第十回 | 进入 main 函数前的最后一跃！

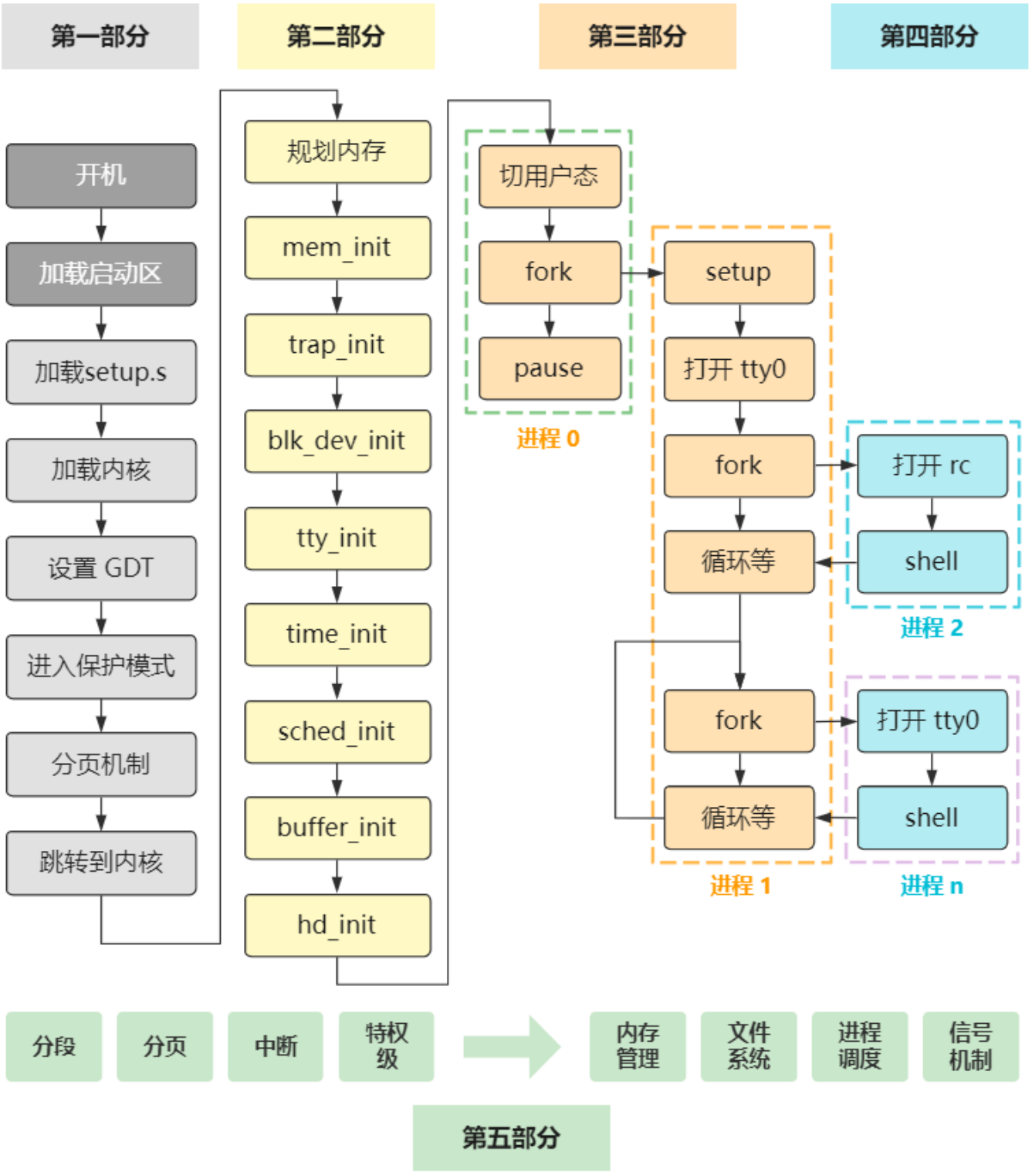
Original 闪客 低并发编程 2021-12-12 16:30

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
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

本系列的 GitHub 地址如下（文末阅读原文可直接跳转）
https://github.com/sunym1993/flash-linux0.11-talk

------- 正文开始 -------

书接上回，上回书咱们说到，我们终于把这些杂七杂八的，idt、gdt、页表都设置好了，并且也开启了保护模式，相当于所有苦力活都做好铺垫了，之后我们就要准备进入 main.c！那里是个新世界！

注意不是进入，而是准备进入哦，就差一哆嗦了。

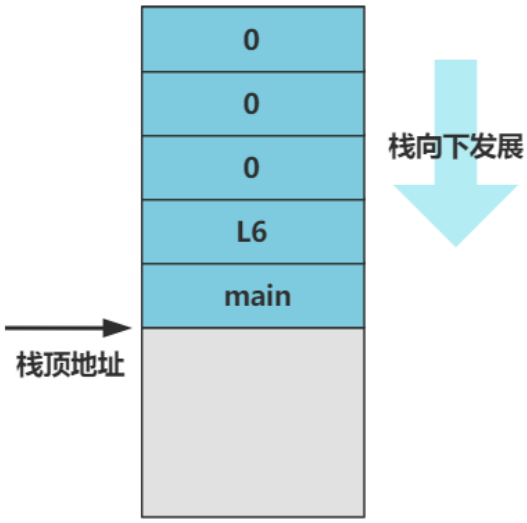
由于上一讲的知识量非常大，所以这一讲将会非常简单，作为进入 main 函数前的衔接，大家放宽心。

这仍然要回到上一讲我们跳转到设置分页代码的那个地方（head.s 里），这里有个骚操作帮我们跳转到 main.c。

```
after_page_tables:
    push 0
    push 0
    push 0
    push L6
    push _main
    jmp setup_paging
...
setup_paging:
    ...
    ret
```

直接解释起来非常简单。

push 指令就是**压栈**，五个 push 指令过去后，栈会变成这个样子。



然后注意，setup_paging 最后一个指令是 **ret**，也就是我们上一回讲的设置分页的代码的最后一个指令，形象地说它叫**返回指令**，但 CPU 可没有那么聪明，它并不知道该返回到哪里执行，只是很机械地**把栈顶的元素值当做返回地址**，跳转去那里执行。

再具体说是，把 esp 寄存器（栈顶地址）所指向的内存处的值，赋值给 eip 寄存器，而 cs:eip 就是 CPU 要执行的下一条指令的地址。而此时栈顶刚好是 main.c 里写的 main 函数的内存地址，是我们刚刚特意压入栈的，所以 CPU 就理所应当跳过来了。

当然 Intel CPU 是设计了 call 和 ret 这一配对儿的指令，意为调用函数和返回，具体可以看后面本回扩展资料里的内容。

至于其他压入栈的 L6 是用作当 main 函数返回时的跳转地址，但由于在操作系统层面的设计上，main 是绝对不会返回的，所以也就没用了。而其他的三个压栈的 0，本意是作为 main 函数的参数，但实际上似乎也没有用到，所以也不必关心。

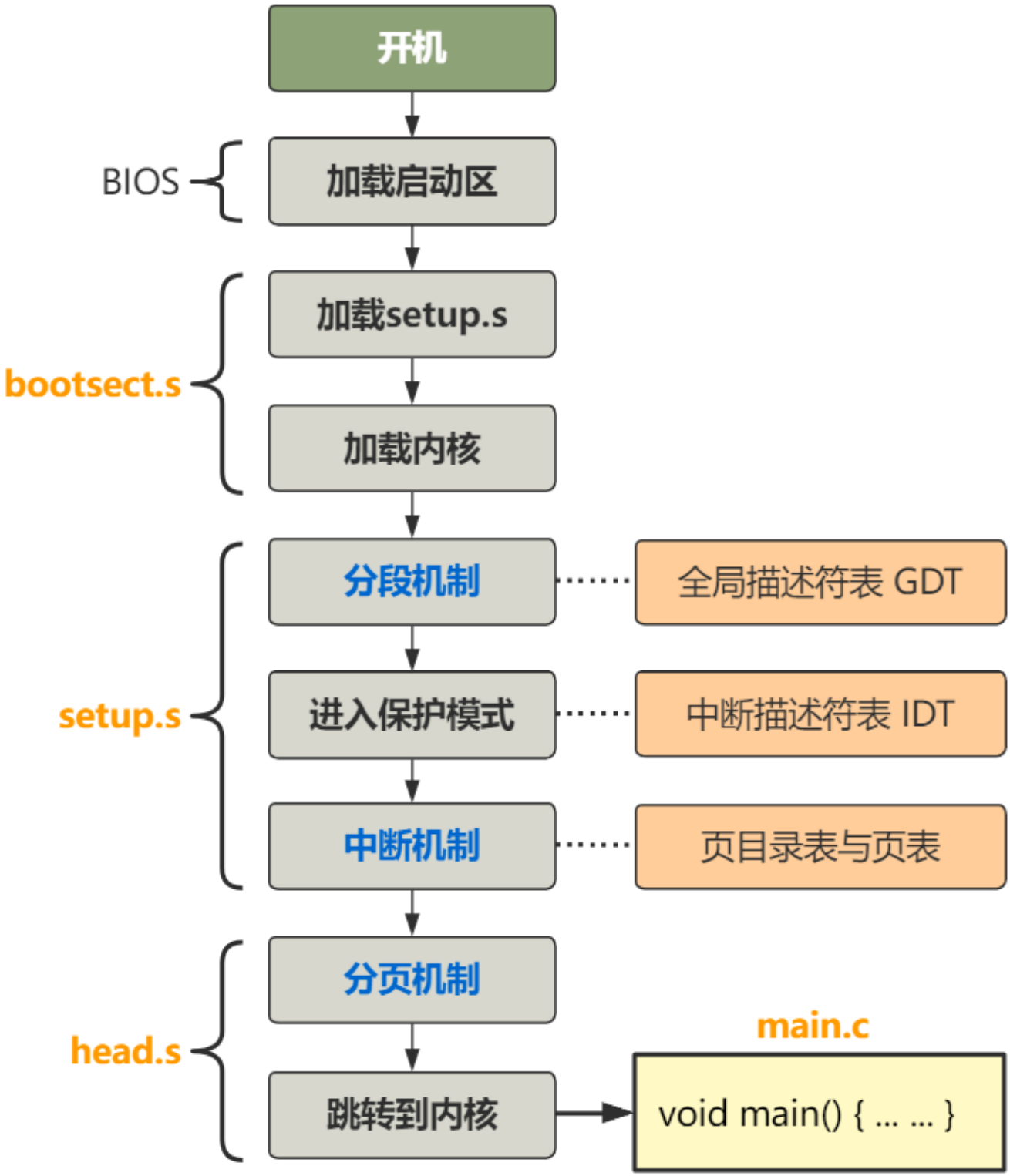
总之，经过这一个小小的骚操作，程序终于跳转到 main.c 这个由 c 语言写就的主函数 main 里了！我们先一睹为快一下。

```
void main(void) {
    ROOT_DEV = ORIG_ROOT_DEV;
    drive_info = DRIVE_INFO;
    memory_end = (1<<20) + (EXT_MEM_K<<10);
    memory_end &= 0xfffff000;
    if (memory_end > 16*1024*1024)
        memory_end = 16*1024*1024;
    if (memory_end > 12*1024*1024)
        buffer_memory_end = 4*1024*1024;
    else if (memory_end > 6*1024*1024)
        buffer_memory_end = 2*1024*1024;
    else
        buffer_memory_end = 1*1024*1024;
    main_memory_start = buffer_memory_end;
    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();
}
```

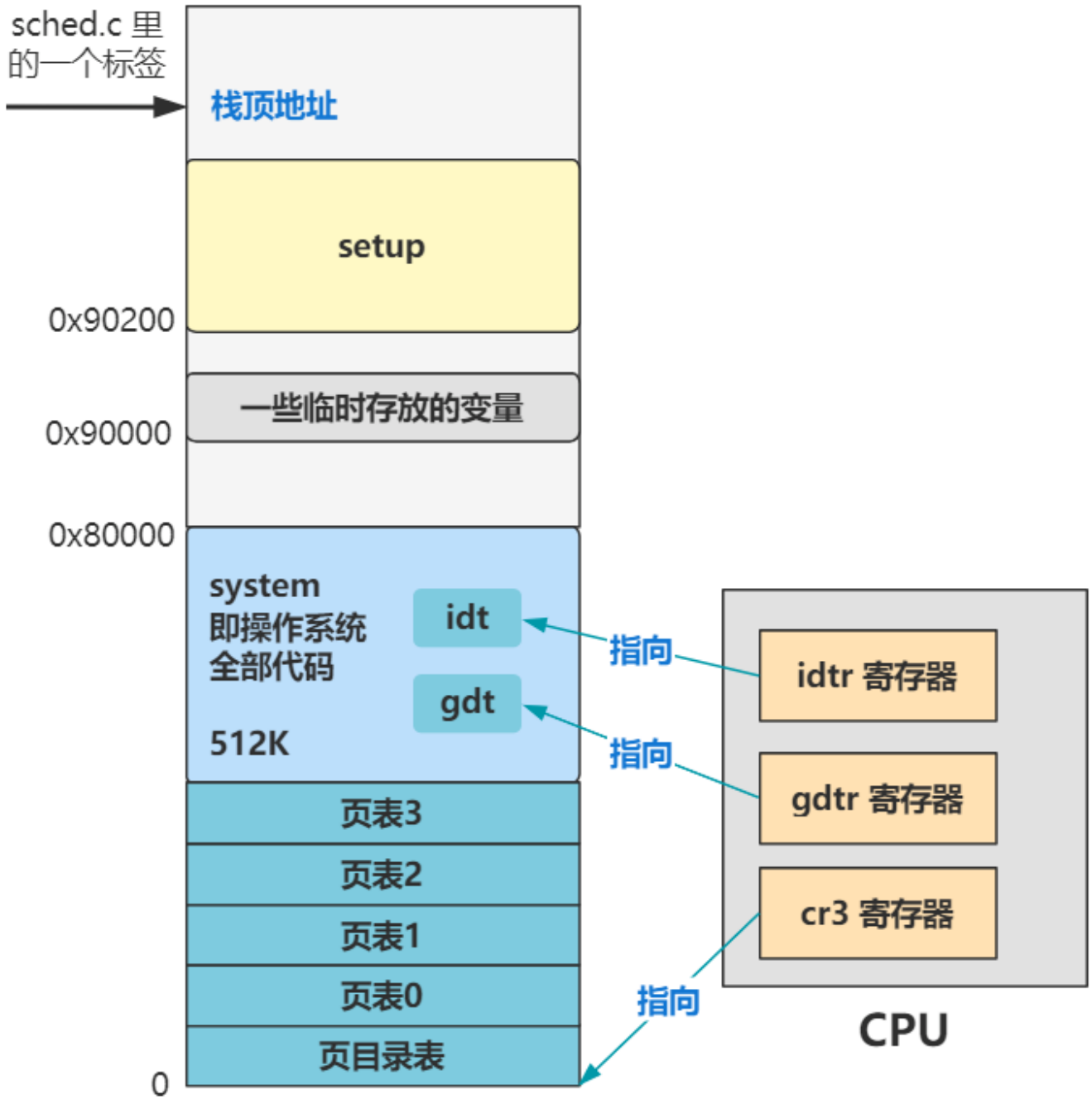
没错，这就是这个 main 函数的全部了。

而整个操作系统也会最终停留在最后一行死循环中，永不返回，直到关机。

好了，至此，整个第一部分就圆满结束了，为了跳进 main 函数的准备工作，我称之为进入内核前的苦力活，就完成了！我们看看我们做了什么。



我把这些称为**进入内核前的苦力活**，经过这样的流程，内存被搞成了这个样子。



之后，main　　方法就开始执行了，靠着我们辛辛苦苦建立起来的内存布局，向崭新的未来前进！

欲知后事如何，且听下回分解。

------- 本回扩展资料 -------

关于 ret 指令，其实 Intel CPU 是配合 call 设计的，有关 call 和 ret 指令，即调用和返回指令，可以参考 Intel 手册：
Intel 1 Chapter 6.4 CALLING PROCEDURES USING CALL AND RET

可以看到还分为不改变段基址的 near call 和 near ret

6.4.1 Near CALL and RET Operation

When executing a near call, the processor does the following (see Figure 6-2):

1. Pushes the current value of the EIP register on the stack.
   If shadow stack is enabled and the displacement value is not 0, pushes the current value of the EIP register on the shadow stack.

2. Loads the offset of the called procedure in the EIP register.
3. Begins execution of the called procedure.

When executing a near return, the processor performs these actions:

1. Pops the top-of-stack value (the return instruction pointer) into the EIP register.
   If shadow stack is enabled, pops the top-of-stack (the return instruction pointer) value from the shadow stack and if it's not the same as the return instruction pointer popped from the stack, then the processor causes a control protection exception with error code NEAR-RET (#CP(NEAR-RET)).
2. If the RET instruction has an optional *n* argument, increments the stack pointer by the number of bytes specified with the *n* operand to release parameters from the stack.
3. Resumes execution of the calling procedure.

以及改变段基址的 far call 和 far ret

6.4.2 Far CALL and RET Operation

When executing a far call, the processor performs these actions (see Figure 6-2):

1. Pushes the current value of the CS register on the stack.

   If shadow stack is enabled:

   a. Temporarily saves the current value of the SSP register internally and aligns the SSP to the next 8 byte boundary.

   b. Pushes the current value of the CS register on the shadow stack.

   c. Pushes the current value of LIP (CS.base + EIP) on the shadow stack.

   d. Pushes the internally saved value of the SSP register on the shadow stack.

2. Pushes the current value of the EIP register on the stack.
3. Loads the segment selector of the segment that contains the called procedure in the CS register.
4. Loads the offset of the called procedure in the EIP register.
5. Begins execution of the called procedure.

When executing a far return, the processor does the following:

1. Pops the top-of-stack value (the return instruction pointer) into the EIP register.
2. Pops the top-of-stack value (the segment selector for the code segment being returned to) into the CS register.

   If shadow stack is enabled:

   a. Causes a control protection exception (#CP(FAR-RET/IRET)) if the SSP is not aligned to 8 bytes.

   b. Compares the values on the shadow stack at address SSP+8 (the LIP) and SSP+16 (the CS) to the CS and (CS.base + EIP) popped from the stack, and causes a control protection exception (#CP(FAR-RET/IRET)) if they do not match.

   c. Pops the top-of-stack value (the SSP of the procedure being returned to) from shadow stack into the SSP register.

3. If the RET instruction has an optional *n* argument, increments the stack pointer by the number of bytes specified with the *n* operand to release parameters from the stack.
4. Resumes execution of the calling procedure.

压栈和出栈的具体过程，上面文字写的清清楚楚，下面 Intel 手册还非常友好地放了张图。

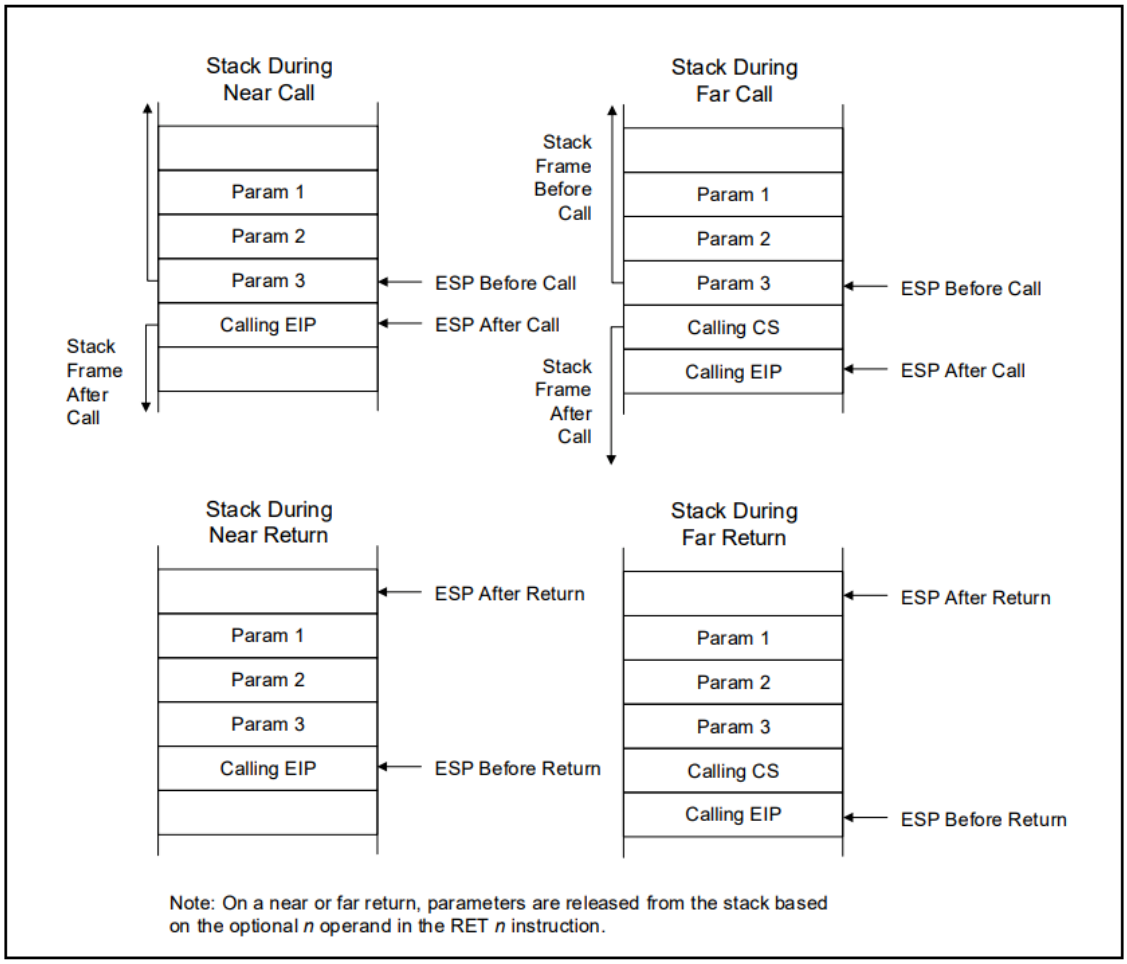


Figure 6-2. Stack on Near and Far Calls

可以看到，我们本文就是左边的那一套，把 main 函数地址值当做 Calling EIP 压入栈，仿佛是执行了 call 指令调用了一个函数一样，但实际上这是我们通过骚操作代码伪造的假象，骗了 CPU。

然后 ret 的时候就把栈顶的那个 Calling EIP 也就是 main 函数地址弹出栈，存入 EIP 寄存器，这样 CPU 就相当于“返回”到了 main 函数开始执行。

------- 关于本系列 -------

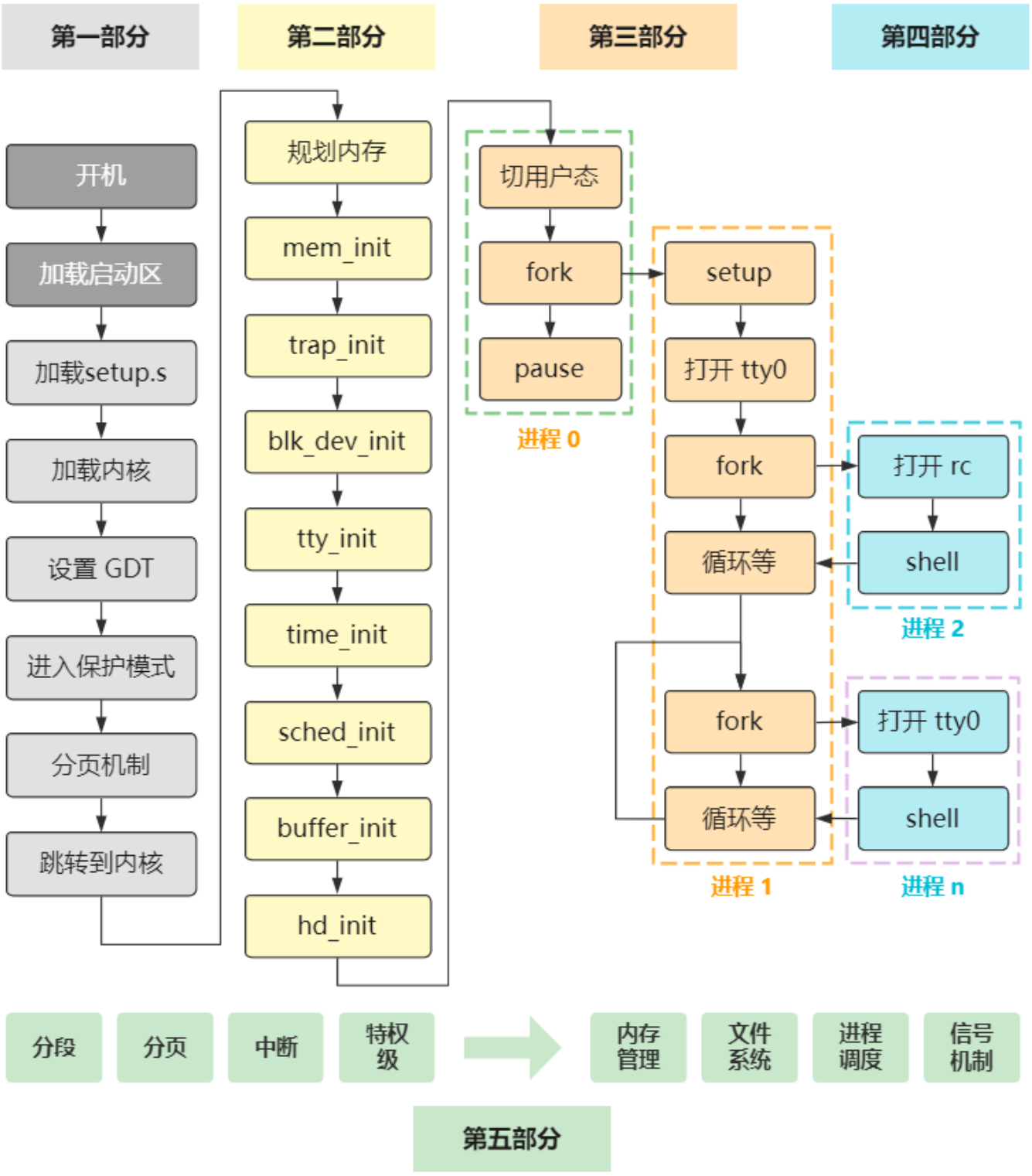
本系列的开篇词看这

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

第九回 | Intel 内存管理两板斧：分段与分页

下一篇

第一部分完结 进入内核前的苦力活

Read more