==========
第十二课　线程同步
==========

一、线程
----

1. 基本概念

1) 线程是可以执行的代码的实例。系统以线程为单位调度程序。
　　一个程序当中可以包含多个线程，实现多任务处理。

2) 线程的特点

A. 每个线程都有自己的唯一标识号——TID。
B. 每个线程都有自己的安全属性。
C. 每个线程都有自己的内存堆栈。
D. 每个线程都有自己的寄存器信息。

3) 进程多任务与线程多任务

A. 系统中的多个进程各自使用自己的地址空间。
B. 一个进程中的多个线程共享同一个地址空间。

4) 线程调度

将CPU的执行时间划分成时间片，分配给不同的线程。
操作系统根据时间片的分配，轮流执行不同的线程。

2. 线程过程函数

```
DWORD WINAPI ThreadProc (
    LPVOID lpParameter // 线程参数
);
```

返回值代表线程执行成功或者失败，可由GetExitCodeThread函数获得。

3. 创建线程

```
HANDLE CreateThread (
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
                              // 安全属性，NULL
    SIZE_T  dwStackSize,      // 线程栈的初始字节数，
                              // 0表示与调用线程栈相等字节数
    LPTHREAD_START_ROUTINE lpStartAddress,
                              // 线程过程函数指针
    LPVOID  lpParameter,      // 线程参数
    DWORD   dwCreationFlags, // 创建方式
    LPDWORD lpThreadId        // 线程ID(输出)
);
```

成功返回线程句柄，失败返回NULL。

dwCreationFlags取值：
    0   - 创建之后立即执行。
    CREATE_SUSPENDED - 创建之后挂起，
                       直到调用ResumeThread函数再执行。

4. 退出线程

```
VOID ExitThread (
    DWORD dwExitCode // 退出码
);
```

5. 终止线程

```
BOOL TerminateThread (
    HANDLE hThread,   // 线程句柄
    DWORD  dwExitCode // 退出码
);
```

成功返回TRUE，失败返回FALSE。

8. 关闭线程句柄

```
CloseHandle
```

并非关闭线程，仅释放线程句柄资源。

9. 挂起线程

```
DWORD SuspendThread (
    HANDLE hThread // 线程句柄
);
```

成功返回线程此前挂起的次数，失败返回-1。

10. 恢复线程

```
DWORD ResumeThread (
    HANDLE hThread // 线程句柄
);
```

成功返回线程此前挂起的次数，失败返回-1。

11. 获取线程ID

```
DWORD GetCurrentThreadId (VOID);
```

返回调用线程的ID。

12. 获取线程句柄

```
HANDLE GetCurrentThread (VOID);
```

返回调用线程的伪句柄。

13. 通过线程ID获取其句柄

```
HANDLE OpenThread (
    DWORD dwDesiredAccess, // 访问权限，THREAD_ALL_ACCESS
    BOOL  bInheritHandle,  // 子进程是否可以继承此函数返回的句柄
    DWORD dwThreadId       // 线程ID
);
```

成功返回对应给定TID的线程句柄，失败返回NULL。

范例：WinST、WinMT、MultiThread

二、线程同步

1. 原子锁(Interlocked)

1) 原子自增

```
LONG InterlockedIncrement (
    LPLONG lpAddend // 自增变量指针
);
```

返回自增结果。

2) 原子自减

```
LONG InterlockedDecrement(
    LPLONG lpAddend // 自减变量指针
);
```

返回自减结果。

范例：WinLock

2. 临界区(Critical Section)

CRITICAL_SECTION - 临界区结构体

1) 初始化临界区

```
VOID InitializeCriticalSection (
    LPCRITICAL_SECTION lpCriticalSection // 临界区对象地址
);
```

2) 进入临界区

```
VOID EnterCriticalSection (
```

```
    LPCRITICAL_SECTION lpCriticalSection // 临界区对象地址
);
```

阻塞，直到调用线程获得对指定临界区对象的所有权才返回。

3) 离开临界区

```
VOID LeaveCriticalSection (
    LPCRITICAL_SECTION lpCriticalSection // 临界区对象地址
);
```

释放对指定临界区对象的所有权。

4) 删除临界区

```
VOID DeleteCriticalSection (
    LPCRITICAL_SECTION lpCriticalSection // 临界区对象地址
);
```

范例：WinCS

3. 互斥体(Mutex)

1) 创建互斥体

```
HANDLE CreateMutex (
    LPSECURITY_ATTRIBUTES lpMutexAttributes,
                          // 安全属性，NULL
    BOOL    bInitialOwner, // 调用线程是否初始拥有该互斥体
    LPCTSTR lpName         // 互斥体名称，跨进程使用，用于线程取NULL
);
```

成功返回互斥体句柄，失败返回NULL。

2) 等待互斥体

```
WaitForSingleObject (hMutex, INFINITE);
```

若其它线程拥有该互斥体，则hMutex无信号，函数阻塞，
直到调用线程获得对该互斥体的所有权，此时hMutex有信号，函数返回。

3) 释放互斥体

```
BOOL ReleaseMutex (
    HANDLE hMutex // 互斥体句柄
);
```

成功返回TRUE，失败返回FALSE。

4) 关闭互斥体

```
CloseHandle (hMutex);
```

范例：WinMutex

4. 信号量(Semaphore)

1) 创建信号量

```
HANDLE CreateSemaphore (
    LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
                          // 安全属性，NULL
    LONG    lInitialCount, // 初始资源计数
    LONG    lMaximumCount, // 最大资源计数
    LPCTSTR lpName         // 信号量名称，跨进程使用，用于线程取NULL
);
```

成功返回信号量句柄，失败返回NULL。

2) 等待信号量

```
WaitForSingleObject (hSemaphore, INFINITE);
```

若资源计数为0，则hSemaphore无信号，函数阻塞，
直到资源计数大于0，此时hSemaphore有信号，函数返回，同时资源计数减1。

3) 释放信号量

```
BOOL ReleaseSemaphore (
    HANDLE hSemaphore,      // 信号量句柄
    LONG   lReleaseCount,   // 资源计数增量
    LPLONG lpPreviousCount // 此前资源计数(输出)，可为NULL
);
```

成功返回TRUE，失败返回FALSE。

4) 关闭信号量

```
CloseHandle (hSemaphore);
```

范例：WinSema

5. 事件(Event)

1) 创建事件

```
HANDLE CreateEvent (
    LPSECURITY_ATTRIBUTES lpEventAttributes,
                          // 安全属性，NULL
    BOOL    bManualReset,  // 是否手动复位
    BOOL    bInitialState, // 是否初始化为有事件
    LPCTSTR lpName         // 事件名称，跨进程使用，用于线程取NULL
);
```

成功返回事件句柄，失败返回NULL。

若bManualReset为FALSE，则WaitFor...函数返回即自动复位，
否则必须通过ResetEvent函数对事件复位。

2) 等待事件

```
WaitForSingleObject (hEvent, INFINITE);
```

若无事件，则hEvent无信号，函数阻塞，直到有事件发生，
此时hEvent有信号，函数返回，同时将事件复位(若bManualReset取FALSE)。

3) 设置事件

```
BOOL SetEvent (
    HANDLE hEvent // 事件句柄
);
```

成功返回TRUE，失败返回FALSE。

4) 复位事件

```
BOOL ResetEvent (
    HANDLE hEvent // 事件句柄
);
```

成功返回TRUE，失败返回FALSE。

5) 关闭事件

```
CloseHandle (hEvent);
```

范例：WinEvent

6. 可等候定时器(Waitable Timer)

1) 创建可等候定时器

```
HANDLE CreateWaitableTimer (
    LPSECURITY_ATTRIBUTES lpTimerAttributes,
                          // 安全属性，NULL
    BOOL    bManualReset, // 是否手动复位
    LPCTSTR lpTimerName   // 可等候定时器名称，用于线程取NULL
);
```

成功返回可等候定时器句柄，失败返回NULL。

2) 启动可等候定时器

```
BOOL SetWaitableTimer (
    HANDLE hTimer,                    // 可等候定期的句柄
    const LARGE_INTEGER* pDueTime,    // 初始时间(正数)/间隔(负数)
                                      // 以100纳秒为单位
    LONG   lPeriod,                   // 重复间隔，0表示一次性定时
                                      // 以毫秒为单位
    PTIMERAPCROUTINE pfnCompletionRoutine,
                                      // 定时处理函数
    LPVOID lpArgToCompletionRoutine, // 定时处理函数参数
    BOOL   fResume                    // 定时到达是否令系统从休眠中恢复
);
```

成功返回TRUE，失败返回FALSE。

定时处理函数参数：

```
VOID CALLBACK TimerAPCProc (
    LPVOID lpArgToCompletionRoutine, // 定时处理函数参数
    DWORD  dwTimerLowValue,          // 系统时间低32位
    DWORD  dwTimerHighValue          // 系统时间高32位
);
```

3) 停止可等候定时器

```
BOOL CancelWaitableTimer (
    HANDLE hTimer // 可等候定期的句柄
);
```

成功返回TRUE，失败返回FALSE。

4) 关闭可等候定时器

```
CloseHandle (hTimer);
```

范例：WinTimer

7. 线程局部存储(TLS)

1) 分配线程局部存储索引

```
DWORD TlsAlloc (VOID);
```

成功返回线程局部存储索引，失败返回-1。

2) 保存数据到调用线程的局部存储槽中

```
BOOL TlsSetValue (
    DWORD  dwTlsIndex, // 线程局部存储索引
    LPVOID lpTlsValue  // 数据
);
```

成功返回TRUE，失败返回FALSE。

3) 从调用线程的局部存储槽中获取数据

```
LPVOID TlsGetValue (
    DWORD dwTlsIndex // 线程局部存储索引
);
```

成功返回线程局部存储中的数据，失败返回NULL。

4) 释放线程局部存储索引

```
BOOL TlsFree(
    DWORD dwTlsIndex // 线程局部存储索引
);
```

成功返回TRUE，失败返回FALSE。

范例：WinTls

5）静态线程局部存储

```
__declspec (thread) int g_cn = 0;
__declspec (thread) static int s_cn = 0;
```

范例：WinTlsEx