程占用一个进程表项。（有些作者称这些表项为进程控制块。）该表项包含了进程状态的重要信息，包括程序计数器、堆栈指针、内存分配状况、所打开文件的状态、账号和调度信息，以及其他在进程由运行态转换到就绪态或阻塞态时必须保存的信息，从而保证该进程随后能再次启动，就像从未被中断过一样。

图2-4中展示了在一个典型系统中的关键字段。第一列中的字段与进程管理有关。其他两列分别与存储管理和文件管理有关。应该注意到进程表中的字段是与系统密切相关的，不过该图给出了所需要信息的大致介绍。

进程管理	存储管理	文件管理
寄存器	正文段指针	根目录
程序计数器	数据段指针	工作目录
程序状态字	堆栈段指针	文件描述符
堆栈指针		用户ID
进程状态		组ID
优先级		
调度参数		
进程ID		
父进程		
进程组		
信号		
进程开始时间		
使用的CPU时间		
子进程的CPU时间		
下次报警时间		

图2-4 典型的进程表表项中的一些字段

在了解进程表后，就可以对在单个（或每一个）CPU上如何维持多个顺序进程的错觉做更多的阐述。与每一I/O类关联的是一个称作中断向量（interrupt vector）的位置（靠近内存底部的固定区域）。它包含中断服务程序的入口地址。假设当一个磁盘中断发生时，用户进程3正在运行，则中断硬件将程序计数器、程序状态字，有时还有一个或多个寄存器压入堆栈，计算机随即跳转到中断向量所指示的地址。这些是硬件完成的所有操作，然后软件，特别是中断服务例程就接管一切剩余的工作。

所有的中断都从保存寄存器开始，对于当前进程而言，通常是在进程表项中。随后，会从堆栈中删除由中断硬件机制存入堆栈的那部分信息，并将堆栈指针指向一个由进程处理程序所使用的临时堆栈。一些诸如保存寄存器值和设置堆栈指针等操作，无法用C语言这一类高级语言描述，所以这些操作通过一个短小的汇编语言例程来完成，通常该例程可以供所有的中断使用，因为无论中断是怎样引起的，有关保存寄存器的工作则是完全一样的。

1. 硬件压入堆栈程序计数器等。
2. 硬件从中断向量装入新的程序计数器。
3. 汇编语言过程保存寄存器值。
4. 汇编语言过程设置新的堆栈。
5. C中断服务例程运行（典型地读和缓冲输入）。
6. 调度程序决定下一个将运行的进程。
7. C过程返回至汇编代码。
8. 汇编语言过程开始运行新的当前进程。

图2-5 中断发生后操作系统最底层的工作步骤

当该例程结束后，它调用一个C过程处理某个特定的中断类型剩下的工作。（假定操作系统由C语言编写，通常这是所有真实操作系统的选择）。在完成有关工作之后，大概就会使某些进程就绪，接着调用调度程序，决定随后该运行哪个进程。随后将控制转给一段汇编语言代码，为当前的进程装入寄存器值以及内存映射并启动该进程运行。图2-5中总结了中断处理和调度的过程。值得注意的是，各种系统之间某些细节会有所不同。

当该进程结束时，操作系统显示一个提示符并等待新的命令。一旦它接到新命令，就装入新的程序进内存，覆盖前一个程序。

2.1.7 多道程序设计模型

采用多道程序设计可以提高CPU的利用率。严格地说，如果进程用于计算的平均时间是进程在内存中停留时间的20%，且内存中同时有5个进程，则CPU将一直满负载运行。然而，这个模型在现实中过于乐观，因为它假设这5个进程不会同时等待I/O。