调用函数来创建一个内核对象时，如果调用失败，那么返回的句柄值通常为0（NULL），这就是为什么第一个有效的句柄值为4的原因，之所以失败，可能是由于系统内存不足，或者遇到了一个安全问题。遗憾的是，有几个函数在调用失败时会返回句柄值-1（也就是在WinBase.h中定义的INVALID\_HANDLE\_VALUE）。例如，如果CreateFile无法打开指定文件，他会返回INVALID\_HANDLE\_VALUE，而不是NULL。凡是用于创建内核对象的函数，在检查它们的返回值时，务必相当仔细。p38

复制句柄DuplicateHandle p58

子进程会继承父进程的环境变量，所以能轻松调用GetEnvironmentVariable来获得这个继承到的内核对象的句柄值。

cui程序可以通过int main(int argc,tchar \*argv[],tchar \*env[])获得环境变量。

GetCommandLine可以获得命令行

调用GetEnvironmentStrings函数获取完整的环境块p74

如果不再需要GetEnvironmentStrings函数返回的内存块，应调用FreeEnvironmentStrings函数来释放它

用户登录windows时，系统会创建外壳进程，并将一组环境字符串与其关联。系统通过检查注册表中的两个注册表项来获得初始的环境字符串。P77

GetModuleFileName，根据基地址获取文件名称p71

GetModuleHandle，根据文件名称返回基地址/句柄p71

GetModuleFileName，根据基地址/句柄返回文件名称（完整路径）

CommandLineToArgvW，将GetCommandLine的命令行分解成单独的标记。P74

通常，子进程会继承一组环境变量，这些环境变量和父进程的环境变量相同。不过，父进程可以控制哪些环境变量允许子进程继承，详情参见后文对CreateProcess函数的讨论。P77

在环境变量许多字符串的内部，都包含了“可替换字符串”。例如%USERPROFILE%\Documents两个百分号之间的部分是“可替换字符串”USERPROFILE环境变量的值是C:\Users\jrichter替换后生成的扩展字符串是C:\Users\jrichter\Documents由于这种形式的字符串替换非常常见，Windows提供了ExpandEnvironmentStrings函数p79

SetEnvironmentVariable函数添加，删除，修改一个变量的值。P79

如果不提供完整的路径名，windows函数会在当前驱动器的当前目录查找文件和目录。例如如果进程中的一个线程调用CreateFile来打开一个文件，系统将在当前驱动器和目录查找该文件。

系统在内部跟踪记录着一个进程的当前驱动器和目录。由于这种信息是以进程为单位来维护的，所以假如进程中的一个线程更改了当前驱动器或目录，那么对于该进程中的所有线程来说，此信息被更改了。

GetCurrentDirectory函数

SetCurrentDirectory函数p80

系统跟踪记录着进程的当前驱动器和目录，但它没有记录每个驱动器的当前目录。不过，利用操作系统提供的支持，可以处理多个驱动器的当前目录。这个支持是通过进程的环境字符串来提供的。例如，一个进程可以有如下所示的两个环境变量；

=c:=c:\Utility\Bin

=D:=D:\Program Files

假如进程的当前目录为C:\Utility\Bin，而且我们调用CreateFile来打开D:ReadMe.Txt，那么系统就会查找环境变量=D:，由于=D:变量是存在的，系统会尝试从D:\Program Files目录打开ReadMe.Txt文件。如果=D:不存在，系统会试着从D盘的根目录打开ReadMe.Txt文件p81

GetFullPathName函数来获得它的当前目录

CreateProcess会按照以下顺序搜索可执行文件。

主调进程.EXE文件所在的目录

主调进程的当前目录。

Windows系统目录，即GetSystemDirectory返回的System32子文件夹。

Windows目录。

PATH环境变量中列出的目录。P87

CreateProcess中的fdwCreate的CREATE\_SUSPENDED标志让系统在创建新进程的同时挂起其主线程，这样父进程就可以修改子进程地址空间中的内存，更改子进程的主线程的优先级p89

创建一个进程内核对象时，系统会为此对象分配一个独一无二的标识符，系统中没有别的进程内核对象会有相同的ID编号。这同样适用于线程内核对象。创建一个线程内核对象时，此对象会被分配一个独一无二的、系同级别的ID编号。进程ID和线程ID分享同一个号码池。P99

可以使用GetCurrentProcessId来得到当前进程的ID，使用GetCurrentThreadId来获得当前正在运行的线程的ID。另外，还可以使用GetProcessId来获得与指定句柄对应的一个进程的ID，使用GetThreadId来获得与指定句柄对应的一个线程的ID。最后，根据一个线程句柄，我们可以调用GetProcessIdOffThread来获得其所在进程的ID。P99

可以使用OpenProcess根据PID获得进程的句柄。

终止进程的4种方式

主线程的入口点函数返回

进程中的一个线程调用ExitProcess函数

另一个进程中的线程调用TerminateProcess函数

进程中的所有线程都“自然死亡”p100

自动提升进程的权限p109

调用ShellExecuteEx函数

枚举系统中正在运行的进程

这些函数在ToolHelp API中p114

OpenProcessToken打开指定进程的令牌

输出调试信息OutputDebugString

打开令牌OpenProcessToken，获取令牌信息GetTokenInformation，获取用户界面特权隔离等级（uipi）基于进程的MIC等级GetSidSubAuthority，获取进程的文件描述符GetSecurityInfo

Toolhelp32ReadProcessMemory

第一个参数是要读取的进程ID，你可以用很多办法获得这个ID。

第二个参数是你要读取的基地址。

第三个参数是你用来接收数据的缓冲区。

第四个参数是你要读取的字节大小。

第五个参数是最后你实际读取的字节大小，如果忽略可以取NULL。

CreateWellKnownSid传递WinbuiltinAministratorsSid参数来创建一个与管理员组一致的SID

CheckTokenMembership检查一个令牌的权限组

ShellExecuteEx提升权限启动进程

创建作业CreateJobObject，检查是否在作业内IsProcessInJob，设置作业限制SetInformationJobObject，关联作业AssignProcessToJobObject，杀死作业TerminateJobObject

一旦创建了I/O完成端口，就可以调用SetInformationJobObject将它与一个作业关联起来p136

GetProcessTimes获取进程时间

获得线程返回值GetExitCodeThread

务必注意ExitThread函数是用于“杀死”线程的Windows函数。如果要写C/C++代码，就绝对不要调用ExitThread，相反，应该使用C++运行库函数\_endthreadex。如果使用的不是Microsoft的C++编译器，那么编译器供应商应该提供它们自己的ExitThread替代函数p148

TerminateThread函数也可以“杀死”一个线程p149

不要使用CreateThread，要使用\_beginthreadex p154

GetCurrentProcess和GetCurrentThread获得进程和线程句柄p162

ResumeThread使进程变为可调度的，返回线程的前一个挂起计数。

SuspendThread挂起线程

挂起线程的所有进程SuspendProcess

CreateToolhelp32Snapshot可以通过获取进程信息为指定的进程、进程使用的堆[HEAP]、模块[MODULE]、线程建立一个快照。

HANDLE WINAPI CreateToolhelp32Snapshot(

DWORD dwFlags, //用来指定“快照”中需要返回的对象，可以是TH32CS\_SNAPPROCESS等

TH32CS\_SNAPALL在快照中包含系统中所有的进程和线程

DWORD th32ProcessID //一个进程ID号，用来指定要获取哪一个进程的快照，当获取系统进程列表或获取 当前进程快照时可以设为0

);

TH32CS\_INHERIT - 声明快照句柄是可继承的。

TH32CS\_SNAPALL - 在快照中包含系统中所有的进程和线程。

TH32CS\_SNAPHEAPLIST - 在快照中包含在th32ProcessID中指定的进程的所有的堆。

TH32CS\_SNAPMODULE - 在快照中包含在th32ProcessID中指定的进程的所有的模块。

TH32CS\_SNAPPROCESS - 在快照中包含系统中所有的进程。

TH32CS\_SNAPTHREAD - 在快照中包含系统中所有的线程。

VOID SuspendProcess(DWORD dwProcessID,BOOL fSuspend){

HANDLE hSnapshot=CreateToolhelp32Snapshot(

TH32CS\_SNAPTHREAD,dwProcessID);

if(hSnapshot!=INVALID\_HANDLE\_VALUE){

THREADENTRY32 te={sizeof(te)};

BOOL fOk=Thread32First(hSnapshot,&te);

for(;fOk;fOk=Thread32Next(hSnapshot,&te)){

if(te.th32OwnerProcessID==dwProcessID){

HANDLE hThread=OpenThread(THREAD\_SUSPEND\_RESUME,

FALSE,te.th32ThreadID);

if(hThread!=NULL){

if(fSuspend)

SuspendThread(hThread);

else

ResumeThread(hThread);

}

CloseHandle(hThread);

}

}

CloseHandle(hSnapshot);

}

} p168

获取执行任务的时间GetTickCount64，GetThreadTimes p171

获取进程时间GetProcessTimes

\_\_int64 FileTimeToQuadWord(PFILETIME pft) {

return (Int64ShllMod32(pft->dwHighDateTime, 32) | pft->dwLowDateTime);

}

GetThreadContext获得寄存器上下文，使用之前要调用SuspendThread p176

SetThreadContext设置寄存器上下文p177

一旦进程运行，便可以通过SetPriorityClass来改变自己的优先级p182

用来获取进程优先级的相应函数如下

DWORD GetPriorityClass(HANDLE hProcess);

在命令行界面输入以下命令，系统将调用计算器，并在idle优先级起始运行：

C:\>START /LOW CALC.EXE

设置线程的相对优先级SetThreadPriority p184

获取线程的相对优先级GetThreadPriority

我们可以通过调用SetThreadPriority并传入THREAD\_MODE\_BACKGROUND\_BEGIN来告诉Windows，线程应该发送低优先级的I/O请求。p187

cpu和线程关联p192

查看cpu的数量，通过调用GetSystemInfo来查询机器上cpu的数量，结果在结构SYSTEM\_INFO中的dwNumberOfProcessors中p193

如果要限制某些线程只在可用cpu的一个子集上运行，则可以调用SetProcessAffinityMask，第一个参数hProcess代表要设置的进程。第二个参数dwProcessAffinityMask是一个位掩码，代表线程可以在哪些cpu上运行。GetProcessAffinityMask可以返回进程的关联性掩码p193

限制线程只在一组cpu上运行SetThreadAffinityMask

要给线程设置一个理想的cpu，可以调用SetThreadIdealProcessor，效果是我们想让一个线程运行在一个cpu上，但系统允许将它移到另一个空闲的cpu p195

原子访问p198

InterlockedExchangeAdd和InterlockedExchangeAdd64，对LONG和LONGLONG类型进行加法运算p199，也可以做减法运算，只需要在第二个参数中传负值。

Interlocked函数执行极快，调用一次Interlocked函数通常只占用几个CPU周期，比内核对象同步速度快。

InterlockedExchange和InterlockedExchange64会把第一个参数所指向的内存地址的当前值，以原子方式替换成为第二个参数指定的值，返回原来的值，在实现旋转锁时InterlockedExchange极其有用，我们必须确保传给这些函数的变量地址是经过对齐的。p200

c运行库提供了一个\_aligned\_malloc函数，我们可以用这个函数来分配一块对齐过的内存p200

InterlockedExchangePointer和InterlockedExchange差不多，只是操作对象是指针

旋转锁会浪费大量的cpu时间，可以使用Sleep改善，许多开发人员会循环指定的次数，如果仍然无法访问资源，那么线程会切换到内核模式，并一直等到资源可供使用为止，这就是关键段的实现方式p201

Interlocked交换函数

PVOID InterlockedCompareExchange(

PLONG plDestination,

LONG lExchange,

LONG lComparand);

PVOID InterlockedCompareExchangePointer(

PVOID \*ppvDestination,

PVOID pvExchange,

PVOID pvComparand);

对32位来说，这两个函数都对32位值进行操作，在64位应用程序中，InterlockedCompareExchangePointer对64位进行操作p202

函数会将plDestination指向的当前值与参数lComparand的值进行比较，如果相同会将plDestination指向值修改为lExchange参数的值，否则plDestination指向的值不变，返回\*plDestination原来的值。p202

InterlockedCompareExchange64处理已对齐的64位值p202

InterlockedAnd64进行与操作。

高速缓存行，我们应该根据高速缓存行的大小来将应用程序的数据组织在一起，并将数据与缓存行的边界对齐。这样做的目的是为了确保不同的cpu能够各自访问不同的内存地址，而且这些地址不在同一个高速缓存行中。p204

如果不给布尔变量加上volatile限定符，编译器可能会对C++代码进行优化，然而变量的地址不需要加上volatile，因为传一个变量的地址给函数，那么函数必须从内存中读取它的值，编译器的优化程序不会对此产生影响p207

关键段p207

EnterCriticalSection进入关键段

LeaveCriticalSection离开关键段

InitializeCriticalSection初始化关键段

DeleteCriticalSection清理关键段

我们也可以用下面的函数来代替EnterCriticalSection，TryEnterCriticalSection

TryEnterCriticalSection从来不会让调用线程进入等待状态，它会通过返回值来表示调用线程是否获准访问资源，如果返回TRUE那么回进入关键段，必须有一个对应的LeaveCriticalSection。p211

读写锁 p214

InitializeSRWLock初始化读写锁

AcquireSRWLockExclusive写入线程上锁

ReleaseSRWLockExclusive写入线程解锁

AcquireSRWLockShared读取线程上锁

ReleaseSRWLockShared读取线程解锁

条件变量p217

我们已经看到，当想让写入者线程和读取者线程以独占模式或共享模式访问同一个资源的时候，可以使用SRWLock。在这些情况下，如果读取者线程没有数据可以读取，那么它应该将锁释放并等待，直到写入者线程产生了新的数据为止。如果用来接收写入者线程产生的数据的数据结构已满，那么写入者线程同样应该释放SRWLock并进入睡眠状态，直到读取者线程把数据结构清空为止。

有时我们想让线程以原子方式把锁释放并将自己阻塞，直到某一个条件成立为止。要实现这样的线程同步是比较复杂的。Windows通过SleepConditionVariableCS或SleepConditionVariableSRW函数，提供一种条件变量，来帮助我们简化这种情形下所需的工作。p217

当另一个线程检测到相应的条件已经满足的时候，会调用WakeConditionVariable或WakeAllConditionVariable，这样阻塞在Sleep\*函数中的线程就会被唤醒。p217

贯穿本书，我们已经讨论了好几种内核对象，包括进程、线程以及作业。几乎所有这些内核对象都可以用来进行同步。对线程同步来说，这些内核对象中的每一种要么处于触发状态，要么处于未触发状态。p230

下面的内核对象既可以处于触发状态，也可以处于未触发状态：

进程

线程

作业

文件以及控制台的标准输入流/输出流/错误流

事件

可等待的计时器

信号量

互斥量p231

等待函数WaitForSigleObject，第一个参数用来标识要等待的内核对象，第二个参数用来指定线程最多愿意花多长的时间来等待对象被触发，如果线程等待对象被触发，那么返回WAIT\_OBJECT\_0 ,如果是因为等待超时，那么返回值是WAIT\_TIMEOUT，如果WaitForSingleObject传入了无效参数，那么返回值是WAIT\_FAILED。p233

WaitForMultipleObjects p233

等待成功所引起的副作用p235

当一个手动重置事件被触发的时候，正在等待该事件的所有线程都将变成可调度状态。而当一个自动重置事件被触发的时候，只有一个正在等待该事件的线程会变成可调度状态。p236

CreateEvent创建事件内核p237

参数dwDesiredAccess允许我们指定在创建事件时返回的句柄对事件有何种访问权限。这是一种创建事件句柄的新方法，他可以减少权限，相比较而言，CreateEvent总是被授予全部权限。但CreateEventEx更有用的地方在于它允许我们以减少权限的方式打开已经存在的事件。p237

SetEvent可以把事件变成触发状态

ResetEvent可以把事件变为未触发状态

Microsoft为自动重置定义了一个等待成功所引起的副作用，当线程成功等到自动重置事件对象的时候，对象会自动地重置为未触发状态。相反，Microsoft并没有为手动重置对象定义一个等待成功所引起的副作用。

PulseEvent，发射一个脉冲，相对于在调用SetEvent之后立即调用ResetEvent，会触发当前等待事件的线程，手动重置的话触发全部等待线程，自动重置触发一个等待线程。

CreateWaitableTimer创建可等待的计时器

OpenWaitableTimer打开一个已经存在的可等待计时器，bManualReset表示手动重置还是自动重置，当手动重置计时器被触发时，正在等待该计时器的所有线程都会变成可调度状态。当自动重置计时器被触发的时候，只有一个正在等待该计时的线程会变成可调度状态。p245

SetWaitableTimer启动计时器p245

由于FILETIME结构和LARGE\_INTEGER结构的二进制格式完全相同，因此我们可能倾向于直接把FILETIME结构传给SetWaitableTimer，如下面的代码所示：

SetWaitableTimer(hTimer,(PLARGE\_INTEGER)&ftUTC,

6\*60\*60\*1000,NULL,NULL,FALSE);

事实上，这也正是我一开始所写的代码。但这是个严重的错误！虽然FILETIME结构和LARGE\_INTEGER结构具有完全相同的二进制格式，但是这两个结构的对齐方式是不同的。所有FILETIME结构的地址必须对齐到32位边界，而所有LARGE\_INTEGER结构的地址则必须对齐到64位边界。正确的做法是先把FILETIME的成员复制到LARGE\_INTEGER的成员中，然后再把LARGE\_INTEGER的地址传给SetWaitableTimer。p247

当计时器触发的时候，Microsoft还允许计时器把一个异步过程调用（asynchronous procedure call，APC）放到SetWaitableTimer的调用线程的队列中。当且仅当SetWaitableTimer的调用线程正处于可提醒状态时，这个函数会被同一线程调用。换句话说，线程必须是由于调用SleepEx、WaitForSingleObjectEx、WaitForMultipleObjectsEx、MsgWaitForMultipleObjectsEx或SignalObjectAndWait而进入的等待状态p248

SingleObjectAndWait 函数会触发一个内核对象并等待另一个内核对象。

DWORD WINAPI SignalObjectAndWait(

\_In\_ HANDLE hObjectToSignal,

\_In\_ HANDLE hObjectToWaitOn,

\_In\_ DWORD dwMilliseconds,

\_In\_ BOOL bAlertable

);p267

线程不应该在等待一个计时器句柄的同时以可提醒的方式等待同一个计时器。让我们看一看下面的代码：

HANDLE hTimer=CreateWaitableTimer(NULL,FALSE,NULL);

SetWaitableTimer(hTimer,...,TimerAPCRoutine,...);

WaitForSingleObjectEx(hTimer,INFINITE,TRUE);p250

信号量p251

信号量的规则如下：

如果当前资源计数大于0，那么信号量处于触发状态

如果当前资源计数等于0，那么信号量处于未触发状态

系统绝对不会让当前资源计数变为负数

当前资源计数绝对不会大于最大资源计数

创建信号量CreateSemaphore，CreateSemaphoreEx

打开存在的信号量句柄OpenSemaphore

线程通过调用ReleaseSemaphore递增信号量的当前资源计数，假如资源总数满了函数返回FALSE p252

很遗憾，我们没有办法在不改变当前资源计数的前提下得到它的值p253

互斥量内核对象用来确保一个线程独占对一个资源的访问。实际上，这也是互斥量名字的由来。互斥量对象包含一个使用计数，线程ID以及一个递归计数。互斥量与关键段的行为完全相同。但是，互斥量是内核对象，而关键段是用户模式下的同步对象。p253

互斥量的规则：

如果线程ID为0，那么该互斥量不为任何线程所占用，它处于触发状态。

如果线程ID为非零值，那么有一个线程已经占用了该互斥量，它处于未触发状态p253

创建互斥量CreateMutex，CreateMutexEx p254

打开互斥量OpenMutex

假如线程试图等待一个未触发的互斥量对象。在这种情况下，线程通常会进入等待状态。但是，系统会检查想要获得互斥量的线程的线程ID与互斥量对象内部记录的线程ID是否相同。如果线程ID一致，那么系统会让线程保持可调度状态——即使该互斥量尚未触发。p254

释放互斥量ReleaseMutex，这个函数会将对象的递归计数减1.如果线程成功地等待了互斥量对象不止一次，那么线程必须调用ReleaseMutex相同的次数才能使对象的递归计数变成0

当互斥量被遗弃的时候，系统会自动将互斥量对象的线程ID设为0，将它的递归计数设为0。p255

互斥量和关键段的比较表p256

线程同步对象速查表p264

线程可以调用WaitForInputIdle函数来将自己挂起：

DWORD WaitForInputIdle(

HANDLE hProcess,

DWORD dwMilliseconds);

这个函数会等待由hProcess标识的进程，直到创建应用程序第一个窗口的线程中没有待处理的输入为止。p265

MsgWaitForMultipleObjects和MsgWaitForMultipleObjectsEx p266

WaitForDebugEvent等待调试信息p267

如果没有SignalObjectAndWait函数，那么一个线程就无法知道另一个线程何时处于等待状态。对PulseEvent之类的函数来说，此类信息非常有用。正如本章前面已经提到过，PulseEvent会先触发一个事件，然后将自己重置。如果当前没有线程在等待该事件，那么没有线程会捕获到这个脉冲事件。我曾经见过有人编写出下面这样的代码：

// Perform some work...

SetEvent(hEventWorkerThreadDone);

WaitForSingleObject(hEventMoreWorkToBeDone,INFINITE);

// Do more work...

另一个线程执行下面这样的代码：

WaitPorSingleObject(hEventWorkerThreadDone);

PulseEvent(hEventMoreWorkToBeDone);

当第一个线程调用SetEvent时，第二个线程可能马上被唤醒并且调用了PulseEvent，此时第一个线程还没有调用WaitForSingleObject。p268

GetThreadWaitChain获得死锁的链表p272

各种设备及其常见用途p276

用来打开各种设备的函数p277

如果有一个设备句柄，那么可以调用GetFileType来查出设备的类型：

DWORD GetFileType(HANDLE hDevice);

我们只要把句柄传给GetFileType函数，该函数会返回FILE\_TYPE\_UNKNOWN、FILE\_TYPE\_DISK、FILE\_TYPE\_CHAR、FILE\_TYPE\_PIPE中的一个值。p278

打开文件CreateFile p278

关闭文件CloseHandle

在打开一个文件的时候，传入的路径名最长不能超过MAX\_PATH（在winDef.h中被定义为260）个字符 p280

获取文件的大小GetFileSizeEx p285

获取文件在磁盘上实际占用的大小 GetCompressedFileSize p286

如果需要随机访问文件，那么我们需要改变与文件内核对象相关联的文件指针。我们通过调用SetFilePointerEx来达到这个目的p287

有时我们可能想要强制使文件变得更小或变得更大，在这些情况下，可以调用下面的函数：

SetEndOfFile，这个SetEndOfFile函数会根据文件对象的文件指针当前所在的位置来截断文件的大小或增大文件的大小。p288

最方便和最常用的对设备数据进行读/写的函数是ReadFile和WriteFile，在打开设备的时候，我们一定不能指定FILE\_FLAG\_OVERLAPPED标志，否则系统会认为我们想要与该设备执行异步I/O。p289

强制系统将缓存数据写入到设备FlushFileBuffers p289

将一个给定线程尚未完成的同步I/O请求取消CancelSynchronousIo p290

异步I/O，为了以异步的方式来访问设备，我们必须先调用CreateFile，并在dwFlagsAndAttributes参数中指定FILE\_FALG\_OVERLAPPED标志来打开设备。这个标志告诉系统我们想要以异步的方式来访问设备。p292

在执行异步设备I/O的时候，我们必须在pOverlapped参数中传入一个已初始化的OVERLAPPED结构p292

如果请求的I/O操作是以异步方式执行的，或者在调用ReadFile或WriteFile的时候发生了错误，那么这两个函数就会返回FALSE，我们必须调用GetLastError来检查到底发生了什么。如果GetLastError返回的是ERROR\_IO\_PENDING，那么I/O请求已经被成功地加入了队列，会在晚些时候完成。p294

在异步I/O请求完成之前，一定不能移动或是销毁在发出I/O请求时所使用的数据缓存和OVERLAPPED结构，p295

有时候，我们可能想要在设备驱动程序对一个已经加入队列的设备I/O请求进行处理之前将其取消。Windows提供了多种方式来达到这一目的

我们可以调用CancelIo来取消由给定句柄所标识的线程添加到队列中的所有I/O请求

我们可以关闭设备句柄，来取消已经添加到队列中的所有I/O请求

当线程终止的时候，系统会自动取消该线程发出的所有I/O请求

发往给定文件句柄的一个指定I/O请求取消，调用CancelIoEx p316

用来接收I/O完成通知的方法

触发设备内核对象、触发事件内核对象、使用可提醒I/O、使用I/O完成端口p297

同步异步的关系

一旦一个线程触发了一个异步I/O请求，该线程将会继续运行，以执行其他有用的任务。但即便如此，线程最终还是需要与I/O操作的完成状态进行同步。换句话说，我们会运行到线程代码中的一个点，在这个点上，除非设备数据已经被载入到缓存中，否则线程将无法继续执行后继操作。p297

同步是在要等待上一个步骤完成才进入下一个步骤，异步是两个步骤一起完成

ReadFile和WriteFile函数在将I/O请求添加到队列之前，会先将设备内核对象设为未触发状态，当设备驱动程序完成了请求之后，驱动程序会将设备内核对象设为触发状态p297

当系统创建一个线程的时候，会同时创建一个与线程相关联的队列。这个队列被称为异步过程调用（asynchronous procedure call，APC）队列。当发出一个I/O请求的时候，我们可以告诉设备驱动程序在调用的APC队列中添加一项。为了将I/O完成通知添加到线程的APC队列中，我们应该调用ReadFileEx和WriteFileEx函数p301

与ReadFile和WriteFile相似，ReadFileEx和WriteFileEx在将I/O请求发给设备驱动程序之后，会立即返回。ReadFileEx和WriteFileEx的大多数参数与ReadFile和WriteFile相同，只有两个例外。首先，\*Ex函数没有一个指向DWORD的指针作为参数来保存已传输的字节数，该信息只有回调函数才能得到。其次，\*Ex函数要求我们传入一个回调函数的地址，这个回调函数被称为完成函数。

Windows提供了6个函数可以将线程置为可提醒状态：

SleepEx、WaitForSingleObjectEx、WaitForMultipleObjectsEx、SignalObjectAndWait、GetQueuedCompletionStatusEx、MsgWaitForMultipleObjectsEx

前5个函数的最后一个参数bAlertable是一个布尔值，表示调用线程是否应该将自己置为可提醒状态。在处理完apc后，可提醒函数的调用会返回p303

这6个函数的返回值表示它们返回的原因是什么。如果返回的是WAIT\_IO\_COMPLETION（或者GetLastError返回的是WAIT\_IO\_COMPLETION），那么我们就知道线程得以继续执行的原因是线程至少处理了APC队列中的一项。p304

Windows提供一个函数，允许我们手动地将一项添加到APC队列中QueueUserAPC

我们能够采用以下两种模型之一来构架一个服务应用程序。

串行模型 一个线程等待一个客户（通常是通过网络）发出请求。当请求到达的时候，线程会被唤醒并对客户请求进行处理。

并发模型 一个线程等待一个客户请求，并创建一个新的线程来处理请求。当新线程正在处理客户请求的时候，原来的线程会进入下一个循环并等待另一个客户请求。当处理客户请求的线程完成整个处理过程的时候，该线程就会终止。

串行模型的问题在于它不能很好地同时处理多个请求。如果两个客户同时发出请求，那么一次只能处理一个p306

CreateIoCompletionPort创建一个I/O完成端口，这个函数执行两项不同的任务，它不仅会创建一个I/O完成端口，而且会将一个设备与一个I/O完成端口关联起来p307

CreateNewCompletionPort

HANDLE CreateNewCompletionPort(DWORD dwNumberOfConcurrentThreads){

return(CreateIoCompletionPort(INVALID\_HANDLE\_VALUE,NULL,0,

dwNumberOfConcurrentThreads));

} p307

BOOL AssociateDeviceWithCompletionPort(

HANDLE hCompletionPort,HANDLE hDevice,DWORD dwCompletionKey){

HANDLE h=CreateIoCompletionPort(hDevice,hCompletionPort,dwCompletionKey,0);

return(h==hCompletionPort);

} p308

向设备发出I/O请求，但不把该项已完成的I/O请求添加到I/O完成端口的队列中也是有可能的p310

等待完成端口GetQueuedCompletionStatus p310

在Windows Vista中，如果预计会不断地收到大量的I/O请求，可以调用下面的函数来同时取得多个I/O请求的结果GetQueuedCompletionStatusEx，它可以指定可唤醒状态bAlertable p312

当我们创建I/O完成端口的时候，需要指定允许多少个线程并发运行。正如前面已经提到过，我们通常会将这个值设为主机的CPU数量。当已完成的I/O项被添加到队列中的时候，I/O完成端口想要唤醒正在等待的线程。但是，完成端口唤醒的线程数量最多不会超过我们指定的数量。因此，如果有4个I/O请求已完成，有4个线程正在等待GetQueue的CompletionStatus，那么I/O完成端口只会唤醒两个线程，而让其他两个线程继续睡眠。当每个线程处理完一个已完成的I/O项时，会再次调用GetQueuedCompletionStatus。这时系统发现队列中还有其他的项，于是会唤醒同一个线程来对剩余的项进行处理。

如果完成端口只允许同时唤醒指定数量的线程，那么为什么还要让更多的线程在线程池中等待呢？p313

我们可以通过以下3种方式之一来结束线程、完成端口的指派：

让线程退出。

让线程调用GetQueue的CompletionStatus，并传入另一个不同的I/O完成端口的句柄。

销毁线程当前被指派的I/O完成端口。p314

当线程数因为从挂起中恢复而使线程数大于CPU数量的时候，CPU不会再唤醒线程直到线程数小于CPU数量。p314

在10.5.3节“可提醒I/O”中，我们介绍了QueueUserAPC函数，该函数允许线程将一个APC项添加到另一个线程的队列中。I/O完成端口有一个类似的函数，名叫PostQueuedCompletionStatus，这个函数用来将一个已经完成的I/O通知追加到I/O完成端口的队列中p316

在Windows Vista中，当我们调用CloseHandle并传入一个完成端口的句柄时，系统会将所有正在等待GetQueuedCompletionStatus返回的线程唤醒，并返回FALSE给她们。此时调用GetLastError会返回ERROR\_INVALID\_HANDLE，线程可以通过这种方式来知道自己应该得体地退出了。p317

FileCopy分配一个CIOReq对象数组，并调用AllocBuffer函数来给每个I/O请求分配一块大小为BUFFSIZE的数据缓存。这个数据缓存是用VirtualAlloc函数分配的。使用VirtualAlloc函数可以确保内存块的起始位置位于分配粒度的边界，从而满足了FILE\_FLAG\_NO\_BUFFERING标志的要求，缓存的起始地址必须是磁盘卷的扇区大小的整数倍。p318

假如指定了FILE\_FLAG\_NO\_BUFFERING标志，那么文件操作必须在扇区的边界上进行。p319

Windows线程池函数允许我们做以下的这些事情：

以异步的方式来调用一个函数

每隔一段时间调用一个函数

当内核对象触发的时候调用一个函数

当异步I/O请求完成的时候调用一个函数p324

为了用线程池来以异步的方式执行一个函数，我们需要定义一个具有以下原型的函数：

VOID NTAPI SimpleCallback(

PTP\_CALLBACK\_INSTANCE pInstance,

PVOID pvContext);

然后为了让线程池中的一个线程执行该函数，我们需要向线程池提交一个请求。为了达到这个目的，我们只需调用下面的函数：

BOOL TrySubmitThreadpoolCallback(

PTP\_SIMPLE\_CALLBACK pfnCallback,

PVOID pvContext,

PTP\_CALLBACK\_ENVIRON pcbe);

该函数（通过调用PostQueuedCompletionStatus来）将一个工作项（work item）添加到线程池的队列中p325

每一次调用TrySubmitThreadpoolCallback的时候，系统会在内部以我们的名义分配一个工作项。如果打算提交大量的工作项，那么出于对性能和内存使用的考虑，创建工作项一次，然后分多次提交它会更好。我们调用下面的函数来创建一个工作项：

PTP\_WORK CreateThreadpoolWork(

PTP\_WORK\_CALLBACK pfnWorkHandler,

PVOID pvContext,

PTP\_CALLBACK\_ENVIRON pcbe);p326

在参数pfnWorkHandler中传入的函数必须符合下面的函数原型：

VOID CALLBACK WorkCallback(

PTP\_CALLBACK\_INSTANCE Instance,

PVOID Context,

PTP\_WORK Work);

当我们想要向线程池提交一个请求的时候，可以调用SubmitThreadpollWork函数

VOID SubmitThreadpoolWork(PTP\_WORK pWork);

如果我们有另一个线程，该线程想要取消已经提交的工作项，或者该线程由于要等待工作项处理完毕而需要将自己挂起，那么可以调用下面的函数：

VOID WaitForThreadpoolWorkCallbacks(

PTP\_WORK pWork,

BOOL bCancelPendingCallbacks);p326

如果我们传TRUE给bCancelPendingCallbacks参数，那么WaitForThreadpoolWorkCallbacks会试图取消先前提交的那个工作项。如果线程中的线程正在处理哪个工作项，那么该过程不会被打断，WaitForThreadpoolWorkCallbacks会一直等到该工作项已经完成后再返回。如果已提交的工作项尚未被任何线程处理，那么函数会先将它标记为已取消，然后立即返回。当完成端口从对列中取出该工作项的时候，线程池知道无需调用回调函数，这样该工作项根本就不会执行。p327

PostMessage传递信息给ui p328

情形2：每隔一段时间调用一个函数

为了将一个工作项安排在某个时间执行，我们必须定义一个回调函数，它的函数原型如下：

VOID CALLBACK TimeoutCallback(

PTP\_CALLBACK\_INSTANCE pInstance,

PVOID pvContext,

PTP\_TIMER pTimer);

然后调用下面的函数来通知线程池应该在何时调用我们的函数：

PTP\_TIMER CreateThreadpoolTimer(

PTP\_TIMER\_CALLBACK pfnTimerCallback,

PVOID pvContext,

PTP\_CALLBACK\_ENVIRON pcbe);

当我们想要向线程池注册计时器的时候，应该调用SetThreadpoolTimer函数p331

我们可以通过调用IsThreadpoolTimerSet来确定某个计时器是否已经被设置（也就是说，它的pftDueTime值不为NULL）p332

FindWindow查找特定名字的窗口句柄p334

如果工作项耗时较长，我们又不喜欢上述行为，而希望能够在每个工作项开始运行之后的10秒钟后将新的工作项添加到队列中，那么我们必须通过另一种途径来构造一种智能的一次性计时器。

我们仍然通过CreateThreadpoolTimer来创建计时器，这一点没有任何变化。

在调用SetThreadpoolTimer的时候给msPeriod参数传0，表示这个计时器是一次性的。

当待处理的工作完成后，重置计时器，仍然将msPeriod设为0。

最后，当最终需要停止计时器的时候，必须在CloseThreadpoolTimer执行之前调用WaitForThreadpoolTimerCallbacks，并传TRUE给最后一个参数，其目的是为了告诉线程池不应该再为该计数器处理任何的工作项。如果不这样做，那么线程池仍然会调用我们的回调函数，从而导致在SetThreadpoolTimer执行的时候引发异常。

注意，如果真的需要一次性计时器，则应该在回调函数中调用SetThreadpoolTimer并传0给msPeriod参数，同时为了确保对线程池的资源进行清理，我们还应该在回调函数返回之前调用CloseThreadpoolTimer。p335

情形3：在内核对象触发时调用一个函数

如果想要注册一个工作项，让它在一个内核对象被触发的时候执行，那么我们需要遵循的流程与本章前面已经介绍过的流程非常相似。首先，编写一个符合下面的原型的函数：

VOID CALLBACK WaitCallback(

PTP\_CALLBACK\_INSTANCE pInstance,

PVOID Context,

PTP\_WAIT Wait,

TP\_WAIT\_RESULT WaitResult);

然后通过调用CreateThreadpoolWait来创建一个线程池等待对象

PTP\_WAIT CreateThreadpoolWait(

PTP\_WAIT\_CALLBACK pfnWaitCallback,

PVOID pvContext,

PTP\_CALLBACK\_ENVIRON pcbe);

当创建完成后，我们调用下面的函数来将一个内核对象绑定到这个线程池：

VOID SetThreadpoolWait(

PTP\_WAIT pWaitItem,

HANDLE hObject,

PFILETIME pftTimeout);p336

当内核对象被触发或超出等待时间的时候，线程池中的某个线程会调用我们的WaitCallback函数

情形4：在异步I/O请求完成时调用一个函数

首先，我们必须编写一个符合以下原型的函数：

VOID CALLBACK OverlappedCompletionRoutine(

PTP\_CALLBACK\_INSTANCE pInstance,

PVOID pvContext,

PVOID pOverlapped,

ULONG IoResult,

ULONG\_PTR NumberOfBytesTransferred,

PTP\_IO pIo);

然后我们通过CreateThreadpoolIo来创建一个线程池I/O对象

PTP\_IO CreateThreadpoolIo(

HANDLE hDevice,

PTP\_WIN32\_IO\_CALLBACK pfnIoCallback,

PVOID pvContext,

PTP\_CALLBACK\_ENVIRON pcbe);

当线程池I/O对象创建完毕后，我们通过调用下面的函数来将嵌入在I/O项中的文件、设备与线程池内部的I/O完成端口关联起来：

VOID StartThreadpoolIo(PTP\_IO pio);

注意，在每次调用ReadFile和WriteFile之前，我们必须调用StartThreadpoolIo。如果每次在发出I/O请求之前没有调用StartThreadpoolIo，那么我们的回调函数将不会被调用。

如果想在发出I/O请求之后让线程池停止调用我们的回调函数，那么可以调用下面的函数：

VOID CancelThreadpoolIo(PTP\_IO pio);

如果在发出请求的时候，ReadFile或WriteFile调用失败了，那么我们仍然必须调用CaucelThreadpoolIo

我们还可以调用下面的函数来让另一个线程等待一个待处理的I/O请求完成：

VOID WaitForThreadpoolIoCallbacks(

PTP\_IO pio,

BOOL bCancelPendingCallbacks);p339

线程池提供了一种便利的方法，用来描述在我们的回调函数返回之后，应该执行的一些操作。回调函数用传给它的不透明的pInstance参数来调用以下这些函数：

LeaveCriticalSectionWhenCallbackReturns

ReleaseMutexWhenCallbackReturns

ReleaseSemaphoreWhenCallbackReturns

SetEventWhenCallbackReturns

FreeLibraryWhenCallbackReturns p339

FreeLibraryWhenCallbackReturns可以让在dll中的回调函数把自己从内存中卸载

除了这些终止函数之外，还有两个函数可用于回调函数的实例

BOOL CallbackMayRunLong(PTP\_CALLBACK\_INSTANCE pci);

VOID DisassociateCurrentThreadFormCallback(PTP\_CALLBACK\_INSTANCE pci);p340

我们可以在自己的应用程序中调用下面的函数来创建一个新的线程池：

PTP\_POOL CreateThreadpool(PVOID reserved);p340

设置最大最小数量

BOOL SetThreadpoolThreadMinimum(PTP\_POOL pThreadPool,DWORD cthrdMin);

BOOL SetThreadpoolThreadMaximum(PTP\_POOL pThreadPool,DWORD cthrdMost);

当应用程序不再需要它为自己定制的线程池时，应该调用CloseThreadpool将其销毁

一旦我们创建了自己的线程池，并指定了线程的最小数量和最大数量，我们就可以初始化一个回调环境\_TP\_CALLBACK\_ENVIRON p341

初始化VOID initializeThreadpoolEnvironment(PTP\_CALLBACK\_ENVIRON pcbe);

清理 VOID DestroyThreadpoolEnvironment(PTP\_CALLBACK\_ENVIRON pcbe);

和PTP\_POOL关联 VOID SetThreadpoolCallbackPool(PTP\_CALLBACK\_ENVIRON pcbe,PTP\_POOL pThreadPool);

告诉环境工作项通常需要较长的时间来处理VOID SetThreadpoolCallbackRunsLong(PTP\_CALLBACK\_ENVIRON pcbe);

SetThreadpoolCallbackLibrary确保只要线程中还有待处理的工作项，就将一个特定的dll一直保持在进程的地址空间p358

线程池清理组

创建CreateThreadpoolCleanupGroup

关联TP\_CALLBACK\_ENVIRON SetThreadpoolCallbackCleanupGroup

回调函数 VOID CALLBACK CleanupGroupCancelCallback(

PVOID pvObjectContext,

PVOID pvCleanupContext);

销毁线程池CloseThreadpoolCleanupGroupMembers p343

每个进程都有自己的虚拟地址空间。对于32位进程来说，这个地址空间的大小为4GB p355

每个进程都有自己私有的地址空间。进程A可以在位于它的地址空间内的0x12345678地址处存储一个数据结构，进程B也可以在“自己的”地址空间内存储一个完全不同的数据结构——同样位于0x12345678地址处。p356

进程的地址空间是如何划分的p356

用户模式分区p357

当系统即将运行一个应用程序时，它会检查应用程序在链接时是否使用了/LARGEADDRESSAWARE连接器开关。如果是，则相当于应用程序在声明它会充分利用大用户模式地址空间，而不会对内存地址进行任何不当的操作。p358

为了让64位应用程序能够访问整个用户模式分区，必须用/LARGEADDRESSAWARE链接器开关来链接应用程序。p359

内核模式分区p359

当系统创建一个进程并赋予它地址空间时，可用地址空间中的大部分都是闲置的或尚未分配的。为了使用这部分地址空间，我们必须调用VirtualAlloc来分配其中的区域。分配区域的操作被称为预订

当应用程序预订地址空间区域时，系统会确保区域的**起始地址**正好是分配粒度64KB p360

当应用程序预订地址空间中的一块区域时，系统会确保**区域的大小**正好是系统页面大小的整数倍。页面是一个内存单元，系统通过它来管理内存。与分配粒度相似，页面大小会根据不同的CPU而有所不同。x86和x64系统使用的页面大小为4KB，而IA-64系统使用的页面大小为8KB。

如果应用程序试图预定一块大小为10KB的地址空间区域，那么系统会自动将该请求取整到页面大小的整数倍。p360

释放预订地址 VirtualFree

当我们调拨物理存储器给区域时，并不需要给整个区域都调拨物理存储器。例如，我们可以预订一块大小为64KB的区域，然后把物理存储器拨给该区域中的第2个页面和第4个页面。p361

调拨物理存储器也是用VirtualAlloc，撤销调拨也是用VirtualFree

当今的操作系统能让磁盘空间看起来像内存一样。磁盘上的文件一般被称为**页交换文件**，其中包含虚拟内存，可供任何进程使用。

页交换文件的使用并不是必须的。如果一台机器没有页交换文件，那么对系统来说，它只是认为可供应用程序使用的内存总量减少了。（VirtualAlloc是直接在内存中分配空间的，经验证VirtualAlloc真的会在页交换文件中拿空间）代码（PVOID P=VirtualAlloc(NULL, 1000000000, MEM\_COMMIT | MEM\_RESERVE, PAGE\_READWRITE);

for (int i = 0; i<1000000000; i++)

((char\*)P)[i] = 1;

VirtualFree(P, 0, MEM\_RELEASE);）

之后用spy++查看，不过在没有分配页交换文件的情况下也是可以运行的，而且增加的是物理内存

最好是把物理存储器看成是保存在磁盘上的页交换文件的数据。当应用程序调用VirtualAlloc函数来把物理存储器调拨给地址空间区域时，该空间实际上是从硬盘上的页交换文件分配得到的（可能对现在来说有误，当年一般只有2G内存）。p362

当访问的数据不在内存中，而是位于页交换文件中的某处。这次不成功的访问称为页面错误。p362

当用户要求执行一个应用程序时，系统会打开该应用程序对应的exe文件并计算出应用程序的代码和数据的大小。然后预订一块地址空间，并注明该区域相关联的物理存储器就是exe文件本身。

当把一个程序位于硬盘上的文件映像用作地址空间区域对应的物理存储器时，我们称这个文件映像为**内存映射文件**，（exe和dll是映像文件）。p364

页面保护属性p365

PAGE\_NOACCESS

PAGE\_READONLY

PAGE\_READWRITE

PAGE\_EXECUTE

PAGE\_EXECUTE\_READ

PAGE\_EXECUTE\_READWRITE

PAGE\_WRITECOPY

PAGE\_EXECUTE\_WRITECOPY P365

PAGE\_READWRITE是更改原始数据，PAGE\_WRITECOPY是写时复制更改备份数据

还有3个保护属性的标志PAGE\_NOCACHE,PAGE\_WRITECOMBINE和PAGE\_GUARD。使用这些标志时，只需将他们与除了PAGE\_NOACCESS之外的任何其他保护属性进行按位操作即可。p367

写时复制：每个内存在写的时候有自己的副本。p366

地址空间映射的实例：区域p367

其中E代表可执行，R代表可读，W代表可写，C是WRITECOPY(?)copy-on-write p370

区域类型，他可能是表13-5所列出的4种类型之一：闲置、私有、印像、已映射

闲置：虚拟地址没有任何后备存储器。该地址空间尚未预订（已经预订而又没有调拨的话算什么？私有吗？好像就是私有，的确是私有）

私有：以页交换文件为后备存储器

映像：一开始以映像文件为后备存储器，但此后不一定以映像文件为后备存储器。假如写入的话可能写时复制会用页交换文件作为后备存储器

已映射：一开始以内存映射文件为后备存储器，但此后不一定以内存映射文件为后备存储器。假如写入的话可能写时复制会用页交换文件作为后备存储器p372

地址空间映射实例：区域和块p372

块的类型：闲置、私有、已映射、映像、保留（保留就是已经保留还没有调拨）

大部分情况下，一个区域中所有已调拨的块都以相同类型的物理存储器为后备存储器，但是也有可能不同p375

（貌似区域的保护属性是块保护属性的或运算，经验证不对，块可以拥有区域没有的保护属性）p374

数据对齐的重要性

x86的话数据不对齐要访问两次

GetSystemInfo函数用来取得与主机相关的值p380

如果想要得到机器中与处理器有关的详细信息，可以调用GetLogicalProcessorInformation函数p380

判断是否在64位机器上运行32位程序（WOW64）IsWow64Process p381

需要调用GetNativeSystemInfo来取得原来的SYSTEM\_INFO结构 p381

Windows函数GlobalMemoryStatus可以用来取得当前内存的动态信息p387

dwMemoryLoad cpu的繁忙程度

dwTotalPhys 物理内存总量

dwAvailPhys 可用物理内存

dwTotalPageFile 页交换文件总量

dwAvailPageFile 可用页交换文件

dwTotalVirtual 地址空间总量

dwAvailVirtual 本进程可用地址空间

如果预计应用程序会在装有4GB的内存的机器上运行，调用新的GlobalMemoryStatusEx函数。

GetNumaAvailableMemoryNode

GetNumaProcessorNode

GetNumaHighestNodeNumber

GetNumaNodeProcessorMask p388

我们把一个进程的地址空间中被保存在内存里的那些页面称为它的**工作集**，GetProcessMemoryInfo可以取得进程当前工作集的大小和最大工作集的大小。p390

Windows提供了一个函数，可以用来查询与地址空间中的内存地址有关的特定信息，这个函数是VirtualQuery，允许一个进程查询另一个进程的内存信息VirtualQueryEx p391

MEMORY\_DBASIC\_INFORMATION结构的成员

BaseAddress块的起始地址

AllocationBase区域的基地址

AllocationProtect区域的保护属性

RegionSize块的大小

State块的状态

Protect块的保护属性，（相邻的页面，页面指的是每4KB内存块，有相同的保护属性，状态和类型的话就是块了）

Type区域的状态

作者自创的VMQuery函数p392

VMQuery和toolhelp函数的区别

toolhelp能查询模块地址，模块的大小，路径名称，句柄，进程id，堆id，线程引用计数，exe的名字，dwFlags等信息

VMQuery能查询每个地址空间是什么，区域和块的状态，大小，保护标志就4样

Windows提供以下三种机制来对内存进行操控。

虚拟内存（virtualalloc）

内存映射文件（filemapping）

堆 p400

VirtualAlloc p400

在预订区域并指定保护属性时，应该考虑在调拨物理存储器时最常用的保护属性。当区域的保护属性和将要调拨的物理存储器的保护属性相一致时，系统内部的处理效率会更高。p402

VirtualAlloc不能使用PAGE\_EXECUTE\_WRITECOPY和PAGE\_WRITECOPY

VirtualAlloc预订时不能使用PAGE\_GUARD，PAGE\_NOCACHE，PAGE\_WRITECOMBINE p402

同时预订和调拨 MEM\_RESERVE|MEM\_COMMIT（其实直接用MEM\_COMMIT也是可以的）

大页面支持

SIZE\_T GetLargePageMinimum();

在VirtualAlloc中传入MEM\_LARGE\_PAGE

得到不可换页 p404

VirtualFree p407

VirtualFree MEM\_RELEASE撤销预订，MEM\_DECOMMIT撤销调拨

撤销预订时dwSize必需为0 p407

修改保护属性VirtualProtect 改变的是页面区域（也就是块，大概？页面区域好像是以4KB为单位的区域）p415

如果我们在一台页面大小为4KB的机器上用下面的代码调用VirtualProtect，那么我们实际上是在给两个物理存储页指定PAGE\_NOACCESS保护属性

VirtualProtect(pvRgnBase+(3\*1024),2\*1024,PAGE\_NOACCESS,&floldProtect); p415

重置物理存储器的内容，调用VirtualAlloc函数，并在第三个参数中传MEM\_RESET标志p416

向下取整到页面大小的整数倍，取一个比当前值少或等于而且刚好是4KB的倍数的地址

向上取整，取刚好大于而且是4KB的倍数。p416

如果传入的是MEM\_RESET，那么VirtualAlloc会从相反的方向进行取整操作。p416

MEM\_RESET感觉在这个不需要页交换文件的时代已经没有意义了。

（MemReset示例程序，可以确定作者的理论已经过期，作者的机器只有1g，我的有8g，作者说VirtualAlloc分配的内存在页交换文件，然而我们分配的是在内存。内存和页交换文件可以看作是两个地位相同的区域，一个数据可以存储在内存中也可以存储在页交换文件中，不存在页交换文件和内存要一一对应的关系，也不存在先在页交换文件中分配然后拷贝到内存的理论。VirtualAlloc也可以在内存中分配。经验证VirtualAlloc真的会在页交换文件中拿空间）p417

地址窗口扩展（Address Windowing Extension，后面称为AWE）。在创建AWE时，Microsoft有以下两个目标。

允许应用程序以一种特殊的方式分配内存，操作系统保证不会将以这种方式分配的内存换出到磁盘上。

允许应用程序访问比进程地址空间还要多的内存

应用程序然后调用一个函数，每调用一次把一块内存指定到该地址窗口。把内存块指定到地址窗口是非常快的。

AWE调用VirtualAlloc并且把MEM\_PHYSICAL传入p419

示例程序中实际分配内存的是new，通过MapUserPhysicalPages映射到VirtualAlloc返回的窗口中

当系统创建线程时，会为线程栈预订一块空间区域（每个线程都有自己的栈），并给区域调拨一些物理存储器。默认的情况下，系统会预订1MB的地址空间并调拨两个页面的存储器。有两种方法可以改变默认值

编译器/F

链接器/STACK p429

CreateThread和\_beginthreadex可以指定空间大小

在预定地址空间区域后，系统会给区域顶部（即地址最高）的两个页面调拨物理存储器。在让线程开始执行之前，系统会把线程栈的指针指向区域顶部的那个页面的末尾（该地址非常接近0x081000000）。这个页面就是线程开始使用栈的地方。区域顶部往下的第二个页面被称为**防护页面** p429

当系统给地址为0x08001000的页面调拨物理存储器时，它会执行一个额外的操作——抛出EXCEPTION\_STACK\_OVERFLOW异常。通过使用结构化异常处理（structured exception handling，后面简称为SEH），系统会在发生这一情况是通知我们的程序，从而使程序能够得体地从这一异常情况下恢复。p431

栈下溢，访问了未知的区域

BYTE aBytes[100];

aBytes[10000]=0; p432

内存映射文件 p440

内存映射文件好像和exe，dll没有关系，exe和dll好像是映像文件（实在搞不清）

创建进程时加载exe和dll的步骤p441

不会共享静态数据p442

共享静态数据p444

SHARED该段的内容为多个实例所共享（这个属性实际上关闭了写时复制机制）p444

颠倒文件内容的例子p451

使用内存映射文件p453

打开一个文件内核对象CreateFile p454

现在我们必须告诉系统文件映射对象需要多大的物理存储器，必须调用CreateFileMapping 创建文件映射内核对象p455

页面保护属性p456

PAGE\_READONLY

PAGE\_READWRITE

PAGE\_WRITECOPY

PAGE\_EXECUTE\_READ

PAGE\_EXECUTE\_READWRITE

5种段属性p456

SEC\_NOCACHE

SEC\_IMAGE

SEC\_RESERVE

SEC\_COMMIT

SEC\_LARGE\_PAGES

dwMaxmumSizeLow和dwMaximumSizeHigh分别表示最大大小的低位和高位p457

在创建了文件映射对象之后，还需要为文件的数据预订一块地址空间区域并将文件的数据作为物理存储器调拨给区域这可以通过调用MapViewOfFile来实现

内存映射文件访问权限p459

FILE\_MAP\_WRITE

FILE\_MAP\_READ

FILE\_MAP\_ALL\_ACCESS

FILE\_MAP\_COPY

FILE\_MAP\_EXECUTE

我们必须告诉系统应该把数据文件中的哪个字节映射到视图中的第一个字节。这是通过参数dwFileOffsetHigh和dwFileOffsetLow来指定的。我们必须告诉系统要把数据文件中的多少映射到地址空间中。这和预订地址空间区域时需要指定区域的大小是一样的道理。参数dwNumberOfBytesToMap用来指定大小，（dwFileOffsetHigh，dwFileOffsetLow，dwNumberOfBytesToMap所指定的范围不能超过CreateFileMapping的dwMaximumSizeHigh和dwMaximumSizeLow）p459

writecopy就是写时复制，写的东西不会保存到映射文件中。

撤销对文件数据的映射UnmapViewOfFile

如果需要确保所做的修改已经被写入到磁盘中，那么可以调用FlushViewOfFile p462

但是，系统在调用MapViewOfFile的时候还会增加文件对象和文件映射对象的引用计数p463

用内存映射文件来处理大文件的方法p469

内存映射文件和一致性

我们可以先把一个文件的前10KB映射到一个视图中，然后再把同一个文件的前4KB映射到另一个视图中。只要我们映射的是同一个文件映射对象，那么系统会确保各视图中的数据是一致的p470

如果打算将打开的文件用于内存映射，那么在调用CreateFile的时候最好是传0给dwShareMode参数

MapViewOfFileEx函数代替MapViewOfFile函数，这样就能建议系统把文件映射到指定的地址p471

MapViewOfFileEx可以解决跨进程的链表p471

Windows提供了多种机制，使得应用程序之间能够快速、方便地共享数据和信息，这些机制包括RPC、COM、OLE、DDE、Windows消息、剪切板、邮件槽、管道、套接字等。在Windows中，在同一台机器上共享数据的最底层的机制就是内存映射文件。p473

exe的载入过程p473

以页交换文件为后备存储器的内存映射文件

系统能够创建以页交换文件为后备存储器的内存映射文件，这样就不需要用磁盘上专门的文件来作为后备存储器了。不必创建或打开一个专门的磁盘文件，因此不需要调用CreateFile。我们只需要像原来那样调用CreateFileMapping，并将INVALID\_HANDLE\_VALUE作为hFile参数传入，所分配的存储器大小由CreateFileMapping的dwMaximumSizeHigh和dwMaximumSizeLow参数决定。如果想和其他进程共享数据，那么可以在调用CreateFileMapping的时候将一个以0位终止符的字符串作为pszName参数传入，这样其他想要访问共享数据的进程就能够以同一个名称为参数来调用CreateFileMapping或OpenFileMapping。p474

稀疏调拨的内存映射文件

我们不希望在一开始就调拨所有的物理存储器。CreateFileMapping为我们提供了一种方法，即在fdwProtect参数中指定SEC\_RESERVE或SEC\_COMMIT标志p478

SEC\_COMMIT标志让CreateFileMapping从页交换文件中调拨存储器，这和不指定任何标志具有相同的效果。

如果在调用CreateFileMapping的时候传入SEC\_RESERVE标志，那么系统不会从页交换文件中调拨物理存储器，它只返回文件映射对象的句柄。现在我们可以调用MapViewOfFile或MapViewOfFileEx来给这个文件映射对象创建一个视图。MapViewOfFile和MapViewOfFileEx会预订一块地址空间区域，但不会给区域调拨任何物理存储器。试图访问区域中的内存地址将会引发访问违规p479

为了给共享区域调拨物理存储器，我们需要调用VirtualAlloc

第15章已经对这个函数进行了非常详细的介绍。无论区域是用VirtualAlloc和MEM\_RESERVE标志预订得到的，还是对内存映射文件进行映射得到的，给它们调拨物理存储器的方法都是一样的，即VirtualAlloc p479

内存文件映射和VirtualAlloc不同，可以使用writecopy，writecopy改变的内容不会反映到原来的映射文件和页交换文件（共享）。代码（HANDLE hFileMap = CreateFileMapping(NULL, NULL, PAGE\_WRITECOPY,

0, 10, TEXT("myMap"));

PVOID P = MapViewOfFile(hFileMap, FILE\_MAP\_COPY, 0, 0, 10);

int ret = GetLastError();

((char\*)P)[0] = 1;

UnmapViewOfFile(P);

CloseHandle(hFileMap);

CloseHandle(hFile);）

NTFS文件系统提供了对稀疏文件的支持，我们可以在普通磁盘文件中进行稀疏文件p479

DeviceIoControl让系统支持稀疏文件

DeviceIoControl(m\_hStream, FSCTL\_SET\_SPARSE, NULL, 0, NULL, 0, &dw, NULL)

DeviceIoControl查询

DeviceIoControl(m\_hStream, FSCTL\_QUERY\_ALLOCATED\_RANGES, &farb, sizeof(farb), pfarb, cb, &cb, NULL);

DeviceIoControl补充空洞

DeviceIoControl(m\_hStream, FSCTL\_SET\_ZERO\_DATA, (PVOID) &fzdi, sizeof(fzdi), NULL, 0, &dw, NULL)

堆

与第15章介绍的虚拟内存和第17章介绍的内存映射文件相比，堆是用来管理链表和树的最佳方式。堆的优点是它能让我们专心解决手头上的问题，而不必理会分配粒度和页面边界这类事情。堆的缺点是分配和释放内存块的速度比其他方式慢。而且也无法在对物理存储器的调拨和撤销调拨进行直接控制p491

系统保证不管在什么时候，一次只让一个线程从默认堆中分配或释放内存块。如果两个线程同时想要从默认堆中分配一块内存，那么只有其中一个线程能够分配内存，另一个线程必须等待第一个线程的分陪完成p491

默认堆是在进程开始运行之前由系统自动创建的，在进程终止后会自动销毁。我们无法销毁金成德默认堆。每个堆都有一个用来标识自己的堆句柄，所有分配和释放内存块的堆函数都会在参数中用到这个堆句柄p491

我们可以通过调用GetProcessHeap来得到进程的默认堆的句柄p492

额外堆好处

对组件进行保护

更有效的内存管理

局部访问

避免线程同步的开销

快速释放p492

创建额外的堆HeapCreate，第一个参数fdwOptions用来表示对堆的操作该如何进行。可以指定0、HEAP\_NO\_SERIALIZE,HEAP\_GENERATE\_EXCEPTIONS,HEAP\_CREATE\_ENABLE\_EXECUTE或这些标志的组合。p495

当任何程序试图从堆中分配一块内存的时候，HeapAlloc函数必须执行以下操作

遍历已分配内存的链表和闲置内存的链表

找到一块足够大的闲置内存块

分配一块新的内存，也就是将刚找到的闲置内存块标记为已分配

将新分配的内存块添加到已分配内存的链表中p495

应该避免使用HEAP\_NO\_SERIALIZE标志，其对堆的访问是非独占的，如果一个线程没有完成对堆的操作，其它线程也可以进程堆操作

满足以下条件之一或更多的时候，使用HEAP\_NO\_SERIALIZE标志才是安全的

进程中只有一个线程

进程中有多个线程，但只有一个线程会访问这个堆

进程中有多个线程，但进程使用了其他方式来管理对堆的独占访问，比如临界区（critical section）、互斥量（mutex）、以及信号量（semaphore）p496

HEAP\_GENERATE\_EXCEPTIONS告诉系统，每当在堆中分配或重新分配内存块失败的时候，抛出一个异常p496

HeapSetInformation p496

在堆中存放可执行代码HEAP\_CREATE\_ENABLE\_EXECUTE p497

HeapCreate的第2个参数dwInitialSize表示一开始要调拨给堆的字节数。如果需要HeapCreate会把这个值向上取整到cpu页面大小的整数倍。最好一个参数dwMaximumSize表示堆所能增长到的最大大小。如果dwMaximumSize大于0，那么创建的堆会有一个最大大小。这时，如果试图分配的内存块可能导致堆超过最大大小，分配操作会失败。p497

堆空间约2g

经验证初始位置大概在0x0f000000附近（分配1g内存）

假如在中间截断的话分配会失败

PVOID p=VirtualAlloc((PVOID)0x40000000, 1024 \* 4, MEM\_COMMIT | MEM\_RESERVE, PAGE\_READWRITE);//第一个参数是0x10000000的话会成功的

char \*pchar = new char[(long long)1024\*1024\*1024\*3/2]();

//或者是char \*pchar=(char\*)HeapAlloc(GetProcessHeap(), HEAP\_ZERO\_MEMORY | HEAP\_GENERATE\_EXCEPTIONS, 1024 \* 1024 \* 1024 );

printf("%x,%x", pchar,pchar+ (long long)1024 \* 1024 \* 1024 \* 3 / 2);

delete[]pchar;

return 0;

HeapAlloc的fdwFlags支持3个标志HEAP\_ZERO\_MEMORY，HEAP\_GENERATE\_EXCEPTIONS，HEAP\_NO\_SERIALIZE p497

HEAP\_ZERO\_MEMORY会让HeapAlloc把内存清零

HEAP\_GENERATE\_EXCEPTIONS和HeapCreate中的一样，如果之前指定了那么就不用写了

最后的标志HEAP\_NO\_SERIALIZE用来强调系统不要把这次HeapAlloc调用与其他线程对同一个堆的访问依次排列起来p498

在分配大块内存的时候应该避免使用堆函数，建议使用VirtualAlloc函数p498

如果分配大量不同大小的内存块，那么堆管理器内部用来处理分配请求的默认算法可能会产生地址空间碎片，由于所有可用的内存块都不够大，因此系统将无法找到一块住够大的闲置内存块，可以使用**低碎片堆**的算法p498

调整内存块的大小HeapReAlloc p499

hHeap用来标识需要调整大小的内存块属于哪个堆

fdwFlags有四个标志HEAP\_GENERATE\_EXCEPTIONS，HEAP\_NO\_SERIALIZE，HEAP\_ZERO\_MEMORY，HEAP\_REALLOC\_IN\_PLACE\_ONLY

HEAP\_REALLOC\_IN\_PLACE\_ONLY用来告诉HeapReAlloc不要移动内存块

pvMem当前内存块的地址

dwBytes内存块新的大小p499

HeapSize函数来得到这块内存的实际大小p499

不再需要一块内存的时候，我们可以调用HeapFree来释放它

如果应用程序不再需要自己创建的堆，则可以调用HeapDestroy来销毁它

对new和delete进行重载p501

Heap32First，Heap32Next，Heap32ListFirst和Heap32ListNext p503

由于进程在自己的地址空间中可以有多个堆，所以GetProcessHeaps函数可以让我们得到这些堆的句柄p503

HeapValidate用来验证堆的完整性p504

为了让堆中闲置的内存块能重新接合在一起，并撤销调拨给堆中闲置内存块的存储器，可以调用下面的函数HeapCompact

HeapLock和HeapUnlock用于线程同步

遍历堆的内容HeapWalk p504

三个最重要的dll分别是：

Kernel32.dll包含函数用来管理内存、进程以及线程

User32.dll包含的函数用来执行与用户界面相关的任务，如创建窗口和发送消息

GDI32.dll包含的函数用来绘制图像和显示文字p509

两种加载dll的方式：隐式载入时链接或显示运行时链接p510

p511例子错误的原因是静态链接时会有自己的一个heap，假如exe和dll一个静态链接一个动态链接的话又或者是两个都是静态链接的话那么就会有两个堆，函数在一个堆中分配内存而在另一个堆中释放内存。所以free调用会失败。p511

dll创建过程及应用程序隐式链接到dll的过程p512

构建dll需要以下步骤p513

一旦构建了dll模块，我们就可以通过下列步骤来构建可执行模块p513

导入段，其中列出了所有它需要的dll模块的名称。此外，对列出的每个dll，该段还记录了可执行文件二进制代码从中引用的函数和变量的符号名。（当调用dll中的函数时，会从导入段跳转到dll的函数中）p513

一个dll可以导出变量、函数或c++类来供其他模块使用。在实际开发中，我们应该避免从dll中导出变量，因为这等于是去掉了代码中的一个抽象层，从而使得dll的代码更加难以维护。此外，只有当导出c++类的模块使用的编译器与导入c++类的模块使用的编译器由同一家厂商提供时，我们才可以导出c++类。否则我们应该避免从dll中导出c++类。

\_\_declspec(dllexport)导出变量、函数或c++类p515

extern"C"，只有在编写c++代码的时候，才应该使用这个修饰符，在编写c代码的时候不应该使用该修饰符。extern"C"用来告诉编译器不要对变量名或函数名进行改编，这样用c、c++或任何编程语言编写的可执行模块都可以访问该变量或函数p516

\_declspec(dllimport)导出p516

在链接dll的时候，连接器会检测到这些与导出的变量、函数或类有关的嵌入信息，并生成一个.lib文件。这个.lib文件列出了该dll导出的符号。在链接任何可执行模块的时候，只要可执行模块引用了该dll导出的符号，那么这个.lib文件当然是必需的。除了创建这个.lib文件之外，链接器还会在生成的dll文件中嵌入一个导出符号表。这个导出段列出了导出的变量、函数和类的符号名。链接器还会保存相对虚拟地址（relative virtual address，RVA），表示每个符号可以在dll模块中的何处找到。p516

我们可以使用Microsoft Visual Studio提供的DumpBin.exe工具来查看一个dll的导出段

当使用\_\_stdcall来导出c函数时Microsoft编译器会对函数名进行改编，如MyFunc改成\_MyFunc@8。p518

不要改编，第一张方法创建一个.def文件，并在.def文件中包含一个类似下面的EXPORTS段

EXPORTS

MyFunc p518

第二种方法

在dll的源文件添加一行类似下面的代码

#pragma comment(linker,"/export:MyFunc=\_MyFunc@8") p519

在导入符号的时候，不必使用\_\_declspec(dllimport)关键字，而可以直接使用标准c语言的extern关键字p520

dll搜索顺序

包含可执行文件的目录

Windows的系统目录，该目录可以通过GetSystemDirectory得到

16位的系统目录，即Windows目录中的System子目录

Windows目录，该目录可以通过GetWindowsDirectory得到

进程的当前目录

PATH环境变量中所列出的目录p522

dll的创建过程以及应用程序显式链接到dll的过程

在任何时候，进程中的一个线程可以调用下面两个函数来将一个dll映射到进程的地址空间中LoadLibrary，LoadLibraryEx p525

LoadLibraryEx的dwFlags参数

DONT\_RESOLVE\_DLL\_REFERENCES，LOAD\_LIBRARY\_AS\_DATAFILE，LOAD\_LIBRARY\_AS\_DATAFILE\_EXCLUSIVE，LOAD\_LIBRARY\_AS\_IMAGE\_RESOURCE，LOAD\_WITH\_ALTERED\_SEARCH\_PATH，LOAD\_IGNORE\_CODE\_AUTHZ\_LEVEL p526

DONT\_RESOLVE\_DLL\_REFERENCES不要调用DllMain

LOAD\_LIBRARY\_AS\_DATAFILE，系统不需要额外的时间准备执行文件中的代码

LOAD\_LIBRARY\_AS\_DATAFILE\_EXCLUSIVE，以独占模式打开

LOAD\_LIBRARY\_AS\_IMAGE\_RESOURCE对虚拟地址RVA进行修复

传入LOAD\_WITH\_ALTERED\_SEARCH\_PATH时LoadLibraryEx的dll搜索顺序

全路径的话直接加载dll

否则将下列文件夹与pszDLLPathName连接起来

进程的当前目录

Windows的系统目录

16位Windows系统目录——即Windows目录下的System子目录

Windows目录

PATH环境变量中列出的目录p528

SetDllDirectory能载入特定文件夹的dll p529

卸载dll FreeLibrary，FreeLibraryAndExitThread p529

实际上，每个dll在进程中有一个与之对应的使用计数，LoadLibrary和LoadLibraryEx会递增使用计数，FreeLibrary和FreeLibraryAndExitThread会递减使用计数

GetModuleFileName函数来得到dll或exe的全路径 p531

获得dll中的函数指针GetProcAddress

FARPROC GetProcAddress(

HMODULE hInstDll,

PCSTR pszSymbolName);

参数pszSymbolName可以有两种形式，以符号名来指定我们想要得到的哪个符号的地址，例如"SimeFuncInDll"，和用序号来指顶我们想要得到哪个符号的地址MAKEINTRESOURCE(2) p532

dll入口函数DllMain，用于接收通知p533

系统是在什么时候调用DllMain函数的呢？静态链接时，或动态链接时调用LoadLibrary和FreeLibrary都会调用DllMain函数。

fdwReason表示系统调用入口点函数的原因。这个参数可能是下列4个值之一

DLL\_PROCESS\_ATTACH，DLL\_PROCESS\_DETACH，DLL\_THREAD\_ATTACH，DLL\_THREAD\_DETACH p533

第一次将一个dll映射到进程的地址空间中时，会传入DLL\_PROCESS\_ATTACH，只有第一次映射的时候才会这样p534

撤销映射DLL\_PROCESS\_DETACH p535

当进程创建一个线程的时候，系统会检查当前映射到该进程的地址空间中的所有dll文件映像，并用DLL\_THREAD\_ATTACH来调用每个DLL的DllMain函数p537

当线程终止时终止的线程会用DLL\_THREAD\_DETACH来调用所有已映射dll的DllMain函数p538

DllMain的序列化调用p538

p539最后一行应该是发送DLL\_THREAD\_ATTACH

DisableThreadLibraryCalls p540

在链接dll的时候，链接器会将dll的入口点函数的地址嵌入到生成的dll文件映像中，默认是\_DllMainCRTStartup

在链接dll的时候，如果链接器无法在dll的.obj文件中找到一个名为DllMain的函数，那么它会链接c/c++运行库的DllMain函数p542

延迟载入dll p542

一个导出了字段的dll是无法延迟载入的

Kernel32.dll模块是无法延迟载入的

不应该在DllMain入口点函数中调用一个延迟载入的函数

延迟载入需要增加两个链接器开关

/Lib:DelayImp.lib

/DelayLoad:MyDll.dll p543

如果\_\_delayLoadHelper2能够找到dll，但无法在dll中找到我们试图调用的函数，那么回产生另一个问题，\_\_delayLoadHelper2会抛出一个异常p545

软件异常码VcppException(ERROR\_SEVERITY\_ERROR，ERROR\_MOD\_NOT\_FOUND)和VcppException(ERROR\_SEVERITY\_ERROR,ERROR\_PROC\_NOT\_FOUND)，它们分别表示dll模块未找到或函数未找到，异常过滤函数DelayLoadDllExceptionFilter会对这两个异常码进行检查，如果抛出的异常不是其中的任何一个，那么过滤器会返回EXCEPTION\_CONTINUE\_SEARCH，否则\_\_delayLoadHelper2函数会提供一个指向DelayLoadInfo结构的指针（参考http://www.cnblogs.com/5iedu/p/5014798.html DelayLoadInfo的指针在pep->ExceptionRecord->ExceptionInformation[0]中） p545

卸载一个延迟载入的dll，要做两件事情。首先，必须在构建可执行文件的时候指定一个额外的链接器开关（/Delay:unload）。其次，必须修改源代码，在我们想要卸载dll的地方调用\_\_FUnloadDelayLoadedDLL2函数p546

函数转发器p553

#pragma comment(linker,"/export:SimeFunc=DllWork.SomeOtherFunc")

这个pragmatic告诉链接器，正在编译的DLL应该输出一个名为SomeFunc的函数，但实际实现SomeFunc的是另一个名为SomeOtherFunc的函数，该函数被包含在另一个名为DllWork.dll的模块中。我们必须为每个想要转发的函数单独创建一行pragma。p553

已知的dll

在注册表中有这么一个注册表项

HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manaer\KnownDLLs

如果我们指定了.dll扩展名，那么这两个函数会先将扩展名去掉，然后再在KnownDLLs注册表项中搜索，看其中是否有与之相符的值名，如果没有那么函数会使用正常的搜索规则。但是如果找到了与之相符的值名，那么系统会查看与值名相对应的数据，并试图用该数据来载入dll

调用下面的函数时，系统会用正常的搜索规则来对这个dll进行定位

LoadLibrary(TEXT("SomeLib"));

但是，如果调用下面的函数，那么系统会发现有一个与之相符的名称

LoadLibrary(TEXT("SomeLib.dll"));p554

dll重定向，为了强制加载程序总是先检查应用程序的目录，我们所要做的就是将一个文件放到应用程序的目录中。这个文件的内容无关紧要，但它的文件名必须是AppName.local，例如SuperApp.exe的话重定向文件的名称必须是SuperApp.exe.local p555

我们还可以创建一个名为.local的文件夹并将自己的dll保存在这个文件夹中

模块的基地址重定位p556

已知的dll和重定向无法隐式加载（静态链接）

如果加载程序无法将模块载入到它的首选基地址，那么系统会打开模块的重定位段并遍历其中的所有的条目。对每一个条目，加载程序会先找到包含机器指令的那个存储页面，然后将模块的首选基地址与模块的实际映射地址之间的差值，加到机器指令当前正在使用的内存地址上。p558

当一个模块无法被载入到它的首选基地址时，存在以下两个主要的缺点

加载程序必须遍历重定位段并修改模块中的大量代码。

当加载程序写入到模块的代码页面中时，系统的写时复制机制会强制这些页面以系统的页交换文件为后备存储器。p558

强制不重定位，我们可以创建一个不包含重定位段的可执行文件或dll模块。在构建模块的时候使用/FIXED开关就能够达到这个目的。这个开关可以让模块的大小变小，但它同时意味着模块无法被重定位。如果模块无法被载入到它的首选基地址，那么它将完全不会被载入。p559

为了创建一个不包含任何重定位信息的映像，应该用/SUBSYSTEM:WINDOWS

指定基地址configuration Properties\Linker\Advanced在Base Address一栏中输入一个数值p559

rebase.exe可以重新指定模块的基地址p560

线程局部存储区（Thread Local Storage，简称TLS）

p566页图中的第一行是所有线程共有的，表示每个线程都有而又线程之间各自独立的变量的索引，例如我们规定buf的索引是3，那么每个线程都会在自己的**线程的数组**中用3去索引buf，然而线程的数组之间相互独立，所以每个线程的buf都是不同的。

要使用动态TLS，我们必须先调用TlsAlloc p567

当系统创建一个线程的时候，会分配TLS\_MINIMUM\_AVAILABLE个PVOID值，将它们都初始化为0，并与线程关联起来。如图21-1下面所示。它们是**线程的数组**p567

为了把一个值放到**线程的数组**中，我们应该调用TlsSetValue函数，dwTlsIndex标识的是线程的数组的位置，也是TlsAlloc的返回值p567

为了从线程的数组中取回一个值，我们应该调用TlsGetValue，它会检查传入的索引是否在已分配的区间内。p568

不需要已经预订的TLS元素，调用TlsFree

通常，如果dll要使用TLS，那它会在DllMain函数处理DLL\_PROCESS\_ATTACH的时候调用TlsAlloc，在DllMain处理DLL\_PROCESS\_DETACH的时候调用TlsFree p568

TlsAlloc在返回之前会遍历进程中的每个线程，并根据新分配的索引，在每个线程的数组中把对应的元素设为0。p569

静态TLS

\_\_declspec(thread) DWORD gt\_dwStartTime=0; p570

应用程序需要跨越线程边界来访问另一个进程的地址空间的情况如下

我们想要从另一个进程创建的窗口派生子类窗口。

我们需要一些手段来辅助调试——例如，我们需要确定另一个进程正在使用哪些dll

我们想要给另一个进程安装挂钩。p572

派生子类窗口，本质上是修改WndProc指向新的函数

如果用下面的代码调用SetWindowLongPtr来从一个窗口派生子类窗口，那等于是告诉系统：所有发到或发往hwnd窗口的消息，应该由MySubclassProc来处理，而不是由该窗口的标准窗口过程来处理

SetWindowLongPtr(hWnd,GWLP\_WNDPROC,MySubclassProc); p573

假如窗口句柄和函数地址不在同一个线程那么SetWindowLongPtr会失败p574

如果通过某种方式让我们的子类窗口的窗口过程进入到进程A的地址空间中，就能够轻易地调用SetWindowLongPtr，并把MySubclassProc在进程A中的地址传给它。我称这项技术为将DLL“注入”到进程的地址空间中p574

使用注册表来注入dll p575

安装挂钩SetWindowsHookEx

第一个参数表示要安装挂钩的类型，第二个参数GetMsgProc是一个函数的地址（在我们的地址空间中），第3个参数标识一个dll，这个dll包含了GetMsgProc函数。在windows中，hInstDll的值是进程地址空间中dll被映射到的虚拟内存地址。最后一个参数表示要给哪个线程安装挂钩。

进程A调用SetWindowsHookEx后，进程B发生了什么p577

撤销映射UnhookWindowsHookEx p577

chASSERT，检查是否为真，如果是假的话那么报错

GetFirstChild，得到窗口句柄的子句柄

FindWindow，查找特定名字的窗口句柄

SendMessage，对窗口传送消息，在proc中处理

GetMessage，调用线程的消息队列里取得一个消息并将其放于指定的结构

PostThreadMessage，其功能是将一个消息放入（寄送）到指定线程的消息队列里，不等待线程处理消息就返回

CreateDialog，创建和原线程同一个线程的另外一个窗口

队列消息，通过GetMessage来的到的消息，一般来说通过DispatchMessage可以把队列消息发给窗口过程处理（proc）

使用远程线程来注入dll

注入dll的第三种方法是使用远程线程，它提供最高的灵活性p587

从根本上说，dll注入技术要求目标进程中的一个线程调用LoadLibrary来载入我们想要的dll。由于我们不能轻易地控制别人进程中的线程，因此这种方法要求我们在目标进程中创建一个新的线程。由于这个线程是我们自己创建的，因此我们可以对它执行的代码加以控制。幸运的是，Windows提供了如下所示的CreateRemoteThread函数，它使得在另一个进程中创建线程变得非常容易

HANDLE CreateRemoteThread(

HANDLE hProcess,

PSECURITY\_ATTRIBUTES psa,

DWORD dwStackSize,

PTHREAD\_START\_ROUTINE pfnStartAddr,

PVOID pvParam,

DWORD fdwCreate,

PDWORD pdwThreadId);

LoadLibrary函数的函数原型和线程函数的函数原型基本相同。下面是线程函数的函数原型

DWORD WINAPI ThreadFunc(PVOID pvParam);p588

在编译和链接一个程序的时候，生成的二进制文件中会包含一个导入段（在第19章中介绍）。这个段由一系列转换函数构成，这些转换函数用来跳转到导入的函数。因此，在代码调用诸如LoadLibraryW之类的函数时，链接器会生成一个调用，来调用我们模块的导入段中的一个转换函数，这个转换函数然后会跳转到实际的函数。

如果在调用CreateRemoteThread的时候直接引用LoadLibraryW，该引用会被解析为我们模块的导入段中的LoadLibraryW转换函数的地址，如果把这个转换函数的地址作为远程线程的起始地址传入，那么天知道远程线程会执行什么代码，其结果很可能是访问违规。为了强制代码略过转换函数并直接调用LoadLibraryW函数，我们必须通过调用GetProcAddress来得到LoadLibraryW的确切地址。p589

必须假定Kernel32.dll映射到同一个地址p589

必须像下面这样来调用CreateRemoteThread

PTHREAD\_START\_ROUTINE pfnThreadRtn=(PTHREAD\_START\_ROUTINE)

GetProcAddress(GetModuleHandle(TEXT("Kernel32")),"LoadLibraryW");

HANDLE hThread=CreateRemoteThread(hProcessRemote,NULL,0,

pfnThreadRtn,L"C:\\MyLib.dll",0,NULL);

字符串"C:\\MyLib.dll"位于调用进程的地址空间中。我们把这个地址传给新创建的远程线程，远程线程再把它传给LoadLibraryW。但当LoadLibraryW去访问这个内存地址的时候，DLL的路径字符串并不在那里，远程进程的线程很可能会引发访问违规，系统会向用户显示一条未处理的异常消息框，然后终止远程进程。没错，被终止的是远程进程，而不是我们的进程。p589

为了解决这个问题，我们需要把dll的路径字符串存放到远程进程的地址空间中去。然后，在调用CreateRemoteThread的时候，我们需要传入存放字符串的地址。幸运的是，Windows为我们提供了VirtualAllocEx函数，他可以让一个进程在另一个进程的地址空间中分配一块内存，VirtualFreeEx可以释放这块内存p590

Windows提供了一些函数，可以让一个进程对另一个进程的地址空间进行读写

BOOL ReadProcessMemory(

HANDLE hProcess,

LPCVOID pvAddressRemote,

PVOID pvBufferLocal,

SIZE\_T dwSize,

SIZE\_T\* pdwNumBytesRead);

BOOL WriteProcessMemory(

HANDLE hProcess,

PVOID pvAddressRemote,

LPCVOID pvBufferLocal,

SIZE\_T dwSize,

SIZE\_T\* pdwNumBytesWritten);p590

远程注入步骤p590

为了试图打开这个正在运行的进程的句柄，程序会用这个标识符来调用OpenProcess，并请求合适的访问权限

hProcess=OpenProcess(

PROCESS\_CREATE\_THREAD|

PROCESS\_VM\_OPERATION|

PROCESS\_VM\_WRITE,

FALSE,dwProcessId);

如果OpenProcess返回NULL，那说明应用程序所在的安全上下文不允许它打开目标进程的句柄

使用木马dll来注入dll

注入dll的另一种方式是，把我们知道的进程必然会载入的一个dll替换掉p598

把dll作为调试器来注入p598

使用CreateProcess来注入代码p599

api拦截p599

修改导入段来拦截api p601

CAPIHook::ReplaceIATEntryInOneMod

CAPIHook::ReplaceEATEntryInOneMod的用途是在一个模块的导出表中替换指定符号（函数）的地址，用自己的函数替换自己的函数，pfnNew = (PROC)((PBYTE)pfnNew - (PBYTE)hmod);的目的是将新的地址写入RVA。

SEH

SEH实际上包含两方面的功能：终止处理和异常处理。p622

终止处理确保不管一个代码块是如何推出的，另一个代码块总能被调用和执行。终止处理的语法如下所示：

\_\_try{

//Grarded body

}

\_\_finally{

//Termination handler

} p622

通过使用终止处理程序可以防止过早地执行return语句。当return语句试图退出try块的时候，编译器会让finally代码块在它之前执行。p623

try语句块中包含一个return语句，所以finally块之后的代码都没有机会执行，返回值为try快中的返回值p624

编译器就会生成一些代码先将返回值保存在一个由它创建的临时变量里，然后再执行finally代码块，这个过程被称之为**局部展开**。p624

可以用\_\_leave关键字代替return语句，它可以避免局部展开开销 p624

在try中使用goto语句 p625

在进程或者线程被提前终止的情况下，系统没法保证finally代码块的执行。调用ExitThread或者ExitProcess可以马上终止线程或者进程，而不会引发finally代码块执行。同样还有TerminateThread或者TerminateProcess p627

在finally块中调用return返回，返回值会覆盖原来的返回值。

一条好的经验法则是，不要在终止处理程序里包含让try块提前退出的语句p628

关键字\_\_leave会导致代码执行控制流跳转到try块的结尾——也可以认为是跳转到了闭花括号处。p631

引起finally块执行的情况

1.从try块到finally的正常代码控制流

2.局部展开：从try块中的提前退出（由goto,longjump,continue,break,return等语句引起）将程序控制流强制转入finally块。

3.全局展开——在本章的Funcfurther1中发生过，如果Funcinator引起内存访问违规，那么全局，那么在vista以前的系统上会引起访问违规

可以调用AbnormalTermination确定finally块的执行理由。p631

使用终止处理程序的理由

因为清理工作集中在一个地方执行，并且保证能得到执行，从而简化了错误处理。

提高了代码的可读性。

让代码更容易维护。

如果正确使用，它们对程序性能和体积的影响是微小的。p632

全局展开，在抛出异常到EXCEPTION\_EXECUTE\_HANDLER之间的所有finally块一个接一个运行

EXCEPTION\_EXECUTE\_HANDLER p641

抛出异常后在except块处理后在except块之后运行

全局展开，当异常过滤程序的计算结果为EXCEPTION\_EXECUTE\_HANDLER时，系统必须执行全局展开。全局展开导致所有已经开始执行但尚未完成的try-finally块得以继续执行，在调用栈中，这些try-finally块位于对异常进行了处理的try-except块的下方p645

全局展开过程，抛出异常，搜索except块，执行except块的异常过滤程序，并等待其返回结果，假如返回结果是EXCEPTION\_EXECUTE\_HANDLER，从栈的底部向上搜索已经开始执行尚未完成的try块，寻找所有带finally块的try块，并一一执行，最后执行except异常处理程序p646

停止全局展开，我们可以通过将return语句置于finally块中以阻止系统完成全局展开p647

一条原则是，尽量避免将return语句置于finally块中。

EXCEPTION\_CONTINUE\_EXECUTION p648

抛出异常经过处理后会从失败的地方开始执行

可以让异常过滤程序调用一个函数，来决定返回三个标识符中的哪一个p648

EXCEPTION\_CONTINUE\_SEARCH p650

会继续向上查找异常处理的except块

GetExceptionCode p652

返回的各种异常p652

内在函数GetExceptionCode只能在异常过滤程序里或异常处理程序的代码里调用，所以要想这样写\_\_except(CoffeeFilter(GetExceptionCode())){}p654

//excpt.h

#define GetExceptionCode \_exception\_code

#define GetExceptionInformation (struct \_EXCEPTION\_POINTERS\*)\_exception\_info

错误码的组成部分

GetExceptionInformation p656

当一个异常发生时，

GetExceptionInformation只能在异常过滤程序中调用，这是因为CONTEXT，EXCEPTION\_RECORD，以及EXCEPTION\_POINTERS数据结构只有在系统计算异常过滤程序时才是有效的。一旦程序控制流被转移到异常处理程序或者别的地方，这些栈上的数据结构就被销毁了。p656

EXCEPTION\_RECORD结构包含了关于刚发生的异常的详细信息p658

CONTEXT结构（请参考第7章）p659

抛出一个软件异常RaiseException p660

未处理异常p663

Microsoft Windows提供了函数SetUnhandledExceptionFilter，它给了我们处理异常的最后机会，否则Windows就会正式认为这个异常没有得到处理。

异常过滤函数的原型必须与下面类似

LONG WINAPI TopLevelUnhandledExceptionFilter(PEXCEPTION\_POINTERS pExceptionInfo);

默认全局异常过滤程序 \_\_CxxUnhandledExceptionFilter p664

如果我们的线程抛出一个异常，并且所有安装的异常过滤程序都返回EXCEPTION\_CONTINUE\_SEARCH，系统提供的一个特殊函数将被自动调用，即UnhandledExceptionFilter。p665

2018年2月5日追记

基地址和句柄不是同一个东西

ReleaseSemaphore时假如超过最大限制会返回false