

Managed Threads

- OS Threads and Managed Threads
- Address Space
- The Thread Class
- ThreadStart Delegate
- Exiting a Thread
- Priority
- Affinity
- Number of Threads
- Suspend/Resume

OS Threads and Managed Threads

- A thread is a unit of execution within a process
 - An OS thread has its own set of CPU register values, together with user mode and kernel mode stacks
 - The CLR host determines the relationship between a managed (.NET) thread, and the OS thread
 - It could be 1:1 for simple CLR hosts, or multiple managed threads could be mapped to a single OS thread in more advanced CLR hosts
 - The MSDN System.Threading.Thread reference page states that for current Windows OS versions, it is 1:1
 - The CLR Host may move a managed thread between OS threads
 - One or more threads run as part of a process
-

Address Space

- Threads use the address space of the process within which they run
 - Local variables (defined within a method) and thread data slots (equivalent concept to Win32's TLS) may be used in each thread without a problem
 - Class variables (either static or instance) may be accessed and changed by multiple threads simultaneously, which can be a problem - hence may need synchronisation
 - Which can occur if either multiple threads get access to the same reference to memory, or multiple references exist which point to the same block of memory
-

Where Code Starts Executing

- When you create a process, the first thread (the primary thread) is automatically created for you
 - It starts executing the Main method of the start object
 - You may create secondary threads using System.Threading namespace
 - Thread class
 - ThreadStart delegate
 - The new secondary thread starts executing the ThreadStart delegate
 - which internally calls instance/static methods in classes it has access to
-

Avoiding Threading

- If you do not want to do multi-threaded programming
 - simply code up your Main function as you would have in the past
 - It will execute in the primary thread as if it were an independent process
 - Works just like as if the code were running in a non-threaded environment
 - The threading infrastructure which .NET provides to those who are interested can be safely ignored by others
-

The Thread Class

- The managed thread-related classes are in the `System.Threading` namespace
 - `System.Diagnostics` also has a `ProcessThread` class, that is used for diagnostics of running process (which may or may not be .NET based)
 - `System.Diagnostics.Process` has a property call `Threads`, of type `ProcessThreadCollection`, which is a collection of `ProcessThread` entries
 - `Thread` is the main managed thread class
 - Important methods: `Abort`, `AllocateDataSlot`, `Set/GetData`, `SpinWait`, `Start`, `Suspend/Resume`
 - Important properties: `IsBackground`, `Priority`, `CurrentPrincipal`, `IsAlive`, `ThreadState`, `CurrentThread`
-

Notes

- Thread is used to start new threads, and to control and query existing threads
 - Thread can be used for synchronisation on thread termination
 - A new thread is started by constructing an instance of the Thread class passing in a ThreadStart delegate, and then calling Thread.Start
 - There is no hierarchical relationship among threads
 - Threads which create other threads are treated the same as the created threads (they are all siblings)
 - though the primary thread has some important attributes
-

The ThreadStart Delegate

- This is where the new thread starts executing
 - When the ThreadStart exits, this is equivalent to the thread exiting
 - Conceptually similar to exiting the main for standard C programs
 - Each thread only has a single ThreadStart method and it must be set in the Thread constructor, and may not be changed later
 - A ThreadStart is a method which you write with this prototype:
 - `void MyThreadProc();`
 - The ThreadStart delegate may be a method of the same class as that which calls Thread.Start, or of a different class
-

Sample

- `public void Run(){`
 - `// do work here!`
 - `}`
 - `Void SomeOtherMethod {`
 - `Thread myThread =`
 - `new Thread(new ThreadStart(Run));`
 - `myThread.Start();`
 - `}`
 - The parameter to the thread constructor must be a valid ThreadStart delegate
 - e.g. if null, an `ArgumentNullException` is thrown
 - A thread may be started only once
-

Passing Information to ThreadStart

- The ThreadStart delegate contains no parameters, unlike Win32's threadproc
 - e.g. Imagine we would like to have the same ThreadStart delegate used for different threads and pass it in different parameters so it works on different data
 - So how do you pass data into a ThreadStart?
 - The ThreadStart delegate may be a member of any class
 - We can create a class that contains the ThreadStart, and whatever data we wish it to operate upon, and initialise this data before we call Thread.Start()
 - If we need to exchange data between two or more threads while they are executing, then we need to consider synchronisation
-

- private void StartThreadWithDataBtn_Click(
 - object sender, System.EventArgs e){
 - MyThreadManager myThreadManager =
 - new MyThreadManager();
 - myThreadManager.StartupValue = 25;
 - Thread myThread = new Thread(
 - new ThreadStart(myThreadManager.Run));
 - myThread.Start();
 - }
 - }
 - class MyThreadManager{
 - int startupValue=0;
 - public int StartupValue{
 - set {startupValue = value;}
 - }
 - public void Run(){
 - // do work here!
 - // startupValue available here
 - }
 - }

Sample

Exiting a Thread

- A thread is destroyed when:
 - It's ThreadStart simply returns
 - Thread.Abort is called (by the thread that is to be aborted, or a different thread with the necessary privileges)
 - When a thread exits it becomes signalled and hence other threads which are waiting on it with Thread.Join are activated
 - The ThreadStart has no return code
 - There is no concept of an exit code for a managed thread
 - Unlike an OS thread - whose exit code is available via the Win32 function GetExitCodeThread
-

OS Thread Priorities

- Threads are assigned priorities
 - In Windows, OS threads have 32 levels of priorities
 - The OS runs the threads with the highest priority, and only when no threads of a certain priority wish to run will it run a thread on a lower priority
 - The thread which is currently executing is interrupted if a thread with a higher priority wishes to run
 - Threads which are blocked (e.g. waiting on a synchronisation object, waiting for I/O etc.) are not scheduled
-

Process and OS Thread Priority

- A process is assigned a process class priority by setting `System.Diagnostics.Process.PriorityClass`, which may be set to a value from the `ProcessPriorityClass` enumeration
 - `ProcessPriorityClass` has the following members:
 - `AboveNormal`, `Normal`, `BelowNormal`, `High`, `Idle`, `RealTime`
 - Be wary about using `High` or `RealTime`, as this could seriously affect the ability of other processes to run
 - A thread can set its relative priority
 - By setting `ProcessThread.ThreadPriority`, which may be set to a value from the `ThreadPriorityLevel` enumeration
 - `AboveNormal`, `Normal`, `BelowNormal`, `Highest`, `Idle`, `Lowest`, `TimeCritical`
 - The `ProcessThread.PriorityBoostEnabled` bool property determines if a thread's priority is boosted when the main window of the thread's process gets the focus
-

Managed Thread Priority

- `System.Threading.Thread` has a `Priority` property
 - May be set to a value from the `System.Threading.ThreadPriority` enumeration: `Highest`, `AboveNormal`, `Normal` (default), `BelowNormal`, `Lowest`
 - This is a hint you give to the CLR host about the scheduling priority you would like to have assigned to a particular thread
 - Hint may or may not be adhered to by the CLR host
-

Ideal Processor & Processor Affinity

- `System.Diagnostics.ProcessThread` has two properties related to the processor a thread uses
- `IdealProcessor (Int)` states the processor the process would prefer to run on
- `ProcessorAffinity (IntPtr)` is a bitmask stating which processors the process would like to use
- Both these properties may be only set (no get)
- Goal is to reduce the frequency with which the CPU cache has to be reloaded
- This is the argument for using web gardens in ASP.NET
- There are no `ProcessorAffinity` or `IdealProcessor` properties for a managed thread (`System.Threading.Thread`) - because there is no guarantee that it corresponds to a single OS thread
- Most developers should let the OS do the scheduling, as ~~getting this wrong can cause serious problems~~
- In some advanced areas, it can lead to significant benefits

Background Threads

- A managed thread can conceptually either run in the foreground or run in the background
 - You can get and set the `Thread.IsBackground` bool property to determine the value for a particular thread
 - The sole difference is that background threads do not keep the process alive, whereas foreground threads do
 - When all the foreground threads have exited, then `Thread.Abort` is called for all the background threads and the process exits
 - Note that the underlying OS thread does not have such a concept
 - It is purely a .NET artifact
-

One Active Thread Per CPU

- Often, it is a good idea for server platforms to have at least one active thread per CPU. An active thread is defined as a thread that is capable of using a CPU if given a timeslice (e.g. it is not blocked waiting on some resource to become available). In this way, if it is the only CPU-intensive app running, it can efficiently use all the available resources
- If you don't know how many threads to use (e.g. you don't know how many could be blocked at any moment), a rule of thumb is to start with the number of threads equal to twice the number of CPUs, and after benchmarking adjust as appropriate
- Too few threads
 - If an app only has a single thread, then its code can only run on a single CPU at a time
 - The OS and other apps (even those which themselves are also single-threaded), can run on the other processors
- Too many threads
 - Need to carefully consider reasons if using large numbers of threads; Having a few additional threads is not going to make any serious difference, but having thousands definitely will
 - E.g. imagine a poorly written web-server, that had one thread per client request (this scenario should be handled by thread pools)

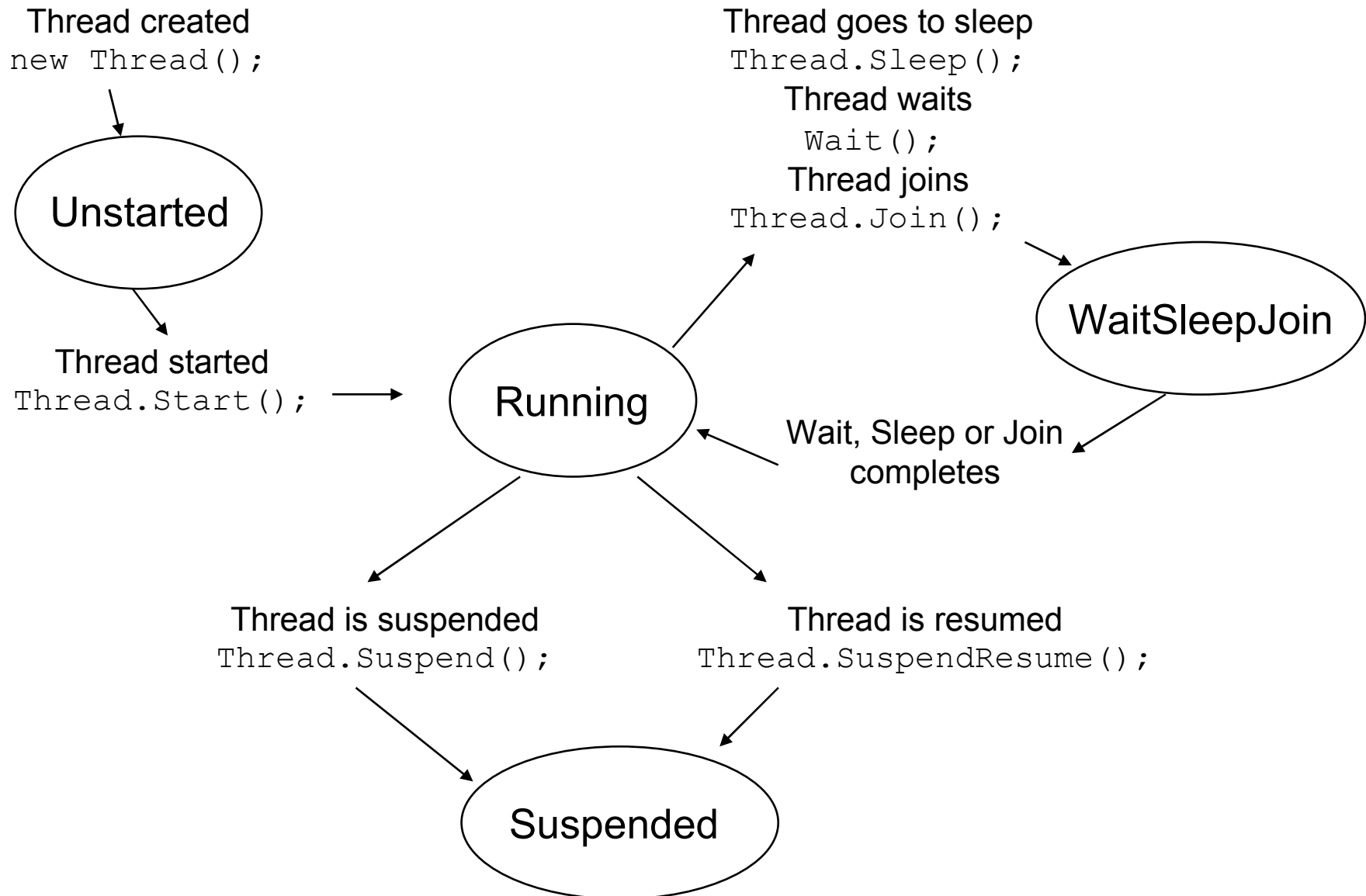
Determining Number of CPUs

- Can determine number of CPUs by calling Win32's GetSystemInfo and checking dwNumberOfProcessors
- .NET does not provide any direct way
- This allows you to ship a single application, that at run-time determines the number of CPUs available, and dynamically adjusts the number of threads it uses
-

-
- [StructLayout(LayoutKind.Sequential, Pack=1)]
 - public struct SYSTEM_INFO{
 - public ushort wProcessorArchitecture;
 - public ushort wReserved;
 - public uint dwPageSize;
 - public IntPtr lpMinimumApplicationAddress;
 - public IntPtr lpMaximumApplicationAddress;
 - public IntPtr dwActiveProcessorMask;
 - public uint dwNumberOfProcessors;
 - public uint dwProcessorType;
 - public uint dwAllocationGranularity;
 - public ushort wProcessorLevel;
 - public ushort wProcessorRevision;
 - };
 - [DllImport("KERNEL32", CharSet=CharSet.Auto)]
 - public static extern void GetSystemInfo(out SYSTEM_INFO si);
 - private void GetNumberOfCPUsBtn_Click(
• object sender, System.EventArgs e){
 - SYSTEM_INFO si;
 - GetSystemInfo(out si);
 - uint numberOfProcessors = si.dwNumberOfProcessors;
 - MessageBox.Show("Number Of Processors =
 - "+numberOfProcessors.ToString());
 - }
-

Sample

States



Thread State

- `Thread.ThreadState` contains `ThreadState` bitmask
 - flag enumeration: multiple may be set at the same time
 - Not all combinations are valid
 - `Aborted`: the thread has been aborted
 - `AbortRequested`: Abort has been requested (via `Thread.Abort`)
 - `Background`: thread is running in the background
 - `Running`: thread is running
 - `Stopped`: thread is stopped
 - `StopRequested`: stop has been requested
 - `Suspended`: thread is suspended
 - `SuspendRequest`: thread suspension has been requested
 - `Unstarted`: thread has yet to be started (`Thread.Start` not yet called)
-
- `WaitSleepJoin`: Awaiting completion of `wait`, `sleep` or `join`

Join

- `Thread.Join` is a blocking call used to wait until a thread terminates
 - `void Join(); // infinite wait`
 - `bool Join(Int32); // wait for x milliseconds`
 - `bool Join(TimeSpan); // wait for timespan`
 - Sample
 - `Thread myThread = new Thread(new ThreadStart(Run));`
 - `myThread.Start();`
 - `myThread.Join();`
 -
-

Suspend/Resume A Thread

- A thread that is running is suspended by calling `Thread.Suspend`
- A thread can suspend itself or a different thread, provided it has sufficient privilege and a reference to the `Thread` instance for the other thread
- A thread that is the target of a suspension request from a different thread is not immediately put in a suspended state, rather it continues until it reaches a safe point and is then suspended
- A safe point is a point at which garbage collection is permitted
- A thread can resume a suspended thread by calling `Thread.Resume`
- A thread that wishes to sleep for a period can call `Thread.Sleep`

Thread Identity

- Options to identify threads
 - For the OS thread, use `System.AppDomain.GetCurrentThreadId`
 - For the managed thread:
 - use `System.Threading.Thread.GetHashCode` to retrieve a unique value that will not collide with the hash codes of other threads in the process (any AppDomain),
 - or simply get a reference to the Thread instance (`System.Threading.Thread.CurrentThread`)
-