# Threads and Thread Synchronization for Win32 and MFC

Jim Fawcett

CSE681 – Software Modeling and Analysis

Fall 2005

# Introduction

- These notes are concerned with Win32 threads and, at the end, with MFC wrapped threads.
- In order to understand how threads work, we will quickly examine the Win32 API and some of its significant objects.

# Win32 API

- ***Creating and supporting windows:***
  - defining, creating, destroying, and setting the style of windows
  - writing text and graphics
  - windows menus and controls
- ***Files and directories:***
  - creating, opening, reading, and writing files
  - searching files and directories
- ***Registry***
  - writing information into the registry
- ***Timers***
- ***Processes, threads, and fibers***
  - Creating and terminating
- ***Errors and Exception handling***
- ***Dynamic link libraries***
  - Loading, creating, and accessing data

# Handles and Objects

- An (operating system) object is a data structure that represents a system resource, e.g., file, thread, bitmap.
- An application does not directly access object data or the resource that an object represents. Instead the application must acquire an object handle which it uses to examine or modify the state of the system resource.
- Each handle refers to an entry in an internal object table that contains the address of a resource and means to identify the resource type.

# Handles and Objects (continued)

- The win32 API provides functions which:
  - create an object
  - get an object handle
  - get information about the object
  - set information about the object
  - close the object handle
  - destroy an object

- Objects fall into one of three categories:
  - ***kernel objects*** used to manage memory, process and thread execution, and inter-process communication
  - ***user objects***, used to support window management
  - ***gdi objects***, supporting graphics operations

# Examples of Win32 Objects

- ***Windows kernel Objects:*** ***kernel32.dll***
  - Events
  - Files
  - Memory-Mapped Files
  - Mailslots and Pipe objects
  - Mutex and Semaphore objects
  - Processes and Threads
- ***GDI Objects:*** ***gdi32.dll***
  - pens and brushes
  - fonts
  - Bitmaps
- ***User Objects:*** ***user32.dll***
  - windows
  - hooks
  - menus
  - mouse cursors

# Kernel Objects

- Most of the win32 functions you use to create, synchronize, and monitor threads rely on kernel objects.
- Kernel objects are operating system resources like processes, threads, events, mutexes, semaphores, shared memory, and files.
- Kernel objects have security attributes and signaled state.

# Signaled Object State

- Except for files, kernel objects are opaque. You don't have access to their internal structure.
- All kernel objects have a signaled state. They are always either signaled or nonsignaled.
- An object that is in the signaled state will not cause a thread that is waiting on the object to block.
- A kernel object that is not in the signaled state will cause any thread that waits on that object to block until the object again becomes signaled.
- Access to a kernel object is controlled by security attributes
- Except when creating or opening a kernel object, you refer to it by a HANDLE rather than a name. The HANDLE is returned by the function that creates or opens the object.
- Kernel object names are system-wide resources and can be used by one process to create and another process to open. HANDLEs are unique and have meaning only within a single process.
- Kernel objects are reference counted. Objects are destroyed only when there are no outstanding references to that object.

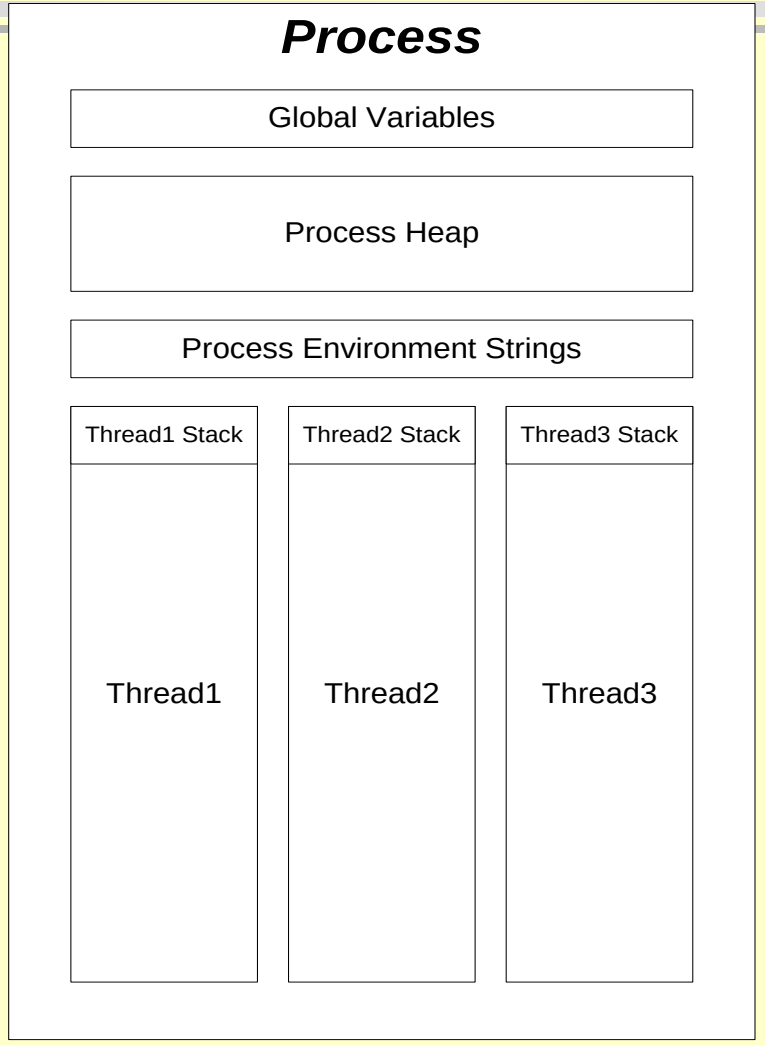
# Threads

- A ***thread*** is a path of execution through a program's code, plus a set of resources (stack, register state, etc) assigned by the operating system.
- A thread lives in one and only one process. A process may have one or more threads.
- Each thread in the process has its own call stack, but shares process code and global data with other threads in the process.
- Pointers are process specific, so threads can share pointers.

# Thread vs. Process

- A Process is inert.  A process never executes anything; it is simply a container for threads.
- Threads run in the context of a process.  Each process has at least one thread.
- A thread represents a path of execution that has its own call stack and CPU state.
- Threads are confined to context of the process that created them.
  - A thread executes code and manipulates data within its process's address space.
  - If two or more threads run in the context of a single process they share a common address space.  They can execute the same code and manipulate the same data.
  - Threads sharing a common process can share kernel object handles because the handles belong to the process, not individual threads.

# Threads vs. Process



# Starting a Process

- Every time a process starts, the system creates a primary thread.
  - The thread begins execution with the C/C++ run-time library's startup code.
  - The startup code calls your main or WinMain and execution continues until the main function returns and the C/C++ library code calls ExitProcess.

# Scheduling Threads

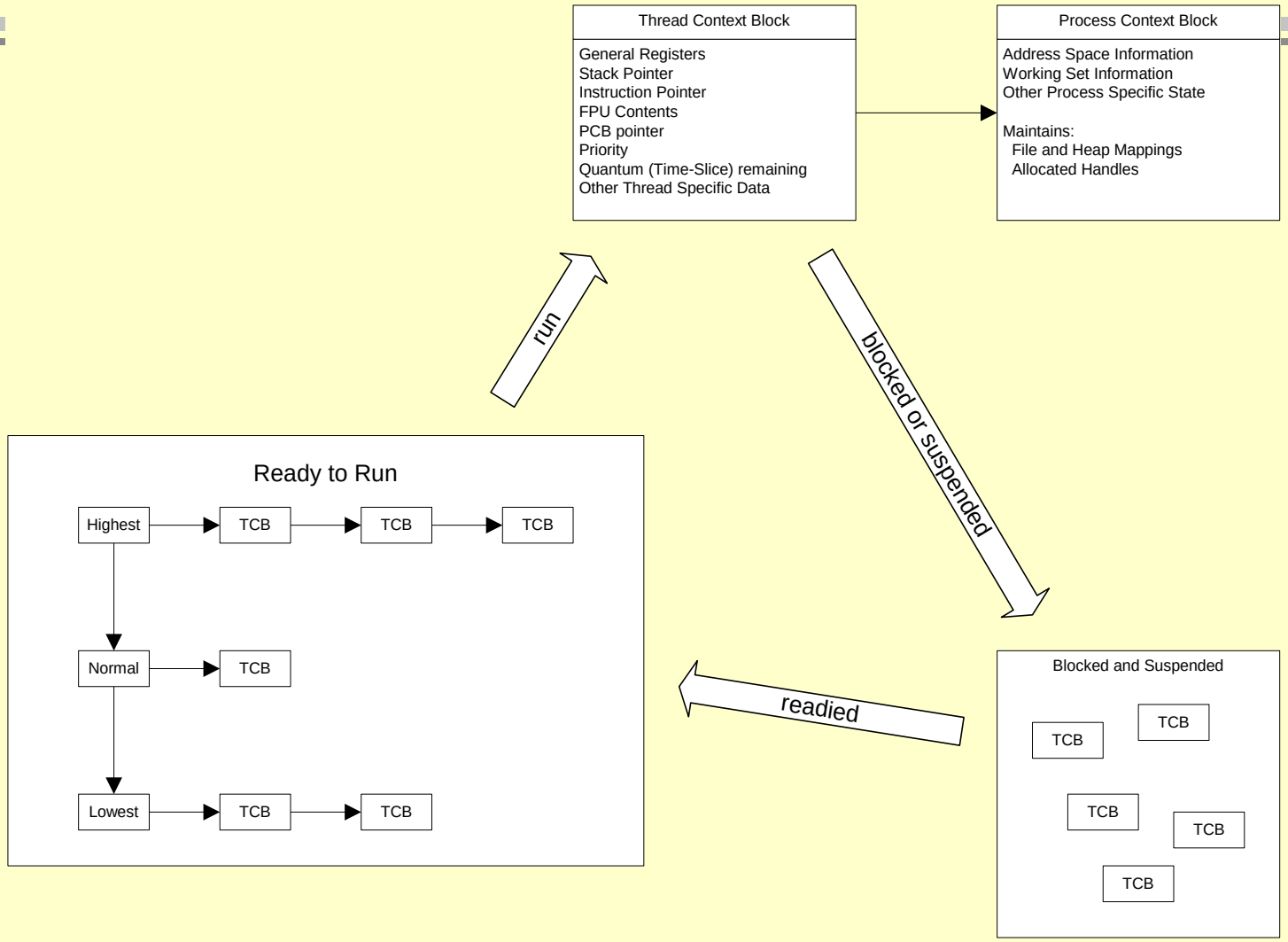
- Windows XP, 2000, NT and Win98 are preemptive multi-tasking systems. Each task is scheduled to run for some brief time period before another task is given control of CPU.

- Threads are the basic unit of scheduling on current Win32 platforms. A thread may be in one of three possible states:
  - running
  - blocked or suspended, using virtually no CPU cycles
  - ready to run, using virtually no CPU cycles

# Scheduling Threads (continued)

- A running task is stopped by the scheduler if:
  - it is blocked waiting for some system event or resource
  - its time time slice expires and is placed back on the queue of ready to run threads
  - it is suspended by putting itself to sleep for some time
  - it is suspended by some other thread
  - it is suspended by the operating system while the OS takes care of some other critical activity.
- Blocked threads become ready to run when an event or resource they wait on becomes available.
- Suspended threads become ready to run when their sleep interval has expired or suspend count is zero.

# Scheduling Threads



# Benefits of using Threads

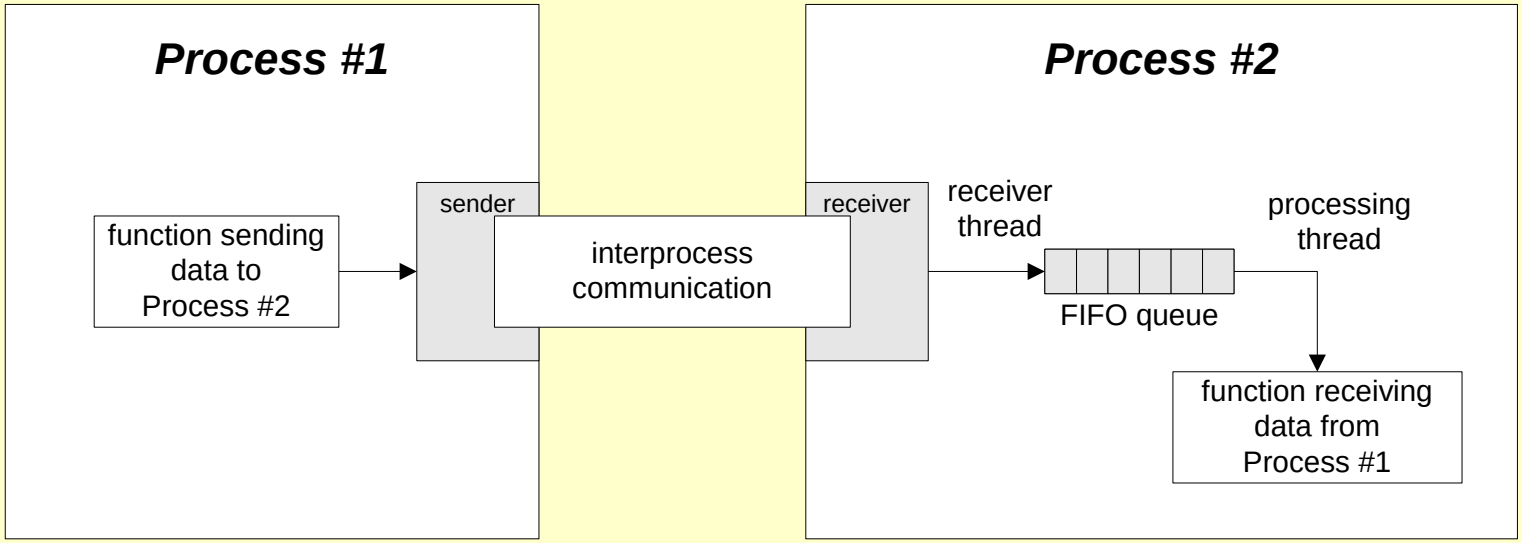
- Keeping user interfaces responsive even if required processing takes a long time to complete.
  - handle background tasks with one or more threads
  - service the user interface with a dedicated thread
- Your program may need to respond to high priority events. In this case, the design is easier to implement if you assign that event handler to a high priority thread.
- Take advantage of multiple processors available for a computation.
- Avoid low CPU activity when a thread is blocked waiting for response from a slow device or human by allowing other threads to continue.

# More Benefits

- Support access to server resources by multiple concurrent clients.
- Improve robustness by isolating critical subsystems on their own threads of control.
- For simulations dealing with several interacting objects the program may be easier to design by assigning one thread to each object.

# Using Threads to Avoid Blocking

***Non-Blocking Communication in Asynchronous System***



# Demonstration Programs

- ProcessDemoWin32
  - Demonstrates creation of Win32 process
- dialogDemo Folder
  - Demonstrates creation of UI and Worker threads

# Potential Problems with Threads

- Conflicting access to shared memory
  - one thread begins an operation on shared memory, is suspended, and leaves that memory region incompletely transformed
  - a second thread is activated and accesses the shared memory in the corrupted state, causing errors in its operation and potentially errors in the operation of the suspended thread when it resumes
- Race Conditions occur when:
  - correct operation depends on the order of completion of two or more independent activities
  - the order of completion is not deterministic
- Starvation
  - a high priority thread dominates CPU resources, preventing lower priority threads from running often enough or at all.

# Problems with Threads (continued)

- Priority inversion
  - a low priority task holds a resource needed by a higher priority task, blocking it from running
- Deadlock
  - two or more tasks each own resources needed by the other preventing either one from running so neither ever completes and never releases its resource

# UI and Worker Threads

- User Interface (UI) threads create windows and process messages sent to those windows
- Worker threads receive no direct input from the user.
  - **Worker threads must not access a window's member functions**. This will often cause a program crash.
  - Worker threads communicate with a program's windows by calling the Win32 API PostMessage and SendMessage functions.
    - Often a program using worker threads will create user defined messages that the worker thread passes to a window to indirectly call some (event-handler) function. Inputs to the function are passed via the message's WPARAM and LPARAM arguments.
    - This is illustrated in the DialogDemo program.

# Creating Win32 Threads

- HANDLE hThrd =
  (HANDLE)_beginthread(ThreadFunc, 0, &ThreadInfo);

- ThreadFunc – the function executed by the new thread

  void _cdecl ThreadFunc(void *pThreadInfo);

- pThreadInfo – pointer to input parameters for the thread

- For threads created with _beginthread the thread function, ThreadFunc, must be a global function or static member function of a class. It can not be a non-static member function.

# Note

- The Win32 API provides a CreateThread(…) function.
- You should not use CreateThread(…) with either C or C++ as it does not properly initialize all the C and C++ libraries for that thread.
- The functions _beginthread(…) and _beginthreadex(…) were designed specifically to work correctly with these libraries.

# Synchronization

- A program may need multiple threads to share some data.
- If access is not controlled to be sequential, then shared data may become corrupted.
  - One thread accesses the data, begins to modify the data, and then is put to sleep because its time slice has expired. The problem arises when the data is in an incomplete state of modification.
  - Another thread awakes and accesses the data, that is only partially modified. The result is very likely to be corrupt data.
- The process of making access serial is called serialization or synchronization.

# Wait For Objects

- ***WaitForSingleObject*** makes one thread wait for:
  - Termination of another thread
  - An event
  - Release of a mutex
  - Syntax: WaitForSingleObject(objHandle, dwMillisec)
- ***WaitForMultipleObjects*** makes one thread wait for the elements of an array of kernel objects, e.g., threads, events, mutexes.
  - Syntax:
    WaitForMultipleObjects(nCount, lpHandles, fwait, dwMillisec)
  - nCount: number of objects in array of handles
  - lpHandles: array of handles to kernel objects
  - fwait: TRUE => wait for all objects, FALSE => wait for first object
  - dwMillisec: time to wait, can be INFINITE

# Process Priority

- IDLE_PRIORITY_CLASS
  - Run when system is idle
- NORMAL_PRIORITY_CLASS
  - Normal operation
- HIGH_PRIORITY_CLASS
  - Receives priority over the preceding two classes
- REAL_TIME_PRIORITY_CLASS
  - Highest Priority
  - Needed to simulate determinism

# Thread Priority

- You use thread priority to balance processing performance between the interfaces and computations.
  - If UI threads have insufficient priority the display freezes while computation proceeds.
  - If UI threads have very high priority the computation may suffer.
  - We will look at an example that shows this clearly.
- Thread priorities take the values:
  - THREAD_PRIORITY_IDLE
  - THREAD_PRIORITY_LOWEST
  - THREAD_PRIORITY_BELOW_NORMAL
  - THREAD_PRIORITY_NORMAL
  - THREAD_PRIORITY_ABOVE_NORMAL
  - THREAD_PRIORITY_HIGHEST
  - THREAD_PRIORITY_TIME_CRITICAL

# Thread Synchronization

- Synchronizing threads means that every access to data shared between threads is protected so that when any thread starts an operation on the shared data no other thread is allowed access until the first thread is done.
- The principle means of synchronizing access to shared data within the Win32 API are:
  - Interlocked increments
    - only for incrementing or decrementing integers
  - Critical Sections
    - Good only inside one process
  - Mutexes
    - Named mutexes can be shared by threads in different processes.
  - Events
    - Useful for synchronization as well as other event notifications.

# Interlocked Operations

- InterlockedIncrement increments a 32 bit integer as an atomic operation. It is guaranteed to complete before the incrementing thread is suspended.

```
long value = 5;
InterlockedIncrement(&value);
```

- InterlockedDecrement decrements a 32 bit integer as an atomic operation:

```
InterlockedDecrement(&value);
```

# Win32 Critical Sections

- Threads within a single process can use critical sections to ensure mutually exclusive access to critical regions of code. To use a critical section you:
  - allocate a critical section structure
  - initialize the critical section structure by calling a win32 API function
  - enter the critical section by invoking a win32 API function
  - leave the critical section by invoking another win32 function.
  - When one thread has entered a critical section, other threads requesting entry are suspended and queued waiting for release by the first thread.

- The win32 API critical section functions are:
  - CRITICAL_SECTION critsec
  - InitializeCriticalSection(&critsec)
  - EnterCriticalSection(&critsec)
  - TryEnterCriticalSection(&critsec)
  - LeaveCriticalSection(&critsec)
  - DeleteCriticalSection(&critsec)

# Win32 Mutexes

- Mutually exclusive access to a resource can be guaranteed through the use of mutexes. To use a mutex object you:
  - identify the resource (section of code, shared data, a device) being shared by two or more threads
  - declare a global mutex object
  - program each thread to call the mutex's acquire operation before using the shared resource
  - call the mutex's release operation after finishing with the shared resource

- The mutex functions are:
  - hMutex = CreateMutex(0,FALSE,0);
  - WaitForSingleObject(hMutex,INFINITE);
  - WaitForMultipleObjects(count,MTXs,TRUE,INFINITE);
  - ReleaseMutex(hMutex);
  - CloseHandle(hMutex);

# Win32 Events

- Events are objects which threads can use to serialize access to resources by setting an event when they have access to a resource and resetting the event when through. All threads use WaitForSingleObject or WaitForMultipleObjects before attempting access to the shared resource.
- Unlike mutexes and semaphores, events have no predefined semantics.
  - An event object stays in the nonsignaled stated until your program sets its state to signaled, presumably because the program detected some corresponding important event.
  - Auto-reset events will be automatically set back to the non-signaled state after a thread completes a wait on that event.
  - After a thread completes a wait on a manual-reset event the event will return to the non-signaled state only when reset by your program.

# Win32 Events (continued)

- Event functions are:
  - HANDLE hEvent = CreateEvent(0,FALSE,TRUE,0);
  - OpenEvent – not used too often
  - SetEvent(hEvent);
  - ResetEvent(hEvent);
  - PulseEvent – not used too often
  - WaitForSingleEvent(hEvent,INFINITE);
  - WaitForMultipleEvents(count,Events,TRUE,INFINITE);

# More Demonstration Programs

- thread problems folder: CSE681\code\threadproblems
  - Illustrates deadlock, race conditions, sharing conflicts, and starvation
- Synch folder: CSE681\code\synch
  - Demonstrates synchronization using Critical Sections, Mutexes, and Events.
  - Also illustrates synchronizing two or more processes with named mutexes.

# MFC Support for Threads

- CWinThread is MFC's encapsulation of threads and the Windows 2000 synchronization mechanisms, e.g.:
  - Events
  - Critical Sections
  - Mutexes
  - Semaphores

# Creating Worker Threads in MFC

- AfxBeginThread – function that creates a thread:

```
CWinThread *pThread
        = AfxBeginThread(ThreadFunc, &ThreadInfo);
```

- ThreadFunc – the function executed by the new thread.

```
AFX_THREADPROC ThreadFunc(LPVOID pThreadInfo)
```

- LPVOID pThreadInfo – a pointer to an arbitrary set of input parameters, often created as a structure.
  - We create a pointer to the structure, then cast to a pointer to void and pass to the thread function.
  - Inside the thread function we cast the pointer back to the structure type to extract its data.

# Creating UI Threads in MFC

- Usually windows are created on the application's main thread.
- You can, however, create windows on a secondary UI thread. Here's how you do that:
  - Create a class, say CUIThread, derived from CWinThread.
  - Use DECLARE_DYNCREATE(CUIThread) macro in the class declaration.
  - Use IMPLEMENT_DYNCREATE(CUIThread, CWinThread) in implementation.
  - Create windows
  - Launch UI thread by calling:

```
CWinThread *pThread =
        AfxBeginThread(RUNTIME_CLASS(CUIThread));
```

# Suspending and Running Threads

- Suspend a thread's execution by calling SuspendThread. This increments a suspend count. If the thread is running, it becomes suspended.

```
pThread -> CWinThread::SuspendThread();
```

- Calling ResumeThread decrements the suspend count. When the count goes to zero the thread is put on the ready to run list and will be resumed by the scheduler.

```
pThread -> CWinThread::ResumeThread();
```

- A thread can suspend itself by calling SuspendThread. It can also relinquish its running status by calling Sleep(nMS), where nMS is the number of milliseconds that the thread wants to sleep.

# Thread Termination

- ThreadFunc returns
  - Worker thread only
  - Return value of 0 a normal return condition code
- WM_QUIT
  - UI thread only
- AfxEndThread( **UINT** *nExitCode* )
  - Must be called by the thread itself
- ::GetExitCode(hThread, &dwExitCode)
  - Returns the exit code of the last work item (thread, process) that has been terminated.

# Thread Safety

- Note that MFC is not inherently thread-safe. The developer must serialize access to all shared data.
- MFC message queues have been designed to be thread safe. Many threads deposit messages in the queue, the thread that created the (window with that) queue retrieves the messages.
- For this reason, a developer can safely use PostMessage and SendMessage from any thread.
- All dispatching of messages from the queue is done by the thread that created the window.
- Also note that Visual C++ implementation of the STL library is not thread-safe, and should not be used in a multi-threaded environment. I hope that will be fixed with the next release of Visual Studio, e.g., Visual Studio.Net.

# MFC Critical Sections

- A critical section synchronizes access to a resource shared between threads, all in the same process.
  - CCriticalSection constructs a critical section object
  - CCriticalSection::Lock() locks access to a shared resource for a single thread.
  - CCriticalSection::Unlock() unlocks access so another thread may access the shared resource

```
CCriticalSection cs;
cs.Lock();
  // operations on a shared resource, e.g., data, an iostream, file
cs.Unlock();
```

# MFC Mutexes

- A mutex synchronizes access to a resource shared between two or more threads. Named mutexes are used to synchronize access for threads that reside in more than one process.
  - CMutex constructs a mutex object
  - Lock locks access for a single thread
  - Unlock releases the resource for acquisition by another thread

```
CMutex cm;
cm.Lock();
  // access a shared resource
cm.Unlock();
```

  - CMutex objects are automatically released if the holding thread terminates.

# MFC Events

- An event can be used to release a thread waiting on some shared resource (refer to the buffer writer/reader example in pages 1018-1021).
- A named event can be used across process boundaries.
- CEvent constructs an event object.
- SetEvent() sets the event.
- Lock() waits for the event to be set, then automatically resets it.

```
CEvent ce;
      :
ce.Lock();    // called by reader thread to wait for writer
      :
ce.SetEvent();   // called by writer thread to release reader
```

# CSingleLock & CMultiLock

- CSingleLock and CMultiLock classes can be used to wrap critical sections, mutexes, events, and semaphores to give them somewhat different lock and unlock semantics.

```
CCriticalSection cs;
CSingleLock slock(cs);
slock.Lock();
   // do some work on a shared resource
slock.Unlock();
```

This CSingleLock object will release its lock if an exception is thrown inside the synchronized area, because its destructor is called. That does not happen for the unadorned critical section.