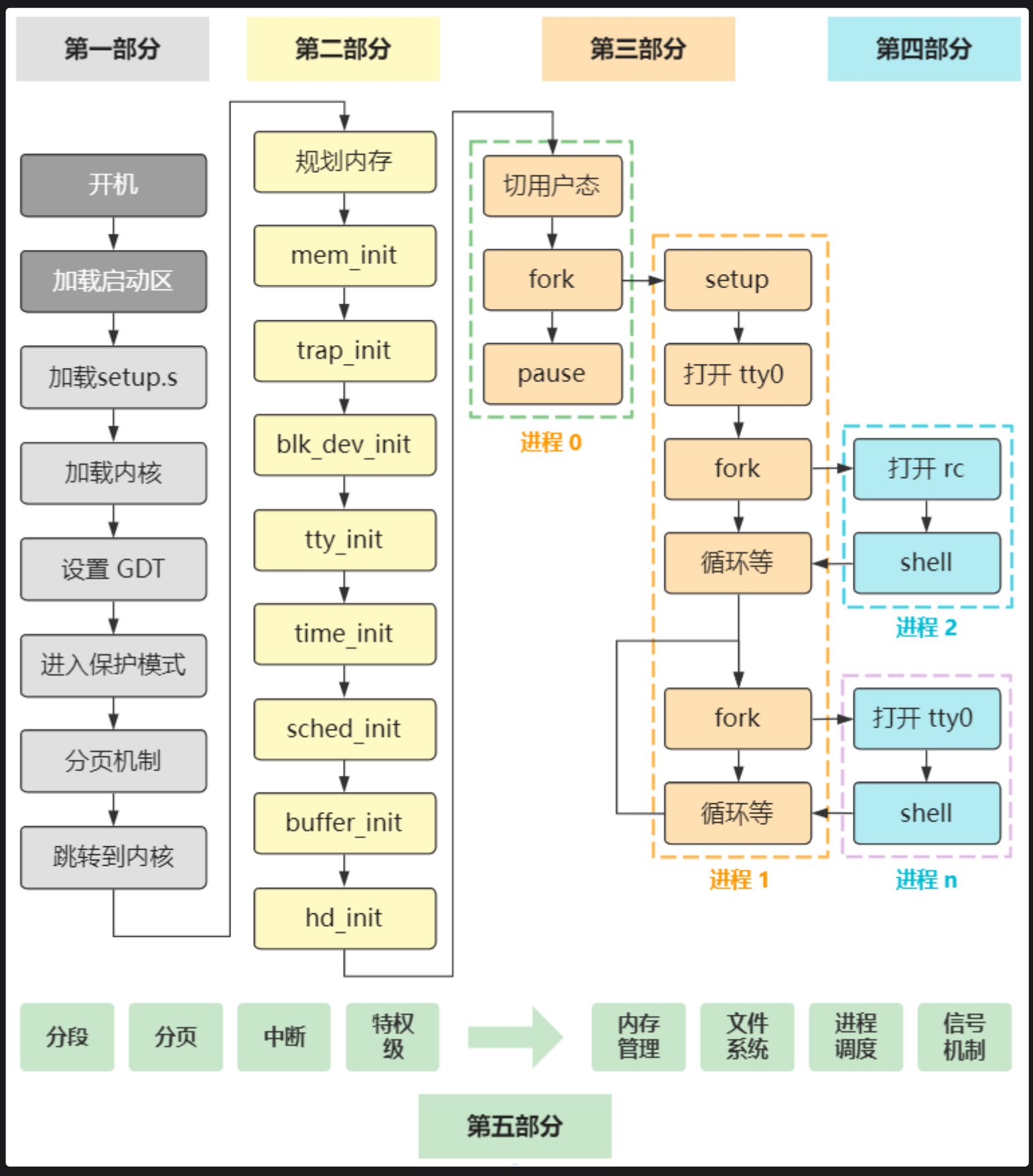
第八回

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

第七回 | 六行代码就进入了保护模式

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，CPU 进入了 32 位保护模式，我们快速回顾一下关键的代码。

首先配置了全局描述符表 gdt 和中断描述符表 idt。

```
    lidt  idt_48lgdt  gdt_48
```

然后打开了 A20 地址线。

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

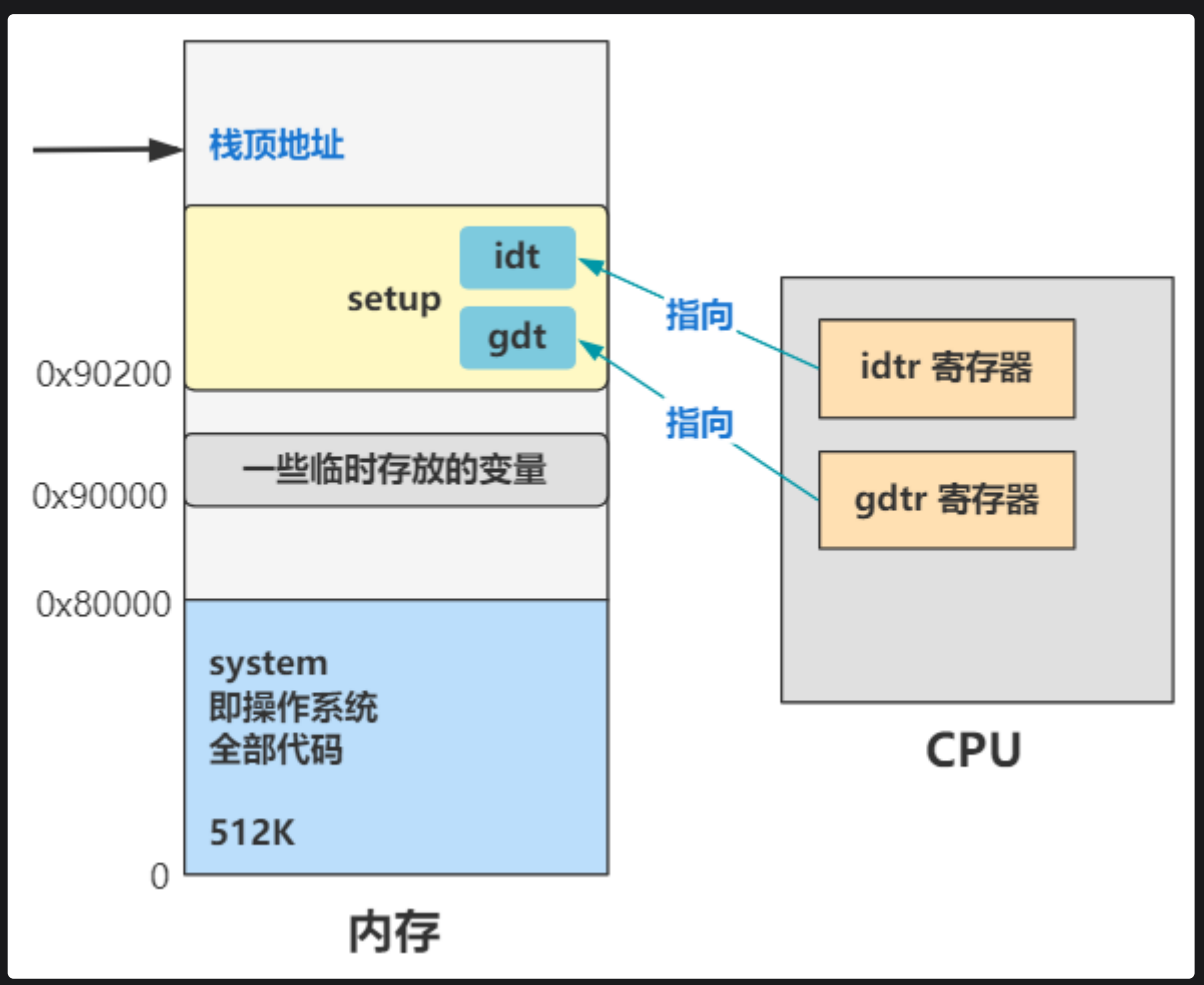
然后更改 cr0 寄存器开启保护模式。

```
    mov ax,#0x0001lmsw ax
```

最后，一个干脆利落的跳转指令，跳到了内存地址 0 处开始执行代码。

```
    jmpi 0,8
```

0 位置处存储着操作系统全部核心代码，是由 head.s 和 main.c 以及后面的无数源代码文件编译并链接在一起而成的 system 模块。



那接下来，我们就品品，正式进入 c 语言写的 main.c 之前的 **head.s** 究竟写了点啥?

head.s 文件很短，我们一点点品。

```
    _pg_dir:_startup_32:    mov eax,0x10    mov ds,ax    mov es,ax    mov fs,ax    mov gs,ax    lss esp,_stack_start
```

注意到开头有个标号 **_pg_dir**。先留个心眼，这个表示**页目录**，之后在设置分页机制时，页目录会存放在这里，也会覆盖这里的代码。

再往下连续五个 **mov** 操作，分别给 ds、es、fs、gs 这几个段寄存器赋值为 **0x10**，根据段描述符结构解析，表示这几个段寄存器的值为指向全局描述符表中的第二个段描述符，也就是数据段描述符。

最后 **lss** 指令相当于让 **ss:esp** 这个栈顶指针指向了 **_stack_start** 这个标号的位置。还记得图里的那个原来的栈顶指针在哪里吧？往上翻一下，**0x9FF00**，现在要变咯。

这个 stack_start 标号定义在了很久之后才会讲到的 **sched.c** 里，我们这里拿出来分析一波。

```
    long user_stack[4096 >> 2];struct{  long *a;  short b;}stack_start = {&user_stack[4096 >> 2], 0x10};
```

这啥意思呢?

首先，stack_start 结构中的高位 8 字节是 **0x10**，将会赋值给 **ss** 栈段寄存器，低位 16 字节是 **user_stack** 这个数组的最后一个元素的地址值，将其赋值给 **esp** 寄存器。

赋值给 ss 的 0x10 仍然按照保护模式下的**段选择子**去解读，其指向的是全局描述符表中的第二个段描述符（数据段描述符），段基址是 0。

赋值给 esp 寄存器的就是 user_stack 数组的最后一个元素的内存地址值，那最终的**栈顶地址**，也指向了这里（user_stack + 0），后面的压栈操作，就是往这个新的栈顶地址处压咯。

继续往下看

```
call setup_idt ;设置中断描述符表 call setup_gdt ;设置全局描述符表 mov eax,10hmov ds,axmov es,axmov fs,axmov gs,axlss esp,_stack_start
```

先设置了 **idt** 和 **gdt**，然后又重新执行了一遍刚刚执行过的代码。

为什么要重新设置这些段寄存器呢？因为上面修改了 gdt，所以要重新设置一遍以刷新才能生效。那我们接下来就把目光放到设置 idt 和 gdt 上。

中断描述符表 idt 我们之前没设置过，所以这里设置具体的值，理所应当。

```
setup_idt:  lea edx,ignore_int  mov eax,00080000h  mov ax,dx  mov dx,8E00h  lea edi,_idt  mov ecx,256rp_sidt:  mov [edi],eax  mov [edi+4],edx  add edi,8  dec ecx  jne rp_sidt  lidt fword ptr idt_descr  retidt_descr:  dw 256*8-1  dd _idt_idt:  DQ 256 dup(0)
```

不用细看，我给你说最终效果。

中断描述符表 idt 里面存储着一个个中断描述符，每一个中断号就对应着一个中断描述符，而中断描述符里面存储着主要是中断程序的地址，这样一个中断号过来后，CPU 就会自动寻找相应的中断程序，然后去执行它。

那这段程序的作用就是，**设置了 256 个中断描述符**，并且让每一个中断描述符中的中断程序例程都指向一个 **ignore_int** 的函数地址，这个是个**默认的中断处理程序**，之后会逐渐被各个具体的中断程序所覆盖。比如之后键盘模块会将自己的键盘中断处理程序，覆盖过去。

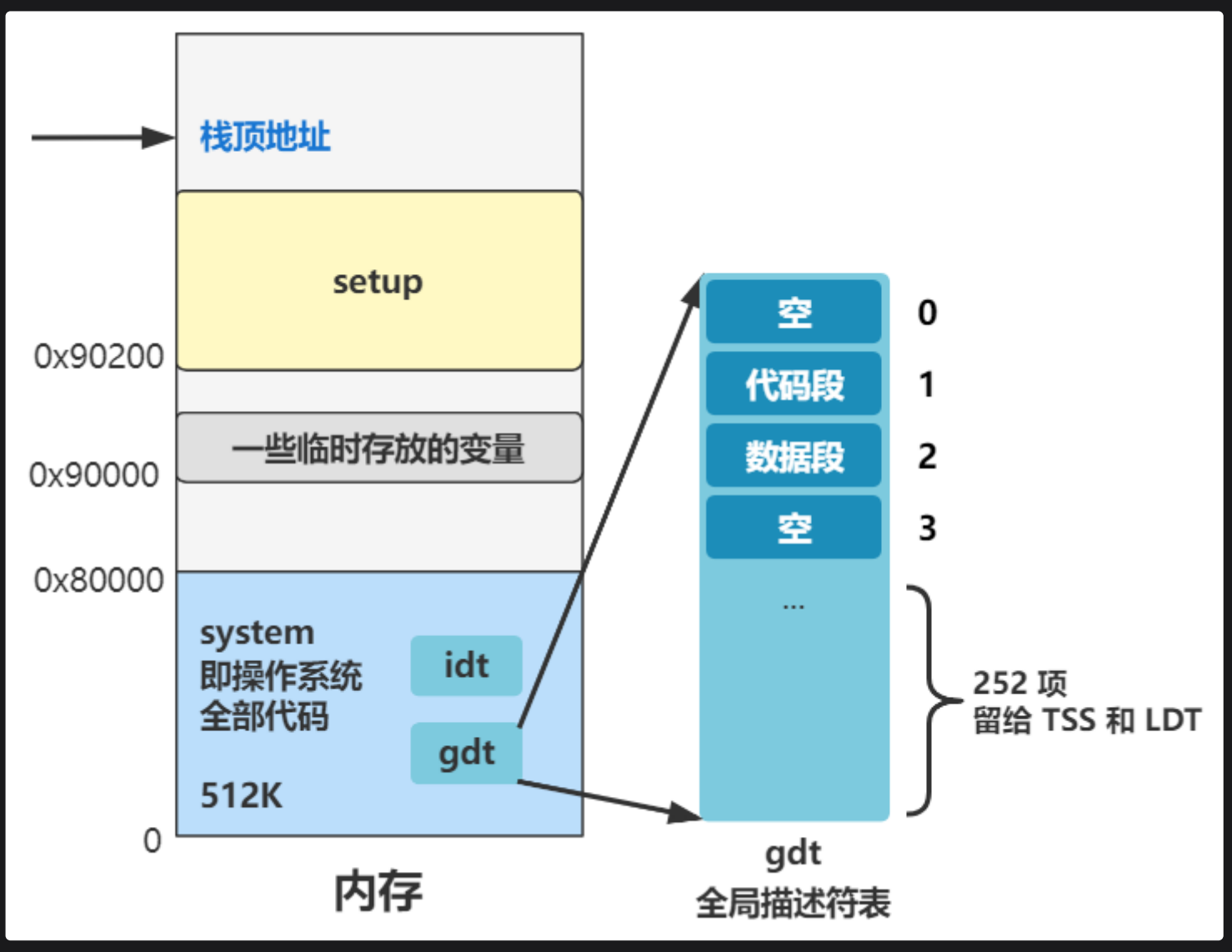
那现在，产生任何中断都会指向这个默认的函数 ignore_int，也就是说现在这个阶段**你按键盘还不好使**。

设置中断描述符表 setup_idt 说完了，那接下来 **setup_gdt** 就同理了。我们就直接看设置好后的新的全局描述符表长什么样吧？

```
_gdt:  DQ 0000000000000000h  ;/* NULL descriptor /  DQ 00c09a0000000fffh  ;/ 16Mb /  DQ 00c0920000000fffh  ;/ 16Mb /  DQ 0000000000000000h  ;/ TEMPORARY - don't use */  DQ 252 dup(0)
```

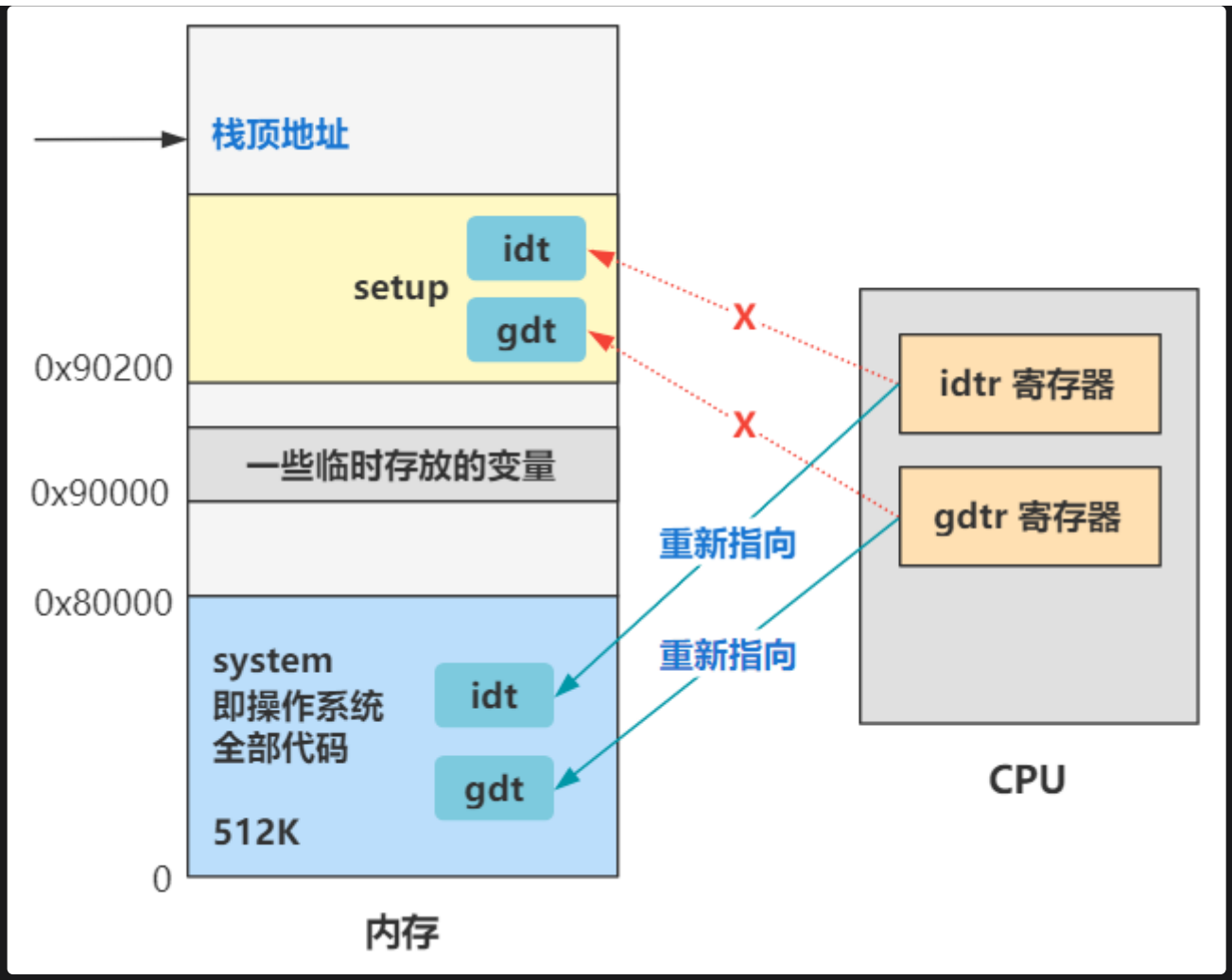
其实和我们原先设置好的 gdt 一模一样。

也是有**代码段描述符**和**数据段描述符**，然后第四项系统段描述符并没有用到，不用管。最后还留了 252 项的空间，这些空间后面会用来放置**任务状态段描述符 TSS** 和**局部描述符 LDT**，这个后面再说。



为什么原来已经设置过一遍了，这里又要重新设置一遍，你可千万别想有什么复杂的原因，就是因为原来设置的 gdt 是在 setup 程序中，之后这个地方要被缓冲区覆盖掉，所以这里重新设置在 head 程序中，这块内存区域之后就不会被其他程序用到并且覆盖了，就这么个事。

说的口干舌燥，还是来张图吧。



如果你本文的内容完全不能理解，那就记住最后这张图就好了，本文代码就是完成了这个图中所示的一个指向转换而已，并且给所有中断设置了一个默认的中断处理程序 ignore_int，然后全局描述符表仍然只有代码段描述符和数据段描述符。

好了，本文就是两个描述符表位置的变化以及重新设置，再后面一行代码就是又一个令人兴奋的功能了！

```
    jmp after_page_tables...after_page_tables:    push 0    push 0    push 0    push L6    push _main    jmp setup_pagingL6:    jmp L6
```

那就是开启分页机制，并且跳转到 main 函数！

这可太令人兴奋了！开启分页后，配合着之前讲的分段，就构成了内存管理的最最底层的机制。而跳转到 main 函数，标志着我们正式进入 c 语言写的操作系统核心代码！

欲知后事如何，且听下回分解。