第18回 | 大名鼎鼎的进程调度就是从这里开始的

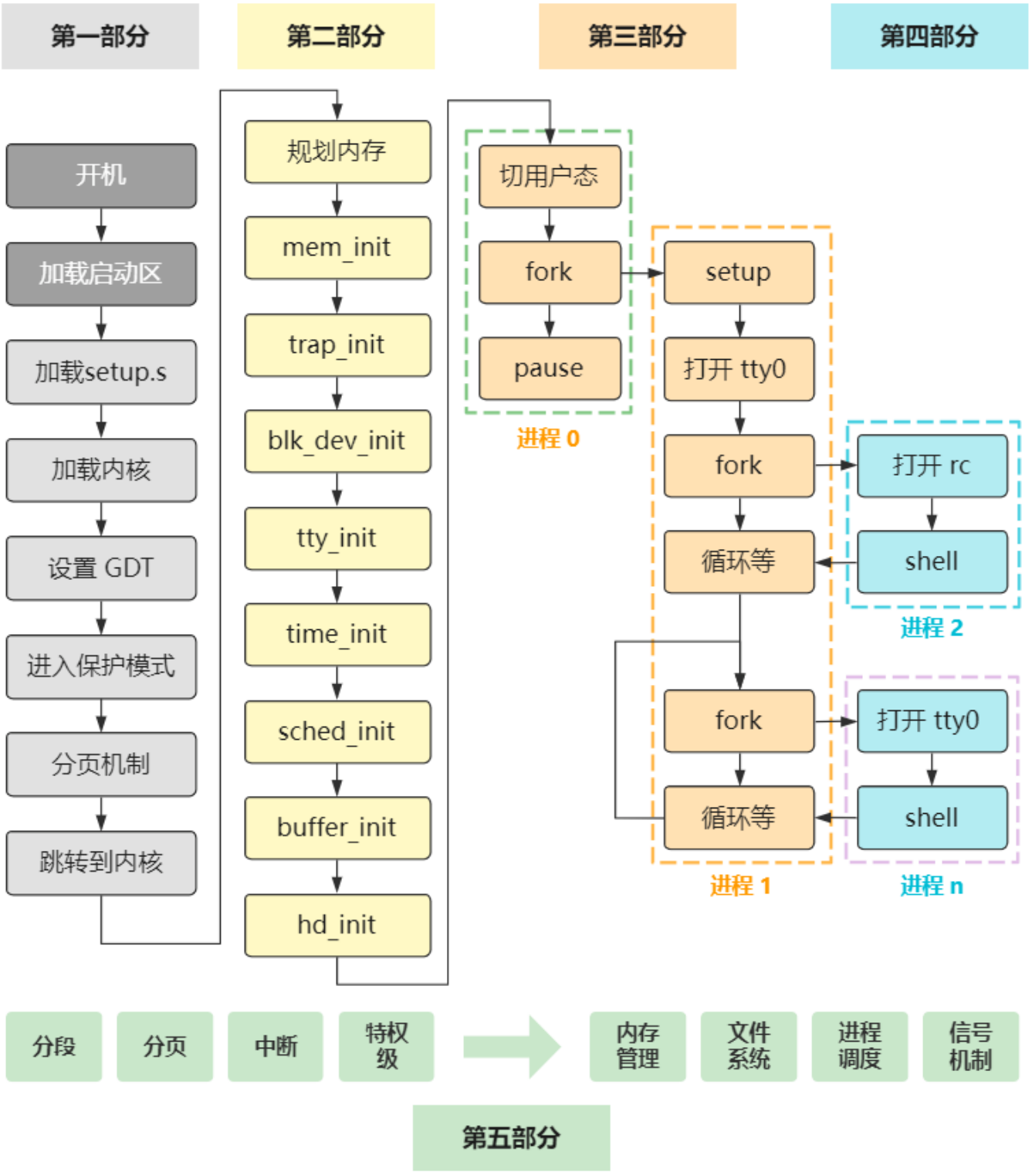
Original 闪客 低并发编程 2022-01-16 16:30

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

第一部分 进入内核前的苦力活

第二部分 大战前期的初始化工作

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，**time_init** 方法通过与 CMOS 端口进行读写交互，获取到了年月日时分秒等数据，并通过这些计算出了开机时间 **startup_time** 变量，是从 1970 年 1 月 1 日 0 时起到开机当时经过的秒数。

我们继续往下看，大名鼎鼎的进程调度初始化，**shed_init**。

```
void main(void) {
    ...
    mem_init(main_memory_start,memory_end);
    trap_init();
    blk_dev_init();
    chr_dev_init();
    tty_init();
    time_init();
    sched_init();
    buffer_init(buffer_memory_end);
    hd_init();
    floppy_init();

    sti();
    move_to_user_mode();
    if (!fork()) {init();}
    for(;;) pause();
}
```

这方法可了不起，因为它就是多进程的基石！

终于来到了兴奋的时刻，是不是很激动？不过先别激动，这里只是进程调度的初始化，也就是为进程调度所需要用到的数据结构做个准备，真正的进程调度还需要调度算法、时钟中断等机制的配合。

当然，对于理解操作系统，流程和数据结构最为重要了，而这一段作为整个流程的起点，以及建立数据结构的地方，就显得格外重要了。

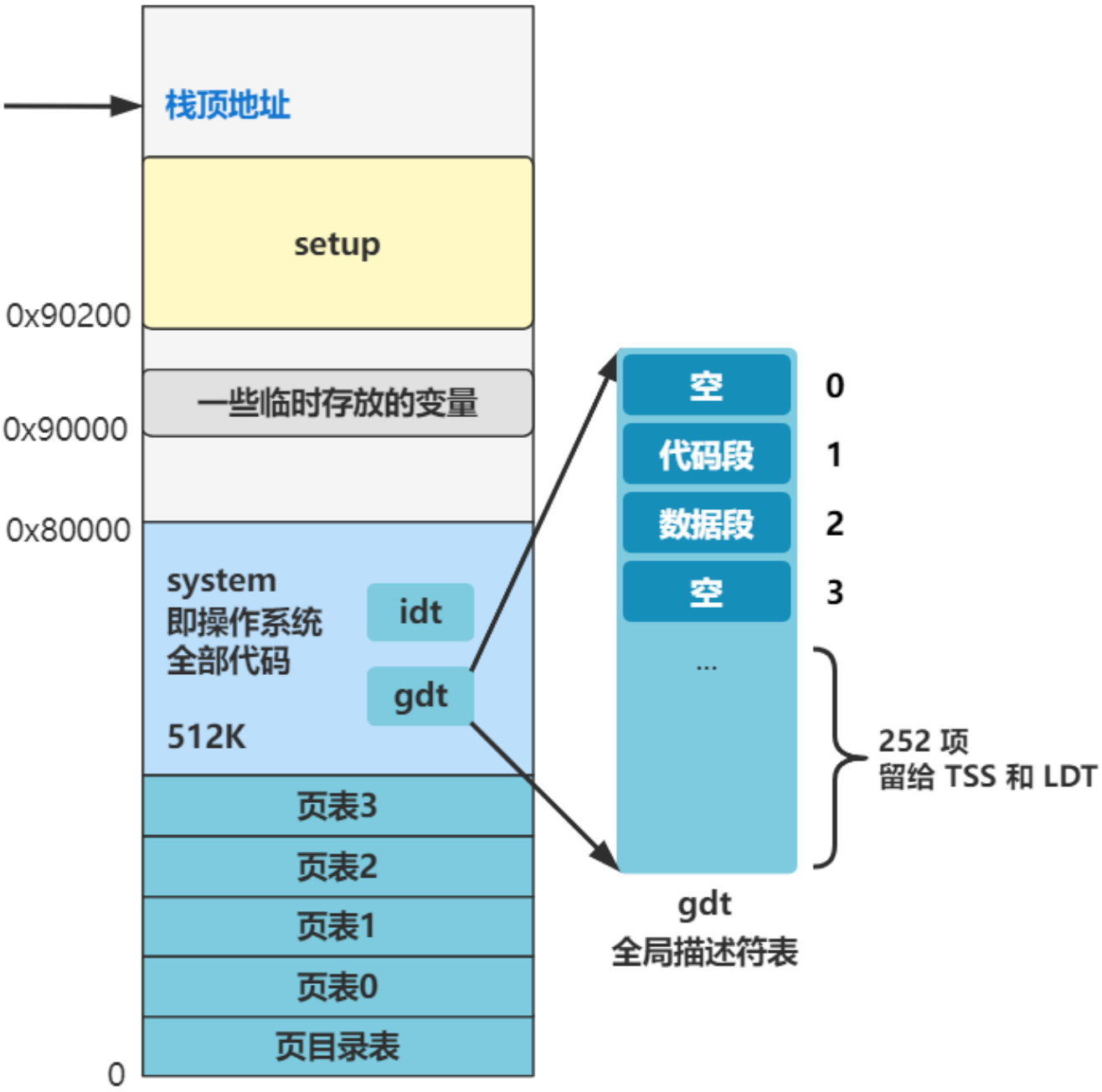
我们进入这个方法，一点点往后看。

```
void sched_init(void) {
    set_tss_desc(gdt+4, &(init_task.task.tss));
    set_ldt_desc(gdt+5, &(init_task.task.ldt));
    ...
}
```

两行代码初始化了下 **TSS** 和 **LDT**。

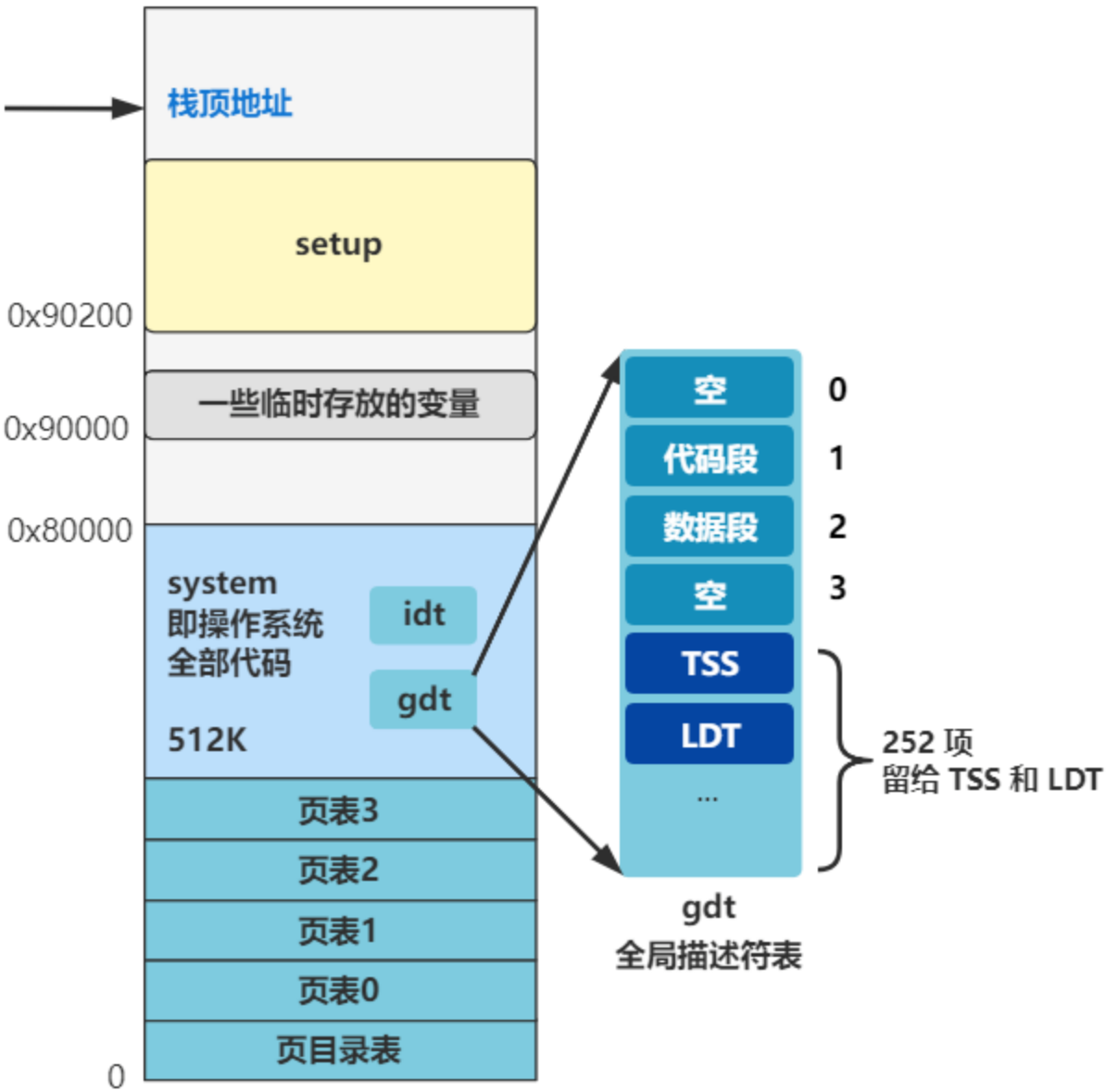
先别急问这俩结构是啥。还记得之前讲的全局描述符表 gdt 么？它在内存的这个位置，并且

被设置成了这个样子。



忘了的看一下第八回 | 烦死了又要重新设置一遍 idt 和 gdt，这就说明之前看似没用的细节有多重要了，大家一定要有耐心。

说回这两行代码，其实就是往后又加了两项，分别是 TSS 和 LDT。



好，那再说说这俩结构是干嘛的，不过本篇先简单理解，后面会详细讲到。

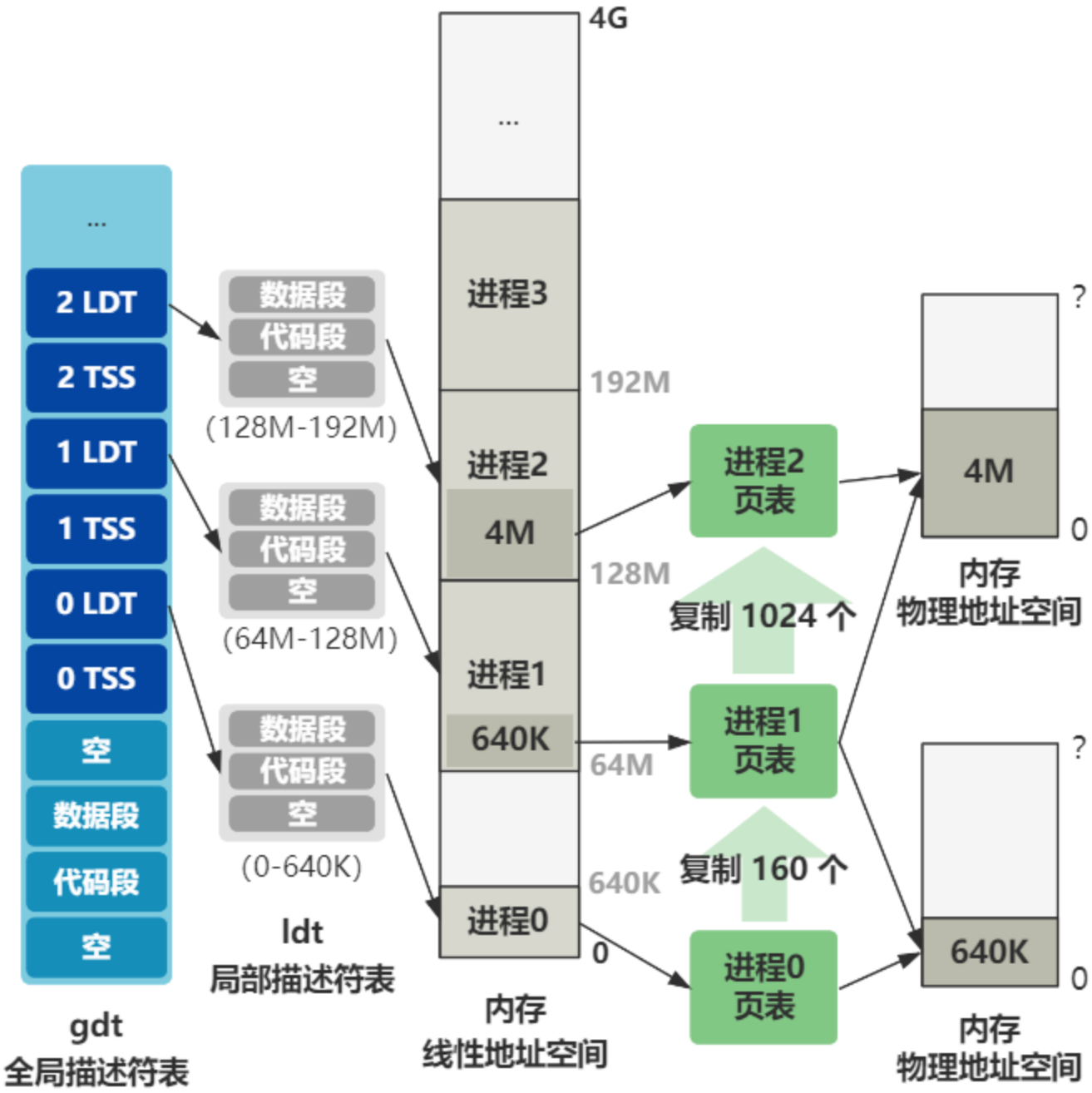
TSS 叫**任务状态段**，就是**保存和恢复进程的上下文的**，所谓上下文，其实就是各个寄存器的信息而已，这样进程切换的时候，才能做到保存和恢复上下文，继续执行。

由它的数据结构你应该可以看出点意思。

```
struct tss_struct{
    long back_link;
    long esp0;
    long ss0;
    long esp1;
    long ss1;
    long esp2;
    long ss2;
    long cr3;
    long eip;
    long eflags;
    long eax, ecx, edx, ebx;
    long esp;
    long ebp;
    long esi;
    long edi;
    long es;
    long cs;
    long ss;
    long ds;
    long fs;
    long gs;
    long ldt;
    long trace_bitmap;
    struct i387_struct i387;
};
```

而 LDT 叫**局部描述符表**，是与 GDT 全局描述符表相对应的，内核态的代码用 GDT 里的数据段和代码段，而用户进程的代码用每个用户进程自己的 LDT 里得数据段和代码段。

先不管它，我这里放一张超纲的图，你先找找感觉。

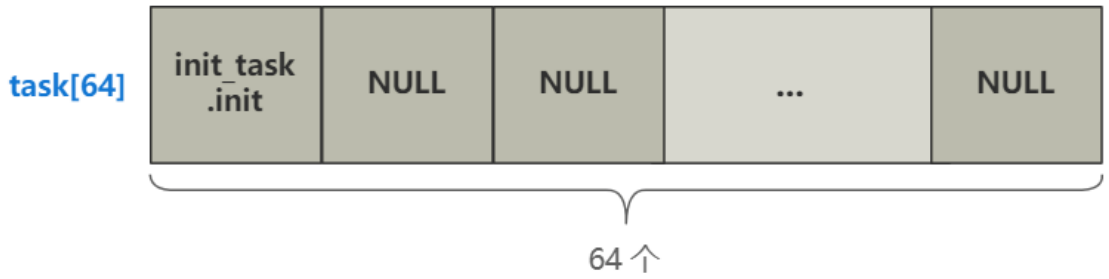

4G
...
进程3
192M
进程2
4M
128M
进程1
640K
64M
进程0
640K
0
2 LDT
2 TSS
1 LDT
1 TSS
0 LDT
0 TSS
空
数据段
代码段
空
gdt
全局描述符表
数据段
代码段
空
(128M-192M)
数据段
代码段
空
(64M-128M)
数据段
代码段
空
(0-640K)
ldt
局部描述符表
内存
线性地址空间
进程2
页表
进程1
页表
进程0
页表
复制 1024 个
复制 160 个
?
4M
0
内存
物理地址空间
?
640K
0
内存
物理地址空间

我们接着往下看。

```
struct desc_struct {
    unsigned long a,b;
}

struct task_struct * task[64] = {&(init_task.task), };

void sched_init(void) {
    ...
    int i;
    struct desc_struct * p;
        p = gdt+6;
    for(i=1;i<64;i++) {
        task[i] = NULL;
        p->a=p->b=0;
        p++;
        p->a=p->b=0;
        p++;
    }
    ...
}
```

这段代码有个循环，干了两件事。

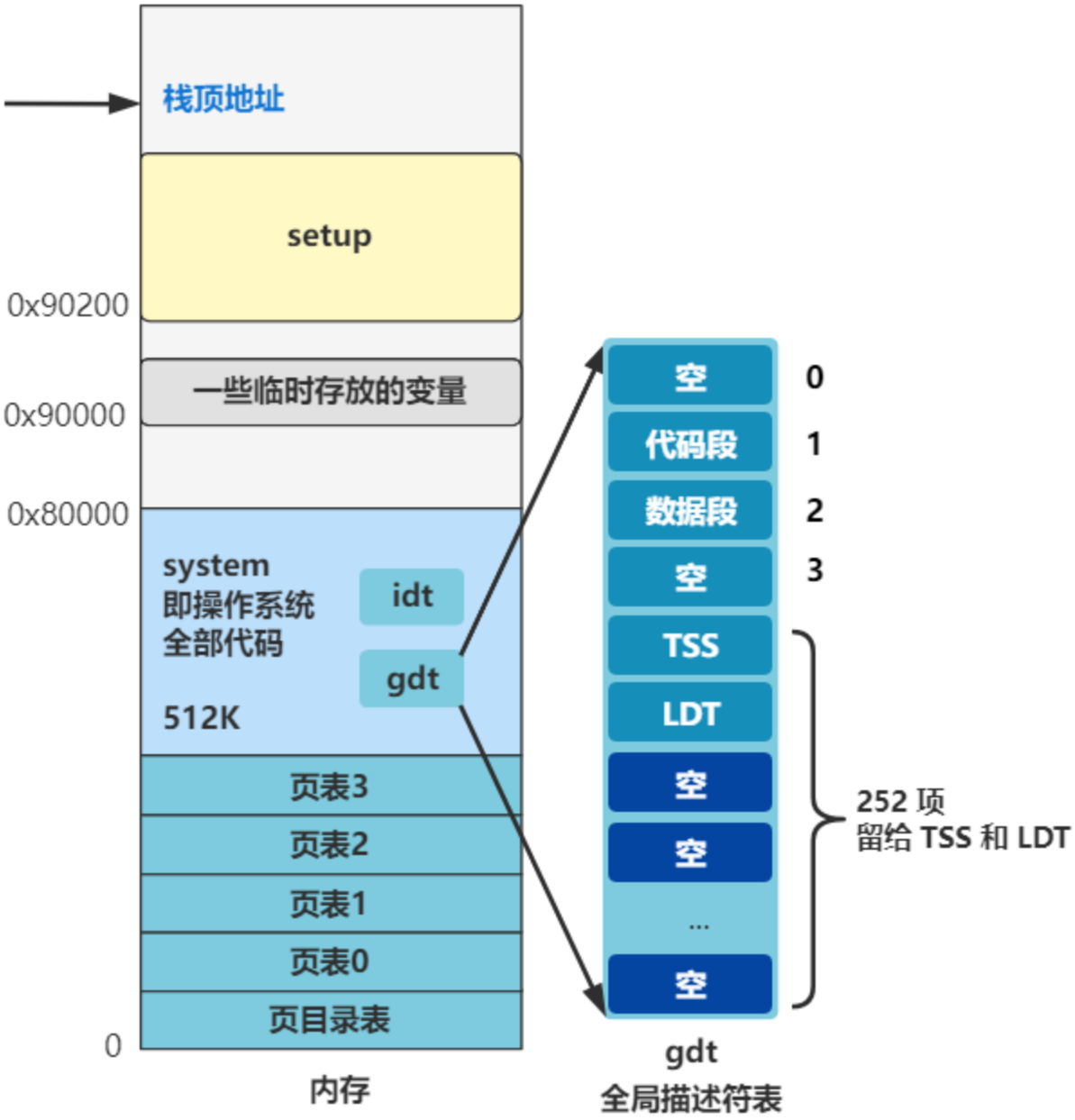
一个是给一个长度为 64，结构为 **task_struct** 的数组 task 附上初始值。



这个 task_struct 结构就是代表**每一个进程的信息**，这可是个相当相当重要的结构了，把它放在心里。

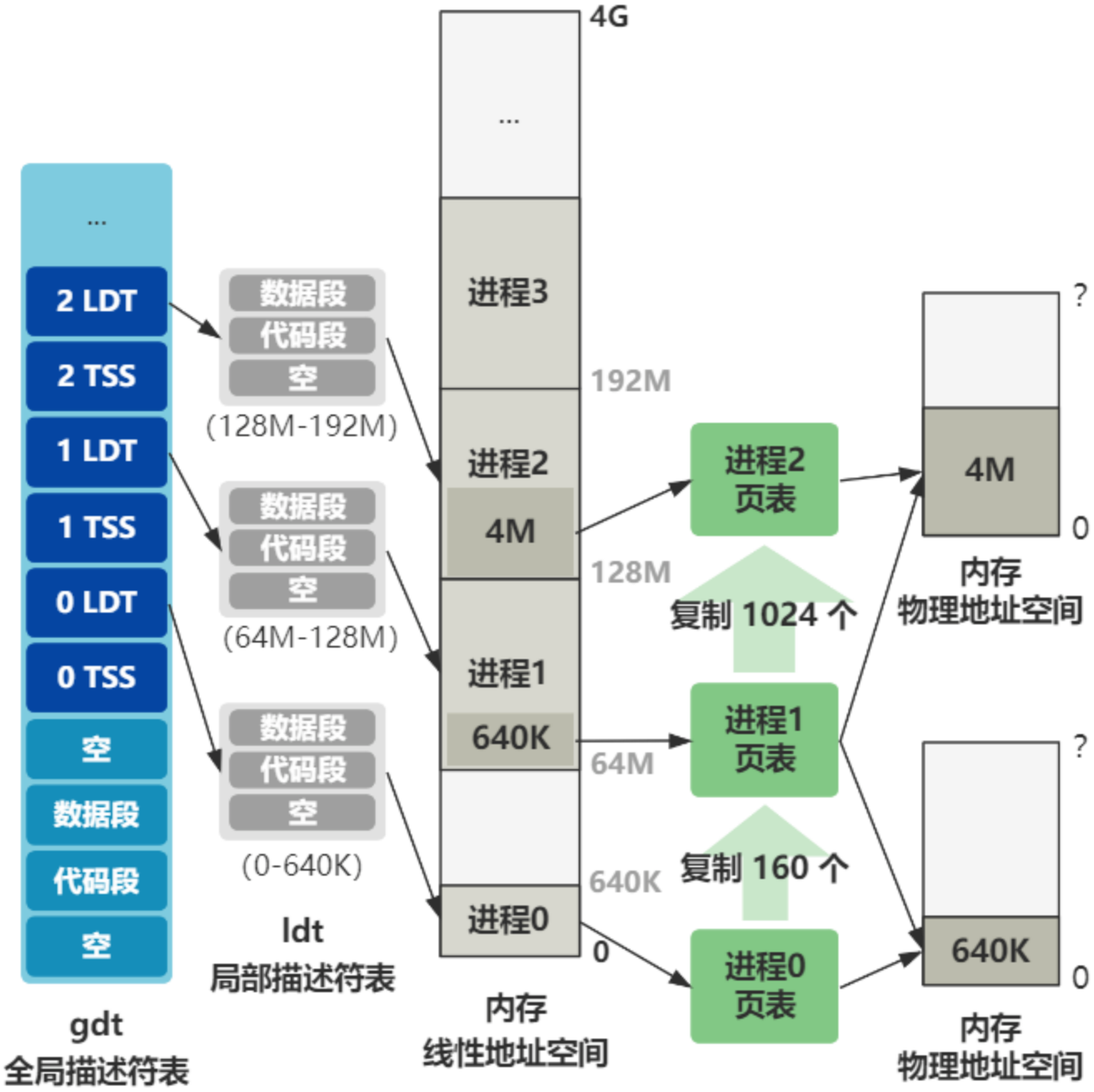
```
struct task_struct {
/* these are hardcoded - don't touch */
    long state; /* -1 unrunnable, 0 runnable, >0 stopped */
    long counter;
    long priority;
    long signal;
    struct sigaction sigaction[32];
    long blocked; /* bitmap of masked signals */
  /* various fields */
    int exit_code;
    unsigned long start_code,end_code,end_data,brk,start_stack;
    long pid,father,pgrp,session,leader;
    unsigned short uid,euid,suid;
    unsigned short gid,egid,sgid;
    long alarm;
    long utime,stime,cutime,cstime,start_time;
    unsigned short used_math;
  /* file system info */
    int tty;  /* -1 if no tty, so it must be signed */
    unsigned short umask;
    struct m_inode * pwd;
    struct m_inode * root;
    struct m_inode * executable;
    unsigned long close_on_exec;
    struct file * filp[NR_OPEN];
  /* ldt for this task 0 - zero 1 - cs 2 - ds&ss */
    struct desc_struct ldt[3];
  /* tss for this task */
    struct tss_struct tss;
};
```

这个循环做的另一件事，是给 gdt 剩下的位置填充上 0，也就是把剩下留给 TSS 和 LDT 的描述符都先附上空值。



往后展望一下的话，就是以后每创建一个新进程，就会在后面添加一组 TSS 和 LDT 表示这个进程的任务状态段以及局部描述符表信息。

还记得刚刚的超纲图吧，未来整个内存的规划就是这样的，不过你先不用理解得很细。



那为什么一开始就先有了一组 TSS 和 LDT 呢？现在也没创建进程呀。错了，现在虽然我们还没有建立起进程调度的机制，但我们正在运行的代码就是会作为**未来的一个进程的指令流。**

也就是当未来进程调度机制一建立起来，正在执行的代码就会化身成为**进程 0** 的代码。所以我们需要提前把这些未来会作为进程 0 的信息写好。

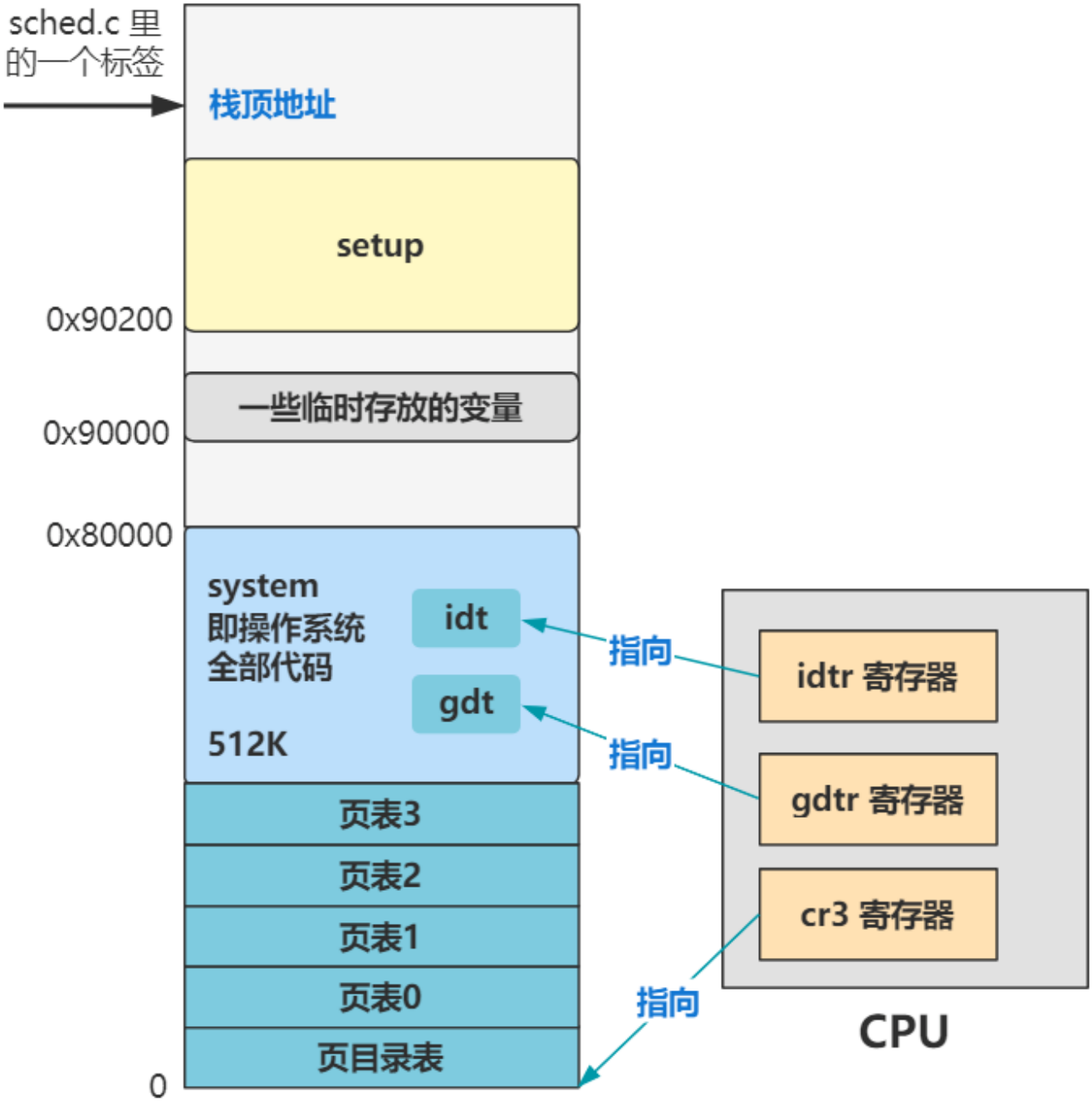
如果你觉得很疑惑，别急，等后面整个进程调度机制建立起来，并且让你亲眼看到进程 0 以及进程 1 的创建，以及它们后面因为进程调度机制而切换，你就明白这一切的意义了。

好，收回来，初始化了一组 TSS 和 LDT 后，再往下看两行。

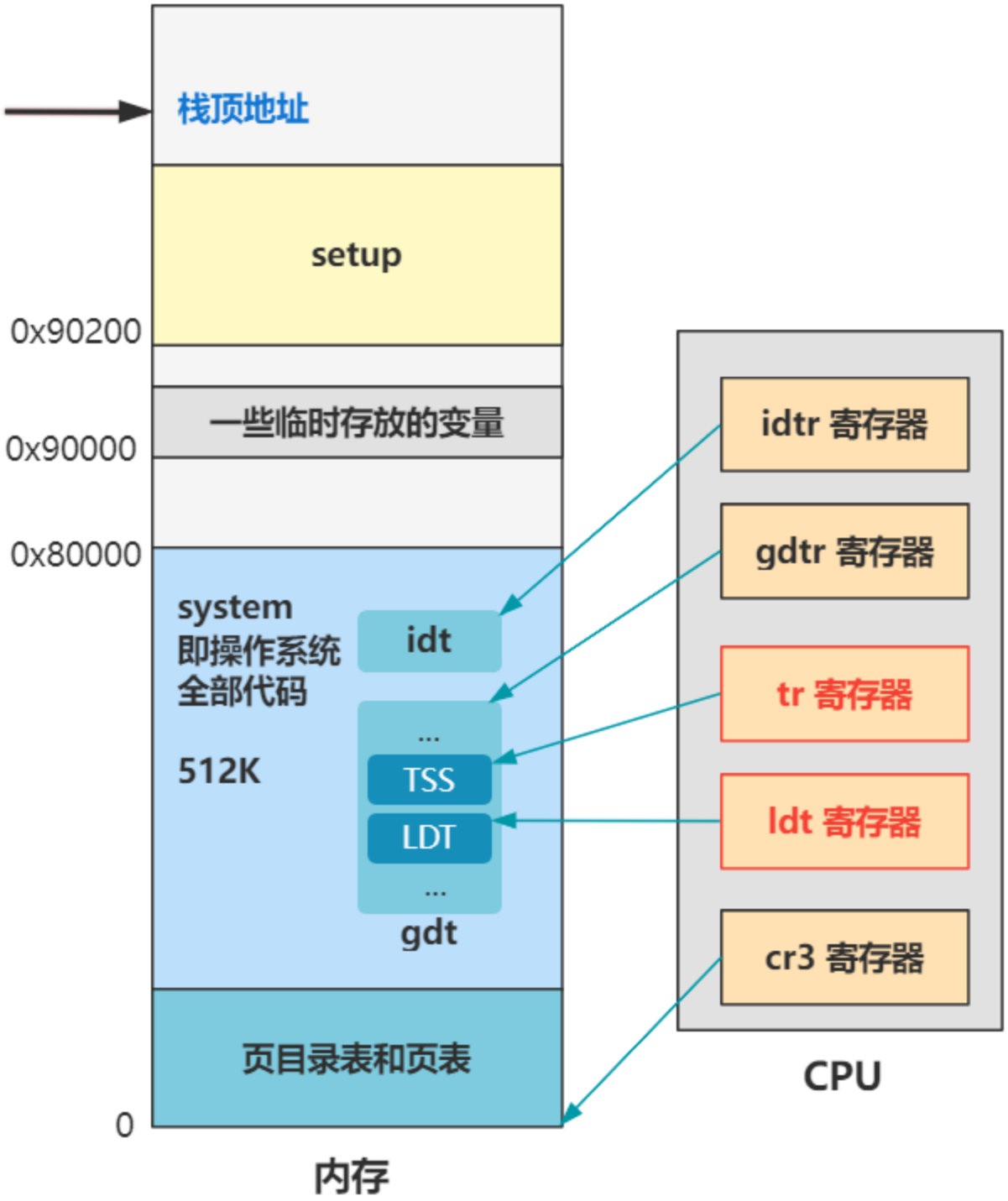
```
#define ltr(n) __asm__("ltr %%ax"::"a" (_TSS(n)))
#define lldt(n) __asm__("lldt %%ax"::"a" (_LDT(n)))

void sched_init(void) {
    ...
    ltr(0);
    lldt(0);
    ...
}
```

这又涉及到之前的知识咯。

还记得 **lidt** 和 **lgdt** 指令么？一个是给 idtr 寄存器赋值，以告诉 CPU 中断描述符表 idt 在内存的位置；一个是给 gdtr 寄存器赋值，以告诉 CPU 全局描述符表 gdt 在内存的位置。



那这两行和刚刚的类似，**ltr** 是给 **tr** 寄存器赋值，以告诉 CPU 任务状态段 TSS 在内存的位置；**lldt** 一个是给 **ldt** 寄存器赋值，以告诉 CPU 局部描述符 LDT 在内存的位置。



这样，CPU 之后就能通过 tr 寄存器找到当前进程的任务状态段信息，也就是上下文信息，以及通过 ldt 寄存器找到当前进程在用的局部描述符表信息。

我们继续看。

```
void sched_init(void) {
    ...
    outb_p(0x36,0x43);      /* binary, mode 3, LSB/MSB, ch 0 */
    outb_p(LATCH & 0xff , 0x40);    /* LSB */
    outb(LATCH >> 8 , 0x40);    /* MSB */
    set_intr_gate(0x20,&timer_interrupt);
    outb(inb_p(0x21)&~0x01,0x21);
    set_system_gate(0x80,&system_call);
    ...
}
```

四行端口读写代码，两行设置中断代码。

端口读写我们已经很熟悉了，就是 CPU 与外设交互的一种方式，之前讲硬盘读写以及 CMOS 读写时，已经接触过了。

而这次交互的外设是一个**可编程定时器**的芯片，这四行代码就开启了这个定时器，之后这个定时器变会**持续的、以一定频率的向 CPU 发出中断信号**。

而这段代码中设置的两个中断，第一个就是**时钟中断**，中断号为 **0x20**，中断处理程序为 **timer_interrupt**。那么每次定时器向 CPU 发出中断后，便会执行这个函数。

这个定时器的触发，以及时钟中断函数的设置，是操作系统主导进程调度的一个关键！没有他们这样的外部信号不断触发中断，操作系统就没有办法作为进程管理的主人，通过强制的手段收回进程的 CPU 执行权限。

第二个设置的中断叫系统调用 **system_call**，中断号是 **0x80**，这个中断又是个非常非常非常非常非常非常非常重要的中断，所有用户态程序想要调用内核提供的方法，都需要基于这个系统调用来进行。

比如 Java 程序员写一个 read，底层会执行汇编指令 **int 0x80**，这就会触发系统调用这个中断，最终调用到 Linux 里的 sys_read 方法。

这个过程之后会重点讲述，现在只需要知道，在这个地方，偷偷把这个极为重要的中断，设置好了。

所以你看这一章的内容，偷偷设置了影响进程和影响用户程序调用系统方法的两个重量级中断处理函数，不简单呀~

到目前为止，中断已经设置了不少了，我们现在看看所设置好的中断有哪些。

中断号	**中断处理函数**
0 ~ 0x10	trap_init 里设置的一堆
0x20	timer_interrupt
0x21	keyboard_interrupt
0x80	system_call

其中 **0-0x10** 这 17 个中断是 trap_init 里初始化设置的，是一些基本的中断，比如除零异常等。这个在 第14回 中断初始化 trap_init 有讲到。

之后，在控制台初始化 con_init 里，我们又设置了 **0x21** 键盘中断，这样按下键盘就有反应了。这个在 第16回 控制台初始化 tty_init 有讲到。

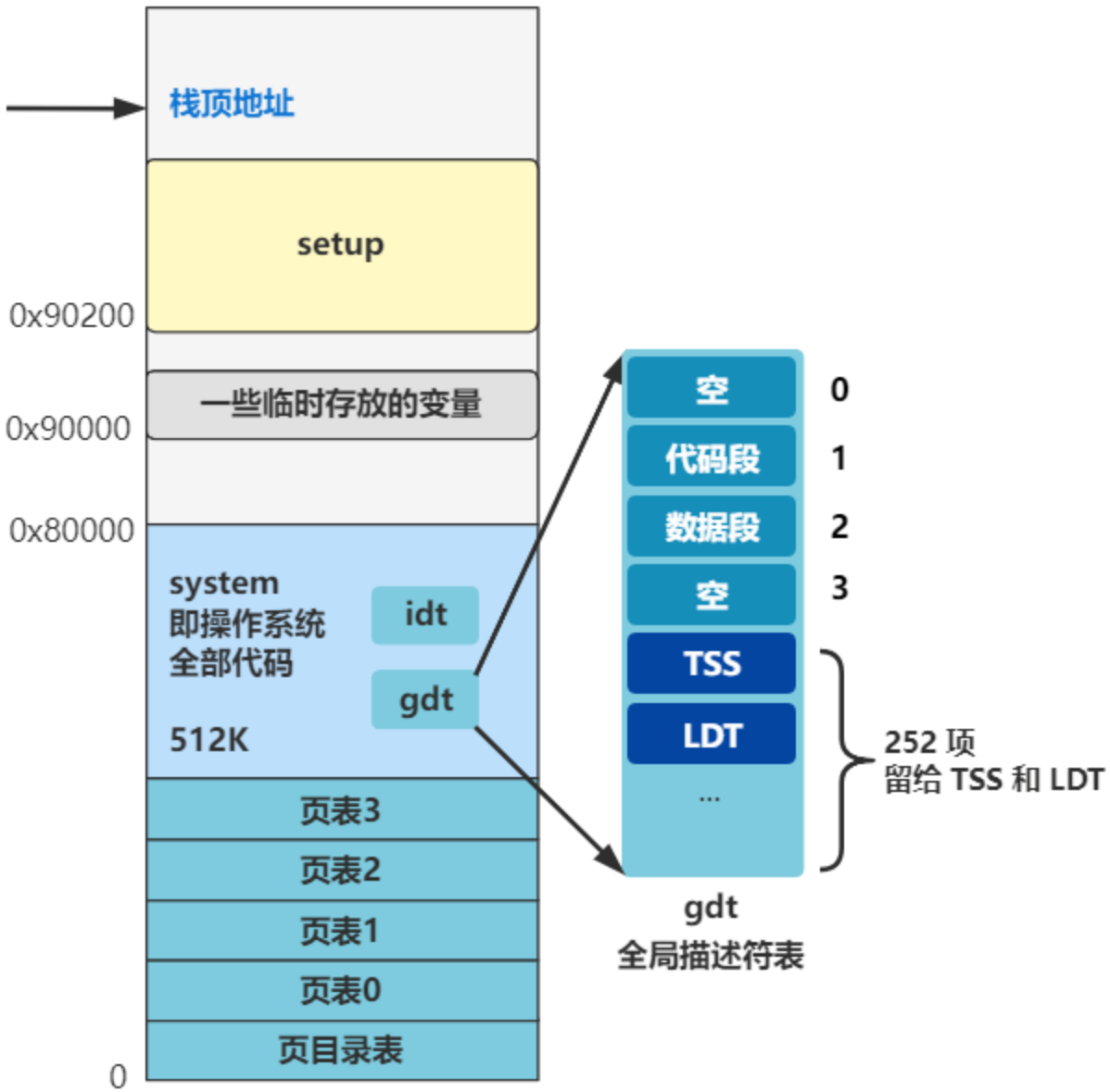
现在，我们又设置了 **0x20** 时钟中断，并且开启定时器。最后又偷偷设置了一个极为重要的 **0x80** 系统调用中断。

找到些感觉没，有没有越来越发现，操作系统有点靠中断驱动的意思，各个模块不断初始化各种中断处理函数，并且开启指定的外设开关，让操作系统自己慢慢“活”了起来，逐渐通过中断忙碌于各种事情中，无法自拔。

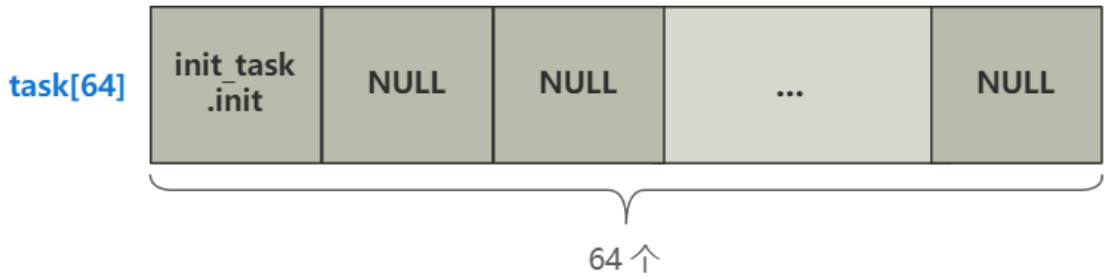
恭喜你，我们已经逐渐在接近操作系统的本质了。

回顾一下我们今天干了什么，就三件事。

第一，我们往全局描述符表写了两个结构，TSS 和 LDT，作为未来进程 0 的任务状态段和局部描述符表信息。



第二，我们初始化了一个结构为 task_struct 的数组，未来这里会存放所有进程的信息，并且我们给数组的第一个位置附上了 init_task.init 这个具体值，也是作为未来进程 0 的信息。



第三，设置了时钟中断 0x20 和系统调用 0x80，一个作为进程调度的起点，一个作为用户程序调用操作系统功能的桥梁，非常之重要。

后面，我们将会逐渐看到，这些重要的事情，是如何紧密且精妙地结合在一起，发挥出奇妙的作用。

欲知后事如何，且听下回分解。

如果觉得还行，给个 star 谢谢。

（点击阅读原文即可进入 Github 页）

------- 关于本系列 -------

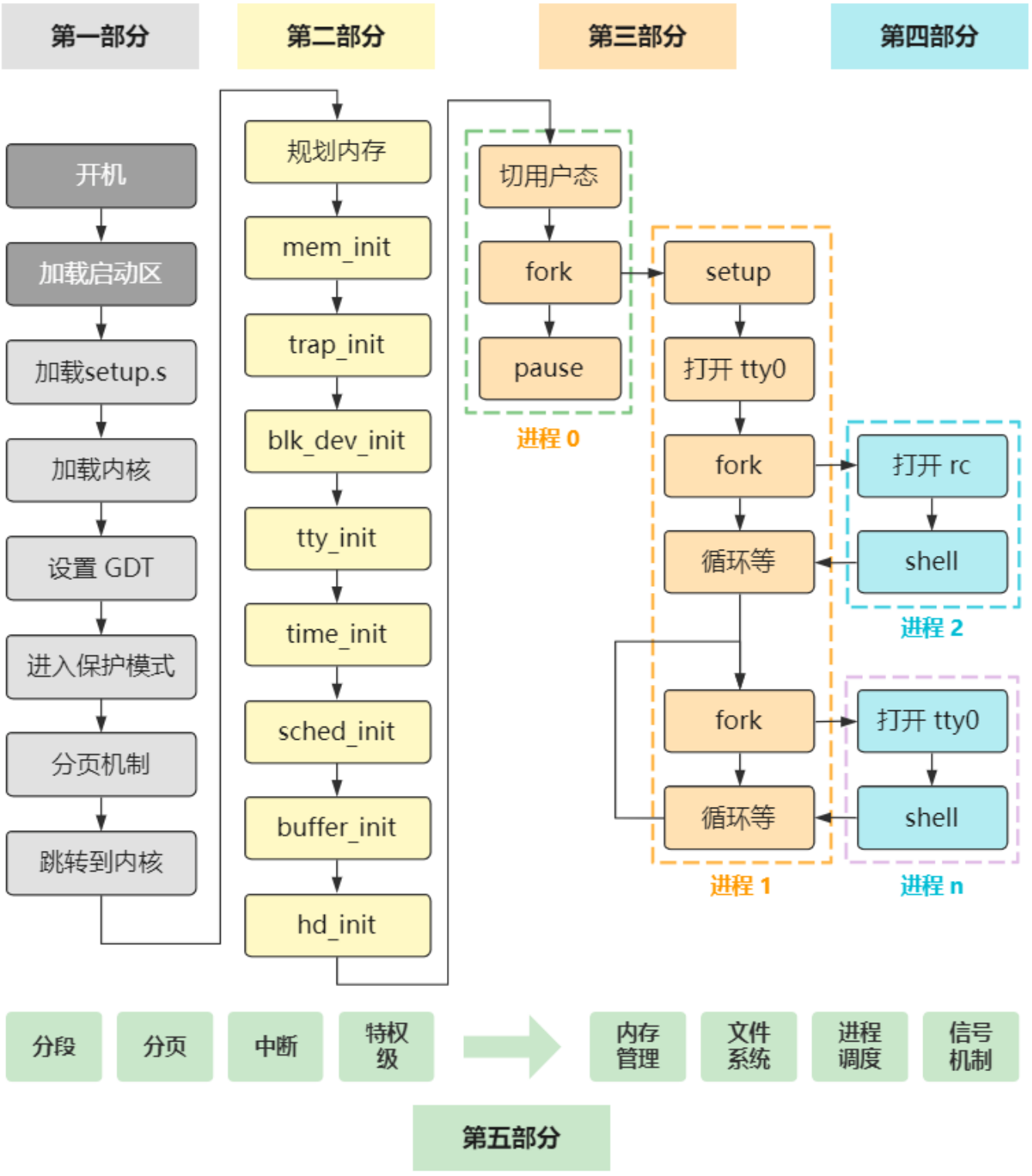
本系列的开篇词看这

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。

公众号更新系列文章不易，阅读量越来越低，希望大家多多传播，不方便的话点个小小的**赞**我也会很开心，谢谢大家咯。

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17回 | 原来操作系统获取时间的方式也这么 low

下一篇

第19回 | 操作系统就是用这两个面试常考的结构管理的缓冲区

Read more