个方案创建了新的作用域层，这些变量对一个线程中所有过程都是可见的。而在原先的作用域层里，变量只对一个过程可见，并在程序中处处可见。

访问私有的全局变量需要有些技巧，不过，多数程序设计语言具有表示局部变量和全局变量的方式，而没有中间的形式。有可能为全局变量分配一块内存，并将它转送给线程中的每个过程作为额外的参数。尽管这不是一个漂亮的方案，但却是一个可用的方案。

还有另一种方案，可以引入新的库过程，以便创建、设置和读取这些线程范围的全局变量。首先一个调用也许是这样的：

```
create_global("bufptr");
```

该调用在堆上或在专门为调用线程所保留的特殊存储区上替一个名为bufptr的指针分配存储空间。无论该存储空间分配在何处，只有调用线程才可访问其全局变量。如果另一个线程创建了同名的全局变量，由于它在不同的存储单元上，所以不会与已有的那个变量产生冲突。

访问全局变量需要两个调用：一个用于写入全局变量，另一个用于读取全局变量。对于写入，类似有

```
set_global("bufptr", &buf);
```

它把指针的值保存在先前通过调用create_global创建的存储单元中。如果要读出一个全局变量，调用的形式类似于

```
bufptr = read_global("bufptr");
```

这个调用返回一个存储在全局变量中的地址，这样就可以访问其中的数据了。

试图将单一线程程序转为多线程程序的另一个问题是，有许多库过程并不是可重入的。也就是说，它们不是被设计成下列工作方式的：对于任何给定的过程，当前面的调用尚没有结束之前，可以进行第二次调用。例如，可以将通过网络发送消息恰当地设计为，在库内部的一个固定缓冲区中进行消息组合，然后陷入内核将其发送。但是，如果一个线程在缓冲区中编好了消息，然后被时钟中断强迫切换到第二个线程，而第二个线程立即用它自己的消息重写了该缓冲区，那会怎样呢？

类似地还有内存分配过程，例如UNIX中的malloc，它维护着内存使用情况的关键表格，如可用内存块链表。在malloc忙于更新表格时，有可能暂时处于一种不一致的状态，指针的指向不定。如果在表格处于一种不一致的状态时发生了线程切换，并且从一个不同的线程中来了一个新的调用，就可能会由于使用了一个无效指针从而导致程序崩溃。要有效的解决所有这些问题意味着重写整个库。做这件事并非是无效的行为。

另一种解决方案是，为每个过程提供一个包装器，该包装器设置一个二进制位从而标志某个库处于使用中。在先前的调用还没有完成之前，任何试图使用该库的其他线程都会被阻塞。尽管这个方式可以工作，但是它会极大地降低系统潜在的并行性。

接着考虑信号。有些信号逻辑上是线程专用的，但是另一些却不是。例如，如果某个线程调用alarm，信号送往进行该调用的线程是有意义的。但是，当线程完全在用户空间实现时，内核根本不知道有线程存在，因此很难将信号发送给正确的线程。如果一个进程一次仅有一个警报信号等待处理，而其中的多个线程又独立地调用alarm，那么情况就更加复杂了。

有些信号，如键盘中断，则不是线程专用的。谁应该捕捉它们？一个指定的线程？所有的线程？还是新创建的弹出式线程？进而，如果某个线程修改了信号处理程序，而没有通知其他线程，会出现什么情况？如果某个线程想捕捉一个特定的信号（比如，用户击键CTRL+C），而另一个线程却想用这个信号终止进程，又会发生什么情况？如果有一个或多个线程运行标准的库过程以及其他用户编写的过程，那么情况还会更复杂。很显然，这些想法是不兼容的。一般而言，在单线程的环境中信号已经是很难管理的了，到了多线程环境中并不会使这一情况变得容易处理。

由多线程引入的最后一个问题是堆栈的管理。在很多系统中，当一个进程的堆栈溢出时，内核只是自动为该进程提供更多的堆栈。当一个进程有多个线程时，就必须有多个堆栈。如果内核不了解所有的堆栈，就不能使它们自动增长，直到造成堆栈出错。事实上，内核有可能还没有意识到内存错是和某个线程栈的增长有关系的。

这些问题当然不是不可克服的，但是却说明了给已有的系统引入线程而不进行实质性的重新设计系