你管这破玩意叫操作系统源码 | 第三回 做好最最基础的准备工作

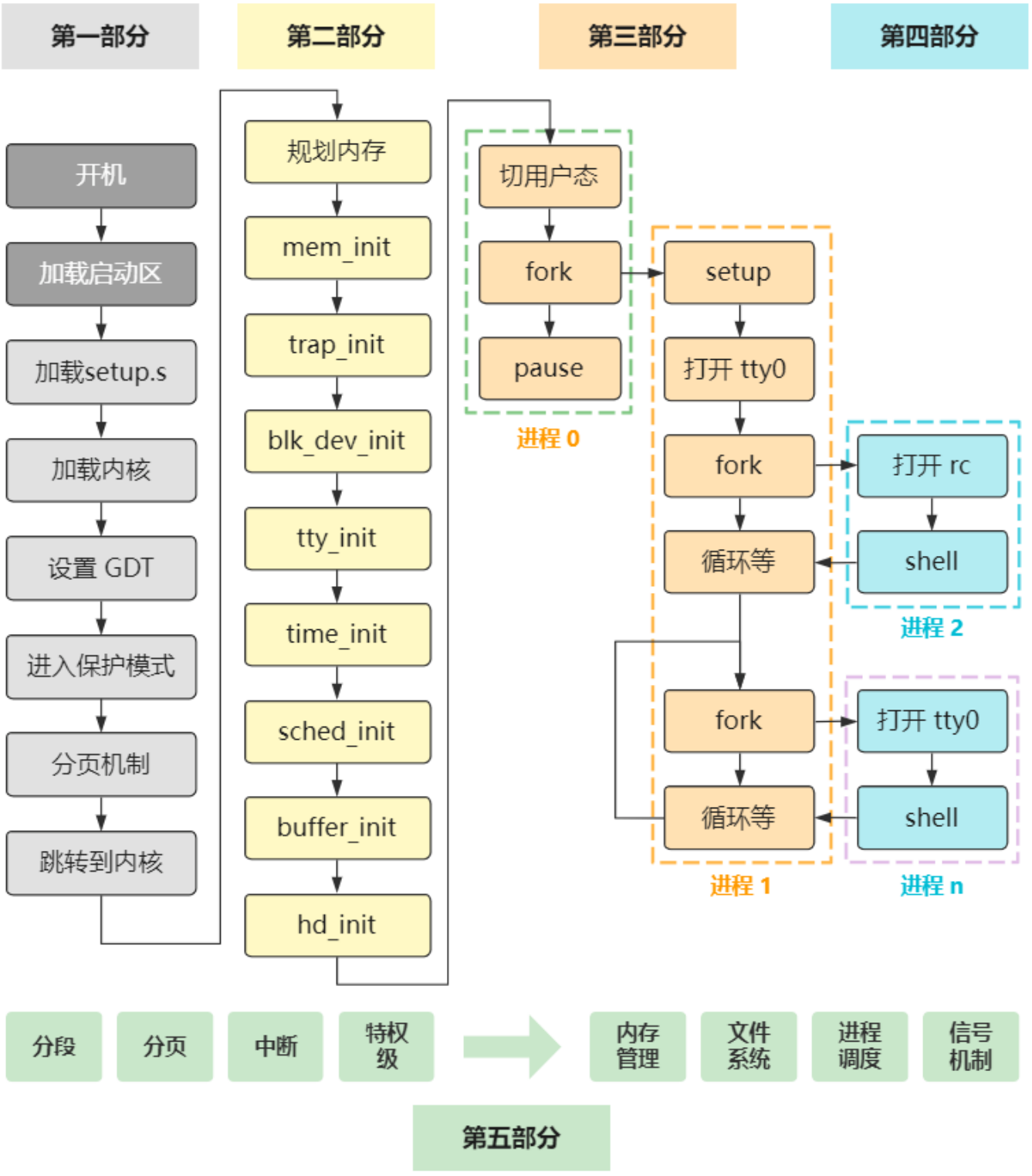
Original 闪客 低并发编程 2021-11-17 16:30

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。



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

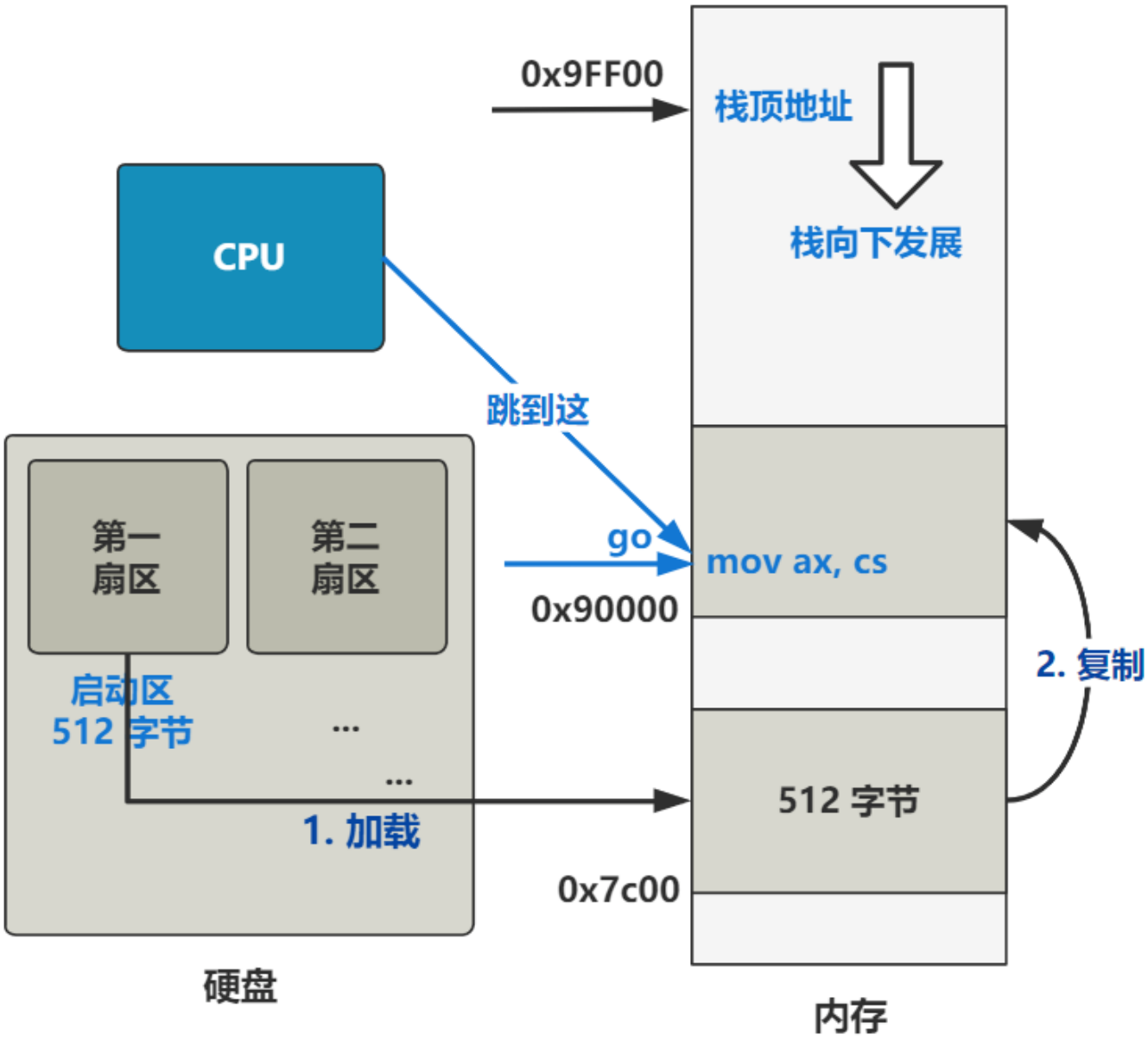
第一回 最开始的两行代码

第二回 自己给自己挪个地儿

本系列的 GitHub 地址如下
https://github.com/sunym1993/flash-linux0.11-talk

------- 正文开始 -------

书接上回，上回书咱们说到，操作系统的代码最开头的 512 字节的数据，从硬盘的启动区先是被移动到了内存 **0x7c00** 处，然后又立刻被移动到 **0x90000** 处，并且跳转到此处往后再稍稍偏移 **go** 这个标签所代表的偏移地址处。



那我们接下来，就继续把我们的目光放在 go 这个标签的位置，跟着 CPU 的步伐往后看。

```
go: mov ax,cs
    mov ds,ax
    mov es,ax
    mov ss,ax
    mov sp,#0xFF00
```

全都是 mov 操作，那好办了。

这段代码的直接意思很容易理解，就是把 **cs** 寄存器的值分别复制给 **ds**、**es** 和 **ss** 寄存器，然后又把 0xFF00 给了 **sp** 寄存器。

回顾下 CPU 寄存器图。

CPU 中的关键寄存器

cs 寄存器表示**代码段寄存器**，CPU 当前正在执行的代码在内存中的位置，就是由 cs:ip 这组寄存器配合指向的，其中 cs 是基址，ip 是偏移地址。

由于之前执行过一个段间跳转指令，还记得不？

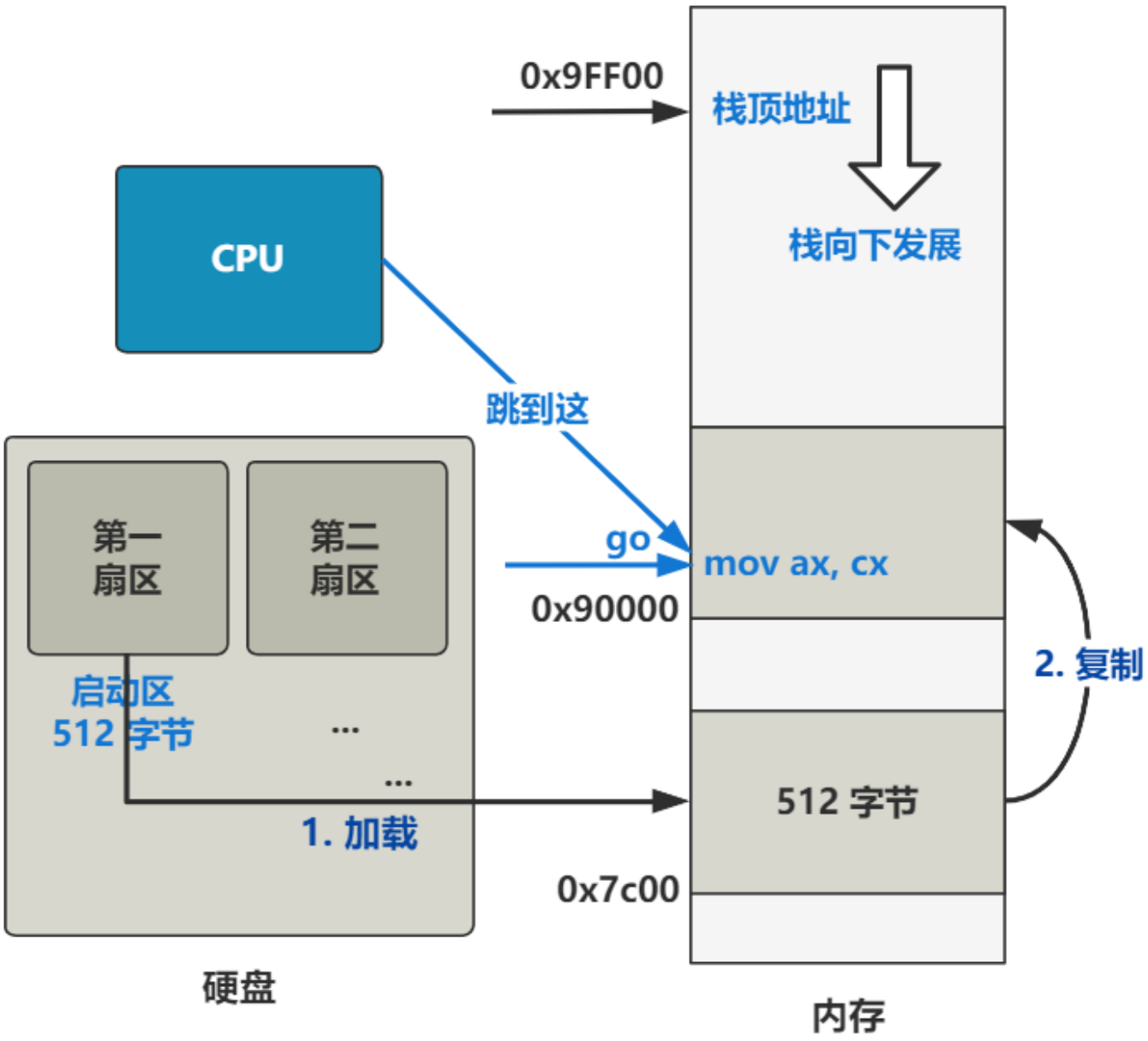
```
jmpi go,0x9000
```

所以现在 cs 寄存器里的值就是 **0x9000**，ip 寄存器里的值是 **go** 这个标签的偏移地址。那这三个 mov 指令就分别给 ds、es 和 ss 寄存器赋值为了 0x9000。

ds　为数据段寄存器，之前我们说过了，当时它被复制为　**0x07c0**，是因为之前的代码在 0x7c00 处，现在代码已经被挪到了 0x90000 处，所以现在自然又改赋值为 **0x9000** 了。

es 是扩展段寄存器，仅仅是个扩展，不是主角，先不用理它。

ss 为**栈段寄存器**，后面要配合栈基址寄存器 sp 来表示此时的栈顶地址。而此时 sp 寄存器被赋值为了 **0xFF00** 了，所以目前的栈顶地址就是 **ss:sp** 所指向的地址 **0x9FF00** 处。



其实到这里，操作系统的一些最最最最基础的准备工作，就做好了。都做了些啥事呢？

第一，代码从硬盘移到内存，又从内存挪了个地方，放在了 **0x90000** 处。

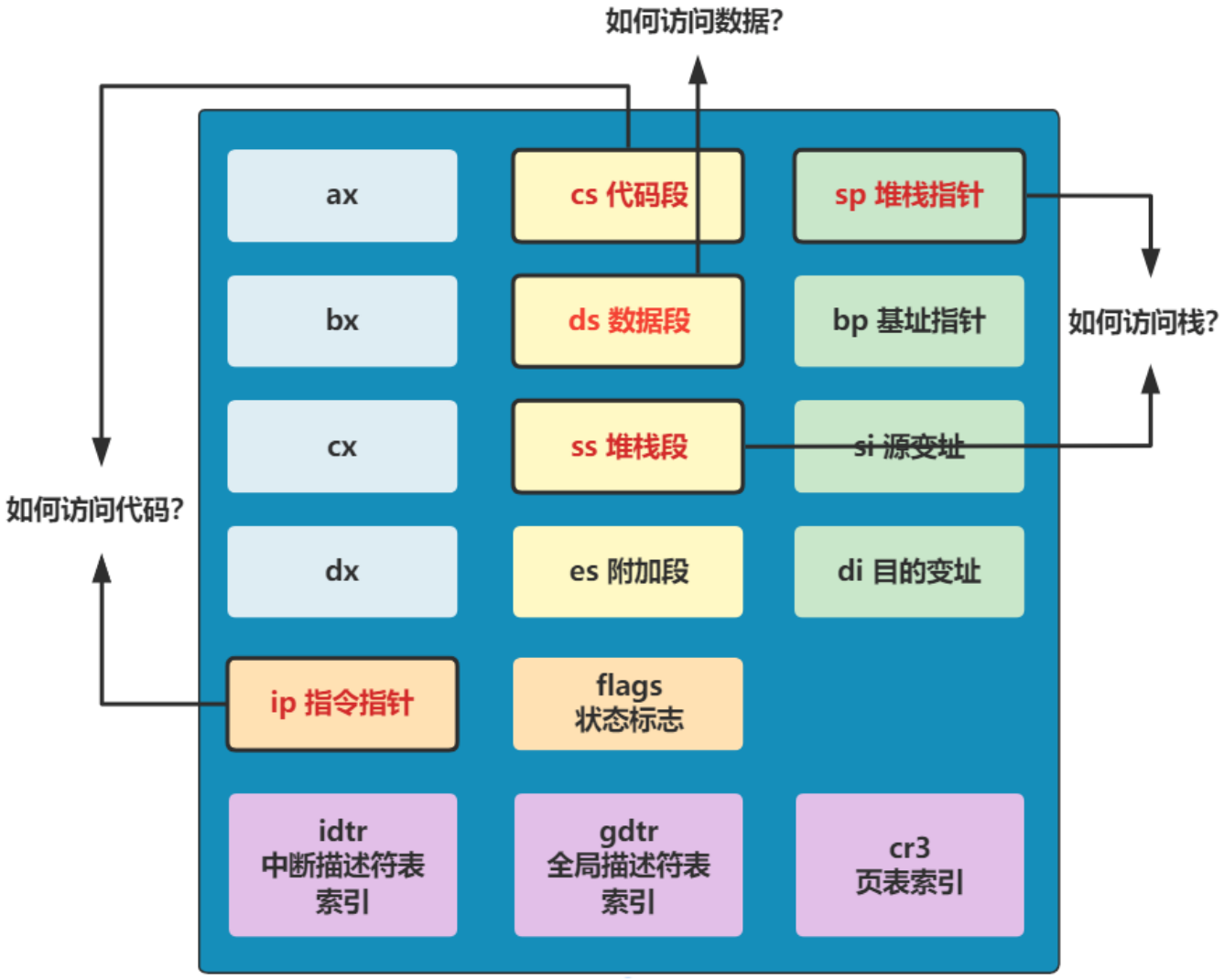
第二，**数据段寄存器 ds** 和**代码段寄存器 cs** 此时都被设置为了 0x9000，也就为跳转代码和访问内存数据，奠定了同一个内存的基址地址，方便了跳转和内存访问，因为仅仅需要指定偏移地址即可了。

第三，栈顶地址被设置为了 0x9FF00，具体表现为**栈段寄存器 ss** 为 0x9000，**栈基址寄存器 sp** 为 0xFF00。栈是向下发展的，这个栈顶地址 0x9FF00 要远远大于此时代码所在的位置 0x90000，所以栈向下发展就很难撞见代码所在的位置，也就比较安全。这也是为什么给栈顶地址设置为这个值的原因，其实只需要离代码的位置远远的即可。

做好这些基础工作后，接下来就又该折腾了其他事了。

总结拔高一下，这一部分其实就是把**代码段寄存器 cs**，**数据段寄存器 ds**，**栈段寄存器 ss** 和**栈基址寄存器 sp** 分别设置好了值，方便后续使用。

再拔高一下，其实操作系统在做的事情，就是给如何访问代码，如何访问数据，如何访问栈进行了一下**内存的初步规划**。其中访问代码和访问数据的规划方式就是设置了一个**基址**而已，访问栈就是把**栈顶指针**指向了一个远离代码位置的地方而已。



所以，千万别多想，就这么点事儿。那再给大家留个作业，把当前的内存布局画出来，告诉我现在 **cs、ip、ds、ss、sp** 这些寄存器的值，在内存布局中的位置。

好了，接下来我们应该干什么呢？我们回忆下，我们目前仅仅把硬盘中 512 字节加载到内存

中了，但操作系统还有很多代码仍然在硬盘里，不能抛下他们不管呀。

所以你猜下一步要干嘛了？

后面的世界越来越精彩，欲知后事如何，且听下回分解。

------- 本回扩展与延伸 -------

有关段寄存器的详细信息，可以参考 Intel 手册：
Volume 1 Chapter 3.4.2 Segment Registers
其中有一张图清晰地描述了三种段寄存器的作用。

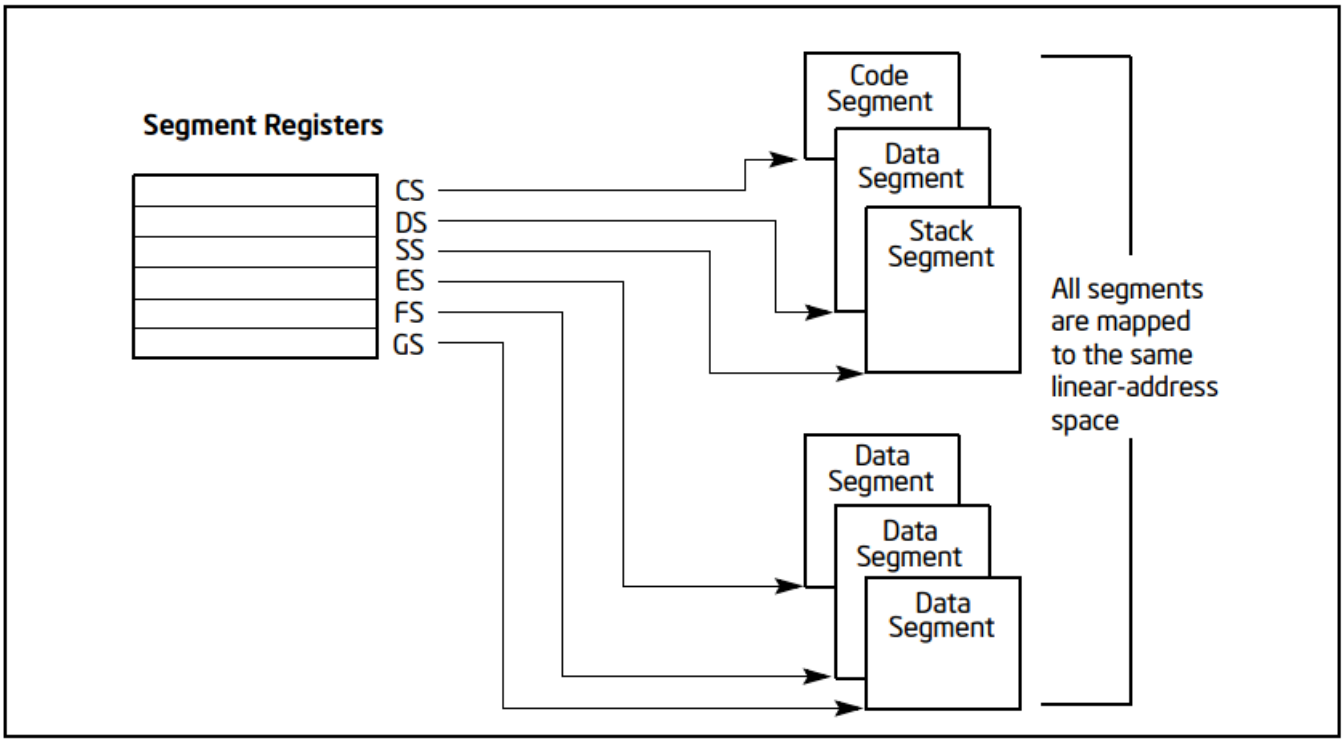


Figure 3-7. Use of Segment Registers in Segmented Memory Model

正如我们本回所涉及到的讲述一样，**CS　是代码段寄存器**，就是执行代码的时候带着这里存的基地址。**DS 是数据段寄存器**，就是访问数据的时候带着这里的基地址。**SS 是栈段寄存器**，就是访问栈时带着这里的基地址。

Each of the segment registers is associated with one of three types of storage: code, data, or stack. For example, the CS register contains the segment selector for the **code segment**, where the instructions being executed are stored. The processor fetches instructions from the code segment, using a logical address that consists of the segment selector in the CS register and the contents of the EIP register. The EIP register contains the offset within the code segment of the next instruction to be executed. The CS register cannot be loaded explicitly by an application program. Instead, it is loaded implicitly by instructions or internal processor operations that change program control (such as procedure calls, interrupt handling, or task switching).

The DS, ES, FS, and GS registers point to four **data segments**. The availability of four data segments permits efficient and secure access to different types of data structures. For example, four separate data segments might be created: one for the data structures of the current module, another for the data exported from a higher-level module, a third for a dynamically created data structure, and a fourth for data shared with another program. To access additional data segments, the application program must load segment selectors for these segments into the DS, ES, FS, and GS registers, as needed.

The SS register contains the segment selector for the **stack segment**, where the procedure stack is stored for the program, task, or handler currently being executed. All stack operations use the SS register to find the stack

所以本回的代码，正如标题所说，就是做好最最基础的准备工作。但要从更伟大的战略意义上讲，它其实是按照 Intel 手册上要求的，老老实实把这三类段寄存器的值设置好，达到了**初步规划内存**的目的。

读到这里，我希望你此时已经稍稍有些，**操作系统原来就是这个破玩意**，的感觉。

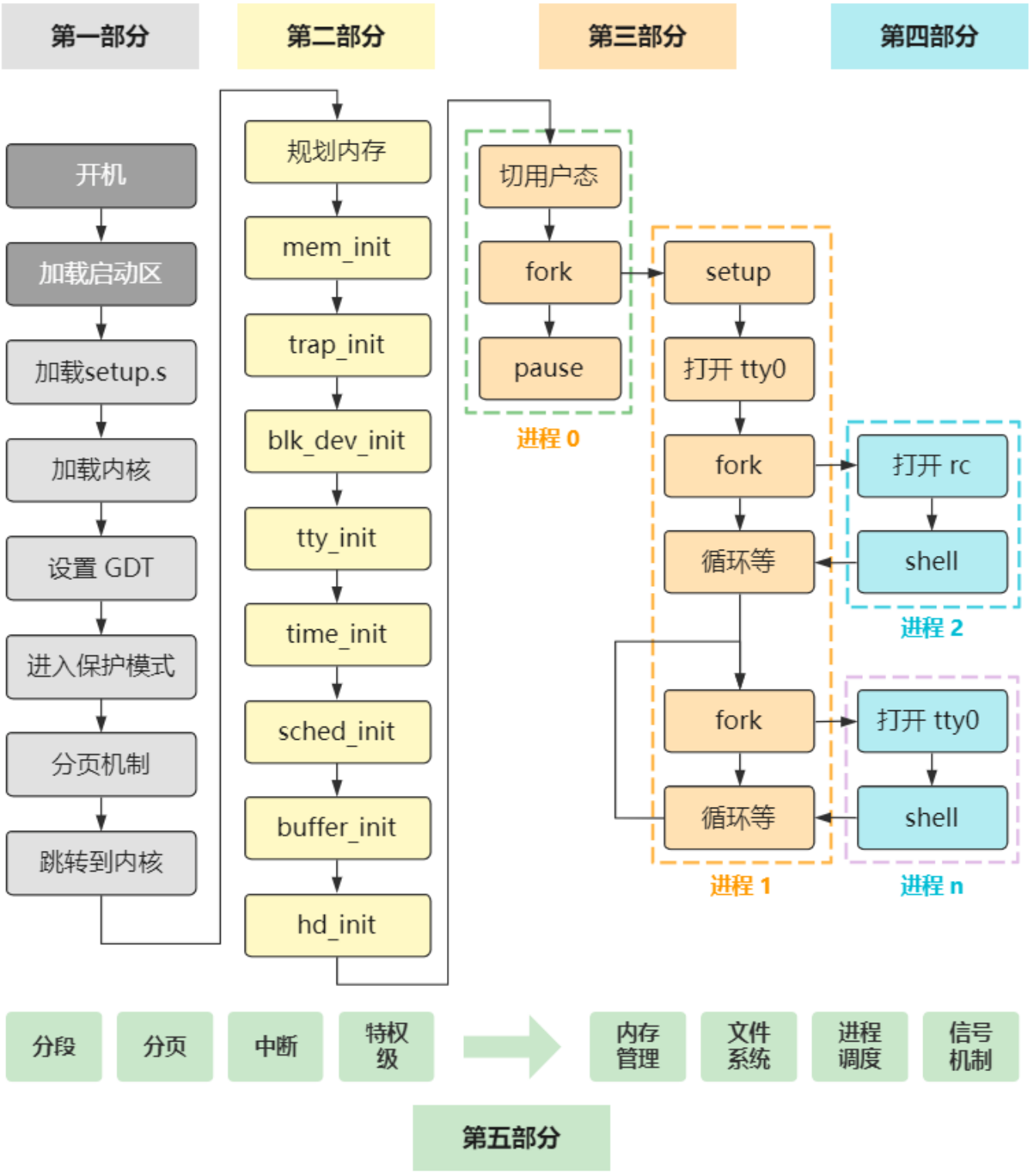
同时也可以看出，Intel 手册对于理解底层知识非常直接有效，但却没有很好的中文翻译版本，因此让许多人望而生畏，只能去看一些错误百出的中文二手资料和博客。因此我也发起了一个 **Intel 手册翻译计划**，就在阅读原文的 GitHub 里，感兴趣的同胞们可以参与进来，我们共同完成一份伟大的事。

------- 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
你管这破玩意叫操作系统源码 | 第二回 自己给自己挪个地儿

下一篇
第四回 | 把自己在硬盘里的其他部分也放到内存来

Read more