操作系统　Operating System

第二章 系统引导

沃天宇
woty@buaa.edu.cn
2023年2月27日

Bootstrapping

- Rudolf Erich Raspe
  - 《The Surprising Adventures of Baron Munchausen》（《吹牛大王历险记》）
- 自举：
  - **Baron Munchausen pulls himself (and his horse) out of a swamp by his hair.**

underwater

riding the cannon ball

flying with ducks

How computer startup?

- Booting is a bootstrapping process that starts operating systems when the user turns on a computer system
- A boot sequence is the set of operations the computer performs when it is switched on that load an operating system

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模式（*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特定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*）
  - e.g. 启动磁盘上的OS，需要先有磁盘驱动程序，而磁盘驱动程序是OS的一部分
- 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

借助外力“创生”

- 早期汽车的启动过程
  - 汽油机需要对压缩的燃料+空气点火后才能工作
  - 只有汽油机工作了才能压缩燃料+空气

- 现在汽车的启动过程
  - 借助外力：人力、电机

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

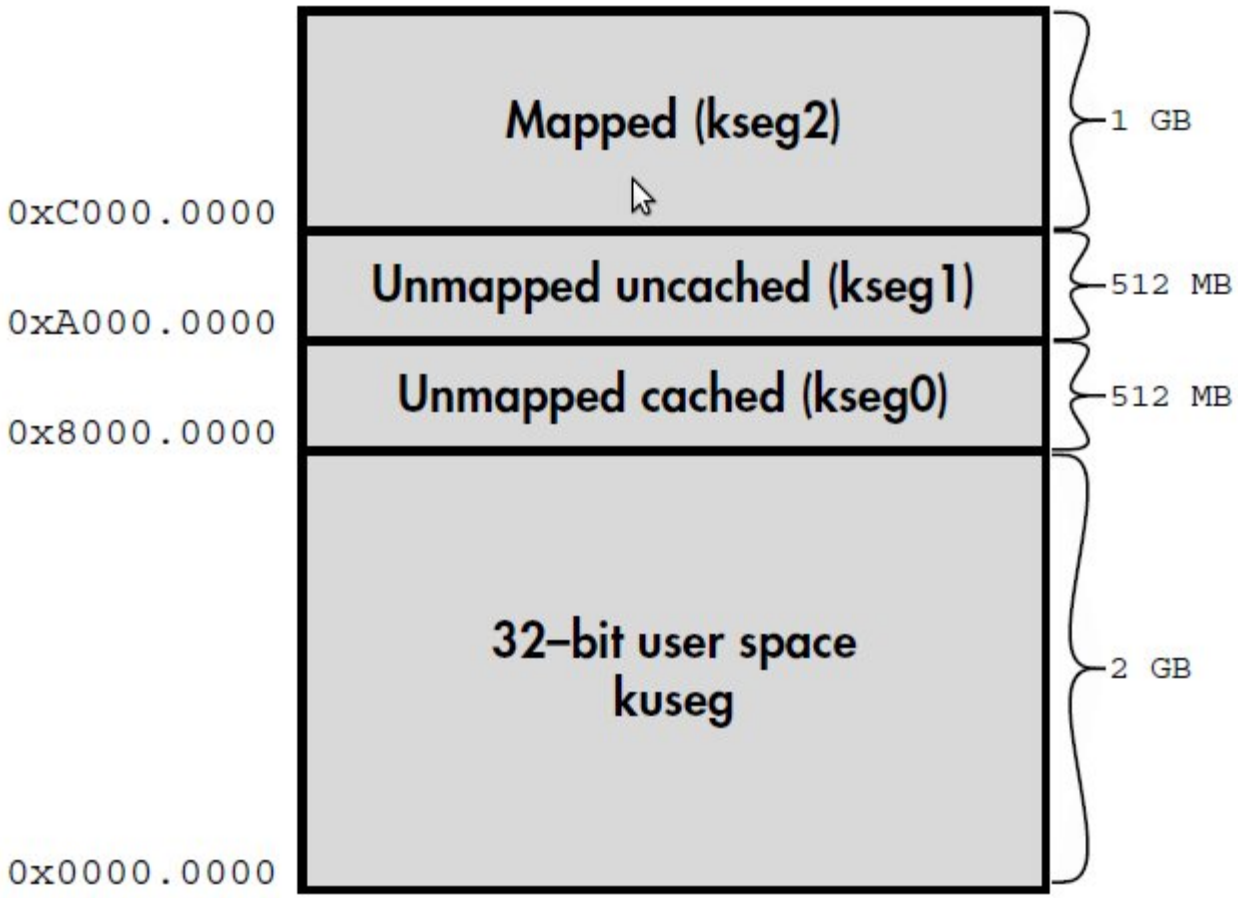
- 开放源码；**https://github.com/u-boot/u-boot**
- 支持多种嵌入式操作系统内核，如Linux、NetBSD, VxWorks, QNX, RTEMS, ARTOS, LynxOS, android；
- 支持多个处理器系列，如PowerPC、ARM、x86、MIPS；
- 较高的可靠性和稳定性；
- 高度灵活的功能设置，适合U-Boot调试、操作系统不同引导要求、产品发布等；
- 丰富的设备驱动源码，如串口、以太网、SDRAM、FLASH、LCD、NVRAM、EEPROM、RTC、键盘等；
- 较为丰富的开发调试文档与强大的网络技术支持；

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

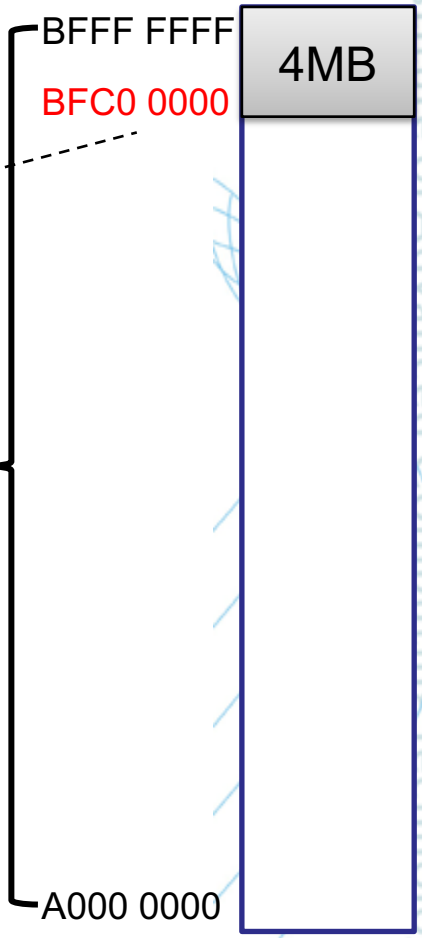
- kuseg: 这些地址是用户态可用的地址,在有MMU的机器里,这些地址将一概被MMU作转换,除非MMU的设置被建立好,否则这2G的地址是不可用的.
- kseg0: 将他们的最高位清零,即可映射到物理地址段512M(0x000 00000 -- 0x1FFF FFFF).这种映射关系很简单,通常称之为"非转换的"地址区域,几乎全部对这段地址的存取都会通过cache,因此cache设置好之前,不能随便使用这段地址.
  - 通常一个没有MMU的系统会使用这段地址作为其绝大多数程序和数据的存放位置;
  - 对于有MMU的系统,操作系统核心会存放在这个区域.

MIPS的基本地址空间

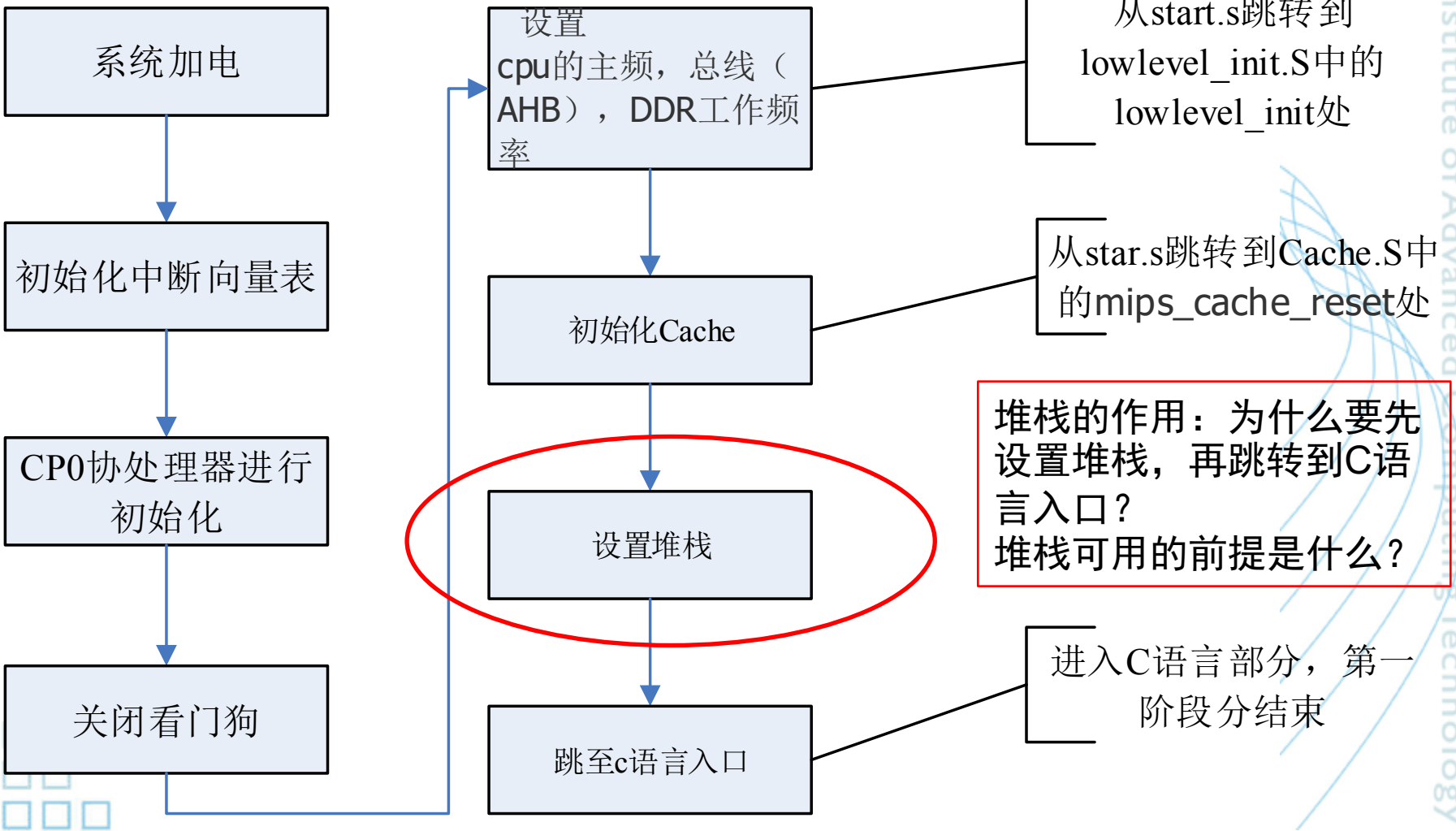
- kseg1：将这些地址的高三位清零可映射到相应的物理地址上,与kseg0映射的物理地址一样,但kseg1是非cache存取的．kseg1是唯一在系统重启时能正常工作的地址空间. (WHY?)
- kseg2：这块区域只能在核心态下使用并且要经过MMU的转换．在MMU设置好之前,不要存取该区域．除非在写一个真正的操作系统,否则没有理由用kseg2．有时会看到该区域被分为kseg2和kseg3,意在强调低半部分(kseg2)可供运行在管理态的程序使用.

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0xBFC0 0000，通过将最高3位清零（&0x1FFF FFFF）的方法，将ROM所在的地址区映射到物理内存的低端512M(0x0000 0000 - 0x1FFF FFFF)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（0xBFC0 0000）会在这个区域。这个向量对应的物理地址是0x1FC00000。（多大？）



MIPS启动stage1



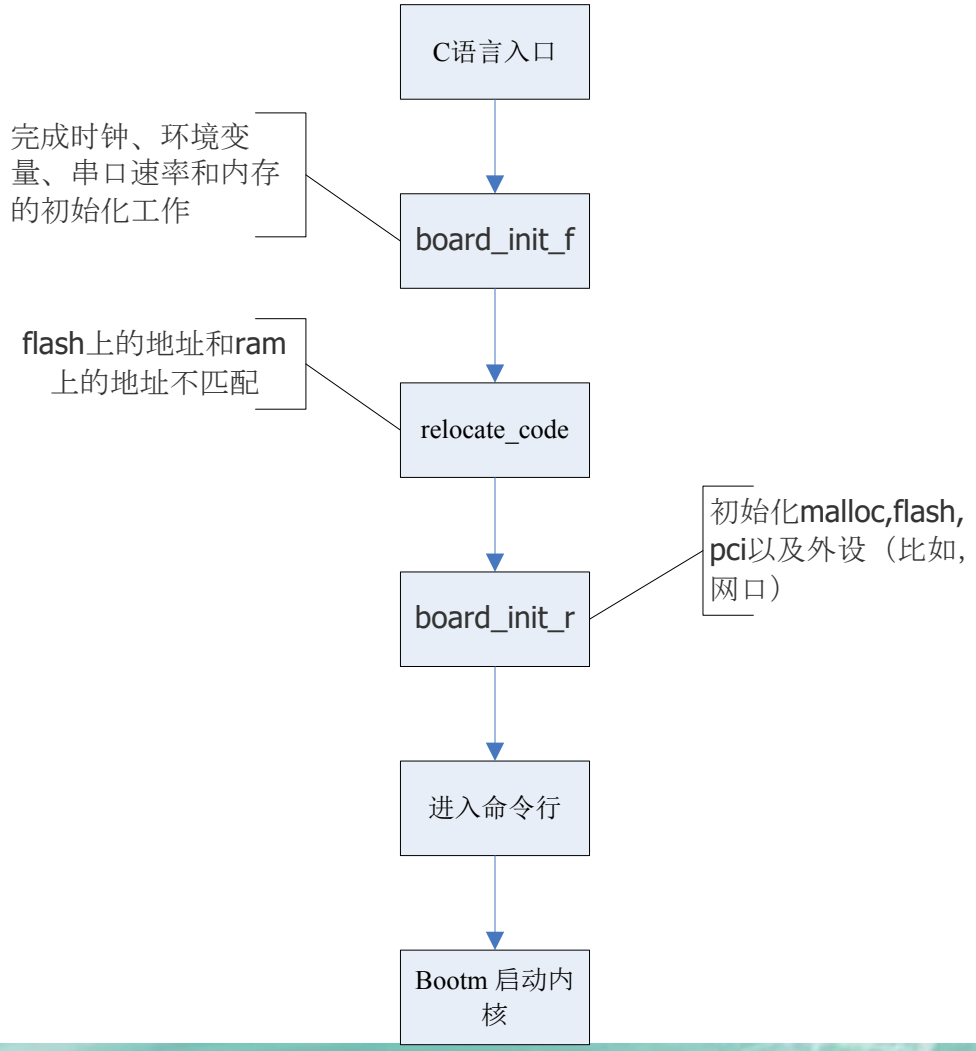
MIPS启动stage2（C代码）

- 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stage2（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。

MIPS启动过程2（C代码）



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。

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/

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

说明

- MIPS的启动过程相对简单，Linux启动的工作除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。
- 本课程MOS实验是在MIPS虚拟机上完成，它比真实的MIPS更简单；
- MOS是一个简单的教学型操作系统；

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）

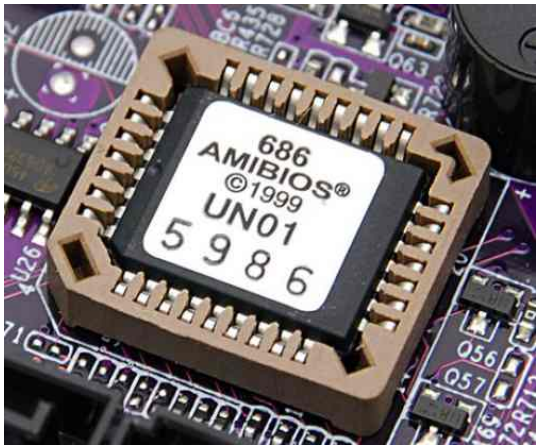
1. Turn on
2. CPU jump to physical address of BIOS (0xFFFF0) （Intel 80386）
3. BIOS runs POST (Power-On Self Test)
4. Find bootable devices
5. Loads boot sector from MBR
6. BIOS yields control to OS BootLoader

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

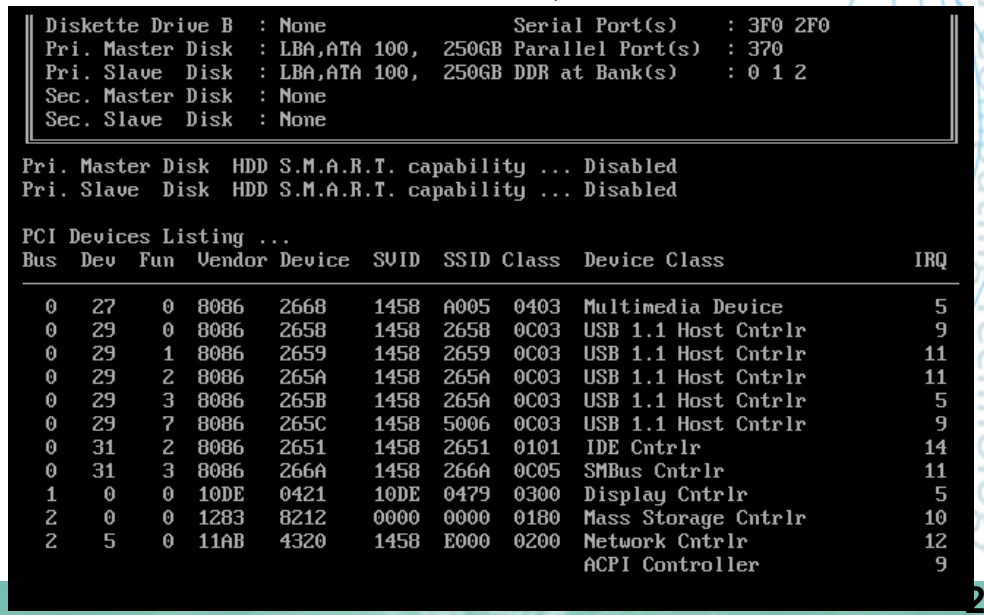
- 当打开计算机电源，计算机会首先加载BIOS信息。BIOS中包含了CPU的相关信息、设备启动顺序信息、硬盘信息、内存信息、时钟信息、PnP特性等等。在此之后，计算机心里就有谱了，知道应该去读取哪个硬件设备了。

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：

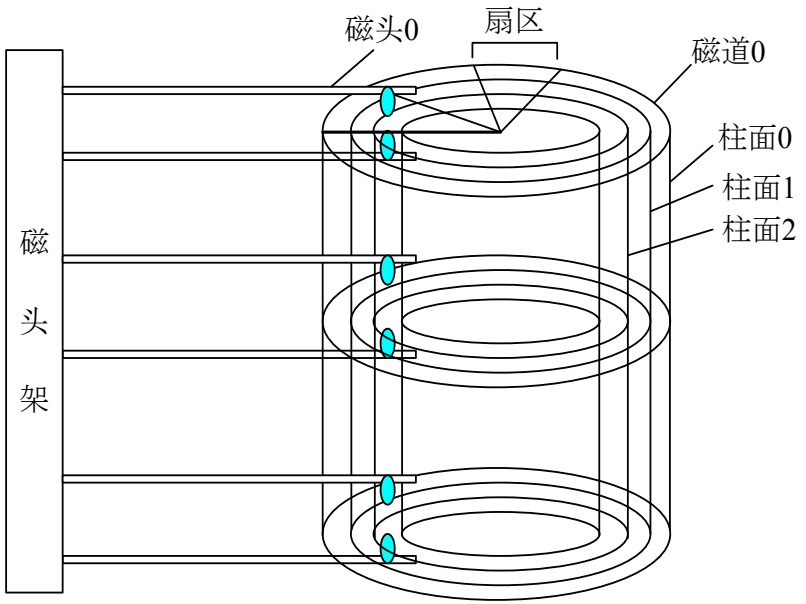
- EFI是用模块化，C语言风格的参数堆栈传递方式，动态链接的形式构建的系统，较BIOS而言更易于实现，容错和纠错特性更强，缩短了系统研发的时间。
- 它运行于32位或64位模式，乃至未来增强的处理器模式下，突破传统BIOS的16位代码的寻址能力，达到处理器的最大寻址。

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)。



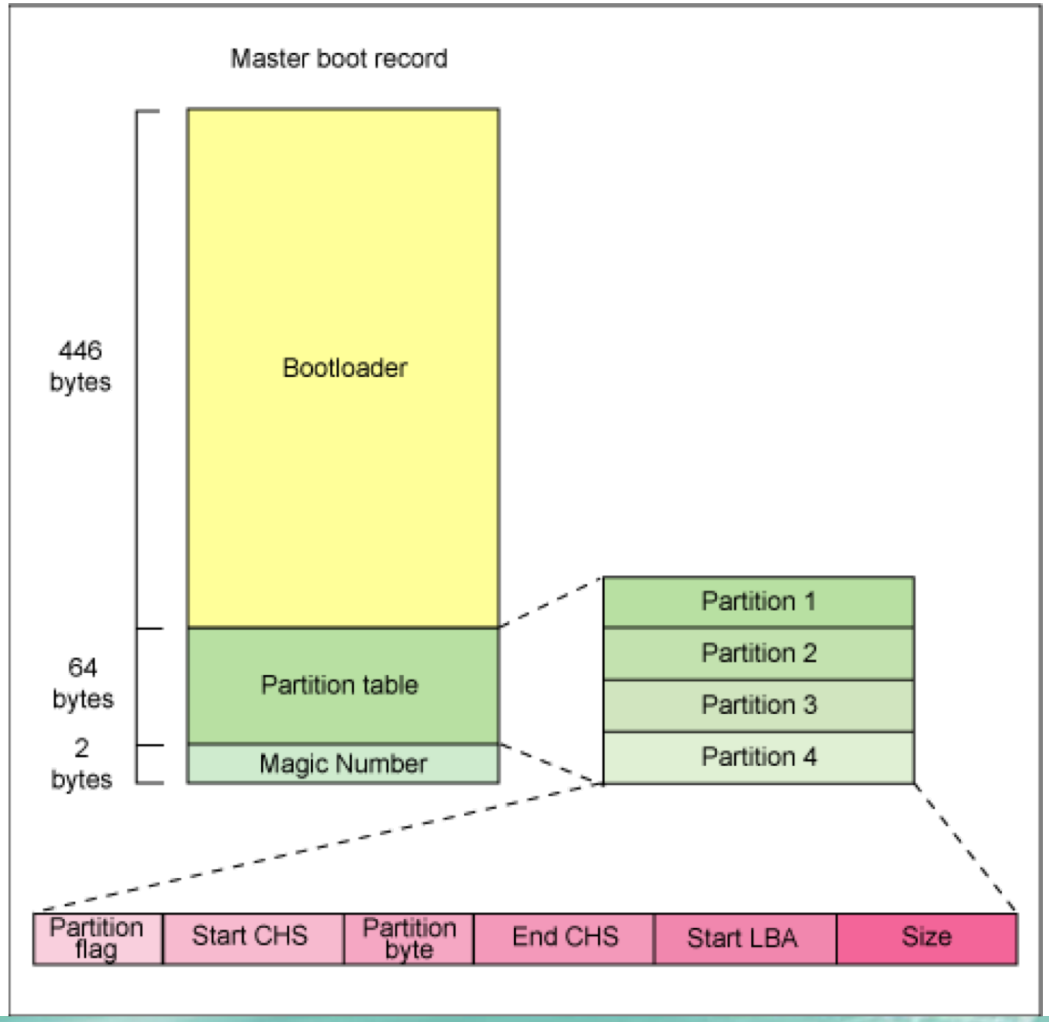
装入MBR

- 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)
- After the MBR is loaded into RAM, the BIOS yields control to it. [How?]

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。

Extracting the MBR

- To see the contents of MBR, use this command:
- # dd if=/dev/hda of=mbr.bin bs=512 count=1
- # od -xa mbr.bin

**The dd command, which needs to be run from root, reads the first 512 bytes from /dev/hda (the first Integrated Drive Electronics, or IDE drive) and writes them to the mbr.bin file.

**The od command prints the binary file in hex and ASCII formats.

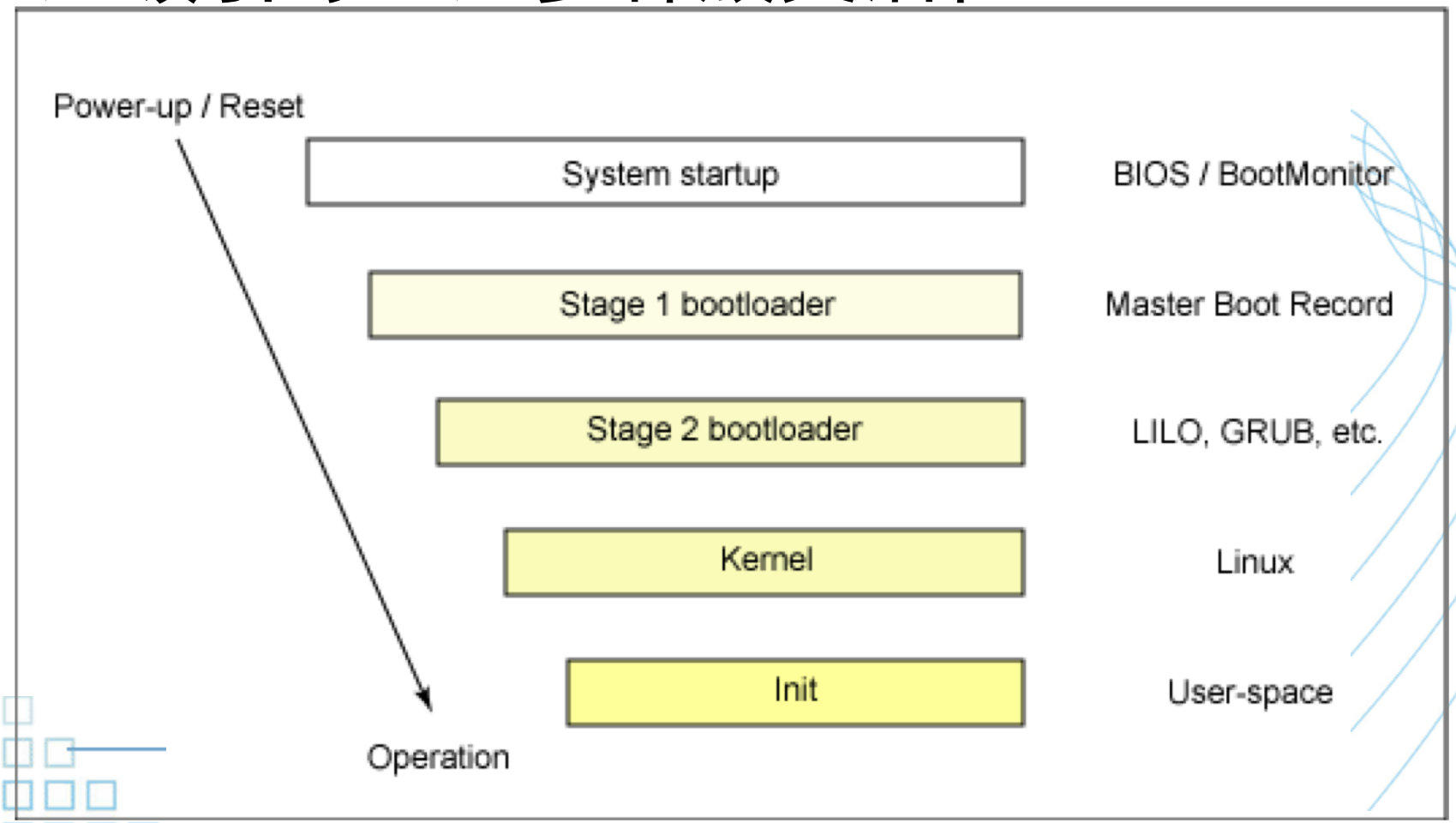
启动及OS引导

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）

Other boot loader (Several OS)

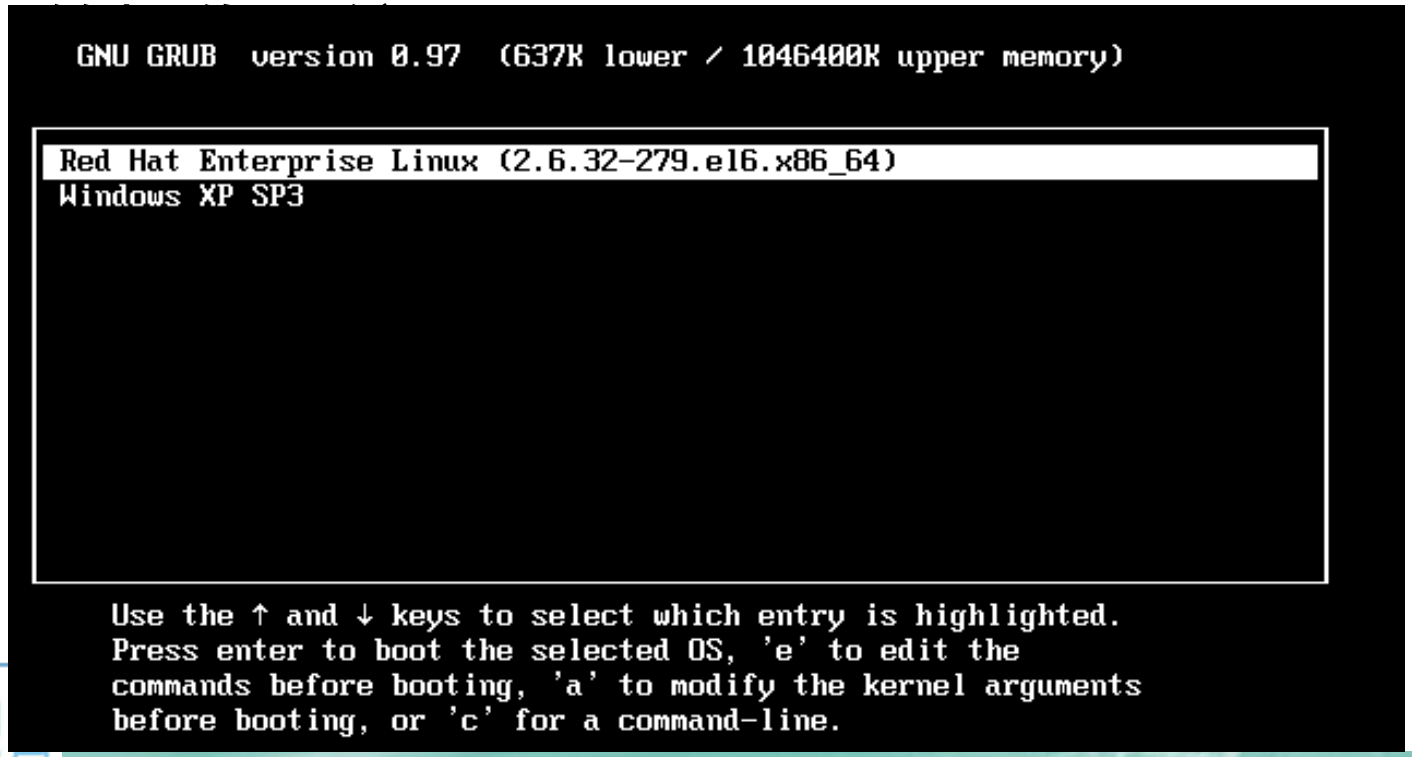
- bootman
- GRUB
- LILO
- NTLDR
- XOSL
- BootX
- loadlin
- Gujin
- Boot Camp
- Syslinux
- GAG
- …

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



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（WHY？）。
- GRUB理解文件系统，可从文件中读取配置
- 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）。

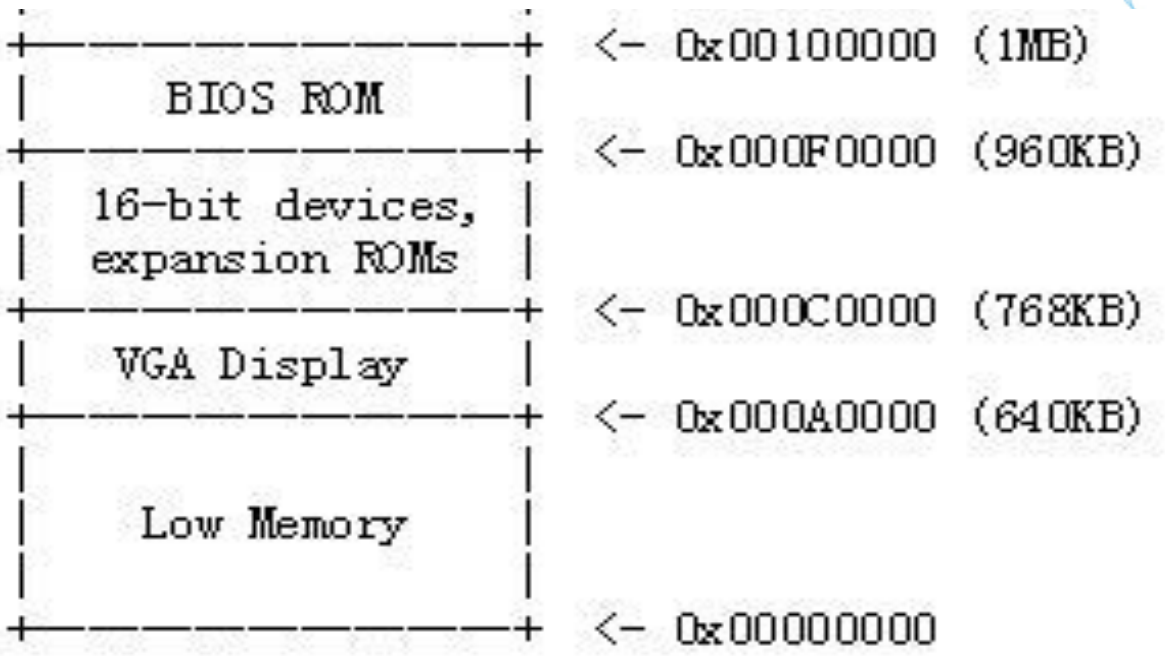
运行主引导程序

- 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

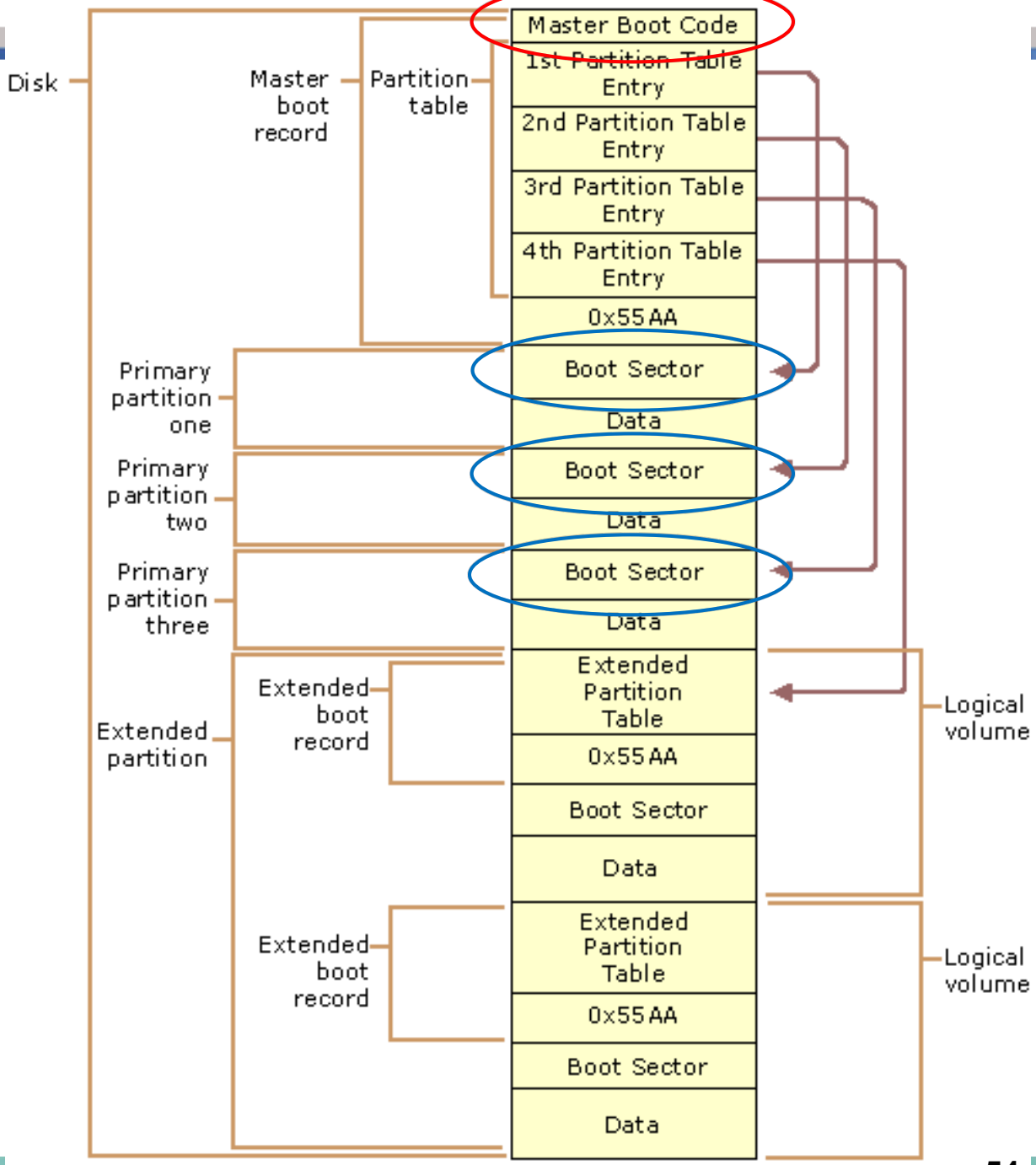
4. 在主分区表中搜索标志为活动的分区。如果发现没有活动分区或者不止一个活动分区，则停止；
5. 将活动分区的第一个扇区读入内存地址0x7c00处；
6. 检查位于地址0x7dfe的（WORD内容）是否等于0xaa55，若不等于则显示 “Missing Operating System”，然后停止，或尝试软盘启动；
7. 跳转到0x7c00处继续执行特定系统的启动程序；

- 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：

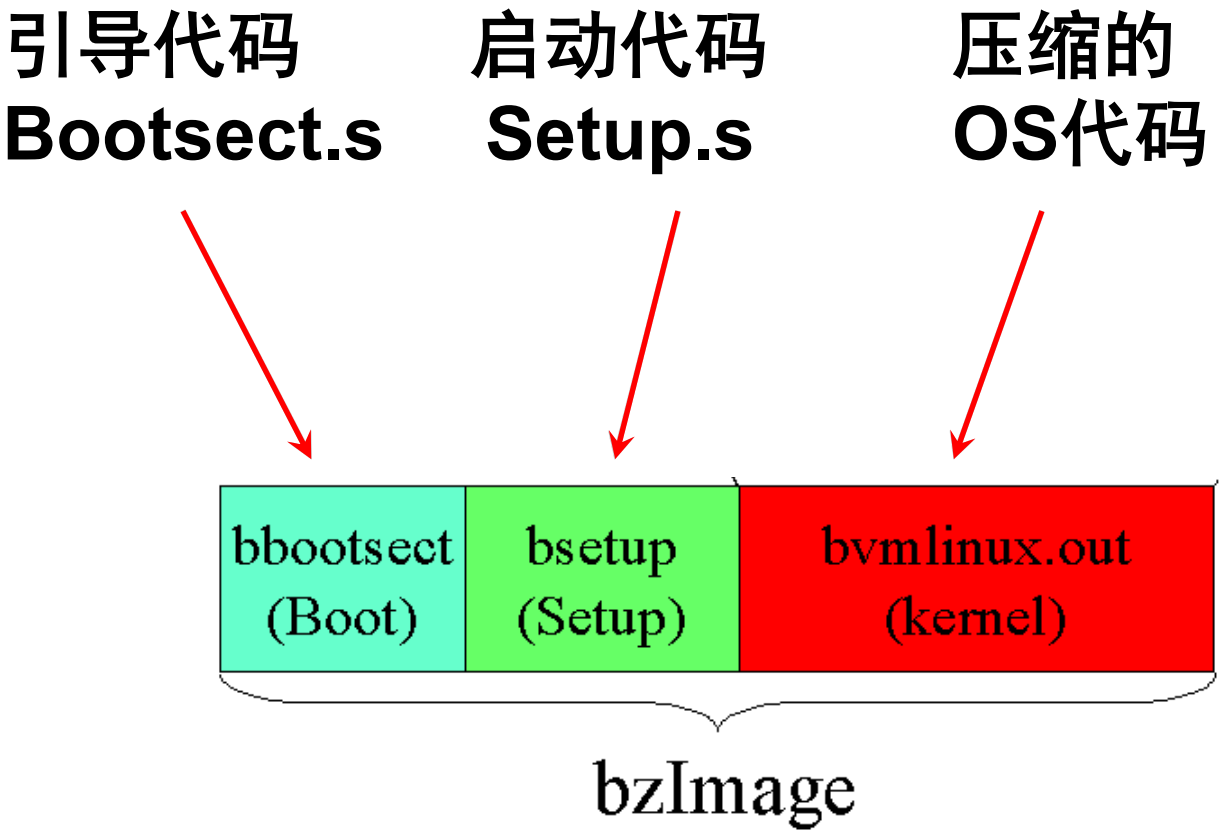


MBR与引导扇区Boot Sector的关系

- **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



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

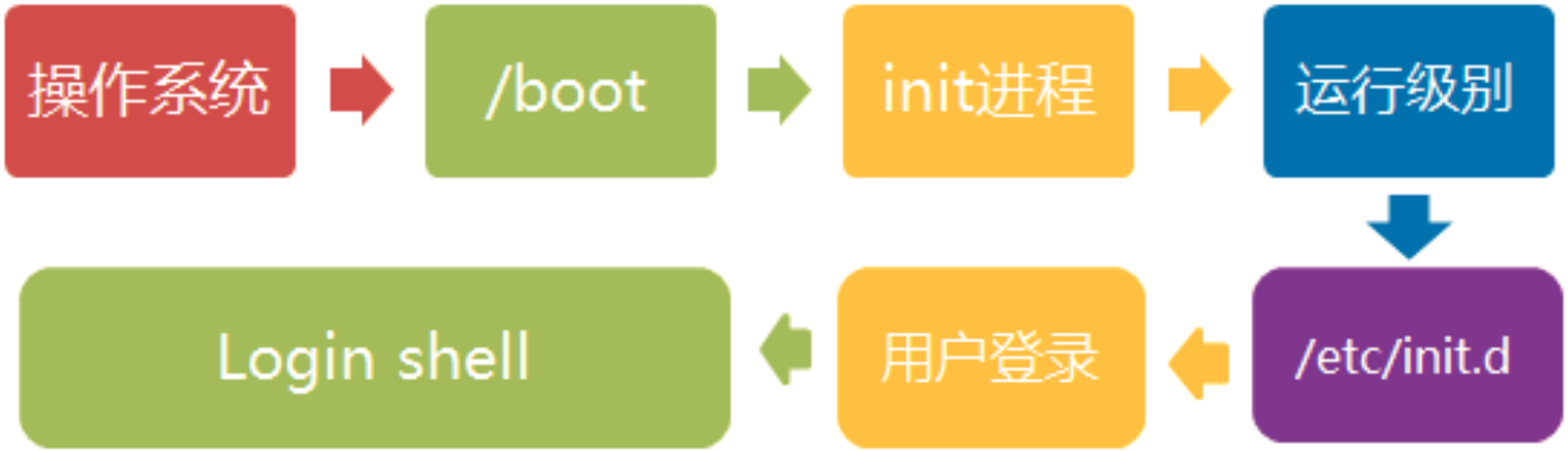
Task of kernel

- 资源管理
  - Process management
  - Memory management
  - Device management
- 用户服务
  - System call

漫长的启动过程结束了……

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……

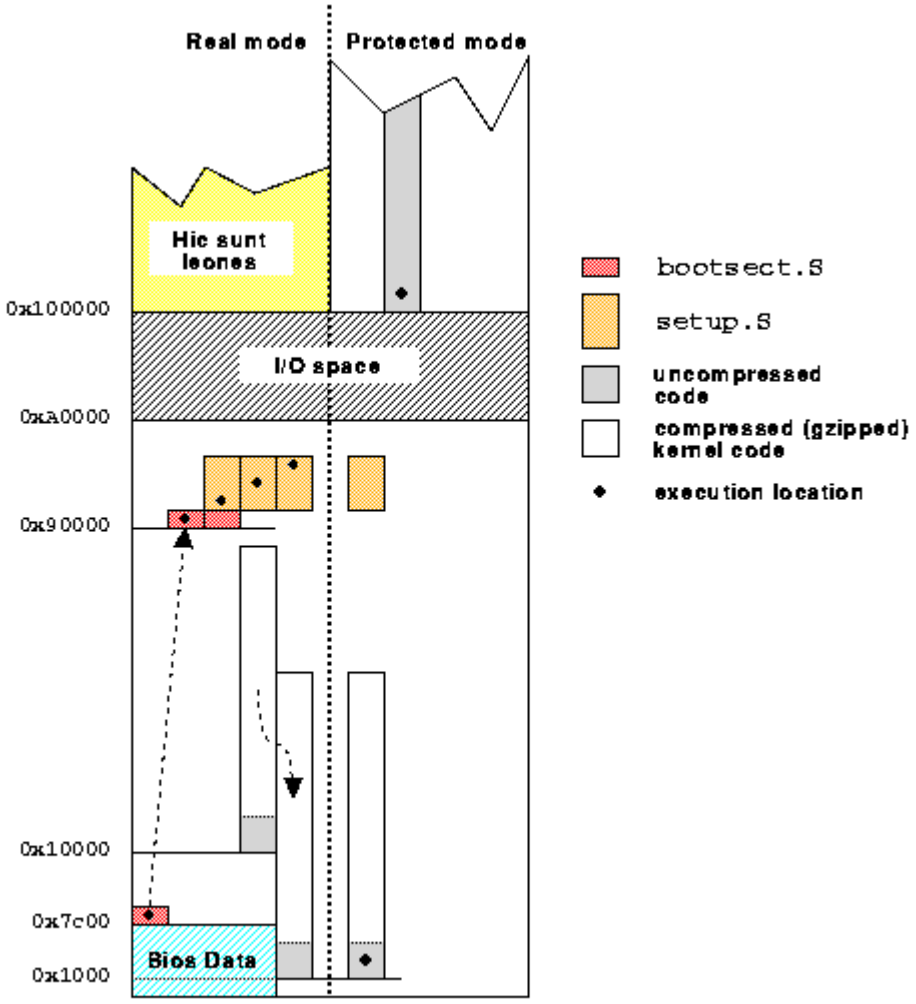
你能否从这些复杂的启动过程中总结出一个最简单的启动需要哪些功能？



详细一些的Linux启动过程

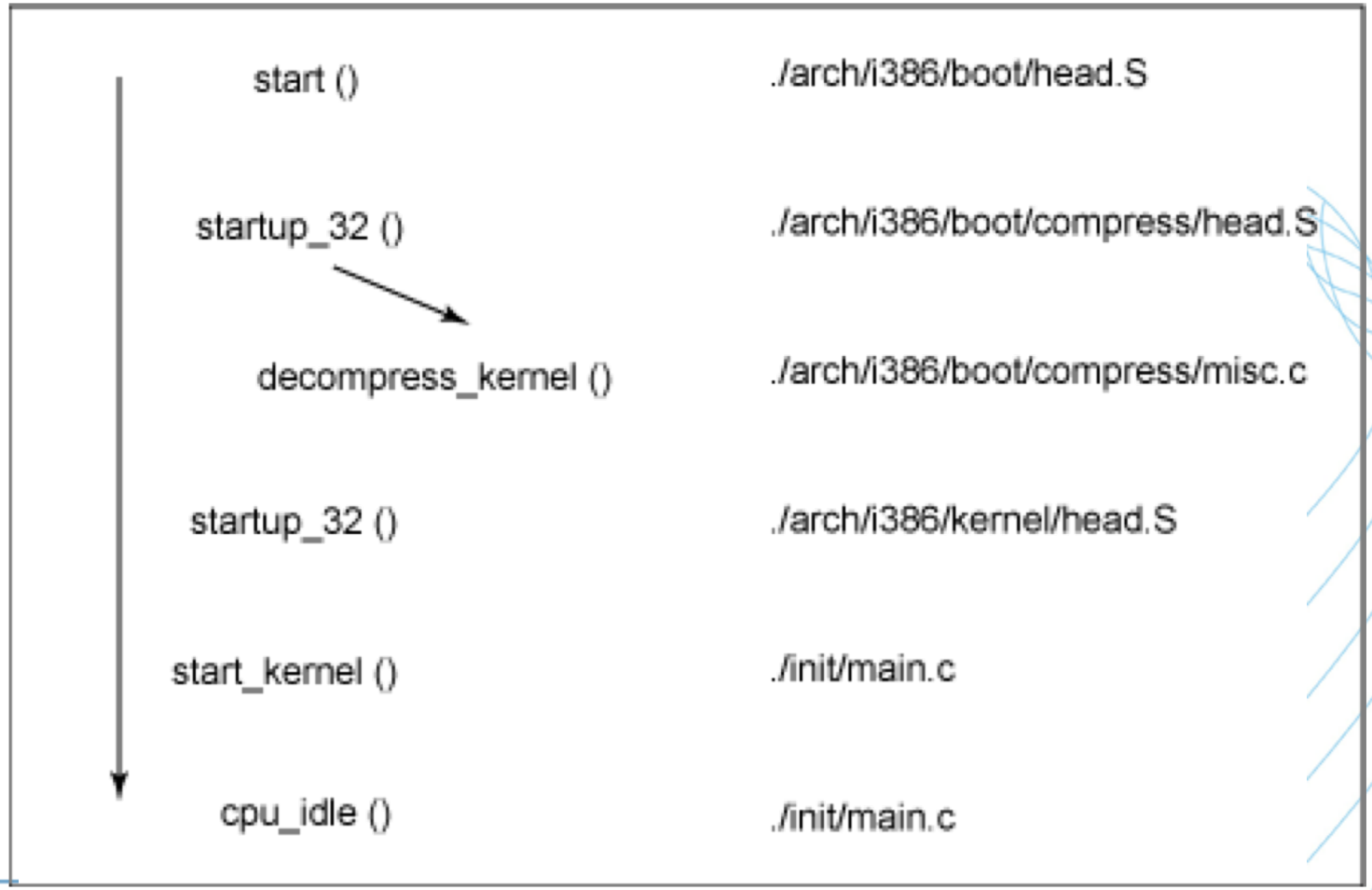
Linux Kernel启动过程

- 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

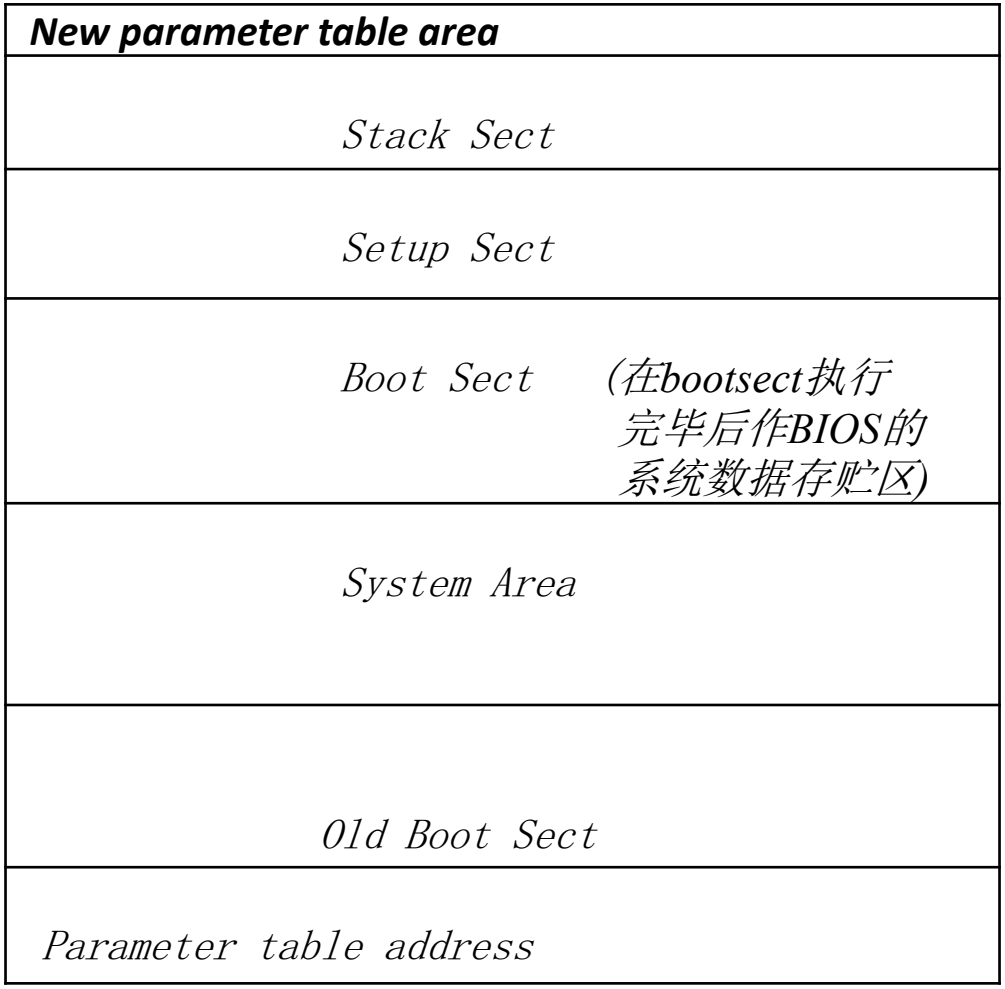
Major functions flow for Linux kernel boot



- 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.

http://tldp.org/HOWTO/Linux-i386-Boot-Code-HOWTO/

重点：规划内存使用



Bootsect.s

- 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(jmpi go,INITSEG ,其中INITSEG=0x9000).
- 设置堆栈(栈基址9000:3ff4).并设置新的磁盘参数表(第一个可用磁盘),把其中的每磁道扇区数设为36(byte 9000:3ff8)，这也是以后一次可读的最大扇区数.
- 重置软盘,并从启动盘读4个扇区到9000:0200(即setup区的头4个扇区).接着测试磁盘的每磁道扇区数（对软盘启动时尤其如此）.
- 装载核心系统于0x10000处(若是big kernel,则调用setup.s中的bootsect_helper).

setup流程

- setup过程首先重置磁盘,检查setup区标志(0xAA55,0x5A5A),若没有则表明setup区大于4个扇区.从system区装载其余扇区到setup区后直至结束,若最后一个扇区没有此标志,则错误.
- 检查是否大内核（big kernel）,若是,则还要检查装载程序（Loader）,老的装载程序不能装载大内核.
- 获取内存大小(主要是扩展内存,放在9000:01e0),(接着往9000:0002存放由ah=0x88,int 15返回的ax值).设置最大击键速度,检查显卡并获取相应参数(调用video过程,在video.S中),存放区9000:0000--9000:0017.
- 获取hd0的数据,存至9000:0080—9000:0090.获取hd1的数据,存至9000:0090—9000:00A0.再检查hd1是否存在,若不,该区域清零.
- 检查是否有PS/2鼠标,若有,9000:01ff置0xaa.检查是否APM_BIOS,其相关参数存于9000:0064—9000:0080区.

- 检查是否有定义的实模式切换过程,否则,使用缺省的过程.然后如果不是大内核（为压缩内核）,则将system移至0100:0000处.
- 检查当前代码是否在9020:0000处,否则,需恢复setup区（反向移动以避免破坏目前正执行代码）.然后把命令行（commandline）512B移至9020:0000,覆盖当前的一部分代码区.
- 装载中断描述表和全局描述表,A20地址线使能,并初始化协处理器及8259外围中断发生器.
- 跳至kernel执行(80x386可通过使用前缀0x66能在16 bit地址下跳至32位代码).　至此,实模式下初始化基本结束,可以放心切换至保护模式.

关于配置装载程序部分（Loader）

- build.c：建立一个磁盘映像，包括：boot区（bootsect.s,521B），setup区（setup.s及video.s部分，2048B），system区（80386代码区，含setup.s剩余部分及kernel的压缩）.该磁盘映像可在软盘上,或存储在hd0的第一个扇区开始。该应用程序对bootsect.s和setup.s的数据区作了修改.
- piggybac.c：该应用程序读入压缩的系统映像并把它装成一个文件.
- extract.c：该应用程序将系统映像解压缩并输出。

保护模式

- setup在把系统核心从start_sys_seg移到0x1000后，装载了在初入保护模式下的GDT和IDT,且GDT设定的系统区基地址为0x00000000，G值为1，即段以页（4KB）为单位长度。
- 然后，使用指令lmsw改变CR0寄存器的低四位为0001，即任务切换，模拟浮点运算和数学运算关闭（TS=0，EM=0，MP=0），保护模式打开（PE=1）。由于CR0的PG位未打开，分页机制未启动，线性地址即物理地址。
- 最后，在当前模式下使用绝对地址跳转至系统核心0x1000处。

head.s

- 即目前系统核心的开始处。它在一获得控制权即装载数据段的选择符(0x18)，设置新堆栈start_stack(位于核心中，在compress/misc.c中定义)，为下面的指令作准备。
- 接着，检查地址线A20是否真的正确打开（循环），重置NT位，初始化BSS区。
- 然后，调用decompress_kernel对kernel进行解压后送入物理地址0x100000处。

/boot/kernel/head.s

- head.s即位于解压后的核心开始处0x100000。它在开始做了与前一个head.s相似的初始化动作：更新数据段寄存器，清BSS区，但接着调用setup_idt建立新的IDT表，然后检查NT位。
- 其后head开始将实模式下得到的系统数据从0x90000——0x907FF和命令行参数区0x90800——0x91000移到0x105000——0x105FFF区中。实际上该区是首先定义的页之一：empty_zero_page页。（关于这一部分的详细介绍参见：页面初始化分析）
- 然后是checkCPUtype过程。
- 其后，head.s检查协处理器的有效并设置标志（call check_x87）。建立初始化时期的4MB页面空间（call setup_paging，其分析见：页面初始化分析）。
- 最后的任务是为下一步的过程建立堆栈数据，并跳转至start_kernel进行核心系统的引导。

体系结构相关

- setup_arch
- Paging_init
- trap_init
- init_IRQ
- time_init
- mem_init
- check_bugs

init process

- The first thing the kernel does is to execute init program
- Init is the root/parent of all processes executing on Linux
- The first processes that init starts is a script /etc/rc.d/rc.sysinit
- Based on the appropriate run-level, scripts are executed to start various processes to run the system and make it functional

The Linux Init Processes

- The init process is identified by process id "1"
- Init is responsible for starting system processes as defined in the /etc/inittab file
- Init typically will start multiple instances of "getty" which waits for console logins which spawn one's user shell process
- Upon shutdown, init controls the sequence and processes for shutdown

System processes

Process ID	Description
0	The Scheduler
1	The init process
2	kflushd
3	kupdate
4	kpiod
5	kswapd
6	mdrecoveryd

inittab file (SysV)

- The inittab file describes which processes are started at bootup and during normal operation
  - /etc/init.d/boot
  - /etc/init.d/rc
- The computer will be booted to the runlevel as defined by the initdefault directive in the /etc/inittab file
  - id:5:initdefault:

Runlevels

- A runlevel is a software configuration of the system which allows only a selected group of processes to exist
- The processes spawned by init for each of these runlevels are defined in the /etc/inittab file
- Init can be in one of eight runlevels: 0-6, S

Runlevels

Runlevel	Scripts Directory (Red Hat/Fedora Core)	State
0	/etc/rc.d/rc0.d/	shutdown/halt system
1	/etc/rc.d/rc1.d/	Single user mode
2	/etc/rc.d/rc2.d/	Multiuser with no network services exported
3	/etc/rc.d/rc3.d/	Default text/console only start. Full multiuser
4	/etc/rc.d/rc4.d/	Reserved for local use. Also X-windows (Slackware/BSD)
5	/etc/rc.d/rc5.d/	XDM X-windows GUI mode (Redhat/System V)
6	/etc/rc.d/rc6.d/	Reboot
s or S		Single user/Maintenance mode (Slackware)
M		Multiuser mode (Slackware)

rc#.d files

- rc#.d files are the scripts for a given run level that run during boot and shutdown
- The scripts are found in the directory /etc/rc.d/rc#.d/ where the symbol # represents the run level

init.d

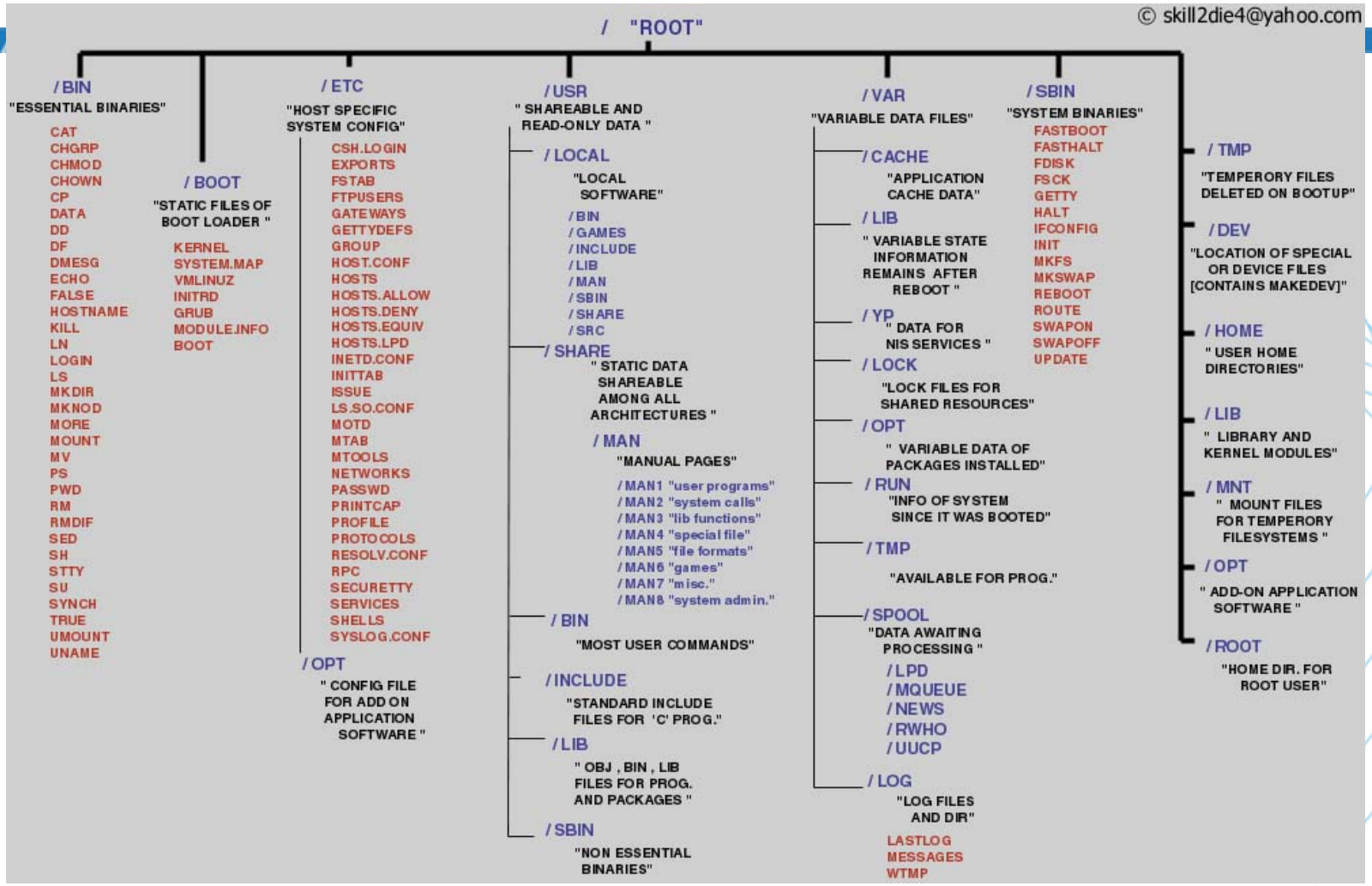
- Deamon is a background process
- init.d is a directory that admin can start/stop individual demons by changing on it
  - /etc/rc.d/init.d/ (Red Hat/Fedora)
  - /etc/init.d/ (SUSE)
  - /etc/init.d/ (Debian)

Start/stop deamon

- Admin can issuing the command and either the start, stop, status, restart or reload option
- i.e. to stop the web server:
  - cd /etc/rc.d/init.d/
  - (or /etc/init.d/ for SUSE and Debian)
  - httpd stop

Linux files structure

Linux files structure



http://www.secguru.com/files/linux_file_structure

FSSTND : (Filesystem standard)

- All directories are grouped under the root entry "/"
- root - The home directory for the root user
- home - Contains the user's home directories along with directories for services
  - ftp
  - HTTP
  - samba

FSSTND : (Filesystem standard)

- bin - Commands needed during booting up that might be needed by normal users
- sbin - Like bin but commands are not intended for normal users. Commands run by LINUX.
- proc - This filesystem is not on a disk. It is a virtual filesystem that exists in the kernels imagination which is memory
  - 1 - A directory with info about process number 1. Each process has a directory below proc.

FSSTND : (Filesystem standard)

- usr - Contains all commands, libraries, man pages, games and static files for normal operation.
  - bin - Almost all user commands. some commands are in /bin or /usr/local/bin.
  - sbin - System admin commands not needed on the root filesystem. e.g., most server programs.
  - include - Header files for the C programming language. Should be below /user/lib for consistency.
  - lib - Unchanging data files for programs and subsystems
  - local - The place for locally installed software and other files.
  - man - Manual pages
  - info - Info documents
  - doc - Documentation
  - tmp
  - X11R6 - The X windows system files. There is a directory similar to usr below this directory.
  - X386 - Like X11R6 but for X11 release 5

FSSTND : (Filesystem standard)

- boot - Files used by the bootstrap loader, LILO. Kernel images are often kept here.
- lib - Shared libraries needed by the programs on the root filesystem
- modules - Loadable kernel modules, especially those needed to boot the system after disasters.
- dev - Device files
- etc - Configuration files specific to the machine.
- skel - When a home directory is created it is initialized with files from this directory
- sysconfig - Files that configure the linux system for devices.

FSSTND : (Filesystem standard)

- var - Contains files that change for mail, news, printers log files, man pages, temp files
  - file
  - lib - Files that change while the system is running normally
  - local - Variable data for programs installed in /usr/local.
  - lock - Lock files. Used by a program to indicate it is using a particular device or file
  - log - Log files from programs such as login and syslog which logs all logins and logouts.
  - run - Files that contain information about the system that is valid until the system is next booted
  - spool - Directories for mail, printer spools, news and other spooled work.
  - tmp - Temporary files that are large or need to exist for longer than they should in /tmp.
  - catman - A cache for man pages that are formatted on demand

FSSTND : (Filesystem standard)

- mnt - Mount points for temporary mounts by the system administrator.
- tmp - Temporary files. Programs running after bootup should use /var/tmp

远程启动

- 客户端在启动前，既无操作系统，又无启动的软盘或者硬盘，它只有计算机的基本部件： CPU, 内存,主板等。 但必须有网卡和启动的BootRom。因此客户机只能通过网络获得操作系统。Linux的远程启动基于标准的BootP/DHCP和TFTP协议，并通过NFS文件系统建立文件系统。

- 客户端开机后，在 Bootrom 获得控制权之前先做自我测试.(bios自检)
- Bootprom 送出 BOOTP/DHCP 要求而取得 IP.
- 如果服务器收到客户端所送出的要求, 就会送回 BOOTP/DHCP 回应,内容包括 IP 地址, 预设网关， 及开机镜像文件.
- Bootprom 由 TFTP 通讯协议从服务器下载开机映像文件。

- 客户端通过这个开机映像文件开机, 这个开机文件可以只是单纯的开机程序，也可以是操作系统.
- 开机映像文件将包含 kernel loader 及压缩过的 kernel，此 kernel 将支持NFS root系统。
- 远程客户端根据下载的文件启动机器.

结合实验

- 阅读《See MIPS Run》，了解MIPS启动过程
- 查阅资料，了解ARM（树莓派）启动过程