May 13, 2016

EE 356
Notes on Threading

A *process* is a program in execution. In an operating system a process has a well-defined state. A five-state model of a process includes: New – a process is admitted for high-level scheduling; Ready – A process is ready to execute and waiting on processor availability; Running – the process is being executed; Waiting – the process is suspended waiting on system resources; Halted – the process is terminated.

A *thread* is a path of execution within an executable process. It has also been defined as a single sequential flow of control and it is the smallest unit of processing executed by an operating system. Typically a thread has its own separate call stack and its own local variables.

Processes are usually independent of one another while threads within a process may be dependent on one another. Processes also typically have separate virtual address spaces whereas threads within a process share an address space. From a programmer's perspective it usually is faster to switch between threads than it is to switch between processes.

Multiple processes can run in parallel on a single computer and multiple threads can run in parallel in a single process.

A single process can spawn multiple threads which can be executed on multiple CPU cores in parallel. Threading gives the programmer the ability to make use of multi-core machines. Without threading, most processes run on a single core regardless of how many cores are available.

Threads are schedule to run by the operating system. There is therefore an advantage to using threads even on machines that have only a single core. On a single core machine, a thread can run in the background leaving a main program responsive to user input from the mouse or keyboard. Without the thread, a computation bound process can appear to freeze up and not respond to user input.

The operating system typically uses either preemptive multi-threading or cooperative multithreading to switch between threads.

Preemptive multi-threading is also called *interleaved multi-threading* or *fine-grained multi-threading*. The CPU works with two or more threads simultaneously and switches between them based on the clock cycle or whether or not a given thread is blocked. In general, a thread has little control over when it gets preempted.

Cooperative multi-threading also called *blocked multi-threading* or *coarse-grained multi-threading.* The CPU executes a single thread until the thread is blocked by some event or until the thread itself releases control.

In C# (and other languages) threading is an easy way to allow the user to take advantage of multi-core processors and do true parallel execution.

May 13, 2016

Consider the following example of how threading works:

```
   {class Program
      {static void Main(string[] args)
        {int i;
         //Must have Using System.Threading
         Thread t = new Thread (WriteY); // Kick off a new thread
         t.Start(); // running WriteY()
         // Simultaneously, do something on the main thread.
         for (i=0;i<500;i++)
            Console.Write ("x");
        }
      static void WriteY()
       {int i;
        for(i=0;i<500;i++)
          Console.Write ("y");
       }
    }
}
/*Prints the following
xyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxx
yyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyxxxxxxxyyyyyyxxxxxxxxxxxxx
xxxxxxxxxxyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyxyxxxxxxxxxxxxxxyyyyyyyy
yyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxyyyyy
yyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxyyxyxxxxxxxyyyyyyyyyyyyyyyxxxxxxxxxxxxxx
xxxxyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyxxxxxxyyyyyyyyy
yyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxyyyyyyyyyyyyyy
yyyyyyxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxyyyyyyyyyyyyyxxxx
xxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxyyxxxxxxxxxxxxxyyyy
yyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxyyyyyyyyyyy
yyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyy
yyyyyxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyxxxxyyyyxxxxxxxxxxxxxxxxxxxxyyy
yyyyyyyyxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyPress any key to continue . . .
*/
```

Figure 1

Two threads printing x and y.

We create a new thread named *t* in the main program which will run the method *WriteY*. After starting the thread the main program proceeds to print out 500 copies of x. The method WriteY prints 500 copies of y. From the printed output we see that the two threads run at the same time – even on a single core machine the two threads will be time sliced to give the appearance of running on two machines.

Creating a Thread

Threads are created in C# using a delegate called ThreadStart

```
public delegate void ThreadStart();
```

The thread itself is created when you activate this delegate

```
Thread t = new Thread(new ThreadStart(MyMethod));
```

To start the thread running you enter

```
t.Start();
```

Note that the user does not create the delegate – it is created by the Thread library and the user does not see it. You can shorten the process with the following syntax

```
Thread t = new Thread(MyMethod);
t.Start();
```

It is also possible to create a thread that takes a parameter since the constructor is overloaded. The syntax looks like this:

```
Thread t = new Thread(MyMethod);
t.Start(count);
...
static void MyMethod(obj MethodObject)
  {int c = (int) MethodObject;  //cast
   ...
```

We can alter the previous example to accept a character parameter like this

```
{class Program
    {static void Main(string[] args)
        {int i;
         //Must have Using System.Threading
         Thread t = new Thread(WriteY); // Kick off a new thread
         t.Start('*'); // running WriteY()
         // Simultaneously, do something on the main thread.
         for (i = 0; i < 500; i++)
             Console.Write("x");
        }
    static void WriteY(object obj)
       {int i;
        for (i = 0; i < 500; i++)
           Console.Write((char)obj);
       }
    }
}
```

Figure 2
A thread with a parameter.

The parameter passage is limited to a single parameter using this technique. If you need to pass more parameters you use *Lambda expressions* or the *ParameterizedThreadStart* delegate. For details on how to use Lambda expressions see Troelson pp. 390-397. The ParameterizedThreadStart method is explained in Troelson pp. 717-719

Thread Names and Other Properties

You do not have to name a thread but doing so allows you to track it when you do debugging. Each thread has a *Name* property which can be set one time.

```
Thread t = new Thread(WriteY); // Kick off a new thread
t.Name = "MyThread";
t.Start('*'); // running WriteY()
```

The following example shows how to obtain other thread properties (from Troelson Fourth Edition pp. 742-743).

```
{class Program
   {static void Main(string[] args)
      {Thread mainThread = Thread.CurrentThread;
       mainThread.Name = "MainThread";
       //AppDomain and Context
       Console.WriteLine("Current AppDomain: {0}",
                         Thread.GetDomain().FriendlyName);
       Console.WriteLine("Current ContextID: {0}",
                         Thread.CurrentContext.ContextID);
       Console.WriteLine("Thread Name: {0}", mainThread.Name);
       Console.WriteLine("Has started: {0}", mainThread.IsAlive);
       Console.WriteLine("Priority level: {0}", mainThread.Priority);
       Console.WriteLine("Thread state: {0}", mainThread.ThreadState);
      }
   }
}
/* Prints the following
Current AppDomain: ThreadProperties.exe
Current ContextID: 0
Thread Name: MainThread
Has started: True
Priority level: Normal
Thread state: Running
Press any key to continue . . .
*/
```

Figure 3
Thread properties.

Foreground and Background Confusion

It is possible for the user to assign a thread either *foreground* or *background* status. This is often confused with the thread's priority but foreground and background status do not imply or convey any priority to the thread.

If a thread is in the foreground, the application cannot end until the thread terminates. If a thread is in the background it is considered expendable and an application can terminate even if a background thread is still working.

A thread created using Thread.Start() is by default a foreground thread. You can make a thread a background thread like this:

```
Thread back = new Thread(new ThreadStart(MyMethod));
back.IsBackground = true;
back.Start();
```

For example, suppose you have a thread that is doing nothing but updating the time on the user's screen. This thread could be assigned as a background thread and need not be closed to terminate the application.

Thread Priority

There are five levels of thread priority: Lowest; BelowNormal; Normal; AboveNormal; and Highest. A thread's priority is meaningful only if you are running multiple threads in one process. Higher priority threads get more execution time than lower priority threads. However, the application in which a process is running also has a priority and the application has to be running in order for the threads to run. A high priority thread running on a low priority application will not get as much run time as a lower priority thread running on a high priority application. By default, threads are given thread priority "normal". Using the code in Figure 2, you can set the thread priority to highest like this

```
t.Priority = ThreadPriority.Highest;
t.Start('*'); // running WriteY()
```

Running the program using this priority allows the printing of the '*' character to finish first even though the two characters are interlaced.

Synchronization Issues

When two or more threads run at the same time, the user has only a very coarse control over which thread will finish first and often the threads will not complete in the same order. Since threads can share resources and often have links to other threads by way of the data they operate on, the user must take care that some operations get done in the proper order. The process of doing this is called *thread synchronization*. There are four mechanisms for doing thread synchronization: Blocking; Locking; Signaling; and Non-blocking synchronization.

Blocking

To use blocking we force one thread to wait for some period of time or we force it to wait until another thread has completed its task. `sleep`, and `join` are typically used in blocking. A method which calls sleep suspends itself for some period of time. Join blocks a calling thread until the specified thread (the one on which join is called) exits.

Locking

In locking we a thread acquires a *lock* which effectively prevents other threads from interrupting until the lock is relinquished. The normal way to do this is:

```
private object threadlock = new object();
…
public void MyMethod()
  {lock(threadlock)
     {//everything in this scope cannot be interrupted.
      ...
     }
  }
```

An example of the use of locking is in Troelson pp. 751-754.

Signaling

For signaling we allow a thread to pause until receiving a notification from another thread. The common signaling device is: Monitor wait/pulse method. These are not discussed in Troelson, but there is a good discussion in C# 4.0 In a Nutshell pp. 840-849.

Non-blocking synchronization

Non-blocking synchronization makes use of something called *memory barriers*. Effectively, memory barriers keep two processes from overwriting the same variable or area of memory. See pp. 825-832 in C# 4.0 In a Nutshell.

Using Threading

Thread Pool Concept
Creating and destroying thread resources can be computationally expensive. In C# a thread pool has been created to reduce this overhead. The idea is that when you need a thread you get it from those available in the pool. This allows the number of threads to be limited and better managed. When you reach the limit of the thread pool, new threads are added in a queue so as not to overburden the CPU. You can access threads from the pool by way of the Background Worker thread, the Task Parallel Library, using delegates, or by the ThreadPool class. For more discussion of the thread pool see Troelson pp. 760-761.

Background Worker
BackgroundWorker is a helper class that is in the System.ComponentModel namespace. It makes it very easy to use a thread to get asynchronous behavior. The following code illustrates a simple application of a BackgroundWorker.

```
{class Program
   {//Create a background worker
    static BackgroundWorker bw = new BackgroundWorker();
    static void Main(string[] args)
      {bw.DoWork += DoWorkMethod;  //Add method to worker
       bw.RunWorkerAsync("Message to Worker");
       Console.WriteLine("Main");
      }
    //Create the method that does the work
    static void DoWorkMethod(object sender, DoWorkEventArgs e)
      {Console.WriteLine(e.Argument);
       Console.WriteLine("Done");
       Thread.Sleep(5000);
      }
   }
```

Figure 4
Background worker in a console program.

The background worker class also provides a way to monitor the progress on the work it is doing, a way to forward errors to the program that started the worker, and a way to easily determine when the work is done. The following example has a GUI. The user enters a number of seconds into the text box. The background worker then counts off the number of quarter seconds and reports progress after each.

```
{public partial class Form1 : Form
  {public Form1()
    {InitializeComponent();
    }
 private void btnStart_Click(object sender, EventArgs e)
    {int s =  Convert.ToInt32(txt1.Text);
     pgb1.Maximum = 4*s;  //maximum value for progress bar
     bgw1.RunWorkerAsync(s); //Do work asynchronously
    }
private void bgw1_DoWork(object sender, DoWorkEventArgs e)
   {int i;
    for(i=0;i<4*(int)e.Argument;i++)
      {Thread.Sleep(250);       //update progress
       bgw1.ReportProgress(i);  //  each quarter second
      }
    e.Result = 4*(int)e.Argument; //Return value when done
   }
private void bgw1_ProgressChanged(object sender,
                                  ProgressChangedEventArgs e)
  {pgb1.Value = e.ProgressPercentage; //Write progress bar
  }
private void bgw1_RunWorkerCompleted(object sender,
                               RunWorkerCompletedEventArgs e)
 {if(e.Cancelled)
    txt1.Text = "Cancelled";
  else if(e.Error != null)
    txt1.Text = (e.Error).ToString();
  else
    txt1.Text = "Done";
 }
}
```

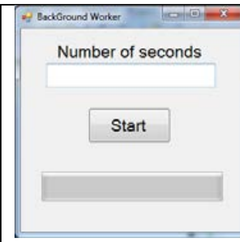

Figure 5
GUI background worker with progress and termination reported.

In a Forms or WFC application the background worker can be dragged as a component into the application. The RunWorkerCompleted, DoWork, and ProgressChanged are events.

The Background Worker provides an easy way to use threads to take advantage of a multi-core machine when the application has a few computationally intensive sections that can be isolated to a thread.

Task Parallel Library
The task parallel library (TPL) is a set of types which will automatically distribute your application across available CPUs using the thread pool in the common language runtime (CLR). As a developer you need not be concerned with how to best partition your application or how to manage your threads. All of the scheduling, partitioning, and thread management is done for you without any intervention.

According to Troelson, the TPL is now the recommended way to write parallel applications in .NET 4.0.

There are a large number of methods in the parallel class. The two that are used most often are the Parallel.For() and the Parallel.ForEach(). If you are writing code that uses a *for* or *foreach* structure to iterate over a large collection of data in an array, ArrayList, a List<T>, or LinQ query, you can substitute the Parallel class versions of these structures and C# will take care of running the loops in parallel. The following example creates three arrays of 10,000,000 random doubles. Each element in the first array is square root of the element in the second array divided by the square root of the element in the third array. The GUI interface has a "Load Data" button which fills the arrays. When it signals that it is done the "Divide" button carries out the operation first as a simple *for* loop and then as a parallel *for* loop. The clock ticks for each is printed on labels.

```
{public partial class Form1 : Form
  {public Form1()
    {InitializeComponent();
    }
   public 8ons tint N = 10000000;
   private Random r = new Random();
   private double [] a1 = new double[N];
   private double [] a2 = new double[N];
   private double [] a3 = new double[N];
   private void Form1_Load(object sender, EventArgs e)
     {lblDone.Text = "";
      lblTime1.Text = "";
      lblTime2.Text = "";
     }
 private void btnLoadData_Click(object sender, EventArgs e)
    {int i;
     for(i=0;i<N;i++)
       {a1[i] = ((double)(r.Next(1, 1001)))/1000;
        a2[i] = ((double)(r.Next(1, 1001)))/1000;
       }
    lblDone.Text = "Done";
   }
 private void btnDivide_Click(object sender, EventArgs e)
   {DivideNonParallel();
    DivideParallel();
   }
 private void DivideNonParallel()
  {int i;
   long timeTicks;
   timeTicks = DateTime.Now.Ticks;
   for(i=0;i<N;i++)
     {a3[i] = Math.Sqrt(a1[i])/Math.Sqrt(a2[i]);
     }
   lblTime1.Text = (DateTime.Now.Ticks –
                       timeTicks).ToString();
  }
 private void DivideParallel()
   {long timeTicks;
    timeTicks = DateTime.Now.Ticks;
    Parallel.For(0, N, i => {a3[i] =
                Math.Sqrt(a1[i])/Math.Sqrt(a2[i]);});
    lblTime2.Text = (DateTime.Now.Ticks –
                timeTicks).ToString();
  }
 }
}
```

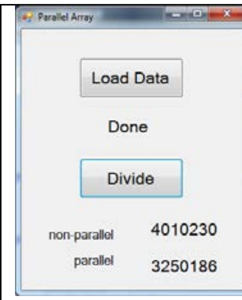

Figure 6

This program does a math operation on 10,000,000 doubles both as a simple loop and as a parallel loop and compares the required number of clock ticks.

The parallel foreach loop is similar but the parameter list is slightly more complicated.

```
{public Form1()
  {InitializeComponent();
  }
  public const int N = 10000000;
  private Random r = new Random();
  private double [] a1 = new double[N];
  private double [] a2 = new double[N];
  private double [] a3 = new double[N];
  private void Form1_Load(object sender, EventArgs e)
    {lblDone.Text = "";
     lblTime1.Text = "";
     lblTime2.Text = "";
    }
private void btnLoadData_Click(object sender, EventArgs e)
  {int i;
   for(i=0;i<N;i++)
     {a1[i] = ((double)(r.Next(1, 1001)))/1000;
     }
   lblDone.Text = "Done";
 }
 private void btnDivide_Click(object sender, EventArgs e)
   {DivideNonParallel();
    DivideParallel();
   }
 private void DivideNonParallel()
  {int i = 0;
   long timeTicks;
   timeTicks = DateTime.Now.Ticks;
   foreach(double d in a1)
    {a3[i] = Math.Sqrt(d) + Math.Sqrt(a2[i]);
     i++;
    }
    lblTime1.Text = (DateTime.Now.Ticks –
                         timeTicks).ToString();
  }
 private void DivideParallel()
  {long timeTicks;
   timeTicks = DateTime.Now.Ticks;
   Parallel.ForEach(a1, (d, state, i) =>
    {a3[i] = Math.Sqrt(d) + Math.Sqrt(a2[i]);
    });
   lblTime2.Text = (DateTime.Now.Ticks –
                       timeTicks).ToString();
  }
 }
}
```

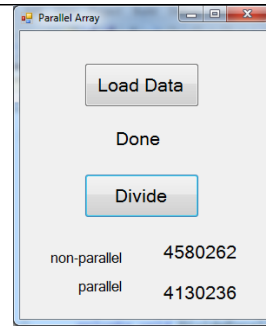

Figure 7
This example illustrates the Parallel.ForEach structure.

The Parallel.ForEach syntax is

```
Parallel.ForEach(a1, (d, state, i) => {a3[i] = Math.Sqrt(d)
                                  + Math.Sqrt(a2[i]);});
```

It can be read in words as "For each *d* in *a1* do the following calculation: $a3[i] = \sqrt{d} + \sqrt{a2[i]}$ where *i* is the index."

The variable named *state* holds the loop state and can be used to force a break in the loop if conditions are met before the loop formally ends. For this simple example the amount of time saved by completing the loop in parallel is not substantial because the parallel loop overhead is close to the time it takes to do the calculation. Also, the numbers shown in the figure are for a dual-core machine. If this same program were to run on a machine with more cores it would be able to automatically scale the threads up to take advantage of the additional resources without further user intervention.

The parallel for and foreach are not without their problems. The user must take care how loops get structured – especially in cases where data is being written by multiple elements to a common variable. Consider the problem of summing the square roots of the first million digits. A parallel for loop might look like this:

```
Parallel.For(1, 1000000, i => {total += Math.Sqrt(i);});
```

The problem is that every iteration of the loop is writing to total and if more than one iteration is done in parallel it is likely that total will be in error because the value of total will be overwritten while it is being added to by another iteration. The solution is to use a *lock* on the total variable so that it can be accessed by only one iteration at a time. This, of course, adds immensely to the loop overhead.