Windows 核心编程笔记

By Ken 2008-7-31

(一) 第一部分——程序员必读

1. 对程序错误的处理

- **Knowledge Points**
- ✧ 在 Windows 系统内部，当一个 Windows 函数检测到一个错误时，它会使用线程本地存储的机制，将相应的错误代码与调用的线程关联起来
- ✧ WinError.h 包含了 Windows 定义的错误代码
- ✧ Windows 函数调用失败时，应该立即调用 GetLastError 函数。如果调用另外一个 Windows 函数，它的错误值可能被改写
- **API Quick Reference**
- ✧ FormatMesage 用于将错误代码转换成文本描述，该函数功能非常丰富
- ✧ SetLastError 用于用户可以定义自己的错误代码（代码格式有要求）
- **Code Style**
- 当写一个函数时，首先应该检查各个参数的有效性。当调用一个函数时，应该始终检查它的返回值的合法性

2. Unicode

- **Knowledge Points**
- ✧ Unicode 的好处
  1) 可以很容易地在不同语言之间进行数据交换
  2) 使你能够发布支持所有语言的单个二进制.exe 或 DLL 文件
  3) 提高应用程序的运行效率（Win2000 是使用 Unicode 开发的）
- ✧ WinCE 完全不支持 ANSI，只能用 Unicode 开发
- ✧ 需要字符串的所有 COM 接口只能接受 Unicode 字符串
- ✧ ANSI C 规定，C 运行期库支持 Unicode 字符和字符串，所以始终都可以调用 C 运行期库来操作 Unicode 字符和字符串。
- ✧ _UNICODE 宏用于 C 运行期头文件，而 UNICODE 宏用于 Windows 头文件。当编译源代码模块时，通常必须同时定义这两个宏。
- ✧ Windows 提供了很多字符串操作函数，应该首先考虑使用这些函数，而不是 C 运行期库函数，这样有助于提高程序的性能。这些函数类似于：StrCat，StrChr，StrCmp 和 StrCpy，都是以 Windows 的命名方式命名的
- **API Quick Reference**
- ✧ CharNext，CharPrev，IsDBCSLeadByte 用于操作 DBCS 字符串
- ✧ 标准 ANSI C 字符串函数和他们等价的 Unicode 函数

strchr	wcschr

strcmp	wcscmp
strcpy	wcscpy
strlen	wcslen
strcat	wcscat
strtok	wcstok
Note :所有的 ANSI 字符串操作函数都以 str 开头；所有的 Unicode 版本都以 wcs（Wide Character Set）开头。若要调用 Unicode 版本函数，只需要将前缀 wcs 来取代 ANSI 版本的前缀 str 即可	

- ✧ Windows 的字符串操作函数

lstrcat
lstrcmp
lstrcmpi
lstrcpy
lstrlen
Note：这些函数都定义为宏，可以操作 Unicode 和 ANSI 字符串，其中的 l 意思是 literal string

- ✧ Windows 定义的用于处理 Unicode 和 ANSI 字符串函数

CharLower	CharUpper
CharLowerBuff	CharUpperBuff
IsCharAlpha	IsCharAlphaNumeric
IsCharLower	IsCharUpper

- ✧ Windows 定义的 C 运行期 printf 函数族

	ASCII	宽字符	常规
参数的形式为 list（参数列表格式）			
标准版	sprintf	swprintf	_stprintf
最大长度版	_snprintf	_snwprintf	_sntprintf
Windows 版	wsprintfA	wsprintfW	wsprintf
参数的形式为 vector（数组格式）			
标准版	vsprintf	vswprintf	_vstprintf
最大长度版	_vsnprintf	_vsnwprintf	_vsntprintf
Windows 版	wvsprintfA	wvsprintfW	wvsprintf

Note：char szA[100];WCHAR szW[100];
spritnf(szA, ”%s”, “ANSI Str”)//normal ANSI str
sprintf(szA, ”%S”,L”Unicode Str”)//Unicode to ANSI
swprintf(szW, L”%s”, L”Unicode Str”)// normal Unicode str
swprintf(szW, L”%S”, “ANSI Str”)// ANSI to Unicode			

- ✧ MultiByteToWideChar 用于将多字节字符串转换为 Unicode 字符串
- ✧ WideCharToMultiByte 用于将 Unicode 字符串转换为多字节字符串
- ✧ GetThreadLocal 用于得到线程的当前语言设置

➢ **Code Style and Technique Tricks**

若要创建动态链接库（DLL），应该考虑这种方法：在 DLL 中提供两个输出函数。一个是 ANSI 版本，另一个是 Unicode 版本。在 ANSI 版本中，只需要分配内存，执行必要的字符串转换，并调用该函数的 Unicode 版本。

3. 内核对象

➢ **Knowledge Points**

✧ 每个内核对象只是内核分配的一个内存块，并且只能由内核访问。该内存块是一种数据结构，它的成员负责维护该内核对象的各种信息。内核对象是由内核所拥有，而不是由创建它的进程所拥有。所以内核对象的生存时间可能比创建它的进程的生存时间要长。内核知道有多少进程在使用该内核对象，因为每个内核对象包含一个引用计数。

✧ 要判断一个对象是否属于内核对象，最容易的办法是观察创建该对象的所用的函数。创建内核对象的所有函数几乎都有 PSECURITY_ATTRIBUTES 参数用于设置安全属性，而用于创建用户对象或 GDI 对象的函数都没有该参数

✧ 当一个进程被初始化时，系统要为它分配一个内核对象句柄表，其实是一个数据结构数组，每个结构包含一个指向内核对象的内存指针、一个访问掩码和其它标志。该句柄表只用于内核对象，不用于用户对象和 GDI 对象。

✧ 创建内核对象的函数都返回与进程相关的句柄。这个句柄只能被在相同的进程中运行的所有线程使用。该句柄值实际上是进程句柄表的索引，用于标识内核对象的信息存放的位置

✧ HANDLE、HINSTANCE、HMODULE 比较。HANDLE 句柄用于标志内核对象，它的值其实就是进程的内核对象句柄表的索引，它和进程紧密相关的，不能由其它进程成功使用。HINSTANCE 和 HMODULE 对象，其实两者是一样的，都是可执行文件的映像加载到进程的地址空间的基址。在要求传递 HMODULE 参数的地方可以传递 HINSTANCE 参数，反之亦然，它们都指向某个模块的内存基址（exe 或 dll）。

✧ 跨越进程边界共享内核对象的三种机制：

1) 利用内核对象句柄的继承性。该方法只能用于父子进程，且子进程中的句柄值和父进程中的句柄值是一样的。虽然内核对象句柄具有继承性，但内核对象本身不具备继承性。
2) 利用命名对象。
3) 复制对象句柄

➢ **API Quick Reference**

✧ CloseHandle 用于关闭句柄

✧ SetHandleInformation 用于操作内核对象，比如设置标志等等

✧ DuplicateHandle 用于复制句柄

✧ Windows 的一些用于操作内核对象的函数

CreateMutex	OpenMutex
CreateEvent	OpenEvent
CreateSemaphore	OpentSemaphore
CreateWaitableTimer	OpenWaitableTimer
CreateFileMapping	OpenFileMapping
CreateJobObject	OpenJobObject

➢ **Code Style and Technique Tricks**

(二)　第二部分——编程的具体方法

1. 进程

➢ **Knowledge Points**

✧ 一个正在运行的程序的实例就是一个进程，它由两部分组成

1) 一个是操作系统用来管理进程的内核对象。该内核对象是系统用来存放关于进程的统计信息的地方
2) 另外一个是地址空间，它包含可执行模块或 DLL 模块的代码和数据。它还包含动态内存分配空间。

✧ Windows 中进程 ID 是全系统唯一的

✧ Windows 应用程序启动时必须有个入口函数，比如 main。操作系统并不调用应用程序的进入点函数，而是调用 C/C++运行期启动函数，然后启动函数再调用应用程序的入口函数。该启动函数负责对 C/C++运行期库进行初始化。应用程序的进入点函数和对应的 C/C++启动函数对应如下：

应用程序类型	应用程序入口函数	C/C++运行期启动函数
ANSI 类型的 GUI 程序	WinMain	WinMainCRTStartup
Unicode 类型的 GUI 程序	wWinMain	wWinMainCRTStartup
ANSI 类型的 CUI 程序	main	mainCRTStartup
Unicode 类型的 CUI 程序	wmain	wmainCRTStartup

✧ C/C++启动函数功能

1) 检索指向新进程的完整命令行指针
2) 检索指向新进程的环境变量指针
3) 对 C/C++运行期的全局变量进行初始化。
4) 对 C 运行期内存单元分配函数（malloc 和 calloc）和其它底层输入/输出例程使用的内存堆进行初始化
5) 为所有的全局和静态 C++类对象调用构造函数

当应用程序的进入点函数返回时，启动函数便调用 C 运行期的 exit 函数。exit 函数负责下面的操作

1) 调用由_onexit 函数注册的所有函数
2) 为所有的全局的和静态的 C++对象调用析构函数
3) 调用操作系统的 ExitProcess 函数结束进程

➢ **API Quick Reference**

✧ GetModuleHandle 用于返回可执行文件或 DLL 文件加载到进程地址空间的句柄/基址，它只查看调用它线程的进程的地址空间

✧ GetModuleFileName 用于获得 DLL 或.exe 的全路径名

✧ GetCommandLine 用于获得指向进程的完整命令行

✧ CommandLineToArgvW 用于把 Unicode 字符串分割成一个个标记

✧ GetEnvironmentVariable 用于得到某个环境变量的值

✧ SetEnvironmentVariable 用于添加、删除或修改环境变量

✧ ExpandEnvironmentStrings 用于把含通配符的环境变量展开

✧ GetCurrentDirectory 用于获得进程的当前驱动器和目录

✧ SetCurrentDirectory 用于设置进程的当前驱动器和目录

✧ GetFullPathName 用于获得进程的当前目录

- GetVersion(Ex)用于获得操作系统版本
- VerifyVersionInfo 用于对系统版本和应用程序需要的版本进行比较
- CreateProcess 用于创建进程
- ExitProcess 用于终止进程（应用程序不应该显示调用）
- TerminateProcess 用于终止进程（应用程序不应该显示调用）
- GetCurrentProcess 用于返回进程内核对象伪句柄，可用 DuplicateHandle 将进程的伪句柄转换为进程的实句柄
- GetCurrentProcessId 用于查询进程自己的 ID
- GetProcessTimes 用于查询进程的各种时间使用情况

Code Style and Technique Tricks

应用程序通常使用环境变量来使用户调整它的行为特性。用户创建一个环境变量并对它进行初始化。然后，当用户启动应用程序运行时，该应用程序查看环境块，并根据环境变量来调整自己的行为特性

2. 作业

Knowledge Points

- 很多时候必须将一组进程当作单个实体来处理，作业（JOB）提供了这个功能，它能够有效地限制进程能够进行的操作。可以把作业看成是一个包含进程的容器
- 作业能够对加入其的进程进行四种限制
  1) 基本限制（用于防止作业中的进程垄断系统资源）
  2) 扩展限制（用于防止作业中的进程垄断系统资源）
  3) 基本 UI 限制（用于防止作业中的进程改变用户界面）
  4) 安全性限制（用于防止作业中的进程访问私有资源）
- 当作业中的所有进程终止运行或分配的所有 CPU 时间用完，或加入了新进程或某个进程终止了，作业都会得到通知

API Quick Reference

- CreateJobObjects 用于创建作业内核对象
- OpenJobObjects 用于对命名的作业对象进行访问
- SetInformationJobObjects 用于对作业施加各种限制
- QueryInformationJobObject 用于获取作业的统计信息
- AssignProcessToJobObject 用于将进程加入作业
- TerminateJobObject 用于终止作业中的所有进程
- GetProcessIoCounters 用于获取不是作业中的进程的读写和非读写的数量

Code Style and Technique Tricks

3. 线程基础知识

Knowledge Points

- 线程由两个部分组成
  1) 一个是线程的内核对象，操作系统用它来对线程实施管理。线程内核对象是系统用来存放线程统计信息的地方
  2) 另一个是线程堆栈，线程用它来维护执行代码时需要的所有函数的参数和局部变量
- Windows 中线程 ID 是全系统唯一的

- 每个线程都有它的一组 CPU 寄存器，称之为上下文（Context），用来反映线程上次运行时该线程的 CPU 寄存器状态。Context 结构被包含在线程内核对象结构里面

➢ **API Quick Reference**

- CreateThread 用于创建线程（C/C++代码中，不应该调用它来创建线程）
- ExitThread 用于终止线程（最好不要使用这种方法终止线程）
- TerminateThread 用于终止线程（最好不要使用这种方法终止线程）
- _beginthreadex 用于创建线程（C/C++运行期库函数，C/C++代码始终）
- _endthreadex 用于终止线程（应该使用这对函数来创建/终止线程）
- GetCurrentThread 用于返回线程内核对象伪句柄，可用 DuplicateHandle 将线程的伪句柄转换为线程的实句柄
- GetCurrentThreadId 用于查询线程自己的 ID
- GetThreadTimes 用于查询线程的各种时间使用情况

➢ **Code Style and Technique Tricks**

- 由于线程开销比进程少，因此始终都应该设法用增加线程来解决编程问题，而要避免创建新进程
- 一个应用程序应该用单个线程来创建所有窗口，并且有一个 GetMessage 循环。进程中的所有其它线程都是工作线程。界面线程应该比工作线程赋予更高的优先级，以便对用户的动作作出更好的响应。单个进程拥有多个用户界面线程并不多见（Windows Explorer 就是一个）

4. 线程调度、优先级和亲缘性

➢ **Knowledge Points**

- Windows 是抢占式多线程操作系统。每隔 20ms 左右，Windows 要查看当前存在的所有线程内核对象。在这些内核对象中，只有某些对象被视为可以调度的对象。Windows 选择可调度的线程内核对象中的一个，将它加载到 CPU 寄存器中，它的值是上次保存的环境中的值。这项操作称之为上下文切换

➢ **API Quick Reference**

- SuspendThread 用于暂停线程
- ResumeThread 用于唤醒暂停的线程，使之成为可调度的
- OpenThread 用于将线程 ID 转换为线程内核句柄
- Sleep 用于线程睡眠
- SwitchToThread 允许其它低优先级线程运行
- GetTickCount 返回当前时间
- QueryPerformanceFrequency 用于高分辨率的线程运行时间查询
- QueryPerformanceConter
- GetThreadContext 用于查看线程内核对象上下文（先 Suspend 线程）
- SetThreadContext 用于设置线程内核对象上下文（先 Suspend 线程）
- GetPriorityClass 用于检索进程的优先级类
- SetPriorityClass 用于设置进程的优先级类
- GetThreadPriority 用于检索线程的相对优先级
- SetThreadPriority 用于设置线程的相对优先级
- GetProcessPriorityBoost 用于确定是激活还是停用优先级提高功能

- SetProcessPriorityBoost 用于禁止或激活动态提高线程优先级的功能
- GetThreadPriorityBoost 用于确定是激活还是停用优先级提高功能
- SetThreadPriorityBoost 用于禁止或激活动态提高线程优先级的功能
- GetProcessAffinityMask 用于获取进程的亲缘性掩码
- SetProcessAffinityMask 用于设置进程的亲缘性掩码
- GetThreadAffinityMask 用于获取线程的亲缘性掩码
- SetThreadAffinityMask 用于设置线程的亲缘性掩码
- SetThreadIdealProcessor 用于为线程设置理想 CPU
- ImageLoad 以下四个函数用于操作可执行文件
- ImageUnload
- GetImageConfigInformation
- SetImageConfigInformation

Code Style and Technique Tricks

- 在实际环境中，调用 SuspendThread 时必须小心，因为不知道暂停的线程运行时它在执行什么操作。如果线程试图从堆中分配内存，那么该线程将在该堆上设置一个锁。当其他线程试图访问该堆时，这些线程的访问就被停止，直到第一个线程恢复运行。只有确切知道目标线程正在做什么，并且采用强有力的措施来避免因暂停线程的运行而带来的问题或死锁状态，SuspendThread 才是安全的
- 应该尽可能的避免使用实时优先级

5. 用户态下的线程同步

Knowledge Points

- 用户态下的线程同步有两种方式，特点是不用进入内核，速度快
  1) 原子访问：使用 InterLocked Family。缺点：只能在单值上运行，无法使线程进入等待状态
  2) 关键代码段（Critical Sections）。缺点：只能在同步同一个进程中的线程；容易进入死锁状态，因为在等待关键代码段的时候不能设置超时
- spinlock 在 X86 上工作方式是：向总线发出一个硬件信号，防止 CPU 访问同一个内存地址

API Quick Reference

- InterlockedExchangeAdd 原子递增长整型变量的值
- InterlockedExchange 原子方式用另外一个值来取代原始值
- InterlockedExchangePointer
- InterlockedCompareExchange 原子方式比较和交换值
- InterlockedCompareExchangePointer
- InitializeCriticalSection 用来初始化关键代码结构
- DeleteCriticalSection 用来删除关键代码段结构
- EnterCriticalSection 进入关键代码段
- TryEnterCriticalSection 尝试进入关键代码段
- LeaveCriticalSection 离开关键代码段
- InitializeCriticalSectionAndSpinCount（spinlock 用于关键代码段）
- SetCriticalSectionSpinCount 设置 spinlock 循环的次数

Code Style and Technique Tricks

- 始终应该将程序的数据以 cache 行大小来组织，并且将它们放在 cache 行边界对齐的位置上。这样做的目的：确保不同的 CPU 能够访问至少由高速缓存边界分开的不同内存地址。还有就是：应该将只读数据（或不常读的数据）与读写数据分开，同时应该将同一时间访问的数据组合在一起
- 使用关键代码段的一些技巧
  1) 对每个共享资源使用一个 CRITICAL_SECTION 变量
  2) 几个线程同时访问共享资源时，每个线程加锁的顺序应该完全相同
  3) 不要长时间占用关键代码段

6. 用内核对象同步线程（内核态）

➢ **Knowledge Points**

✧ 用内核对象同步线程唯一的缺点就是要进入内核，速度慢

✧ 可处于 signaled 和 nonsignaled 状态的内核对象包括：

Process	File change notification
Thread	Events
Jobs	Waitable timers
Files	Semaphore
Console input	Mutexes

✧ Event 内核对象能够通知某一操作已经完成。有两种不同类型的事件
  1) 人工重置事件。当人工重置事件得到通知时，等待该事件的所有线程均变为可调度线程
  2) 自动重置事件。当自动重置事件得到通知时，等待该事件的所有线程中只有一个线程变为可调度线程

✧ Waitable Timer 内核对象是在某个时间或按规定的时间间隔发出信号通知自己的内核对象。

✧ Waitable Timer 和用户定时器（SetTimer 函数设置）的区别：
  1) 用户定时器需要在应用程序中设置许多附加的用户界面结构，这使得用户定时器变得资源更加密集；等待定时器属于内核对象，这意味着它能够供多个线程共享，并且是安全的
  2) 用户定时器能够生成 WM_TIMER 消息，这些消息将返回给调用 SetTimer 的线程（用于回调定时器时）或返回给创建窗口的线程（用于基于窗口的定时器时）。因此当用户定时器报时的时候，只有一个线程会得到通知。另一方面，多个线程可以在等待定时器上等待，如果定时器是个人工重置定时器，那么它可以调度若干个线程
  3) 如果要执行与界面相关的事件，以便对定时器作出响应，那么使用用户定时器来组织代码更加容易一些。最后，运用等待定时器，当到了规定时间的时候，更有可能得到通知。因为 WM_TIMER 消息始终属于最低优先级的消息，当消息队列中没有其它消息时才处理该消息

✧ Semaphore 内核对象用于对资源进行计数，当前资源的数量绝不会为负值

✧ Mutex 内核对象能够确保线程拥有对单个资源的互斥访问

➢ **API Quick Reference**

✧ WaitForSingleObject 用于等待某个内核对象直到其变为 signaled 状态为止

✧ WaitForMultipleObjects 等待若干个内核对象

✧ CreateEvent 用于创建事件内核对象

- ✧ OpenEvent 用于引用已经创建的内核对象
- ✧ SetEvent 用于设置事件内核对象为 signaled 状态
- ✧ ResetEvent 用于设置事件内核对象为 nonsignaled 状态
- ✧ PulseEvent 用于使事件变为 signaled 状态，然后又变为 nonsignaled 状态
- ✧ CreateWaitableTimer 用于创建等待定时器
- ✧ OpenWaitableTimer 用于引用已经创建的定时器内核对象
- ✧ SetWaitableTimer 用于告诉定时器在何时让它变为 signal 状态
- ✧ CancelWaitableTimer 用于取消定时器
- ✧ QueueUserAPC 请参考 MSDN，这个函数很重要，涉及的内容很多
- ✧ CreateSemaphore 用于创建信号量内核对象
- ✧ OpenSemaphore 用于引用已经创建了的信号量内核对象
- ✧ ReleaseSemaphore 用于对信号量当前的资源数量进行递增
- ✧ WaitForInputIdle//the functions following，please refer to MSDN
- ✧ MsgWaitForMultipleObjects(Ex)
- ✧ WaitForDebugEvent
- ✧ SignalObjectAndWait
- ➢ **Code Style and Technique Tricks**

7. 线程同步工具箱

- ➢ **Knowledge Points**
- ➢ **API Quick Reference**
- ➢ **Code Style and Technique Tricks**

8. 线程池

- ➢ **Knowledge Points**
- ✧ Windows 的线程池包括四个独立的组件
  1) Timer
  2) Wait
  3) I/O
  4) Non-I/O
- ✧ Windows 的线程池能够执行下面的操作
  1) 异步调用函数（I/O 组件）
  2) 按照规定的时间间隔调用函数（Timer 组件）
  3) 在单个内核对象变为 signaled 状态时调用函数（Wait、Non-I/O 组件）
  4) 当异步 I/O 请求完成时调用函数（I/O、Non-I/O 组件）
- ✧ 异步调用函数。它允许单个线程处理来自不同客户机的请求。该线程不必顺序处理这些请求，也不必在等待 I/O 请求运行结束时暂停运行，所以可以说异步 I/O 是创建高性能可伸缩的应用程序的秘诀。
- ✧ 按照规定的时间间隔调用函数。它允许程序员不必为应用程序执行的每个基于时间的操作任务创建一个等待定时器对象，这可以节省系统资源。它维护一个定时器队列（这是个轻便的资源），并且共享 Timer 组件的线程和等待定时器内核对象。
- ✧ 在单个内核对象变为 signaled 状态时调用函数。它允许不必让若干个线程等待一个对象（节省资源，多个线程也会占用比较多的资源），而是让 Wait

组件中的一个线程来等待该内核对象，然后对等待该对象的工作项目排队。

- 当异步 I/O 请求完成时调用函数。它能够让服务器应用程序发出某些异步 I/O 请求，当这些请求完成时，让一个准备好的线程池来处理已完成的 I/O 请求。

API Quick Reference

- QueueUserWorkItem 用于将一个工作项目排队放入线程池中的一个线程
- PostQueuedCompletionStatus 用于将工作项目信息传递给 I/O 完成端口
- GetQueuedCompletionStatus 用于在 I/O 完成端口上等待的线程取出信息
- CreateTimerQueue 用于创建一个定时器队列
- DeleteTimerQueueEx 用于删除定时器队列
- CreateTimerQueueTimer 用于在定时器队列中加入定时器
- DeleteTimerQueueTimer 用于删除定时器队列中的定时器
- ChangeTimerQueueTimer 用于改变定时器到期时间和到期周期
- RegisterWaitForSingleObject 用于注册一个在内核对象变为 signaled 状态时要执行的函数
- UnregisterWaitEx 用于取消 Wait 组件的注册状态
- BindIoCompletionCallBack 用于将一个设备与该组件关联起来

Code Style and Technique Tricks

- 线程池的开销并不小，必须认真考虑线程池能够做什么和不能做什么，切忌盲目使用线程池
- 当使用线程池函数时，应该查找潜在的死锁条件。有两点值得注意：
  1) 如果工作项目函数在关键代码段、信号量和互斥对象上阻塞，那么必须十分小心，因为这更有可能导致死锁现象。始终应该了解哪个组件（I/O、Non-I/O、Wait、Timer）的线程正在在运行你的代码。
  2) 如果工作项目函数位于可能被动态卸载的 DLL 中，也要小心。调用已卸载的 DLL 中的函数的线程将会产生违规访问。若要确保不卸载带有已经排队的工作项目的 DLL，必须对已经排队的工作项目进行引用计数，在调用 QueueUserWorkItem 函数之前递增引用计数器的值，当工作项目函数完成运行时则递减该引用计数器的值。只有当引用计数降为 0 时，才能安全地卸载 DLL。
- 使用 Timer 组件要注意的问题：正在使用 Timer 组件的线程，不应该试图对任何定时器进行中断删除，否则就会发生死锁。因为，如果试图删除一个定时器，就会将一个 APC 通知放入该定时器组件的线程队列中。如果该线程正在等待一个定时器删除，而它又不能删除该定时器，那么就会发生死锁。
- 当使用 Wait 组件时要注意下面的问题：
  1) 当将 INVALID_HANDLE_VALUE 传递给 UnregisterWaitEx 时，必须小心避免死锁状态。在试图取消注册引起工作项目执行的 Wait 组件时，该工作组件不能阻塞自己。这好像说：暂停我的运行，直到我完成运行为止——这将导致死锁
  2) 在取消 Wait 组件的注册状态之前，不要关闭内核对象的句柄。这会导致句柄无效。因为 Wait 组件在内部调用 WaitForMultipleObjects 函数，传递一个无效句柄给它会导致它的调用失败，整个 Wait 组件将无法正

常工作

3) 不应该调用 PulseEvent 函数来通知注册的事件对象。如果这样做了，等待组件的线程可能忙于执行某些别的操作，从而错过了时间的触发。

- 使用“当异步 I/O 请求完成时调用函数时”要注意：关闭设备会导致它的所有待处理的 I/O 请求立即完成，并产生一个错误代码。要做好准备，在回调函数中处理这种情况。如果想确认关闭设备后没有任何回调函数在执行，那么必须在应用程序中进行引用计数。也就是说，在每次发出 I/O 请求时，必须将引用计数器递增，每次完成 I/O 请求时，则递减引用计数器。

9. 纤程

➢ **Knowledge Points**

✧ 实现线程的是 Windows 内核。操作系统知道线程的情况，并且根据 Microsoft 定义的算法对线程进行调度。纤程是以用户态的代码来运行的，内核并不知道纤程，并且它们根据用户定义的算法来调度，所以纤程采用非抢占式调度方式。纤程在线程上运行，一个线程上可以有多个纤程，但同一时刻只有一个纤程在运行

➢ **API Quick Reference**

✧ ConvertThreadToFiber 用于将一个线程转换成纤程
✧ CreateFiber 用于创建一个纤程
✧ SwitchToFiber 用于切换到别的纤程执行
✧ DeleteFiber 用于删除纤程
✧ GetCurrentFiber 用于得到当前运行的纤程的执行环境的地址
✧ GetFiberData 用于得到当前执行纤程的执行环境

➢ **Code Style and Technique Tricks**

- 当能够用线程很好解决应用程序设计时，应该避免使用纤程

(三) 第三部分——内存管理

1. Windows 内存体系结构

➢ **Knowledge Points**

✧ NULL 指针分配区。这个分区的设置是为了帮助程序员捕捉对 NULL 地址赋值的情况。若果进程中的线程对该地址空间进行读/写操作时，CPU 会引发一个访问违规异常

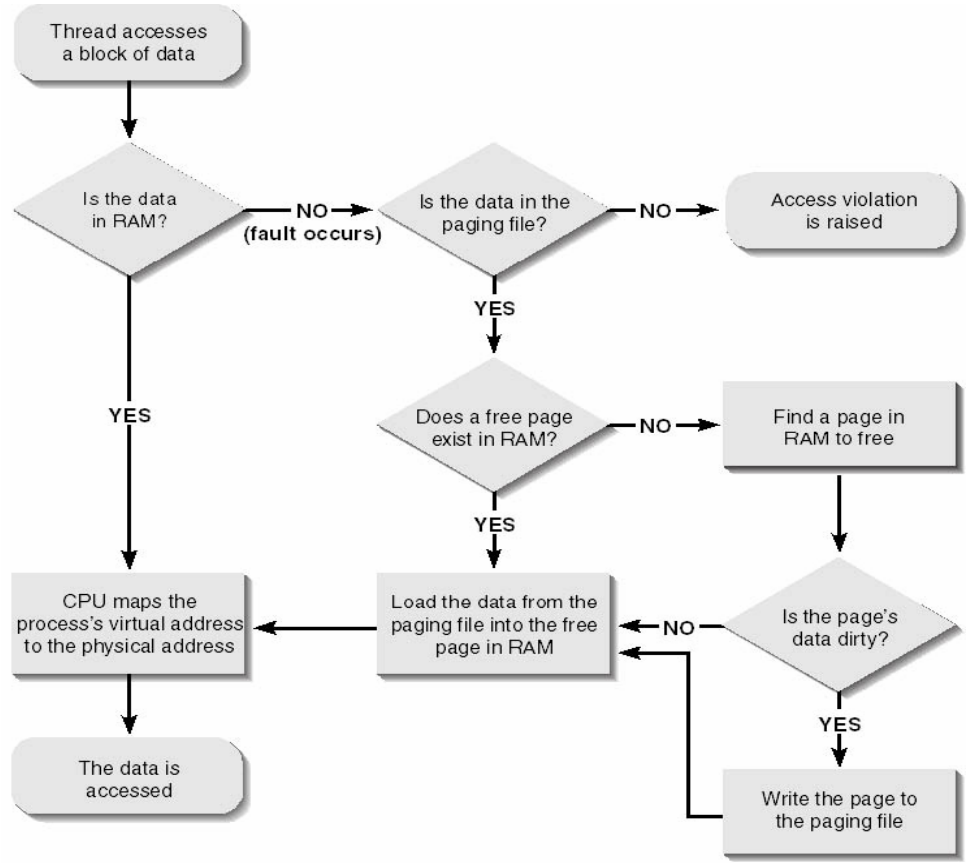
✧ 用户态分区。这个分区是进程的私有地址空间。一个进程不能读取、写入或者以任何方式访问另外一个进程驻留在该分区中的数据。该分区是应用程序维护进程的大部分数据的地方。

1) 在 Windows2000 中，所有的 exe 和 DLL 模块均加载到用户态分区。每个进程可以将这些 DLL 加载到用户态分区的不同地址中（不过可能性很小）。系统也是在用户态分区映射该进程可访问的所有内存映射文件

2) 在 Windows98 中，主要的 Win32 系统 DLL（kernal32.dll，AdvAPI32.dll，User32.dll 和 GDI32.dll）均加载到共享内存映射文件分区中。exe 和其它的 DLL 模块则加载到用户态分区中。共享的 DLL 都映射到所有

进程的相同虚拟地址中，但是其它的 DLL 可以加载到用户态分区的不同地址中（不过可能性很小）。另外，在 Windows98 中，用户态分区中绝不会出现内存映射文件（它有一个专门的分区来处理内存映射文件）

✧ Windows 下物理存储器有四类：RAM、Paging File 页文件、exe 和 DLL 映像文件、Data File 数据文件。应用程序可以将 RAM 和 Paging File 的大小之和看作系统的总物理内存大小。在操作系统和 CPU 的协调下 RAM 中的页和 Paging File 中的页可以来回交换。Windows 下虚拟地址转换成物理地址的流程图如下：



✧ Copy-On-Write：操作系统给共享的内存块赋予了 Copy-On-Write 保护属性。当一个.exe 或 DLL 模块被映射到内存地址时，系统计算有多少个页面可以写入，然后从页文件中分配内存，以容纳这些可写入页面的需要。除非该模块的可写入页面发生了实际的写入，否则这些页文件是不用的。当一个系统试图将数据写入一个共享页面时，系统就会进行干预并执行下列步骤：

1) 系统查找 RAM 中的一个空闲内存页面
2) 系统将试图被修改的页面内容拷贝到第一步中找到的页面。该空闲页面将被赋予 PAGE_READWRITE 或 PAGE_EXECUTE_READWRITE 保护属性。原始页面的保护属性和数据不发生任何变化
3) 然后系统更新进程的页表，使得被访问的虚拟地址被转换成新的 RAM 页面的地址

✧ 32 位的进程的虚拟地址空间为 0x0000,0000-0xFFFF,FFFF（4G）。虚拟地址空间的分布如下：

分区	32 位 Windows 2000（X86 和 Alpha 处理器）	32 位 Windows 2000（X86 和 Alpha 处理器 w/3GB 用户方式）	Windows98
NULL 指针分区	0x0000,0000 0x0000,FFFF (64KB)	0x0000,0000 0x0000,FFFF (64KB)	0x0000,0000 0x0000,0FFF (4KB)
DOS/16 位 Windows 程序兼容分区	NA	NA	0x0000,1000 0x003F,FFFF (4MB-4KB)
用户态方式分区	0x0001,0000 0x7FFE,FFFF (2GB-64*2KB)	0x0001,0000 0xBFFE,FFFF (3GB-64*2KB)	0x0040,0000 0x7FFF,FFFF (2GB-4MB)
64KB 禁止进入分区	0x7FFF,0000 0x7FFF,FFFF (64KB)	0xBFFF,0000 0xBFFF,FFFF (64KB)	NA
共享内存映射文件分区	NA	NA	0x8000,0000 0xBFFF,FFFF (1GB)
内核态分区	0x8000,0000 0xFFFF,FFFF (2GB)	0xC000,0000 0xFFFF,FFFF (1GB)	0xC000,0000 0xFFFF,FFFF (1GB)

➢ **API Quick Reference**

➢ **Code Style and Technique Tricks**

2. 虚拟内存

➢ **Knowledge Points**

➢ **API Quick Reference**

✧ GetSystemInfo 用于检索和主机相关的各种信息

✧ GlobalMemoryStatus 用于检索当前的内存状态信息

✧ VirtualQuery(Ex)用于查询特定地址空间的某些信息

➢ **Code Style and Technique Tricks**

3. 在程序中使用虚拟内存

➢ **Knowledge Points**

✧ Windows 提供了三种方法来管理内存

1) 虚拟内存，最适合用来管理大型对象或结构数组
2) 内存映射文件，最适合用来管理大型数据流（通常来自文件）以及单个计算机上运行的多个进程之间共享数据
3) 内存堆，最适合用来管理大量小型对象

➢ **API Quick Reference**

✧ VirtualAlloc 用于在进程地址空间保留一个区域或提交物理内存

✧ VirtualFree 用于释放地址空间区域

- VirtualProtect 用于改变页面的保护属性
- AllocateUserPhysicalPages 用于 RAM 物理内存的分配
- FreeUserPhysicalPages 用于释放 RAM 块
- MapUserPhysicalPages 用于将 RAM 块赋予给地址窗口

➢ **Code Style and Technique Tricks**

虚拟内存的提交常常和 SEH 配合使用，这是提高内存使用效率和程序效率的绝佳途径

4. 线程的堆栈

➢ **Knowledge Points**

➢ **API Quick Reference**

➢ **Code Style and Technique Tricks**

按照 Windows 默认设置，系统保留 1MB 的地址空间并提交两个页面的内存。可以给链接程序加上/STACK:reserve[,commit]来改变这个默认设置

5. 内存映射文件

➢ **Knowledge Points**

- 与虚拟内存一样，内存映射文件可以用来预留一个地址空间的区域，并将物理内存提交给该区域。它们之间的差别是：内存映射文件的物理内存来自一个已经位于磁盘上的文件，而不是系统的页文件。一旦文件被映射，就可以访问它，就像整个文件已经加载到内存一样。内存映射文件可以用于 3 个不同的目的：
  1) 系统使用内存映射文件，以便加载和执行.exe 和 DLL 文件。这可以大大节省页文件空间和应用程序启动所需时间
  2) 可以使用内存映射文件来访问磁盘上的数据文件。这使得不需要对文件执行 I/O 操作，并且可以不必对文件进行缓存
  3) 可以使用内存映射文件，使同一台计算机上运行的多个进程相互共享数据
- 当线程调用 CreateProcess 是，系统将执行系列操作
  1) 系统指出在调用 CreateProcess 时设定的.exe 文件
  2) 系统创建一个新进程内核对象
  3) 系统为这个新进程创建一个私有地址空间
  4) 系统保留一个足够大的地址空间区域，用于存放.exe 文件。该区域的位置在.exe 文件中有设定。默认设置.exe 文件映射的基址是 0x0040,0000。可以给链接程序用/BASE 选项改写该基址
  5) 系统注意到支持保留区域的物理存储器是磁盘上的.exe 文件，而不是系统的页文件
  6) 如果.exe 文件里面引用了别的 DLL 文件，那么系统要为每个 DLL 文件递归调用 LoadLibrary 函数，来加载所有的 DLL 文件，执行的步骤如下
  7) 系统保留一个足够大的地址空间区域，用于存放 DLL 文件。该区域的位置在 DLL 文件中有设定。默认设置为 0x1000,0000，可以给链接程序用/BASE 选项改写该基址。Windows 提供的所有标准系统 DLL 都拥有不同的基地址。这样加载到单地址空间，它们就不会重叠

8) 如果系统无法在该 DLL 首选基地址上保留一个区域，其原因可能是该区域已经被另一个 DLL 或.exe 占用，也可能是该区域不够大，此时系统将设法寻找另外一个地址空间的区域来保留该 DLL。如果一个 DLL 不能加载到它的首选基址，这将是非常不利的，原因有二：首先，如果 DLL 没有重定位信息，那么系统就无法加载该 DLL；第二系统必须在 DLL 中执行某些重定位操作，这往往需要更多的页文件支持，也增加了 DLL 加载的时间

9) 系统会注意到支持已保留区域的物理存储器位于磁盘上的 DLL 文件，而不是在系统的页文件中。如果 DLL 无法加载到它的首选基址，系统必须执行重定位操作。那么，系统也将知道 DLL 的某些物理存储器已经被映射到页文件中

- 系统允许应用程序映射一个文件的相同数据的多个视图，只要是映射相同的文件映射内核对象，系统就可以保证映射视图数据的相关性，也即一个视图中的数据改变，另外的视图也会更新以放映这个变化。多个进程映射单个文件映射内核对象的多个视图，数据仍然是相关的。Windows 允许创建若干个由单数据文件支持的文件映射内核对象，但不能保证这些不同的文件映射内核对象的视图的相关性。它只能保证单个文件映射内核对象的的多个视图之间的相关性。
- 在 Win2000 中，第一个进程调用 MapViewOfFile 函数返回的内存地址，很可能不同于第二个进程调用 MapViewOfFile 返回的内存地址，即使这两个进程映射了相同文件映射内核对象。在 Win98 中如果是对相同文件映射内核对象进行映射，那么返回的地址是相同的。
- Windows 中，在单个计算机上共享数据的最底层机制是内存映射文件。它是单机上内存共享数据最佳最有效的途径。数据共享的方法是通过多个进程映射同一个文件映射内核对象的视图来实现的，它们共享物理存储器的同一个页面。
- 内存映射文件除了可以受磁盘数据文件支持外，还可以用受页文件支持，要创建受页文件支持的内存映射文件，不需要调用 CreateFile 函数，只要调用 CreateFileMapping 时传递 INVALID_HANDLE_VALUE 作为 hFile 参数即可。

➢ **API Quick Reference**

- CreateFile 用于创建或打开一个文件内核对象
- CreateFileMapping 用来创建一个文件映射内核对象
- MapViewOfFile(Ex)用于将文件数据映射到进程的地址空间
- UnmapViewOfFile 用于从进程地址空间中撤销文件数据的映像
- FlushViewOfFile 用于强制系统将修改过数据重新写入磁盘映像中
- GetFileSize 用于得到文件的大小
- SetFilePointer 用于将文件指针定位到指定的位置
- SetEndOfFile 用于设置文件结尾
- CopyFile 用于复制文件

➢ **Code Style and Technique Tricks**

- 要实现在可执行文件或 DLL 的多个实例之间共享数据，可以用#pragma data_seg 来创建自己的节，并且配合/SECTION 的链接开关，然后要把要共享的变量放到自己的节中，那么该.exe 或 DLL 的多个实例就可以共享

这些数据了，系统并不为.exe 或 DLL 文件的每个映像都创建这些变量。Microsoft 并不鼓励使用共享节来在多个实例之间共享数据。

6. 堆

➢ **Knowledge Points**

✧ 从内部来讲，堆是保留的地址空间的一个区域。堆的优点是可以不考虑分配粒度和页面边界之类的问题，可以集中精力处理手头任务；堆的缺点是分配和释放堆的内存块的速度比其它机制要慢，并且无法直接控制物理存储器的提交和回收。

✧ 当进程初始化时，系统在进程地址空间创建一个默认堆，大小为 1MB。可以用/HEAP 链接开关来改变这个默认大小。进程的默认堆可供许多 Windows 函数使用，因此对进程默认堆的访问时顺序进行的。系统必须保证在规定时间内，每次只有一个线程能够分配和释放堆中的内存块。所以如果想要应用程序更快的访问堆，那么应该创建自己的堆。

➢ **API Quick Reference**

✧ GetProcessHeap 用于得到进程的默认堆
✧ HeapCreate 用于创建辅助堆
✧ HeapAlloc 用于从堆中分配内存
✧ HeapReAlloc 用于改变内存块的大小
✧ HeapSize 用于检索内存块的大小
✧ HeapFree 用于释放内存块
✧ HeapDestroy 用于撤销堆
✧ GetProcessHeaps 用于获取现有堆的句柄
✧ HeapValidate 用于验证堆的完整性
✧ HeapCompact 用于合并地址中的空闲内存块
✧ HeapLock
✧ HeapUnlock
✧ HeapWalk 用于遍历堆
✧ Heap32First//refer to MSDN，ToolHelp 函数
✧ Heap32Next
✧ Heap32ListFirst
✧ Heap32ListNext

➢ **Code Style and Technique Tricks**

(四)　第四部分——动态链接库

1. 动态链接库基础知识

➢ **Knowledge Points**

✧ 使用动态链接库有以下好处：

1) 它们扩展了应用程序的特性
2) 它们可以用多种语言来编写
3) 它们简化了软件项目的管理
4) 它们有助于节省内存
5) 它们有助于资源共享

6) 它们有助于应用程序本地化
7) 它们有助于解决平台差异
8) 它们可以用于一些特殊目的（Hook 等）

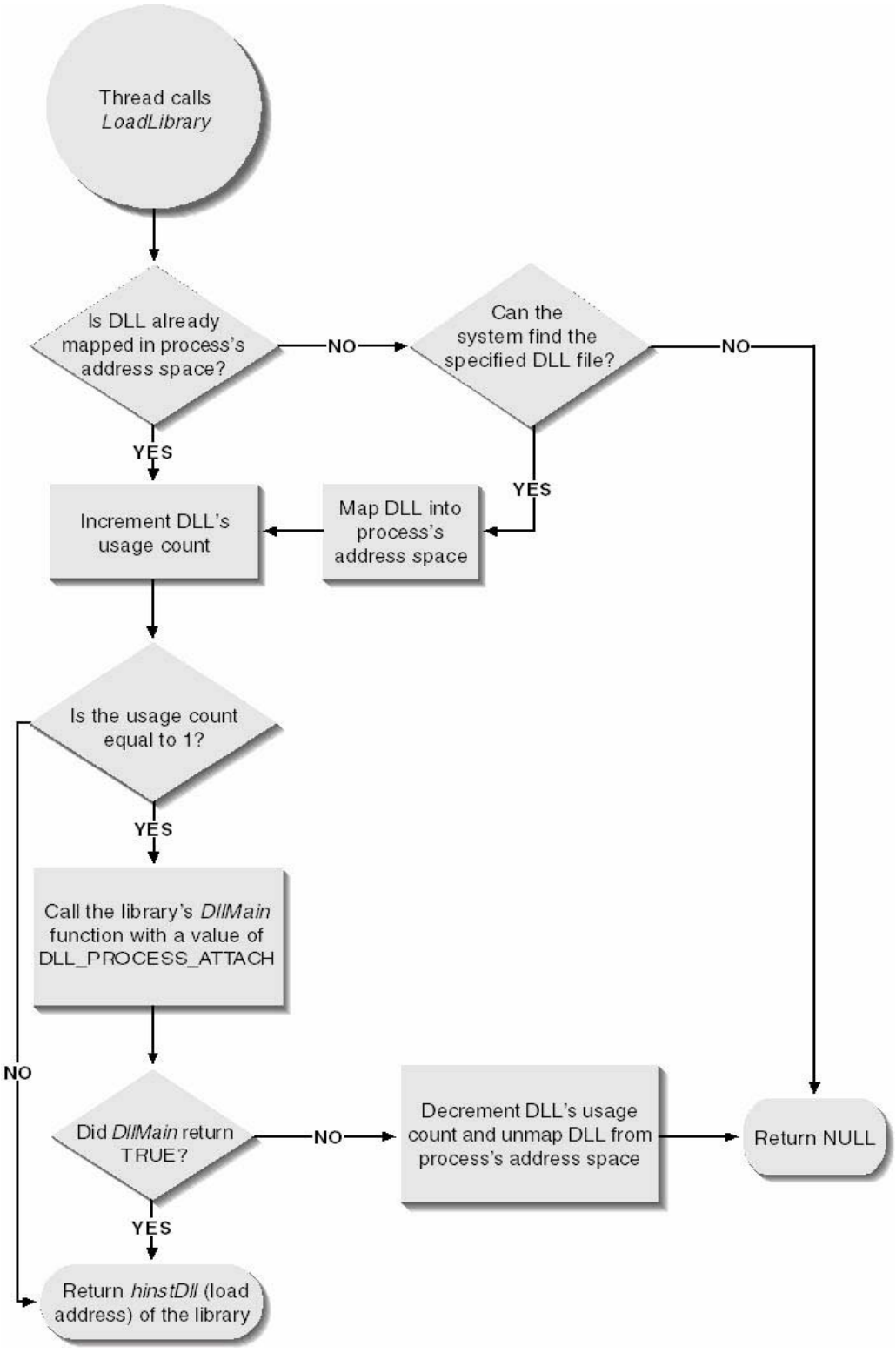
- ✧ 在 DLL 中通常没有用来处理消息循环或创建窗口的支持代码。DLL 中函数的代码创建的任何对象均由调用线程所拥有，而 DLL 本身从来不拥有任何东西
- ✧ 可执行文件的全局变量和静态变量不能被同一个可执行文件的多个实例共享。Windows98 能够确保这一点，方法是在可执行文件被映射到进程的地址空间时为可执行文件的全局变量和静态变量分配相应的存储器。Windows2000 确保这一点的方法是使用 copy-on-write 机制

- ➢ **API Quick Reference**
- ➢ **Code Style and Technique Tricks**

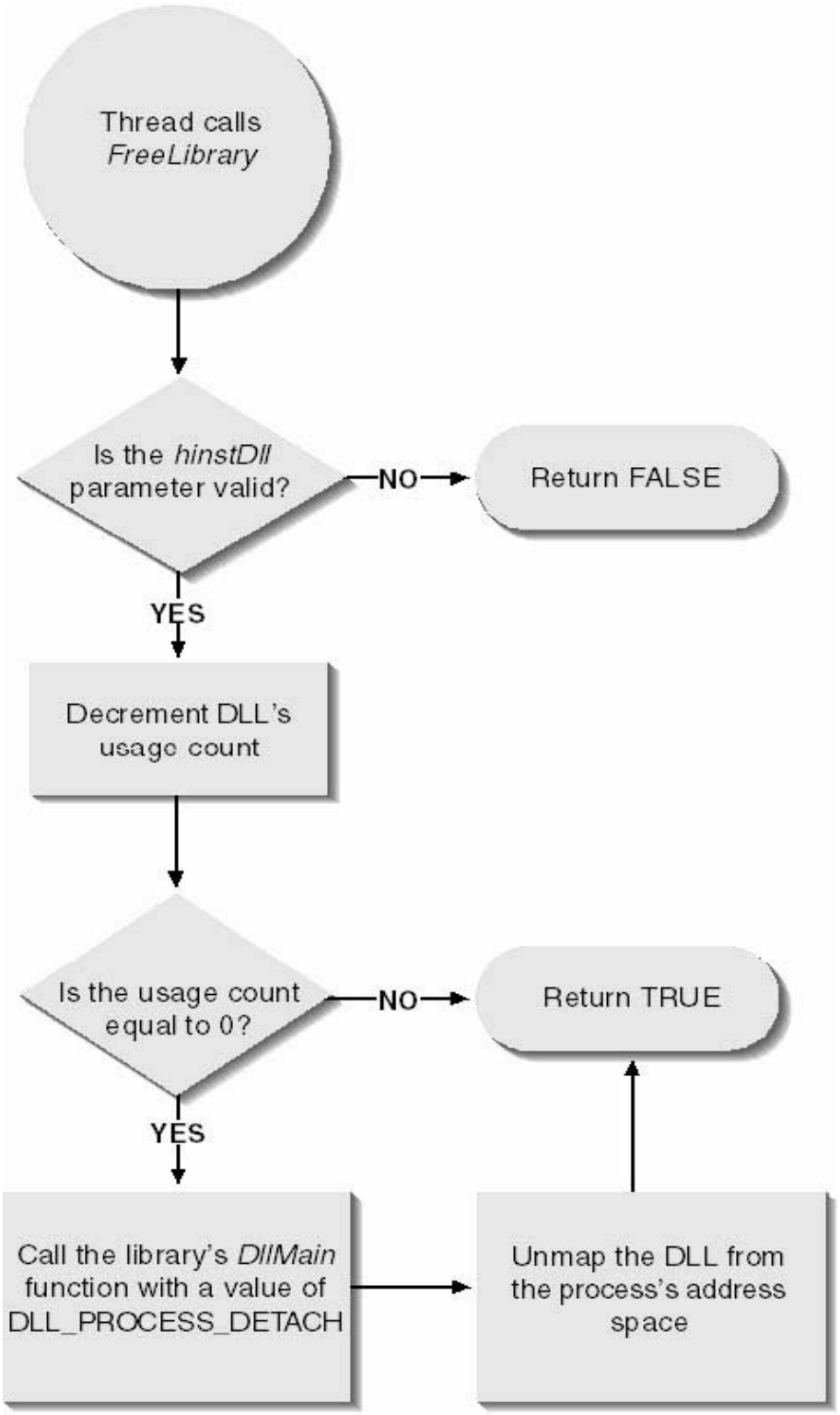
2. 动态链接库高级技巧

- ➢ **Knowledge Points**

- ✧ 系统是顺序调用 DLL 中 DllMain 函数的（通过进程内部创建的互斥对象来实现）。意思是：假设有一个进程，它拥有两个线程 A 和 B；并且该进程还有一个 DLL 为 SomeDll.dll，它被映射到进程地址空间中。线程 A 和 B 都准备调用 CreateTread 来创建两个线程 C 和 D。当线程 A 创建线程 C 的时，系统调用带有 DLL_THREAD_ATTACH 值的 SomeDll.dll 的 DllMain 函数。当线程 C 执行 DllMain 函数中的代码时，线程 B 调用创建线程 D。这时系统必须再次调用带有 DLL_THREAD_ATTACH 值的 SomeDll.dll 的 DllMain 函数，这次让线程 D 执行代码。但是由于系统是顺序执行 DllMain 函数的，因此系统会暂停线程 D 的运行直到线程 C 完成了对 DllMain 函数的代码处理并且返回为止。由于这个原因在 DllMain 函数中调用 WaitForSingleObject 等待函数时要小心，很可能导致死锁的发生。
- ✧ 当 DLL 收到 DLL_PROCESS_ATTACH 通知时，进入 DllMain 之前，系统要负责对 DLL 中定义的全局的和静态的 C++对象初始化。当 DLL 收到 DLL_PROCESS_DETACH 通知时，在 DllMain 函数返回后，系统要负责对 DLL 中定义的全局的和静态的 C++对象进行析构操作
- ✧ 线程调用 LoadLibrary 时系统的执行步骤如下如所示：



✧ 线程调用 FreeLibrary 时系统的执行步骤如下如所示：



- **API Quick Reference**
- LoadLibrary(Ex)用于加载 DLL 模块
- FreeLibrary 用于卸载 DLL 模块
- FreeLibraryAndExitThread 用于卸载 DLL 模块并终止线程
- GetProcAddress 用于获取函数的地址
- DisableThreadLibraryCalls 用于禁止将 DLL_THREAD_ATTACH 和 DLL_THREAD_DETTACH 的通知发送给 DllMain 函数

➢ **Code Style and Technique Tricks**

- 在一个 DLL 中分配的内存，不要在另外的 DLL 或 exe 中删除该内存。当一个模块提供一个用于分配内存块的函数，该模块也必须提供释放内存的函数
- 在 DLL 中应避免输出变量，因为这会删除代码的一个抽象层，使得对 DLL 的维护更加困难。还有应该避免输出 C++类，除非可执行模块和 DLL 模块的开发是使用同一工具
- 在 DLL 的进入点函数 DllMain 中，应该避免调用从其它 DLL 中输入的函数，因为可能其它的 DLL 还没有初始化；还因该避免从 DllMain 内部调用 LoadLibrary(Ex)函数，因为这些函数可能会形成一个依赖性循环；还应避免在 DllMain 中调用 User、Shell、ODBC、COM、RPC 和 Socket 等函数，因为它们的 DLL 有可能还没有初始化，或者这些函数内部调用 LoadLibrary(Ex)函数。
- VS 的 Rebase.exe 和 Bind.exe 工具的使用

3. 线程本地存储器

➢ **Knowledge Points**

✧ 线程本地存储器将数据与执行的特定线程联系起来，可以解决多线程问题和解决应用程序对全局变量和静态变量的依赖性

➢ **API Quick Reference**

✧ SetWindowsWord 用于将数据与特定的窗口关联起来
✧ SetWindowLong 同上
✧ TlsAlloc 用于动态 TLS，分配 TLS 数组 Slot
✧ TlsFree 用于释放 TLS 数组 Slot
✧ TlsSetValue 用于将值放入线程的 TLS 数组
✧ TlsGetValue 用于从线程的 TLS 数组检索值

➢ **Code Style and Technique Tricks**

4. DLL Injection 和 API Hooking

➢ **Knowledge Points**

✧ DLL 的插入是指一个进程将一个 DLL 模块插入到另外一个进程的地址空间中，这可以使另一个进程来执行插入的 DLL 模块代码。DLL 插入的方法有下面几种：
✧ 使用注册表来插入。优点是方法简单，只要将一个值添加到一个已经存在的注册表关键字中。不足之处有：1）修改注册表之后要重启机子；2）要插入的 DLL 只能映射到使用 User32.dll 的进程中；4）DLL 被映射到每个基于 GUI 的应用进程中；
✧ 使用 Windows Hook 来插入
✧ 使用远程线程来插入
✧ 使用 Trojan DLL 来插入
✧ 将 DLL 作为调试程序来插入
✧ 用 CreateProcess 来插入

➢ **API Quick Reference**

✧ SetWindowsHookEx 用于安装挂钩

- UnhookWindowsHookEx 用于删除挂钩
- CreateRemoteThread 用于在另外一个进程中创建线程
- VirtualAllocEx 用于一个进程分配另外一个进程地址空间中的内存
- VirtualFreeEx 用于释放内存
- ReadProcessMemory 用于一个进程读取另外一个进程地址空间的数据
- WriteProcessMemory 用于一个进程将数据写入另外一个进程地址空间

➢ **Code Style and Technique Tricks**

- 可以跨越进程边界发送窗口消息，以便与内置控件（如 button，edit，static，combo box, list box 等等）进行交互（Windows 内部使用 MMF 来实现），但是对于新的常用控件不能这样做。

(五) 第五部分——结构化异常处理

1. 结束处理器

➢ **Knowledge Points**

- 结构化异常处理（SEH）主要包含两个功能
  1) 结束处理（termination handling）
  2) 异常处理（exception handling）

  它带来的好处是：
  1) 简化了错误处理，因为所有的清理工作都在一个位置并且保证被执行
  2) 提高了程序的可读性
  3) 使代码更容易维护
  4) 如果使用得当，具有最小的系统开销
- 结束处理器（a termination handler）能够保证去调用和执行一个代码（the termination handler）块而不管另外一段代码（guard body）是如何退出的。也就是说__try{}__finally{}结构中，不管__try 块是如何退出的，__finally 块始终会执行。但又几种特殊情况：当 try 块调用 ExitThread、ExitProcess、TerminateThread、TerminateProcess 函数时，finally 块不会被执行。

➢ **API Quick Reference**

➢ **Code Style and Technique Tricks**

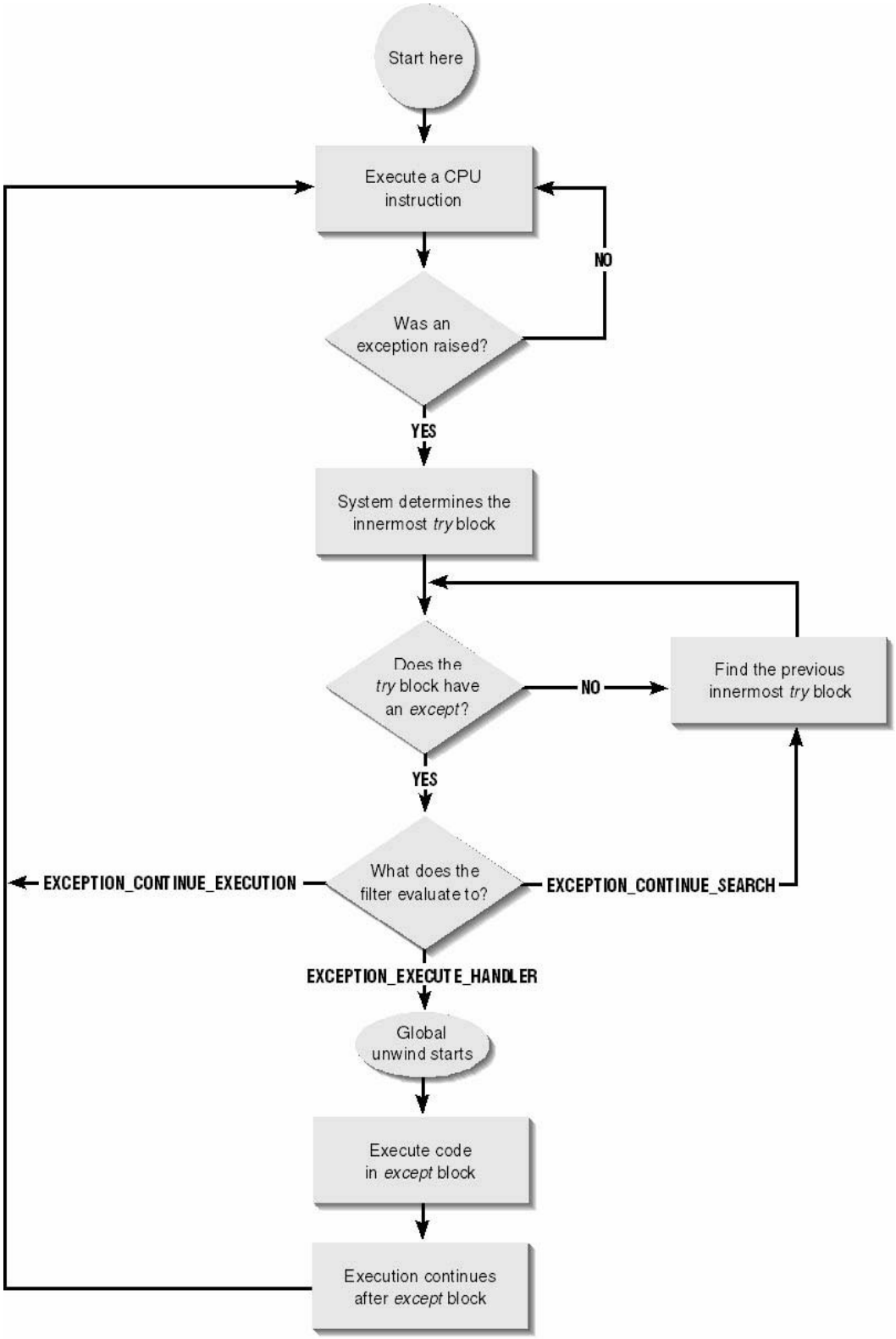
- 结束处理器的 try 块和 finally 块中最好不要出现 return、continue、break、goto、longjump 等语句，因为这些语句会使 try 块过早的退出，这样会使 finally 局部展开，生成更多的代码，这样会影响程序效率。

2. 异常处理器和软件异常

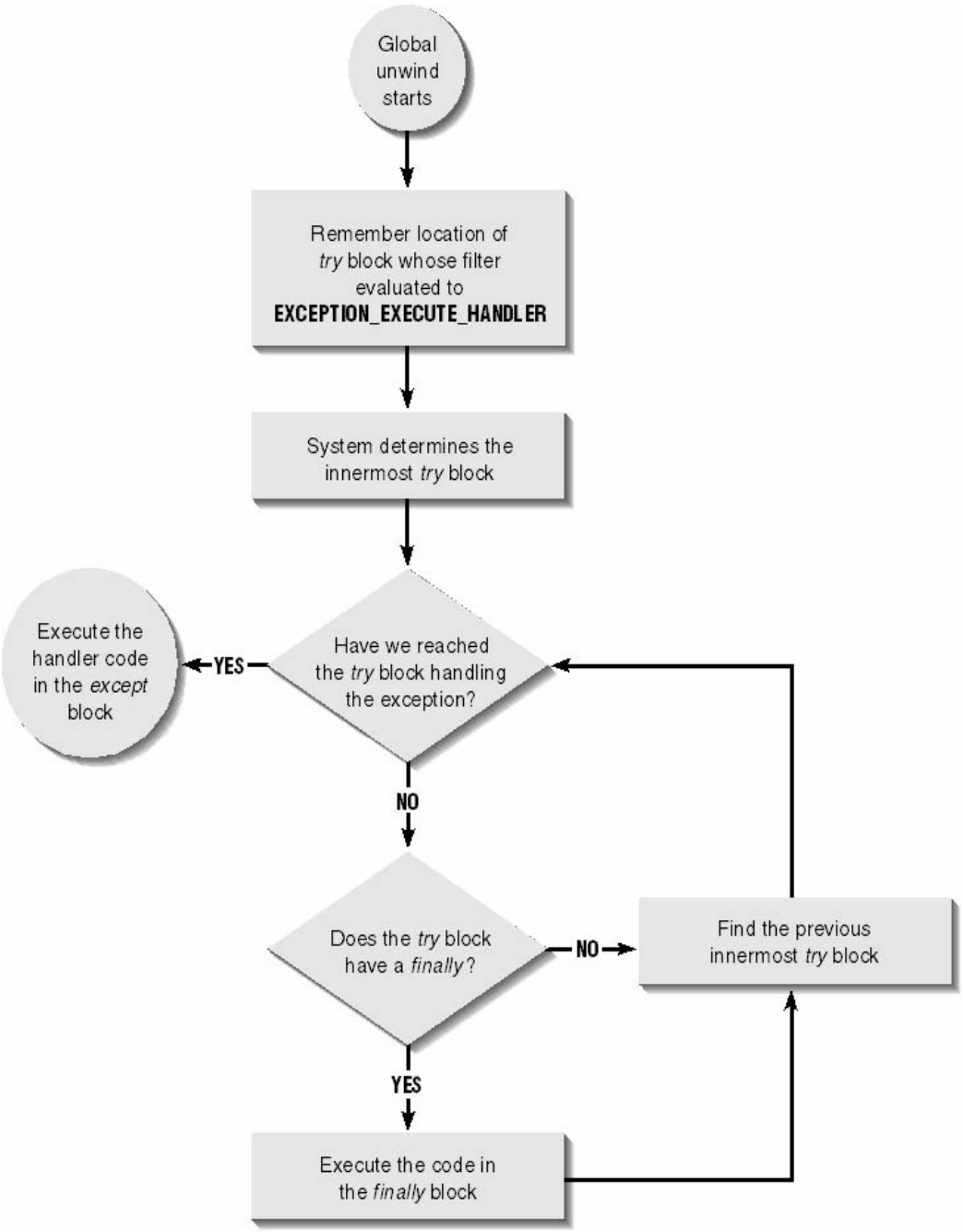
➢ **Knowledge Points**

- 由 CPU 引发的异常称之为硬件异常，由操作系统或应用程序引发的异常称之为软件异常
- __try 块必须和__finally 块或__except 块联用，但一次只能和其中之一联用
- 尽管在结束处理器（termination handler）的 try 块中使用 return、goto、continue 和 break 语句是遭强烈反对的，但在异常处理器（exception handler）的 try 块中使用这些语句是不会产生速度和代码规模方面的不良影响。这些语句出现在与__except 块相结合的 try 块中是不会引起局部展开的系统开销

✧ 系统处理异常的步骤如下图所示：



✧ 当异常过滤器的值为 EXCEPTION_EXCUTE_HANDLER 时，系统必须执行一个全局展开。全局展开的流程图如下：



- **API Quick Reference**
  - GetExceptionCode 用于获取异常代码
  - GetExceptionInformation 用于获取异常的信息
  - RaiseException 用于引发一个软件异常
- **Code Style and Technique Tricks**

传统上一个失败的函数返回一些特殊值来指出失败。函数的调用者应该检查这些特殊值并采取相应动作。通常这个调用者要清除所做的事情并将它自己的失败代码返回给它的调用者。这种错误代码的逐层传递会给源程序的代码变得非常难以编写和维护。另外一种方法是让函数在失败时引发异常。用这种方法，代码更容易编写和维护，而且也执行的更好，因为通常不需要执行那些错误测试代码。实际上，仅当发生失败时也就是异常发生

时才执行错误测试代码

3. 未处理异常和 C++异常

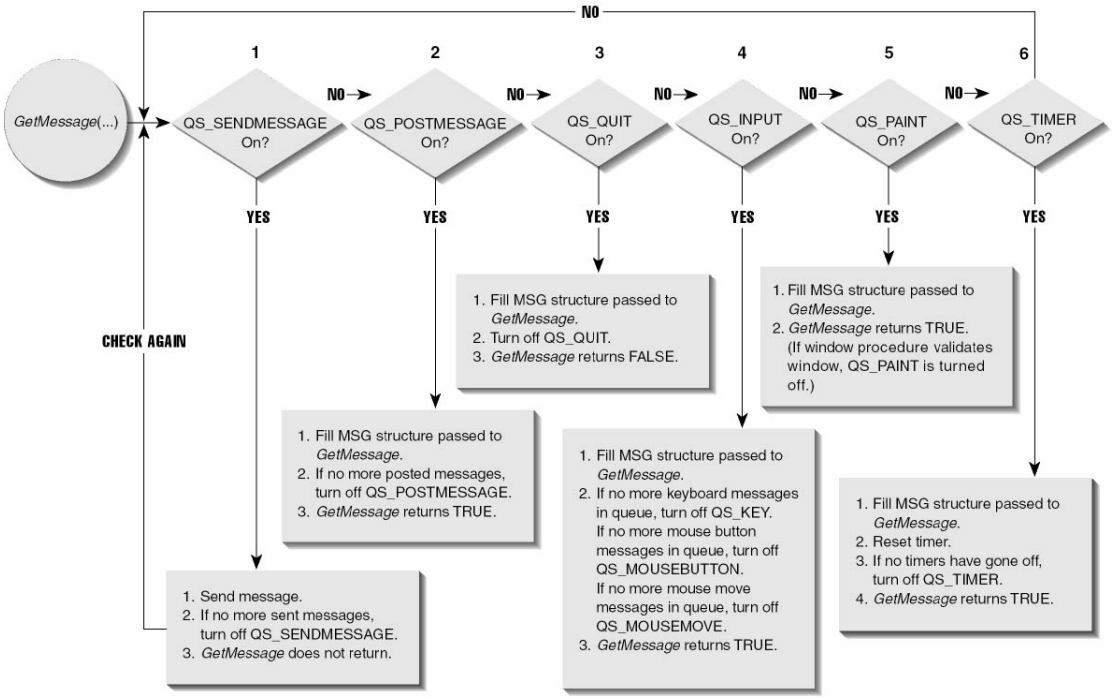
- **Knowledge Points**
- C++异常和 SEH 的对比
  1) SEH 是可用于任何编程语言的操作系统设施，而 C++异常处理只能用于编写 C++代码。
  2) VC++编译器已经利用操作系统的 SEH 实现了 C++异常处理
- **API Quick Reference**
- **Code Style and Technique Tricks**
- 在不要求再度恢复异常处理，并且编写的是 C++程序，那么应该使用 C++异常处理，因为 C++异常处理是 C++语言的一部分，编译器能够自动生成代码来调用 C++对象的析构函数，保证对象的清除

(六) 第六部分——窗口

1. Windows 消息

- **Knowledge Points**
- 对于窗口和 hook 这两种 User 对象它们分别由创建窗口和安装 hook 的线程所拥有。如果一个线程建立窗口或安装一个 hook，然后线程结束，那么系统将自动删除窗口和卸载 hook。建立窗口的线程必须是为窗口处理所有消息的线程，这也意味着，如果一个线程至少建立了一个窗口，操作系统会为它分配一个消息队列。这个队列用于窗口消息的派送。为了使窗口接收这些消息，线程必须有它自己的消息循环
- 当一个线程被建立时，系统分配一个 THREADINFO 结构，将数据与线程关联起来。THREADINFO 结构包含四个消息队列，分别为
  1) Posted message queue
  2) Send message queue
  3) Reply message queue
  4) Virtual input queue
- 以下函数发送的消息会被放到创建窗口的线程的 Posted message queue 中
  1) PostMessage
  2) PostThreadMessage
- 以下函数发送的消息会被放到创建窗口的线程的 Send message queue 中
  1) SendMessage（当发送的目的窗口是由另外的线程创建时）
  2) SendMessageTimeout
  3) SendMessageCallback
  4) SendNotifyMessage
- 以下函数发送的消息会被放到创建窗口的线程的 Reply message queue 中
  1) ReplyMessage
  2) SendMessage 可能引起 Reply message 放入此队列
  3) SendMessageCallback 可能引起 Reply message 放入此队列
- 硬件事件（鼠标/键盘）转换后的消息存放到 Virtual input queue 中
- 当一个线程调用 GetMessage 或 PeekMessage 时，系统必须检查线程的队

列标志的情况，并确定应该处理哪个消息。从线程队列中提取消息的算法如下图所示：



✧ 为什么要在检查 QS_SENDMESSAGE、QS_POSTMESSAGE、QS_QUIT 标志在检查硬件输入事件之前？微软设计这个算法时有个大前提，就是应用程序应该是用户驱动的，用户通过建立硬件输入事件（键盘和鼠标）来驱动应用程序。在使用应用程序时，用户可能按一个鼠标按钮，引起一系列要发生的事件。应用程序通过向线程的消息队列中登记消息使每个个别的事件发生。

举个例子说明。比如用户按鼠标按钮，处理 WM_LBUTTONDOWN 消息的窗口可能向不同的窗口投送三个消息。由于这是硬件事件引发的三个软件事件，所以系统要在读取用户的下一个硬件事件之前，先处理这三个软件事件。这也说明了为什么 Posted message queue 要在虚拟输入队列之前检查。这种事件序列的一个很好的例子是调用 TranslateMessage 函数。这个函数检查是否有一个 WM_KEYDOWN 或一个 WM_SYSKEYDOWN 消息从输入队列中取出。这些消息中的其中一个被取出来，系统检查虚拟键信息是否能够转换成等价的字符。如果虚拟键信息能够转换，TranslateMessage 调用 PostMessage 将一个 WM_CHAR 或 WM_SYSCHAR 消息放置在 Posted message queue 中。下次调用 GetMessage 时，系统首先检查 Posted message queue 中的内容，如果其中有消息存在，就从队列中取出消息并将其返回。返回的消息将是 WM_CHAR 或是 WM_SYSCHAR。下一次再调用 GetMessage 时，系统检查 Posted message queue，发现队列已空。系统再检查输入队列，在其中找到 WM_(SYS)KEYUP 消息。GetMessage 返回这个消息。所以硬件事件序列 WM_KEYDOWN、WM_KEYUP 生成 WM_KEYDOWN、WM_CHAR、WM_KEYUP 窗口过程的消息序列（假设虚拟键信息可以转换成等价的字符）

✧ GetMessage 或 PeekMessage 函数只检查唤醒标志和调用线程的消息队列。这意味着一个线程不能其它的线程队列中取得消息，包括同一进程内其它

线程的消息通过消息发送数据。一些窗口消息在其 lParam 参数中指出了一个内存块地址。例如 WM_SETTEXT 消息使用 lParam 参数作为指向一个以 0 结尾的字符串指针，这个字符串为窗口设定了新的文本标题。比如下面的调用：

```
char szBuf[200]
SendMessage(FindWindow(NULL, ”Calculator”), WM_SETTEXT,
        0, (LPARAM) szBuf);
```

因为 lParam 中的地址指向调用进程的地址空间中的字符串，而不是 Caculator 的地址空间。这本来会发生内存访问违规的严重问题。但为什么上面的调用会成功。答案是系统特别要检查 WM_SETTEXT 消息，并用与处理其它消息不同的方法来处理这个消息。上面的调用，系统实际上向那个窗口发送了两个消息。首先发送一个 WM_GETTEXTLENGTH。窗口过程通过返回窗口标题的字符数来响应这个消息。系统利用这个数字来建立一个内存映射文件，用于两个进程之间共享。对于系统已经知道的消息，发送消息时都可以按相应的方式来处理。

对于用户定义的消息（WM_USER+x）,Windows 建立了一个特殊的窗口消息 WM_COPYDATA 以解决这个问题

- 对于 WM_COPYDATA 消息，应该注意三个重要问题
  1) 只能 Send 这个消息，而不能 Post 这个消息。不能 Post 一个 WM_COPYDATA 消息是因为在接收消息的窗口过程处理完消息之后，系统必须释放内存映射文件。如果 Post 这个消息，系统不知道这个消息何时被处理，所以会导致不能释放那些内存块。
  2) 系统从另外的进程的地址空间中复制数据要花费一些时间。所以在 SendMessage 调用返回之前，不应让发送程序中运行的其它线程修改这个内存块
  3) 利用 WM_COPYDATA 消息，可以实现 16 为和 32 位之间的通信，也能实现 32 位和 64 位之间的通信。这是使新旧程序交流的便捷方法。在解决进程间通信的问题方面，WM_COPYDATA 消息是一个非常好的工具
- Windows 对 ANSI/Unicode 字符和字符串的处理原理。告诉系统一个窗口过程是要求 ANSI 字符串还是 Unicode 字符串，取决于注册窗口类时使用的函数。如果构造 WNDCLASS 结构，并调用 RegisterClassA，系统就认为窗口过程要求的字符串和字符都属于 ANSI。而用 RegisterClassA 注册窗口类，则系统就向窗口过程派送 Unicode 字符串和字符。发送消息之前，系统自动做了字符串的转换工作。
- 当对窗口进行子类化时，如何能保证原窗口过程得到正确的字符和字符串？系统需要两条信息：1）第一条是，字符和字符串当前具有的形式，即当前要传递的字符串是 ANSI 还是 Unicode。可以通过调用 CallWindowProcA 和 CallWindowProcW 来告诉系统。如果子类过程要把 ANSI 字符串传递给原来的窗口过程，子类过程必须调用 CallWindowProcA; 如果子类过程要把 Unicode 字符串传递给原来的窗口过程，那么子类过程必须调用 CallWindowProcW；2）第二条是，原来的窗口过程所要求的字符类型。系统从原窗口过程的地址获取这个信息。当调用了 SetWindowLongPtrA 或 SetWindowLongPtrW 函数时，系统要查看

是否使用一个 ANSI 过程来子类化一个 Unicode 窗口过程，或用一个 Unicode 过程来子类化一个 ANSI 窗口过程。如果没有改变原窗口过程所要求的字符类串型，那么 SetWindowLongPtr 只返回原窗口过程。如果改变了原窗口过程所要求的字符串类型，SetWindowLongPtr 不是返回原窗口过程的实际地址，而是返回一个内部子系统数据结构的句柄。这个数据结构包含原窗口过程的地址和一个数值，用来指示窗口过程是要求 ANSI 还是要求 Unicode 字符串。当调用 CallWindowProc 时，系统要查看是传递了一个内部数据结构的句柄，还是传递了一个窗口过程的地址。如果传递了一个窗口过程的地址，则调用原先的窗口过程，不需要执行字符串转换。如果传递了一个内部数据结构的句柄，则系统要将字符串转换成适当的类型（ANSI 或 Unicode），然后调用原窗口过程

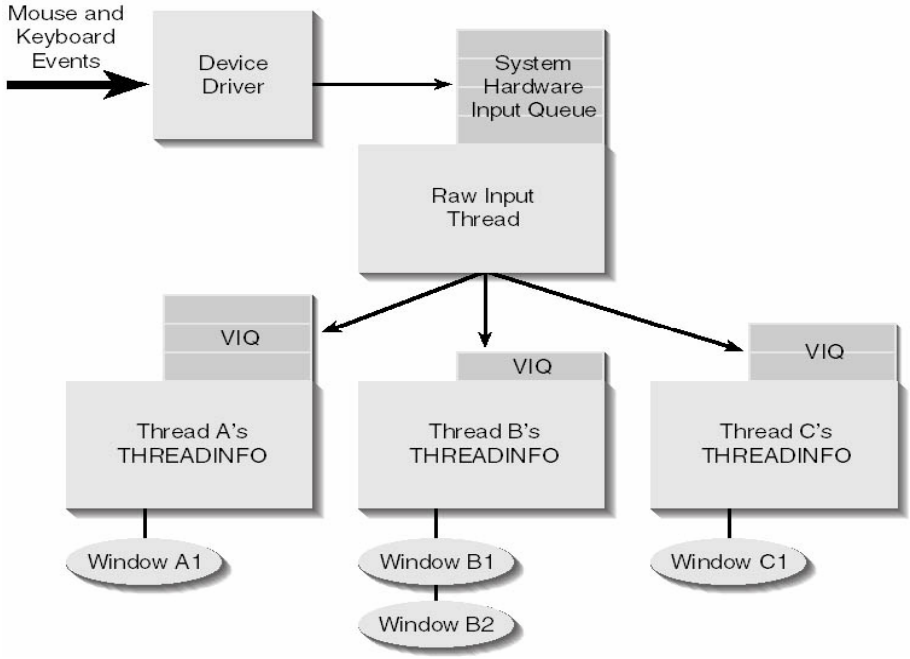
- **API Quick Reference**
- GetWindowThreadProcessId 用来确定哪个线程创建了窗口
- PostMessage 用于发送一个信息给特定窗口，立即返回（在线程的 Posted message queue 消息队列中登记消息）
- PostThreadMessage 用于发送一个信息给特定线程，立即返回（在线程的 Posted message queue 消息队列中登记消息）
- PostQuitMessage 用于终止消息循环（设置 THREADINFO 中的 QS_QUIT 标志就返回，并不在线程的消息队列中登记消息）
- SendMessage 用于将窗口消息直接发送给一个窗口过程，等待窗口过程处理完成后才返回。如果目的窗口是别的线程创建的，那么发送线程等待自己的 Posted message queue 消息队列有消息出现，这个消息是接收消息的线程处理完消息后发送过来的
- SendMessageTimeout
- SendMessageCallback 用于发送消息到接收线程的 Send message queue 消息队列，并立即返回到发送线程继续执行。当接收线程完成对消息的处理时，一个消息将会被登记到发送线程的 Reply message queue 消息队列。然后，系统通过调用回调函数将这个应答通知给发送进程
- SendNotifyMessage 用于将一个消息置于接收线程的 Send message queue 消息队列中，并立即返回到调用线程
- ReplyMessage 用于接收消息的线程将一个消息发送到发送消息线程的 Reply message queue 消息队列中，这可以使得发送消息线程醒来，继续执行
- InSendMessage 用于判断一个消息是线程间的，还是线程内的消息
- InSendMessage 用于确定窗口过程正在处理的消息类型
- GetMessage 用于等待窗口消息
- WaitMessage
- PeekMessage
- GetQueueStatus 用于查询队列的状态
- MsgWaitForMultipleObjects(Ex)用于线程等待它自己的消息
- IsWindowUnicode 用于确定窗口过程所要求的字符和字符串的类型
- SetWindowLongPtr 用于窗口子类化，设置自己的窗口过程，以后 Windows 系统直接把消息派发到我们自己的窗口过程
- CallWindowProc 用于告诉系统用什么类型的字符串传递给原窗口过程

➢ **Code Style and Technique Tricks**

2. 硬件输入模型和局部输入状态

➢ **Knowledge Points**

✧ 当系统初始化时，要建立一个特殊的线程，叫做原始输入线程（raw input thread，RIT）。此外系统还要建立一个队列，称之为系统硬件输入队列（system hardware input queue，SHIQ）。RIT 和 SHIQ 构成系统硬件输入模型的核心。硬件输入队列模型如下图所示：



✧ RIT 工作原理：RIT 通常处于睡眠状态，等待一个新项出现在 SHIQ 中。当用户按下或放开一个按键时，按下和松开鼠标按钮或移动鼠标的时候，这些硬件设备的硬件驱动程序就像 SHIQ 中增加一个硬件事件，这将唤醒 RIT。然后 RIT 从 RHIQ 中提取这个新项，并将这个事件转换成适当的 WM_KEY*、WM_?BUTTON*或 WM_MOUSEMOVE 消息。转换成的消息再添加到适当的线程的虚拟输入队列 VIQ。然后 RIT 再循环等待更多的消息出现在 SHIQ 中

✧ RIT 是如何知道要想那个线程的 VIQ 队列中增加硬件输入消息呢？

1) 对于鼠标消息的派发，RIT 只是确定哪一个窗口在鼠标光标下。利用这个窗口，RIT 调用 GetWindowThreadProcessId 来确定是哪个线程创建了这个窗口。返回的线程 ID 指出哪一个线程应该得到该鼠标消息
2) 对于键盘按键消息的处理稍有不同。在任何时刻，只有一个线程和 RIT“连接”这个线程称为前景线程（foreground thread），键盘消息就发往前景线程的 VIQ 队列中

✧ 线程是如何连接到 RIT 的呢？有几种情况：

1) 当产生一个进程时，这个进程的线程创建了一个窗口。这个窗口处于前景，其建立的窗口的线程同 RIT 连接
2) RIT 要负责处理特殊进的组合，如 Alt+Tab、Alt+Esc 和 Ctrl+Alt+Del 等。因为 RIT 在内部处理这些组合键，就可以保证用户总能用键盘激活窗口。应用程序不能拦截和废弃这些组合键。当用户按动了某个组

合键时，RIT 激活选定的窗口，并将窗口的线程连接到 RIT。

3) Windows 还提供激活窗口的功能，使窗口线程连接到 RIT

✧ 每个线程都有自己的局部输入状态，并在线程的 THREADINFO 结构内进行管理。这个局部输入状态由线程的虚拟输入队列和一组变量构成。这些变量跟踪下面的输入状态管理信息：

键盘输入和窗口焦点信息	鼠标光标管理信息
哪一个窗口有键盘焦点	哪一个窗口有鼠标捕获
哪一个窗口是活动的	鼠标光标的形状
哪一个按键认为是按下的	鼠标光标的可见性
插入符的状态	

➢ **API Quick Reference**

✧ SetActiveWindow 用于激活系统中一个最高层的窗口并对窗口设定焦点
✧ GetActiveWindow
✧ SetFocus 用于设置焦点窗口
✧ GetFocus
✧ BringWindowToTop
✧ SetWindowPos
✧ SetForgroundWindow 用于将窗口设为前景，并将它的线程连接到 RIT
✧ GetForegroundWindow
✧ AllowSetForegroundWindow
✧ LockSetForegroundWindow
✧ GetAsyncKeyStatus
✧ GetKeyState
✧ ClipCursor 用于将鼠标光标剪切到一个矩形区域
✧ SetCapture 用于窗口捕获鼠标
✧ ReleaseCapture
✧ AttachThreadInput 用来强制两个或多个线程共享一个虚拟输入队列和一组局部输入状态变量

➢ **Code Style and Technique Tricks**

(七) 第七部分——附录

1. 环境搭建

➢ **Knowledge Points**
➢ **API Quick Reference**
➢ **Code Style and Technique Tricks**

2. Message Cracks，Child Control Macros 和 API Macros

➢ **Knowledge Points**
➢ **API Quick Reference**
➢ **Code Style and Technique Tricks**