一个新进程的诞生（六）fork 中进程基本信息的复制

Original 闪客 低并发编程 2022-03-02 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。

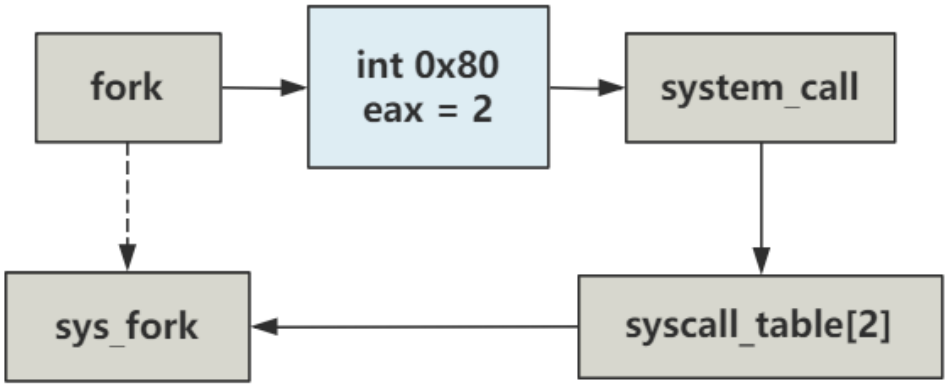
第一部分 进入内核前的苦力活
第二部分 大战前期的初始化工作

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 触发系统调用中断，最终调用到了 sys_fork 函数，借这个过程介绍了一次**系统调用**的流程。



那今天我们回到正题，开始讲 **fork** 函数的原理，实际上就是 **sys_fork** 函数干了啥。

还是个汇编代码，但我们要关注的地方不多。

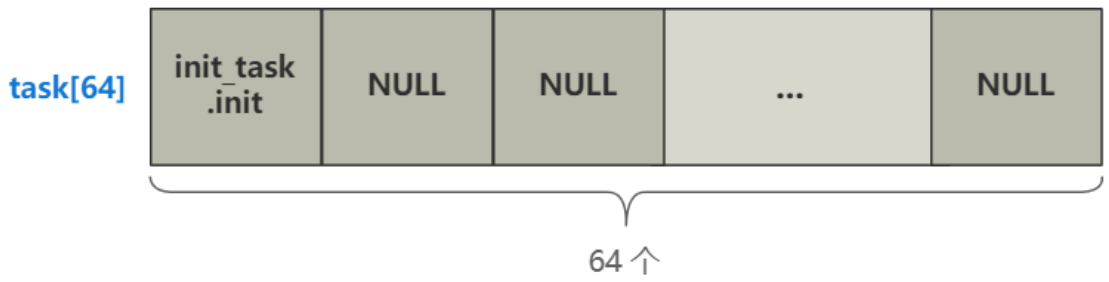
```
_sys_fork:
    call _find_empty_process
    testl %eax,%eax
    js 1f
    push %gs
    pushl %esi
    pushl %edi
    pushl %ebp
    pushl %eax
    call _copy_process
    addl $20,%esp
1:  ret
```

其实就是调用了两个函数。我们先从方法名直接翻译一下，猜猜意思。

先是 **find_empty_process**，就是找到空闲的进程槽位。

然后 **copy_process**，就是复制进程。

那妥了，这个方法的意思非常简单，因为存储进程的数据结构是一个 task[64] 数组，这个是在之前 第18回 | 大名鼎鼎的进程调度就是从这里开始的 **sched_init** 函数的时候设置的。



就是先在这个数组中找一个空闲的位置，准备存一个新的进程的结构 **task_struct**，这个结构之前在 一个新进程的诞生（三）如果让你来设计进程调度 也简单说过了。

```
struct task_struct {
    long state;
    long counter;
    long priority;
    ...
    struct tss_struct tss;
}
```

这个结构各个字段具体赋什么值呢？

通过 **copy_process** 这个名字我们知道，就是复制原来的进程，也就是当前进程。

当前只有一个进程，就是数组中位置 0 处的 **init_task.init**，也就是零号进程，那自然就复制它咯。

好了，以上只是我们的猜测，有了猜测再看代码会非常轻松，我们一个个函数看。

先来 **find_empty_process**。

```
long last_pid = 0;

int find_empty_process(void) {
    int i;
    repeat:
        if ((++last_pid)<0) last_pid=1;
        for(i=0 ; i<64 ; i++)
            if (task[i] && task[i]->pid == last_pid) goto repeat;
    for(i=1 ; i<64; i++)
        if (!task[i])
            return i;
    return -EAGAIN;
}
```

一共三步，很简单。

第一步，判断 ++last_pid 是不是小于零了，小于零说明已经超过 long 的最大值了，重新赋

值为 1，起到一个保护作用，这没什么好说的。

第二步，一个 for 循环，看看刚刚的 last_pid 在所有 task[] 数组中，是否已经被某进程占用了。如果被占用了，那就重复执行，再次加一，然后再次判断，直到找到一个 pid 号没有被任何进程用为止。

第三步，又是个 for 循环，刚刚已经找到一个可用的 pid 号了，那这一步就是再次遍历这个 task[] 试图找到一个空闲项，找到了就返回素组索引下标。

最终，这个方法就返回 task[] 数组的索引，表示找到了一个空闲项，之后就开始往这里塞一个新的进程吧。

由于我们现在只有 0 号进程，且 task[] 除了 0 号索引位置，其他地方都是空的，所以这个方法运行完，last_**pid 就是 1，也就是新进程被分配的 pid 就是 1**，然后即将要加入的 task[] 数组的索引位置，也是 1。

好的，那我们接下来就看，怎么构造这个进程结构，塞到这个 1 索引位置的 task[] 中？

来看 **copy_process** 方法。

```
int copy_process(int nr,long ebp,long edi,long esi,long gs,long none,
        long ebx,long ecx,long edx,
        long fs,long es,long ds,
        long eip,long cs,long eflags,long esp,long ss)
{
    struct task_struct *p;
    int i;
    struct file *f;


    p = (struct task_struct *) get_free_page();
    if (!p)
        return -EAGAIN;
    task[nr] = p;
    *p = *current;  /* NOTE! this doesn't copy the supervisor stack */
    p->state = TASK_UNINTERRUPTIBLE;
    p->pid = last_pid;
    p->father = current->pid;
    p->counter = p->priority;
    p->signal = 0;
    p->alarm = 0;
    p->leader = 0;      /* process leadership doesn't inherit */
    p->utime = p->stime = 0;
    p->cutime = p->cstime = 0;
    p->start_time = jiffies;
    p->tss.back_link = 0;
    p->tss.esp0 = PAGE_SIZE + (long) p;
    p->tss.ss0 = 0x10;
    p->tss.eip = eip;
    p->tss.eflags = eflags;
    p->tss.eax = 0;
    p->tss.ecx = ecx;
    p->tss.edx = edx;
    p->tss.ebx = ebx;
    p->tss.esp = esp;
    p->tss.ebp = ebp;
    p->tss.esi = esi;
    p->tss.edi = edi;
    p->tss.es = es & 0xffff;
    p->tss.cs = cs & 0xffff;
    p->tss.ss = ss & 0xffff;
```

```
    p->tss.ss = ss & 0xffff;
    p->tss.ds = ds & 0xffff;
    p->tss.fs = fs & 0xffff;
    p->tss.gs = gs & 0xffff;
    p->tss.ldt = _LDT(nr);
    p->tss.trace_bitmap = 0x80000000;
    if (last_task_used_math == current)
        __asm__("clts ; fnsave %0"::"m" (p->tss.i387));
    if (copy_mem(nr,p)) {
        task[nr] = NULL;
        free_page((long) p);
        return -EAGAIN;
    }
    for (i=0; i<NR_OPEN;i++)
        if (f=p->filp[i])
            f->f_count++;
    if (current->pwd)
        current->pwd->i_count++;
    if (current->root)
        current->root->i_count++;
    if (current->executable)
        current->executable->i_count++;
    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;    /* do this last, just in case */
    return last_pid;
}
```

艾玛，这也太多了！

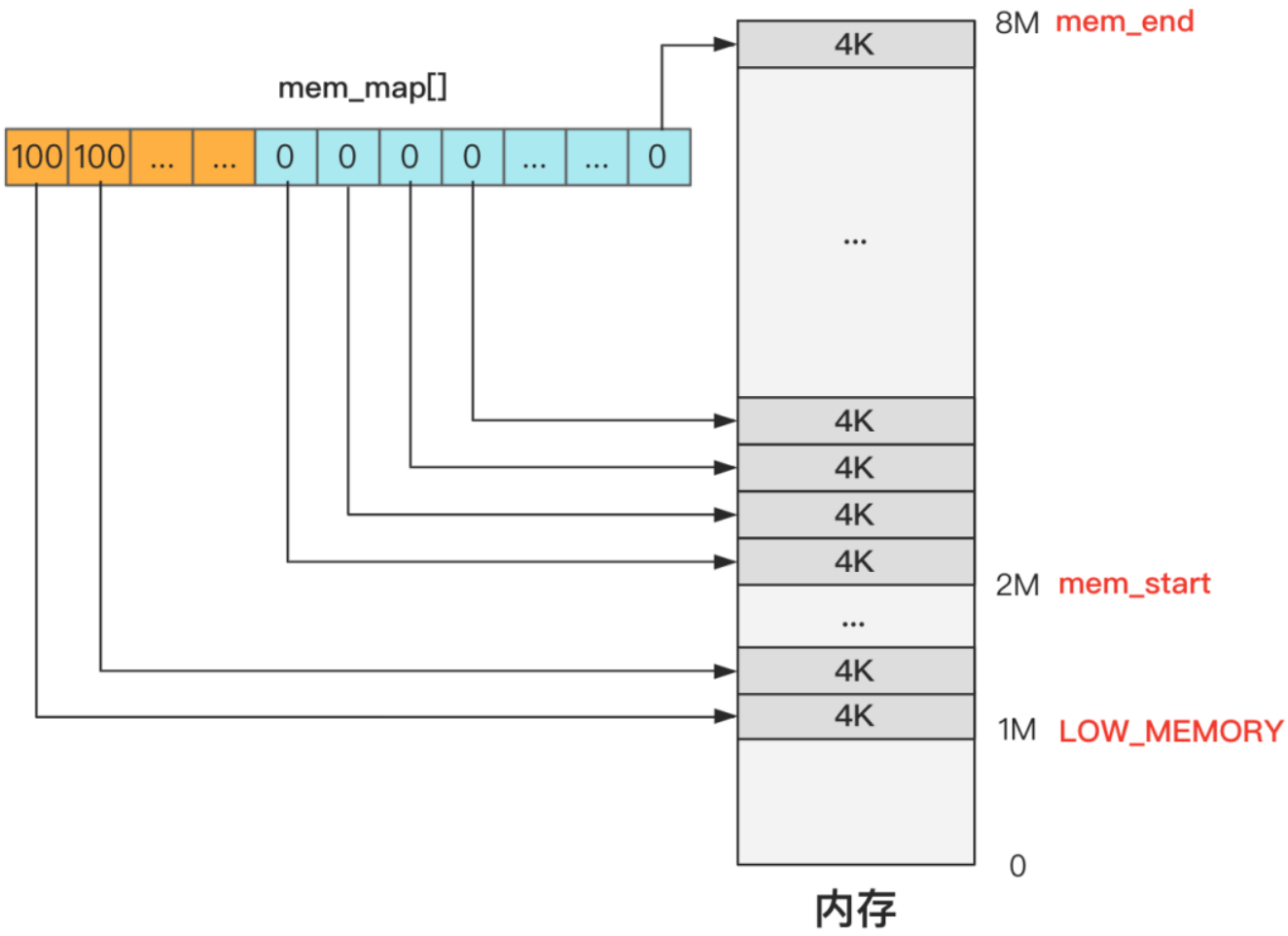
别急，大部分都是 tss 结构的复制，以及一些无关紧要的分支，看我简化下。

```
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;
}
```

这个函数本来就是 fork 的难点了，所以我们慢慢来。

首先 **get_free_page** 会在主内存末端申请一个空闲页面，还记得我们之前在 第13回 内存初始化 mem_init 里是怎么管理内存的吧？



那 get_free_page 这个函数就很简单了，**就是遍历 mem_map[] 这个数组，找出值为零的项，就表示找到了空闲的一页内存。**然后把该项置为 1，表示该页已经被使用。最后，算出这个页的内存起始地址，返回。

然后，拿到的这个内存起始地址，就给了 task_struct 结构的 p。

```
int copy_process(int nr, ...) {
    struct task_struct p =
        (struct task_struct *) get_free_page();
    task[nr] = p;
    *p = *current;
    ...
}
```

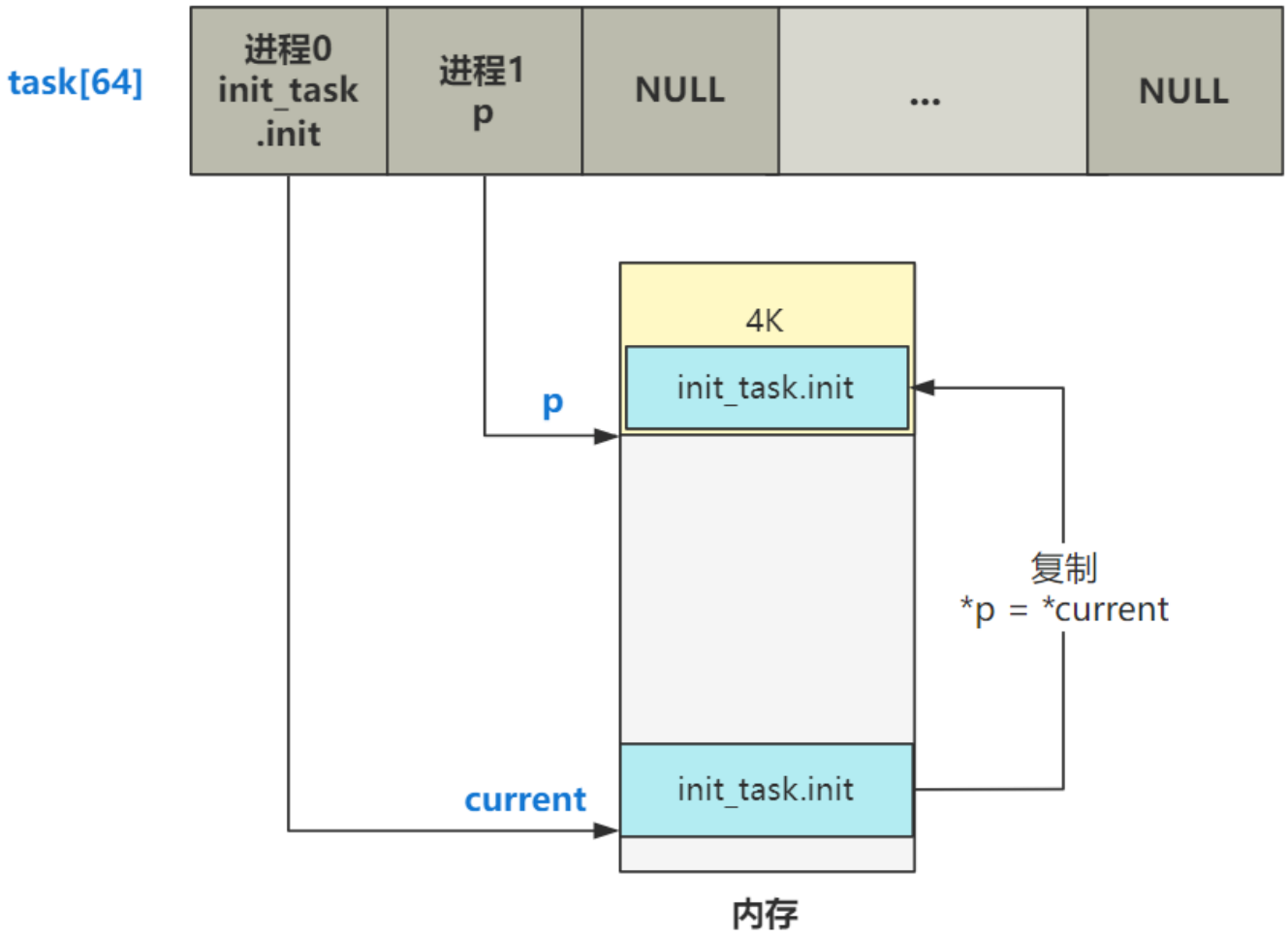
于是乎，一个进程结构 task_struct 就在内存中有了一块空间，但此时还没有赋值具体的字段。别急。

首先将这个 p 记录在进程管理结构 task[] 中。

然后下一句 *p = *current 很简单，**就是把当前进程，也就是 0 号进程的 task_struct 的全部值都复制给即将创建的进程 p**，目前它们两者就完全一样了。

嗯，这就附上值了，就完全复制之前的进程的 task_struct 而已，很粗暴。

最后的内存布局的效果就是这样。



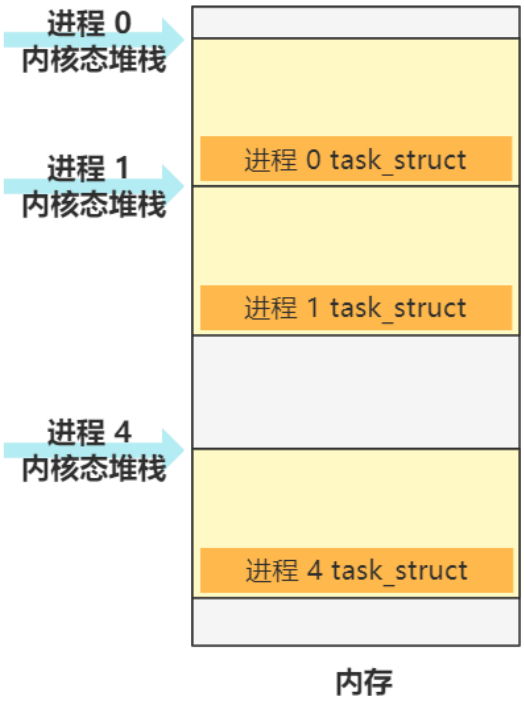
然后，进程 1 和进程 0 目前是完全复制的关系，但有一些值是需要个性化处理的，下面的代码就是把这些不一样的值覆盖掉。

```
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;
    ...
    p->tss.esp0 = PAGE_SIZE + (long) p;
    p->tss.ss0 = 0x10;
    ...
}
```

不一样的值，一部分是 **state**，**pid**，**counter** 这种**进程的元信息**，另一部分是 **tss** 里面保存的各种寄存器的信息，即**上下文**。

这里有两个寄存器的值的赋值有些特殊，就是 ss0 和 esp0，这个表示 0 特权级也就是内核态时的 ss:esp 的指向。

根据代码我们得知，其含义是将代码在内核态时使用的堆栈栈顶指针指向进程 task_struct 所在的 4K 内存页的最顶端，而且之后的每个进程都是这样被设置的。



好了，进程槽位的申请，以及基本信息的复制，就讲完了。

今天就这么点内容，**就是内存中找个地方存一个 task_struct 结构的东东，并添加到 task[] 数组里的空闲位置处，这个东东的具体字段赋值的大部分都是复制原来进程的。**

接下来将是进程页表和段表的复制，这将会决定进程之间的内存规划问题，很是精彩，也是 fork 真正的难点所在。

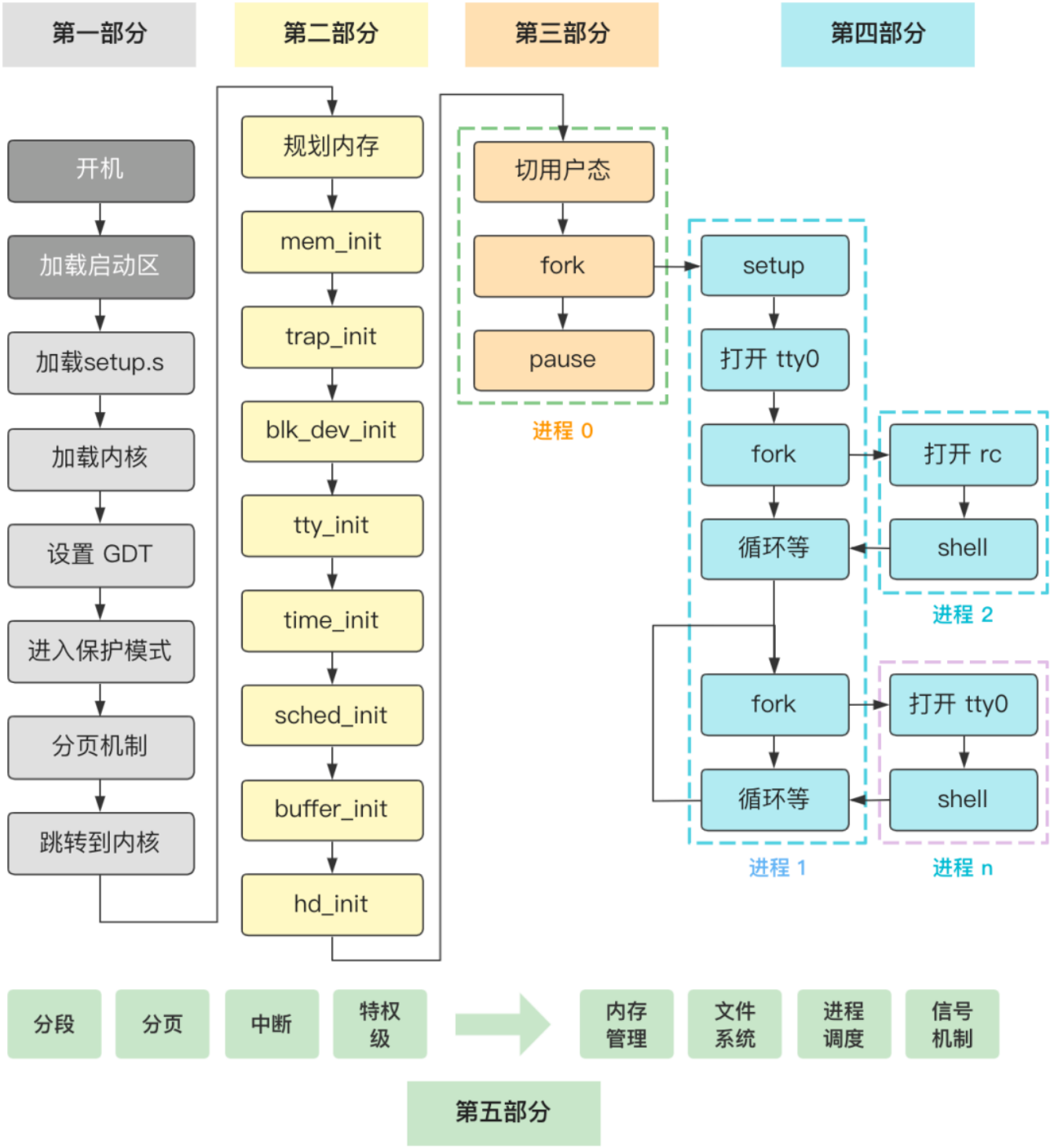
欲知后事如何，且听下回分解。

------- 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，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**在看**我也会很开心，我相信星火燎原的力量，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
一个新进程的诞生（五）通过 fork 看一次系统调用

下一篇
一个新进程的诞生（七）透过 fork 来看进程的内存规划

Read more