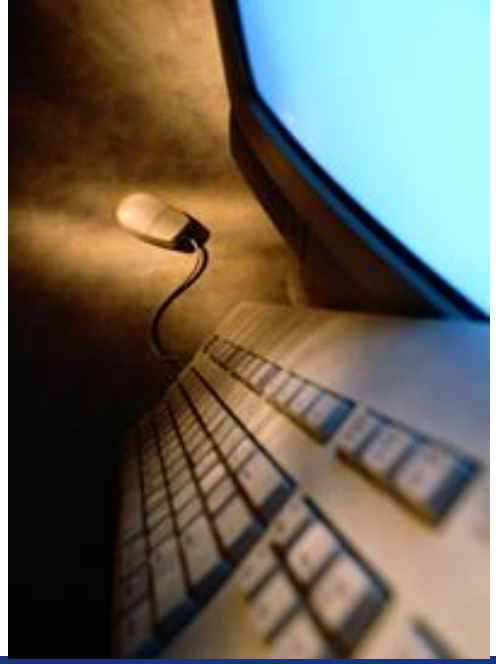
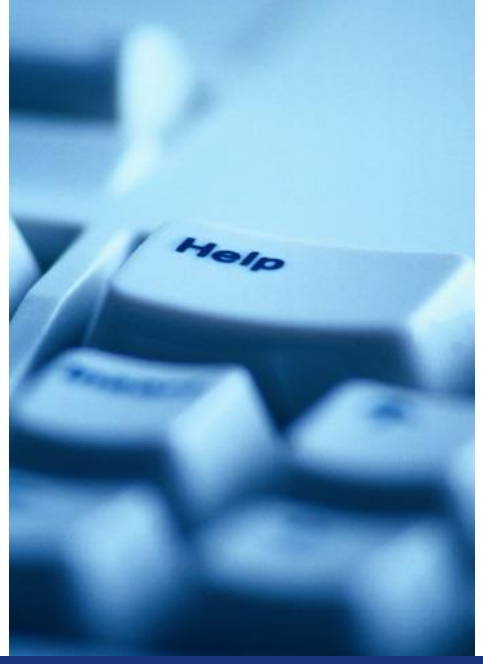



操作系统实验

Labs of Operating Systems

陈鹏飞
计算机学院
2021-03-09

实验安排

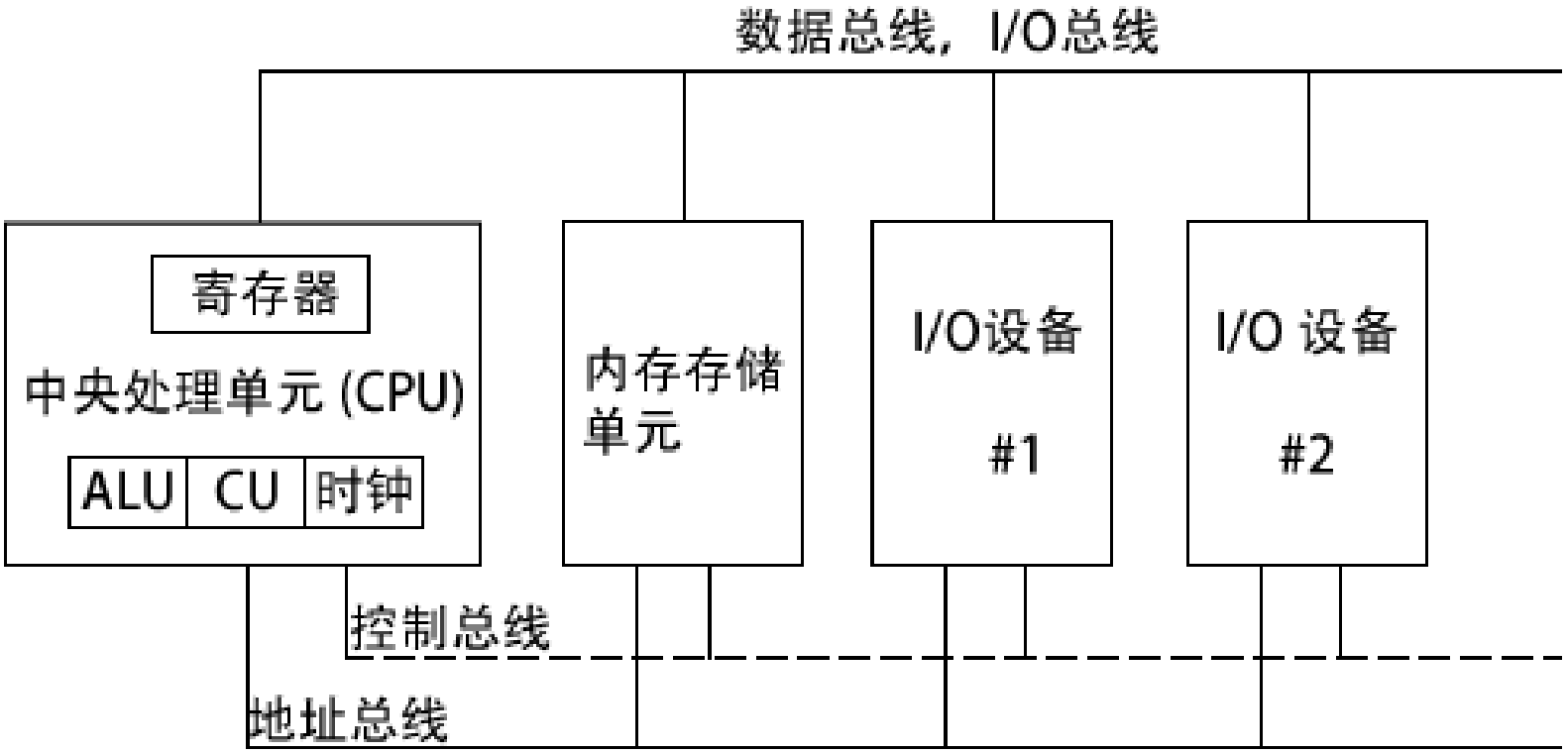
分成两次课程

实验-2： 实模式和保护模式下的OS启动

了解操作系统启动的原理，利用汇编语言实模式即（20位地址空间）的启动和保护模式即（32位地址空间）下的启动方法，并能够在此基础上利用汇编或者C程序实现简单的应用；

1. 回顾、学习32位汇编语言的基本语法；
2. 编写简单的汇编程序，进行中断、输入输出测试；
3. 实现实模式下OS启动；
4. 在实模式下利用汇编/C/Rust等实现简单的应用；
5. 实现保护模式下OS启动；
6. 在保护模式下利用汇编/C/Rust等实现简单的应用；
7. 比较实模式和保护模式的不同；

处理器架构


数据总线，I/O总线
寄存器
中央处理单元 (CPU)
ALU
CU
时钟
内存存储单元
I/O设备 #1
I/O 设备 #2
控制总线
地址总线

加载程序

在程序执行之前，需要用一种工具程序将其加载到内存，这种工具程序称为程序加载器（program loader）。加载后，操作系统必须将 CPU 向程序的入口，即程序开始执行的地址。以下步骤是对这一过程的详细分解。

1）操作系统（OS）在当前磁盘目录下搜索程序的文件名；

2）如果程序文件被找到，OS 就访问磁盘目录中的程序文件基本信息，包括文件大小，及其在磁盘驱动器上的物理位置；

3）OS 确定内存中下一个可使用的位置，将程序文件加载到内存；

4）OS 开始执行程序的第一条机器指令（程序入口）。当程序开始执行后，就成为一个进程（process）；

5）进程自动运行。OS 的工作是追踪进程的执行，并响应系统资源的请求；

6）进程结束后，就会从内存中移除；

IA-32处理器基本架构

x86 处理器有三个主要的操作模式：保护模式、实地址模式和系统管理模式；以及一个子模式：虚拟 8086 (virtual-8086) 模式，这是保护模式的特殊情况。

1) 保护模式 (Protected Mode)

保护模式是处理器的原生状态，在这种模式下，所有的指令和特性都是可用的。分配给程序的独立内存区域被称为段，而处理器会阻止程序使用自身段范围之外的内存。

2) 虚拟 8086 模式 (Virtual-8086 Mode)

保护模式下，处理器可以在一个安全环境中，直接执行实地址模式软件，如 MS-DOS 程序。换句话说，如果一个程序崩溃了或是试图向系统内存区域写数据，都不会影响到同一时间内执行的其他程序。现代操作系统可以同时执行多个独立虚拟 8086 会话。

3) 实地址模式 (Real-Address Mode)

实地址模式实现的是早期 Intel 处理器的编程环境，但是增加了一些其他的特性，如切换到其他模式的功能。当程序需要直接访问系统内存和硬件设备时，这种模式就很有用。

4) 系统管理模式 (System Management Mode)

系统管理模式 (SMM) 向操作系统提供了实现诸如电源管理和系统安全等功能的机制。这些功能通常是由计算机制造商实现的，他们为了一个特定的系统设置而定制处理器。

地址空间

- 在 32 位保护模式下，一个任务或程序最大可以寻址 4GB 的线性地址空间。从 P6 处理器开始，一种被称为扩展物理寻址 (extended physical addressing) 的技术使得可以被寻址的物理内存空间增加到 64GB。

- 在实地址模式下，IA-32处理器使用20位的地址线，可以访问220=1MB的内存，范围时0x0000到0xFFFFF。但是，我们看到寄存器的访问模式只有32位，16位和8位，形如eax，ax，ah，al。那么我们如何才能使用16位的寄存器表示20位的地址空间呢？这在当时也给Intel工程师带来了极大的困扰，但是聪明的工程师想出来一种“段地址+偏移地址”的解决方案。段地址和偏移地址均为16位。此时，一个1MB中的地址，称为物理地址，按如下方式计算出来。

物理地址=(段地址<<4)+偏移地址

寄存器

- 基本寄存器。寄存器是CPU内部的高速存储单元。IA-32处理器主要有8个通用寄存器eax，ebx，ecx，edx，ebp，esp，esi，edi、6个段寄存器cs，ss，ds，es，fs，gs、标志寄存器eflags、指令地址寄存器eip。
- 通用寄存器用于算术运算和数据传输。32位寄存器用于保护模式，为了兼容16位的实模式，每一个32位寄存器又可以拆分成16位寄存器和8位寄存器来访问。例如ax是eax的低16位，ah是ax高8位，al是ax的低8位。ebx，ecx，edx也有相同的访问模式。如下所示。

32 位	16 位	8 位（高）	8 位（低）
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

32 位	16 位	32 位	16 位
ESI	SI	EBP	BP
EDI	DI	ESP	SP

寄存器

•某些通用寄存器有特殊用法:
乘除指令默认使用EAX。它常常被称为扩展累加器（extended accumulator）寄存器。
•CPU 默认使用 ECX 为循环计数器。
•ESP 用于寻址堆栈（一种系统内存结构）数据。它极少用于一般算术运算和数据传输，通常被称为扩展堆栈指针（extended stack pointer）寄存器。
•ESI 和 EDI 用于高速存储器传输指令，有时也被称为扩展源变址（extended source index）寄存器和扩展目的变址（extended destination index）寄存器。
•高级语言通过 EBP 来引用堆栈中的函数参数和局部变量。除了高级编程，它不用于一般算术运算和数据传输。它常常被称为扩展帧指针（extended frame pointer）寄存器。

指令指针

指令指针（EIP）寄存器中包含下一条将要执行指令的地址。某些机器指令能控制EIP，使得程序分支转向到一个新位置。

EFLAGS 寄存器

EFLAGS （或 Flags）寄存器包含了独立的二进制位，用于控制 CPU 的操作，或是反映一些 CPU 操作的结果。有些指令可以测试和控制这些单独的处理器标志位。

汇编语言

- 采用Intel x86汇编语言格式;

```
mov ax, 3
mov bx, 2
add  ax,bx
```

编译器采用nasm;

- 样例程序

```
1  section .text
2  global main
3  main:
4  mov eax,4;
5  mov ebx,1;
6  mov ecx,msg;
7  mov edx,14;
8  int 80h;
9  mov eax,1;
10 int 80h;
11 msg:
12 db "hello world",0ah,0dh
```

1. 编译成可执行二进制过程:
   nasm –f elf32 –o *.o *.s;
   gcc –m32 –g -o *.bin *.o ;
2. 编译成可装载的二进制镜像:
   nasm –f bin –o *.o *.s;
3. 使用ld链接形成可执行二进制;
   ld -m elf_i386 -o *.bin *.o;

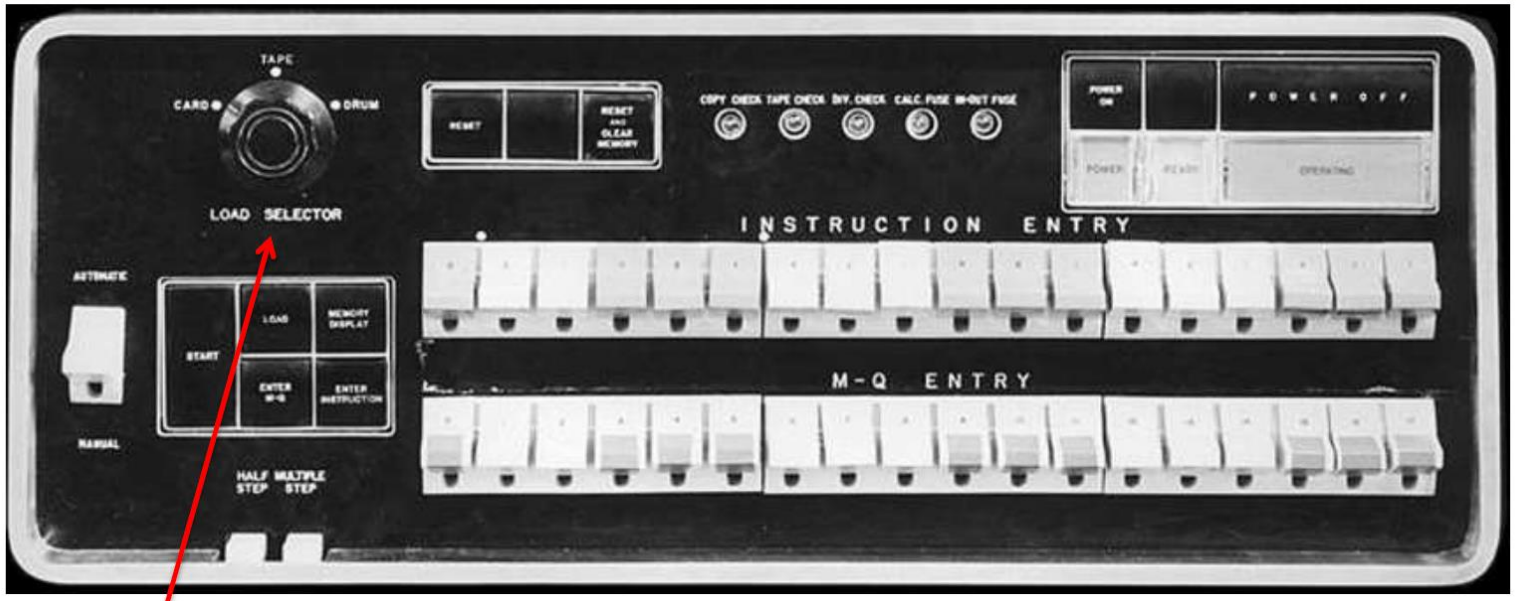
操作系统启动

What runs first?

- Boot loader
  - A program that loads a bigger program (e.g., the OS)

操作系统启动

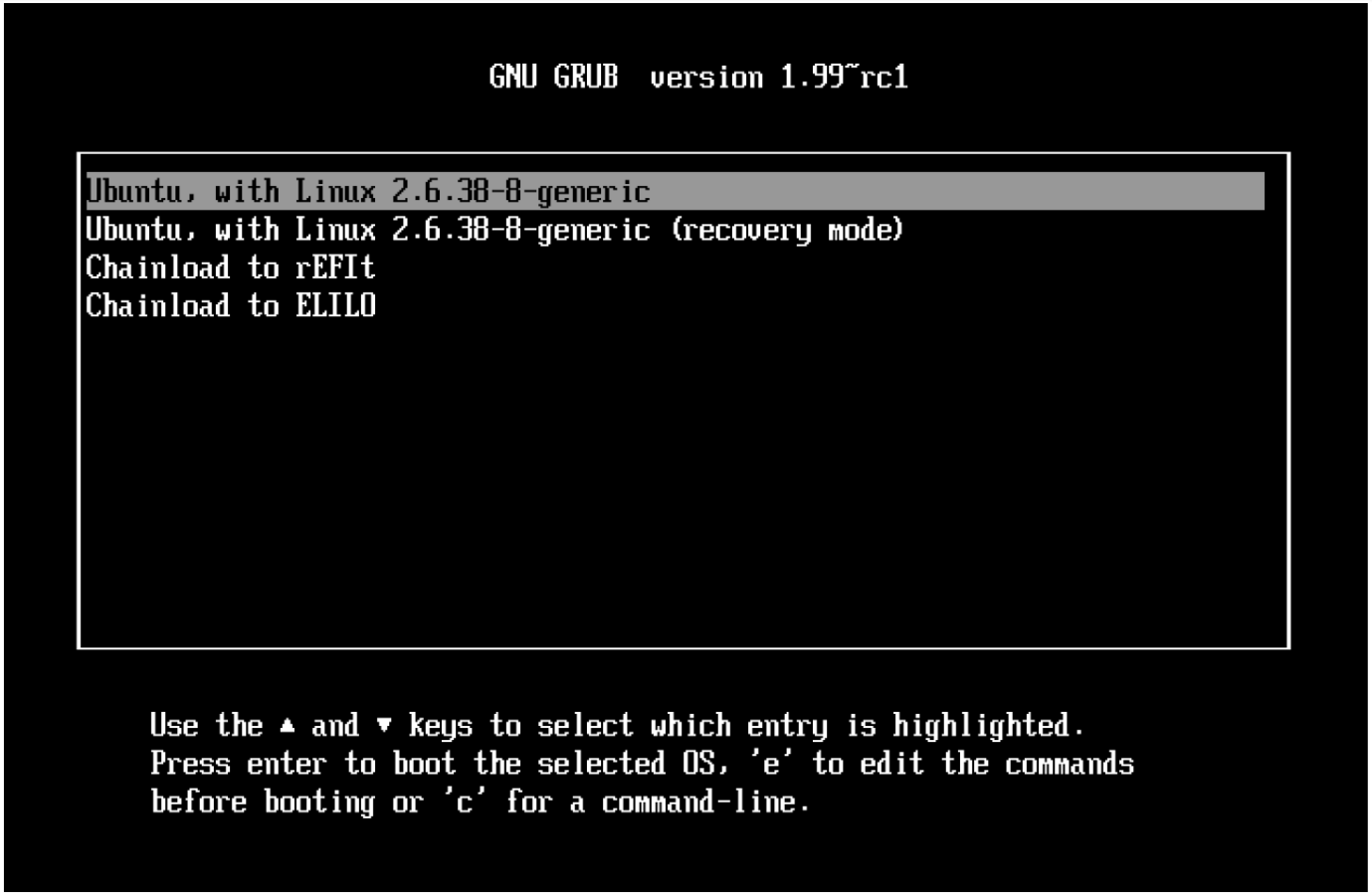
Booting



Load selector: Card, Tape, Drum

操作系统启动

Booting



操作系统启动

Multi-stage boot loader (chain loading)

- First stage boot loader
  - Often primitive enough that an operator could enter the code via front panel switches … or it could sit in the first block of a disk
- Second stage loader
  - More sophisticated and included error checking
  - Second stage loader may give the user a choice:
    - Different operating systems
    - Boot a test program
    - Enable diagnostic modes (e.g., safe boot) in the OS

操作系统启动

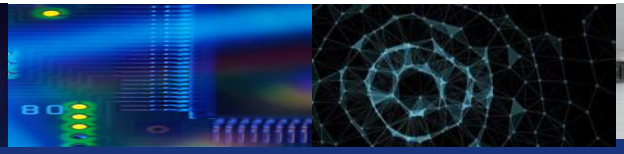
Transfer of control

- When the boot loader finishes loading the OS, it transfers control to it
- The OS will initialize itself and load various modules as needed (for example, device drivers and various file systems)

操作系统启动

Intel/AMD PC Startup

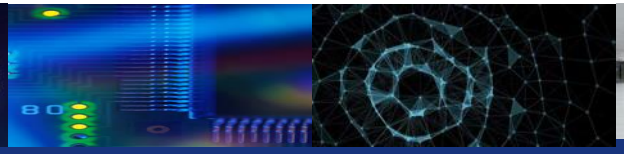
- CPU reset at startup
- Start execution at `0xfffffff0`
  - Jump instruction to BIOS code in non-volatile memory
    - Near the top of 32-bit addressable memory map
    - *Reset vector:* jump to firmware initialization code
  - Processor starts in Real Mode
    - 20-bit address space (top 12 address lines held high)
    - Direct access to I/O, interrupts, and memory



操作系统启动

BIOS

- BIOS = Basic Input/Output System
- Found in Intel-based 16- and 32-bit PCs
- Code resident in ROM or non-volatile flash memory
- Background: CP/M (MS-DOS was almost a clone)
  - Console Command Processor (CCP): *user interface*
  - Basic Disk Operating System (BDOS): *generic code*
  - Basic Input/Output System (BIOS): *all the device interfaces*



操作系统启动

1、显示服务(Video Service——INT 10H)

00H —设置显示器模式0CH —写图形象素
01H —设置光标形状0DH —读图形象素
02H —设置光标位置0EH —在Teletype模式下显示字符
03H —读取光标信息0FH —读取显示器模式
04H —读取光笔位置10H —颜色
05H —设置显示页11H —字体

3、串行口服务(Serial Port Service——INT 14H)

00H —初始化通信口03H —读取通信口状态
01H —向通信口输出字符04H —扩充初始化通信口
02H —从通信口读入字符
(1)、功能00H
功能描述：初始化通信口
入口参数：AH = 00H

4、键盘服务(Keyboard Service——INT 16H)

00H、10H —从键盘读入字符03H —设置重复率
01H、11H —读取键盘状态04H —设置键盘点击
02H, 12H —读取键盘标志05H —字符及其扫描码进栈
(1)、功能00H和10H
功能描述：从键盘读入字符
入口参数：AH = 00H——读键盘

2、直接磁盘服务(Direct Disk Service——INT 13H)

00H —磁盘系统复位0EH —读扇区缓冲区
01H —读取磁盘系统状态0FH —写扇区缓冲区
02H —读扇区10H —读取驱动器状态
03H —写扇区11H —校准驱动器
04H —检验扇区12H —控制器RAM诊断
05H —格式化磁道13H —控制器驱动诊断
06H —格式化坏磁道14H —控制器内部诊断

操作系统启动

PC Startup

- BIOS executes:
  - Power-on self-test (POST)
  - Detect video card's BIOS – execute video initialization
  - Detect other device BIOS – initialize
  - Display start-up screen
  - Brief memory test
  - Set memory, drive parameters
  - Configure Plug & Play devices: PCIe, USB, SATA, SPI
    - Assign resources (DMA channels & IRQs)
  - Identify boot device:
    - Load block 0 (Master Boot Record) to `0x7c00` and jump there

操作系统启动

Booting Windows (NT/Windows 20xx,7,8)

- BIOS-based booting
  - The BIOS does *not* know file systems but can read disk blocks
- MBR = Master Boot Record = Block 0 of disk (512 bytes)
  - Small boot loader (chain loader, ≤ 440 bytes)
  - Disk signature (4 bytes)
  - Disk partition table (16 bytes per partition * 4)
- BIOS firmware loads and executes the contents of the MBR
- MBR code scans through partition table and loads the Volume Boot Record (VBR) for that partition
  - Identifies partition type & size
  - Contains Instruction Program Loader that executes startup code
  - IPL reads additional sectors to load BOOTMGR (Windows 7, 8)
    - The loader is called NTLDR for Windows NT, XP, 2003

操作系统启动

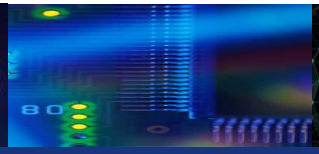
Booting other systems on a PC

- Example: GRUB (Grand Unified Boot Loader)
- MBR contains GRUB Stage 1
  - Or another boot loader that may boot GRUB Stage 1 from the Volume Boot Record
- Stage 1 loads Stage 2
  - Present user with choice of operating systems to boot
  - Optionally specify boot parameters
  - Load selected kernel and run the kernel
  - For Windows (which is not Multiboot compliant),
    - Run MBR code or Windows boot menu
    - Multiboot specification:
      - Free Software Foundation spec on loading multiple kernels using a single boot loader

操作系统启动

Good-bye BIOS: PCs and UEFI

- ~2005: Unified Extensible Firmware Interface (UEFI)
  - Originally called EFI; then changed to UEFI
    You still see both names in use
- Created for 32- and 64-bit architectures
  - Including Macs, which also have BIOS support for Windows
- Goal:
  - Create a successor to the BIOS
    - no restrictions on running in 16-bit 8086 mode with 20-bit addressing



操作系统启动

UEFI Includes

- Preserved from BIOS:
  - Power management (Advanced Configuration & Power Interface, ACPI)
  - System management components from the BIOS
- Support for larger disks
  - BIOS only supported 4 partitions per disk, each up to 2.2 TB per partition
  - EFI supports max partition size of 9.4 ZB (9.4×10^{21} bytes)
- Pre-boot execution environment with direct access to all memory
- Device drivers, including the ability to interpret architecture-independent EFI Byte Code (EBC)
- Boot manager: lets you select and load an OS
  - *No need for a dedicated boot loader (but they may be present anyway)*
  - *Stick your files in the EFI boot partition and EFI can load them*
- Extensible: extensions can be loaded into non-volatile memory

操作系统启动

UEFI Booting

- No need for MBR code *(ignore block 0)*
- Read GUID Partition Table (GPT)
  - Describes layout of the partition table on a disk (blocks 1-33)
- EFI understands Microsoft FAT file systems
  - Apple's EFI knows HFS+ in addition
- Read programs stored as *files* in the EFI System Partition:
  - Windows 7/8, Windows 2008/2012 (64-bit Microsoft systems):
    - Windows Boot Manager (BOOTMGR) is in the EFI partition
  - NT (IA-64): IA64ldr
  - Linux: elilo.efi (ELILO = EFI Linux Boot Loader)
  - OS X: boot.efi

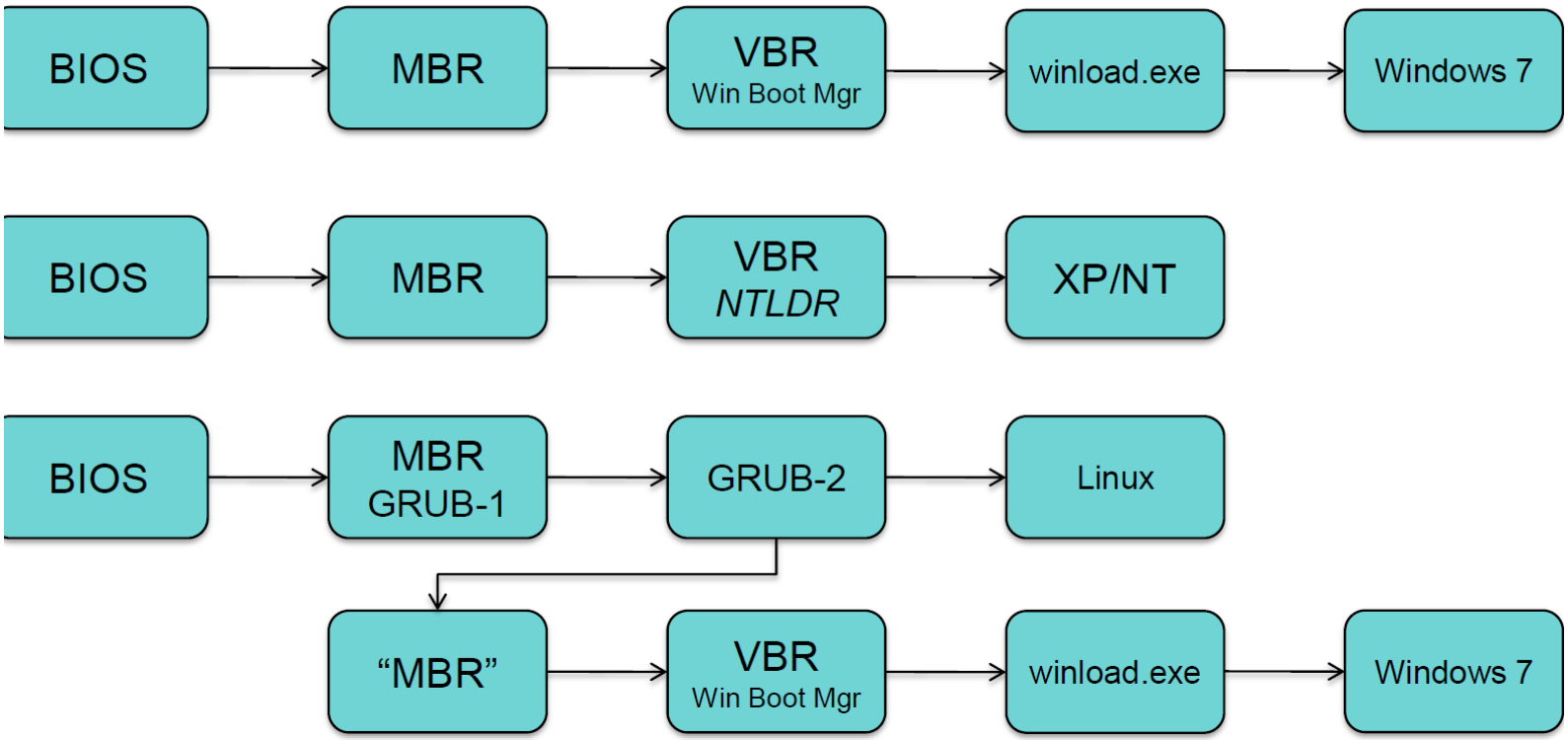
操作系统启动

Non-Intel Systems

- Power on: execute boot ROM code (typically NOR Flash)
  - Often embedded in the CPU ASIC
- Boot ROM code detects boot media
  - Loads first stage boot loader (sometimes to internal RAM)
  - Initialize RAM
  - Execute boot loader
- Second stage boot loader loads kernel into RAM
  - For Linux, typically GRUB for larger systems
  - uBoot for embedded systems
  - Set up network support, memory protection, security options

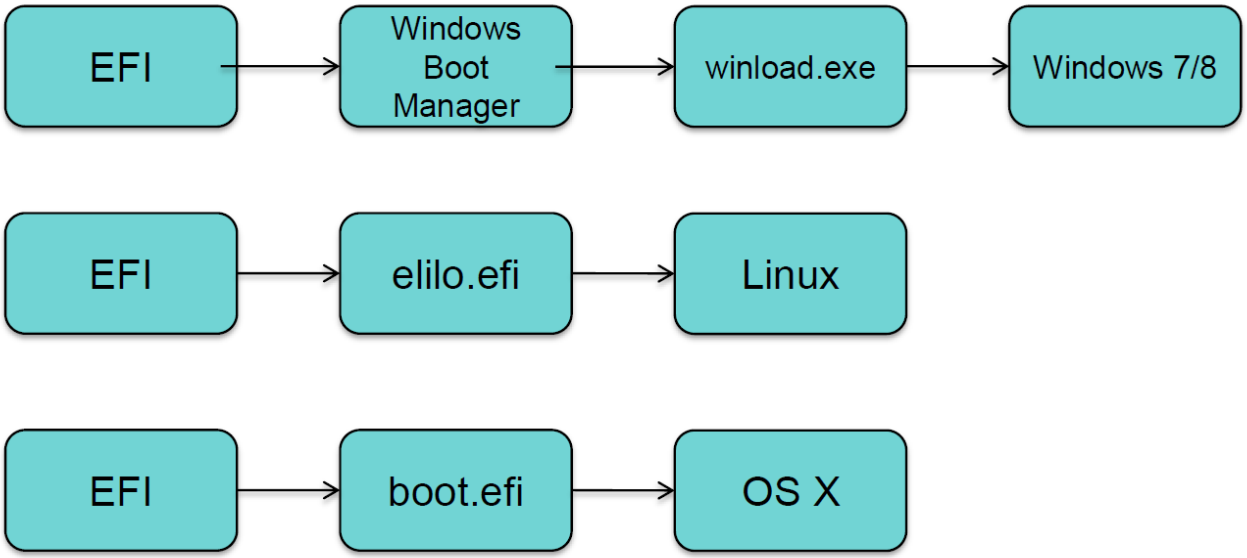
操作系统启动

Summary



操作系统启动

Summary

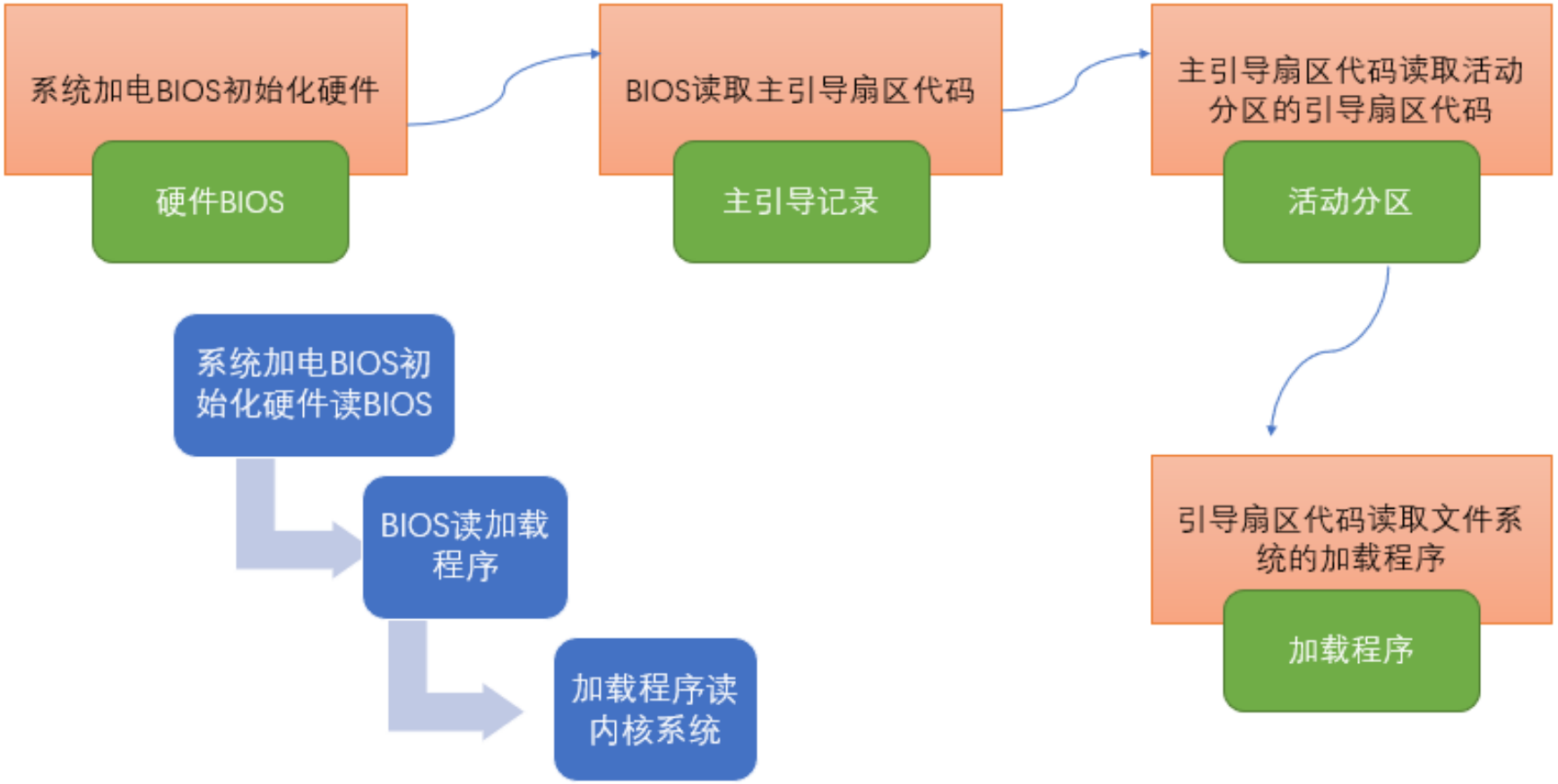


实模式启动代码

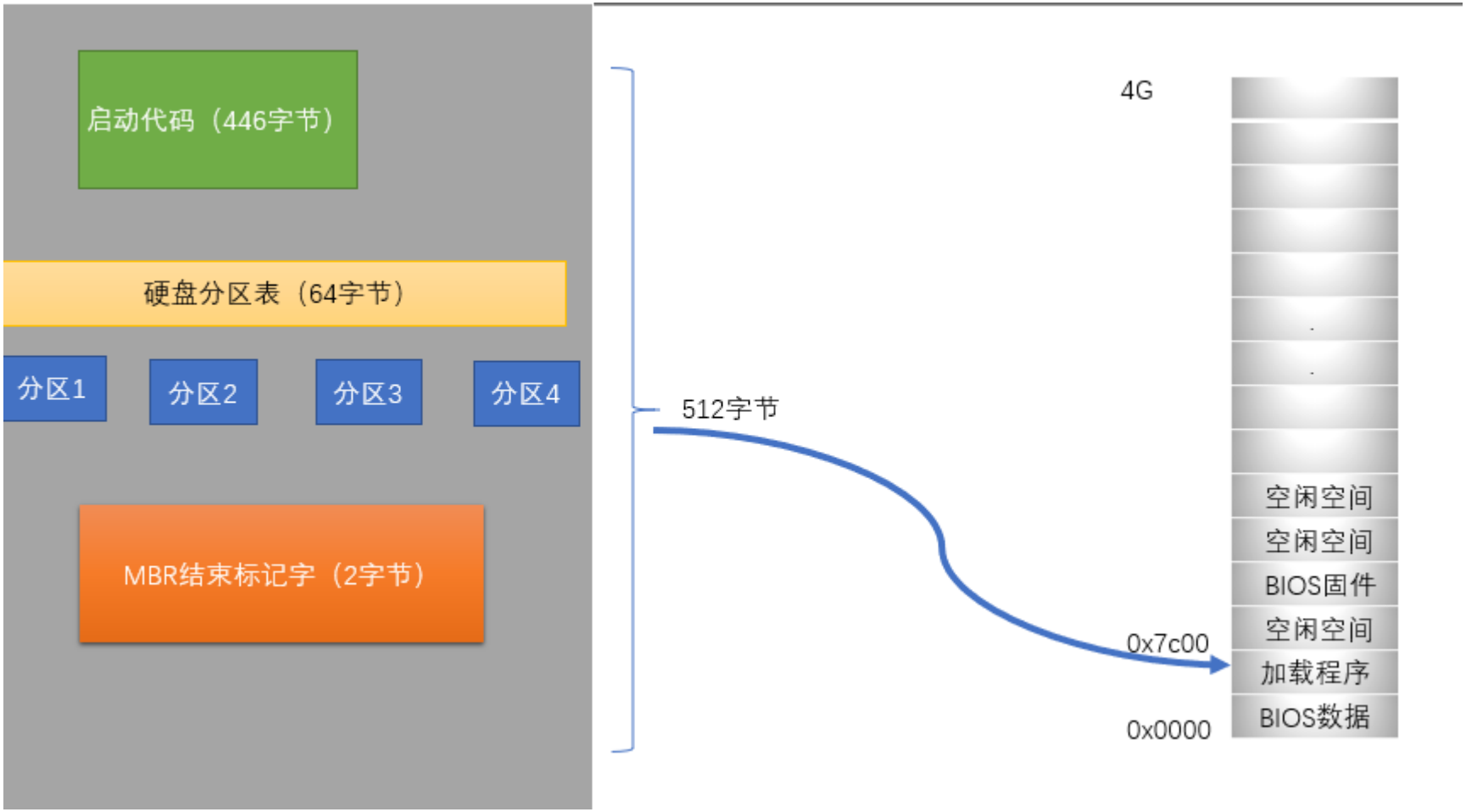
```
;
;bits 16

[bits 16]
mov ah, 0x0e            ;设置tele-type mode
mov al, 'H'             ;设置待显示字符
int 0x10                ;screen相关的中断
mov al, 'e'
int 0x10
mov al, 'l'
int 0x10
mov al, 'l'
int 0x10
mov al, 'o'
int 0x10

jmp $                   ;跳到当前地址，无限循环
times 510-($-$$) db 0   ;填充程序到512个字节
dw 0xaa55               ;让bios识别此扇区为可启动扇区的魔数
```

1. nasm -f bin *.asm -o *.bin;
2. qemu-img create hd.img 10m;
3. dd if=*.bin of=hd.img bs=512 count=1 seek=0 conv=notrunc;
4. qemu-system-i386 -hda hd.img -serial null -parallel stdio ;

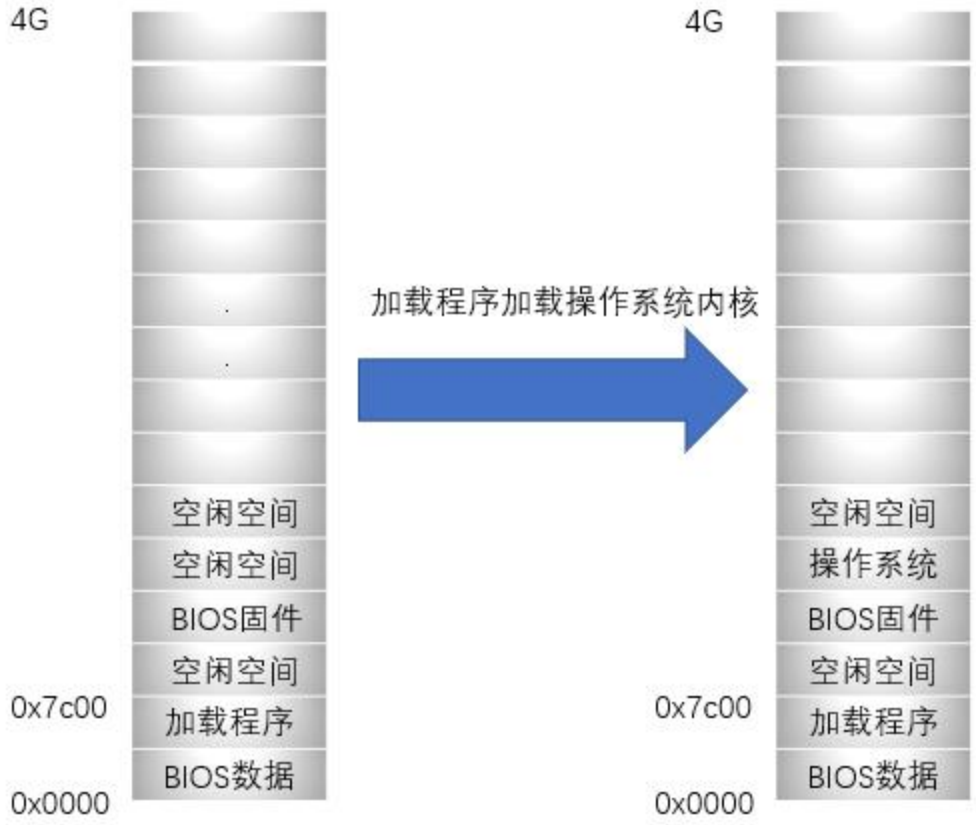
回顾：操作系统启动过程


系统加电BIOS初始化硬件
硬件BIOS
BIOS读取主引导扇区代码
主引导记录
主引导扇区代码读取活动分区的引导扇区代码
活动分区
引导扇区代码读取文件系统的加载程序
加载程序
系统加电BIOS初始化硬件读BIOS
BIOS读加载程序
加载程序读内核系统

引导扇区（512字节）的内容及在内存中分配的地址



操作系统加载过程


4G
空闲空间
空闲空间
BIOS固件
空闲空间
0x7c00
加载程序
BIOS数据
0x0000
加载程序加载操作系统内核
4G
空闲空间
操作系统
BIOS固件
空闲空间
0x7c00
加载程序
BIOS数据
0x0000

突破512字节的限制

在运行操作系统前，我们需要将操作系统内核程序从外存加载到内存中。显然，MBR无法胜任这项工作

1、通过BIOS中断（int 13h）实现磁盘内容读取；

```
19
20 disk_load:
21     push dx
22     mov ah, 0x02     ;BIOS读扇区功能
23     mov al, dh       ;读dh个扇区
24     mov ch, 0x00     ;选择柱面0
25     mov dh, 0x00     ;选择磁头0
26     mov cl, 0x02     ;读扇区2（从boot_sector之后的第一个扇区）
27
28     int 0x13         ;BIOS interrupt
29     jc disk_error_flag
30     pop dx
31     cmp dh, al       ;al代表已读扇区数，dh是预期读扇区数
32     jne disk_error_count
33     ret
34 disk_error_flag:
35     mov bx, DISK_ERROR_FLAG_MSG
36     call print_string
37     jmp $
38 disk_error_count:
39     mov bx, DISK_ERROR_COUNT_MSG
40     call print_string
41     jmp $
```

展示

突破512字节的限制

2、通过LBA（(Logical Block Addressing）方式读写磁盘；

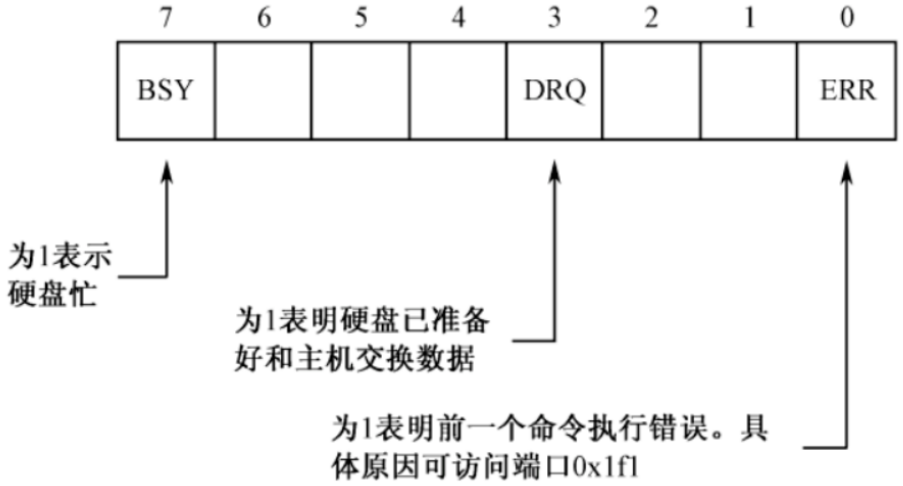
- ✓ 硬盘是外围设备的一种，处理器和外围设备的交换是通过I/O端口进行的；
- ✓ 每一个端口在I/O电路中都会被统一编址。例如，主硬盘分配的端口地址是0x1f0~0x1f7；
- ✓ 因为端口是独立编址的，因此我们无法使用mov指令来对端口赋值，我们使用的是in,out指令；

```
in al, 0x21 ; 表示从0x21端口读取一字节数据到al
in ax, 0x21 ; 表示从端口地址0x21读取1字节数据到al，从端口地址0x22读取1字节到ah

mov dx,0x379
in al, dx  ; 从端口0x379读取1字节到al

; out指令
out 0x21, al ; 将al的值写入0x21端口
out 0x21, ax ; 将ax的值写入端口地址0x21开始的连续两个字节
mov dx, 0x378
out dx, ax  ; 将ah和al分别写入端口0x379和0x378
```

突破512字节的限制

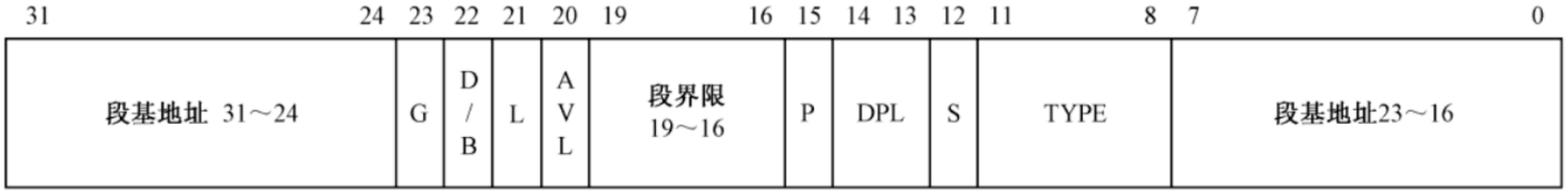
2、通过LBA（(Logical Block Addressing）方式读写磁盘；

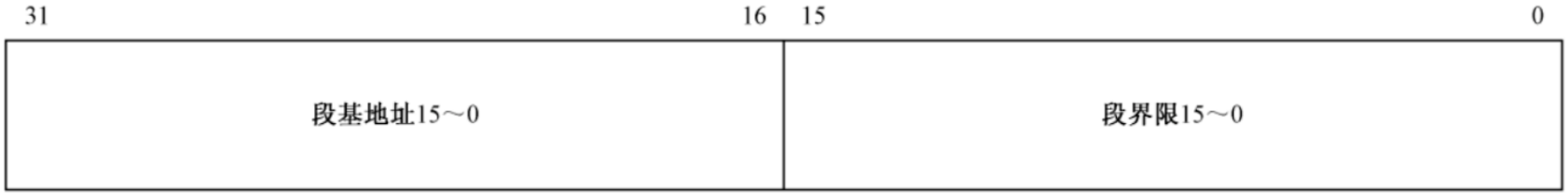
- **设置起始的逻辑扇区号 （LBA28）；**
- **将要读取的扇区数量写入0x1F2端口；**
- **向0x1F7端口写入0x20，请求硬盘读；**
- **等待其他读写操作完成；**
- **若在第4步中检测到其他操作已经完成，那么我们就可以正式从硬盘中读取数据**



从实模式到保护模式

- 在保护模式下，所有的程序都会运行在自己的段中
- 段地址空间信息是通过段描述符(segment descriptor)来给出的；

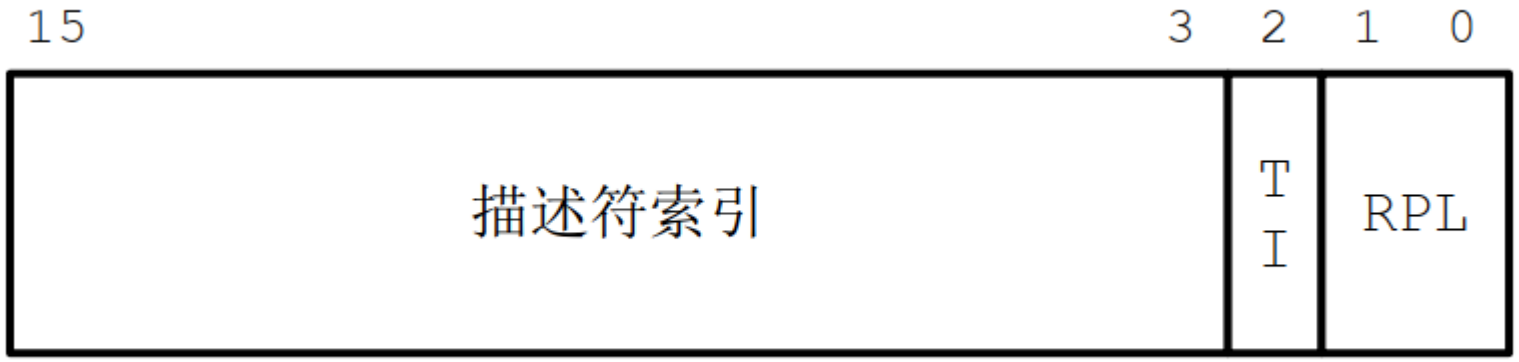




- 保护模式下的段寄存器依然是 16 位，但其中保存的不再是段地址，而是段选择子；

从实模式到保护模式

所有的段都会被保存在全局描述符表(GDT)中，实际上段选择子是全局描述符表的索引，类似数组访问array[i]中的i，但段选择子中还会包含其他信息，如下所示。



保护模式启动代码

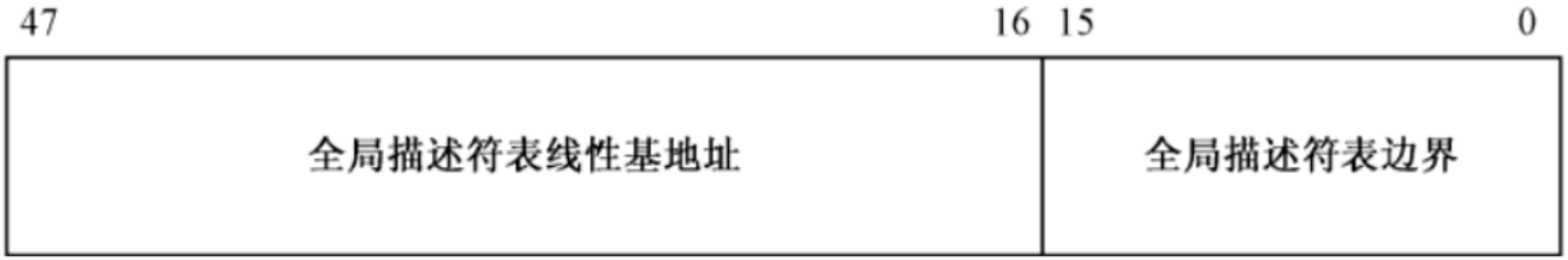
Intel x86系列CPU有实模式和保护模式，实模式从8086开始就有，保护模式从80386开始引入。为了兼容，Intel x86系列CPU都支持实模式。现代操作系统都是运行在保护模式下（Intel x86系列CPU）。计算机启动时，默认的工作模式是实模式，为了让内核能运行在保护模式下，Bootloader需要从实模式切换到保护模式，切换步骤如下：

1. 准备好GDT(Global Descriptor Table)
2. 关中断
3. 加载GDT到GDTR寄存器
4. 开启A20，让CPU寻址大于1M
5. 开启CPU的保护模式，即把cr0寄存器第一个bit置1
6. 跳转到保护模式代码

GDT是Intel CPU保护模式运行的核心数据结构，所有保护模式操作的数据都从GDT表开始查找， 这里有GDT的详细介绍。

GDT

GDT实际上是一个段描述符数组，保存在内存中。GDT的起始位置和大小由我们来确定，保存在寄存器GDTR中，GDTR的内容如下所示。



第21根地址线

在实模式下，第21根地址线的值恒为0,想进入保护模式时，首先需要打开第 21 根地址线;

```
in al, 0x92 ; 南桥芯片内的端口
or al, 0000_0010B
out 0x92, al ; 打开 A20
```

保护模式开关——CR0

CR0 是 32 位的寄存器，包含了一系列用于控制处理器操作模式和运行状态的标志位，其第0位是保护模式的开关位，称为PE（protect mode enable）位。

```
cli             ; 保护模式下中断机制尚未建立，应禁止中断
mov eax, cr0
or eax, 1
mov cr0, eax    ; 设置 PE 位
```

参考资料

- x86汇编(Intel汇编)入门
- 《Intel汇编语言程序设计》第1-8章
- 《从实模式到保护模式》第1-8章
- http://c.biancheng.net/makefile/
- How to write a simple operating system
- The little book about OS development

谢谢