Windows 核心编程(相关笔记）

第六章　　线程的基础知识

线程是由两个部分构成的，一个是线程的内核对象。另一个是线程堆栈。（维护线程在执行代码时需要的所有函数参数和局部变量）跟进程类似。
进程是不活泼的，进程从来不执行任何东西。它只是线程的容器。

线程只有一个内核对象和一个堆栈，保存的记录很少，所以占的内存也很少。
每当进程被初始化时，系统就要创建一个主线程。
该主线程与 C/C++运行期库的启动代码一同开始运行。C/C++启动代码调用进入点函数(main,winmain)，并且继续运行直到进入点函数返回，
并且(C/C++启动代码）调用 ExitProcess 为止。

每个线程必须拥有一个进入点函数，线程从这个进入点开始运行。。。（也是个进入点函数）。。。

```
DWORD     WINAPI    ThreadFunc(PVOID pvParam){
DWORD     dwResult=0;
....
return (dwResult);
}
```

最终线程到达它的结尾处并且返回。这时线程终止运行，该堆栈的内存被释放，同时，线程的内核对象
的使用计数递减，如果为 0，则撤销。
线程函数可以用任何名字。
线程函数不必担心 ANSCII 和 UNICODE 的问题。
与主线程一样，线程函数也必须返回一个值作为该线程的退出代码。
线程函数（实际上是你的所有函数)应该尽可能使用函数参数和局部变量，因为他们是在线程堆栈中创建的。
因此不太可能被另一个线程破坏。

CreateThread 函数

```
HANDLE    CreateThread(
PSECURITY_ATTRIBUTES      psa,
DWORD       cbStack,
PTHREAD_START_ROUTINE    pfnStartAddr,
PVOID    pvParam,
DWORD    fdwCreate,
PDWORD pdwThreadID);
```

系统从进程的地址空间中分配内存，供线程的堆栈使用

单个进程中的多个线程非常容易的实现互相通信
CreateThread 函数是用来创建线程的 Windows 函数。不过如果你在编写 C/C++代码，决不应该调用 CreateThread....如果是 Visual c++编译器，用它的替代函数_beginthreadex.

PSA 通常传递 NULL..如果希望所有的子进程能够继承该线程对象的句柄，必须设定一个 SECURITY_ATTRIBUTES 结构。。。
它的 bInheritHandle 成员被初始化为 TRUE..

cbStack 参数用于设定线程可以将多少地址空间用于它自己的堆栈。每个线程都有它自己的堆栈。
可以使用链接程序的/STACK 开关来控制这个值.
/STACK:[reserve][,commit]

分配的存储器容量应该与传递的 cbStack 值相一致

pfnStartAddr　pvParam...
pfnStartAddr 参数用于指明新线程执行的线程函数的地址

CreateThread 使用 pvParam 只是在线程启动执行时将参数传递给线程函数。。pvParam 参数提供了一个将
初始化值传递给线程函数的手段。。该初始化数据既可以是数字值，也可以是指向包含其他信息的一个
数据结构的指针。。。
创建多个线程，使这些线程拥有与起始点相同的函数地址，这是完全合乎逻辑且非常有用的。。。

Windows 是一个抢占式多线程系统

```
DWORD　WINAPI　FirstThread(PVOID　pvParam){
int x=0;
DWORD　dwThreadID;

//Create a new　thread..
HANDLE　hThread=CreateThread(NULL,0,SecondThread,(PVOID)&x,0,&dwThreadId);

CloseHandle(hThread);
return(0);
}

DWORD　WINAPI　SecondThread(PVOID pvParam){
……
*((int *)pvParam)=5;
```

.....
return(0);
}

参数 fdwCreate
可以设定用于控制创建线程的其他标志
0 或 CREATE_SUSPENDED
0:线程创建后可以立即进行调度
CREATE_SUSPENDED: 系统可以完整的创建线程并对它进行初始化，但是要暂停该线程的运行，这样它就无法
进行调度

pdwThreadID
必须是 DWORD 的一个有效地址
系统用这个地址来存放分配给新线程的 ID

终止线程的运行
终止线程的运行的方法：
1：线程函数的返回（最好使用这种方法）；
2：通过调用 ExitThread 函数，线程将自行撤销（最好不要使用这种方法);
3：同一个或另一个进程中的线程调用 TerminateThread 函数（应该避免使用这种方法）；
4：包含线程的进程终止运行（避免）

1：
确保所有线程资源被正确清除的唯一办法
实现的事项：
创建的所有 C++对象将通过他们的撤销函数正确的撤销
操作系统将正确释放线程堆栈使用的内存
系统将线程的退出代码（在线程的内核对象中维护）设置为线程函数的返回值
系统将递减线程内核对象的使用计数

2：
ExitThread 函数
VOID ExitThread(DWORD dwExitCode);
该函数终止线程的运行，并导致操作系统清除线程所使用的所有操作系统资源。但是，C++资源（如 C++类对象）将不被撤销由于这个原因，最好从线程函数返回，而不是通过调

用这个函数来返回

3：
TerminateThread 函数
BOOL　TereminateThread(
HANDLE　hThread,
DOWRD　dwExitCode);

TerminateThread 函数是异步运行的函数，但是当函数返回时不能保证线程被撤销。如果需要知道线程是否终止运行，必须调用 WaitForSingleObject 或类似的函数，传递线程的句

柄。线程终止时，DLL 通常接受到通知。如果使用 TerminateThread 函数强制终止线程，DLL 就不接受通知。这能阻止适当的清除。

4：
在进程终止运行时撤销线程。

线程终止运行时发生的操作。线程拥有的所有用户对象均被释放。

BOOL GetExitCodeThread(
HANDLE　hThread,
PDWORD　pdwExitCode);

线程的一些性质
刚开始时，线程的内核对象设置为未通知状态。

线程堆栈：高地址-》低地址　pvParam-pfnStartAddrj
IP-》kernel32.dll 中的 BaseThreadStart()函数。

写入堆栈的第一个值是传递给 CreateThread 的 pvParam 参数的值。紧靠它的下面是传递 CreateThread 的 pfnStartAddr 参数每个线程都有它自己的一组 CPU 寄存器，称为线程的上下

文。它反映了线程上次运行时该线程 的 CPU 寄存器的状态。线程的这组 CPU 寄存器保存在一个 CONTEXT 结构。CONTEXT 结构本身则包含在现存的内核对象中。

指令指针和堆栈指针寄存器。

当线程的内核对象被初始化时，CONTEXT 结构的堆栈指针寄存器被设置为线程堆栈上用来放置 pfnStartAddr

的地址。
指令指针寄存器置为 BaseThreadStart 的未文档化的函数的地址中。

```
VOID BaseTHreadeStart(PTHREAD_START_ROUTINE pfnStartAddr,PVOID pvPam){
__try{
ExitThread((pfnStartAddr)(pvParam));
}
__except(UnhandledExceptionFilter(GetExceptionInformation())){
ExitProcess(GetExceptionCode());
}

}
```

当线程完全初始化后，系统就要查看 CREATE_SUSPENDED 标志是否已经传递给 CreateThread。。如果该标志没有传递
系统便将线程的暂停计数减为 0.。该线程可以调度到一个进程中。。然后系统用上次保存在线程上下文中的值
加载到实际的 CPU 寄存器中，这时线程就可以执行代码，并对它的进程的地址空间中的数据进行操作。

由于新线程的指令指针寄存器被置为 BaseThreadStart,因此该函数实际上是线程开始执行的地方。

当新线程执行 BaseThreadStart 函数时，将会出现下列情况。
在线程函数中建立一个结构化异常处理（SEH）帧，这样，在线程执行时产生的任何异常情况都会得到系统的
某种默认处理。。

系统调用线程函数，并将你传递给 CreateThread 函数的 pvParam 参数传递给它。
当线程函数返回时，BaseThreadStart 调用 ExitThread,并将线程函数的返回值传递给它。
该线程内核对象的使用计数被递减，线程停止执行。。
如果线程产生一个没有处理的异常条件，由 BaseThreadStart 函数建立的 SEH 帧将负责处理该异常条件。。
通常情况下，这意味着想用户显示一个消息框，并且在用户撤销该消息框时，BzsethreadStart 调用 ExitThread,
以终止整个进程的运行。

当进程的主线程初始化时，他的指令指针被设置为另一个为文档化的函数，成为 BaseProcessStart...
这两个函数的唯一差别是，BaseProcessStart 没有引用 pvParam 参数。
当 BaseProcessStart 开始运行时，它调用 C/C++运行期库的启动代码，该启动代码先要初始化 main,wmain,

WinMain,wWinMain 函数，然后调用这些函数，当进入点函数返回时，C/C++运行期库的启动代码调用 ExitProcess。
因此，对于 C/C++应用程序来说，主线程从不返回 BaseProcessStart 函数。

第 7 章　线程的调度.优先级和亲缘性

上一章介绍了每个线程是如何拥有一个上下文结构，这个结构维护在线程的内核对象中。。这个上下文结构反映了
线程上次运行时该线程的 CPU 寄存器的状态。。每隔 20MS 左右，WINDOWS 要查看当前存在的所有线程内核对象。。
在这些对象中，只有某些对象被视为可以调度的对象。。。WINDOWS 选择可调度的所有线程内核对象中的
一个，将它加载到 CPU 的寄存器中，它的值是上次保存在线程的环境中的值。。这称为上下文转换。。

使用 MICROSOFT　SPY++这个工具就可以观察线程的情况。。。
系统对线程进行调度的过程：
当线程正在执行代码，并对它的进程的地址空间的数据进行操作。再过 20ms 左右，Windows 就将 CPU 的
寄存器重新保存在线程的上下文中。。。线程不再运行。系统再次查看其余的可调度线程的内核对象，选定另一个
线程的内核对象。将该线程的上下文加载到 CPU 的寄存器中，然后继续运行。。当系统引导时，便开始加载线程的上下文，
让线程运行，保存上下文和重复这些操作，直到系统关闭。。

Windows 被称为抢占式多线程操作系统，因为一个线程可以随时停止运行，随后另一个线程可进行调度。。。

除了暂停的线程外，其他许多线程也是不可调度的线程，因为他们正在等待某些事情的发生。

例如：
如果运行 NOTEPAD，但是不键入任何数据，那么 Notepad 的线程就没有什么事情要做。。。系统不给无事可做的
线程分配 CPU 时间。当移动 NOTEPAD 的窗口时，或NOTEPAD 的窗口需要刷新他的内容，或将数据键入 NOTEPAD，
系统就会自动使 NOTEPAD 的线程称为可调度的线程。。。这并不意味着 NOTEPAD 的线程立即获得了 CPU 时间。它只是表示
NOTEPAD 的线程有事情可做，系统将设法在某个时间对它进行调度。。

7.1　暂停和恢复线程的运行。。

在线程内核对象的内部有一个值，用于指明线程的暂停计数。。
当调用 CreateProcess 或 CreateThread 函数，就创建了线程的内核对象，并且它 的暂停计数被初始化为 1.
这可以防止线程被调度到 CPU 中。。这是很有用的，因为现场的初始化需要时间，你不希望在系统做好准备之前就开始执行线程。。
当线程完全初始化好了之后， CreateProcess 或 CreateThread 要查看是否传递了 CREATE_SUSPENDED 标志。。。
如果传递了，那么这些就返回，同时新线程处于暂停状态。。如果尚未传递该标志，那么将该线程的暂停计数减为 0.
当暂停计数是 0 的时候，除非线程正在等待其他某种事情的发生，否则该线程就处于可调度的状态。。

在暂停状态中创建了一个线程，就能在线程有机会执行任何代码之前改变线程的运行环境（如优先级）。。。

DWORD　ResumeThread(HANDLE　hThread);

DWORD　SuspendThread(HANDLE　hThread);

任何线程都可因调用该函数来暂停另一个线程的运行。。。。线程可以自行暂停运行，但是不能自行
恢复运行。。

7.2　暂停和恢复进程的运行
进程调用 WaitForDebugEvent 和 ContinueDebugEvent 之类的函数。。

作者的 SuspendProcess 函数的实现代码：

```
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=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);
}
}
```

这个 SuspendProcess 函数使用 ToolHelp 函数来枚举系统中的线程列表。。当我找到作为指定进程的组成部分的线程
时，调用 OpenThread:

```
HANDLE    OpenThread(
DWORD     dwDesireAccess,
BOOL      bInheritHandle,
DWORD     dwThreadID);
```

7.3　睡眠方式

线程也能告诉系统，他不想再某个时间段内被调度。这是通过调用 Sleep 函数来实现的。。

```
VOID Sleep(DWORD dwMilliseconds);
```

该函数可是线程暂停自己的运行，直到 dwMilliseconds 过去为止。关于这个函数有几个问题注意：

调用 Sleep,可使线程自愿放弃他剩余的时间片。。

系统将在大约的指定的毫秒数内使线程不可调度。。

可以调用 Sleep，并且为 dwMilliseconds 参数传递 INFINITE...这将告诉系统永远不要调度该线程。。这不是
一件值得去做的事情。。最好是让线程退出。。并还原他的堆栈和内核对象。。。

可以将 0 传递给 Sleep...这将告诉系统，调用线程将释放剩余的时间片，并迫使系统调度另一个线程。。。

7.4　转换到另一个线程。。

```
BOOL    SwitchToThread();
```

该函数允许一个需要资源的线程强制另一个优先级较低。而目前拥有该资源的线程放弃该资源。。如果调用 SwitchToThread
函数时没有其他线程能够运行，那么该函数返回 FALSE，否则放回非 0 值 。。。

另外，SwitchToThread 允许优先级较低的线程运行。。

调用 SwitchToThread 函数与调用 Sleep 是相似的。。。

7.5　线程的运行时间

```
DWROD dwStartTime=GetTickCount();
DWORD dwElapseTime=GetTickCount()-dwStartTime;


BOOL    GetThreadTimes(
HANDLE hThread,
PFILETIME    pftCreationTime,
PFILETIME    pftExitTime,
PFILETIME    pftKernelTime,
PFILETIME    pftUserTime);

__int64    FileTimeToQuadWord(PFILETIME pft){
        return(Int64ShllMod32(pft->dwHighDateTime,32)|pft->dwLowDateTime);
}

void    PerformLongOperation(){
FILETIME ftKernelTimeStart,ftKernelTimeEnd;
FILETIME ftUserTimeStart,ftUserTimeEnd;
FILETIME tfDummy;
__int64 qwKernelTimeElapsed,qwUserTimeElapsed,qwTotalTimeElapsed;

GetThreadTimes(GetCurrentThread(),&ftDummy,&ftDummy,&ftKernelTimeStart,&ftUserTimeSt
art);

GetThreadTimes(GetCurrentThread(),&ftDummy,&ftDummy,&ftKernelTimeEnd,&ftUserTimeEn
d);

qwUserTimeElapsed=FileTimeToQuadaWord(&ftUserTimeEnd)-
FileTimeToQuadWord(&ftUserTImeStart);

qwKernelTimeElapsed=FileTimeToQuadWord(&ftKernelTimeEnd)-
FileTimeToQuadWord(&ftKernelTimeStart);

qwtotalTimeElapsed=qwKernelTimeElapsed+qwUserTimeElapsed;


}



BOOL  GetProcessTimes(
HANDLE    hProcess,
```

```
PFILETIME    pftCreationTime,
PFILETIME    pftExitTime,
PFILETIME    pftKernelTime,
PFILETIME    pftUserTime);
```

对于高分辨率的配置文件，系统提供了一些高分辨率性能函数：

```
BOOL  QueryPerformanceFrequency(LARGE_INTEGER* pliFrequency);
BOOL  QueryPerformanceCounter(LARGE_INTEGER* pliCount);
```

为了使这些函数运行起来更容易些。。。创建了一个类。。。

```
class    CStopwatch{
public:
        CStopwatch(){    QueryPerformanceFrequency(&m_liPerfFreq);Start();}
void    Start(){
        QueryPerformanceCounter(&m_liPerfStart);}

__int64    Now()    const{
      QueryPerformanceCounter(&liPerfNow);
      return(((liPerfNow.QuadPart-m_liPerfStart.QuadPart)*1000)
}

private:
        LARGE_INTEGER    m_liPerfFreq;
        LARGE_INTEGER    m_liPerfStart;
};
```

使用这个类如下：

```
CStopwatch    stopwatch;
__int64    qwElapsedTime=stopwatch.Now();
```

7.6　运用结构环境

CONTEXT 结构包含了特定处理器的寄存器数据。。。参加头文件 WinNT.h...
CONTEXT 结构包含了主机 CPU 上的每个寄存器的数据结构。。
在 X86 计算机上，数据成员是 EAX,EBX,ECX,EDX...
如果是 Alpha 处理器，那么数据成员包括 IntV0,IntT0,IntT1,IntS0,IntRa 和 IntZero 等等。。
CONTEXT_CONTROL 包含 CPU 的控制寄存器。。。
CONTEXT_INTEGER 包含标识 CPU 的整数寄存器。。
CONTEXT_SEGMENTS 标识 CPU 的段寄存器。。
CONTEXT_DEBUG_REGISTER 标识 CPU 的调试寄存器。。

CONTEXT_EXTENDED_REGISTER 标识 CPU 的扩展寄存器。。。

Windows 实际上允许查看线程内核对象的内部情况。。。需调用 GetThreadContext 函数。。。
BOOL GetThreadContext(
HANDLE hThead,
PCONTEXT pContext);

在调用 GetThreadContext 函数之前，应该调用 SuspendThread.否则线程可能被调度。。。而且线程的环境可能与你收回的不同。。。
一个线程实际上有两个环境：用户方式，内核方式。。。
GetThreadContext 只能返回线程的用户方式环境
CONTEXT 结构的 ContextFlags 成员用于向 GetThreadText 函数指明你想检索哪些寄存器。。。
例如：想获得线程的控制寄存器，可以编写下面的代码：

```
CONTEXT Context;

Context.ContextFlags=CONTEXT_CONTROL;

GetThreadContext(hThread,&Context);
```

```
Context.ContextFlags=CONTEXT_CONTROL|CONTEXT_INTEGER;
```

CONTEXT_FULL 获得所有重要的寄存器。。。

要查看只需要调用 GetThreadText 就行。。

Windows 使你能够修改 CONTEXT 结构中的成员，通过调用 SetThreadContext 将新寄存器值放回线程的内核对象中。。。

```
BOOL SetThreadContext(
HANDLE hThread,
CONST CONTEXT *pContext);
```

同样，修改其环境的线程应该首先暂停。。

代码：

```
CONTEXT Context;
```

```
SuspendThread(hThread);

Context.ContextFlags=CONTEXT_CONTROL;
GetThreadText(hThread,&Context);

#if    defined(_ALPHA_)
Context.Fir=0x00010000;
#elif    defined(_X86_)
Context.Eip=0x00010000;
#else
#error     Module contains CPU_specific code;modify and fecompile...
#endif


Context.ControlFlags=CONTEXT_CONTROL;
SetThreadContext(hThread,&Context);

ResumeThread(hThread);
```

7.7 线程的优先级

每个线程都会被赋予一个从 0 到 31 的优先级号码。。。当系统确定将哪个线程分配给 CPU 时，它首先观察优先级为 31 的线程。。并
以循环方式对他们进行调度。。。
渴求调度。。。当高优先级的线程使用大量的 CPU 时间，从而使低优先级的线程无法运行时，便会出现渴求情况。。。
在多处理器计算机上这种情况出现的少的多。。。
在任何一个时段，系统中的大多数线程是不能调度的。。例如进程的主线程调用 GetMessage 函数，而系统发现
没有现场可以供它使用。那么系统就暂停线程的运行。。。释放该线程的剩余时间片。。并且立即将 CPU 分分配给另一个
等待运行的线程。。。
如果没有为 GetMessage 函数显示可供检索的消息，那么进程的线程就保持暂停状态，并且决不会被分配 CPU。。但是，当消息被置于线程的队列中，
系统就知道该线程不应再处于暂停状态。。。此时，如果没偶遇更搞优先级的线程需要运行，该线程就会被分配 CPU。。。
当系统引导时，它会创建一个特殊的线程，称为 0 页线程。。优先级为 0.。是整个系统中唯一的一个在优先级 0 上
运行的线程。。。当系统中没有任何线程需要执行操作时，0 页线程负责将系统中的所有空闲 RAM 页面置 0.。。

7.8 对优先级的抽象说明

Windows API 展示了系统的调度程序上的一个抽象层，这样就永远不会直接与调度程序进行

通信。。
相反，要调用 windows 函数，以便根据运行的系统版本“转换"参数。。。
优先级类：实时，高，高于正常，正常，低于正常，空闲。。。
Task Manager 任务管理器处在高优先级类。。
Windows 支持 7 个相对的线程优先级：空闲，最低，低于正常，正常，高于正常，最高，和关键时间优先级。。。。
这些优先级是相对于进程的优先级类而言的。。。大多数线程都是用正常线程优先级。。。
进程根本不能调度的，只有线程才能被调度。。。

大多数时候高优先级的线程不应该处于可调度状态。当线程要进行某种操作时，他能迅速获得 CPU 时间。。
这时线程应该尽可能少地执行 CPU 指令。。。并返回睡眠状态。。等待再次变成可调度状态。。
相反，低优先级的线程可以保持可调度状态。。。执行大量的 CPU 指令来进行它的操作。。。
如果按这些原则来办，
整个操作系统就能正确对用户作出相应。。。。

7.9　程序的优先级。。。
当调用 CreateProcess 时，可以在 fdwCreate 参数中传递需要的优先级类。。。
优先级类 id 标识符。。。
实时　　REALTIME_PRIORITY_CLASS
高　　HIGH_PRIORITY_CLASS
高于正常　ABOVE_NORMAL_PRIORITY_CLASS
正常　　NORMAL_PRIORITY_CLASS
低于正常　BELOW_NORMAL_PRIORITY_CLASS
空闲　　IDLE_PRIORTIY_CLASS

创建子进程的进程负责选择子进程的优先级类。例如 Explorer...
当使用 Explorer 来运行一个应用程序时，新进程按正常优先级运行。。。
Explorer 不知道进程在做什么，也不知道隔多久它的线程需要进行调度。。。但是，一旦子进程运行，它就能通过调用 SetPriorityClass 来改变它自己的优先级类：

```
BOOL SetPriorityClass(
HANDLE　hProcess,
DWORD fdwPriority);
```

检索进程的优先级类的补充函数：

```
DWORD　GetPriorityClass(HANDLE　hProcess);
```

当使用命令外壳启动一个程序时，该程序的其实优先级是正常优先级。但是
如果使用 Start 命令来启动该程序，可以使用一个开关来设定应用程序的起始优先级。。例如在命令外壳输入下面的命令可使系统启动 Calculator，并在开始时按空闲优先级来运行它。。。

c:\start /low calc.exe

Start 命令还能识别其他标识符开关。。。以便按他们各自的优先级启动执行一个应用程序。。当然，一旦应用程序启动运行，他就可以调用 SetPriorityClass 函数，将它自己的优先级该为它选择的任何优先级。。。

Windows 2000 的 Task Manager 使得用户可以改变进程的优先级类。。。。

```
BOOL SetThreadPriority(
HANDLE hThread,
int nPriority);
```

nPriority 是线程相对优先级的标识符常量。。。。
THREAD_PRIORTIY_TIME_CRITICAL
.....

```
int GetThreadPriority(HANDLE hThread);
```

若要创建一个带有相对优先级为空闲的线程，可以执行类似下面的代码：

```
DWORD dwThreadID;
HANDLE
hThread=CreateThread(NULL,0,ThreadFunc,NULL,CREATE_SUSPENDED,&dwThreadID);
SetThreadPriority(hThread,THREAD_PRIORITY_IDLE);
ResumeThread(hThread);
CloseHandle(hThread);
```