

第二部分完结撒花！大战前期的初始化工作

Original 闪客 低并发编程 2022-01-26 16:30

收录于合集

#操作系统源码

43个

目录在此

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

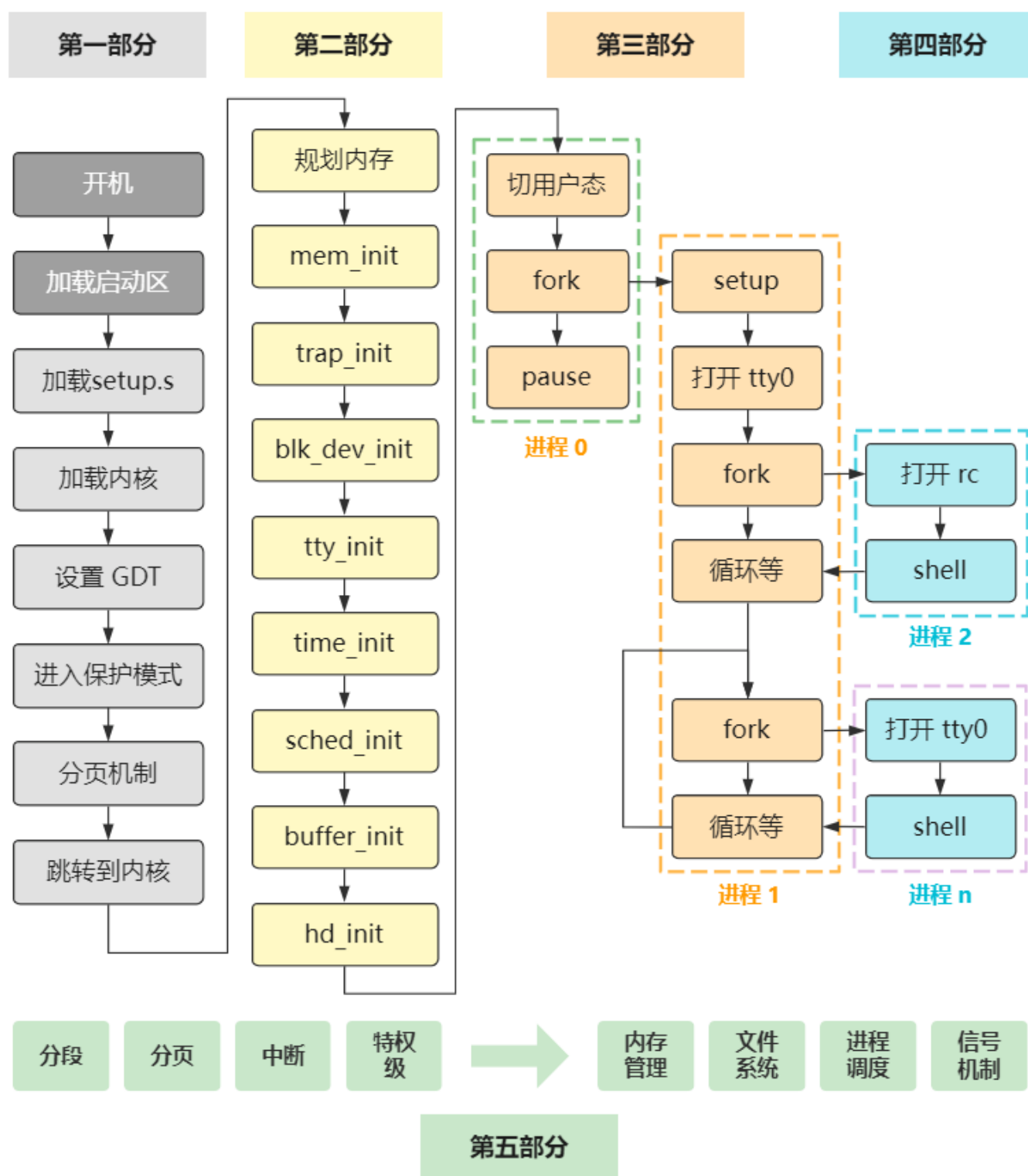
第17回 | 时间初始化 time_init

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

这张图展示了整个系列的结构



那我们今天就来给第二部分做个梳理。

第二部分所讲的代码，就和第二部分的目录一样规整，一个 `init` 方法对应一个章节，简单粗暴。

```
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();  
}
```

如果坚持到这里了，先给自己鼓鼓掌！

这个过程，你可能觉得无聊，因为全是各种数据结构、中断、外设的初始化工作，后面将会怎么用它们，并没有展开讲解。

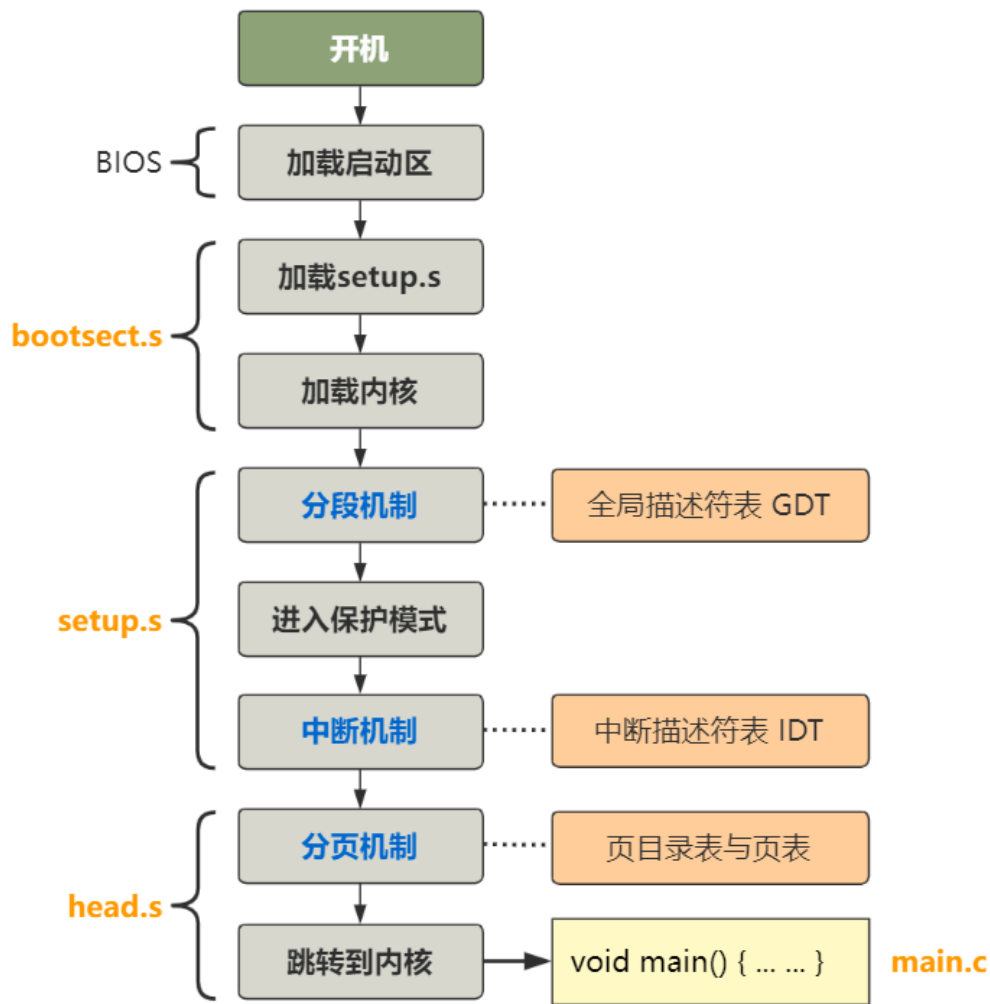
但你也可能觉得兴奋，因为后面操作系统的全部工作，都是围绕着这几个初始化了的结构展开的，而它们却都是那么的好理解。

其实我是蛮喜欢这个过程的，比如我看电影，其实我对高潮部分并不是很感兴趣，我就喜欢看一场大战或者一场阴谋前各部门的准备工作，看着它们为了后面一个完美的计划，所做的前期筹备，是一种享受，你懂的！

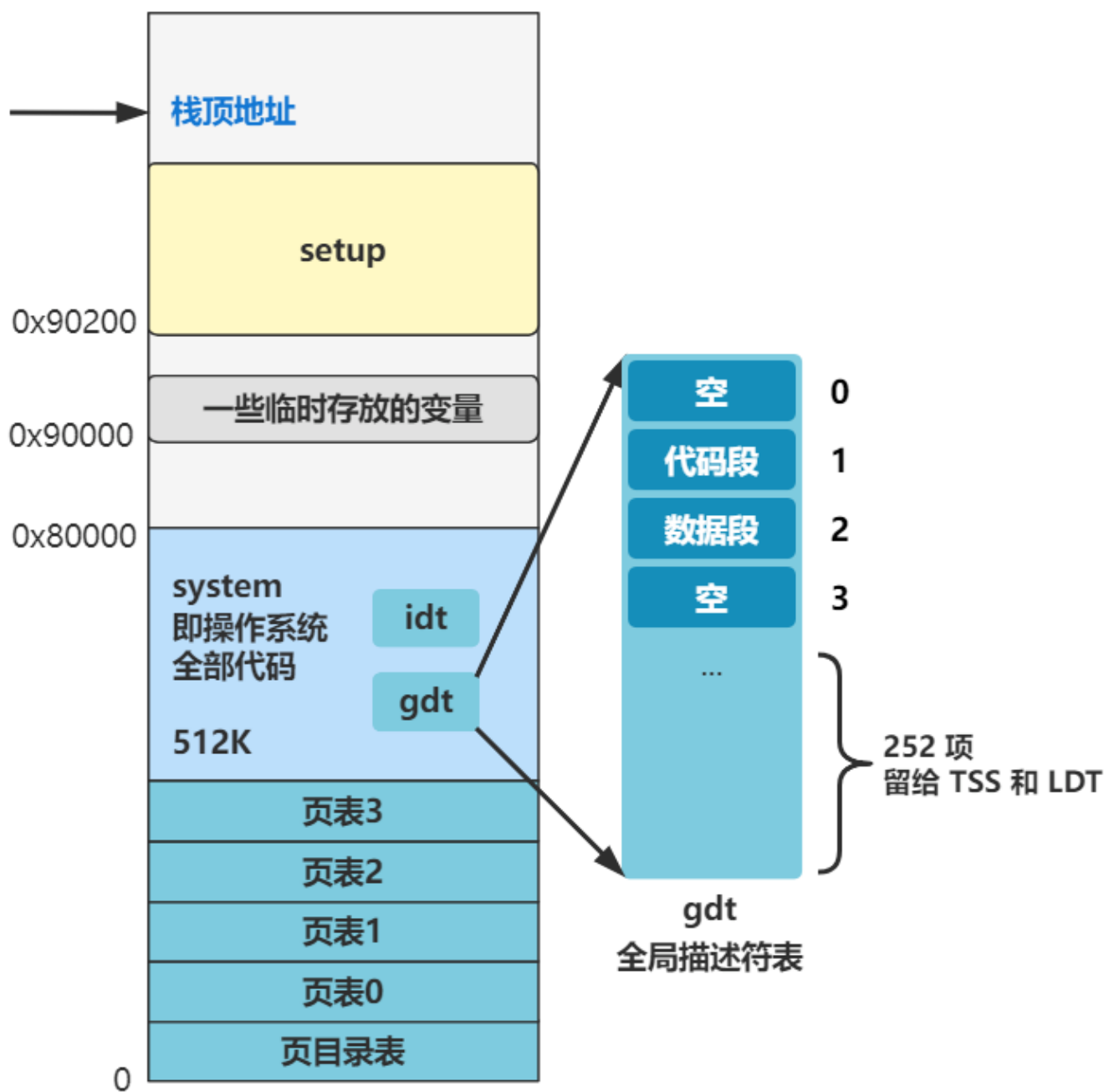
所以今天特地花一章的功夫，把之前的初始化工作梳理一遍，之前没仔细看的同学，这章是个重新开始的机会！

----- 开始 -----

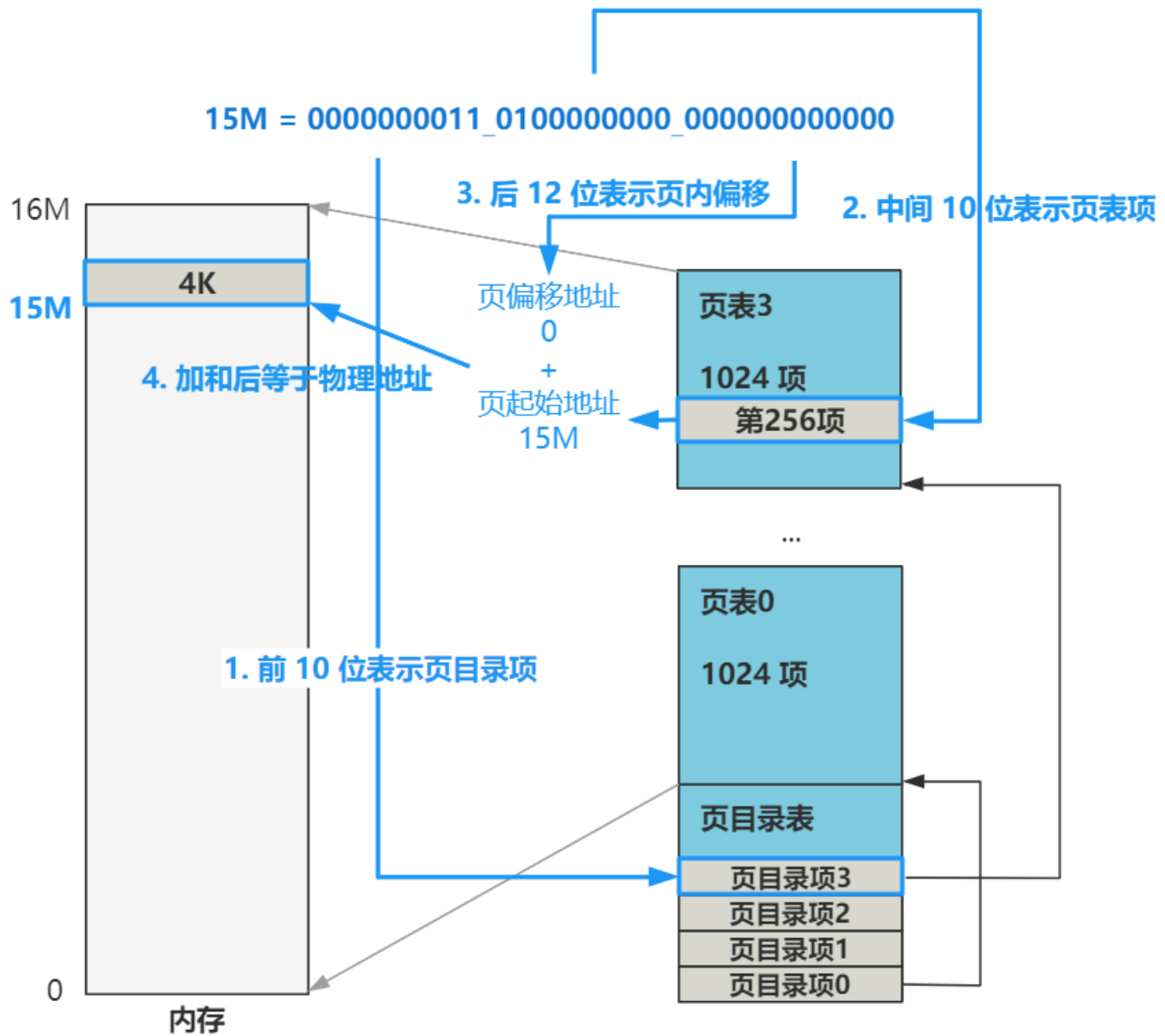
电脑开机后，首先由 BIOS 将操作系统程序加载到内存，之后在进入 main 函数前，我们用汇编语言（boot 包下的三个汇编文件）做了好多苦力活。



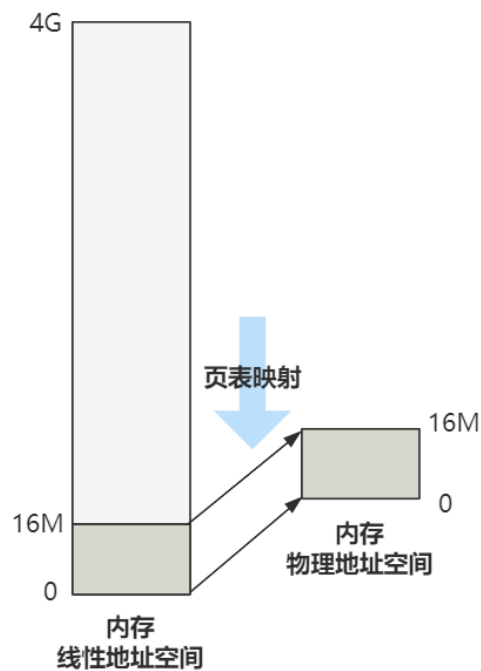
这些苦力活做好后，内存布局变成了这个样子。



其中页表的映射关系，被做成了线性地址与物理地址相同。



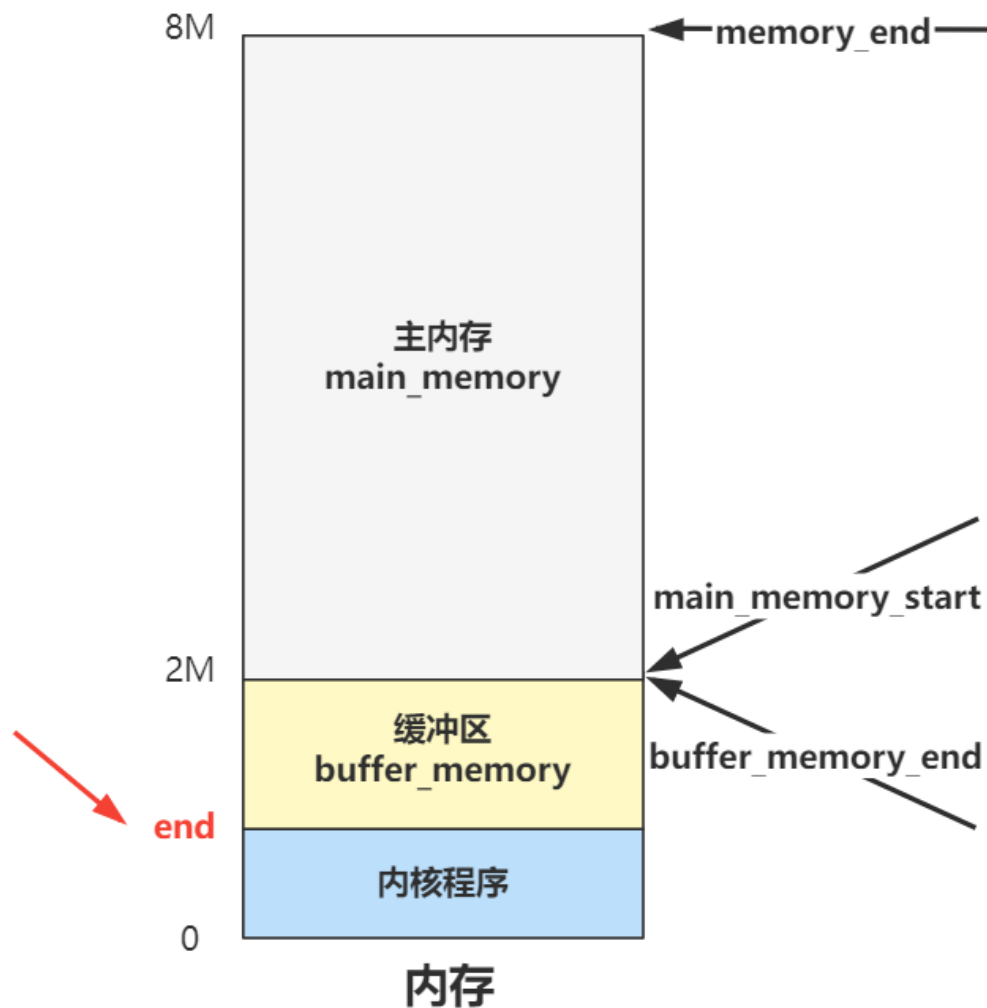
也因为有了页表的存在，所以多了线性地址空间的概念，即经过分段机制转化后，分页机制转化前的地址，不考虑段限长的话，32 位的 CPU 线性地址空间应为 4G。



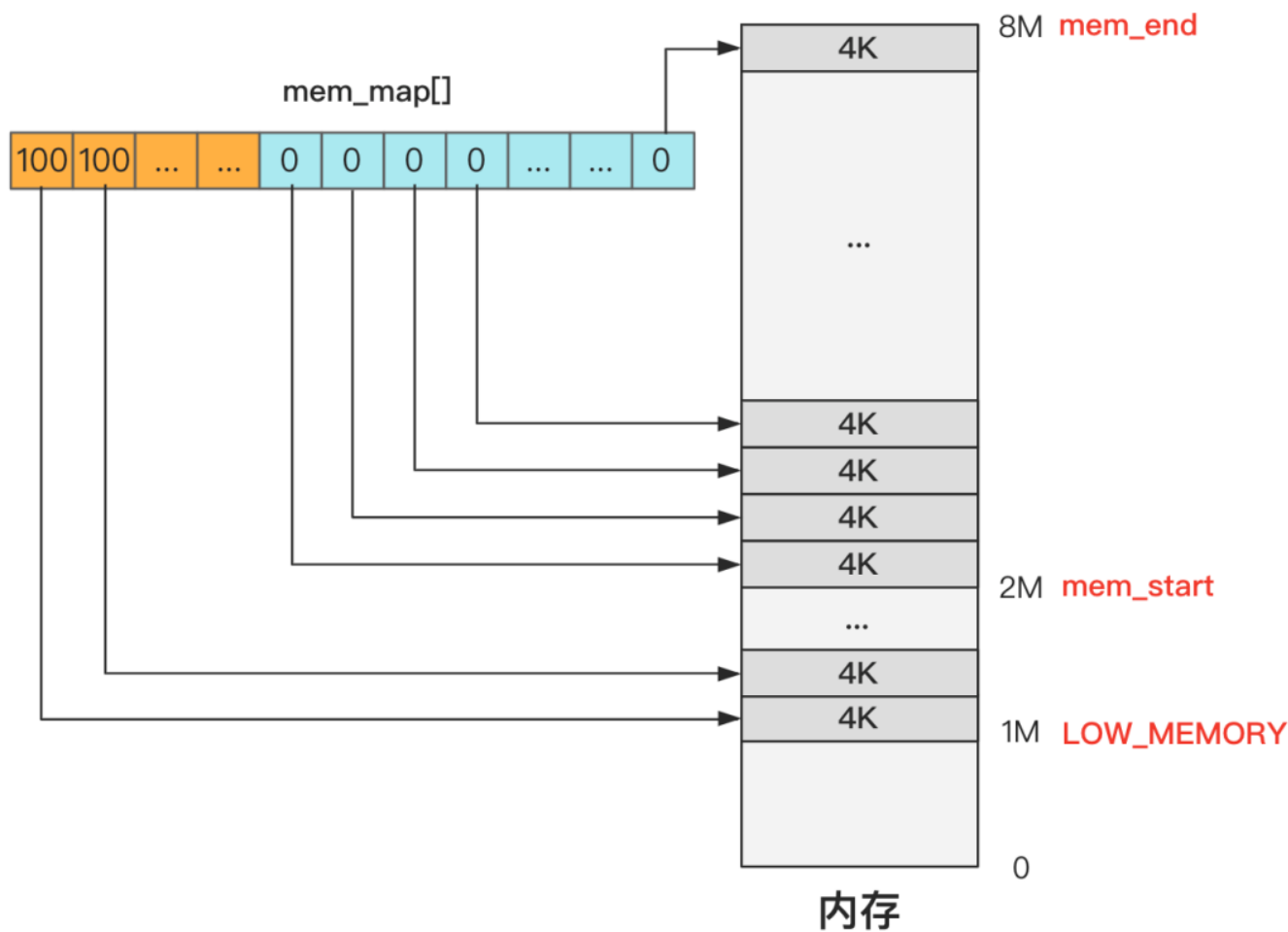
以上这些，是进入 main 函数之前的事情，由 boot 文件夹下的三个汇编文件完成，具体可以看整个第一部分的总结：第一部分完结 进入内核前的苦力活

----- 进入 main 函数后 -----

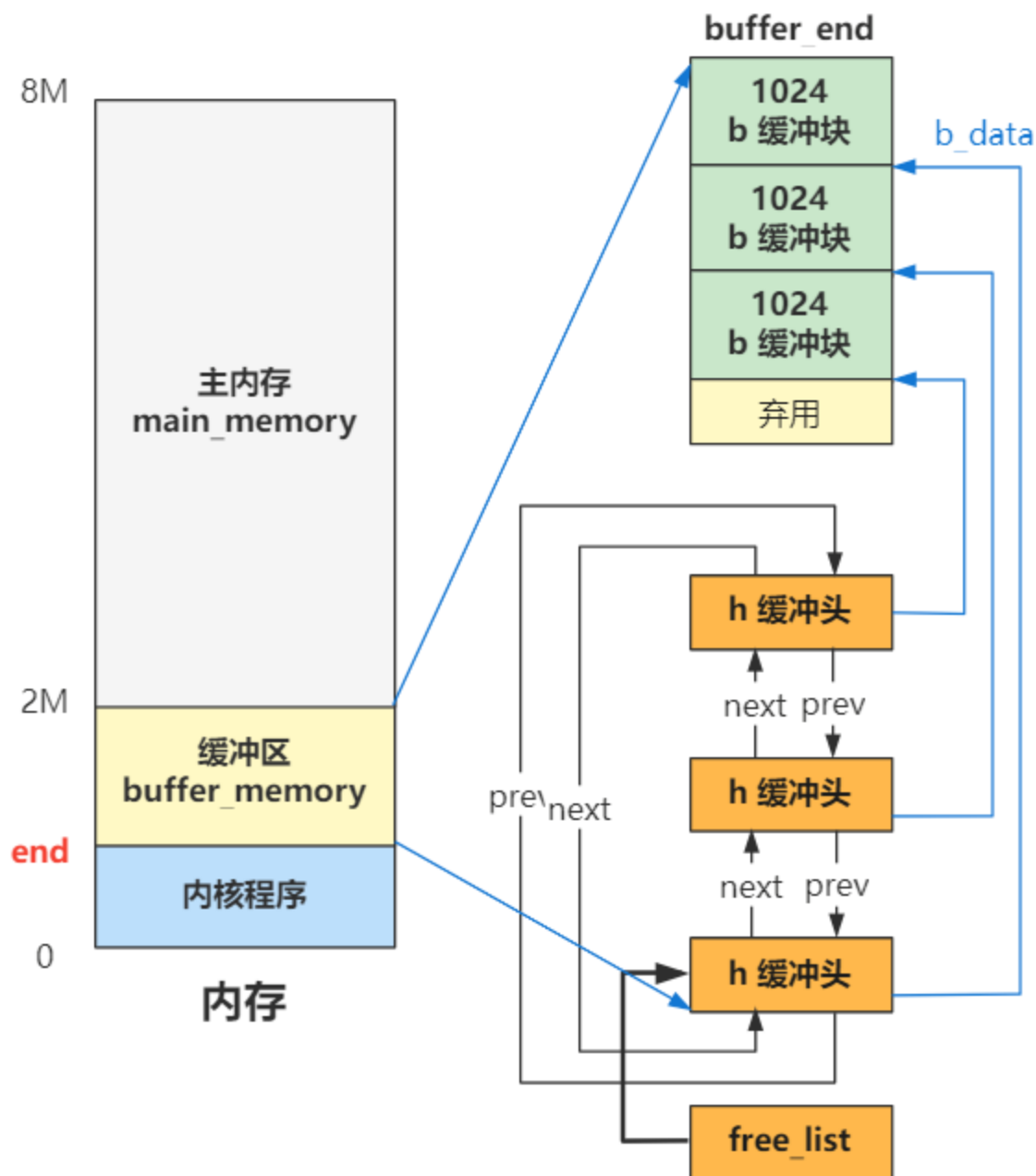
进入 main 函数后，首先进行了内存划分，其实就是设置几个边界值，将内核程序、缓冲区、主内存三个部分划分开界限。这就是 第12回 | 管理内存前先划分出三个边界值 所做的事情。



随后，通过 `mem_init` 函数，对主内存区域用 `mem_map[]` 数组管理了起来，其实就是每个位置表示一个 4K 大小的内存页的使用次数而已，今后对主内存的申请和释放，其实都是对 `mem_map` 数组的操作。这是 [第13回 | 主内存初始化 mem_init](#) 所做的事。

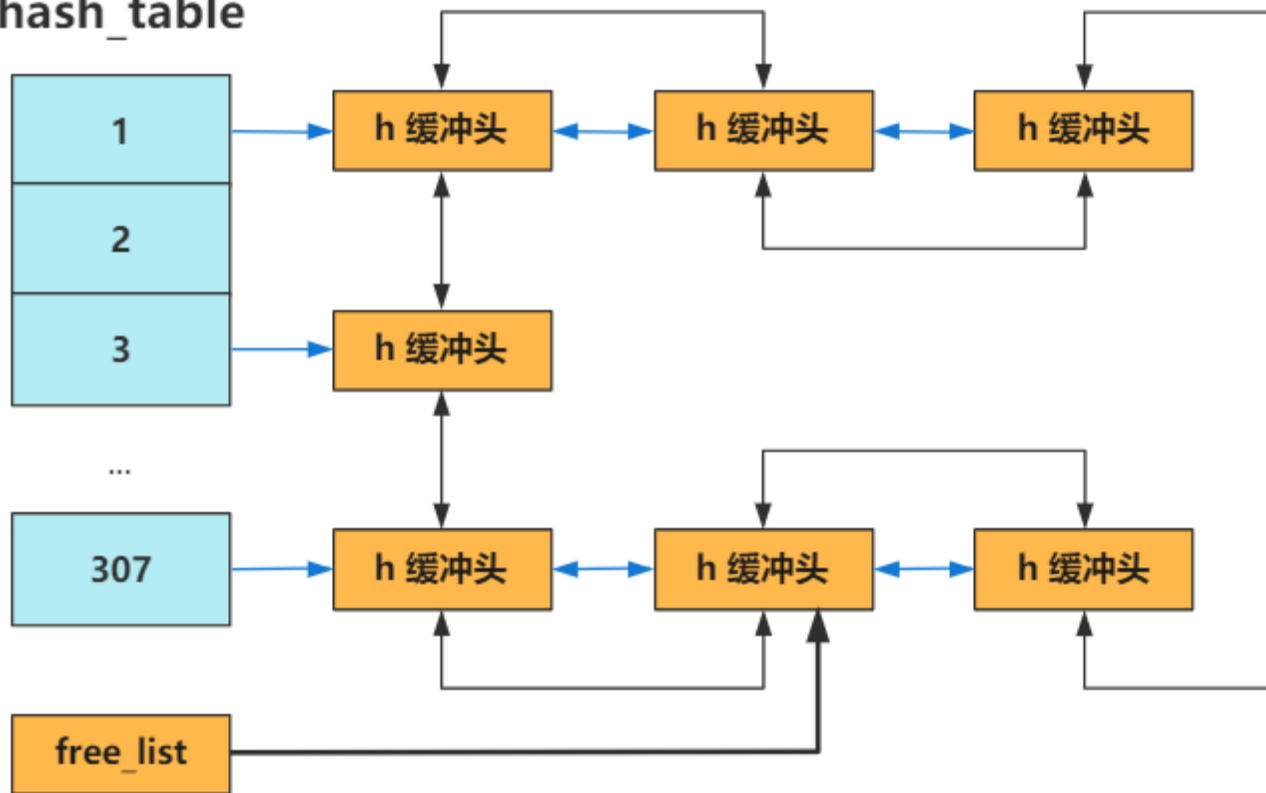


后面又通过 `buffer_init` 函数，对缓冲区区域用多种数据结构管理起来。其中包括双向链表缓冲头 `h` 和每个缓冲头管理的 1024 字节大小的缓冲块 `b`。这是 [第19回 | 缓冲区初始化](#) `buffer_init` 的内容。



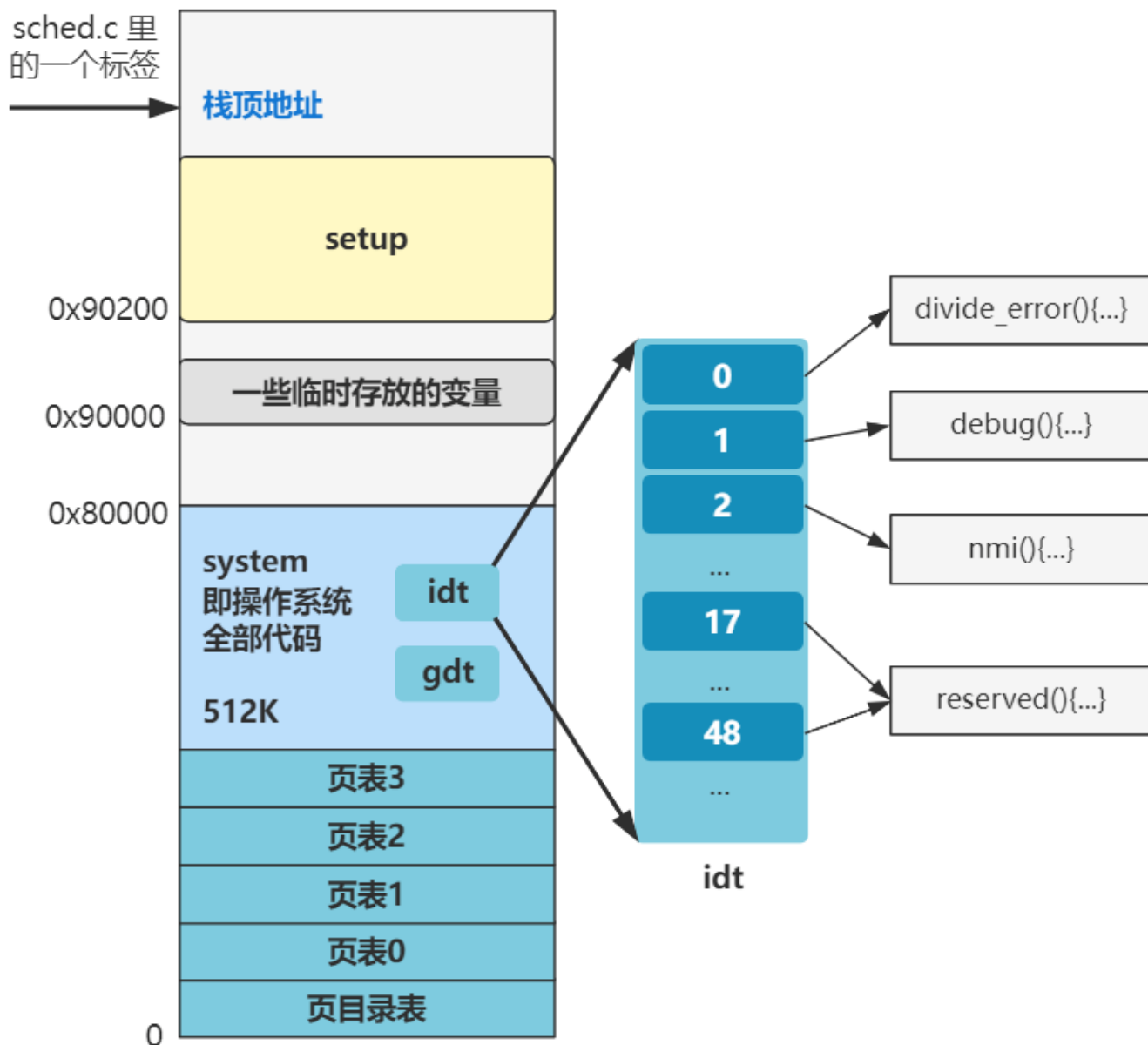
同时，又用一个 hashmap 结构，索引到所有缓冲头，方便快速查找，为之后的通过 LRU 算法使用缓冲区做准备。

hash_table

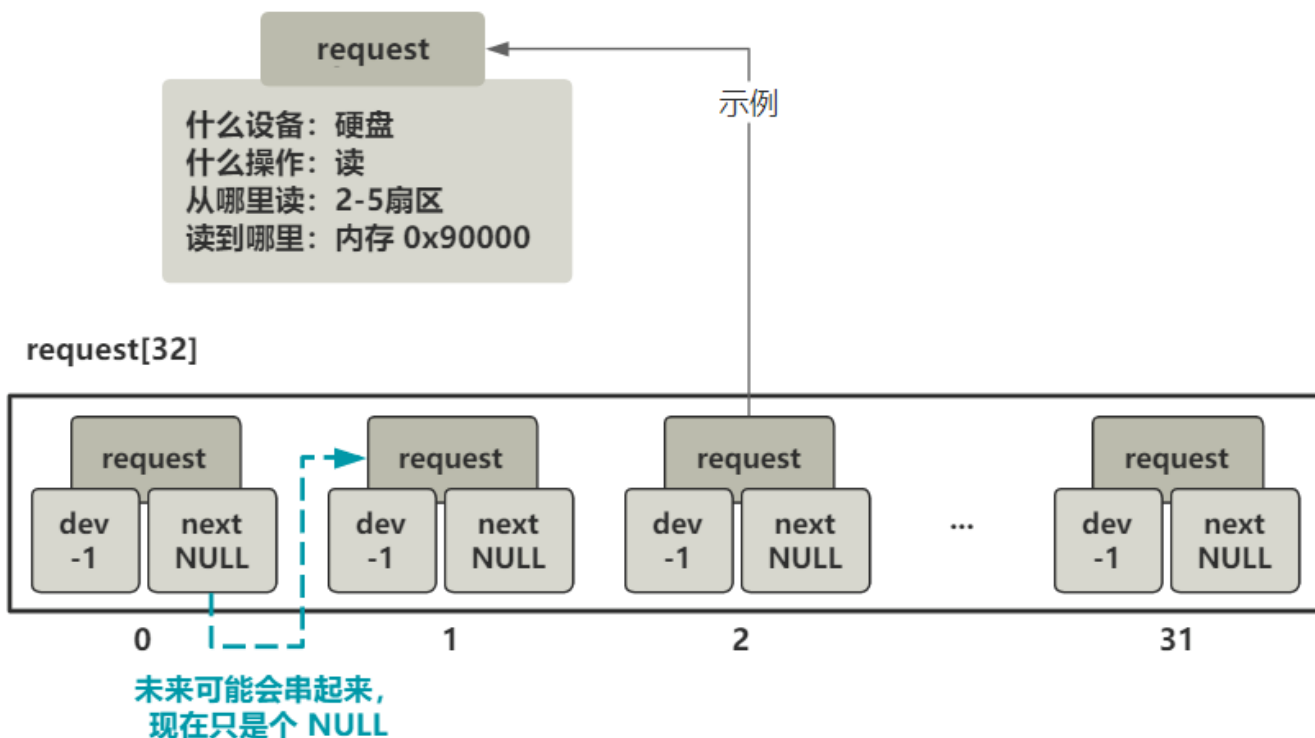


这些结构，就是缓冲区部分的管理，而缓冲区的目的是为了加速磁盘的读写效率，后面将读写文件全流程的时候，你会看到它在整个流程中起到中流砥柱的作用。

再往后，通过 `trap_init` 函数把中断描述符表的一些默认中断都设置好了，随后再由各个模块设置它们自己需要的个性化的中断（比如硬盘中断、时钟中断、键盘中断等）。这是 第14回 | 中断初始化 `trap_init` 的内容。



再之后，通过 `blk_dev_init` 对读写块设备（比如硬盘）的管理进行了初始化，比如对硬盘的读写操作，都要封装为一个 `request` 结构放在 `request[]` 数组里，后面用电梯调度算法进行排队读写硬盘。这是 第15回 | 块设备请求项初始化 `blk_dev_init` 的内容。



再往后，通过 `tty_init` 里的 `con_init`，实现了在控制台输出字符的功能，并且可以支持换行、滚屏等效果。当然此处也开启了键盘中断，如果此时中断已经处于打开状态，我们就可以用键盘往屏幕上输出字符啦。这是 第16回 | 控制台初始化 `tty_init` 的内容。

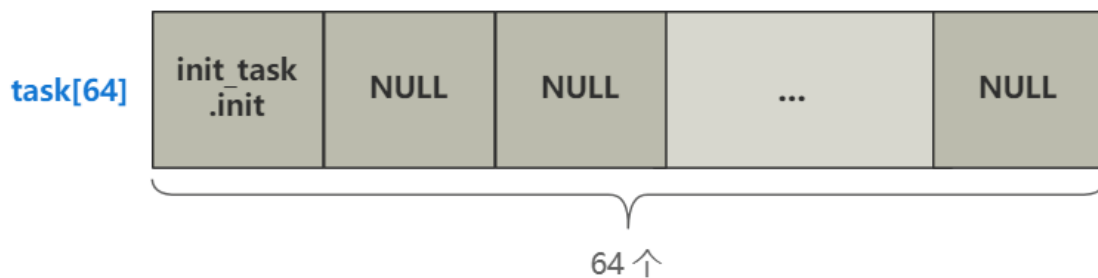
```
<-- keyboard_interrupt

[input] _
```

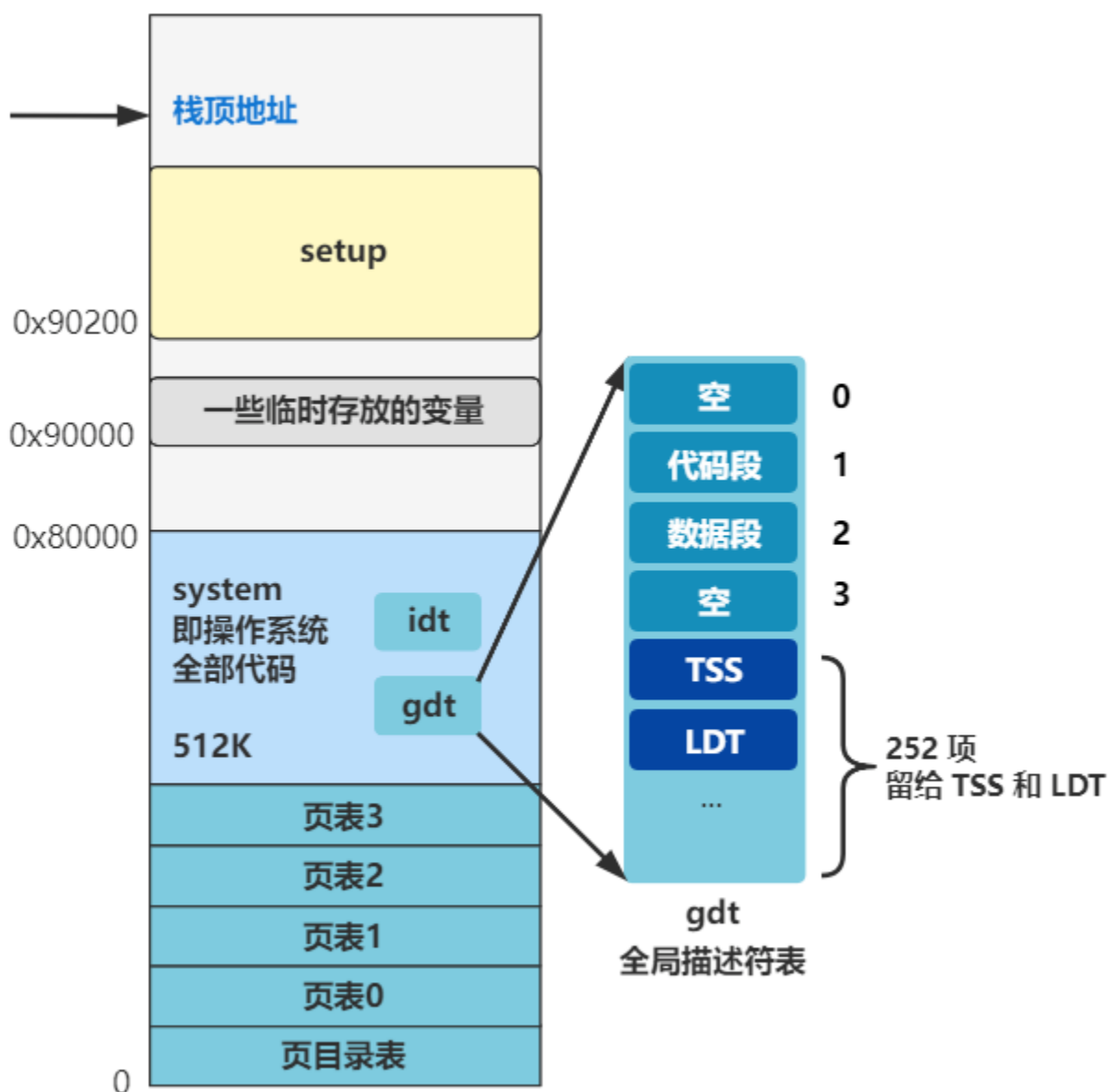


再之后，整个操作系统的精髓，进程调度，其初始化函数 `shed_init`，定义好了全部进程的管

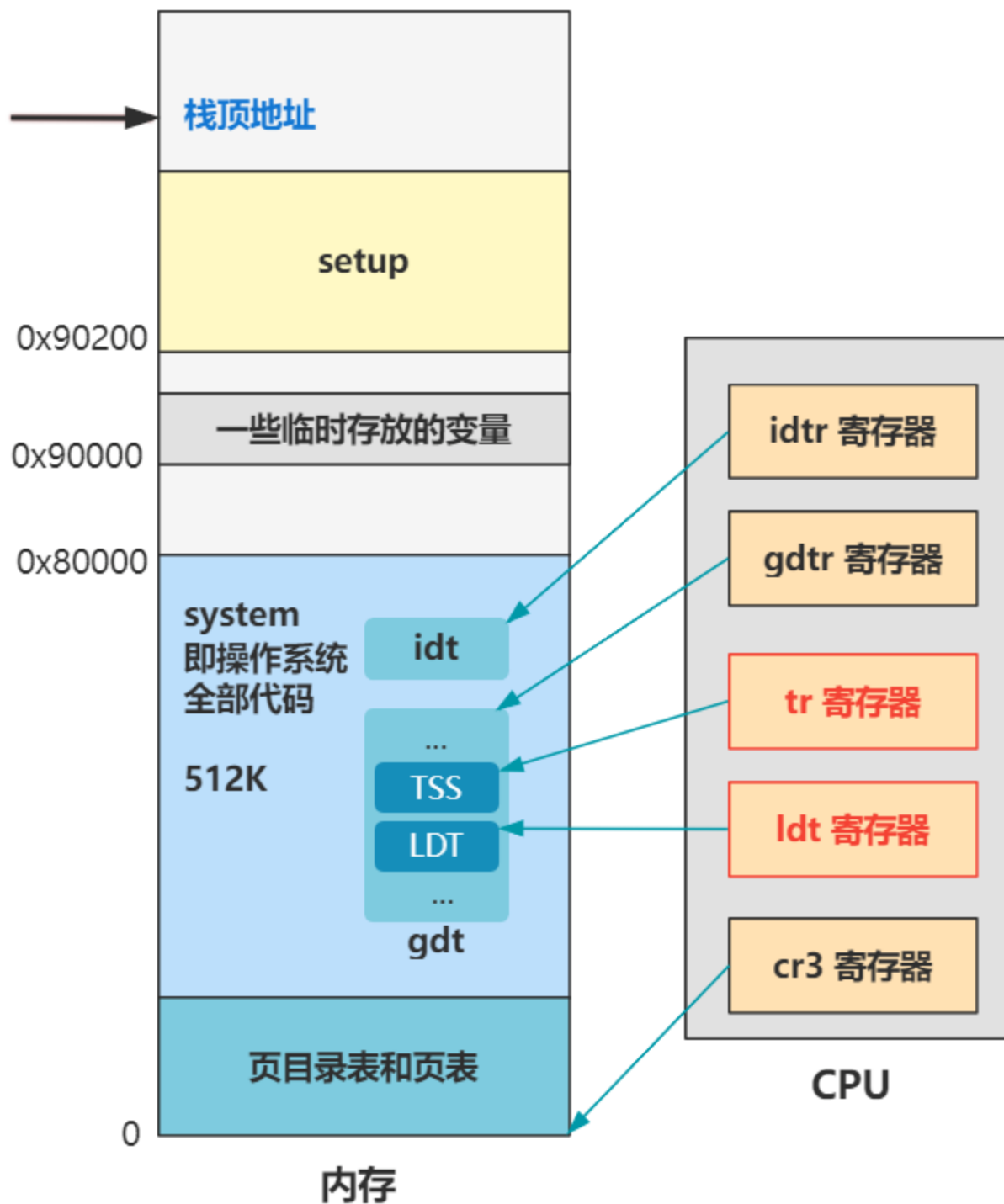
理结构 `task[64]` 数组，并在索引 0 位置处赋上了初始值，作为零号进程的结构体。这是 第 18回 | 进程调度初始化 `sched_init` 的内容。



然后将全局描述符表增添了 TSS 和 LDT，用来管理 0 号进程的上下文信息以及内存规划，结构里面具体是什么，先不用管哟。



同时，将这两个结构的地址，告诉 tr 寄存器和 ldt 寄存器，让 CPU 能够找到它们。



随后，开启定时器，以及设置了时钟中断，用于响应定时器每隔 100ms 发来的中断信号。



这样就算把进程调度的初始化工作完成了，之后进程调度就从定时器发出中断开始，先判断当前进程时间片是不是到了，如果到了就去 `task[64]` 数组里找下一个被调度的进程的信息，切换过去。

这就是进程调度的简单流程，也是后面要讲的一个非常精彩的环节。

最后最后，一个简单的硬盘初始化 `hd_init`，为我们开启了硬盘中断，并设置了硬盘中断处理函数，此时我们便可以真正通过硬盘的端口与其进行读写交互了。这是 [第20回 | 硬盘初始化 `hd_init` 的内容](#)。

把之前几个模块设置的中断放一块，此时的中断表我们看一下。

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
0x2E	hd_interrupt
0x80	system_call

这里我又提了一嘴，操作系统本质上就是个中断驱动的死循环，这个后面你会慢慢体会到。

而我们再往下看一行 `main` 方法。

```
#define sti() __asm__ ("sti::")
void main(void) {
    ...
    sti();
    ...
}
```


是一个 **sti** 汇编指令，意思是打开中断。其本质上是将 **eflags** 寄存器里的中断允许标志位 **IF** 位置 1。（由于已经是 32 位保护模式了，所以我把寄存器也都偷偷换成了 32 位的名字）

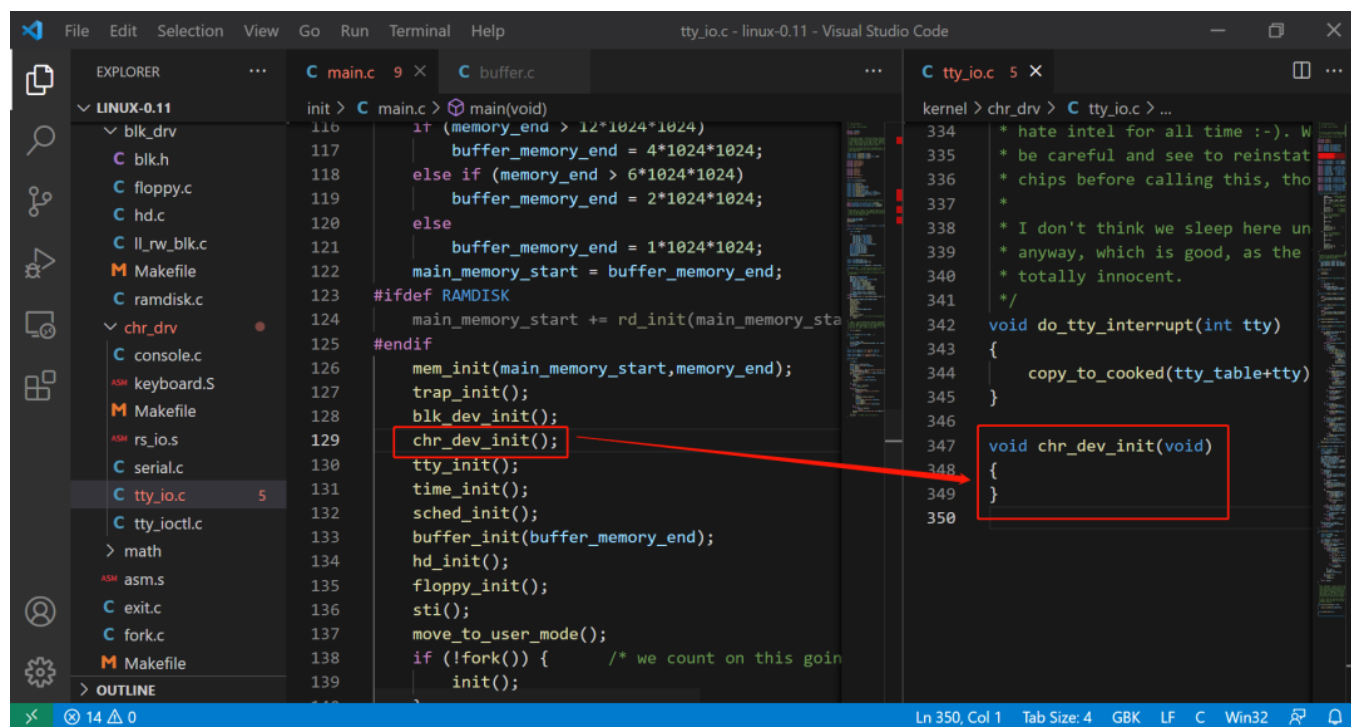


这样 CPU 就开始可以接收并处理中断信号了，键盘可以按了，硬盘可以读写了，时钟可以震荡了，系统调用也可以生效了！

这就代表着，操作系统具有了控制台交互能力，硬盘读写能力，进程调度能力，以及响应用户进程的系统调用请求！

至此，全部初始化工作，就结束了！这里有几个初始化函数没有讲，都是可以忽略的，不要担心。

一个是 `chr_dev_init`，因为这个函数里面本身就是空的，什么也没做。



```
116 int main(void)
117 {
118     (memory_end > 12*1024*1024)
119     buffer_memory_end = 4*1024*1024;
120     else if (memory_end > 6*1024*1024)
121         buffer_memory_end = 2*1024*1024;
122     else
123         buffer_memory_end = 1*1024*1024;
124     main_memory_start = buffer_memory_end;
125     #ifdef RAMDISK
126         main_memory_start += rd_init(main_memory_start);
127     #endif
128     mem_init(main_memory_start, memory_end);
129     trap_init();
130     blk_dev_init();
131     chr_dev_init();
132     tty_init();
133     time_init();
134     sched_init();
135     buffer_init(buffer_memory_end);
136     hd_init();
137     floppy_init();
138     sti();
139     move_to_user_mode();
140     if (!fork()) { /* we count on this going away */
141         init();
142     }
143 }
```

```
334 * hate intel for all time :-). We
335 * be careful and see to reinstat
336 * chips before calling this, tho
337 *
338 * I don't think we sleep here un
339 * anyway, which is good, as the
340 * totally innocent.
341 */
342 void do_tty_interrupt(int tty)
343 {
344     copy_to_cooked(tty_table+tty)
345 }
346
347 void chr_dev_init(void)
348 {
349 }
350
```

一个是 `tty_init` 里的 `rs_init`，这个方法是串口中断的开启，以及设置对应的中断处理程序，串口在我们现在的 PC 机上已经很少用到了，所以这个直接忽略。

还有一个是 `floppy_init`，这个是软盘的初始化，软盘现在已经被淘汰了，且电脑上也没有软盘控制器了，所以也忽略即可。



除了这些之外，全部的初始化工作，我们就全部梳理清楚了！再次为我们这一阶段性的胜利，鼓掌吧！！！！

同时，这章也会作为之后工作的一个索引章节，初始化工作所设置的所有数据结构都十分重要，后面如果你忘了，可以常来这里看看，祝大家好运。

欲知后事如何，且听下回分解。我要去休假了，刚好节前把第二部分收了尾巴，等节后我们开始大战第三部分！大家给我报销个回家的路费吧~

----- 关于本系列 -----

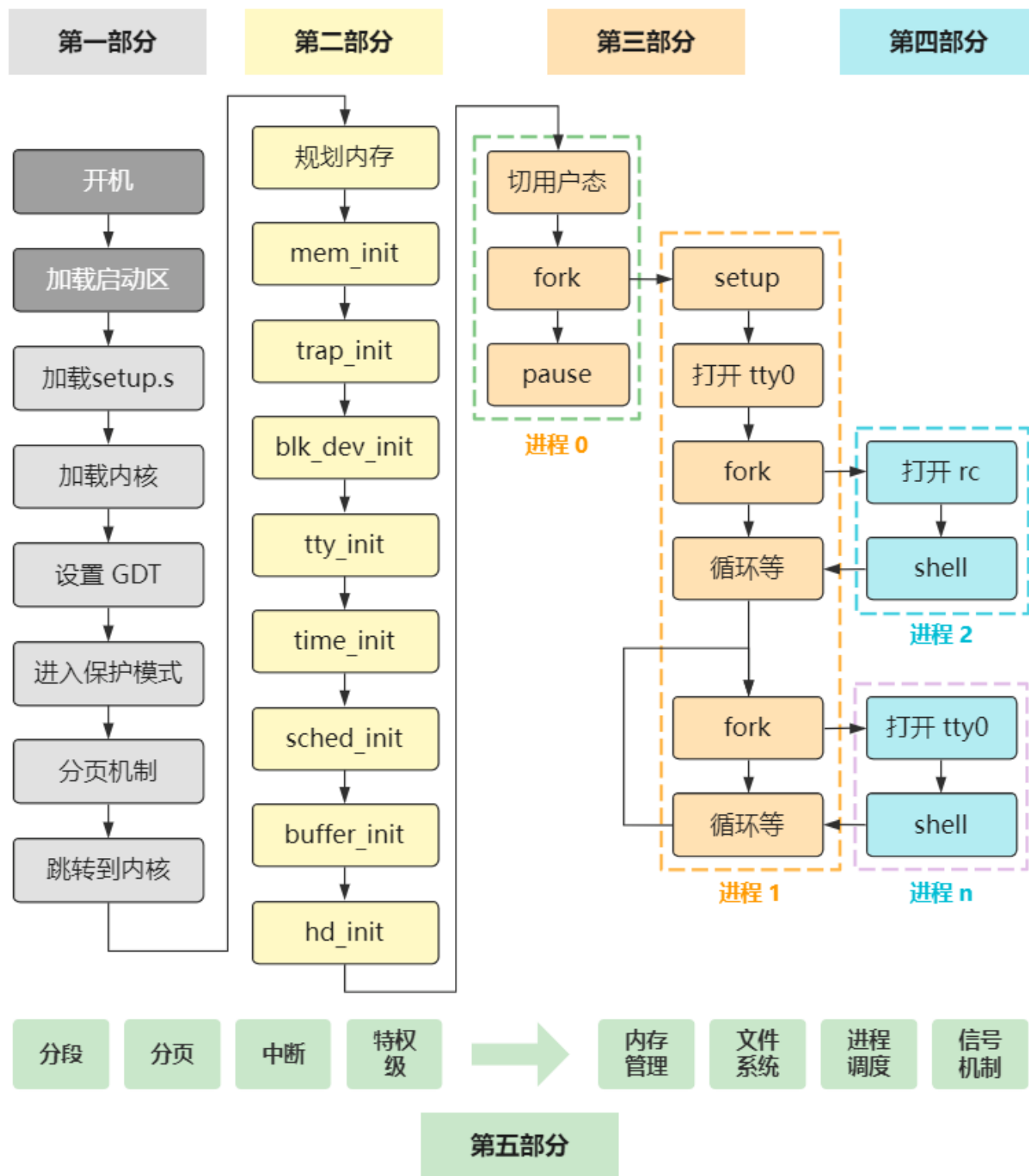
本系列的开篇词看这

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。

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20回 | 硬盘初始化 hd_init

下一篇

一个新进程的诞生（一）先整体看一下

Read more

People who liked this content also liked

外部函数如何访问其它类的私有成员

程序喵大人



如何在 Go 函数中获取调用者的函数名、文件名、行号...

Go编程时光



jmeter函数助手二次开发之加解密

测试新青年

