

Programming in .NET

Microsoft Development Center Serbia programming course

Lesson 8 – Parallelism and Threading in .NET

Example 1

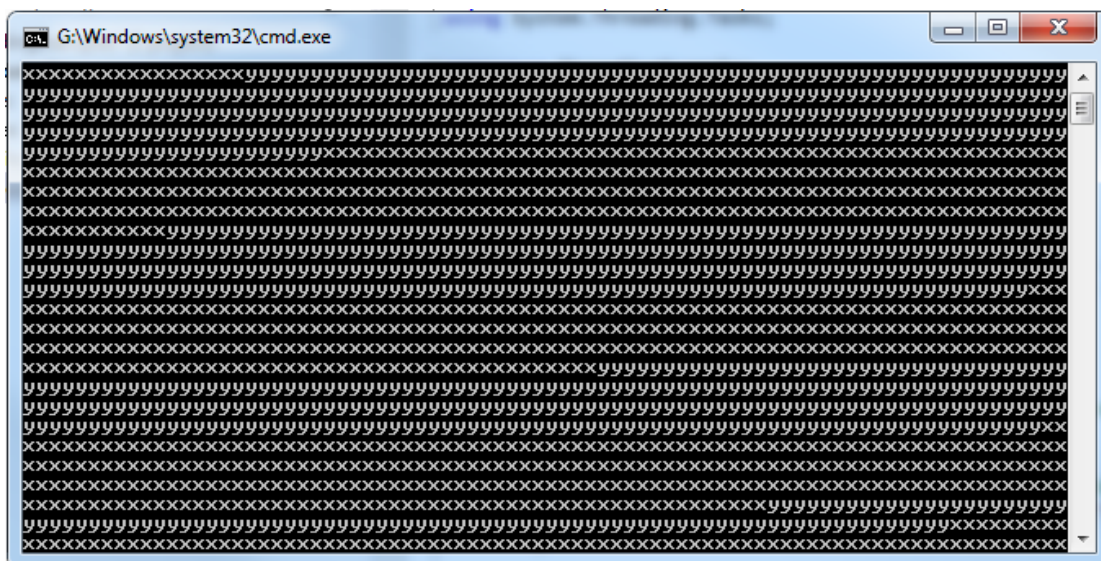
Our first example shows basics about threading in C#. It covers basic thread creation, thread start method and parallel thread execution.

```
private static void Example1()
{
    Thread t = new Thread(WriteY); // Kick off a new thread
    t.Start(); // running WriteY()

    // Simultaneously, do something on the main thread.
    for (int i = 0; i < 1000; i++) Console.Write("x");
}

private static void WriteY()
{
    for (int i = 0; i < 1000; i++) Console.Write("y");
}
```

The expected output follows:



Here we can see parallel thread execution, context switching and unpredictable outcome.

Example 2

Example 2 is a basic race condition scenario. Here we have test and set operations on the shared (not guarded) variable.

```
private static bool done; // Static fields are shared between all threads

private static void Example2()
{
    var t = new Thread(Go);
    t.Name = "Thread 2";
    t.Start();
    Go();
}

private static void Go()
{
    if (!done)
    {
        done = true;
        Console.WriteLine("Done");
    }
}
```

In most of the cases we will have single “Done” written, still in some of the cases we will have “Done” string written from both threads. This piece of code is considered unsafe!

While working with shared memory you should always keep in mind

*Shared data is the primary cause of complexity and obscure errors in multithreading. Although often essential, it pays to keep it **as simple as possible**.*

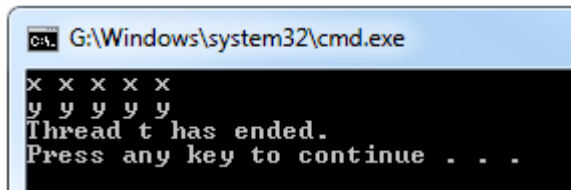
Example 3

This example is focusing on some basic thread operations – sleep, join and yield

```
private static void Example3()
{
    Thread t = new Thread(Go3);
    t.Start();
    t.Join(); // timeout can be added
    Console.WriteLine("Thread t has ended.");
}

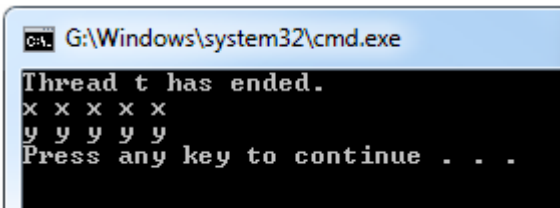
private static void Go3()
{
    for (int i = 0; i < 5; i++)
    {
        Console.Write("x ");
    }
    Console.Write(Environment.NewLine);
    Thread.Sleep(500); // or we can use Thread.Yield();
    for (int i = 0; i < 5; i++)
    {
        Console.Write("y ");
    }
    Console.Write(Environment.NewLine);
}
```

In the example above the outcome is deterministic:



```
C:\G:\Windows\system32\cmd.exe
x x x x x
y y y y y
Thread t has ended.
Press any key to continue . . .
```

Still, if we are about to change an example and skip `Thread.Join` command, we can get three different results, where to most common one would be:



```
C:\G:\Windows\system32\cmd.exe
Thread t has ended.
x x x x x
y y y y y
Press any key to continue . . .
```

A few useful notes related to this example:

Thread.Sleep(0) relinquishes the thread's current time slice immediately, voluntarily handing over the CPU to other threads. Framework 4.0's new Thread.Yield() method does the same thing — except that it relinquishes only to threads running on the same processor.

Sleep(0) or Yield is occasionally useful in production code for advanced performance tweaks. It's also an excellent diagnostic tool for helping to uncover thread safety issues: if inserting Thread.Yield() anywhere in your code makes or breaks the program, you almost certainly have a bug.

Example 4

This example describes one of the common mistakes in parallel programming. It is an example of the access to modified closure.

```
private static void Example4()
{
    for (int i = 0; i < 10; i++)
        new Thread(() => Console.Write(i)).Start();
}
```

In this case output will be completely unpredictable. Still this problem is easy to overcome by copying shared value into the local variable:

```
private static void Example4()
{
    for (int i = 0; i < 10; i++)
    {
        int i1 = i;
        new Thread(() => Console.Write(i1)).Start();
    }
}
```

Example 5

The following example is all about Tasks. It describes Task as a mechanism for parallel, asynchronous code execution. The example is also showing basic usage of the one of the Task's most interesting property – Result.

```
private static void Example5()
{
    // Start the task executing:
    Task<string> task = Task.Factory.StartNew<string>
        (() => DownloadString("http://www.bing.com"));

    // We can do other work here and it will execute in parallel:
    RunSomeOtherMethod();

    // When we need the task's return value, we query its Result property:
    // If it's still executing, the current thread will now block (wait)
    // until the task finishes:
    string result = task.Result;
    Console.WriteLine(result);
}

private static void RunSomeOtherMethod()
{
    // Do some smart things here..
}

private static string DownloadString(string uri)
{
    using (var wc = new WebClient())
        return wc.DownloadString(uri);
}
```

Example 6

Sixth example is presenting two threading concepts – Interrupt and Abort.

Calling Interrupt on a blocked thread forcibly releases it, throwing a ThreadInterruptedException. Interrupting a thread does not cause the thread to end, unless the ThreadInterruptedException is unhandled.

If Interrupt is called on a thread that's not blocked, the thread continues executing until it next blocks, at which point a ThreadInterruptedException is thrown.

A blocked thread can also be forcibly released via its **Abort** method. This has an effect similar to calling **Interrupt**, except that a **ThreadAbortException** is thrown instead of a **ThreadInterruptedException**. Furthermore, the exception will be re-thrown at the end of the **catch** block (in an attempt to terminate the thread for good) unless **Thread.ResetAbort** is called within the **catch** block.

The big difference between **Interrupt** and **Abort** is what happens when it's called on a thread that is not blocked. Whereas **Interrupt** waits until the thread next blocks before doing anything, **Abort** throws an exception on the thread right where it's executing (unmanaged code excepted). This is a problem because .NET Framework code might be aborted — code that is not abort-safe. For example, if an abort occurs while a **FileStream** is being constructed, it's possible that an unmanaged file handle will remain open until the application domain ends. This rules out using **Abort** in almost any nontrivial context.

```

private static void Example6()
{
    Thread t = new Thread(delegate()
    {
        try
        {
            Thread.Sleep(Timeout.Infinite);
        }
        catch (ThreadInterruptedException)
        {
            Console.WriteLine("Forcibly ");
        }
        catch (ThreadAbortException)
        {
            Console.WriteLine("Forcibly ");
            // ThreadAbortException re-thrown
        }
        // This won't execute in case of thread abort
        Console.WriteLine("Woken!");
    });

    t.Start();
    Thread.Sleep(50);
    t.Abort();
    // t.Interrupt();
}

```

And, of course, a friendly notes to keep in mind:

Interrupt and Abort can cause considerable trouble: it's precisely because they seem like obvious choices in solving a range of problems that it's worth examining their pitfalls.

*An unhandled **ThreadAbortException** is one of only two types of exception that does not cause application shutdown (the other is **AppDomainUnloadException**).*

Example 7

This example is trying to show a pattern for a safe thread cancelation. It is here to demonstrate a mechanism how to overcome problems with **Abort** and **Interrupt**. Of course this logic has already been implemented in .NET (please take a look at **Cancellation Tokens**).

```

internal class RulyCanceler
{
    private object cancelLocker = new object();
    private bool cancelRequest;

    public bool IsCancellationRequested
    {
        get { lock (_cancelLocker) return _cancelRequest; }
    }

    public void Cancel()
    {
        lock (_cancelLocker) _cancelRequest = true;
    }

    public void ThrowIfCancellationRequested()
    {
        if (IsCancellationRequested) throw new OperationCanceledException();
    }
}

```

```

private static void Example7()
{
    RulyCanceler canceler = new RulyCanceler();
    new Thread(() =>
        {
            try
            {
                Work(canceler);
            }
            catch (OperationCanceledException)
            {
                Console.WriteLine("Canceled!");
            }
        })
        .Start();
    Thread.Sleep(1000);
    canceler.Cancel(); // Safely cancel worker.
}

private static void Work(RulyCanceler c)
{
    while (true)
    {
        c.ThrowIfCancellationRequested();
        // ...
        try
        {
            OtherMethod(c);
        }
        finally
        {
            /* any required cleanup */
        }
    }
}

private static void OtherMethod(RulyCanceler c)
{
    // Do stuff...
    c.ThrowIfCancellationRequested();
}

```

Example 8

This is just another short example explaining very useful thread property – `IsBackground`. **Foreground** threads keep the application alive for as long as any one of them is running, whereas **background** threads do not. Once all foreground threads finish, the application ends, and any **background** threads still running abruptly terminate. By default, threads you create explicitly are **foreground** threads.

```

public static void Example8()
{
    Thread t = new Thread(Go8) {IsBackground = true};
    t.Start();
    Thread.Sleep(500);
}

public static void Go8()
{
    while (true)
    {
        Console.Write("a");
    }
}

```

So, the expected behavior of this piece of code is to write letter “a” for a half of the second and then terminate.

Example 9

In the following example we are about to show basic usage of the .NET *BackgroundWorker* class. *BackgroundWorker* is a helper class in the *System.ComponentModel* namespace for managing a worker thread. It can be considered a general-purpose implementation of the Event-Based Asynchronous Pattern (EAP), and provides the following features:

- A cooperative cancellation model
- The ability to safely update WPF or Windows Forms controls when the worker completes
- Forwarding of exceptions to the completion event
- A protocol for reporting progress
- An implementation of *IComponent* allowing it to be sited in Visual Studio's designer

```
private static BackgroundWorker _bw9 = new BackgroundWorker();

private static void Example9()
{
    bw9.DoWork += Bw9DoWork;
    _bw9.RunWorkerAsync("Message to worker");
    Console.ReadLine();
}

private static void Bw9DoWork(object sender, DoWorkEventArgs e)
{
    // This is called on the worker thread
    Console.WriteLine(e.Argument); // writes "Message to worker"
    // Perform time-consuming task...
}
```

One fact you should follow:

***BackgroundWorker* uses the thread pool, which means you should never call *Abort* on a *BackgroundWorker* thread.**

Example 10

In the following example we have shown more advance use of the *BackgroundWorker* class. *ProgressChanged* and *RunWorkerCompleted* are the same (event based) construct as *DoWork*. This example is providing more realistic use case for the *BackgroundWorker* as a helper class.

```
private static void Example10()
{
    bw10 = new BackgroundWorker
    {
        WorkerReportsProgress = true,
        WorkerSupportsCancellation = true
    };
    bw10.DoWork += Bw10DoWork;
    bw10.ProgressChanged += Bw10ProgressChanged;
    _bw10.RunWorkerCompleted += Bw10RunWorkerCompleted;

    _bw10.RunWorkerAsync("Hello to worker");
}
```

```

    Console.WriteLine("Press Enter in the next 5 seconds to cancel");
    Console.ReadLine();
    if ( bw10.IsBusy) bw10.CancelAsync();
    Console.ReadLine();
}

private static void Bw10DoWork(object sender, DoWorkEventArgs e)
{
    for (int i = 0; i <= 100; i += 20)
    {
        if (_bw10.CancellationPending)
        {
            e.Cancel = true;
            return;
        }
        _bw10.ReportProgress(i);
        Thread.Sleep(1000); // Just for the demo... don't go sleeping
    } // for real in pooled threads!

    e.Result = 123; // This gets passed to RunWorkerCompleted
}

private static void Bw11RunWorkerCompleted(object sender,
                                           RunWorkerCompletedEventArgs e)
{
    if (e.Cancelled)
        Console.WriteLine("You canceled!");
    else if (e.Error != null)
        Console.WriteLine("Worker exception: " + e.Error.ToString());
    else
        Console.WriteLine("Complete: " + e.Result); // from DoWork
}

private static void Bw10ProgressChanged(object sender,
                                         ProgressChangedEventArgs e)
{
    Console.WriteLine("Reached " + e.ProgressPercentage + "%");
}

```

Example 11

This example is all about effectiveness and proper use of the thread safe and thread unsafe collections. For this example we have used *List<int>* and *ConcurrentBag<int>* as a collections of the integers. We are trying to insert a million random integers in the collections both sequential and parallel.

```

private static int N = 1000000;
private static int MaxInt = 1000;
private static Random randgen = new Random();

private static int CalcRandom()
{
    return randgen.Next(MaxInt);
}

private static void Test1ParallelSimpleList()
{
    List<int> results = new List<int>();

    try
    {
        Stopwatch sw1 = new Stopwatch();
        sw1.Start();

        Parallel.For(0, N, i =>
            {
                try
                {
                    results.Add(CalcRandom());
                }
            }
        );
    }
}

```



```

        }
        catch (Exception)
        {
            //do nothing
        }
    });

    sw1.Stop();

    Console.WriteLine("Elapsed: " + sw1.ElapsedMilliseconds);
    Console.WriteLine(results.Count);

}
catch (Exception ex)
{
    // Console.WriteLine(ex);
}
}

private static void Test1ParallelConcurrentBag()
{
    ConcurrentBag<int> results = new ConcurrentBag<int>();

    try
    {
        Stopwatch sw1 = new Stopwatch();
        sw1.Start();

        Parallel.For(0, N, i =>
            {
                results.Add(CalcRandom());
            });

        sw1.Stop();

        Console.WriteLine("Elapsed: " + sw1.ElapsedMilliseconds);
        Console.WriteLine(results.Count);

    }
    catch (Exception ex)
    {
        Console.WriteLine(ex);
    }
}

private static void Test1SerialSimpleList()
{
    List<int> results = new List<int>();

    try
    {
        Stopwatch sw2 = new Stopwatch();
        sw2.Start();

        for (int i = 0; i < N; ++i)
        {
            results.Add(CalcRandom());
        }

        sw2.Stop();

        Console.WriteLine("Elapsed: " + sw2.ElapsedMilliseconds);
        Console.WriteLine(results.Count);

    }
    catch (Exception ex)
    {
        Console.WriteLine(ex);
    }
}

private static void Test1SerialConcurrentBag()
{
    ConcurrentBag<int> results = new ConcurrentBag<int>();

```

```

try
{
    Stopwatch sw2 = new Stopwatch();
    sw2.Start();

    for (int i = 0; i < N; ++i)
    {
        results.Add(CalcRandom());
    }

    sw2.Stop();

    Console.WriteLine("Elapsed: " + sw2.ElapsedMilliseconds);
    Console.WriteLine(results.Count);
}
catch (Exception ex)
{
    Console.WriteLine(ex);
}
}

private static void Example12()
{
    Test1ParallelConcurrentBag();
    Test1ParallelSimpleList();
    Test1SerialConcurrentBag();
    Test1SerialSimpleList();
}

```

This is one of the possible outcomes:

```

G:\Windows\system32\cmd.exe
[ConcurrentBag parallel] elapsed: 114 ms
Number of keys inserted: 1000000
[Simple list - parallel] elapsed: 6751 ms
Number of keys inserted: 973208
[Concurrent bag - serial] elapsed: 162 ms
Number of keys inserted: 1000000
[Simple list - serial] elapsed: 24 ms
Number of keys inserted: 1000000
Press any key to continue . . .

```

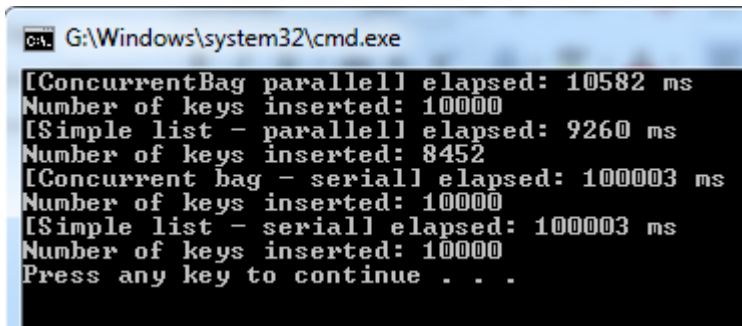
First thing we should notice is that in case of simple list with parallel execution we don't have correct result. Therefore time needed for this operation won't be considered – we have to avoid usage of the thread unsafe structures in the parallel context. Second observation: Simple list with the sequential execution is an order of magnitude faster than concurrent bag. This is the fact mainly because generation of the random integer is a cheap operation and the overhead produced by thread safe collection is way bigger than actual meaningful operation. And finally what we are seeing is that ConcurrentBag in the parallel context is still faster than ConcurrentBag with the serial inserts. Basically, this means that the overhead of the parallelism (thread initialization, context switching, etc...) is smaller than the actual benefit of the parallel execution. For practice, you are encouraged to implement more intensive *CalcRandom* method and discuss the results. Here is one example:

```

private static int N = 10000;
private static int CalcRandom()
{
    Thread.Sleep(10);
    return randgen.Next(MaxInt);
}

```

And the results also:



```
CA: G:\Windows\system32\cmd.exe
[ConcurrentBag parallel] elapsed: 10582 ms
Number of keys inserted: 10000
[Simple list - parallel] elapsed: 9260 ms
Number of keys inserted: 8452
[Concurrent bag - serial] elapsed: 100003 ms
Number of keys inserted: 10000
[Simple list - serial] elapsed: 100003 ms
Number of keys inserted: 10000
Press any key to continue . . .
```

Example 12

PLINQ automatically parallelizes local LINQ queries. PLINQ has the advantage of being easy to use in that it offloads the burden of both work partitioning and result collation to the Framework.

To use PLINQ, simply call *AsParallel()* on the input sequence and then continue the LINQ query as usual. The following query calculates the prime numbers between 3 and 100,000 — making full use of all cores on the target machine:

```
// Calculate prime numbers using a simple (unoptimized) algorithm.
private static void Example13()
{
    IEnumerable<int> numbers = Enumerable.Range(3, 100 - 3);
    var parallelQuery =
        from n in numbers.AsParallel()
        where Enumerable.Range(2, (int) Math.Sqrt(n)).All(i => n%i > 0)
        select n;

    parallelQuery.ToList().ForEach(Console.WriteLine);
}
```

Good to know:

*If a PLINQ query throws an exception, it's re-thrown as an **AggregateException** whose **InnerExceptions** property contains the real exception (or exceptions).*