如果没有其他中断悬而未决，中断控制器将立刻对中断进行处理。如果有另一个中断正在处理中，或者另一个设备在总线上具有更高优先级的一条中断请求线上同时发出中断请求，该设备将暂时不被理睬。在这种情况下，该设备将继续在总线上置起中断信号，直到得到CPU的服务。

为了处理中断，中断控制器在地址线上放置一个数字表明哪个设备需要关注，并且置起一个中断CPU的信号。

中断信号导致CPU停止当前正在做的工作并且开始做其他的事情。地址线上的数字被用做指向一个称为中断向量（interrupt vector）的表格的索引，以便读取一个新的程序计数器。这一程序计数器指向相应的中断服务过程的开始。一般情况下，陷阱和中断从这一点上看使用相同的机制，并且常常共享相同的中断向量。中断向量的位置可以硬布线到机器中，也可以在内存中的任何地方通过一个CPU寄存器（由操作系统装载）指向其起点。

中断服务过程开始运行后，它立刻通过将一个确定的值写到中断控制器的某个I/O端口来对中断做出应答。这一应答告诉中断控制器可以自由地发出另一个中断。通过让CPU延迟这一应答直到它准备好处理下一个中断，就可以避免与多个几乎同时发生的中断相牵涉的竞争状态。说句题外的话，某些（老式的）计算机没有集中的中断控制器，所以每个设备控制器请求自己的中断。

在开始服务程序之前，硬件总是要保存一定的信息。哪些信息要保存以及将其保存到什么地方，不同的CPU之间存在巨大的差别。作为最低限度，必须保存程序计数器，这样被中断的进程才能够重新开始。在另一个极端，所有可见的寄存器和很多内部寄存器或许也要保存。

将这些信息保存到什么地方是一个问题。一种选择是将其放入内部寄存器中，在需要时操作系统可以读出这些内部寄存器。这一方法的问题是，中断控制器之后无法得到应答，直到所有可能的相关信息被读出，以免第二个中断重写内部寄存器保存状态。这一策略在中断被禁止时将导致长时间的死机，并且可能丢失中断和丢失数据。

因此，大多数CPU在堆栈中保存信息。然而，这种方法也有问题。首先，使用谁的堆栈？如果使用当前堆栈，则它很可能是用户进程的堆栈。堆栈指针甚至可能不是合法的，这样当硬件试图在它所指的地址处写某些字时，将导致致命错误。此外，它可能指向一个页面的末端。若干次内存写之后，页面边界可能被超出并且产生一个页面故障。在硬件中断处理期间如果发生页面故障将引起更大的问题：在何处保存状态以处理页面故障？

如果使用内核堆栈，将存在更多的堆栈指针是合法的并且指向一个固定的页面的机会。然而，切换到核心态可能要求改变MMU上下文，并且可能使高速缓存和TLB的大部分或全部失效。静态地或动态地重新装载所有这些东西将增加处理一个中断的时间，因而浪费CPU的时间。

精确中断和不精确中断

另一个问题是由下面这样的事实引起的：现代CPU大量地采用流水线并且有时还采用超标量（内部并行）。在老式的系统中，每条指令完成执行之后，微程序或硬件将检查是否存在悬而未决的中断。如果存在，那么程序计数器和PSW将被压入堆栈中而中断序列将开始。在中断处理程序运行之后，相反的过程将会发生，旧的PSW和程序计数器将从堆栈中弹出并且先前的进程继续运行。

这一模型使用了隐含的假设，这就是如果一个中断正好在某一指令之后发生，那么这条指令前的所有指令（包括这条指令）都完整地执行过了，而这条指令后的指令一条也没有执行。在老式的机器上，这一假设总是正确的，而在现代计算机上，这一假设则未必是正确的。

首先，考虑图1-6a的流水线模型。在流水线满的时候（通常的情形），如果出现一个中断，那么会发生什么情况？许多指令正处于各种不同的执行阶段，当中断出现时，程序计数器的值可能无法正确地反映已经执行过的指令和尚未执行的指令之间的边界。事实上，许多指令可能部分地执行了，不同的指令完成的程度或多或少。在这种情况下，程序计数器更有可能反映的是将要被取出并压入流水线的下一条指令的地址，而不是刚刚被执行单元处理过的指令的地址。

在如图1-7b所示的超标量计算机上，事情更加糟糕。指令可能分解成微操作，而微操作有可能乱序执行，这取决于内部资源（如功能单元和寄存器）的可用性。当中断发生时，某些很久以前启动的指令可能还没开始执行，而其他最近启动的指令可能几乎要完成了。当中断信号出现时，可能存在许多指令