究竟什么是用户态，什么是内核态，这两个基本概念以前一直理解得不是很清楚，根本原因个人觉得是在于因为大部分时候我们在写程序时关注的重点和着眼的角度放在了实现的功能和代码的逻辑性上，先看一个例子：

1）例子

C代码

1. void testfork(){

2. if(0 = = fork()){

3. printf(“create new process success!\n”);

4. }

5. printf(“testfork ok\n”);

6. }

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。

2）特权级

熟悉Unix/Linux系统的人都知道，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。其实无论是不是Unix或者Linux，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。

特权级显然是非常有效的管理和控制程序执行的手段，因此在硬件上对特权级做了很多支持，就Intel x86架构的CPU来说一共有0~3四个特权级，0级最高，3级最低，硬件上在执行每条指令时都会对指令所具有的特权级做相应的检查，相关的概念有 CPL、DPL和RPL，这里不再过多阐述。硬件已经提供了一套特权级使用的相关机制，软件自然就是好好利用的问题，这属于操作系统要做的事情，对于 Unix/Linux来说，只使用了0级特权级和3级特权级。也就是说在Unix/Linux系统中，一条工作在级特权级的指令具有了CPU能提供的最高权力，而一条工作在3级特权级的指令具有CPU提供的最低或者说最基本权力。

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。

2. 用户态和内核态的转换

1）用户态切换到内核态的3种方式

a. 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。

b. 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。

c. 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。

这3种方式是系统在运行时由用户态转到内核态的最主要方式，其中系统调用可以认为是用户进程主动发起的，异常和外围设备中断则是被动的。

2）具体的切换操作

从触发方式上看，可以认为存在前述3种不同的类型，但是从最终实际完成由用户态到内核态的切换操作上来说，涉及的关键步骤是完全一致的，没有任何区别，都相当于执行了一个中断响应的过程，因为系统调用实际上最终是中断机制实现的，而异常和中断的处理机制基本上也是一致的，关于它们的具体区别这里不再赘述。关于中断处理机制的细节和步骤这里也不做过多分析，涉及到由用户态切换到内核态的步骤主要包括：

[1] 从当前进程的描述符中提取其内核栈的ss0及esp0信息。

[2] 使用ss0和esp0指向的内核栈将当前进程的cs,eip,eflags,ss,esp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。

[3] 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，这时就转到了内核态的程序执行了。\

**特权级**

在IA32处理器的分段机制中，采用的是环保护方式，有0~3共4个特权级（Privilege Level），数字越小级别越高。内层的级别高，外层的级别低(三个同心圆)

处理器通过识别CPL、DPL和RPL这三种特权级，来进行特权级检验，从而实现分段内存的保护功能。

l  CPL（Current Privilege Level，当前特权级）——为当前执行的程序或任务的特权级，存储在段寄存器CS和SS的低2位中。

l  DPL（Descriptor Privilege Level，描述符特权级）——为段或门的特权级，存储在段或门描述符的属性低字节中的DPL字段中。

l  RPL（Requested Privilege Level，请求特权级）——为取代（override）特权级，存储在指向调用目标段的段/门选择符的低2位中。

只有CPL的级别高于或等于（即级别数小于或等于）DPL时，当前的程序才有权访问对应的段。

请求/取代特权级被存放在指向调用目标的段选择符中的RPL字段中，当OS的（被调用）过程从一个应用程序（调用过程）接收一个选择符时，就会用该调用过程的特权级RPL，而不是操作系统过程的特权级CPL，来进行特权级检验。只有当RPL的级别高于或等于（即级别数小于或等于）被访问段的DPL时，其对该段数据的访问请求才被允许。所以，OS过程往往使用RPL，来避免低特权级的应用程序，访问高特权级段内的数据。