一个新进程的诞生（七）透过 fork 来看进程的内存规划

Original 闪客 低并发编程 2022-03-09 16:30

收录于合集
#操作系统源码 43 #一个新进程的诞生 8

本系列作为 你管这破玩意叫操作系统源码 的第三大部分，讲述了操作系统第一个进程从无到有的诞生过程，这一部分你将看到内核态与用户态的转换、进程调度的上帝视角、系统调用的全链路、fork 函数的深度剖析。

不要听到这些陌生的名词就害怕，跟着我一点一点了解他们的全貌，你会发现，这些概念竟然如此活灵活现，如此顺其自然且合理地出现在操作系统的启动过程中。

本篇章作为一个全新的篇章，需要前置篇章的知识体系支撑。

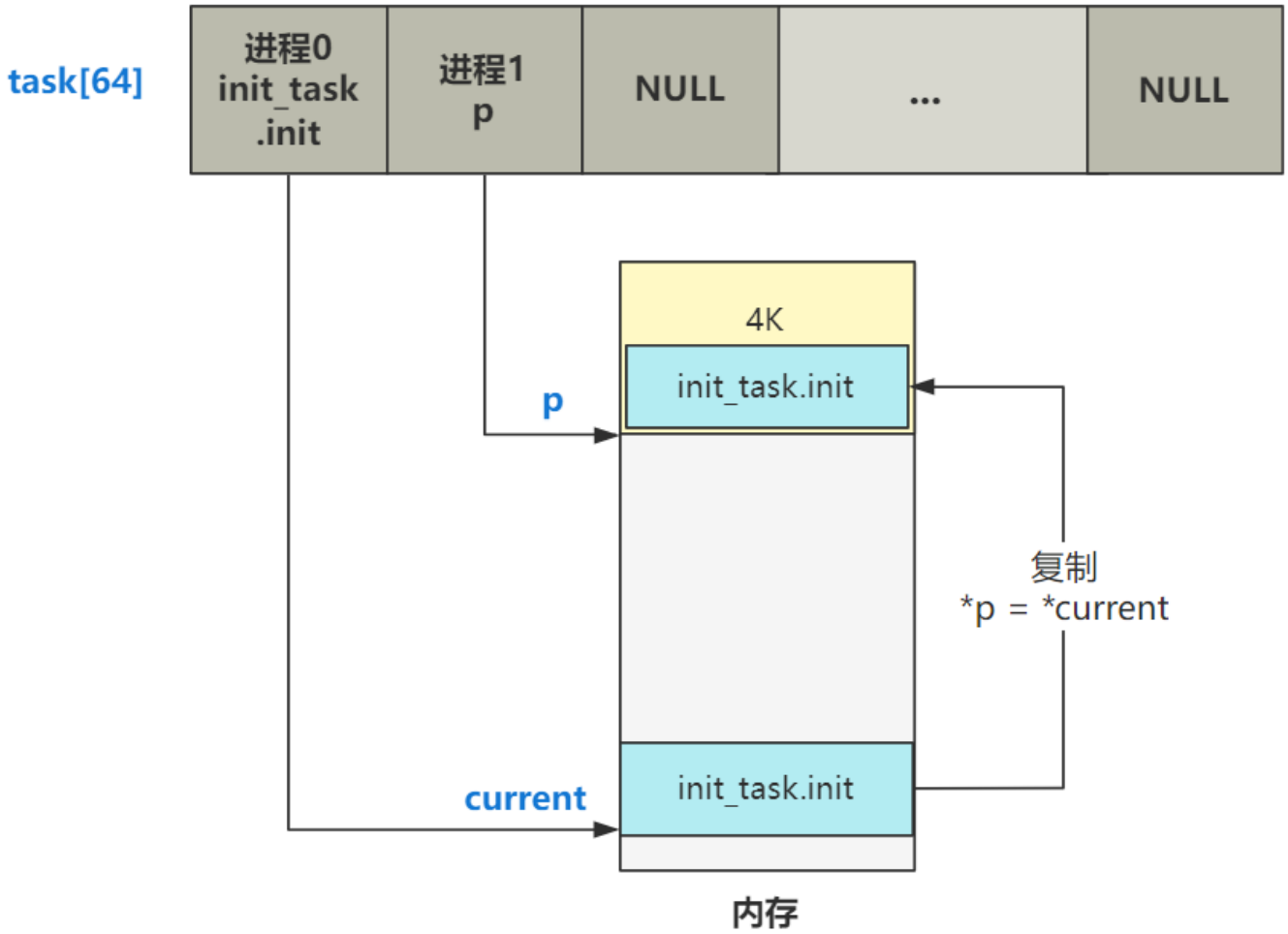
第一部分 进入内核前的苦力活
第二部分 大战前期的初始化工作

当然，没读过的也问题不大，我都会在文章里做说明，如果你觉得有困惑，就去我告诉你的相应章节回顾就好了，放宽心。

------- 第三部分目录 -------

------- 正文开始 -------

书接上回，上回书咱们说到，**fork** 函数为新的进程（进程 1）申请了槽位，并把全部 **task_struct** 结构的值都从进程零复制了过来。



之后，覆盖了新进程自己的基本信息，包括元信息和 tss 里的寄存器信息。

```
int copy_process(int nr, ...) {
    ...
    p->state = TASK_UNINTERRUPTIBLE;
    p->pid = last_pid;
    p->counter = p->priority;
    ..
    p->tss.edx = edx;
    p->tss.ebx = ebx;
    p->tss.esp = esp;
    ...
}
```

这可以说将 fork 函数的一半都讲完了，那我们今天展开讲讲另一半，也就是 **copy_mem** 函数。

```
int copy_process(int nr, ...) {
    ...
    copy_mem(nr,p);
    ...
}
```

这将会决定进程之间的内存规划问题，十分精彩，我们开始吧。

整个函数不长，我们还是试着先直译一下。

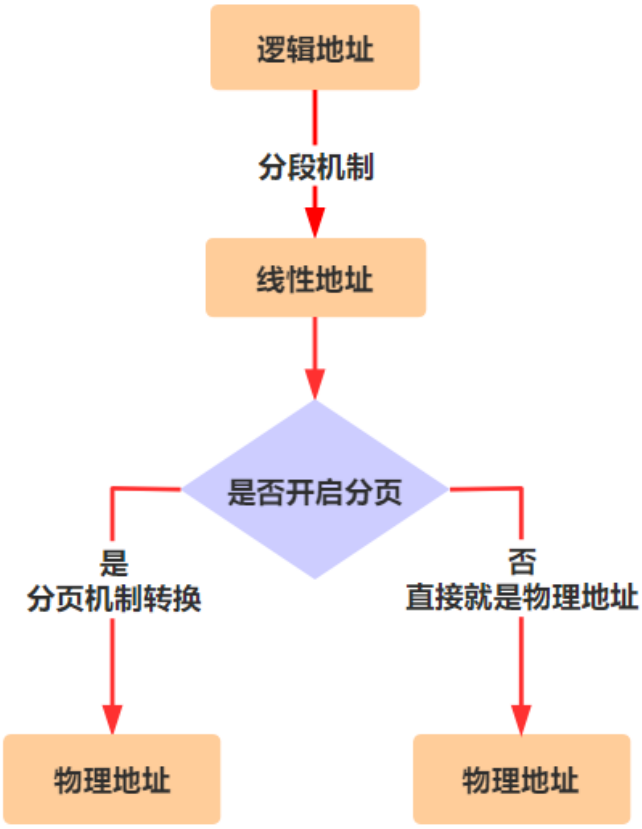
```
int copy_mem(int nr,struct task_struct * p) {
    // 局部描述符表 LDT 赋值
    unsigned long old_data_base,new_data_base,data_limit;
    unsigned long old_code_base,new_code_base,code_limit;
    code_limit = get_limit(0x0f);
    data_limit = get_limit(0x17);
    new_code_base = nr * 0x4000000;
    new_data_base = nr * 0x4000000;
    set_base(p->ldt[1],new_code_base);
    set_base(p->ldt[2],new_data_base);
    // 拷贝页表
    old_code_base = get_base(current->ldt[1]);
    old_data_base = get_base(current->ldt[2]);
    copy_page_tables(old_data_base,new_data_base,data_limit);
    return 0;
}
```

看，其实就是**新进程 LDT 表项的赋值，以及页表的拷贝。**

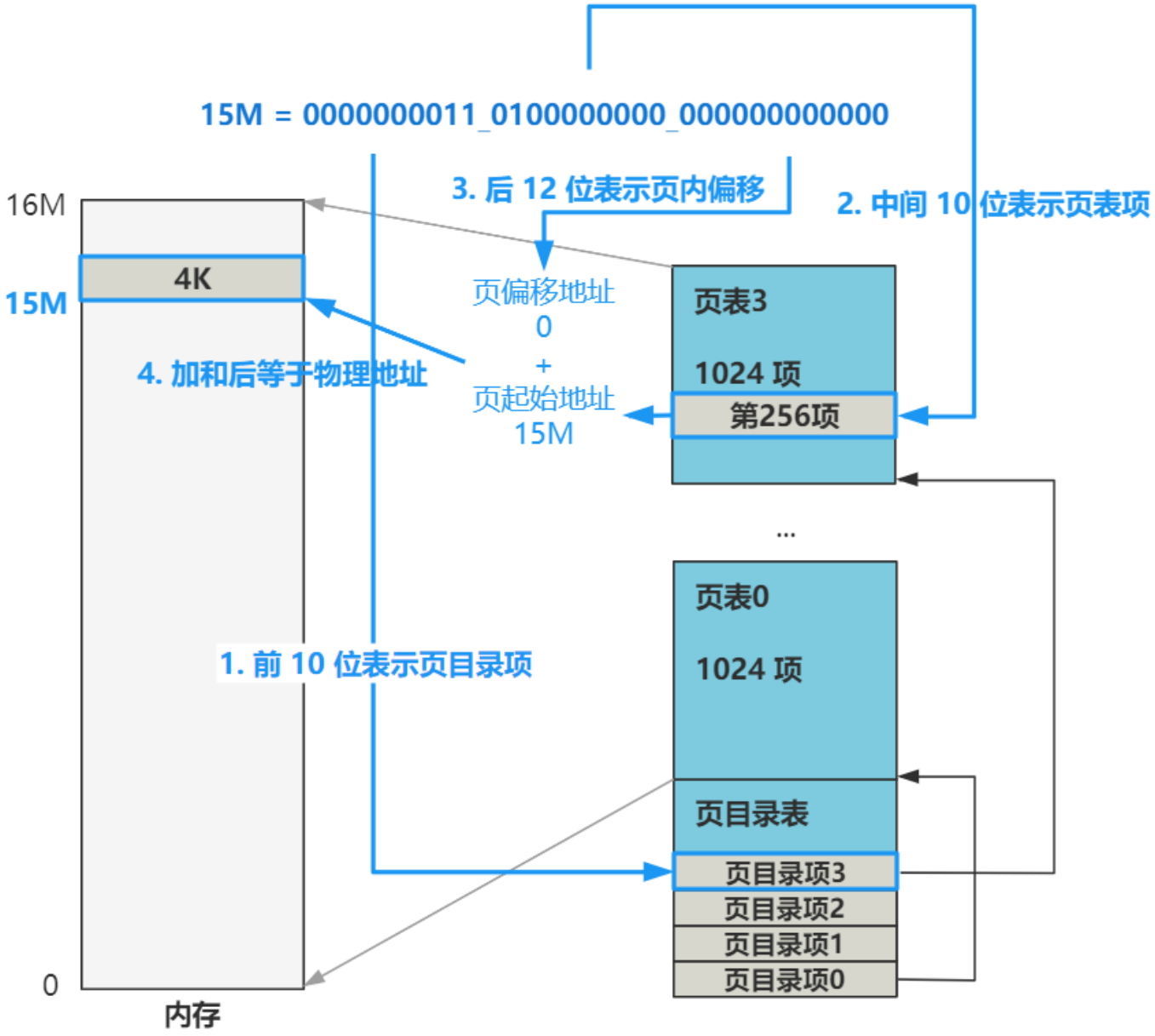
LDT 的赋值

那我们先看 LDT 表项的赋值，要说明白这个赋值的意义，得先回忆一下我们在 第九回 | Intel 内存管理两板斧：分段与分页 刚设置完页表时说过的问题。

程序员给出的逻辑地址最终转化为物理地址要经过这几步骤。

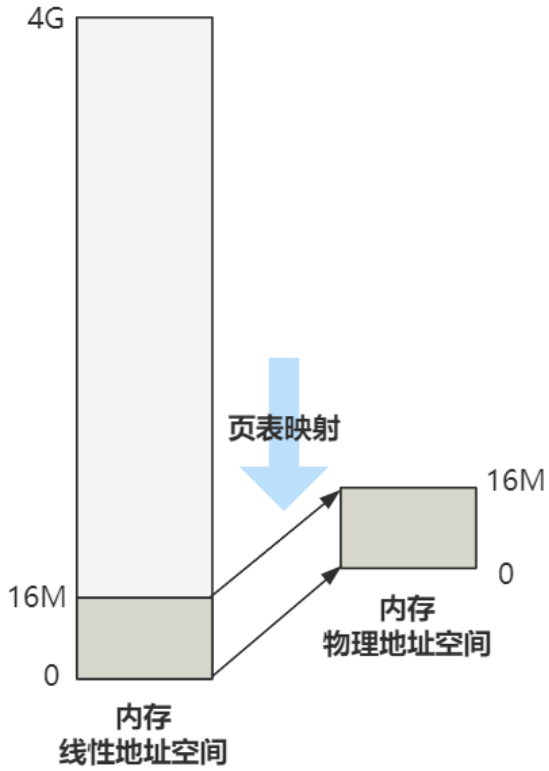


而我们已经开启了分页，那么分页机制的具体转化是这样的。

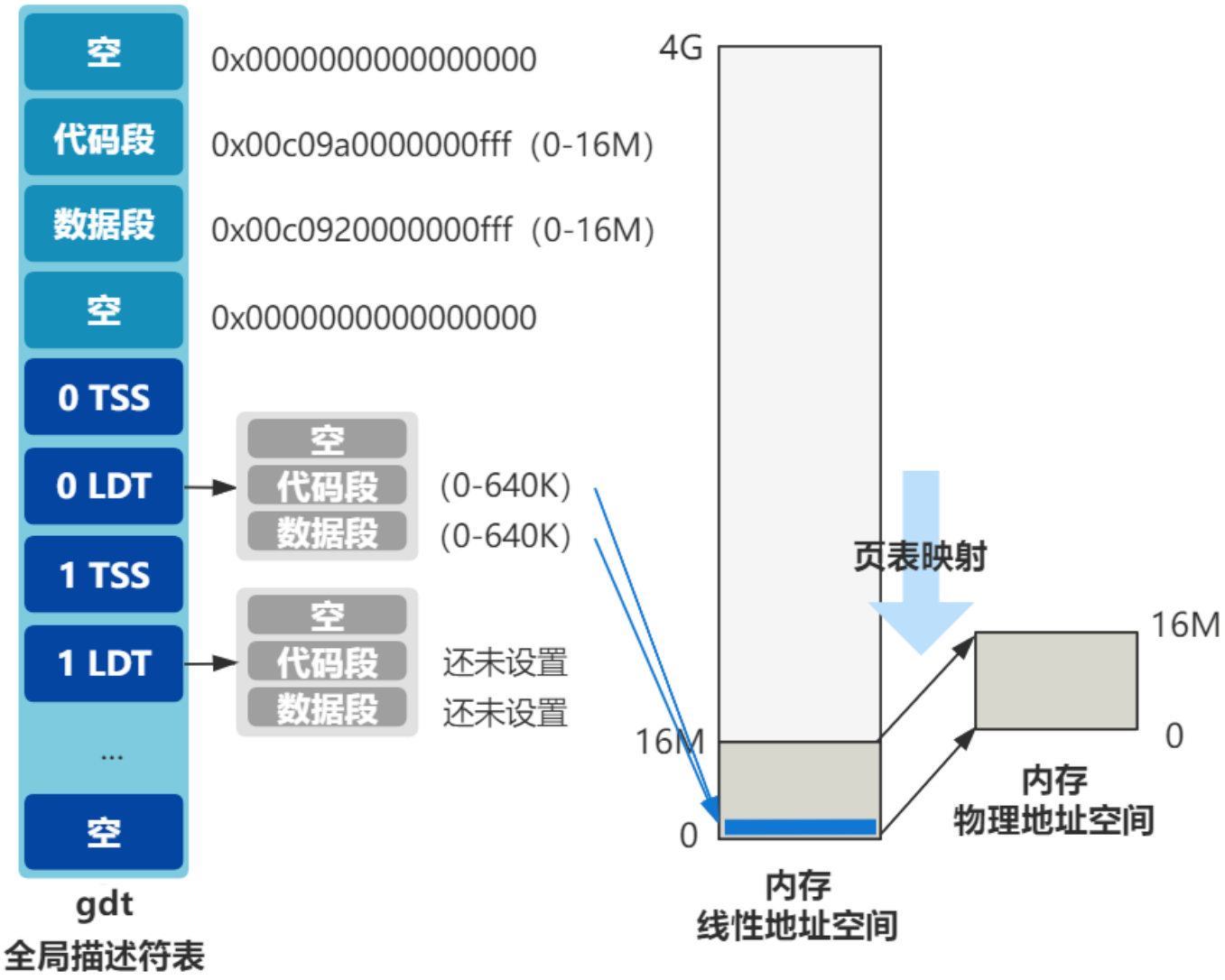


因为有了页表的存在，所以多了**线性地址空间**的概念，即经过分段机制转化后，分页机制转化前的地址。

不考虑段限长的话，32 位的 CPU 线性地址空间应为 4G。现在只有四个页目录表，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。



把这个图和全局描述符表 GDT 联系起来，这个线性地址空间，就是经过分段机制（段可能是 GDT 也可能是 LDT）后的地址，是这样对应的。



我们给进程 0 准备的 LDT 的代码段和数据段，段基址都是 0，段限长是 640K。给进程 1，也就是我们现在正在 fork 的这个进程，其代码段和数据段还没有设置。

所以第一步，**局部描述符表 LDT 的赋值**，就是给上图中那两个还未设置的代码段和数据段赋值。

其中**段限长**，就是取自进程 0 设置好的段限长，也就是 640K。

```
int copy_mem(int nr,struct task_struct * p) {
    ...
    code_limit = get_limit(0x0f);
    data_limit = get_limit(0x17);
    ...
}
```

而**段基址**有点意思，是取决于当前是几号进程，也就是 nr 的值。

```
int copy_mem(int nr,struct task_struct * p) {
    ...
    new_code_base = nr * 0x4000000;
    new_data_base = nr * 0x4000000;
    ...
}
```

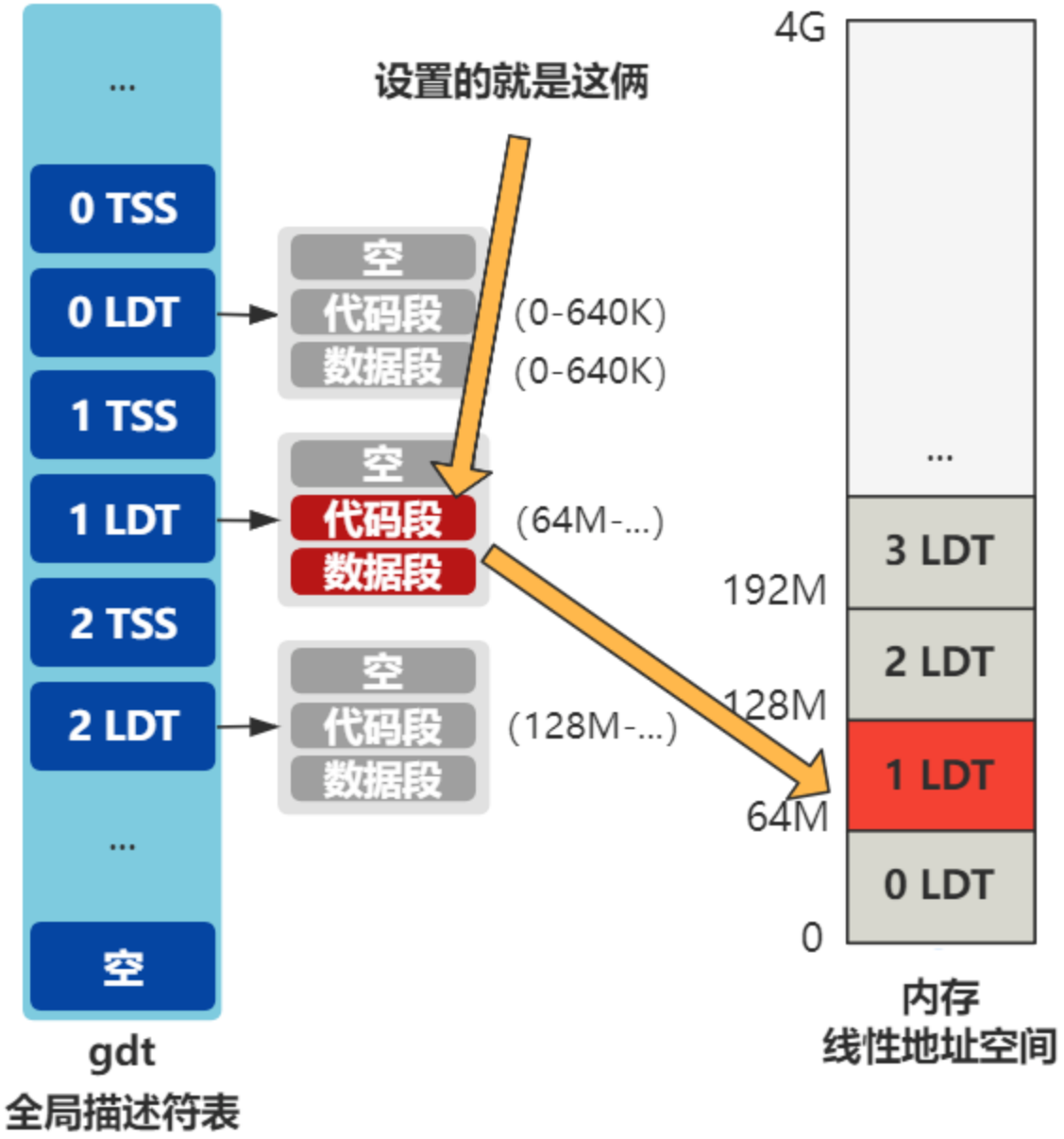
这里的 0x4000000 等于 64M。

也就是说，今后每个进程通过段基址的手段，分别在线性地址空间中占用 64M 的空间（暂不考虑段限长），且紧挨着。

接着就把 LDT 设置进了 LDT 表里。

```
int copy_mem(int nr,struct task_struct * p) {
    ...
    set_base(p->ldt[1],new_code_base);
    set_base(p->ldt[2],new_data_base);
    ...
}
```

最终效果如图。



经过以上的步骤，就通过分段的方式，将进程映射到了相互隔离的线性地址空间里，这就是**段式**管理。

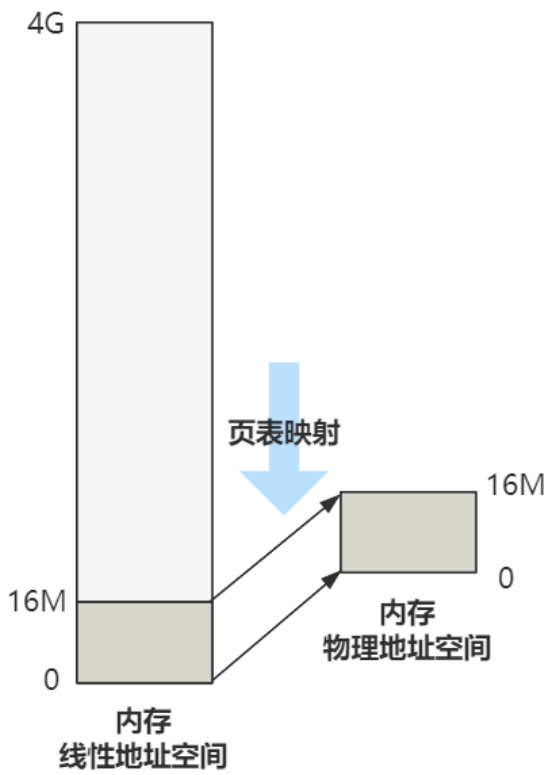
当然，Linux 0.11 不但是分段管理，也开启了分页管理，最终形成**段页式**的管理方式。这就涉及到下面要说的，页表的复制。

页表的复制

OK，上面刚刚讲完段表的赋值，接下来就是页表的复制了，这也是 copy_mem 函数里的最后一行代码。

```
int copy_mem(int nr,struct task_struct * p) {
    ...
    // old=0, new=64M, limit=640K
    copy_page_tables(old_data_base,new_data_base,data_limit)
}
```

原来进程 0 有**一个页目录表**和**四个页表**，将线性地址空间的 0-16M 原封不动映射到了物理地址空间的 0-16M。



那么新诞生的这个进程 2，也需要一套映射关系的页表，那我们看看这些页表是怎么建立的。

```
/*
 *  Well, here is one of the most complicated functions in mm. It
 * copies a range of linerar addresses by copying only the pages.
 * Let's hope this is bug-free, 'cause this one I don't want to debug :-)
 */
int copy_page_tables(unsigned long from,unsigned long to,long size)
{
    unsigned long * from_page_table;
    unsigned long * to_page_table;
    unsigned long this_page;
    unsigned long * from_dir, * to_dir;
    unsigned long nr;

    from_dir = (unsigned long *) ((from>>20) & 0xffc);
    to_dir = (unsigned long *) ((to>>20) & 0xffc);
    size = ((unsigned) (size+0x3fffff)) >> 22;
    for( ; size-->0 ; from_dir++,to_dir++) {
        if (!(1 & *from_dir))
            continue;
        from_page_table = (unsigned long *) (0xfffff000 & *from_dir);
        to_page_table = (unsigned long *) get_free_page()
        *to_dir = ((unsigned long) to_page_table) | 7;
        nr = (from==0)?0xA0:1024;
        for ( ; nr-- > 0 ; from_page_table++,to_page_table++) {
            this_page = *from_page_table;
            if (!(1 & this_page))
                continue;
            this_page &= ~2;
            *to_page_table = this_page;
            if (this_page > LOW_MEM) {
                *from_page_table = this_page;
                this_page -= LOW_MEM;
                this_page >>= 12;
                mem_map[this_page]++;
            }
        }
    }
    invalidate();
```

```
        return 0;
    }
```

先不讲这个函数，我们先看看注释。

注释是 Linus 自己写的，他说：

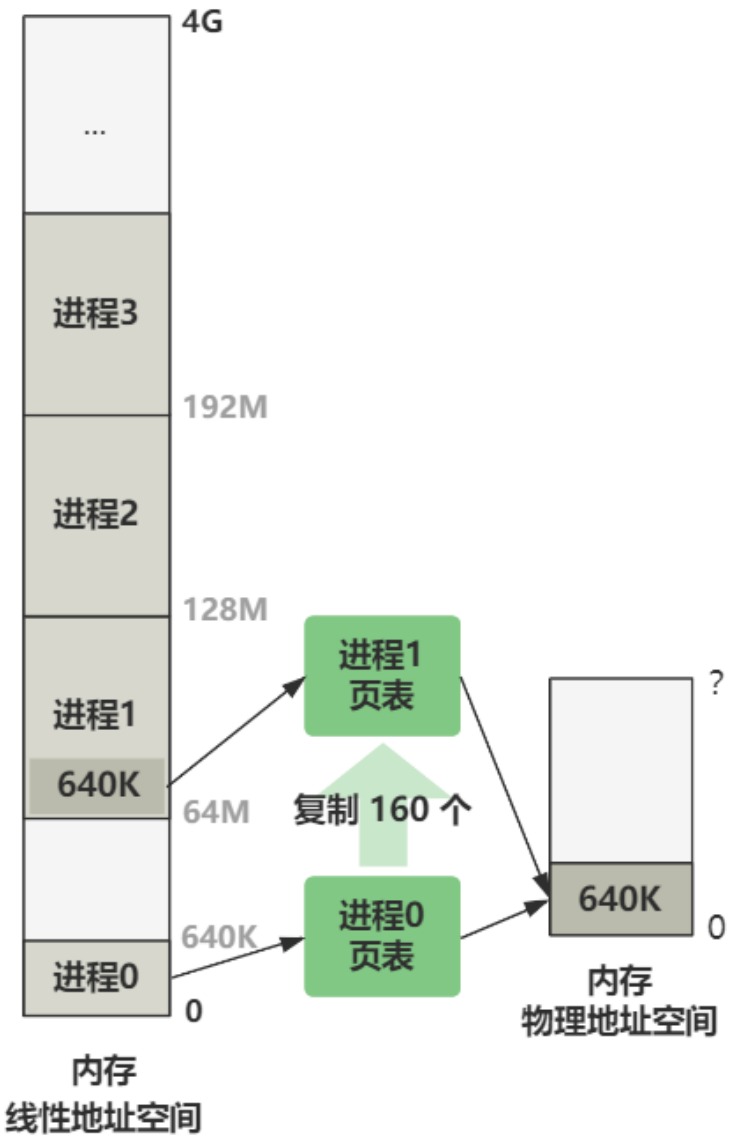
"这部分是内存管理中最复杂的代码，希望这段代码没有错误（bug-free），因为我实在不想调试它！"

可见这是一套让 Linus 都觉得烧脑的逻辑。

虽说代码实现很复杂，但要完成的事情确实非常简单！我想我们要是产品经理，一定会和 Linus 说这么简单的功能有啥难实现的？哈哈。

回归正题，这个函数要完成什么事情呢？

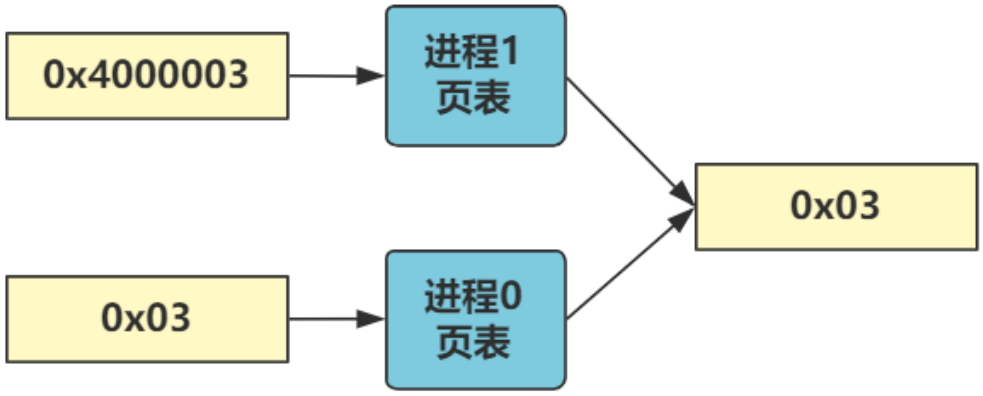
你想，现在进程 0 的线性地址空间是 0 - 64M，进程 1 的线性地址空间是 64M - 128M。**我们现在要造一个进程 1 的页表，使得进程 1 和进程 0 最终被映射到的物理空间都是 0 - 64M**，这样进程 1 才能顺利运行起来，不然就乱套了。



总之，最终的效果就是：

假设现在正在运行进程 0，代码中给出一个虚拟地址 0x03，由于进程 0 的 LDT 中代码段基址是 0，所以线性地址也是 0x03，最终由进程 0 页表映射到物理地址 0x03 处。

假设现在正在运行进程 1，代码中给出一个虚拟地址 0x03，由于进程 1 的 LDT 中代码段基址是 64M，所以线性地址是 64M + 3，最终由进程 1 页表映射到物理地址也同样是 0x03 处。



即，进程 0 和进程 1 目前共同映射物理内存的前 640K 的空间。

至于如何将不同地址通过不同页表映射到相同物理地址空间，很简单，举个刚刚的例子。

刚刚的进程 1 的线性地址 64M + 0x03 用二进制表示是：
0000010000_0000000000_000000000011

刚刚的进程 0 的线性地址 0x03 用二进制表示是：
0000000000_0000000000_000000000011

根据分页机制的转化规则，**前 10 位表示页目录项，中间 10 位表示页表项，后 12 位表页内偏移。**

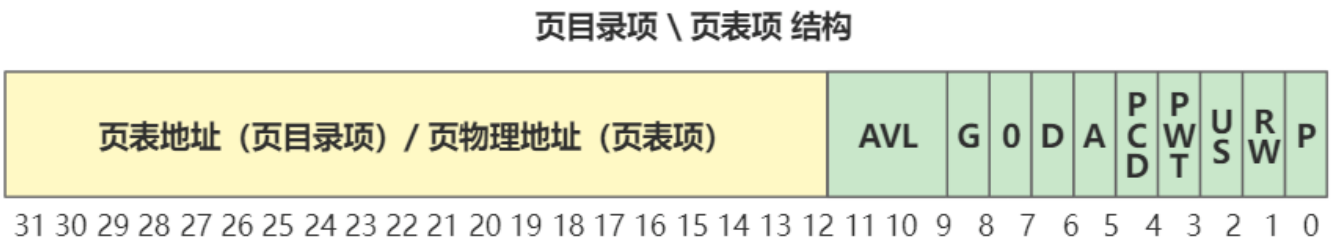
进程 1 要找的是页目录项 16 中的第 0 号页表
进程 0 要找的是页目录项 0 中的第 0 号页表

那只要让这俩最终找到的两个页表里的数据一模一样即可。

我居然会认为权威书籍写错了...

由于理解起来非常简单，但代码中的计算就非常绕，所以我们就不细致分析代码了，只要理解其最终的作用就好。

OK，本章的内容就讲完了，再稍稍展开一个未来要说的东西。还记得页表的结构吧？



其中 RW 位表示读写状态，0 表示只读（或可执行），1表示可读写（或可执行）。当然，在内核态也就是 0 特权级时，这个标志位是没用的。

那我们看下面的代码。

```
int copy_page_tables(unsigned long from,unsigned long to,long size) {
    ...
    for( ; size-->0 ; from_dir++,to_dir++) {
        ...
        for ( ; nr-- > 0 ; from_page_table++,to_page_table++) {
            ...
            this_page &= ~2;
            ...
            if (this_page > LOW_MEM) {
                *from_page_table = this_page;
                ...
            }
        }
    }
    ...
}
```

~2 表示取反，2 用二进制表示是 10，取反就是 01，其目的是把 this_page 也就是当前的页表的 RW 位置零，也就是是**把该页变成只读。**

而 *from_page_table = this_page 表示**又把源页表也变成只读。**

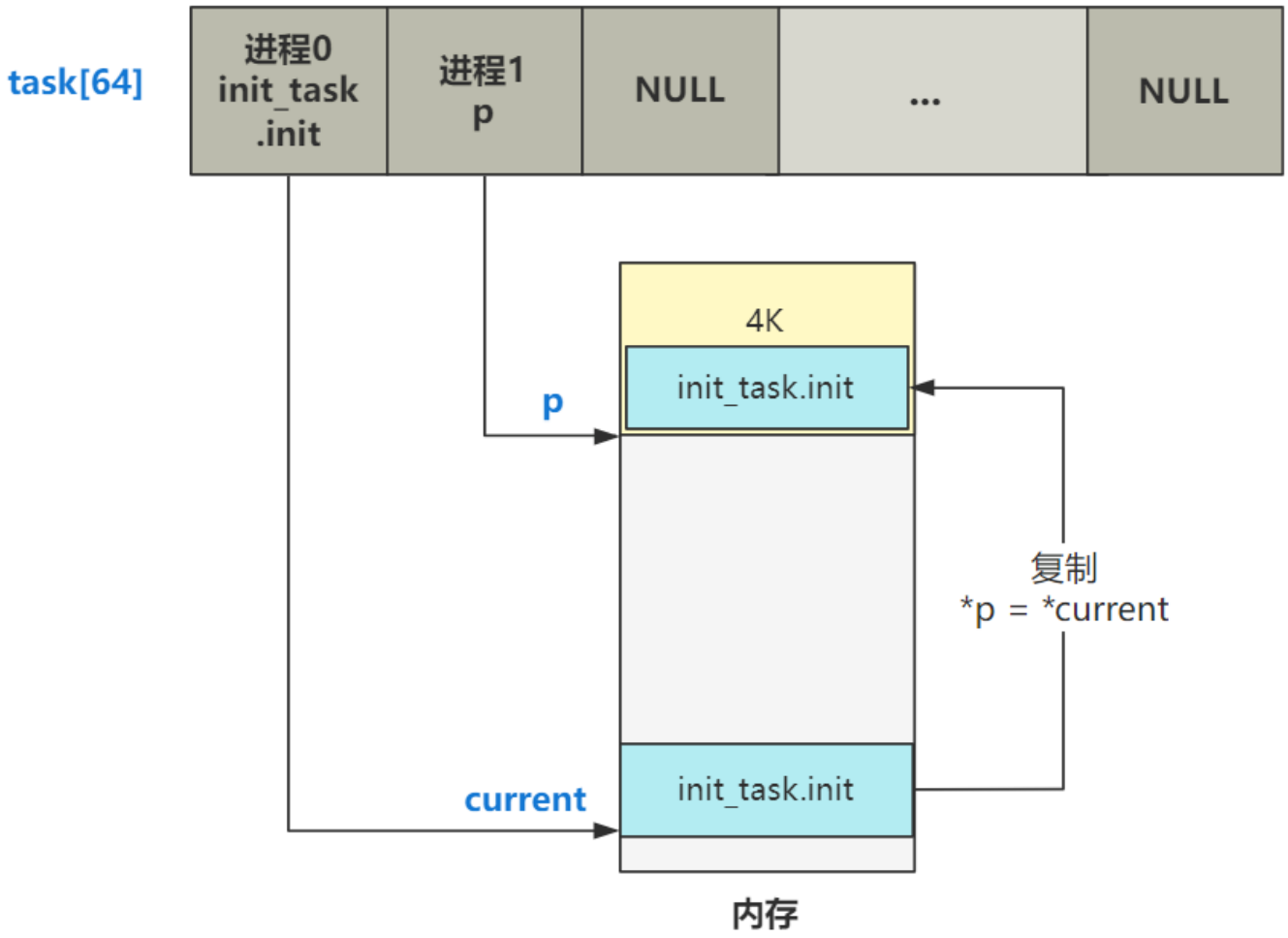
也就是说，经过 fork 创建出的新进程，其页表项都是只读的，而且导致源进程的页表项也变成了只读。

这个就是**写时复制**的基础，新老进程一开始共享同一个物理内存空间，如果只有读，那就相安无事，但如果任何一方有写操作，由于页面是只读的，将触发缺页中断，然后就会分配一块新的物理内存给产生写操作的那个进程，此时这一块内存就不再共享了。

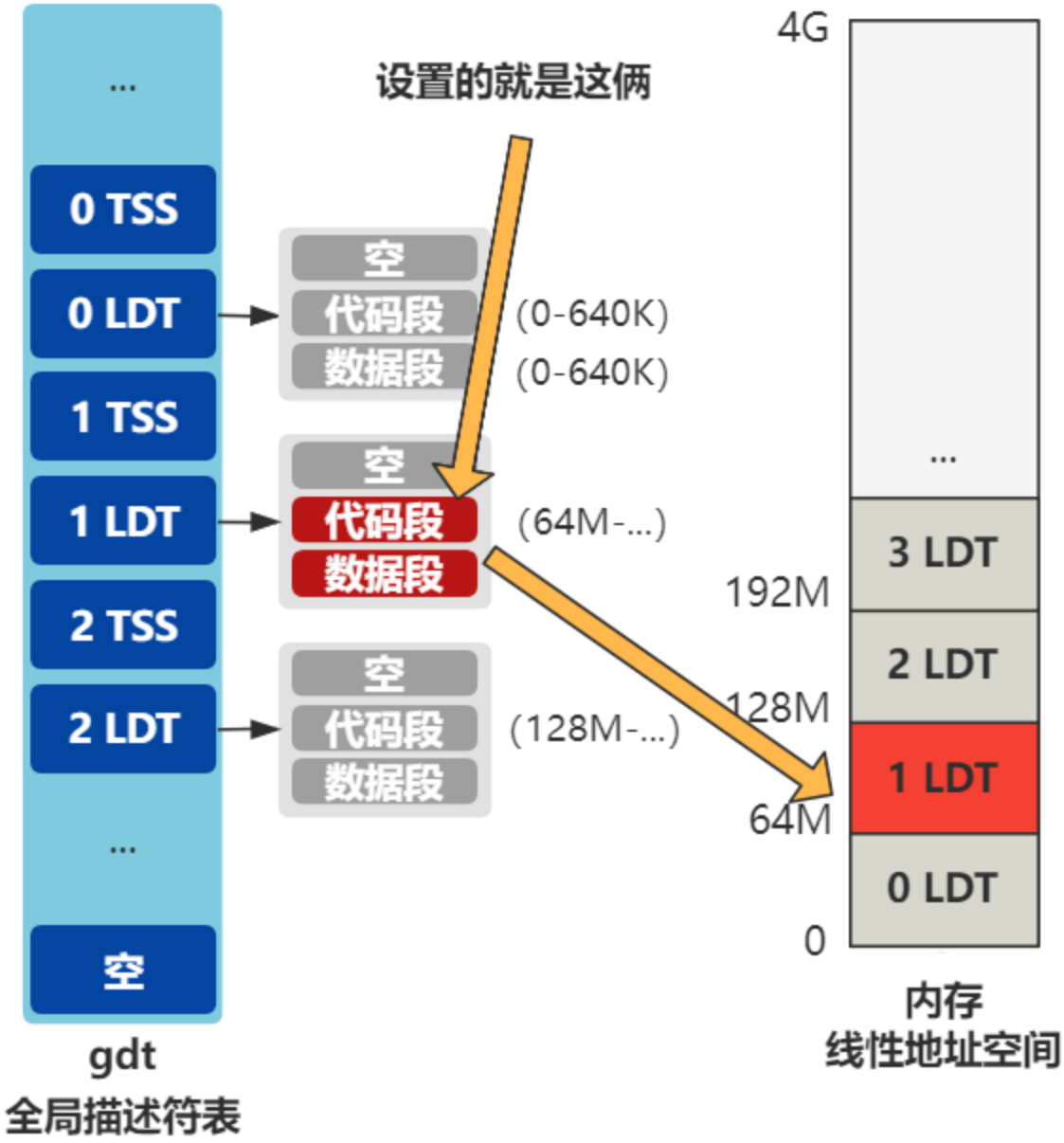
这是后话了，这里先埋个伏笔。

好了，至此 fork 中的 **copy_process** 函数就全部被我们读完了，总共做了三件事，把整个进程的数据结构个性化地从进程 0 复制给了进程 1。

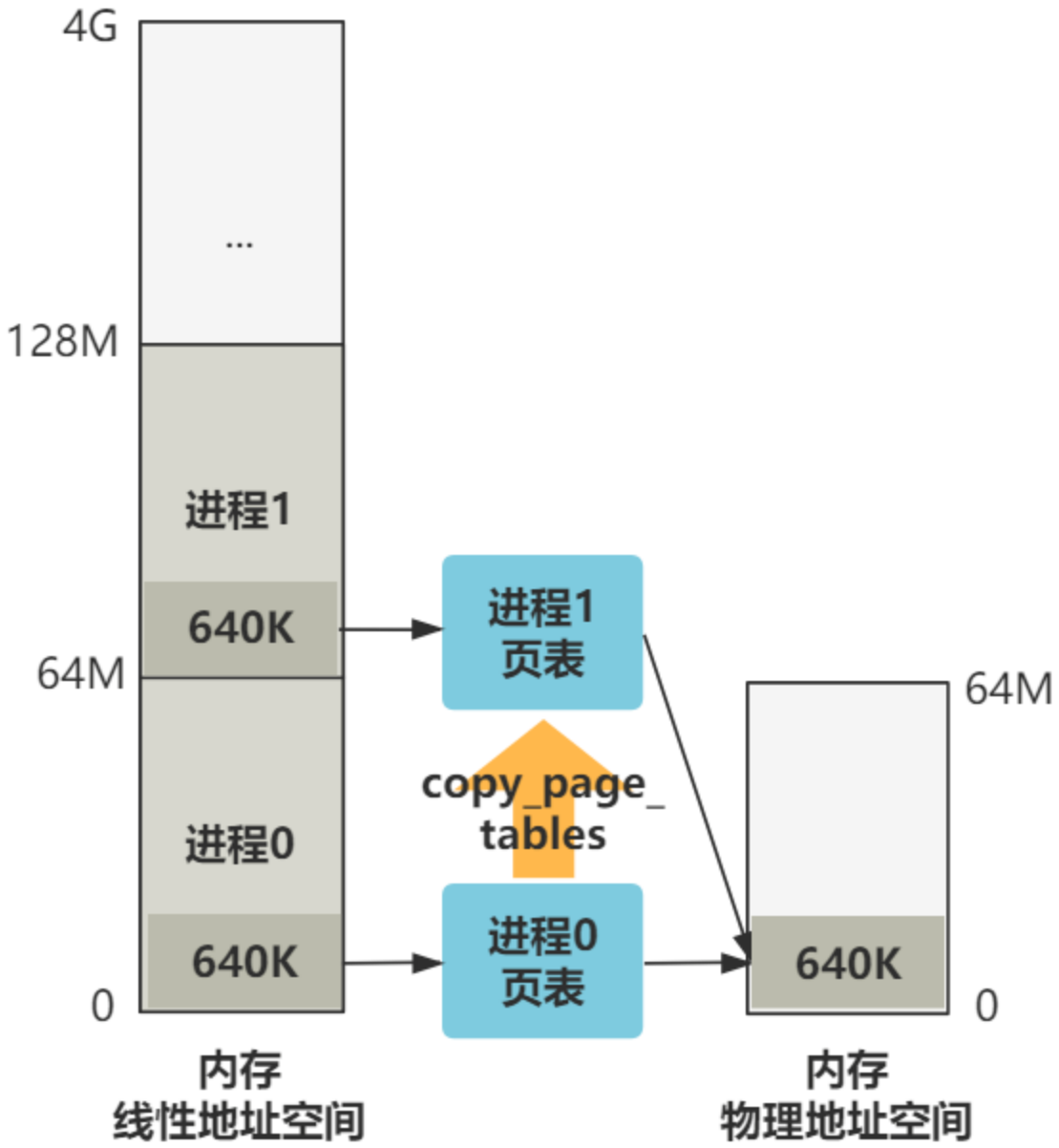
第一，原封不动复制了一下 task_struct。



第二，LDT 的复制和改造，使得进程 0 和进程 1 分别映射到了不同的线性地址空间。



第三，页表的复制，使得进程 0 和进程 1 又从不同的线性地址空间，被映射到了相同的物理地址空间。



最后，将新老进程的页表都变成只读状态，为后面写时复制的缺页中断做准备。

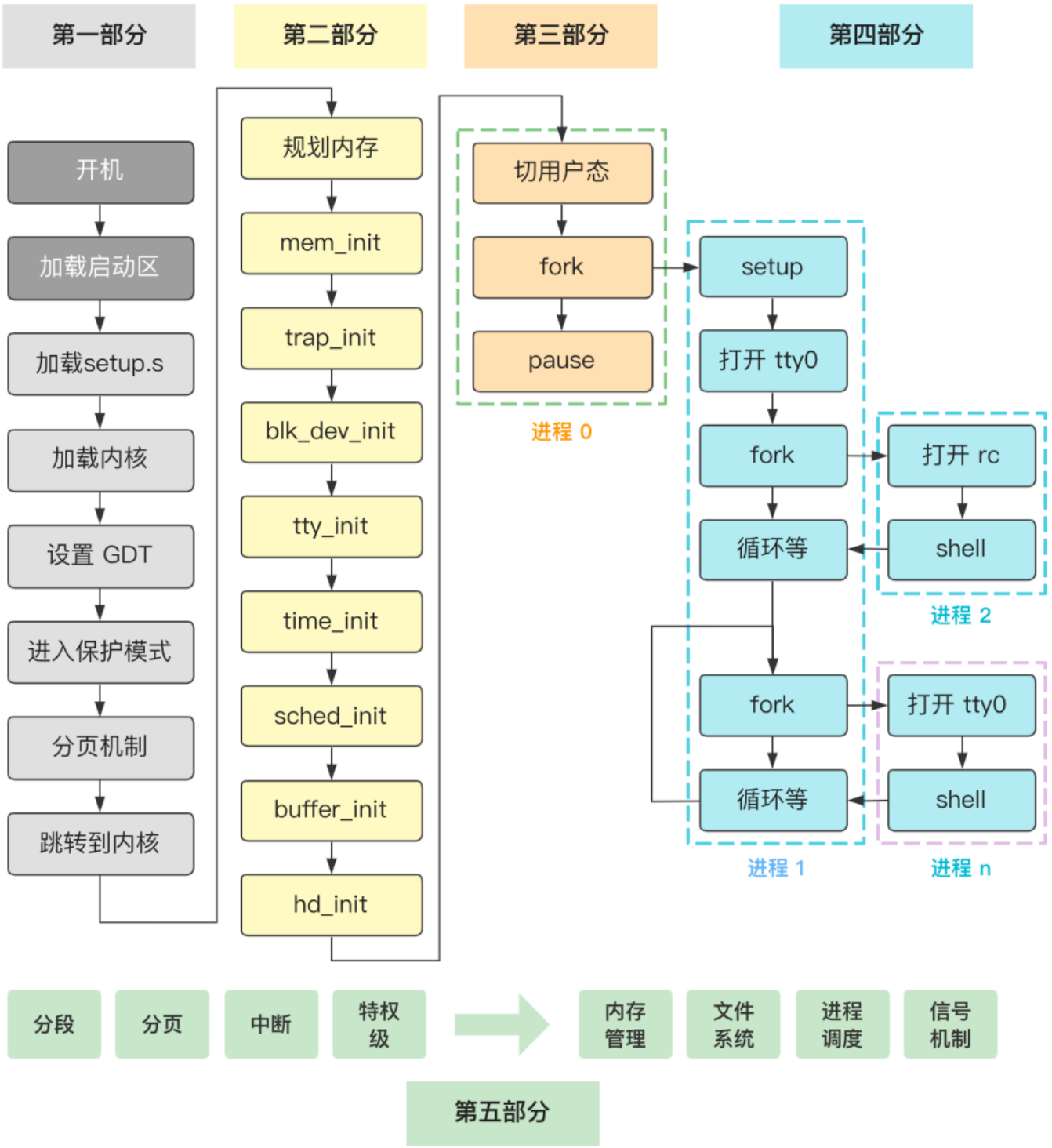
欲知后事如何，且听下回分解。

------- 关于本系列的完整内容 -------

本系列的开篇词看这，开篇词

本系列的番外故事看这，让我们一起来写本书？

本系列全局视角



最后，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

收录于合集 #操作系统源码 43

上一篇
一个新进程的诞生（六）fork 中进程基本信息的复制

下一篇
一个新进程的诞生 完结撒花！！！

Read more