Using Selected Windows Base System Services

Contents

- Using the Win32 API
  - Calling API functions
  - Handling errors
  - Accessing objects by handles

- Asynchronous I/O

- I/O Completion Ports and Thread Pools

- Synchronization
  - Synchronization Objects
  - Wait Functions
  - Mutexes, Semaphores, Events, and Timer
  - Interlocked Functions
  - Critical Sections
Events
Win32-API Functions for Events
Waitable Timer
Win32-API Functions for Waitable Timer
Time Formats (1)
Time Formats (2)
Converting Times
Interlocked Functions
Critical Sections
Functions for Critical Sections
Slim Reader Writer Locks
Functions for SRW locks
Literature


The .NET Framework is layered on top of the Windows API. For a good description see

Win32 API

- Thousands of functions in seven main groups:
  - Base System Services
  - Component Services
  - User Interface Services
  - Graphics and Multimedia Services
  - Messaging and Collaboration
  - Networking
  - Web Services

- The Base System Services are not the „real“ system calls into the operating system. The latter have not been published by Microsoft.

- The Win32 API is documented in the Microsoft Developer's Network (MSDN) Library.
- Include the header file \texttt{windows.h} in your programs.
Tips for Using Win32 API functions

- Always use the special datatypes given in the documentation of the functions, e.g. DWORD.

- The documentation of a function also tells how to find out whether the function completed successfully or not. For most functions, this information is contained in the return value.

- After calling an API function
  - immediately check whether an error occurred. If so,
  - get the error code of the error by calling the function DWORD GetLastError(VOID).
  - The last error value will be overwritten by the next call (explicit or implicit) to a Win32 API function.

- The texts associated with the error codes returned by GetLastError
  - are listed in the header file WinError.h
  - can be looked up in Visual Studio by using the Error Lookup Tool,
  - can be output by the program by using the FormatMessage function (see example on next page).

Example for the Use of FormatMessage

```c
wchar_t  lpMsgBuf[200];

FormatMessage(  
    FORMAT_MESSAGE_FROM_SYSTEM, // Use the System Message Table  
    NULL,  
    GetLastError(),  
    0,  
    lpMsgBuf,  
    200,  
    NULL  
  );

wprintf(L"%s\n", lpMsgBuf);
```

Handles to Objects

- Win32 API functions are the methods for manipulating kernel objects.
- Objects are accessed by specifying a handle to it (data type: HANDLE).
- Handles
  - are returned to the caller of a Create_xyz function, when the object is newly created,
  - can be obtained by „opening“ an existing object with an Open_xyz function,
  - are stored in a process-wide handle table, and are thus available to all threads of the process,
  - are removed from the handle table by calling CloseHandle.
  - When the last handle to an object is closed, the object will be deleted.
- Pseudohandles to the current process resp. the current thread
  - can be obtained with
    HANDLE GetCurrentProcess(VOID) resp.
    HANDLE GetCurrentThread(VOID)
  - need not (and cannot) be closed.

Asynchronous I/O
Overview of Asynchronous I/O

- Asynchronous I/O must be requested when opening a file.
- Each read or write operation needs an OVERLAPPED data structure which – among other things – contains the file pointer, indicating where in the file the operation should be performed.

Remark: In Windows, asynchronous I/Os are often referred to as overlapped I/Os.

- There are several ways of synchronizing with the completion of the I/O:
  - On I/O completion, the file object will be signalled.
  - The OVERLAPPED structure may contain a handle to an event object, which will also be signalled on I/O completion.
  - Hence, a thread may synchronize with either the file or event object, i.e. wait for the object to go to the signalled state (e.g. using WaitForSingleObject).
  - There are further possibilities, which will not be discussed here.

Requesting Asynchronous I/O

- The CreateFile function is used both for creating a new, and for opening an existing file.
- The FILE_FLAG_OVERLAPPED flag requests asynchronous I/O operations.

Example: Creating a file for asynchronous writing (or overwriting, if the file already exists):

```c
HANDLE hFile;
if (INVALID_HANDLE_VALUE == (hFile = CreateFile("MYFILE.TXT",
                                            GENERIC_WRITE,
                                            0,
                                            NULL,
                                            CREATE_ALWAYS,
                                            FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED,
                                            NULL)))
{
    ErrorHandler("Could not open file.");
    return;
}
```
**The OVERLAPPED Structure**

```c
typedef struct _OVERLAPPED { // o
    DWORD Internal;
    DWORD InternalHigh;
    DWORD Offset;
    DWORD OffsetHigh;
    HANDLE hEvent;
} OVERLAPPED;
```

- Before asynchronously reading from or writing to a file
  - specify the 64-bit file pointer in Offset and OffsetHigh.
  - specify a handle to an event that is to be notified upon I/O completion, or NULL.
  - The fields Internal and InternalHigh will be filled in by the system when the I/O completes.

**The Functions ReadFile and WriteFile**

```c
BOOL ReadFile(
    HANDLE hFile, // handle of file to read from
    LPVOID lpBuffer, // pointer to buffer that is to receive data
    DWORD nNumberOfBytesToRead, // number of bytes to read
    LPDWORD lpNumberOfBytesRead, // pointer to number of bytes read
    LPOVERLAPPED lpOverlapped // pointer to Overlapped structure
);

BOOL WriteFile(
    HANDLE hFile, // handle of file to write to
    LPVOID lpBuffer, // pointer to data to write to file
    DWORD nNumberOfBytesToWrite, // number of bytes to write
    LPDWORD lpNumberOfBytesWritten, // pointer to number of bytes written
    LPOVERLAPPED lpOverlapped // pointer to Overlapped structure
);
```
Reading and Writing Asynchronously

- The pointer to the initialized OVERLAPPED structure must be specified.
- Use NULL for the parameters `lpNumberOfBytesRead` resp. `lpNumberOfBytesWritten`.
- The return value is
  - TRUE, when the I/O was completed synchronously,
  - FALSE, when the I/O was started asynchronously (or the I/O request failed). `GetLastError` will return `ERROR_IO_PENDING` (unless some other error occurred).
- When a read operation encounters an end-of-file condition
  - either the return value is FALSE and `GetLastError` returns `ERROR_HANDLE_EOF`,
  - or `LastError` is `ERROR_IO_PENDING`, and the condition will be detected later when evaluating the results of the I/O (see below).

Synchronizing with I/O Completion

- When an asynchronous I/O completes, the kernel
  - signals the device object,
  - signals the event object specified in the OVERLAPPED structure (if any).
- You can synchronize with an object, i.e. wait for it to go to the signalled state, using
  ```
  DWORD WaitForSingleObject(
      HANDLE hHandle,     // handle of object to wait for
      DWORD dwMilliseconds); // time-out interval in milliseconds
  ```
  You may also specify 0 or `INFINITE` for the timeout interval.
- The fastest method to check whether an I/O is complete is by using the macro
  ```
  BOOL HasOverlappedIoCompleted(
      LPOVERLAPPED lpOverlapped);
  ```
  which expands to
  ```
  (lpOverlapped)->Internal != STATUS_PENDING
  ```
Obtaining the Result of an Asynchronous I/O

- The following function both:
  - checks whether an I/O is complete, and
  - returns the number of bytes transferred

  ```c
  BOOL GetOverlappedResult(
    HANDLE hFile,          // handle of file
    LPOVERLAPPED lpOverlapped, // address of overlapped structure
    LPDWORD lpNumberOfBytesTransferred, // address of actual bytes count
    BOOL bWait           // wait flag
  );
  ```

- If the I/O is not yet complete, the function either blocks (if bWait = TRUE), or returns FALSE, and GetLastError returns ERROR_IO_INCOMPLETE.

- If the I/O is a read operation which encountered an end-of-file condition, the function returns FALSE, and GetLastError returns ERROR_HANDLE_EOF.

Using APCs with Asynchronous I/Os (1)

- Another way of post-processing an asynchronous I/O is by specifying an APC (asynchronous procedure call) routine that is to be executed after I/O completion.

- In this case, the I/O request must be done with one of the functions

  ```c
  BOOL WINAPI ReadFileEx(
    __in HANDLE hFile,
    __out_opt LPVOID lpBuffer,
    __in DWORD nNumberOfBytesToRead,
    __inout LPOVERLAPPED lpOverlapped,
    __in_opt LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine);
  ```

  ```c
  BOOL WINAPI WriteFileEx(
    __in HANDLE hFile,
    __in_opt LPCVOID lpBuffer,
    __in DWORD nNumberOfBytesToWrite,
    __inout LPOVERLAPPED lpOverlapped,
    __in_opt LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine);
  ```
Using APCs with Asynchronous I/Os (2)

- **LPOVERLAPPED_COMPLETION_ROUTINE** must be declared as follows:
  - `VOID CALLBACK FileIOCompletionRoutine(
    __in DWORD dwErrorCode,
    __in DWORD dwNumberOfBytesTransfered,
    __in LPOVERLAPPED lpOverlapped );`

- The completion routine
  - will be entered into the thread’s APC queue when the I/O completes,
  - will (only) be executed when the thread is in an **alertable wait state**.

- The following functions cause a thread to enter an alertable wait state:
  - `SleepEx`,
  - `WaitForSingleObjectEx`,
  - `WaitForMultipleObjectsEx`,
  - `SignalObjectAndWait`,
  - `MsgWaitForMultipleObjectsEx`

Further Remarks on APCs

- The APC mechanism is also available for
  - sockets,
  - named pipes,
  - waitable timer.

- An APC routine can also explicitly be queued to a thread (independently of I/O completion or timer expiration) with the function
  - `DWORD WINAPI QueueUserAPC(
    __in PAPCFUNC pfnAPC,
    __in HANDLE hThread,
    __in ULONG_PTR dwData );`

- Windows itself can also post a kernel-mode APC to a thread, which will be executed immediately (independently of the thread state).
I/O Completion Ports and Thread Pools

The Idea Behind I/O Completion Ports (1)

- An I/O completion port maintains
  - a queue of "work items" that need to be processed,
  - a queue of threads waiting for processing work items,
  - a concurrency value, indicating the desired maximum number of active threads,
  - a counter for the number of threads actively processing work items (states ready, or running),
  - a counter for the number of threads processing work items, but which are currently blocked.

- The system
  - deals out work items to the threads, initially at most as many as the concurrency value specifies,
  - keeps track of the state of threads currently processing work items,
  - deals out another work item to a further thread when a currently active thread blocks,
  - does not block any threads when a blocked thread becomes ready again, so at times there may be more active threads than the concurrency value specifies.
The Idea Behind I/O Completion Ports (2)

- “Work items”
  - typically are I/O completion notification packets queued on completion of an asynchronous I/O to a device (e.g., file, or socket) that was associated with the port.
  - can be a callback function that is to be executed when a certain object goes to the signalled state (e.g., an event, or a timer).
  - can be any other work item (function) explicitly queued to the completion port.

- The use of I/O completion ports
  - simplifies the writing, and
  - improves the performance of multi-threaded applications by
    - maximizing parallelism on SMP systems,
    - reducing the number of thread creations, and thread deletions (by reusing already existing threads),
    - reducing the number of context switches.

Creating an I/O Completion port, and Associating Files with it

```
HANDLE WINAPI CreateIoCompletionPort(
    __in HANDLE FileHandle,
    __in_opt HANDLE ExistingCompletionPort,
    __in ULONG_PTR CompletionKey,
    __in DWORD NumberOfConcurrentThreads );
```

- Creates a new I/O completion port (if ExistingCompletionPort is NULL).
- Associates the "file" specified by FileHandle with the (existing or newly created) I/O completion port. The handle must have been opened for overlapped I/O.
- CompletionKey will be included in every completion packet resulting from an I/O to FileHandle.
  - Use different CompletionKey values for different files associated with the same port.
- NumberOfConcurrentThreads is the maximum number of threads that should actively process I/O completion packets.
  - If this parameter is 0, NumberOfConcurrentThreads will be set to the number of processors in the system.
Retrieving a Completion Packet from a Completion Port

```c
BOOL WINAPI GetQueuedCompletionStatus(
    __in HANDLE CompletionPort,
    __out LPDWORD lpNumberOfBytes,
    __out PULONG_PTR lpCompletionKey,
    __out LPOVERLAPPED *lpOverlapped,
    __in DWORD dwMilliseconds);
```

**Remark:** GetQueuedCompletionStatusEx (available as of Vista / Server 2008) can retrieve multiple completion port entries simultaneously.

---

Queuing a Work Item to an I/O Completion Port

```c
BOOL WINAPI QueueUserWorkItem(
    __in LPTHREAD_START_ROUTINE Function,
    __in_opt PVOID Context,
    __in ULONG Flags);
```

- **LPTHREAD_START_ROUTINE Function** must be declared as
  - `DWORD WINAPI Function ( __in LPVOID lpParameter );`
    (same type as in CreateThread)
- **Context** is a single parameter value to be passed to the thread function.
- See the documentation (MSDN) for the supported flag values.
Making a Worker Thread Wait for Something

```
BOOL WINAPI RegisterWaitForSingleObject(
    __out PHANDLE phNewWaitObject,
    __in HANDLE hObject,
    __in WAITORTIMERCALLBACK Callback,
    __in_opt PVOID Context,
    __in ULONG dwMilliseconds,
    __in ULONG dwFlags);
```

- WAITORTIMERCALLBACK Callback must be declared as
  VOID CALLBACK Callback(
    __in PVOID lpParameter,
    __in BOOLEAN TimerOrWaitFired);

- hObject can be a handle to a
  - Synchronization object (process, thread, mutex, semaphore, event, waitable timer)
  - Change notification (indicating a change in a directory)
  - Console input (indicating that there is unread input)
  - Memory resource notification (indicating a memory low, or memory high condition)

---

Thread Pools (as of Vista / Server 2008)

- Windows Vista / Server 2008
  - introduced a new kernel object called worker factories,
  - offers a new Thread Pooling facility based on worker factories,
  - introduced a new API for working with thread pools.

- Thread Pools
  - require and use an I/O completion port,
  - can be permanent with a fixed number of threads,
  - can be dynamic, with the system creating and deleting worker threads in order to maximize parallelism, and minimize overhead,
  - support the new Vista/Server 2008 feature of dynamic processors.
Thread Pool API

- New, more consistent interface with an object-based design, using the object types
  - Pool object: Pool of threads to execute callbacks
  - Callback Environment: Binds a thread pool to its callback objects
  - Clean-up group: Tracks thread pool callback objects to simplify their clean-up (optional)
  - Work object: Invokes a callback function asynchronously
  - Timer object: Invokes a callback function at a due time
  - Wait object: Invokes a callback function when a kernel object is signaled
  - I/O object: Invokes a callback function when an asynchronous I/O completes

- New API functions for "traditional" I/O completion port functionality, for example
  - QueueUserWorkItem is replaced by SubmitThreadpoolWork
  - RegisterWaitForSingleObject is replaced by SetThreadpoolWait

Before, objects ThreadpoolWork resp. ThreadpoolWait have to be created.

Creating a Thread Pool

- Every process has a default thread pool with default parameters (e.g. a thread maximum of 500).
- Additional thread pools can be created with
  
  ```
  PTP_POOL WINAPI CreateThreadpool( _Reserved_ PVOID reserved );
  (the reserved parameter must be NULL)
  ```

- The number of threads in the pool can be limited with the functions
  ```
  VOID WINAPI SetThreadpoolThreadMaximum( _Inout_ PTP_POOL ptpp,
  _In_ DWORD cthrdMost ); // Default is 500
  ```
  ```
  BOOL WINAPI SetThreadpoolThreadMinimum( _Inout_ PTP_POOL ptpp,
  _In_ DWORD cthrdMic ); // Default is 0
  ```

- A thread pool can be released with the function
  ```
  VOID WINAPI CloseThreadpool( _Inout_ PTP_POOL ptpp );
  ```
Creating a Thread Pool Environment

- You can use the default thread pool without explicitly creating a thread pool environment.
- A different Thread Pool Environment can be created with
  \[
  \text{VOID InitializeThreadpoolEnvironment(}_\text{Out}_\text{ PTP\_CALLBACK\_ENVIRON pcbe });
  \]
- You can modify a thread pool environment with (among others)
  \[
  \begin{align*}
  &\text{VOID SetThreadpoolCallbackPool(}_\text{InOut}_\text{ PTP\_CALLBACK\_ENVIRON pcbe,}\\
  &\hspace{1em} _\text{In}_\text{ PTP\_POOL ptpp }; \quad \text{Associates the callback environment with a pool other}\\
  &\hspace{1em} \text{than the default pool}\\
  &\text{SetThreadpoolCallbackCleanupGroup (after calling CreateThreadpoolCleanupGroup(void);)}\\
  &\text{SetThreadpoolCallbackPriority Priority of callback functions in this environment relative to}\\
  &\hspace{1em} \text{other work items in the same thread pool.}\\
  &\text{SetThreadpoolCallbackRunsLong Indicate that callbacks may not return quickly. This}\\
  &\hspace{1em} \text{information is used by the system to determine when}\\
  &\hspace{1em} \text{new threads should be created.}\\
  \end{align*}
  \]
- \[
  \text{VOID DestroyThreadpoolEnvironment(}_\text{InOut}_\text{ PTP\_CALLBACK\_ENVIRON pcbe });
  \]

Work Objects

- The following function creates a work object:
  \[
  \text{PTP\_WORK WINAPI CreateThreadpoolWork(}\\
  \hspace{1em} _\text{In}_\text{ PTP\_WORK\_CALLBACK pfnwk,}\\
  \hspace{1em} _\text{Inout\_opt}_\text{ PVOID pv,} \quad \text{// parameter passed to callback}\\
  \hspace{1em} _\text{In\_opt}_\text{ PTP\_CALLBACK\_ENVIRON pcbe });
  \]
- The signature of the callback function is
  \[
  \text{VOID CALLBACK WorkCallback(}_\text{Inout\_opt}_\text{ PTP\_CALLBACK\_INSTANCE Instance,}\\
  \hspace{1em} _\text{Inout\_opt}_\text{ PVOID Context,} \quad \text{// parameter passed to callback}\\
  \hspace{1em} _\text{Inout}_\text{ PTP\_WORK Work });
  \]
  where WorkCallback is a placeholder for the application-defined function name.
- A work object is freed with
  \[
  \text{VOID WINAPI CloseThreadpoolWork(}_\text{Inout}_\text{ PTP\_WORK pwk });
  \]
Wait Objects

- The following function creates a wait object:

  ```
  PTP_WAIT WINAPI CreateThreadpoolWait( _In_ PTP_WAIT_CALLBACK pfnwa,
  _Inout_opt_ PVOID pv, // parameter passed to callback
  _In_opt_ PTP_CALLBACK_ENVIRON pcbe );
  ```

- The signature of the callback function is

  ```
  VOID CALLBACK WaitCallback( _Inout_ PTP_CALLBACK_INSTANCE Instance,
  _Inout_opt_ PVOID Context, // parameter passed to callback
  _In_ PTP_WAIT Wait,
  _In_ TP_WAIT_RESULT WaitResult );
  ```

  where `WaitCallback` is a placeholder for the application-defined function name.

- A wait object is freed with

  ```
  VOID WINAPI CloseThreadpoolWait( _Inout_ PTP_WAIT pwa );
  ```

Timer Objects

- The following function creates a timer object:

  ```
  PTP_TIMER WINAPI CreateThreadpoolTimer( _In_ PTP_TIMER_CALLBACK pfnti,
  _Inout_opt_ PVOID pv, // parameter passed to callback
  _In_opt_ PTP_CALLBACK_ENVIRON pcbe );
  ```

- The signature of the callback function is

  ```
  VOID CALLBACK TimerCallback( _Inout_ PTP_CALLBACK_INSTANCE Instance,
  _Inout_opt_ PVOID Context, // parameter passed to callback
  _In_ PTP_TIMER Timer );
  ```

  where `TimerCallback` is a placeholder for the application-defined function name.

- A timer object is freed with

  ```
  VOID WINAPI CloseThreadpoolTimer( _Inout_ PTP_TIMER pti );
  ```
IO Objects

- The following function creates an IO object:
  ```c
  PTP_IO WINAPI CreateThreadpoolIo( _In_ HANDLE h,                   // a file handle
                                     _In_ PTP_WIN32_IO_CALLBACK pfio,    // parameter
                                     _Inout_opt_ PVOID pv, _In_opt_ PTP_CALLBACK_ENVIRON pcbe );
  ```

- The signature of the callback function is
  ```c
  VOID CALLBACK IoCompletionCallback(
    _Inout_ PTP_CALLBACK_INSTANCE Instance,
    _Inout_opt_ PVOID Context,  // parameter passed to callback
    _Inout_opt_ PVOID Overlapped, // pointer to OVERLAPPED struct.
    _In_ ULONG IoResult,        // NO_ERROR or an error code
    _In_ ULONG_PTR NumberOfBytesTransferred,
    _Inout_ PTP_IO Io );
  ```
  where `IoCompletionCallback` is a placeholder for the application-defined function name.

- An IO object is freed with
  ```c
  VOID WINAPI CloseThreadpoolIo( _Inout_ PTP_IO pio );
  ```

Putting Callback Objects in the Work Queue

- Callback Objects are put in the work queue with the functions
  ```c
  VOID WINAPI SubmitThreadpoolWork( _Inout_ PTP_WORK pkw );
  ```

- When an asynchronous I/O was preceded by
  ```c
  VOID WINAPI StartThreadpoolIo( _Inout_ PTP_IO pio );
  ```
  an I/O completion object will be put in the work queue automatically.
Waiting for Callback Completion

- You can wait for the completion (i.e. synchronize with the completion) of outstanding callbacks of a particular callback object with the functions:

  VOID WINAPI WaitForThreadpoolWorkCallbacks(_Inout_ PTP_WORK pwk, _In_ BOOL fCancelPendingCallbacks);

  VOID WINAPI WaitForThreadpoolWaitCallbacks(_Inout_ PTP_WAIT pwa, _In_ BOOL fCancelPendingCallbacks);

  VOID WINAPI WaitForThreadpoolTimerCallbacks(_Inout_ PTP_TIMER pti, _In_ BOOL fCancelPendingCallbacks);

  VOID WINAPI WaitForThreadpoolIoCallbacks(_Inout_ PTP_IO pio, _In_ BOOL fCancelPendingCallbacks);

Further I/O Optimizations

I/O Priorities and Bandwidth Reservation
I/O Prioritization Concepts

- As of Windows Vista / Server 2008, I/Os can be given one of five priorities:
  - Critical: used by the memory manager
  - High: not currently used
  - Normal: used by normal applications
  - Low: not currently used
  - Very Low: used for background activities

- These priorities are used according to two different strategies:
  - Hierarchy prioritization for the upper four priority values (critical, high, normal, low),
  - Idle prioritization for I/Os with the I/O priority very low.

Hierarchy Prioritization

- With the hierarchy prioritization strategy, all higher priority I/Os are processed before any lower-priority I/Os:
  - critical before high,
  - high before normal,
  - normal before low.

- Hierarchy prioritization is implemented by the storage port drivers.

- Not all port drivers (have to) support I/O prioritization:
  - The ATA and the USB port drivers support I/O prioritization.
  - The SCSI port driver does not support I/O prioritization.
  - Therefore, the specified I/O priority only is a priority hint.
Idle Prioritization

- The idle prioritization strategy
  - only processes an I/O request when there is no higher priority I/O,
  - waits 50 milliseconds after the completion of the last non-idle I/O before issuing an idle I/O,
  - guarantees that at least one I/O is processed per unit of time (half a second),
  - causes the cache manager to
    - flush written data immediately,
    - not do any read-ahead.

- The idle prioritization strategy
  - is implemented in the Microsoft-provided system storage class device driver,
  - automatically applies to I/Os directed at all storage devices.

Using I/O Prioritization

The I/O priority can be set at three different levels:

- to very low for a process or a single thread with SetPriorityClass resp. SetThreadPriority, by specifying the priority values PROCESS_MODE_BACKGROUND_BEGIN and PROCESS_MODE_BACKGROUND_END (resp. THREAD_MODE_BACKGROUND_...),

- to very low, low, or normal for a file object (i.e. an open file) with SetFileInformationByHandle, by specifying FileIoPriorityHintInfo as the FileInformationClass,

- for individual I/Os (IRPs) with IoSetIoPriorityHint (by device drivers).

The I/O priority can be watched with tools like
- Process Explorer
- Process Monitor
I/O Bandwidth Reservation

- Bandwidth Reservation
  - asks the I/O system to guarantee the reading of data at a specified rate,
  - is implemented at the port driver level,
  - is only available for IDE, SATA, or USB based storage.

- A maximum of 75% of the bandwidth of a device may be reserved.

- All other I/Os will have to wait until granted bandwidth reservations have been satisfied.

Using I/O Bandwidth Reservation

- BOOL WINAPI SetFileBandwidthReservation(
  __in HANDLE hFile,
  __in DWORD nPeriodMilliseconds,
  __in DWORD nBytesPerPeriod,
  __in BOOL bDiscardable,
  __out LPDWORD lpTransferSize,
  __out LPDWORD lpNumOutstandingRequests);

  - requests bandwidth for nBytesPerPeriod many bytes per period of length nPeriodMilliseconds,
  - specifies whether an I/O should be discarded when it cannot be processed in time,
  - returns the minimum size that should be used for any I/O request in lpTransferSize,
  - returns the maximum number of I/Os that should be outstanding in lpNumOutstandingRequests,
  - returns ERROR_NO_SYSTEM_RESOURCES when the request cannot be satisfied.

The values for nPeriodMilliseconds, and for nBytesPerPeriod that are supported by a particular volume can be obtained with GetFileBandwidthReservation.
SATA NCQ Prioritization

- Disk Scheduling may have an adverse effect on I/O prioritization.
- Traditionally, the reordering of I/Os to reduce seek times does not take I/O priorities into account.
- The SATA specification introduced I/O prioritization to its NCQ (native command queueing) to solve this issue.

Working with Threads
Creating Threads

- HANDLE WINAPI CreateThread(
  LPSECURITY_ATTRIBUTES lpThreadAttributes,  // pointer to thread security attributes
  DWORD dwStackSize,  // initial thread stack size, in bytes
  LPTHREAD_START_ROUTINE lpStartAddress,   // pointer to thread function
  LPVOID lpParameter,   // argument for new thread
  DWORD dwCreationFlags,  // creation flags
  LPDWORD lpThreadId    // pointer to returned thread identifier
);

- Possible Flags: CREATE_SUSPENDED
  STACK_SIZE_PARAM_IS_A_RESERVATION

- DWORD WINAPI ThreadFunction(LPVOID lpThreadParameter);

Setting Thread Priority

- BOOL WINAPI SetThreadPriority(
  HANDLE hThread,
  int nPriority
);

- The specified priority is the so-called priority level, the priority relative to the priority class of the process.

- The possible priority level values are:
  - THREAD_PRIORITY_IDLE (1 resp. 16)
  - THREAD_PRIORITY_LOWEST (-2)
  - THREAD_PRIORITY_BELOW_NORMAL (-1)
  - THREAD_PRIORITY_NORMAL (0)
  - THREAD_PRIORITY_ABOVE_NORMAL (1)
  - THREAD_PRIORITY_HIGHEST (2)
  - THREAD_PRIORITY_TIME_CRITICAL (15 resp. 31)
Further Thread Functions

- DWORD WINAPI SuspendThread(HANDLE hThread);
- DWORD WINAPI ResumeThread(HANDLE hThread);
- VOID WINAPI Sleep(DWORD dwMilliseconds);
- HANDLE WINAPI GetCurrentThread(void);

Synchronization
Overview

- Synchronization Objects
  - Signalled and Unsignalled State
  - Wait-Functions
- Mutexes
- Semaphores
- Events
- Waitable Timer
- Critical Sections (implemented by the Win32 Subsystem)
- Slim Reader Writer Locks (SRW, implemented by the Win32 Subsystem)
- Interlocked Functions

Synchronization Objects (1)

- Certain kernel objects are implemented as so-called synchronization objects, which can be in one of two states:
  - in the signalled state
  - in the unsignalled state

- A thread can synchronize itself with one or more objects by using the wait services of the object manager:
  - The thread blocks until the synchronization object(s) enter(s) the signalled state.
  - When the kernel sets a synchronization object to the signalled state, one or several of the threads waiting for this object are made ready to run.
### Synchronization Objects (2)

<table>
<thead>
<tr>
<th>Object Type</th>
<th>Signalled state when</th>
<th>Wakes up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>the process terminates</td>
<td>all threads</td>
</tr>
<tr>
<td>Thread</td>
<td>the thread terminates</td>
<td>all threads</td>
</tr>
<tr>
<td>File</td>
<td>an I/O for this file completes</td>
<td>all threads</td>
</tr>
<tr>
<td>Mutex</td>
<td>the mutex is not owned by a thread</td>
<td>one thread</td>
</tr>
<tr>
<td>Semaphore</td>
<td>the value of the semaphore is bigger than 0</td>
<td>several threads</td>
</tr>
<tr>
<td>Event</td>
<td>a thread explicitly sets the event</td>
<td>depends on the event type</td>
</tr>
<tr>
<td>Timer</td>
<td>the specified time is reached</td>
<td>depends on the timer type</td>
</tr>
</tbody>
</table>

### Wait Functions (1)

- **DWORD WaitForSingleObject**
  ```
  DWORD WaitForSingleObject(  
      HANDLE hObject,  
      DWORD dwMilliseconds  
  );
  ```
  // handle of object to wait for
  // time-out interval in milliseconds

  - For `dwMilliseconds` you may also specify INFINITE.

- **Possible Return Values:**
  - **WAIT_OBJECT_0** The object is in the signalled state.
  - **WAIT_TIMEOUT** A timeout occurred, the object is in the nonsignalled state.
  - **WAIT_ABANDONED** The object is a mutex which was „abandoned“ (see below).
  - **WAIT_FAILED** An error occurred.
Wait Functions (2)

- DWORD WaitForMultipleObjects(
    DWORD nCount,         // number of handles in the object handle array
    const HANDLE* lpHandles,   // pointer to the object-handle array
    BOOL bWaitAll,         // wait flag
    DWORD dwMilliseconds,   // time-out interval in milliseconds
);

- The parameter bWaitAll has the following meaning:
  - TRUE: The function atomically waits for all objects to be in the signalled state.
  - FALSE: The function waits for at least one of the objects to be in the signalled state.

Wait Functions (3)

- Return Values of WaitForMultipleObjects:
  - WAIT_OBJECT_0 to (WAIT_OBJECT_0 + nCount – 1)
    bWaitAll = TRUE: All objects are in the signalled state.
    bWaitAll = FALSE: (return value - WAIT_OBJECT_0) designates the index of the handle in the array lpHandles, for which the corresponding object is in the signalled state.
  - WAIT_TIMEOUT A timeout occurred.
    bWaitAll = TRUE: not all objects are in the signalled state.
    bWaitAll = FALSE: none of the objects is in the signalled state.
  - WAIT_ABANDONED_0 to (WAIT_ABANDONED_0 + nCount – 1)
    bWaitAll = TRUE: All objects are in the signalled state, and at least one of them is an "abandoned" mutex.
    bWaitAll = FALSE: (return value - WAIT_OBJECT_0) designates the index of the handle in the array lpHandles, for which the corresponding object is in the signalled state, and this object is an "abandoned" mutex.
  - WAIT_FAILED An error occurred.
Mutexes

- A synchronization object of type mutex is
  - in the signalled state, when it is not owned by a thread,
  - in the unsignalled state, when it is owned by a thread.

- When a mutex object goes to the signalled state, exactly one waiting thread will be woken up.

- When a thread ends without having released a mutex it currently holds,
  - the mutex is considered „abandoned“,  
  - the mutex goes to the signalled state,
  - one thread waiting for the mutex is awakened,
  - the awakened thread is informed by the return status of its wait call that the mutex was not released normally, but is „abandoned“.

Win32-API Functions for Mutexes

- HANDLE CreateMutex(
    LPSECURITY_ATTRIBUTES lpMutexAttributes,  
    BOOL bInitialOwner,  
    LPCTSTR lpName    
);  

  Creates a new mutex object or opens an existing one.

- BOOL ReleaseMutex(
    HANDLE hMutex
);  

  Only succeeds when the mutex is held by the caller.
Semaphores

- A synchronization object of type semaphore is
  - in the signalled state, when its value is bigger than 0,
  - in the unsignalled state, when its value is 0.

- Consider the following when using Windows semaphores:
  - Calling a wait function decrements the value of the semaphore by 1.
  - It is not possible to atomically request (wait for) two or more „copies“ of the same semaphore.
  - Every thread can call ReleaseSemaphore (without having requested it before).
  - Several „copies“ of a semaphore can be released with one call to ReleaseSemaphore.
  - When n „copies“ of a semaphore are released, (up to) n waiting threads will be woken up.
  - ReleaseSemaphore fails if the resulting value for the semaphore would exceed the maximum.
  - It is not possible to query the value of a semaphore without changing the value.
  - When a thread terminates, its previous semaphore operations will not be undone.

Win32-API Functions for Semaphores

- HANDLE CreateSemaphore(
  LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
  LONG InitialCount,
  LONG MaximumCount,
  LPCTSTR IpName
);

  Creates a new semaphore object or opens an existing one.

- BOOL ReleaseSemaphore(
  HANDLE hSemaphore,
  LONG lReleaseCount,
  LPLONG lpPreviousCount
);

  Releases ReleaseCount „copies“ of a semaphore.
Events

- Events are very primitive but versatile synchronization objects.
  - Whether an event is in the signalled or the unsignalled state is solely determined by the application.
  - The operating system only takes care of blocking and waking threads.

There are two types of event objects:

- **Manual-reset events**
  - remain in the signalled state, until they are explicitly reset,
  - wake up all waiting threads when they go to the signalled state.

- **Auto-reset events**
  - are automatically reset to the unsignalled state as soon as one thread was awakened.

Win32-API Functions for Events

- **HANDLE CreateEvent(**
  - LPSECURITY_ATTRIBUTES lpEventAttributes,
  - BOOL bManualReset,
  - BOOL bInitialState,
  - LPCTSTR lpName
  - **);**
    Creates a new event object or opens an existing one. The type is specified by the $bManualReset$ parameter.

- **BOOL SetEvent( HANDLE hEvent );**
  Sets an event to the signalled state.

- **BOOL ResetEvent( HANDLE hEvent );**
  Resets an event to the nonsignalled state.
Waitable Timer

- A waitable timer object goes to the signalled state at a specified point in time.

There are two types of waitable timer objects:

- **Manual-reset timers**
  - remain in the signalled state, until they are explicitly reset (or set to a new time),
  - wake up all waiting threads when they go to the signalled state.

- **Synchronization timers**
  - are automatically reset to the unsignalled state as soon as one thread was awakened.

Both types of timer objects can also be defined as periodical timers.

Win32-API Functions for Waitable Timer

- **HANDLE CreateWaitableTimer(**
  - LPSECURITY_ATTRIBUTES lpTimerAttributes,
  - BOOL bManualReset,
  - LPCTSTR lpTimerName **);**

- **BOOL SetWaitableTimer(**
  - HANDLE hTimer,
  - const LARGE_INTEGER* pDueTime, // in FILETIME format
  - LONG lPeriod, // in milliseconds
  - PTIMERAPCROUTINE pfnCompletionRoutine,
  - LPVOID lpArgToCompletionRoutine,
  - BOOL fResume **);**

- **BOOL CancelWaitableTimer( HANDLE hTimer );**
Time Formats (1)

- `typedef struct _FILETIME {
    DWORD dwLowDateTime;
    DWORD dwHighDateTime;
} FILETIME`;

- `FILETIME` is a 64-bit value representing the number of 100-nanosecond intervals since 00:00 on January 1, 1601 (UTC).

- Calculations with time values should be done by:
  - copying the `FILETIME` structure to a `ULARGE_INTEGER` structure (using `memcpy`),
  - then using normal 64-bit arithmetic.

Time Formats (2)

- `typedef struct _SYSTEMTIME {
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME`;

- `wMonth`: January = 1, February = 2, etc.
- `wDayOfWeek`: Sunday = 0, Monday = 1, etc.
Converting Times

- BOOL SystemTimeToFileTime(
  const SYSTEMTIME* lpSystemTime,
  LPFILETIME lpFileTime);  

- BOOL FileTimeToSystemTime(
  const FILETIME* lpFileTime,
  LPSYSTEMTIME lpSystemTime);  

- BOOL FileTimeToLocalFileTime(
  const FILETIME* lpFileTime,
  LPFILETIME lpLocalFileTime);  

- BOOL LocalFileTimeToFileTime(
  const FILETIME* lpLocalFileTime,
  LPFILETIME lpFileTime);  

Interlocked Functions

- The interlocked functions perform operations on variables in an atomic manner.

  - Arithmetic and logic functions:
    - InterlockedDecrement, InterlockedIncrement, InterlockedExchangeAdd
    - InterlockedExchange, InterlockedCompareExchange
    - InterlockedAdd
    - InterlockedBitTestAndReset, InterlockedBitTestAndSet
    - InterlockedAnd, InterlockedOr, InterlockedXor

  - List functions:
    - InterlockedPopEntrySList, InterlockedPushEntrySList

- All functions also exist in versions
  - for 64-bit values on 64-bit systems,
  - using different memory semantics ("acquire" and "release"), if supported by the processor.
Critical Sections

- Critical Sections are a synchronization mechanism implemented in the Win32 subsystem.
  - When the critical section is available, no call to the kernel is necessary.
  - Keeping wait queues and waking up threads is done by the kernel.

- Optionally, a spincount can be specified for a Critical Section.
  - If the critical section is unavailable, the thread checks for its availability in a busy loop.
  - When the critical section is still unavailable after spincount many loops, the thread blocks.
  - If the critical section becomes free during the spin operation, the calling thread avoids the wait operation.

- Critical Sections
  - can only be used by the threads of a single process,
  - can be requested in a blocking or non-blocking fashion (but no finite timeout),
  - are not released automatically when a thread ends.

Functions for Critical Sections

- VOID InitializeCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection);
- BOOL InitializeCriticalSectionAndSpinCount(
  LPCRITICAL_SECTION lpCriticalSection,
  DWORD dwSpinCount);
- VOID EnterCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection);
- BOOL TryEnterCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection);
- VOID LeaveCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection);
- VOID DeleteCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection);
**Slim Reader Writer Locks**

- Slim Reader Writer Locks (SRW locks) - which were introduced with Windows Vista - are another synchronization mechanism implemented in the Win32 subsystem.

- A SRW lock can be requested in two different modes:
  - Shared mode
  - Exclusive mode

- As the names of the modes imply
  - any number of threads can be granted access to the lock in shared mode concurrently,
  - only a single thread can be granted access to the lock in exclusive mode.

- A SRW lock held in shared mode cannot be upgraded to exclusive mode.

---

**Functions for SRW locks**

- VOID WINAPI InitializeSRWLock(
  _Out_ PSRWLOCK SRWLock );
- VOID WINAPI AcquireSRWLockExclusive(
  _Inout_ PSRWLOCK SRWLock );
- VOID WINAPI AcquireSRWLockShared(
  _Inout_ PSRWLOCK SRWLock );
- BOOLEAN TryAcquireSRWLockExclusive(
  _Inout_ PSRWLOCK SRWLock );
- BOOLEAN TryAcquireSRWLockShared(
  _Inout_ PSRWLOCK SRWLock );
- VOID WINAPI ReleaseSRWLockExclusive(
  _Inout_ PSRWLOCK SRWLock );
- VOID WINAPI ReleaseSRWLockShared(
  _Inout_ PSRWLOCK SRWLock );