BASIC CONCEPTS

Handle

A mumber that refers to a system object (e.g. window, brush, pen, file, memory block)

Process

Has a virtual address space, executable code, data, object handles, environment variables, a base priority, minimum and maximum working set sizes, and initially a single thread called the "primary thread". Identified by a handle and/or process ID.

Thread

Shares the virtual address space and system resources of the process it is contained in, but has its own exception handlers, priority, and thread context (registers, stack, etc.). Identified by a handle and/or thread ID.

BASIC CONCEPTS

Dynamic Link Library (DLL)

External collection of functions and/or data that can be loaded during load time of an executable, or during runtime via LoadLibrary(). Can optionally have a DllMain() which is called on a process attach and detach, and on a thread attach and detach.

Hungarian Notation

Created by Charles Simonyi who used it in the early 1970s when a doctoral student at Berkeley, and adopted by Microsoft when he went to work for them in 1981. Variable names are prefixed by their type for clarity, and function names indicate their parameters and return types. Called Hungarian Notation both as the variables look as if they are written in a foreign language, and because Simonyi himself was Hungarian. All Microsoft code uses this format.

PROCESS CREATION

```
BOOL CreateProcess(
    LPCTSTR lpApplicationName, LPTSTR lpCommandLine,
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    BOOL bInheritHandles,  DWORD dwCreationFlags,
    LPVOID lpEnvironment, LPCTSTR lpCurrentDirectory,
    LPSTARTUPINFO lpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation)
```

lpApplicationName or lpCommandLine contains the path to the executable image.

lpProcessInformation returns the handles and IDs of the new process and its primary thread.

PROCESS TERMINATION

exit from the main function

End the current process when there is no more code to execute. Attached DLLs are notified of process termination.

VOID ExitProcess(UINT uExitCode)

End the current process when there is no more code to execute. Attached DLLs are notified of process termination.

BOOL TerminateProcess(HANDLE hProcess, UINT uExitCode)

Force another process to exit. Unsafe as the process cannot detect the shutdown and take action, and attached DLLs do not know the process is terminating.

PROCESS IDENTIFICATION

HANDLE GetCurrentProcess()

Obtain a handle to the process executing this function.

DWORD GetCurrentProcessId()

Obtain an identifier for the process which is unique in the system.

THREAD CREATION

```
HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    DWORD dwStackSize,
    LPTHREAD_START_ROUTINE lpStartAddress,
    LPVOID lpParameter, DWORD dwCreationFlags,
    LPDWORD lpThreadId)
```

lpStartAddress is the thread function of the form

```
DWORD WINAPI ThreadProc(LPVOID lpParameter)
```

which receives the lpParameter from CreateThread(). WINAPI is defined as __stdcall, the calling convention of the Win32 API.

CreateThread() returns a handle to the new thread, plus lpThreadId contains the thread ID.

THREAD TERMINATION

exit from the thread function

End the current thread when there is no more code to execute.
Attached DLLs are notified of thread termination.

VOID ExitThread(DWORD dwExitCode)

End the current thread when there is no more code to execute.
Attached DLLs are notified of thread termination.

BOOL TerminateThread(HANDLE hThread, DWORD dwExitCode)

Force another thread to exit. Unsafe as the thread cannot detect the shutdown and take action, and attached DLLs do not know the thread is terminating.

THREAD IDENTIFICATION

HANDLE GetCurrentThread()

Obtain a handle to the thread executing this function.

DWORD GetCurrentThreadId()

Obtain an identifier for the thread which is unique in the system.

SCHEDULING

Preemptive multitasking

On Windows NT multiple CPUs are used if available

32 levels (1-31 accessible, 0 is for system use only)

Set the priority class of a process, then modify the thread priority of the various threads within that class, using:

BOOL SetPriorityClass(HANDLE hProcess,
DWORD dwPriorityClass)

BOOL SetThreadPriority(HANDLE hThread,
int nPriority)

SCHEDULING PRIORITY

dwPriorityClass of SetPriorityClass()	Effect on Priority	Example
IDLE_PRIORITY_CLASS	=4	screensaver
NORMAL_PRIORITY_CLASS (running in the background) (running in the foreground)	 =7 =9	default for a process
HIGH_PRIORITY_CLASS	=13	Task List - want it to be responsive
REALTIME_PRIORITY_CLASS	=24	hardware interfacing, but isn't hard realtime (no guarantees)

SCHEDULING PRIORITY

nPriority of SetThreadPriority()	Effect on Priority
THREAD_PRIORITY_HIGHEST	+2
THREAD_PRIORITY_ABOVE_NORMAL	+1
THREAD_PRIORITY_NORMAL	+0
THREAD_PRIORITY_BELOW_NORMAL	-1
THREAD_PRIORITY_LOWEST	-2
THREAD_PRIORITY_IDLE	=16 (if REALTIME_P_C) =1 (otherwise)
THREAD_PRIORITY_TIME_CRITICAL	=31 (if REALTIME_P_C) =15 (otherwise)

SYNCHRONIZATION - WAIT FOR COMPLETION

To block until a process or thread exits, use the wait functions.

DWORD WaitForSingleObject(HANDLE hHandle,
    DWORD dwMilliseconds)

The return value is the result of the wait:

WAIT_OBJECT_0 if the handle was signaled (the process or thread terminated)
WAIT_ABANDONED_0 if the handle "disappeared"
WAIT_TIMEOUT if the timeout interval was exceeded

The timeout can be INFINITE to wait forever. The timeout can also be 0, causing the state of the handle to be tested and the function immediately return.

SYNCHRONIZATION - WAIT FOR COMPLETION

To wait on more than one handle at a time, use this function:

```
DWORD WaitForMultipleObjects(DWORD nCount,
    CONST HANDLE *lpHandles, BOOL fWaitAll,
    DWORD dwMilliseconds)
```

The return value is again the result of the wait, though offset by the handle that triggered the return (e.g. WAIT_OBJECT_0+3 if the third handle in the array was signaled).

The size of the array is limited by MAXIMUM_WAIT_OBJECTS (currently a system limit of 64).

If fWaitAll is false, the first handle that is signaled causes the function to return. If fWaitAll is true, all handles must be signaled for the function to return.

SYNCHRONIZATION - MUTUAL EXCLUSION BY CRITICAL SECTION

Fast (~24ms on PII-450), but only between threads in a single process.
Allows one thread access to the critical region at one time.

Create:
VOID InitializeCriticalSection(LPCRITICAL_SECTION lpCS)

Enter critical region:
VOID EnterCriticalSection(LPCRITICAL_SECTION lpCS)
BOOL TryEnterCriticalSection(LPCRITICAL_SECTION lpCS)

Exit critical region:
VOID LeaveCriticalSection(LPCRITICAL_SECTION lpCS)

Destroy:
VOID DeleteCriticalSection(LPCRITICAL_SECTION lpCS)

SYNCHRONIZATION - MUTUAL EXCLUSION BY MUTEX

Slow (~444ms on PII-450), but can be used by threads in different processes. Allows one thread access to the critical region at one time.

Create:

HANDLE CreateMutex(LPSECURITY_ATTRIBUTES lpAttr,
    BOOL bInitialOwner, LPCTSTR lpName)
HANDLE OpenMutex(DWORD dwDesiredAccess,
    BOOL bInheritHandle, LPCTSTR lpName)

bInitialOwner is true if the mutex is created in the acquired state.

lpName non-NULL allows the mutex to be accessed by other processes in the system by its name.

SYNCHRONIZATION - MUTUAL EXCLUSION BY MUTEX

Enter critical region:
WaitForSingleObject()/WaitForMultipleObjects()

Exit critical region:
BOOL ReleaseMutex(HANDLE hMutex)

Destroy:
BOOL CloseHandle(HANDLE hMutex)

SYNCHRONIZATION - MUTUAL EXCLUSION BY SEMAPHORE

Slow (~441ms on PII-450), but can be used by threads in different processes. Allows multiple threads based on the semaphore count access to the critical region at one time.

Create:

```
HANDLE CreateSemaphore(LPSECURITY_ATTRIBUTES lpAttr,
    LONG lInitialCount, LONG lMaximumCount,
    LPCTSTR lpName)
HANDLE OpenSemaphore(DWORD dwDesiredAccess,
    BOOL bInheritHandle, LPCTSTR lpName)
```

The counts control the state of the semaphore.

lpName non-NULL allows the semaphore to be accessed by other processes in the system by its name.

SYNCHRONIZATION - MUTUAL EXCLUSION BY SEMAPHORE

Enter critical region:
WaitForSingleObject()/WaitForMultipleObjects()

Exit critical region:
BOOL ReleaseSemaphore(HANDLE hSemaphore,
LONG lReleaseCount, LPLONG lpPreviousCount)

Usually lReleaseCount=1, but it does not need to be

Destroy:
BOOL CloseHandle(HANDLE hSemaphore)

SYNCHRONIZATION - SIGNAL BY EVENT

To "raise a flag" use an event. Events are boolean (signaled or nonsignaled) and can be used by threads in different processes.

Create:

```
HANDLE CreateEvent(LPSECURITY_ATTRIBUTES lpAttr,
    BOOL bManualReset, BOOL bInitialState, LPCTSTR lpName)
HANDLE OpenEvent(DWORD dwDesiredAccess,
    BOOL bInheritHandle, LPCTSTR lpName)
```

If bManualReset is true, the event is a "manual-reset event." When signaled, all threads waiting on the event will be woken. A manual-reset event must be reset to a nonsignaled state by the programmer.

If bManualReset is false, the event is an "auto-reset event." When signalled, only one thread waiting on the event will be woken. An auto-reset event will automatically reset to a nonsignaled state.

SYNCHRONIZATION - SIGNAL BY EVENT

Block until an event is signaled:
WaitForSingleObject()/WaitForMultipleObjects()

Change to signaled state:
BOOL SetEvent(HANDLE hEvent)

Change to nonsignaled state:
BOOL ResetEvent(HANDLE hEvent)

Change to signaled state, then back to nonsignaled state:
BOOL PulseEvent(HANDLE hEvent)

Destroy:
BOOL CloseHandle(HANDLE hEvent)

NOT QUITE SO SIMPLE

Applications for Windows are in two styles:

Console mode - as a usual C/C++ program for UNIX. Its first function is main() and it cannot display graphical windows.

Windows - more like a UNIX program with X-Window support. Messages are processed and Windows GUI functions are available. Its first function is WinMain(), but does not have a console (where messages printed to stdout can be displayed) unless one is explicitly allocated.

Applications that must process messages can have user interfaces that "lock up" if WaitForSingleObject() or WaitForMultipleObjects() is called. MsgWaitForMultipleObjects() must be used instead to allow Windows messages, in addition to signaled handles, to cause the wait to end.

NOT QUITE SO SIMPLE

If the C library is being used, the wrapper functions _beginthread() [or _beginthreadex()] and _endthread() [or _endthreadex()] must be used instead of CreateThread() and ExitThread() directly.

Various buffers (such as the one used for the internal operation of strtok()) when a multithreaded application is compiled are allocated on a per-thread basis. These structures are not freed unless the wrapper functions are used.

Applications that create multiple threads must link with the multithreaded libraries. Not using these libraries will cause compilation to fail at best, or incorrect application behavior at worst.