《openEuler内核编程》

课程讲稿

第一章 第2讲

操作系统的启动过程

软件所制

第1章 第2讲 操作系统的启动过程

**学时：**1学时

**教学目的：**了解操作系统启动过程中的一些关键步骤。对CPU模式之间的切换有感性的理解。明白内核态与用户态的区别，了解不同状态下系统权限的差异。

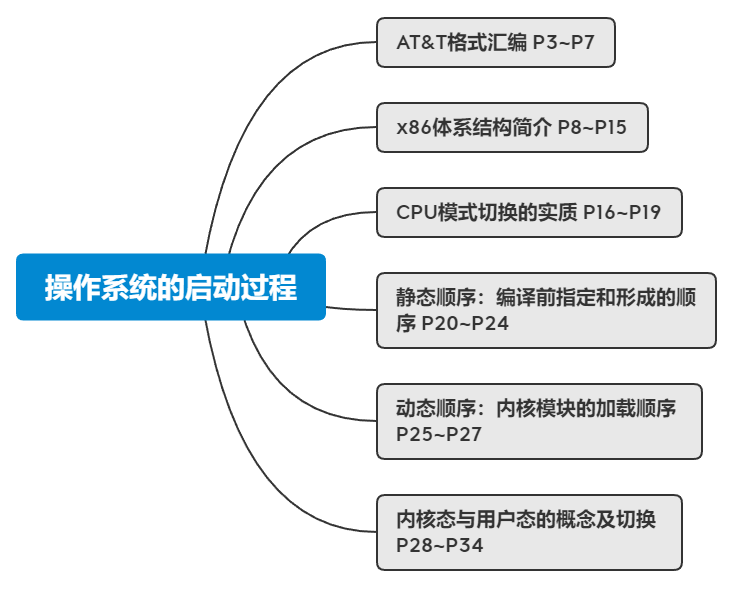
**课程时间线：**



**课外参考读物：**

https://www.cnblogs.com/a-lai/articles/7293828.html

**知识框图：**

****

1. 这节课我们讲操作系统的启动过程，首先就要了解一下汇编。
2. 汇编语言是各种CPU指令的助记符的集合。我们用汇编语言就可以直接控制硬件系统进行工作。现在主要有两种形式的汇编语言，一种是英特尔的，像DOS啊Windows的这些系统会用。另一种是at&t。比如unix Linux用的都是这一种格式。我们使用的openEuler操作系统呢，它是基于linux的，所以我们要了解后者的汇编语言的格式。
3. 每一条汇编指令呢，都是由4个部分组成的。分别是标号，操作码，操作数和注释。其中第1个和第4个是可选的，那么中间的两个是必选的。这里面的操作数呢，可以不只有一个，这个数量呢，是根据你的操作码来确定的。
4. 在汇编中呢，操作码名称的最后一个字符，指的是操作数的宽度。比如像mov后面可以跟b wl。其中b代表byte字节，w是word字，l是long，双字。举个例子，如 movl var, %eax 表示把内存地址var处的4个字节载入到寄存器eax中。操作码前缀，常用语重复字符串，指令执行总线锁定操作，或者指定操作数和地址宽度等等。下面的例子呢就是一个重复的操作。
5. 操作数呢，一个指令可以有0~3个操作数，中间呢是用逗号来分隔的。如果有两个操作数的话，那么前面那个操作数呢表示是原操作数，后面呢表示是目的操作数。看这的例子啊，这个就表示把ax寄存器的内容复制到bx寄存器。操作。数的种类呢分为三种，分别是立即数寄存器以及内存，那么在前面加上前置进行一个区分如果是利益数的话就加上一个美元符号，如果是寄存器的话，那么前面就加上一个百分号。内存的话稍微复杂一点。像这里的一个例子。内存地址就是机制加上index×scale，再加上d三分钟就下呢就是把内存地址var的内容载入到寄存器eax。
6. 接着我们再来了解一下x86的体系结构。
7. X86系统下呢，有很多不同的寄存器。第1类是数据寄存器。一共有4个分别是ax、bx、cx、dx前面加一个e。第2个是指针寄存器，分别是si、di、bp、sp前面也加一个e。第3类呢是段寄存器分别是cs、ds、ss、es、fs、gs。然后呢，是指令指针寄存器eip。表示下一个将要执行的指令的地址。还有一类是标志寄存器eflags。
8. X86的系统还有4个段基址寄存器。分别是gdtr, ldtr, idtr, tr。用来保存内存分段管理所用的系统表的地址
9. 还有控制寄存器四个，cr0, cr1, cr2, cr3
10. 简单说一下内存寻址。这个我们下一章还会详细的讲啊。先产生一个概念。。叉86系统呢把内存分成了很多个段。每一个段呢相当于一个独立的小空间。在这个小空间以内呀，就用段内的偏移地址。所以当我们要查找一块内存的位置，只需要知道这个段的位置以及这个数据所在段内的偏移地址就可以了。就像是我们去一个小区。先要知道楼号，然后再知道是门牌号就可以了。这里的楼号呢就是段机制，门牌号就是偏移地址。每个段呢都有自己的描述服来记录这个段的一些基本信息。那所有段的描述符呢，都会保存在一个统一的段描述符表中。
11. 这一张图呢就能看出来啊，我们首先从段选择符里面啊，去选择我们的段描述符来获得段基址。然后段基址再加上偏移量就能得到我们实际的地址了。
12. 为了更好的管理内存，x86系统对内存进行了分页。虚拟内存和物理内存的页是一一对应的。但并不是按顺序对应的。这样呢，我们在访问虚拟内存的时候，可以时刻的保证，我们看起来的内存空间是连续的，但其实呢，它在物理内存里面可能是相隔着存放的。这样做的好处呢，是可以最大限度的利用我们物理内存的空间。我们的只要记录。虚拟内存的页和物理内存的页之间的映射关系就可以了。
13. 那为了更好的对页进行管理。 X86系统还引入了页表。专门用来对这一些页进行一个索引记录。这样做的好处呢，就是可以更加的省空间。比如在我们程序的一段虚拟内存空间中，如果只有很少的几个页里面有内容。我们的页表呢，就会标记我只使用这几个页，那剩下的页我就会标记这个是空的，所以呢，空的页是不会占用实际的物理内存空间的，达到了节省内存的目的。页呢，还会分为几级的页表。也可以看作是列表的列表，具体的这些细节呢，我们下一章还会讲。
14. 再来看一下cpu模式的切换。
15. Cpu的第1种模式啊，叫做实模式。实模式是直接访问物理地址。一般只在操作系统刚刚启动的很短暂的一个时间内使用。实模式下的地址的宽是20，所以呢最多只能利用一兆的内存空间。在实模式下呢，所有段都是可读可写可执行的。没有系统权限分级多任务的概念。主要就是处理一些。操作系统初始化的操作
16. 经过了短暂的时模式以后呢，系统会进入保护模式。在保护模式下，总线的宽度是32。也就是说可以访问4G的内存空间。这样呢，可以满足我们绝大的应用需求了。通过分页呢，我们还能支持虚拟内存，从而实现多任务处理。在保护模式下，最大特点就是有了保护。段呢有保护，比如说是否可读，是否可写，是否可执行。那页也是有保护的，每个页也有这些啊，包括一些权限的限制。
17. 在系统启动的时候，x86呢是先进入实模式，然后内核初始化一定程度之后才切换到保护模式，要注意啊，我们是没有办法从保护模式切换到实模式的。
18. Linux驱动先注册总线，总线上可以先挂device，也可以先挂driver，那么究竟怎么控制先后的顺序呢？Linux系统使用两种方式去加载系统中的模块：动态和静态。我们先看静态。
19. 。内核模块呢是linux内核向外部提供的一个插口就像玩具一样，我们可以将其他的模块放到我们的内核上来实现一些功能。一个模块呢，一般是由函数以及内部的数据来组成的，比如实现一种文件系统，一个驱动程序啊，或者其他的一些功能。这些模块，可以直接被编译到内核里面，或者作为独立的一个模块，动态的加载需要的时候再加载。所以linux的内核，就是vmlinux加上内核的一些模块所组成的。
20. 我们编译内核文件的时候，都要有对应的makefile配置文件。在这里，我们可以设置是否编译这个模块，是否编译到内核或者是作为独立的模块存在的。
21. .initcall.init段保存内核模块入口函数的地址。通过vmlinux.lds文件可以看到.initcall.init段中的内容。
22. 我们再看下编译的函数调用顺序。这些函数在启动过程中会逐个被调用。
23. 除了静态加载，在操作系统启动后，也会有模块的动态加载。
24. 内核初始化的最后一步，会启动init进程，init进程会依次执行/etc/rc.d中的启动脚本，其中实现了内核模块的加载、服务的启动等功能系统运行时调用modprobe加载
25. sys\_init\_module()首先调用load\_module()，将内核模块的镜像从用户空间拷贝到内核空间，然后执行符号解析、重定位等工作，将模块对内核接口的调用替换成真实的地址。sys\_init\_module()调用模块自己的init函数，完成模块的初始化。 init由struct module中的init函数指针指向
26. 最后来看一下内核态与用户态的概念以及相互之间的切换。
27. Cpu一共有0~3三种级别的运行模式。其中0级别最高，三级别最低。最高级别呢是留给内核以及驱动程序使用的，它们处于系统的核心态。三级别呢是给普通用户使用的，它是工作在用户态的。
28. 那两种不同的状态就能做一些什么事儿呢？零级别状态可以访问代码，而不受到任何的限制，可以自由的访问任何的地址，可以直接的进行端口的访问。可以说是在操作系统内部畅通无阻。三级别下呢，代码需要受到处理器的很多检查。他们只能访问这些映射地址，空间的列表项中规定的，在用户看一下可以访问的页面的地址。并且呢，对于端口的访问也必须是。
29. 总结来说呢，内核态和用户态，是linux形成的两种运行级别。那么为什么要把内核态和用户态进行区分呢？这段代码很简单，从功能的角度来看，就是实际执行了一个fork()，生成一个新的进程，从逻辑的角度看，就是判断了如果fork()返回的是0则打印相关语句，然后函数最后再打印一句表示执行完整个testfork()函数。代码的执行逻辑和功能上看就是如此简单，一共四行代码，从上到下一句一句执行而已，完全看不出来哪里有体现出用户态和进程态的概念。如果说前面两种是静态观察的角度看的话，我们还可以从动态的角度来看这段代码，即它被转换成CPU执行的指令后加载执行的过程，这时这段程序就是一个动态执行的指令序列。而究竟加载了哪些代码，如何加载就是和操作系统密切相关了。fork的工作实际上是以系统调用的方式完成相应功能的，具体的工作是由sys\_fork负责实施。对于任何操作系统来说，创建一个新的进程都是属于核心功能，因为它要做很多底层细致地工作，消耗系统的物理资源，比如分配物理内存，从父进程拷贝相关信息，拷贝设置页目录页表等等，这些显然不能随便让哪个程序就能去做，于是就自然引出特权级别的概念，显然，最关键性的权力必须由高特权级的程序来执行，这样才可以做到集中管理，减少有限资源的访问和使用冲突。3）用户态和内核态 现在我们从特权级的调度来理解用户态和内核态就比较好理解了，当程序运行在3级特权级上时，就可以称之为运行在用户态，因为这是最低特权级，是普通的用户进程运行的特权级，大部分用户直接面对的程序都是运行在用户态；反之，当程序运行在0级特权级上时，就可以称之为运行在内核态。虽然用户态下和内核态下工作的程序有很多差别，但最重要的差别就在于特权级的不同，即权力的不同。运行在用户态下的程序不能直接访问操作系统内核数据结构和程序，比如上面例子中的testfork()就不能直接调用sys\_fork()，因为前者是工作在用户态，属于用户态程序，而sys\_fork()是工作在内核态，属于内核态程序。当我们在系统中执行一个程序时，大部分时间是运行在用户态下的，在其需要操作系统帮助完成某些它没有权力和能力完成的工作时就会切换到内核态，比如testfork()最初运行在用户态进程下，当它调用fork()最终触发sys\_fork()的执行时，就切换到了内核态。
30. 用户态切换到内核态的3种方式 1 系统调用 这是用户态进程主动要求切换到内核态的一种方式，用户态进程通过系统调用申请使用操作系统提供的服务程序完成工作，比如前例中fork()实际上就是执行了一个创建新进程的系统调用。而系统调用的机制其核心还是使用了操作系统为用户特别开放的一个中断来实现，例如Linux的int 80h中断。2 外围设备的中断 当外围设备完成用户请求的操作后，会向CPU发出相应的中断信号，这时CPU会暂停执行下一条即将要执行的指令转而去执行与中断信号对应的处理程序，如果先前执行的指令是用户态下的程序，那么这个转换的过程自然也就发生了由用户态到内核态的切换。比如硬盘读写操作完成，系统会切换到硬盘读写的中断处理程序中执行后续操作等。3 异常 当CPU在执行运行在用户态下的程序时，发生了某些事先不可知的异常，这时会触发由当前运行进程切换到处理此异常的内核相关程序中，也就转到了内核态，比如缺页异常。
31. 内核态和用户态之间的切换的关键是保存当前进程的上下文环境首先从ss0和esp0指向的内核栈。将当前进程的一些信息保存起来。这个过程也完成了由用户栈到内核栈的切换过程，同时保存了被暂停执行的程序的下一条指令，最后将先前由中断向量检索得到的中断处理程序的cs,eip信息装入相应的寄存器，开始执行中断处理程序