

# 90分钟学会写操作系统

瓦利哥 2014/10/7



# 目标群体

- 已学课程：

- 汇编语言、C语言、操作系统概念、计算机组成、编译原理。



# 大纲

- ① 1 操作系统是如何启动的
- ② 2 从实模式到保护模式
- ③ 3 硬件中断与系统调用
- ④ 4 多任务的实现原理
- ⑤ 5 程序的加载与执行
- ⑥ 6 总结



# 思考

- 什么是操作系统 (Operating System) ?



# 思考

- BIOS (Basic Input Output System) 是不是一个操作系统?



# 重新思考BIOS

- 从名字可知，它也是一个软件系统，甚至可以说是一个简单的操作系统。
- 负责检测内存等硬件的状态，并加载Boot扇区。
- 提供一整套的硬件操作接口（通过int中断指令，先简单理解为函数），方便Boot程序调用。



# 操作系统的启动过程

- 1, 按下电源
- 2, 运行BIOS (Basic Input Output System)
- 3, 加电自检POST (Power On Self Test)
  - 检查硬件是否正常
- 4, 加载Boot Loader引导程序
- 5, 引导操作系统

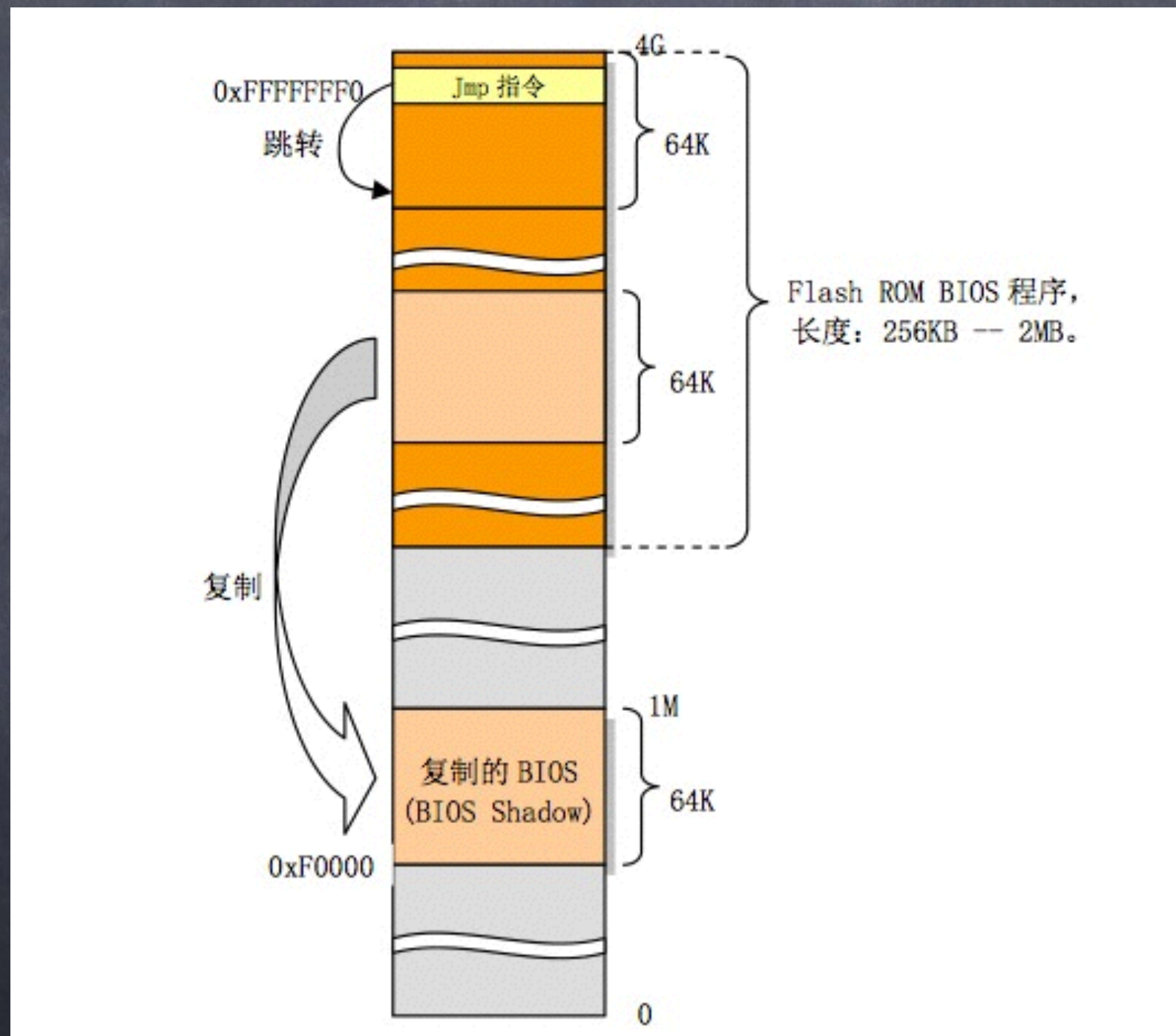


# 启动BIOS

- 开机后，BIOS所在的ROM可以被CPU直接访问，执行一段硬件检测指令后，会复制自身到0xF0000-0xFFFFF内存区域去，即1M内存的最后64K，并跳转过去继续执行。



# BIOS 内存映射图





# 加载boot扇区

- 依次检测硬件存储器的第0个扇区的第510和511字节的值，如果分别是0x55，0xAA，则说明是个有效的Boot扇区。
- 将其复制到内存地址0x7C00处，并将CS设置为0，且IP设置为0x7C00，然后跳转到此地址开始执行，即运行Boot扇区的代码。



# Boot的功能

- Boot程序会通过BIOS提供的存储器读取中断指令，读取更多的内容到内存中，并从实模式到保护模式（寻址范围从1M到4G），然后就可以加载进来真正的内核了。
- 内核初始化。



# BIOS提供的中断调用

- `int 0x10`; 用于屏幕显示
  - 寄存器 `AH = 0x0E` 时, 表示显示 `AL` 对应的字符。
  - 寄存器 `AH = 0x13` 时, 表示显示一个字符串。`ES: BP` 对应字符串的地址, `CX` 存放字符串的长度。 (`DH`、`DL`) = 坐标 (行、列)



# 实验1

- 在屏幕打印一个字符串Hello, world!



# 代码

```
1 BOOTSEG equ 0x07C0 ;宏定义，用于初始化ES
2
3 mov bp, HelloMsg ;将HelloMsg的地址赋值给bp
4 mov ax, BOOTSEG
5 mov es, ax        ;es:bp = 字符串地址
6                   ;HelloMsg只是段内的偏移地址
7                   ;而代码被加载到0x7C00，故设置段地址
8                   ;即 es * 16 + bp为实际内存地址
9 mov cx, 13 ;串的长度
10 mov ax, 0x1301 ;AH = 0x13，表示串打印
11                ;AL = 0x01，表示打印后光标移动
12 mov bx, 0x000C ;BH = 0x00，表示页码
13                ;BL = 0x0C，表示红色
14 mov dx, 0x0000 ;第0行0列开始
15 int 0x10
```



# 代码

```
17 loop1:
18     jmp loop1 ;jmp段内跳转，是相对当前地址
19             ;翻译成机器指令后，是0xeb 0xfe
20             ;0xeb表示jmp段内跳转的指令号
21             ;0xfe即-2，表示往前跳2个字节
22             ;这是因为执行本指令时，IP已经移到下条
23             ;指令开头了。
24             ;所以不会是jmp 0x00这样的写法。
25
26 HelloMsg:
27     db "Hello, world!" ;共13个字符
28
29 times 510 - ($ - $$) db 0 ;填充一堆的0
30                             ;$表示当前位置，$$表示文件头部
31 db 0x55
32 db 0xAA ;上面两行用于设置引导扇区的标识
```



# 生成软盘镜像

- 汇编:

- `nasm -f bin printhello.s -o  
printhello.bin`

- 生成镜像:

- `dd conv=sync if=printhello.bin  
of=helloboot.img bs=1440k count=1`

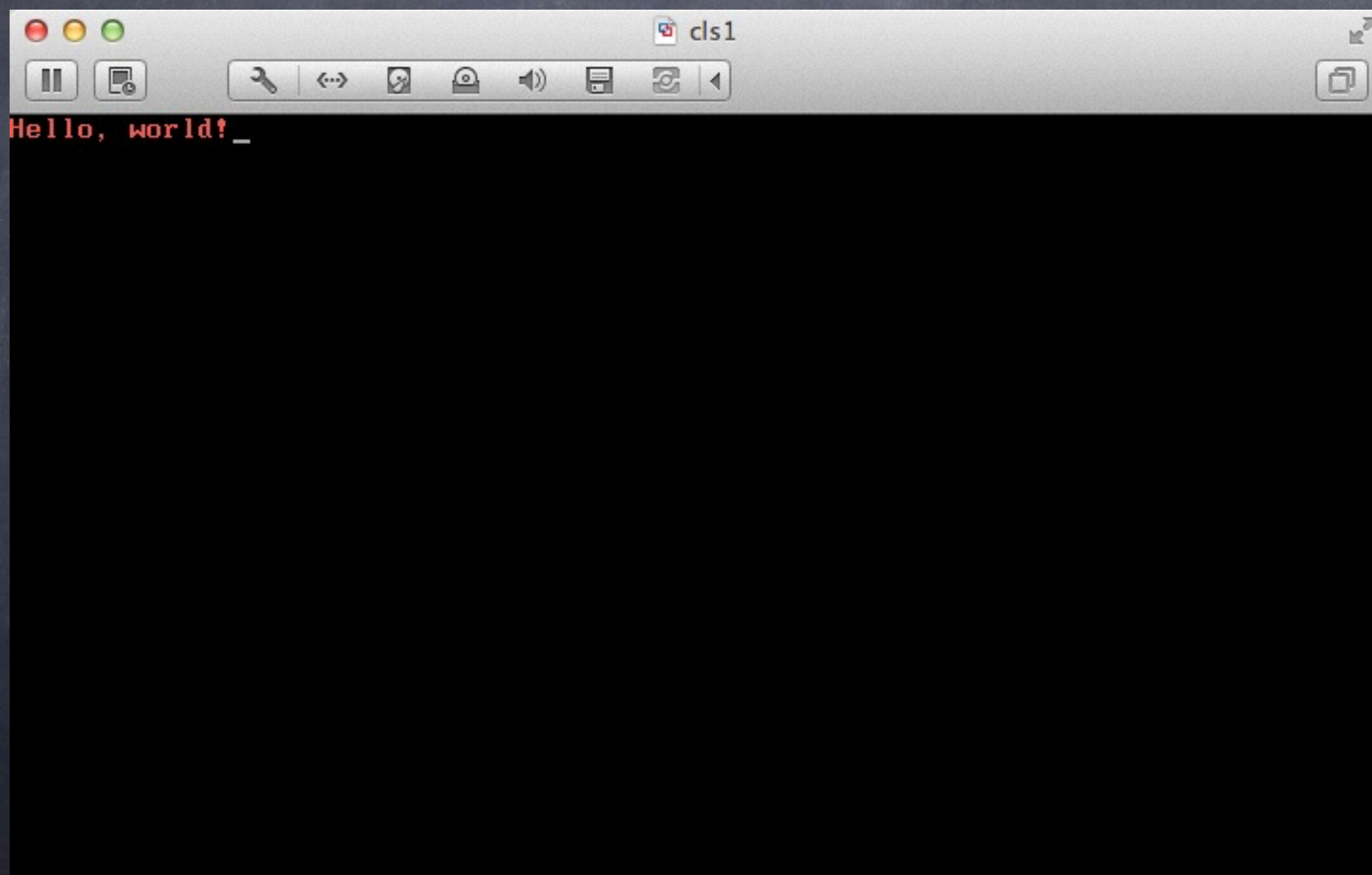


# 使用虚拟机VMWare运行

- 打开虚拟机资源库，添加一个新系统，类型选（其它、其它），并添加设备软盘。
- 添加软盘时，会让你选择软盘文件，就选择刚只制作的boot.img文件，然后启动运行。



# 运行结果



A screenshot of a Java IDE window titled "cls1". The window has a standard macOS-style title bar with red, yellow, and green window control buttons. Below the title bar is a toolbar with icons for running, debugging, and other IDE functions. The main area of the window is black, and the text "Hello, world!\_" is displayed in red, indicating the output of a Java program.

```
Hello, world!_
```



# 实验1总结

- 本质上，我们的实验是BIOS系统下写的一个应用。



# 大纲

- ① 1 操作系统是如何启动的
- ② 2 从实模式到保护模式
- ③ 3 硬件中断与系统调用
- ④ 4 多任务的实现原理
- ⑤ 5 程序的加载与执行
- ⑥ 6 总结



# 实模式寻址

- 地址换算：段选择符 \* 16 + 偏移地址
- 最多寻址20位，即1M访问空间



# 实模式的问题

- 能访问的内存太小，只有1M
- 缺乏保护，程序间内存没有隔离，不安全
  - 如：0x1001: 0x0010和0x1000: 0x0020访问的地址一样。



# 保护模式的解决之道

- 地址变为32位, EIP、EAX、ESP。。。。
- 复用段寄存器 (CS、DS、SS。。。)
- 但表达的含义变了, 成了一个段选择符 (索引)。
- 从哪里选择呢?

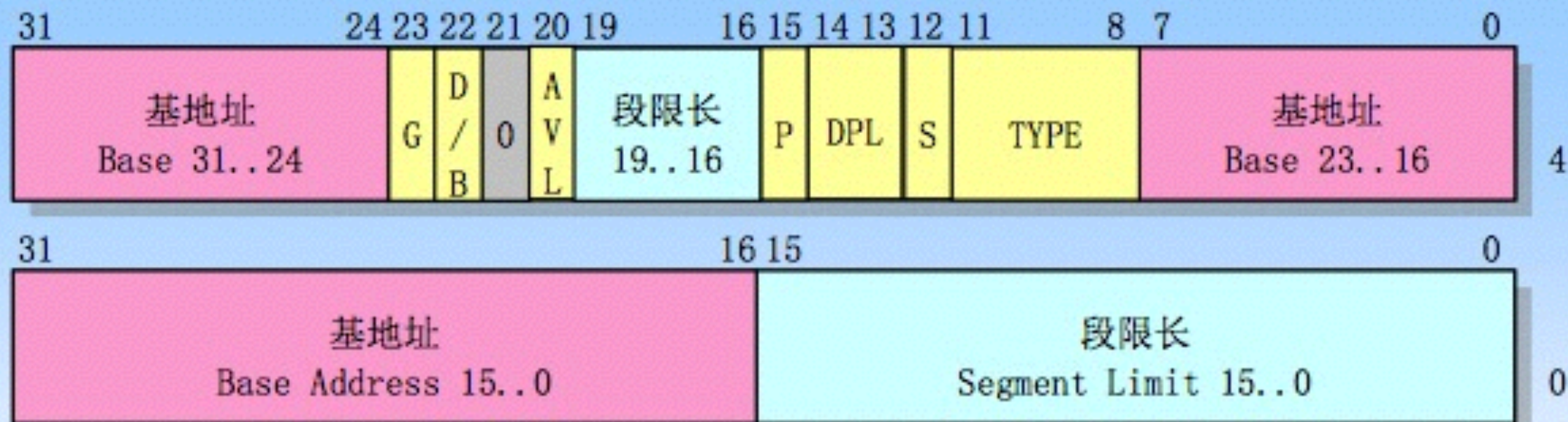


# GDT

- Global Descriptor Table: 全局描述符表
- 段的详细信息在GDT中记录, 包括: 段的起始地址, 限长, 访问权限控制等。
- 就是一个大数组, 每一条就是一个段描述符。



# 段描述符通用格式



AVL -- 系统软件可用位

BASE -- 段基地址

B/D -- 默认大小 (0-16 位; 1-32 位段)

DPL -- 描述符特权级

G -- 颗粒度

LIMIT -- 段限长

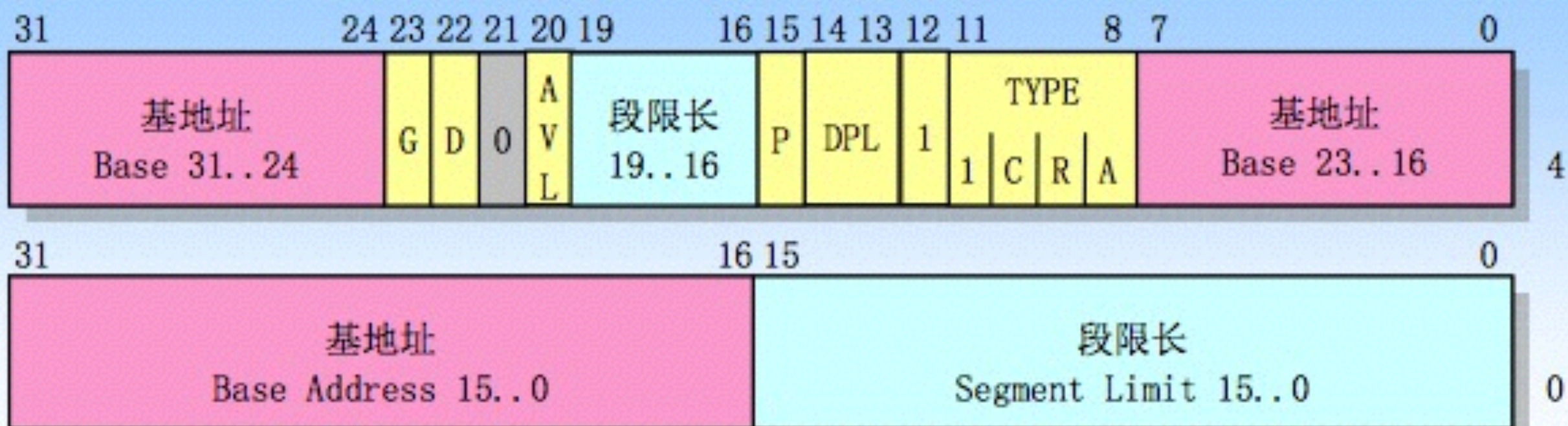
P -- 段存在

S -- 描述符类型 (0-系统; 1-代码或数据)

TYPE -- 段类型

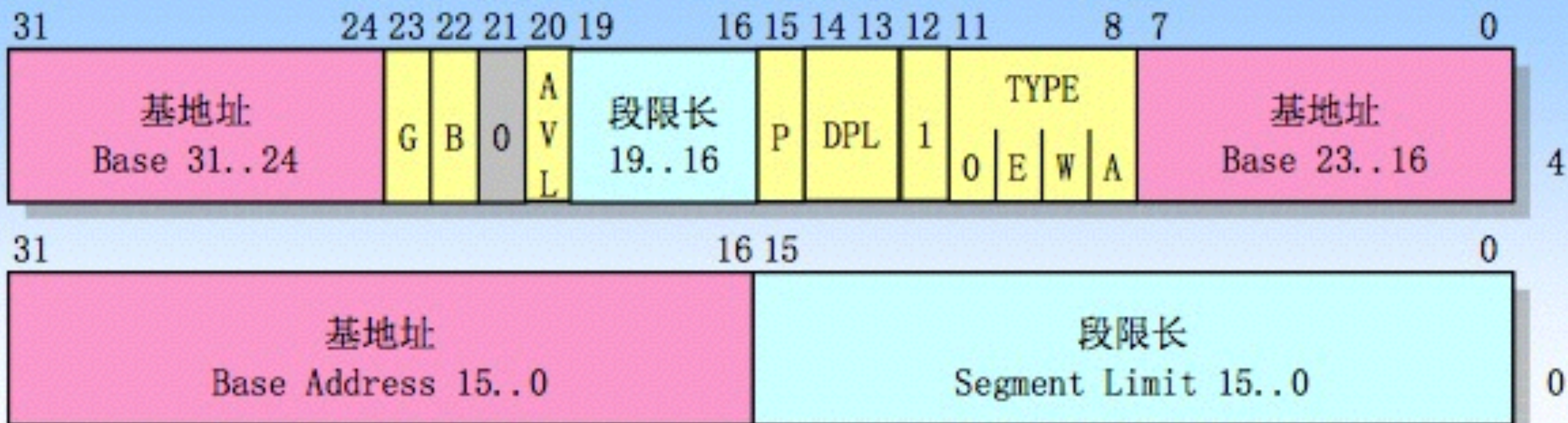


# 代码段





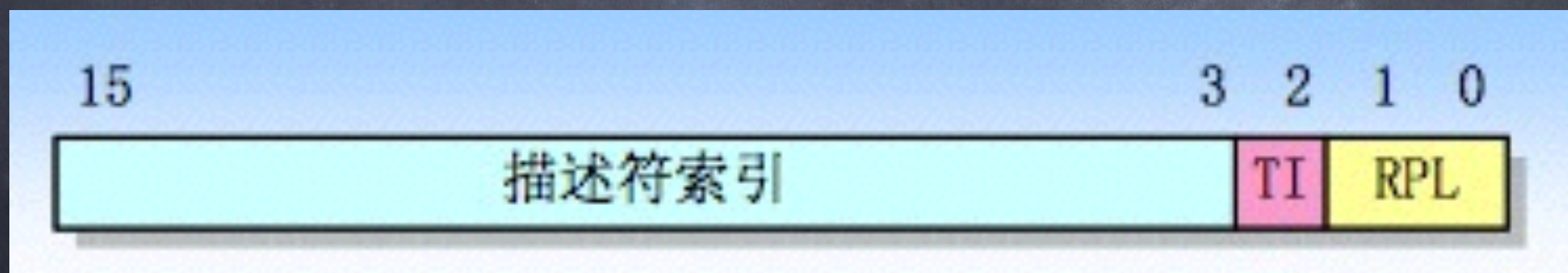
# 数据段





# 段选择符

- 共包括三个部分：
  - GDT/LDT 表索引号
  - TI: 0 表示 GDT, 1 表示 LDT (暂时不用)
  - RPL: 请求特权级



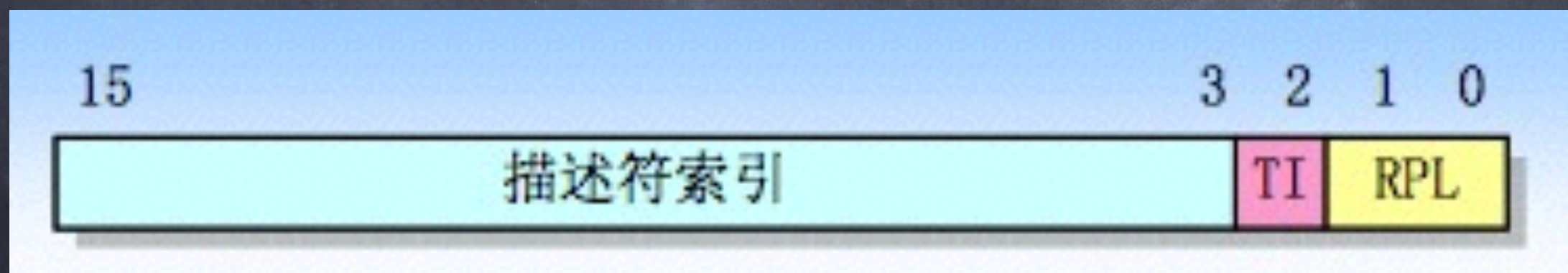


# 段选择符

• 例子:

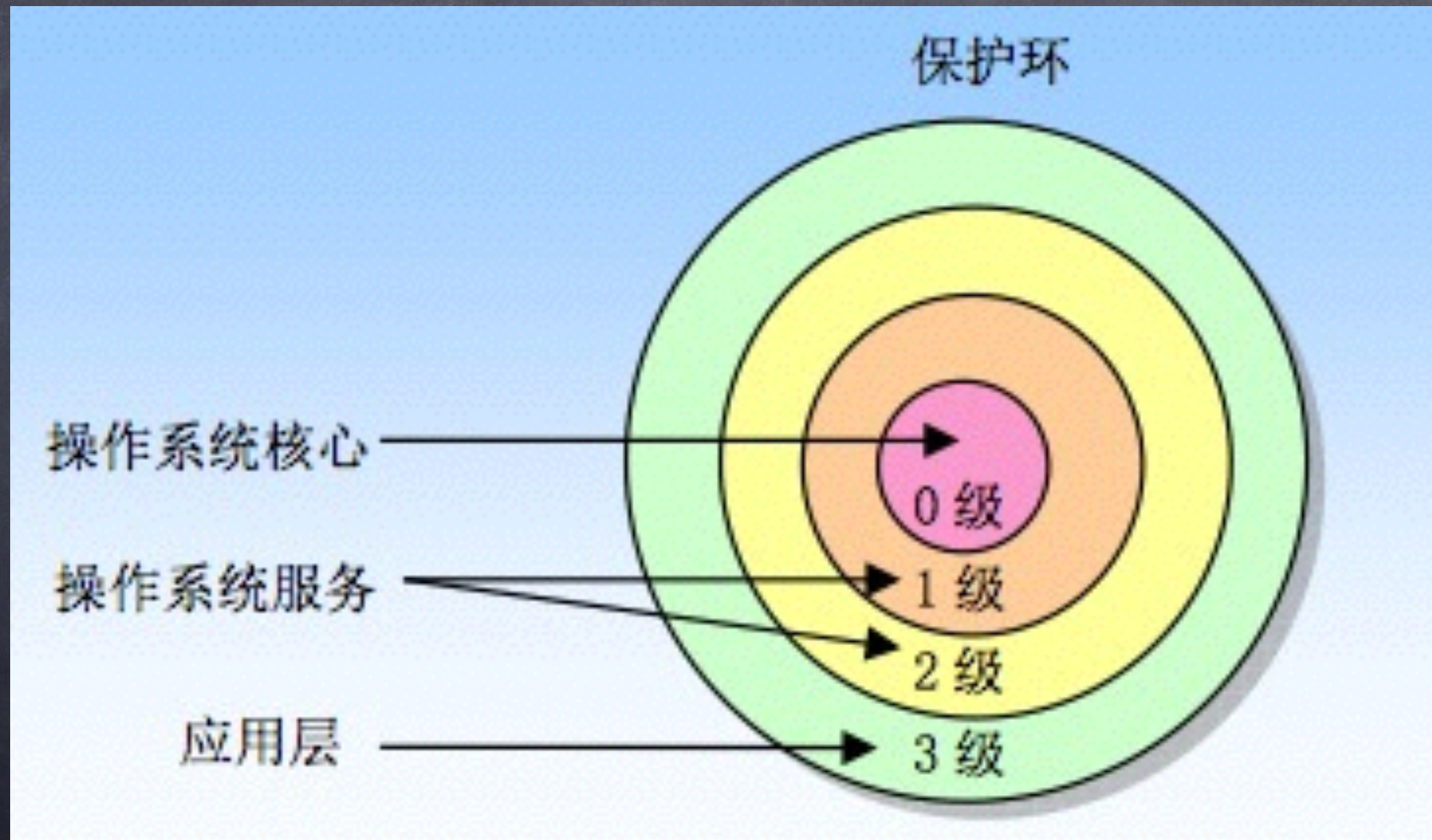
• 全局描述符表中的第1项: 00001 0 00 = 0x08

• 第2项: 00010 0 00 = 0x10





# 特权级





# 如何进入保护模式

- 1, 设置GDT描述符表, 通过LGDT加载表。
- 2, 将寄存器CR0的PE位设置为1, 进入保护模式。
- 3, 紧跟一条jmp指令, 使用一个段选择符, 跳转到保护模式下的有效代码段。
- 4, 进入保护模式。



# 实验2

- 保护模式下，在屏幕打印一个字符串Hello, world!



# 代码 (boot.s)

```
1 BOOTSEG equ 0x07C0 ;宏定义，用于初始化ES
2
3 ;使用bios中断0x13，功能号(AH = 2)进行软盘读取
4 mov dx, 0x0000 ;DH = 0，磁头号
5                ;DL = 0，驱动器号
6 mov cx, 0x0002 ;CH, 10位磁道号低8位
7                ;CL - 位7、6是磁道号高2位
8                ;CL - 位5~0表示扇区号（从1开始）
9                ;本指令表示读取0号驱动器0号磁道第2号扇区
10               ;第1扇区就是boot扇区
11 mov ax, 0x1000 ;设置拷贝的目的地段描述符为0x1000
12 mov es, ax
13 xor bx, bx     ;设置bx为0，[es:bx]表示目的地内存地址
14               ;即0x1000 * 16 + 0 = 0x10000
15 mov al, 1      ;读取的扇区数为1
16               ;注意：如果读取的内容超过512byte，则加大
17 mov ah, 0x02   ;AH = 2，表示读扇区
18 int 0x13
```



# 代码 (boot.s)

```
20 cli ;关闭中断
21 mov ax, 0x1000
22 mov ds, ax
23 xor ax, ax ;移动到内存0地址
24 mov es, ax
25 mov cx, 128 ;移动128个double word(4 bytes)
26 sub si, si
27 sub di, di
28 rep movsd
29
30 mov ax, BOOTSEG
31 mov ds, ax ;要访问内存数据，则需设置数据段
32 lidt [idt_48] ;加载中断表，请忽略，以后会讲
33 lgdt [gdt_48] ;加载GDT描述符
34
35
36 mov eax, cr0
37 or al, 1 ;将保护模式的标记PE设置为1
38 mov cr0, eax
```



# 代码 (boot.s)

```
40 ;这里加 dword是为了生成一个32位偏移地址的指令
41 ;这条指令执行的时候，已经是在保护模式了
42 ;是利用了CPU预取的机制，在执行 mov cr0, eax时，
43 ;就将其读入了指令寄存器
44 ;0x0008表示GDT中第一个描述符
45 ;0x00000000是跳转的32位偏移地址，有效的
46 jmp dword 0x0008: 0x00000000
```



# 代码段



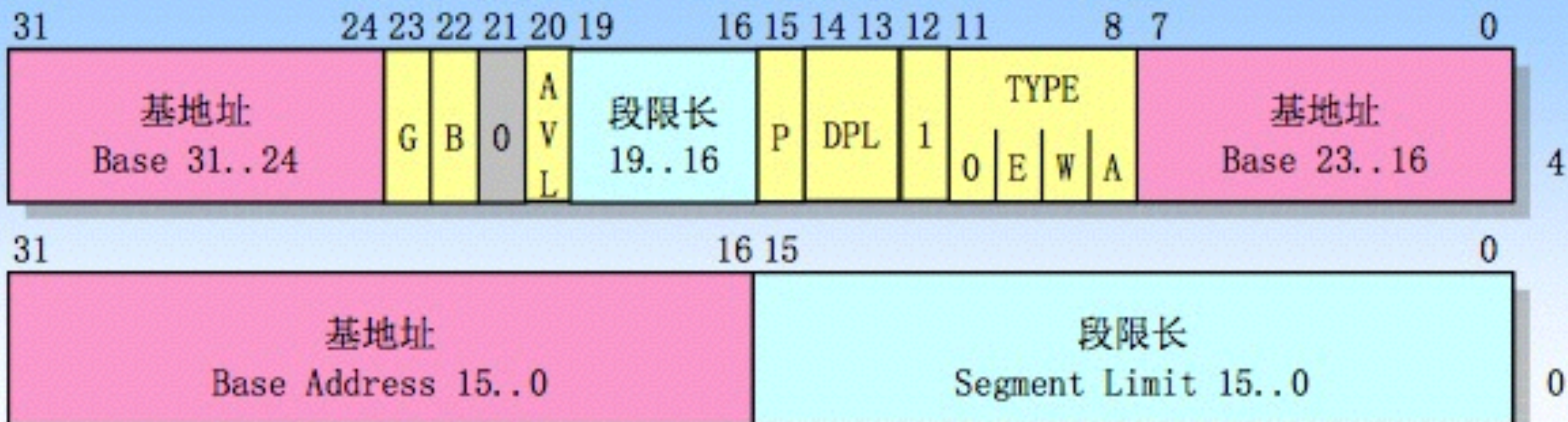


# 代码 (boot.s)

```
48 ;注意：以下GDT的设置一定要对照格式图表
49 gdt:
50     dw 0, 0, 0, 0 ;第0个描述符，不使用
51
52     ;第一个描述符(0x08)，表示代码段
53     dw 0x07FF ;段限长低16位，则段限长为：
54             ; (0x07FF + 1) * 4KB = 8MB
55     dw 0x0000 ;基地址低16位0，所以基地址为0
56     dw 0x9A00 ;0x9A = 1001 1010
57             ;高4位分别表示P = 1段在内存中
58             ;DPL = 0，最高权限
59             ;S = 1，非系统段
60             ;且TYPE的bit 3为1，表示代码段
61             ;低8位表示基地址23~16为0
62     dw 0x00C0 ;高8位表示基地址31~24为0
63             ;G标志(bit 7) = 1，表示颗粒度为4KB
64             ;注意：保护模式下颗粒度一般都设4KB
65             ;最后4位为0，表示段限长19~16位是0
```



# 数据段





# 代码 (boot.s)

```
67      ;第2个段描述符 (0x10) , 表示显卡内存
68      ;显卡默认在CGA模式, 字符彩屏
69      ;映射在内存地址0xb8000~0xc8000, 共16K
70      ;我们只用前面的4K
71      ;所以颗粒度G = 1, 且段限长Limit = 0
72      dw 0x0002
73      dw 0x8000
74      dw 0x920B ;0x92 = 1001 0010
75                ;S = 1且TYPE(bit 3)为0, 表示数据段
76                ;DPL = 0
77                ;E = 0, 表示地址是从小往大增长
78                ;W = 1表示可写入
79      dw 0x00C0
80
81 end_gdt:
82
83 idt_48:
84      dw 0
85      dw 0
86      dw 0
```



# 代码 (boot.s)

```
88 gdt_48:
89     dw (end_gdt - gdt) - 1 ;gdt的限长
90                                     ; - 1的作用和段描述符的Limit类似
91                                     ; 都是表示最后一个有效的地址 (<=)
92     dw BOOTSEG * 16 + gdt ;gdt的起始地址
93     dw 0
94
95 times 510 - ($ - $$) db 0 ;填充一堆的0
96                                     ; $表示当前位置, $$表示文件头部
97 db 0x55
98 db 0xAA ;上面两行用于设置引导扇区的标识
```



# 代码

(printhello\_pe.s)

```
1 KERNEL_SEL equ 0x08
2 SCREEN_SEL equ 0x10
3
4 bits 32 ;这是32位的指令
5 mov eax, SCREEN_SEL
6 mov ds, eax
7
8 mov eax, 0
9 mov byte [eax], 'H'
10 mov byte [eax + 1], 0x02 ;color
11 add eax, 2
12 mov byte [eax], 'e'
13 mov byte [eax + 1], 0x02 ;color
14 add eax, 2
15 mov byte [eax], 'l'
16 mov byte [eax + 1], 0x02 ;color
17 add eax, 2
18 mov byte [eax], 'l'
19 mov byte [eax + 1], 0x02 ;color
20 add eax, 2
```



# 代码

(printfkhello\_pe.s)

```
36 mov byte [eax], 'r'
37 mov byte [eax + 1], 0x02 ;color
38 add eax, 2
39 mov byte [eax], 'l'
40 mov byte [eax + 1], 0x02 ;color
41 add eax, 2
42 mov byte [eax], 'd'
43 mov byte [eax + 1], 0x02 ;color
44 add eax, 2
45 mov byte [eax], '!'
46 mov byte [eax + 1], 0x02 ;color
47
48 LOOP1:
49     jmp LOOP1
```



# 生成软盘镜像

- 汇编:

- `nasm -f bin boot.s -o boot.bin`

- `nasm -f bin printhello_pe.s -o printhello.bin`

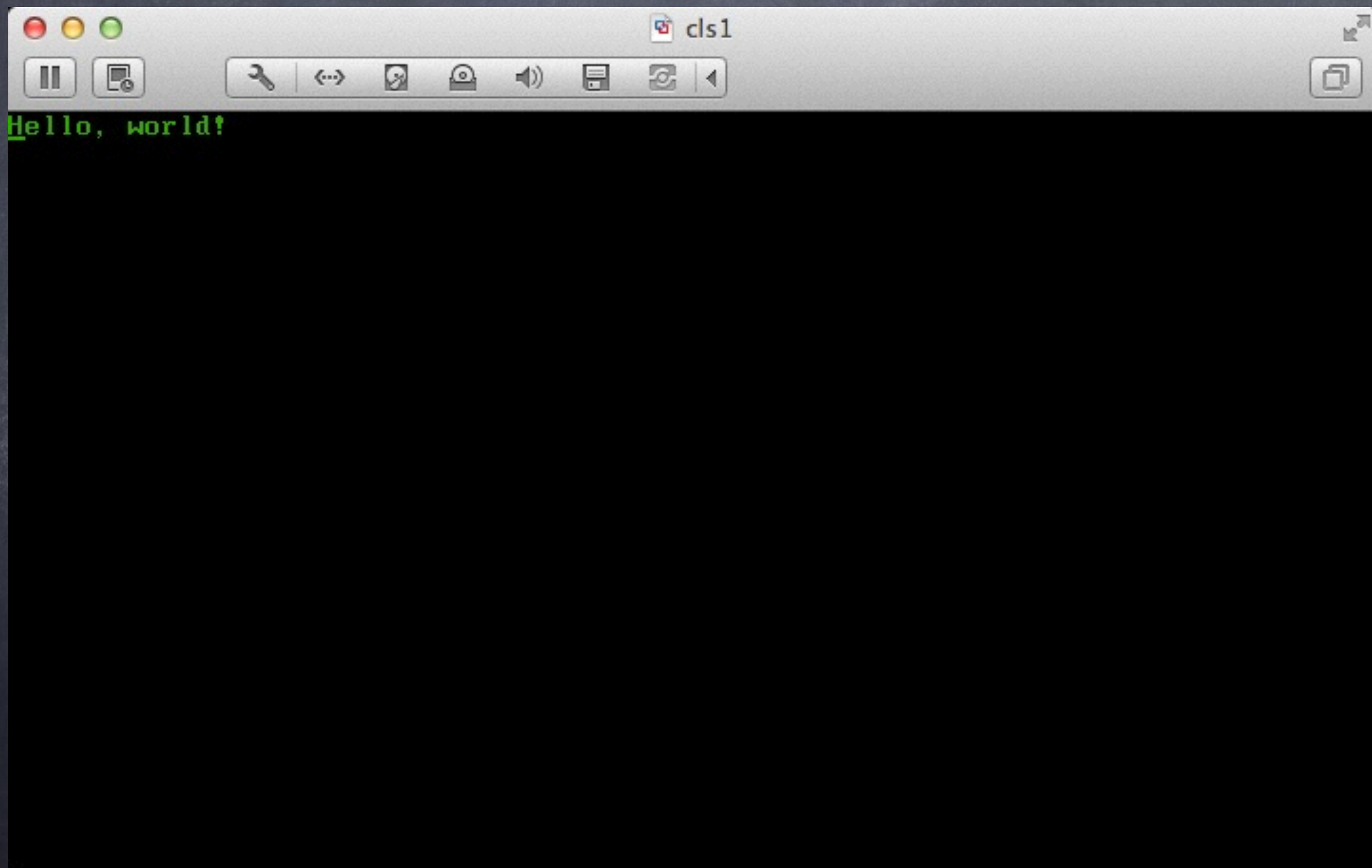
- 生成镜像:

- `cat boot.bin printhello.bin > system.bin`

- `dd conv=sync if=system.bin  
of=helloboot.img bs=1440k count=1`



# 运行结果



The image shows a screenshot of a Java IDE window titled "cls1". The window has a standard macOS-style title bar with red, yellow, and green window control buttons. Below the title bar is a toolbar with various icons for editing and development. The main area of the window is black, and the text "Hello, world!" is displayed in green, indicating the output of a Java program.

```
Hello, world!
```



# 分段的问题

- 段占用的物理内存连续，且前后无法扩展，就要整体移位。
- 内存不足时，需要将整个段置换到磁盘（回顾段描述符的P标记位），来回置换代价大。

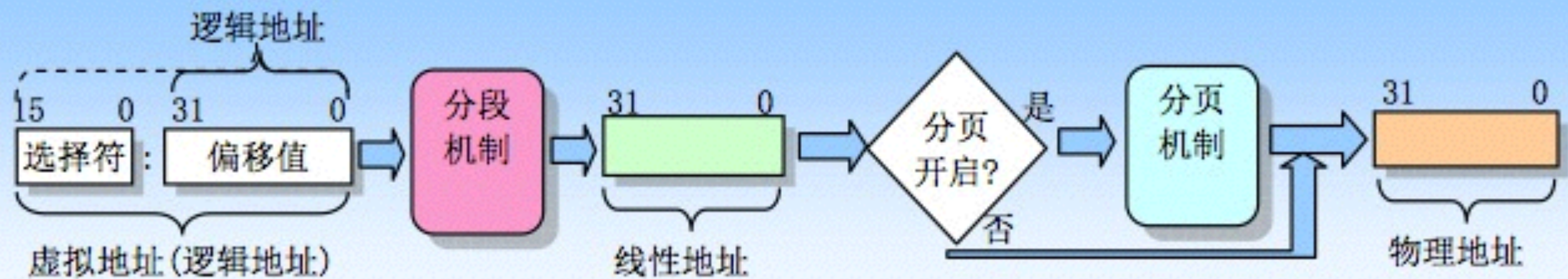


# 分页的解决之道

- 在保护模式下，可以开启分页，每页**4K**，可以按需分配。
- 好处：
  - 页面之间独立，不要求连续。
  - 实际是将内存当成了磁盘的缓存，按需加载。  
(传说中的虚拟内存)

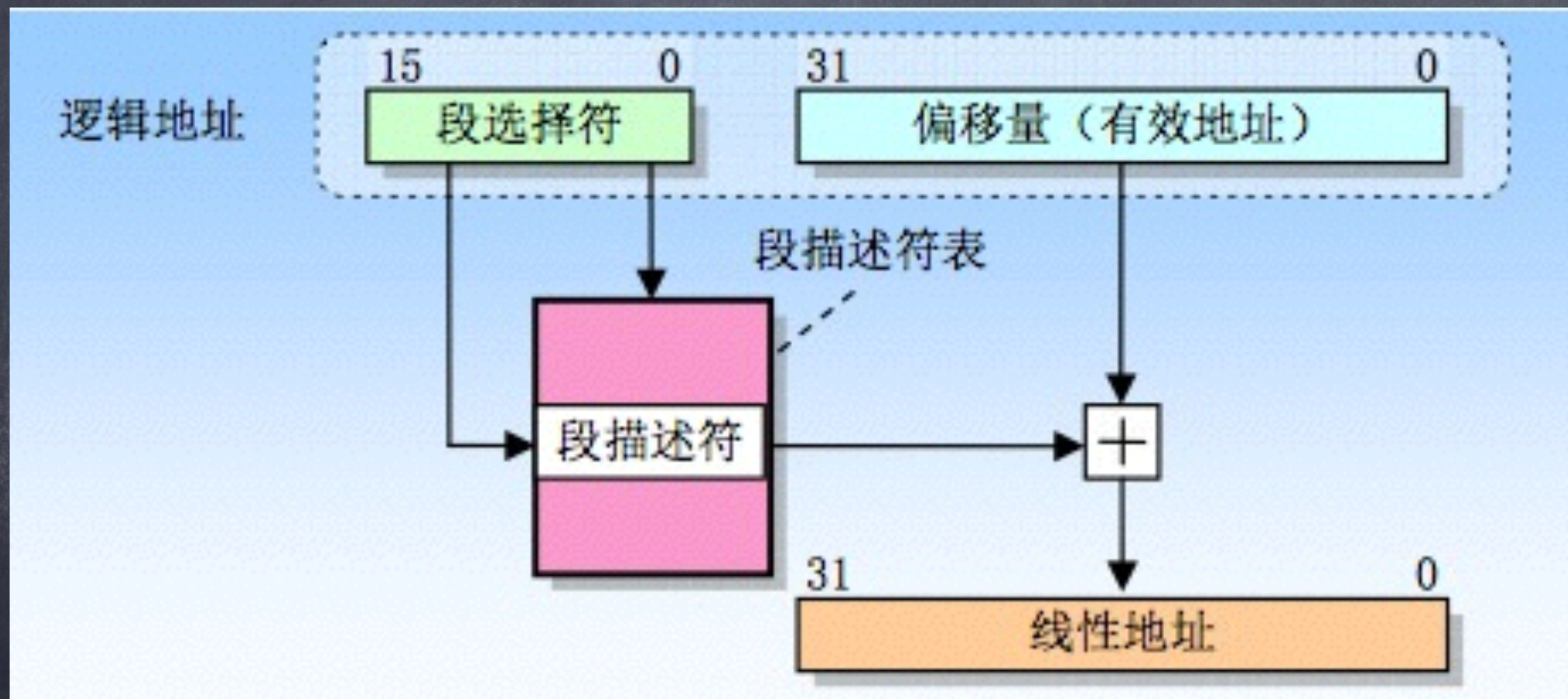


# 从虚拟地址到物理地址



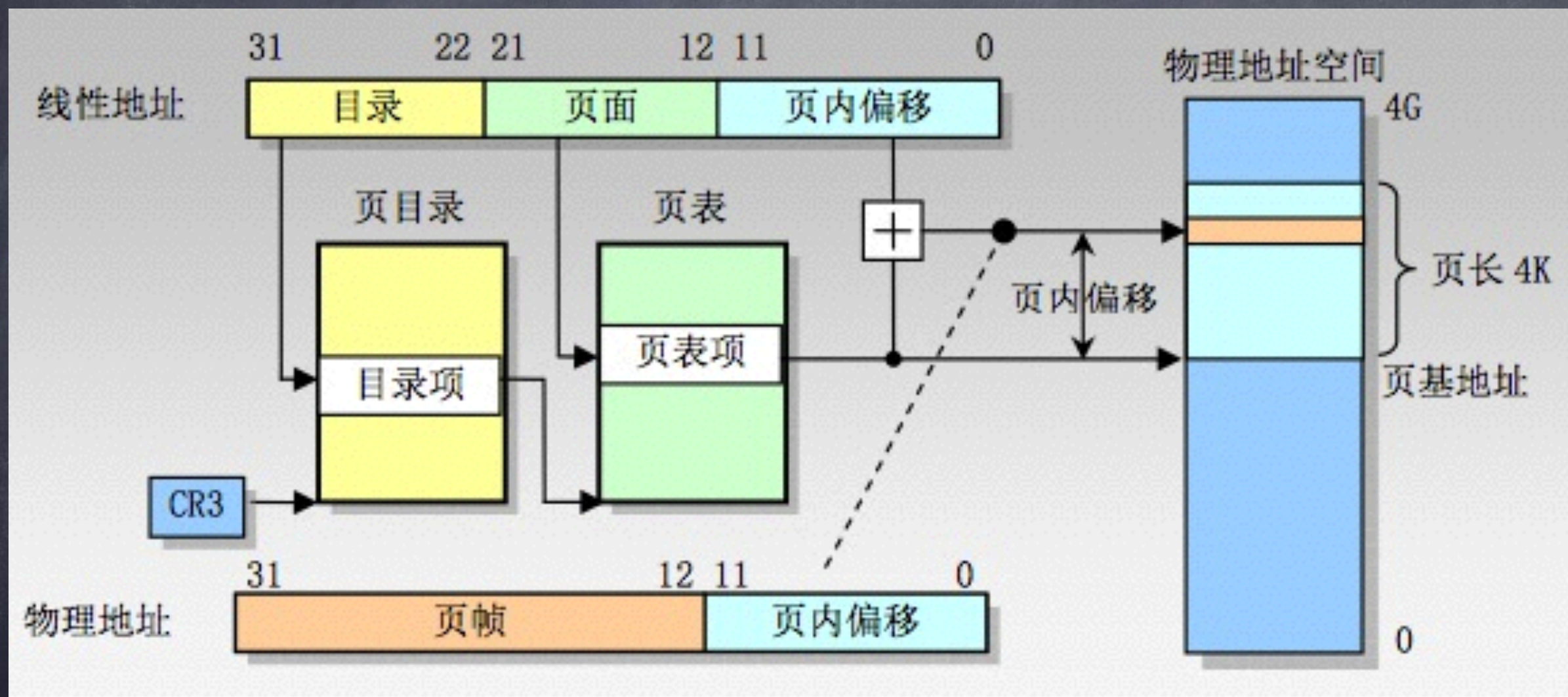


# 段地址的转换



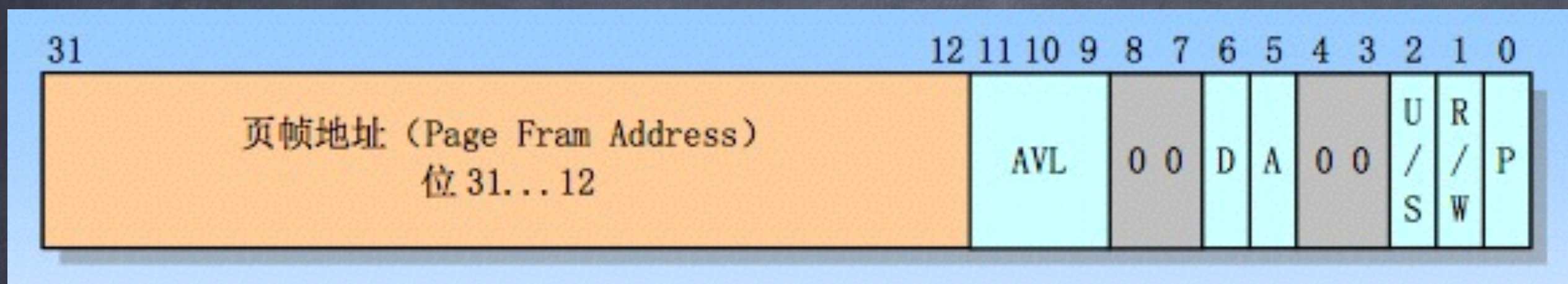


# 分页的地址转换





# 页目录表/页表的格式



**P:** 表示页是否在物理内存中，是虚拟内存工作的最关键的标记。

**D:** 页面是否被修改。当页面要被置换到磁盘时，该位决定了是否要回写。

**A:** 页面是否被访问。没被访问过的页面，优先被换出到磁盘。



# 大纲

- ① 1 操作系统是如何启动的
- ② 2 从实模式到保护模式
- ③ 3 硬件中断与系统调用
- ④ 4 多任务的实现原理
- ⑤ 5 程序的加载与执行
- ⑥ 6 总结



# 硬件中断 (Interrupt)

- 硬件中断是外围设备（时钟、键盘等）与CPU交互的方式。
- 每一个硬件中断都会有个唯一的硬件编号，当发生中断时，CPU可以通过这个编号，到中断描述符表里找到对应的处理程序入口地址，类似执行一个函数调用。
- 但中断程序执行完成之后，CPU怎么知道该怎么继续执行呢？这就靠在函数调用前，先将EIP的值压入内核栈，中断调用完成时，再弹出到EIP，这样就可以继续执行了。



# 异常 (Exception)

- CPU执行过程中，软件触发的。



# 异常的分类

- **Fault** (故障) : 可以被纠正的异常, 如缺页异常 (**Page Fault**), 异常处理后, 引起异常的指令会重复执行。
- **Trap** (陷阱) : 专门引起陷阱的指令, 有意为之, 如 **int n**。 **int n** 又称为软件中断。
- **Abort** (终止) : 严重错误, 指令位置无法确切定位, 关键系统数据存在异常, 最好关闭。



# 系统调用 (System Call)

- 想象你在C程序中，会通过 `fwrite` 调用文件写操作。`fwrite` 是C标准库实现的对系统函数 `write` 的一个封装。
- `write` 本身也可以直接使用，且核心系统软件都是通过 `write` 调用。`write` 内部又会嵌入汇编，比如 `write(...){asm_code(mov eax, 5; int 0x80;)} 这种方式。`



# int 0x80

- `int 0x80` 又是怎么回事呢？这就是系统调用的总入口。在Linux的实现中，通过`eax`保存系统调用的编号，比如`fork`的编号是2。`open`、`write`、`close`都有对应的编号。
- 根据参数的多少，使用寄存器传递调用参数或者使用栈。
- 某种意义上，实现OS的本质就是实现一组系统调用。



# int 0x80

- **Linus**在**Just for fun**自传中，就描述一段经历，在不断的找手册，实现几十个系统调用。特别是在移植**unix**程序时，如果某个系统调用不存在，就在屏幕上打印出来，他就知道必须要实现它了。
- **Posix: Portable Operating System Interface of Unix**，关键就是定义了操作系统的系统调用接口，要求按照这一标准。所以你会听说某某系统实现了**Posix**标准。



# 大纲

- ① 1 操作系统是如何启动的
- ② 2 从实模式到保护模式
- ③ 3 硬件中断与系统调用
- ④ 4 多任务的实现原理
- ⑤ 5 程序的加载与执行
- ⑥ 6 总结



# 回顾操作系统课程

- 操作系统的多任务（进程/线程）是怎么实现的？
- 是不是有一个超级的调度进程？那这个进程挂了又怎么办？



# 多任务的解决之道

- 答案就在时钟中断！
- 通过时钟中断，我们可以将任务状态保存，并跳转到新的任务。



# 任务状态段 (TSS)

- TSS (Task State Segment) 用于存放任务的执行状态信息，如EIP、SS、EAX等。这样只要保存了TSS，就可以把任务的时间静止了。然后换一个任务执行。



# TSS的全貌 (104 bytes)

31	16 15	0	
I/O 位图基地址		T	0x64
		LDT 段选择符	0x60
		GS	0x5C
		FS	0x58
		DS	0x54
		SS	0x50
		CS	0x4C
		ES	0x48
EDI			0x44
ESI			0x40
EBP			0x3C
ESP			0x38
EBX			0x34
EDX			0x30
ECX			0x2C
EAX			0x28
EFLAGS			0x24
EIP			0x20
页目录基地址寄存器 CR3 (PDBR)			0x1C
		SS2	0x18
ESP2			0x14
		SS1	0x10
ESP1			0x0C
		SS0	0x08
ESP0			0x04
		前一任务链接 (TSS 选择符)	0x00

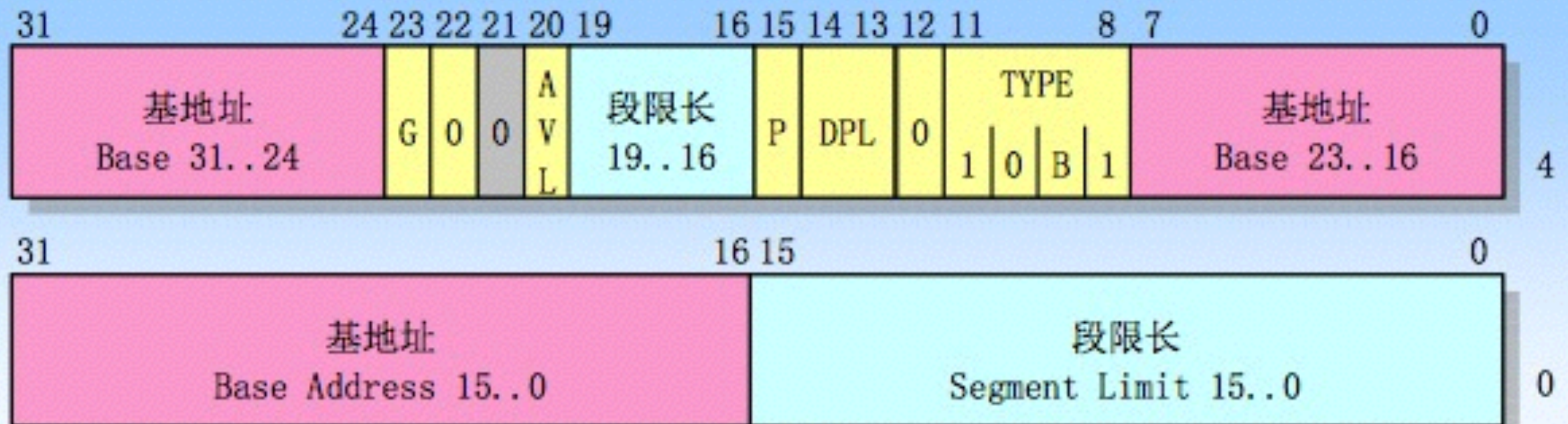


# 内核栈与用户栈

- TSS中，包括SS0/ESP0、SS1/ESP1、SS2/ESP2，分别对应于CPU特权级中的Ring 0、Ring 1、Ring 2所使用的栈。
- 在Linux中，只使用了Ring 0和Ring 3，Ring 3所对应的栈就是用户栈。



# TSS描述符（存放于GDT中）



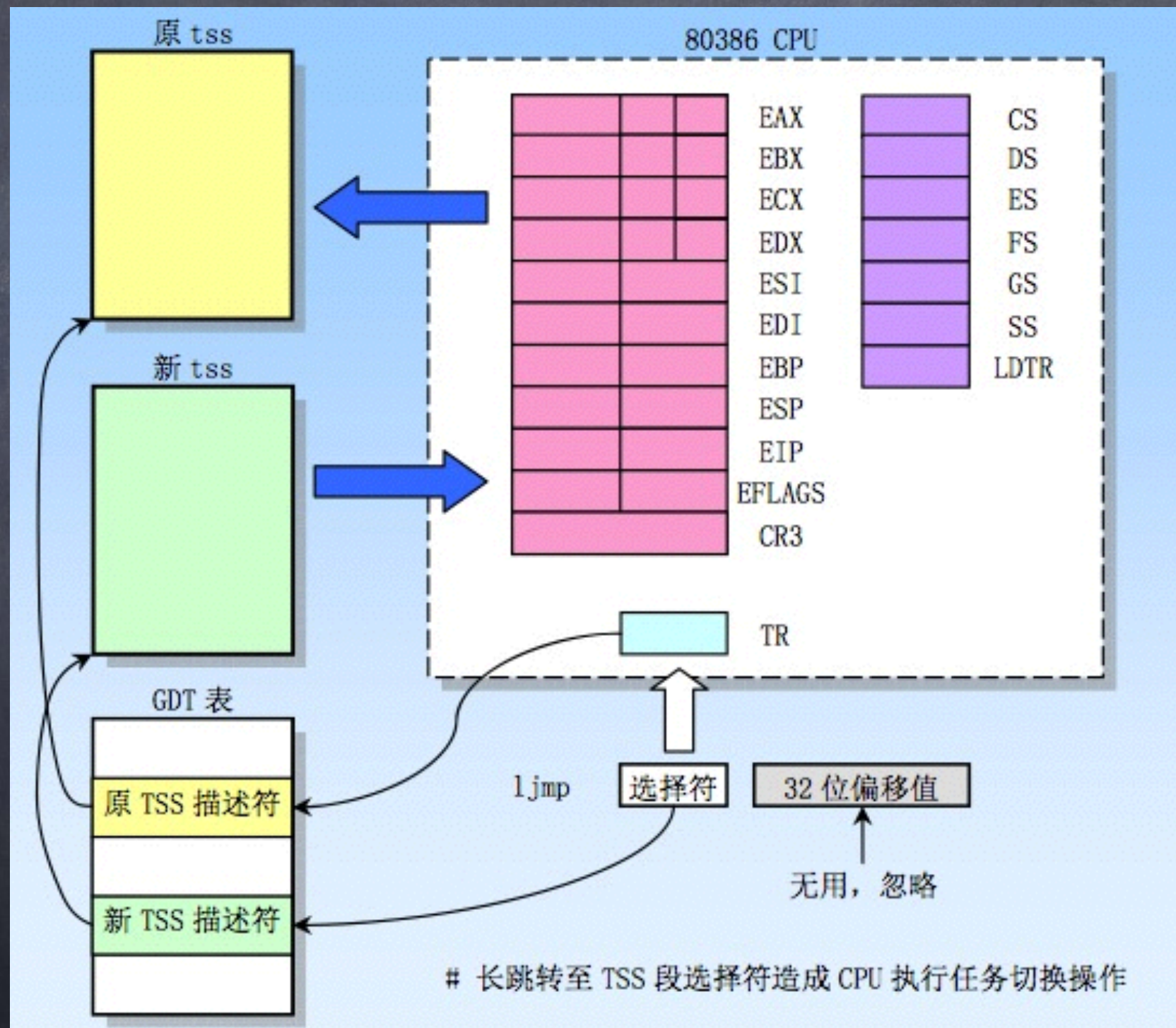


# 任务寄存器

- TR (Task Register) 存放了16位的GDT中的TSS选择符，以及在不可见部分存放了整个TSS的描述符（出于效率考虑）。
- 通过jmp TSS选择符：0x000000000000，可以实现TR中存放的TSS选择符的切换。后面的0x000000000000是无效的。



# 任务切换示意图





# 局部描述符表

- LDT (Local Descriptor Table) 与 GDT 相对，用于存放用户态任务 (Ring 3) 的局部使用的描述符，对其他任务不可见。
- LDTR 也是 TSS 中的一个字段。

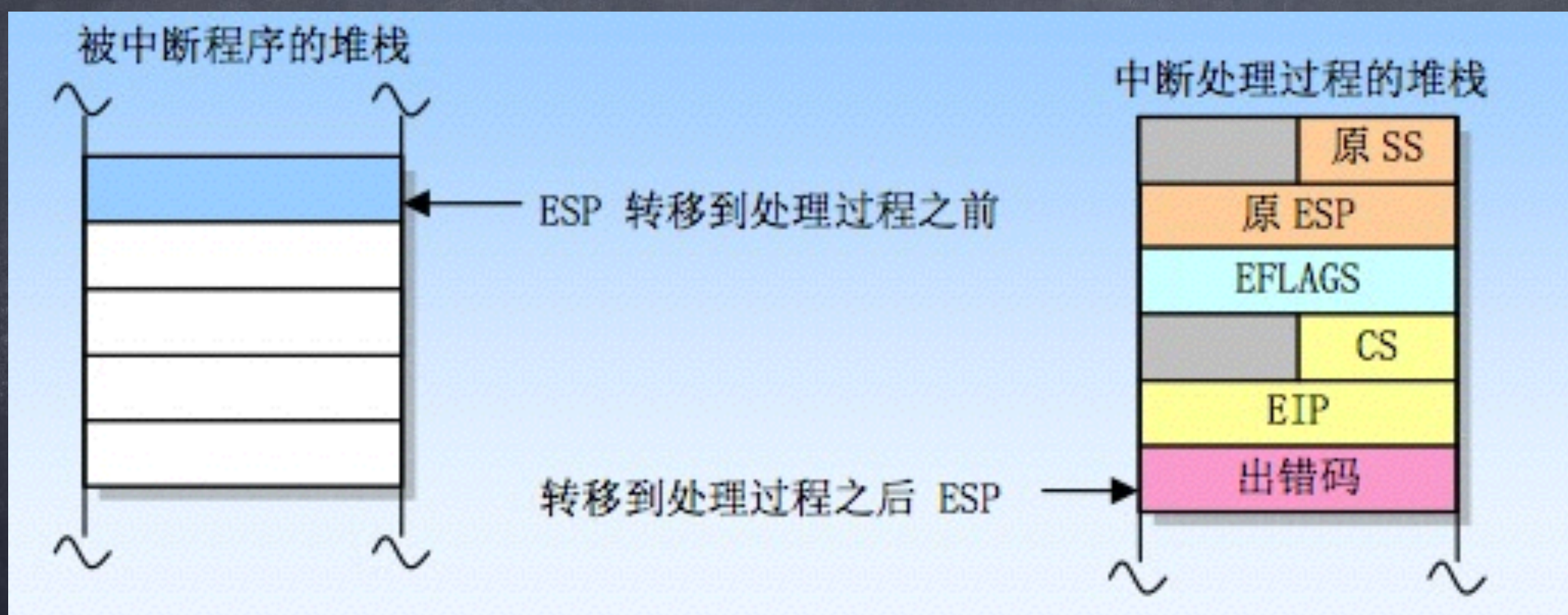


# 从用户态到内核态

- 硬件中断，或`int n`。如果是用户态的任务，牵涉到用户栈到内核栈的切换。
- 从当前TSS中，获取到SS0、ESP0，然后将当前SS、ESP等压入该栈，并更新SS、ESP为SS0、ESP0。



# 用户栈到内核栈的切换





# 从内核态到用户态

- 通过`iret`。
- 在实验中，我们会构造一个中断压栈的场景，然后调用`iret`跳转到用户态。



# 实现多任务的步骤

- 1, 设置时钟中断, 在时钟中断中, 实现任务切换的逻辑。
- 2, 设置GDT, 在GDT中, 设置任务的TSS/LDT描述符。
- 3, 设置TR、LDT寄存器, 并构造一个中断场景。
- 4, 调用iret跳转到用户态任务。



# 实验3

- 实现两个用户态任务，分别不断打印A和B，通过时钟中断实现任务切换。
- 遥想当年Linus实现Linux之前，就写了这么一个程序，很激动的拿给自己的亲妹妹看，她看了一眼觉得还不错，可是不理解。这就是传说中的Linux 0.00，是它孕育了后来的Linux。



# 关键代码1

```
int_timer: ;时钟中断处理函数
    push ds
    push eax

    mov al, 0x20
    out 0x20, al

    mov eax, DATA_SEL
    mov ds, ax

    mov eax, 1
    cmp [current], eax
    je .switch_0
    mov dword [current], 1
    jmp TSS1_SEL: 0 ;注意，执行这句后，马上保存了当前现场，
                    ;并跳转到了任务1之前的现场去执行，也就
                    ;下次切换回来时，是执行了下一句。是在内
                    ;核态时，别切换出去了。

    jmp .switch_finish
.switch_0:
    mov dword [current], 0
    jmp TSS0_SEL: 0
.switch_finish:
    pop eax
    pop ds
    iret
```



# 关键代码2

```
tss0:
    dd 0
    dd krn_stk0, DATA_SEL ; 第1个dd是内核栈
    dd 0, 0, 0, 0, 0
    dd 0, 0, 0, 0, 0
    dd 0
    dd 0, 0, 0, 0 ; 第14个dd是用户栈
    dd 0, 0, 0, 0x17, 0, 0
    dd LDT0_SEL, 0x80000000

times 128 dd 0
krn_stk0:

align 8
ldt0:
    dw 0, 0, 0, 0
    dw 0x03ff, 0x0000, 0xfa00, 0x00c0
    dw 0x03ff, 0x0000, 0xf200, 0x00c0
```



# 关键代码3

```
pushfd ;复位eflags中的任务嵌套标志，不用关注  
and dword [esp], 0xffffbfff  
popfd
```

```
mov eax, TSS0_SEL  
ltr ax  
mov eax, LDT0_SEL  
lldt ax  
mov dword [current], 0  
sti ;开中断
```

```
push 0x17 ;任务0的局部数据段用于ss  
push init_stack  
pushfd  
push 0x0f ;CS  
push task0 ;EIP  
iret
```

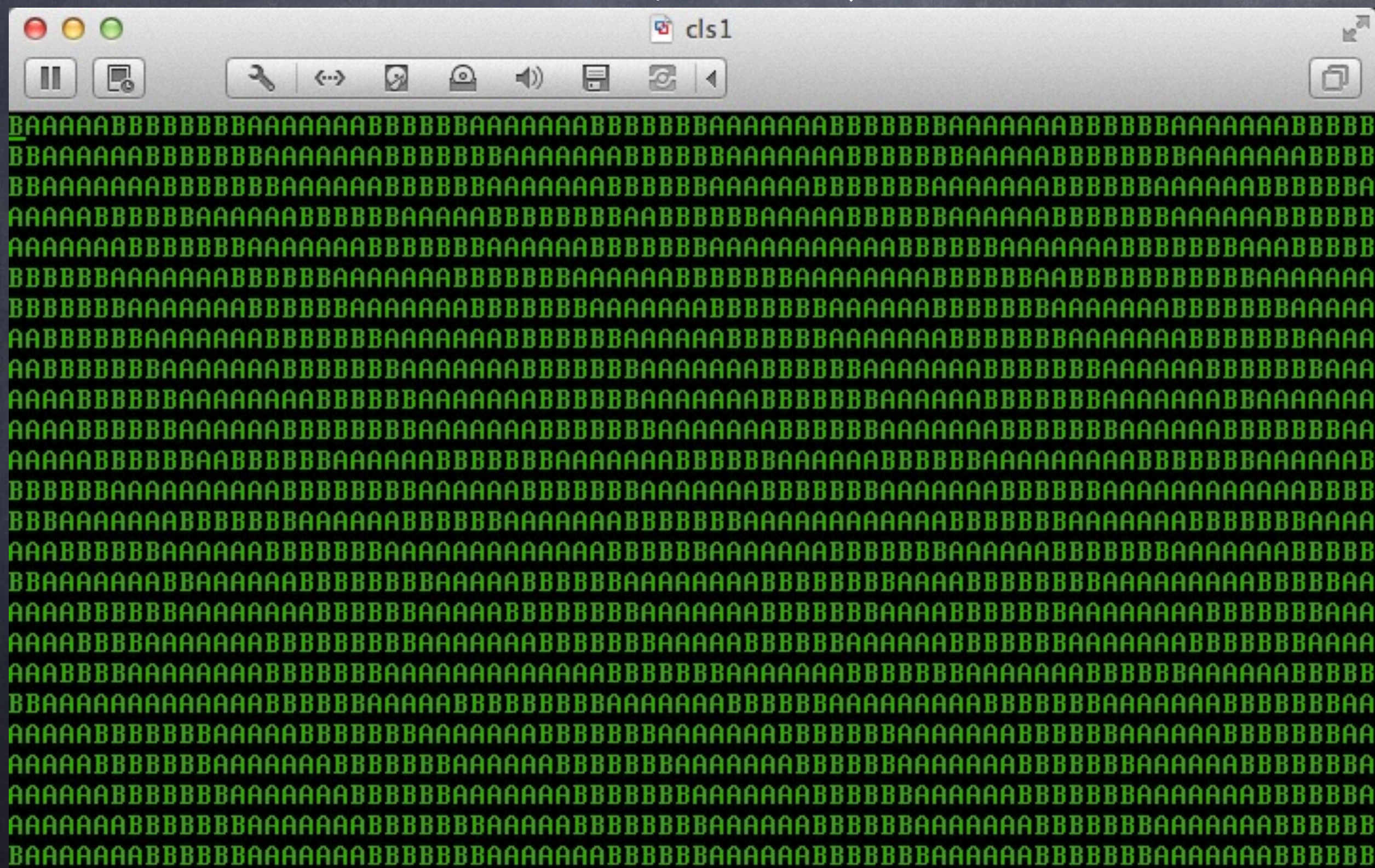


# 关键代码4

```
task0:
    mov al, 'A'
    int 0x81
    mov ecx, 0xffffffff
    .t0:
    loop .t0
    jmp task0
task1:
    mov al, 'B'
    int 0x81
    mov ecx, 0xffffffff
    .t1:
    loop .t1
    jmp task1
```



## 运行结果





# 大纲

- ① 1 操作系统是如何启动的
- ② 2 从实模式到保护模式
- ③ 3 硬件中断与系统调用
- ④ 4 多任务的实现原理
- ⑤ 5 程序的加载与执行
- ⑥ 6 总结



# 可执行程序

- 一种特殊格式的文件，里面包含机器指令，可以被操作系统加载执行。



# 常见可执行文件的格式

- ELF — Executable and Linkable Format, Linux/Unix 的默认格式。
- PE — Portable Execute, Windows 的默认格式。
- Macho — Mac OS 的默认格式。



# ELF格式

Linking View

ELF header
Program header table <i>optional</i>
Section 1
...
Section $n$
...
...
Section header table

Execution View

ELF header
Program header table
Segment 1
Segment 2
...
Section header table <i>optional</i>



# ELF Header

```
typedef struct {
    unsigned char    e_ident[EI_NIDENT];
    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;
} Elf32_Ehdr;
```



# ELF Header

- `e_ident`: ELF 文件标识, 以 `0x7f` 开头。
- `e_type`: 文件具体类型, 可重定向文件/可执行文件/Core 文件等。
- `e_entry`: 可执行程序执行时的入口地址。



# 可重定向文件

- 代码的绝对地址在编译时无法确定，需要在链接时确定和调整。



# 实验4：认识ELF文件

```
1 #include <stdio.h>
2
3 int main(int argc, char* argv[])
4 {
5     printf("Hello, world!\n");
6
7     return 0;
8 }
```



# 实验4：认识ELF文件

- `$ i586-pc-linux-gcc helloworld.c`
- `$ i586-pc-linux-readelf -h a.out`
- `$ i586-pc-linux-objdump -d a.out`



# 实验4：认识ELF文件

## ELF Header:

Magic:	7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
Class:	ELF32
Data:	2's complement, little endian
Version:	1 (current)
OS/ABI:	UNIX - System V
ABI Version:	0
Type:	EXEC (Executable file)
Machine:	Intel 80386
Version:	0x1
Entry point address:	0x80482f0
Start of program headers:	52 (bytes into file)
Start of section headers:	3776 (bytes into file)
Flags:	0x0



# 实验4：认识ELF文件

```
a.out:      file format elf32-i386
```

```
Disassembly of section .init:
```

```
08048274 <_init>:
```

8048274:	55	push	%ebp
8048275:	89 e5	mov	%esp,%ebp
8048277:	53	push	%ebx
8048278:	83 ec 04	sub	\$0x4,%esp
804827b:	e8 00 00 00 00	call	8048280 <_init+0xc>
8048280:	5b	pop	%ebx



# 回顾bootloader的加载

- 读取第0号扇区，加载到0x7C00地址，然后从0x7C00执行。



# 程序的加载

File Offset	File	Virtual Address
0	ELF header	
Program header table		
	Other information	
0x100	Text segment ... 0x2be00 bytes	0x8048100  0x8073eff
0x2bf00	Data segment ... 0x4e00 bytes	0x8074f00  0x8079cff
0x30d00	Other information ...	



# 程序的加载步骤 (exec系统调用)

- 1, 读取ELF Header;
- 2, 读取各个段并放到内存指定位置 (或设置好页表映射);
- 3, 修改栈上的函数返回地址EIP为e\_entry;
- 4, 中断返回后, 即从e\_entry开始执行。



# 实验5: `exec`系统调用

- 实现一个简单的`exec`系统调用
- 如果传入文件名为“`print_hello`”，则执行`print_hello`程序，否则，执行`print_world`程序。
- `print_hello/print_world`均为Binary格式而非ELF格式，简化实现。

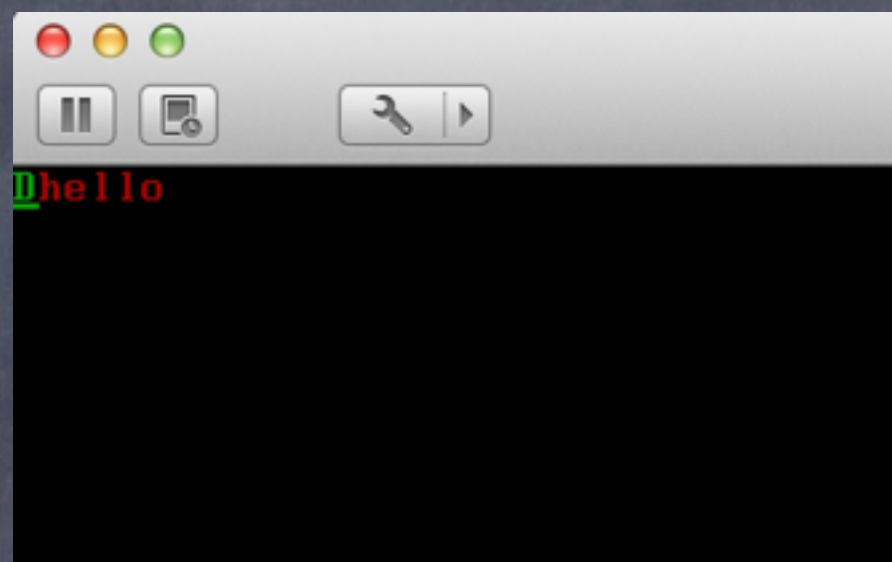


# 代码1

```
1 #include "system.h"
2
3 int my_entrance()
4 {
5     if (fork_syscall() == 0) {
6         // son
7         exec("print_hello");
8         while(1) ;
9     } else {
10        // dad
11        print_char('D');
12        while(1) ;
13    }
14
15    return 0;
16 }
```



运行结果:

A screenshot of a terminal window with a light gray title bar. The title bar contains three colored window control buttons (red, yellow, green) on the left, and three icons (a pause button, a copy button, and a run button) on the right. The main area of the terminal is black, and the text 'hello' is displayed in red. The first letter 'h' is underlined with a green cursor.

```
hhello
```



# 大纲

- ① 1 操作系统是如何启动的
- ② 2 从实模式到保护模式
- ③ 3 硬件中断与系统调用
- ④ 4 多任务的实现原理
- ⑤ 5 程序的加载与执行
- ⑥ 6 总结



# 总结

- 持续两年的业余时间学习，才有了上面的成果，挺有成就感。
- 单枪匹马写一个可用的商业操作系统是不现实的。



# 推荐资料

- Linux内核完全剖析
- 自己动手写操作系统
- <https://github.com/sangwf/walleclass>



谢谢！