘上的某一键被按下时，CPU 就会产生一个中断并把这个键的信息读入，在操作系统没有被装入以前（如 Linux 的 Bootsect.S 还没有被读入）或操作系统没有专门提供另外的中断响应程序的情况下，中断的响应程序就是由 BIOS 提供的。

这里介绍一个具体的 BIOS 系统，它的上电自检（POST）程序包含 14 个项目，具体内容如表 13.1 所示，执行过 POST 后，该系统将调入硬盘上的 Boot Loader。

表 13.1　　POST 程序包含的 14 个项目

序号	相应内容	序号	相应内容
1	CPU 处理器内部寄存器测试	8	键盘复位和测试
2	32K RAM 存储器测试	9	键盘复位和测试
3	DMA 控制器测试	10	附加 RAM 存储器测试
4	32K RAM 存储器测试	11	其他包含在系统中的 BIOS 测试
5	CRT 视频接口测试	12	软盘设备测试
6	8259 中断控制器测试	13	硬盘设备测试

7	8253 定时器测试	14	打印机接口和串行接口测试

BIOS 中的中断程序、BIOS 数据区中的信息这里就不作详细介绍了，如果你想进一步了解，请查阅相关资料，在本章后面的内容中，只会对所要涉及到的部分 BIOS 内容进行详细解释。

13.1.3　Boot Loader

Boot Loader 通常是一段汇编代码，存放在 MBR 中，它的主要作用就是将系统启动代码读入内存，有关这方面的内容相当复杂，这里请你先记住它的功能，至于详细情况，比如，怎样把系统读入，后面将会介绍。

13.1.4　操作系统的初始化

这部分实际上是初始化的关键。Boot Loader 将控制权交给操作系统的初始化代码后，操作系统所要完成的存储管理、设备管理、文件管理、进程管理等任务的初始化必须马上进行，以便进入用户态。其实不管是单任务的 DOS 操作系统还是这里介绍的多任务 Linux 操作系统，当启动过程完成以后，系统都进入用户态，等待用户的操作命令。而 Linux 要到达这个状态是相当复杂的一件工作，这一章主要就是围绕这部分内容写而。

13.2　初始化的任务

13.2.1　处理器对初始化的影响

每个操作系统都是基于计算机的硬件设备的，不管它的设计，实现，还是特性，都要依赖于一定的硬件。所有的硬件中，中央处理器（CPU）对它的影响最大。我们知道，Linux 是一个可以运行于多个不同平台的操作系统，但这并不意味着它可以抛开不同种类计算机的硬件特性。事实上，Linux 是靠在不同机器上运行不同的代码来实现跨平台特性的。Linux 巧妙地把与设备相关的代码按照设备型号分类安排，以便在编译时把对应的部分编入内核。如果你看过了在/usr/sur/arch 目录下组织的源文件，就会发现，所有 Intel 386 相关的代码在一个子目录下，而与 spark 相关的代码在另一个子目录下。代码在编译时会得到关于平台的信息，根据这个信息，编译器决定到底包含哪一段代码。

以上这些是为了说明一点，即操作系统必须支持硬件设备特别是 CPU 的特性，反过来说，硬件设备也决定操作系统的特性。

具有代表性的 Intel 80386 处理器大家一定不会陌生，它支持多任务并发执行，它的结构和机能完全是为此设计的。根据对 80386 保护模式的了解（详见第二章），我们可以看出，操作系统根据 80386 提供的机制，对计算机的资源（主存储器空间、执行时间及外围设备）进行分配和保护。通过把这些资源分配给系统中的各个任务，并对资源进行保护，是所有的

任务得以有效的运行直至完成。80386 的存储管理及保护机制，保护（系统中的）每一个任务不被另外的任务破坏。例如，操作系统通过使用存储管理机制，保证分配给不同任务的存储区互不重叠（共享存储区域除外）；通过使用保护机制，保证系统中任何一个用户任务都不能访问分配给操作系统的存储区域。

请注意，80386 提供保护机制，也提供段页式的两层内存管理，但在操作系统初始化之前，它却运行于一个既不支持保护机制，也不支持页机制的实模式下的。在这个模式下，根本没法实现多任务并发处理，所以，在一个要求实现多任务并发处理的操作系统的初始化程序中，就必须加入使 80386 进入保护模式的代码。这就是处理器对启动任务影响的一个例子。

13.2.2 其他硬件设备对处理器的影响

除了处理器以外，许多硬件设备也对初始化产生影响，刚才介绍的 BIOS 就在很大程度上影响初始化的步骤。另外，每加入一种新的硬件设备，为了它能被正常使用，你必须在操作系统中对它进行配置（PC 机的标准配置设备除外）。你或许要提到 Windows 的即插即用技术，事实上，即插即用设备是由操作系统自动完成配置的，而不是不需要配置，所以，如果你编制的是 Windows 的初始化程序，那么你需要在你的代码中加入支持自动配置即插即用设备的代码。

硬件对初始化的影响并不仅仅局限于这些方面，由于硬件在计算机系统中的中心地位，它对初始化的的影响是从始至终的。

13.3 Linux 的 Boot Loader

从这里开始，我们将对 Linux 的启动过程进程分析。首先要介绍的是 Boot Loader，因为 Boot Loader 比较复杂，并且与启动密不可分。但是要想了解 Boot Loader 的内在机制，必须从了解磁盘结构开始。

13.3.1 软盘的结构

软盘是由一个引导扇区，一个管理块（比如说 MS 的 FAT 或 EXT2 的 inode）和一个基本数据区的形式组织在一起的。由于管理块是与特定文件系统相关的，我们也可以把它归为数据区的一部分，软盘的结构如图 13.2 所示。

如果你想了解有关 inode 或数据区的相关知识，请参看第八章和第九章。在这里，我们主要关心的是引导扇区。

引导扇区中存放着用于启动的代码，以及一些有关特定文件系统的信息，在它的最后，存放着一个启动标志，如果它是 0xAA55，代表这个引导扇区是可用于启动的。

MS-DOS 的引导记录及记录内各个项目的偏移如图 13.3 所示，Linux 引导记录跟它大体一致，只是没有磁盘参数表。

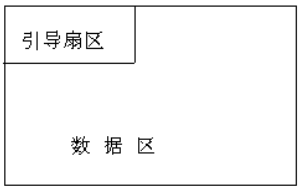

图 13.2　软盘的结构

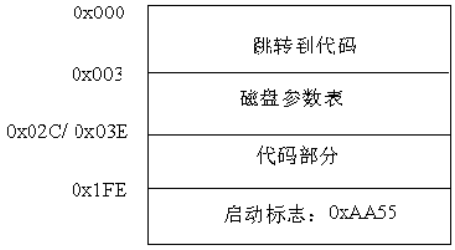

图 13.3　软盘引导区

13.3.2　硬盘的结构

硬盘相对于软盘来说要复杂一些，一个硬盘在 DOS 的文件系统下可被分为 4 个基本分区（Pramiray Partition）。如果分区数目达不到所需，那么可以把一个基本分区定义为一个扩展分区(Extended Partition)，然后再把这个基本分区分为一个或多个逻辑分区。整个硬盘有一张分区表，它存放在硬盘的第一个扇区（MBR）里，而每个扩展分区也对应一个分区表，它存放在该扩展分区对应的第一个扇区里。图 13.4 是一张硬盘分区层次图。

同样地，我们也要介绍硬盘的引导扇区内容，不过，硬盘的分区机制使它的引导扇区也有了几个层次。这里，我们先看一下整个硬盘的主引导扇区（MBR），它的结构与软盘的结构是很相似的，如图 13.5 所示。

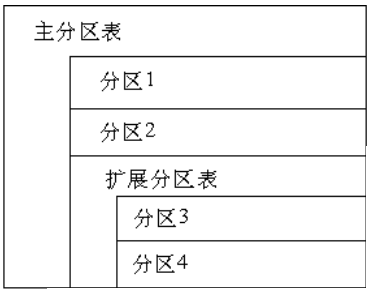

图 13.4　硬盘分区层次图

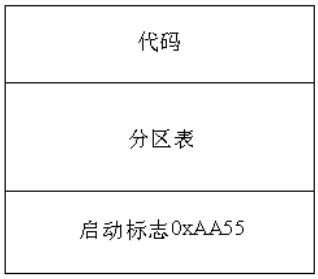

图 13.5　硬盘引导扇区

除了主引导扇区外，每个基本分区和扩展分区也有自己的引导扇区，它们的结构与主引导扇区是相同的。而逻辑分区的引导扇区通常不能用于启动。如果你使用 DOS 下的 FDISK 程序，你就会发现逻辑分区不能设置为 active 状态，也就是说 DOS 不能从该分区启动。

13.3.3　Boot Loader

如前所述，在启动的过程中，BIOS 会把 Boot Loaer 读入内存，并把控制权交给它。MBR（硬盘启动）或软盘的启动扇区（软盘启动）内的代码就是 Boot Loader 或者 Boot Loader 的一部分。Boot Loader 的实现是很复杂的，这也是我们什么要了解磁盘结构的原因。

实际上 Boot Loader 的来源有多种，最常见的一种是你的操作系统就是 DOS，而 Boot

Loader 是 DOS 系统提供的 MS-Boot Loader。这种情况下比较简单：如果是软盘启动，Boot Loader 会检查盘上是否存在两个隐含的系统文件（IBMBIO.COM、IBMDOS.COM），若有，读出并送至内存中指定的区域，把控制权转移给 IBMBIO 这个模块，否则显示出错信息。如果是硬盘启动，Boot Loader 将查找主分区表中标记为活动分区的表项，把该表项对应的分区的引导扇区读入，然后把控制权交给该扇区内的引导程序，这段程序也可以被看作是 Boot Loader 的一部分，它完成的工作与软盘的 Boot Loader 大致相同。

有时候一台计算机上所装的操作系统并不是 DOS，或者并不仅仅是 DOS。在这种情况，如果你是一个 Linux 的使用者，那么，你的计算机上现在就需要两套操作系统了（Linux 和 Windows），于是启动碰上了一个新问题——怎么能引导多个系统?

仅仅 MS-DOS 的 Boot Loader 无法完成这种工作，你所要的是一个可以多重启动的工具，怎么办？幸运的是，有很多用来实现这一功能的软件（大部分是共享或自由软件）已经被编制了，如在 DOS 环境下启动 Linux 的 LOADLIN，Linux 下最常用的 LILO 等等，使用它们，你可以方便地从各种操作系统启动。不幸的是，你是一个操作系统的研究者，很可能你会需要编写自己的 Boot Loader 程序，所以，你必须了解这个程序的工作原理。由于 LILO 的强大功能和方便使用的特性，很多人都在使用它，因此我们将在这里详细介绍 LILO。

13.3.4 LILO

LILO 是一个在 Linux 环境编写的 Boot Loader 程序(所以安装和配置它都要在 Linux 下)。它的主要功能就是引导 Linux 操作系统的启动。但是它不仅可以引导 Linux，它还可以引导其他操作系统，如 DOS，Windows 等等。它不但可以作为 Linux 分区的引导扇区内的启动程序，还可以放入 MRB 中完全控制 Boot Loadr 的全过程。下面让我们看看几种典型情况下硬盘的主引导扇区和各个分区的引导扇区内程序的内容。

(1) 计算机上只装了 DOS 一个操作系统

这种情况和刚才介绍的 DOS 硬盘启动相对应，如图 13.6 所示。

图 13.6 只有 DOS 的硬盘分区图

(2) 计算机上装了 DOS 和 Linux 操作系统，Linux 由 LOADIN 启动，如图 13.7 所示。

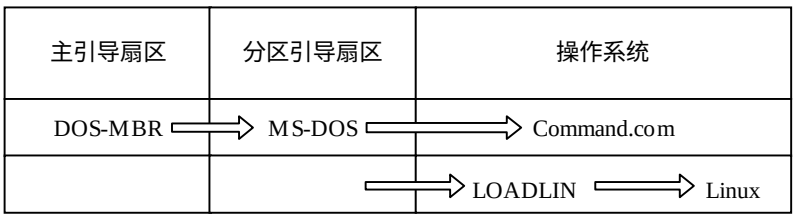

图 13.7　用 LOADLIN 从 DOS 下启动

Linux 在这种情况下，DOS 的主引导区没有发生变化，分区的引导扇区也没有变化，只是在 DOS 的配置文件 Autoexec.bat 中加入了 Loadin 程序而已。

(3) LILO 存放在 Linux 分区的引导扇区内，如图 13.8 所示。

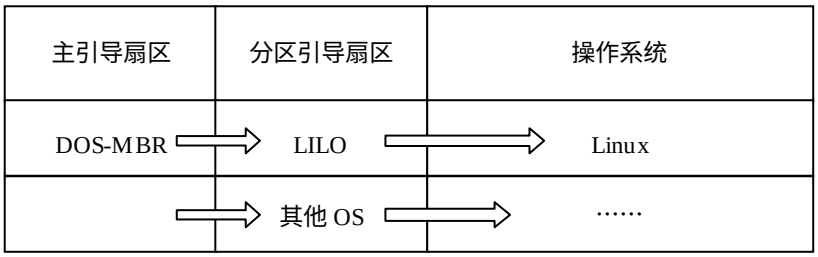

图 13.8　LILO 存放在 Linux 分区的引导扇区内

在这种情况下，LILO 存放在硬盘上的一个基本分区内。如果希望从 Linux 启动，必须把 Linux 分区设为活动分区。而如果想使用 Windows，就必须把 Windows 所在的分区激活，然后重新启动以进入 Windows，也就是说，你没办法在启动的时候选择从哪个操作系统进入，这样的多重启动显得太麻烦。回想 Windows 下的多重启动，你只要在引导时输入一个 F3 键，便能自动进入 DOS 6.22，这才是我们所希望的方式。Windows 能做到，LILO 当然可以做到，它还能做得更好（LILO 不仅允许你选择从哪个系统引导，它还允许你给 Linux 的内核传递参数）请看下面这种模式，如图 13.9 所示。

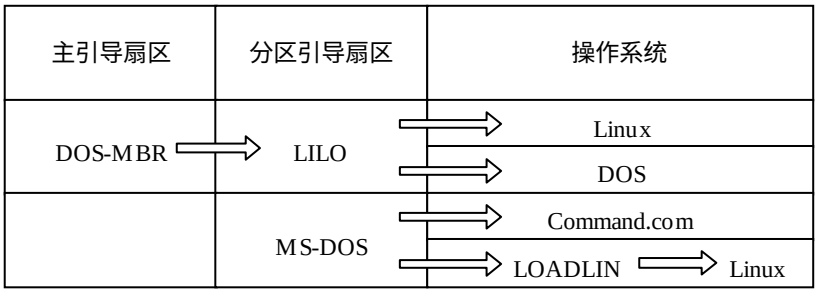

图 13.9　LILO 在分区引导扇区内的多重引导

无论从哪个分区引导，你都可以选择是进入 Linux 或是 DOS，不过从 DOS 分区启动时，如果你不想进入 Linux，你需要单步执行 autoexec.bat 以跳过 LOADLIN。而从 Linux 分区启动时，你仅需要在启动时敲键盘输入操作系统的名字（这个名字可以由你在配置 LILO 时自己设定）便可以选择进入哪个操作系统。这张表完全是用于说明 LILO 安装位置的，其实你可能已经看出来了，既然无论从哪个分区都能进入所有的操作系统，那么，只要有一个活动分区就够了，从方便的角度来讲，从 Linux 分区启动是个不错的选择。

(4) LILO 放在硬盘的主引导扇区里。

如图 13.10 所示。LILO 如果在安装时选择作为 MRB，它将负责 Boot Loader 的全过程，不过这样做有一定的风险，因为它将覆盖 MBR，有可能使你原来的系统无法启动，所以你需

要先备份主引导扇区。

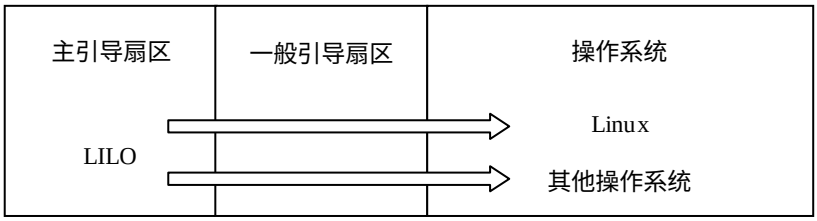

图 13.10　LILO 放在主引导扇区里

LILO 的功能实际上是由几个程序共同实现的，它们是：

① Map Installer：这是 LILO 用于管理启动文件的程序。它可以将 LILO 启动时所需的文件放置到合适的位置(这些文件的位置由 LILO 本身决定)并且记录下这些位置,以便 LILO 访问。其实,当运行/sbin/lilo 这个程序时,Map installer 就已经工作了,它将 Boot loader 写入引导分区（原来的 Boot Loader 将被备份），创建记录文件——map file 以映射内核的启动文件。每当内核发生变化时（比如说内核升级了），你必须运行/sbin/lilo 来保证系统的正常运行。

② Boot Loader：这就是由 BIOS 读入内存的那部分 LILO 的程序，它负责把 Linux 的内核或其他操作系统的引导分区读入内存。另外，Linux 的 Boot Loader 还提供一个命令行接口，可以让用户选择从哪个操作系统启动和加入启动参数。

③ 其他文件：这些文件主要包括用于存放 Map installer 记录的 map 文件（/boot/map）和存放 LILO 配置信息的配置文件（/etc/lilo.conf），这些文件都是 LILO 启动时必需的，它们一般存放在/boot 目录下。

LILO 在引导 Linux 的同时还可以向 Linux 的内核传送参数。前面我们提到了，LILO 提供了一个命令行解释程序，当系统加载 LILO，并在屏幕上显示了“LILO” 字样时，你可以按下 Ctrl 或者 Shift 键（不同版本的 LILO 可能有所不同，笔者的系统需要按下 Tab 键），这时会出现“LILO boot”字样，表明命令行解释程序已经被激活，可以从键盘输入了。如果相应的系统引导提示符是“Linux”，“DOS”的话，你可以键入“Linux”启动 Linux，或者键入“DOS”启动 Windows。如果选择启动 Linux，此时你还可以在“Linux”后面加入一些参数，LILO 可以把这些参数传递给内核。例如：

LILO boot: Linux 1 告诉内核按照单用户模式启动。

LILO boot: Linux ether=eth0,0x280,10 告诉内核你的第一块网卡的端口地址是 0x280，中断号是 10。

LILO 提供许多种参数，如 Debug，等等，具体这些参数和它们的作用，请你查阅 LILO 的文挡。此外，并不是所有的硬件都需要加参数才能支持的。如果硬件设备在编译内核时已经被支持了，那么完全没有必要加参数。事实上，只有那些比较特殊（也比较不常用）的设备，才需要在启动时设定参数值，明确它的端口地址和中断号，以节省大量的用于检测端口地址和中断号的启动时间。

像可以预设默认的启动选项一样，在/etc/lilo.conf 中也可以预先定义启动时要输入的参数，这样就可以避免每次启动都要重复输入。让我们看一个具体的 lilo.conf 的例子，例

子的左边是 Script 的脚本程序，右面是对程序的解释。从这个例子可以看出，lilo.conf 的编制思想，同 DOS 下的 config.sys 差不多。

```
# /etc/lilo.conf
# LILO configuration file
# gernerated by 'liloconfig'
#
# Start LILO global section      /*LILO 的通用配置块*/
append = "ether=eth0，0x280，10" /*请注意，这就是向内核传递的参数，我们把它写在这里，就可以免去每次在启动时输入的麻烦*/
 boot = /dev/hda2                    /*LILO 安装在硬盘 1 的二号分区的分区表上*/
delay = 50                                /*给用户选择从哪个操作系统启动的等待时间*/
vga = normal                         /*显示器设置为标准 VGA*/
# ramdisk = 0                             /*未安装虚拟启动盘*/
# End LILO global section           /*通用配置块结束*/
# Linux bootable partition config begins      /*用于启动 Linux 的配置块*/
root = /dev/hda2                     /*Linux 的根文件系统安装在硬盘 1 的二号分区上*/
image = /vmlinuz                  /*选择根目录下的 vmlinuz 作为内核*/
lable = linux                      /*启动选择的标识符为 linux*/
image = /zimage-2.4.18           /*在引导 Linux 时，可以选择多个内核。比如说我们编译了一个新的内核，并想从它启动，只需把这行程序写在这里，当然，别忘了先运行 LILO 来改变 Map 文件。*/
label = Newkernel
read-only                          /*以只读方式安装，防止启动中的误操作*/
#Linux bootable partition config ends      /*Linux 配置块结束*/
#DOS bootable partition config begens          /*用于启动 DOS 的配置块*/
other = /dev/hda1  /*该操作系统的 Boot Loader 安装在硬盘 1 的一号分区的分区表内*/
label = dos                  /*启动选择的标识符为 dos*/
table = /dev/hda           /*该操作系统的根目录在硬盘 1 的一号分区上*/
# Dos bootab le partion config ends   /*Dos 配置块结束*/
```

13.3.5　LILO 的运行分析

我们知道了 LILO 怎么安装，包含什么东西，有什么功能，但 LILO 到底是怎么运行的呢？下面是代码分析层次的 LILO 运行过程，通过介绍这个过程，希望你能对整个 Boot Loader 这部分内容有一个深入的认识。

1．从软盘启动

Linux 内核可以存入一张 1.44MB 的软盘中，这样做的前提是对“Linux 内核映像”进行压缩，压缩是在编译内核时进行的，而解压是由装入程序在引导时进行的。

当从软盘引导 Linux 时，Boot Loader 比较简单，其代码在 arch/i386/boot/bootsect.S 汇编语言文件中。当编译 Linux 内核源码时，就获得一个新的内核映像，这个汇编语言文件所产生的可执行代码就放在内核映像文件的开始处。因此，制作一个包含 Linux 内核的软磁盘并不是一件困难的事。

把内核映像的开始处拷贝到软盘的第 1 个扇区就创建了一张启动软盘。当 BIOS 装入软盘的第 1 个扇区时，实际上就是拷贝 Boot Loader 的代码。BIOS 将 Boot Loader 读入至内存中物理地址 0x07c00 处，控制权转给 Boot Loader，Boot Loader 执行如下操作。

- 把自己从地址 0x07c00 移到 0x90000。
- 利用地址 0x03ff，建立“实模式”栈。
- 建立磁盘参数表，这个表由 BIOS 用来处理软盘设备驱动程序。
- 通过调用 BIOS 的一个过程显示“Loading”信息。
- 然后，调用 BIOS 的一个过程从软盘装入内核映像的 setup( )代码，并把这段代码放入从地址 0x90200 开始的地方。
- 最后再调用 BIOS 的一个过程。这个过程从软盘装入内核映像的其余部分，并把映像放在内存中从地址 0x10000 开始的地方，或者从地址 0x100000 开始的地方，前者叫做“低地址”的小内核映像（以“make zImage”进行的编译），后者叫做“高地址”的大内核映像（以“make bzImage”）进行的编译。

2．从硬盘启动

一般情况下，Linux 内核都是从硬盘装入的。BIOS 照样将引导扇区读入至内存中的 0x00007c00 处，控制权转给 Boot Loader，Boot Loader 把自身移动至 0x90000 处，并在 0x9B000 处建立堆栈（从 0x9B000 处向 0x9A200 增长），将第 2 级的引导扇区读入至内存的 0x9B000 处，把控制权交给它。在引导扇区移动之后，将显示一个大写的 L 字符，而在启动第 2 级的引导扇区之前，将显示一个大写的 I 字符。如果读入第 2 级的引导扇区的过程有错误，屏幕上的 LI 之后会显示一个十六进制的错误号。

二级引导扇区内的代码将把描述符表读入至内存中的 0x9D200 处，把包含有命令行解释程序的扇区读入至内存的 0x9D600 处。接着，二级引导扇区将等待用户的输入，不管这时用户输入了一个选择还是使用缺省配置，都将把对应的扇区读入至内存的 0x9D600（覆盖命令行解释程序的空间），把生成的启动参数保存在 0x9D800 处。

如果用户定义了用于启动的 RAM 盘的话，这部分文件将被读入到物理内存的末尾。如果你的内存大于 16MB 的话，它会被读入至 16MB 内存的结尾，这是因为 BIOS 程序不支持对 16MB 以上内存的访问（它用于寻址的指令中只有 24 位的地址描述位）。并且它开始于一个新的页，以便于启动后系统把它所占的内存回收到内存池。

接下来，操作系统的初始化代码将被读入到内存的 0x90200 处。而系统的内核将被读入到 0x10000 处。如果该内核是以 make bzImage 方式编译的，它将被读入到内存的 0x100000 处。在读入的过程中，存放 map 文件的扇区被读入至内存的 0x9D000 处。

如果读入的 image 是 Linux 的内核，控制权将交给处于 0x90200 的 Setup.S。如果读入的是另外的操作系统，过程要稍微麻烦一点:chain loader 被读入到内存的 0x90200 处。该系统用于启动的扇区被读入到 0x90400。chain loader 将把它所包含的分区表移到 0x00600 处，把引导扇区读入到 0x07c00。做完这一切，它把控制权交给引导扇区。

第 2 级引导扇区在得到控制权以后马上显示一个大写的 L 字符。读入命令行解释程序后显示一个大写的 O 字符。

图 13.11 是 LILO 运行完后，内存的分布情况。

0x0000		1982 字节
0x007BE	分区表	64 字节
0x007FE		29K 字节
0x07C00	MBA 或软盘的引导扇区	512 字节
0x7E00		32.5K 字节
0x10000	内核	448K 字节
0x90000	软盘引导扇区	512 字节
0x90200	Linux 的 Setup.s	39.5K 字节(其中占用了 2K 字节)
0x9A000	主引导扇区	512 字节
0x9A200	堆栈	3.5K 字节
0x9B000	第 2 级引导扇区	8K 字节，占用了 2.5K 字节
0x9D000	Map 信息	512 字节
0x9D200	描述符表	1K 字节
0x9D600	命令行参数	512 字节
0x9D800	键盘信息交换区	512 字节
0x9DA00	启动参数	1K 字节
0x9DC00		7.5K 字节
0xA0000	driver swapper	1K 字节

图 13.11　LILO 运行后的计算机内存分布情况

13.4　进入操作系统

Boot Loader 作了这么多工作，一言以蔽之，只是把操作系统的代码调入内存，所以，当它执行完后，自然该把控制权交给操作系统，由操作系统的启动程序来完成剩下的工作。上面已经提到了，LILO 此时把控制权交给了 Setup.S 这段程序。该程序是用汇编语言编写的 16 位启动程序，它作了些什么呢？

13.4.1　Setup.S

首先，Setup.S 对已经调入内存的操作系统代码进行检查，如果没有错误（所有的代码都已经被调入，并放至合适的位置），它会通过 BIOS 中断获取内存容量信息，设置键盘的响应速度，设置显示器的基本模式，获取硬盘信息，检测是否有 PS/2 鼠标，这些操作，都是在 386 的实模式下进行的，这时，操作系统就准备让 CPU 进入保护模式了。当然，要先屏蔽中

断信号，否则，系统可能会因为一个中断信号的干扰而陷入不可知状态，然后再次设置 32 位启动代码的位置,这是因为虽然预先对 32 位启动程序的存储位置有规定,但是 Boot Loader (通常是 LILO)有可能把 32 位的启动代码读入一个与预先定义的位置不同的内存区域，为了保证下一个启动过程能顺利进行，这一步是必不可少的。

完成上面的工作后，操作系统指令 lidt 和 lgdt 被调用了，中断向量表（idt）和全局描述符表（gdt）终于浮出水面了，此时的中断描述符表放置的就是开机时由 BIOS 设定的那张表，gdt 虽不完善，但它也有了 4 项确定的内容，也就是说，这里已经定义了下面 4 个保护模式下的段。

```
(1) .word   0，0，0，0       ! 系统所定义的 NULL 段
(2) .word   0，0，0，0       ! 空段，未使用
(3) .word   0xFFFF  ! 4Gb   (0x100000*0x1000 = 4Gb) 大小的系统代码段
    .word   0x0000       !base address=0
    .word   0x9A00       ! 可执行代码段
    .word   0x00CF       !粒度=4096
(4) .word   0xFFFF       ! 4Gb (0x100000*0x1000 = 4Gb) 大小的系统数据段
    .word   0x0000       ! base address=0
    .word   0x9200       !可读写段
    .word   0x00CF       !粒度=4096
```

注意：这里关于段描述符的格式请看第二章图 2.10 及相关内容。

在实模式下，还有几件事要作，如下所述。

我们需要对 8259 中断控制器进行编程，当然，这很简单，让它们的功能与标准的 PC 相一致就可以了，这已经在第四章进行了介绍，在此不准备再进行详细介绍。

此外，协处理器也需要重新复位。这几件事做完以后，Setup.S 设置保护模式的标志位，重新取指令以后，再用一条跳转指令：

```
jmpi    0x100000, KERNEL_CS
```

进入保护模式下的启动阶段，同时把控制权交给 Head.S 这段纯 32 位汇编代码。

13.4.2 Head.S

Head.S 也要先做一些屏蔽中断一类的准备工作,然后,它会对中断向量表做一定的处理：用一个默认的表项把所有的 256 个中断向量填满。这个默认表项指向一个特殊的中断服务程序，事实上，该程序什么都不做。为什么这样呢？这是因为，在 Linux 系统初始化完成后，BIOS 的中断服务程序是不会再被使用了。Linux 采用了很完善的设备驱动程序使用机制，该机制使特定硬件设备的中断服务程序很容易被系统本身或用户直接调用，而且，调用时所需的参数通常都要比 BIOS 调用来得简单。由于设备驱动程序是专门针对一个设备的，所以，它所包含的中断服务程序在功能上通常也比 BIOS 中断要完善，所以，BIOS 的中断向量在这里就被覆盖了。启动处在这个阶段，第一，还不需要开启中断，第二，相应的设备驱动程序还未被加载，所以把中断向量表置空显然是个合理的选择。事实上，直到初始化到了最后的阶段（start_kernel()被调用后），中断向量表才被各个中断服务程序重新填充。

Boot Loader 读入内存中的启动参数和命令行参数，这些参数在启动过程中是必须的数据资料，他们不但在启动的过程中要使用，在启动后也是必须的，Head.S 把它们保存在 empty_zero_page 页中，这就涉及到了页机制，之所以先映射这个页，是因为以后运行的程序可能会占用启动参数和命令行参数的内存区，所以要先保存它们。

Head.S 此后会检查 CPU 的类型：虽然 Intel 系列 CPU 保持兼容，但无疑后继产品会有很多新的特性和功能（如奔腾芯片支持 4MB 大小的页）。作为一个操作系统，Linux 当然要对他们作出相应的支持并发挥他们更优越的处理能力，此外，由于 Intel 已成为业界标准，所以也有必要把与之兼容的设备归入一类（如 AMDK6 就与 Intel 奔腾芯片兼容）。这里只是对处理器类型进行判断，在 Start_kernel()中要根据这里的结果对系统进行设置。此外，还要对协处理器进行检查，80387 与 80487 当然也该区别对待。

下面是一个非常重要的初始化步骤——页初始化。

Head.S 调用了 Setup_paging 这个子函数，这个函数只对 4MB 大小的内存空间进行了映射,剩下的部分在 start_kernel()中分配,这个函数并不复杂,它先把从线性地址 0xC0000000 处开始的两个大小为 4KB 的内存空间映射为两个页：一个为页目录表——swap_page_dir，另一个为零页——pg0，他们分别指向物理地址的 0x1000 和 0x2000。此后设置页目录表的属性（可读写、存在、任意特权级可访问），并使它的表项指向为内核分配的 4MB 空间上的各个页。在 Head.S 里定义了几个比较特殊的页，除了上面提到的 swap_page_dir，pg0 以外，还有 empty_bad_page 、 empty_bad_page_table 、 empty_zero_page 等等。刚才提到了 empty_zero_page，这里详细介绍一下这个页，以便对页机制及页初始化有更进一步的认识。

empty_zero_page 这个页存放的是系统启动参数和命令行参数，他们各占 2KB 大小，即各占半页。该页的具体内容如下：

```
偏移量    数据类型            描述
------    ----      -----------
    0     32 bytes       一段屏幕显示信息
2    unsigned shortEXT_MEM_K,  extended memory size in Kb (BIOS 的 15 号
中断得到的内存大小)
 0x20     unsigned shortCL_MAGIC,  命令行标志数 (=0xA33F)
 0x22     unsigned shortCL_OFFSET,  经计算所得的命令行程序的偏移地址,
                0x90000 + contents of CL_OFFSET
              (只在 CL_MAGIC = 0xA33F 时计算)
 0x40     20 bytes  APM_BIOS_INFO 结构体
 0x80     16 bytes  hd0-disk-parameter  (硬盘 1 参数，由 BIOS 的 40 号中断得到)
 0x90     16 bytes  hd1-disk-parameter  (硬盘 1 参数，由 BIOS 的 46 号中断得到)

 0xa0     16 bytes       系统描述符表截为 16 字节的信息
              ( sys_desc_table_struct 结构)
 0xb0 - 0x1df       未占用
0x1e0     unsigned long  ALT_MEM_K，可变化的内存大小
0x1f1     char      setup.s 所占的扇区数
0x1f2     unsigned shortMOUNT_ROOT_RDONLY (if !=0)
0x1f4     unsigned short压缩内核占用的空间
0x1f6     unsigned shortswap_dev (unused AFAIK) (交换分区)
0x1f8     unsigned shortRAMDISK_FLAGS (虚盘的标志)
0x1fa     unsigned shortVGA-Mode
0x1fc     unsigned shortORIG_ROOT_DEV (high=Major,  low=minor) (主设备的从
```

```
设备号)
    0x1ff    char     AUX_DEVICE_INFO
    0x200    short          jump to start of setup code aka "reserved" field.
    0x202    4 bytes       SETUP-header 的标志， ="HdrS"
                 目前的版本是 0x0241...
    0x206    unsigned shortheader 的标志
    0x208    8 bytes        setup.S 获得 boot loaders 信息的内存区
    0x210    char     LOADER_TYPE = 0， 老式引导程序，
                 否则由 boot loader 自己设置
                 0xTV: T=0 ： LILO
                      1 ： Loadlin
                      2 ： bootsect-loader
                      3 ： SYSLINUX
                      4 ： ETHERBOOT
                      V = 引导程序版本号
    0x211    char     loadflags（调入标志）:
                 bit0 = 1: 读入高内存区 (bzImage)
                 bit7 = 1: boot loader 设置了堆和指针。
    0x212    unsigned short(setup.S)
    0x214    unsigned long KERNEL_START， loader 从何处调入内核
    0x218    unsigned long INITRD_START， 调入内存的 image 的位置
    0x21c    unsigned long INITRD_SIZE， 虚盘上的 image 的大小
    0x220    4 bytes       (setup.S)
    0x224    unsigned shortsetup.S 的堆和指针
    0x226 - 0x7ff      setup.S 的程序体

    0x800    string， 2K max    命令行参数
```

这个页中的内容是从启动的多个程序中总结出来的，从中我们可以看出，页的大小为0x800～4KB，页内的数据按照一定的偏移量顺序排列，对页的读写操作也要通过这些偏移量来完成。在你分析 Linux 的启动时，通过和这张表的对照，希望你能对在这种汇编语言环境中准确的定位读写及对页与段的安排，有一个更明确的认识。

此外，你应该发现，物理地址的 0x0000 没有被页映射，这部分空间保存着中断向量表，全局描述符表等系统的比较重要的数据结构，他们不必要进行页交换，而且，对线性空间的访问页机制也完全不能察觉，所以，没有必要用页来映射这个空间。这样做还有另外一层用意：系统中凡是无效的指针（NULL），都可以自动对应到这个空间。

关于页初始化更详细的内容请参见第六章。那么，段机制在 Head.S 中有没有变化呢？我们知道，一旦系统进入保护模式，那么，系统的多任务特性就完全可以体现了。这里，段机制的多任务特性，当然要进一步地体现出来了。下面是 Head.S 中定义的全局描述符表：

```
ENTRY(gdt_table)
     .quad 0x0000000000000000        /* NULL descriptor */
     .quad 0x0000000000000000        /* not used */
     .quad 0x00cf9a000000ffff        /* 0x10 kernel 4GB code at 0x00000000 */
     .quad 0x00cf92000000ffff        /* 0x18 kernel 4GB data at 0x00000000 */
     .quad 0x00cffa000000ffff        /* 0x23 user   4GB code at 0x00000000 */
     .quad 0x00cff2000000ffff        /* 0x2b user   4GB data at 0x00000000 */
     .quad 0x0000000000000000        /* not used */
     .quad 0x0000000000000000        /* not used */
     /*
```

```
 * The APM segments have byte granularity and their bases
 * and limits are set at run time.
 */
.quad 0x0040920000000000        /* 0x40 APM set up for bad BIOS's */
.quad 0x00409a0000000000        /* 0x48 APM CS    code */
.quad 0x00009a0000000000        /* 0x50 APM CS 16 code (16 bit) */
.quad 0x0040920000000000        /* 0x58 APM DS    data */
.fill NR_CPUS*4,8,0              /* space for TSS's and LDT's */
```

关于这部分内容的详细解释请参见 2.3 节。有了新的全局描述符表和中断向量表，当然要重新装入描述符，所以 lgdt，lidt，又被执行了一遍，而以前预取的指令当然要重取，各个寄存器也要重新赋值，核心的堆栈自然也要重置。此外，虽然此时并没有用户以和任务，但还是可以用 lldt 让局部描述符表的寄存器指向空，以便初始化的顺利进行。

到这一步，保护机制下内存管理，中断管理的框架已经建好了，下一步，就是怎么样具体实现操作系统的功能了，于是，Head.S 调用/init/main.C 中的 start_kernel 函数，把控制权交给启动的下一部分代码。

13.5　main.c 中的初始化

head.s 在最后部分调用 main.c 中的 start_kernel（）函数，从而把控制权交给了它。所以启动程序从 start_kernel（）函数继续执行。这个函数是 main.c 乃至整个操作系统初始化的最重要的函数，一旦它执行完了，整个操作系统的初始化也就完成了。

如前所述，计算机在执行 start_kernel（）前处已经进入了 386 的保护模式，设立了中断向量表并部分初始化了其中的几项，建立了段和页机制，设立了 9 个段，把线性空间中用于存放系统数据和代码的地址映射到了物理空间的头 4MB，可以说我们已经使 386 处理器完全进入了全面执行操作系统代码的状态。但直到目前为止，我们所做的一切可以说都是针对 386 处理器所做的工作，也就是说几乎所有的多任务操作系统只要使用 386 处理器，都需要作这一切。而一旦 start_kernel（）开始执行，Linux 内核的真实面目就一步步地展现在你的眼前了。start_kernel（）执行后，你就可以以一个用户的身份登录和使用 Linux 了。

让我们来看看 start_kernel 到底做了些什么，这里，我们通过介绍 start_kernel（）所调用的函数，来讨论 start_kernel（）的流程和功能。

我们仿照 C 语言函数的形式来进行这种描述，不过请注意，真正的 start_kernel（）函数调用子函数并不象我们在下面所写的这样简单，毕竟这本书的目的是帮助你深入分析 Linux。我们只能给你提供从哪儿入手和该怎么看的建议，真正深入分析 Linux，还需要你自己来研究代码。start_kernel（）这个函数是在/init/main.c 中，这里也只是将 main.c 中较为重要的函数列举出来。

```
start_kernel ()            /*定义于 init/main.c */
{
……
setup_arch () ;
}
```

它主要用于对处理器、内存等最基本的硬件相关部分的初始化，如初始化处理器的类型

（是在 386，486，还是 586 的状态下工作，这是有必要的，比如说，Pentium 芯片支持 4MB 大小的页，而 386 就不支持），初始化 RAM 盘所占用的空间（如果你安装了 RAM 盘的话）等。其中，setup_arch（）给系统分配了 intel 系列芯片统一使用的几个 I/O 端口的地址。

```
paging_init（）；    /*该函数定义于 arch/i386/mm/init.c */
```

它的具体作用是把线性地址中尚未映射到物理地址上的部分通过页机制进行映射。这一部分在本书第六章有详细的描述，在这里需要特别强调的是，当 paging_init（）函数调用完后，页的初始化就整个完成了。

```
trap_init（）;/*该函数在 arch/i386/kernel/traps.c 中定义*/
```

这个初始化程序是对中断向量表进行初始化，详见第四章。它通过调用 set_trap_gate（或 set_system_gate 等）宏对中断向量表的各个表项填写相应的中断响应程序的偏移地址。

事实上，Linux 操作系统仅仅在运行 trap_init（）函数前使用 BIOS 的中断响应程序（我们这里先不考虑 V86 模式）。一旦真正进入了 Linux 操作系统，BIOS 的中断向量将不再使用。对于软中断，Linux 提供一套调用十分方便的中断响应程序，对于硬件设备，Linux 要求设备驱动程序提供完善的中断响应程序,而调用使用多个参数的 BIOS 中断就被这些中断响应程序完全代替了。

另外，在 trap_init（）函数里，还要初始化第一个任务的 LDT 和 TSS，把它们填入 Gdt 相应的表项中。第一个任务就是 init_task 这个进程，填写完后，还要把 init_task 的 TSS 和 LDT 描述符分别读入系统的 TSS 和 LDT 寄存器。

```
init_IRQ（）    /* 在 arch/i386/kernel/irq.c 中定义*/
```

这个函数也是与中断有关的初始化函数。不过这个函数与硬件设备的中断关系更密切一些。

我们知道 intel 的 80386 系列采用两片 8259 作为它的中断控制器。这两片级连的芯片一共可以提供 16 个引脚，其中 15 个与外部设备相连，一个用于级连。可是，从操作系统的角度来看，怎么知道这些引脚是否已经使用;如果一个引脚已被使用，Linux 操作系统又怎么知道这个引脚上连的是什么设备呢?在内核中，同样是一个数组（静态链表）来纪录这些信息的。这个数组的结构在 irq.h 中定义：

```
struct irqaction {
    void (*handler) (int, void *, struct pt_regs *);
      unsigned long flags;
    unsigned long mask;
    const char *name;
    void *dev_id;
  struct irqaction *next;}
```

具体内容请参见第四章。我们来看一个例子：

```
static void math_error_irq (int cpl, void *dev_id, struct pt_regs *regs)
  {
      outb (0, 0xF0);
      if (ignore_irq13 || !hard_math)
          return;
      math_error ();
  }
static struct irqaction irq13 = { math_error_irq, 0, 0, "math error", NULL, NULL };
```

该例子就是这个数组结构的一个应用，这个中断是用于协处理器的。在 init_irq（）这

个函数中，除了协处理器所占用的引脚，只初始化另外一个引脚，即用于级连的 2 引脚。不过，这个函数并不仅仅做这些，它还为两片 8259 分配了 I/O 地址，对应于连接在管脚上的硬中断，它初始化了从 0x20 开始的中断向量表的 15 个表项（386 中断门），不过，这时的中断响应程序由于中断控制器的引脚还未被占用，自然是空程序了。当我们确切地知道了一个引脚到底连接了什么设备，并知道了该设备的驱动程序后，使用 setup_x86_irq 这个函数填写该引脚对应的 386 的中断门时，中断响应程序的偏移地址才被填写进中断向量表。

sched_init（） /*在/kernel/sched.c 中定义*/

看到这个函数的名字可能令你精神一振，终于到了进程调度部分了，但在这里，你非但看不到进程调度程序的影子，甚至连进程都看不到一个，这个程序是名副其实的初始化程序：仅仅为进程调度程序的执行做准备。它所做的具体工作是调用 init_bh 函数（在 kernel/softirq.c 中）把 timer、tqueue、immediate 三个任务队列加入下半部分的数组。

time_init（）/*在 arch/i386/kernel/time.c 中定义*/

时间在操作系统中是个非常重要的概念。特别是在 Linux、UNIX 这些多任务的操作系统中它更是作为主线索贯穿始终，之所以这样说，是因为无论进程调度（特别是时间片轮转算法）还是各种守护进程（也可以称为系统线程，如页表刷新的守护进程）都是根据时间运作的。可以说，时间是他们运行的基准。那么，在进程和线程没有真正启动之前，设定系统的时间就是一件理所当然的事情了。

我们知道计算机中使用的时间一般情况下是与现实世界的时间一致的。当然，为了避开 CIH，把时间跳过每月 26 号也是种明智的选择。不过如果你在银行或证交所工作，你恐怕就一定要让你计算机上的时钟与挂在墙上的钟表分秒不差了。还记得 CMOS 吗？计算机的时间标准也是存在那里面的。所以，我们首先通过 get_cmos_time（）函数设定 Linux 的时间，不幸的是，CMOS 提供的时间的最小单位是秒，这完全不能满足需要，否则 CPU 的频率 1 赫兹就够了。Linux 要求系统中的时间精确到纳秒级，所以，我们把当前时间的纳秒设置为 0。

完成了当前时间的基准的设置，还要完成对 8259 的一号引脚上的 8253（计时器）的中断响应程序的设置，即把它的偏移地址注册到中断向量表中去。

parse_options（）　/*在 main.c 中定义*/

这个函数把启动时得到的参数如 debug、init 等从命令行的字符串中分离出来，并把这些参数赋给相应的变量。这其实是一个简单的词法分析程序。

console_init（） /*在 linux/drivers/char/tty_io.c 中定义*/

这个函数用于对终端的初始化。在这里定义的终端并不是一个完整意义上的 TTY 设备，它只是一个用于打印各种系统信息和有可能发生错误的出错信息的终端。真正的 TTY 设备以后还会进一步定义。

kmalloc_init（） /*在 linux/mm/kmalloc.c 中定义*/

kmalloc 代表的是 kernel_malloc 的意思，它是用于内核的内存分配函数。而这个针对 kmalloc 的初始化函数用来对内存中可用内存的大小进行检查，以确定 kmalloc 所能分配的内存的大小。所以，这种检查只是检测当前在系统段内可分配的内存块的大小，具体内容参见第六章内存分配与回收一节。

下面的几个函数是用来对 Linux 的文件系统进行初始化的，为了便于理解，这里需要把 Linux 的文件系统的机制稍做介绍。不过，这里是很笼统的描述，目的只在于使我们对初始化的解释工作能进行下去，详细内容参见第八章的虚拟文件系统。

虚拟文件系统是一个用于消灭不同种类的实际文件系统间（相对于 VFS 而言，如 ext2、fat 等实际文件系统存在于某个磁盘设备上）差别的接口层。在这里，您不妨把它理解为一个存放在内存中的文件系统。它具体的作用非常明显：Linux 对文件系统的所有操作都是靠 VFS 实现的。它把系统支持的各种以不同形式存放于磁盘上或内存中（如 proc 文件系统）的数据以统一的形式调入内存，从而完成对其的读写操作。（Linux 可以同时支持许多不同的实际文件系统，就是说，你可以让你的一个磁盘分区使用 Windows 的 FAT 文件系统，一个分区使用 UNIX 的 SYS5 文件系统，然后可以在这两个分区间拷贝文件）。为了完成以及加速这些操作，VFS 采用了块缓存，目录缓存（name_cach）、索引节点（inode）缓存等各种机制，以下的这些函数，就是对这些机制的初始化。

inode_init（） /*在 Linux/fs/inode.c 中定义*/

这个函数是对 VFS 的索引节点管理机制进行初始化。这个函数非常简单：把用于索引节点查找的哈希表置入内存，再把指向第一个索引节点的全局变量置为空。

name_cache_init（） /*在 linux/fs/dcache.c 中定义*/

这个函数用来对 VFS 的目录缓存机制进行初始化。先初始化 LRU1 链表，再初始化 LRU2 链表。

Buffer_init（）/*在 linux/fs/buffer.c 中定义*/

这个函数用来对用于指示块缓存的 buffer free list 初始化。

mem_init（） /* 在 arch/i386/mm/init.c 中定义*/

启动到了目前这种状态，只剩下运行/etc 下的启动配置文件。这些文件一旦运行，启动的全过程就结束了，系统也最终将进入我们所期待的用户态。现在，让我们回顾一下，到目前为止，我们到底做了哪些工作。

其实，启动的每一个过程都有相应的程序在屏幕上打印与这些过程相应的信息。我们回顾一下这些信息，整个启动的过程就一目了然了。

当然，你的计算机也许速度很快，你甚至来不及看清这些信息，系统就已经就绪，即“Login：”就已经出现了。不要紧，登录以后，你只要打一条 dmesg | more 命令，所有这些信息就会再现在屏幕上。

【Loading ……】出自 bootsect.S ，表明内核正被读入。

【uncompress ……】很多情况下，内核是以压缩过的形式存放在磁盘上的，这里是解压缩的过程。

下面这部分信息是在 main.c 的 start_kernel 函数被调用时显示的。

【Linux version 2.2.6 （root@lance）（gcc version 2.7.2.3）】Linux 的版本信息和编译该内核时所用的 gcc 的版本。

【Detected 199908264 Hz processor】调用 init_time（）时打出的信息。

【Console:colour VGA+ 80x25，1 virtaul console（max 63）】调用 console_init（）打出的信息。初始化的终端屏幕使用彩色 VGA 模式，最大可以支持 63 个终端。

【Memory: 63396k/65536k available（848k kernel code，408k reserved， 856k data）】调用 init_mem（）时打印的信息。内存共计 65536KB，其中空闲内存为 63396KB，已经使用的内存中，有 848KB 用于存放内核代码，404KB 保留，856KB 用于内核数据。

【VFS:Diskquotas version dquot_6.4.0 initialized】调用 dquote_init（）打出的信息。quota 是用来分配用户磁盘定额的程序。关于这个程序请参看第八章。

以下是对设备的初始化：

```
【PCI: PCI BIOS revision 2.10 entry at 0xfd8d1          |
PCI: Using configuration type 1                        |
```

PCI: Probing PCI hardware 】调用 pci_init（）函数时显示的信息。

```
【Linux NET4.0 for Linux 2.2
Based upon Swansea University Computer Society NET3.039
NET4: UNIX domain sockets 1.0 for Linux NET4.0.
NET4: Linux TCP/IP 1.0 for NET4.0
```

IP Protocols: ICMP，UDP，TCP】调用 socket_init（）函数时打印的信息。使用 Linux 的 4.0 版本的网络包，采用 sockets 1.0 和 1.0 版本的 TCP/IP 协议，TCP/IP 协议中包含有 ICMP、UDP、TCP 三组协议。

```
【Detected PS/2 Mouse Port。
Sound initialization started
Sound initialization complete
Floppy drive (s) : fd0 is 1.44M
Floppy drive (s) : fd0 is 1.44M
```

FDC 0 is a National Semiconductor PC87306 】调用 device_setup（）函数时打印的信息。包括对 ps/2 型鼠标、声卡和软驱的初始化。

看完上面这一部分代码和与之相应的信息，你应该发现，这些初始化程序并没有完成操作系统的各个部分的初始化，比如说，文件系统的初始化只是初始化了几个内存中的数据结构，而更关键的文件系统的安装还没有涉及，其实，这是在 init 进程建立后完成的。下面，就是 start_kernel（）的最后一部分内容。

13.6　建立 init 进程

在完成了上面所有的初始化工作后，Linux 的运行环境已经基本上完备了。此时，Linux 开始逐步建立进程了。

13.6.1　init 进程的建立

Linux 将要建立的第一个进程是 init 进程，建立该进程是以调用 kernel_thread（init，NULL，0）这个函数的形式进行的。init 进程是很特殊的——它是 Linux 的第 1 个进程，也是其他所有进程的父进程。让我们来看一下它是怎样产生的。

在调用 kernel_thread（init，NULL，0）函数时，会调用 main.c 中的另外一个函数——init（）。请注意 init（）函数和 init 进程是不同的概念。通过执行 inin（）函数，系统完成了下述工作。

- 建立 dbflush、kswapd 两个新的内核线程。
- 初始化 tty1 设备。该设备对应了多个终端（concole），用户登录时，就是登录在这些终端上的。
- 启动 init 进程。Linux 首先寻找“ /etc/init ”文件，如果找不到，就接着找“ /bin/init ”文件，若仍找不到，再去找“ /sbin/init ”。如果仍无法找到，启动将无法进行下去。否则，

便执行 init 文件，从而建立 init 进程。

当 etc/init（假定它存在）执行时，建立好的 init 进程将根据启动脚本文件的内容创建其它必要的进程去完成一些重要的操作。

（1）文件系统检查。

（2）启动系统的守护进程。

（3）对每个联机终端建立一个“ getty ”进程。

（4）执行“ /etc/rc ”下的命令文件。

此后，“ getty ”会在每个终端上显示“ login ”提示符，以等待用户的登录。此时“ getty ”会调用“ exec ”执行“ login ”程序，“ login ”将核对用户帐户和密码，如果密码正确，“ login ”调用“ exec ”执行 shell 的命令行解释程序（当然，也可以执行 X Windows 如果用户设置了的话）。shell 接着去执行用户默认的系统环境配置脚本文件（通常是用户的 home 目录下的 profile 文件）。

init 还有另外一个任务，当某个终端或虚拟控制台上的用户注销之后，init 进程要为该终端或虚拟控制台重新启动一个 “ getty ”，以便能够让其他用户登录。这是为什么呢？你应该发现，当用户登录时，“ getty ”用的是“ exec ”而不是“ fork ”系统调用来执行“ login ”，这样，“ login ”在执行的时候会覆盖“ getty ”的执行环境（同理，用户注册成功后，“ login ”的执行环境也会被 shell 占用）。所以，如果想再次使用同一终端，必须再启动一个“ getty ”。

此外，init 进程还负责管理系统中的“ 孤儿 ”进程。如果某个进程创建子进程之后，在子进程终止之前终止，则子进程成为孤儿进程。init 进程负责“ 收养 ”该进程，即孤儿进程会立即成为 init 进程的子进程。这是为了保持进程树的完整性。

init 进程的变种较多，大多数 Linux 的发行版本采用 sysvinit（由 Miquel van Smoorenburg）。这是一个编译好的软件包。由于 System V 而得名。UNIX 的 BSD 版本有不同的 init，主要区别在于是否具有运行级别（关于运行级别的问题下面会有专门的描述）：System V 有运行级别，而 BSD 没有运行级别。但这种区别并不是本质的区别。

图 5.12 是上述流程的流程图。

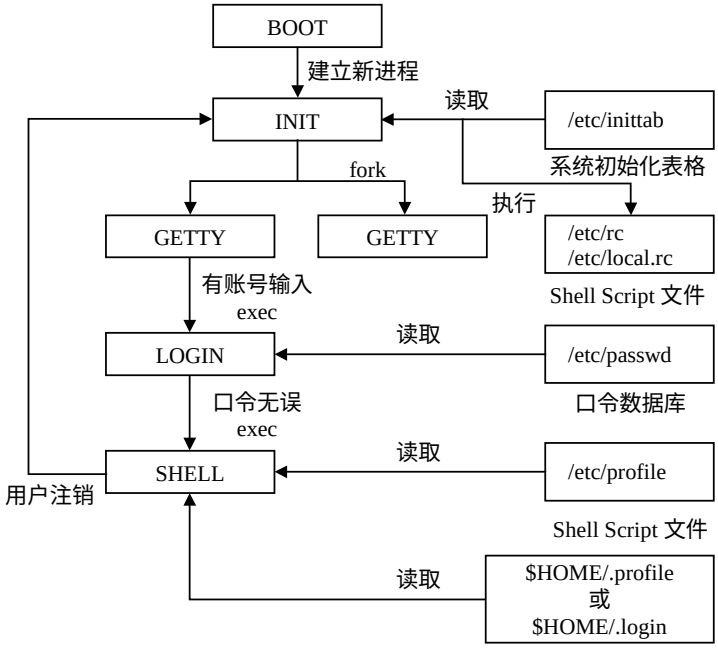

图 13.12　init 进程的启动流程

13.6.2　启动所需的 Shell 脚本文件

在启动的过程中，多次用到了 Shell 的脚本文件——Shell Script，如“$HOME/profile”、“/etc/inittab”等等。这里有必要把它们的格式和作用稍加说明。

我们把启动所需要的脚本文件分为两部分，一部分是 Linux 系统启动所必需的，也就是从/etc/inittab 开始直到出现“Login:”提示符时要用到的所用脚本，另外一部分是用户登录后自己设定的用于支持个性化的操作环境的脚本。在后者中，我们可以设定提示符用“$”或是其他什么任意你喜欢的字符，可以设所用的 Shell 是 bash、ksh，还是 zsh。显然，这部分不是我们的重点，我们要重点描述的是前一部分——系统启动所必需的脚本。

系统启动所必需的脚本存放在系统默认的配置文件目录/etc 下。用一条 ls 指令你可以看到所用的配置文件。不过，/etc 下面还有一些子目录，比如说，rc.d 就是启动中非常重要的一部分。我们主要介绍的是/etc/inittab 和 rc.d 下的一些文件，我们还是按启动时 init 进程调用它们的顺序来一一介绍。

首先调用的是/etc/inittab。init 进程将会读取它并依据其中所记载的内容进入不同的启动级别，从而启动不同的进程。所谓运行级别就是系统中定义了许多不同的级别，根据这些级别，系统在启动时给用户分配资源。比如说，以系统管理员级别登录的用户，就拥有使用几乎所有系统资源的权力，而一般用户显然不会被赋予如此大的特权。

下面是系统的 7 个启动级别。

0　系统停止。如果在启动时选择该级别，系统每次运行到 inittab 就会自动停止，无法启动。

1　单用户模式。该模式只允许一个用户从本地计算机上登录，该模式主要用于系统管理员检查和修复系统错误。

2　多用户模式。与 3 级别的区别在于用于网络的时候，该模式不支持 NFS（网络文件系统）。

3　完全多用户模式。可以支持 Linux 的所有功能，是 Linux 安装的默认选项。

4　未使用的模式。

5　启动后自动进入 X Windwos。

6　重新启动模式。如果在启动时选择该级别，系统每次运行到 inittab 就会自动重新启动，无法进入系统。

以上这 7 种模式并非在启动后一成不变，如果用户是系统管理员，那么就可以使用“ init X ”（X 取 1、2……6 中的一个）来改变运行状态。另外，如果重新启动时，也可以看到显示器上的提示，当前运行状态已经变为 6。

让我们看一个 inittab 文件的实例。

```
id:3:initdefault:                    系统默认模式为 3。
#System initialization.
si::sysinit:/etc/rc.d/rc.sysinit     无论从哪个级别启动，都执行/etc/rc.d/rd.sysinit。
10:0:wait:/etc/rc.d/rc.0             从 0 级别启动，将运行 rc.0。
11:1:wait:/etc/rc.d/rc.1             从 1 级别启动，将运行 rc.1。
12:2:wait:/etc/rc.d/rc.2             从 2 级别启动，将运行 rc.2。
13:3:wait:/etc/rc.d/rc.3             从 3 级别启动，将运行 rc.3。
14:4:wait:/etc/rc.d/rc.4             从 4 级别启动，将运行 rc.4。
15:5:wait:/etc/rc.d/rc.5             从 5 级别启动，将运行 rc.5。
16:6:wait:/etc/rc.d/rc.6             从 6 级别启动，将运行 rc.6。
#Things to run in every runlevel任何级别都执行的配置文件。
ud::once:/sbin/update

#Run gettys in standard runlevels          对虚拟终端的初始化。
1:12345:respqwn:/sbin/mingetty tty1       tty1 运行于 1、2、3、4、5 五个级别。
2:2345:respqwn:/sbin/mingetty tty2        tty2 运行于 2、3、4、5 四个级别。
3:2345:respqwn:/sbin/mingetty tty3        tty3 运行于 2、3、4、5 四个级别。
4:2345:respqwn:/sbin/mingetty tty4        tty4 运行于 2、3、4、5 四个级别。
5:2345:respqwn:/sbin/mingetty tty5        tty5 运行于 2、3、4、5 四个级别。
6:2345:respqwn:/sbin/mingetty tty6        tty6 运行于 2、3、4、5 四个级别。

#Run xdm in runlevel 5                    在级别 5 启动 X Window。
x:5:respawn:/usr/bin/X11/xdm -nodaemon
```

现在详细解释一些 inittab 的内容。

从上面的文件可以看出，inittab 的每一行分成 4 个部分，这 4 个部分的格式如下：

id:runleveld:action:process

它们代表的意义分别如下。

id：代表有几个字符所组成的标识符。在 inittab 中任意两行的标识符不能相同。

runlevels：指出本行中第 3 部分的 action 以及第 4 部分的进程会在哪些 runlevel 中被执行，这一栏的合法值有 0、1、2...…6，s 以及 S。

action：这个部分记录 init 进程在启动过程中调用进程时，对进程所采取的应答方式，合法的应答方式有下面几项。

- initdefault：指出系统在启动时预设的运行级别。上例中的第 1 行就用了这个方式。所以系统将在启动时，进入 runlevel 为 3 的模式。当然，可以把 3 改为 5，那将会执行/etc/rc.d/rc.5，也就是 X Window。
- sysinit：在系统启动时，这个进程肯定会被执行。而所有的 inittab 的行中，如果它的 action 中有 boot 及 bootwait，则该行必须等到这些 action 为 sysinit 的进程执行完之后才能够执行。
- wait：在启动一个进程之后，若要再启动另一个进程，则必须等到这个进程结束之后才能继续。
- respawn：代表这个 process 即使在结束之后，也可能会重新被启动，最典型的例子就是 getty。

明白了 inittab 的意思，让我们回过头来看看启动过程。

首先，执行的是/etc/rd.c/rc.sysinit。这里不再给出它的程序清单，只给出它的主要功能：

（1）检查文件系统

包括启用系统交换分区，检查根文件系统的情况，使用磁盘定额程序 quato（可选项），安装内核映像文件系统 proc，安装其他文件系统。

（2）设置硬件设备

设定主机名，检查并设置 PNP 设备，初始化串行接口，初始化其他设备（根据你的机器配置情况决定）。

（3）检查并载入模块

执行完 rc.sysinit 并返回 inittab 后，init 进程会根据 inittab 所设定的运行级别去执行/etc/rc.d 目录下的相应的 rc 文件。比如说运行级别为 3，相应的 rc 文件即为 rc.3。这些文件将运行不同的启动程序去初始化各个运行级别下的系统环境，这部分启动程序最重要的作用之一是启动系统的守护进程，如在 rc.3 中，就要启动 cron、sendmial 等守护进程。

做完这一步，init 进程将执行 getty 进程从而等待用户的登录，也就是说，Linux 的启动全过程已经结束了，剩下的部分，就是整个系统等待用户需求，并为用户提供服务了。

Linux 的初始化到此就结束了，回过头来看看，它确实跟我们在开始时假设的那个简单的操作系统的初始化有许多相似之处，这些是所有初始化都应该有的相似点，希望您在分析完 Linux 源代码后，能编制出自己的操作系统的初始化代码。