Windows and .Net Threads

Jim Fawcett

Software Modeling

Windows API

- Create, style, and manage windows
- Manage files and directories
- Create and manage processes, threads, and synchronizers
- Load and unload dynamic-link libraries
- Create and manage timers
- Read and write to the registry

Windows Resources—MSDN

- Windows System Services
- Processes and Threads
- Synchronization
  - Using Synchronization
  - Synchronization Reference
- System.Threading Namespaces
  - System.Threading Namespace
  - Thread Class
  - System.Threading.Tasks Namespace
    - Task Class
    - Task<TResult> Class

Other Threading Resources

- 7 Ways to Start a Task in .NET C#
- Best Practices in Asynchronous Programming
- Asynchronous Programming with Async and Await – MSDN
- Event-Based Asynchronous Pattern
- Async in 4.5: Worth the Await

Windows Processes and Virtual Memory

- Windows processes are containers for:
  - Threads—one primary thread at process start
  - File handles
  - Other windows objects
  - Allocated memory
  - Process Diagram
- Windows uses a paged virtual memory system
  - Virtual Memory Diagram

Windows Objects

- Windows supports internal objects and provides an API to access and manage them through handles.
- Kernel objects—named, secured, system-wide
  - Devices, files, symbolic links, registry keys, threads and processes, events, mutexes, semiphores, memory-mapped files, callbacks, pipes
- User objects—GUI objects
  - Window, menu, icon, hook, …
- GDI objects—drawing objects
  - Brush, pen, DeviceContext, bitmap, metafile, …

Windows Objects and Handles

- A windows object is a data structure that represents a system resource, e.g., file, thread, bitmap
  - Windows objects are created and accessed using the Win32 API
  - They reside in Kernel32.dll, User32.dll, or GDI32.dll
- An application accesses those resources through a handle, usually returned from a Create function
- 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

Object API

- The Windows API provides functions which:
  - Create an object, returning a handle
  - Get an object handle using other information
  - Get and set information about an object
  - Close the object handle, possibly destroying the internal object
    - Objects are reference counted and will be destroyed when all referring handles have been closed
  - Kernel objects have security functions that manage ACLs

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.
  - A kernel object is always either signaled or unsignaled.
  - An object in the unsignaled state will cause any thread that waits on the object's handle to block.
  - An object in the signaled state will not block a thread that called wait on its handle.

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.
  - Thus local data is unique to each thread.
- Pointers are process specific, so threads can share pointers.

Starting a Process

- Every time a process starts Windows creates a primary thread.
  - The thread begins execution with the application's startup code that initializes libraries and enters main
  - The process continues until main exits and library code calls ExitProcess.
- You will find demo code for starting a Windows process here:
  - http://www.ecs.syr.edu/faculty/fawcett/handouts/Coretechnologies/ThreadsAndSynchronization/code/ProcessDemoWin32/
- Here is a demo for starting up a process using C#:
  - http://www.ecs.syr.edu/faculty/fawcett/handouts/Coretechnologies/ThreadsAndSynchronization/code/ProcessDemoDotNet/

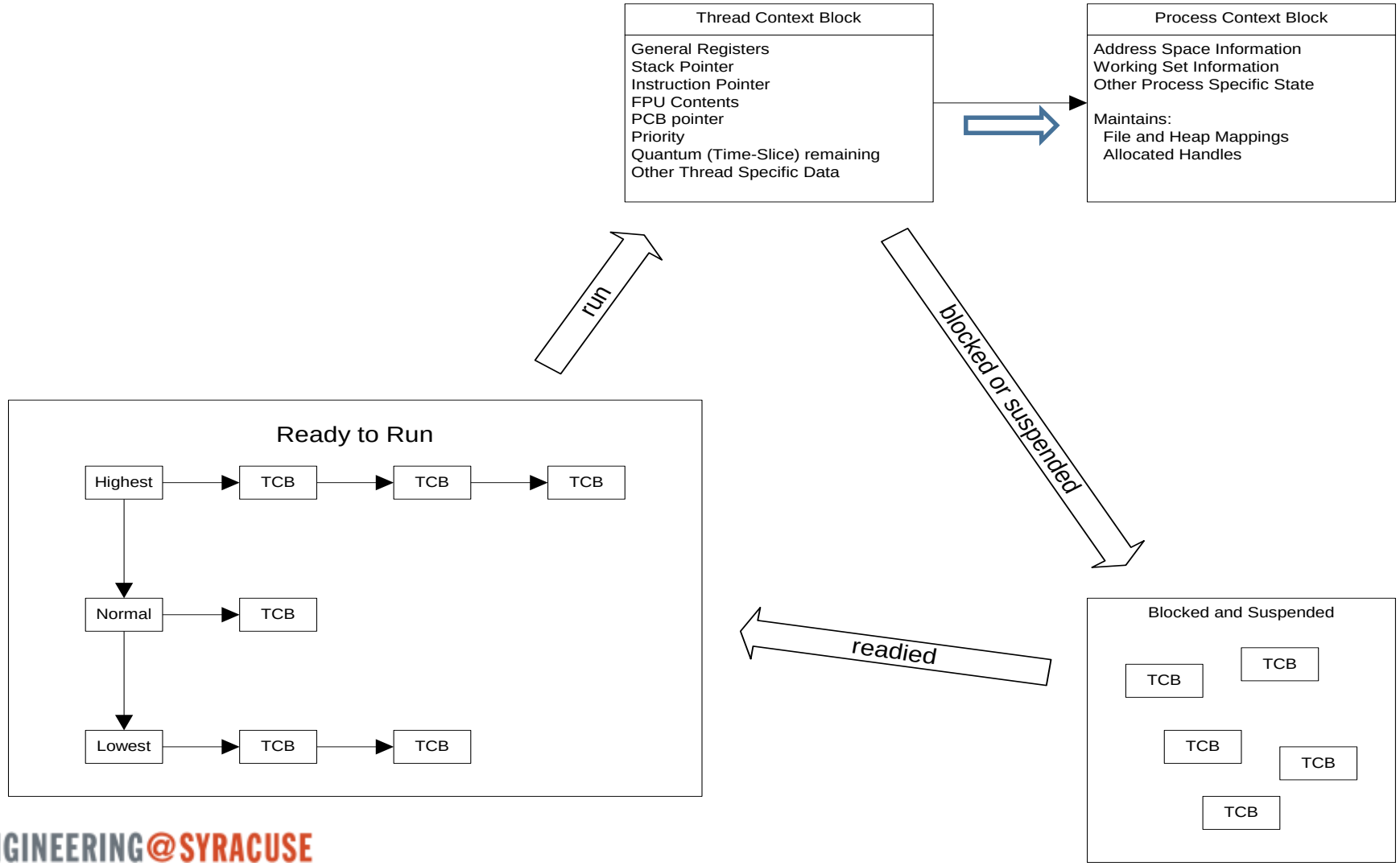
Scheduling Threads

- Windows is a preemptive multitasking system. Each task is scheduled to run for some brief time period before another task is given control of a CPU core.
- Unlike Unix and Linux, in Windows threads are the basic unit of scheduling. A thread can be in one of three possible states:
  - Running
  - Blocked or suspended, using virtually no CPU cycles, but consuming about 1 MB of memory per thread
  - Ready to run, using virtually no CPU cycles

Scheduling Activities

- A running task is stopped by the scheduler if:
  - It is blocked waiting for some system event or resource
  - Its 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, e.g., waiting on a timer
  - It is suspended by some other thread
  - It is suspended by Windows while the OS takes care of some critical activity
- Blocked threads become ready to run when an event or resource they wait on becomes available, e.g., its handle becomes signaled
- Suspended threads become ready to run when their suspend count is zero

Scheduling Threads



Benefits of Using Threads

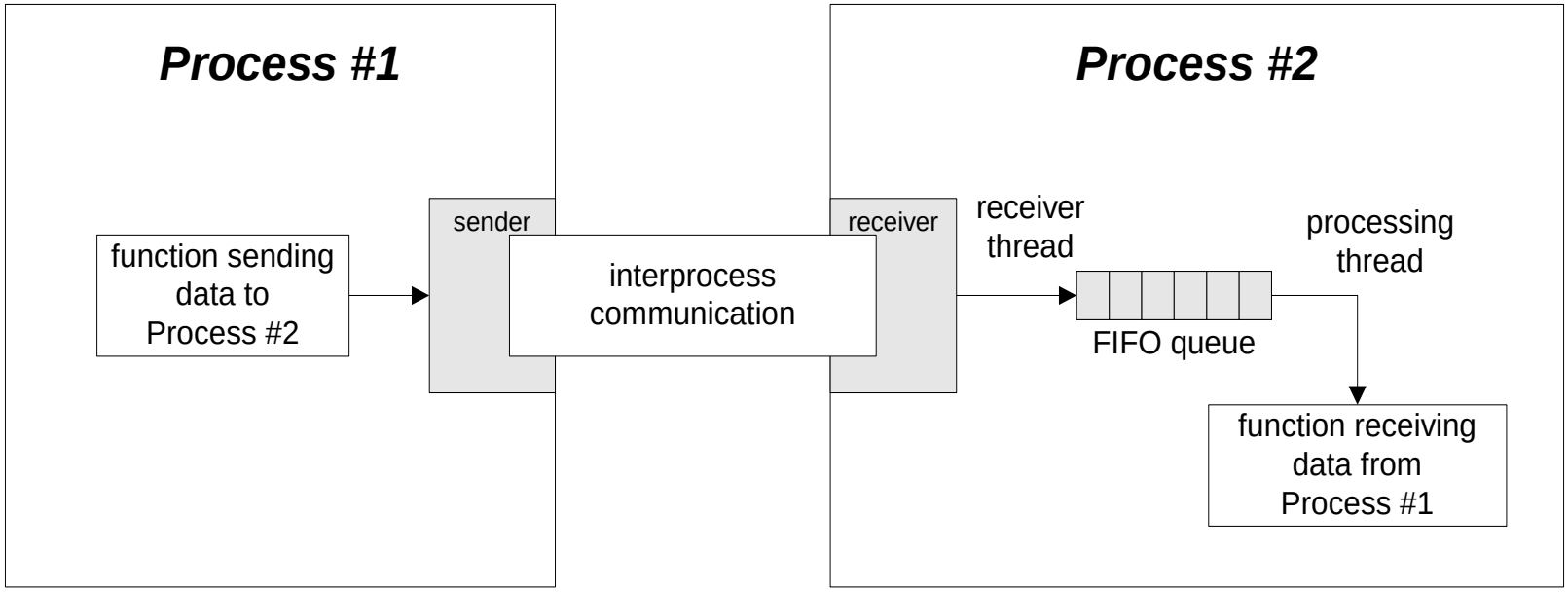
- Keep user interfaces responsive even if required processing takes a long time to complete.
  - Handle background tasks with one or more threads
  - Service the user interface with a dedicated UI thread
- Your program may need to respond to high-priority events, so you can assign that event handler to a high priority thread.
- Take advantage of multiple cores available for a computation.
- Avoid low CPU activity when a thread is blocked waiting for response from a slow device or human, allowing other threads to continue.

More Benefits

- Support access to server resources by multiple concurrent clients.
- For processing with several interacting objects the program may be significantly easier to design by assigning one thread to each object.

Using Threads to Avoid Blocking

Non-Blocking Communication in Asynchronous System



Potential Problems with Threads

- Conflicting access to shared memory
  - One thread begins an operation on shared memory, is suspended, and leaves the memory region incompletely transformed.
  - A second thread is activated and accesses the shared memory in the incomplete 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 due to use of threads.

More Problems with Threads

- Starvation
  - A high-priority thread dominates CPU resources, preventing lower priority threads from running often enough or at all.
- 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 resources.

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 directly access a window's member functions. This will cause exceptions.
  - Worker threads communicate with a program's windows by calling the Win32 API PostMessage and SendMessage functions.
  - With modern GUI frameworks that is handled by calling Form.Invoke or Dispatcher.Invoke passing a delegate to the UI thread bound to a function that handles the worker's data.

Creating Win32 Threads

- Call CreateThread(…) only if you won't be using ANY language libraries, as these are not initialized by Win32 API functions.
- 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 cannot be a nonstatic member function.

Creating Win32 Threads

- HANDLE hThrd = (HANDLE)_beginthreadex(// returns 0 on failure

  pSecurity, stack_size, ThreadFunc, pThreadInfo, initflg, pThrd

  );
- SECURITY_ATTRIBUTES *pSecurity – null for user priviledges
- unsigned int stack_size – size of stack to use, 0 gives default size
- ThreadFunc – the function executed by the new thread

  unsigned _stdcall ThreadFunc(void *pThreadInfo); // returns exit code
- void *pThreadInfo – pointer to input parameter structure for use by ThreadFunc
- Enum initflg – 0 to start running or CREATE_SUSPENDED to start suspended
- Int32 *pThrdID – returns pointer to threadID if non-null on call, otherwise not used
- For threads created with _beginthreadex the thread function, ThreadFunc, must be a global function or static member function of a class. It cannot be a nonstatic member function.

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

Creating .Net Threads Using C#

- Thread t = new Thread(new ThreadStart(tProc));
  t.Start()
  - void tProc() is a static or nonstatic member function of some class.

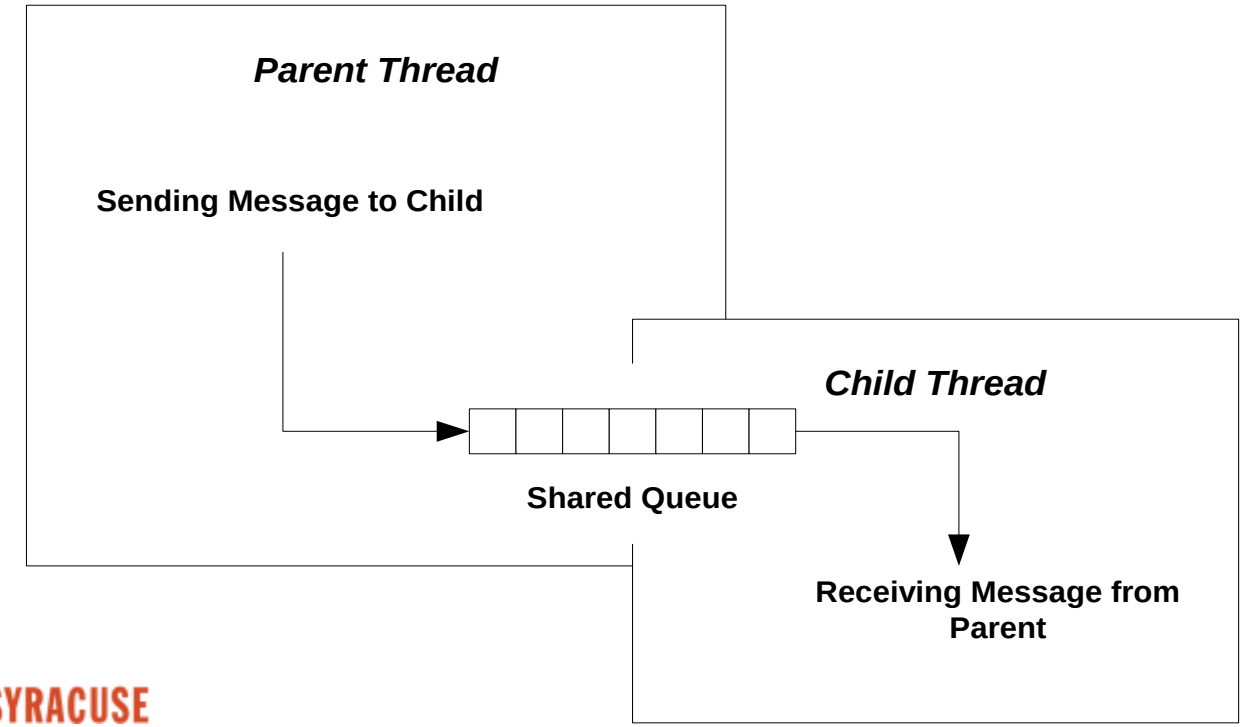
- Thread t = new Thread(new ParameterizedThreadStart(tProc))
  t.Start(inputArgument)
  - where void tProc(object inputArgument) is a static or nonstatic function of some class.

Thread Properties

- ***IsBackground*—**get, set
  - Process does not end until all foreground threads have ended.
  - Background threads are terminated when application ends.
- **CurrentThread—**get, static
  - Returns thread reference to calling thread
- ***IsAlive*—**get
  - Has thread started but not terminated?
- ***Priority*—**get, set
  - Highest, AboveNormal, Normal, BelowNormal, Lowest
- ***ThreadState*—**get
  - Unstarted, Running, Suspended, Stopped, WaitSleepJoin, ..

Shared Resources

- A child thread often needs to communicate with its parent thread. It does this via some shared resource, like a queue.



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, which 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

Win32 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 increment
    - Only for incrementing or decrementing integral types
  - Critical section
    - Good only inside one process
  - Mutex (kernel object)
    - Named mutexes can be shared by threads in different processes.
  - Event (kernel object)
    - Useful for synchronization as well as other event notifications
  - Semiphore (kernel object)
    - Allows a specified number of threads to use a resource
  - SlimReaderWriter lock
    - Supports concurrent reads while making writes and reads sequential
  - Condition variable (kernel object)
    - Used in conjunction with a critical section or mutex to signal an event that allows one or more threads to proceed.

.Net Synchronization

.Net wraps most of the Win32 synchronizers:

- Monitor
  - Wraps a critical section and a condition variable to support both locking and notification
- Lock
  - Encapsulates a monitor and try {} finally {} to ensure that the lock is released even if an exception is thrown
- Mutex
  - Wraps the Win32 mutex
- ReaderWriterLock
  - Wraps the Win32 SlimReaderWriter
- Interlocked
  - Wraps the Win32 interlocked
- SpinLock
  - Supports low overhead short-term locks
- Semaphore
  - Wraps the Win32 semaphore
- WaitHandle
  - Threads block by calling WaitOne on the handle. They are released by the kernel object to which the handle refers.

Tasks and Async Await Pattern

.Net 4 introduced some new asynchronous programming constructs centered on the Task class.

- Task.Run(…) runs a task defined by an action or lambda that has no return value on a thread-pool thread.
- Task<T>.Run(…) runs a task defined by a Funct<T> or lambda that returns an instance of T.

This topic is quite accessible if you look at some simple code demos:

- Tasks, Threads, and Continuations
- Abstract Task Model
- WPF Thread Demos

Appendix:
More Details on Some of the Synchronization Operations

Win32 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);
```

.Net Interlocked—Atomic Operations

- static int count = 0;
- Interlocked.Increment(ref count);
- Interlocked.Decrement(ref count);
- Interlocked.Exchange(ref count, 1);
- Long cnt = Interlocked.Read(ref count);

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);

.Net Lock—Most Commonly Used

- static object synch_;
- Lock(synch_)
  {
  // use shared resource safely
  // lock is released if exception is thrown in
  // locked region
  }
- Note: synch_ must be a reference type, as shown here.

.Net Monitor

Used for locking and signaling

- static object sync_;
- Monitor.Enter(sync_); // enters protected region
- Monitor.Exit(sync_); // leaves protected region
- Monitor.Pulse(resource); // notifies waiting thread
- Monitor.PulseAll(resource);

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);

.Net Mutex

Used for system-wide synchronization

- Mutex mutex = new Mutex(false);
- mutex.WaitOne(); // enter protected region
- mutex.ReleaseMutex(); // leave protected region
- mutex.Close(); // decrement object's ref count

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 nonsignaled 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 nonsignaled state only when reset by your program.

Win32 Events

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

.Net Manual ResetEvent

- static ManualResetEvent mre = new ManualResetEvent(false);
- mre.Set(); // threads will not block on WaitOne()
- mre.Reset(); // threads that call WaitOne() will block
- mre.WaitOne();
- mre.close()

ENGINEERING@SYRACUSE