

Performing Asynchronous Operations with the CLR's Thread Pool



Jeffrey Richter
Copyright © 2008
www.wintellect.com

Introduction

- Performing asynchronous operations is the secret to creating responsive, scalable applications
 - Threads don't block waiting for some operation to complete
 - A blocked thread wastes resources (stack, kernel object, etc)
 - A blocked thread causes you to create more threads
 - More threads & deep stacks increase GC time
 - You can take advantage of multiple CPUs in the machine
- Two kinds of asynchronous operations
 - Compute-bound: Requires a thread to do the work
 - I/O-bound: Device driver has hardware do the work; no threads required!

Introducing the CLR's Thread Pool

- The CLR ThreadPool manages thread creation for you
 - The TP has threads for your application's use (initially 0)
 - There is one TP per process, shared by all AppDomains
- TP has a queue of async op requests called *work items*
 - A work item is a delegate referring to a callback method
- To start an async op, call method to append a delegate to the queue
- TP thread is created (if necessary), extracts the work item and executes the callback method
 - Creating a thread causes a performance hit



Introducing the CLR's Thread Pool

- When method returns, the next work item is dequeued
 - The thread doesn't die (improving performance)
- Queued work items are serviced by 1 thread (sequentially)
- If work items queue quickly, more threads are created
 - Eventually, a small # of threads handle lots of work items
- If work items reduce, TP threads sit idle (doing nothing)
 - This wastes resources
 - A TP thread that waits 2 minutes, wakes and kills itself
 - Perf hit but not a problem since app isn't doing much anyway

Performing Asynchronous Compute-Bound Operations



Queuing a Work Item

- Use ThreadPool's static methods to queue a work item
 - Optionally pass some state data
 - Methods return immediately
 - Return 'true' or throw OutOfMemoryException
- If callback method throws an unhandled exception, the CLR terminates the process
 - Unless the host imposes its own policy

```
static Boolean QueueUserWorkItem(WaitCallback callback, Object state);  
delegate void WaitCallback(Object state);
```

Using a Dedicated Thread to Perform an Asynchronous Compute-Bound Operation

- Use a dedicated thread if work item requires unusual thread state
 - Non-normal thread priority
 - Foreground thread to keep app alive until work item is done
 - Long-running compute-bound task (usually low-priority)

```
public sealed class Thread : CriticalFinalizerObject, ... {  
    public Thread(ParameterizedThreadStart start);  
    // Less commonly used constructors are not shown here  
}
```

```
// Same signature as WaitCallback  
delegate void ParameterizedThreadStart(Object state);
```

Periodically Performing an Asynchronous Compute-Bound Operation

- A Timer object tells the CLR to call a method periodically
- Internally, the CLR uses 1 thread for all Timer objects
 - When a Timer's time comes due, thread posts a TP work item
- If work item is "slow", the timer could go off again
 - Work Item method should be re-entrant and thread safe
- When a Timer object is GC'd, the CLR cancels the timer

```
public sealed class Timer : MarshalByRefObject, IDisposable {  
    public Timer(TimerCallback callback, Object state,  
                Int32 dueTime, Int32 period);  
  
    public Boolean Change(Int32 dueTime, Int32 period);  
    public void Dispose();  
    // Less commonly used methods are not shown here  
}
```

```
delegate void TimerCallback(Object state);
```

Timers

```
using System;  
using System.Threading;  
  
class App {  
    public static void Main() {  
        Console.WriteLine("Checking status updates every 2 seconds.");  
        Console.WriteLine("    (Hit Enter to terminate the sample)");  
        Timer timer = new Timer(CheckStatus, null, 0, 2000);  
        Console.ReadLine();  
        timer.Dispose(); // Stop the timer  
    }  
  
    static void CheckStatus(Object state) {  
        Console.WriteLine("Checking Status.");  
        // ...  
    }  
}
```

Discussion
