指针、寄存器和状态等。该线程表由运行时系统管理。当一个线程转换到就绪状态或阻塞状态时，在该线程表中存放重新启动该线程所需的信息，与内核在进程表中存放进程的信息完全一样。

当某个线程做了一些会引起在本地阻塞的事情之后，例如，等待进程中另一个线程完成某项工作，它调用一个运行时系统的过程，这个过程检查该线程是否必须进入阻塞状态。如果是，它在线程表中保存该线程的寄存器（即它本身的），查看表中可运行的就绪线程，并把新线程的保存值重新装入机器的寄存器中。只要堆栈指针和程序计数器一被切换，新的线程就又自动投入运行。如果机器有一条保存所有寄存器的指令和另一条装入全部寄存器的指令，那么整个线程的切换可以在几条指令内完成。进行类似于这样的线程切换至少比陷入内核要快一个数量级（或许更多），这是使用用户级线程包的极大的优点。

不过，线程与进程有一个关键的差别。在线程完成运行时，例如，在它调用thread_yield时，pthread_yield代码可以把该线程的信息保存在线程表中，进而，它可以调用线程调度程序来选择另一个要运行的线程。保存该线程状态的过程和调度程序都只是本地过程，所以启动它们比进行内核调用效率更高。另一方面，不需要陷阱，不需要上下文切换，也不需要对内存高速缓存进行刷新，这就使得线程调度非常快捷。

用户级线程还有另一个优点。它允许每个进程有自己定制的调度算法。例如，在某些应用程序中，那些有垃圾收集线程的应用程序就不用担心线程会在不合适的时刻停止，这是一个长处。用户级线程还具有较好的可扩展性，这是因为在内核空间中内核线程需要一些固定表格空间和堆栈空间，如果内核线程的数量非常大，就会出现问题。

尽管用户级线程包有更好的性能，但它也存在一些明显的问题。其中第一个问题是如何实现阻塞系统调用。假设在还没有任何击键之前，一个线程读取键盘。让该线程实际进行该系统调用是不可接受的，因为这会停止所有的线程。使用线程的一个主要目标是，首先要允许每个线程使用阻塞调用，但是还要避免被阻塞的线程影响其他的线程。有了阻塞系统调用，这个目标不是轻易地能够实现的。

系统调用可以全部改成非阻塞的（例如，如果没有被缓冲的字符，对键盘的read操作可以只返回0字节），但是这需要修改操作系统，所以这个办法也不吸引人。而且，用户级线程的一个长处就是它可以在现有的操作系统上运行。另外，改变read操作的语义需要修改许多用户程序。

在这个过程中，还有一种可能的替代方案，就是如果某个调用会阻塞，就提前通知。在某些UNIX版本中，有一个系统调用select可以允许调用者通知预期的read是否会阻塞。若有这个调用，那么库过程read就可以被新的操作替代，首先进行select调用，然后只有在安全的情形下（即不会阻塞）才进行read调用。如果read调用会被阻塞，有关的调用就不进行，代之以运行另一个线程。到了下次有关的运行系统取得控制权之后，就可以再次检查看看现在进行read调用是否安全。这个处理方法需要重写部分系统调用库，所以效率不高也不优雅，不过没有其他的可选方案了。在系统调用周围从事检查的这类代码称为包装器（jacket或wrapper）。

与阻塞系统调用问题有些类似的是页面故障问题。我们将在第3章讨论这些问题。此刻可以认为，把计算机设置成这样一种工作方式，即并不是所有的程序都一次性放在内存中。如果某个程序调用或者跳转到了一条不在内存的指令上，就会发生页面故障，而操作系统将到磁盘上取回这个丢失的指令（和该指令的“邻居们”），这就称为页面故障。在对所需的指令进行定位和读入时，相关的进程就被阻塞。如果有一个线程引起页面故障，内核由于甚至不知道有线程存在，通常会把整个进程阻塞直到磁盘I/O完成为止，尽管其他的线程是可以运行的。

用户级线程包的另一个问题是，如果一个线程开始运行，那么在该进程中的其他线程就不能运行，除非第一个线程自动放弃CPU。在一个单独的进程内部，没有时钟中断，所以不可能用轮转调度（轮流）的方式调度进程。除非某个线程能够按照自己的意志进入运行时系统，否则调度程序就没有任何机会。

对线程永久运行问题的一个可能的解决方案是让运行时系统请求每秒一次的时钟信号（中断），但是这样对程序也是生硬和无序的。不可能总是高频率地发生周期性的时钟中断，即使可能，总的开销也是可观的。而且，线程可能也需要时钟中断，这就会扰乱运行时系统使用的时钟。

再者，也许反对用户级线程的最大负面争论意见是，程序员通常在经常发生线程阻塞的应用中才希望使用多个线程。例如，在多线程Web服务器里。这些线程持续地进行系统调用，而一旦发生内核陷阱进行系统调用，如果原有的线程已经阻塞，就很难让内核进行线程的切换，如果要让内核消除这种情形，