一个新进程的诞生（二）从内核态到用户态

Original 闪客 低并发编程 2022-02-13 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，上回书咱们从整体上鸟瞰了一下第三部分要讲的内容，代码上就是还差四句话就走到了 main 函数的尽头。

```
void main(void) {
    ...
    move_to_user_mode();
    if (!fork()) {
        init();
    }
    for(;;) pause();
}
```

今天我们就重点讲这第一句代码，**move_to_user_mode。**

让进程无法逃出用户态

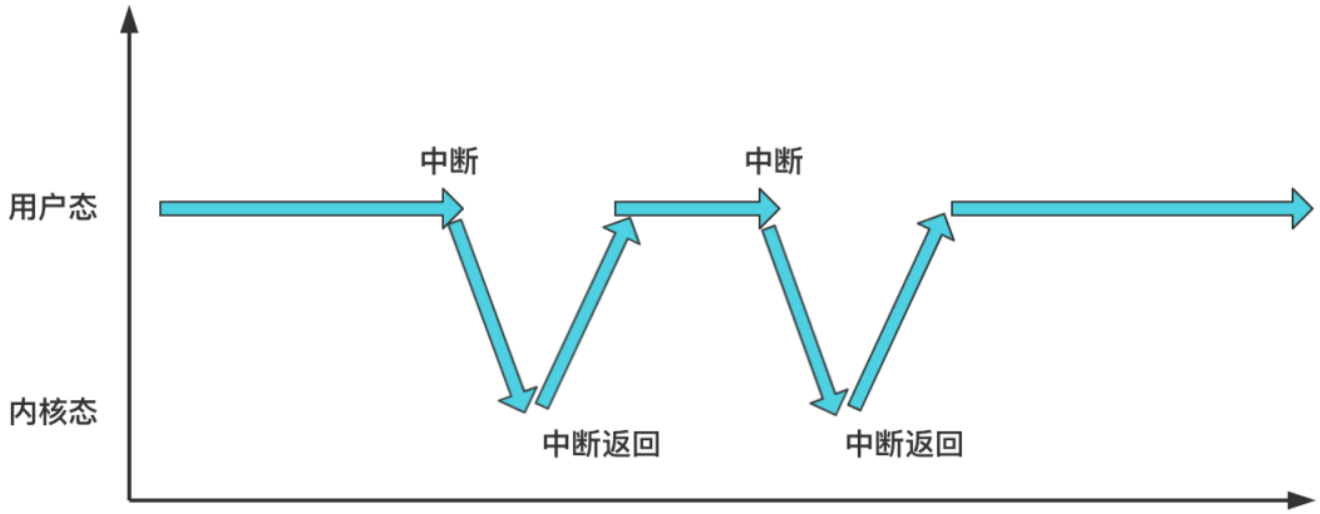
这行代码的意思直接说非常简单，就是**从内核态转变为了用户态**，但要解释清楚这个意思，还需要听我慢慢道来。

我相信你肯定听说过操作系统的内核态与用户态，用户进程都在用户态这个特权级下运行，而有时程序想要做一些内核态才允许做的事情，比如读取硬盘的数据，就需要通过系统调用，来请求操作系统在内核态特权级下执行一些指令。

我们现在的代码，还是在内核态下运行，之后操作系统达到怠速状态时，是以用户态的 shell 进程运行，随时等待着来自用户输入的命令。

所以，就在这一步，也就是 move_to_user_mode 这行代码，作用就是将当前代码的特权级，从内核态变为用户态。

一旦转变为了用户态，那么之后的代码将一直处于用户态的模式，除非发生了中断，比如用户发出了系统调用的中断指令，那么此时将会从用户态陷入内核态，不过当中断处理程序执行完之后，又会通过中断返回指令从内核态回到用户态。

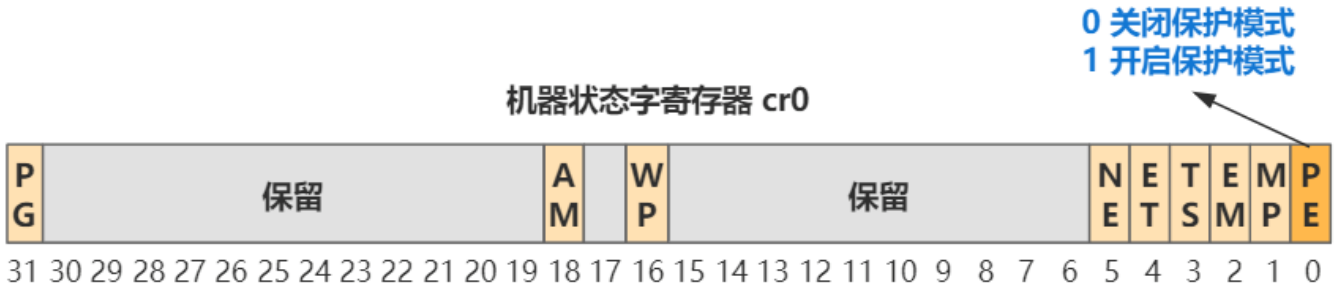


整个过程被操作系统的机制拿捏的死死的，始终让用户进程处于用户态运行，必要的时候陷入一下内核态，但很快就会被返回而再次回到用户态，是不是非常无奈？这样操作系统就掌控了控制权，而用户进程再怎么折腾也无法逃出这个模式。

内核态与用户态的本质-特权级

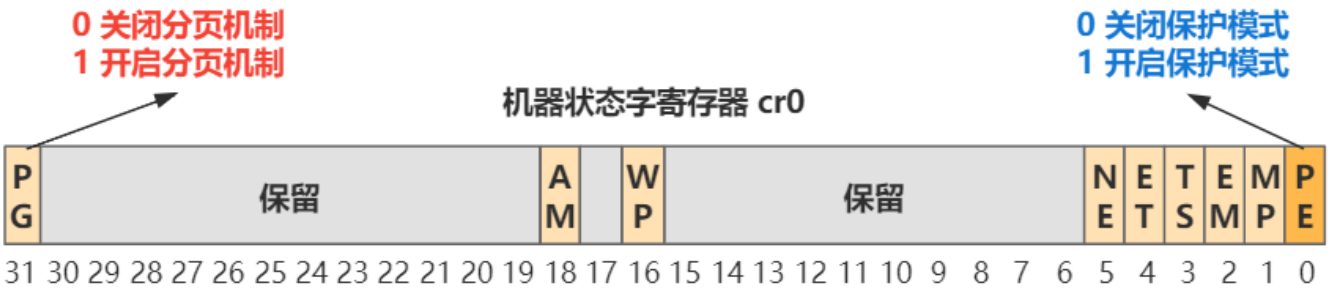
首先从一个最大的视角来看，这一切都源于 CPU 的保护机制。CPU 为了配合操作系统完成保护机制这一特性，分别设计了**分段保护机制**与**分页保护机制**。

当我们在 第七回 | 六行代码就进入了保护模式 将 cr0 寄存器的 PE 位开启时，就开启了保护模式，也即开启了**分段保护机制**。



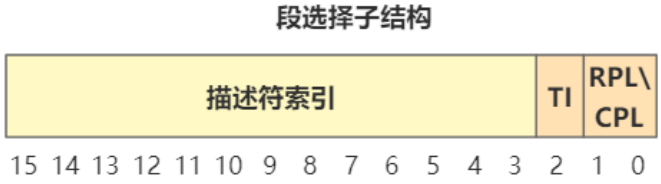
当我们在 第九回 | Intel 内存管理两板斧：分段与分页 将 cr0 寄存器的 PG 位开启时，就开

启了分页模式，也即开启了**分页保护机制。**



有关特权级的保护，实际上属于分段保护机制的一种。具体怎么保护的呢？由于这里的细节比较繁琐，所以我举个例子简单理解下即可，实际上的特权级检查规则要比我说的多好多内容。

我们目前正在执行的代码地址，是通过 CPU 中的两个寄存器 cs：eip 指向的对吧？cs 寄存器是代码段寄存器，里面存着的是段选择子，还记得它的结构么？

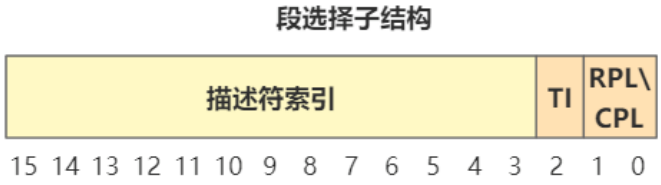


这里面的低端两位，此时表示 **CPL**，也就是**当前所处的特权级**，假如我们现在这个时刻，CS 寄存器的后两位为 3，二进制就是 11，就表示是当前处理器处于用户态这个特权级。

假如我们此时要跳转到另一处内存地址执行，在最终的汇编指令层面无非就是 jmp、call 和中断。我们拿 jmp 跳转来举例。

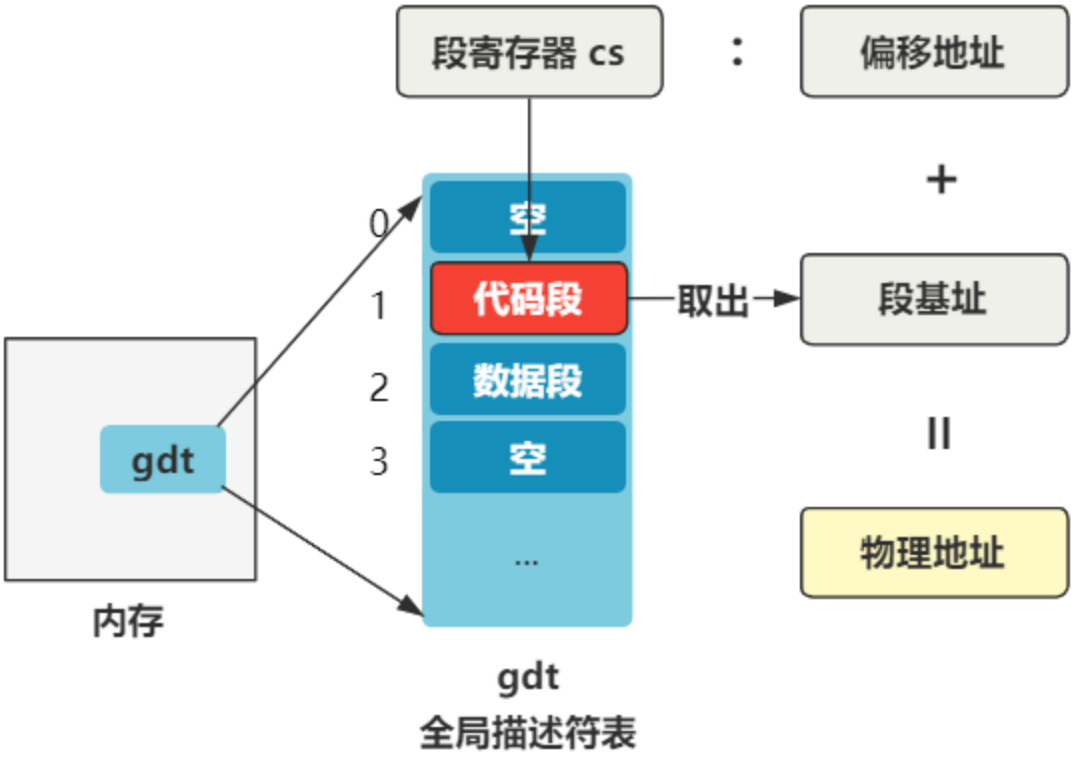
如果是短跳转，也就是直接 jmp xxx，那不涉及到段的变换，也就没有特权级检查这回事。

如果是长跳转，也就是 jmp yyy：xxx，这里的 yyy 就是另一个要跳转到的段的段选择子结构。



这个结构仍然是一样的段选择子结构，只不过这里的低端两位，表示 **RPL**，也就是**请求特权级**，表示我想请求的特权级是什么。同时，CPU 会拿这个段选择子去全局描述符表中寻找段

描述符，从中找到段基址。



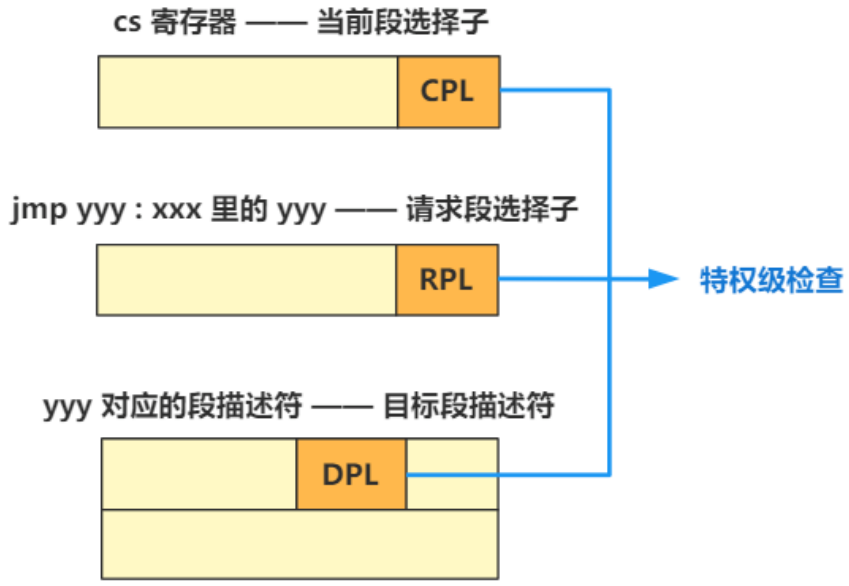
保护模式下物理地址的转换（仅段机制）

那还记得段描述符的样子么？

段描述符结构

你看，这里面又有个 **DPL**，这表示**目标代码段特权级**，也就是即将要跳转过去的那个段的特权级。

好了，我们总结一下简图，就是这三个玩意的比较。



这里的检查规则比较多，简单说，绝大多数情况下，**要求 CPL 必须等于 DPL**，才会跳转成功，否则就会报错。

也就是说，当前代码所处段的特权级，必须要等于要跳转过去的代码所处的段的特权级，那就只能**用户态往用户态跳**，**内核态往内核态跳**，这样就防止了处于用户态的程序，跳转到内核态的代码段中做坏事。

这只是代码段跳转时所做的特权级检查，还有访问内存数据时也会有数据段的特权级检查，这里就不展开了。最终的效果是，**处于内核态的代码可以访问任何特权级的数据段，处于用户态的代码则只可以访问用户态的数据段**，这也就实现了内存数据读写的保护。

说了这么多，其实就是，**代码跳转只能同特权级，数据访问只能高特权级访问低特权级。**

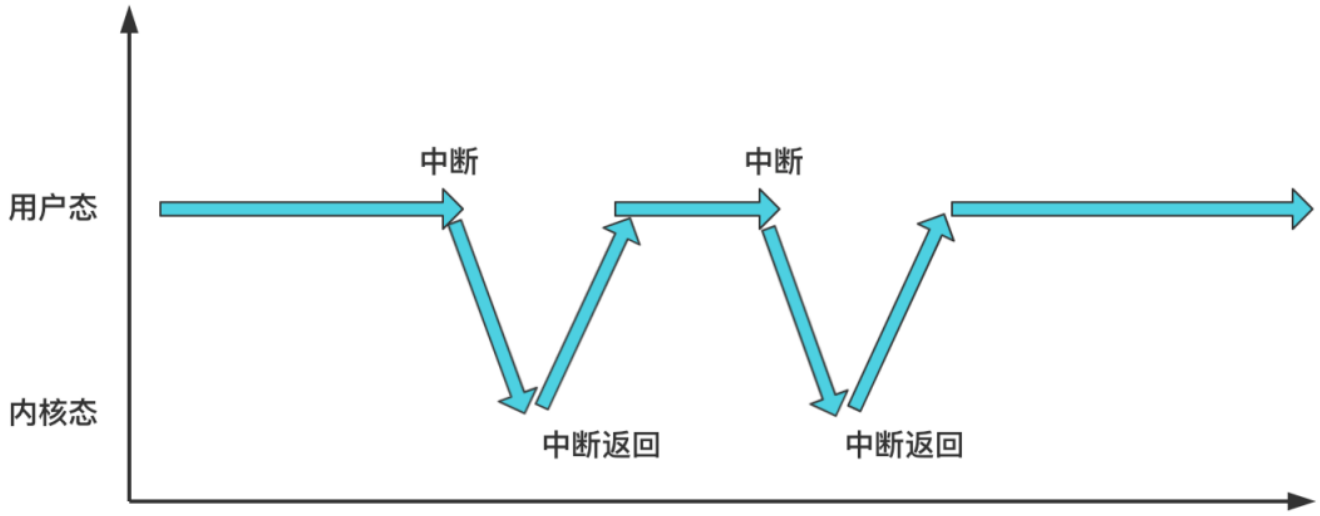
特权级转换的方式

诶不对呀，那我们今天要讲的是，从内核态转变为用户态，那如果代码跳转只能同特权级跳，我们现在处于内核态，要怎么样才能跳转到用户态呢？

Intel 设计了好多种特权级转换的方式，**中断**和**中断返回**就是其中的一种。

处于用户态的程序，通过触发中断，可以进入内核态，之后再通过中断返回，又可以恢复为用户态。

就是刚刚的图所表示的。



而**系统调用**就是这么玩的，用户通过 **int 0x80** 中断指令触发了中断，CPU 切换至内核态，执行中断处理程序，之后中断程序返回，又从内核态切换回用户态。

但有个问题是，我们当前的代码，此时就是处于内核态，并不是由一个用户态程序通过中断而切换到的内核态，那怎么回到原来的用户态呢？答案还是，通过中断返回。

没有中断也能中断返回？可以的，Intel 设计的 CPU 就是这样不符合人们的直觉，中断和中断返回的确是应该配套使用的，但也可以单独使用，我们看代码。

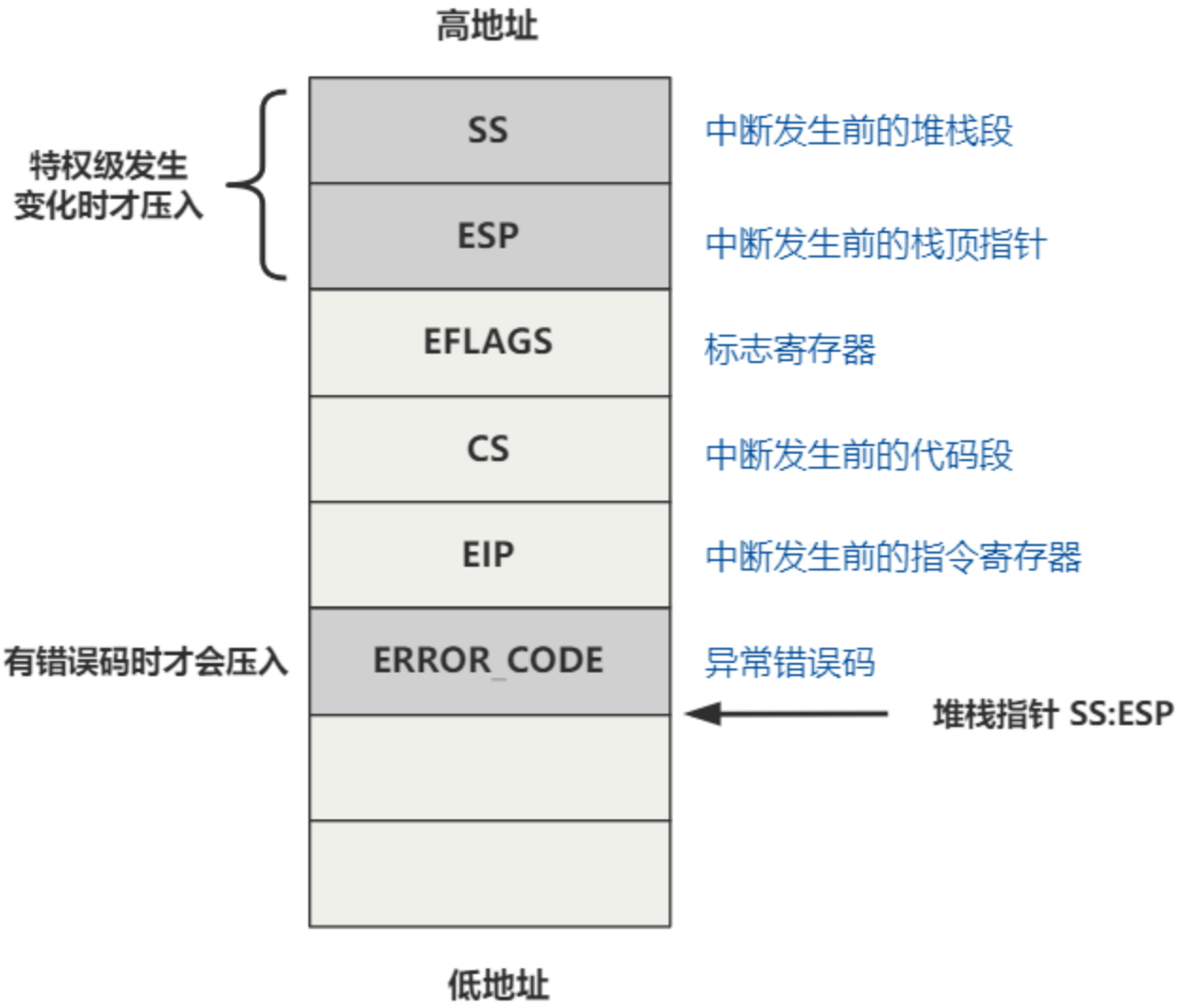
```
void main(void) {
    ...
    move_to_user_mode();
    ...
}

#define move_to_user_mode() \
_asm { \
    _asm mov eax,esp \
    _asm push 00000017h \
    _asm push eax \
    _asm pushfd \
    _asm push 0000000fh \
    _asm push offset l1 \
    _asm iretd /* 执行中断返回指令*/ \
_asm l1: mov eax,17h \
    _asm mov ds,ax \
    _asm mov es,ax \
    _asm mov fs,ax \
    _asm mov gs,ax \
}
```

你看，这个方法里直接就执行了中断返回指令 iretd。

那么为什么之前进行了一共**五次的压栈操作呢**？因为中断返回理论上就是应该和中断配合使用的，而此时并不是真的发生了中断到这里，所以我们得**假装发生了中断**才行。

怎么假装呢？其实就把栈做做工作就好了，中断发生时，CPU 会自动帮我们做如下的压栈操作。而中断返回时，CPU 又会帮我们把压栈的这些值返序赋值给响应的寄存器。



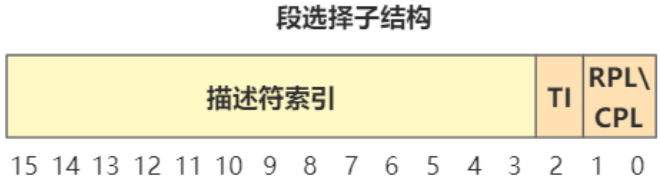
去掉错误码，刚好是五个参数，所以我们在代码中模仿 CPU 进行了五次压栈操作，这样在执行 iretd 指令时，硬件会按顺序将刚刚压入栈中的数据，分别赋值给 SS、ESP、EFLAGS、CS、EIP 这几个寄存器，这就感觉像是正确返回了一样，让其**误以为这是通过中断进来的。**

压入栈的 CS 和 EIP 就表示中断发生前代码所处的位置，这样中断返回后好继续去那里执行。

压入栈的 SS 和 ESP 表示中断发生前的栈的位置，这样中断返回后才好恢复原来的栈。

其中，特权级的转换，就体现在 CS 和 SS 寄存器的值里，都是细节！

CS 和 SS 寄存器是段寄存器的一种，段寄存器里的值是段选择子，其结构上面已经提过两遍了，在 第六回 | 先解决段寄存器的历史包袱问题 中也专门讲了这个结构的作用。



对着这个结构，我们看代码。

```
#define move_to_user_mode() \
_asm { \
    _asm mov eax,esp \
    _asm push 00000017h \ ; 给 SS 赋值
    _asm push eax \
    _asm pushfd \
    _asm push 0000000fh \ ; 给 CS 赋值
    _asm push offset l1 \
    _asm iretd /* 执行中断返回指令*/ \
_asm l1: mov eax,17h \
    _asm mov ds,ax \
    _asm mov es,ax \
    _asm mov fs,ax \
    _asm mov gs,ax \
}
```

拿 CS 举例，给它赋的值是，**0000000fh**，用二进制表示为：

0000000000001111

最后两位 11 表示特权级为 3，即用户态。而我们刚刚说了，CS 寄存器里的特权级，表示 CPL，即当前处理器特权级。

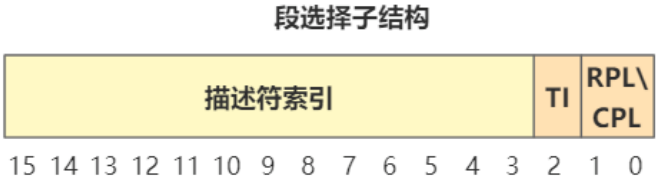
所以经过 iretd 返回之后，CS 的值就变成了它，而当前处理器特权级，也就变成了用户态特权级。

除了改变特权级之外

除了改变了特权级之外，还做了什么事情呢？

刚刚我们关注段寄存器，只关注了特权级的部分，我们再详细看看。

刚刚说了 CS 寄存器为 **0000000000001111**，最后两位表示用户态的含义。



那继续解读，倒数第三位 TI 表示，前面的描述符索引，是从 GDT 还是 LDT 中取，1 表示 LDT，也就是从局部描述符表中取。

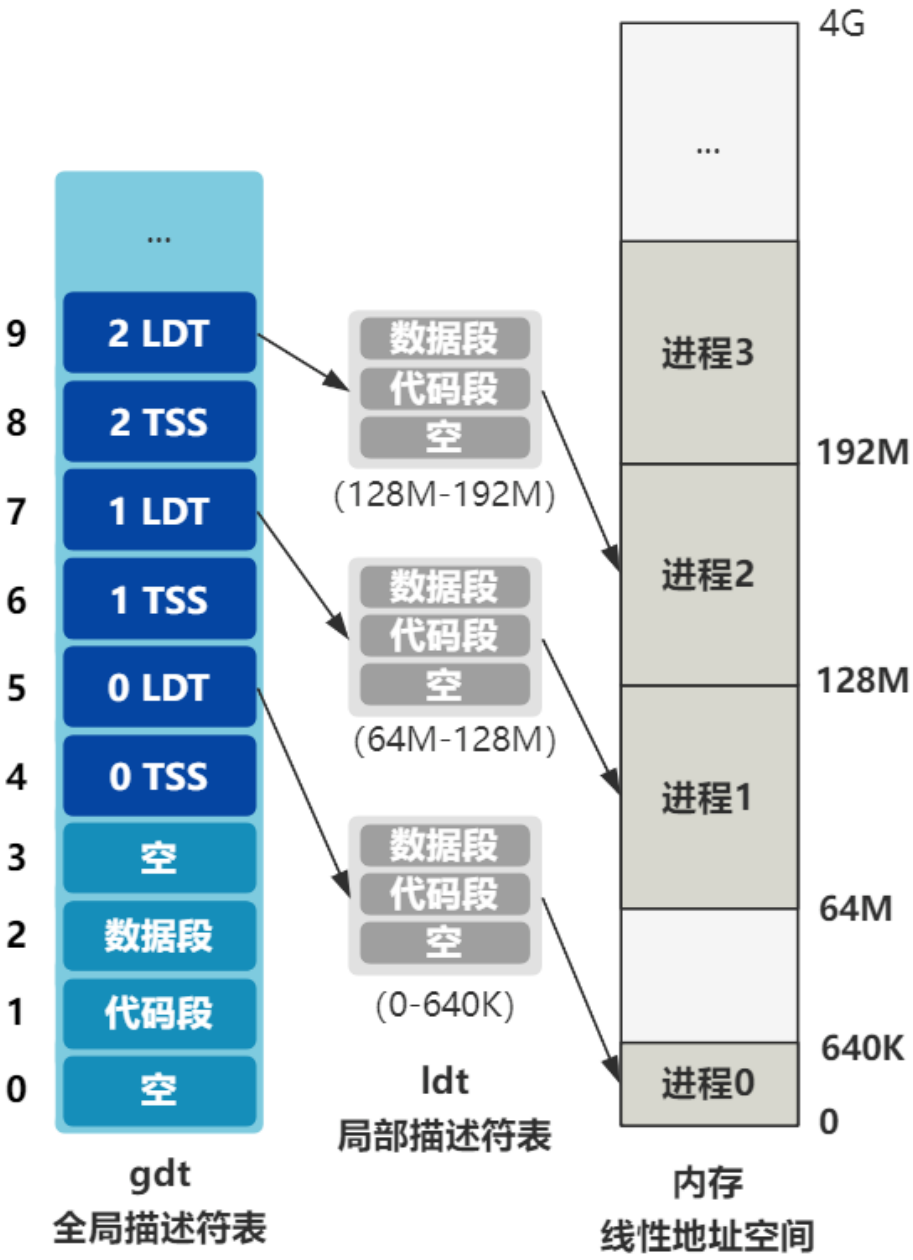
前面的描述符索引为 1，表示从局部描述符表中取到代码段描述符，如果你熟悉前面我讲过的内容，你将会直接得出上述结论。不过我还是帮你回忆一下。

在 第18回 | 大名鼎鼎的进程调度就是从这里开始的 中，将 0 号 LDT 作为当前的 LDT 索引，记录在了 CPU 的 lldt 寄存器中。

```
#define lldt(n) __asm__("lldt %%ax"::"a" (_LDT(n)))

void sched_init(void) {
    ...
    lldt(0);
    ...
}
```

而整个 GDT 与 LDT 表的设计，经过整个 第一部分 进入内核前的苦力活 和 第二部分 大战前期的初始化工作 的设计后，成了这个样子。



所以，一目了然。

再看这行代码，把 EIP 寄存器赋值为了那行标号的地址。

```
void main(void) {
    ...
    move_to_user_mode();
    ...
}

#define move_to_user_mode() \
_asm { \
    _asm mov eax,esp \
    _asm push 00000017h \
    _asm push eax \
    _asm pushfd \
    _asm push 0000000fh \
    _asm push offset l1 \
    _asm iretd /* 执行中断返回指令*/ \
_asm l1: mov eax,17h \
    _asm mov ds,ax \
    _asm mov es,ax \
    _asm mov fs,ax \
    _asm mov gs,ax \
}
```

这里刚好设置的是下面标号 l1 的位置，所以 iretd 之后 CPU 就乖乖去那里执行了。所以其实从效果上看，就是顺序往下执行，只不过利用了 iretd 做了些特权级转换等工作。

同理，这里的栈段 ss 和数据段 ds，都被赋值为了 17h，大家可以展开二进制算一下，他们又是什么特权级，对应的描述符又是谁。

总结

所以其实，最终效果上看就是按顺序执行了我们所写的指令，仿佛没有经过什么中断和中断返回的过程，但却通过中断返回实现了特权级的翻转，也就是从内核态变为了用户态，顺便设置了栈段、代码段和数据段的基地址。

好了，我们兜兜转转终于把这个 mov_to_user_mode 讲完了，特权级这块的检查细节非常繁

琐，为了理解操作系统，我们只需要暂且记住如下一句话就好了：

数据访问只能高特权级访问低特权级，代码跳转只能同特权级跳转，要想实现特权级转换，可以通过中断和中断返回来实现。

OK，我们现在已经进入了用户态，也即表明了需要内核态来完成的工作已经全部安排妥当了，其实就是整个 第一部分 进入内核前的苦力活 和 第二部分 大战前期的初始化工作 的内容，对全局描述符表、中断描述符表、页表等关键内存结构进行设置，以及对 CPU 特殊寄存器如 cr0 和 cr3 的设置，还有对外设如硬盘、键盘、定时器的设置等。

看来我们又完成了一大堆苦力活呀，内核态做的工作也真是枯燥乏味呢。接下来只需要在用户态进行工作即可了！

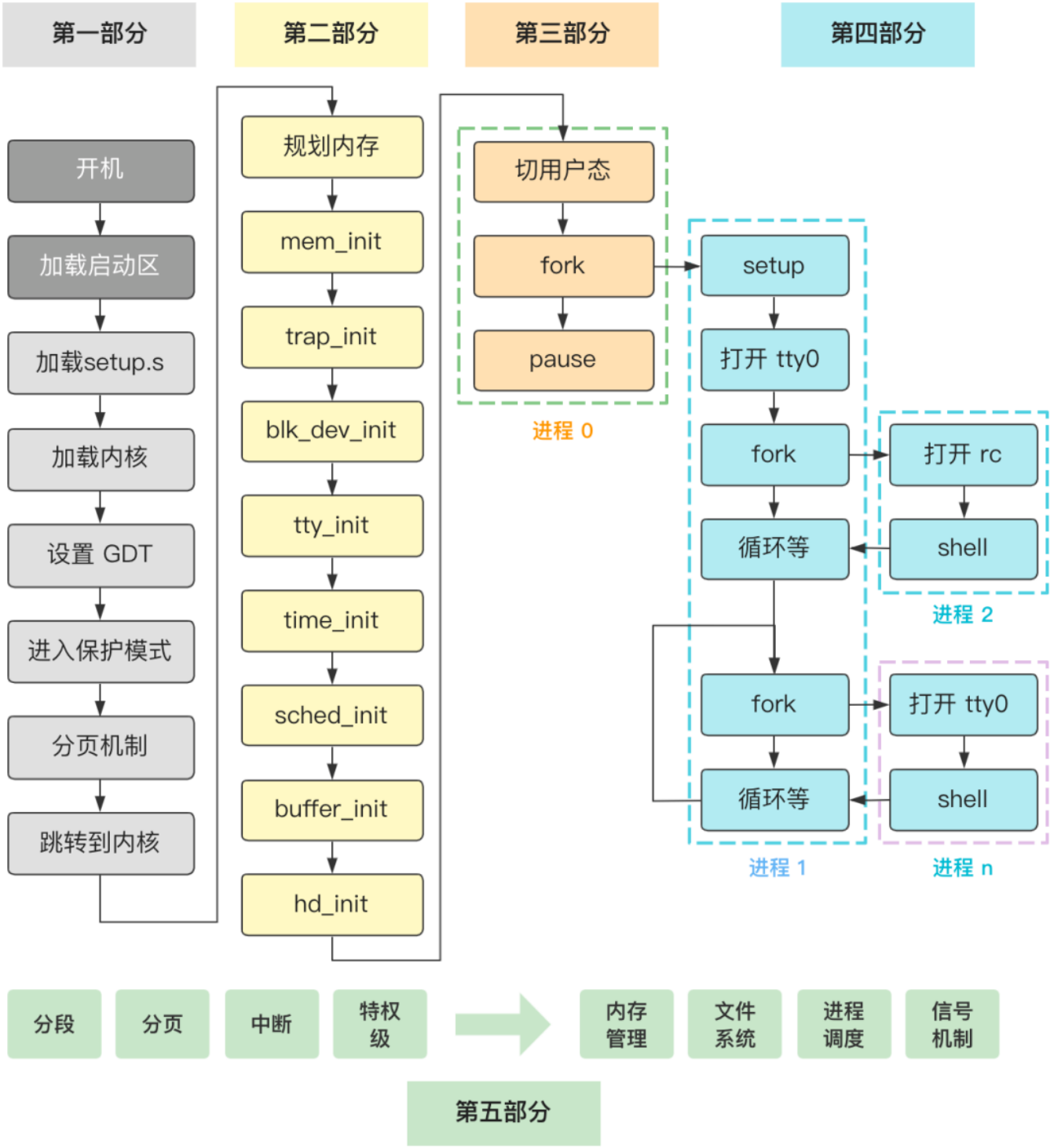
欲知后事如何，且听下回分解。

------- 关于本系列的完整内容 -------

本系列的开篇词看这
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。

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（一）先整体看一下

下一篇
一个新进程的诞生（三）如果让你来设计进程调度

Read more

People who liked this content also liked

vivado axi master ip core状态机分析
IC打工魂

西门子标准化之路(3)—程序的复用性和内存管理
自动化玩家

服务端网站架构的演进：从100个并发到千万级并发
程序员黑哥