一个新进程的诞生 完结撒花！！！

Original 闪客 低并发编程 2022-03-16 17:30

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。

到这里，第三部分终于也完结了，其实核心就是讲述了一个 fork 函数的原理，我们本篇文章就回顾下整个第三部分的事情。

------- 整个系列的目录 -------

第一部分 进入内核前的苦力活

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
第八回 | 烦死了又要重新设置一遍 idt 和 gdt
第九回 | Intel 内存管理两板斧：分段与分页
第十回 | 进入 main 函数前的最后一跃！
第一部分完结 进入内核前的苦力活

第二部分 大战前期的初始化工作

第11回 | 整个操作系统就 20 几行代码
第12回 | 管理内存前先划分出三个边界值
第13回 | 主内存初始化 mem_init
第14回 | 中断初始化 trap_init
第15回 | 块设备请求项初始化 blk_dev_init
第16回 | 控制台初始化 tty_init
第17回 | 时间初始化 time_init
第18回 | 进程调度初始化 sched_init
第19回 | 缓冲区初始化 buffer_init
第20回 | 硬盘初始化 hd_init
第二部分总结与回顾

第三部分：一个新进程的诞生

第21回 | 新进程诞生全局概述
第22回 | 从内核态切换到用户态
第23回 | 如果让你来设计进程调度
第24回 | 从一次定时器滴答来看进程调度
第25回 | 通过 fork 看一次系统调用
第26回 | fork 中进程基本信息的复制
第27回 | 透过 fork 来看进程的内存规划

------- 正文开始 -------

整个第三部分，我们用前四回的内容讲述了**进程调度机制**，又用后三回内容讲述了 **fork** 函数的全部细节。先看进程调度机制。

进程调度机制

前四回内容循序渐进地讲述了进程调度机制的设计思路和细节。

第21回 | 新进程诞生全局概述
第22回 | 从内核态切换到用户态
第23回 | 如果让你来设计进程调度
第24回 | 从一次定时器滴答来看进程调度

进程调度的始作俑者，就是那个每 10ms 触发一次的定时器滴答。

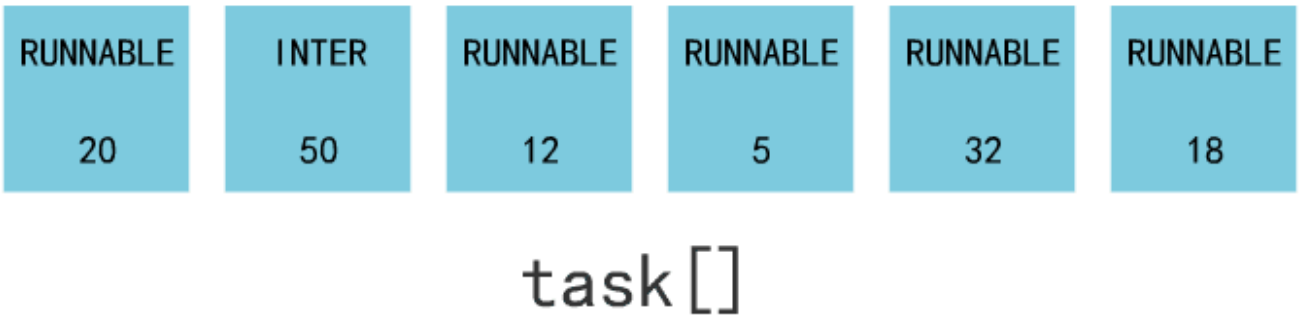
而这个滴答将会给 CPU 产生一个**时钟中断**信号。

而这个中断信号会使 CPU 查找中断向量表，找到操作系统写好的一个时钟中断处理函数 **do_timer**。

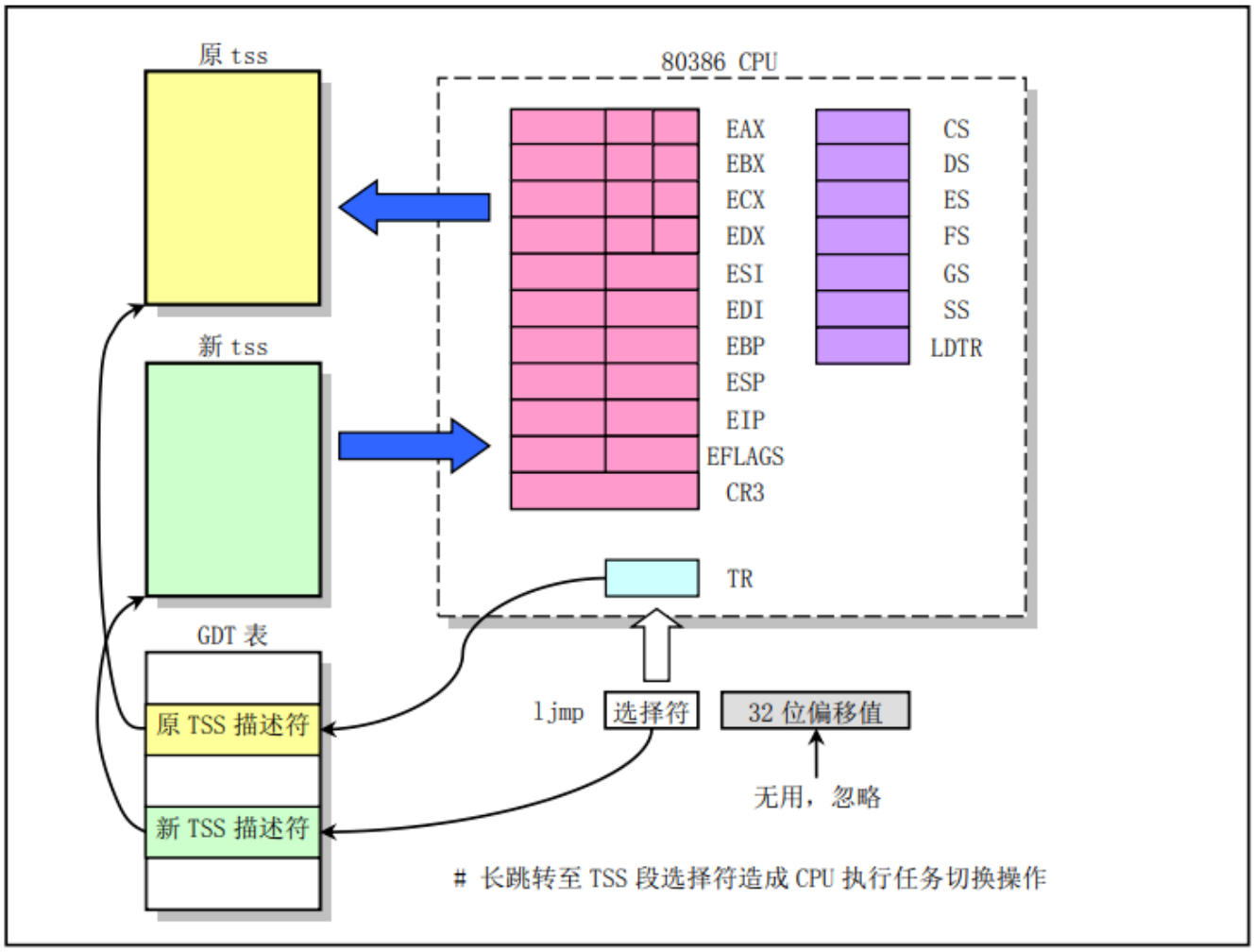
do_timer 会首先将当前进程的 **counter** 变量 -1，如果 counter 此时仍然大于 0，则就此结束。

但如果 counter = 0 了，就开始进行进程的调度。

进程调度就是找到所有处于 **RUNNABLE** 状态的进程，并找到一个 counter 值最大的进程，把它丢进 **switch_to** 函数的入参里。



switch_to 这个终极函数，会保存当前进程上下文，恢复要跳转到的这个进程的上下文，同时使得 CPU 跳转到这个进程的偏移地址处。



上图来源于《Linux内核完全注释V5.0》

接着，这个进程就舒舒服服地运行了起来，等待着下一次**时钟中断**的来临。

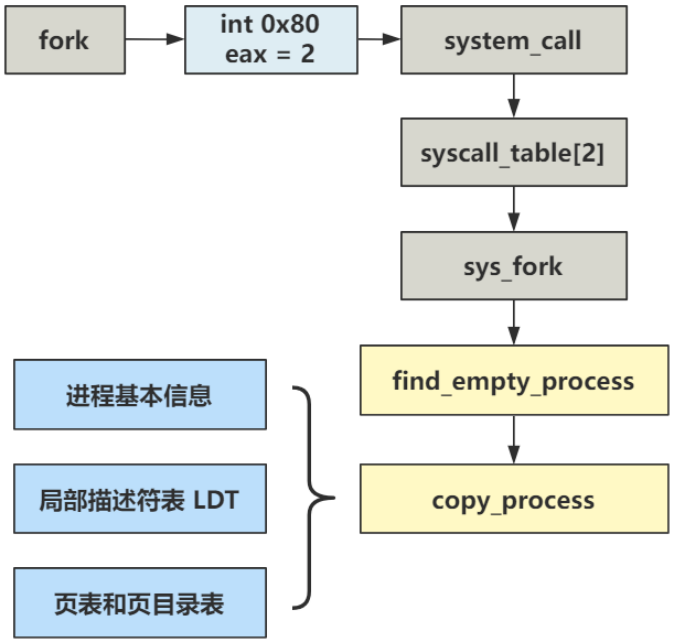
聊完进程调度机制，我们再看看 fork 函数的原理。

fork

后三回内容讲述了 fork 函数的全部细节。

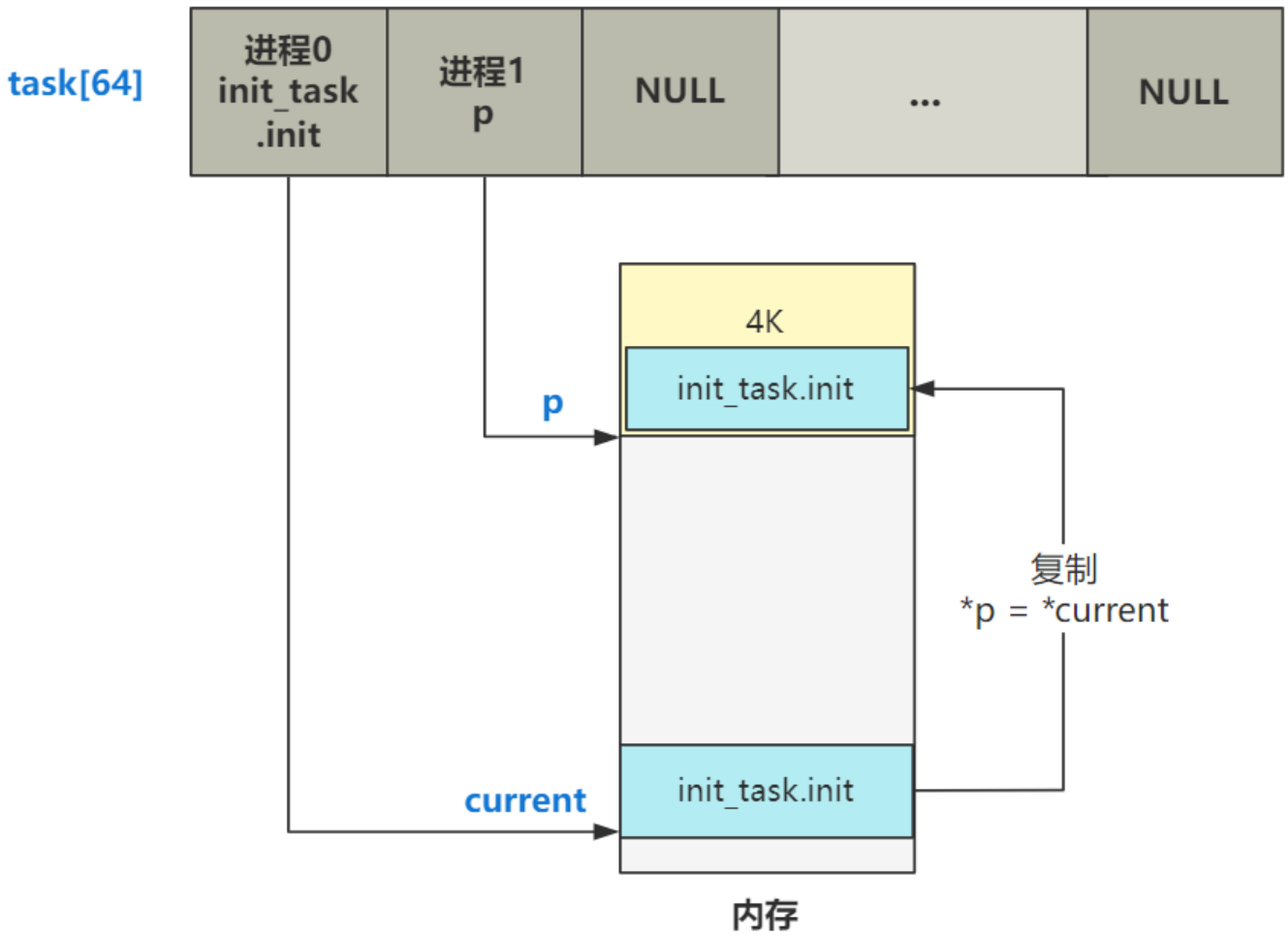
第25回 | 通过 fork 看一次系统调用
第26回 | fork 中进程基本信息的复制
第27回 | 透过 fork 来看进程的内存规划

用一张图来表示的话，就是。

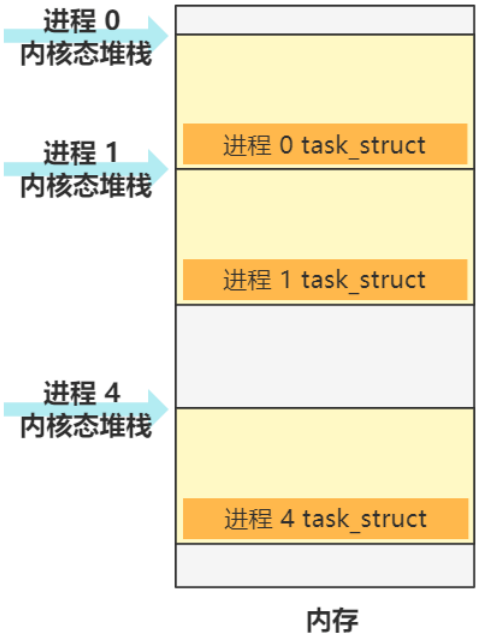


其中 **copy_process** 是复制进程的关键，总共分三步来。

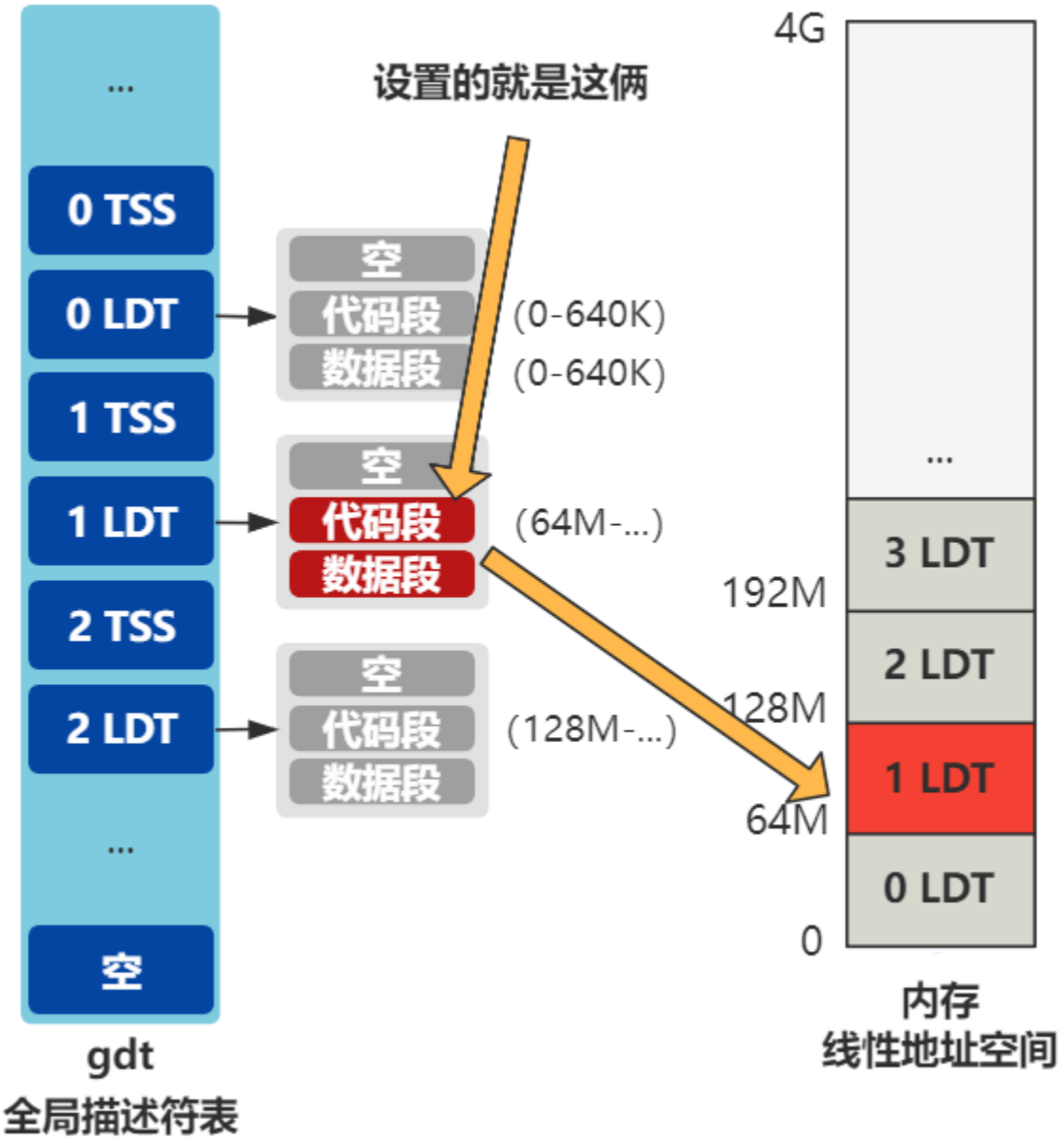
第一，原封不动复制了一下 task_struct。



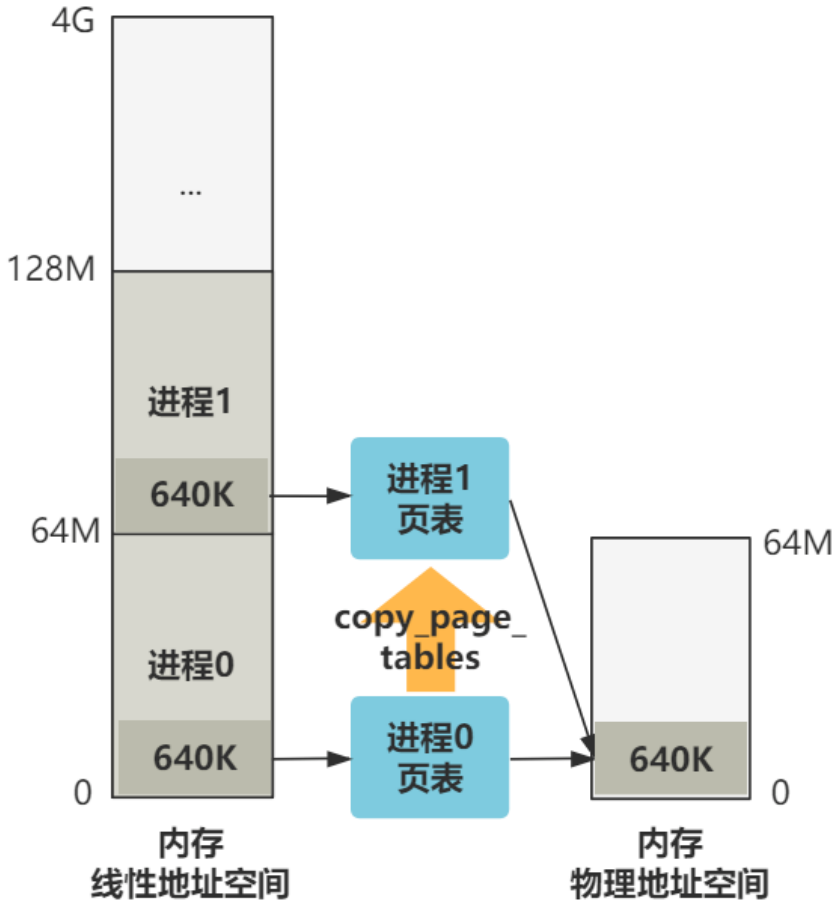
并且覆盖了一些基本信息，包括元信息和一些寄存器的信息。其中比较重要的是将内核态堆栈栈顶指针的指向了自己进程结构所在 4K 内存页的最顶端。



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。

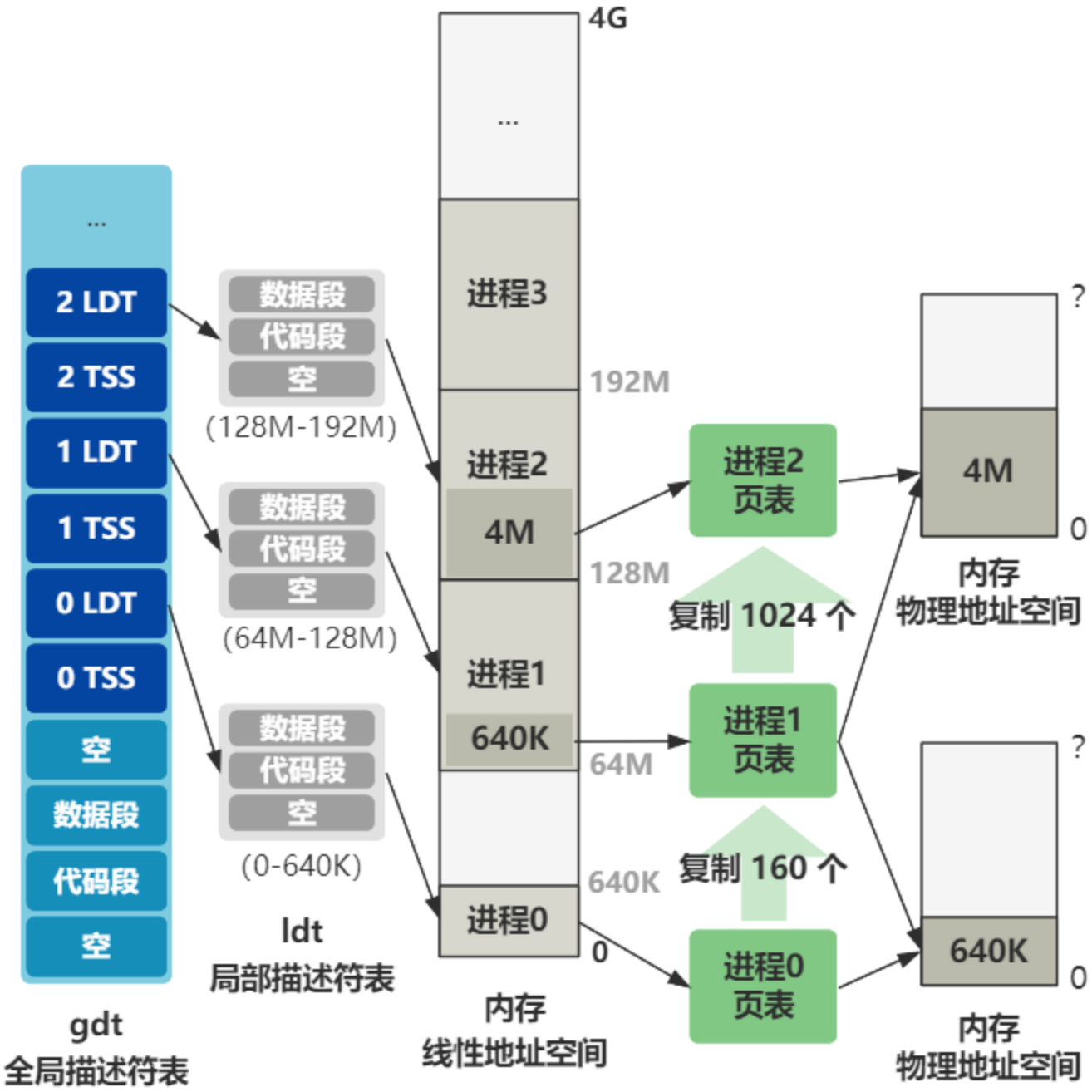


第三，页表的复制，使得进程 0 和进程 1 又从不同的线性地址空间，被映射到了相同的物理地址空间。



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。

这一部分的 fork 函数只用于进程 0 创造进程 1 的过程，而之后的新进程创建，比如进程 1 里 fork 创建进程 2，也都是这样的套路。



整个核心函数 copy_process 的代码如下。

```
int copy_process(int nr, ...) {
    struct task_struct p =
        (struct task_struct *) get_free_page();
    task[nr] = p;
    *p = *current;
    p->state = TASK_UNINTERRUPTIBLE;
    p->pid = last_pid;
    p->counter = p->priority;
    ..
    p->tss.edx = edx;
    p->tss.ebx = ebx;
    p->tss.esp = esp;
    ...
    copy_mem(nr,p);
    ...
    set_tss_desc(gdt+(nr<<1)+FIRST_TSS_ENTRY,&(p->tss));
    set_ldt_desc(gdt+(nr<<1)+FIRST_LDT_ENTRY,&(p->ldt));
    p->state = TASK_RUNNING;
    return last_pid;
}
```

注意图中的两个标黄的代码。

开始复制进程信息的时候，由于进程 1 的结构还没弄好，此时如果进程调度到了进程 1，那就坏事了。

所以一开始把进程 1 的状态先设置为 **TASK_UNINTERRUPTIBLE**，使得其不会被进程调度算法选中。

而所有复制工作完成后，进程 1 就拥有了运行的内容，进程基本信息也有了，进程的内存规划也完成了。

此时就把进程设置为 **TASK_RUNNING**，允许被 CPU 调度。

看到这行代码，其实我们也可以很自信地认为，**到这里进程 1 的初步建立工作已经圆满结束，可以达到运行在 CPU 上的标准了。**

第四部分的展望

那我们此时又该回到之前的 main 方法，是不是都忘了最初的目的了？哈哈。

```
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();
}
```

看，下一行代码，是 **init**。

fork 只是把进程 1 搞成可以在 CPU 中运行的进程，之后创建新进程，都可以用这个 fork 方法。

不过进程 1 具体要做什么事情呢？那就是 init 这个函数的故事了。

虽然就一行代码，但这里的事情可多了去了，我们先看一下整体结构。我已经把单纯的日志打印和错误校验逻辑去掉了。

```
void init(void) {
    int pid,i;
    setup((void *) &drive_info);
    (void) open("/dev/tty0",O_RDWR,0);
    (void) dup(0);
    (void) dup(0);
    if (!(pid=fork())) {
        open("/etc/rc",O_RDONLY,0);
        execve("/bin/sh",argv_rc,envp_rc);
    }
    if (pid>0)
        while (pid != wait(&i))
            /* nothing */;
    while (1) {
        if (!pid=fork()) {
            close(0);close(1);close(2);
            setsid();
            (void) open("/dev/tty0",O_RDWR,0);
            (void) dup(0);
            (void) dup(0);
            _exit(execve("/bin/sh",argv,envp));
        }
        while (1)
            if (pid == wait(&i))
                break;
        sync();
    }
    _exit(0);   /* NOTE! _exit, not exit() */
}
```

是不是看着还挺复杂?

不过还好，我们几乎已经把计算机体系结构，和操作系统的设计思想，通过前面的源码阅读，不知不觉建立起来了。

接下来的工作，就是基于这些建立好的能力，站在巨人的肩膀上，做些更伟大的事情!

说伟大其实也没什么伟大的，就是最终建立好一个人机交互的 shell 程序，无限等待用户输入的命令。

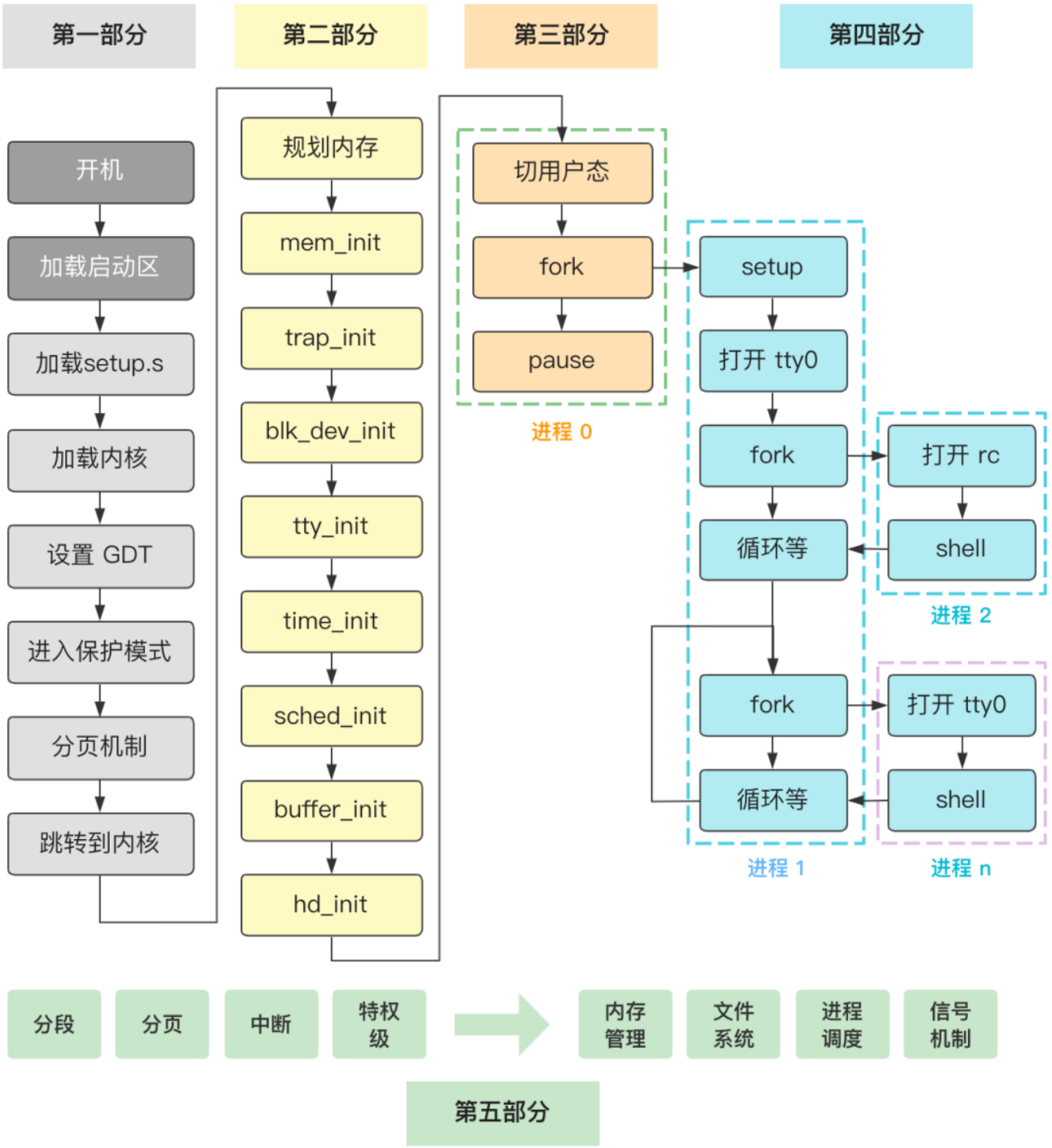
欲知后事如何，且听第四部分的分解！

------- 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。

收录于合集 #操作系统源码 43

上一篇
一个新进程的诞生（七）透过 fork 来看进程的内存规划

下一篇
写时复制就这么几行代码，麻烦你先看看再 BB 行吗?

Read more

People who liked this content also liked

RabitMQ 发布确认
ylcoder

用户态 tcpdump 如何实现抓到内核网络包的?
CSDN云计算

代理应用-
聊聊IT技术