操作系统 Operation System

第二章 系统引导

王雷 wanglei@buaa.edu.cn
82316284

计算机的启动过程

- 英文：Bootstrapping（简称boot）
  - 它来自一句谚语：“pull oneself up by one‘s bootstraps”(据考证来自于《吹牛大王历险记》)字面意思是"拽着鞋带把自己拉起来"，这当然是不可能的事情。最早的时候，工程师们用它来比喻，计算机启动是一个很矛盾的过程：必须先运行程序，然后计算机才能启动，但是计算机不启动就无法运行程序！
  - 早期必须想尽各种办法，把一小段程序装进内存，然后计算机才能正常运行。工程师们把这个过程叫做"拉鞋带"，久而久之就简称为boot了。

启动，是一个很“纠结”的过程

- 现代计算机 —— 硬件 + 软件
- 计算机功能的多样性和灵活性 vs 启动状态的单一性
  - 一方面：必须通过程序控制使得计算机进入特定工作状态（*必须运行启动程序来启动计算机*）
  - 另一方面：程序必须运行在设置好工作模式的硬件环境上（*启动程序必须运行在启动好的计算机上*）
- 因此：启动前硬件状态必须假设在一个最安全、通用，因此也是功能最弱的状态，需要逐步设置硬件，以提升硬件环境能力。
- OS启动是一个逐步释放系统灵活性的过程。

关于Bootloader

- 引导加载程序是系统加电后运行的第一段软件代码，称为Bootloader，是在操作系统内核运行之前运行的一段小程序；
- BootLoader是Booter和Loader的合写：
  - 前者要初始化嵌入式系统硬件使之运行起来，至少是部分运行起来；
  - 后者将操作系统映像加载到内存中，并跳转到操作系统的代码运行。
- MIPS处理器大多用于嵌入式系统，嵌入式系统常用U-boot作为OS启动装载程序，U-Boot，全称 Universal Boot Loader；X86处理器通常采用LILO和GRUB。

关于Bootloader

- Bootloader的实现严重依赖于具体硬件，在嵌入式系统中硬件配置千差万别，即使是相同的CPU，它的外设(比如Flash)也可能不同，所以不可能有一个Bootloader支持所有的CPU、所有的开发板。
- 即使是支持CPU架构比较多的U-Boot，也不是一拿来就可以使用的(除非里面的配置刚好与你的板子相同)，需要进行一些移植。

启动及OS引导

当我们打开计算机的电源开关，到我们看到OS的登录界面，计算机内部经历了怎样的过程?

- 计算机的启动过程（MIPS）
- MIPS下Linux系统引导过程
- 计算机的启动过程（X86）
- X86下Linux系统引导过程

选择U-Boot的理由

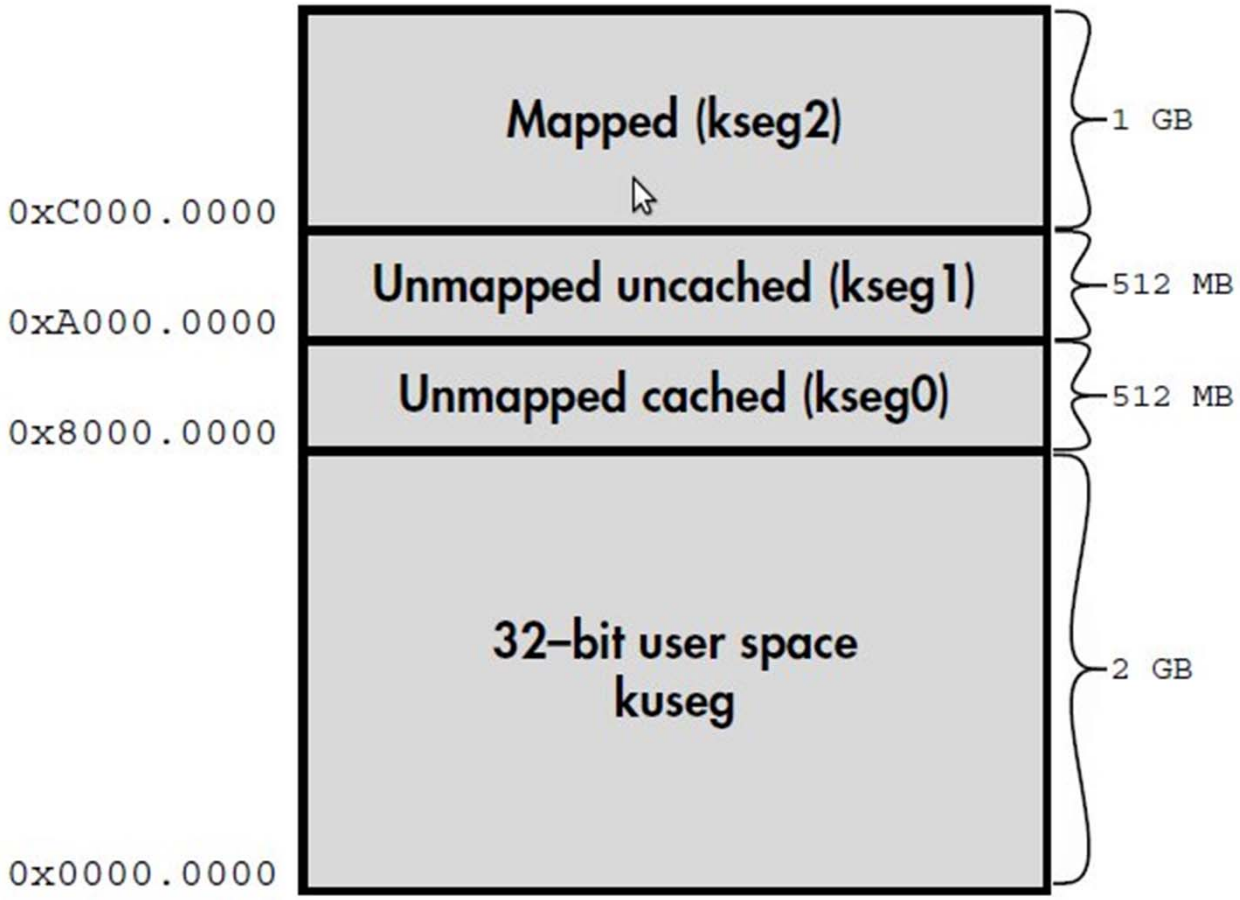
- 开放源码；
- 支持多种嵌入式操作系统内核，如Linux、NetBSD, VxWorks, QNX, RTEMS, ARTOS, LynxOS, android；
- 支持多个处理器系列，如PowerPC、ARM、x86、MIPS；
- 较高的可靠性和稳定性；
- 高度灵活的功能设置，适合U-Boot调试、操作系统不同引导要求、产品发布等；
- 丰富的设备驱动源码，如串口、以太网、SDRAM、FLASH、LCD、NVRAM、EEPROM、RTC、键盘等；
- 较为丰富的开发调试文档与强大的网络技术支持；

U-Boot启动流程

- 大多数BootLoader都分为stage1和stage2两大部分，U-boot也不例外。
- 依赖于cpu体系结构的代码（如设备初始化代码等）通常都放在stage1且可以用汇编语言来实现；
- stage2则通常用C语言来实现，这样可以实现复杂的功能，而且有更好的可读性和移植性。

MIPS的基本地址空间

在32位下,程序地址空间(4G)划分为四大区域,不同区域有不同的属性



MIPS的基本地址空间

- kuseg: 这些地址是用户态可用的地址,在有MMU的机器里,这些地址将一概被MMU作转换,除非MMU的设置被建立好,否则这2G的地址是不可用的.
- kseg0: 将他们的最高位清零,即可映射到物理地址段512M(0x00000000 -- 0x1fffffff).这种映射关系很简单,通常称之为"非转换的"地址区域,几乎全部对这段地址的存取都会通过cache,因此cache设置好之前,不能随便使用这段地址.
  - 通常一个没有MMU的系统会使用这段地址作为其绝大多数程序和数据的存放位置;
  - 对于有MMU的系统,操作系统核心会存放在这个区域.

MIPS的基本地址空间

- kseg1: 将这些地址的高三位清零可映射到相应的物理地址上,与kseg0映射的物理地址一样,但kseg1是非cache存取的. kseg1是唯一在系统重启时能正常工作的地址空间.
- kseg2: 这块区域只能在核心态下使用并且要经过MMU的转换. 在MMU设置好之前,不要存取该区域. 除非在写一个真正的操作系统,否则没有理由用kseg2. 有时会看到该区域被分为kseg2和kseg3,意在强调低半部分(kseg2)可供运行在管理态的程序使用.

MIPS ROM/Flash启动地址

- MIPS上电启动时，由于OS尚未接管系统，不能采用TLB、Cache机制。从MIPS的初始内存划分可知，kseg1是唯一的在系统重启时能正常工作的内存映射地址空间。
- MIPS的启动入口地址是0xBFC00000，通过将最高3位清零（&0x1fffffff）的方法，将ROM所在的地址区映射到物理内存的低端512M(0x00000000 - 0x1FFFFFFF)空间，也是“非翻译无需转换的”（Unmapped）地址区域。
- 因此，kseg1是唯一的在系统重启时能正常工作的内存映射地址空间，这也是为什么重新启动时的入口向量是（0xBFC00000）会在这个区域。这个向量对应的物理地址是0x1FC00000。

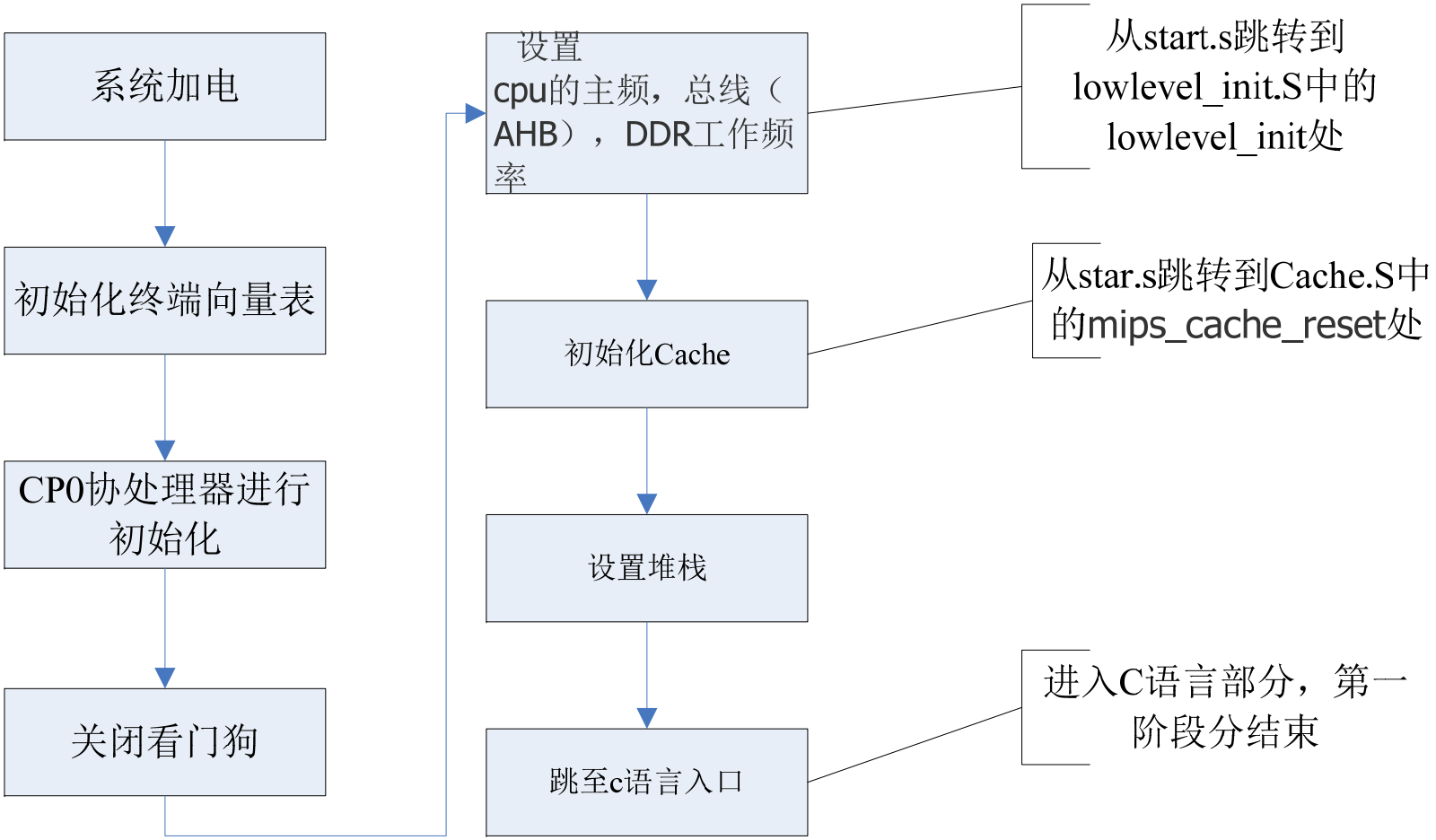
MIPS启动过程1

- 启动涉及到几个文件： start.S， cache.S， lowlevel_init.S 和 board.c, 前三个都是汇编代码。
- 程序从start.S的_start开始执行。
  - 初始化中断向量，寄存器清零，大致包括32个通用寄存器reg0-reg31和协处理器的一些寄存器；
  - 配置寄存器CP0_STATUS，设置所使用的协处理器，中断以及cpu 运行级别（核心级）。
  - 配置gp寄存器。
- 执行lowlevel_init.S的lowlevel_init，主要目的是工作频率配置，比如cpu的主频，总线（AHB），DDR工作频率等。

MIPS启动过程1

- 调用cache.S的mips_cache_reset对cache进行初始化。
- 接着调用cache.S的mips_cache_lock。
- 由于此时ddr ram并没有配置好，而如果直接调用c语言的函数必须完成栈的设置，而栈必定要在ram中。所以，只有先把一部分cache拿来当ram用。做法就是把一部分cache配置为栈的地址并锁定。这样，当读写栈的内存空间时，只会访问cache，而不会访问真的ram地址了。

MIPS启动过程1

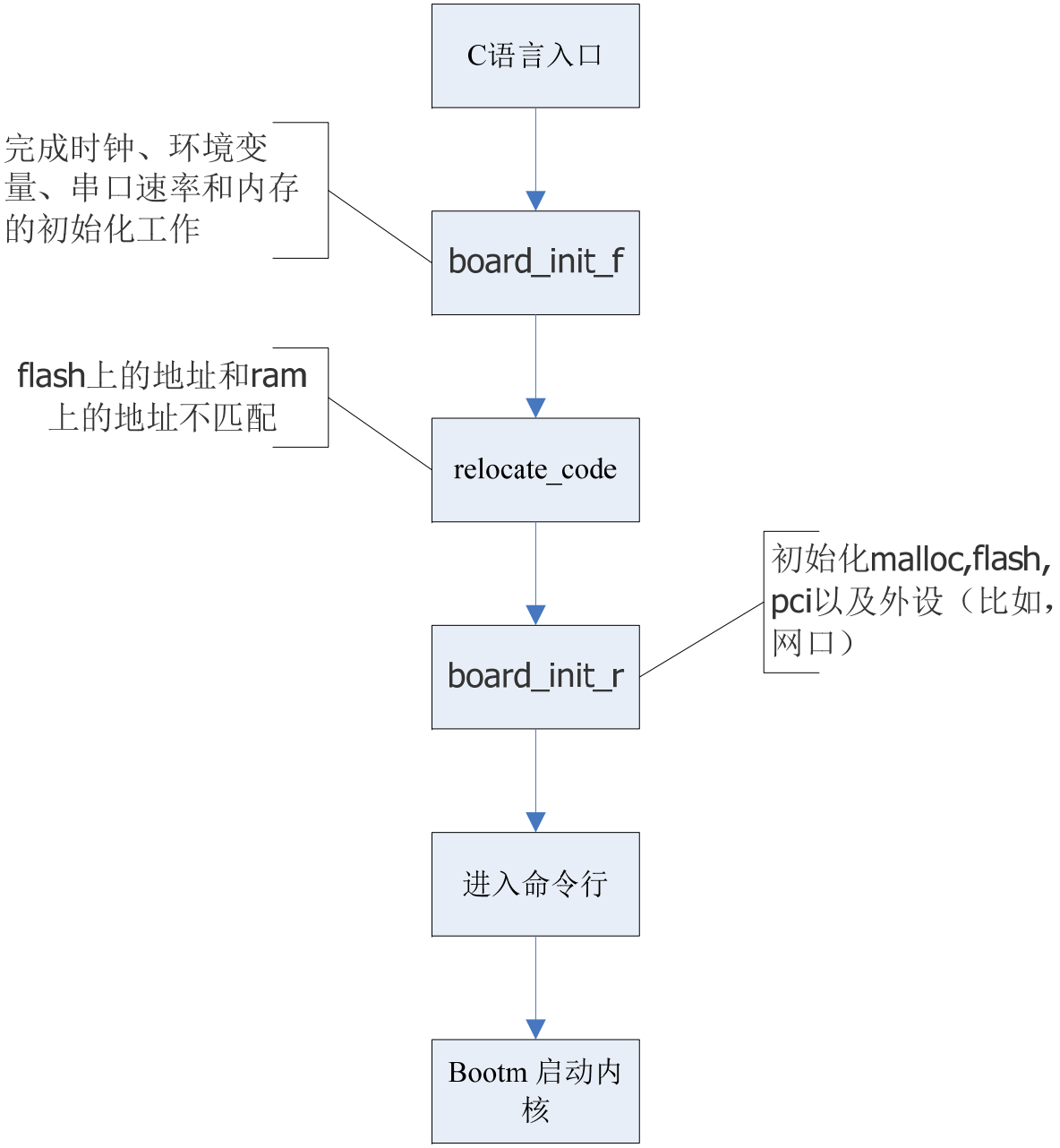

系统加电
初始化终端向量表
CP0协处理器进行初始化
关闭看门狗
设置cpu的主频，总线（AHB），DDR工作频率
初始化Cache
设置堆栈
跳至c语言入口
从start.s跳转到lowlevel_init.S中的lowlevel_init处
从star.s跳转到Cache.S中的mips_cache_reset处
进入C语言部分，第一阶段分结束

MIPS启动过程2（C代码）

- 调用board.c中的函数board_init_f做一系列初始化：
  - timer_init 时钟初始化
  - env_init 环境变量初始化（取得环境变量存放的地址）
  - init_baudrate 串口速率
  - serial_init 串口初始化
  - console_init_f 配置控制台
  - display_banner 显示u-boot启动信息，版本号等
  - init_func_ram 初始化内存，配置ddr controller

MIPS启动过程2（C代码）

- 上述工作完成后，串口和内存都已经可以用了。然后进行内存划分，对堆和栈初始化，并留出u-boot代码大小的空间，把代码从flash上搬到ram上，继续执行。
- 之后进入board.c的board_init_r函数，在这个函数里初始化 flash, pci 以及外设（比如，网口），最后进入命令行或者直接启动Linux kernel。


C语言入口
完成时钟、环境变量、串口速率和内存的初始化工作
board_init_f
flash上的地址和ram上的地址不匹配
relocate_code
初始化malloc,flash,pci以及外设（比如，网口）
board_init_r
进入命令行
Bootm 启动内核

启动及OS引导

当我们打开计算机的电源开关，到我们看到OS的登录界面，计算机内部经历了怎样的过程?

- 计算机的启动过程（MIPS）
- MIPS下Linux系统引导过程
- 计算机的启动过程（X86）
- X86下Linux系统引导过程

Linux启动第一阶段Head.s

- Bootloader将 Linux 内核映像拷贝到 RAM 中某个空闲地址处，然后一般有个内存移动操作，将内核移到指定的物理地址处。即内核取得控制权后执行的第一条指令的地址。
- linux 内核启动的第一个阶段从 /arch/mips/kernel/head.s文件开始的。而此处正是内核入口函数kernel_entry()，该函数是体系结构相关的汇编语言，它首先初始化内核堆栈段，为创建系统中的第一个进程进行准备，接着用一段循环将内核映像的未初始化数据段清零，最后跳转到 /init/main.c 中的 start_kernel()初始化硬件平台相关的代码。

Linux启动第二阶段start_kernel

1. 设置CPU ID，为多和环境做准备smp_setup_processor_id()
2. 初始化kernel要用到的数据结构
3. 关掉中断
4. 挂接tick回调
5. 在void __init setup_arch(char **cmdline_p)中初始化内存和页表
6. 如果是多核CPU，会给不同core分配物理地址空间
7. 初始化调度器sched_init
8. timer初始化：init_timers（普通）/hrtimers_init（硬实时）
9. 软中断初始化softirq_init
10. 时间初始化time_init
11. 开中断
12. 开console：console_init()
13. vmalloc_init()
14. 内存管理模块初始化mem_init();
15. rest_init()中，初始化kernel_init和kthreadd线程，并将自己设置为idle线程
16. 调用schedule()开始多任务状态下的系统运行
17. kernel_init线程中，使用run_init_process()接口来实现启动用户进程的工作

- MIPS的启动过程相对简单，Linux启动的工作处体系结构相关部分外，其它都几乎一样。
- Linux是一个很复杂的OS，所以它的启动过程很复杂。
- 我的OS实验是在MIPS虚拟机上完成，它比真实的MIPS更简单；
- 我们课程的OS是一个简单的教学型操作系统；
- 所以……

启动及OS引导

当我们打开计算机的电源开关，到我们看到OS的登录界面，计算机内部经历了怎样的过程?

- 计算机的启动过程（MIPS）
- MIPS下Linux系统引导过程
- 计算机的启动过程（X86）
- X86下Linux系统引导过程

X86 启动过程（与OS无关）

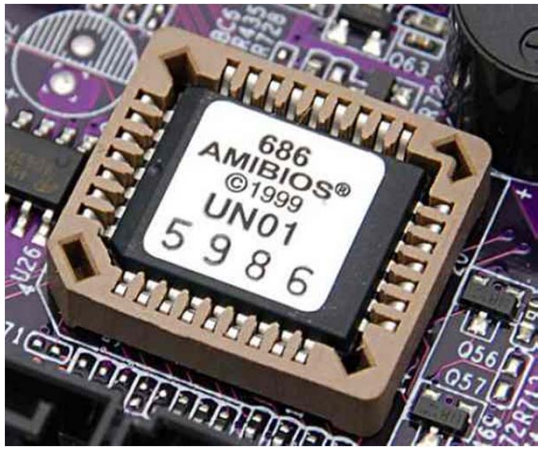
1. Turn on
2. CPU jump to physical address of BIOS (0xFFFF0) （Intel 80386）
3. BIOS runs POST (Power-On Self Test)
4. Find bootable devices
5. Loads boot sector from MBR
6. BIOS yields control to OS BootLoader

第一阶段

第二阶段

BIOS (Basic Input/Output System)

- BIOS设置程序是被固化到电脑主板上地ROM芯片中的一组程序，其主要功能是为电脑提供最底层的、最直接的硬件设置和控制。BIOS通常与硬件系统集成在一起（在计算机主板的ROM或EEPROM中），所以也被称为固件。



BIOS on board

BIOS on screen

BIOS

- BIOS程序存放于一个断电后内容不会丢失的只读存储器中；系统上电或被重置（reset）时，处理器要执行第一条指令的地址会被定位到BIOS的存储器中，让初始化程序开始运行。
- 在X86系统中，CPU加电后将跳转到BIOS的固定物理地址0xFFFF0。（Intel 80386）

启动第一步——加载BIOS

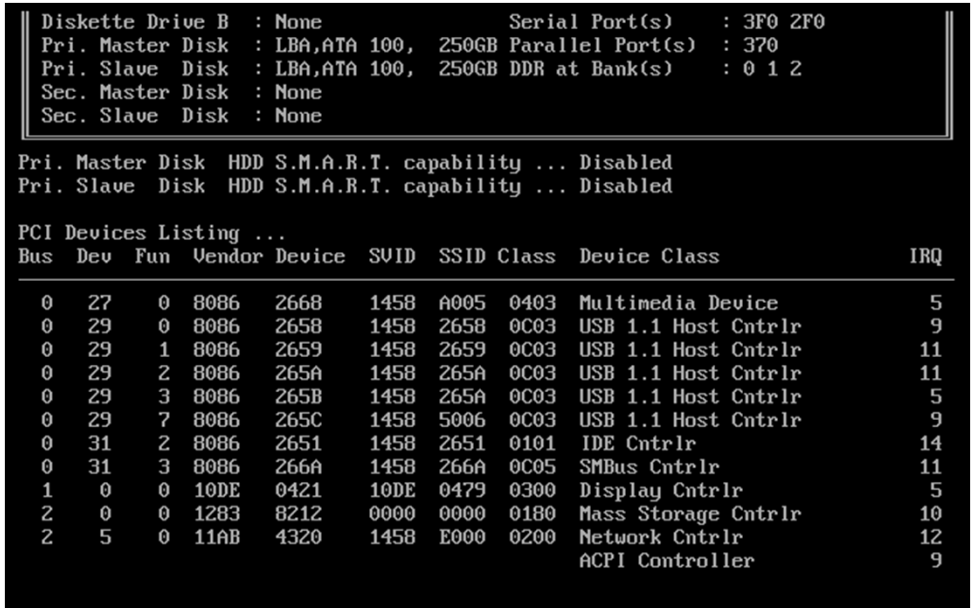
- 当打开计算机电源，计算机会首先加载BIOS信息。BIOS中包含了CPU的相关信息、设备启动顺序信息、硬盘信息、内存信息、时钟信息、PnP特性等等。在此之后，计算机心里就有谱了，知道应该去读取哪个硬件设备了。

BIOS

硬件自检（Power-On Self-Test）

- BIOS代码包含诊断功能，以保证某些重要硬件组件，像是键盘、磁盘设备、输出输入端口等等，可以正常运作且正确地初始化。几乎所有的BIOS都可以选择性地运行CMOS存储器的设置程序；也就是保存BIOS会访问的用户自定义设置数据（时间、日期、硬盘细节，等等）。

如果硬件出现问题，主板会发出不同含义的蜂鸣，启动中止。如果没有问题，屏幕就会显示出CPU、内存、硬盘等信息。



BIOS

读取启动顺序（Boot Sequeue）

- 现代的BIOS可以让用户选择由哪个设备引导电脑，如光盘驱动器、硬盘、软盘、USB U盘等等。这项功能对于安装操作系统、以CD引导电脑、以及改变电脑找寻开机媒体的顺序特别有用。

打开**BIOS**的操作界面，里面有一项就是"设定启动顺序"。

BIOS的问题

- 16位~20位实模式寻址能力（问：地址空间？）
- 实现结构、可移植性

- 问题根源
  - 历史的局限性、向前兼容的压力
    - 支持遗留软件：老设备驱动等
  - 经典≈（成熟、稳定、共识），来之不易，维持整个产业生态正常运转的必要Tradeoff
  - IT发展太快，对“历史局限”的继承，导致改变成本越来越高。——“另起炉灶”（UEFI）来解决。

UEFI——统一可扩展固件接口

- Unified Extensible Firmware Interface
  - 2000年提出，Intel组建生态
- 功能特性
  - 支持从超过2TB的大容量硬盘引导 (GUID Partition Table，GPT分区) (硬件支持)
  - CPU-independent architecture(可移植性)
  - CPU-independent drivers (可移植性)
  - Flexible pre-OS environment, including network capability (硬件支持)
  - Modular design (可移植性)

https://en.wikipedia.org/wiki/Unified_Extensible_Firmware_Interface#cite_note-note1-15

UEFI和BIOS的比较

二者显著的区别是:

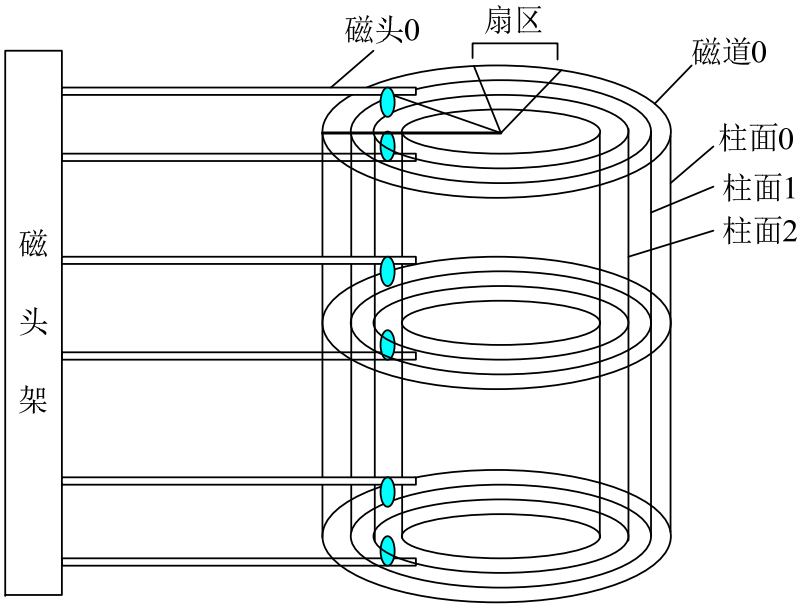
- EFI是用模块化，C语言风格的参数堆栈传递方式，动态链接的形式构建的系统，较BIOS而言更易于实现，容错和纠错特性更强，缩短了系统研发的时间。
- 它运行于32位或64位模式，乃至未来增强的处理器模式下，突破传统BIOS的16位代码的寻址能力，达到处理器的最大寻址。

启动第二步——读取MBR

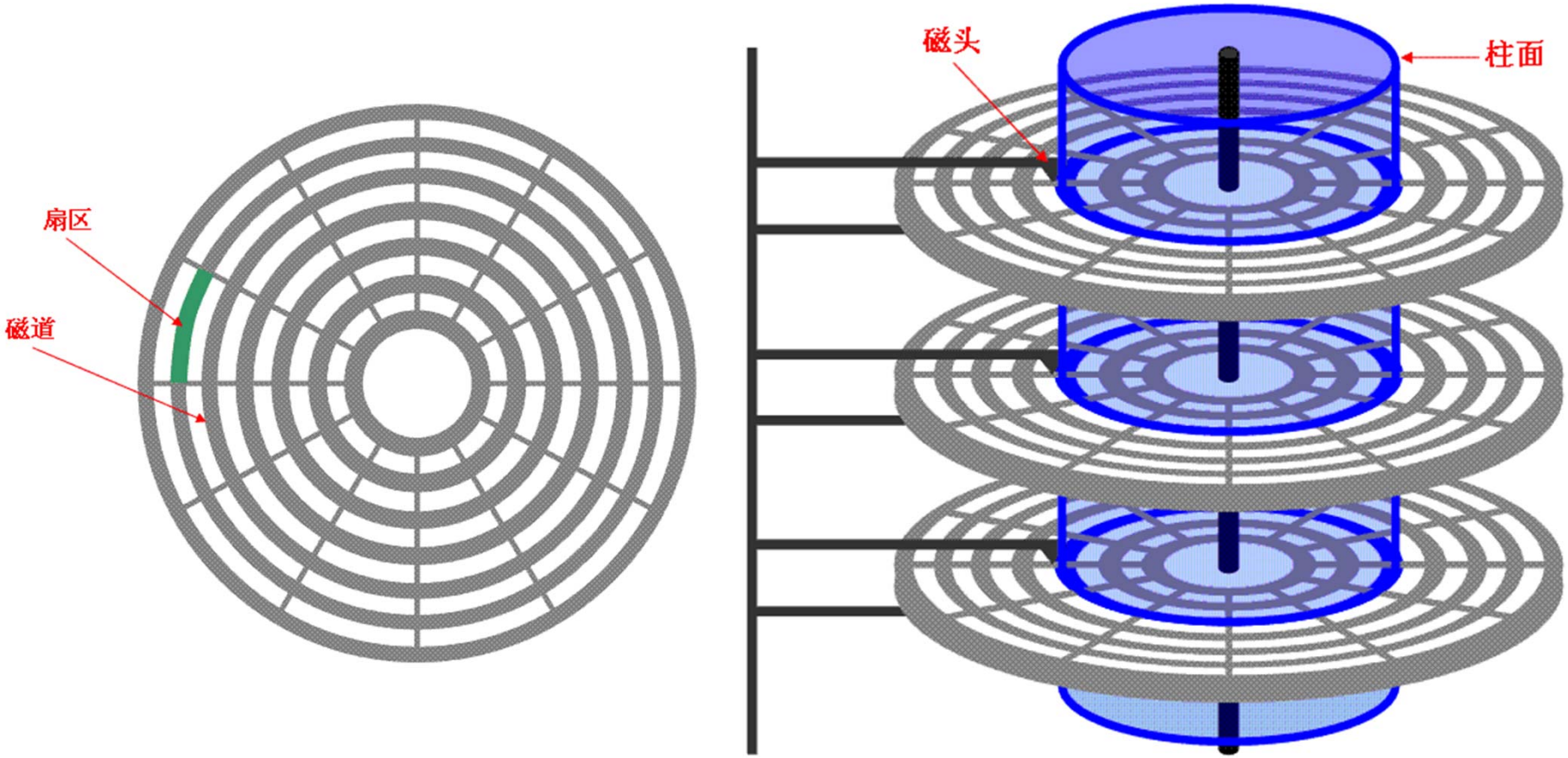
- 硬盘上第0磁头第0磁道第一个扇区被称为MBR，也就是Master Boot Record，即主引导记录，它的大小是512字节，别看地方不大，可里面却存放了预启动信息、分区表信息。

磁盘小知识

- 扇区（sector）
  - 盘片被分成许多扇形的区域
- 磁道（track）
  - 盘片上以盘片中心为圆心，不同半径的同心圆。
- 柱面（cylinder)
  - 硬盘中，不同盘片相同半径的磁道所组成的圆柱。
- 每个磁盘有两个面，每个面都有一个磁头(head)。



磁盘小知识



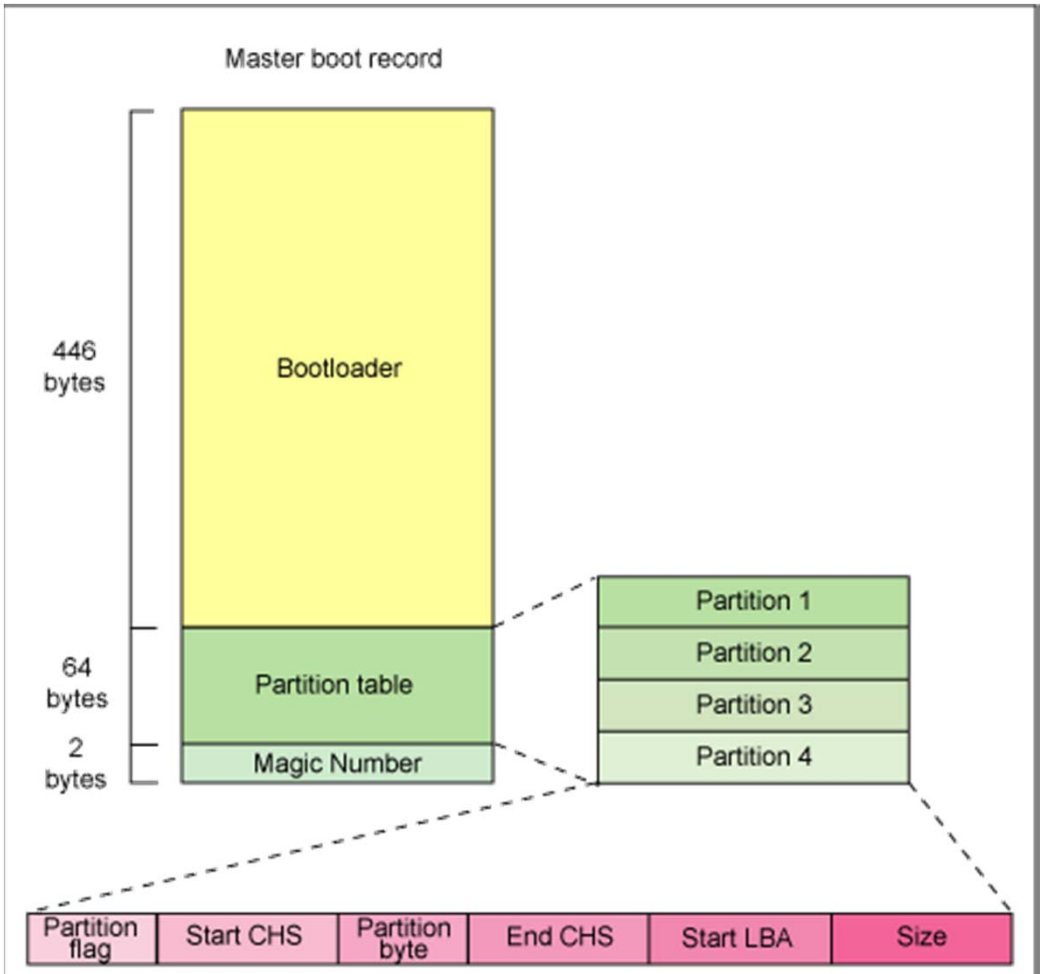
装入MBR

- MBR的全称是Master Boot Record（主引导记录），MBR早在1983年IBM PC DOS 2.0中就已经提出。之所以叫“主引导记录”，是因为它是存在于驱动器开始部分的一个特殊的启动扇区。
- 这个扇区包含了已安装的操作系统的启动加载器(BootLoader)和驱动器的逻辑分区信息。
- MBR 是一个512-byte的扇区，位于磁盘的固定位置(sector 1 of cylinder 0, head 0)

MBR的结构

- MBR(Master Boot Record)主引导记录包含两部分的内容，前446字节为启动代码及数据；
- 之后则是分区表（DPT, Disk Partition Table），分区表由四个分区项组成，每个分区项数据为16字节，记录了启动时需要的分区参数。这64个字节分布在MBR的第447-510字节。
- 后面紧接着两个字节AA和55被称为幻数(Magic Number), BOIS读取MBR的时候总是检查最后是不是有这两个幻数,如果没有就被认为是一个没有被分区的硬盘。

MBR (Master Boot Record)



MBR

Address		Description		Size (bytes)
Hex	Dec			
+0x0000	+0	Bootstrap code area		446
+0x01BE	+446	Partition entry #1	*Partition table* (for primary partitions)	16
+0x01CE	+462	Partition entry #2		16
+0x01DE	+478	Partition entry #3		16
+0x01EE	+494	Partition entry #4		16
+0x01FE	+510	55h	*Boot signature*	2
+0x01FF	+511	AAh		
		Total size: 446 + 4×16 + 2		512

存贮字节位	内容及含义
第1字节	引导标志。若值为80H表示活动分区，若值为00H表示非活动分区。
第2、3、4字节	本分区的起始磁头号、扇区号、柱面号。其中： 磁头号——第2字节； 扇区号——第3字节的低6位； 柱面号——为第3字节高2位+第4字节8位。
第5字节	分区类型符。 00H——表示该分区未用（即没有指定）； 06H——FAT16基本分区； 0BH——FAT32基本分区； 05H——扩展分区； 07H——NTFS分区； 0FH——（LBA模式）扩展分区（83H为Linux分区等）。
第6、7、8字节	本分区的结束磁头号、扇区号、柱面号。其中： 磁头号——第6字节； 扇区号——第7字节的低6位； 柱面号——第7字节的高2位+第8字节。
第9、10、11、12字节	本分区之前已用了的扇区数。
第13、14、15、16字节	本分区的总扇区数

MBR

- 由于MBR的限制 只能有4个主分区，系统必须装在主分区上面。
- 硬盘分区有三种，主磁盘分区、扩展磁盘分区、逻辑分区。
- 一个硬盘主分区至少有1个，最多4个，扩展分区可以没有，最多1个。且主分区+扩展分区总共不能超过4个。逻辑分区可以有若干个。
- 主分区只能有一个是激活的（active），其余为inactive。

MBR

- 分出主分区后，其余的部分可以分成扩展分区，一般是剩下的部分全部分成扩展分区，也可以不全分，剩下的部分就浪费了。
- 扩展分区不能直接使用，必须分成若干逻辑分区。所有的逻辑分区都是扩展分区的一部分。

硬盘的容量＝主分区的容量＋扩展分区的容量

扩展分区的容量＝各个逻辑分区的容量之和

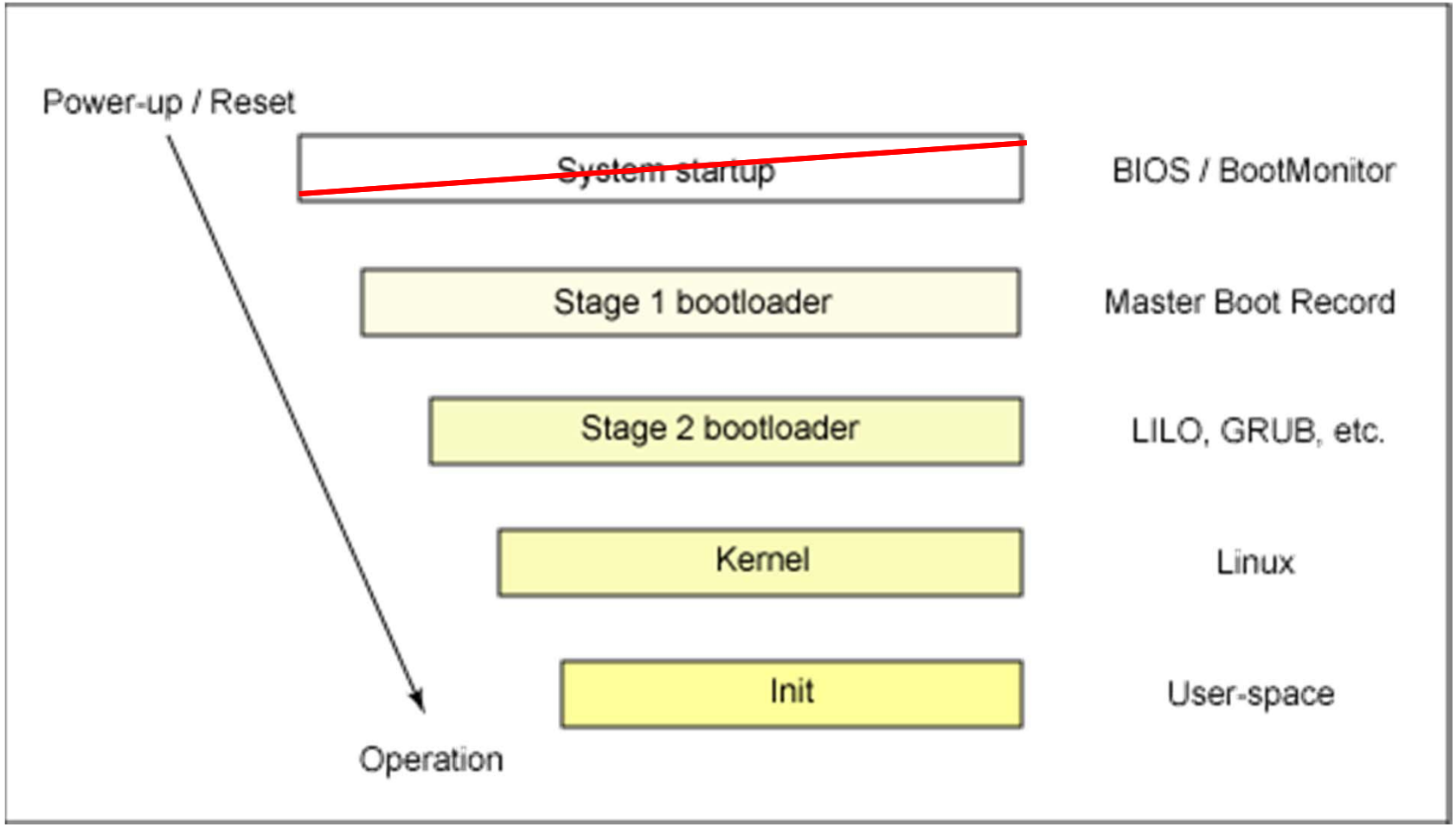
启动及OS引导

当我们打开计算机的电源开关，到我们看到OS的登录界面，计算机内部经历了怎样的过程?

- 计算机的启动过程（MIPS）
- MIPS下Linux系统引导过程
- 计算机的启动过程（X86）
- X86下Linux系统引导过程

How Linux boot?

- 逐级引导、逐步释放灵活性



启动第三步——Boot Loader

- Boot Loader 就是在操作系统内核运行之前运行的一段小程序。通过这段小程序，可以初始化硬件设备、建立内存空间的映射图，从而将系统的软硬件环境带到一个合适的状态，以便为最终调用操作系统内核做好一切准备。

Boot loader

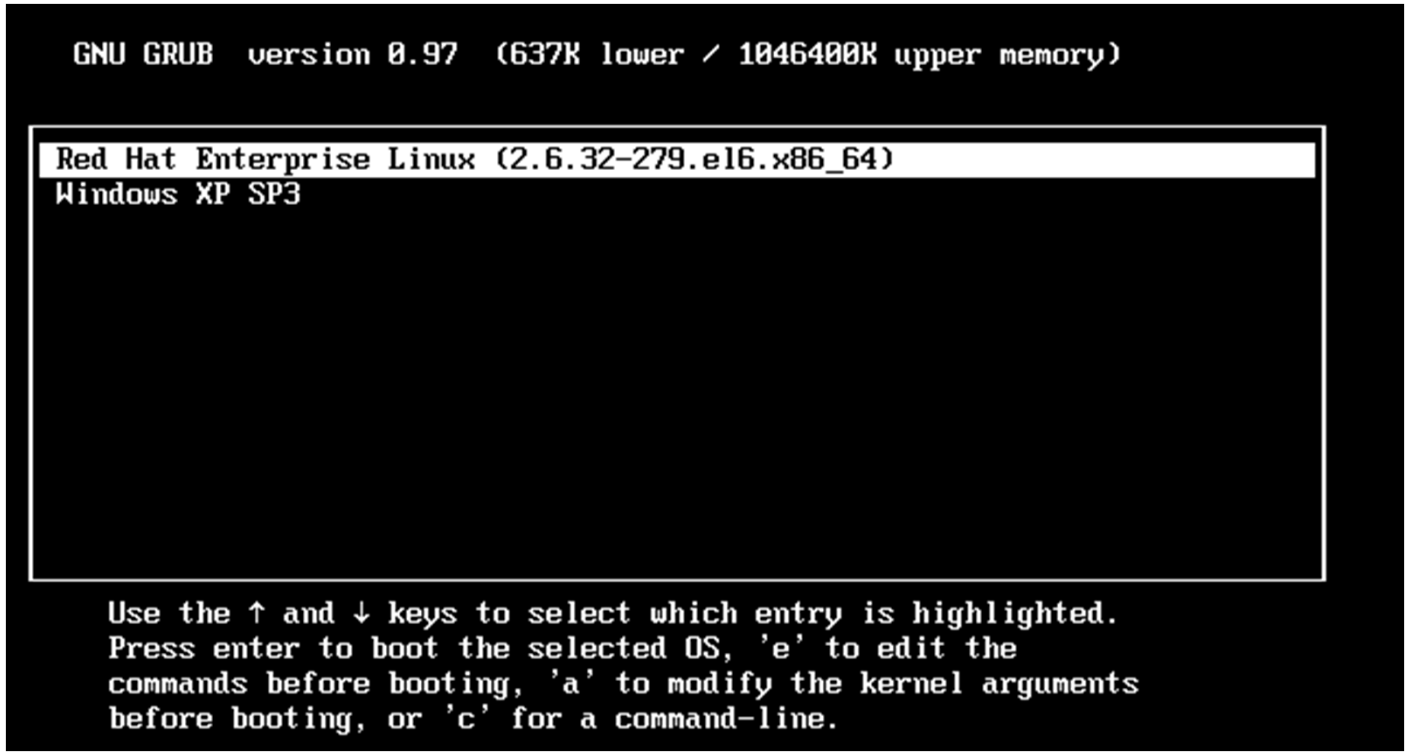
- Boot loader 也可以称之为操作系统内核加载器(OS kernel loader), 是操作系统内核运行之前运行的一段小程序。通过这段小程序，我们可以初始化硬件设备、建立内存空间的映射图，从而将系统的软硬件环境带到一个合适的状 态，以便为最终调用操作系统内核做好一切准备。通常是严重地依赖于硬件而实现的。
- GRUB 和 LILO 最重要的Linux加载器。
  - Linux Loader（LILO）
  - GRand Unified Bootloader（GRUB）

LILO: LInux LOader

- *LInux LOader（LILO）* 已经成为所有 Linux 发行版的标准组成部分，是最老的 Linux 引导加载程序。
- LILO的主要优点是，它可以快速启动安装在主启动记录中的Linux操作系统。
- LILO的主要局限是，LILO 配置文件被反复更改时，主启动记录（MBR）也需要反复重写，但重写可能发生错误，这将导致系统无法引导。

GNU GRUB

- GNU GRand Unified Bootloader
  - 允许用户可以在计算机内同时拥有多个操作系统，并在计算机启动时选择希望运行的操作系统。



GRUB 与 LILO 的比较

- LILO 没有交互式命令界面，而 GRUB 拥有。
- LILO 不支持网络引导，而 GRUB 支持。
- LILO 将关于可以引导的操作系统位置的信息物理上存储在 MBR 中。如果修改了 LILO 配置文件，必须将 LILO 第一阶段引导加载 程序重写到 MBR。错误配置的 MBR 可能会让系统无法引导。
- 使用 GRUB，如果配置文件配置错误，则只是默认转到 GRUB 命令行界面。

GRUB磁盘引导过程

- stage1: grub读取磁盘第一个512字节（硬盘的0道0面1扇区，被称为MBR（主引导记录）,也称为bootsect）。MBR由一部分bootloader的引导代码、分区表和魔数三部分组成。（启动的第二步）
- Stage1.5: 识别各种不同的文件系统格式。这使得grub识别到文件系统。
- stage2: 加载系统引导菜单(/boot/grub/menu.lst或grub.lst)，加载内核映像(kernel image)和RAM磁盘initrd（可选）。

运行主引导程序

- BIOS将硬盘的主引导记录（位于0柱面、0磁道、1扇区）读入7C00处，然后将控制权交给主引导程序（GRUB的第一阶段)。任务包括：

  1. **检查（WORD）0x7dfe是否等于0xaa55。若不等于则转去尝试其他介质；如果没有其他启动介质，则显示“No ROM BASIC”，然后死机；**
  2. **跳转到0x7c00处执行MBR中的程序；**
  3. **将自己复制到0x0600处，然后继续执行；**

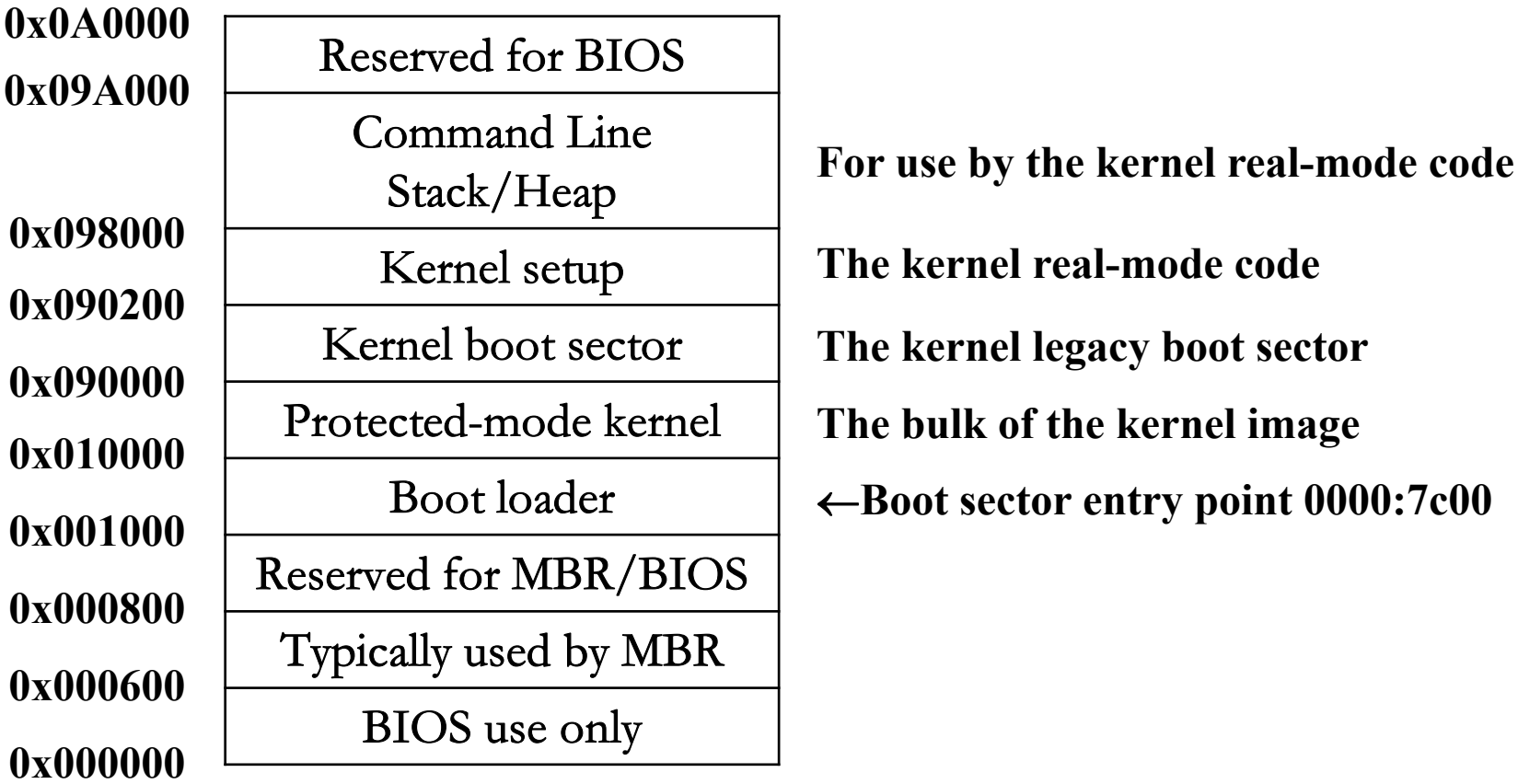
运行主引导程序

4. 在主分区表中搜索标志为活动的分区。如果发现没有活动分区或者不止一个活动分区，则停止；
5. 将活动分区的第一个扇区读入内存地址0x7c00处；
6. 检查位于地址0x7dfe的（WORD内容）是否等于0xaa55，若不等于则显示“Missing Operating System”，然后停止，或尝试软盘启动；
7. 跳转到0x7c00处继续执行特定系统的启动程序；

- PC机最早是由IBM生产，使用的是Intel 8088处理器。这个处理器只有20根地址线，可以寻址1M的空间。这1M空间大概有如下的结构：

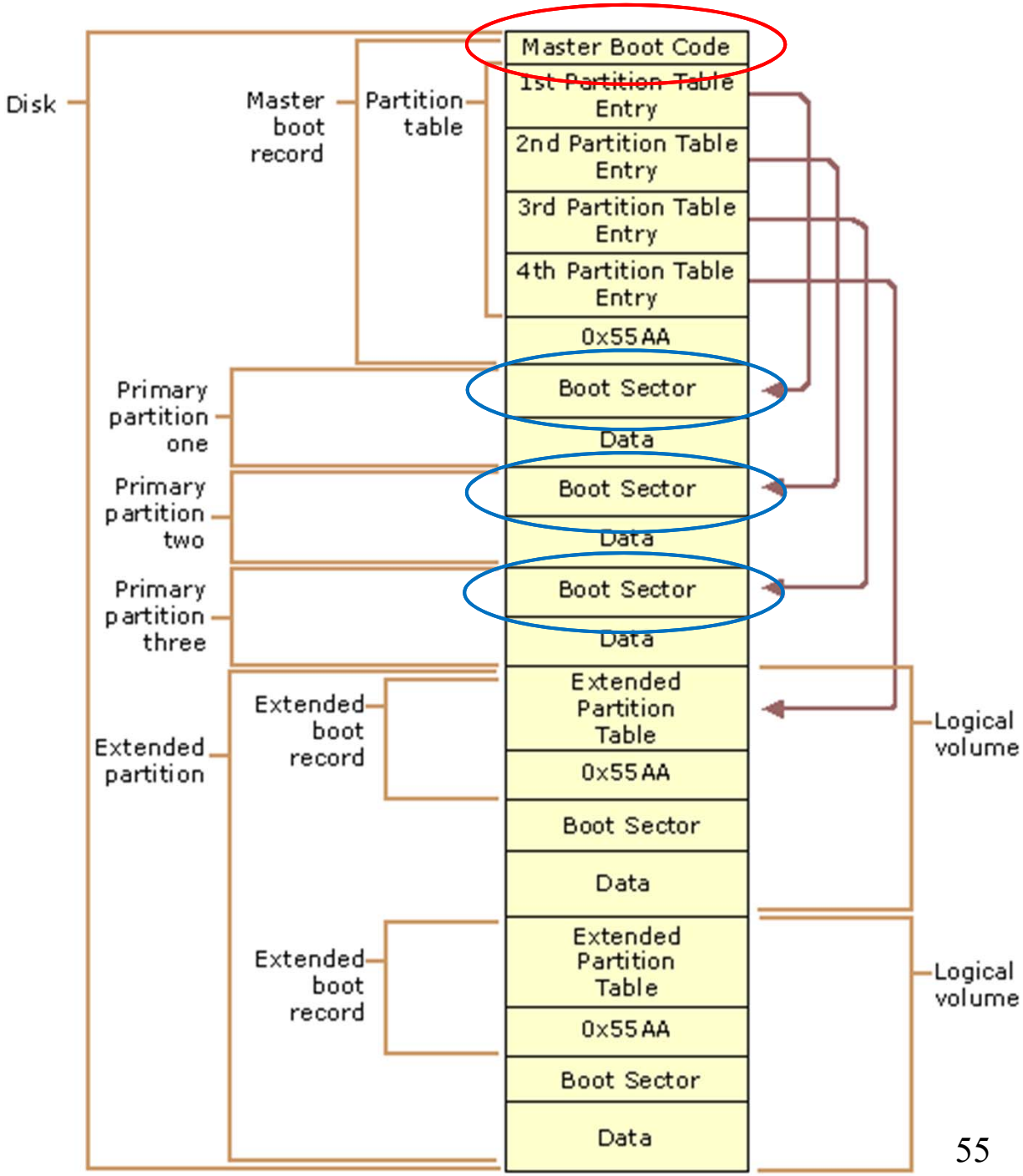
```
+-------------------+  <- 0x00100000 (1MB)
|     BIOS ROM      |
+-------------------+  <- 0x000F0000 (960KB)
|  16-bit devices,  |
|  expansion ROMs   |
+-------------------+  <- 0x000C0000 (768KB)
|   VGA Display     |
+-------------------+  <- 0x000A0000 (640KB)
|                   |
|    Low Memory     |
|                   |
+-------------------+  <- 0x00000000
```

传统支持zImage内核的启动装载内存布局



MBR与引导扇区Boot Sector的关系

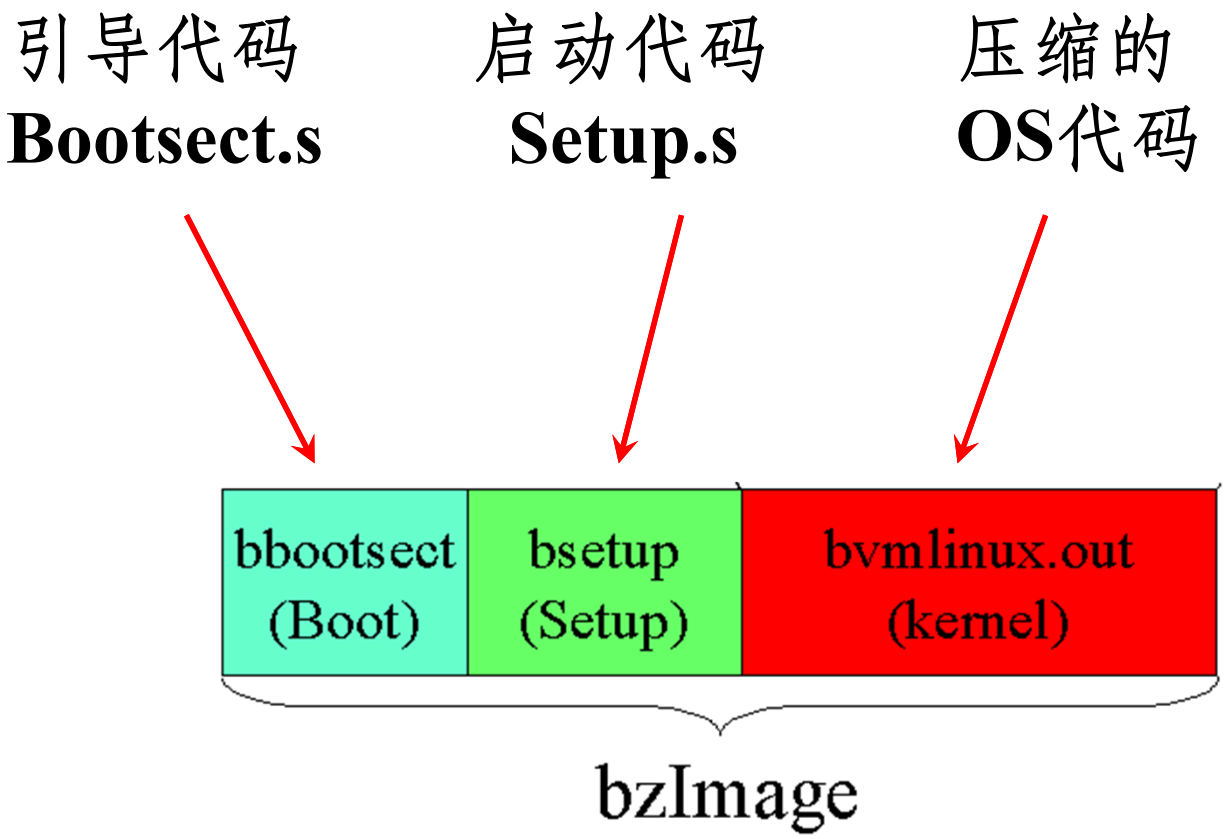
- **MBR**存放的位置是整个硬盘第一个扇区。
- **Boot Sector**是硬盘上每个分区的第一个扇区。



Kernel image

- The kernel is the central part in most computer operating systems because of its task, which is the management of the system's resources and the communication between hardware and software components.
- Kernel is always store on memory until computer is turn off.
- Kernel image is not an executable kernel, but a compress kernel image.
  - zImage size less than 512 KB
  - bzImage size greater than 512 KB

Kernel Image



Task of kernel

- 资源管理
  - Process management
  - Memory management
  - Device management

- 用户服务
  - System call

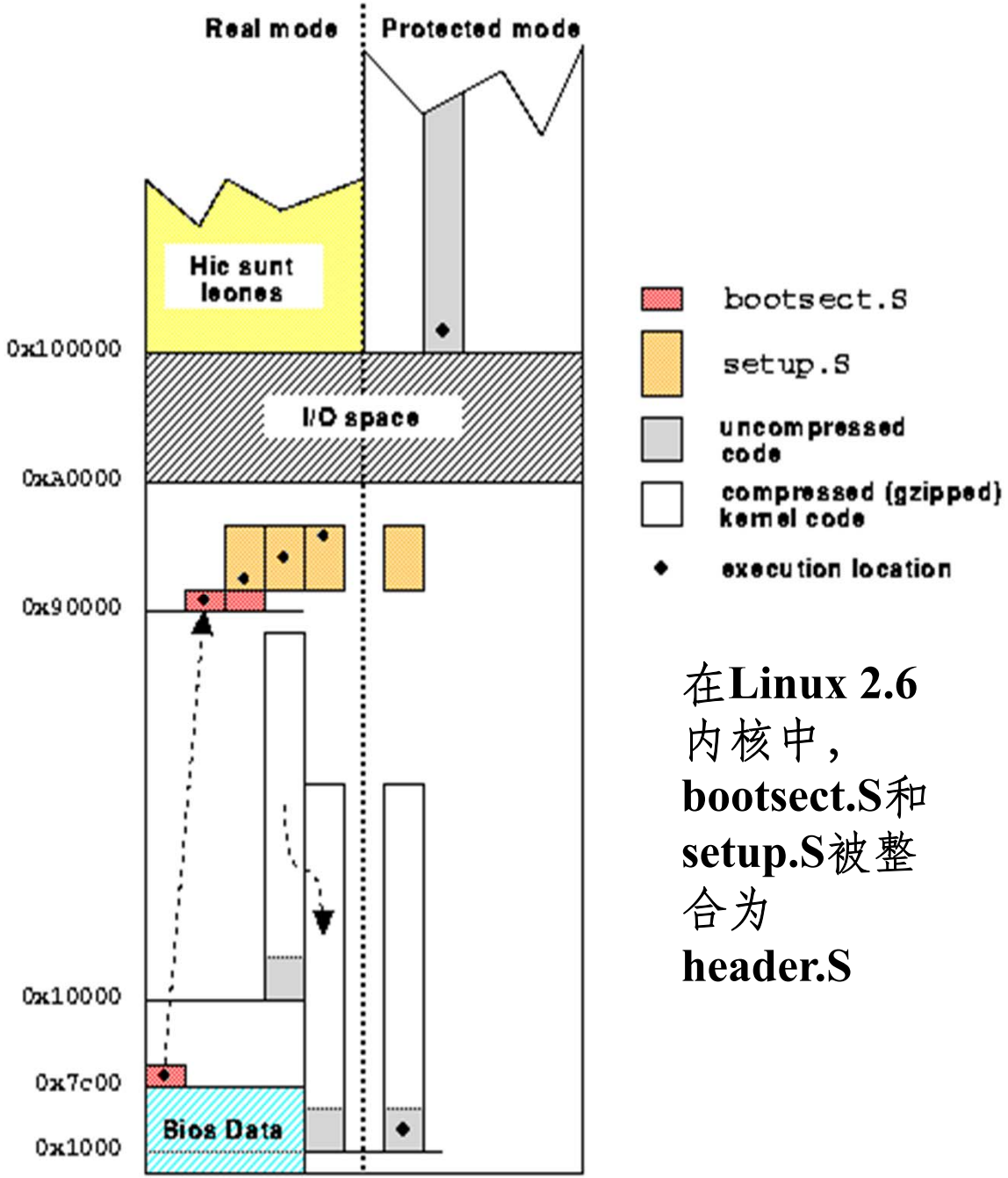
启动第四步——加载内核

- 根据grub设定的内核映像所在路径，系统读取内存映像，并进行解压缩操作。
- 系统将解压后的内核放置在内存之中，初始化函数并初始化各种设备，完成Linux核心环境的建立。

Linux Kernel 加载过程

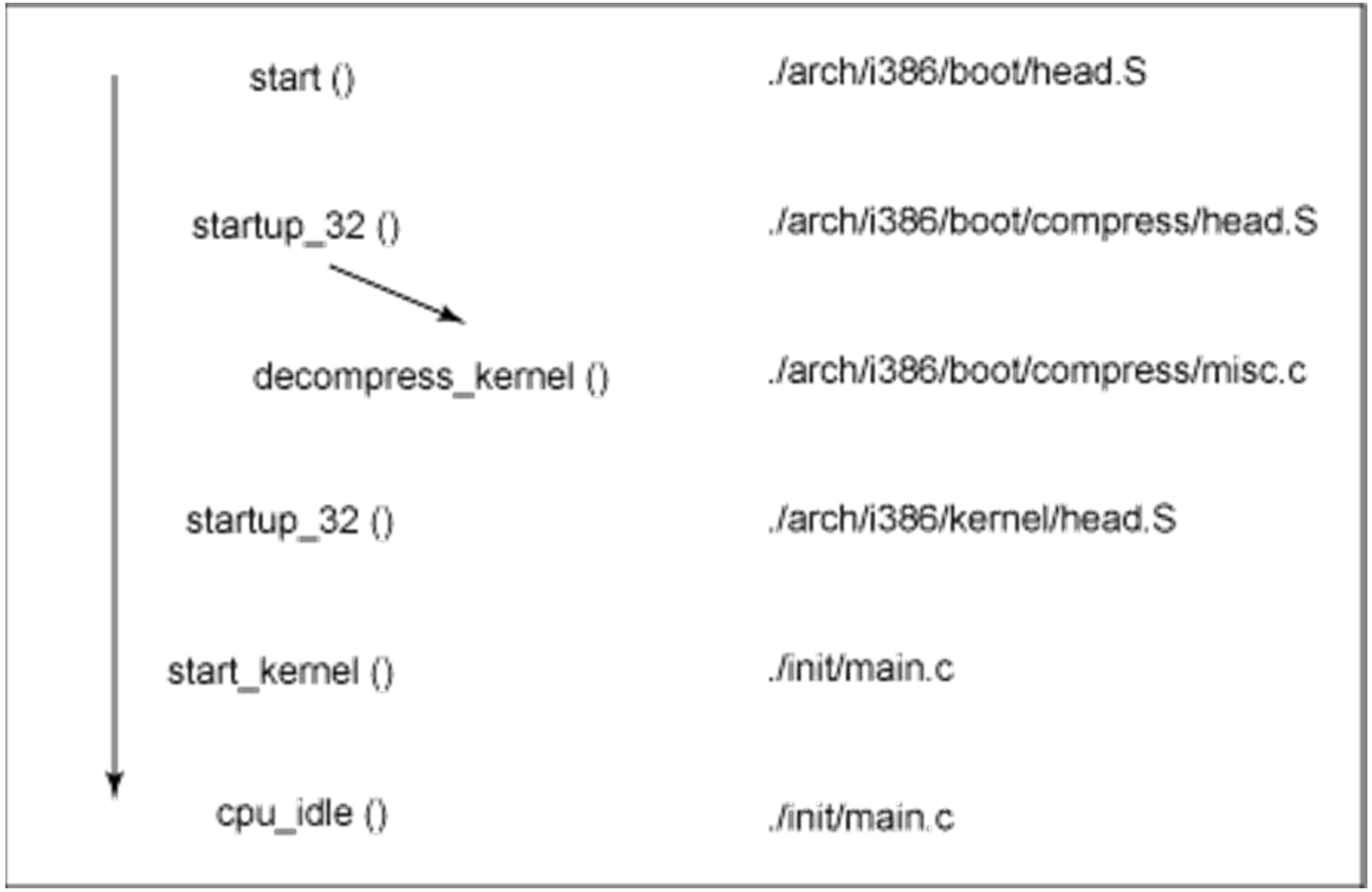
- 不断装载下一段可执行代码
  - 扇区拷贝
  - 支持文件系统
  - 设置内存
  - 解压缩
  - 切换CPU模式
  - …

https://en.wikipedia.org/wiki/Vmlinux



在Linux 2.6内核中，bootsect.S和setup.S被整合为header.S

Major functions flow for Linux kernel boot



- bootsect.S :装载系统启动过程并设置系统启动过程的相关参数.
- setup.S　:初始化系统及硬件, 并切换至保护模式.
- video.S　:初始化显示设备.

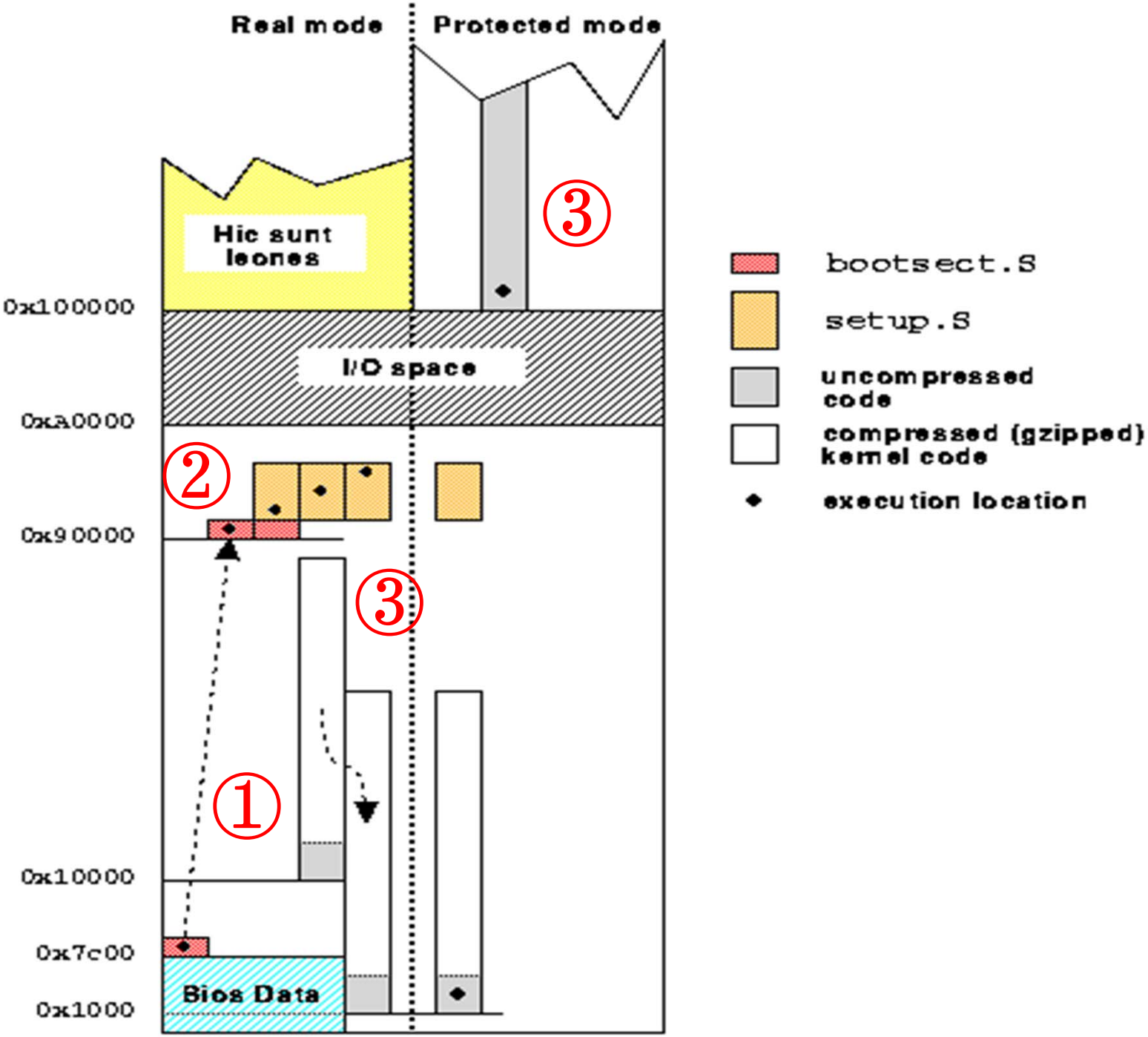
Bootsect.s

① boot过程首先将自身从原始启动区0x7c00—0x7dff移至0x90000—0x901ff,并跳至下一条指令。

② 读引导扇区的后4个扇区到0x90200(即setup区的头4个扇区)。设置一些参数：如堆栈(栈基址9000:3ff4). 新的磁盘参数表等

③ 加载内核映像：如果是大内核，加载到0x100000（1M以上高端内存），否则加载到0x010000（64K低端内存）。

- 跳转到Setup的入口点（即转到地址为0x90020的地址处执行，也就是开始执行setup部分的代码了）。


Real mode
Protected mode
Hic sunt leones
0x100000
I/O space
0xA0000
0x90000
0x10000
0x7c00
0x1000
Bios Data
①
②
③
③
bootsect.S
setup.S
uncompressed code
compressed (gzipped) kernel code
execution location

几个小问题

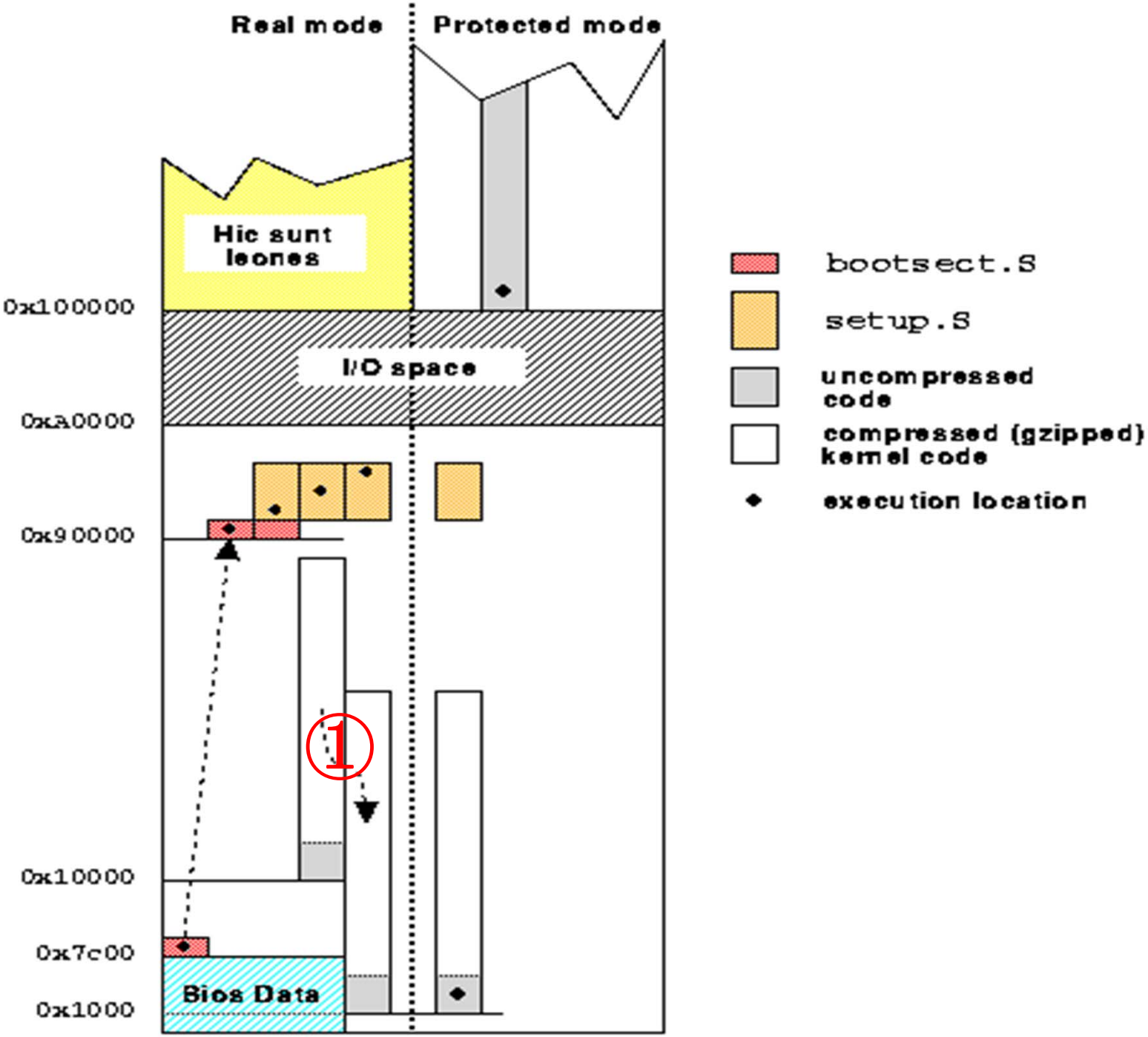
- 为什么需要磁盘引导程序
  - 为什么需要磁盘引导程序而不是直接载入操作系统？或许是因为历史原因，BIOS最初是为最原始的8位PC所创，8位的PC磁盘很小，所以BIOS只能先载入磁盘引导程序然后通过磁盘引导程序导入操作系统。
- 磁盘引导程序为什么要从1扇区，0磁头，0磁道加载到7c00H处
  - “引导程序加载器--int19”。由于BIOS加载磁盘引导程序时需要调用系统中断int19来加载引导程序，而这个中断指定了将磁盘1扇区，0磁头，0磁道加载到7c00H处。
- 磁盘引导程序为什么是16位
  - 因为DOS系统是16位，X86系统都向后兼容引导时的16位模式。

Setup.S

- 初始化硬件设备。如：调用BIOS例程建立描述系统物理内存布局的表；设置键盘的重复延迟和速率；初始化显卡；检测IBM MCA总线、PS/2鼠标设备、APM BIOS支持等……。

① 如果内核加载在低RAM地址0x00010000，则把它移动到0x00001000处；如果映像加载在高内存1M位置，则不动；

- 为内核程序的执行建立环境。如：启动位于8042键盘控制器的A20 pin。建立一个中断描述表IDT和全局描述表GDT表；重启FPU单元；对可编程中断控制器进行重新编程，屏蔽所有中断；设置CR0状态寄存器的PE位使CPU从实模式切换到保护模式，PG位清0，禁止分页功能等……；
- 跳转到startup_32()汇编函数, jmpi 0x100000, __BOOT_CS，终于进入内核Head.S；


Real mode
Protected mode
Hic sunt leones
0x100000
I/O space
0xA0000
0x90000
1
0x10000
0x7c00
Bios Data
0x1000
bootsect.S
setup.S
uncompressed code
compressed (gzipped) kernel code
execution location

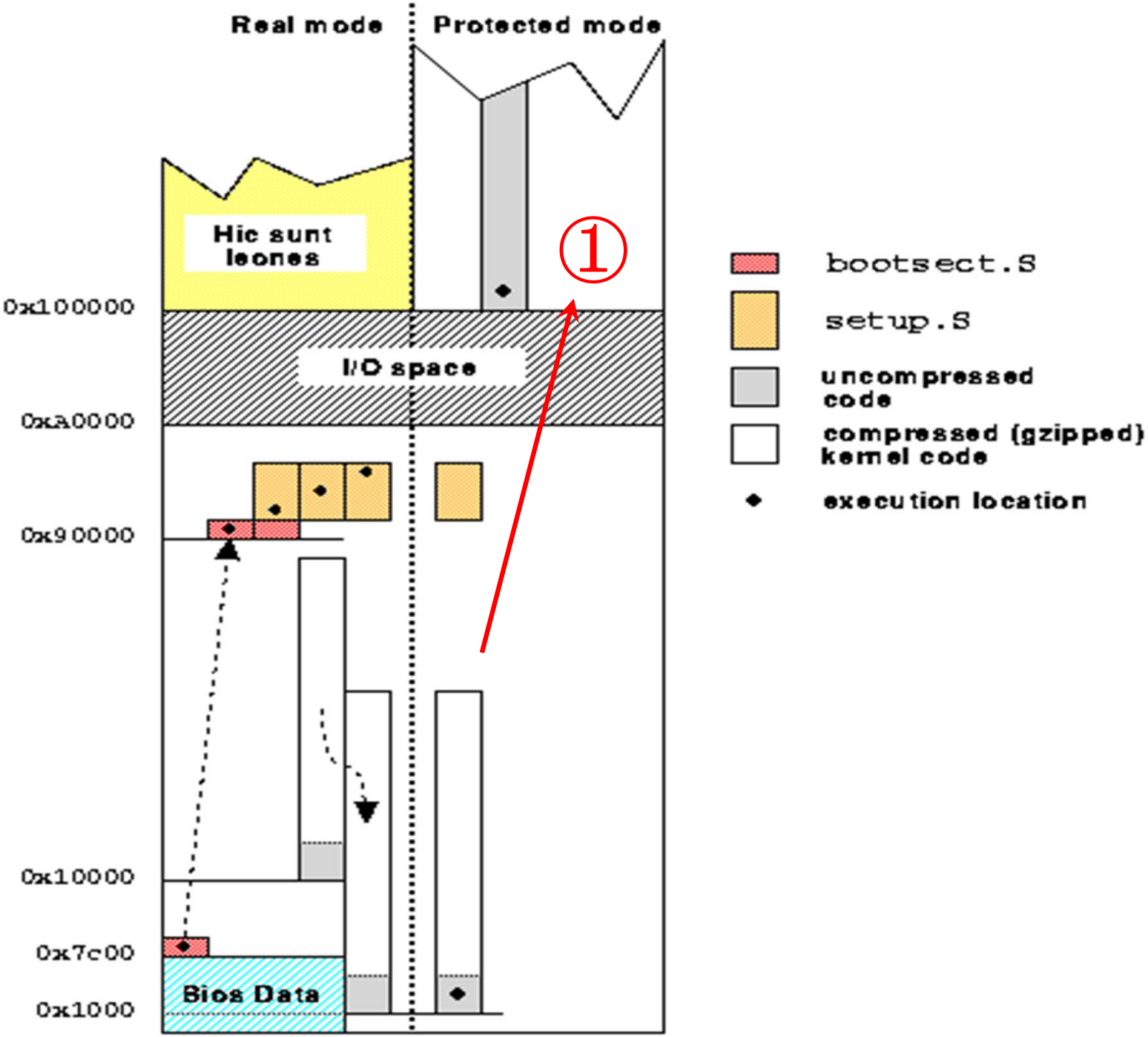
Head.S（第一个start_32()函数）

setup结束后，该函数被放在0x00001000或者0x00100000位置，该函数主要操作：

- 首先初始化段寄存器和临时堆栈；
- 清除eflags寄存器的所有位；
- 将_edata和_end区间的所有内核未初始化区填充0；

① 调用decompress_kernel( )函数解压内核映像。首先显示"Uncompressing Linux..."信息，解压完成后显示 "OK, booting the kernel."。内核解压后，如果时低地址载入，则放在0x00100000位置；否则解压后的映像先放在压缩映像后的临时缓存里，最后解压后的映像被放置到物理位置0x00100000处；

- 跳转到0x00100000物理内存处执行；


Real mode
Protected mode
Hic sunt leones
0x100000
I/O space
0xA0000
0x90000
0x10000
0x7c00
0x1000
Bios Data
①
bootsect.S
setup.S
uncompressed code
compressed (gzipped) kernel code
execution location

Head.s（第二个start_32()函数）

- 解压后的映像开始于arch/i386/kernel/head.S 文件中的startup_32()函数，因为通过物理地址的跳转执行该函数的，所以相同的函数名并没有什么问题。该函数为Linux第一个进程建立(x86)执行环境，操作如下：
  a) 初始化ds,es,fs,gs段寄存器的最终值；
  b) 用0填充内核bss段；
  c) 初始化swapper_pg_dir数组和pg0包含的临时内核页表
  d) 建立进程0idle进程的内核模式的堆栈；
  ……
  x) 跳转到start_kernel函数，这个函数是第一个C编制的函数，内核又有了一个新的开始。

调用Start_kernel()

- 调用start_kernel()函数来启动一系列的初始化函数并初始化核心数据结构和各种设备，完成Linux核心环境的建立：
  - 调度器初始化，调用sched_init();
  - 调用build_all_zonelists函数初始化内存区；
  - 调用page_alloc_init()和mem_init()初始化伙伴系统分配器；
  - 调用trap_init()和init_IRQ()对中断控制表IDT进行最后的初始化；
  - 调用softirq_init() 初始化TASKLET_SOFTIRQ和HI_SOFTIRQ；
  - Time_init()对系统日期和时间进行初始化；
  - 调用kmem_cache_init()初始化slab分配器；
  - 调用calibrate_delay()计算CPU时钟频率；

至此，Linux内核已经建立起来了，基于Linux的程序应该可以正常运行了。

启动第五步——用户层init依据inittab文件来设定运行等级

- 内核被加载后，第一个运行的程序便是**/sbin/init**，该文件会读取**/etc/inittab**文件，并依据此文件来进行初始化工作。
- **/etc/inittab**文件最主要的作用就是设定**Linux**的运行等级，其设定形式是“**:id:5:initdefault:**”，这就表明**Linux**需要运行在等级**5**上。

Linux的运行等级

Linux的运行等级设定如下：

- 0：关机
- 1：单用户模式
- 2：无网络支持的多用户模式
- 3：有网络支持的多用户模式
- 4：保留，未使用
- 5：有网络支持有X-Window支持的多用户模式
- 6：重新引导系统，即重启

启动第六步－－init进程执行rc.sysinit

- 在设定了运行等级后，**Linux**系统执行的第一个用户层文件就是**/etc/rc.d/rc.sysinit**脚本程序，它做的工作非常多，包括设定**PATH**、设定网络配置（**/etc/sysconfig/network**）、启动**swap**分区、设定**/proc**等等。
- 如果你有兴趣，可以到**/etc/rc.d**中查看一下**rc.sysinit**文件，里面的脚本够你看的☺

启动第七步－－启动内核模块

- 具体是依据**/etc/modules.conf**文件或**/etc/modules.d**目录下的文件来装载内核模块。

启动第八步——执行不同运行级别的脚本程序

- 根据运行级别的不同，系统会运行**rc0.d**到**rc6.d**中的相应的脚本程序，来完成相应的初始化工作和启动相应的服务。

启动第九步－－执行/etc/rc.d/rc.local

- 你如果打开了此文件，里面有一句话，读过之后，你就会对此命令的作用一目了然：

  # This script will be executed *after* all the other init scripts.

  # You can put your own initialization stuff in here if you don't

  # want to do the full Sys V style init stuff.

- **rc.local**就是在一切初始化工作后，**Linux**留给用户进行个性化的地方。你可以把你想设置和启动的东西放到这里。

启动第十步－－执行/bin/login程序，进入登录状态

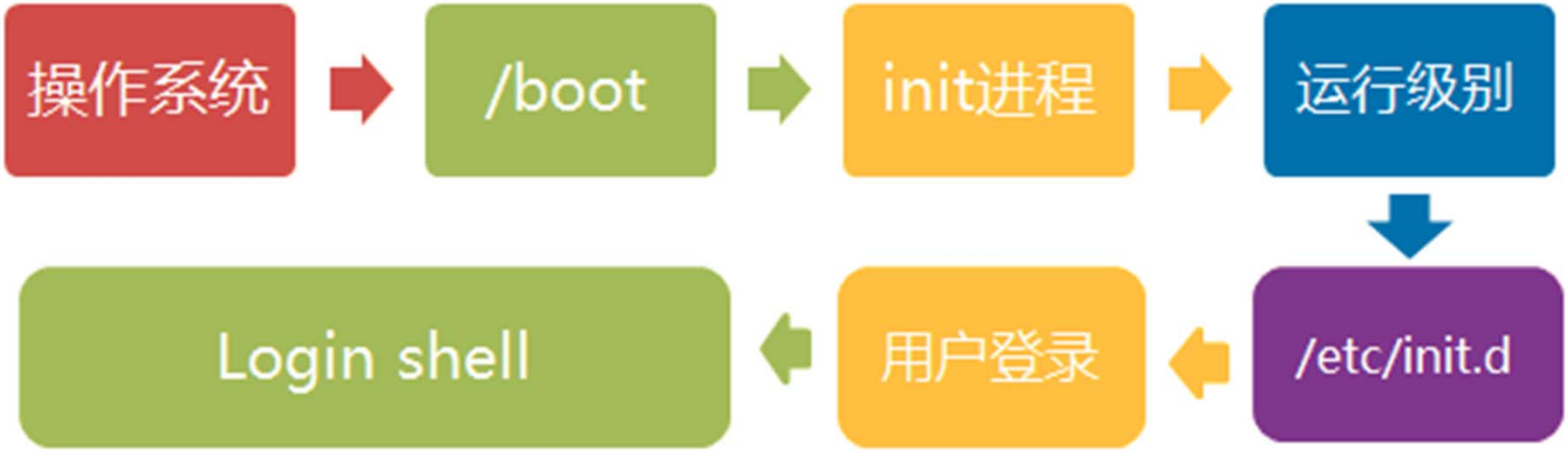
- 此时，系统已经进入到了等待用户输入**username**和**password**的时候了，你已经可以用自己的帐号登入系统了。

漫长的启动过程结束了，一切都清静了……

其实在这背后，还有着更加复杂的底层函数调用，等待着你去研究……

课堂上的内容就算抛砖引玉了

操作系统
/boot
操作系统
/boot
init进程
操作系统
/boot
init进程
运行级别
操作系统
/boot
init进程
运行级别
/etc/init.d
操作系统
/boot
init进程
运行级别
/etc/init.d
用户登录


操作系统
/boot
init进程
运行级别
/etc/init.d
用户登录
Login shell