<<windows核心编程>>阅读笔记

2017年10月17日 星期二　　19:10

韦侃忠

学前知识-错误处理

2017年1月22日 9:42

API返回值类型预示函数调用状态

Windows的大多数API参数繁多，功能不一 ，常常会在无意中调用失败，失败的原因往往并不能直接观察到．因此，Windows的API通常会通过返回值来告知函数是否被调用成功，这些返回值一般有如下类型:

数据类型	指出函数是否调用失败或失败的错误码
VOID	返回这种类型的API一般都不可能调用失败,但只有极少的API的返回值是这种类型
BOOL	成功返回TRUE,失败返回FALSE
HANDLE	成功返回一个有效的句柄,失败一般返回NULL,少部分API会返回INVALID_HANDLE_VALUE
PVOID	成功返回一个有效的地址，失败返回NULL
LONG/DWORD	如果API返回值类型是这一种，那么说明这个API失败的原因是多种的，返回值将会是一个4字节的整型，这个整型值是一个错误代码.错误代码能够直观的说明导致函数失败的原因．

错误代码列表

大多数Windows的API的返回值类型是DWORD，这种API的返回值返回的是一个错误代码，Windows将所有的错误代码列出在一个表里，这个表就是错误代码表．这个表被放在WinError.h这个头文件里．在这个头文件中，Windows将一个错误代码使用一个宏来表示，宏一般都是以ERROR_开头．

获取最后一次错误代码

大多数Windows的API在被调用之后，除了使用返回值来表明API本身是否调用失败，还会将错误代码保存到线程本地存储区中(每个线程都有一个块内存,这块内存是线程独有的,线程和线程之间的对这块内存的操作互不影响,这块内存就叫做线程本地存储区)，通过GetLastError函数就能够获取到这个错误代码．

GetLastError的返回值是一个DWORD ，这个DWORD就是一个错误代码．

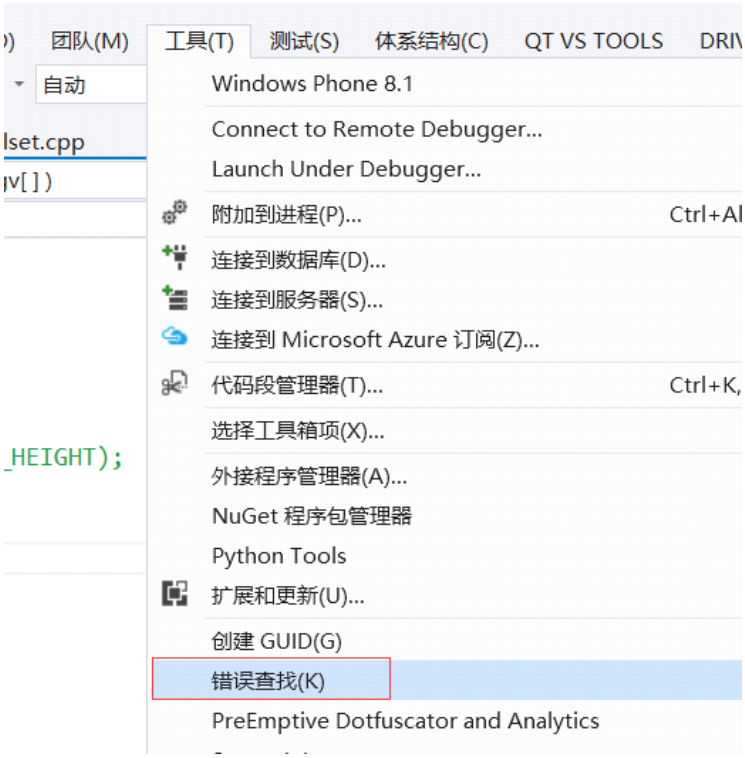
由于每个API调用之后都会设置这块内存，因此，想要通过GetLastError来获取API出错的具体原因，必须在API调用之后，调用其他API之前就调用GetLastError来获取．

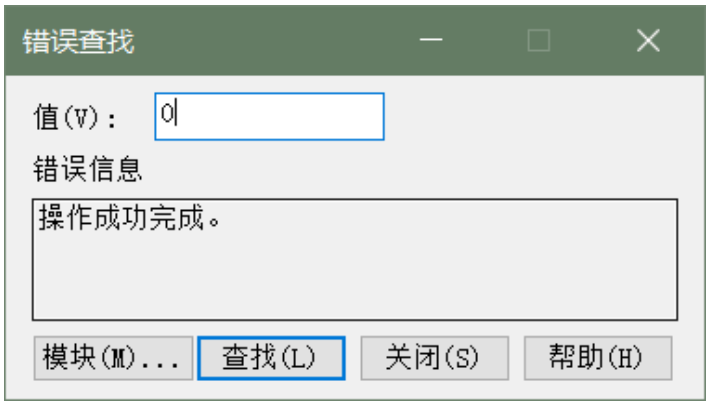
$err,hr

在VS调试程序时，可以在监视窗口中输入 $err,hr来实时获取最后一次错误代码．

错误查找程序

得到的错误代码也可以在VS自带的小程序中查找到对应的文本信息：





FormatMessage

Windows提供一个API,这个API可以获取整型的错误代码对应的错误信息(字符串). 这个函数就是FormatMessage.

这个函数的参数很多，但是如果只是想将一个错误码对应的文本获取出来，第一个参数传入：
FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_IGNORE_INSERTS 即可.

示例：

```
LPTSTR lpErrorString;
FormatMessage( FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_IGNORE_INSERTS ,
               NULL ,
               dwLastErrorCode , // 错误代码
               0 ,
               (LPTSTR) &lpErrorString , // 接收错误代码字符串的缓冲区
               0 ,
               NULL
               );

MessageBox( 0 , lpErrorString , 0 , 0 );

LocalFree(lpErrorString);  // 释放内存
```

设置最后一次错误代码

Windows提供SetLastError函数来设置最后一次错误代码，函数的编写者可以根据函数的的执行状况来设置最后一次错误代码.

通过这个函数 ，可设置一个已有的错误代码(以ERROR_开头的错误代码宏)，也可以设置自己定义的错误代码，但为了和系统定义的起冲突，因此，Windows对错误代码的格式作出如下定义:

位	31 - 30	29	28	27 - 16	15 - 0
内容	严重性	系统/客户	保留	Facility代码	错误代码
含义	0 : 成功 1 : 信息(提示) 2 : 警告 3 : 错误	0 : 系统定义 1 : 客户定义	必须为0	前256个值由微软保留	错误的代码

这个格式可以定义成如下结构体:

```
typedef struct _MS_ERRORCODE
{
    DWORD   status          : 2;
    DWORD   system          : 1;
    DWORD   reserve         : 1;
    DWORD   facility        : 12;
    DWORD   code            : 16;
}MS_ERRORCODE , *PMS_ERRORCODE;
```

学前知识-字符串处理

2017年1月22日 10:42

字符版本

ANSI字符串和Unicode字符串

字符版本对应的数据类型

ANSI字符串	**char**(c语言标准类型) 或 **CHAR**(由windows定义)
Unicode字符串	**wchar_t**((c语言标准类型)) 或 **WCHAR**(由windows定义)

Unicode字符串的定义方法

定义Unicode字符时，在字符前面加大写字母：L ，如 ：wchar_t ch = L'A';
定义Unicode字符串时，在字符串前面加大写字母：L，如 ：wchar_t wString=L"hello 15pb";

Windows定义的字符串相关类型

原型	Windows的定义	含义
char	CHAR	字符
char*	PSTR	字符串
const char*	PCSTR	常量字符串
wchar_t	WCHAR	宽字符
wchar_t*	PWSTR	宽字符串
const wchar_t*	PCWSTR	常量宽字符串

Windows定义的通用字符串类型

Windows的编译器会能够设置一个项目的字符编码集，如果在设置中使用了Unicode字符集，则Windows会定义一个UNICODE的宏，如果在设置中使用ANSI字符集,则不定义这个宏.

Windows提供了一种TCHAR的字符类型,这个类型可能表示CHAR类型，也可能表示WCHAR类型．实际上这种类型在检测项目使用了UNICODE字符编码集时，会被自动替换成WCHAR，否则被自动替换成CHAR.

使用这种类型，在改变项目的字符编码集时就不需要代码开发者逐个将CHAR和WCHAR互相替换了.

Windows API的版本

Windows的很多API都接收字符串的参数，或者内部有处理与字符串相关的代码 ，为了兼容两种版本的字符串，在这些API上，Windows提供了两种版本，一种是用于处理Unicode字符串的W版本的函数,W版本函数指的是,在API的名称后面有一个大写的字面:W. 另一版本是用于处理ANSI字符串的A版本函数(实际上,这种版本的函数内部会将ANSI的字符串转换成UNICODE字符串,再调用W版本的函数)

两种版本的字符串处理函数

对于char类型的字符串，一般有***strlen,strcpy*** 等函数，对于wchar_t类型的字符串，一般使用w开头的函数,如: ***wcslen*** ， ***wscpy*** 等函数 .

通用版本的字符串处理函数

既然有了通用字符类型 ，势必也会有通用字符串处理函数，这些通用字符串处理函数一般以**_t**开头，如***_tcslen***，***_tcscpy***等.

Windows的安全版字符串处理函数

Windows中，对于字符串的处理函数一般都采用了安全班 ，这种版本的函数有效的避免缓冲区写入越界的隐患. 这些函数一般都是以**_s**结尾，如: ***_cscpy_s*** ， ***_tcscat***.

C运行时库提供的安全版字符串处理函数

Windows提供的安全版函数只能在VS编译器中使用，当这些代码被其他编译器所编译时，其他编译器会识别不出. 但C提供的安全版处理函数在任何一个编译器中都会得到支持.这些函数一般会在**StrSave.h**中声明.

这些函数一般以**StringCch**开头，如字符串拷贝: *StringCchCopy* ， 字符串拼接: *StringCchCat*，格式化组合字符串: *StringCchPrintf*.

Cch指的是 Count of characters,即字符数，这系列的函数都是以字符个数作为单位来处理字符串的，例如，一个wchar_t类型的字符是两个字节，在处理wchar_t类型时，这系列函数使用的是字符个数，而不是字符字节数.

此外，还有另一种版本按字节来处理字符的，它们和前面所述的字符串处理函数的名字不同的是:它们以*StringCcb*开头.

字符串转换函数

函数名	作用
MultiByteTo**Wide**Char	将**多**字节字符串转换成**宽**字节字符串
WideCharTo**Multi**Byte	将**宽**字节字符串转换成**多**字节字符串

判断字符串的字符版本

Windows中，提供一个函数用于判断一个字符串是否是Unicode字符集的字符串，该函数的名称是：***IsTextUnicode*** .

由于多种原因，这个函数的判断并不精确，因此，这个函数可能会返回错误的结果.该函数测试的字节数越多，得到的结果越准确.

学前知识-内核对象

2017年1月22日 11:40

内核对象的种类
访问令牌对象
事件对象
互斥量对象
信号量对象
文件对象
文件映射对象
I/O完成端口对象
作业对象
邮件槽对象
管道对象
进程对象
线程对象
可等待的计时器对象
线程池工程对象

内核对象的由来

每个内核对象都只是一个内存块，它们由操作系统的内核所维护(创建,读写,释放)，并只能由操作系统所维护.

每个这样的内存块实际上是一个数据结构，这个数据结构中有大量的字段,用于描述一个内核对象的信息.

内核对象的数据保存在内核中,应用程序并不能够随意地直接地读写这些内存. 因为应用程序随意修改这些内存可能会引发系统崩溃.

为了能够让不能直接访问这些内存的应用程序也能修改和读取这些内核对象的数据，操作系统提供了一些API专门去操纵这些结构. 这些API会以最恰当的方式来操纵这些内核对象. 它们之间的操作也保证操作系统运行的安全性(保证不会无缘无故的崩溃,蓝屏).

这些操作内核对象的API用一个句柄来表示操作的是哪个内核对象(类似C++中的类成员函数用this指针来识别操作的是哪个类对象的内存).

句柄和进程

句柄的值并非内存地址，而是一个数组的索引.

系统在进程被创建的时候，会为每个进程创建一个句柄表，这个句柄表实际上就是一个数组. 每当我

们在进程中创建一个内核对象时(用户对象和GDI对象不是内核对象) ，系统会主动将这个内核对象在内核中的地址，访问掩码，标志等信息保存到数组的一个元素中(因此你可以推测得到,这个数组是一个结构体数组)，然后操作系统会把数组下标作为句柄来返回.

上述的言论仅仅是猜测，Windows并没有将句柄表的结构做出官方说明，因此，在这里关于句柄表所保存的元素的结构的说法并不是正确的.

每个进程都有自己的句柄表，而句柄只是一个句柄表的索引，因此，每个进程的句柄都是独立的，不能同时使用.

内核对象的引用次数

内核对象保存在内核的内存中. 为了节约内存,同一个内核对象无论被打开多少次,它在内存中都只有一份内存. 比如当多个程序打开同一个文件,内核只会在内存中创建一个内核对象用于表示一个被打开的文件. 当其中一个进程关闭内核对象时，内核并不会立即销毁内核对象所占用的内存. 因为其他进程有可能仍然还在使用这块内存.

为了记录还有多少进程在使用内核对象，当同一个内核对象每被打开一次，它的引用计数就被增加1 ，内核对象每被关闭一次，它的引用计数会建1 ，当引用计数小于等于0时，内核才会把这个对象所占用的内存释放.

一般地，内核对象由***CreateXXXX***系列函数所创建 ，内核对象由***CloseHandle***所关闭

句柄泄露

如果进程创建一个内核对象忘记关闭，那么在进程运行期间会造成句柄泄露，但是进程一旦被关闭，内核会将该进程的所有内核对象都关闭.

跨进程共享内核对象

进程和进程之间的句柄并不能直接使用，但是通过一些API还是能够使另一个进程使用本进程所创建出的句柄.

1. 父进程创建子进程时，通过允许继承句柄，让子进程得以使用父进程的句柄.
    a. 子进程在继承父进程的句柄时，并不是能够将父进程所有的句柄都继承. 父进程的内核对象在创建时，必须使用父进程必须指定一个***SECURITY_ATTRIBUTES***结构并对它进行初始化，然后将该结构的地址传递给特定的CreateXXXEx函数，这样创建出来的句柄才具备可继承性.
    代码如下:
    ```
    SECURITY_ATTRIBUTES sa={0};
    sa.nLength = sizeof(sa);
    sa.lpSecurityDescriptor = NULL;
    sa.bInheritHandle = TRUE; // 让创建出来的句柄句柄可继承性
    ```

```
HANDLE hMutex = CreateMutEx(&sa,FALSE,NULL);
```

2. 创建全局命名对象
   a. 通过在创建一个内核对象时，给这个内核对象命名来完成共享的目的.
3. 复制对象句柄
   a. DuplicateHandle

使用winddbg查看句柄信息

使用!process 命令查看进程的PID ：

```
kd> !process 0 0 handle.exe
PROCESS 875a9d40  SessionId: 1  Cid: 0b1c    Peb: 7ffd4000  ParentCid: 04c8
    DirBase: 3eb69320  ObjectTable: 9521a008  HandleCount:  13.
    Image: handle.exe
```

使用!handle 命令查看对应PID进程的句柄表

```
kd> !handle 0 0 0xb1c

Searching for Process with Cid == b1c
PROCESS 875a9d40  SessionId: 1  Cid: 0b1c    Peb: 7ffd4000  ParentCid: 04c8
    DirBase: 3eb69320  ObjectTable: 9521a008  HandleCount:  13.
    Image: handle.exe

Handle table at 9521a008 with 13 entries in use

0004: Object: 8f207e68  GrantedAccess: 00000003

0008: Object: 87d2af80  GrantedAccess: 00100020

000c: Object: 86bd93d0  GrantedAccess: 001f0003

0010: Object: 928f9a90  GrantedAccess: 00020019

0014: Object: 87a66038  GrantedAccess: 001f0001

0018: Object: 875a5d20  GrantedAccess: 001f0001

001c: Object: 9624d820  GrantedAccess: 00000001

0020: Object: 87c56d40  GrantedAccess: 00001400

0024: Object: 87bfd6f0  GrantedAccess: 001fffff

0028: Object: 87ace030  GrantedAccess: 001fffff

0034: Object: 83a8f528  GrantedAccess: 00000008

0038: Object: 95338ed0  GrantedAccess: 00000008

003c: Object: 984f82b8  GrantedAccess: 00000009
```

使用**!handle** 句柄索引 f PID 命令来查看句柄的详细的信息

WinDbg会输出上面这些信息， 在这些信息中， 我们可以看到，一个句柄它保存着一个内核对象在内核中的内存地址， 并且可以看到这个对象是什么类型的对象，如上图所示的这个对象，是一个进程对象，对象地址就是这个进程内核对象的地址。

如果是进程对象， 我们可以使用 *!process 对象地址 0* 命令来查看这个进程的信息：

```
kd> !process 86186a38  0
PROCESS 86186a38  SessionId: 1  Cid: 0ad0    Peb: 7ffdd000  ParentCid: 04c8
    DirBase: 3eb69500  ObjectTable: 95231c40  HandleCount:  61.
    Image: notepad.exe
```

可以看得出， 这个对象就是*notepad.exe*

再看一例：
查看索引为0x24的句柄信息：

```
Handle table at 96381968 with 14 entries in use

0004: Object: 8f207e68  GrantedAccess: 00000003

0008: Object: 85870110  GrantedAccess: 00100020

000c: Object: 8587ddf8  GrantedAccess: 001f0003

0010: Object: 89ab0600  GrantedAccess: 00020019

0014: Object: 86abe160  GrantedAccess: 001f0001

0018: Object: 85f1b290  GrantedAccess: 001f0001

001c: Object: 8a7aacc8  GrantedAccess: 00000001

0020: Object: 86186a38  GrantedAccess: 00001400

0024: Object: 858e58c8  GrantedAccess: 00120089

0028: Object: 85f8b868  GrantedAccess: 001fffff

002c: Object: 858f1d40  GrantedAccess: 001fffff

0038: Object: 8a73e030  GrantedAccess: 00000008

003c: Object: 8a798f10  GrantedAccess: 00000008

0040: Object: 8a780b08  GrantedAccess: 00000009


kd> !handle 24 f 0bc0

Searching for Process with Cid == bc0
PROCESS 87b41c90  SessionId: 1  Cid: 0bc0    Peb: 7ffd7000  ParentCid: 04c8
    DirBase: 3eb69320  ObjectTable: 96381968  HandleCount:  14.
    Image: handle.exe

Handle table at 96381968 with 14 entries in use

0024: Object: 858e58c8  GrantedAccess: 00120089 Entry: 8995d048
Object: 858e58c8  Type: (857dc378) File
    ObjectHeader: 858e58b0 (new version)
        HandleCount: 1  PointerCount: 1
        Directory Object: 00000000  Name: \Users\zr\Desktop\1.temp {HarddiskVolume1}
```

这个句柄保存的是一个文件对象 ， 这个文件对象的文件路径在上图中的最后一行已经给出， 如果想查看更详细的信息， 可以使用 *dt _FILE_OBJECT 文件对象地址* 命令来查看：

```
kd> dt _FILE_OBJECT 858e58c8
nt!_FILE_OBJECT
   +0x000 Type             : 0n5
   +0x002 Size             : 0n128
   +0x004 DeviceObject     : 0x869f6d38 _DEVICE_OBJECT
   +0x008 Vpb              : 0x869f5d18 _VPB
   +0x00c FsContext        : 0x89b7b8c0 Void
   +0x010 FsContext2       : 0x96301cb8 Void
   +0x014 SectionObjectPointer : 0x858e4400 _SECTION_OBJECT_POINTERS
   +0x018 PrivateCacheMap  : (null)
   +0x01c FinalStatus      : 0n0
   +0x020 RelatedFileObject : 0x85870110 _FILE_OBJECT
   +0x024 LockOperation    : 0 ''
   +0x025 DeletePending    : 0 ''
   +0x026 ReadAccess       : 0x1 ''
   +0x027 WriteAccess      : 0 ''
   +0x028 DeleteAccess     : 0 ''
   +0x029 SharedRead       : 0x1 ''
   +0x02a SharedWrite      : 0 ''
   +0x02b SharedDelete     : 0 ''
   +0x02c Flags            : 0x40042
   +0x030 FileName         : _UNICODE_STRING "\Users\zr\Desktop\1.temp"
   +0x038 CurrentByteOffset : _LARGE_INTEGER 0x0
   +0x040 Waiters          : 0
   +0x044 Busy             : 0
   +0x048 LastLock         : (null)
   +0x04c Lock             : _KEVENT
   +0x05c Event            : _KEVENT
   +0x06c CompletionContext : (null)
   +0x070 IrpListLock      : 0
   +0x074 IrpList          : _LIST_ENTRY [ 0x858e593c - 0x858e593c ]
   +0x07c FileObjectExtension : (null)
```

进程

2017年1月23日 10:46

Windows进程的类型

CUI进程 ： VS中的配置: ***/SUBSYSTEM:CONDOLE***
GUI进程 ： VS中的配置: ***/SUBSYSTEM:WINDOWS***

进程的启动函数

入口函数名	C/C++运行时库启动函数
main	mainCRTStartup
wmain	wmainCRTStartup
WinMain	WinMainCRTStartup
wWinMain	wWinMainCRTStartup

通过修改配置可以修改一个项目的的类型

C/C++运行时库启动函数的作用

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的指针
3. 对C/C++的全局变量进行初始化
4. 为所有全局和静态C++类对象调用构造函数
5. 对C的堆空间分配函数(*malloc*和*calloc*)和其他底层输入/输出例程使用的内存栈进行初始化
6. 调用程序的入口函数

进程命令行参数

1. 可使用*GetCommandLineW*来取得命令行参数
2. 可使用*CommandLineToArgW*来取得命令行参数的指针数组

进程环境块

每个进程都一个与它相关的环境块．环境块是进程内存中的一个内存块．一个进程可以有多个环境块,每个环境块都包含一组字符串，形式如下:

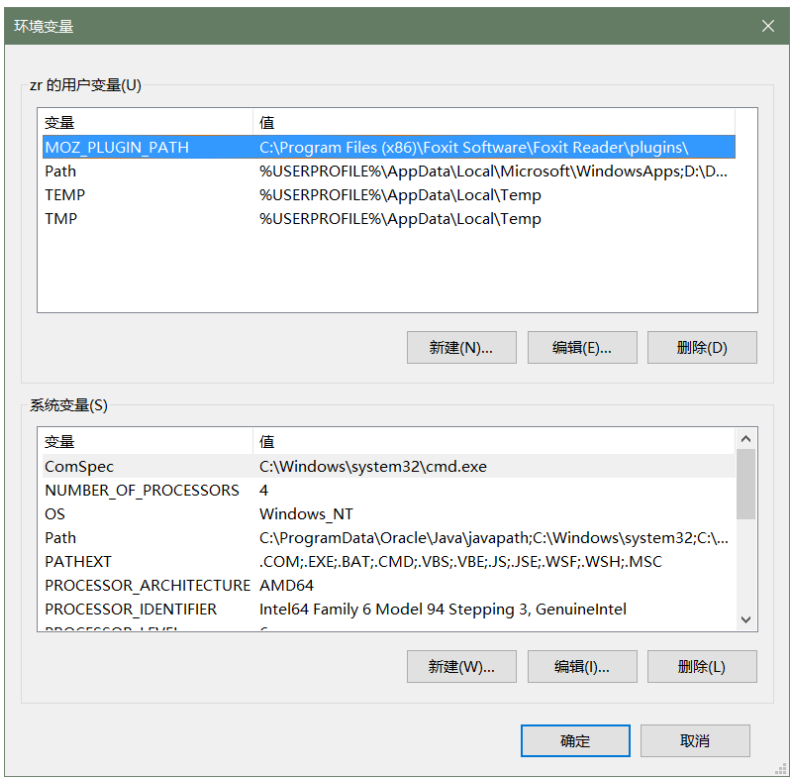
环境块名=环境块值\0

每个环境块中的字符串的第一部分是环境块的名字，后跟一个等号，等号后面就是环境块名字所拥有的值,这个值被称为环境块变量，一个环境名可以拥有多个变量，每个变量用分号来分隔．
如:

PATH=C:\windows;C:\windows\system32

环境块中的变量的值中的每个字符都是有意义的,包括空格．并且，每个环境变量中的值都是以字母顺序进行排序存储的．

每个进程环境变量可以在系统属性中可以看到



环境变量又分为用户环境变量和系统环境变量.

获取和设置环境变量

1. 通过***GetEnvironmentVariable***函数可获得某个环境变量以及它的值.
2. 通过***ExpandEnvironmentString***函数可以展开环境变量中具有可取代的字符串,如:
   %USERPROFILE%\My Documents
3. 通过***SetEnvironmentVariable***函数可设置,删除环境变量
   3.1 当系统中已经存在同名的环境变量时,则函数会修改这个环境变量中的值
   3.2 当不存在这个环境变量时, 系统会创建一个.
   3.3 当传入的环境变量的值为NULL时, 则函数会删除传入的环境变量.

进程的当前工作目录

系统在创建一个进程之后, 会将进程的二进制映像在磁盘中的路径保存起来, 一旦我们在代码中使用相对路径创建文件,或加载DLL时, 系统会自动使用进程的当前工作目录搜索这个文件或DLL.

进程的当前工作目录可以通过***GetCurrentDirectory***函数来获取

进程的当前工作目录可以通过***SetCurrentDirectory***函数来改变.

进程的当前工作目录也可以在创建一个进程时,在***CreateProcess***函数中的参数指定.

创建进程

通过***CreateProcess***函数来创建

系统创建进程的过程:

1. 系统创建一个内核对象(该内核对象不并代表进程本身)
2. 系统为新进程创建一个虚拟内存地址空间(4GB)
3. 系统将进程的可执行文件 , 进程需要到的DLL文件加载到进程的地址空间中.
4. 系统为新进程创建一个**线程内核对象**, 并将C/C++运行期启动函数作为线程的回调函数.
5. C/C++运行期启动函数在内部调用***WinMain/wWinMain/main/wmain***其中的一个函数.

CreatePreocess的参数

[参数1]LPCWSTR lpApplicationName :

1. 指定进程的可执行文件路径. 此参数可以为空.
2. 当这个参数为空的时候, [参数2]中就必须要传入具有可执行文件路径的命令行参数.
3. 当这个参数中传入了可执行文件路径时, 这个文件如果是相对目录, 则系统之会在当前调用CreateProcess函数的当前目录下搜索文件, 如果找不到文件,则创建进程失败. 并且,文件路径中的扩展名不能缺省.

[参数2]LPWSTR lpCommandLine :

1. 当CreatePress分析lpCommandLine 字符串时, 它将查看字符串中的第一个标记, 并假设该标记是想运行的可执行文件的名字。如果可执行文件的文件名没有扩展名, 便假设它的扩展名为.exe。 当CreatePress分析lpCommandLine的文件路径使一个相对目录路径时,会按下面的顺序搜索该可执行文件
   1.1 包含调用*CreateProcess*进程的.exe文件目录
   1.2 调用进程的当前目录
   1.3 Windows的系统目录

1.4 Windows目录
1.5 PATH环境变量中列出的目录.

2. 当进程被创建出来之后, ***WinMain/wWinMain/main/wmain***入口函数收到的的命令行参数就是通过这个形参传递进去的.

[参数3]LPSECURITY_ATTRIBUTES lpProcessAttributes : 进程安全描述符

[参数4]LPSECURITY_ATTRIBUTES lpThreadAttributes : 线程安全描述符

[参数5]BOOL bInheritHandles : 创建出的进程是否继承父进程的可继承句柄

[参数6]DWORD dwCreationFlags :
创建标志. 规定如何创建新进程. 这个参数可以用或运算符来讲多个标志组合起来,可以使用的标志以及含义如下:

DEBUG_PROCESS	以调试方式创建子进程
DEBUG_ONLU_THIS_PROCESS	和上一个一样,区别在于,子进程再创建子进程时,调试事件将不会得到响应
CREATE_SUSPENDED	创建新进程时, 新进程的主线程会被挂起.
CREATE_NEW_CONSOLE	为新进程创建一个新的控制台窗口,这个标志不能和DETACHED_PROCESS共同使用
DETACHED_PROCESS	不创建新的控制台窗口,和父进程共同使用一个
CREATE_NO_WINDOW	不要为新进程创建任何控制台窗口.

[参数7]LPVOID lpEnvironment : 进程环境变量块

[参数8]LPCWSTR lpCurrentDirectory : 可以传入一个目录来设置新进程的当前工作目录. 如果没有传入,则使用默认的(父进程的)目录作为新进程的当前工作目录

[参数9]LPSTARTUPINFOW lpStartupInfo : 指定新进程的启动信息.

[参数10]LPPROCESS_INFORMATION lpProcessInformation :
CreateProcess函数输出的信息,主要包括新进程的进程句柄和进程ID, 新进程的主线程句柄和线程ID.
进程ID和线程ID是一个可复用的值, 意思说, 进程ID在进程运行时是有效的, 而且系统会保证一个进程ID对应一个进程, 线程ID也是如此. 但进程退出后, 系统会回收这个ID, 当其它的新进程被创建时,系统会把这个ID和新进程进行关联.

终止进程

1. ***ExitProcess*** : 退出本进程,C/C++全局对象的析构函数将不会被调用.
2. 在***WinMain/wWinMain/main/wmain***函数中返回时, 进程自动退出
3. ***TerminateProcess*** : 终止其他进程.

在主函数自动返回退出进程是一种最可靠的进程终止方式. 只有在这种方式退出的进程,以下几点才能够被保证:

1. C++对象的析构函数才能被保证被调用.
2. 操作系统能正确释放该线程的堆栈使用的内存.
3. 系统将进程的退出代码设置为主函数的返回值.
4. 系统将进程内核对象的值递减1.

获取正在执行的进程列表

Windows中提供三种获取进程列表的方法:

1. 通过**performance data Helper**来遍历(使用艰难)
2. 通过***Process32Firest***和***Process32Next***来遍历
3. 通过***EnumProcesss***来枚举进程

获取进程所有模块

1. ***Module32First*** 和 ***Module32Next*** 遍历模块快照
2. ***EnumProcessModules*** 枚举进程的所有模块

进程间通讯

1. WM_COPYDATA
   只能在窗口程序上使用，控制台程序没有消息循环.
2. 邮槽
   使用ReadFile和WriteFile读写文件的形式来读写邮槽中的数据.
   邮槽是单向的，创建邮槽的一方只能往邮槽里写数据(服务端)，打开邮槽的一方只能往邮槽读数据(客户端.)

作业

2017年1月23日 15:28

作业的概念

通常，必须将一组进程当作单个实体来处理。例如，当让 Microsoft Developer Studio为你创建一个应用程序项目时，它会生成 cl.exe， cl.exe则必须生成其他的进程（比如编译器的各个函数传递）。如果用户想要永远停止该应用程序的创建，那么 Developer Studio必须能够终止cl.exe和它的所有子进程的运行。在 Windows中解决这个简单（和常见的）的问题是极其困难的，因为Windows并不维护进程之间的父/子关系。即使父进程已经终止运行，子进程仍然会继续运行。

当设计一个服务器时，也必须将一组进程作为单个进程组来处理。例如，客户机可能要求服务器执行一个应用程序（这可以成它自己的子应用程序），并给客户机返回其结果。由于可能有许多客户机与该服务器相连接，如果服务器能够限制客户机的要求，即用什么手段来防止任何一个客户机垄断它的所有资源，那么这是非常有用的。这些限制包括：可以分配给客户机请求的最大C P U时间，最小和最大的工作区的大小，防止客户机的应用程序关闭计算机，以及安全性限制等。

Microsoft Windoss 2000提供了一个新的作业内核对象，使你能够将进程组合在一起，并且创建一个“沙框”，以便限制进程能够进行的操作。最好将作业对象视为一个进程的容器。但是，创建包含单个进程的作业是有用的，因为这样一来，就可以对该进程加上通常情况下不能加的限制。

创建/获得作业对象

创建作业对象：***CreateJobObject***

打开作业对象：***OpenJobObject***

将进程加入到作业对象中

AssignProcessToJobObject

对作业进程的限制

1. 基本限制和扩展基本限制(用于防止作业中的进程垄断系统的资源)
2. 基本的UI限制(用于防止作业中的进程改变用于界面)
3. 安全性限制(用于防止作业中的进程访问保密资源(文件,注册表子关键字等)

通过函数SetInformationJobObject来进行限制

线程

2017年1月23日　　16:08

创建线程

Windows原生API ： ***CreateThread***
C/C++运行时库函数: ***_begintheadex***

终止线程执行

1. 线程函数返回(最好使用这种方法)
2. ***ExitThread*** ： 终止本线程(最好不要使用这种方法)
   如果编写的是C++的代码，呢么最后使用***_endthreadex***来结束当前线程，因为ExitThread将不会负责清理C++资源(c++类对象)
3. ***TerminateThread*** ： 终止指定线程(应该避免使用这种方法)
   当使用这种方法终止一个线程时，系统将不会释放线程占用的堆栈等资源，DLL也不会收到线程的通知(DllMain函数不会被调用)
4. 进程被关闭时,进程的所有线程都会被终止.

线程的性质

1. 每个线程都有自己独立的堆栈，在***CreateThread*** 被调用后，系统首先会创建一个线程内核对象． 之后就从进程的内存中分配一块出来作为线程的堆栈空间.

2. 每个线程都有一套CPU寄存器．这一样的一套CPU寄存器被称为线程上下文,也叫线程环境，这些CPU寄存器中的值在Windows中用一个结构体来表示，这个结构体的名字叫:***CONTEXT*** .

3. 每个被***CreateThread*** 函数创建出来的线程，其回调函数实际上是由***Kernel32.dll*** 中的***BaseThreadInitThunk***来调用，微软并未对这个函数进行文档化.

4. 线程被切换时，系统会将CPU线程当前的CPU寄存器保存到线程环境块中，当线程下一次被执行时，CPU会把线程环境块中的全部的寄存器的值设置到真正的CPU寄存器中.

获取进程和线程自己的标识

线程ID和进程ID，线程句柄和进程句柄经常会被使用．Windows提供了两组API来专门获取这些信息:

获取当前进程的ID　　: *GetCurrentProcessId*
获取当前进程的句柄 : *GetCurrentProcess*

获取当前线程的ID　　: *GetCurrentThreadId*
获取当前线程的句柄 : *GetCurrentThread*

获取当前继承的运行时间: *GetProcessTimes*
获取当前线程的运行时间: *GetThreadTimes*

需要注意的是，*GetCurrentProcess* 和 *GetCurrentThread* 得到的句柄都是伪句柄，这个句柄并不被保存到句柄表中，要想将这些伪句柄转换成真正的句柄需要用到***DuplicateHandle*** 来进行转换

线程调度

Windows内核每隔一段时间就会遍历当前所有的线程内核对象．并根据线程内核对象的优先级来选择一其中一个来执行，在执行前，Windows内核会先将线程的线程环境块设置到真实的寄存器中．然后就执行线程 ．线程被执行一段时间后，内核会将当前的线程挂起，再次遍历当前所有线程内核对象，准备再次执行另外的线程．

线程被挂起．重新执行的过程称为**线程切换**．线程环境块被写入到真实寄存器的过程被称为**上下文切换** ．

线程切换的时间间隔过长和多短都会影响系统的性能．通过***GetSystemTimeAdjustment*** 来获取到这个时间间隔，这个间隔大概是20毫秒．

线程的挂起和恢复

线程可以被挂起，并可以被挂起多次，且被挂起的线程不会被CPU执行到．

线程内核对象的结构体中，有一个字段被用于记录线程挂起计数，这个计数用于记录一个线程被挂起多少次．当这个挂起次数大于0时，内核在选择将要切换的线程时，不会将这个线程考虑在内．
除了被挂起的线程不会被切换，还有一些其他的线程也不会被切换，这些其他线程一般是只有当某种事件被触发时才会被切换．

挂起指定线程的函数 ：***SuspendThread***

恢复线程的函数　　：***ResumeThread***

这两个函数如果被调用成功，它们都会返回一个线程被挂起的次数．否则将会返回0xFFFFFFFF．
一个线程最多被挂起 ***MAXIMUM_SUSPEND_COUNT*** 次，一般为127次．

进程的挂起和恢复

Windows其实不存在挂起和恢复进程的概念，一般挂起一个进程时，可以把这个进程的所有线程都挂起．恢复一个进程，再把这个进程的所有线程都恢复一遍即可．

线程睡眠

线程可以放弃自己当前的CPU时间片，并在一段时间内不需要被执行(相当于挂起一段时间)．
Sleep函数可以实现这种功能． Sleep的调用需要注意一下几点:

1. 调用Sleep函数，使线程放弃属于它的时间片剩下的部分．
2. 睡眠的时间并不是一个准确的时间，比如调用函数时,可能仅仅是想睡眠100ms，但windows并不是一个实时操作系统，线程可能在长达数秒之后才能醒来．

3. 给***Sleep***传入0时，等于告诉内核放弃剩下的CPU时间片，强制切换到下一个线程．但是如果没有相同或者更高优先级的可调度线程时，内核有可能会再次调用刚刚***Sleep***的函数的线程．

切换线程

通过***SwitchToThread*** 函数来让内核自动切换到下一个线程， 这个函数的功能和Sleep(0)几乎一致，只不过Sleep(0)时，系统如果没有找到大于等于当前线程优先级的线程,则会切换回线程本身．但***SwitchToThread*** 不同，它能够切换到比当前线程优先级低的饥饿线程．
如果不存在任何一个饥饿线程，则该函数会返回***FALSE***，否则会返回一个非零值．

在超线程上切换到另一个线程

超线程处理器芯片内部有多个逻辑CPU ，这个CPU能够同时支持几条线程同步在执行代码．这些线程是内核无法直接管理和切换的．CPU使用***PAUSE*** 指令来让这些线程暂停．
操作系统将这样的指令定义成了一个宏:***YieldProcess*** ，通过这个宏就可以从CPU级别上切换线程．

线程的执行时间

GetTickCount64 ：

获取CPU开机到函数被调用那一刻的时钟毫秒数，缺点是线程中途会产生切换，在线程切换时，程序的代码不会得到执行,但计数器仍然还在计数，因此如果这个函数来计算某段代码的执行时间时并不精确．

GetThreadTiemes ：

获取线程的当前执行时间．
函数会返回四个时间:

创建时间	表示线程创建的时间
退出时间	表示线程退出时的时间,线程如果没有退出则该时间无效
内核时间	线程在内核中运行所消耗的总时间,以100纳秒为单位
用户时间	线程在用户态中运行所消耗的总时间，以100纳秒为单位

创建时间和内核时间都是以100纳秒为单位，且这个时间是从格林尼治时间1601年1月1日子夜开始计算的时间.

线程优先级

操作系统将每个线程分为 0~31 个等级，这些等级就是一个线程的优先级．在切换线程时内核会按照线程的优先级来决定是否要切换它．

数值越高，优先级越高．

饥饿线程

正在排队切换的线程中，当存在一个优先级为31的线程时，内核只会选择优先级为31的线程来执行．所以只要正在排队的线程中,存在一个优先级为31的线程，系统就不会给优先级0~30的线程分配CPU时间片．这些得不到CPU时间片的线程被称为**饥饿线程．**

线程调度算法

线程的调度算法并不是靠谱的，微软并没有出具书面文档说明在微软的所有操作系统上都保持同一套线程调度算法．

这是因为在设计系统内核时，并没有出现当今的众多复杂的需求．

因此，微软的NT内核在开发时，有一些特性是无法出具书面文档来说明它们是在任何一个系统中都可以使用同一套方法来使用的．微软公司可能会在未来

线程调度算法正是一个未文档化的特性．

正是由于这个特性，使得大多数代码都不会特别依赖线程的优先级调度算法．

优先级的分类

Windows为0~31级别的优先级提供了一个抽象层．这个抽象成使得用户在编写与线程优先级相关的代码时不必直接使用和线程调度算法的相关函数．

这个抽象层把0~31级的优先级映射成6大类：

优先级	描述
real-time(实时)	最高优先级
high(高)	被立即响应的优先级
Above normal(高于标准)	
Normal(标准)	99%的应用程序都是使用这个优先级
Below normal(低于标准)	
Idle(空闲)	一般只有一个,即系统页面清零线程(一个在空闲时将内存中所有闲置页面清零的线程)

进程优先级和线程优先级

线程的优先级越高┌┐那么就可以分占相对多的CPU时间片。每个进程都有相应的优先级，线程优先级决定它何时运行和占用CPU时间。最终的优先级共分32级┌┐是从0到31的数值,称为基本优先级别。

进程有进程的优先级，但是操作系统在进行调度时依然是以线程为单位进行调度的，因此，进程的优先级对进程本身而言是没有起作用的．但是，在创建线程时， 新线程的优先级会受到进程的优先级的影响，进程优先级和线程优先级二者决定了线程最终的优先级．

为了避免造成语义上的错误，未受到进程优先级影响的线程优先级被称为**相对线程优先级**．由相对线程优先级和进程优先级联合决定的才是真正的线程优先级．

一个线程的优先级通常由以下这个表交叉的值决定：

相对线程优先级		进	程	优	先	级
	idle	below normal	normal	above normal	high	real-time
Time-critical	15	15	15	15	15	31
highest	6	8	10	12	15	26
above normal	5	7	9	11	14	25
normal	4	6	8	10	13	24
below normal	3	5	7	9	12	23
lowest	2	4	6	8	11	22
idle	1	1	1	1	1	16

PS： *在这里必须要说明的是，线程的优先级(由进程优先级和相对线程优先级映射出来的值)由操作系统来决定，也就是说只有操作系统才能决定进程优先级和相对线程优先级映射出来的值是多少，这个映射过程可能会在不同版本的操作系统下产生变化．*

进程优先级获取和设置

创建进程时指定优先级：

在CreateProcess的第六个参数***dwCreationFlags*** 中可以传入以下宏来设置新进程的优先级

优先级	宏
real-time	***REALTIME_PRIORITYY_CLASS***
High	***HIGH_PRIORITY_CLASS***
Above normal	***ABOVE_NORMAL_PRIORITY_CLASS***
Normal	***NORMAL_PRIORITY_CLASS***
Below normal	***BELOW_NORMAL_PRIORITY_CLASS***
Idle	***IDLE_PRIORITY_CLASS***

对已经创建完成的进程，可以使用以下函数来获取,设置它们的优先级，这两个函数第一个参数接收的是要设置的进程的句柄，第二个参数就是上述表中的任意一个宏．

SetPriorityClass ： 设置进程优先级
GetPriorityClass ： 获取进程优先级

相对线程优先级的设置

相对线程的优先级也有对应的宏来表示：

优先级	宏
Time-critical	***THREAD_PRIORITY_TIME_CRITICAL***
Highest	***THREAD_PRIORITY_HIGHEST***
Above normal	***THREAD_PRIORITY_ABOVE_NORMAL***
Normal	***THREAD_PRIORITY_NORMAL***

Below normal	***THREAD_PRIORITY_BELOW_NORMAL***
Idle	***THREAD_PRIORITY_IDLE***

设置和获取线程的函数为:
SetThreadPriority : 设置相对线程优先级
~~***GetThreadPriority*** : 获取线程优先级~~

Windows并没有提供任何获取线程优先级的函数 , 这是微软公司故意为之的一件事.

线程优先级的动态调整

操作系统在切换线程时, 会自动调整线程的优先级. 但只会对优先级在 ***1~15*** 范围内的线程进行动态调整.

这个动态调整可以使用一组API来设置和获取:
SetThreadPriorityBoost : 设置一个线程是否允许优先级动态调整.
GetThreadPriorityBoost : 获取当前线程是否允许优先级动态调整.

操作系统还提供面向整个进程中的所有线程的API:
SetProcessPriorityBoost : 设置整个进程的所有线程是否允许优先级动态调整.
GetProcessPriorityBoost : 获取整个进程的所有线程是否允许优先级动态调整.

线程同步_原子操作

2017年1月25日　　22:22

原子操作系列函数

Windows中提供一系列函数来提供多线程同步的支持．这些函数内部会确保变量在同一时刻只会被一个线程更改它的值．

根据对变量的值修改的方式，Windows提供了不同的函数，这些函数有：

```
InterlockedIncrement( ); // 使变量自增1
InterlockedExchangeAdd(  ); // 使变量加上一个数(可以加上负数)
InterlockedExchange( ); // 将变量设置成另一个数

InterlockedDecrement( ); // 使变量自减1
InterlockedCompareExchange( );// 将目标变量和一个数比较，如果相同就将第三个数赋值到目标变量．

InterlockedXor( ) ;//按位异或
InterlockedAnd( ); //按位与
InterlockedXor( ); //按位或
```

原子操作的单向链表栈函数

函数	描述
InitializeSListHead	创建一个空栈
InterlockPushEntrySList	将一个元素入栈
InterlockPopEntrySList	将一个元素出栈
InterlockFlushSList	清空栈
QueryDepthSList	获取栈中元素个数

CPU高速缓存行

CPU在读取内存中的数据时，并不是一次只读取一个字节,或者4个字节．而是取回一个高速缓存行所能容纳的字节个数据.高速缓存行一般能够保存32字节,64字节,128字节．

CPU这样做是为了提高性能，一般的应用程序代码会对一组相邻的字节进行操作(结构体,数组,字符串等)，如果所有字节都在高速缓存行中，CPU就不用访问内存，因为CPU访问高速缓存行的速度比访问内存的速度要快得多．

但是现在的CPU一般都有多个核心，每个核心都有自己的高速缓存行，这导致了一些问题的产生：

1. CPU1读取一个字节，它会把剩下的一些字节也读取进高速缓存行中．
2. CPU2也读取同一个地址上的字节(多线程时会有这种事情发生)，CPU2也会把剩下的一些字节读取到CPU2的高速缓存行中．
3. CPU1修改高速缓存行中的某一个字节．但高速缓存行中被修改的数据并没有被写回到内存中．
4. CPU2读取高速缓存行中的某一个字节．这个读取到的字节本应该是被CPU1修改过的，但由于

CPU1和CPU2的高速缓存行不是同一个，因此,CPU2读取到的不是被修改过的新字节.

CPU的设计者当然非常清楚这个问题，他们的解决办法是:

当任意一个CPU核心修改了高速缓存行中的字节时，它会把修改过的高速缓存行的内存写回内存，并通知其他CPU核心，其他CPU核心接收到通知时，会将自己的高速缓存行中的数据作废，重新读取内存中的数据到高速缓存行中.

如果希望能得到更优秀的执行效率 ，就需要好好设计一个结构体中每个字段的位置与对齐粒度.
字段位置的设计:

1. **被读取次数比较多的字段和被写入次数比较多的字段的位置分开定义.**

   如:

```
struct TANK{
    DWORD     id;    // 大多数都是被读取
    DWORD     color; // 大多数都是被读取
    DWORD     x;     // 大多数都是被写入
    DWORD     y;     // 大多数都是被写入
}
```

2. 将结构体的对齐粒度设置为高速缓存行的字节数.

   1. 使用***GetLogicalProcessInformation*** 获取***SYSTEM_LOGICAL_PROCESS_INFORMATION*** 结构体数组，这个结构体中有一个***Cache***字段,这个字段的类型是一个***CACHE_DESCRIPTOR***结构体，在这个结构体中,***LineSize***字段表示CPU的高速缓存行的字节数.

   2. 使用***__declspec(align(#))*** 来设置结构体的对齐粒度，其中 ***#*** 应用一个数值来替代，如:

```
struct __declspec(align(64))  TANK{
    DWORD     id;    // 大多数都是被读取
    DWORD     color; // 大多数都是被读取
    DWORD     x;     // 大多数都是被写入
    DWORD     y;     // 大多数都是被写入
}
```

线程同步_临界区

2017年1月27日　　12:51

多个线程同时访问同一个全局变量时，原子操作只能够这个全局变量的读写是原子操作的，如果有多个变量，或者说一段代码，原子操作就支持不了．

临界区

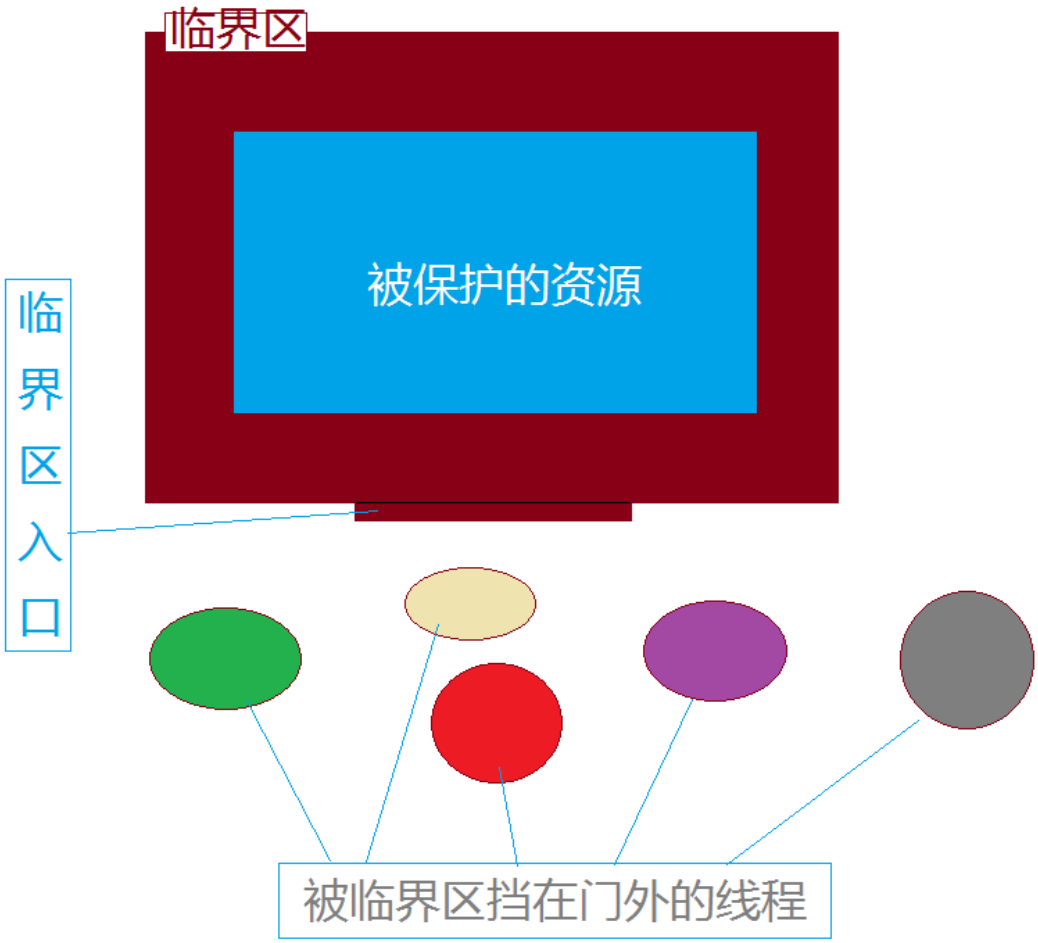
临界区是一小段代码，它能够让这一小段代码以原子操作的形式执行．这里的原子操作指的是：除了当前线程能够执行这一小段代码，其它线程都不会执行这段代码．

这样一来，这一小段代码如果在操作全局的资源，这些全局资源的原子性操作就能够得到保证．

临界区是很有必要的．在使用一些简单的数据结构的时候，使用原子操作系列函数就足够了，但是在操作像链表等数据结构时，使用原子操作系列函数会显得比较狼狈，使用临界区就能显得比较从容．

比如 ，当一个线程早搜索链表中的元素时，另一个线程在往同一个链表中插入数据．或者两个线程同时往同一链表中插入数据，那么两个新的节点的位置该如何确定?

需要支持原子操作的代码在没有使用任何多线程保护手段进行保护时，任何一个线程都可以随意的进行访问．
临界区的作用就是讲需要支持原子操作的代码保护起来，就相当于在这些代码的上下左右建了一堵墙，堵住任何线程的随意访问 ，临界区会在这四堵墙中的一个位置开一个门，当有多个线程同时要访问这个资源时，临界区会打开门，并只允许一个线程进入门内访问被保护起来的资源，其他线程在外面拥挤的排队．当这个线程完事了之后，临界区会把该线程送出，并迎接下一个线程．如此往复．



临界区的使用方法

1. 初始化临界区
   InitializeCriticalSection(PCRITICAL_SECTION pcs);
2. 进入临界区
   EnterCriticalSection(PCRITICAL_SECTION pcs);
3. 离开临界区
   ***LevelCriticalSection*(PCRITICAL_SECTION pcs);**
4. 删除临界区
   DeleteCriticalSection(PCRITICAL_SECTION pcs);

例子:

```
CRITICAL_SECTION cs; // 定义临界区变量
InitializeCriticalSection(&cs); // 初始化临界区

EnterCriticalSection(&cs); // 进入临界区

// do something

LevelCriticalSection(&cs); // 离开临界区
```

临界区函数细节

InitializeCriticalSection:

函数内部会初始化***CRITICALSECTION*** 结构体中的字段，这个结构体字段微软并没有公开，因此，我们并不直接对这个结构体赋值，而是调用初始化函数来更准确的进行初始化.

EnterCriticalSection:

函数会检查传入的临界区结构体变量中的某个字段，用以判断是否已经有线程进入了这个临界区,如果没有，则将传入的临界区结构体中的某个字段更新，用于记录已经有线程进入了这个临界区内部.

如果这个临界区的内部已经有一个线程在里面，则该函数会将当前的线程挂起，直到另一个线程调用了***LevelCriticalSection*** 函数离开临界区. 线程才会被唤醒. 线程被挂起后，这个线程就不会浪费任何CPU的时间.

但是线程被挂起时，需要从用户态进入到内核态，在内核态中才能把线程挂起. 从用户态到内核态大约需要1000个CPU周期.
因此微软把旋转锁和临界区融合到了一起. 在一定周期之内，临界区在用户态中等待其他线程调用***LevelCriticalSection*** 离开临界区. 如果超出这个周期，临界区才会将线程挂起.

这个等待周期可以通过***InitializeCriticalSectionAndSpinCount*** 来初始化临界区的同时设置这个等待周期. 这个等待周期设置为4000时，会得到比较好的性能.

LevelCriticalSection :

离开临界区，将传入的临界区结构体变量中的某个字段更新，表示线程已经从临界区中离开，使得正在等待临界区的线程得以进入临界区.

如果一个线程进入临界区后，一直不调用离开临界区的函数，那么其他线程就没有机会进入到临界区.

线程同步_事件

2017年1月29日 1:39

使用内核对象进行线程同步的原因

临界区并不是内核对象，因此，临界区不能跨进程使用.当多进程需要进程线程同步时(每个进程有多个线程,不同进程的线程之间需要进行同步)，就需要内核对象来支持.

内核对象的性能虽然比原子操作和临界区要差．但是内核对象的用途更广泛．

事件内核对象

事件内核对象和临界区不一样，临界区是为了确保多个线程中只允许一个线程在一个时间点上访问同一全局变量．

而事件对象是为了确保多个线程之间的顺序．加入有10个线程，它们被循环创建出来:

```
for(int i =0 ;i < 10 ; ++i)
    _beginthreadex(…);
```

但是,你并不能保证被创建出来的线程时按顺序被启动的．事件对象就可以用来确保这一点．当然,这只是事件对象的应用方向之一．

事件内核对象的概念很简单，一个事件内核对象只有两种状态

1. 触发状态(有信号状态)
2. 未触发状态(无信号状态)

触发状态实际上就是一个BOOL值．

当一个线程因为等待一个事件内核对象而被挂起时，如果事件处于触发状态，那么线程会从挂起状态中恢复执行．

如果一个线程需要等待一个事件内核对象，事件内核对象在当时是未触发状态的，则线程会被挂起，直到事件内核对象变成触发状态为止．

事件内核对象的创建

```
CreateEvent(
    LPSECURITY_ATTRIBUTES lpEventAttributes ,
    BOOL bManualReset ,
    BOOL bInitialState ,
    LPCWSTR lpName
)
```

[参数1] : 内核对象的安全描述符
[参数2] : ***TRUE*** ->事件内核对象的状态为人工设置，***FALSE*** ->事件内核对象的状态为自动设置
[参数3] : 事件内核对象的初始状态,***TRUE*** -> 触发状态，***FALSE*** -> 未触发状态
[参数4] : 事件内核对象的名字(可选),如果传入了一个名字，那么这个事件对象就是一个可以跨进程
使用的内核对象，其他进程可以通过***OpenEvent*** 并传入这个名字来打开同一个事件对象．

事件内核对象的使用:等待事件对象

通过***WaitForSingleObject*** 来等待事件对象的信号．在调用该函数时，会发生如下的事情:

1. 如果等待的事件对象当前是处于未触发状态的，则调用这个函数的线程会被挂起．直到等待的事件对象有信号为止．
2. 如果在等待的过程中超时了，线程会从挂起状态中恢复．
3. 如果在等待的过程中，事件对象有信号了，线程会从挂起状态中恢复．如果事件对象在创建时，在参数2中传入了FALSE(指定了自动设置状态)，则***WaitForSingleObject*** 在返回前会将事件对象设置为未触发状态．

事件内核对象的使用： 设置触发/未触发状态

将事件对象设置为触发状态:

SetEvent

PS:函数调用之后，事件对象的状态编程触发状态，调用 ***WaitForSingleObject*** 函数而进入挂起状态的函数将会被唤醒．

将事件对象设置为未触发状态:

ResetEvent

PS: 这个函数一般情况下都不会被调用，当创建内核对象时,在第二个参数中传入了***TRUE***(使用人工设置来设置事件对象的状态)时才会考虑调用这个函数．比如，事件内核对象的状态由于被设置成了人工设置,***WaitForSingleObject*** 函数不会自动设置事件对象的状态，因此,在***WaitForSingleObject*** 函数返回之后，需要通过代码调用***ResetEvent*** 来设置事件对象为未触发状态．

线程同步_信号量

2017年1月29日 1:39

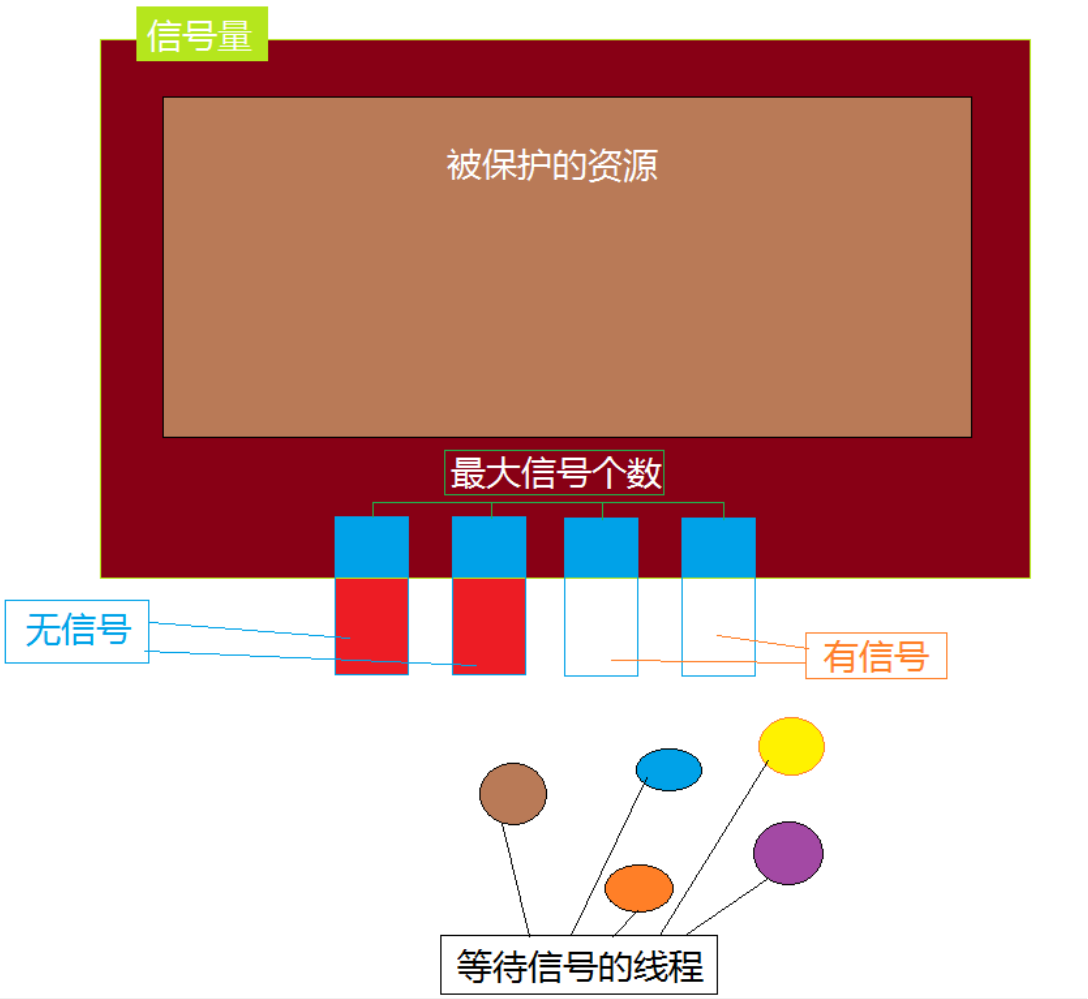
信号量的概念

信号量可以对保护的资源进行使用计数，之前谈及的线程同步手段中，只能在同一时刻允许一个线程访问被保护的资源．信号量不同之处在于，它能够允许多个线程在同一时刻同时访问被保护起来的资源．

为了支持这个特性，信号量内核对象中提供了两个重要的字段：

1. 最大信号个数
2. 当前信号个数

最大信号个数保存着最多能够允许多少个线程同时访问被保护的资源．
当前信号个数保存着，当前有多少线程在访问被保护的资源．



最大信号个数是在创建信号量时就必须要确定的．
当前使用计数的值也能够在创建信号量的时候设置，可以在参数2中进行设置．
当有线程进入信号量后，信号量的信号个数加1 ，当前信号个数大于等于最大信号个数时，信号量会拒绝再放入线程．线程只能进入挂起的等待状态．

当有一个线程离开信号量时，当前信号个数会被减1．

信号量内核对象的使用:信号量的创建

```
CreateSemaphoreW(
        LPSECURITY_ATTRIBUTES lpSemaphoreAttributes ,
        LONG lInitialCount ,
        LONG lMaximumCount ,
        LPCWSTR lpName
        );
```

[参数1] : 内核对象的安全描述符
[参数2] : 信号量的信号个数(见附注1)
[参数3] : 信号量的最大使用计数
[参数4] : 信号量的名字(可选),传入名字后,信号量可以跨进程使用,可以通过***OpenSemaphore***来打开一个已经存在的信号量.

附注1: 参数2比较奇怪, 这个参数表示有信号的个数, 如果传入了0 , 任何等待这个信号量的线程都会进入到挂起状态, 因为这个信号量没有信号. 因此, 不要将这个参数和信号计数弄混.

信号量内核对象的使用:等待信号

通过***WaitForSingleObject*** 来等待事件对象的信号. 在调用该函数时, 会发生如下的事情:

1. 如果等待的信号量当前信号个数已经满了, 则调用这个函数的线程会被挂起. 直到当前信号个数有空位为止.
2. 如果在等待的过程中超时了, 线程会从挂起状态中恢复.
3. 如果在等待的过程中信号量的当前信号个数有空位了(当前信号个数 小于 最大信号个数时), 线程会从挂起状态中恢复. 并将信号量的当前信号个数加1.
4. 当有线程调用了***ReleaseSemaphore*** 释放了信号量 ,则信号量的当前信号个数减1 , 并且在减1的时候, 系统会保证信号量的当前信号个数不会小于0.

信号量内核对象的使用: 释放信号量

```
ReleaseSemaphore(
       HANDLE hSemaphore ,
       LONG lReleaseCount ,
       LPLONG lpPreviousCount
       );
```

[参数1] : 要被释放的信号量对象
[参数2] : 释放的次数,如果释放的次数超出了最大使用计数, 则信号量不会被使用,并且函数会返回FALSE
[参数3] : 信号量被释放前的当前使用计数(可选,可以传NULL)

函数会将信号量的当前信号个数减少. 减少的数量可以从参数2中指定, 参数3会输出信号量释放前的当前信号个数

线程同步_互斥体

2017年1月29日　　1:39

互斥体和临界区的功能几乎相同，是为了确保只有一个线程能对资源进行访问.

互斥体内核对象的使用:创建互斥体

```
CreateMutexW(
        LPSECURITY_ATTRIBUTES lpMutexAttributes ,
        BOOL bInitialOwner ,
        LPCWSTR lpName
        );
```

[参数1] : 安全描述符
[参数2] : 创建时是否有信号
[参数3] : 内核对象名称

互斥体内核对象的使用:释放互斥体

ReleaseMutex : 释放互斥体

互斥体和临界区不一样 , 互斥体会在内部保存线程的ID, 如果线程在释放互斥体之前被终止了, 系统会将互斥体释放.

内核对象等待函数

2017年1月30日 1:45

WaitForInputIdle

等待一个进程进入可接收输入事件.

MsgWiatForMultipleObject

功能和***WiatForMultipObject*** 函数几乎相同.
区别在于，当产生窗口消息时，线程也会被唤醒,并从该函数返回.

WaitForDebugEvent

当有调试事件产生时，线程会被唤醒.

SignalObjectAndWiat

将一个内核对象设置为有信号，并等待另一个内核对象.
函数内部会根据传入的内核对象自动识别出它们的类型(事件对象,信号量对象,互斥体对象),并调用它们的设置信号的函数(***SetEvent,ReleaseSemaphore,ReleaseMutex***).

设备

2017年1月29日 1:39

Windows 中的设备

```
CreateFileW(LPCWSTR lpFileName ,
            DWORD dwDesiredAccess ,
            DWORD dwShareMode ,
            LPSECURITY_ATTRIBUTES lpSecurityAttributes ,
            DWORD dwCreationDisposition ,
            DWORD dwFlagsAndAttributes ,
            HANDLE hTemplateFile
            );
```

[参数1] : 设备名
[参数2] : 设备的访问权限
[参数3] : 设备的共享权限
[参数4] : 设备内核对象的安全描述符
[参数5] : 设备的创建标志
[参数6] : 设备的属性
[参数7] : 设备的全部属性的模板句柄

设备	打开设备的函数
文件	CreateFile , 第一个参数传文件路径
目录	CreateFile, 第一个参数传目录路径,并使用FILE_FLAG_BACKUP_SEMANTICS
逻辑磁盘驱动器	CreateFile, 第一个参数必须是"\\.\x:" 其中,x表示盘符
物理磁盘驱动器	CreateFile, 第一个参数是"\\.\\PHYSICALDRIVEx" , 其中x表示物理驱动器号
串口	CreateFile, 第一个参数是"COMx" x是一个数字
并口	CreateFile, 第一个参数是"LPTx" x是一个数字
邮件槽服务器	CreateFile, 第一个参数是"\\.\mailslot\邮槽名称"
邮件槽客户端	CreateFile, 第一个参数是"\\服务器名或IP地址\mailslot\邮件槽名称"
命名管道服务器	CreateFile, 第一个参数是"\\.\pipe\命名管道名称"
命名管道客户端	CreateFile, 第一个参数是"\\服务器名或IP地址\pipe\命名管道名称"
匿名管道	CretePipe
套接字	Socket, accpet
控制台	CreateConsoleScreenbuffer 或 GetStdHandle

同步设备I/O

2017年1月30日 1:57

设备一般都是通过***CreateFile***进行打开.

设备的操作只有两种:

1. 将数据写入到设备
2. 从设备读取出数据

这两种操作对应着两个API:

1. ***WriteFile***
2. ***ReadFile***

在读写设备文件时，Windows提供了两种方式，默认情况下，Windows是以同步的方式来完成文件数据的读写.即:只有等数据完全写入到设备,或完全从设备中读取出来后，函数才能返回.

异步设备I/O

2017年1月30日 2:00

当从设备中读取的数据，或者写入到设备中的数据都比较小的时候，使用同步设备I/O并不影响程序的运行．但是当读写的数据非常大．或者读写的时间耗时久，那么，应用程序可能会被`卡死`．

这个时候就需要用到异步I/O．

当使用异步I/O时，在读写一个设备的时候，ReadFile 或 WriteFile 只是将一个读写请求发送到内核．随后立即从函数中返回．

因此，在同步I/O中,只要ReadFile或WriteFile调用完毕，我们就能够知道数据有没有被读写完毕．使用异步I/O时，我们就不能仅仅从函数是否调用完毕来评判数据有没有被读写完毕了．

异步I/O基础

默认情况下，Windows使用同步I/O来完成设备数据的读写．如果要想启用异步I/O，则需要在创建设备文件的时候,在第6个参数中,传入***FILE_FLAG_OVERLAPPED*** 标志．

传入这个标志以后，在调用***ReadFile*** 或***WriteFile*** 时，需要传入***OVERLAPPED 结构体变量***.
这是因为同步I/O时， ***ReadFile*** 或***WriteFile*** 使用文件指针来定位文件的读写位置，但启用异步I/O后，已经不存在文件读写位置．而且在同步I/O时，函数调用完毕，数据的读写就完成了，启用异步I/O后，函数调用完毕，数据的读写可能还没有完成．

因此．***ReadFile*** 或***WriteFile*** 需要记录本次数据操作的读写位置在哪，读写操作完成之后,读写了多少字节，有无发生错误等信息．而这些信息就需要保存在***OVERLAPPED*** 结构中．

```
typedef struct _OVERLAPPED {
    ULONG_PTR Internal;          // 保存错误码(输出)
    ULONG_PTR InternalHigh;      // 保存读写了多少字节(输出)
    union {
        struct {
            DWORD Offset;        // 读写位置(低32位)(输入)
            DWORD OffsetHigh;    // 读写位置(高32位)(输入)
        } DUMMYSTRUCTNAME;
        PVOID Pointer;           // 联合体中的一员,也表示读写位置
    } DUMMYUNIONNAME;

    HANDLE  hEvent;              // 事件对象(输入)
} OVERLAPPED, *LPOVERLAPPED;
```

接收异步I/O完成通知

同步I/O时，***ReadFile*** 或***WriteFile*** 调用完毕就可以知道数据已经读写完毕了，异步I/O时，需要用其他的方法来获知数据有无读写完毕，方法有以下4种

1. 触发设备内核对象
2. 触发事件内核对象

3. 使用可提醒I/O
4. 使用I/O完成端口

接收异步I/O的完成通知

2017年1月30日　　2:21

一共有四种方式可以知道数据有无读写完毕

1. 触发设备内核对象
2. 触发事件内核对象
3. 使用可提醒I/O
4. 使用I/O完成端口

1. 触发设备内核对象

当异步I/O读写完数据之后，设备内核对象会被设置成有信号状态，这种方法没什么用.
当同时发出两个I/O请求时，只要其中一个请求被完成. 则设备内核对象会被设置为有信号状态，这样一来，就无法判断出是哪一个I/O请求被完成.

2. 触发事件内核对象

在OVERLAPPED结构体重，有一个hEvent的结构，当我们在提交一个I/O任务时，如果我们创建一个事件对象，并赋值到这个结构体，那么，I/O任务被完成时，系统会将这个事件对象设置为有信号状态.

因此，我们只需要等待这个事件对象，就能够知道I/O任务在什么时候完成了.

需要一次性提交多个I/O任务时，可以为每一个I/O任务创建事件对象，这样一来，哪个事件对象被触发了，就说明哪个I/O任务被完成了.

3. 可提醒I/O

在Windows 中，ReadFile和WriteFile还有另外一个版本的函数，那就是***ReadFileEx*** 和 ***WriteFileEx,*** 这两个增强版的函数中，都是添加了一个参数，这个参数是一个函数指针.

当I/O任务完成之后，系统内部会在一段时间内调用这个函数指针. 但是系统并不是无条件调用该函数指针的.

每个线程都有一个APC队列(异步过程调用队列)，这个队列实际上就是一个数组，这个数组中的每一个元素都是一个函数的地址. 线程被挂起时，这些函数会被系统调用，系统会从APC队列中移除已经被调用过的函数.

线程挂起时，默认是不执行APC队列中的函数的，想要在线程挂起时让系统自动调用APC队列中的函数，必须使用以下的函数来挂起线程:

1. ***SleepEx***

2. ***WaitForSignalObjectEx***
3. ***WaitForMultipleObjectEx***
4. ***SignalObjectAndWait***
5. ***GetQueuedCompletionStatusEx***
6. ***MsgWaitMultipelObjectEx***

这些函数中，有一个参数 ***BOOL bAlertble*** 这个参数就用于指定是否调用APC队列中的函数的.

异步I/O中的可提醒I/O通知方式实际的原理是这样的: 当I/O任务完成后，系统会将完成函数的地址发送到线程APC队列(通过***QueueUserAPC*** 发送)，然后当线程被指定的函数挂起后，可提醒I/O的完成函数才会被调用.

4. 完成端口

事件对象和可提醒IO已经能够让我们及时处理完成的IO任务. 但是这两种办法并不是最有效率的.

事件对象模型中，每处理一个IO任务就需要创建一个线程. 这就好比一个银行，每处理一个客户就新建一个窗口.用户走了就把窗口砸了，这是非常不合理的. 线程的创建和销毁会耗费系统非常多的资源，而且线程多了，就会产生线程切换. 这样一来，线程越多，效率反而越低.

可提醒IO虽然好用，但如果IO任务多了，那么. 线程的APC队列中的元素就会变多，APC队列中函数的地调用无疑会将整个线程都拖慢.

完成端口就是一种综合了两者的有点，避开了两者的缺点的优秀IO通知模型.

完成端口的运行原理比较复杂.

完成端口的创建函数

```
CreateIoCompletionPort(
        HANDLE FileHandle ,
        HANDLE ExistingCompletionPort ,
        ULONG_PTR CompletionKey ,
        DWORD NumberOfConcurrentThreads
        );
```

[参数1] : 和完成端口关联的文件句柄
[参数2] : **完成端口句柄**
[参数3] : 完成键
[参数4] : 完成端口的最大的同时运行的线程个数
[返回值]: **完成端口句柄**

这个函数的使用比较复杂，比如参数2中的完成端口句柄，这个参数需要使用 ***CreateIoCompletionPort*** 函数本身返回的句柄，但是在调用这个函数的之时，没有存在完成端口句柄. 因此,这个函数实际上需要调用两次.

第一次调用的时候是为了创建出一个完成端口,并配置完成端口的属性.
第二次调用的时候是将已经创建好的完成端口和已经存在的文件对象进行关联.

完成端口运行机制

在第二次调用***CreateIoCompletionPort*** 后(将文件对象和完成端口对象进行关联后)，所有发生在文件对象上的IO任务都会被系统发送到完成端口的队列中.

比如，当我们***ReadFile*** 一个文件对象时，在使用事件对象模型时 ，系统会在IO任务完成后触发 ***OVERLAPPED*** 中的***hEvent*** 事件对象．我们使用***WaitForSignalObject*** 函数进行等待的线程会别唤醒．

在使用可提醒IO的完成函数中，系统会在IO任务完成后,将完成函数发送到线程的APC队列．APC队列会在线程挂起时会被调用，我们的完成函数也就是在APC队列中的函数被调用时被调用的.并且在调用函数时，将***OVERLAPPED*** 结构体变量的地址通过形参传递进函数．

在使用完成端口时，系统会检测我们的文件句柄有没有和完成端口进行关联.如果是有关联的，那么系统会在IO任务完成后，将**OVERLAPPED**结构体和一些其它信息发送到完成端口的队列中(可以使用 ***PostQueuedCompletionStatus*** 函数来发送)．当我们在一个线程中使用 ***GetQueuedCompletionStatus*** 时,这个函数实际上就是在等待完成端口中的队列是否有值，如果没有值，那么调用这个函数的线程会被挂起．如果队列中有值了，这个线程会被唤醒，并且 ***GetQueuedCompletionStatus*** 函数会将已经完成的IO任务的***OVERLAPPED*** 结构体变量的地址获取出来．

至于***GetQueuedCompletionStatus*** 的调用地点，就必须是在线程当中，因为这个函数会因为完成队列中没有值时将线程挂起．

而调用***GetQueuedCompletionStatus*** 函数的线程如果只有一个，那么完成端口的效率并不会太好．因为完成队列中如果同时保存了几百个值，那么只有一个线程在调用***GetQueuedCompletionStatus*** 函数时，线程就会将队列中的值一个接着一个地进行处理．

但如果调用***GetQueuedCompletionStatus*** 函数的线程有十个，当完成队列中存在几百个值，那么十个线程会同时从完成队列中取出值来进行处理，这样一来，就能每次处理10个．

但是前面说了，线程并非越多越好．线程多了，反而可能将效率拖得更低．微软官方的文档有说过，线程的个数，最好是CPU个数的2倍．这是能够达到最优效率的线程个数了．

GetQueuedCompletionStatus 的细节

1. 函数最重要的作用就是检查完成队列，如果队列没有值，则将线程挂起，有过有值,则从完成队列中取出一个值，并从函数中返回．
2. 在获取到完成队列中的值时，完成端口的内部会唤醒一个线程来对这个值来进行处理，唤醒的原则并不是随机的，而是以后入先出的方式来进行唤醒，比如创建了4个线程来调用 ***GetQueuedCompletionStatus*** 函数,函数内部会维护一个等待线程的队列．***GetQueuedCompletionStatus*** 函数会将调用者线程的线程标识符保存到等待线程的队列中，当完成队列中出现一个值的时候，***GetQueuedCompletionStatus*** 函数会将最后一个被唤醒的线程唤醒．并使用它来处理这个值，也就是，当完成队列中的值出现得特别慢时，这4个线程中只有一个线程在运行，其他线程会一直进入挂起状态．完成端口内部还会使用一定的算法，将处于挂起状态时间比较长的线程从内存交换到硬盘中．

线程池

2017年1月30日　　20:03

线程池的使用方式有4中：

异步调用函数	和普通线程一样的简单的使用方式	TrySubmitThreadpoolCallback
间隔性的调用函数	和定时器中的回调函数的使用方式差不多	CreateThreadpoolTimer和SetThreadpoolTimer
在内核对象触发时调用函数	创建线程来等待一个事件对象的响应,这个等待的线程由线程池来负责管理	CreateThreadpoolWait和SetThreadpoolWait
在异步I/O完成时调用函数	在调用ReadFile或WriteFile异步IO时，创建一个线程来等待IO任务的完成,这个等待的线程由线程池来负责管理	CreateThreadpoolIo和StartThreadpoolIo

定制线程池

线程池的定制在于线程池环境.
线程池环境中，有最小线程数，最大线程数两个选项可以配置.
配置的函数如下：

InitializeThreadpoolEnviroment	初始化一个线程池环境
SetThreadpoolThreadMinimum	设置最小线程数
SetThreadpoolThreadMaximum	设置最大线程数
SetThreadpoolCallbackPool	将线程池环境与线程池关联

权限管理与UAC

2017年2月1日　　21:26

安全等级的概念

在计算机中，有一个安全等级的概念.
安全等级用于核定软件(包括操作系统)的安全性.

安全等级分为以下几个级别:

等级	说明
A1	可检验的设计
B3	安全域
B2	结构化的保护
B1	分类的安全保护
C2	受控制的访问保护
C1	自主访问保护(已不再使用)
D	最少的保护

当前没有一个操作系统满足A1或"可检验的设计"这个等级，只有极少数的操作系统获得了B类等级.

对于一个通用的操作系统，Windows Nt4的操作系统获得了C2等级.

C2安全等级的要求

要获得C2安全等级，则操作系统需要满足一下的要求:

1. **安全的登录措施**
   要求用户可以被唯一标识，而且只有当它们被通过某种方式认证身份后(用户名密码,指纹认证等),才可以被授予对该计算机的访问权.
2. **自主的访问控制**
   这使得资源(比如文件)的所拥有者能够决定谁可以访问该资源，以及能够访问该资源的用户可以对资源做些什么(读写执行).
3. **安全审计**
   提供相应的能力来检测和记录与安全相关的事件,或提供相应的能力来检测和记录任何创建,访问和删除系统资源的意图. 登录标识符记录了所有用户的身份，从而使得跟踪谁在执行一个未授权的动作比较容易.
4. **对象重用保护**
   防止用户看到其他用户已经删除的数据. 或者访问到其他用户原先使用过后来又释放的内存.对象重用保护设施一般的做法是，在将对象(包括文件和内存)分配给其他用户以前,先对对象进行初始化.
5. **可信路径功能**
   防止木马程序在用户登录的时候能够截取到用户名和用户口令.在Windows中，Windows以Ctrl+Alt+Del登录注意序列(Ctrl+Alt+Del登录注意序列是一个名词). 非特权应用不能截取到此序列. 该键击序列也称为安全注意序列(SAS)，它总是显示一个由系统控制的安全屏幕,或者是登录屏幕.
6. **可信设施管理**
   它要求针对各种管理功能有单独的账户角色作为支持. 例如,针对管理工作(管理员),负责计算机备份的用户和标准用户分别提供单独的账户.

Windows实现C2安全等级的组件及组件的详细功能

Windows通过它的安全子系统和相关组件满足了上述要求.
它的相关组件以及组件所完成的功能如下:

1. **安全引用监视器(SRM)**
   组件路径: *%SystemRoot%\system32\Ntoskrnl.exe*
   组建功能:
   1.1 创建访问令牌数据结构来表示一个安全环境
   1.2 执行对象的安全访问检查
   1.3 管理特权(用户的权限)
   1.4 生成所有的结果安全审计消息

2. **本地安全权威子系统**
   组件路径: *%SystemRoot%\system32\Lsass.exe*
   组建功能:
   2.1 本地系统安全策略(允许哪些用户登录到本地机器,密码策略,授予用户和用户组的特权,系统安全审计设置)
   2.2 用户认证
   2.3 发送安全审计消息到事件日志中.

3. **LSASS策略数据库**
   数据库路径: 注册表的*HKLM\SECURITY* 下
   这个数据存储一下类型的信息:
   3.1 可信任的的域
   3.2 可以访问系统的用户
   3.3 访问系统的方式(交互式登录,网络登录,服务登录)
   3.4 用户的特权信息
   3.5 执行的安全审计类型

4. **安全账户管理器(SAM)服务**
   服务路径: *%SystemRoot%\system32\Samsrv.dll*
   服务功能: 管理数据库. 所管理的数据库包含了本地机器上已经定义的用户名名和用户组. 这个DLL被LSASS进程所加载.

5. **SAM数据库**
   数据库路径: 注册表的*HKLM\SAM*下
   被Samsrv.dll所管理的数据库. 数据库的存储的信息有:
   已定义的用户和用户组. 以及它们的口令和其它属性

6. **活动目录**
   服务路径:*%SystemRoot%\system32\Ntdsa.dll*
   该DLL被LSASS进程所加载.
   是一个目录服务 , 该服务包含一个数据库, 这个数据用户存储关于域中对象的信息. 这里的域更多的是指同一局域网中一组计算机构成的. 域中的计算机被当做单个实体来管理. 活动目录存储了有关该域中的对象的信息. 对象包括用户, 组合计算机.
   域用户和组的口令信息和特权被存储放在活动目录中.

7. **认证包**
   认证包的功能由运行在LSASS进程和客户进程环境中的DLL实现.
   这些DLL主要实现了Windows的认证策略, 负责认证一个用户. 其做法是: 检查一个给定的用户名和口令是否匹配, 如果匹配的话, 则向LSASS返回有关用户安全身份的细节信息, LSASS利用这些信息来生成一个令牌.

8. **交互式登录管理器(Winlogon)**
   组件路径 : *%SystemRoot%\system32\Winlogo.exe*
   组件功能: 负责响应SAS(安全键击序列), 管理交互式登录会话. 例如, 当用户登录的时候, Winlogon创建用户的第一个进程.

9. **登录用户界面(LogoUI)**
   组件路径：*%SystemRoot%\system32\LogonUI.exe*
   组件功能：它负责显示用户界面(用户名和密码输入界面)，并在此界面中认证用户名和密码在系统上的身份．LogonUI通过各种不同的方法，向各个凭证提供者查询用户的登录凭证．

10. **凭证提供者(CP)**
    获取用户登录信息的COM对象．运行在LogonUI进程．它用户获取用户的名称和口令，这些口令的获取不仅仅是可以通过键盘输入，还有指纹,智能卡的PIN码等多种途径．

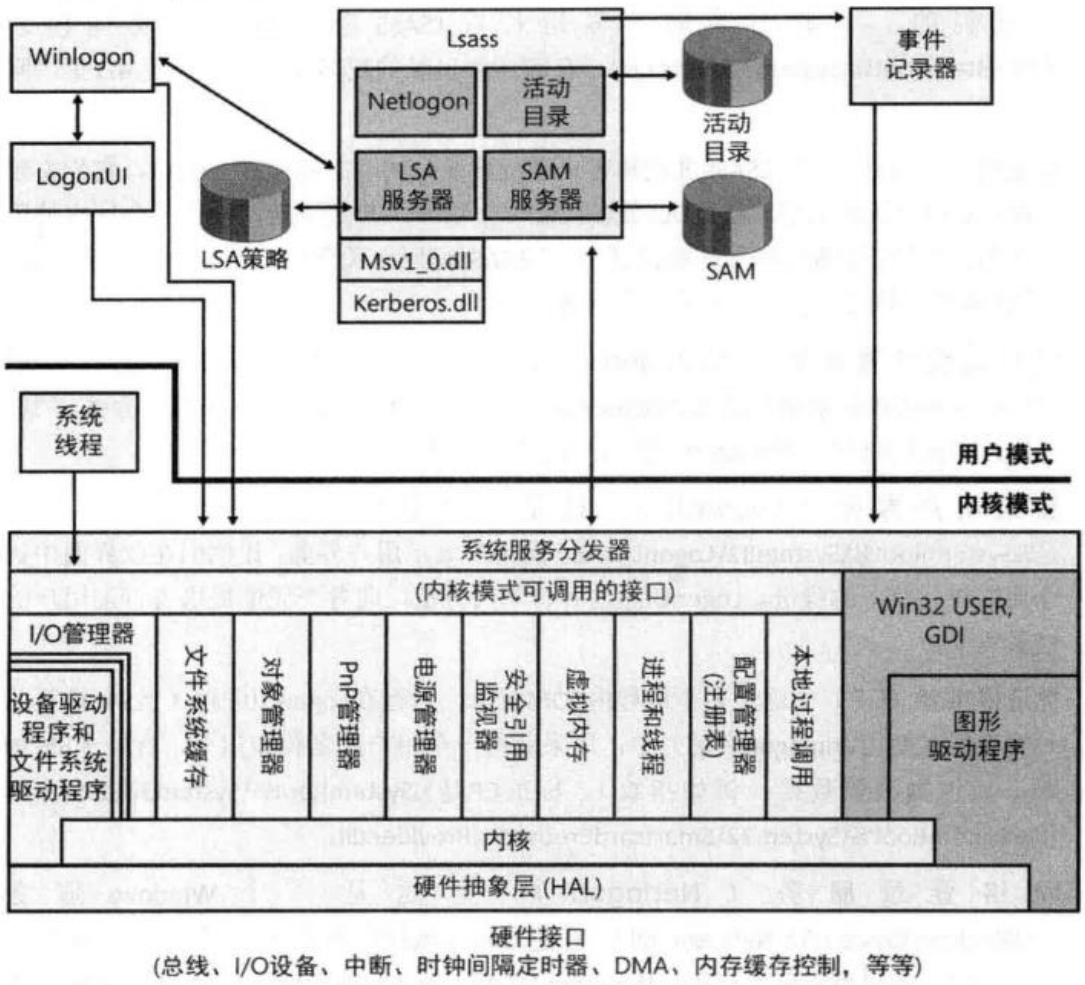
11. **网络登录服务(Netlogon)**
    服务路径：*%SystemRoot%\system32\Netlogon.dll*
    服务功能：建立起与域控制器之间的安全通道，也用于活动目录中的登录．

12. **内核安全设备驱动程序(KSecDD)**
    组件路径：*%SystemRoot%\system32\Drivers\Ksecdd.sys*
    组件功能：提供与用户模式下的LSASS进行通信．

13. **AppLocker机制**
    一种机制,这种机制实现让管理员指定哪些可执行文件,DLL,脚本可以被指定的用户或用户组执行．
    AppLocker由一个驱动程序(*SystemRoot%\system32\Drivers\Appld.sys*) 和一个运行在SvcHost进程的服务(*SystemRoot%\system32\Drivers\AppldSvc.dll*)所组成．

Windows组件和他们管理数据之间的关系

图6.1显示了其中一些组件和它们管理的数据库之间的关系。



安全等级所保护的对象

Windows平台下，被保护的对象主要有：文件，设备，邮件槽,管道(命名管道和匿名管道)，作业，进程，线程，事件，带键的的事件，事件对，互斥体，信号量，共享内存区，IO完成端口，LPC端口，可等待的定时器，访问令牌，卷，窗口站，桌面，网络共享体，服务，注册表键，打印机,活动目录对象等等.

只有那些只在内核使用，而不暴露给用户模式的对象才不受保护.
因为内核模式下的的代码是受信任的．用户模式下的代码不是.

令牌

Windows用令牌来控制哪些对象可以被访问，哪些安全操作可以被执行.
令牌由两部分组成.

1. 用户账户SID和组SID域
2. 特权数组(与该令牌相关的权限数组)

在windbg中查看令牌:
使用命令 ***dt _TOKEN*** 查看内核令牌对象的结构体格式.
通过***!token*** 命令可以查看令牌具体的值．命令的参数是令牌的地址.

令牌的地址可以通过 ***!process*** 命令来得到.如:

!process 0 1 cmd.exe

```
kd> !process 0 1 cmd.exe
PROCESS 839029b8  SessionId: 1  Cid: 0bc4    Peb: 7ffdb000  ParentCid: 0a1c
    DirBase: 3f2f7520  ObjectTable: 94f5b588  HandleCount:  18.
    Image: cmd.exe
    VadRoot 838f39c0 Vads 36 Clone 0 Private 107. Modified 0. Locked 0.
    DeviceMap 8b0828b0
    Token                             94b9ec88
    ElapsedTime                       00:26:05.470
    UserTime                          00:00:00.000
    KernelTime                        00:00:00.000
    QuotaPoolUsage[PagedPool]         70884
    QuotaPoolUsage[NonPagedPool]      2160
    Working Set Sizes (now,min,max)  (590, 50, 345) (2360KB, 200KB, 1380KB)
    PeakWorkingSetSize                590
    VirtualSize                       34 Mb
    PeakVirtualSize                   34 Mb
    PageFaultCount                    623
    MemoryPriority                    BACKGROUND
    BasePriority                      8
    CommitCharge                      397
```

输出结果中的Token 94b9ec88, 就是token的地址.

通过命令***!token 94b9ec88*** 可得到该令牌的信息:

```
kd> !token 94b9ec88
_TOKEN 94b9ec88
TS Session ID: 0x1
User: S-1-5-21-812292560-3787043718-3631103563-1000
User Groups:
 00 S-1-5-21-812292560-3787043718-3631103563-513
    Attributes - Mandatory Default Enabled
 01 S-1-1-0
    Attributes - Mandatory Default Enabled
 02 S-1-5-32-544
    Attributes - Mandatory Default Enabled Owner
 03 S-1-5-32-545
    Attributes - Mandatory Default Enabled
 04 S-1-5-4
    Attributes - Mandatory Default Enabled
 05 S-1-2-1
    Attributes - Mandatory Default Enabled
 06 S-1-5-11
    Attributes - Mandatory Default Enabled
 07 S-1-5-15
    Attributes - Mandatory Default Enabled
 08 S-1-5-5-0-84422
    Attributes - Mandatory Default Enabled LogonId
 09 S-1-2-0
    Attributes - Mandatory Default Enabled
 10 S-1-5-64-10
    Attributes - Mandatory Default Enabled
 11 S-1-16-12288
    Attributes - GroupIntegrity GroupIntegrityEnabled
Primary Group: S-1-5-21-812292560-3787043718-3631103563-513
Privs:
 05 0x000000005 SeIncreaseQuotaPrivilege          Attributes -
 08 0x000000008 SeSecurityPrivilege               Attributes -
 09 0x000000009 SeTakeOwnershipPrivilege          Attributes -
 10 0x00000000a SeLoadDriverPrivilege             Attributes -
 11 0x00000000b SeSystemProfilePrivilege          Attributes -
 12 0x00000000c SeSystemtimePrivilege             Attributes -
 13 0x00000000d SeProfileSingleProcessPrivilege   Attributes -
 14 0x00000000e SeIncreaseBasePriorityPrivilege   Attributes -
 15 0x00000000f SeCreatePagefilePrivilege         Attributes -
 17 0x000000011 SeBackupPrivilege                 Attributes -
 18 0x000000012 SeRestorePrivilege                Attributes -
 19 0x000000013 SeShutdownPrivilege               Attributes -
 20 0x000000014 SeDebugPrivilege                  Attributes -
 22 0x000000016 SeSystemEnvironmentPrivilege      Attributes -
 23 0x000000017 SeChangeNotifyPrivilege           Attributes - Enabled Default
 24 0x000000018 SeRemoteShutdownPrivilege         Attributes -
 25 0x000000019 SeUndockPrivilege                 Attributes -
 28 0x00000001c SeManageVolumePrivilege           Attributes -
 29 0x00000001d SeImpersonatePrivilege            Attributes - Enabled Default
 30 0x00000001e SeCreateGlobalPrivilege           Attributes - Enabled Default
 33 0x000000021 SeIncreaseWorkingSetPrivilege     Attributes -
 34 0x000000022 SeTimeZonePrivilege               Attributes -
 35 0x000000023 SeCreateSymbolicLinkPrivilege     Attributes -
Authentication ID:         (0,14b2e)
Impersonation Level:       Anonymous
TokenType:                 Primary
Source: User32             TokenFlags: 0x2000 ( Token in use )
Token ID: 652dc            ParentToken ID: 0
Modified ID:               (0, 65041)
RestrictedSidCount: 0      RestrictedSids: 00000000
OriginatingLogonSession: 3e7
```

安全标识符SID

SID即安全标识符(System IDentifier)，它用来标识用户身份的。当系统每次创建用户都会分配一个唯一的SID，每个帐户的SID都是不重复的。正因为SID有这样的特性，从Windows 2000以后的W系统对SID的依赖性较高，包括很多系统应用在内的系统内部进程引用帐户的SID而不是帐户的用户名和组名。因为用户的登陆名、显示名和归属的组等都可以修改，将一个帐户删除后再建立一个同名帐户，该账户的SID不同于被删除的那个帐户，所以它也不具有授权给前一个帐户的权利和权限。

来自 <http://beacon.blog.51cto.com/442731/116217/>

系统如何使用SID

当用户通过输入用户名口令得到身份验证(authentication)后，系统内部进程会给用户发放一个访问令牌，其实也就相当于一个票证(ticket)。此后用户访问系统资源时不再需要提供用户名和口令，

只需要将访问令牌提供给系统，然后系统检查用户试图访问对象(Resources)上的访问控制列表(ACL)。如果用户被允许访问该对象，系统将会分配给用户相应的访问权限，这也就完成了授权(authorization)的过程。

来自 <http://beacon.blog.51cto.com/442731/116217/>

令牌 - 用户SID和用户组SID的作用

用户SID 标识了这是哪个用户，拥有何种权限，用户组标识了该用户是哪些用户组的成员.

这些SID的作用:

安全对象具有访问控制列表(简称ACL) ，ACL说明了哪些用户(SID)能访问本对象，哪些不能. 以及哪些SID可以进行那种访问(读写执行等).

令牌 - 特权数组的作用

特权数组里面保存的是进程能够进行的特定系统操作，如关闭系统，修改系统时间，加载设备驱动等. 比如关机特权的进程，调用关机的API后会失败.

特权的种类和用途有:

特权	用户权限	用途
SeAssignPrimaryTokenPrivilege	替换一个进程层面的令牌	各种组件会检查此特权，比如NtSetInformationJob (设置一个进程的令牌)
SeAuditPrivilege	产生安全审计	想要利用ReportEvent 产生事件,并记录到安全事件日志中,这是必需的特权
SeBackupPrivilege	备份文件和目录	使得NTFS(文件系统)不管实际的安全描述符是什么，都授予对任何文件或目录下列访问权: READ_CONTROL ACCESS_SYSTEM_SECURITY FILE_GENERIC_READ FILE_TRAVERSE
SeChangNotifyPrivilege	绕过穿越检查	
SeCreateGlobalPrivilege	创建全局对象	当一个进程在对象管理器名字空间的目录中创建内存区和符号链接对象.
SeCreatePermanentPrivilege	创建一个页面文件	NtCreatePagingFile函数需要此特权
SeCreateSymbolicLinkPrivilege	创建符号链接	当NTFS利用CreateSymbolicLink函数在文件系统上创建符号链接时需要此特权
SeCreateTokenPrivilege	**创建一个令牌对象**	**NtCreateToken函数需要此特权，该函数会创建一个令牌对象**
SeDebugPrivilege	**调试程序**	**进程管理器会允许持有该特权的进程使用OpenProcess或OpenThread打开所有进程和线程.(受保护进程除外)**

SeShutdownPrivilege	**系统停机**	**NtShutdownSystem(关机,重启,注销)和NtRaiseHardError需要此特权**
SeEnableDelegationPrivilege	允许计算机账户和用户账户被用于委托	
SeImpersonatePrivilege	模仿一个客户	
SeIncreaseBasePriorityPrivilege	增加调度优先级	
SeIncreaseQuotaPrivilege	调整一个进程的内存配额	
SeIncreaseWorkingSetPrivilege	增加一个进程的工作集	
SeLoadDriverPrivilege	**加载和卸载设备驱动程序**	**驱动加载函数NtLoadDriver和驱动卸载函数NtUnloadDriver需要此特权**
SeLockMemoryPrivilege	将页面锁定在物理内存中(不允许交换到硬盘上)	**NtLockVirtualMemory需要此特权,该函数时VirtualLock的内核实现**
SeTcbPrivilege	**作为操作系统的一部分来执行**	
SeRestorePrivilege	**恢复文件和目录**	
SeTakeOwnershipPrivilege	**接管文件和其他对象的所有权**	

还有很多权限在这里没有列出，标粗的特权是功能强大的超级特权

1. **SeDebugPrivilege(调试程序)**
   拥有此特权的进程能够打开任何的进程，并且能够在任何进程上读写进程内存，创建进程,线程，因为进程线程的权限是由父进程继承的，因此，如果拥有该权限的进程注入了一个权限更多高的进程(如***LSASS***)，那么它所注入的线程或进程就拥有等同的权限.

2. **SeRestorePrivilege(恢复文件和目录)**
   拥有此特权的用户能够用他自己的文件来替换系统中任何文件.

3. **SeLoadDriverPrivilege(加载和卸载设备驱动程序)**
   一个恶意用户可以使用这种特权将一个设备驱动程序加载系统内核中．内核中的代码被视为操作系统的可信任部分，操作系统可能会在System账户凭证下执行设备驱动程序中的代码．所以一个驱动程序可以启动有特权的层序,这些特权程序再授予用户其他权限.

4. **SeCreateTokenPrivilege(创建一个令牌对象)**
   此特权很明显的用法是，生成一些代表任意用户账户的令牌．而且这些账户可以有任意的组成员关系和特权.

5. **SeTcbPrivilege(作为操作系统的一部分来执行)**
   LsaRegisterLogonProcess 函数需要此特权，进程可以调用此函数来建立起它与LSASS之间的可信连接．具有此特权的恶意用户可以建立一个可信的LSASS连接，然后执行LsaLogonUser来创建新登录会话，

以管理员身份运行

通过***ShellExecute 传入"runas"*** 参数将一个文件以管理员权限进行启动.
在启动过程中，会根据当前的用户权限不同而弹出不同的对话框.

1. 权限低于标准用户的用户权限，将会弹出一个带有密码输入框的AAM对话框来要求提升管理员权

限
2. 标准用户(即受限制的管理员用户) , 弹出一个有是和否两个按钮的对话框.如果点否, 程序启动失败. 根据进程的情况, 对话框的外观也会变得不同
   如果可执行文件拥有Windows发行的数字签名, 并且可执行文件是保存于系统安全目录中的一个, 则对话框是一个带绿条的对话框.并对话框上有个一金蓝相间的盾牌.
   如果可执行文件的数字签名是一个Windows之外的机构的数字签名, 则对话框有一个蓝色的带问号的盾牌.
3. 如果可执行文件没有数字签名,则对话框上又一个橙色且带问号的盾牌.

令牌和令牌信息的获取,修改

获取进程令牌: ***OpenProcessToken***
返回令牌句柄.

获取线程令牌: ***OpenThreadToken***
返回令牌句柄.

获取令牌信息: ***GetTokenInformation***
获取令牌信息
令牌信息有多种类型, 想要获取何种类型的信息,需要传入一个指定的枚举值.这些枚举值有:
TokenUser : 获取用户名密码对应的SID
TokenGroups, 获取用户组.
TokenPrivileges, 令牌的权限
TokenOwner, 令牌拥有者
TokenPrimaryGroup, 获取主用户组
TokenDefaultDacl,
TokenSource,
TokenType, 令牌类型,主类型和模仿类型
TokenImpersonationLevel, 令牌模仿等级
TokenStatistics,
TokenRestrictedSids,
TokenSessionId, 令牌会话ID
TokenGroupsAndPrivileges, 用户组优先级
TokenSessionReference,
TokenSandBoxInert,
TokenAuditPolicy,
TokenOrigin,
TokenElevationType, 令牌当前的提升等级
TokenLinkedToken, 和令牌链接在一起的另一个令牌
TokenElevation,
TokenHasRestrictions,
TokenAccessInformation,
TokenVirtualizationAllowed,
TokenVirtualizationEnabled,
TokenIntegrityLevel,
TokenUIAccess,
TokenMandatoryPolicy,
TokenLogonSid,
TokenIsAppContainer,
TokenCapabilities,
TokenAppContainerSid,
TokenAppContainerNumber,
TokenUserClaimAttributes,
TokenDeviceClaimAttributes,
TokenRestrictedUserClaimAttributes,
TokenRestrictedDeviceClaimAttributes,
TokenDeviceGroups,

```
TokenRestrictedDeviceGroups,
TokenSecurityAttributes,
TokenIsRestricted,
MaxTokenInfoClass
```

查询特权名称的LUID: ***LookupPrivilegeValue***

查看一个权限名对应的LUID值，
系统每次开机后，GUID是一个全球唯一标识，而LUID是本地唯一标识,它只保证每次系统开机之后的值都是不一样的.

修改令牌权限: ***AdjustTokenPrivileges***

修改权限 ，在这个函数中,权限使用LUID来表示. 因此，要想修改成何种权限，需要先获得这种权限对应的LUID值(可以使用***LookupPrivilegeValue***)

令牌

这篇文章有比较详细的介绍 http://blog.csdn.net/holandstone/article/details/7602032

三大网络模型_消息选择模型

2017年2月9日　　18:54

消息选择模型的相关函数

使用***WSAAsyncSelec*** 函数来将一个已经绑定,监听的套接字句柄和一个已经创建完毕的窗口句柄进行关联.

消息选择模型的工作原理

将一个套接字和窗口句柄关联，在关联之时，将一个自定义消息的消息码传入函数保存，并将需要接收的网络事件传入到函数，在响应的网络事件被触发之时，模型会向窗口句柄发送自定义的消息，并在LPARAM的低16位中保存被触发的网络事件代码．在LPARAM中的高16位中保存着错误码.

WSAAsyncSelec 函数返回值

当函数调用成功，返回值等于0，失败则返回***SOCKET_ERROR*** ，具体的错误可以通过WSAGetLastError函数的返回值得知．函数的返回值将会是以下***WSAAsyncSelec*** 函数调用失败的错误代码:

错误码	意义
WSANOTINITIALISED	在调用此函数前，必须先调用WSAStartup函数来进行初始化
WSAENETDOWN	网络子系统失败
WSAEINVAL	其中一个参数无效,可能是窗口句柄,窗口句柄必须是一个有效的窗口句柄,也可能是套接字句柄，如果是流式TCP套接字,则句柄必须使用***socket***创建,***bind***绑定，***listen***监听.
WSAEINPROGRESS	套接字可能由于其他原因处于阻塞状态中.
WSAENOTSOCK	不是一个套接字.

网络事件消息

网络事件如果被触发，消息选择模型就会为我们发送一条自定义消息.
在这个消息中，

uMsg	就是在***WSAAsyncSelec*** 中所注册的消息码

wParam	客户端套接字
lParam	低16位 被触发的网络事件代码 高16位 错误代码

主要的网络事件码

事件：***FD_ACCEPT*** 连接事件
事件：***FD_READ*** 读取事件
事件：***FD_WRITE*** 写入事件
事件：***FD_CLOSE*** 关闭事件
事件：***FD_CONNECT*** 连接事件

事件名	事件被触发时机
FD_READ	**recv, recvfrom, WSARecv, 或 WSARecvFrom.**
FD_WRITE	**send, sendto, WSASend, 或 WSASendTo.**
FD_ACCEPT	**accept 或 WSAAccept**
FD_CONNECT	None.
FD_CLOSE	None.

网络事件错误码描述

不同的网络事件，会有不同错误码，下面列出了所有不同事件所产生的错误码的描述

事件：***FD_CONNECT***

错误码	意义
WSAEAFNOSUPPORT	地址在这个套接字的协议组中不可用(可能是使用LPv6的地址,或IPv6使用了IPv4的地址).
WSAECONNREFUSED	连接被拒绝
WSAENETUNREACH	在当前的的这台主机上无法获得网络联系
WSAEFAULT	参数 *namelen* 无效
WSAEINVAL	套接字已经绑定到地址上
WSAEISCONN	套接字已经连接(当错误码是这个时,才说明连接事件是成功的.)
WSAEMFILE	没有更多的文件描述符是可用的
WSAENOBUFS	没有可用的缓冲区控件，套接字连接不了
WSAENOTCONN	套接字没有连接上
WSAETIMEDOUT	尝试连接超时

事件：***FD_CLOSE***

错误码	意义
WSAENETDOWN	网络子系统失败了
WSAECONNRESET	连接被对方重置
WSAECONNABORTED	因为超时或其他故障终止连接

事件：***FD_ACCEPT***
事件：***FD_READ***
事件：***FD_WRITE***

错误码	意义
WSAENETDOWN	网络子系统失败了

三大网络模型_事件选择模型

2017年3月31日 15:04

事件选择模型相关函数

WSAEventSelect	将一个套接字和一个事件对象进行关联，在套接字上发生了网络事件，与之关联的事件对象就会被设置为有信号
WSAWaitForMultipleEvents	等待多个事件对象. 和WaitForMultipleObjects的原理一样，函数会等待一个内核对象数组，当数组中有一个对象被触发为有信号的状态后，函数就会返回被触发的内核对象在数组中的索引.
WSAEnumNetworkEvents	枚举出事件对象对应的网络活动事件代码. 比如，事件A与套接字A进行了关联，当事件对象A被触发后，你可能仍然不知道是什么网络活动触发了这个事件对象.该函数的功能就是为了解决这个问题，它需要将被触发的事件和与之关联的套接字传入，函数会枚举出一个网络活动事件代码(FD_ACCEPT,FD_READ之类的网络活动事件代码)

WSAEnumNetworkEvents 详解

函数的第三个参数是输出参数，它就收一个结构体，函数会将枚举到的信息填写到该结构体中，结构体的具体字段为:

```
typedef struct _WSANETWORKEVENTS {
       long lNetworkEvents;
       int iErrorCode[FD_MAX_EVENTS];
} WSANETWORKEVENTS, FAR * LPWSANETWORKEVENTS;
```

[字段1] lNetworkEvents ： 在套接字上正在发生的网络活动事件代码，这个字段可能会保存多个事件代码，它们被 or 运算符组合在一个整型中，因此，如果你需要知道这个字段中有没有FD_READ事件，那么，你不能这样做:

```
     if(WSANETWORKEVENTS.lNetworkEvents == FD_READ){
          // 其它代码
     }
```

你只能这样做:

```
     if(WSANETWORKEVENTS.lNetworkEvents & FD_READ){
          // 其它代码
     }
```

[字段2] iErrorCode ： 这是一个数组，这个数组中的每个元素保存的都是错误代码. 但有点奇怪的时候，当特定网络活动被触发时，这个数组中的某个元素是有效的，其它的都是无效. 这是因为，在枚举网络活动事件时，有可能一次性出现集中网络活动(**lNetworkEvents** 就保存着多个事件代码)，因此，错误代码在此处使用了一个数组来存放.
所以，每个网络活动事件对应着数组中的一个元素，一般地，***FD_ACCEPT*** 事件对应第***FD_ACCEPT_BIT*** 个元素，***FD_READ*** 对应着第***FD_READ_BIT*** 个元素，以此类推，***FD_XXX*** 对应着***FD_XXX_BIT*** .
因此，你需要检查错误代码的话，你可以这样做:

```
if((WSANETWORKEVENTS.lNetworkEvents & FD_READ) && WSANETWORKEVENTS.iErrorCode[FD_READ_BIT]
== ERROR_SUCESS ){
     // 其它代码
}
```

三大网络模型_完成端口选择模型

2017年3月31日　　15:04

完成端口模型

accept函数一次只能等待一个客户端的接入,如果有多个客户端接入，就需要调用accept多次.

send函数是一个阻塞函数 ,它和WriteFile函数一样,在调用WrietFile函数时如果一次写入的数据过多,就会阻塞在函数内,等待数据写入完毕，调用send函数时，如果一次性发送的数据过大，那么它会阻塞在send函数中，如果数据发送需要一个小时，send就会阻塞一个小时.

recv函数是一个阻塞函数，它会等到一个远程套接字把数据发送过来，如果远程套接字迟迟没有将数据发送过来，那么就会一直阻塞在recv函数内部.

完成端口模式中，提供了异步IO版本的send和recv函数，分别叫:*WSASend* 和 *WSARecv* ，这两个函数可以使用异步IO.

完成端口模型分解

网络模型中的完成端口实际上分为两部分，每个部分都各有线程来处理.

第一部分:

第一部分的线程用于等待客户端接入，在这个工作线程中做的事情有两件:

1. 调用accept函数接受客户端的连接，并将客户端套接字和完成端口绑定在一起(因为完成端口只能等待和完成端口绑定在一起的内核对象)
2. 调用异步IO版的*WSARecv*函数接收客户端套接字发送过来的数据. 由于使用了异步IO版的*WSARecv*，因此在这里调用的时候，函数并不会阻塞住. 调用完毕就重新循环，回到第一步继续等待新客户端接入. 在上一步进行了绑定，因此，在第二部分的线程函数中的*GetQueuedCompletionStatus* 会收到新套接字的IO任务.

第二部分:

第二部分的线程用于等待WSARecv的IO任务完成. 在这个工作线程中，做的事情有:

1. 调用*GetQueuedCompletionStatus* 函数等待IO任务完成
2. 处理完成的IO任务.如果想知道已经完成的IO任务是何种任务，那么在第一部份的第2个步骤中，调用WSARecv投递IO任务时，需要将任务的细节存入到OVERLAPPED结构中(如任务的缓冲区,缓冲区的字节数等细节)，当IO任务完成的时候，就可以在这一步中得到OVERLAPPED,进一步就可以得到OVERLAPPED中的具体细节.

虚拟内存管理

2017年4月7日　　20:22

进程的虚拟地址空间

32位进程拥有 0~4Gb的地址空间 ，64位进程拥有0 ~ 16EB的地址空间．各个进程的空间是独立的，无法直接相互访问．

虽然应用程序有这么大的地址空间可用，但是这只是虚拟地址空间，并不是真正的物理内存空间．它们仅仅是一个地址，为了能够让这些虚拟地址和真正的物理内存一样能够进行读写操作，我们还需要把物理内存分配或映射到相应的虚拟地址空间中，这样一来，当我们读写虚拟地址的时候，操作系统就自动在对应的物理内存上进行读写．

进程虚拟地址空间的分区

分区	在x86 32位Windows的范围	在x64 64位Windows的范围	IA-64 64位windwos
空指针赋值区	0 ~0xFFFF	0~0xFFFF	0 ~ 0xFFFF
用户模式分区	0x10000 ~ 0x7FFEFFFF	0x10000 ~ 0x7FFFFFEFFFF	0x10000 ~ 0x6FBFFFEFFFF
64KB禁入分区	0x7FFF000~0x7FFFFFFF	0x7FFFFFF0000 ~ 0x7FFFFFFFF	0x6FBFFFF0000 ~ 0x6FbFFFFFFFF
内核模式分区	0x80000000 ~ 0xFFFFFFFF	0x80000000000 ~ 0xFFFFFFFFFFFFFFFF	0x6FC00000000 ~ 0xFFFFFFFFFFFFFFFF

地址空间中的区域

当系统创建一个进程之后，进程中的大部分地址空间都是闲置的或尚未分配的．闲置的地址空间是没有对应的物理内存的，因此，一个进程虽然有4Gb的地址空间，但它真正占用的物理内存可能会很小.只有当进程使用***VirtualAlloc***预定并调拨了虚拟地址空间，它才会占用一定的物理内存．

一个没有经过预定并调拨的空闲地址空间或尚未分配的地址空间被使用时，就会引起一个错误．

预定地址空间

预定地址空间可以使用VirtualAlloc函数，这个函数会在程序的闲置地址空间中找打一段合适的地址，并进行预定，经过预定之后，这段地址空间会从闲置状态变成已预定状态，当再次调用***VirtualAlloc***函数时，已经被预定的地址空间就不会被再次预定．

预定地址空间时，需要指定地址空间的大小．系统会确保这个大小正好是系统页面大小的整数倍．一般x86个x64系统使用的页面大小为4Kb． IA-64系统使用的页面为8Kb．

```
LPVOID pBuff = NULL;
pBuff = VirtualAlloc(NULL, /*预定时,不需传入地址*/
                     1024*64, /*预定的字节数*/
```

```
    MEM_RESERVE, /*预定内存的标志*/
    0       /*只是在预定,因此,无法设置内存的保护属性*/
);
```

给已预定的地址空间调拨物理内存

地址空间只是被预定时，不能用它来进行读写数据，因为这些地址空间仅仅是被打上了一个已被预定的标志，系统并没有将之与真正的物理内存进行关联．要进行关联，需要进行调拨操作．
调拨操作也是通过***VirtualAlloc*** 函数来完成．

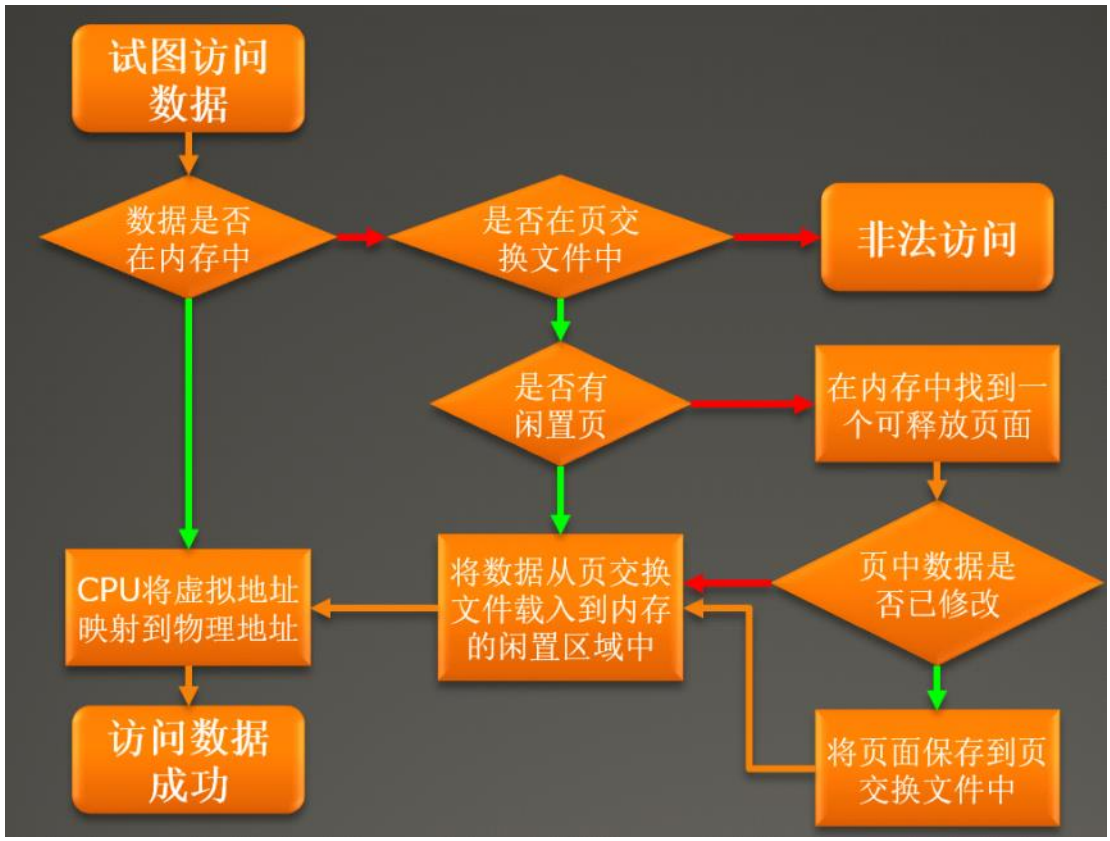
当使用***VirtualAlloc***进行内存调拨时，系统会确保区域的起始地址正好是分配粒度的整数倍，也就是***VirtualAlloc***函数返回的地址总是分配粒度的整数倍，一般这个分配粒度是***64KB*** ，因此，在分配内存空间时，大小应取整为64Kb的倍数，否则,即使没有使用那么多内存空间，系统再下一次调拨空间时，也会为了保证新空间的起始地址是分配粒度的整数倍而跳过剩余的地址空间.

```
VirtualAlloc(
    pBuff ,                  // 需要进行提交的已预定的地址
    1024*64*1,               // 预定并调拨的字节数: 1024 * 64KB * 1
    MEM_RESERVE|MEM_COMMIT, // 预定并调拨内存
    PAGE_READWRITE);         // 内存分页保护属性: 可读写
```

当分配出来的物理内存不再需要时 ，可以使用***VirtualFree*** 来释放物理存储器

物理内存和页交换文件

当真正的物理内存被占用完时，系统会有选择地把一些不常用的物理内存保存到磁盘的文件中，这些文件就被称为页交换文件.
当这些物理内存再次被使用时，系统会把这些内存分页从磁盘重新载入到物理内存当中.
为了能够使用虚拟内存，操作系统需要CPU的大力协助，当线程视图访问某个字节时，CPU必须知道这个字节是在物理内存中还是在磁盘上.
CPU在访问一个数据时，将需要判断这个数据是在内存中还是在磁盘中，如果这个数据是在内存中,则CPU就能够将虚拟地址转换成物理内存地址去访问到这个数据，如果不在内存中,CPU就会产生一个页错误的中断，产生这个中断之后，操作系统的的中断处理程序就会被立即调用,这个处理程序负责将位于磁盘中的内存页重新载入到物理内存，如果物理内存已经没有剩余的空间，则中断程序会将当前物理内存中的一些不常用的内存页交换到磁盘中，这样就有物理内存可以使用了.
上述流程如下图所示:



内存页的保护属性

windwos系统在管理内存分页时，给每个分页都设置了以下属性

内存分页保护属性	描述
PAGE NOACCESS	不具有任何权限,试图读,写内存分页上的数据,或在内存分页上执行代码都将引发页面访问违规
PAGE_READONLY	只读权限，不能在该分页上进行写入,也不能执行代码.
PAGE_READWRITE	可读可写，不能执行代码
PAGE_EXECUTE_READ	只读和可执行，不可写入
PAGE_EXECUTE_READWRITE	可读可写可执行
PAGE_WRITECOPY	写时拷贝,不能执行代码,进行写入时,系统会为进程单独创建一份该页面的副本.
PAGE_EXECUTE_WRITECOPY	可读,可执行,写时拷贝,进行写入时,系统会为进程单独创建一份该页面的副本.

遍历进程的虚拟内存

VirtualQuery 函数

虚拟内存有三种性质:

1. **虚拟内存状态**

   内存状态描述一块虚拟地址空间有无被预定，有无和物理存储器进行映射. 内存状态一共有3种:闲置(free),保留(reserve),提交(commit).

闲置 ： 说明虚拟地址没有被使用(没有被预定,更没有被提交).
保留 ： 说明虚拟地址已经被预定， 但是没有提交,也就是没有和物理存储器进行映射.
提交 ： 说明虚拟地址已经被映射到物理存储器.

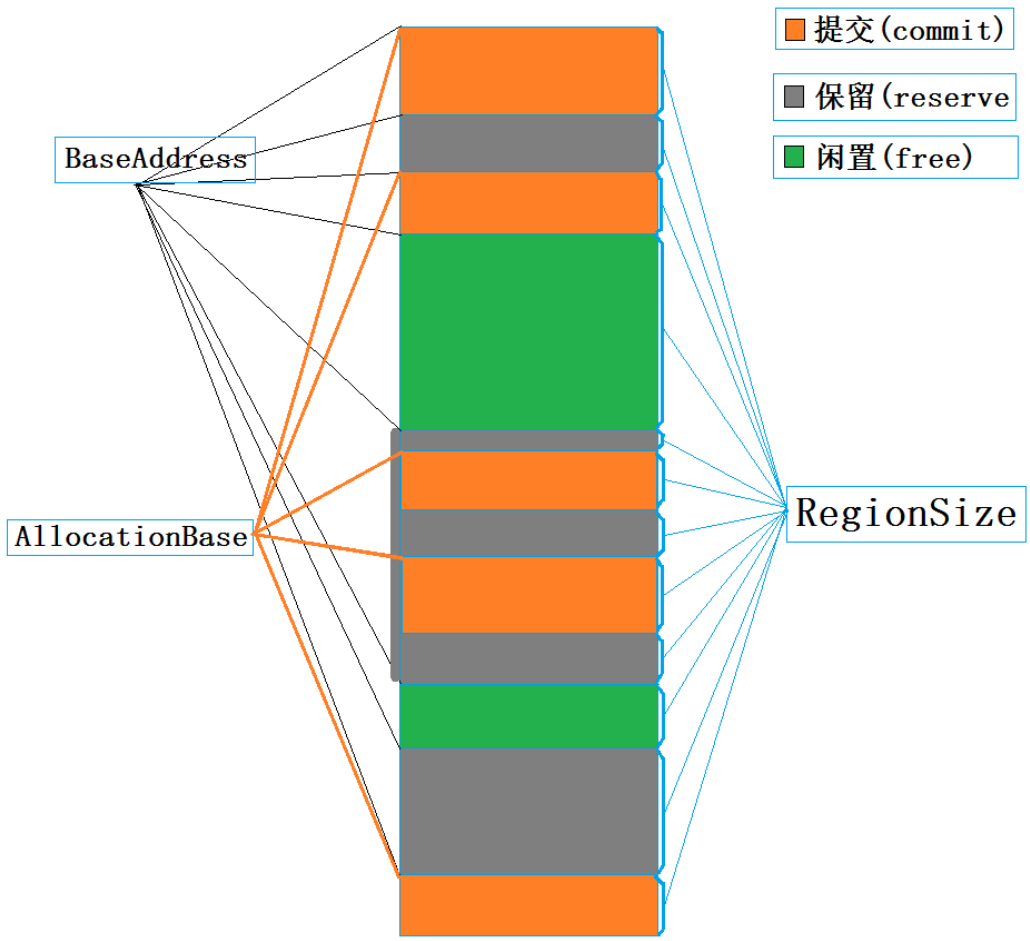
2. **虚拟内存映射方式**
   当虚拟地址被提交后， 一个虚拟地址就能够映射到物理存取器， 以下是虚拟地址和物理存储器进行映射的方式:
   1. private ： 私有映射， 这是属于进程独有的内存,不和其它进程共享,一般是堆,栈等.
   2. mapped ： 映射到其它物理存储器， 是从其它物理存储器映射而来.
   3. image ： 来自PE映像文件， 当程序加载后， 可执行文件的各个区段就是以image方式呗映射到内存.

3. **虚拟内存分页属性**
   只有那些被提交的虚拟内存才具有内存分页属性.
   windwos对内存分页设定了一些属性(windwos内存管理的最小单位是1个分页， 一个分页一般是4096Kb.)， 当程序访问一个虚拟地址时， 这个虚拟地址所在的内存分页属性就被操作系统检查,只有当内存分页具有了一定属性， 程序才能正常访问.

VirtualQuery 函数会使用*MEMORY_BASE_INFORMATION* 结构体来保存虚拟内存信息

```
typedef struct _MEMORY_BASIC_INFORMATION {
    PVOID BaseAddress;          // 内存基地址
    PVOID AllocationBase;       // 预定内存时的首地址
    DWORD AllocationProtect;    // 提交内存时的内存分页属性
    SIZE_T RegionSize;          // 提交内存时的大小
    DWORD State;                // 内存状态
    DWORD Protect;              // 内存属性,和AllocationProtect一样
    DWORD Type;                 // 内存映射类型
} MEMORY_BASIC_INFORMATION , *PMEMORY_BASIC_INFORMATION;
```

下图是一个进程的虚拟内存分布图:



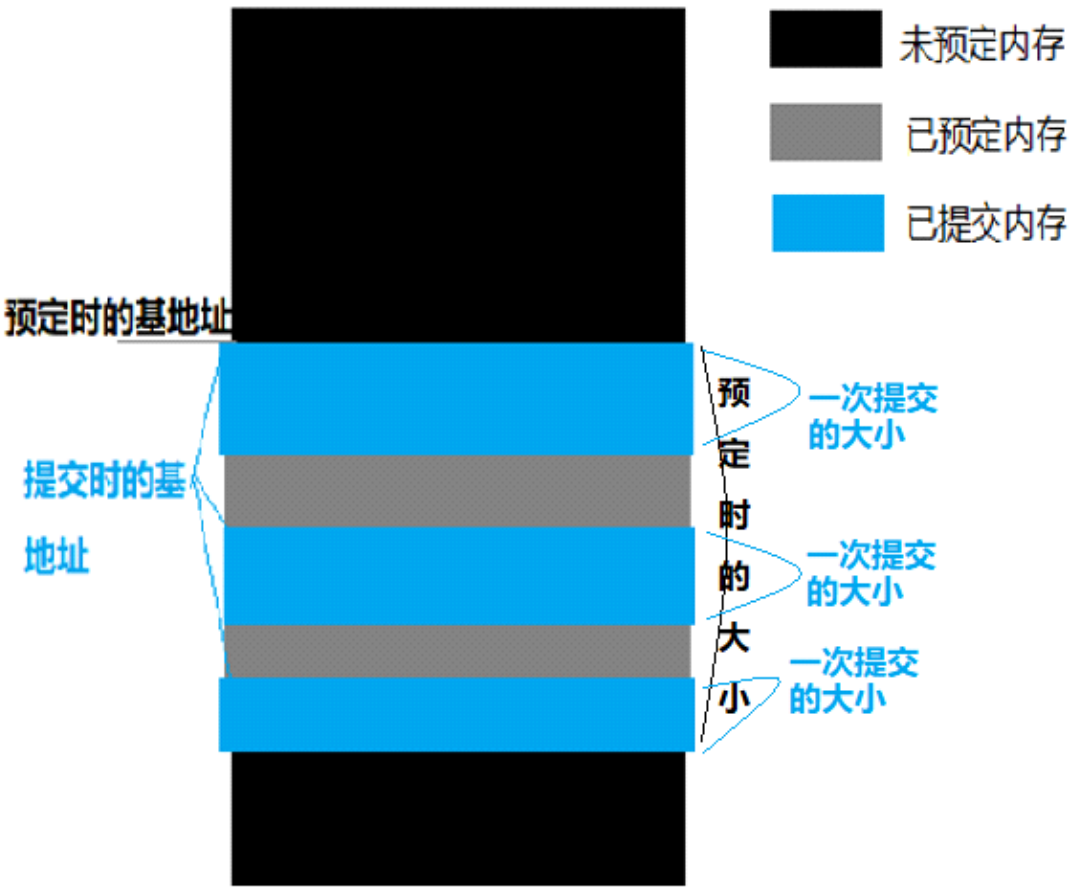
当我们将一个地址传入到*VirtualQuery* 函数后,这个函数就会将这个地址所在的内存块的信息通过结构体输出给我们.
如果想要遍历一个进程的所有内存块，就必须知道每一个内存块的首地址，然后将这些首地址出传入函数即可.
若想获取每一个内存块的首地址，可以将0(也就是第一个内存块首地址)传入函数，这样,函数就会返回第一个内存的大小，知道大小后，也能够知道下一个内存块的起始地址了.如此往复，就能找到所有的内存块.

分配基地址和基地址的关系

虚拟内存经过预定才能进行使用，因此，一块内存先需要进行分配(预定），然后在提交，在这两次操作中，就会出现一个分配基地址 ，一个提交基地址.

在预定内存之后 ，内存仍然还是不能直接使用，因此，一般在预定时会预定一块比较大的页面.在使用内存时，需要进行提交，提交时，只能在已经预定的内存页面上进行提交，而已预定的虚拟内存页面都比较打，提交时不会一次性都将之提交，而是几个字节，或几个页面的提交，这样一来，就形成一个预定的内存块拥有多个提交的内存块的情形了.

因此，分配基地址和基地址的关系如下图:



HOOK

2017年10月17日 星期二　　17:05

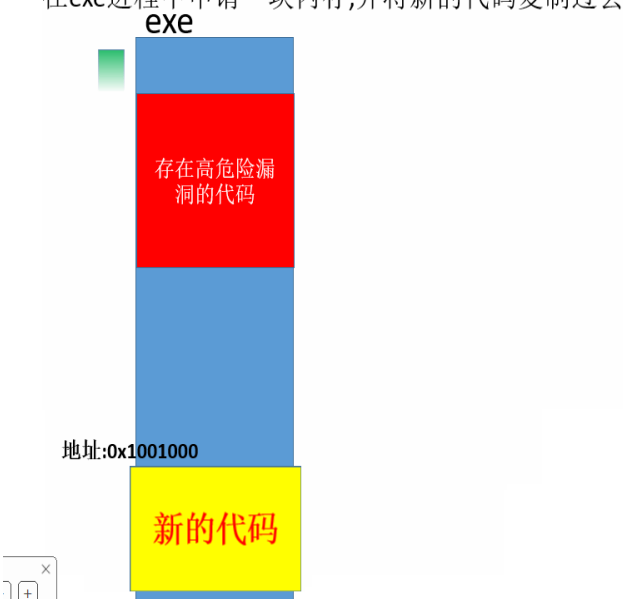
HOOK的描述

hook译文为钩子 ， 在计算机中， 也被译为拦截.

在window操作系统中 ， 就有HOOK的使用场景. Windows服务器系统是一种要求比较高的系统 ,作为服务器在系统运行期间不能随便重启,一旦重启,客户端就都掉线了… 但是当服务器系统被发现存在高危漏洞时， 必须要进行修复,否则系统就会收到入侵的安全威胁， 修复的过程说白了其实就是将某个具有高位漏洞的模块(exe文件或dll文件)进行替换， 但是当exe或dll处于运行状态时是无法被修改,删除,覆盖的. 因此,必须关掉服务器程序或者说重启系统,在exe或dll运行起来之前， 才能将新的文件覆盖旧文件. 这是合乎情理的安排， 要不是有更好的实现方法出现的话， 这种措施就被应用了. Windows使用的是一种热修补的技术， 当一个模块出了高危漏洞， 并不是说整个模块都不能用了,实际上,这些漏洞有可能仅仅是由一个或几个函数引发出来的， 只要能将这几个函数的代码换掉就好了. 所以， 在修复这些高位漏洞的时候， 可以先将已经准备好的二进制代码保存到内存中 ， 在存在高危漏洞的二进制代码上使用jmp跳转命令直接跳转到没有漏洞的代码， 这样就可以完美做到漏洞修复.

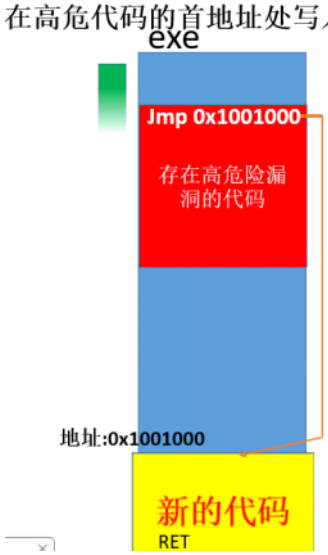
1.

在exe进程中申请一块内存,并将新的代码复制过去



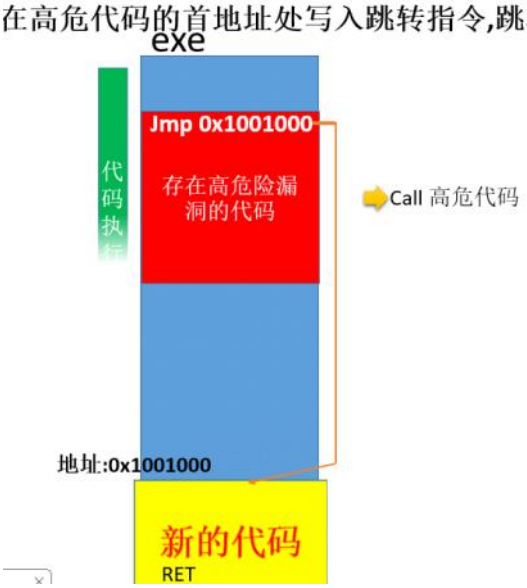
2.

在高危代码的首地址处写入跳转指令,跳转到新代码

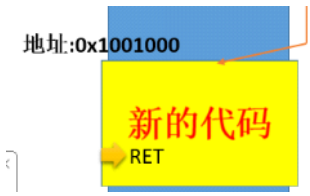


3.

在高危代码的首地址处写入跳转指令,跳转到新代码



4.



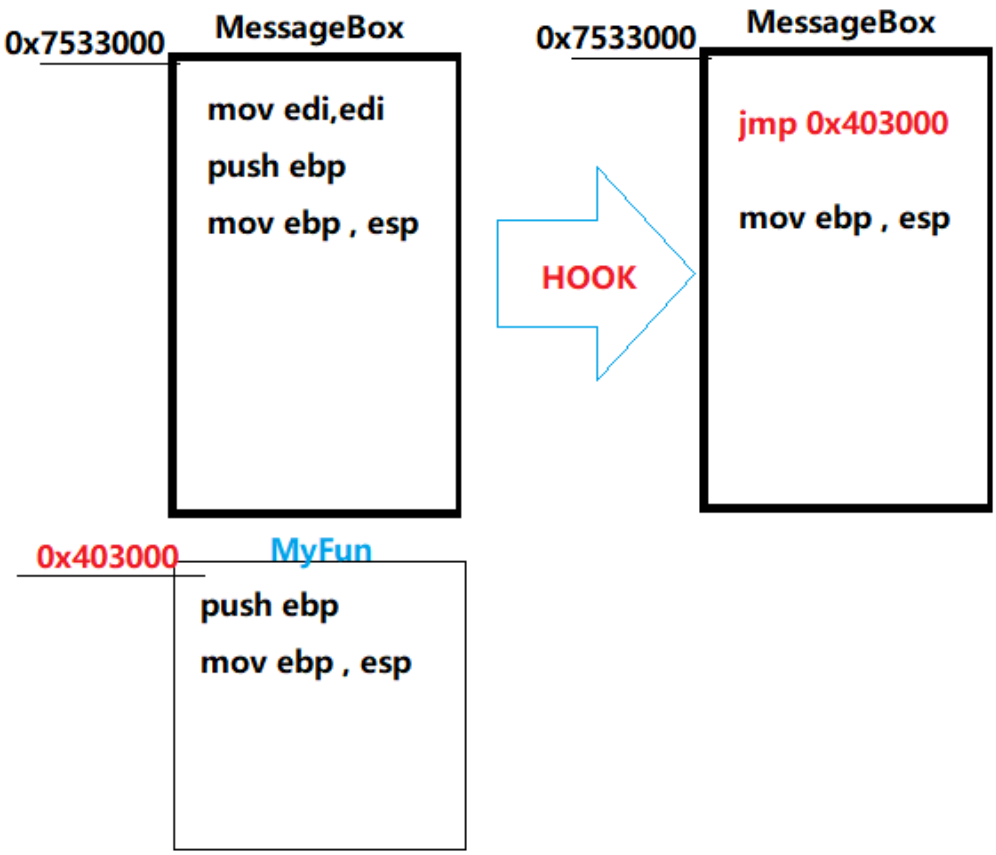
5

上面的热修补过程,实际上就是一次**拦截过程**, 我们在高危代码的首地址处使用一条跳转指令,将执行流的eip拦截住了， 使得EIP不再走向高危代码， 而是转到新的代码中执行， 当新的代码被执行完之后,通过ret语句就能够返回到调用高位代码的call语句的下一条语句了.

这样的一次绝妙的拦截过程就是一次HOOK过程.
实际上，很多病毒程序也正式通过这项技术去拦截一些至关重要的API ， 使得我们应用程序每次调用API之前，EIP都会跑到病毒程序预先设置的代码中去执行病毒代码.

HOOK的实现原理以及应用

上述的热修补过程已经将HOOK的原理展现得淋漓尽致 ， 而下面的描述,则是一次对一个真实API进行真材实料的拦截.
当我们在程序中调用MessageBox时，作为API的MessageBox就会被调用，EIP将指向位于User32.dll中的MessageBox函数的二进制代码. 并在最后，通过ret指令回到调用MessageBox的地方. 如果我们在调用MessageBox之前，找到MessageBox二进制代码的首地址，在它的首地址处写入一条Jmp指令，那么MessageBox被调用的时候，我们的写入进去的Jmp指令也将会被调用. 我们在Jmp指令后面编写一个指定的地址，比如一个函数的地址，那么MessgaeBox函数的调用将变成另一个函数的调用.
实现如图所示:



这里有一个非常绝妙的巧合. Windows的API的头是mov edi,edi,push ebp这两条指令的机器码,这两条指令的机器码的长度是5个字节，而一条跳转指令恰好也是5个字节. 这样的状况实际并非巧合，而是编译器在生成代码故意为之,是为了更方便操作系统进行热修补而出现的，在visual studio中就有这样的一个选项可以让我们生成类似的函数:

这些过程看起来是比较简单的，但我们在实际的操作中,需要考虑一些其它因素，例如，代码所在的内存分页一般是可读可执行,不可写的，如果冒然将jmp指令的机器码写入进去，就会导致内存页面访问错误.因此,在写入之前,需要使用VirtualProtect函数来修改分页属性. 在写入jmp机器码时，jmp指令的操作数是一个跳转偏移,而非要跳转到的地址. 这个偏移是需要我们在写入jmp指令前就得计算出来得，因此,我们需要知道jmp指令跳转偏移的计算公式.一般，jmp指令和call指令的跳转偏移计算公式是这样的:

跳转偏移 = 目标地址 - jmp指令所在地址 - jmp指令的总长度

另外，在对函数拦截成功之后，如果我们还想继续使用被拦截函数的功能(比如拦截了MessageBox,却还想调用MessageBox函数),此时是不可以直接调用被拦截函数的，因此被拦截函数已经写入了jmp指令,一旦被调用,就会跳转到拦截函数中，这样一来就会造成无限递归. 解决的方法是，在调用之前，将jmp指令撤销,将函数原先的5个字节恢复回去,这样一来就能调用了，在调用之后在将jmp指令写回即可.

注入

2017年10月18日 星期三　　12:32

注入背景介绍

通过HOOK,我们可以改变原函数的功能 ， 我们可以让代码执行到一个函数的时候跳转到另一个函数执行代码, 前提是另一个函数的代码在同一个进程的虚拟地址空间中.
有时候, 我们需要通过一个进程, 去修改另一个进程的某个函数 ， 即一个进程HOOK另一个进程 ， 比如外挂程序就是这样的一个程序 ， 外挂程序不需要HOOK自己, 它要修改的是游戏进程的代码, 比如说在游戏中,要购买某种装备后,金币就会减掉, 如果使金币减少是通过一个函数来实现的, 那么拦截这个函数 ， 使之在被调用时跳转到另一块代码执行, 在另一块代码中, 把保存在栈中的参数修改掉(将金币数修改成0),在跳回原函数执行代码, 这样一来,购买时就不会被扣金币了.
但需要注意的是,拦截游戏进程.手段.
我们可以将函数放在DLL工程中, 然后将这个DLL加载到游戏进程, 这样外挂代码就进入到了游戏进程的虚拟地址空间中了, 只要找到那个修改游戏金币个数的函数的地址, jmp命令就可以跳转此函数中了.
将一个DLL加载到指定进程这样的行为, 我们称之为注入.
至于加载DLL,目前为止, 我们知道的能够实现加载DLL的API,恐怕只有LoadLibrary了, 但LoadLibrary只能将DLL加载到本进程, 也就是在哪个进程调用了这个API,它就把DLL加载到哪个进程.
但是, Windows系统中, 存在着一些让人意想不到的巧合, 这些巧合能够使我们轻而易举地将DLL注入到指定的进程中.

远程线程注入

特色API : CreateRemoteThread
功能　　: 为一个进程创建一个线程, 并可以指定一个位于对方进程的函数作为线程回调函数

远程线程注入中的远程二字指的是对方线程,本线程之外的另一个线程. 这个注入方式利用到的就是上面的这个API . 我们知道, 一个进程想要加载DLL, 就需要调用LoadLibrary函数, 而LoadLibrary函数的调用却只能为本进程加载模块, 不能为别的进程加载模块, 要是能够让对方进程的LoadLibrary函数自己调用就好了.
CreateRemoteThread正好有这个能力. 此API运行之后,能够在对方进程中创建一个线程. 线程的回调函数可以在调用函数时指定, 如果将LoadLibrary函数作为回调函数传给这个API ， 那么, 对方进程的LoadLibrary函数就会被调用. 这一巧妙的组合就能够将DLL注入到指定的进程中.

注册表注入

注册表注入利用的是注册表.一个exe在运行之前, 系统检查注册表的路径 *HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows* ,在这个路径下如果存在一个键*LoadAppInit_Dlls*并且其值为1时 ， 系统就会将注册表同一目录下的*AppInit_Dlls*键中记录的DLL路径一一加载到进程中, 这个键中的DLL路径可以存在多个, 使用空格作为分隔符, 并且第一个路径可以使用绝对路径, 后续的都只能使用相对路径(相对于系统路径), 我们可以事先将要注入的DLL准备好,然后将DLL路径记录在这个键中, 这样一来exe运行起来后,DLL就能自动加载到exe中.

添加导入表记录

当exe被双击时,pe加载器就会解析exe的PE文件格式, 修复导入表, 将导入表中记录的DLL加载到进程中. 我们可以修改exe的导入表, 为其增加一个DLL记录. 这样一来, exe运行时, PE加载器就能自动加载此DLL. 但这样的方法需要我们将dll和exe放在同一个目录, 或者将dll放在系统路径下.

其它注入方式

1. ComRes注入
2. APC注入
3. 消息钩子注入
4. 依赖可信进程注入
5. 劫持进程创建注入
6. 输入法注入