

**Microsoft .NET Threading in C#:
Building Responsive, Reliable and
Scalable Applications**



Jeffrey Richter
Copyright © 2008
www.wintellect.com



what we do
Consulting | Debugging | Training

who we are
Founded by top technical and business experts, we are a fast-growing group of outstanding consulting and training professionals who pull out all the stops to solve their clients' problems.

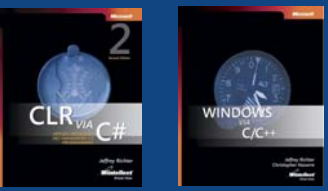
how we do it

Consulting & Debugging <ul style="list-style-type: none">- Architecture, analysis, and design services- Full Lifecycle custom software development- Content creation- Project management- Debugging & performance tuning	Training <ul style="list-style-type: none">- On-site instructor-led training- Virtual instructor-led training- Devscovery conferences
---	--



Jeffrey Richter

- Cofounder of Wintellect
- Author of several .NET Framework & Win32 Books
- MSDN Magazine Contributing Editor/Columnist
- Consultant for many companies
 - Microsoft, Intel, HP, DreamWorks & many others



Course Assumptions & Philosophy

- You should be comfortable with
 - The .NET Framework: The CLR and the C# programming language
 - The Visual Studio development environment
- Philosophy
 - This course teaches how Windows and the CLR work together to provide a threading architecture
 - You will take away a foundation of knowledge allowing you to effectively use threads to build responsive, reliable, and scalable applications and components

Motivating this Course

- Early OSes didn't support threads (there was just 1 thread)
 - Problem: Long-running tasks affected all apps and the OS
- Windows supports multiple threads for robustness (1 or more thread per process)
- Threads are overhead
 - Kernel object, user-mode & kernel-mode stack; DLL attach/detach notifications
- A computer with 1 CPU can run 1 thread at a time
- Windows pre-empts running thread and schedules another thread each *quantum* (~20ms)
 - This is called a *context switch*

Motivating this Course

- Every context switch requires that Windows
 - Enter kernel mode
 - Save registers from CPU to running thread's kernel object
 - x86 = ~700 bytes, x64 = ~1240 bytes, IA64 = ~2500 bytes
 - Determine which thread to schedule next
 - If thread owned by other process, switch address space
 - Load registers from selected thread's kernel object into CPU
 - Leave kernel mode
- Context switches are pure overhead and hurt performance
 - But required for a robust OS
- Conclusion
 - Avoid threads: they are time/memory overhead
 - Use threads: they enable robustness & scalability on multi-CPU systems
 - This class is about wrestling with this tension

CPU Trends



CPU Trends

- Past: CPUs used to get faster each year
 - Our applications just ran faster on a newer CPU
- Present: CPU manufacturing can't continue this trend
 - New trend is more (slow) CPUs on 1 chip
 - We use threads to take advantage of these CPUs
 - Multiple threads run simultaneously
- There are 3 "multi-CPU" technologies
 - Multiple CPU chips: Machine has multiple physical chips
 - Hyper-threaded chips
 - Multi-core chips

Hyper-Threaded Chips
Examples: Intel Xeon & Pentium 4

- Single chip with 2 logical CPUs
- Each logical CPU has a architectural state (CPU registers)
- All logical CPUs share execution resources (CPU cache)
- When logical CPU pauses, chip switches to other CPU
 - Pause: cache miss, branch misprediction, or waiting for the results of a previous instruction
- Performance
 - Ideally, you'd hope to get a 100% improvement (2 CPUs vs 1)
 - Sharing resources hurts performance
 - Intel reports just a 10% to 30% improvement



Multi-Core Chips

Examples: Intel Pentium D & AMD Athlon 64 X2

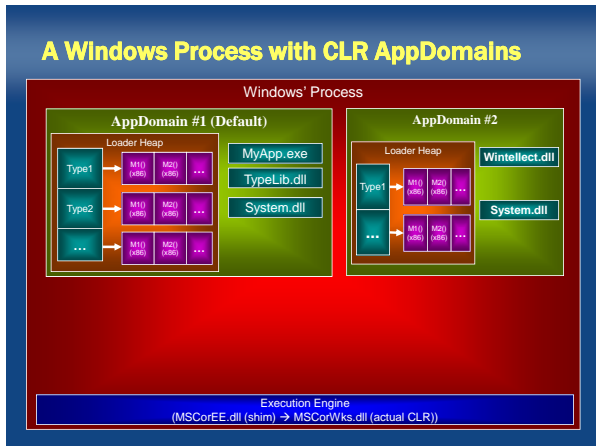
- Single chip with 2/4 physical CPUs
 - Next few years: 8, 16 or 32 core chips; a ton of computing power
- Each CPU has its own architectural state (CPU registers) and execution resources (CPU cache)
- All CPUs truly run simultaneously
- Performance
 - 100% improvement if CPUs run independent tasks
- Intel's Pentium Extreme
 - Uses both hyper-threading and multi-core technologies
 - To Windows, 1 Pentium Extreme chip looks like four CPUs!

Processes, AppDomains, and Threads



What is a Process & an AppDomain?

- Process (a Windows concept)
 - 32-/64- bit virtual address space with EXE/DLL code & data
 - The CLR's DLLs can load into a process address space
- AppDomain (a CLR concept)
 - Logically sub-divides a process' virtual address space
 - Created/destroyed more cheaply than a process
 - Provides managed EXE/DLL code & data isolation
 - Unloading, security sandbox, configuration
 - There is one managed heap per process
 - But an AppDomain "owns" the objects its code constructs



What is a Thread? (A Windows and CLR Concept)

- Executes code in a process/AppDomain
 - Has CPU register set (called its context)
 - x86 = ~700 bytes, x64 = ~1240 bytes, IA64 = ~2500 bytes
 - Has stack memory allocated
 - User-mode=1MB committed, Kernel-mode=12KB/24KB
 - For parameter/local variables, exception-handling chain
- Windows: Threads are confined to a process address space
 - When process dies, all threads die
- CLR: Thread can cross AppDomain boundaries
 - A thread is in 1 AppDomain at any one time
 - Thread executes code with AppDomain's security/configuration
 - Other AppDomains can be on the thread's stack

Windows Schedules Thread Execution

- All threads appear to run "simultaneously"
- Windows schedules a thread to each CPU
- Windows allows a thread to run for a *time quantum* (~20ms)
 - When quantum expires, Windows performs a *context switch*
 - A thread can voluntarily end its time quantum early
 - Usually by waiting for input (keyboard, mouse, network, file)
- Windows won't schedule a waiting thread
 - In reality, most threads in the system are waiting for something
- If a machine has multiple CPUs, hyper-threaded CPUs, or multi-core CPUs, Windows schedules threads so that several are *truly* executing simultaneously

Creating a Thread

```
using System;
using System.Threading;

public static class Program {
    // 1. Define the thread method
    private static void ThreadMethod(Object state) {
        // Do whatever here...
    }

    public static void Main() {
        // 2. Construct a Thread object
        Thread t = new Thread(ThreadMethod);

        // 3. Create a thread and let it run
        t.Start("Initialization data");

        // Do whatever here...

        // 4. Optional: Wait for thread to die
        t.Join();
    }
}
```

Reasons to Create Threads

- Two reasons to create threads
 - Isolate code from other code: Reliability/easier coding
 - Concurrent execution: Scalability on multi-processor machines
- Benefits
 - Keep the CPU busy
 - User gets more features with no learning curve
 - App reliability (no hangs, ability to cancel, responsive UI)
- Examples
 - Indexing files for fast searching
 - Defragmenting disk for better I/O speed
 - Building your project when you stop typing
 - Recalculating spreadsheet cells
 - Spell/grammar checking documents
 - Copying/printing files
 - Resizing an app's window while processing I/O

Thread Scheduling



Temporarily Suspending a Thread

- Call Sleep to suspend a thread for an approximate time
 - Thread relinquishes the remainder of its quantum
 - Windows schedules another thread
 - May be the same thread if time has elapsed

```
public class Thread {
    public static void Sleep(int milliseconds);
    public static void Sleep(TimeSpan timeout);
}
```

Priority Scheduling

NOTE: MS periodically tweaks the scheduler

- Internally, every thread has a base priority number
 - 0 (lowest & reserved) to 31 (highest)
 - Windows schedules highest-priority runnable threads to the CPUs
 - High-priority threads prevent low-priority threads from running
 - Unless multiple CPUs are available
 - Windows boosts a thread's priority when an "event" occurs
 - A thread is never boosted above 15
 - Boost decays after each quantum to the base priority
- Externally, Windows provides an abstract view
 - Processes: 6 priority classes
 - Threads: 7 relative thread priorities

Windows Priority Mapping

Relative Thread Priority	Process Priority Class					
	Idle	Below Normal	Normal	Above Normal	High	Real time
Time critical	15	15	15	15	15	31
Highest	6	8	10	12	15	26
Above normal	5	7	9	11	14	25
Normal	4	6	8	10	13	24
Below normal	3	5	7	9	12	23
Lowest	2	4	6	8	11	22
Idle	1	1	1	1	1	16

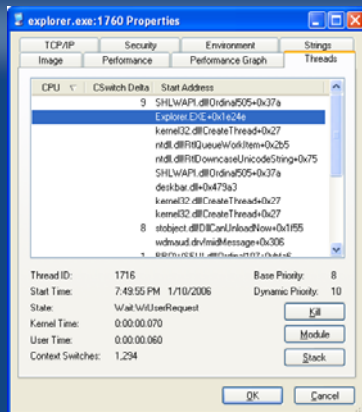
Priority Scheduling Notes

- OS threads run in the real time range
 - Threads in this range adversely affect OS threads
 - You must be an Administrator to run a process in this range
- You can use CMD's Start command to run a process in a specific process priority class
- Typically, it is best to lower a thread's priority
 - Used for long-running compute-bound tasks
 - Won't adversely affect other processes
- Typically, you should avoid increasing a thread's priority
 - Used for tasks that need to react immediately to something, execute for a very short time, and then block
 - Can adversely affect other processes

FCL Process/Thread Classes

- System.Diagnostics: Process & ProcessThread
 - Classes provide Windows' view of process/thread (avoid)
 - Callers require high security to access members
- System.Threading: Thread
 - Class provides CLR (logical/managed) view of thread (use)
 - No process class; managed code can't control the host process
 - Callers require less security to access members
- Properties that affect scheduling
 - Process: PriorityClass(g/s), PriorityBoostEnabled(g/s)
 - Internal view: BasePriority(g)
 - ProcessThread: PriorityLevel(g/s), PriorityBoostEnabled(g/s)
 - Internal view: BasePriority(g), CurrentPriority(g),
 - Thread: Priority(g/s)

Process Explorer's Threads Tab



Discussion
