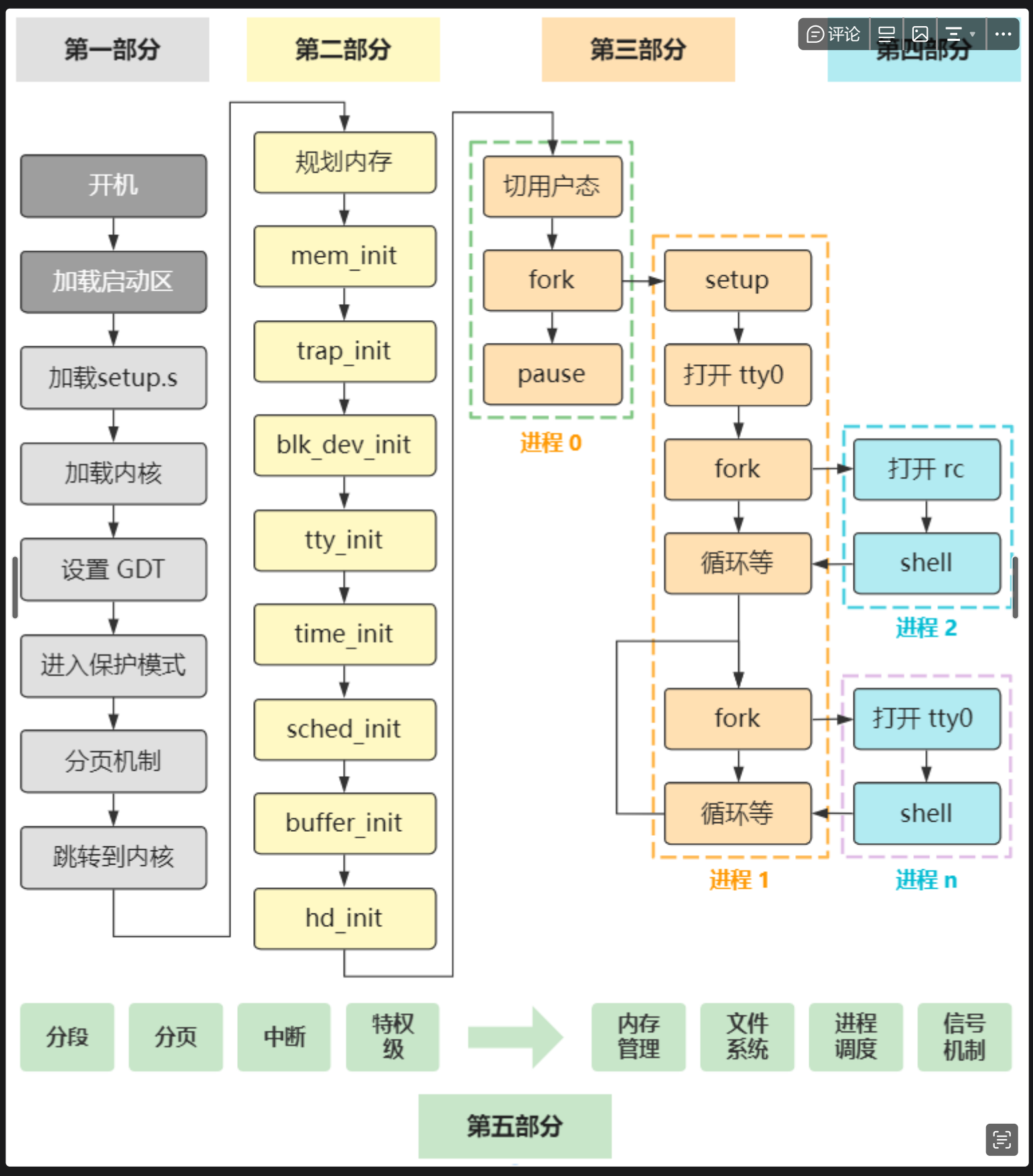
第七回

新读者看这里，老读者直接跳过。

本系列会以一个读小说的心态，从开机启动后的代码执行顺序，带着大家阅读和赏析 Linux 0.11 全部核心代码，了解操作系统的技术细节和设计思想。



你会跟着我一起，看着一个操作系统从啥都没有开始，一步一步最终实现它复杂又精巧的设计，读完这个系列后希望你能发出感叹，原来操作系统源码就是这破玩意。

以下是**已发布文章**的列表，详细了解本系列可以先从开篇词看起。

开篇词

第一回 | 最开始的两行代码

第二回 | 自己给自己挪个地儿

第三回 | 做好最最基础的准备工作

第四回 | 把自己在硬盘里的其他部分也放到内存来

第五回 | 进入保护模式前的最后一次折腾内存

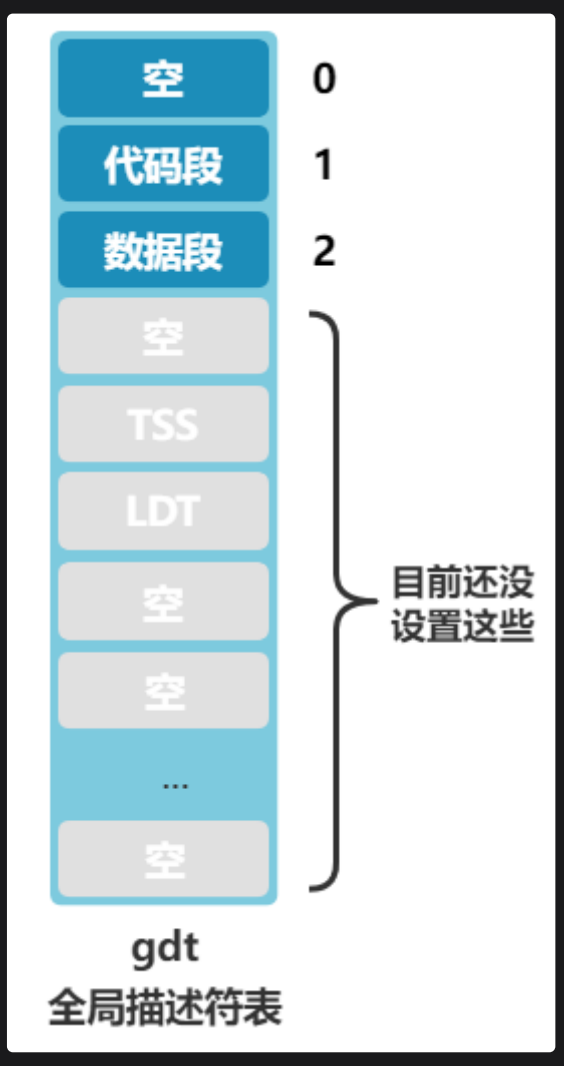
第六回 | 先解决段寄存器的历史包袱问题

本系列的 GitHub 地址如下（文末阅读原文可直接跳转）

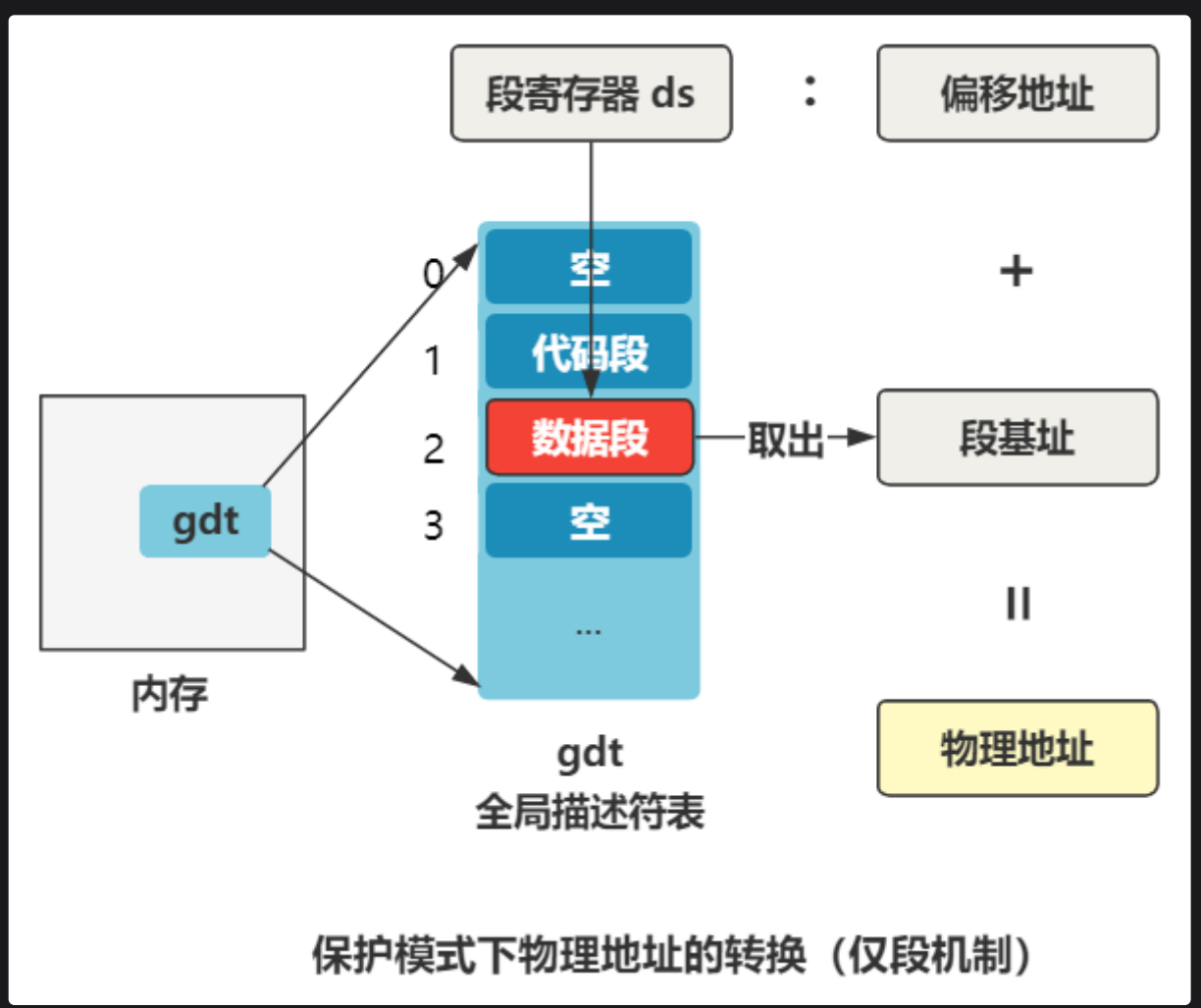
https://github.com/sunym1993/flash-linux0.11-talk

------- **正文开始** -------

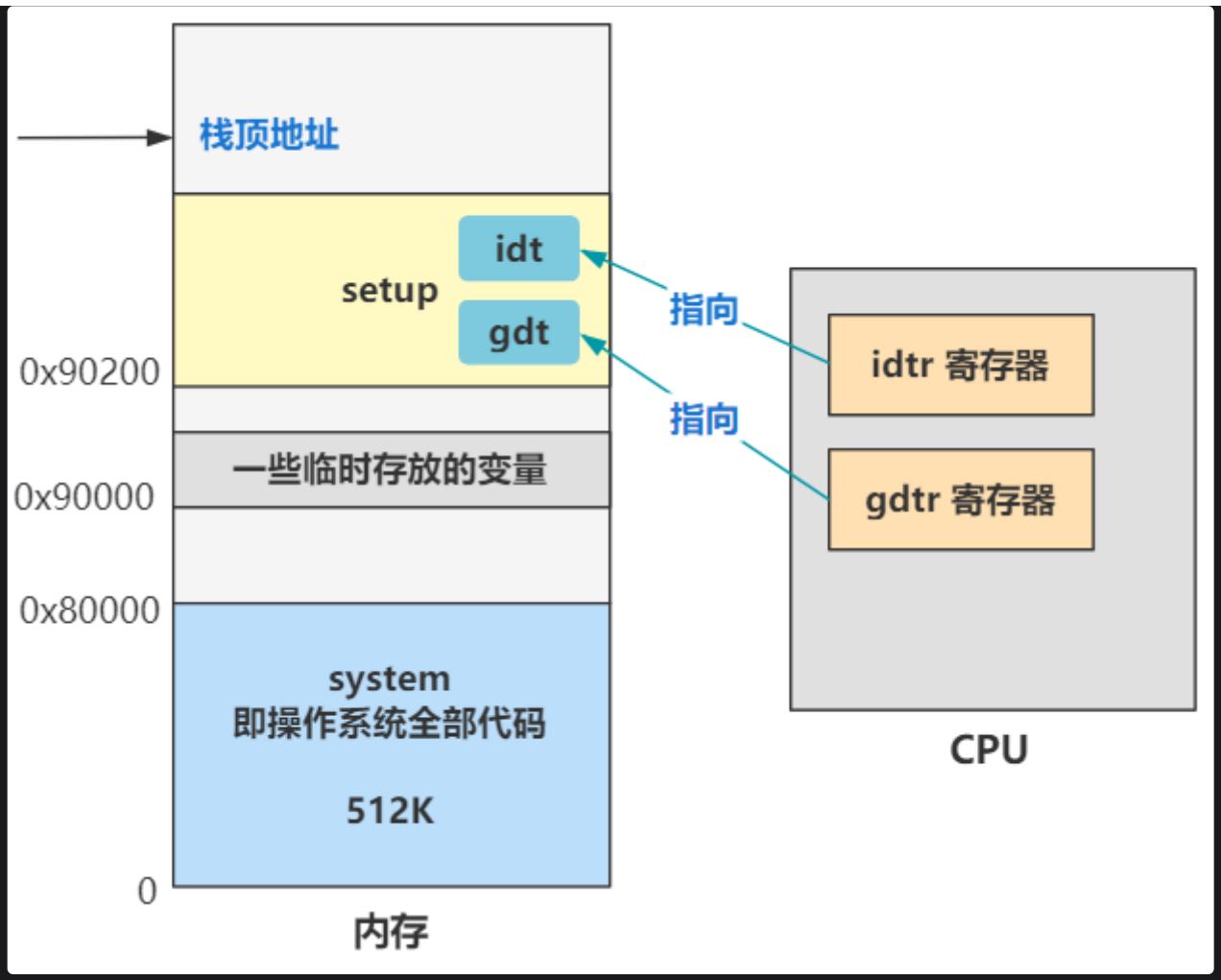
书接上回，上回书咱们说到，操作系统设置了个全局描述符表 gdt。



为后面切换到保护模式后，能去那里寻找到段描述符，然后拼凑成最终的物理地址。



而此时我们的内存布局变成了这个样子。



这仅仅是进入保护模式前准备工作的其中一个，我们接着往下看。代码仍然是 setup.s 中的。

```
    mov al,#0xD1        ; command writeout #0x64,almov al,#0xDF        ; A20 onout #0x60,al
```

这段代码的意思是，**打开 A20 地址线**。

说人话就是，打开 A20 地址线。哈哈，开玩笑，到底什么是 A20 地址线呢？

简单理解，这一步就是为了突破地址信号线 20 位的宽度，变成 32 位可用。这是由于 8086 CPU 只有 20 位的地址线，所以如果程序给出 21 位的内存地址数据，那多出的一位就被忽略了，比如如果经过计算得出一个内存地址为

1 0000 00000000 00000000

那实际上内存地址相当于 0，因为高位的那个 1 被忽略了，地方不够。

当 CPU 到了 32 位时代之后，由于要考虑**兼容性**，还必须保持一个只能用 20 位地址线的模式，所以如果你不手动开启的话，即使地址线已经有 32 位了，仍然会限制只能使用其中的 20 位。

简单吧？我们继续。

接下来的一段代码，你完全完全不用看，但为了防止你一直记挂在心上，我给你截出来说道说道，这样以后我说完全不用看的代码时，你就真的可以放宽心完全不看了。

就是这一大坨，还有 Linus 自己的注释。

```
    ; well, that went ok, I hope. Now we have to reprogram the interrupts :-(; we put them right after the intel-reserved hardware interrupts, at; int 0x20-0x2F. There they won't mess up anything. Sadly IBM really; messed this up with the original PC, and they haven't been able to; rectify it afterwards. Thus the bios puts interrupts at 0x08-0x0f,; which is used for the internal hardware interrupts as well. We just; have to reprogram the 8259's, and it isn't fun.    mov al,#0x11        ; initialization sequence    out #0x20,al    ; send it to 8259A-1    .word    0x00eb,0x00eb        ; jmp $+2, jmp $+2    out #0xA0,al        ; and to 8259A-2    .word    0x00eb,0x00eb    mov al,#0x20        ; start of hardware int's (0x20)    out #0x21,al    .word    0x00eb,0x00eb    mov al,#0x28        ; start of hardware int's 2 (0x28)    out #0xA1,al    .word    0x00eb,0x00eb    mov al,#0x04        ; 8259-1 is master    out #0x21,al    .word    0x00eb,0x00eb    mov al,#0x02        ; 8259-2 is slave    out #0xA1,al    .word    0x00eb,0x00eb    mov al,#0x01        ; 8086 mode for both    out #0x21,al    .word    0x00eb,0x00eb    out #0xA1,al    .word    0x00eb,0x00eb    mov al,#0xFF        ; mask off all interrupts for now    out #0x21,al    .word    0x00eb,0x00eb    out #0xA1,al
```

这里是对**可编程中断控制器 8259 芯片**进行的编程。

因为中断号是不能冲突的， Intel 把 0 到 0x19 号中断都作为**保留中断**，比如 0 号中断就规定为**除零异常**，软件自定义的中断都应该放在这之后，但是 IBM 在原 PC 机中搞砸了，跟保留中断号发生了冲突，以后也没有纠正过来，所以我们得重新对其进行编程，不得不做，却又一点意思也没有。这是 Linus 在上面注释上的原话。

所以我们也不必在意，只要知道重新编程之后，8259 这个芯片的引脚与中断号的对应关系，变成了如下的样子就好。

PIC 请求号	中断号	用途
IRQ0	0x20	时钟中断
IRQ1	0x21	键盘中断
IRQ2	0x22	接连从芯片
IRQ3	0x23	串口 2
IRQ4	0x24	串口 1
IRQ5	0x25	并口 2
IRQ6	0x26	软盘驱动器
IRQ7	0x27	并口 1
IRQ8	0x28	实时钟中断
IRQ9	0x29	保留
IRQ10	0x2a	保留
IRQ11	0x2b	保留
IRQ12	0x2c	鼠标中断
IRQ13	0x2d	数学协处理器
IRQ14	0x2e	硬盘中断
IRQ15	0x2f	保留

好了，接下来的一步，就是真正切换模式的一步了，从代码上看就两行。

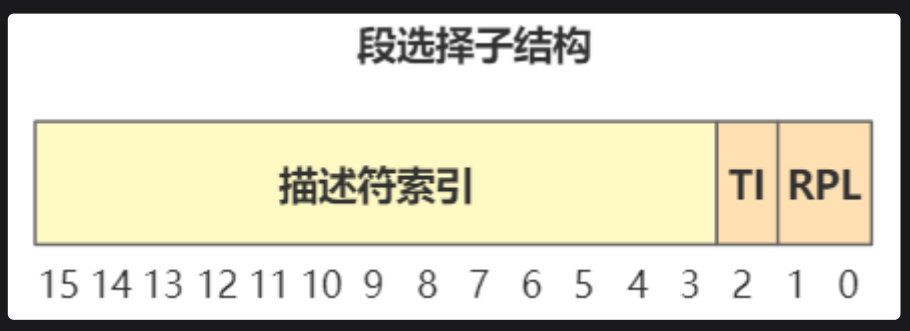
```
    mov ax,#0x0001  ; protected mode (PE) bitlmsw ax      ; This is it;jmpi 0,8     ; jmp offset 0 of segment 8 (cs)
```

前两行，将 cr0 这个寄存器的位 0 置 1，模式就从实模式切换到保护模式了。

所以真正的模式切换十分简单，重要的是之前做的准备工作。

再往后，又是一个段间跳转指令 **jmpi**，后面的 8 表示 cs（代码段寄存器）的值，0 表示偏移地址。请注意，此时已经是保护模式了，之前也说过，保护模式下内存寻址方式变了，段寄存器里的值被当做段选择子。

回顾下段选择子的模样。



8 用二进制表示就是

00000,0000,0000,1000

对照上面段选择子的结构，可以知道**描述符索引值是 1**，也就是要去**全局描述符表（gdt）**中找第一项段描述符。

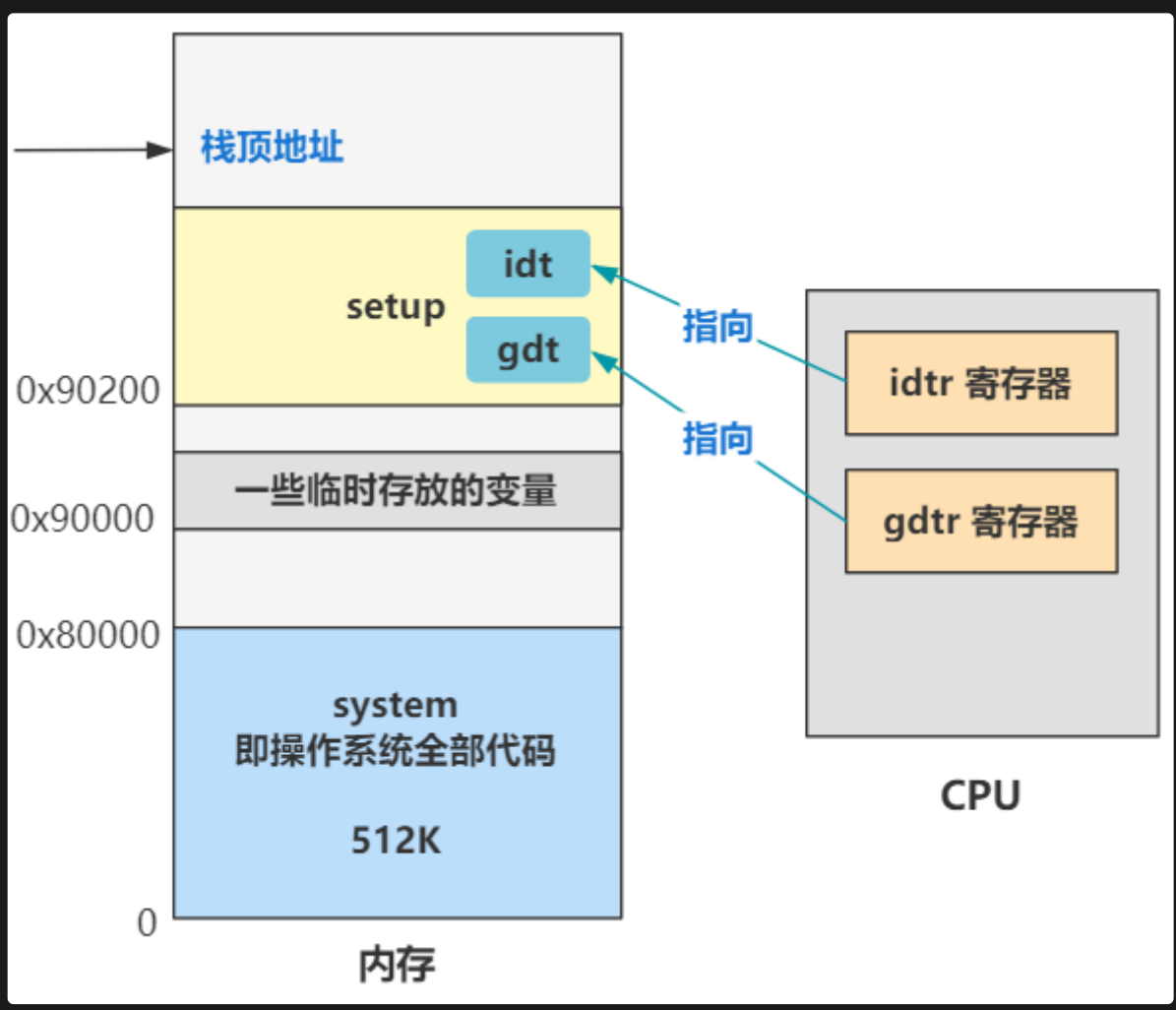
还记得上一讲中的全局描述符的具体内容么？

```
    gdt:    .word    0,0,0,0     ; dummy    .word    0x07FF      ; 8Mb - limit=2047 (20484096=8Mb)    .word    0x0000      ; base address=0
    .word    0x9A00      ; code read/exec    .word    0x00C0      ; granularity=4096, 386    .word    0x07FF      ; 8Mb - limit=2047
    (20484096=8Mb)    .word    0x0000      ; base address=0    .word    0x9200      ; data read/write    .word    0x00C0      ; granularity=4096,
    386
```

我们说了，第 0 项是空值，第一项被表示为**代码段描述符**，是个可读可执行的段，第二项为**数据段描述符**，是个可读可写段，不过他们的段基址都是 0。

所以，这里取的就是这个代码段描述符，**段基址是 0**，偏移也是 0，那加一块就还是 0 咯，所以最终这个跳转指令，就是跳转到内存地址的 0 地址处，开始执行。

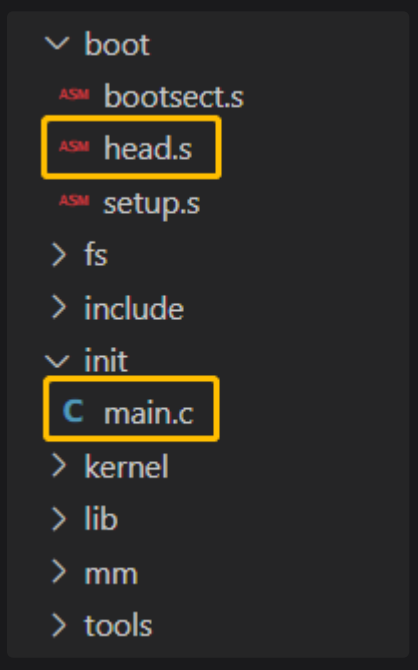
零地址处是什么呢？还是回顾之前的内存布局图。



就是操作系统全部代码的 system 这个大模块，system 模块怎么生成的呢？由 Makefile 文件可知，是由 head.s 和 main.c 以及其余各模块的操作系统代码合并来的，可以理解为操作系统的全部核心代码编译后的结果。

tools/system: boot/head.o init/main.o \ $(ARCHIVES)$ (DRIVERS) $(MATH)$ (LIBS) (LD) (LDFLAGS) boot/head.o init/main.o \ ƒ(x) 公式错误 (DRIVERS) \ ƒ(x) 公式错误 (LIBS) \ -o tools/system > System.map

所以，接下来，我们就要重点阅读 **head.s** 了。



这也是 boot 文件夹下的最后一个由汇编写就的源代码文件，哎呀，不知不觉就把两个操作系统源码文件（**bootsect.s** 和 **setup.s**）讲完了，而且是汇编写的令人头疼的代码。

head.s 这个文件仅仅是为了顺利进入由后面的 c 语言写就的 main.c 做的准备，所以咬咬牙看完这个之后，我们就终于可以进入 c 语言的世界了！也终于可以看到我们熟悉的 main 函数了！

在那里，操作系统真正秀操作的地方，才刚刚开始！欲知后事如何，且听下回分解。