该文件对该进程中的其他线程都可见，这些线程可以对该文件进行读写。由于资源管理的单位是进程而非线程，所以这种情形是合理的。如果每个线程有其自己的地址空间、打开文件、即将发生的报警等，那么它们就应该是不同的进程了。线程概念试图实现的是，共享一组资源的多个线程的执行能力，以便这些线程可以为完成某一任务而共同工作。

和传统进程一样（即只有一个线程的进程），线程可以处于若干种状态的任何一个：运行、阻塞、就绪或终止。正在运行的线程拥有CPU并且是活跃的。被阻塞的线程正在等待某个释放它的事件。例如，当一个线程执行从键盘读入数据的系统调用时，该线程就被阻塞直到键入了输入为止。线程可以被阻塞，以便等待某个外部事件的发生或者等待其他线程来释放它。就绪线程可被调度运行，并且只要轮到它就很快可以运行。线程状态之间的转换和进程状态之间的转换是一样的，如图2-2所示。

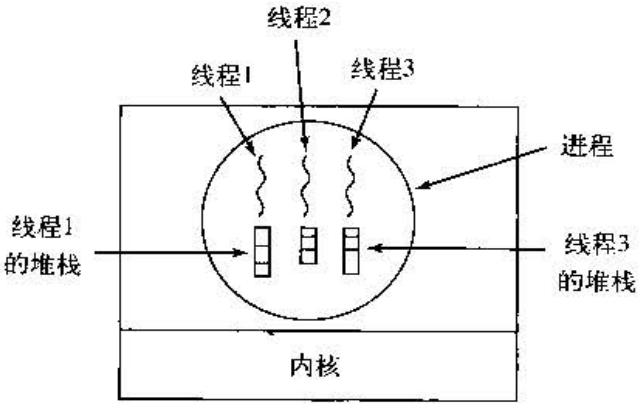

图2-13 每个线程有其自己的堆栈

认识到每个线程有其自己的堆栈很重要，如图2-13所示。每个线程的堆栈有一帧，供各个被调用但是还没有从中返回的过程使用。在该帧中存放了相应过程的局部变量以及过程调用完成之后使用的返回地址。例如，如果过程X调用过程Y，而Y又调用Z，那么当Z执行时，供X、Y和Z使用的帧会全部存在堆栈中。通常每个线程会调用不同的过程，从而有一个各自不同的执行历史。这就是为什么每个线程需要有自己的堆栈的原因。

在多线程的情况下，进程通常会从当前的单个线程开始。这个线程有能力通过调用一个库函数（如thread_create）创建新的线程。thread_create的参数专门指定了新线程要运行的过程名。这里，没有必要对新线程的地址空间加以规定，因为新线程会自动在创建线程的地址空间中运行。有时，线程是有层次的，它们具有一种父子关系，但是，通常不存在这样一种关系，所有的线程都是平等的。不论有无层次关系，创建线程通常都返回一个线程标识符，该标识符就是新线程的名字。

当一个线程完成工作后，可以通过调用一个库过程（如thread_exit）退出。该线程接着消失，不再可调度。在某些线程系统中，通过调用一个过程，例如thread_join，一个线程可以等待一个（特定）线程退出。这个过程阻塞调用线程直到那个（特定）线程退出。在这种情况下，线程的创建和终止非常类似于进程的创建和终止，并且也有着同样的选项。

另一个常见的线程调用是thread_yield，它允许线程自动放弃CPU从而让另一个线程运行。这样一个调用是很重要的，因为不同于进程，（线程库）无法利用时钟中断强制线程让出CPU。所以设法使线程行为“高尚”起来，并且随着时间的推移自动交出CPU，以便让其他线程有机会运行，就变得非常重要。有的调用允许某个线程等待另一个线程完成某些任务，或等待一个线程宣称它已经完成了有关的工作等。

通常而言，线程是有益的，但是线程也在程序设计模式中引入了某种程度的复杂性。考虑一下UNIX中的fork系统调用。如果父进程有多个线程，那么它的子进程也应该拥有这些线程吗？如果不是，则该子进程可能会工作不正常，因为在该子进程中的线程都是绝对必要的。

然而，如果子进程拥有了与父进程一样的多个线程，如果父进程在read系统调用（比如键盘）上被阻塞了会发生什么情况？是两个线程被阻塞在键盘上（一个属于父进程，另一个属于子进程）吗？在键入一行输入之后，这两个线程都得到该输入的副本吗？还是仅有父进程得到该输入的副本？或是仅有子进程得到？类似的问题在进行网络连接时也会出现。

另一类问题和线程共享许多数据结构的事实有关。如果一个线程关闭了某个文件，而另一个线程还在该文件上进行读操作时会怎样？假设有一个线程注意到几乎没有内存了，并开始分配更多的内存。在工作一半的时候，发生线程切换，新线程也注意到几乎没有内存了，并且也开始分配更多的内存。这样，内存可能会分配两次。不过这些问题通过努力是可以解决的。总之，要使多线程的程序正确工作，就需要仔细思考和设计。