第九回 | Intel 内存管理两板斧：分段与分页

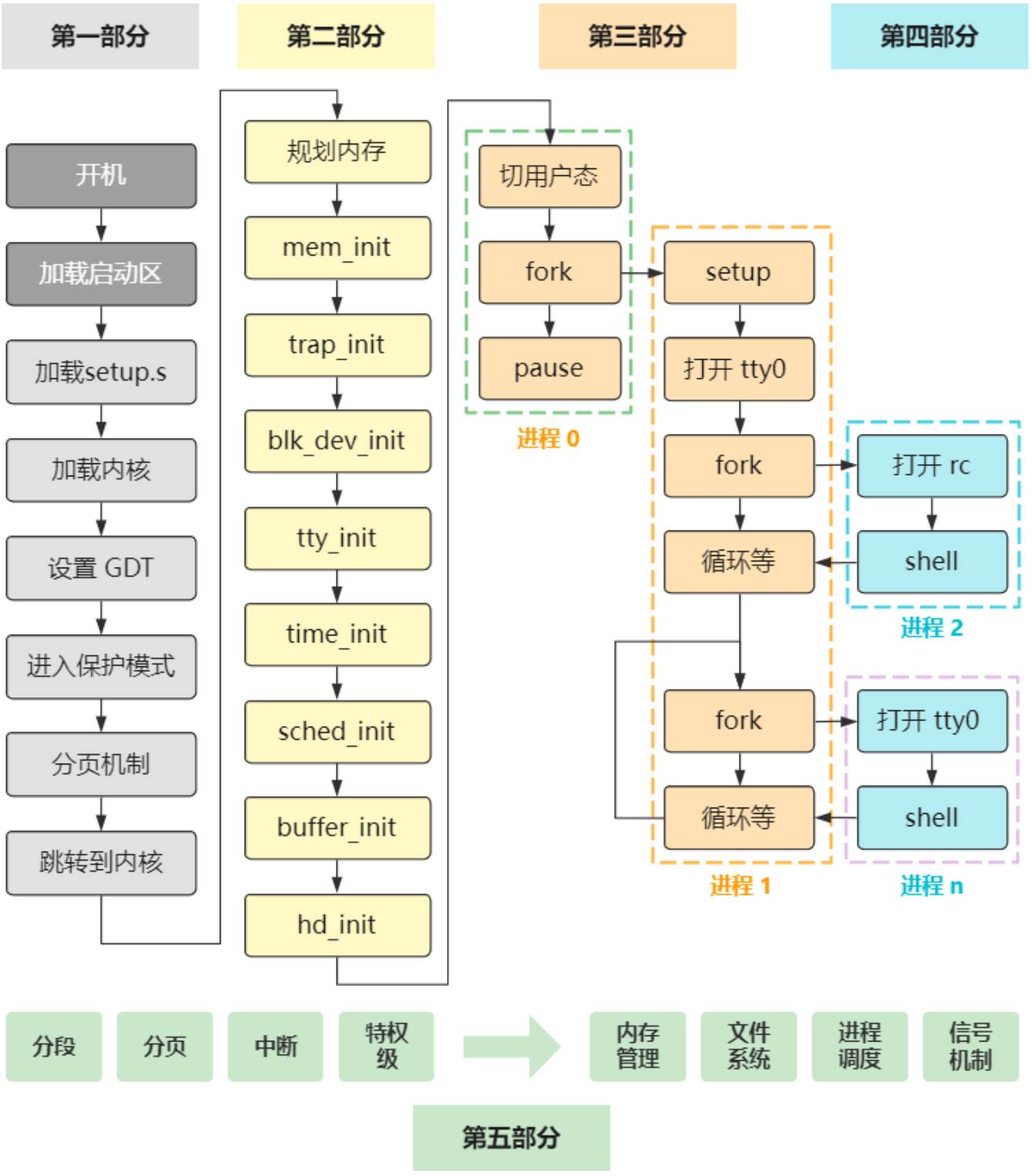
Original 闪客 低并发编程 2021-12-08 16:30

收录于合集
#操作系统源码

43个

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。

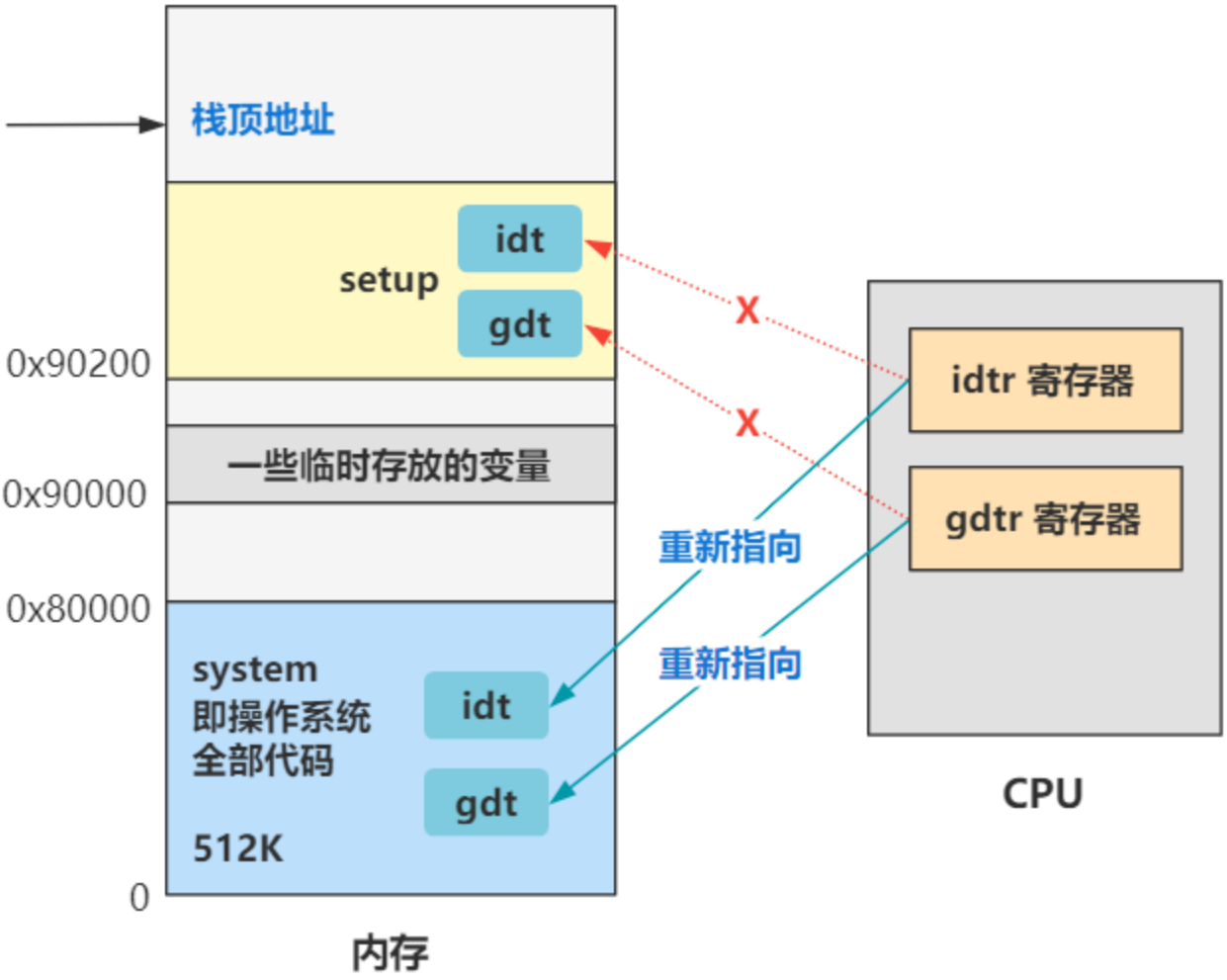
以下是**已发布文章**的列表，详细了解本系列可以先从开篇词看起。

开篇词

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，head.s 代码在重新设置了 gdt 与 idt 后。



来到了这样一段代码。

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

那就是开启分页机制，并且跳转到 main 函数。

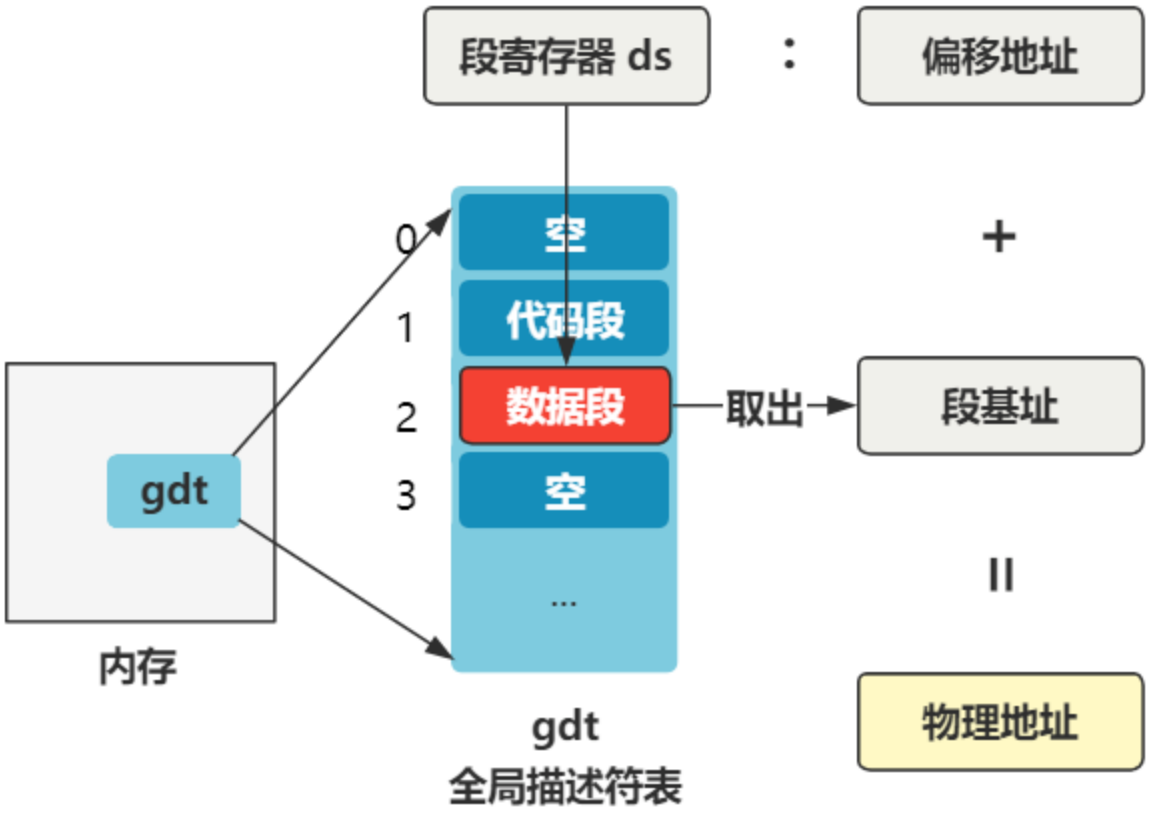
如何跳转到之后用 c 语言写的 main.c 里的 main 函数，是个有趣的事，也包含在这段代码里。不过我们先瞧瞧这**分页机制**是如何开启的，也就是 **setup_paging** 这个标签处的代码。

```
setup_paging:
    mov ecx,1024*5
    xor eax,eax
    xor edi,edi
    pushf
    cld
    rep stosd
    mov eax,_pg_dir
    mov [eax],pg0+7
    mov [eax+4],pg1+7
    mov [eax+8],pg2+7
    mov [eax+12],pg3+7
    mov edi,pg3+4092
    mov eax,00fff007h
    std
L3: stosd
    sub eax,00001000h
    jge L3
    popf
    xor eax,eax
    mov cr3,eax
    mov eax,cr0
    or  eax,80000000h
    mov cr0,eax
    ret
```

别怕，我们一点点来分析。

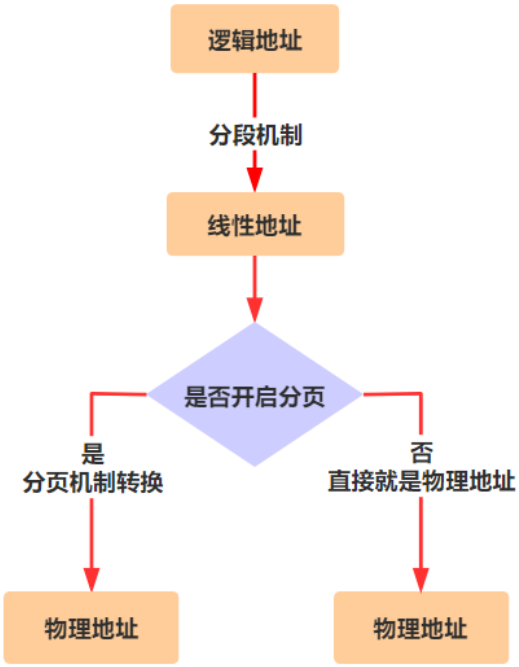
首先要了解的就是，啥是分页机制？

还记不记得之前我们在代码中给出一个内存地址，在保护模式下要先经过分段机制的转换，才能最终变成物理地址，就是这样。



保护模式下物理地址的转换（仅段机制）

这是在没有开启分页机制的时候，只需要经过这一步转换即可得到最终的物理地址了，但是在开启了分页机制后，又会**多一步转换**。



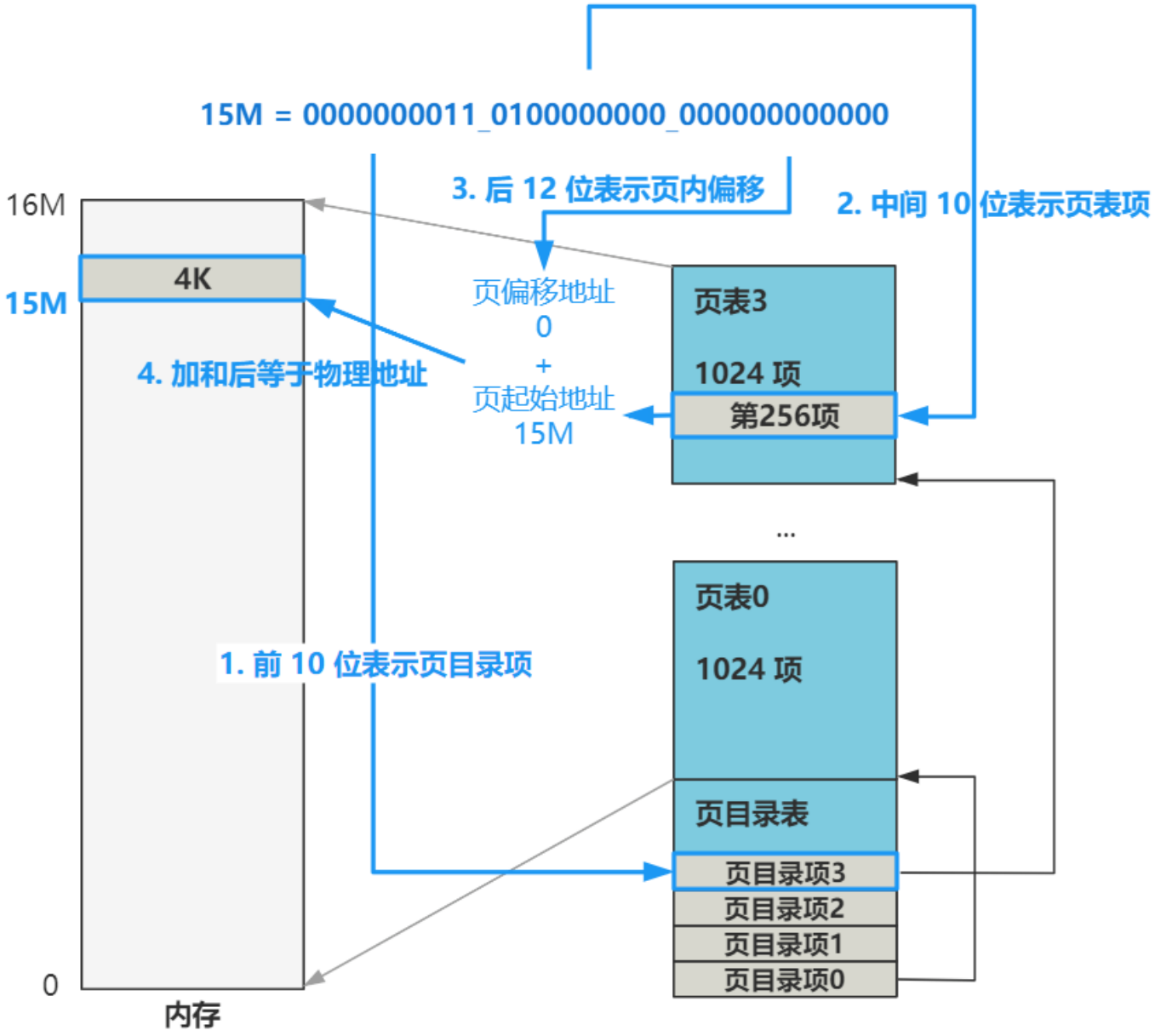
也就是说，在没有开启分页机制时，由程序员给出的**逻辑地址**，需要先通过分段机制转换成物理地址。但在开启分页机制后，逻辑地址仍然要先通过分段机制进行转换，只不过转换后不再是最终的物理地址，而是**线性地址**，然后再通过一次分页机制转换，得到最终的物理地址。

分段机制我们已经清楚如何对地址进行变换了，那分页机制又是如何变换的呢？我们直接以一个例子来学习过程。

比如我们的线性地址（已经经过了分段机制的转换）是
15M

二进制表示就是
0000000011_0100000000_000000000000

我们看一下它的转换过程



也就是说，CPU 在看到我们给出的内存地址后，首先把线性地址被拆分成

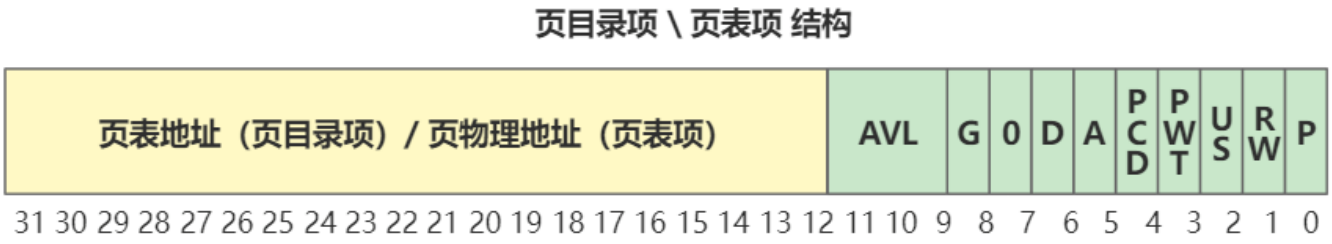
高 10 位：中间 10 位：后 12 位

高 10 位负责在**页目录表**中找到一个**页目录项**，这个页目录项的值加上中间 10 位拼接后的地址去**页表**中去寻找一个**页表项**，这个页表项的值，再加上后 12 位偏移地址，就是最终的物理地址。

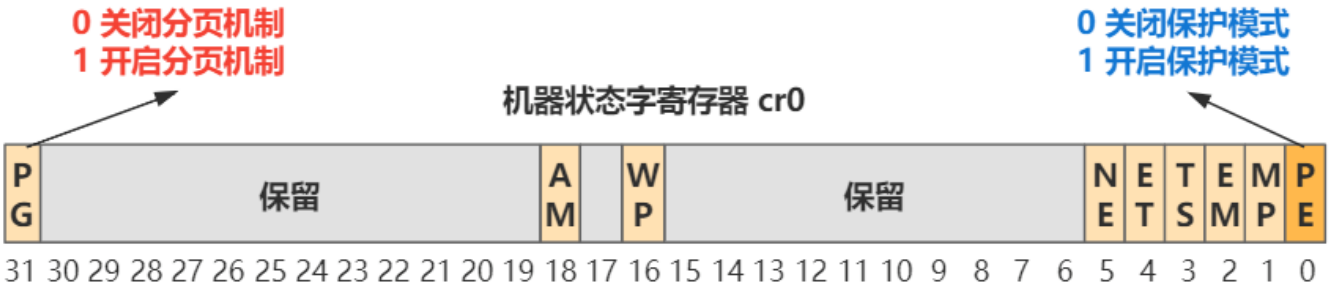
而这一切的操作，都由计算机的一个硬件叫 **MMU**，中文名字叫**内存管理单元**，有时也叫 PMMU，分页内存管理单元。由这个部件来负责将虚拟地址转换为物理地址。

所以整个过程我们不用操心，作为操作系统这个软件层，只需要提供好页目录表和页表即可，

这种页表方案叫做**二级页表**，第一级叫**页目录表　PDE**，第二级叫**页表　PTE**。他们的结构如下。



之后再开启分页机制的开关。其实就是更改 **cr0** 寄存器中的一位即可（31 位），还记得我们开启保护模式么，也是改这个寄存器中的一位的值。



然后，MMU 就可以帮我们进行分页的转换了。此后指令中的内存地址（就是程序员提供的逻辑地址），就统统要先经过分段机制的转换，再通过分页机制的转换，才能最终变成物理地址。

所以这段代码，就是帮我们把页表和页目录表在内存中写好，之后开启　cr0　寄存器的分页开关，仅此而已，我们再把代码贴上来。

```
setup_paging:
    mov ecx,1024*5
    xor eax,eax
    xor edi,edi
    pushf
    cld
    rep stosd
    mov eax,_pg_dir
    mov [eax],pg0+7
    mov [eax+4],pg1+7
    mov [eax+8],pg2+7
    mov [eax+12],pg3+7
    mov edi,pg3+4092
    mov eax,00fff007h
    std
L3: stosd
    sub eax,00001000h
    jge L3
    popf
    xor eax,eax
    mov cr3,eax
    mov eax,cr0
    or  eax,80000000h
    mov cr0,eax
    ret
```

我们先说这段代码最终产生的效果吧。

当时 linux-0.11 认为，总共可以使用的内存不会超过 **16M**，也即最大地址空间为 **0xFFFFFF**。

而按照当前的页目录表和页表这种机制，1 个页目录表最多包含 1024 个页目录项（也就是 1024 个页表），1 个页表最多包含 1024 个页表项（也就是 1024 个页），1 页为 4KB（因为有 12 位偏移地址），因此，16M 的地址空间可以用 1 个页目录表 + 4 个页表搞定。

4（页表数）* 1024（页表项数） * 4KB（一页大小）= 16MB

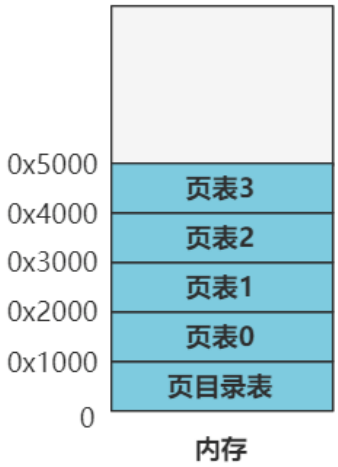
所以，上面这段代码就是，**将页目录表放在内存地址的最开头**，还记得上一讲开头让你留意的 _pg_dir 这个标签吧？

```
_pg_dir:
_startup_32:
    mov eax,0x10
    mov ds,ax
    ...
```

之后紧挨着这个页目录表，放置 4 个页表，代码里也有这四个页表的标签项。

```
.org 0x1000 pg0:
.org 0x2000 pg1:
.org 0x3000 pg2:
.org 0x4000 pg3:
.org 0x5000
```

最终将页目录表和页表填写好数值，来覆盖整个 16MB 的内存。随后，开启分页机制。此时内存中的页表相关的布局如下。



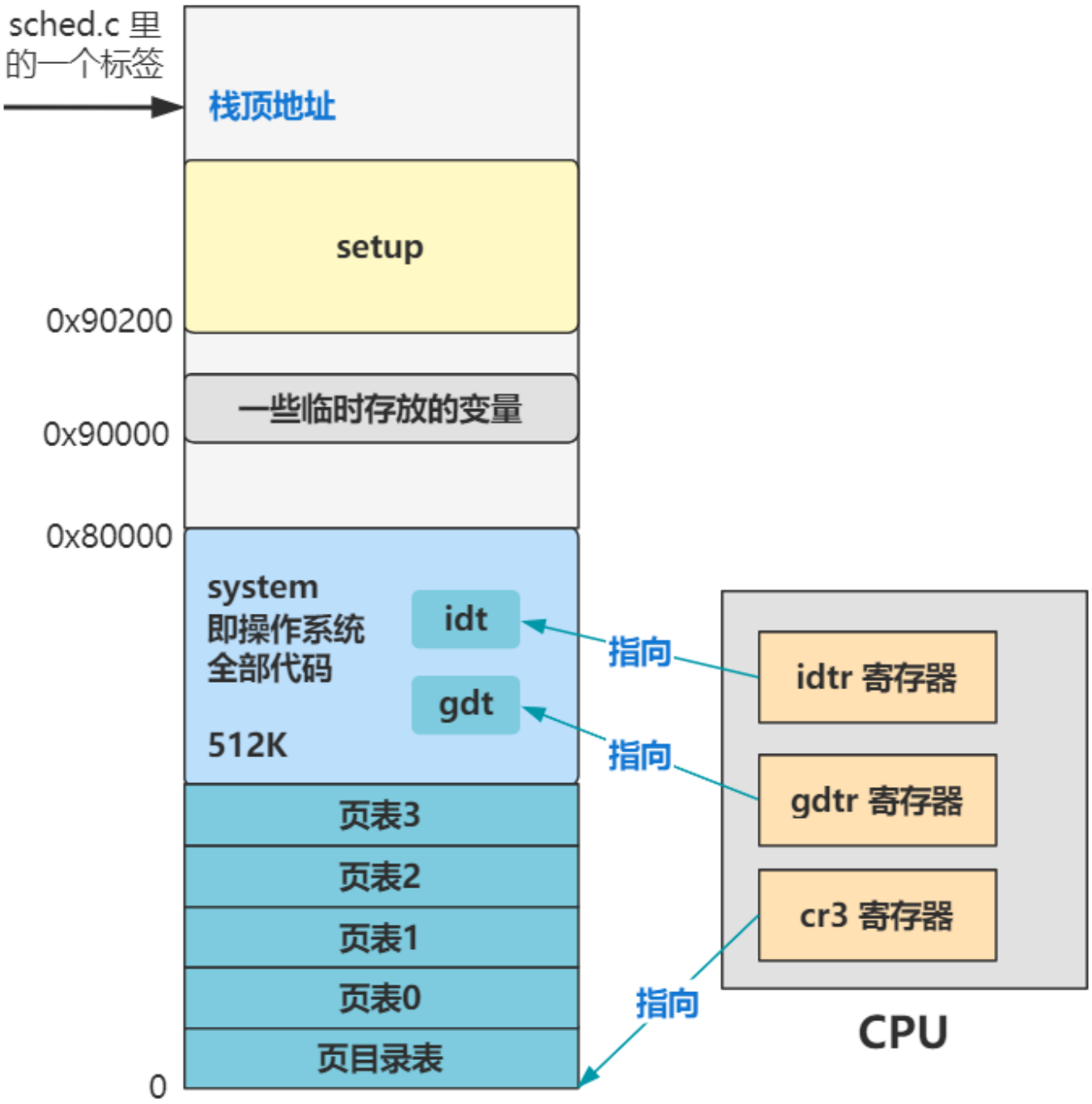
这些页目录表和页表放到了整个内存布局中最开头的位置，就是覆盖了开头的 system 代码了，不过被覆盖的 system 代码已经执行过了，所以无所谓。

同时，如 idt 和 gdt 一样，我们也需要通过一个寄存器告诉 CPU 我们把这些页表放在了哪里，就是这段代码。

```
xor eax,eax
mov cr3,eax
```

你看，我们相当于告诉 cr3 寄存器，**0 地址处就是页目录表，再通过页目录表可以找到所有的页表**，也就相当于 CPU 知道了分页机制的全貌了。

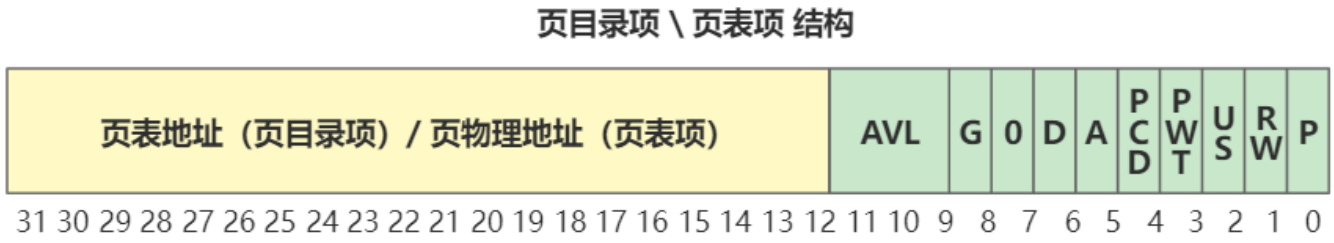
至此后，整个内存布局如下。



那么具体页表设置好后，映射的内存是怎样的情况呢？那就要看页表的具体数据了，就是这一坨代码。

```
setup_paging:
    ...
    mov eax,_pg_dir
    mov [eax],pg0+7
    mov [eax+4],pg1+7
    mov [eax+8],pg2+7
    mov [eax+12],pg3+7
    mov edi,pg3+4092
    mov eax,00fff007h
    std
L3: stosd
    sub eax, 1000h
    jpe L3
    ...
```

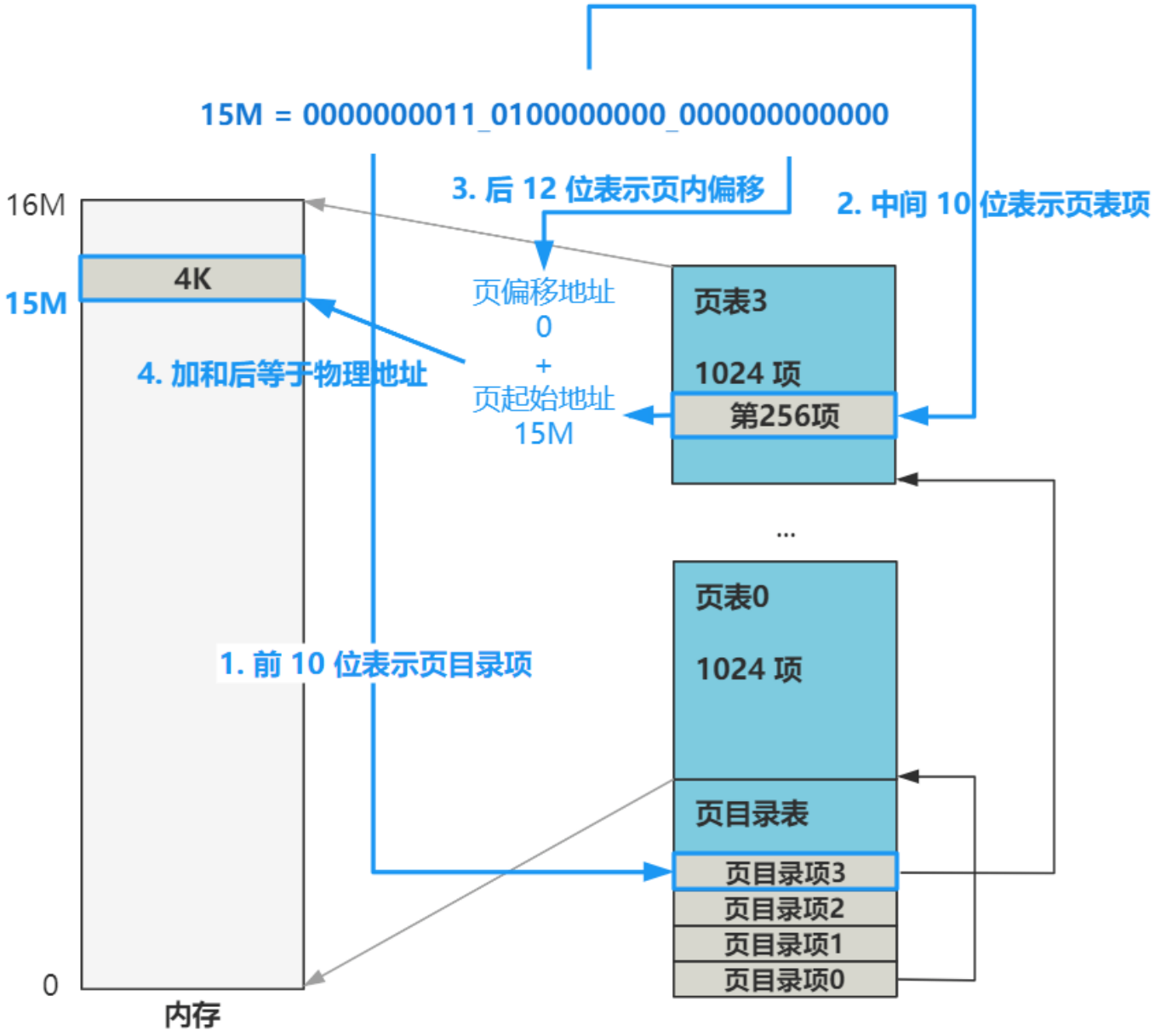
很简单，对照刚刚的页目录表与页表结构看。



前五行表示，页目录表的前 4 个页目录项，分别指向 4 个页表。比如页目录项中的第一项 **[eax]** 被赋值为 **pg0+7**，也就是 **0x00001007**，根据页目录项的格式，表示页表地址为 **0x1000**，页属性为 **0x07** 表示改页存在、用户可读写。

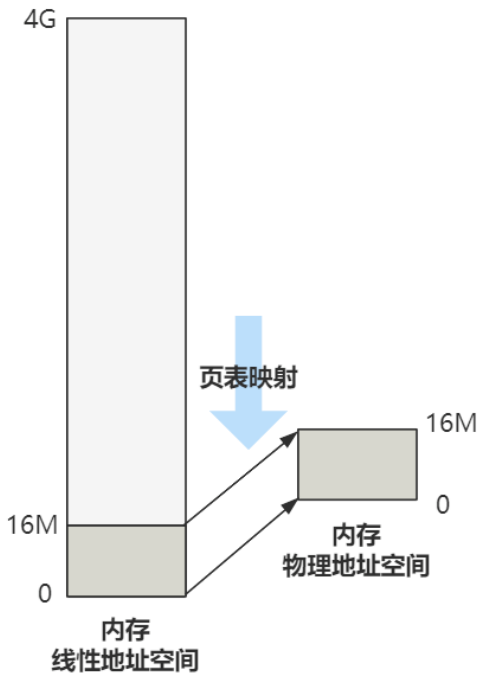
后面几行表示，填充 4 个页表的每一项，一共 **4*1024=4096** 项，依次映射到内存的前 16MB 空间。

画出图就是这个样子，其实刚刚的图就是。



看，最终的效果就是，经过这套分页机制，**线性地址将恰好和最终转换的物理地址一样。**

现在只有四个页目录项，也就是将前 16M 的线性地址空间，与 16M 的物理地址空间一一对应起来了。



好了，我知道你目前可能有点晕头转向，关于地址，我们已经出现了好多词了，包括**逻辑地址**、**线性地址**、**物理地址**，以及本文中没出现的，你可能在很多地方看到过的**虚拟地址**。

而这些地址后面加上空间两个字，似乎又成为了一个新词，比如**线性地址空间**、**物理地址空间**、**虚拟地址空间**等。

那就是时候展开一波讨论，将这块的内容梳理一番了，且听我说。

Intel 体系结构的**内存管理**可以分成两大部分，也就是标题中的两板斧，**分段**和**分页**。

分段机制在之前几回已经讨论过多次了，其目的是为了为每个程序或任务提供单独的代码段（cs）、数据段（ds）、栈段（ss），使其不会相互干扰。

分页机制是本回讲的内容，开机后分页机制默认是关闭状态，需要我们手动开启，并且设置好页目录表（PDE）和页表（PTE）。其目的在于可以按需使用物理内存，同时也可以在多任务时起到隔离的作用，这个在后面将多任务时将会有所体会。

在 Intel 的保护模式下，分段机制是没有开启和关闭一说的，它必须存在，而分页机制是可以选择开启或关闭的。所以如果有人和你说，它实现了一个没有分段机制的操作系统，那一定是个外行。

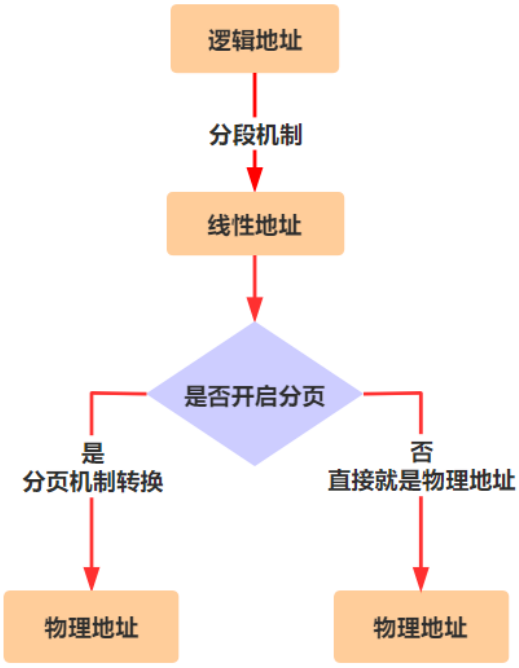
再说说那些地址：

逻辑地址：我们程序员写代码时给出的地址叫逻辑地址，其中包含段选择子和偏移地址两部分。

线性地址：通过分段机制，将逻辑地址转换后的地址，叫做线性地址。而这个线性地址是有个范围的，这个范围就叫做线性地址空间，32 位模式下，线性地址空间就是 4G。

物理地址：就是真正在内存中的地址，它也是有范围的，叫做物理地址空间。那这个范围的大小，就取决于你的内存有多大了。

虚拟地址：如果没有开启分页机制，那么线性地址就和物理地址是一一对应的，可以理解为相等。如果开启了分页机制，那么线性地址将被视为虚拟地址，这个虚拟地址将会通过分页机制的转换，最终转换成物理地址。



但实际上，我本人是不喜欢虚拟地址这个叫法的，因为它在 Intel 标准手册上出现的次数很少，我觉得知道逻辑地址、线性地址、物理地址这三个概念就够了，逻辑地址是程序员给出的，经过分段机制转换后变成线性地址，然后再经过分页机制转换后变成物理地址，就这么简单。

好了，我们终于把这些杂七杂八的，idt、gdt、页表都设置好了，并且也开启了保护模式，之后我们就要做好进入 main.c 的准备了，那里是个新世界！

不过进入 main.c 之前还差最后一哆嗦，就是 head.s 最后的代码，也就是本文开头的那段代码。

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

看到没，这里有个 push _main，把 main 函数的地址压栈了，那最终跳转到这个 main.c 里的 main 函数，一定和这个压栈有关。

压栈为什么和跳转到这里还能联系上呢？留作本文思考题，下一篇将揭秘这个过程，你会发现仍然简单得要死。

欲知后事如何，且听下回分解。

------- 本回扩展资料 -------

关于逻辑地址-线性地址-物理地址的转换，可以参考 Intel 手册：
Intel 3A Chapter 3 Protected-Mode Memory Management

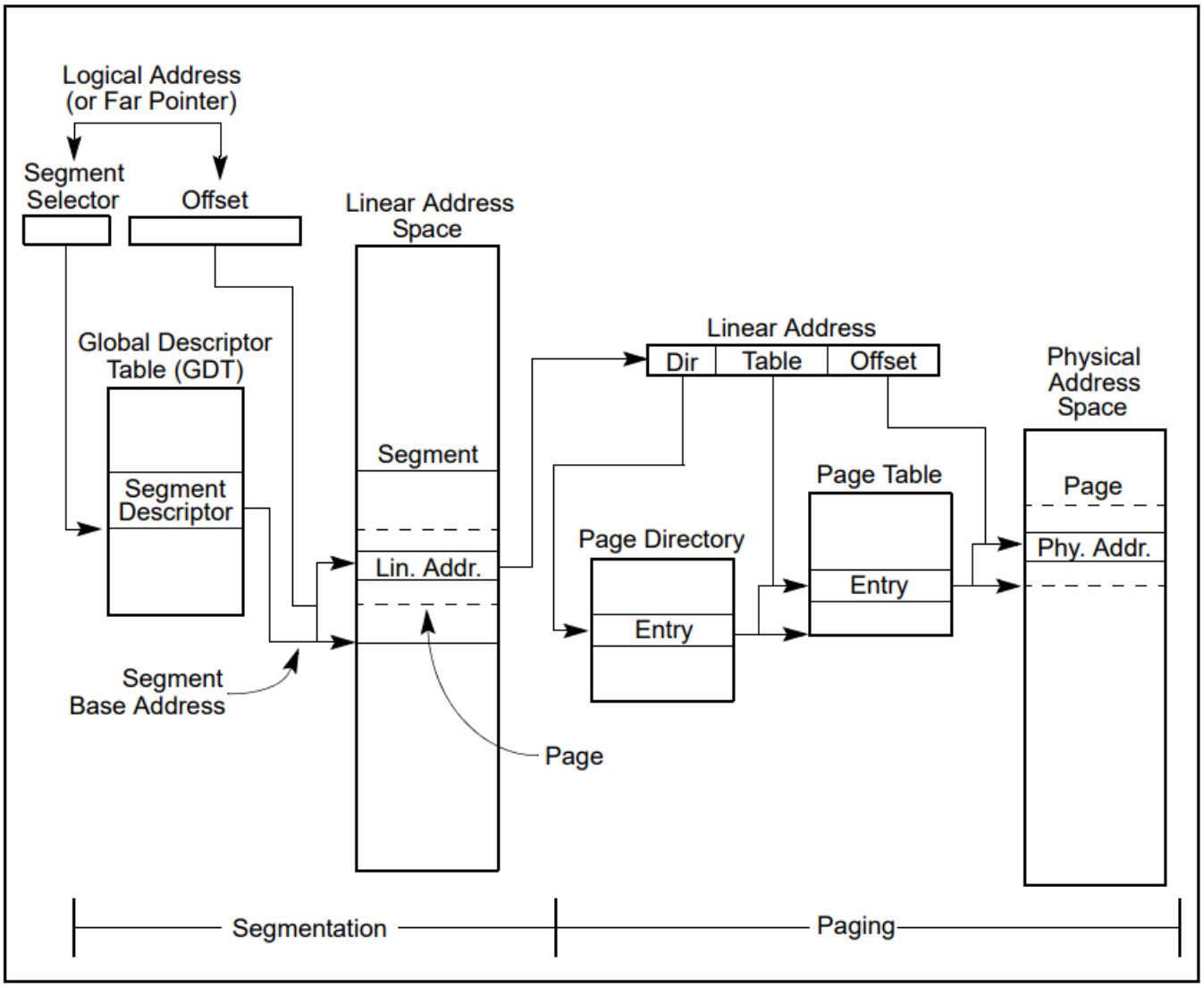


Figure 3-1. Segmentation and Paging

而有关这些地址的定义和说明，在本小节中也做了详细的说明，看这里的介绍是最权威也是最透彻的。相信我，它很简单。

3.1 MEMORY MANAGEMENT OVERVIEW

The memory management facilities of the IA-32 architecture are divided into two parts: segmentation and paging. Segmentation provides a mechanism of isolating individual code, data, and stack modules so that multiple programs (or tasks) can run on the same processor without interfering with one another. Paging provides a mechanism for implementing a conventional demand-paged, virtual-memory system where sections of a program's execution environment are mapped into physical memory as needed. Paging can also be used to provide isolation between multiple tasks. When operating in protected mode, some form of segmentation must be used. **There is no mode bit to disable segmentation.** The use of paging, however, is optional.

These two mechanisms (segmentation and paging) can be configured to support simple single-program (or single-task) systems, multitasking systems, or multiple-processor systems that used shared memory.

As shown in Figure 3-1, segmentation provides a mechanism for dividing the processor's addressable memory space (called the **linear address space**) into smaller protected address spaces called **segments**. Segments can be used to hold the code, data, and stack for a program or to hold system data structures (such as a TSS or LDT). If more than one program (or task) is running on a processor, each program can be assigned its own set of segments. The processor then enforces the boundaries between these segments and ensures that one program does not interfere with the execution of another program by writing into the other program's segments. The segmentation mechanism also allows typing of segments so that the operations that may be performed on a particular type of segment can be restricted.

All the segments in a system are contained in the processor's linear address space. To locate a byte in a particular segment, a **logical address** (also called a far pointer) must be provided. A logical address consists of a segment selector and an offset. The segment selector is a unique identifier for a segment. Among other things it provides an offset into a descriptor table (such as the global descriptor table, GDT) to a data structure called a segment descriptor. Each segment has a segment descriptor, which specifies the size of the segment, the access rights and privilege level for the segment, the segment type, and the location of the first byte of the segment in the linear address space (called the base address of the segment). The offset part of the logical address is added to the base address for the segment to locate a byte within the segment. The base address plus the offset thus forms a **linear address** in the processor's linear address space.

页目录表和页表的具体结构，可以看

Intel 3A Chapter 4.3 32-bit paging

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page directory[1]																				Ignored							PCD	PWT	Ignored			CR3
Bits 31:22 of address of 4MB page frame										Reserved (must be 0)						Bits 39:32 of address[2]			PAT	Ignored			G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 4MB page
Address of page table																				Ignored				0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored																															0	PDE: not present
Address of 4KB page frame																				Ignored			G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored																															0	PTE: not present

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

Table 4-5. Format of a 32-Bit Page-Directory Entry that References a Page Table

Bit Position(s)	Contents
0 (P)	Present; must be 1 to reference a page table
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-MByte region controlled by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-MByte region controlled by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9)
5 (A)	Accessed; indicates whether this entry has been used for linear-address translation (see Section 4.8)
6	Ignored
7 (PS)	If CR4.PSE = 1, must be 0 (otherwise, this entry maps a 4-MByte page; see Table 4-4); otherwise, ignored
11:8	Ignored
31:12	Physical address of 4-KByte aligned page table referenced by this entry

------- 关于本系列 -------

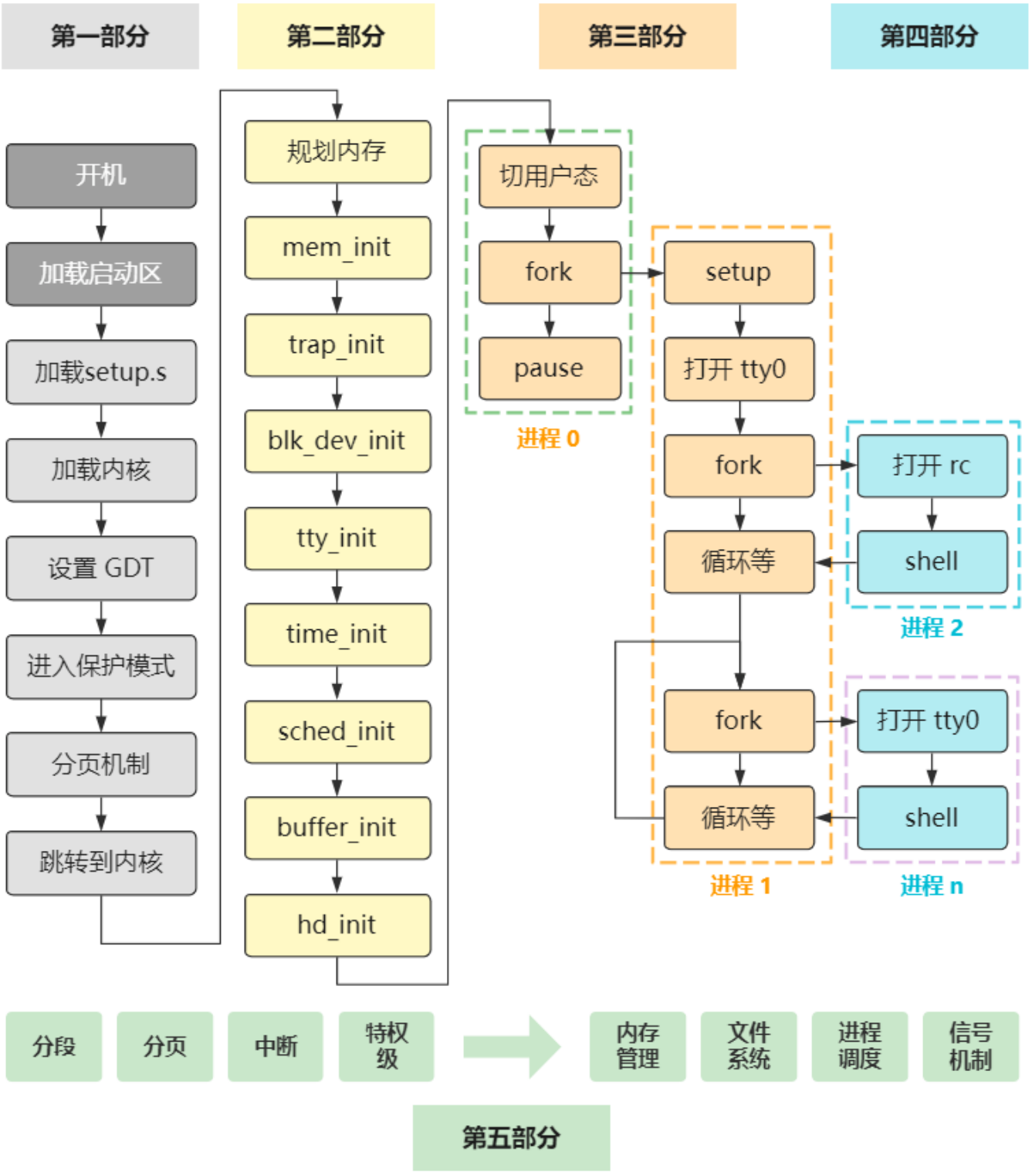
本系列的开篇词看这

闪客新系列！你管这破玩意叫操作系统源码

本系列的扩展资料看这（也可点击**阅读原文**），这里有很多有趣的资料、答疑、互动参与项目，持续更新中，希望有你的参与。

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

本系列全局视角



最后，祝大家都能追更到系列结束，只要你敢持续追更，并且把每一回的内容搞懂，我就敢让你在系列结束后说一句，我对 Linux 0.11 很熟悉。

另外，本系列**完全免费**，希望大家能多多传播给同样喜欢的人，同时给我的 GitHub 项目点个 star，就在**阅读原文**处，这些就足够让我坚持写下去了！我们下回见。

低并发编程
战略上藐视技术，战术上重视技术
175篇原创内容

Official Account

收录于合集 #操作系统源码 43

上一篇
第八回 | 烦死了又要重新设置一遍 idt 和 gdt

下一篇
第十回 | 进入 main 函数前的最后一跃！

Read more