一、内核对象可在多个进程共享，方法是：

1）子进程继承父进程，但默认是不能继承的，需要手动更改S E C U R I T Y \_ AT T R I B U T E S结构中的标明继承性的字段；

2）在不同进程中create名字相同的内核对象，或者在一个进程中create，另一个进程中open；

3）使用服务器的名字空间；

4）使用 DuplicateHandle函数；

二、

8.1 原子访问：互锁的函数家族（InterlockedExchange等）

原子访问即为线程在访问资源时能够确保所有其他线程都不在同一时间内访问相同资源；

没有任何互锁函数仅仅负责对值进行读取操作而不改变值，因为这样的函数根本是不需要的。如果线程只是试图读取值的内同，而这个值始终都由互锁函数来修改，那么被读取的值总是一个很好的值，虽然不知道读取的是原始值还是更新值。

8.4 线程同步

用户方式（即使用CRITICAL\_SECTION变量去进行控制）同步的优点是它的同步速度非常快。只能对单个进程中的线程实施同步。

当线程试图进入另一个线程拥有的关键代码段时，调用线程就立即被置于等待状态。这意味着该线程必须从用户方式转入内核方式（大约 1 0 0 0个C P U周期）。这种转换是要付出很大代价的。在多处理器计算机上，当前拥有资源的线程可以在不同的处理器上运行，并且能够很快放弃对资源的控制。 实际上拥有资源的线程可以在另一个线程完成转入内核方式之前释放资源。如果出现这种情况，就会浪费许多C P U时间。为了提高关键代码段的运行性能， M i c r o s o f t将循环锁纳入了这些代码段。因此，当

EnterCriticalSection函数被调用时，它就使用循环锁进行循环，以便设法多次取得该资源。只有当为了取得该资源的每次试图都失败时，该线程才转入内核方式，以便进入等待状态。

与InitializeCriticalSection中的情况一样， InitializeCriticalSectionAndSpinCount的第一个参数是关键代码段结构的地址。但是在第二个参数 dwSpinCount中，传递的是在使线程等待之前它试图获得资源时想要循环锁循环迭代的次数。 这个值可以是0至0x00FFFFFF之间的任何数字。如果在单处理器计算机上运行时调用该函数， dwSpinCount参数将被忽略，它的计数始终被置为0。这是对的，因为在单处理器计算机上设置循环次数是毫无用处的，如果另一个线程正在循环运行，那么拥有资源的线程就不能放弃它。

InitializeCriticalSection函数返回void，若失败则无法追踪问题；InitializeCriticalSectionAndSpinCount，如果内存无法分配，那么它就返回 FA L S E

9.1使用内核对象进行同步

内核对象机制的适应性远远优于用户方式机制。实际上，内核对象机制的唯一不足之处是它的速度比较慢。当调用本章中提到的任何新函数时，调用线程必须从用户方式转为内核方式。这个转换需要很大的代价：往返一次需要占用 x 8 6平台上的大约 1 0 0 0个C P U周期，当然，这还不包括执行内核方式代码，即实现线程调用的函数的代码所需的时间。

1）WaitForSingleObject：等待函数可使线程自愿进入等待状态，直到一个特定的内核对象变为已通知状态为止；如果线程等待的对象变为已通知状态，那么返回值是 WAIT\_OBJECT\_0。如果设置的超时已经到期，则返回值是 WAIT\_TIMEOUT。如果将一个错误的值（如一个无效句柄）传递给 WaitForSingleObject，那么返回值将是WAIT\_FAILED；

WaitForMultipleObject

2）event事件内核对象：可设置为自动重置和人工重置，当设为自动重置时，调用setevent，事件变为已通知状态，但多个线程在waitforsingleobject时只能有一个线程变为可调度，然后该event自动重置为未通知状态；当设为人工重置时，多个等待该事件的线程都会变成可调度，之后需要人工设置event为未通知状态；

3）等待定时器内核对象：createwaitabletimer,setwaitabletimer,也是使用waitforsingleobject去等待；可以在某个指定的时间触发，也可以每隔一定时间触发一次；

4）信标内核对象：用于资源计数，比如服务器进程的线程池有5个线程，当前资源数量大于0，发出信标信号，使得线程变为可调度，接收用户请求；当前资源数量等于0，不发出信标信号

createsemaphore,releasesemphor

5）互斥对象内核对象：mutex。不同进程的多个线程能够访问单个互斥对象。

createmutex

十一章、Windows线程池

Windows提供了一个（与I/O完成端口想配套的）线程池机制来简化线程的创建、销毁及日常管理。这个行的通用线程池不可能适用于全部情况，但很多情况下能为我们节省大量时间。这些线程池允许我们做以下这些事情：

1）以异步方式调用一个函数；

2）每隔一段时间调用一个函数；

3）当内核对象触发的时候调用一个函数；

4）当异步I/O请求完成的时候调用一个函数；

11.1 以异步方式调用一个函数

首先定义一个SimpleCallback回调函数(有函数原型)，再通过TrySubmitThreadppolCallback函数将该回调函数作为工作项添加到线程池的工作项队列；

注意，我们从来不需要自己调用CreateThread，系统会自动为我们的进程创建一个默认的线程池，炳然线程池中的一个线程来调用我们的回调函数，当这个线程处理完一个客户请求后，它不会立刻被销毁，而是会回到线程池中，准备好处理队列中的其他工作项。

当不需要一个工作项时，应调用CloseThreadpoolWork并在参数中传入该工作项的指针；

11.2每隔一段时间调用一个函数

许多开发人员会为应用程序需要执行的每隔基于时间的任务创建一个可等待的计时器，但这是不必要的，而且会浪费系统资源。我们应该只创建一个可等待的计时器，设置它下一次应该触发的时间，将它重置，并等待下一次出发后，再重复这一过程。

为了将一个工作项安排在某个时间执行，我们定义一个TimeoutCallback回调函数(有函数原型)，再调用CreateThreadpoolTimer来通知线程池何时调用我们的函数。通过SetThreadpoolTimer函数向线程池注册计时器

11.3、在内核对象触发时调用一个函数

定义一个WaitCallback函数，通过CreateThreadpoolWait来创建一个线程池等待对象，再调用SetThreadpoolWait来将一个内核对象绑定到这个线程池。当该对象被触发时，会导致线程池调用我们的WaitCallback函数；

11.4、在异步I/O请求完成时调用一个函数

在打开一个文件或者设备时，我们必须先将改文件/设备与线程池的I/O完成端口关联起来，然后告诉线程池，当发往温江/设备的异步I/O请求完成时，应该调用哪个函数。

定义一个overlappedCompletionRoutine函数，再调用CreateThreadpoolio来创建一个线程池I/O对象，并将想要与线程池内部的I/O完成端口相关联的文件设备句柄

传入。

11.5.1对线程池进行定制

在调用CreateThreadpoolWork、CreateThreadpoolTimer、CreateThreadpoolWait、CreateThreadpoolio的时候，我们能传入一个PTP\_CALLBACK\_ENVIRON参数，如果传给这个参数的值是NULL，我们将工作项添加到进程默认的线程池中，默认的线程池配置能够很好满足大多数应用程序的要求；

当然，我们也可以手动创建一个线程池，使用CreateThreadpool函数。通过SetThreadpoolThreadMinimun和SetThreadpoolThreadMaximum来设置线程中额最大数量和最小数量；

默认线程池的最小数量为1，最大数量为500.

一旦创建了自己的线程池，并指定线程的最小最大数量，我们就可以初始化一个回调环境，它包含了一些可应用与工作项的额外的设置和配置。线程池的回调环境的数据结构为\_TP\_CALLBACK\_ENVIRON，在WinNT.h中定义，可以调用InitializeThreadpoolEnvironment函数来初始化这个结构，可以调用DestroyThreadpoolEnvironment函数来初清理这个结构。

为了将一个工作项添加到线程池的队列中，回调环境必须标明该工作项应该由哪个线程池来处理，可以调用SetThreadpoolCallbackPool来指定一个线程池，某则工作项将被添加到进程默认的线程池。使用SetThreadpoolCallbackRunsLong函数来告诉回调环境，工作项通常需要较长时间来处理。这使得线程池会更快地创建线程。

CloseThreadpool可以销毁自定制的线程池。

11.5.2、销毁线程：清理组

默认的线程池的生命期与进程相同，在进程终止时Windows会将其销毁并负责所有清理工作；

对于自定义的线程池，线程池提供了清理组来进行销毁；使用CreateThreadpoolCleanupGroup来创建一个清理组，再调用SetThreadpoolCallbackCleanupGroup将清理组与一个绑定到线程池的回调环境关联起来。

三、动态库DLL

1、动态库的优点：

1）便于应用程序的扩展；

2）有助于资源共享，减少RAM的占用，因为多个应用程序可以用一份动态库；

3）有助于解决平台差异。可以根据平台或者系统版本选择性的导入DLL中的函数；

2、DLL被映射到进程的地址空间中后，则可以被进程中的所有线程使用。DLL函数代码创建的任何对象均由调用线程所拥有，而DLL本身从来不拥有任何东西。

3、DLL提供分配内存的接口函数时，也必须提供释放资源的接口函数；

4、LoadLibrary(DLLpathName)和LoadLibraryEx(DLLPathName, HANDLE hFile, DWORD dwFlags)都能load入DLL，后者能使用dwFlags参数进行一些设置；

dwFlags为LOAD\_WITH\_ALTERED\_SEARCH\_PATH用来改变LoadLibraryEx搜索指定DLL时的搜索路径，分为以下几种情形：

1）DLLPathName不包含 \ 字符，按标准路径去搜索（包含可执行映像文件的目录->进程的当前目录->Windows系统目录->Windows目录->PATH环境变量中列出的各个目录）;

2）DLLPathName包含绝对路径，则直接搜索该路径文件，若不存在则返回NULL，不再搜索其他路径；

3）DLLPathName包含相对路径，则会将DLLPathName与下列文件夹连接起来：进程的当前目录，Windows系统目录，16位的Windows系统目录，Windows目录，PATH环境变量中列出的各个目录。

5、SetDllDirectory函数可以指定LoadLibrary和LoadLibraryEx的搜索路径：进程的当前目录->SetDllDirectory指定的路径->Windows系统目录->16位的Windows系统目录->Windows目录->PATH环境变量中列出的各个目录.

6、系统将一个DLL第一次导入到进程的地址空间时，会调用DllMain函数，并收到DLL\_PROCESS\_ATTACK通知，但不会收到DLL\_THREAD\_ATTACK通知。应该避免在DLLMain中调用LoadLibrary和FreeLibrary，因为这些函数可能会产生循环依赖；

7、需要注意一种情况，在一个线程中LoadLibrary一个DLL，系统调用DLLMain，收到的是DLL\_PROCESS\_ATTACK通知，当该线程退出时，DLLMain收到的是DLL\_THREAD\_DETACK通知。

8、DLLMain的序列化调用：一个进程中有多个线程，进程中已经加载了一个DLL，当多个子线程又同时加载该DLL，则会按调用先后顺序执行DLLMain，后者将会挂起。

9、VS编译器可以选择延迟载入DLL，以实现根据系统选择性加载DLL。