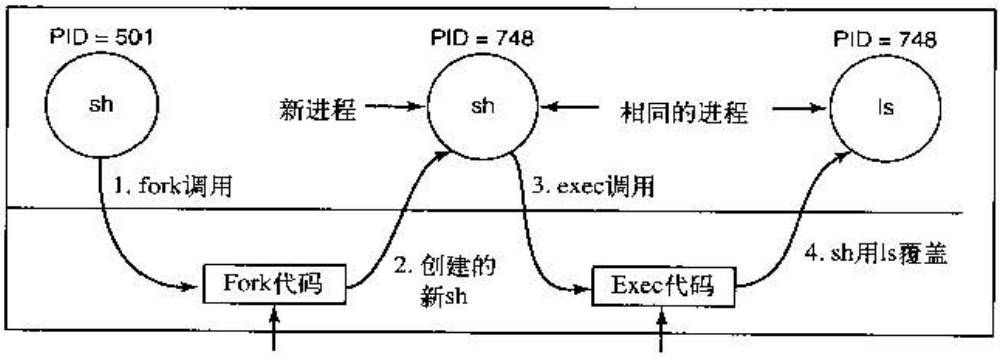
图10-8通过下面的例子解释了上述的步骤：某用户在终端键入一个命令ls，shell调用fork函数复制自身以创建一个新进程。新的shell调用exec函数用可执行文件ls的内容覆盖它的内存。



分配子进程任务数据结构
从父进程处得到数据填写子进程任务数据结构
分配子进程堆栈和用户空间
从父进程处得到数据填写子进程用户空间
为子进程分配PID
设置子进程以共享父进程的正文
为数据和堆栈复制页表
设置共享打开文件
为子进程复制父进程的寄存器

寻找可执行程序
验证执行许可
读取和验证头文件
给内核复制变量、环境参数
释放旧的地址空间
分配新的地址空间
为堆栈复制变量、环境参数
信号复位
初始化寄存器

图10-8 shell执行命令ls的步骤

Linux中的线程

我们在第2章中概括性的介绍了线程。在这里，我们重点关注Linux系统的内核线程，特别是Linux系统中线程模型与其他UNIX系统的不同之处。为了能更好地理解Linux模型所提供的独一无二的性能，我们先来讨论一些多线程操作系统中存在的有争议的决策。

引入线程的最大争议在于维护传统UNIX语义的正确性。首先来考虑fork函数。假设一个多（内核）线程的进程调用了fork系统调用。所有其他的线程都应该在新进程中被创建吗？我们暂时认为答案是肯定的。再假设其他线程中的其中一个线程在从键盘读取数据时被阻塞。那么，新进程中对应的线程也应该被阻塞么？如果是的话，那么哪一个线程应该获得下一行的输入？如果不是的话，新进程中对应的线程又应该做什么呢？同样的问题还大量存在于线程可以完成的很多其他的事情上。在单线程进程中，由于调用fork函数的时候，惟一的进程是不可能被阻塞的，所以不存在这样的问题。现在，考虑这样的情况——其他的线程不会在子进程中被创建。再假设一个没有在子进程中被创建的线程持有一个互斥变量，而子进程中惟一的线程在fork函数结束之后要获得这个互斥变量。那么由于这个互斥变量永远不会被释放，所以子进程中惟一的线程也会永远挂起。还有大量其他的问题存在。但是没有简单的解决办法。

文件输入/输出是另一个问题。假设一个线程由于要读取文件而被阻塞，而另一个线程关闭了这个文件，或者调用lseek函数改变了当前的文件指针。下面会发生什么事情呢？谁能知道？

信号的处理是另一个棘手的问题。信号是应该发送给某一个特定的线程还是发送给线程所在的进程呢？一个浮点运算异常信号SIGFPE应该被引起浮点运算异常的线程所捕获。但是如果它没有捕获到呢？是应该只杀死这个线程，还是杀死线程所属进程中的全部线程？再来考虑由用户通过键盘输入的信号SIGINT。哪一个线程应该捕获这个信号？所有的线程应该共享同样的信号掩码吗？通常，解决这些或其他问题的所有方法会引发另一些问题。使线程的语义正确（不涉及代码）不是一件容易的事。

Linux系统用一种非常值得关注的有趣的方式支持内核线程。具体实现基于4.4BSD的思想，但是在那个版本中内核线程没能实现，因为在能够解决上述问题的C语言程序库被重新编写之前，Berkeley就资金短缺了。

从历史观点上说，进程是资源容器，而线程是执行单元。一个进程包含一个或多个线程，线程之间共享地址空间、已打开的文件、信号处理函数、警报信号和其他。像上面描述的一样，所有的事情简单而清晰。