

Linux 操作系统设计与实现

All problems in computer science can be solved by another level of
indirection.

作 者: 罗裕辉
时 间: 2024 年上

目录

一、绪论	4
1.1 操作系统架构介绍	4
1.2 一些不那么显然的问题及回答	4
1.2.1 程序编译的过程	4
1.2.2 解释型语言解释的过程	4
1.2.3 半编译半解释型语言的过程	4
1.2.4 为什么说 Linux 一切皆文件	5
1.2.5 为什么 Linux 的应用程序不能在 windows 上运行	5
1.2.6 什么是系统调用	5
1.2.7 为什么说陷入内核	5
1.2.8 先有 C 编译器还是先有 C 语言	5
1.2.9 什么是交叉编译	5
1.2.10 体系结构、指令集、微架构	5
1.2.11 几种存储结构的比较	5
二、工作环境	5
2.1 编译器 GCC	5
2.2 汇编器 NASM	6
2.3 虚拟机	6
2.4 bochs 的调试方法	7
2.4.1 “Debugger control” 类	7
2.4.2 “Execution control” 类	7
2.4.3 “Breakpoint management” 类	8
2.4.4 “CPU and memory contents” 类	8
2.5 开始部署	10
三、计算机的启动	10
3.1 汇编语言的一些知识	10
3.1.1 地址	10
3.1.2 section	10
3.1.3 \$、\$\$	10
3.1.4 vstart	10
3.2 实模式	11

3.2.1 CPU 的原理	11
3.2.2 寄存器	12
3.2.3 实模式下的分段	14
3.2.4 实模式下 CPU 寻址方式	14
3.2.5 调用函数的堆栈框架	14
3.3 CPU 访问外设	16
3.3.1 IO 接口	16
3.3.2 显卡的访问	17
3.4 硬盘	21
3.4.1 发展简史	21
3.4.2 硬盘的原理	21
3.4.3 硬盘的操作	24
3.5 BIOS	24
3.6 让 MBR 读取硬盘	26
四、保护模式	32
4.1 实模式的缺陷	32
4.2 扩展	32
4.2.1 寄存器扩展	32
4.2.2 寻址扩展	33
4.2.3 运行模式	34
4.2.4 指令扩展	34
4.3 全局描述符表 GDT	34
4.3.1 段描述符	34
4.3.2 GDT	36
4.3.3 segment selector	36
4.3.4 LDT	37
4.3.5 进入保护模式	37
4.4 处理器微架构	48
4.4.1 流水线	48
4.4.2 乱序执行	49
4.4.3 分支预测	49
4.4.4 缓存	50
4.5 段的保护	50

4.5.1 内存段的保护	50
五、内核的完善	51
5.1 获取物理内存容量	51
5.2 内存分页机制-由段式管理到页式管理	58
5.2.1 页目录表及页表	58
5.2.2 用户进程共享操作系统	59
5.2.3 启动分页机制	60
5.3 加载内核	62
5.3.1 ELF 文件结构	62
5.3.2 载入内存	64
参考文献	64

一、绪论

1.1 操作系统架构介绍

1.2 一些不那么显然的问题及回答

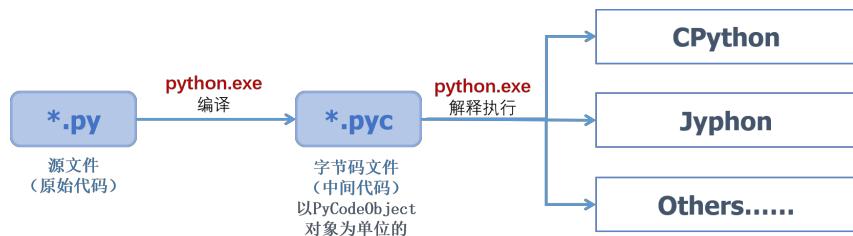
1.2.1 程序编译的过程

1.2.2 解释型语言解释的过程

除了编译类型的语言，其他类型的语言一般都有一个中间文件，这个生成的中间代码类似编译过程中的前端文件。如 python 语言。

.py -> 编译成 .pyc 字节码 -> python 解释器逐行解释。

需要注意的是，尽管 python 有中间的字节码文件，但这个中间表示相对简单，因为 python 是动态语言，动态确定数据类型，通常这个过程会和 python 解释器集合在一起，是透明的，主要是为了提高执行效率而非跨平台。



*每一个PyCodeObject对象包括了静态变量的信息和字节码，与环境PyFrameObject所维护的动态内存空间相对应，完成一次特定环境下的动态执行。

图 1

1.2.3 半编译半解释型语言的过程

如 Java 语言。

.java -> javac 编译成 .class 字节码 -> JVM 解释或即时编译 (热点代码)。

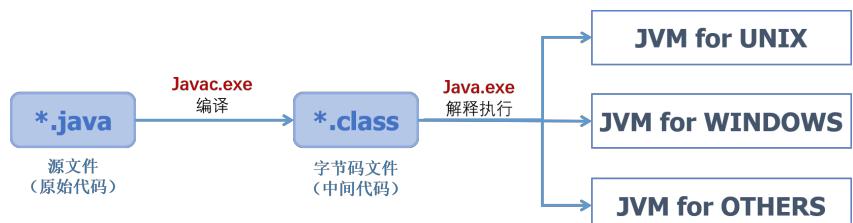


图 2

总的来说：

- 解释执行就是说逐行读取字节码并执行，在这种情况下，字节码被解释为执行指令，但不会生成持久的机器码。而即时编译，即时编译器会在程序运行时将字节码动态编译为机器码，然后直接执行这些机器码。
- 虚拟机实际上可以包含解释器，用于解释执行字节码。但并非所有虚拟机都使用解释执行；有些使用即时编译技术将字节码转换为机器码。
- 通常，这些非编译型语言本身不会成为一个进程在内存上执行，而是被其他进程解释执行 (JIT 除外)。

1.2.4 为什么说 Linux 一切皆文件

1.2.5 为什么 Linux 的应用程序不能在 windows 上运行

1.2.6 什么是系统调用

1.2.7 为什么说陷入内核

1.2.8 先有 C 编译器还是先有 C 语言

1.2.9 什么是交叉编译

1.2.10 体系结构、指令集、微架构

1.2.11 几种存储结构的比较

二、工作环境

2.1 编译器 GCC

GNU 编译器套装 (GNU Compiler Collection, GCC)，是一套由 GNU 开发的编程语言编译器。它是一套以 GPL 及 LGPL 许可证所发行的自由软体，也是 GNU 计划的关键部分，亦是自由的类 Unix 及苹果电脑 Mac OS X 操作系统的标准编译器。GCC（特别是其中的 C 语言编译器）也常被认为是跨平台编译器的标准。

GCC 经过扩展可编译的语言有很多，包括 C 语言 (GCC)、C++ (G++)、GO、Java (编译器:GCJ; 解释器:GIJ) 等。

GCC 作者是理查德·马修·斯托曼 (Richard Matthew Stallman, 自由软件运动和 GNU 的领袖，还是著名黑客)，大神级别人物。

这里介绍一下 GCC 常见的几种用法。

- 预处理：

```
gcc -E hello.c -o hello.i
```

- 预处理 + 编译:

```
1 gcc -S hello.c -o hello.s
```

- 预处理 + 编译 + 汇编:

```
1 gcc -c hello.c -o hello.o
```

- 预处理 + 编译 + 汇编 + 链接:

```
1 gcc hello.c -o hello.out
```

- 指定编译优化级别 (通常 O2, 它包括包括函数内联、循环优化、指令调度和启用更多特定 CPU 的优化。):

```
1 gcc -O2 hello.c -o hello.out
```

2.2 汇编器 NASM

NASM 是一个为可移植性与模块化而设计的一个 80x86 的汇编器。它支持相当多的目标文件格式，包括 Linux 和'NetBSD/FreeBSD'，'a.out'，'ELF'，'COFF'，微软 16 位的'OBJ' 和'Win32'。它还可以输出纯二进制文件。它的语法设计得相当的简洁易懂，和 Intel 语法相似但更简单。它支持'Pentium'，'P6'，'MMX'，'3DNow!'，'SSE' 和'SSE2' 指令集，

注意: 在看汇编代码时，通常有两种语法，intel 和 AT&T

- intel: 源操作数在左侧，目标操作数在右侧，同时直接使用寄存器名称。

```
1 MOV AX, 5 ; 将立即数 5 移动到寄存器 AX
```

- AT&T: 源操作数在右侧，目标操作数在左侧，同时用% 代表取值。

```
1 mov $5, %ax ; 将立即数 5 移动到寄存器 AX
```

2.3 虚拟机

虚拟机就是用软件来模拟硬件。虚拟机只是一个普通的进程，该进程模拟了硬件资源，在虚拟机中运行的程序其所做出的任何行为都先被虚拟机检查，由虚拟机分析后，代为向操作系统申请。

使用虚拟机方便、功能强大并且安全。我们选用 vmware+ubuntu。此外 PC 模拟器选择 bochs。

Bochs (发音: box) 是一个以 LGPL 许可证发放的开放源代码的 x86、x86-64IBM PC 兼容机模拟器和调试工具。它支持处理器（包括保护模式）、内存、硬盘、显示器、以太网、BIOS、IBM PC 兼容机的常见硬件外设的仿真。(当然著名的还有 qemu)。

2.4 bochs 的调试方法

bochs 是一个开源 x86 虚拟机软件，调试风格是参照 gdb 来设计。

大体上 bochs 的调试命令分为“Debugger control”类、“Execution control”类、“Breakpoint management”类、“CPU and memory contents”类

2.4.1 “Debugger control” 类

- q|quit|exit: 退出虚拟机
- set 是指令族，通常用 set 设置寄存器的值。
 - set reg = val。可以设置的寄存器包括通用寄存器和段寄存器。
 - 2) 也可以设置每次停止执行时，是否反汇编指令：set u on|off。
- show 是指令族，有很多子功能
 - show mode: 每次 CPU 变换模式时就提示，模式是指保护模式、实模式，比如从实模式进入到保护模式时会有提示。
 - show int: 每次有中断时就提示，同时显示三种中断类型，这三种中断类型包括“softint”、“extint” 和 “iret”。可以单独显示某类中断，如执行 show softint 只显示软件主动触发的中断，show extint 则只显示来自外部设备的中断，show iret 只显示 iretd 指令有关的信息。
 - show call: 每次有函数调用发生时就会提示。
- trace on|off 如果此项设为 on，每次执行一条指令，bochs 都会将反汇编的代码打印到控制台，这样在单步调试时免得看源码了。
- u|disasm: u|disasm [/num] [start] [end]
将物理地址 start 到 end 之间的代码反汇编，如果不指定地址，则反汇编 EIP 指向的内存。num 指定反汇编的指令数。
- ctrl+c 中断执行，回到 bochs 控制台。

2.4.2 “Execution control” 类

- c|cont|continue，左边列出的三个命令都意为向下持续执行，若没断点则一直运行下去。最常用的是 c。

- s|step [count] 执行 count 条指令, count 是指定单步执行的指令数, 若不指定, count 默认为 1。此指令若遇到函数调用, 则会进入函数中去执行。最常用的是 s。
- p|n|next 执行 1 条指令, 若待执行的指令是函数调用, 不管函数内有多少指令, 把整个函数当作一个整体来执行。最常用的是 n。

2.4.3 “Breakpoint management” 类

- 通过地址打断点
 - vb|vbreak [seg: off] 以虚拟地址添加断点, 程序执行到此虚拟地址时停下来, 注意虚拟地址是“段: 段内偏移”的形式。最常用的是 vb。
 - lb|lbreak [addr] 以线性地址添加断点, 程序执行到此线性地址时停下来。最常用的是 lb。
 - pb|pbreak|b|break [addr] 以物理地址添加断点。程序执行到此物理地址时停下来。
b 比较常用。
- 以指令数打断点
 - sb [delta] delta 表示增量, 意味再执行 delta 条指令程序就中断。
 - sba [time] CPU 从运行开始, 执行第 time 条指令时中断, 从 0 开始的指令数。
- 以读写 IO 打断点
 - watch 显示所有读写断点。
 - watch r|read [phy_addr] 设置读断点, 如果物理地址 phy_addr 有读操作则停止运行。
 - watch w|write [phy_addr] 设置写断点, 如果物理地址 phy_addr 有写操作则停止运行。此命令非常有用, 如果某块内存不知何时被改写了, 可以设置此中断。
 - unwatch 清除所有断点。
 - unwatch [phy_addr] 清除在此地址上的读写断点。
 - blist 显示所有断点信息, 功能等同于 info b。
 - bpd|bpe [n] 禁用断点 (break point disable) / 启用断点 (break point enable), n 是断点号, 可以用 blist 命令先检查出来。
 - d|del|delete [n] 删除某断点, n 是断点号, 可以用 blist 命令先检查出来。D 最常用。

2.4.4 “CPU and memory contents” 类

- x /nuf [line_addr] 显示线性地址的内容。n、u、f 是三个参数, 都是可选的, 如果没有指定, 则 n 为 1, u 是 4 字节, f 是十六进制
 - n 显示的单元数

- u 每个显示单元的大小, u 可以是下列之一: (1) b 1 字节; (2) h 2 字节; (3) w 4 字节; (4) g 8 字节。
 - f 显示格式, f 可以是下列之一: (1) x 按照十六进制显示; (2) d 十进制显示; (3) u 按照无符号十进制显示; (4) o 按照八进制显示; (5) t 按照二进制显示; (6) c 按照字符显示; (7) s 按照 ASCIIz 显示; (8) i 按照 instr 显示。
- `xp /nuf [phy_addr]` 显示物理地址 phy_addr 处的内容, 注意和 x 的区别, x 是线性地址。
- `setpmem [phy_addr] [size] [val]` 设置以物理内存 phy_addr 为起始, 连续 size 个字节的内容为 val。此命令非常有用, 在某些情况下不易调试时, 可以在程序中通过某个地址的值来判断分支, 需要用 setpmem 来配合。
size 最多只能设置 4 个字节宽度的数据, 如果 size 大于 4 便会报错: Error: setpmem: bad length value = 8。size 小于等于 4 是正确的, `setpmem 0x7c00 4 0x9`。
- `r|reg|regs|registers` 任意四个命令之一便可以显示 8 个通用寄存器的值 +eflags 寄存器 +eip 寄存器。r 是常用的查看寄存器的命令。
- `ptime` 显示 Bochs 自启动之后, 总执行指令数。其实这个命令不常用, 感兴趣的同学可以用 ptime 和“sb 指令数”来验证结果是否正确。
- `print-stack [num]` 显示堆栈, num 默认为 16, 表示打印的栈条目数。输出的栈内容是栈顶在上, 低地址在上, 高地址在下。这和栈的实际扩展方向相反, 这一点请注意。
- `?|calc` 内置的计算器。
- `info` 是个指令族, 执行 `help info` 时可查看其所有支持的子命令
 - `info pb|pbreak|b|break` 查看断点信息, 等同于 `blist`。
 - `info CPU` 显示 CPU 所有寄存器的值, 包括不可见寄存器。
 - `info fpu` 显示 FPU 状态。
 - `info idt` 显示中断向量表 IDT。
 - `info gdt [num]` 显示全局描述符表 GDT, 如果加了 num, 只显示 gdt 中第 num 项描述符。
 - `info ldt` 显示局部描述符表 LDT。
 - `info tss` 显示任务状态段 TSS。
 - `info ivt [num]` 显示中断向量表 IVT。和 gdt 一样, 如果指定了 num, 则只会显示第 num 项的中断向量。
 - `info flags|eflags` 显示状态寄存器, 其实在用 r 命令显示寄存器值时也会输出 eflags 的状态, 还会输出通用寄存器的值, 通常会用 r 来看。
- `sreg` 显示所有段寄存器的值。
- `dreg` 显示所有调试寄存器的值。
- `creg` 显示所有控制寄存器的值。

- info tab 显示页表中线性地址到物理地址的映射。
- page line_addr 显示线性地址到物理地址间的映射。

2.5 开始部署

参考博客: [部署工作环境](#).

三、计算机的启动

要完成一个操作系统，首先我们需要知道计算机启动后发生了什么，操作系统是怎么样被加载的。

3.1 汇编语言的一些知识

3.1.1 地址

汇编语言中编译器给程序中各符号（变量名或函数名等）分配的地址，就是各符号相对于文件开头的偏移量。

3.1.2 section

section 是一个伪指令，它声明一个节 (尽管有翻译是段，但为了避免和后面的段冲突还是叫节)，它是逻辑上划分将代码进行划分，便于程序员管理。

在默认情况下，有没有 section 都一个样，section 中数据的地址依然是相对于整个文件的顺延，仅仅是在逻辑上让开发人员梳理程序之用。

3.1.3 \$、\$\$

- \$ 表示当前行的地址
- \$\$ 表示本节的地址

3.1.4 vstart

vstart 预先声明程序将来被加载到某地址处 (注意它不会影响加载器的加载)，只是程序员预先知道，进而引用。

section 可以用 vstart=xxxx 修饰，此时 \$\$ 的值则是此 section 的虚拟起始地址 xxxx。\$ 的值是以 xxxx 为起始地址的顺延。

如果用了 vstart 关键字，想获得本 section 在文件中的真实偏移量，可以使用 section. 节名.start。

比如, BIOS 进入 mbr 是通过 jmp 0: 7c00 来实现的, 故此时 cs 已经变成 0, 相当于“平坦模型”了, 我们可以设置 vstart=0x7c00。(当然, 也可以通过其他指令设置 CS)

3.2 实模式

实模式是指早期的 8086 CPU 的寻址方式、寄存器大小、指令用法等。

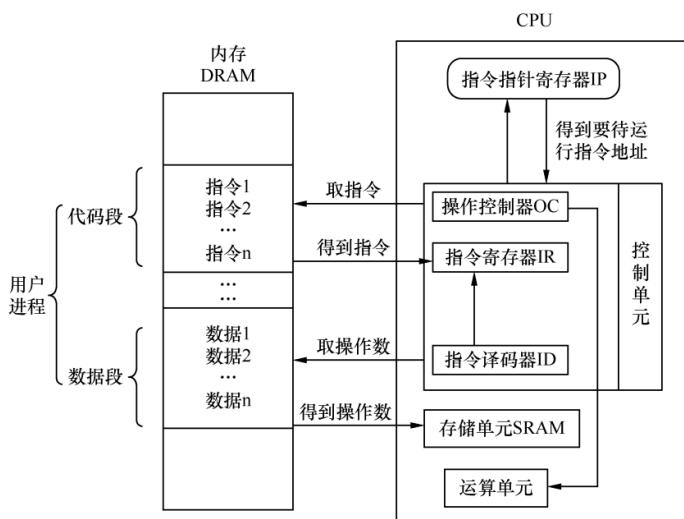
3.2.1 CPU 的原理

CPU(Central Processing Unit) 主要由三部分组成: 控制单元、运算单元、存储单元。

- **控制单元:** 它负责从内存中读取计算机指令, 解码这些指令, 然后控制其他部分(如运算单元和存储单元)以执行这些指令。包括指令指针寄存器 IP、指令寄存器 IR、指令译码器 ID。
- **运算单元:** 负责执行所有的算术运算(如加、减、乘、除)和逻辑运算(如 AND、OR、NOT)。包括 ALU(算术逻辑单元) 和 FPU(浮点单元)。
- **存储单元:** CPU 内的临时存储设备, 包括寄存器和 L1、L2 高速缓存 (Cache)

CPU 本质上进行的是取指令-指令译码-执行指令, 具体执行过程如下:

1. 根据指令指针寄存器 IP 确定下一条要执行的指令的地址
2. 取出该指令放到指令寄存器 IR
3. 指令译码器 ID 译码, 确定操作码、操作数等
4. 根据指令的类型, 控制单元会向运算单元 (ALU) 或其他部件发送控制信号, 进行相应的操作, 同时 IP 指向下一条要执行的指令。



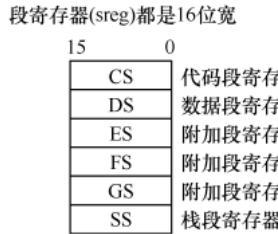
▲图 3-2 CPU 工作原理示意图

图 3

3.2.2 寄存器

段寄存器

段寄存器的作用就是指定一片内存的起始地址，故也称为段基址寄存器。无论是在实模式下，还是保护模式下，它们都是 16 位宽，包括代码段寄存器、数据段寄存器、栈段寄存器，还有三个附加段寄存器。



▲图 3-3 实模式下的段寄存器

图 4

指令指针寄存器

IP 寄存器存储下一条要执行的指令的地址，本质上是代码段段内偏移地址。IP 寄存器是不可见寄存器，CS 寄存器是可见寄存器。CS:IP 最终才是当前 CPU 要执行的指令的地址。

flags 寄存器

flags 寄存器是计算机的窗口，展示了 CPU 内部各项设置、指标。任何一个指令的执行、其执行过程的细节、对计算机造成了哪些影响，都在 flags 寄存器中通过一些标志位反映出来。

31...	21	20	19	18	17	16	15	14	13-12	11	10	9	8	7	6	5	4	3	2	1	0
	ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF				

图 5

控制寄存器

控制寄存器用来控制 CPU 的运行机制。其中，CR0 的内容如下：

控制寄存器CR0										保留位									
P G	C D	N W			A M	W P				N E	E T	T S	E M	M P	P E				
31	30	29	28		19	18	17	16	15	6	5	4	3	2	1	0			

图 6

表 4-11

控制寄存器 CR0 字段

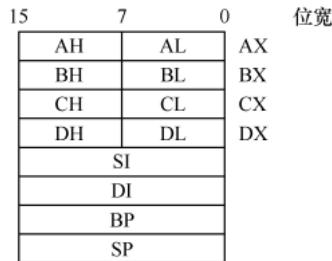
标志位	描述
PE	Protection Enable
MP	Monitor coProcessor/Math Present
EM	Emulation
TS	Task Switched
ET	Extention Type
NE	Numeric Error
WP	Write Protect
AM	Alignment Mask
NW	Not Writethrough
CD	Cache Disable
PG	Paging

图 7

通用寄存器

有 8 个通用寄存器，AX、BX、CX、DX、SI、DI、BP、SP(前四个还可以再分)。

虽说通用，但还是约定了它们的惯用法，除了通用的用途外每个寄存器肩负特定的功用。



▲图 3-6 实模式下通用寄存器

图 8

表 3-2

通用寄存器介绍

寄存器	助记名称	功能描述
ax	累加器 (accumulator)	使用频度最高，常用于算术运算、逻辑运算、保存与外设输入输出的数据
bx	基址寄存器 (base)	常用来存储内存地址，用此地址作为基址，用来遍历一片内存区域
计数器 (counter)	顾名思义，计数器的作用就是计数，所以常用于循环指令中的循环次数	
dx	数据寄存器 (data)	可用于存放数据，通常情况下只用于保存外设控制器的端口号地址
si	源变址寄存器 (source index)	常用于字符串操作中的数据源地址，即被传送的数据在哪里。通常需要与其他指令配合使用，如批量数据传送指令族 movs[bwd]
di	目的变址寄存器 (destination index)	和 si 一样，常用于字符串操作。但 di 是用于数据的目的地址，即数据被传送到哪里
sp	栈指针寄存器 (stack pointer)	其段基址是 SS，用来指向栈顶。随着栈中数据的进出，push 和 pop 这两个对栈操作的指令会修改 sp 的值
bp	基址指针 (base pointer)	访问栈有两种方式，一种是用 push 和 pop 指令操作栈，sp 指针的值会自动更新，但我们只能获取栈顶 sp 指针指向的数据。很多时候，我们需要读写在栈底和栈顶之间的数据，处理器为了让开发人员方便控制栈中数据，还提供了把栈当成数据段来访问的方式，即提供了寄存器 bp，所以 bp 默认的段寄存器就是 SS，可通过 SS: bp 的方式把栈当成普通的数据段来访问，只不过 bp 不像 sp 那样随 push、pop 自动改变

图 9

3.2.3 实模式下的分段

实模式的“实”体现在：程序中用到的地址都是真实的物理地址，“段基址：段内偏移”产生的逻辑地址就是物理地址，也就是程序员看到的完全是真的内存。

8086 的地址总线是 20 位宽，也就是其寻址范围是 2^{20} 次方 =1MB。

为了让 16 位的寄存器寻址能够访问 20 位的地址空，访问内存用“段基址：段内偏移地址”的策略，它是首次在 8086 上出现的。通过先把 16 位的段基址左移 4 位后变成 20 位，再加段内偏移地址，这样便形成了 20 位地址

3.2.4 实模式下 CPU 寻址方式

8086 的寻址模式分类如下：

- 寄存器寻址：“数”在寄存器中，直接从寄存器中拿数据就行了
- 立即数寻址
- 内存寻址：操作数在内存中的寻址方式称为内存寻址。
 - 直接寻址：直接在操作数中给出的数字作为内存地址，通过中括号的形式告诉 CPU，取此地址中的值作为操作数。
 - 基址寻址：就是在操作数中用 bx 寄存器或 bp 寄存器作为地址的起始，地址的变化以它为基础。（保护模式下可以是全部的通用寄存器。），bx 寄存器的默认段寄存器是 DS，而 bp 寄存器的默认段寄存器是 SS。
 - 变址寻址：变址寻址其实和基址寻址类似，只是寄存器由 bx、bp 换成了 si 和 di。si 是指源索引寄存器（source index），di 是指目的索引寄存器（destination index）。两个寄存器的默认段寄存器也是 ds。
 - 基址变址寻址：基址寻址和变址寻址的结合，即基址寄存器 bx 或 bp 加一个变址寄存器 si 或 di。

3.2.5 调用函数的堆栈框架

为了说明 bp 可以用来作为基址寻址，尤其是将栈作为数据段访问，我们来看下面这段程序。

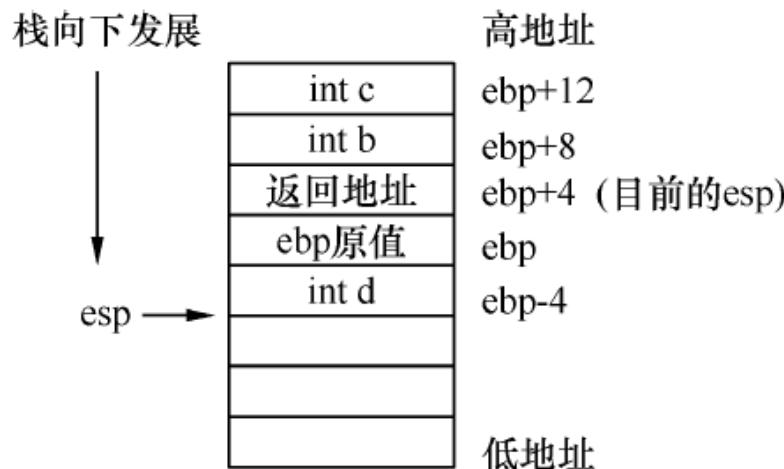
```
1 void function(int a, int b){  
2     int d = 1;  
3 }  
4  
5 int main(){  
6     function(1, 2);  
7     return 0;
```

```
8 }
```

这个简单的程序调用了 function(1, 2)，我们看它对应的汇编语言及解释：

```
1 section .text
2 global _start
3 _start:
4     push 2 ;在执行call指令前会将函数参数按照从右向左的顺序压栈
5     push 1
6     call function ;调用function，同时将返回地址(也就是下一条指令的地址)压栈
7     add esp, 8 ;清理堆栈
8
9     mov eax, 1 ;系统调用号(退出程序)
10    xor ebx, ebx ;退出代码0
11    int 0x80 ;执行系统调用
12
13 function:
14     push ebp
15     → ;ebp是基址寄存器，它是上一个函数栈帧地址，函数之间是层级的调用关系，我们需要先
16     mov ebp, esp ;将当前栈顶指针esp赋给ebp，此时它就是当前函数的栈帧地址
17
18     sub esp, 4
19     → ;栈是向下增长的，分配4个字节变量，并且之后通过ebp相对偏移访问该变量
20     mov dword [ebp - 4], 1; int d = 1;
21
22     mov esp, ebp ;恢复esp，相当于跳过了局部变量
23     pop ebp ;恢复ebp
24     ret ;返回，同时会弹出返回地址，写入eip指令指针寄存器
```

总的来说，该过程图示如下：



▲图 3-8 堆栈框架栈中布局示意图

图 10

当然，为了方便有 enter 指令 (push ebp 再 mov ebp, esp) 和 leave 指令 (mov esp, ebp 再 pop ebp) 这两条集成的指令。

3.3 CPU 访问外设

3.3.1 IO 接口

IO 接口是 CPU 访问外设的桥梁，其功能是协调 CPU 和外设之间的种种不匹配，比如显卡和声卡都是 IO 接口，一个 IO 接口上面有多个端口（寄存器），同一时刻 CPU 只能与一个端口通信，IO 接口的功能如下：

- 设置数据缓冲，解决 CPU 与外设的速度不匹配。
- 设置信号电平转换电路。CPU 和外设的信号电平不同，如 CPU 所用的信号是 TTL 电平，而外设大多数是机电设备，可以在接口电路中设置电平转换电路来解决。
- 设置数据格式转换。CPU 只能处理数字信号，而外设是多种多样的，输出的信息可能是数字信号、模拟信号等，接口电路中需要包括 A/D 转换器和 D/A 转换器，此外还需要并行或串行数据的转换。
- 设置时序控制电路来同步 CPU 和外部设备。
- 提供地址译码。IO 接口提供地址译码电路，使 CPU 可以选中某个端口，使其可以访问数据总线。

输入输出控制中心 (I/O control hub, ICH)，也就是南桥芯片，CPU 通过内部总线连接到南桥芯片中的内部。

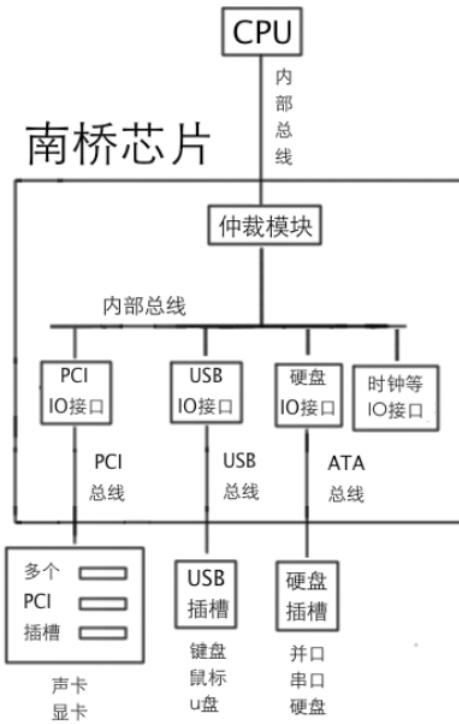


图 11

CPU 提供了专门的指令访问端口，分别是 in/out。

- in 指令用于从端口中读取数据。

1 (1) `in al, dx; ;al` 和 `ax` 用来存储从端口获取的数据，`dx` 是指端口号。
2 (2) `in ax, dx.`; `in` 指令中，端口号只能用 `dx` 寄存器。

- out 指令用于往端口中写数据

1 (1) `out dx, al;`
2 (2) `out dx, ax;`
3 (3) `out 立即数, al;`
4 (4) `out 立即数, ax.`; `out` 指令中，可以选用 `dx` 寄存器或立即数充当端口号。

3.3.2 显卡的访问

某些 IO 接口也叫适配器，适配器是驱动某一外部设备的功能模块。显卡也称为显示适配器，专门用来连接 CPU 和显示器。我们想操作显示器，没有直接的办法，只能通过它的 IO 接口—显卡。

显卡提供给我们的可编程接口都是一样的：IO 端口和显存。显卡的工作就是不断地读取这块内存，随后将其内容发送到显示器。

屏幕是由密密麻麻的像素组成的，显存中的每一位都对应屏幕上的一个像素点。而用 R 红色、G 绿色、B 蓝色这三种颜色以任意比例混合，可以搭配出其他颜色，在真彩色中，是用 24 位对应一个像素，所以才呈现出彩色 (RGB)。

美国信息互换标准代码 (American Standard Code for Information Interchange, ASCII)。是标准单字节字符编码方案，用于描述纯文本。标准 ASCII 码也叫基本 ASCII 码，用 7 位二进制数表示。要想往显示器或任何一个文本处理系统中输出文本信息，必须按照这套规则来编码了。

显卡的内存布局如下：

显存地址分布			
起 始	结 束	大 小	用 途
C0000	C7FFF	32KB	显示适配器 BIOS
B8000	BFFFF	32KB	用于文本模式显示适配器
B0000	B7FFF	32KB	用于黑白显示适配器
A0000	AFFFF	64KB	用于彩色显示适配器

图 12

显卡支持三种模式，文本模式、黑白图形模式、彩色图形模式。我们只关注文本模式就好了，最终我们要实现类似 Linux 终端那样的字符界面。

显卡的文本模式也是分为多种模式的，用“列数 * 行数”来表示，如 80*25, 40*25, 80*43 或者 80*50，它们的乘积是整个屏幕上可以容纳的字符数。显卡在加电后，默认就置为模式 80*25，也就是一屏可以打印 2000 个字符。我们也在这个默认模式下工作了。

即使在文本模式下，也可以打印出彩色字符。这是因为每个字符在屏幕上都是由 2 个字节来表示的，而且是连续的 2 个字节。屏幕上每个字符的低字节是字符的 ASCII 码，高字节是字符属性元信息。

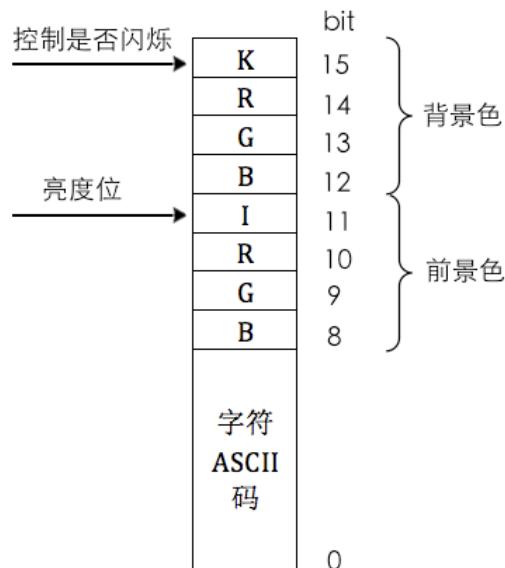


图 13

表 3-16 文本模式中字符颜色

R	G	B	颜 色	
			I=0	I=1
0	0	0	黑	灰
0	0	1	蓝	浅蓝
0	1	0	绿	浅绿
0	1	1	青	浅青
1	0	0	红	浅红
1	0	1	品红	浅品红
1	1	0	棕	黄
1	1	1	白	亮白

图 14

编写一段 MBR 程序如下：

```

1 SECTION MBR vstart=0x7c00
2     mov ax,cs
3     mov ds,ax
4     mov es,ax
5     mov ss,ax
6     mov fs,ax
7     mov sp,0x7c00
8     mov ax,0xb800 ;0xb800是显存文本模式地址
9     mov gs,ax
10

```

```
11 mov ax, 0600h ;这段用于清屏
12 mov bx, 0700h
13 mov cx, 0
14 mov dx, 184fh
15 int 10h
16
17 mov byte [gs:0x00], 'L'
18 mov byte [gs:0x01], 0xA4 ; A表示绿色背景字符闪烁, 4表示前景色为红色
19 mov byte [gs:0x02], 'i'
20 mov byte [gs:0x03], 0xA4
21 mov byte [gs:0x04], 'n'
22 mov byte [gs:0x05], 0xA4
23 mov byte [gs:0x06], 'u'
24 mov byte [gs:0x07], 0xA4
25 mov byte [gs:0x08], 'x'
26 mov byte [gs:0x09], 0xA4
27
28 jmp $           ; 通过死循环使程序悬停在此
29
30 times 510-($-$) db 0
31 db 0x55,0xaa
```

执行结果如下图所示：



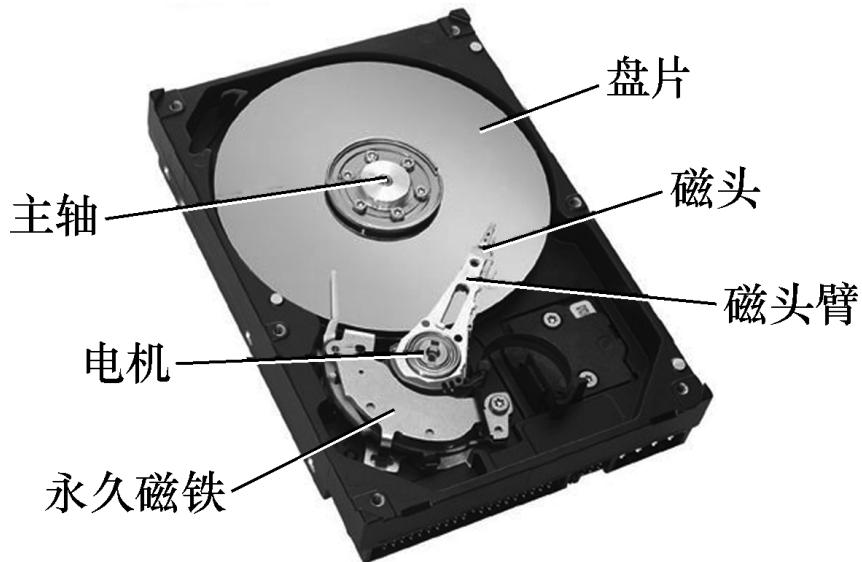
图 15

3.4 硬盘

3.4.1 发展简史

在计算机世界中，实现随机存取具有划时代的意义，程序中的算法不用再把存储时间考虑进去，访问任意数据所用的时间几乎相等，这一改之前的存储设备其存取时间呈线性的历史。

硬盘的随机存取是靠磁头臂不断移动实现的，磁头臂移动到目标位置的时间称为寻道时间。



传统硬盘是机械式硬盘，而 SSD 同 U 盘一样，基于闪存，也就是 Flash Memory，它不存在机械部件，这就是它快的原因。



图 17

3.4.2 硬盘的原理

- 盘片和盘面：每个盘片分上下两面，每个盘面都各由一个磁头来读取数据，一个盘片上对应 2 个磁头。

- 磁头号：用磁头号来表示盘面。磁头编号从上到下以 0 开始计数，用磁头 0 代表第一个盘面。
- 磁道：将整个盘面划分为多个同心环，就是磁道。
- 扇区：以圆心画扇形，扇形与每个同心环相交的弧状区域作为最基本的数据存储单元，大小通常是 512 字节。
- 柱面：磁道的编号从 0 开始，从内到外递增，相同编号的磁道组成的管状区域就称为柱面。柱面的作用是减少寻道次数，当 0 面上的某磁道空间不足时，其他数据写入第 1 面相同编号的磁道上。若新磁道空间还是不足，再写第 2 面相同编号的磁道上，直到同一柱面上的磁道（所有盘面上的编号相同的磁道）都不够用时才会写到新的柱面上。
- 磁头臂带动磁头在盘片上方移动，是在寻找磁道位置，盘片高速自转，是在磁道内定位扇区(由磁头号、磁道号和扇区号确定)。

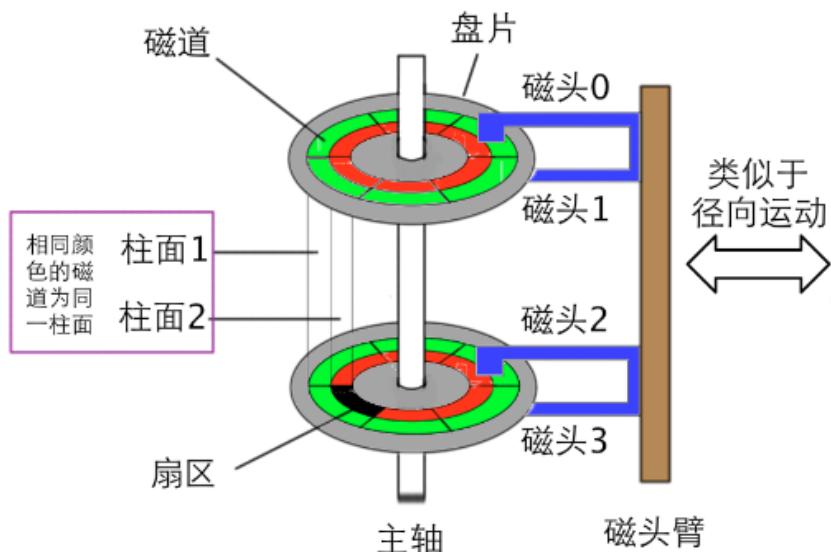


图 18

硬盘的 IO 接口 (寄存器) 是硬盘控制器。, 硬盘控制器和硬盘是连接在一起的，现在一般是硬盘串行接口 (Serial ATA, SATA)。

硬盘控制器的常用端口号如下：

表 3-17

硬盘控制器主要端口寄存器

IO 端口		端口用途	
Primary 通道	Secondary 通道	读操作时	写操作时
Command Block registers			
0x1F0	0x170	Data	Data
0x1F1	0x171	Error	Features
0x1F2	0x172	Sector count	Sector count
0x1F3	0x173	LBA low	LBA low
0x1F4	0x174	LBA mid	LBA mid
0x1F5	0x175	LBA high	LBA high
0x1F6	0x176	Device	device
0x1F7	0x177	Status	Command
Control Block registers			
0x3F6	0x376	Alternate status	Device Control

图 19

端口可以被分为两组，Command Block registers 和 Control Block registers。Command Block registers 用于向硬盘驱动器写入命令字或者从硬盘控制器获得硬盘状态，Control Block registers 用于控制硬盘工作状态。

需要指出的是，上图中的通道对应的是插槽 (，IDE0 叫作 Primary 通道，IDE1 叫作 Secondary 通道)，一个插槽对应两个硬盘，一个主硬盘和一个从硬盘 (可以由 device 寄存器中的第四位指定)，而一个通道往往有多个端口 (寄存器)。

这几个寄存器的介绍如下：

- data 寄存器：data 寄存器就是读取或写入数据的。16 位 (其他 8 位)
- Error/Features 寄存器：读硬盘记录状态信息，写硬盘存储命令的参数。
- Sector count 寄存器用来指定待读取或待写入的扇区数。
- 硬盘中的扇区在物理上是用“磁头-柱面-扇区”来定位的 (Cylinder Head Sector)，简称为 CHS，如果只考虑扇区的逻辑顺序，则用 LBA 编址，全称为逻辑块地址 (Logical Block Address)。LBA 有两种，一种是 LBA28，用 28 位比特来描述一个扇区的地址。另外一种是 LBA48，用 48 位比特来描述一个扇区的地址。
- LBA 寄存器：有 LBA low、LBA mid、LBA high 三个，它们三个都是 8 位宽度的。LBA low 寄存器用来存储 28 位地址的第 0~7 位，LBA mid 寄存器用来存储第 8~15 位，LBA high 寄存器存储第 16~23 位。
- device 寄存器。此寄存器的低 4 位用来存储 LBA 地址的第 24~27 位，第 4 位用来指定通道上的主盘或从盘，0 代表主盘，1 代表从盘。第 6 位用来设置是否启用 LBA 方式，1 代表启用 LBA 模式，0 代表启用 CHS 模式。另外的两位：第 5 位和第 7 位是固定为 1 的，称为 MBS 位。

- status/command 寄存器：读硬盘用来给出硬盘的状态信息。写硬盘用来存储让硬盘执行的命令，通常使用以下三个命令。
 - identify: 0xEC，即硬盘识别。
 - read sector: 0x20，即读扇区。
 - write sector: 0x30，即写扇区。

3.4.3 硬盘的操作

硬盘的一般操作如下：

1. 先选择通道，然后往该通道的 sector count 寄存器中写入待操作的扇区数。
2. 往该通道上的三个 LBA 寄存器写入扇区起始地址的低 24 位。
3. 往 device 寄存器中写入 LBA 地址的 24~27 位，并置第 6 位为 1，使其为 LBA 模式，设置第 4 位，选择操作的硬盘（master 硬盘或 slave 硬盘）。
4. 往该通道上的 command 寄存器写入操作命令。
5. 读取该通道上的 status 寄存器，判断硬盘工作是否完成。
6. 如果以上步骤是读硬盘，进入下一个步骤。否则，完工。
7. 将硬盘数据读出。

3.5 BIOS

事实上，按下主机的 power 启动通电后，第一个运行的程序是 BIOS(Base Input & Output System)，即基本输入输出系统。

intel8086 实模式下的内存布局如下：

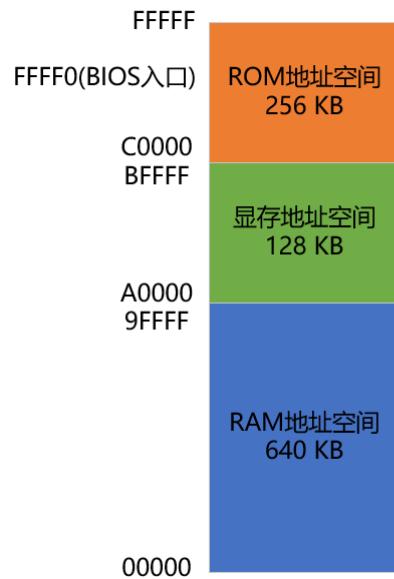


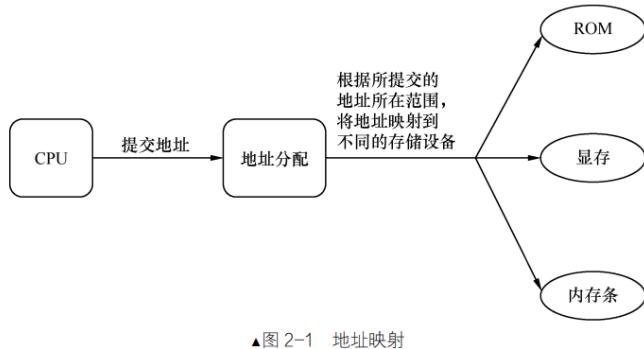
图 20

表 2-1 实模式下的内存布局

起始	结束	大小	用途
FFFF0	FFFFF	16B	BIOS 入口地址，此地址也属于 BIOS 代码，同样属于顶部的 640KB 字节。只是为了强调其入口地址才单独贴出来。此处 16 字节的内容是跳转指令 jmp f000: e05b
F0000	FFFEF	64KB-16B	系统 BIOS 范围是 F0000~FFFFF 共 640KB，为说明入口地址，将最上面的 16 字节从此处去掉了，所以此处终止地址是 0XFFEF
C8000	EFFFF	160KB	映射硬件适配器的 ROM 或内存映射式 I/O
C0000	C7FFF	32KB	显示适配器 BIOS
B8000	BFFFF	32KB	用于文本模式显示适配器
B0000	B7FFF	32KB	用于黑白显示适配器
A0000	AFFFF	64KB	用于彩色显示适配器
9FC00	9FFFF	1KB	EBDA (Extended BIOS Data Area) 扩展 BIOS 数据区
7E00	9FBFF	622080B 约 608KB	可用区域
7C00	7DFF	512B	MBR 被 BIOS 加载到此处，共 512 字节
500	7BFF	30464B 约 30KB	可用区域
400	4FF	256B	BIOS Data Area (BIOS 数据区)
000	3FF	1KB	Interrupt Vector Table (中断向量表)

图 21

这 1MB 空间实际上并没有完全给用户使用，会按照地址空间的不同映射到不同的设备上。



▲图 2-1 地址映射

图 22

具体来说，计算机启动之后的过程如下：

1. 首先 CPU 的 cs: ip 寄存器被强制初始化为 0xf000: 0xffff, 跳转到 BIOS 的入口地址。
2. BIOS 的入口地址实际上是一条跳转指令，jmp far f000: e05b
3. 接着 BIOS 进行一系列操作，包括硬件检测和初始化，以及建立基本的中断向量表 IVT 并填写中断例程
4. 然后 BIOS 会将 0 盘 0 道 1 扇区的内容 (这里是 CHS 编号) 加载到物理地址 0x7c00 处。(原书有误，应该是先加载再检测)
5. 如果此扇区末尾的两个字节分别是魔数 0x55 和 0xaa，BIOS 便认为此扇区中确实存在可执行的程序，也就是说，它是 MBR(master boot record)，MBR 的任务是加载某个程序 (这个程序一般是内核加载器，很少有直接加载内核的)，随后 jmp 0: 0x7c00，也就是执行 MBR。

有一个小问题：为什么是 0x7c00?

当然简单的说就是这么规定的(不过 x86 手册里找不到它的说明)，但还是有一些原因的。

- 最早是 IBM PC 5150 BIOS 的规范
- IBM PC 5150 BIOS 按照最小内存 32KB 开发，考虑到内存大小和布局问题，不影响其他程序，MBR 放在 31KB 处，也就是 0x7c00。

3.6 让 MBR 读取硬盘

MBR 的功能是负责从硬盘上把 loader 加载到内存，并将接力棒交给它。

根据实模式内存空间的可用空间，我们读取 LBA 的第 2 个扇区，将其放置到地址 0x900 处，代码如下：

```

1 ;主引导程序
2 %include "boot.inc"

```

```

3 section MBR vstart=0x7c00
4 mov ax,cs
5 mov ds,ax
6 mov es,ax
7 mov ss,ax
8 mov fs,ax
9 mov sp,0x7c00
10 mov ax,0xb800
11 mov gs,ax
12
13 ;清屏
14 mov ax, 0600h
15 mov bx, 0700h
16 mov cx, 0           ; 左上角: (0, 0)
17 mov dx, 184fh      ; 右下角: (80, 25),
18 ; 因为VGA文本模式中, 一行只能容纳80个字符, 共25行。
19 ; 下标从0开始, 所以0x18=24, 0x4f=79
20 int 10h             ; int 10h
21
22 mov byte [gs:0x00], 'W'
23 mov byte [gs:0x01], 0x24
24
25 mov byte [gs:0x02], 'e'
26 mov byte [gs:0x03], 0x24
27
28 mov byte [gs:0x04], 'l'
29 mov byte [gs:0x05], 0x24 ;A表示绿色背景闪烁, 4表示前景色为红色
30
31 mov byte [gs:0x06], 'c'
32 mov byte [gs:0x07], 0x24
33
34 mov byte [gs:0x08], 'o'
35 mov byte [gs:0x09], 0x24
36
37 mov byte [gs:0x0a], 'm'

```

```
38 mov byte [gs:0x0b], 0x24
39
40 mov byte [gs:0x0c], 'e'
41 mov byte [gs:0x0d], 0x24
42
43 mov byte [gs:0x0e], ' '
44 mov byte [gs:0x0f], 0x24
45
46 mov byte [gs:0x10], 't'
47 mov byte [gs:0x11], 0x24
48 mov byte [gs:0x12], 'o'
49 mov byte [gs:0x13], 0x24
50
51 mov byte [gs:0x14], ' '
52 mov byte [gs:0x15], 0x24
53
54 mov byte [gs:0x16], 'm'
55 mov byte [gs:0x17], 0x24
56
57 mov byte [gs:0x18], 'y'
58 mov byte [gs:0x19], 0x24
59
60 mov byte [gs:0x1a], ' '
61 mov byte [gs:0x1b], 0x24
62
63 mov byte [gs:0x1c], 'L'
64 mov byte [gs:0x1d], 0x24
65
66
67 mov byte [gs:0x1e], 'i'
68 mov byte [gs:0x1f], 0x24
69
70 mov byte [gs:0x20], 'n'
71 mov byte [gs:0x21], 0x24
72
```

```
73 mov byte [gs:0x22], 'u'  
74 mov byte [gs:0x23], 0x24  
75  
76 mov byte [gs:0x24], 'x'  
77 mov byte [gs:0x25], 0x24  
78  
79 mov byte [gs:0x26], '!'  
80 mov byte [gs:0x27], 0x24  
81  
82 mov byte [gs:0x28], ' '  
83 mov byte [gs:0x29], 0x24  
84  
85 mov byte [gs:0x2a], 'a'  
86 mov byte [gs:0x2b], 0x24  
87 mov byte [gs:0x2c], 'u'  
88 mov byte [gs:0x2d], 0x24  
89 mov byte [gs:0x2e], 't'  
90 mov byte [gs:0x2f], 0x24  
91 mov byte [gs:0x30], 'h'  
92 mov byte [gs:0x31], 0x24  
93 mov byte [gs:0x32], 'o'  
94 mov byte [gs:0x33], 0x24  
95 mov byte [gs:0x34], 'r'  
96 mov byte [gs:0x35], 0x24  
97 mov byte [gs:0x36], ':'  
98 mov byte [gs:0x37], 0x24  
99 mov byte [gs:0x38], 'l'  
100 mov byte [gs:0x39], 0x24  
101 mov byte [gs:0x3a], 'y'  
102 mov byte [gs:0x3b], 0x24  
103 mov byte [gs:0x3c], 'h'  
104 mov byte [gs:0x3d], 0x24  
105  
106 mov eax,LOADER_START_SECTOR ; 起始扇区lba地址  
107 mov bx,LOADER_BASE_ADDR ; 写入的地址
```

```
108 mov cx,4      ; 待读入的扇区数
109 call rd_disk_m_16 ; 以下读取程序的起始部分（一个扇区）
110
111
112
113 jmp LOADER_BASE_ADDR
114
115
116
117 ;功能:读取硬盘n个扇区
118 rd_disk_m_16: ;在16位下读取硬盘
119 ;-----
120 ; eax=LBA扇区号
121 ; ebx=将数据写入的内存地址
122 ; ecx=读入的扇区数
123 mov esi,eax ;备份eax
124 mov di,cx    ;备份cx
125 ;读写硬盘:
126 ;第1步：设置要读取的扇区数
127 mov dx,0x1f2
128 mov al,cl
129 out dx,al      ;读取的扇区数
130
131 mov eax,esi ;恢复ax
132
133 ;第2步：将LBA地址存入0x1f3 ~ 0x1f6
134
135 ;LBA地址7~0位写入端口0x1f3
136 mov dx,0x1f3
137 out dx,al
138
139 ;LBA地址15~8位写入端口0x1f4
140 mov cl,8
141 shr eax,cl
142 mov dx,0x1f4
```

```

143 out dx,al

144

145 ;LBA地址23~16位写入端口0x1f5

146 shr eax,cl

147 mov dx,0x1f5

148 out dx,al

149

150 shr eax,cl

151 and al,0x0f ;lba第24~27位

152 or al,0xe0 ; 设置7~4位为1110,表示lba模式

153 mov dx,0x1f6

154 out dx,al

155

156 ;第3步：向0x1f7端口写入读命令，0x20

157 mov dx,0x1f7

158 mov al,0x20

159 out dx,al

160

161 ;第4步：检测硬盘状态

162 .not_ready:

163 ;同一端口，写时表示写入命令字，读时表示读入硬盘状态

164 nop

165 in al,dx

166 and al,0x88 ;第4位为1表示硬盘控制器已准备好数据传输，第7位为1表示硬盘忙

167 cmp al,0x08

168 jnz .not_ready ;若未准备好，继续等。

169

170 ;第5步：从0x1f0端口读数据

171 mov ax, di

172 mov dx, 256

173 mul dx

174 mov cx, ax ; di为要读取的扇区数，一个扇区有512字节，每次读入一个字，

175 ; 共需di*512/2次，所以di*256

176 mov dx, 0x1f0

177 .go_on_read:

```

```
178 in ax,dx  
179 mov [bx],ax  
180 add bx,2  
181 loop .go_on_read  
182 ret  
183  
184 times 510-($-$) db 0  
185 db 0x55,0xaa
```

Listing 1: mbr.S

四、保护模式

Intel 80286 CPU 中首次出现，是 16 位的保护模式，到了 80386 中就是 32 位的保护模式。

4.1 实模式的缺陷

- 实模式操作系统和用户程序特权级相同
- 逻辑地址等于物理地址
- 用户可以自由访问内存地址
- 单道程序
- 最大可用内存 1MB，并且利用效率不高。

注意：32 位 CPU 具有保护模式和实模式两种运行模式，可以兼容实模式下的程序，但并不是变成了 16 位的 CPU，指 32 位的 CPU 处于 16 位运行模式下的状态，其本质上还是 32 位的 CPU。

4.2 扩展

4.2.1 寄存器扩展

寄存器要保持向下兼容，不能推翻之前的方案从头再来，必须在原有的基础上扩展 (extend)，各寄存器在原有 16 位的基础上，再次向高位扩展了 16 位，成为了 32 位寄存器。经过 extend 后的寄存器，统一在名字前加了 e 表示扩展。

但段寄存器还是 16 位。

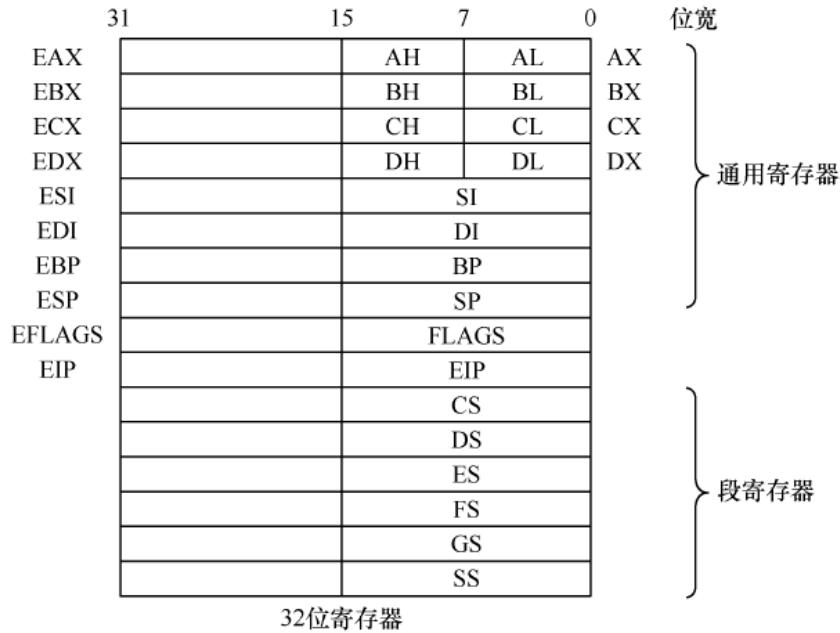


图 23

首款 32 位处理器 80386，它的地址总线和寄存器都是 32 位的，段基址是 32 位，单独的一个寄存器也是 32 位，任意一个段都可以访问到 4GB 空间，段内偏移也可以访问到 4GB 空间。此外，为了兼容之前的实模式程序，有虚拟 8086 模式。

4.2.2 寻址扩展

实模式下基址和变址寄存器有限制，但是在保护模式下不再有限制，如下：

$$\left\{ \begin{array}{l} BX \\ BP \end{array} \right\} + \left\{ \begin{array}{l} SI \\ DI \end{array} \right\} + \left\{ \begin{array}{l} \text{立即数} \end{array} \right\}$$

图 24

$$\left\{ \begin{array}{l} eax \\\dots \\ edx \end{array} \right\} + \left\{ \begin{array}{l} eax \\\dots \\ edx \end{array} \right\} \times \left\{ \begin{array}{l} 1 \\ 2 \\ 4 \\ 8 \end{array} \right\} + \left\{ \begin{array}{l} \text{立即数} \end{array} \right\}$$

图 25

4.2.3 运行模式

CPU 需要兼容实模式和保护模式，为了指定编译后的指令的位数，可用伪指令 bits。
bits 16 是告诉编译器，下面的代码帮我编译成 16 位的机器码。
bits 32 是告诉编译器，下面的代码帮我编译成 32 位的机器码。
•，在未使用 bits 指令的地方，默认是 [bits 16]。

此外，指令的格式如下：



图 26

前缀存放的是指令选项，比如指令重复前缀 rep、段跨越前缀“段寄存器”，操作数反转前缀 0x66 和寻址方式反转前缀 0x67。汇编时会自动根据汇编语言的操作数和寻址方式修改前缀。

4.2.4 指令扩展

在 16 位的实模式下，CPU 照样可以处理 32 位的数据。

4.3 全局描述符表 GDT

全局描述符表（Global Descriptor Table，GDT）是保护模式下内存段的登记表，这是不同于实模式的显著特征之一。

4.3.1 段描述符

全局描述符表中的表项就是段描述符，段描述符是用来描述内存段的属性，该结构是 8 字节大小。



图 27

- 12 位是 S 字段：S 为 0 时表示系统段，S 为 1 时表示数据段。各种称为“门”的结构是系统段，如调用门、任务门。其他都是数据段，包括代码段、数据段、栈段。

- 8~11 位是 type 字段，该字段共 4 位，用于表示内存段或门的子类型。

表 4-10 段描述符的 type 类型

系统段类型	第 3~0 位				说明	
	3	2	1	0		
系统段	未定义	0	0	0	0	保留
	可用的 80286 TSS	0	0	0	1	仅限 286 的任务状态段
	LDT	0	0	1	0	局部描述符表，只有第 1 位为 1
	忙碌的 80286 TSS	0	0	1	1	仅限 286，type 中的第 1 位称 B 位，若为 1，则表示当前任务忙碌。由 CPU 将此位置 1
	80286 调用门	0	1	0	0	仅限 286
	任务门	0	1	0	1	任务门在现代操作系统中很少用到
	80286 中断门	0	1	1	0	仅限 286
	80286 陷阱门	0	1	1	1	仅限 286
	未定义	1	0	0	0	保留
	可用的 80386TSS	1	0	0	1	386 以上 CPU 的 TSS，type 第 3 位为 1
	未定义	1	0	1	0	保留
	忙碌的 80386 TSS	1	0	1	1	386 以上 CPU 的 TSS，type 第 3 位为 1
	80386 调用门	1	1	0	0	386 以上 CPU 的调用门，type 第 3 位为 1
	未定义	1	1	0	1	保留
非系统段	中断门	1	1	1	0	386 以上 CPU 的中断门
	陷阱门	1	1	1	1	386 以上 CPU 的陷阱门
对于非系统段，按代码段和数据段划分，这 4 位分别有不同的意义						
内存段类型	X	R	C	A	说明	
代码段	1	0	0	*	只执行代码段	
	1	1	0	*	可执行、可读代码段	
	1	0	1	*	可执行、一致性代码段	
	1	1	1	*	可执行、可读、一致性代码段	
非系统段	内存段类型	X	W	E	A	说明
	数据段	0	0	0	*	只读数据段
		0	1	0	*	可读写数据段
		0	0	1	*	只读，向下扩展的数据段
		0	1	1	*	可读写，向下扩展的数据段

图 28

当然，经过检验，上面的 R、C 位以及 W、E 位标反了，正确的顺序是下面：

X	E	W	A	描述符类别	含义
0	0	0	×	数据	只读
0	0	1	×		读、写
0	1	0	×		只读，向下扩展
0	1	1	×		读、写，向下扩展
X	C	R	A	描述符类别	含义
1	0	0	×	代码	只执行
1	0	1	×		执行、读
1	1	0	×		只执行，依从的代码段
1	1	1	×		执行、读，依从的代码段

图 29

- 13~14 位是 DPL，Descriptor Privilege Level，即描述符特权级。这两位能表示 4 种特权级，分别是 0、1、2、3 级特权，数字越小，特权级越大。
- 第 15 位是 P 字段，Present，即段是否存在。如果段存在于内存中，P 为 1，否则 P 为 0。
- 第 20 位为 AVL 字段，可用的。
- 第 21 位为 L 字段，用来设置是否是 64 位代码段。L 为 1 表示 64 位代码段，否则表示 32 位代码段。

- 第 22 位是 D/B 字段，用来指示有效地址（段内偏移地址）及操作数的大小。对于代码段来说，此位是 D 位，对于栈段来说，此位是 B 位。
- 第 23 位是 G 字段，Granularity，粒度，用来指定段界限的单位大小。若 G 为 0，表示段界限的单位是 1 字节，这样段最大是 2 的 20 次方 *1 字节，即 1MB。若 G 为 1，表示段界限的单位是 4KB，这样段最大是 2 的 20 次方 *4KB 字节，即 4GB。

4.3.2 GDT

段描述符放在 GDT (Global Descriptor Table)，全局描述符表 GDT 相当于是一个描述符的数组，数组中的每个元素都是 8 字节的描述符。可以用选择子中提供的下标在 GDT 中索引描述符。(GDT 中第 0 个段描述符不可用)

全局体现在多个程序都可以在里面定义自己的段描述符，是公用的。全局描述符表位于内存中，需要用专门的寄存器指向它，这个专门的寄存器便是 GDTR，即 GDT Register。GDTR 是一个 48 位的寄存器。

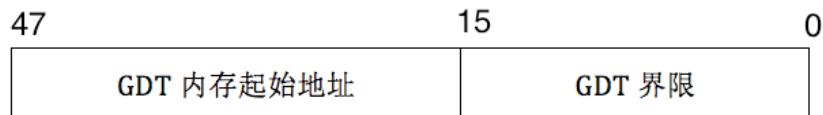


图 30

lgdt 指令用来初始化 gdtr，格式是：lgdt48 位数据。

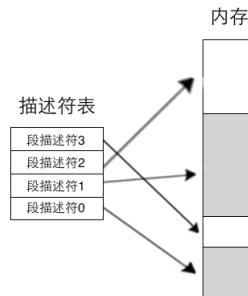


图 31

4.3.3 segment selector

段寄存器 CS、DS、ES、FS、GS、SS，在实模式下时，段中存储的是段基址，即内存段的起始地址。而在保护模式下时，在段寄存器中存入的是一个叫作选择子的东西 —selector。

段选择子的结构如下：

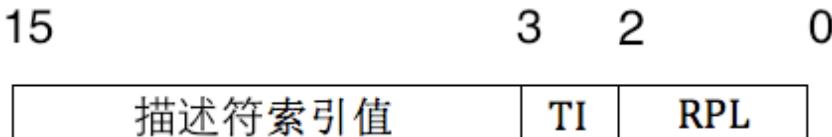


图 32

- 在选择子的第 2 位是 TI 位，即 Table Indicator，用来指示选择子是在 GDT 中，还是 LDT 中索引描述符。TI 为 0 表示在 GDT 中索引描述符，TI 为 1 表示在 LDT 中索引描述符。
- 第 3~15 位是描述符的索引值，用此值在 GDT 中索引描述符。
- 选择子对应的“段描述符中的段基址”加上“段内偏移地址”就是要访问的内存地址。

GDT 中的第 0 个段描述符是不可用的，否则会抛出异常，这是为了避免忘记初始化选择子。

4.3.4 LDT

局部描述符表，叫 LDT，Local Descriptor Table。CPU 厂商建议每个任务的私有内存段都应该放到自己的段描述符表中，该表就是 LDT，即每个任务都有自己的 LDT，随着任务切换，也要切换相应任务的 LDT。每切换任务时，都要用 lldt 指令重新加载任务的私有内存段。

LDT 虽然是个表，但也是一个系统段，需要用个描述符在 GDT 中先注册，再用选择子进行访问。因此 lldt 指令的格式是：lldt 16 位寄存器/16 位内存(相当于一个段选择子)。

在 LDT 被加载到 ldtr 寄存器后，之后再访问某个段时，选择子中的 TI 位若为 1，就会用该选择子中的高 13 位在 ldtr 寄存器所指向的 LDT 中去索引相应段描述符。

当然，其实在现代操作系统中很少有用 LDT 的，我们系统中也未用 LDT。

4.3.5 进入保护模式

经过测试和 debug，原书其实有个小错误，那就是定义显存段的高 4 位段界限时后面少了一个 0(已经补上，尽管这不影响结果)。此外，显存段的段基地址的高 8 位是 0x0b(书上错了，给的代码没错)。

首先是 boot.inc，我们提前定义一些需要的宏：

```

1 ;这个文件定义一些需要的宏
2
3 LOADER_BASE_ADDR equ 0x900 ;loader在内存中的地址

```

```

4 LOADER_START_SECTOR equ 0x2 ;loader的逻辑扇区地址
5
6
7 ;下面是GDT描述符所需的一些属性，我们借助二进制预先定义，由于使用平坦模型，因此段基址是0
8 DESC_G_4K equ 1_000000000000000000000000000000b
    ↪ ;granularity,粒度，为1代表段界限单位4KB
9 DESC_D_32 equ 1_000000000000000000000000000000b ;
    ↪ D/B位，代码段是D位，栈段是B位，32位下设置为1
10 DESC_L equ 0_000000000000000000000000000000b ;
    ↪ L=1代表64位代码段，L=0代表32位代码段
11 DESC_AVL equ 0_000000000000000000000000000000b ; cpu不用此位，暂置为0
12 DESC_LIMIT_CODE2 equ 1111_000000000000000000000000b ;
    ↪ 段界限位，这里设置成最大，也就是4GB
13 DESC_LIMIT_DATA2 equ DESC_LIMIT_CODE2 ; 同样是4GB
14 DESC_LIMIT_VIDEO2 equ 0000_000000000000000000b ; 显存段的高4位是0
15 DESC_P equ 1_0000000000000000b ; Present位，为1代表存在内存中
16 DESC_DPL_0 equ 00_00000000000000b ;0号特权级
17 DESC_DPL_1 equ 01_00000000000000b ;1号特权级
18 DESC_DPL_2 equ 10_00000000000000b ;2号特权级
19 DESC_DPL_3 equ 11_00000000000000b ;3号特权级
20 DESC_S_CODE equ 1_000000000000b ;S位，为1代表非系统段
21 DESC_S_DATA equ DESC_S_CODE ;数据段同样是非系统段
22 DESC_S_sys equ 0_000000000000b ;系统段的S位设置为0
23 DESC_TYPE_CODE equ 1000_00000000b ;这是type位，x=1,c=0,r=0,a=0
    ↪ 代码段是可执行，非可读，非一致性，已访问位a清0.
24 DESC_TYPE_DATA equ 0010_00000000b ;x=0,e=0,w=1,a=0
    ↪ 数据段是不可执行的，向上扩展，可读可写，已访问位a清0.
25
26 ;下面是定义的几个段
27 DESC_CODE_HIGH4 equ (0x00 << 24) + DESC_G_4K + DESC_D_32 + DESC_L +
    ↪ DESC_AVL + DESC_LIMIT_CODE2 + DESC_P + DESC_DPL_0 + DESC_S_CODE +
    ↪ DESC_TYPE_CODE + 0x00
28 DESC_DATA_HIGH4 equ (0x00 << 24) + DESC_G_4K + DESC_D_32 + DESC_L +
    ↪ DESC_AVL + DESC_LIMIT_DATA2 + DESC_P + DESC_DPL_0 + DESC_S_DATA +
    ↪ DESC_TYPE_DATA + 0x00

```

```

29 DESC_VIDEO_HIGH4 equ (0x00 << 24) + DESC_G_4K + DESC_D_32 + DESC_L +
    ↪ DESC_AVL + DESC_LIMIT_VIDEO02 + DESC_P + DESC_DPL_0 + DESC_S_DATA +
    ↪ DESC_TYPE_DATA + 0x0b

30
31
32
33 ;下面是段选择子的属性
34 RPL0 equ 00b
35 RPL1 equ 01b
36 RPL2 equ 10b
37 RPL3 equ 11b
38 TI_GDT equ 000b
39 TI_LDT equ 100b

```

Listing 2: boot.inc

此外，是主引导扇区 mbr:

```

1 ;主引导程序
2 %include "boot.inc"
3 section MBR vstart=0x7c00
4 mov ax,cs
5 mov ds,ax
6 mov es,ax
7 mov ss,ax
8 mov fs,ax
9 mov sp,0x7c00
10 mov ax,0xb800
11 mov gs,ax
12
13 ;下面代码的功能是实现清屏
14 mov ax, 0600h
15 mov bx, 0700h ; 因为VGA文本模式中，一行只能容纳80个字符，共25行。
16 mov cx, 0           ; 左上角：(0, 0)
17 mov dx, 184fh      ; 右下角：(80, 25),；下标从0开始，所以0x18=24,0x4f=79
18 int 10h            ; int 10h，调用中断处理程序
19

```

```
20  
21 mov byte [gs:0x00], 'W'  
22 mov byte [gs:0x01], 0x24  
23 mov byte [gs:0x02], 'e'  
24 mov byte [gs:0x03], 0x24  
25 mov byte [gs:0x04], 'l'  
26 mov byte [gs:0x05], 0x24 ;A表示绿色背景闪烁, 4表示前景色为红色  
27 mov byte [gs:0x06], 'c'  
28 mov byte [gs:0x07], 0x24  
29 mov byte [gs:0x08], 'o'  
30 mov byte [gs:0x09], 0x24  
31 mov byte [gs:0x0a], 'm'  
32 mov byte [gs:0x0b], 0x24  
33 mov byte [gs:0x0c], 'e'  
34 mov byte [gs:0x0d], 0x24  
35 mov byte [gs:0x0e], ' '  
36 mov byte [gs:0x0f], 0x24  
37  
38 mov byte [gs:0x10], 't'  
39 mov byte [gs:0x11], 0x24  
40 mov byte [gs:0x12], 'o'  
41 mov byte [gs:0x13], 0x24  
42 mov byte [gs:0x14], ' '  
43 mov byte [gs:0x15], 0x24  
44  
45 mov byte [gs:0x16], 'm'  
46 mov byte [gs:0x17], 0x24  
47 mov byte [gs:0x18], 'y'  
48 mov byte [gs:0x19], 0x24  
49 mov byte [gs:0x1a], ' '  
50 mov byte [gs:0x1b], 0x24  
51  
52 mov byte [gs:0x1c], 'L'  
53 mov byte [gs:0x1d], 0x24  
54 mov byte [gs:0x1e], 'i'
```

```
55 mov byte [gs:0x1f], 0x24
56 mov byte [gs:0x20], 'n'
57 mov byte [gs:0x21], 0x24
58 mov byte [gs:0x22], 'u'
59 mov byte [gs:0x23], 0x24
60 mov byte [gs:0x24], 'x'
61 mov byte [gs:0x25], 0x24
62 mov byte [gs:0x26], '!'
63 mov byte [gs:0x27], 0x24
64 mov byte [gs:0x28], ' '
65 mov byte [gs:0x29], 0x24
66
67 mov byte [gs:0x2a], 'A'
68 mov byte [gs:0x2b], 0x24
69 mov byte [gs:0x2c], 'u'
70 mov byte [gs:0x2d], 0x24
71 mov byte [gs:0x2e], 't'
72 mov byte [gs:0x2f], 0x24
73 mov byte [gs:0x30], 'h'
74 mov byte [gs:0x31], 0x24
75 mov byte [gs:0x32], 'o'
76 mov byte [gs:0x33], 0x24
77 mov byte [gs:0x34], 'r'
78 mov byte [gs:0x35], 0x24
79 mov byte [gs:0x36], ':'
80 mov byte [gs:0x37], 0x24
81 mov byte [gs:0x38], 'l'
82 mov byte [gs:0x39], 0x24
83 mov byte [gs:0x3a], 'y'
84 mov byte [gs:0x3b], 0x24
85 mov byte [gs:0x3c], 'h'
86 mov byte [gs:0x3d], 0x24
87
88
89
```

```

90 mov eax,LOADER_START_SECTOR ; 起始扇区lba地址
91 mov bx,LOADER_BASE_ADDR    ; 写入到内存中的地址
92 mov cx,4                  ; 待读入的扇区数
93 call rd_disk_m_16          ; 近调用读取扇区的代码, 读取loader, 最后会ret近返回
94
95
96
97 jmp LOADER_BASE_ADDR      ;跳转到loader程序处的内存地址去执行
98
99
100
101 ;功能:读取硬盘n个扇区
102 rd_disk_m_16: ;在16位下读取硬盘
103 ;-----
104 ; eax=LBA扇区号
105 ; ebx=将数据写入的内存地址
106 ; ecx=读入的扇区数
107 mov esi,eax ;备份eax
108 mov di,cx    ;备份cx
109 ;读写硬盘:
110 ;第1步: 设置要读取的扇区数
111 mov dx,0x1f2
112 mov al,cl
113 out dx,al      ;读取的扇区数
114
115 mov eax,esi    ;恢复ax
116
117 ;第2步: 将LBA地址存入0x1f3 ~ 0x1f6
118
119 ;LBA地址7~0位写入端口0x1f3
120 mov dx,0x1f3
121 out dx,al
122
123 ;LBA地址15~8位写入端口0x1f4
124 mov cl,8

```

```

125 shr eax,cl
126 mov dx,0x1f4
127 out dx,al
128
129 ;LBA地址23~16位写入端口0x1f5
130 shr eax,cl
131 mov dx,0x1f5
132 out dx,al
133
134 shr eax,cl
135 and al,0x0f ;lba第24~27位
136 or al,0xe0 ; 设置7~4位为1110,表示lba模式
137 mov dx,0x1f6
138 out dx,al
139
140 ;第3步: 向0x1f7端口写入读命令, 0x20
141 mov dx,0x1f7
142 mov al,0x20
143 out dx,al
144
145 ;第4步: 检测硬盘状态
146 .not_ready:
147 ;同一端口, 写时表示写入命令字, 读时表示读入硬盘状态
148 nop
149 in al,dx
150 and al,0x88 ;第4位为1表示硬盘控制器已准备好数据传输, 第7位为1表示硬盘忙
151 cmp al,0x08
152 jnz .not_ready ;若未准备好, 继续等。
153
154 ;第5步: 从0x1f0端口读数据
155 mov ax, di
156 mov dx, 256
157 mul dx
158 mov cx, ax ; di为要读取的扇区数, 一个扇区有512字节, 每次读入一个字,
159 ; 共需di*512/2次, 所以di*256

```

```

160 mov dx, 0x1f0
161 .go_on_read:
162 in ax,dx
163 mov [bx],ax
164 add bx,2
165 loop .go_on_read
166 ret
167
168 times 510-($-$) db 0
169 db 0x55,0xaa

```

Listing 3: mbr.S

最后，是 loader.S:

```

1 %include "boot.inc"
2
3 section loader vstart=LOADER_BASE_ADDR
4 LOADER_STACK_TOP equ LOADER_BASE_ADDR
5 ;将loader加载到的地址0x900下面的地址作为栈
6 jmp loader_start
7
8 ;构建gdt及其内部的描述符
9 GDT_BASE:
10 dd 0x00000000 ;第一个描述符没用
11 dd 0x00000000
12
13 CODE_DESC:
14 dd 0x0000FFFF ;代码段描述符
15 dd DESC_CODE_HIGH4
16
17 DATA_STACK_DESC:
18 dd 0x0000FFFF ;栈段和数据段的描述符
19 dd DESC_DATA_HIGH4
20
21 VIDEO_DESC:
22 dd 0x80000007 ;limit=(0xffff-0xb8000)/4k=0x7
23 dd DESC_VIDEO_HIGH4 ; 此时dpl已改为0

```

```

22
23 GDT_SIZE equ $ - GDT_BASE ;得到GDT表的长度
24 GDT_LIMIT equ GDT_SIZE - 1 ;得到偏移长度
25 times 60 dq 0 ;此处预留60个描述符的
26
27 ;下面构建相应的段选择子
28 SELECTOR_CODE equ (0x0001<<3) + TI_GDT + RPL0 ;
29 SELECTOR_DATA equ (0x0002<<3) + TI_GDT + RPL0 ;
30 SELECTOR_VIDEO equ (0x0003<<3) + TI_GDT + RPL0 ;
31
32 ;以下是定义gdt的指针，前2字节是gdt界限，后4字节是gdt起始地址
33
34 gdt_ptr dw GDT_LIMIT
35 dd GDT_BASE
36
37 loadermsg db '2 loader in real.'
    ↪ ;实模式下利用中断打印字符串显示要进入保护模式
38
39 loader_start: ;此处的物理地址是0xb1a
40 ;打印字符，"2 LOADER"说明loader已经成功加载
41 ; 输出背景色绿色，前景色红色，并且跳动的字符串"1 MBR"
42 mov byte [gs:160], '2'
43 mov byte [gs:161], 0xA4 ; A表示绿色背景闪烁，4表示前景色为红色
44
45 mov byte [gs:162], ' '
46 mov byte [gs:163], 0xA4
47
48 mov byte [gs:164], 'L'
49 mov byte [gs:165], 0xA4
50
51 mov byte [gs:166], 'O'
52 mov byte [gs:167], 0xA4
53
54 mov byte [gs:168], 'A'
55 mov byte [gs:169], 0xA4

```

```

56
57 mov byte [gs:170], 'D'
58 mov byte [gs:171], 0xA4
59
60 mov byte [gs:172], 'E'
61 mov byte [gs:173], 0xA4
62
63 mov byte [gs:174], 'R'
64 mov byte [gs:175], 0xA4
65
66
67
68
69 ;-----
70 ; INT 0x10 功能号:0x13 功能描述:打印字符串
71 ;-----
72 ;输入:
73 ;AH 子功能号=13H
74 ;BH = 页码
75 ;BL = 属性(若AL=00H或01H)
76 ;CX=字符串长度
77 ;(DH、DL)=坐标(行、列)
78 ;ES:BP=字符串地址
79 ;AL=显示输出方式
80 ; 0——字符串中只含显示字符，其显示属性在BL中。显示后，光标位置不变
81 ; 1——字符串中只含显示字符，其显示属性在BL中。显示后，光标位置改变
82 ; 2——字符串中含显示字符和显示属性。显示后，光标位置不变
83 ; 3——字符串中含显示字符和显示属性。显示后，光标位置改变
84 ;无返回值
85 mov sp, LOADER_BASE_ADDR
86 mov bp, loadermsg      ; ES:BP = 字符串地址
87 mov cx, 17      ; CX = 字符串长度
88 mov ax, 0x1301    ; AH = 13, AL = 01h
89 mov bx, 0x001f    ; 页号为0(BH = 0) 蓝底粉红字(BL = 1fh)
90 mov dx, 0x1800    ;

```

```

91 int    0x10          ; 10h 号中断
92
93 ;----- 准备进入保护模式
94 ; 1 打开A20
95 ; 2 加载gdt
96 ; 3 将cr0的pe位置1
97
98
99 ;----- 1.打开A20地址线 -----
100 in al,0x92
101 or al,0000_0010B
102 out 0x92,al
103
104 ;----- 2.加载GDT -----
105 lgdt [gdt_ptr]
106
107
108 ;----- 3.cr0第0位置1, 即PE 位, Protection Enable
109 mov eax, cr0
110 or eax, 0x00000001
111 mov cr0, eax
112
113 ;jmp dword SELECTOR_CODE:p_mode_start ;
114     → 刷新流水线, 避免分支预测的影响, 这种cpu优化策略, 最怕jmp跳转,
114 jmp  SELECTOR_CODE:p_mode_start
115     → ;由于下面是32位操作码, 为了避免上面指令分支预测的影响, 使用jmp无条件跳转, 从而刷新
116
117 [bits 32]
118 p_mode_start: ;用选择子初始化各段寄存器
119     mov ax, SELECTOR_DATA
120     mov ds, ax
121     mov es, ax
122     mov ss, ax

```

```

122 mov esp,LOADER_STACK_TOP
123 mov ax, SELECTOR_VIDEO
124 mov gs, ax
125
126 mov byte [gs:320], 'P'
127
128 jmp $

```

Listing 4: loader.S

运行代码后的结果如下, 成功实现保护模式的跳转:

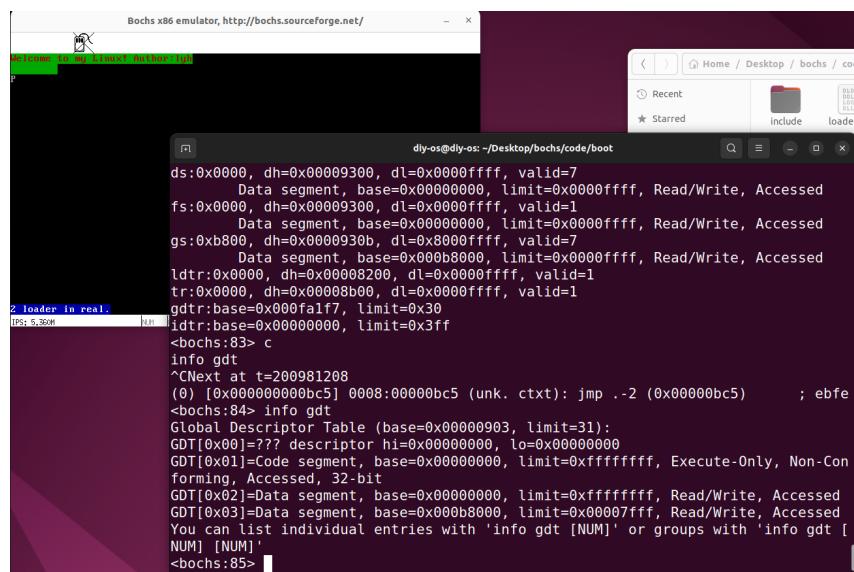


图 33

4.4 处理器微架构

我们先了解一下处理器的微架构。

4.4.1 流水线

流水线是 CPU 内的一项十分重要的技术。

单核 CPU 一次只能执行一条指令, 指令执行的过程分为取指令、译码和执行指令的过程, 但是, 每个步骤都是独立执行, CPU 可以在每个周期内并行执行这三个步骤。

表 4-14 三级流水线						
指令	周期 1	周期 2	周期 3	周期 4	周期 5	周期 6
第一条指令	取指	译码	执行			
第二条指令		取指	译码	执行		
第三条指令			取指	译码	执行	

图 34

CPU 是按照程序中指令顺序来填充流水线的，也就是说按照程序计数器 PC (x86 中是 cs: ip) 中的值来装载流水线的，当前指令和下一条指令在空间上是挨着的，但是，如果遇到无条件转移指令 jmp 时，会清空流水线。

当然，流水线还可以优化，也就是将指令执行的每一步操作进一步细分，缩短最终指令执行的最后一步所需的时间，提升效率。

4.4.2 乱序执行

乱序执行就是对于那些彼此无关的指令打乱顺序执行，后面的操作可以放到前面来做，利于装载到流水线上提高效率。

x86 最初用的指令集是 CISC (Complex Instruction Set Computer)，意为复杂指令集计算机，当初的 CPU 工程师们为了让 CPU 更加强大，不断地往 CPU 中添加各种指令，甚至在 CPU 硬件一级直接支持软件中的某些操作，以至于指令集越来越庞大复杂。比如 push 指令，它相当于两个个微操作的合成：

1. 先将栈指针 esp 减去操作数的字长，如 sub esp,4。
2. 再将操作数 mov 到新的 esp 指向的地址，如 mov [esp],eax。

而 RISC (Reduced Instruction Set Computer)，意为精简指令集计算机，固定了指令的格式和长度，精简保留了那些常用的指令，这些指令大多数都是不可再细分的，也就是微操作级别的指令。

x86 后面虽然还是 CISC 指令集，但其内部已经采用 RISC 内核，译码对于 x86 体系来说，除了按照指令格式分析机器码外，还要将 CISC 指令分解成多个 RISC 指令。当一个“大”操作被分解成多个“微”操作时，它们之间通常独立无关联，所以非常适合乱序执行。

4.4.3 分支预测

CPU 中的指令是在流水线上执行。分支预测，是指当处理器遇到一个分支指令时，考虑将何条指令放置于流水线上。

根据程序的局部性，最简单的想法是根据上一次跳转的结果来预测本次，如果上一次跳转啦，这一次也预测为跳转，否则不跳。

Intel 的分支预测部件中用了分支目标缓冲器 (Branch Target Buffer, BTB)。BTB 中记录着分支指令地址，CPU 遇到分支指令时，先用分支指令的地址在 BTB 中查找，若找到相同地址的指令，根据跳转统计信息判断是否把相应的预测分支地址上的指令送上流水线。在真正执行时，根据实际分支流向，更新 BTB 中跳转统计信息。

当然，如果指令地址在该表中未查到，是第一次，使用 Static Predictor，静态预测器，存储在里面的预测策略是固定写死的，它是由人们经过大量统计之后，根据某些特

征总结出来的。。

分支指令所在地址	预测的分支地址	跳转统计

图 35

4.4.4 缓存

缓存就是用一些存取速度较快的存储设备作为数据缓冲区，避免频繁访问速度较慢的低速存储设备。

CPU 和 DRAM 之间的缓存是 SRAM，CPU 中有一级缓存 L1、二级缓存 L2，甚至三级缓存 L3 等。它们都是 SRAM，即静态随机访问存储器。

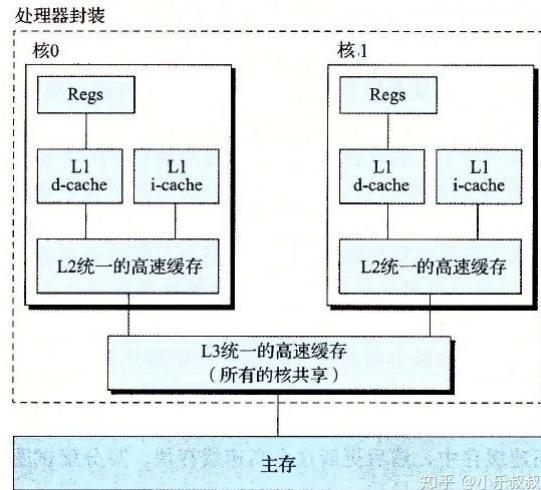


图 36

缓存利用了程序的局部性。

- 时间局部性：最近访问过的指令和数据，在将来一段时间内依然经常被访问。
- 空间局部性：靠近当前访问内存空间的内存地址，在将来一段时间也会被访问。

4.5 段的保护

4.5.1 内存段的保护

加载段选择子时首先会检查段的类型：

表 4-17

段寄存器	代码段 (X=1)		数据段 (X=0)	
	只执行 (R=0)	执行+可读 (R=1)	只读 (R=1, W=0)	读写 (W=1)
CS	通过	通过	不通过	不通过
DS	不通过	通过	通过	通过
ES	不通过	通过	通过	通过
FS	不通过	通过	通过	通过
GS	不通过	通过	通过	通过
SS	不通过	不通过	不通过	通过

图 37

此外，还会检查不能超出段的界限。

五、内核的完善

5.1 获取物理内存容量

Linux 获取内存容量通常是通过 BIOS 中断 0x15 实现的，它的子功能号存放在 eax 寄存器中，它有三个子功能：

1. EAX=0xE820：遍历主机上全部内存(内存最大的即为主板上配置的物理内存容量)。
 2. AX=0xE801：分别检测低 15MB 和 16MB~4GB 的内存，最大支持 4GB。
 3. AH=0x88：最多检测出 64MB 内存，实际内存超过此容量也按照 64MB 返回。
- BIOS 的 0x15 中断子功能 0xE820 如下：

表 5-3 BIOS 中断 0x15 子功能 0xE820 说明

调用或返回	寄存器或状态位	参 数 用 途
调用前输入	EAX	子功能号：EAX 寄存器用来指定子功能号，此处输入为 0xE820
	EBX	ARDS 后续值：内存信息需要按类型分多次返回，由于每次执行一次中断都只返回一种类型的内存的 ARDS 结构，所以要记录下一个待返回的内存 ARDS，在下一次中断调用时通过此值告诉 BIOS 该返回哪个 ARDS，这就是后续值的作用。第一次调用时一定要置为 0，EBX 具体值我们不用关注，取决于具体 BIOS 的实现。每次中断返回后，BIOS 会更新此值
	ES: DI	ARDS 缓冲区：BIOS 将获取到的内存信息写入此寄存器指向的内存，每次都以 ARDS 格式返回

图 38

续表

调用或返回	寄存器或状态位	参 数 用 途
调用前输入	ECX	ARDS 结构的字节大小：用来指示 BIOS 写入的字节数。调用者和 BIOS 都同时支持的大小是 20 字节，将来也许会扩展此结构
	EDX	固定为签名标记 0x534d4150，此十六进制数字是字符串 SMAP 的 ASCII 码：BIOS 将调用者正在请求的内存信息写入 ES: DI 寄存器所指向的 ARDS 缓冲区后，再用此签名校验其中的信息
返回后输出	CF 位	若 CF 位为 0 表示调用未出错，CF 为 1，表示调用出错
	EAX	字符串 SMAP 的 ASCII 码 0x534d4150
	ES:DI	ARDS 缓冲区地址，同输入值是一样的，返回时此结构中已经被 BIOS 填充了内存信息
	ECX	BIOS 写入到 ES:DI 所指向的 ARDS 结构中的字节数，BIOS 最小写入 20 字节
	EBX	后续值：下一个 ARDS 的位置。每次中断返回后，BIOS 会更新此值，BIOS 通过此值可以找到下一个待返回的 ARDS 结构，咱们不需要改变 EBX 的值，下一次中断调用时还会用到它。在 CF 位为 0 的情况下，若返回后的 EBX 值为 0，表示这是最后一个 ARDS 结构

图 39

- BIOS0x15 中断的子功能 0xE801 如下：

表 5-4 BIOS 中断 0x15 子功能 0xE801 说明			
调用或返回	寄存器或状态位	用途	描述
返回后输出	AX	Function Code	子功能号: 0xE801
	CF 位	Carry Flag	若 CF 位为 0 表示调用未出错, CF 为 1, 表示调用出错
	AX	Extended 1	以 1KB 为单位, 只显示 15MB 以下的内存容量, 故最大值为 0x3C00, 即 AX 表示的最大内存为 $0x3C00 * 1024 = 15MB$
	BX	Extended 2	以 64KB 为单位, 内存空间 16MB ~ 4GB 中连续的单位数量, 即内存大小为 $BX * 64 * 1024$ 字节
	CX	Configured 1	同 AX
	DX	Configured 2	同 BX

图 40

使用的一个示例如下:

表 5-5 BIOS 中断 0x15 子功能 0xE801 实例			
实际物理内存	AX	BX	检测到的内存大小
14MB	0x3400	0	$AX * 1024 + BX * 64 * 1024 = 13MB$
15MB	0x3800	0	$AX * 1024 + BX * 64 * 1024 = 14MB$
16MB	0x3C00	0	$AX * 1024 + BX * 64 * 1024 = 15MB$
17MB	0x3C00	0x10	$AX * 1024 + BX * 64 * 1024 = 16MB$
18MB	0x3C00	0x20	$AX * 1024 + BX * 64 * 1024 = 17MB$

图 41

由于历史原因存在内存空洞, 实际物理内存大小要加 1MB。

- BIOS 中断 0x15 子功能 0x88 如下:

表 5-6 BIOS 中断 0x15 子功能 0x88 说明		
调用或返回	寄存器或状态位	参数用途
返回后输出	AH	子功能号: 0x88
	CF 位	若 CF 位为 0 表示调用未出错, CF 为 1, 表示调用出错
	AX	以 1KB 为单位大小, 内存空间 1MB 之上的连续单位数量, 不包括低端 1MB 内存。故内存大小为 $AX * 1024$ 字节 + 1MB

图 42

测试代码如下 (注意由于是 BIOS 中断, 只能在实模式下运行, 主要修改了 loader.S)

```

1 %include "boot.inc"
2 section loader vstart=LOADER_BASE_ADDR ;预计加载到0x900地址
3 LOADER_STACK_TOP equ LOADER_BASE_ADDR
4     ;将loader加载到的地址0x900下面的地址作为栈
5 ;构建gdt及其内部的描述符
6 GDT_BASE:
7 dd 0x00000000 ;第一个描述符没用
8 dd 0x00000000
9
10 CODE_DESC:

```

```

11 dd    0x0000FFFF ;代码段描述符
12 dd    DESC_CODE_HIGH4
13
14 DATA_STACK_DESC:
15 dd    0x0000FFFF ;栈段和数据段的描述符
16 dd    DESC_DATA_HIGH4
17
18 VIDEO_DESC:
19 dd    0x80000007      ;limit=(0xffff-0xb8000)/4k=0x7, 低16位的段界限是0x7
20 dd    DESC_VIDEO_HIGH4
21
22 GDT_SIZE equ  $ - GDT_BASE ;得到GDT表的长度
23 GDT_LIMIT equ  GDT_SIZE - 1 ;得到偏移长度
24 times 60 dq 0           ;此处预留60个描述符的
25
26 ;下面构建相应的段选择子
27 SELECTOR_CODE equ (0x0001<<3) + TI_GDT + RPL0 ;
28 SELECTOR_DATA equ (0x0002<<3) + TI_GDT + RPL0 ;
29 SELECTOR_VIDEO equ (0x0003<<3) + TI_GDT + RPL0 ;
30
31
32 total_mem_bytes dd 0
   ↪ ;存储获取到的内存容量, 它的物理地址会是0x900+0x200=0xb00
33
34 ;以下是定义gdt的指针, 前2字节是gdt界限, 后4字节是gdt起始地址
35 gdt_ptr dw GDT_LIMIT
36 dd GDT_BASE
37
38 ards_buf times 244 db 0
   ↪ ;用于存储所有返回的ards数据, 这里设置244只是为了让loader_start偏移地址对齐到0x300
39 ards_nr dw 0          ;用于记录ards结构体数量
40
41 loader_start: ;loader的代码
42 ;下面是在实模式下获取内存容量
43 ;-----1.利用0x15子功能号eax = 0000e820获取内存容量-----

```

```

44 xor ebx, ebx          ;第一次调用时，ebx值要为0
45 mov edx, 0x534d4150  ;edx只赋值一次，是"SMAP"，循环体中不会改变
46 mov di, ards_buf     ;指定结果写到前面定义的ards_buf缓冲区中
47 .e820_mem_get_loop:   ;循环获取每个ARDS内存范围描述结构
48 mov eax, 0x0000e820  ;执行int
    ↪ 0x15后，eax值变为0x534d4150，所以每次执行中断前都要更新为子功能号。
49 mov ecx, 20           ;ARDS地址范围描述符结构大小是20字节
50 int 0x15              ;执行0x15中断
51 jc .e820_failed_so_try_e801 ;若cf位为1则有错误发生，尝试第二种方法
52 add di, cx            ;使di增加20字节指向缓冲区中新的ARDS结构位置
53 inc word [ards_nr]    ;让ards_nr也就是ards数据+1
54 cmp ebx, 0
    ↪ ;若ebx为0且cf不为1，这说明ards全部返回，当前已是最后一个，不再循环
55 jnz .e820_mem_get_loop
56
57 ;在所有ards结构中，找出(base_add_low + length_low)的最大值，即内存的容量。
58 mov cx, [ards_nr]      ;遍历每一个ARDS结构体，循环次数是ARDS的数量
59 mov ebx, ards_buf
60 xor edx, edx          ;edx为最大的内存容量，在此先清0
61 .find_max_mem_area:   ;无须判断type是否为1，最大的内存块一定是可被使用
62 mov eax, [ebx]         ;base_add_low
63 add eax, [ebx+8]       ;length_low
64 add ebx, 20            ;指向缓冲区中下一个ARDS结构
65 cmp edx, eax          ;比较取max
66 jge .next_ards
67 mov edx, eax
68 .next_ards: ;遍历每一个ards
69 loop .find_max_mem_area
70 jmp .mem_get_ok ;遍历完所有的ards后就存储内存容量值到前面定义的变量中
71
72
73 ;-----2.利用0x15子功能号ax = e801获取内存容量-----
74 ; 返回后，ax cx 值一样，以KB为单位，bx dx值一样，以64KB为单位
75 ; 在ax和cx寄存器中为低15M，在bx和dx寄存器中为16MB到4G。
76 .e820_failed_so_try_e801:

```

```

77 mov ax,0xe801
78 int 0x15
79 jc .e801_failed_so_try88 ;若当前e801方法失败,就尝试0x88方法
80
81 ;算出低15M的内存, 注意ax和cx中是以KB为单位的内存数量,将其转换为以B为单位
82 mov cx,0x400      ;cx和ax值一样,cx用做乘数
83 mul cx
84 shl edx,16
85 and eax,0x0000FFFF
86 or edx,eax
87 add edx, 0x100000 ;由于存在内存空洞,故要加1MB
88 mov esi,edx      ;先把低15MB的内存容量存入esi寄存器备份
89
90 ;算出16MB以上的内存, bx和dx中是以64KB为单位, 将其转换为以B为单位
91 xor eax,eax
92 mov ax,bx
93 mov ecx, 0x10000 ;0x10000十进制为64KB
94 mul ecx
    → ;32位乘法,默认的被乘数是eax,积为64位,高32位存入edx,低32位存入eax.
95 add esi,eax
    → ;由于此方法只能测出4G以内的内存,故32位eax足够了,edx肯定为0,只加eax便可
96 mov edx,esi      ;edx为总内存大小
97 jmp .mem_get_ok
98
99
100
101 ;-----3.利用0x15子功能号ah = 0x88 获取内存容量-----
102 .e801_failed_so_try88:
103 ;x存入的是以KB为单位的内存容量
104 mov ah, 0x88
105 int 0x15
106 jc .error_hlt ;这种方法也不行就停止
107 and eax,0x0000FFFF
108
109 ;16位乘法, 被乘数是ax,积为32位.积的高16位在dx中, 积的低16位在ax中

```

```

110 mov cx, 0x400 ;0x400等于1024,将ax中的内存容量换为以byte为单位
111 mul cx
112 shl edx, 16 ;把dx移到高16位
113 or edx, eax ;把积的低16位组合到edx,获得32位的积
114 add edx, 0x100000 ;0x88子功能只会返回1MB以上的内存,故实际内存大小要加上1MB
115
116
117 .mem_get_ok:
118 mov [total_mem_bytes], edx
    ↪ ;将内存换为B单位后存入我们上面定义的total_mem_bytes处。
119
120
121
122
123 ;-----准备进入保护模式
    ↪ -----
124 ;1 打开A20
125 ;2 加载gdt
126 ;3 将cr0的pe位置1
127
128 ;1.打开A20地址线
129 in al,0x92
130 or al,0000_0010B
131 out 0x92,al
132
133 ;2.加载GDT
134 lgdt [gdt_ptr]
135
136
137 ;3.cr0第0位置1, 即PE 位, Protection Enable
138 mov eax, cr0
139 or eax, 0x00000001
140 mov cr0, eax
141
142 ;jmp dword SELECTOR_CODE:p_mode_start ;

```

```

    ↳ 刷新流水线，避免分支预测的影响，这种cpu优化策略，最怕jmp跳转，

143 jmp dword SELECTOR_CODE:p_mode_start
    ↳ ;由于下面是32位操作码，为了避免分支预测的影响，使用jmp无条件跳转，从而刷新流水线

144
145 .error_hlt:          ;出错则挂起
146 hlt    ;出错则让处理器进入halt停止状态
147
148 [bits 32]
149 p_mode_start: ;用选择子初始化各段寄存器
150 mov ax, SELECTOR_DATA
151 mov ds, ax
152 mov es, ax
153 mov ss, ax
154 mov esp,LOADER_STACK_TOP
155 mov ax, SELECTOR_VIDEO
156 mov gs, ax
157
158 mov byte [gs:320], 'P'
159
160 jmp $

```

Listing 5: loader.S

运行结果如下，显示的内存容量是 32MB，和预设的一样：

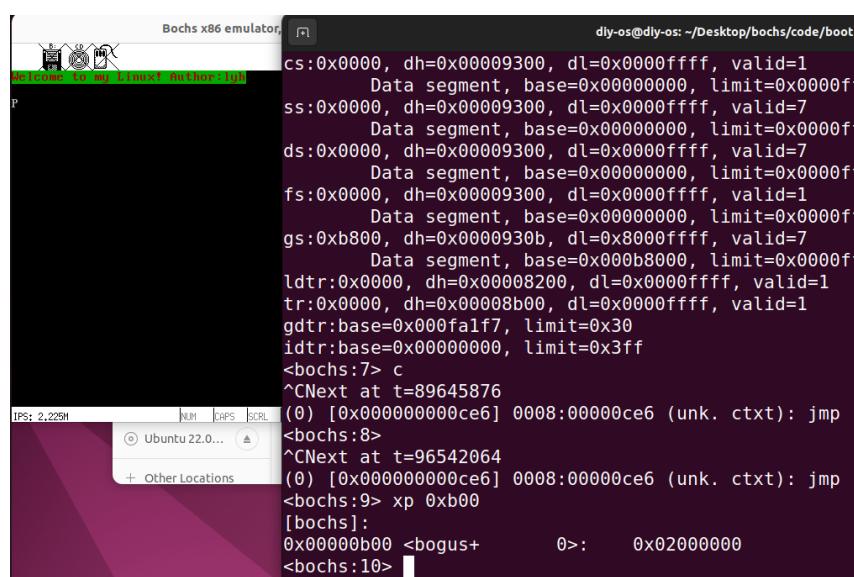


图 43

5.2 内存分页机制-由段式管理到页式管理

启用分页机制，我们要按顺序做好三件事。

- (1) 准备好页目录表及页表。
- (2) 将页表地址写入控制寄存器 cr3。
- (3) 寄存器 cr0 的 PG 位置 1。

5.2.1 页目录表及页表

为了解决内存碎片、内存利用效率和内存不足的问题，引入分页机制，也就是线性地址成为虚拟地址，经过映射后才成为物理地址，使得逻辑上连续的线性地址其对应的物理地址可以不连续。

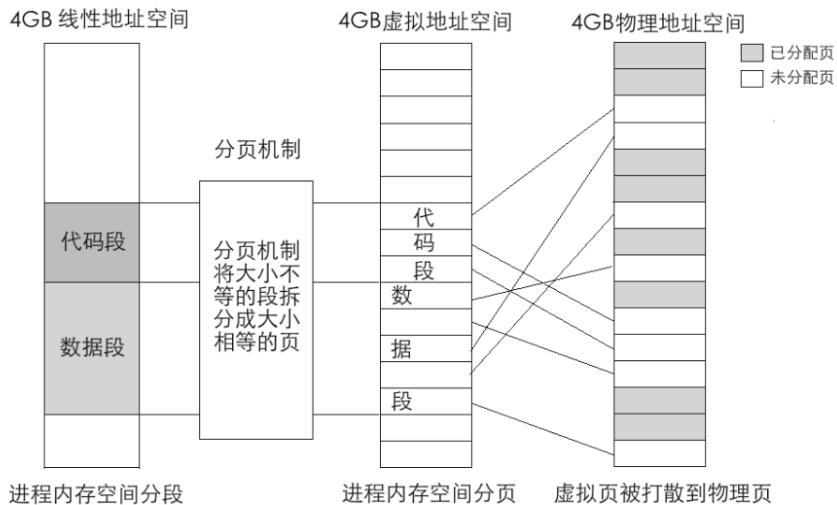


图 44

标准的页是 4KB，二级页表中，页目录表和页表本身也是一个页，页目录表项和页表项是 4B，其结构如下(标准页大小是 4KB，故地址都是 4K 的倍数，也就是地址的低 12 位是 0，所以只需要记录物理地址高 20 位就可以):



图 45

虚拟地址的转换过程如下(注意页目录表项和页表项是 4B，所以地址要乘以 4，但是表项索引的话不用):

(1) 用虚拟地址的高 10 位乘以 4，作为页目录表内的偏移地址，加上页目录表的物理地址，所得的和，便是页目录项的物理地址。读取该页目录项，从中获取到页表的物理地址。

(2) 用虚拟地址的中间 10 位乘以 4，作为页表内的偏移地址，加上在第 1 步中得到的页表物理地址，所得的和，便是页表项的物理地址。读取该页表项，从中获取到分配的物理页地址。

(3) 虚拟地址的高 10 位和中间 10 位分别是 PDE 和 PTE 的索引值，所以它们需要乘以 4。但低 12 位不是索引值，其表示的范围是 0~0xffff，作为页内偏移最合适，所以虚拟地址的低 12 位加上第 2 步中得到的物理页地址，所得的和便是最终转换的物理地址。

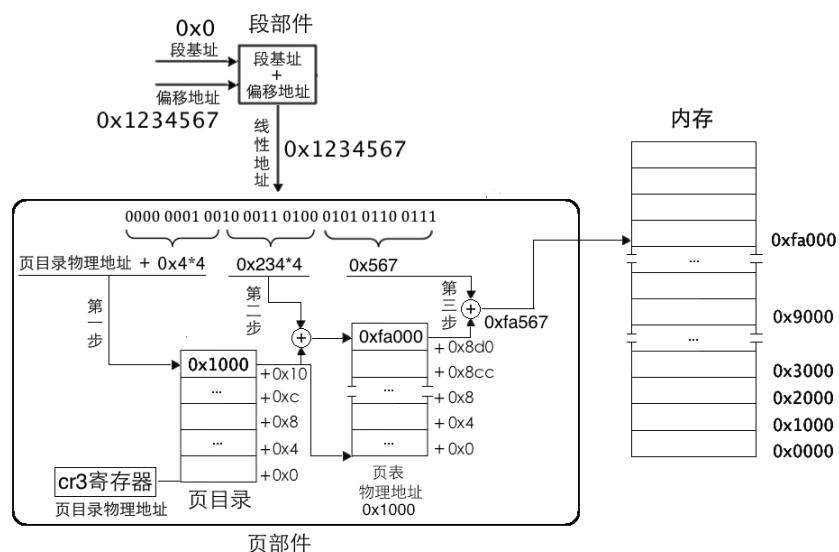
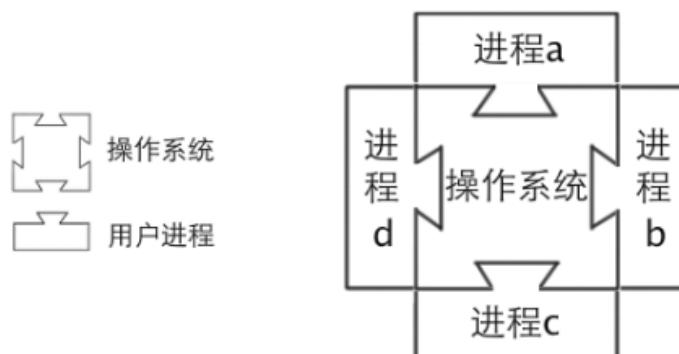


图 46

5.2.2 用户进程共享操作系统



- 1 用户进程共享操作系统
- 2 用户进程与操作系统配合“在一起”才能完成工作

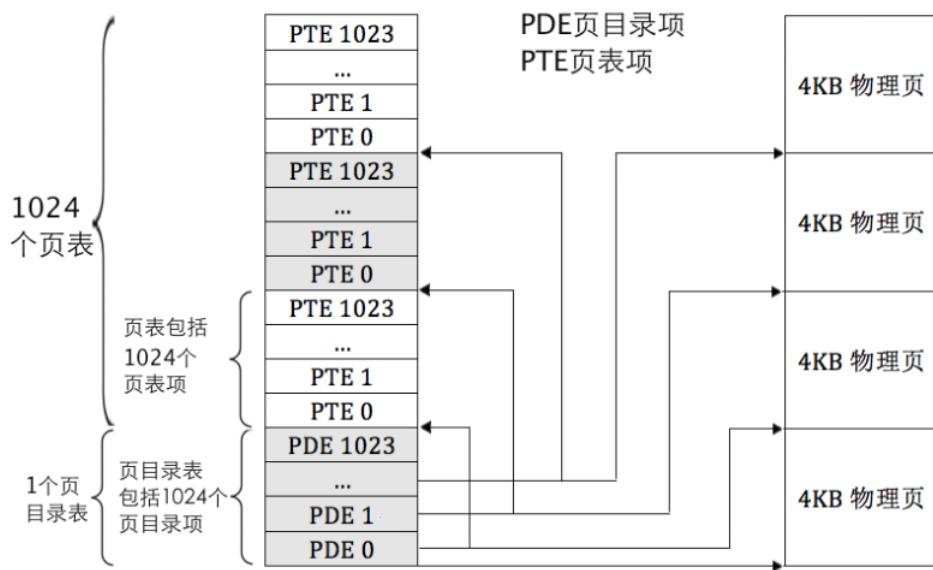
图 47

用户完成某个功能，需要操作系统配合，不同用户之间共享操作系统。

为了实现共享操作系统，让所有用户进程 3GB~4GB 的虚拟地址空间都指向同一个操作系统，也就是所有进程的虚拟地址 3GB~4GB 本质上都是指向的同一片物理页地址，这片物理页上是操作系统的实体代码。

5.2.3 启动分页机制

页目录表和页表的关系如下：



页目录表与页表的关系

图 48

页目录表及页表布局

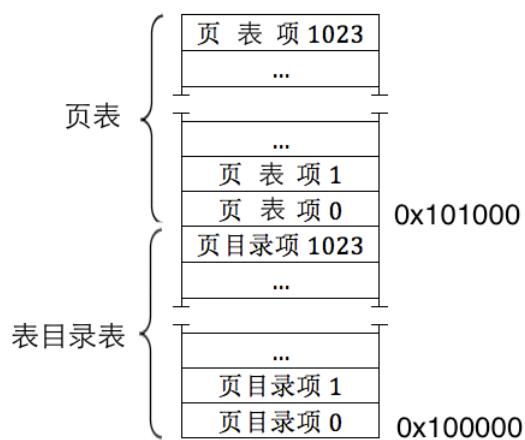


图 49

这里分页会将 0~0xfffff 和 0xc0000000~0xc00ffff 的地址都映射到内核地址 0~0xfffff。

同时还会将 gdt、显存段、栈指针也虚拟到内核地址中。

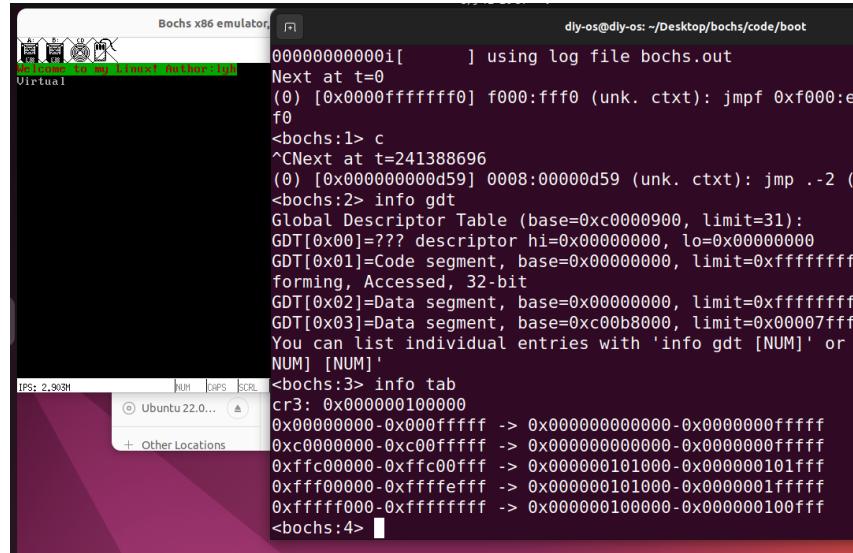


图 50

由于我们将最后一个页目录项存储的地址设置页目录表本身地址，因此可以用虚拟地址访问到页表自身(把页表本身的地址当作虚拟地址访问，类似递归的处理方法)。

- 获取页目录表物理地址：让虚拟地址的高 20 位为 0xfffff，低 12 位为 0x000，即 0xfffff000，这也是页目录表中第 0 个页目录项自身的物理地址。
- 访问页目录中的页目录项，即获取页表物理地址：要使虚拟地址为 0xfffffxxx，其中 xxx 是页目录项的索引乘以 4 的积。
- 访问页表中的页表项：要使虚拟地址高 10 位为 0x3ff，中间 10 位为页表的索引，低 12 位为页表内的偏移地址，用来定位页表项，它必须是已经乘以 4 后的值。

为了加速虚拟地址的转换，专门用来存放虚拟地址页框与物理地址页框的映射关系的 TLB 快表。

虚拟地址的高 20 位 (虚拟页框号)	属性	物理地址的高 20 位 (物理页框号)
...		
...		

图 51

更新 TLB 有两种方法，一个是针对 TLB 中所有条目的方法—重新加载 CR3，比如将 CR3 寄存器的数据读出来后再写入 CR3，这会使整个 TLB 失效。另一个方法是针对

TLB 中某个条目的更新。处理器提供了指令 invlpg (invalidate page)，它用于在 TLB 中刷新某个虚拟地址对应的条目，如 invlpg [0x1234]，。

5.3 加载内核

5.3.1 ELF 文件结构

为了让程序可以运行，需要在程序中表明入口地址 (加载的地址)，以及其他信息，这些信息就放在程序头部，其余代码和数据部分则形成了程序体，形成了一种文件头 header+ 文件体 body 的文件格式。

在不同操作系统下的这种文件格式不同，在 Linux 操作系统中生成的是 elf(Executable and Linkable Format) 文件格式。

ELF header 从“全局上”给出程序文件的组织结构，如给出程序头表的大小及位置和节头表的大小及位置。而程序头表和节头表再给出各个段和节的位置、大小等信息。

链接视图	运行视图
ELF header (elf 头)	ELF header (elf 头)
Program header table (程序头表) 可选	Program header table (程序头表)
Section 1 (节 1)	Segment 1 (段 1)
...	
...	
Section n (节 n)	Segment 2 (段 2)
...	...
Section header table (节头表)	Section header table (节头表) 可选
...	...
待重定位文件体	可执行文件体

图 52

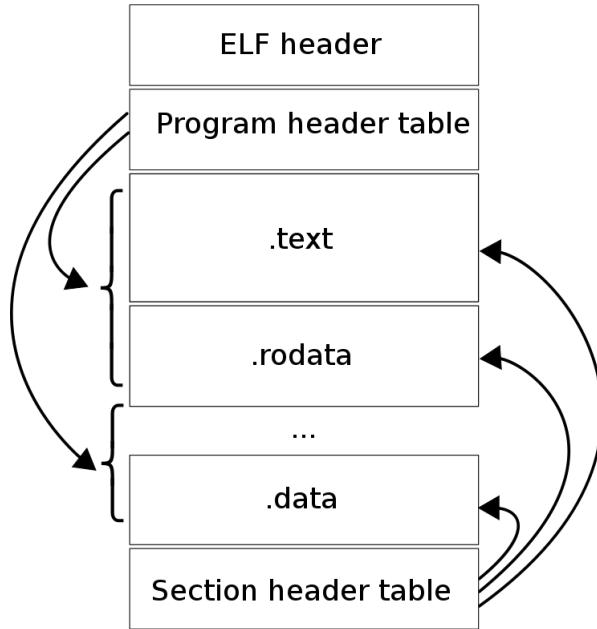


图 53

elf header 的结构如下：

```
struct Elf32_Ehdr {
    unsigned char e_ident[16];
    Elf32_Half   e_type;
    Elf32_Half   e_machine;
    Elf32_Word   e_version;
    Elf32_Addr   e_entry;
    Elf32_Off    e_phoff;
    Elf32_Off    e_shoff;
    Elf32_Word   e_flags;
    Elf32_Half   e_ehsize;
    Elf32_Half   e_phentsize;
    Elf32_Half   e_phnum;
    Elf32_Half   e_shentsize;
    Elf32_Half   e_shnum;
    Elf32_Half   e_shstrndx;
};
```

图 54

program header talbe 中的 program header 的结构如下：

```
struct Elf32_Phdr {
    Elf32_Word   p_type;
    Elf32_Off    p_offset;
    Elf32_Addr   p_vaddr;
    Elf32_Addr   p_paddr;
    Elf32_Word   p_filesz;
    Elf32_Word   p_memsz;
    Elf32_Word   p_flags;
    Elf32_Word   p_align;
};
```

图 55

5.3.2 载入内存

根据我们的硬盘分布：

1. 第 0 扇区放置 MBR。
2. 第 24 扇区放置 loader
3. 第 9 扇区开始放置 kernel(预留给 loader 一定空间)

由于这里我们的内核是由 C 语言编写的，通过编译形成二进制程序，属于 elf 可执行文件，因此这里我们将内核载入内存不仅需要将 kernel 加载到内存，还需要对 elf 格式的内核进一步解析，加载其中的程序体，也就是将内核文件中的段（segment）展开到（复制到）内存中的相应位置。

将内核 kernel.bin 加载到 0x70000，由于编译时指定了入口虚拟地址 0xc0001500，kernel.bin 会被解析到物理地址 0x1500 处。

参考文献

[1] 《操作系统真相还原》