Multithreading
Multi-Threaded Programming

Prof. Lin Weiguo

Oct. 2015

Note

- You should not assume that an example in this presentation is complete. Items may have been selected for illustration. It is best to get your code examples directly from the textbook or course website and modify them to work. Use the lectures to understand the general principles.

Outline

- Windows Message Processing
- Starting and Terminatinging a Worker Thread
- Messaging among Threads
- Thread Synchronization

Windows Message Processing

- Windows Messages
  - Windows-based applications are event-driven. They do not make explicit function calls to obtain input. Instead, they wait for the system to pass input to them.
  - The system passes input to a window procedure in the form of *messages*. Messages are generated by both the system and applications.
- How a Single-Threaded Program Processes Messages

```
MSG message;
while (::GetMessage(&message, NULL, 0, 0)) {
    ::TranslateMessage(&message);
    ::DispatchMessage(&message);
}
```

MSG structure

- The system sends a message to a ***window procedure*** with a set of four parameters:
  - a window handle
  - a message identifier
  - two values called *message parameters*.

```
typedef struct {
    HWND hwnd;   //identifies the window for which the msg is intended
    UINT message; //a named constant that identifies the purpose of a msg.
    WPARAM wParam; //specify data  or
    LPARAM lParam;    //the location of data when processing a message.
    DWORD time;       //the time when posted
    POINT pt;         //cursor position when posted
} MSG, *PMSG;
```

Window Procedure

- A window procedure is
  - a function that receives and processes all messages sent to the window.
  - Every window class has a window procedure, and every window created with that class uses that same window procedure to respond to messages.
- The system sends a message to a window procedure by passing the message data as arguments to the procedure.
  - The window procedure then performs an appropriate action for the message; it checks the message identifier and, while processing the message, uses the information specified by the message parameters.
- A window procedure does not usually ignore a message.
  - If it does not process a message, it must send the message back to the system for default processing. The window procedure does this by calling the **DefWindowProc** function, which performs a default action and returns a message result.
- To identify the specific window affected by the message, a window procedure can examine the window handle passed with a message.
  - Because a window procedure is shared by all windows belonging to the same class, it can process messages for several different windows.

```
#include <windows.h>
#include <process.h>
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
HWND hwnd ;
int cxClient, cyClient ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow)
{
        static TCHAR szAppName[] = TEXT (”HelloWin") ;
        MSG msg ;
        WNDCLASS wndclass ;
        wndclass.style = CS_HREDRAW | CS_VREDRAW ;
        wndclass.lpfnWndProc = WndProc ;
        ... …
        … …
        if (!RegisterClass (&wndclass))
        {
                MessageBox (NULL, TEXT ("This program requires Windows NT!"), szAppNam, MB_ICONERROR) ;
                return 0 ;
        }
        hwnd = CreateWindow (szAppName, TEXT (” The Hello Program"),"),
                                        WS_OVERLAPPEDWINDOW,
                                        CW_USEDEFAULT, CW_USEDEFAULT,
                                        CW_USEDEFAULT, CW_USEDEFAULT,
                                        NULL, NULL, hInstance, NULL) ;
        ShowWindow (hwnd, iCmdShow) ;
        UpdateWindow (hwnd) ;
        while (GetMessage (&msg, NULL, 0, 0))
        {
                TranslateMessage (&msg) ;
                DispatchMessage (&msg) ;
        }
        return msg.wParam ;
}
```

Programming Windows, Fifth Edition - Charles Petzold. 1998

WndProc

```
LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
        HDC hdc ;
        PAINTSTRUCT ps ;
        RECT rect ;
        switch (message)
        {
                case WM_CREATE:
                        PlaySound (TEXT ("hellowin.wav"), NULL, SND_FILENAME | SND_ASYNC) ;
                        return 0 ;
                case WM_PAINT:
                        hdc = BeginPaint (hwnd, &ps) ;
                        GetClientRect (hwnd, &rect) ;
                        DrawText (hdc, TEXT ("Hello, Windows 98!"), -1, &rect,
                        DT_SINGLELINE | DT_CENTER | DT_VCENTER) ;
                        EndPaint (hwnd, &ps) ;
                        return 0 ;
                case WM_DESTROY:
                        PostQuitMessage (0) ;
                        return 0 ;
                }
        return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

Yielding Control

- What would happen if one of your handler functions were a pig and chewed up 10 seconds of CPU time?
  - It won't be able to process any messages because DispatchMessage won't return until the pig returns.
- Way around:
  - Yield control once in a while by inserting the following instructions inside the main loop.

```
MSG message;
if (::PeekMessage(&message, NULL, 0, 0, PM_REMOVE)) {
    ::TranslateMessage(&message);
    ::DispatchMessage(&message);
}
```

Multi-Threaded Programming

- Process
  - A process is a running program that owns some memory, file handles, and other system resources.
- Threads
  - An individual process can contain separate execution paths, called threads. All of a process's code and data space is available to all of the threads in the process.
- Windows offers two kinds of threads
  - Worker threads and user interface threads.
  - A user interface thread has windows and therefore has its own message loop. A worker thread doesn't have windows, so it doesn't need to process messages.

Writing the Worker Thread Function

- Why threads?
  - Using a worker thread for a long computation is more efficient than using a message handler that contains a PeekMessage call.
- Thread Function
  - A global function or static member function
  - This thread function should return a UINT, and it should take a single 32-bit value (LPVOID) as a parameter. You can use the parameter to pass anything to your thread when you start it.
  - The thread does its computation, and when the function returns, the thread terminates.
  - The thread will also be terminated if the process terminated,

A template of a Thread Function

```
UINT ComputeThreadProc(LPVOID pParam)
{
    // Do thread processing
    return 0;
}
```

Starting the Thread

- To start the thread (with function name ComputeThreadProc):

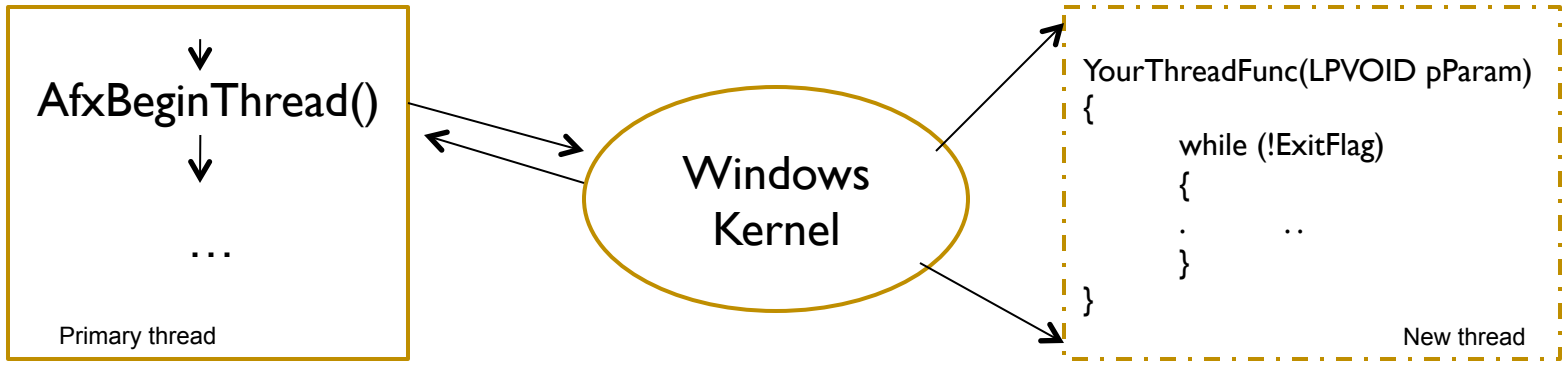
```
CWinThread* pThread =
    AfxBeginThread(
        ComputeThreadProc, // The address of the thread function
        GetSafeHwnd(),  // 32-bit value that passes to the thread function
        THREAD_PRIORITY_NORMAL); // The desired priority of the thread
```

```
CWinThread* pThread =
    AfxBeginThread(
        ComputeThreadProc,
        hTerminateSignal,     // a handle
        THREAD_PRIORITY_BELOW_NORMAL,
        0,  // The desired stack size for the thread
        CREATE_SUSPENDED); // if you want the thread to be created in a
                                        //suspended state.
……
pThread->ResumeThread();
```

AfxBeginThread

```
CWinThread* AfxBeginThread(
        AFX_THREADPROC pfnThreadProc,
        LPVOID pParam,
        int nPriority = THREAD_PRIORITY_NORMAL,
        UINT nStackSize = 0,
        DWORD dwCreateFlags = 0,
        LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL
);
```

```
UINT YourThreadProc(LPVOID pParam);
```



The Running Threads

- The AfxBeginThread function returns immediately;
  - The return value is a pointer to the newly created thread object. You can use that pointer to suspend and resume the thread (CWinThread::SuspendThread and ResumeThread), but the thread object has no member function to terminate the thread.
- Once the worker thread starts, all threads run independently.
  - Windows divides the time among the threads (and among the threads that belong to other processes) according to their priority.
  - If the main thread is waiting for a message, the compute thread can still run.

Terminating a running thread

- Normal Thread Termination
  - For a worker thread, normal thread termination is simple: Exit the controlling function and return a value that signifies the reason for termination.
- *not* a good idea to call TerminateThread.
  - The target thread's initial stack is not freed, causing a resource leak.
- Premature Thread Termination
  - Call AfxEndThread from **within the thread**. Pass the desired exit code as the only parameter.
  - AfxEndThread must be called from within the thread to be terminated.

How the **Main** Thread Talks to a **Worker** Thread

- Does not work: Windows message sent to the worker
  - the worker thread doesn't have a message loop.
- What work:
  - global variables :The simplest way, because all the threads in the process have access to all the globals.

```
UINT ComputeThreadProc(LPVOID pParam)
{
    g_nCount = 0;
    while (g_nCount < 100) {
        // Do some computation here
        ::InterlockedIncrement((long*) &g_nCount);
    }
    return 0;
}
```

  - More , wait…

How the **Worker** Thread Talks to the **Main** Thread

- Check a *global variable* in a loop in main thread? *No*
  - That would waste CPU cycles and stop the program's message processing.
- A *Windows message* is the preferred way for a worker thread to communicate with the main thread because the main thread always has a message loop.
  However , this implies:
  - The main thread has a window (visible or invisible) and
  - The worker thread has *a handle to that window*.

How does the worker thread get the handle?

- A global variable

```
HWND g_hWndMain = GetSafeHwnd();
```

- Pass the handle in the AfxBeginThread call

```
AfxBeginThread(
        ComputeThreadProc, // The address of the thread function
        GetSafeHwnd(),  // 32-bit value that passes to the thread function
        THREAD_PRIORITY_NORMAL); // The desired priority of the thread
```

Note: Why not pass the C++ window pointer instead?
Doing so would be dangerous because you can't depend on the continued existence of the object and you're not allowed to share objects of **MFC classes** among threads. (This rule does not apply to objects derived directly from CObject or to simple classes such as CRect and CString.).

Do you *send* the message or *post* it?

- PostMessage is better
  - The **PostMessage** function places (posts) a message in the message queue associated with the thread that created the specified window and returns without waiting for the thread to process the message.

```
BOOL PostMessage(
   HWND hWnd,      // Handle to the window who is to receive the message.
   UINT Msg,       // the message to be posted.
   WPARAM wParam, // specify data (WORD)  or.
   LPARAM lParam     // the location (LONG) of data in the MSG struct.
);

::PostMessage((HWND)pParam, WM_COMPLETE,0,0);
```

- SendMessage could cause reentry of the main thread's MFC message pump code,
  - Sends the specified message to a window and calls the window procedure for the specified window and does not return until the window procedure has processed the message.

What kind ***Msg*** in the PostMessage do you post?

- Any user-defined message will do.
- Assume you defined a private window message called WM_COMPLETE:
  - Declare user message ID

```
#define       WM_COMPLETE     (WM_USER + 100)
```

  - Declare message handling function in the .h file

```
…
afx_msg LRESULT OnComplete(WPARAM wParam, LPARAM lParam);
DECLARE_MESSAGE_MAP()
```

  - Map the message to the handling function in .cpp file

```
BEGIN_MESSAGE_MAP(CMYDlg, CDialog)
…
    ON_MESSAGE(WM_COMPLETE,OnComplete)
END_MESSAGE_MAP()
```

  - Implement the message handling functon in .cpp file

```
LRESULT CMyDlg::OnComplete(WPARAM wParam, LPARAM lParam)
{   ...   return 0;    }
```

PostMessage inside the thread function

```
UINT ComputeThreadProc(LPVOID pparams)
{
        WaitForSingleObject (pparams-> hTerminateEvent, INFINITE) ;
        lTime = timeGetTime() ;
        PostMessage (pparams->hwnd, WM_COMPLETE, 0, lTime);
}
```

Tips for Parameters of (WPARAM wParam, LPARAM lParam)

- Send/PostMessage with a pointer: not safe

```
//Handle to a global memory block,handled by main process
char *Buffer=new char[100];
strcpy(Buffer, aString, strlen(aString));
::SendMessage((pparams->m_hWnd, WM_COMPLETE, (WPARAM)     0,
(LPARAM)Buffer);
```

- Send/PostMessage with a Handle

```
//Handle to a global memory block,handled by main process
HGLOBAL hShareBuff =     GlobalAlloc (GMEM_MOVEABLE, BufferSize);
BYTE *Buffer   = (BYTE*) GlobalLock (hShareBuff);
memcpy(Buffer,RXBuff, BufferSize);
::SendMessage((pparams->m_hWnd, WM_COMPLETE, (WPARAM) BufferSize,
(LPARAM) hShareBuff);
```

Process the Parameters in the Thread Func

```
LRESULT OnComplete(WPARAM BufferSize, LPARAM hShareBuff)
{
        DWORD dwBytesReceived=(DWORD) BufferSize;
        char  *pShareBuffer = (char*)GlobalLock((HGLOBAL) hShareBuff) ;
        char  *LocalBuffer=new char[dwBytesReceived]
        memcpy(LocalBuffer, pShareBuffer,dwBytesReceived);
        GlobalUnlock((HGLOBAL) hShareBuff);
        GlobalFree((HGLOBAL) hShareBuff);
         ….
        return 0;
}
```

Using Events for Thread Synchronization

- Event
  - An event is one type of kernel object that Windows provides for thread synchronization. (Processes and threads are also kernel objects.)
  - An event is identified by a unique 32-bit handle within a process. It can be identified by name, or its handle can be duplicated for sharing among processes.
  - An event can be either in the signaled (or true) state or in the unsignaled (or false) state.

CEvent MFC class

- #include <afxmt.h>
- *CEvent* class of MFC is derived from *CSyncObject*.
- By default, the constructor creates a Win32 autoreset event object in the unsignaled state.
- When the main thread wants to send a signal to the worker thread, it sets the appropriate event to the signaled state by calling CEvent::
  - CEvent::SetEvent( ) - Sets the event to available (signaled) and releases any waiting threads.
  - CEvent::ResetEvent( ) - Sets the event to unavailable (nonsignaled).

```
CEvent g_eventStart;
CEvent g_eventKill;
```

Win32 Event HANDLE

- CreateEvent function is used to create the event thread synchronization object.

```
HANDLE  m_hShutdownEvent;
m_hShutdownEvent = CreateEvent(NULL, FALSE, FALSE, NULL);

SetEvent (m_hShutdownEvent);

ResetEvent (m_hShutdownEvent);
```

```
HANDLE CreateEvent (
  LPSECURITY_ATTRIBUTES lpsa,
  BOOL bManualReset,
  BOOL bInitialState,
  LPTCSTR lpEventName)
```

WaitForSingleObject in worker thread

- Worker thread must monitor the events and respond when a event is signaled
- Waits until the specified object is in the signaled state or the time-out interval elapses. This function suspends the thread until the specified object becomes signaled.
  - When the thread is suspended, it's not using any CPU cycles, which is good.

```
::WaitForSingleObject(
        g_eventStart,  // the event handle or a CEvent object
        INFINITE);     // the time-out interval. INFINITE means the function will
                       //wait forever until the event becomes signaled
::WaitForSingleObject(g_eventKill,
        0); //If you set the time-out to 0, WaitForSingleObject will return immediately,
            //with a return value of WAIT_OBJECT_0 if the event was signaled.
```

WaitForSingleObject function

- Waits until the specified object is in the signaled state or the time-out interval elapses.

```
DWORD WINAPI WaitForSingleObject(
        _In_  HANDLE hHandle,     //A handle to the object.
        _In_  DWORD dwMilliseconds ); //The time-out interval, in milliseconds.
```

- Return value
  - WAIT_ABANDONED: The specified object is a mutex object that was not released by the thread that owned the mutex object before the owning thread terminated.
  - WAIT_OBJECT_0: The state of the specified object is signaled.
  - WAIT_TIMEOUT: The time-out interval elapsed, and the object's state is nonsignaled.
  - WAIT_FAILED: The function has failed. To get extended error information, call **GetLastError**.

WaitForMultipleObjects in worker thread

```
HANDLE  hEventArray[3];
…
DWORD Event = 0;
// If the 3rd parameter is TRUE, the function returns when the state of all objects in the lpHandles array
is signaled. If FALSE, the function returns when the state of any one of the objects is set to signaled. In
the latter case, the return value indicates the object whose state caused the function to return.
Event = WaitForMultipleObjects(3, hEventArray, FALSE, INFINITE);
switch (Event)
 {
       case WAIT_OBJECT_0 :  …
       case WAIT_OBJECT_0 + 1:  …
       case WAIT_OBJECT_0 + 2: …
}
```

WaitForMultipleObjects function

- Waits until one or all of the specified objects are in the signaled state or the time-out interval elapses.

```
DWORD WINAPI WaitForMultipleObjects(
        _In_  DWORD nCount, //The number of object handles in the array pointed to by lpHandles.
        _In_  const HANDLE *lpHandles, //An array of object handles.
        _In_  BOOL bWaitAll, //If TRUE, the function returns when the state of all objects in
                             //the lpHandles array is signaled.
        _In_  DWORD dwMilliseconds ); //The time-out interval, in milliseconds.
```

- Return value
  - WAIT_OBJECT_0 to (WAIT_OBJECT_0 + nCount– 1): If bWaitAll is FALSE, the return value minus WAIT_OBJECT_0 indicates the lpHandles array index of the object that satisfied the wait.
  - WAIT_ABANDONED_0 to (WAIT_ABANDONED_0 + nCount– 1): the return value indicates that the state of the specified objects is signaled and is an abandoned mutex object.
  - WAIT_TIMEOUT: The time-out interval elapsed and the conditions specified by the bWaitAll parameter are not satisfied.
  - WAIT_FAILED: The function has failed. To get extended error information, call **GetLastError**.

Example worker thread

- Uses two events to synchronize the worker thread with the main thread

```
UINT ComputeThreadProc(LPVOID pParam)
{
   volatile int nTemp;
   ::WaitForSingleObject(g_eventStart, INFINITE);
   TRACE("starting computation\n");
   for (g_nCount = 0; g_nCount < CComputeDlg::nMaxCount;  g_nCount++) {
      for (nTemp = 0; nTemp < 10000; nTemp++)
        {  // Simulate computation }
      if (::WaitForSingleObject(g_eventKill, 0) == WAIT_OBJECT_0) {
         break;
      }
   }
   // Tell owner window we're finished
   ::PostMessage((HWND) pParam, WM_THREADFINISHED, 0, 0);
   g_nCount = 0;
   return 0; // ends the thread
}
```

Example main thread

- Start the thread

```
AfxBeginThread(ComputeThreadProc, GetSafeHwnd());
```

- Set the start event

```
void CComputeDlg::OnBnClickedStart()
{
   GetDlgItem(IDC_START)->EnableWindow(FALSE);
    g_eventStart.SetEvent();
}
```

- Set the terminate event

```
void CComputeDlg::OnBnClickedCancel()
{
    g_eventKill.SetEvent();
}
```

Other Thread Blocking methods

- WaitForSingleObject call is an example of thread blocking.
  - The thread simply stops executing until an event becomes signaled.
- A thread can be blocked in many other ways.
  - Win32 Sleep function:
    - Sleep(500); //put your thread to "sleep" for 500 ms.
  - Many functions block threads, particularly those functions that access hardware devices or Internet hosts.
    - Windows Socket function: Accept()
- You should avoid putting blocking calls in your main user interface thread.
  - Remember that if your main thread is blocked, it can't process its messages, and that makes the program appear sluggish. If you have a task that requires heavy file I/O, put the code in a worker thread and synchronize it with your main thread.

Timer in worker thread

//the while loop will continue sending packets every MT_INTERVAL until the FrameCapture thread signals the g_event_FrameCaptured which means the corresponding response frame has arrived.

```
UINT SendThreadProc( LPVOID pParam )
{
    pcap_sendpacket( … …);          //send a frame out
    while(::WaitForSingleObject(g_event_FrameCaptured,  MT_INTERVAL) != WAIT_OBJECT_0) {
       pcap_sendpacket( … …);       //no response until timeout, so send the frame again, or skip to send the next.
        ……
    }
    return 0;
}

//signal the event when a frame is captured
UINT ReceiveThreadProc( LPVOID pParam )
{
    while(::WaitForSingleObject(g_event_Done,0) != WAIT_OBJECT_0){
      res = pcap_next_ex(……);
      if(res == 0){ continue;}                  //return 0 if timeout, invalid caputre
      else{                                     // a frame captured
           ……
          g_event_FrameCaptured.SetEvent();
           …….
     }
    return 0;
}
```

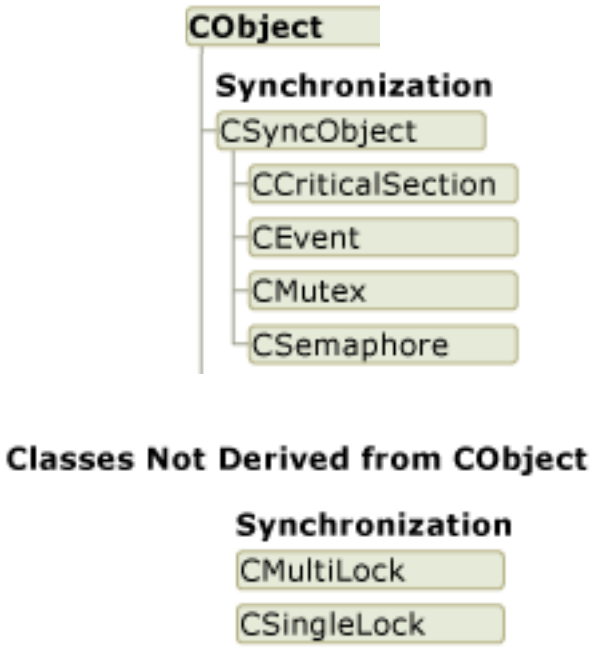
Another Timer Thread Function

```
//a timer for 5 seconds
UINT Timer5s( LPVOID pParam )
{
        int timeInterval = 5000;   // five seconds
        int iRet = WaitForSingleObject( g_hAbortEvent, timeInterval );
        Switch(iRet)
        {
            case WAIT_OBJECT_0: break;
            case WAIT_TIMEOUT:
                    g_hTimeoutEvent. SetEvent(); //Signal timeout
                    break;
        }

        return iRet;
}
```

Synchronization Classes

- The multithreaded classes provided with MFC fall into two categories:
  - Synchronization objects derived from CSyncObject(**CSyncObject** is never used directly. It is the base class for the other four synchronization classes):
    - CSemaphore,
    - CMutex,
    - CCriticalSection
    - CEvent
  - Synchronization access objects
    - CMultiLock
    - CSingleLock



CSyncObject

```
class CSyncObject : public CObject
{
   DECLARE_DYNAMIC(CSyncObject)
   public:
      CSyncObject(LPCTSTR pstrName); // Constructor
      // Attributes
   public:
      operator HANDLE() const;
      HANDLE m_hObject;
      // Operations
      virtual BOOL Lock(DWORD dwTimeout = INFINITE);
      virtual BOOL Unlock() = 0;
      virtual BOOL Unlock(LONG /* lCount */, LPLONG /* lpPrevCount=NULL */)    { return TRUE; }
      // Implementation
   public:
      virtual ~CSyncObject();
      #ifdef _DEBUG
         CString m_strName;
         virtual void AssertValid() const;
         virtual void Dump(CDumpContext& dc) const;
      #endif
      friend class CSingleLock;
      friend class CMultiLock;
};
```

When to Use Which?

- *Synchronization classes* are used when access to a resource must be controlled to ensure integrity of the resource.

```
m_CritSection.Lock();
// resource locked
//usage of shared resource...

m_CritSection.Unlock();
```

- *Synchronization access classes* are used to gain access to these controlled resources.

```
void  f()
{
      CSingleLock singleLock(&m_CritSection);
      singleLock.Lock();
      // resource locked
      //singleLock.Unlock();  //no need. the lock is released  when its scope is exited
}
```

Difference between CSingleLock ::Lock and CSyncObject::Lock

- CCriticalSections are usually (not always) class members or global objects that never go out of scope. So if you use CCriticalSection::Lock() you have to remember to call CCriticalSection::Unlock().
  - This can be tricky to implement if your code can take many divergent paths inside the lock.
- Using CSingleLock makes this much easier as you would normally declare the CSingleLock inside a function or loop or some other small bit of code where it goes out of scope and is destroyed when no longer needed. CSingleLock automatically unlocks the sync object from within its destructor, you do not have to explicitly call CSingleLock::Unlock().

How to determine which is suitable?

- To determine which synchronization class you should use, ask the following series of questions:
  - Does the application have to *wait for something to happen before* it can *access* the resource (for example, data must be received from a communications port before it can be written to a file)?
    - *If yes, use **CEvent**.*
  - Can *more than one thread* within *the same application access* this resource at one time (for example, your application allows up to five windows with views on the same document)?
    - *If yes, use **CSemaphore**.*
  - Can *more than one application use* this resource (for example, the resource is in a DLL)?
    - *If yes, use **CMutex**. Mutex object allows synchronizing objects across the process.*
    - *If no, use **CCriticalSection**. The critical section object synchronizes threads within the process. Critical section allows accessing only one thread at a time.*

Using Three Synchronization Classes

- As an example, take an application that maintains a linked list of accounts.
  - This application allows up to three accounts to be examined in separate windows,
  - but only one can be updated at any particular time. When an account is updated, the updated data is sent over the network to a data archive.
- This example application uses all three types of synchronization classes.
  - Because it allows up to three accounts to be examined at one time, it uses **CSemaphore** to limit access to three view objects. When an attempt to view a fourth account occurs, the application either waits until one of the first three windows closes or it fails.
  - When an account is updated, the application uses **CCriticalSection** to ensure that only one account is updated at a time.
  - After the update succeeds, it signals **CEvent**, which releases a thread waiting for the event to be signaled. This thread sends the new data to the data archive.

Example of using CCriticalSection

- Thread functions

```
// Global Critical section
CCriticalSection c_s;
static int g_C;
```

```
//Thread 1, just replace 1 with 2 for ThreadFunction2
UINT ThreadFunction1(LPVOID lParam)
{
        CSingleLock lock(&c_s); // Create object for Single Lock
        lock.Lock();
        for(int i=0;i<10;i++)
        {
                g_C++;
                cout << "Thread 1 : " << g_C << endl;
        }
        lock.Unlock();
        return 0;
}
```

Example of using CCriticalSection, cont.

- main( )

```
CWinThread *Thread[2];
Thread[0] = AfxBeginThread(ThreadFunction1,LPVOID(NULL));
Thread[1] = AfxBeginThread(ThreadFunction2,LPVOID(NULL));

// Copy to Thread Handle
HANDLE hand[2];
for(int i=0;i<2;i++)
    hand[i] = Thread[i]->m_hThread;

// Wait for complete the child thread , otherwise main thread quit
WaitForMultipleObjects(2,hand,TRUE,INFINITE);
```

Note: *The Critical section object is same as the Mutex object. But the Critical section object does not allow synchronization with different processes. The critical section is used to synchronize the threads within the process boundary.*

Example of using CMutex

- Mutex is the synchronization object used to synchronize the threads with more than one process. The Mutex is as the name tells, mutually exclusive.
- It is possible to use Mutex instead of critical section. But, the critical section is slightly faster compared to other synchronization objects.

```
CMutex g_m;
int g_C;
UINT ThreadFunction1(LPVOID lParam)
{
    CSingleLock lock(&g_m);
    lock.Lock();
    g_C++;// Process
    lock.Unlock();
    return 0;
}
```

Example of using Win32 Mutex Handle

```
HANDLE g_Mutex;
DWORD dwWaitResult;

// Create a mutex with initial owner.
g_Mutex = CreateMutex( NULL, TRUE, "MutexToProtectDatabase");

 // Wait for ownership
dwWaitResult = WaitForSingleObject( g_Mutex, 5000L);
….
ReleaseMutex(g_Mutex)); // Release Mutex
```

Example of using CSemaphore - 1

- CSemaphore, CMutex, and CCriticalSection all work in a similar manner.
- Semaphore is a thread synchronization object that allows accessing the resource for a count between zero and maximum number of threads.
  - If the Thread enters the semaphore, the count is incremented.
  - If the thread completed the work and is removed from the thread queue, the count is decremented.
  - When the thread count goes to zero, the synchronization object is non-signaled. Otherwise, the thread is signaled.

Example of using CSemaphore - 2

- Constructs a named or unnamed CSemaphore object.

```
CSemaphore(
  LONG lInitialCount = 1,      //The initial usage count for the semaphore.
                               // Each time we lock the semaphore, it will decrement the
                               // reference count by 1, until it reaches zero.
  LONG lMaxCount = 1,          // Max threads that can simultaneously access.
  LPCTSTR pstrName = NULL,
  LPSECURITY_ATTRIBUTES lpsaAttributes = NULL
);
```

```
CSemaphore g_sema(4,4);
UINT ThreadFunction1(LPVOID lParam)
{
    CSingleLock lock(&g_sema);
    lock.Lock();
    // Process …
    lock.Unlock();
    return 0;
}
```

Win32 Semaphore Synchronization Object

```
HANDLE g_Semaphore;

// Create a semaphore with initial and max.
g_Semaphore = CreateSemaphore(  NULL,   4, 4,  NULL);

DWORD dwWaitResult;

//take ownership
dwWaitResult = WaitForSingleObject(   g_Semaphore, 0L);

// Check the ownership

// Release Semaphore
ReleaseSemaphore(  g_Semaphore,  1,  NULL) ;
```

CMultiLock

- This class allows you to block, or wait, on up to 64 synchornization objects at once. You create an array of references to the objects, and pass this to the constructor. In addition, you can specify whether you want it to unblock when one object unlocks, or when all of them do.

```
CCriticalSection cs;
CMutex mu;
CEvent ev;
CSemaphore sem[3];
...
CSynObject* objects[6]={&cs, &mu, &ev, &sem[0], &sem[1], &sem[2]};
CMultiLock mlock(objects,6);
int result=mlock.Lock(INFINITE, FALSE);
...
```

Example of Threads Pool – Thread data

```
class THREAD_DATA : public CObject
{
	public:
		HANDLE		hThread;
		HANDLE		hTerminateSignal;
		DWORD		nThreadId;
};
```

```
CList<THREAD_DATA*, THREAD_DATA*> m_ThreadList;

HWND	g_hWndMain = GetSafeHwnd();
```

Example of Threads Pool – Start a thread

```
BOOL StartThreadComputing()
{
    CWinThread* pThread;
    HANDLE hTerminateSignal = CreateEvent(NULL, TRUE, FALSE, NULL);
    pThread = AfxBeginThread(Computingthread, hTerminateSignal,
              THREAD_PRIORITY_BELOW_NORMAL, 0, CREATE_SUSPENDED);
    if (pThread)   {
        HANDLE hThread;
        DuplicateHandle(GetCurrentProcess(), pThread->m_hThread,
                GetCurrentProcess(), &hThread, 0, TRUE, DUPLICATE_SAME_ACCESS);
       THREAD_DATA* pThreadData = new THREAD_DATA;
       pThreadData->hThread = hThread;
       pThreadData->hTerminateSignal = hTerminateSignal;
       pThreadData->nThreadId = pThread->m_nThreadID;
       m_ThreadList.AddTail(pThreadData);
       pThread->ResumeThread();
       return TRUE;
   } else {  AfxMessageBox(_T("Error strarting new adding thread!"));
       CloseHandle(hTerminateSignal);
       return 0;
   }
}
```

Example of Threads Pool – Stop all threads

```
BOOL StopAllThreads()
{
    THREAD_DATA* pThreadData;
    if(m_ThreadList.IsEmpty())
        return FALSE;
    CWaitCursor theWait;
    POSITION pos = m_ThreadList.GetHeadPosition();
    while(pos!=NULL)
    {
        pThreadData = m_ThreadList.GetNext(pos);
        // Signal the thread to terminate.
        SetEvent(pThreadData->hTerminateSignal);
        DWORD dwWait;
        dwWait= MsgWaitForSingleObjects(pThreadData->hThread, INFINITE);
         // Note: The thread will post a terminating message
         // which will remove it from the list
     }
     return TRUE;
}
```

Example of Threads Pool – OnThreadTerminating

```
LRESULT OnThreadTerminating(WPARAM wParam /*Thread ID*/, LPARAM)
{
	THREAD_DATA* pThreadData;
	POSITION pos, prev_pos;
	for(pos = m_ThreadList.GetHeadPosition();pos!=NULL;)
	{
		prev_pos = pos;
		pThreadData = m_ThreadList.GetNext(pos);
		ASSERT_VALID(pThreadData);
		if(pThreadData->nThreadId == wParam)
		{
			CloseHandle(pThreadData->hThread);
			CloseHandle(pThreadData->hTerminateSignal);
			m_ThreadList.RemoveAt(prev_pos);
			delete pThreadData;
			break;
		}
	}
	return 0;
}
```

Example of Threads Pool –Thread function

```
UINT ComputingThread(LPVOID lpvData)
{
    DWORD dwWait; BOOL bStop=FALSE;
    HANDLE hTerminateSignal = (HANDLE)lpvData;
    while(!bStop)
    {
        dwWait = MsgWaitForMultipleObjects(1, &hTerminateSignal, FALSE, 0,
QS_ALLINPUT);
        MSG msg;
        while(PeekMessage(&msg, NULL, NULL, NULL, PM_REMOVE))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
       if(dwWait == WAIT_TIMEOUT)
        {    … /*computing */    }
       }else if(dwWait == WAIT_OBJECT_0) {
            break; // Terminate signal set.  Break out of loop
        } //End Of if(dwWait == WAIT_TIMEOUT)
    }//End Of while(!bStop)
    ::PostMessage(g_hWndMain, MYWM_TERMINATING, GetCurrentThreadId(), 0);
}
```

More Interlocked Functions

- InterlockedIncrement
  - InterlockedIncrement(PLONG Addend);
- InterlockedDecrement
  - InterlockedDecrement(PLONG Addend);
- InterlockedExchange returns the current value of *Target and sets *Target to Value.
  - LONG InterlockedExchange (LPLONG Target, LONG Value)
- InterlockedExchangeAdd adds the second value to the first.
  - LONG InterlockedExchangeAdd (PLONG Addend, LONG Increment)
- InterlockedCompareExchange is similar to InterlockedExchange except that the exchange is done only if a comparison is satisfied.
  - PVOID InterlockedCompareExchange (PVOID *Destination, PVOID Exchange, PVOID Comparand)

References

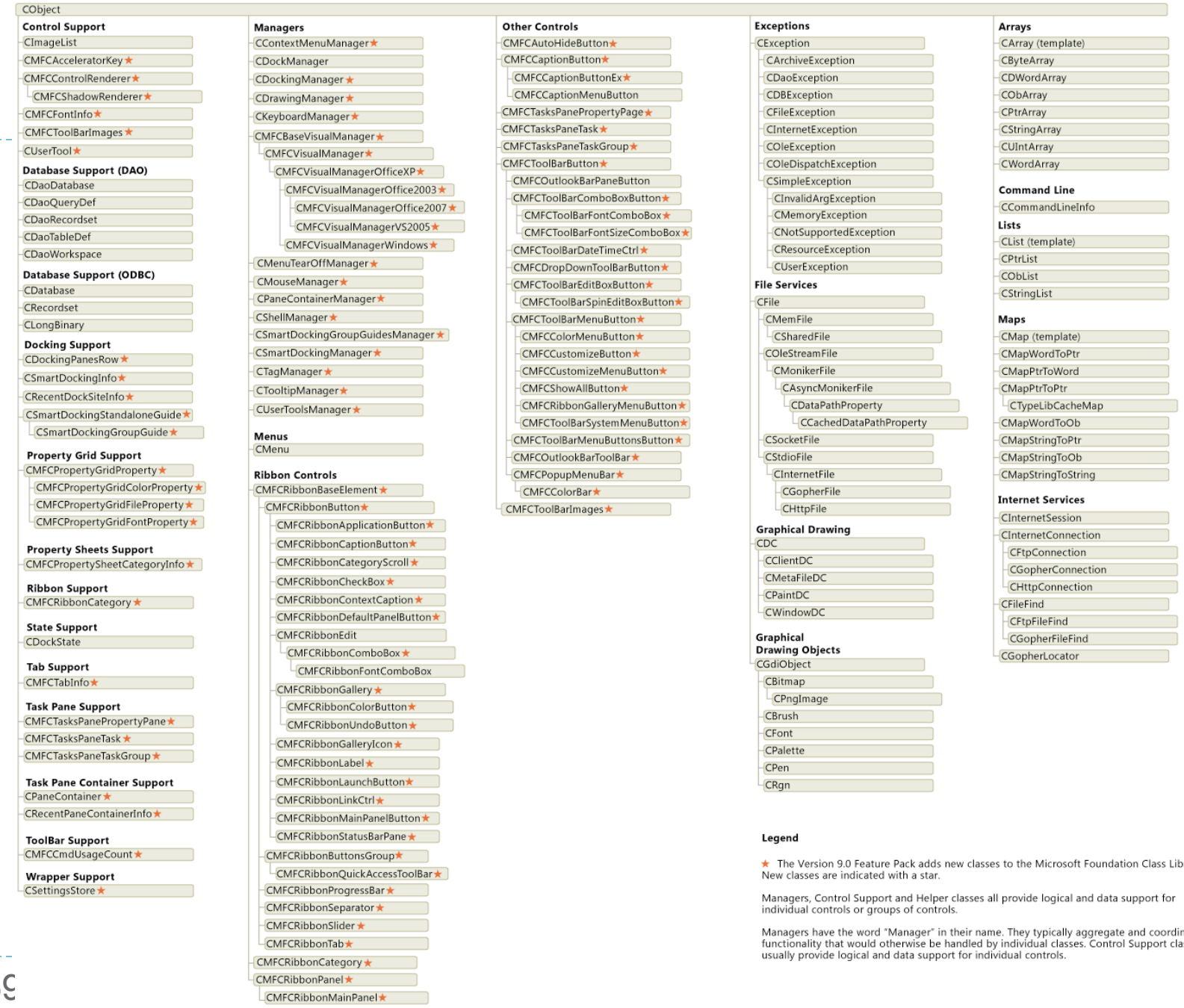
- Chapter 11, Windows Message Processing and Multi-Threaded Programming, Programming With Microsoft Visual C++ NET 6th Ed. - George/Kruglinski Shepherd. 2002
- MSDN: Messages and Message Queues
- MSDN: Multithreading with C++ and MFC
- MSDN: Memory Management Functions
- MSDN: Synchronization
- MSDN: CEvent Class
- Chaper 20, Multitasking and Multithreading, Programming Windows, Fifth Edition - Charles Petzold. 1998
- Chapter 18 – Heaps, Windows Via C/C++ by Jeffrey Richter and Christophe Nasarre Microsoft Press © 2008
- Multithreading Applications in Win32: The Complete Guide to Threads. Jim Beveridge, 1996
- CodeProject: Thread Synchronization for Beginners

  http://www.codeproject.com/KB/threads/Synchronization.aspx?msg=1782286
- Why Worker Threads? www.flounder.com/workerthreads.htm
- The *n* Habits of Highly Defective Windows Applications, http://www.flounder.com/badprogram.htm#TerminateThread

Annex:
MFC Hierarchy Chart
Visual Studio 2010

MFC Hierarchy Chart Part 1 of 3

Classes Derived From CObject



Legend

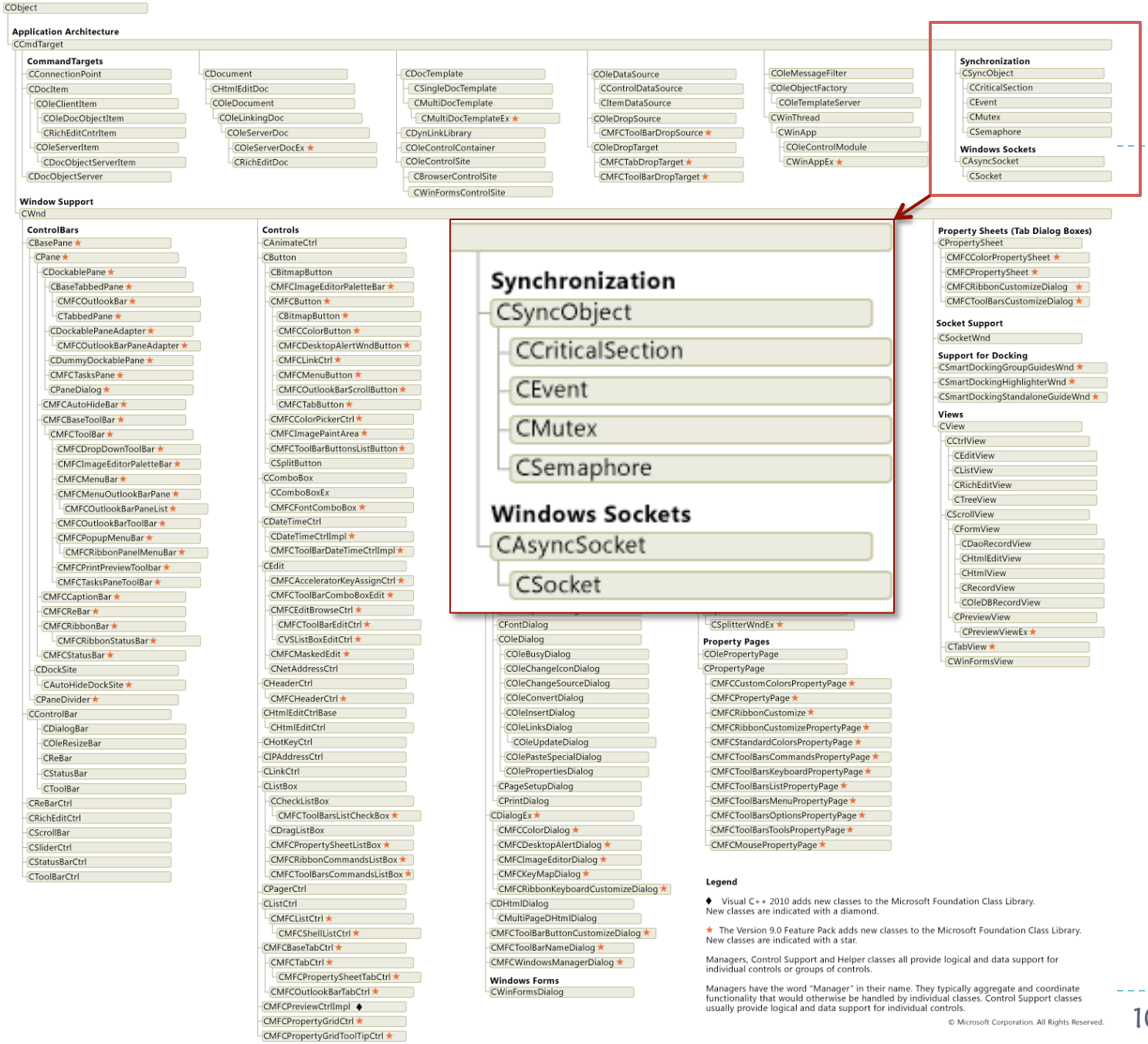
★ The Version 9.0 Feature Pack adds new classes to the Microsoft Foundation Class Library. New classes are indicated with a star.

Managers, Control Support and Helper classes all provide logical and data support for individual controls or groups of controls.

Managers have the word "Manager" in their name. They typically aggregate and coordinate functionality that would otherwise be handled by individual classes. Control Support classes usually provide logical and data support for individual controls.

MFC Hierarchy Chart Part 2 of 3

Classes That Derive From CCmdTarget or CWnd



Classes not derived from CObject