LINUX 内核解读

（1）

内　　容：LINUX **系统引导和初始化**

撰　　稿：刘 立

参加讨论：Linux **内核研讨小组**

修订日期：

智 能 中 心 体 系 结 构 组

Linux 的系统引导和初始化

----------Linux2.4.22 内核解读之一

一、 系统引导和初始化概述

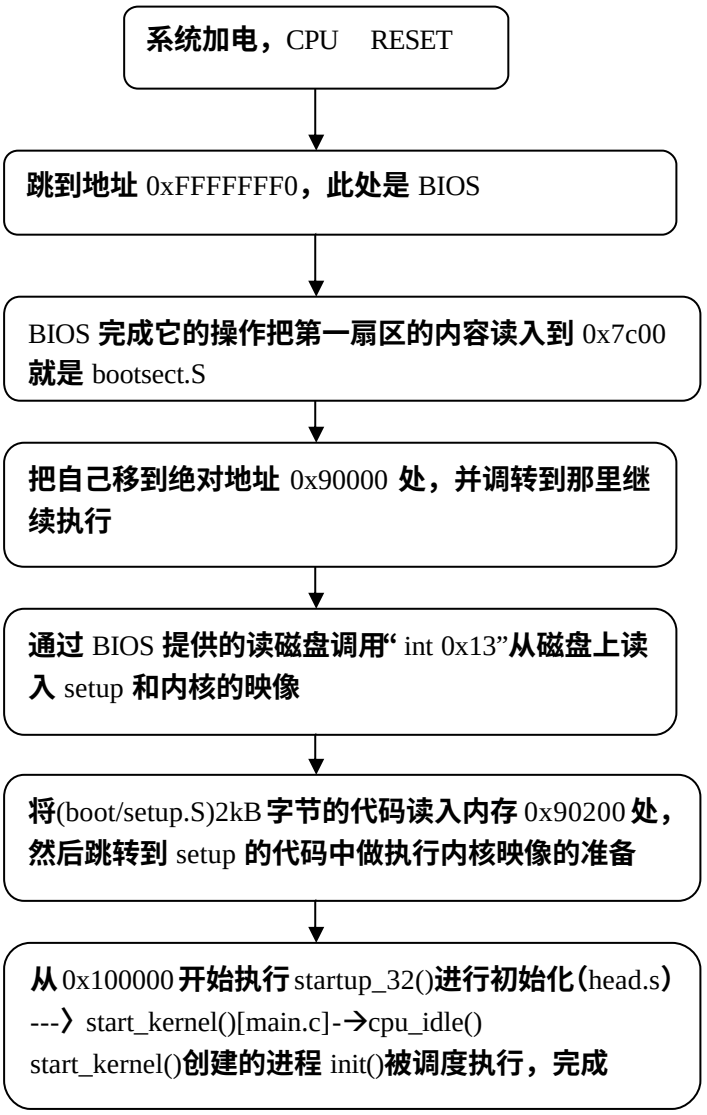
- 相关代码（引导扇区的程序及其辅助程序，以 x86 体系为例）：

|linux-2.4.22|arch|i386|boot|bootsect.S：Linux 引导扇区的源代码；512 字节
|linux-2.4.22|arch|i386|boot|setup.S：辅助程序；
|linux-2.4.22|arch|i386|boot|video.S：辅助程序，用于引导过程中的屏幕显示。
|linux-2.4.22|arch|i386|boot|compressed|head.S,
|linux-2.4.22|arch|i386|boot|compressed|misc.c ：用于对内核映像解压缩

\linux-2.4.22\arch\i386\kernel\head.s 系统初始化入口
\linux-2.4.22\init\main.c 系统初始化入口

- 参考文档：\linux-2.5.75\Documentation\i386\boot.txt

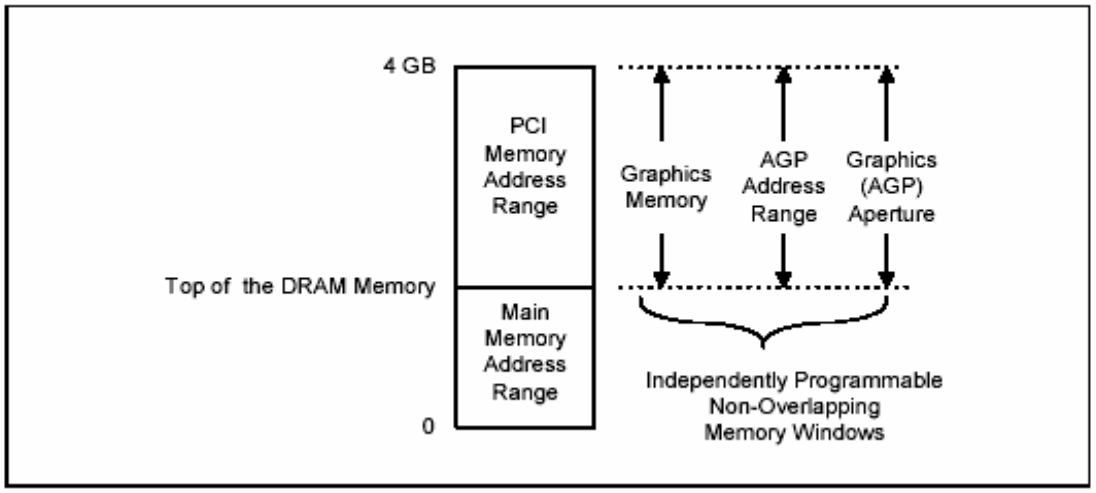
过程描述



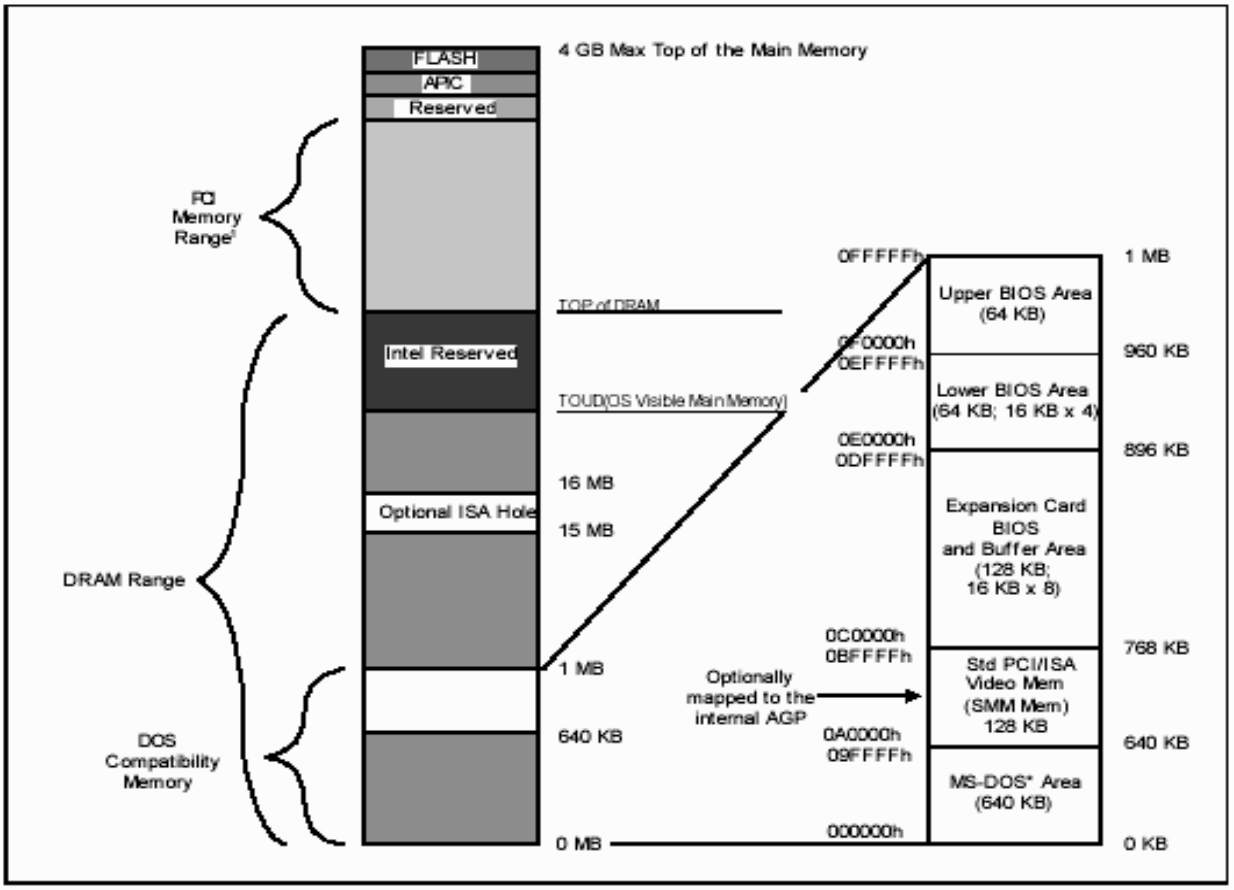
1、指令的跳转及其机理

- 80x86/Pentium 的地址映射

. Memory System Address Map



). Detailed Memory System Address Map



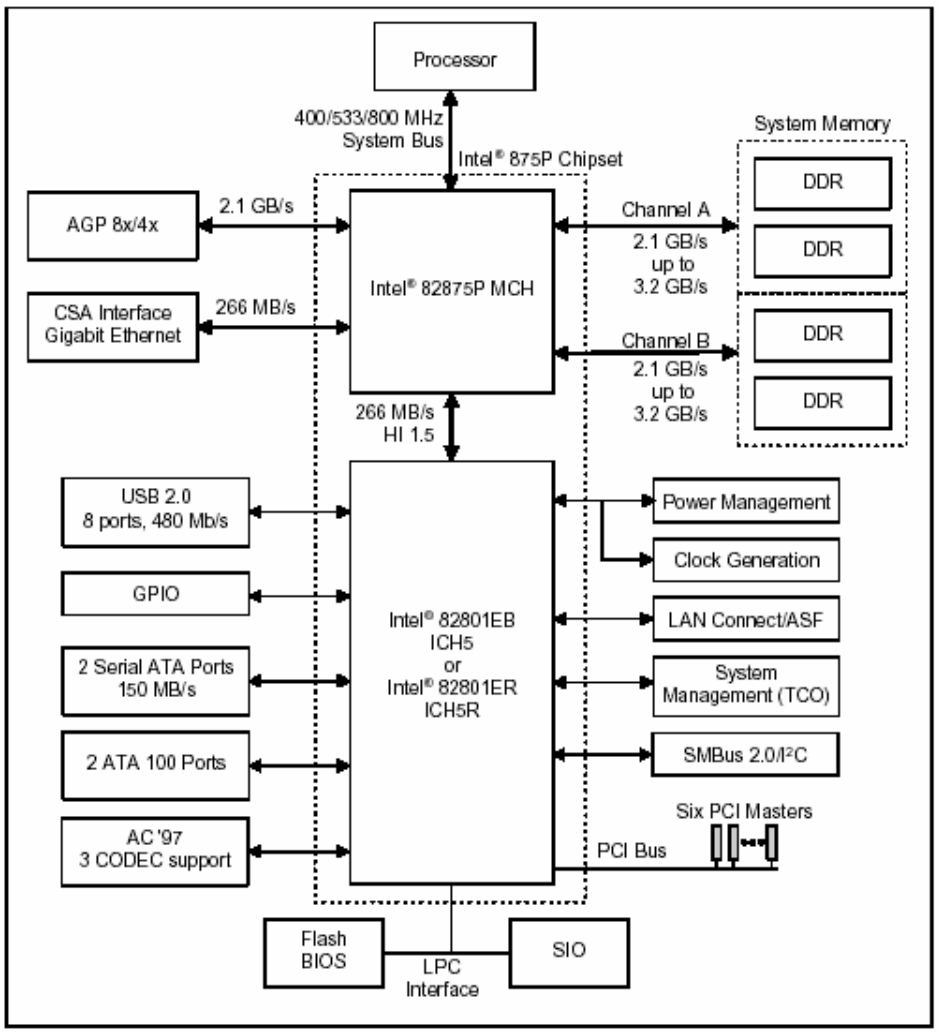
NOTE:
1. Contains AGP Window, GFX Aperture, PCI, and ICH5 ranges.

? 0 – 640 KB　　MS-DOS Area.
? 640 – 768 KB　　Video Buffer Area.
? 768 – 896 KB　in 16-KB sections (total of eight sections) - Expansion Area.
? 896 -960 KB　　in 16-KB sections (total of four sections) - Extended System BIOS Area.
? 960-KB – 1-MB　memory (BIOS Area) - System BIOS Area.
扩展内存区：由 1M 到 4GB－1
— High BIOS area from 4 GB to 4 GB – 2 MB

Intel ? ? 82875P Memory Controller Hub (MCH)

Figure 2-5 illustrates a typical 875P chipset-based system configuration.

Figure 2-5. Typical System Configuration



memery 的监测和初始化：在对内存接口做操作前，必须初始化 MCH DRAM 寄存器。MCH 必须配制成针对所安装的内存的类型进行操作。对内存类型和大小的检测是通过 ICH5 上的 System Management Bus (SMBus)来完成。这个两线的总线通过 DRAM DIMM 上的 Serial Presence Detect 端口获得 DRAM 的类型和大小信息。BIOS 需要确定每行内存的大小和类型来配置 MCH 内存接口。

2、　x86MPU 启动时的初始化

复位输入提供一种初始化的硬件手段。通过复位接口电路向 mpu 提供信号,Reset 要保持 1 至少 15 个

CLK2 周期，当返回 0 后，MPU 启动内部初始化程序，进入实地址模式。初始化完成后，标志寄存器设为 0xUUUU0002(u 代表未定义，实模式下 9 位标志可用，这里是奇偶标志为 1)；指令指针设为 0x0000FFF0,CS 寄存器设为 0xF000,DS,SS,ES,FS 和 GS 寄存器都设为 0x0000，指令队列清空。

实模式下地址的形成：段基址+指令指针偏移

MPU 在识别出 Reset 信号后把数据总线设在高阻状态，地址线强行设为 1。由于清空中断标志是初始化的一部分，外部中断被禁止。因为代码段寄存器为 0xF000，指令指针为 0x0000FFF0，地址线 A20-A31 全部是 1，从而复位后实模式程序从地址 0xFFFFFFF0 开始（只用于实模式高地址位忽略，从地址 0xFFFF0 开始。该地址处可以包含一条转移指令跳到启动程序处。

物理地址为 0xfffffff0 的代码将被执行。这个地址被存储在一个只读存储器（ROM）里。BIOS（基本输入/输出系统）实际上是一段存储在 ROM 里的程序。它包含了一系列可以被某些操作系统调用，用于处理计算机各种硬件设备的中断驱动和低级程序。其中微软的 DOS 就是这样的一种操作系统。

所有的 BIOS 程序都是在实模式下运行的。但是，Linux 内核是在保护模式下运行，而不是在实模式下。因此，一旦初始化完成后，Linux 就不再使用 BIOS，而是完全由自己来为计算机上的所有硬件提供驱动程序。

那么什么时候 Linux 使用保护模式？为什么 BIOS 不能使用相同的模式？BIOS 使用实模式是因为其在操作过程中使用的是实模式地址，并且在计算机刚打开电源时，只有实模式地址可用。一个实模式地址由段地址和偏移地址组成，因此，相应的物理地址就为段地址×（2×8）+偏移。

这是不是意味着在整个启动过程中,Linux 就从来不使用 BIOS 了呢？答案是否定的。在启动阶段,Linux 从硬盘或者其它外部设备加载内核时，需要使用 BIOS。

3、BIOS 的作用：

BIOS 要对硬件进行一系列彻底的检测。这个步骤主要是检查系统安装有哪些设备,以及它们工作是否正常。通常把这个步骤叫做自检（Power-On Self-Test,POST），这时会显示版本及其它很多相关的硬件信息。

BIOS 要对硬件进行初始化。这一步非常重要，因为它要保证所有的硬件设备在 IRQ（中断请求）和 I/O 端口操作时都没有冲突。等这步完成以后，它会显示一个已经安装的 PCI 设备表。

接着到了操作系统，BIOS 将查找一个可以引导的操作系统。这取决于 BIOS 的设置，它可以从软盘、硬盘或者光盘启动。

一旦发现一个合法的设备，BIOS 就会把其第一扇区的内容复制到物理地址，即从 0x00007c00 开始的内存中，然后跳至刚加载的地址并执行之。

BIOS 调用一个专门的程序，这个程序的任务就是把操作系统的内核调入内存。这个程序就叫做自举程序（Boot Loader）。

4、详细启动和初始化过程描述

1）从软盘启动 Linux

从软盘启动时，存储在软盘第一扇区的指令将被加载并执行。这个指令然后就会把其余的内核复制到内存中。

Linux 内核可以装在 1.44MB 的软盘里，不过为了减少磁盘占用量，它们都进行了压缩。这个压缩过程是在编译时完成的，而解压缩的过程则由自举程序完成。

从软盘启动 Linux 时，自举程序要做的工作非常简单。它是一个位于/usr/src/linux-2.4.2/arch/i386/boot/bootsect.S 的汇编语言文件。当我们编译 Linux 内核源代码，或者获取一个新的内核时，这个可执行的汇编代码就会被放在内核程序的前端。由此可见，要制作一个可启动的 Linux 软盘其实很简单。我们只要从磁盘的第一个扇区拷贝 Linux 内核，就可以创建一个可启动软盘。当 BIOS 加载软盘的第一个扇区时，它实际上拷贝的是自举程序。自举程序由 BIOS 调用（跳到物理地址为

0x00007c00 的位置），然后执行以下的操作：

（1）把自已从地址 0x00007c00 移动到 0x00090000；

（2）使用地址 0x00003ff4，创建“实模式”栈；

（3）设置磁盘参数表，这里使用的是 BIOS 提供的软盘驱动程序；

通过调用 BIOS 程序显示“Loading”信息；

（4）自举程序调用 BIOS 程序来加载软盘上内核的 setup()函数，并把它放在起始地址为 0x00090200 的内存中；

（5）接下来自举程序调用一个 BIOS 程序，这个程序从软盘加载剩余的内核程序，并将其放入起始地址为 0x00010000（所谓的低地址）或者 0x00100000（所谓的高地址）；

（6）然后，跳转到 setup()函数。

2）从硬盘启动 Linux

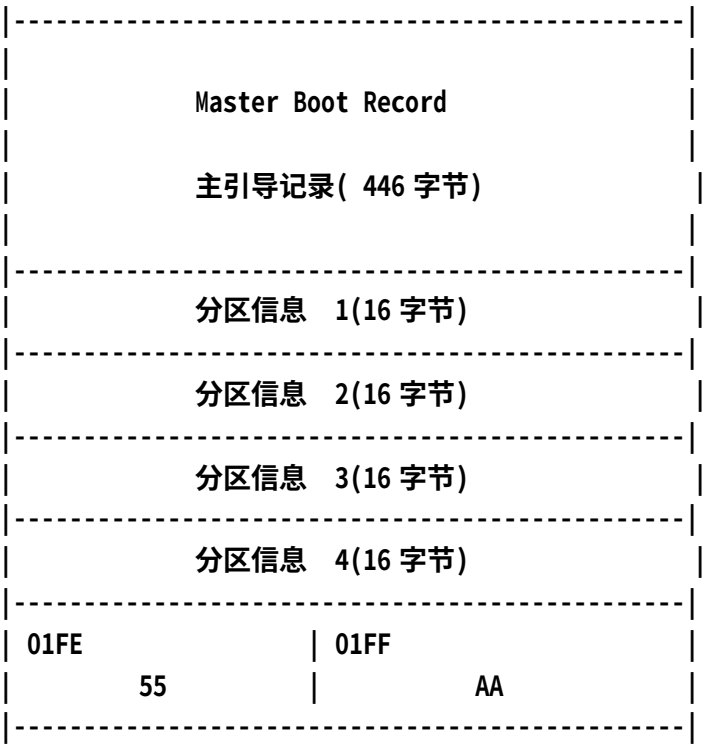
当系统从硬盘启动时，启动过程又有所不同。硬盘的第一个扇区（0 头 0 道 1 扇区）叫做 MBR（Master Boot Record），其上存储着分区表和一个小程序。这个程序加载存储由操作系统的第一扇区来开始启动。Linux 是一个高度灵活且非常优秀的软件，所以在 MBR 里，它使用一个叫做 LILO 的程序来代替上述的那个程序。LILO 允许用户选择所要启动的操作系统。

Boot Sector 也就是硬盘的第一个扇区，它由 MBR (Master Boot Record),DPT (Disk Partition Table) 和 Boot Record ID 三部分组成.

MBR 又称作主引导记录占用 Boot Sector 的前 446 个字节 (0 to 0x1BD),存放系统主引导程序（它负责从活动分区中装载并运行系统引导程序).

DPT 即主分区表占用 64 个字节 (0x1BE to 0x1FD)，记录了磁盘的基本分区信息．主分区表分为四个分区项，每项 16 字节，分别记录了每个主分区的信息(因此最多可以有四个主分区). Boot Record ID 即引导区标记占用两个字节 (0x1FE and 0x1FF)，对于合法引导区，它等于 0xAA55，这是判别引导区是否合法的标志.

Boot Sector 的具体结构如下图所示:



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。 BIOS 首先检查 0000:7dfe 的位置是不是 0x55AA,若不是就去启动其他介质。

一般来说，Linux 是从硬盘启动的。这就需要不同的自举程序。在 Intel 系统里，用得最多的自举程序就是 LILO。对于其它的体系结构，还存在着别的自举程序。LILO 可以安装在 MBR 上（请注意：在安装 Red Hat Linux 时，有一个步骤会让用户选择把 LILO 安装到 MBR 或者引导扇区）或一个活动分区的引导扇区上。

由于 LILO 太大，MBR 无法容纳，所以它被分成两部分。MBR（或者磁盘分区的引导扇区）包含有一个小的自举程序，它被 BIOS 载入到起始地址为 0x00007c00 的内存中。然后，这个小程序再把自己移到 0x0009a000 地址处，接着设置实模式栈，最后加载第二部分的 LILO 自举程序（请注意：实模式栈地址范围是 0x0009b000 到 0x0009a200）。

第二部分的 LILO 会从磁盘读取所有可用的操作系统，并且给用户列出，以选择所要启动的系统。一旦用户选择完成，自举程序就会加载相应的扇区内容到内存中并且执行之。

- **自举程序 bootsect.S**

自举程序 bootsect.S 被 BIOS 调用时（跳到物理地址为 0x00007c00 处），要执行以下操作：

（1）把自已从地址 0x00007c00 移动到 0x00090000；

(2)使用地址 0x00003ff4，创建“实模式”栈；

(3)设置磁盘参数表。

(4)通过调用 BIOS 程序显示“Loading Linux”信息；

(5)自举程序 BIOS 调用来加载的 setup()函数，并把它放在起始地址为 0x00090200 的内存中；

(6)自举程序 BIOS 调用加载剩余的内核程序，并将其放入起始地址为 0x00010000 或者 0x00100000；
(根据内核类型，对于小内核 zImage 放在 0x10000 ，大内核 bzImage 放在 0x100000)
此处是如何判断要加载的内核是什么的？？？？

(7)然后，跳转到 setup()函数。

- **Setup（）函数的功用**

Setup（）函数可以在/linux-2.4.22/arch/i386/boot/setup.S 文件中找到。

Setup（）函数代码是在完整的内核自举程序加载以后，才会跳到相应的函数代码处。在内核文件中，其偏移地址是 0x200。这使得自举程序很容易找到这段代码，并将其拷贝到起始物理地址为 0x00090200 的内存中。

这个 Setup()文函数到底是做什么用的？在计算机时里，内核要正确地操作所有硬件就必需首先要检测到它们，并且以一种有序的方式进行初始化。Setup()函数初始化所有的硬件设备，从而为内核操作它创造了一个环境。

但是,前面我们不是已经提到过BIOS会检测所有的硬件吗？虽然BIOS初始化了所有的硬件,但是Linux内核并不放心，它还要以自己的方式对所有的硬件进行初始化。Linux 内核之所以要设计成这样，是为了增强可移植性和稳定性。这也是 Linux 内核要优于很多目前可用的 Unix 和类 Unix 内核的原因之一，并且也使得它在很多方面表现的非常出众。

Setup()函数主要完成以下任务：

(1)首先是检测系统可用内存的总量，它是通过 BIOS 程序来完成检测的；

(2)设置键盘重复延迟时间和重复速度；

(3)检测视频卡；

(4)重新初始化硬盘控制器和硬盘参数；

(5)检测一个 MCA；

(5)检测一个 PS/2 定点设备（鼠标总线）；

(6)检测高级电源管理器（APM）BIOS 支持；

(7)检测内核在内存中的位置，如果在低地址 0x00010000，就将其移到高地址 0x00001000，如在高地址则不做任何移动；

(8)设置设备中断描述表（IDT）和全局描述表（GDT）；

(9)如已经有了浮点单位（FPU），则重置之；

(10)重新调用程序中断控制器；

(11)通过设置 cr0 状态寄存器的 PE 位，把 CPU 从“实模式”切换到“保护模式”；

(12)跳转到 stratup_32( )汇编语言函数。

因为在内核中不能做 BIOS 调用，内存信息由 setup 通过 INT 0X15 来加以查询，并根据获得的信息生成一张物理内存构成图，称为 e820 图，再通过参数块传给内核，使内核知道系统中内存资源的配置。因为在做 int 0x15 来查询内存构成是要把调用参数之一设置成 0xe820，所以叫 e820 图。

- **第一个 stratup_32( )函数**

在启动过程中要用到两个 stratup_32( )函数，虽然它们都是汇编语言函数，但是却是两个完全不同的函数。我们这里所说的函数包含在/usr/src/linux-2.4.2/arch/i386/boot/compressed/head.S 文件里。Setup()文件执行后,这个函数就被加载到物理地址为 0x00100000 或者物理地址为 0x00001000 的内存中(取决于内核是载入高或者低内存)。

当执行这个函数时，会执行以下的操作：

(1)初始化段寄存器和一个临时栈。

(2)内核中没有初始化的数据都用 0 填充。它是通过 symbols _edata 和 _end 来识别的。

(3)执行 decompress_kernel( )函数。这个函数用于对 Linux 内核解压缩。这个时候，屏幕上将显示“Uncompressing Linux……”信息。解压缩完成后，就会显示“OK, booting the kernel”信息。现在有一个问题,就是解完压缩的内核被放置在什么位置？答案是如果 Linux 内核被加载低地址,那么解压缩的内核将被置于物理地址为 0x00100000 的地方。如果在高地址，则内核会被先解压到一个临时缓冲区中，待完成后再将其加载到物理地址为 0x00100000 的地方。

(4)最后，跳转到物理地址为 0x00100000 的地方执行。

到此为止，代码执行操作就由另外一个 startup_32( )函数来接管。也就是说，第二个 startup_32( )函数接管了启动过程。

- **第二个 startup_32( )函数完成的功能**

解压缩 Linux 内核的工作由另外一个 startup_32( )函数来完成。该函数位于/usr/src/linux-2.4.2/arch/i386/kernel/head.S 文件中。

这时你可能会说两个不同的函数用同一个名字不会出错吗？答案是不会的。因为两个函数都是到自己初始地址去执行，并且都有自己的执行环境，所以不会出错。

下面我们来看一下第二个 startup_32( )函数的功能。当执行这个函数时，实际上是为第一个 Linux 进程（process 0）设置环境。这个函数将执行下面的操作：

(1) 寄存器将以最后的值进行初始化；

(2) 为 process 0 设置内核模式栈；

(3) 调用并且执行 setup_idt( )函数，该函数将把所有的 IDT 填充空值；

(4) 把从 BIOS 中获得的参数放在第一页的框架中；

(5) 识别处理器的模式；

(6) 使用 GDT 和 IDT 表加载 gdtr 和 idtr 寄存器；

(7) 最后跳到 start_kernel( )函数。

- **start_kernel( )函数功能**

start_kernel( ) 函数完成 Linux 内核的初始化工作。这个函数执行后，所有的基本内核组件都将被初始化。这也是整个启动过程的最后一步。

该函数将完成以下的功能：

(1) 输出 Linux 版本信息（printk(linux_banner)）
(2) 设置与体系结构相关的环境（setup_arch()） ——>页表结构初始化（paging_init()）
(3)提取并分析核心启动参数(从环境变量中读取参数,设置相应标志位等待处理,(parse_options())
(4) 使用"arch/alpha/kernel/entry.S"中的入口点设置系统自陷入口（trap_init()）
(5) 使用 alpha_mv 结构和 entry.S 入口初始化系统 IRQ（init_IRQ()）
(6) 核心进程调度器初始化（包括初始化几个缺省的 Bottom-half，sched_init()）
(7) 时间、定时器初始化（包括读取 CMOS 时钟、估测主频、初始化定时器中断等，time_init()）
(8) 控制台初始化（为输出信息而先于 PCI 初始化，console_init()）
(9) 初始化可安装模块机制,计算出内核符号表的大小 init_modules()
(10) 剖析器数据结构初始化（prof_buffer 和 prof_len 变量）
(11) 核心 Cache 初始化（描述 Cache 信息的 Cache，kmem_cache_init()）
(12) 延迟校准（获得时钟 jiffies 与 CPU 主频 ticks 的延迟，calibrate_delay()）
(13) 内存初始化（设置内存上下界和页表项初始值，mem_init()）
(14) 创建和设置内部及通用 cache（"slab_cache"，kmem_cache_sizes_init()）
(15) 创建页 cache（内存页 hash 表初始化，pgtable_cache_init()）
(16) 根据物理内存的大小计算出允许创建线程(包括进程)的数量 fork_init(num_mappedpages);
(17) proc_caches_init()； vfs_caches_init(num_physpages)；buffer_init(num_physpages)。都是为有关的管理机制建立起专用的 slab 缓冲区队列。
(18) 分配空间建立起缓冲页面杂凑表 page_hash_table （ page_cache_init())
(19) 对 Sys V 进程间通信机制的初始化 ipc_init();
(20) 创建信号队列 cache（"signal_queue"，signals_init()）
(21) 检查体系结构漏洞（对于 alpha，此函数为空，check_bugs()）
(22)SMP 机器其余 CPU(除当前引导 CPU)初始化(对于没有配置 SMP 的内核,此函数为空,smp_init())
启动 init 过程（创建第一个核心线程，调用 init()函数，原执行序列调用 cpu_idle() 等待调度，init())
至此 start_kernel()结束，基本的核心环境已经建立起来了

- **init（）函数作用**

init()函数作为核心线程，首先锁定内核（仅对 SMP 机器有效），然后调用 do_basic_setup()完成外设及其驱动程序的加载和初始化。过程如下：
总线初始化（比如 pci_init()） ；
网络初始化（所有协议的初始化过程，sock_init()）；
创建事件管理核心线程（start_context_thread()函数启动 context_thread()过程，并重命名为 keventd）；
启动任何使用__initcall 标识的函数（方便核心开发者添加启动函数，do_initcalls())
do_initcalls()-→partion_setup()-→device_init()是所有外设初始化的总入口；
文件系统初始化（filesystem_setup()，主要是 devfs）；
安装 root 文件系统（mount_root()）
至此 do_basic_setup()函数返回 init()，在释放启动内存段（free_initmem()）并给内核解锁以后，init()打开/dev/console 设备，重定向 stdin、stdout 和 stderr 到控制台，最后，搜索文件系统中的 init 程序（或者由 init=命令行参数指定的程序），并使用 execve()系统调用加载执行 init 程序。
init()函数到此结束，内核的引导部分也到此结束了，这个由 start_kernel()创建的第一个线程已经成为一个用户模式下的进程了。此时系统中存在着六个运行实体：
start_kernel()本身所在的执行体，这其实是一个"手工"创建的线程，它在创建了 init()线程以后就进入 cpu_idle()循环了，它不会在进程（线程）列表中出现

6、模式向保护模式的转换以及内核使用物理地址向虚拟地址的转换

CPU 在跳转到 bootsect 时还处于 16 位实地址模式，然后在 setup 的执行过程中转入 32 位保护模式的段式寻址方式。在 bootsect 和 setup 中都利用 BIOS 提供的调用来完成一些比较大的操作，如读磁盘，取得 BIOS 在加电自检是搜集到的有关内存的信息等等。一旦转入内核映像本身的执行就不再需要 BIOS 了。

- **模式转换的控制：**

x86 保护模式比实模式多了几个寄存器：全局描述符表寄存器（GDTR）,中断描述符表寄存器（IDTR），局部描述符表寄存器（LDTR）和任务寄存器（TR）。另外，一些寄存器的功能得到了扩展，例如指令指针成为 EIP,长度 32 位；标志寄存器(EFLAGS)的更多位得到了利用。四个控制寄存器 CR0－CR3 都得到了利用。
GDTR（48 位）在物理存储器地址空间定义了全局描述符表 GDT（每个描述符 8 字节）。
IDTR（长度同 GDTR）定义了中断描述符表 IDT。(256 个中断，每个中断门 8 字节)

在从实模式转到保护模式前必须将 GDTR 的基址 BASE 和限长 limit (16 位) 的值装入 GDTR ;IDTR 也是。这两个表的装入和保存有特殊的指令 LGDT SGDT LIDT SIDT

16 位的 LDTR 并不直接定义一个局部描述符表，他只是一个指向 GDT 中 LDT 描述符的选择符。选择符装入 LDTR 相应的描述符就能从全局存储器中读出来装入局部描述符表高速缓存，该描述符为当前任务创建了一个 LDT。

控制寄存器

31	23	15	7	0	
页目录基址寄存器（PDBR）			保留		Cr3
缺页线性地址					Cr2
保留					Cr1
PG	保留	R	TS	EM MP PE	Cr0

CR0 的低五位是机器状态字 MSW，包含保护模式配置和状态信息。PE（保护模式允许）位重启时清零允许实模式操作，为进入保护模式将 PE 设为 1。一旦处于保护模式就不能再设回到实模式，除非重启。MP（数学协处理器存在）设为 1 表示该系统有一个数学协处理器。如果用到了软件模拟器执行数学运算，那么 EM（模拟）位应设为 1，一次只能设置其中的 1 位。R（扩展类型）表示用的是 80287 还是 80387，R 为 1 表示 80387。TS（任务切换）在做任务切换的时候自动设置。

在保护模式下支持分页，CR0 的 PG 位设为 1 表示允许分页，CR3 包含页目录基址寄存器 PDBR。
任务寄存器（TR）是在保护模式下的任务切换中用到。该寄存器存放 16 位的选择符用来指示全局描述符表中描述符的位置，当选择符装入 TR，相应的任务状态段（TSS）描述符自动由存储器读出并装入到任务描述符缓存中。

几条重要的系统指令：LGDT SGDT LIDT SIDT LMSW SMSW
指令 LMSW 和 SMSW 分别用于装入和保存机器状态字信息。他们是用于从实模式转换到保护模式的指令，需要将 MSW 的最右位设为 1。

```
SMSW AX；
OR   AX，1；
LMSW AX；
```

- **地址的转换**

在正常运行时整个内核映像都应该在系统空间，系统空间地址实连续的线性的，虚拟地址和物理地址间有个固定的位移：0xC0000000。内核影像的起点是_stext,引导核压缩后的整个映像放在内存中从 0x100000 即 1Mb 开始的空间。CPU 执行内核映像的入口 start_32 就在内核映像开头的地方，因此物理地址是 0x100000,虚拟地址就成了 0xC0100000。

CPU 在进入 start_32 时运行于保护模式下的段式寻址方式。段描述表中与__KERNEL_CS 和__KERNEL_DS

相对应的描述项所提供的基地址都是 0，所以实际产生的是线性地址。其中代码段寄存器 CS 已在进入 start_32 之前设置成__KERNEL_CS，数据段寄存器尚未设置为__KERNEL_DS。不过虽然代码段已设置，从而 start_32 的地址为 0xC0100000。但是在转入这个入口时使用的指令是“ ljmp 0X10000 ”而不是“ ljmp startup_32 ”,所以装入 COU 寄存器 IP 的地址是物理地址 0x100000 而不是虚拟地址,CPU 在进入 startup_32 后会继续以物理地址取指令。只要不在代码段中引用某个地址，例如项某个地址做绝对转移，或者调用某个子程序，就可以一直这样运行下去，而与 CS 的内容无关，此外，CPU 的中断已在进入 startup_32 前关闭。