第四部分完结！操作系统启动完毕！

Original 闪客 低并发编程 2022-06-05 17:30 Posted on 北京

收录于合集
#操作系统源码

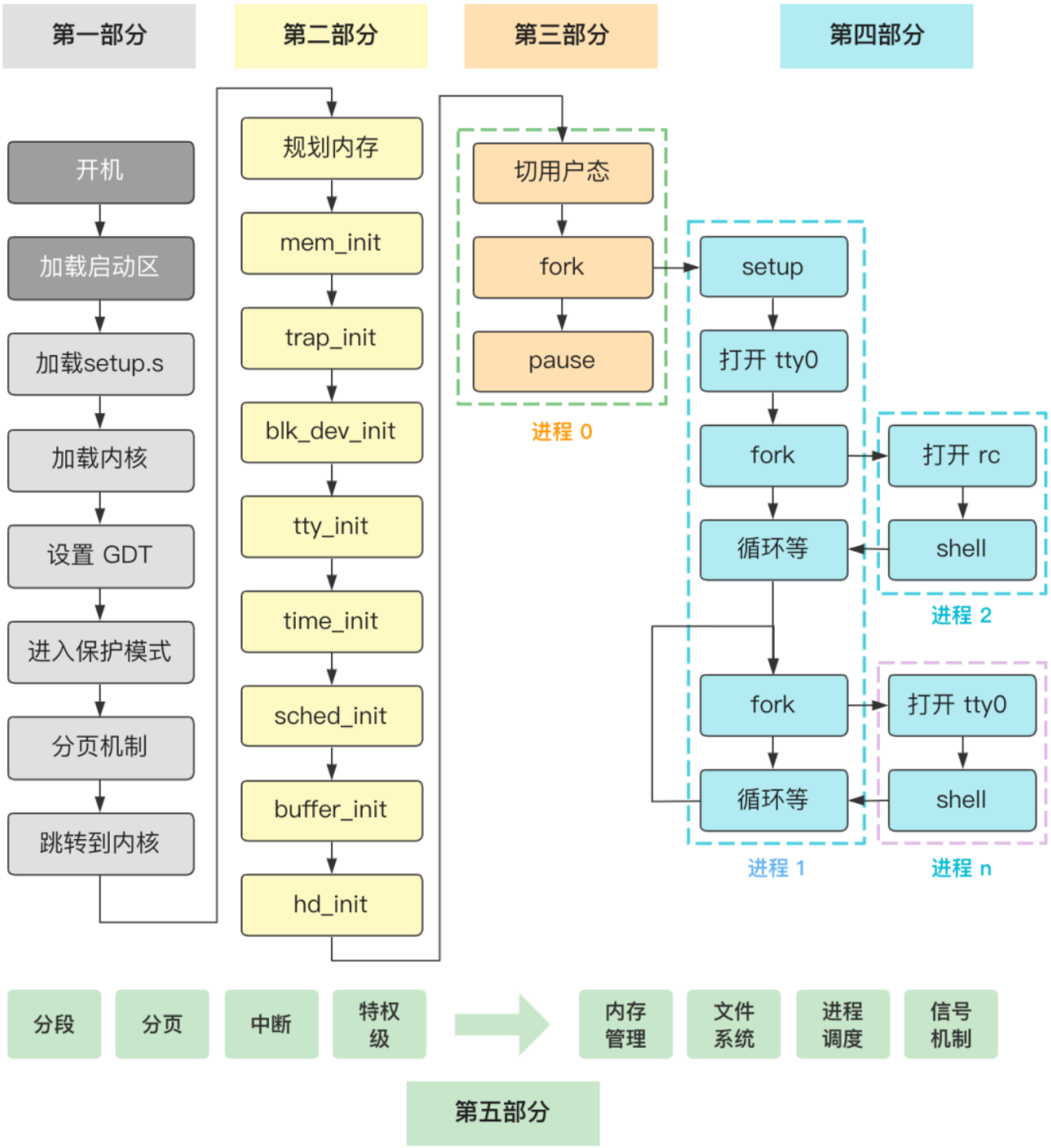
43个

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。

本系列的 GitHub 地址如下，希望给个 star 以示鼓励（文末**阅读原文**可直接跳转，也可以将下面的链接复制到浏览器里打开）
https://github.com/sunym1993/flash-linux0.11-talk

本回的内容属于第四部分。



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

第三部分：一个新进程的诞生

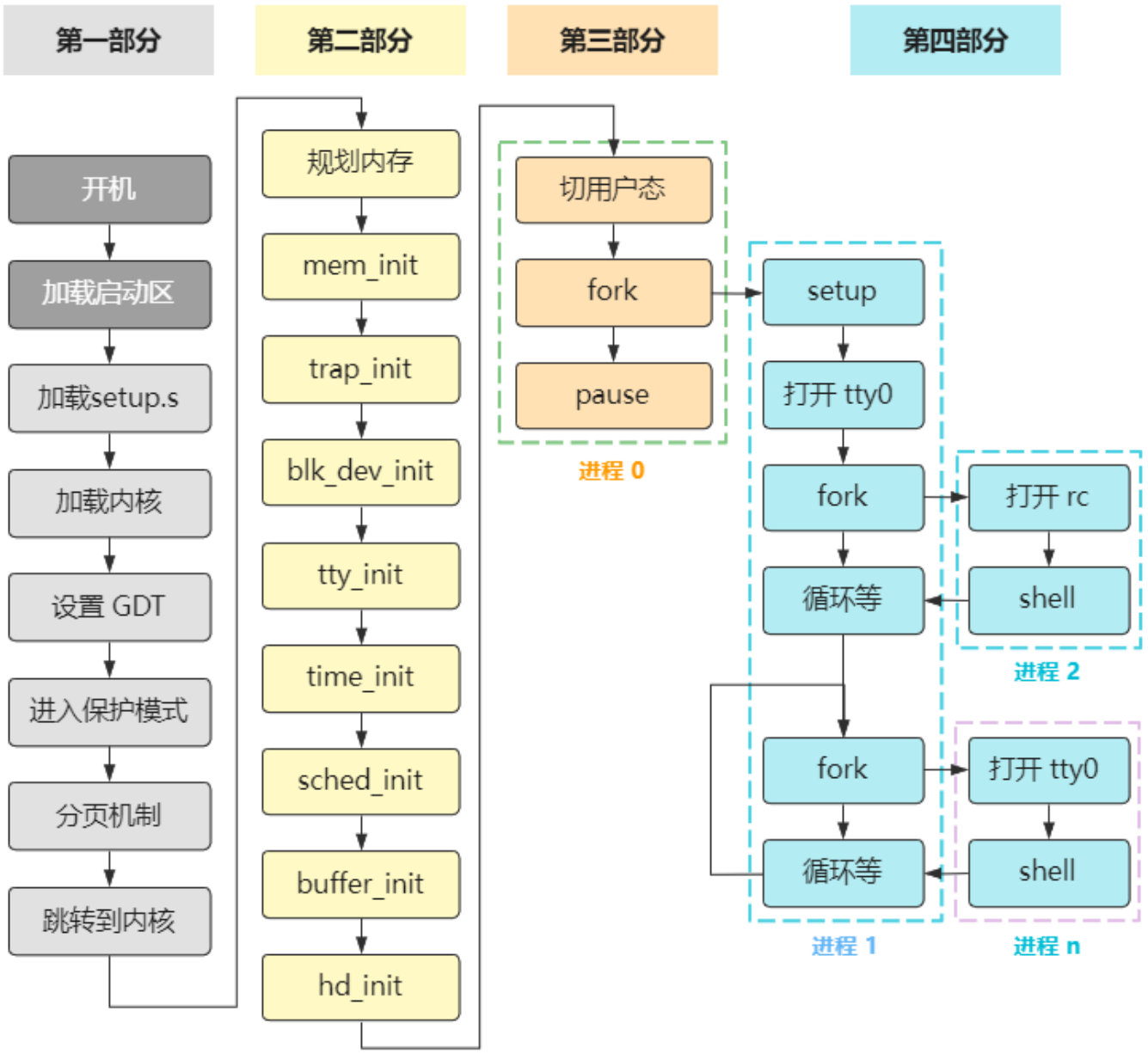
第四部分：shell 程序的到来

第五部分：一条 shell 命令的执行

持续更新中...

------- 正文开始 -------

整个操作系统终于通过四个部分的讲解，完成了它的启动，达到了一个怠速状态，留下了一个 shell 程序等待用户指令的输入并执行。



具体来说。

通过 第一部分 | 进入内核前的苦力活 完成了执行 main 方法前的准备工作，如加载内核代码，开启保护模式，开启分页机制等工作，对应内核源码中 boot 文件夹里的三个汇编文件 **bootsect.s setup.s head.s**。

通过 第二部分 | 大战前期的初始化工作 完成了内核中各种管理结构的初始化，如内存管理结构初始化 mem_init，进程调度管理结构初始化 shed_init 等，对应 main 方法中的 **xxx_init** 系列方法。

通过 第三部分 | 一个新进程的诞生 讲述了 fork 函数的原理，也就是进程 0 创建进程 1 的过

程，对应 main 方法中的 **fork** 函数。

通过 第四部分 | shell 程序的到来 讲述了从加载根文件系统到最终创建出与用户交互的 shell 进程的过程，对应 main 方法中的 init 函数。

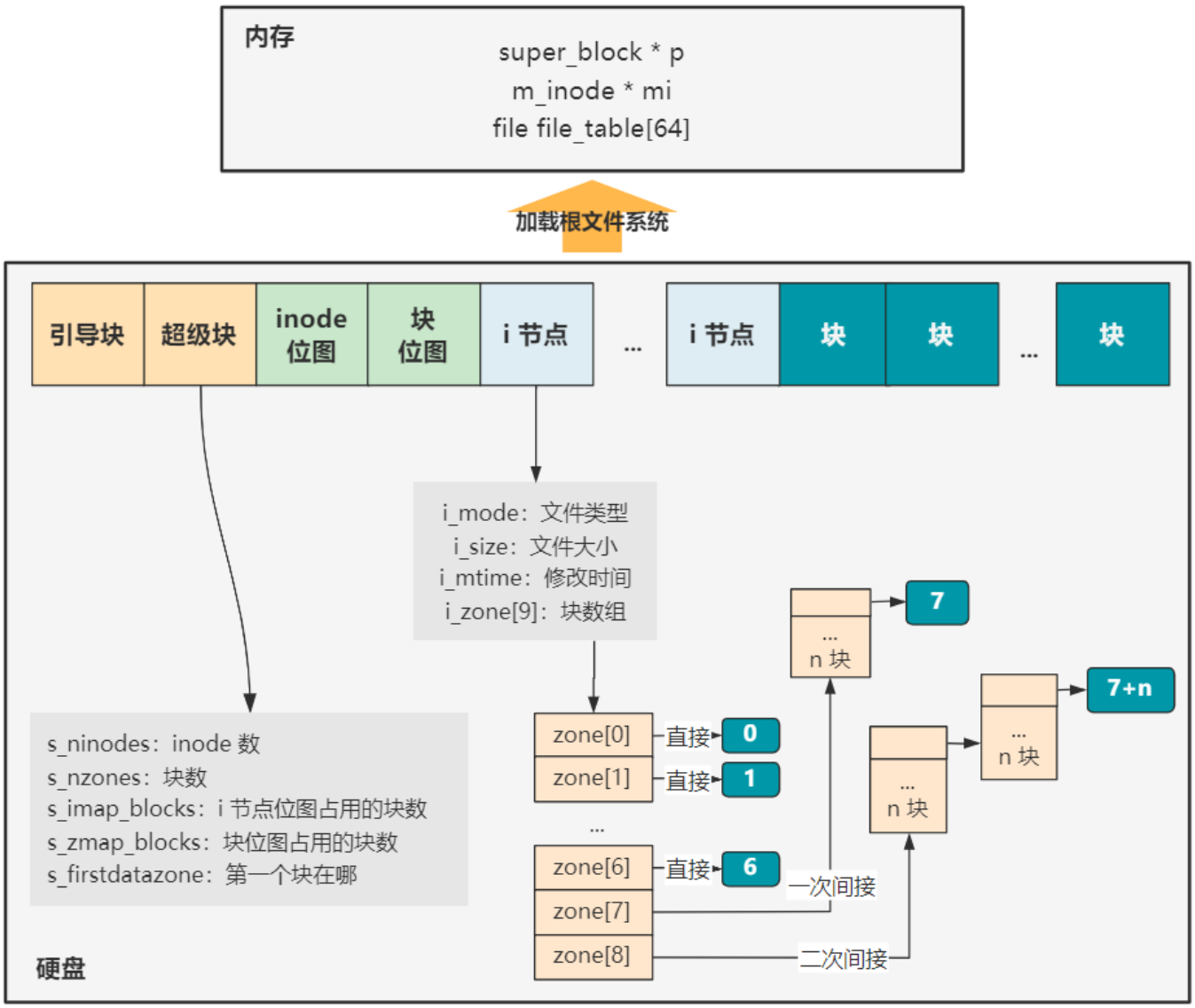
至此操作系统启动完毕，达到怠速状态。

纵观整个操作系统的源码，前四部分对应的代码如下，这就是启动流程中的全部代码了。

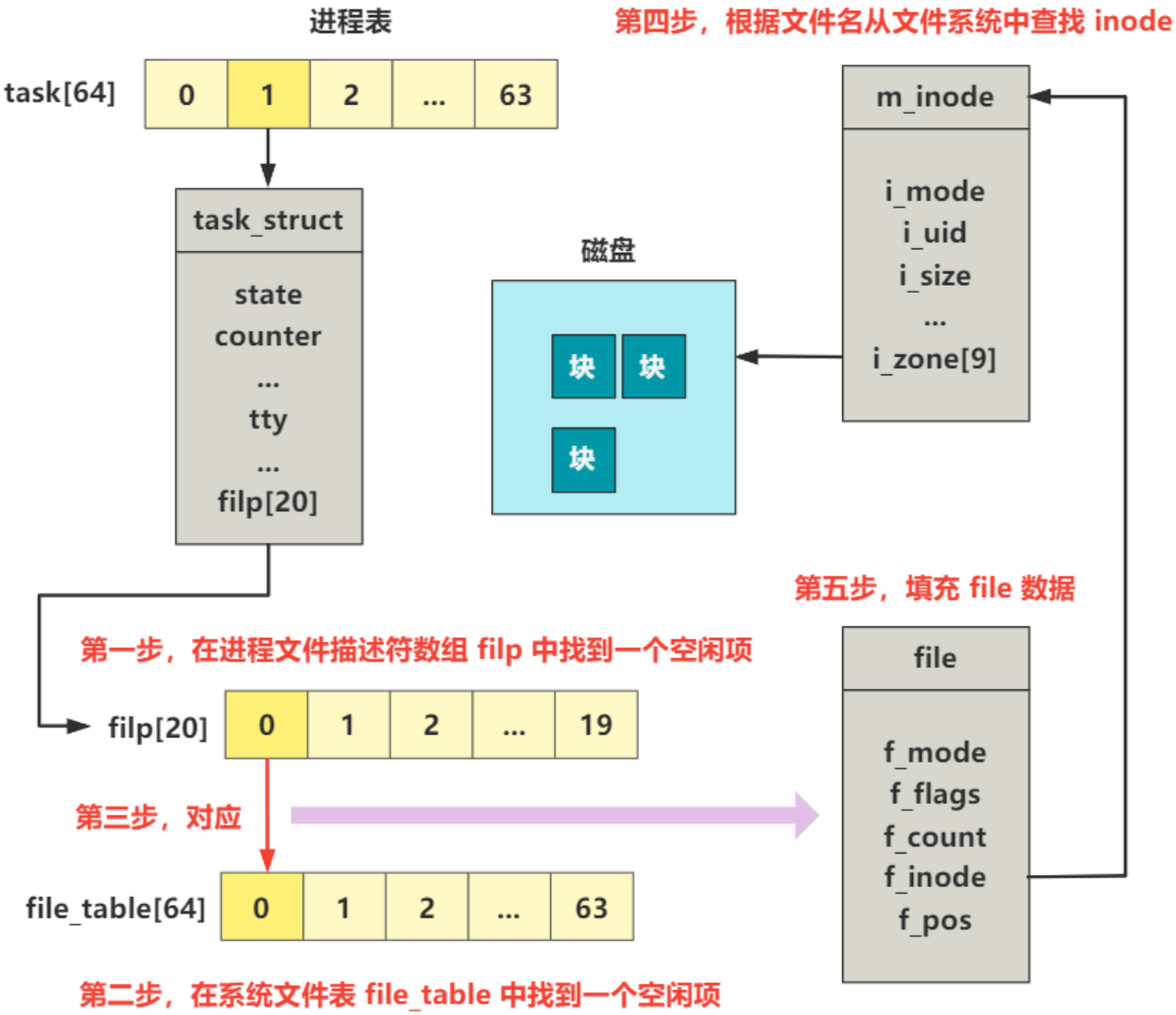
```
--- 第一部分 进入内核前的苦力活 ---
bootsect.s
setup.s
head.s

main.c
void main(void) {
--- 第二部分 大战前期的初始化工作 ---
    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();
--- 第三部分 一个新进程的诞生 ---
    move_to_user_mode();
    if (!fork()) {
--- 第四部分 shell程序的到来 ---
        init();
    }
    for(;;) pause();
}
```

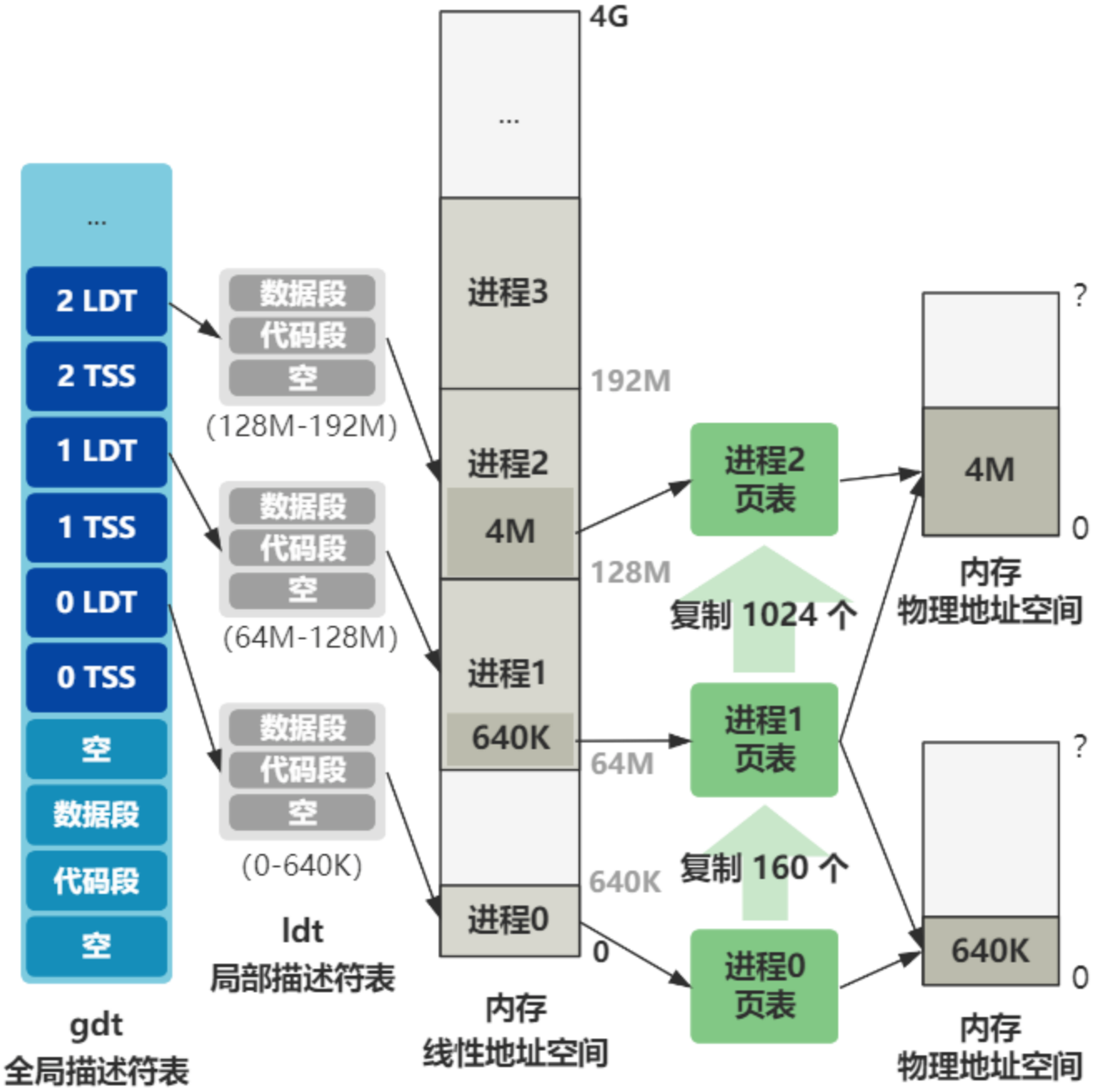
具体展开第四部分，我们首先通过 第31回 | 拿到硬盘信息 和 第32回 | 加载根文件系统 使得内核具有了以**文件系统**的形式管理硬盘中的数据的能力。



接下来 第33回 | 打开终端设备文件 使用刚刚建立好的文件系统能力，打开了 /dev/tty0 这个终端设备文件，此时内核便具有了**与外设交互的能力**，具体可以体现为调用 printf 函数可以往屏幕上打印字符串了。

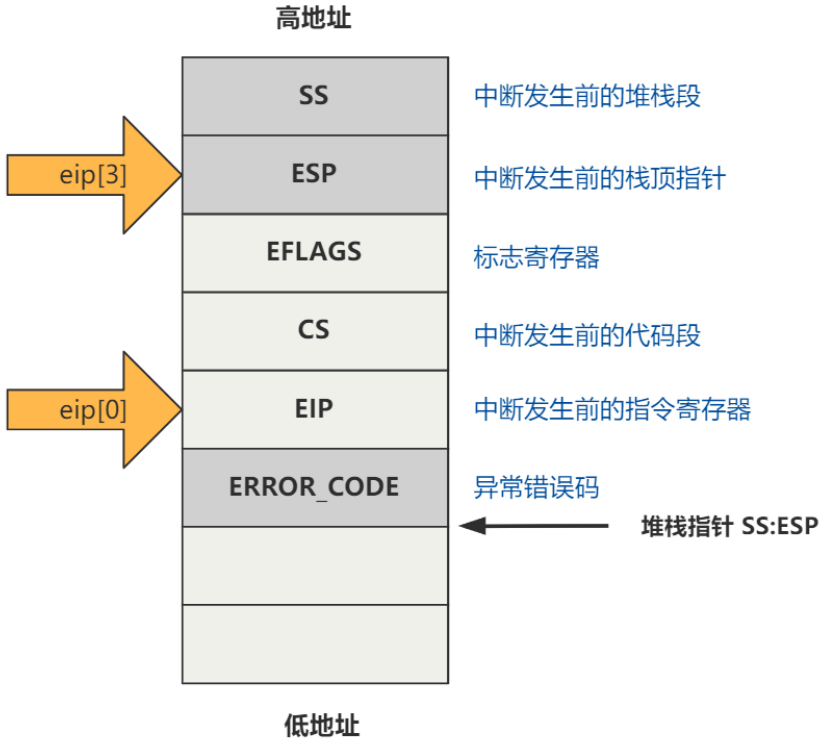


再接下来，第34回 | 进程2的创建 利用刚刚建立好的文件系统，以及进程 1 的与外设交互的能力，创建出了进程 2，此时进程 2 与进程 1 一样也具有与外设交互的能力，这为后面 shell 程序的创建打好了基础。

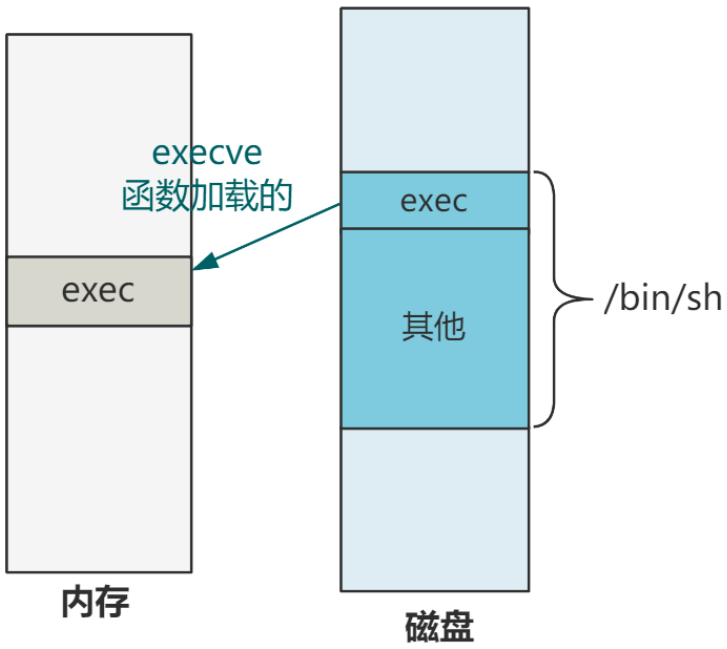


然后，进程 2 此时摇身一变，在 第35回 | execve 加载并执行 shell 程序 利用 **execve** 函数使自己变成了 shell 程序，配合上一回 fork 的进程 2 的过程，这就是 Linux 里经典的 **fork + execve** 函数。

execve 函数摇身一变的关键，其实就是改变了栈空间中的 **EIP** 和 **ESP** 的值，使得中断返回后的地址被程序进行了魔改，改到了 shell 程序加载到的内存地址上。



此时，execve 系统调用的中断返回后，指向了 shell 程序所在的内存地址起始处，就要开始执行 shell 程序了。但此时 shell 程序还没有从硬盘中加载到内存呢，所以此时会触发**缺页中断**，将硬盘中的 shell 程序（除 exec 头部的其他部分）按需加载到内存，这就是 第36回 | 缺页中断 里讲述的过程。



这回，终于可以开始执行 **shell** 程序了，在 第37回 | shell 程序跑起来了 中我们以 xv6 源码

中的超级简单的 shell 程序源码为例，讲解了 shell 程序的原理。

就是不断读取我们用户输入的命令，创建一个新的进程并执行刚刚读取到的命令，最后等待进程退出，再次进入读取下一条命令的循环中。

```
// xv6-public sh.c
int main(void) {
    static char buf[100];
    // 读取命令
    while(getcmd(buf, sizeof(buf)) >= 0){
        // 创建新进程
        if(fork() == 0)
            // 执行命令
            runcmd(parsecmd(buf));
        // 等待进程退出
        wait();
    }
}
```

shell 程序是个死循环，我们再回过头来看操作系统的死循环。

在 第38回 | 操作系统启动完毕 中给出了整个操作系统启动代码的鸟瞰视角。

```
// main.c
void main() {
    // 初始化环境
    ...
    // 外层操作系统大循环
    while(1) {
        // 内层 shell 程序小循环
        while(1) {
            // 读取命令 read
            ...
            // 创建进程 fork
            ...
            // 执行命令 execve
            ...
        }
    }
}
```

可以看出，不仅 shell 程序是个死循环，整个操作系统也是个死循环。

除此之外，这里所有的键盘输入、系统调用、进程调度，统统都需要**中断**来驱动，所以很久之前我说过，**操作系统就是个中断驱动的死循环**，就是这个道理。

OK！到此为止，操作系统终于启动完毕，达到了怠速的状态，它本身设置好了一堆中断处理程序，随时等待着中断的到来进行处理，同时它运行了一个 shell 程序用来接受我们普通用户的命令，以同人类友好的方式进行交互。

我们前四个部分，终于把整个操作系统的启动流程讲述清楚了，如果你头脑中已经有像过电影般把整个启动流程清晰地印在脑子里，相信你已经不再恐惧操作系统源码了。

但理解操作系统不单单是启动流程这个视角，还需要**内存管理、文件系统、进程调度、设备管理、系统调用**等操作系统提供的功能的视角看。

启动流程是一次性的，就这么来一下子，而这些功能是持续不断的，用户程序不断通过系统调用和操作系统提供的这些功能，完成自己想要让计算机帮忙做的事情。

所以接下来的第五部分，我打算用一条 shell 命令的执行过程，来把操作系统这些模块和所提供的功能讲述清楚。

因为一条 shell 命令的执行，包括了内存管理、文件系统、进程调度、设备管理、中断控制、特权级切换等等各方面的内容，实在是把它们都串起来的好办法。

那接下来就跟我一起，期待第五部分的到来吧！

欲知后事如何，且听下回分解。

------- 关于本系列 -------

本系列的开篇词看这里，开篇词

本系列的番外故事看这里，让我们一起来写本书？也可以直接无脑加入星球，共同参与这场旅行。

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 · 第38回 | 操作系统启动完毕！

今天我下了个JDK

低并发编程

MySQL 启停过程了解一二

GreatSQL社区