Threading in C-SHARP

<http://www.albahari.com/threading/>

Table of Contents

[GETTING STARTED 5](#_Toc155633256)

[Threading 5](#_Toc155633257)

[Lock 5](#_Toc155633258)

[Join 5](#_Toc155633259)

[Sleep 6](#_Toc155633260)

[Yield 6](#_Toc155633261)

[Passing Data to a Thread 6](#_Toc155633262)

[Lambda 6](#_Toc155633263)

[Function - object 6](#_Toc155633264)

[Naming Threads 6](#_Toc155633265)

[Foreground and Background Threads 7](#_Toc155633266)

[Thread Priority 7](#_Toc155633267)

[Process Priority 7](#_Toc155633268)

[QueueUserWorkItem 7](#_Toc155633269)

[Asynchronous delegates 7](#_Toc155633270)

[Optimizing the Thread Pool 8](#_Toc155633271)

[BASIC SYNCHRONIZATION 9](#_Toc155633272)

[Blocking 9](#_Toc155633273)

[Spinning 9](#_Toc155633274)

[Hybrid between blocking and spinning 9](#_Toc155633275)

[ThreadState 9](#_Toc155633276)

[Locking 10](#_Toc155633277)

[Lock 10](#_Toc155633278)

[Monitor.Enter and Monitor.Exit 11](#_Toc155633279)

[Synchronization Object 11](#_Toc155633280)

[When to Lock 11](#_Toc155633281)

[Locking and Atomicity 11](#_Toc155633282)

[Nested Locking 11](#_Toc155633283)

[Deadlocks 11](#_Toc155633284)

[Mutex 11](#_Toc155633285)

[Semaphore 12](#_Toc155633286)

[Thread Safety 13](#_Toc155633287)

[Thread Safety and .NET Framework Types 13](#_Toc155633288)

[Thread Safety in Application Servers 13](#_Toc155633289)

[UI Thread 14](#_Toc155633290)

[WPF 14](#_Toc155633291)

[WinForms 14](#_Toc155633292)

[Signaling 15](#_Toc155633293)

[Signaling with Event Wait Handles 15](#_Toc155633294)

[AutoResetEvent 15](#_Toc155633295)

[Two-way signaling 16](#_Toc155633296)

[CountdownEvent 17](#_Toc155633297)

[USING THREADS 18](#_Toc155633298)

[Async - Event-Based Asynchronous Pattern 18](#_Toc155633299)

[BackgroundWorker 18](#_Toc155633300)

[Using BackgroundWorker 18](#_Toc155633301)

[RunWorkerCompleted / WorkerReportsProgress / WorkerSupportsCancellation 18](#_Toc155633302)

[Subclassing BackgroundWorker 20](#_Toc155633303)

[Abort 20](#_Toc155633304)

[Lazy<T> 21](#_Toc155633305)

[Thread-Local Storage 21](#_Toc155633306)

[[ThreadStatic] 21](#_Toc155633307)

[ThreadLocal<T> 21](#_Toc155633308)

[ThreadLocal<T> and instance fields 21](#_Toc155633309)

[GetData and SetData 21](#_Toc155633310)

[Timers 22](#_Toc155633311)

[Multithreaded Timers 22](#_Toc155633312)

[System.Threading.Timer 22](#_Toc155633313)

[System.Timers.Timer 22](#_Toc155633314)

[Single-Threaded Timers 23](#_Toc155633315)

[ADVANCED THREADING 24](#_Toc155633316)

[Reader/Writer Locks 24](#_Toc155633317)

[Aborting Threads 25](#_Toc155633318)

[Ending Processes 25](#_Toc155633319)

[PARALLEL PROGRAMMING 26](#_Toc155633320)

[PFX Concepts 26](#_Toc155633321)

[PFX Components 26](#_Toc155633322)

[When to Use PFX 27](#_Toc155633323)

[PLINQ 27](#_Toc155633324)

[The Parallel Class 28](#_Toc155633325)

[Parallel.Invoke 28](#_Toc155633326)

[Parallel.For and Parallel.ForEach 29](#_Toc155633327)

[ParallelLoopState: Breaking early out of loops 29](#_Toc155633328)

[Task 31](#_Toc155633329)

[Task 31](#_Toc155633330)

[TaskCreationOptions 31](#_Toc155633331)

[Child tasks 31](#_Toc155633332)

[Waiting on Tasks 32](#_Toc155633333)

[Exception-Handling Tasks 32](#_Toc155633334)

[UnobservedTaskException 32](#_Toc155633335)

[Canceling Tasks 32](#_Toc155633336)

[Continuations 33](#_Toc155633337)

[Continuations and Task<TResult> 33](#_Toc155633338)

[Continuations and child tasks 33](#_Toc155633339)

[Conditional continuations 34](#_Toc155633340)

[Continuations with multiple antecedents 35](#_Toc155633341)

[Multiple continuations on a single antecedent 35](#_Toc155633342)

[Task Schedulers and UIs 35](#_Toc155633343)

[TaskFactory 36](#_Toc155633344)

[Reuse task factory 36](#_Toc155633345)

# GETTING STARTED

## Threading

using System;

using System.Threading;

class ThreadTest {

  static void Main() {

    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");

  }

  static void WriteY() {

    for (int i = 0; i < 1000; i++) Console.Write ("y");

  }

}

xxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyy

yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxx

xxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy

yyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

...

## Lock

class ThreadSafe {

  static bool done;

  static readonly object locker = new object();

  static void Main() {

    new Thread (Go).Start();

    Go();

  }

  static void Go() {

    lock (locker) {

      if (!done) { Console.WriteLine ("Done"); done = true; }

    }

  }

}

## Join

Wait for thread to finish execution

static void Main() {

  Thread t = new Thread (Go);

  t.Start();

  t.Join();

  Console.WriteLine ("Thread t has ended!");

}

static void Go() {

  for (int i = 0; i < 1000; i++) Console.Write ("y");

}

## Sleep

pauses the current thread for a specified period

Thread.Sleep (TimeSpan.FromHours (1));  // sleep for 1 hour

Thread.Sleep (500);                     // sleep for 500 milliseconds

## Yield

Same as sleep except that it relinquishes only to threads running on the same processor

## Passing Data to a Thread

### Lambda

static void Main() {

  Thread t = new Thread ( () => Print ("Hello from t!") );

  t.Start();

}

static void Print (string message) {

  Console.WriteLine (message);

}

### Function - object

static void Main() {

  Thread t = new Thread (Print);

  t.Start ("Hello from t!");

}

static void Print (object messageObj) {

  string message = (string) messageObj;   // We need to cast here

  Console.WriteLine (message);

}

## Naming Threads

Each thread has a **Name** property that you can set for the benefit of debugging. This is particularly useful in Visual Studio, since the thread’s name is displayed in the Threads Window and Debug Location toolbar. You can set a thread’s name just once; attempts to change it later will throw an exception.

class ThreadNaming {

  static void Main() {

    Thread.CurrentThread.Name = "main";

    Thread worker = new Thread (Go);

    worker.Name = "worker";

    worker.Start();

    Go();

  }

  static void Go() {

    Console.WriteLine ("Hello from " + Thread.CurrentThread.Name);

  }

}

## Foreground and Background Threads

By default, threads you create explicitly are foreground threads. 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.

## Thread Priority

enum ThreadPriority { Lowest, BelowNormal, Normal, AboveNormal, Highest }

## Process Priority

using (Process p = Process.GetCurrentProcess())

  p.PriorityClass = ProcessPriorityClass.High;

## QueueUserWorkItem

To use **QueueUserWorkItem**, simply call this method with a delegate that you want to run on a pooled thread:

static void Main() {

  ThreadPool.QueueUserWorkItem (Go);

  ThreadPool.QueueUserWorkItem (Go, 123);

  Console.ReadLine();

}

static void Go (object data) {   // data will be null with the first call.

  Console.WriteLine ("Hello from the thread pool! " + data);

}

Hello from the thread pool!

Hello from the thread pool! 123

## Asynchronous delegates

static void Main() {

  Func<string, int> method = Work;

  IAsyncResult cookie = method.BeginInvoke ("test", null, null);

  //

  // ... here's where we can do other work in parallel...

  //

  int result = method.EndInvoke (cookie);

  Console.WriteLine ("String length is: " + result);

}

static int Work (string s) { return s.Length; }

static void Main() {

  Func<string, int> method = Work;

  method.BeginInvoke ("test", Done, method);

  // ...

  //

}

static int Work (string s) { return s.Length; }

static void Done (IAsyncResult cookie) {

  var target = (Func<string, int>) cookie.AsyncState;

  int result = target.EndInvoke (cookie);

  Console.WriteLine ("String length is: " + result);

}

## Optimizing the Thread Pool

The thread pool starts out with one thread in its pool. As tasks are assigned, the pool manager “injects” new threads to cope with the extra concurrent workload, up to a maximum limit. After a sufficient period of inactivity, the pool manager may “retire” threads if it suspects that doing so will lead to better throughput.

You can set the upper limit of threads that the pool will create by calling **ThreadPool.SetMaxThreads**; the defaults are:

* 1023 in Framework 4.0 in a 32-bit environment
* 32768 in Framework 4.0 in a 64-bit environment
* 250 per core in Framework 3.5
* 25 per core in Framework 2.0

# BASIC SYNCHRONIZATION

## Blocking

A thread is deemed blocked when its execution is paused for some reason, such as when **Sleep**ing or waiting for another to end via **Join** or **EndInvoke**. A blocked thread immediately yields its processor time slice, and from then on consumes no processor time until its blocking condition is satisfied.

Unblocking happens in one of four ways (the computer's power button doesn't count!):

* by the blocking condition being satisfied
* by the operation timing out (if a timeout is specified)
* by being interrupted via [Thread.Interrupt](http://www.albahari.com/threading/part3.aspx" \l "_Interrupt)
* by being aborted via [Thread.Abort](http://www.albahari.com/threading/part3.aspx" \l "_Abort)

A thread is not deemed blocked if its execution is paused via the (deprecated) [Suspend](http://www.albahari.com/threading/part4.aspx#_Suspend_and_Resume) method.

## Spinning

Sometimes a thread must pause until a certain condition is met. [Signaling](http://www.albahari.com/threading/part2.aspx#_Signaling_with_Event_Wait_Handles) and [locking](http://www.albahari.com/threading/part2.aspx#_Locking) constructs achieve this efficiently by [blocking](http://www.albahari.com/threading/part2.aspx#_Blocking) until a condition is satisfied. However, there is a simpler alternative: a thread can await a condition by spinning in a polling loop. For example:

while (!proceed);

or:

while (DateTime.Now < nextStartTime);

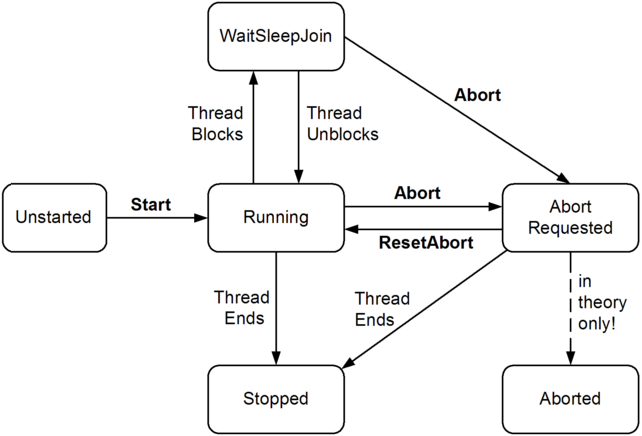
## Hybrid between blocking and spinning

while (!proceed) Thread.Sleep (10);

Spinning very briefly can be effective when you expect a condition to be satisfied soon (perhaps within a few microseconds) because it avoids the overhead and latency of a context switch. The .NET Framework provides special methods and classes to assist — these are covered [in the parallel programming section](http://www.albahari.com/threading/part5.aspx#_SpinLock_and_SpinWait).

## ThreadState

You can query a thread's execution status via its **ThreadState** property. This returns a flags enum of type **ThreadState**, which combines three “layers” of data in a bitwise fashion. Most values, however, are redundant, unused, or deprecated. The following diagram shows one “layer”:



## Locking

**A Comparison of Locking Constructs**

|  |  |  |  |
| --- | --- | --- | --- |
| Construct | Purpose | Cross-process? | Overhead\* |
| [lock](http://www.albahari.com/threading/part2.aspx#_Locking) (**Monitor.Enter** / **Monitor.Exit**) | Ensures just one thread can access a resource, or section of code at a time | - | 20ns |
| [Mutex](http://www.albahari.com/threading/part2.aspx#_Mutex) | Yes | 1000ns |
| [SemaphoreSlim](http://www.albahari.com/threading/part2.aspx#_Semaphore) (introduced in Framework 4.0) | Ensures not more than a specified number of concurrent threads can access a resource, or section of code | - | 200ns |
| [Semaphore](http://www.albahari.com/threading/part2.aspx#_Semaphore) | Yes | 1000ns |
| [ReaderWriterLockSlim](http://www.albahari.com/threading/part4.aspx#_Reader_Writer_Locks)(introduced in Framework 3.5) | Allows multiple readers to coexist with a single writer | - | 40ns |
| [ReaderWriterLock](http://www.albahari.com/threading/part4.aspx#_Reader_Writer_Locks) (effectively deprecated) | - | 100ns |

## Lock

Limited to the same thread, must be released by the same thread

class ThreadSafe {

  static readonly object \_locker = new object();

  static int \_val1, \_val2;

  static void Go() {

lock (\_locker) {

      if (\_val2 != 0) Console.WriteLine (\_val1 / \_val2);

      \_val2 = 0;

    }

  }

}

### Monitor.Enter and Monitor.Exit

C-SHARP’s **lock** statement is in fact a syntactic shortcut for a call to the methods **Monitor.Enter** and **Monitor.Exit**

bool lockTaken = false;

try {

  Monitor.Enter (\_locker, ref lockTaken);

  // Do your stuff...

}

finally { if (lockTaken) Monitor.Exit (\_locker); }

### Synchronization Object

Any object visible to each of the partaking threads can be used as a synchronizing object, subject to one hard rule: it must be a reference type. The synchronizing object is typically private (because this helps to encapsulate the locking logic) and is typically an instance or static field. The synchronizing object can double as the object it’s protecting

class ThreadSafe {

  List <string> \_list = new List <string>();

  void Test() {

    lock (\_list) {

      \_list.Add ("Item 1");

      ...

### When to Lock

As a basic rule, you need to lock around accessing any writable shared field.

### Locking and Atomicity

If a group of variables are always read and written within the same lock, you can say the variables are read and written atomically.

lock (locker) { if (x != 0) y /= x; }

### Nested Locking

The atomicity provided by a lock is violated if an exception is thrown within a **lock** block.

decimal \_savingsBalance, \_checkBalance;

void Transfer (decimal amount) {

  lock (\_locker) {

    \_savingsBalance += amount;

    \_checkBalance -= amount + GetBankFee();

  }

}

### Deadlocks

A deadlock happens when two threads each wait for a resource held by the other, so neither can proceed.

## Mutex

A **Mutex** is like a C-SHARP **lock**, but it can work across multiple processes. In other words, **Mutex** can be computer-wide as well as application-wide.

class OneAtATimePlease {

  static void Main() {

    // Naming a Mutex makes it available computer-wide. Use a name that's

    // unique to your company and application (e.g., include your URL).

    using (var mutex = new Mutex (false, "oreilly.com OneAtATimeDemo")) {

      // Wait a few seconds if contended, in case another instance

      // of the program is still in the process of shutting down.

      if (!mutex.WaitOne (TimeSpan.FromSeconds (3), false)) {

        Console.WriteLine ("Another app instance is running. Bye!");

        return;

      }

      RunProgram();

    }

  }

  static void RunProgram() {

    Console.WriteLine ("Running. Press Enter to exit");

    Console.ReadLine();

  }

}

## Semaphore

Nightclub queue lock / file system access lock

A semaphore is like a nightclub: it has a certain capacity, enforced by a bouncer. Once it’s full, no more people can enter, and a queue builds up outside. Then, for each person that leaves, one person enters from the head of the queue. The constructor requires a minimum of two arguments: the number of places currently available in the nightclub and the club’s total capacity.

class TheClub {

  static SemaphoreSlim \_sem = new SemaphoreSlim (3);    // Capacity of 3

  static void Main() {

    for (int i = 1; i <= 5; i++) new Thread (Enter).Start (i);

  }

  static void Enter (object id) {

    Console.WriteLine (id + " wants to enter");

\_sem.Wait();

    Console.WriteLine (id + " is in!");           // Only three threads

    Thread.Sleep (1000 \* (int) id);               // can be here at

    Console.WriteLine (id + " is leaving");       // a time.

\_sem.Release();

  }

}

1 wants to enter

1 is in!

2 wants to enter

2 is in!

3 wants to enter

3 is in!

4 wants to enter

5 wants to enter

1 is leaving

4 is in!

2 is leaving

5 is in!

## Thread Safety

### Thread Safety and .NET Framework Types

Locking can be used to convert thread-unsafe code into thread-safe code

class ThreadSafe {

  static List <string> \_list = new List <string>();

  static void Main() {

    new Thread (AddItem).Start();

    new Thread (AddItem).Start();

  }

  static void AddItem() {

    lock (\_list) \_list.Add ("Item " + \_list.Count);

    string[] items;

    lock (\_list) items = \_list.ToArray();

    foreach (string s in items) Console.WriteLine (s);

  }

}

Enumerating .NET collections is also thread-unsafe in the sense that an exception is thrown if the list is modified during enumeration. Rather than locking for the duration of enumeration, in this example we first copy the items to an array. This avoids holding the lock excessively if what we’re doing during enumeration is potentially time-consuming.

### Thread Safety in Application Servers

Application servers need to be multithreaded to handle simultaneous client requests. WCF, ASP.NET, and Web Services applications are implicitly multithreaded; the same holds true for Remoting server applications that use a network channel such as TCP or HTTP. This means that when writing code on the server side, you must consider thread safety if there’s any possibility of interaction among the threads processing client requests. Fortunately, such a possibility is rare; a typical server class is either stateless (no fields) or has an activation model that creates a separate object instance for each client or each request. Interaction usually arises only through static fields, sometimes used for caching in memory parts of a database to improve performance.

// User is a custom class with fields for user data

internal User RetrieveUser (int id) { ... }

If this method was called frequently, you could improve performance by caching the results in a static **Dictionary**.

static class UserCache {

  static Dictionary <int, User> \_users = new Dictionary <int, User>();

  internal static User GetUser (int id) {

    User u = null;

    lock (\_users)

      if (\_users.TryGetValue (id, out u))

        return u;

    u = RetrieveUser (id);   // Method to retrieve user from database

    lock (\_users) \_users [id] = u;

    return u;

  }

}

## UI Thread

The objects that make up a rich client are based primarily on **DependencyObject** in the case of WPF, or **Control** in the case of Windows Forms. These objects have thread affinity, which means that only the thread that instantiates them can subsequently access their members. Violating this causes either unpredictable behavior, or an exception to be thrown.

On the positive side, this means you don’t need to lock around accessing a UI object. On the negative side, if you want to call a member on object X created on another thread Y, you must marshal the request to thread Y. You can do this explicitly as follows:

* In WPF, call **Invoke** or **BeginInvoke** on the element’s **Dispatcher** object.
* In Windows Forms, call **Invoke** or **BeginInvoke** on the control.

### WPF

public partial class MyWindow : Window {

  public MyWindow() {

    InitializeComponent();

    new Thread (Work).Start();

  }

  void Work() {

    Thread.Sleep (5000);           // Simulate time-consuming task

    UpdateMessage ("The answer");

  }

  void UpdateMessage (string message) {

    Action action = () => txtMessage.Text = message;

Dispatcher.Invoke (action);

  }

}

### WinForms

  void UpdateMessage (string message) {

    Action action = () => txtMessage.Text = message;

this.Invoke (action);

  }

## Signaling

### Signaling with Event Wait Handles

Event wait handles are used for signaling. Signaling is when one thread waits until it receives notification from another. Event wait handles are the simplest of the signaling constructs, and they are unrelated to C-SHARP events. They come in three flavors: **[AutoResetEvent](http://www.albahari.com/threading/part2.aspx" \l "_AutoResetEvent)**, **[ManualResetEvent](http://www.albahari.com/threading/part2.aspx" \l "_ManualResetEvent)**, and (from Framework 4.0) **[CountdownEvent](http://www.albahari.com/threading/part2.aspx" \l "_CountdownEvent)**. The former two are based on the common **EventWaitHandle** class, where they derive all their functionality.

**A Comparison of Signaling Constructs**

|  |  |  |  |
| --- | --- | --- | --- |
| Construct | Purpose | Cross-process? | Overhead\* |
| [AutoResetEvent](http://www.albahari.com/threading/part2.aspx#_AutoResetEvent) | Allows a thread to unblock once when it receives a signal from another | Yes | 1000ns |
| [ManualResetEvent](http://www.albahari.com/threading/part2.aspx#_ManualResetEvent) | Allows a thread to unblock indefinitely when it receives a signal from another (until reset) | Yes | 1000ns |
| [ManualResetEventSlim](http://www.albahari.com/threading/part2.aspx#_ManualResetEvent)(introduced in Framework 4.0) | - | 40ns |
| [CountdownEvent](http://www.albahari.com/threading/part2.aspx#_CountdownEvent) (introduced in Framework 4.0) | Allows a thread to unblock when it receives a predetermined number of signals | - | 40ns |
| [Barrier](http://www.albahari.com/threading/part4.aspx#_The_Barrier_Class) (introduced in Framework 4.0) | Implements a thread execution barrier | - | 80ns |
| [Wait and Pulse](http://www.albahari.com/threading/part4.aspx#_Signaling_with_Wait_and_Pulse) | Allows a thread to block until a custom condition is met | - | 120ns for a **Pulse** |

### AutoResetEvent

An **AutoResetEvent** is like a ticket turnstile: inserting a ticket lets exactly one person through. The “auto” in the class’s name refers to the fact that an open turnstile automatically closes or “resets” after someone steps through. A thread waits, or [blocks](http://www.albahari.com/threading/part2.aspx#_Blocking), at the turnstile by calling **WaitOne** (wait at this “one” turnstile until it opens), and a ticket is inserted by calling the **Set** method. If a number of threads call **WaitOne**, a queue builds up behind the turnstile. (As with locks, the fairness of the queue can sometimes be violated due to nuances in the operating system). A ticket can come from any thread; in other words, any (unblocked) thread with access to the **AutoResetEvent** object can call **Set** on it to release one blocked thread.

You can create an **AutoResetEvent** in two ways. The first is via its constructor:

var auto = new AutoResetEvent (false);

(Passing **true** into the constructor is equivalent to immediately calling **Set** upon it.) The second way to create an **AutoResetEvent** is as follows:

var auto = new EventWaitHandle (false, EventResetMode.AutoReset);

In the following example, a thread is started whose job is simply to wait until signaled by another thread:

class BasicWaitHandle {

  static EventWaitHandle \_waitHandle = new AutoResetEvent (false);

  static void Main() {

    new Thread (Waiter).Start();

    Thread.Sleep (1000);                  // Pause for a second...

    \_waitHandle.Set();                    // Wake up the Waiter.

  }

  static void Waiter() {

    Console.WriteLine ("Waiting...");

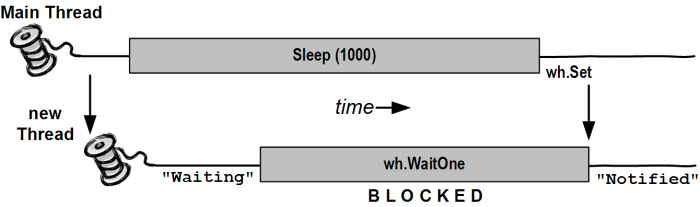
    \_waitHandle.WaitOne();                // Wait for notification

    Console.WriteLine ("Notified");

  }

}

Waiting... (pause) Notified.



### Two-way signaling

class TwoWaySignaling {

  static EventWaitHandle \_ready = new AutoResetEvent (false);

  static EventWaitHandle \_go = new AutoResetEvent (false);

  static readonly object \_locker = new object();

  static string \_message;

  static void Main() {

    new Thread (Work).Start();

    \_ready.WaitOne();                  // First wait until worker is ready

    lock (\_locker) \_message = "ooo";

    \_go.Set();                         // Tell worker to go

\_ready.WaitOne();

    lock (\_locker) \_message = "ahhh";  // Give the worker another message

\_go.Set();

\_ready.WaitOne();

    lock (\_locker) \_message = null;    // Signal the worker to exit

\_go.Set();

  }

  static void Work() {

    while (true) {

      \_ready.Set();                          // Indicate that we're ready

      \_go.WaitOne();                         // Wait to be kicked off...

      lock (\_locker) {

        if (\_message == null) return;        // Gracefully exit

        Console.WriteLine (\_message);

      }

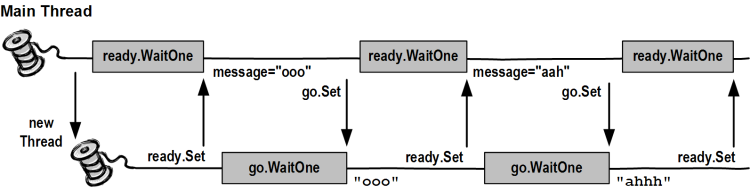
    }

  }

}

ooo

ahhh



### CountdownEvent

**CountdownEvent** lets you wait on more than one thread. The class is new to Framework 4.0 and has an efficient fully managed implementation.

var countdown = new CountdownEvent (3);  // Initialize with "count" of 3.

static CountdownEvent \_countdown = new CountdownEvent (3);

static void Main() {

  new Thread (SaySomething).Start ("I am thread 1");

  new Thread (SaySomething).Start ("I am thread 2");

  new Thread (SaySomething).Start ("I am thread 3");

\_countdown.Wait();   // Blocks until Signal has been called 3 times

  Console.WriteLine ("All threads have finished speaking!");

}

static void SaySomething (object thing) {

  Thread.Sleep (1000);

  Console.WriteLine (thing);

\_countdown.Signal();

}

# USING THREADS

## Async - Event-Based Asynchronous Pattern

Sample in WebClient class

// These members are from the WebClient class:

public byte[] DownloadData (Uri address);    // Synchronous version

public void DownloadDataAsync (Uri address);

public void DownloadDataAsync (Uri address, object userToken);

public event DownloadDataCompletedEventHandler DownloadDataCompleted;

public void CancelAsync (object userState);  // Cancels an operation

public bool IsBusy { get; }                  // Indicates if still running

Implementation

var wc = new WebClient();

wc.DownloadStringCompleted += (sender, args) => {

  if (args.Cancelled)

    Console.WriteLine ("Canceled");

  else if (args.Error != null)

    Console.WriteLine ("Exception: " + args.Error.Message);

  Else {

    Console.WriteLine (args.Result.Length + " chars were downloaded");

    // We could update the UI from here...

  }

};

wc.DownloadStringAsync (new Uri ("http://www.linqpad.net"));  // Start it

## BackgroundWorker

### Using BackgroundWorker

class Program {

  static BackgroundWorker \_bw = new BackgroundWorker();

  static void Main() {

    \_bw.DoWork += bw\_DoWork;

    \_bw.RunWorkerAsync ("Message to worker");

    Console.ReadLine();

  }

  static void bw\_DoWork (object sender, DoWorkEventArgs e) {

    // This is called on the worker thread

    Console.WriteLine (e.Argument);        // writes "Message to worker"

    // Perform time-consuming task...

  }

}

### RunWorkerCompleted / WorkerReportsProgress / WorkerSupportsCancellation

using System;

using System.Threading;

using System.ComponentModel;

class Program {

  static BackgroundWorker \_bw;

  static void Main() {

    \_bw = new BackgroundWorker {

      WorkerReportsProgress = true,

      WorkerSupportsCancellation = true

    };

    \_bw.DoWork += bw\_DoWork;

    \_bw.ProgressChanged += bw\_ProgressChanged;

    \_bw.RunWorkerCompleted += bw\_RunWorkerCompleted;

    \_bw.RunWorkerAsync ("Hello to worker");

    Console.WriteLine ("Press Enter in the next 5 seconds to cancel");

    Console.ReadLine();

    if (\_bw.IsBusy) \_bw.CancelAsync();

    Console.ReadLine();

  }

  static void bw\_DoWork (object sender, DoWorkEventArgs e) {

    for (int i = 0; i <= 100; i += 20) {

      if (\_bw.CancellationPending) { e.Cancel = true; return; }

      \_bw.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

  }

  static void bw\_RunWorkerCompleted (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

  }

  static void bw\_ProgressChanged (object sender, ProgressChangedEventArgs e) {

    Console.WriteLine ("Reached " + e.ProgressPercentage + "%");

  }

}

Press Enter in the next 5 seconds to cancel

Reached 0%

Reached 20%

Reached 40%

Reached 60%

Reached 80%

Reached 100%

Complete: 123

Press Enter in the next 5 seconds to cancel

Reached 0%

Reached 20%

Reached 40%

You canceled!

## Subclassing BackgroundWorker

public class Client {

  Dictionary <string,int> GetFinancialTotals (int foo, int bar) { ... }

  ...

}

We could refactor it as follows:

public class Client {

  public FinancialWorker GetFinancialTotalsBackground (int foo, int bar) {

    return new FinancialWorker (foo, bar);

  }

}

public class FinancialWorker : BackgroundWorker {

  public Dictionary <string,int> Result;   // You can add typed fields.

  public readonly int Foo, Bar;

  public FinancialWorker() {

    WorkerReportsProgress = true;

    WorkerSupportsCancellation = true;

  }

  public FinancialWorker (int foo, int bar) : this() {

    this.Foo = foo; this.Bar = bar;

  }

  protected override void OnDoWork (DoWorkEventArgs e) {

    ReportProgress (0, "Working hard on this report...");

    // Initialize financial report data

    // ...

    while (!<finished report>) {

      if (CancellationPending) { e.Cancel = true; return; }

      // Perform another calculation step ...

      // ...

      ReportProgress (percentCompleteCalc, "Getting there...");

    }

    ReportProgress (100, "Done!");

    e.Result = Result = <completed report data>;

  }

}

## Abort

A [blocked thread](http://www.albahari.com/threading/part2.aspx#_Blocking) can also be forcibly released via its **Abort** method. a **ThreadAbortException** is thrown.

## Lazy<T>

To use **Lazy<T>**, instantiate the class with a value factory delegate that tells it how to initialize a new value, and the argument **true**. Then access its value via the **Value** property:

Lazy<Expensive> \_expensive = new Lazy<Expensive>

  (() => new Expensive(), true);

public Expensive Expensive { get { return \_expensive.Value; } }

## Thread-Local Storage

There are three ways to implement thread-local storage.

### [ThreadStatic]

[ThreadStatic] static int \_x;

Each thread then sees a separate copy of **\_x**.

### ThreadLocal<T>

static ThreadLocal<int> \_x = new ThreadLocal<int> (() => 3);

You then use **\_x**’s **Value** property to get or set its thread-local value. A bonus of using **ThreadLocal** is that values are lazily evaluated: the factory function evaluates on the first call (for each thread).

### ThreadLocal<T> and instance fields

**ThreadLocal<T>** is also useful with instance fields and captured local variables. For example, consider the problem of generating random numbers in a multithreaded environment. The **Random** class is not thread-safe, so we have to either lock around using **Random** (limiting concurrency) or generate a separate **Random** object for each thread. **ThreadLocal<T>** makes the latter easy:

var localRandom = new ThreadLocal<Random>(() => new Random());

Console.WriteLine (localRandom.Value.Next());

Our factory function for creating the **Random** object is a bit simplistic, though, in that **Random**’s parameterless constructor relies on the system clock for a random number seed. This may be the same for two **Random** objects created within ~10 ms of each other. Here’s one way to fix it:

var localRandom = new ThreadLocal<Random>

 ( () => new Random (Guid.NewGuid().GetHashCode()) );

### GetData and SetData

The third approach is to use two methods in the **Thread** class: **GetData** and **SetData**. These store data in thread-specific “slots”. **Thread.GetData** reads from a thread’s isolated data store; **Thread.SetData** writes to it. Both methods require a **LocalDataStoreSlot** object to identify the slot. The same slot can be used across all threads and they’ll still get separate values. Here’s an example:

class Test {

  // The same LocalDataStoreSlot object can be used across all threads.

  LocalDataStoreSlot \_secSlot = Thread.GetNamedDataSlot ("securityLevel");

  // This property has a separate value on each thread.

  int SecurityLevel {

    get {

      object data = Thread.GetData (\_secSlot);

      return data == null ? 0 : (int) data;    // null == uninitialized

    }

    set { Thread.SetData (\_secSlot, value); }

  }

  ...

In this instance, we called **Thread.GetNamedDataSlot**, which creates a named slot — this allows sharing of that slot across the application. Alternatively, you can control a slot’s scope yourself with an unnamed slot, obtained by calling **Thread.AllocateDataSlot**:

class Test {

  LocalDataStoreSlot \_secSlot = Thread.AllocateDataSlot();

  ...

## Timers

The .NET Framework provides four timers. Two of these are general-purpose multithreaded timers:

* **System.Threading.Timer**
* **System.Timers.Timer**

The other two are special-purpose single-threaded timers:

* **System.Windows.Forms.Timer** (Windows Forms timer)
* **System.Windows.Threading.DispatcherTimer** (WPF timer)

The multithreaded timers are more powerful, accurate, and flexible; the single-threaded timers are safer and more convenient for running simple tasks that update Windows Forms controls or WPF elements.

## Multithreaded Timers

### System.Threading.Timer

using System;

using System.Threading;

class Program {

  static void Main() {

    // First interval = 5000ms; subsequent intervals = 1000ms

Timer tmr = new Timer (Tick, "tick...", 5000, 1000);

    Console.ReadLine();

    tmr.Dispose();         // This both stops the timer and cleans up.

  }

  static void Tick (object data) {

    // This runs on a pooled thread

    Console.WriteLine (data);          // Writes "tick..."

  }

}

### System.Timers.Timer

Wrapper of the 1st with additional elements

using System;

using System.Timers;   // Timers namespace rather than Threading

class SystemTimer {

  static void Main() {

    Timer tmr = new Timer();       // Doesn't require any args

    tmr.Interval = 500;

    tmr.Elapsed += tmr\_Elapsed;    // Uses an event instead of a delegate

    tmr.Start();                   // Start the timer

    Console.ReadLine();

    tmr.Stop();                    // Stop the timer

    Console.ReadLine();

    tmr.Start();                   // Restart the timer

    Console.ReadLine();

    tmr.Dispose();                 // Permanently stop the timer

  }

  static void tmr\_Elapsed (object sender, EventArgs e) {

    Console.WriteLine ("Tick");

  }

}

Multithreaded timers use the [thread pool](http://www.albahari.com/threading/#_Thread_Pooling) to allow a few threads to serve many timers. This means that the callback method or **Elapsed** event may fire on a different thread each time it is called. Furthermore, **Elapsed** always fires (approximately) on time — regardless of whether the previous **Elapsed** has finished executing. Hence, callbacks or event handlers must be thread-safe.

## Single-Threaded Timers

WPF and Windows Forms timers suitable for only small jobs, typically those that involve updating some aspect of the user interface (e.g., a clock or countdown display). Otherwise, you need a multithreaded timer.

# ADVANCED THREADING

## Reader/Writer Locks

Many readers and just occasional updates.

The **ReaderWriterLockSlim** class is designed to provide maximum-availability locking in just this scenario.

When compared to an ordinary [**lock**](http://www.albahari.com/threading/part2.aspx#_Locking) (**Monitor.Enter**/**Exit**), **ReaderWriterLockSlim** is twice as slow, though.

With both classes, there are two basic kinds of lock — a read lock and a write lock:

* A write lock is universally exclusive.
* A read lock is compatible with other read locks.

So, a thread holding a write lock [blocks](http://www.albahari.com/threading/part2.aspx#_Blocking) all other threads trying to obtain a read or write lock (and vice versa). But if no thread holds a write lock, any number of threads may concurrently obtain a read lock.

**ReaderWriterLockSlim** defines the following methods for obtaining and releasing read/write locks:

public void EnterReadLock();

public void ExitReadLock();

public void EnterWriteLock();

public void ExitWriteLock();

class SlimDemo {

static ReaderWriterLockSlim \_rw = new ReaderWriterLockSlim();

  static List<int> \_items = new List<int>();

  static Random \_rand = new Random();

  static void Main() {

    new Thread (Read).Start();

    new Thread (Read).Start();

    new Thread (Read).Start();

    new Thread (Write).Start ("A");

    new Thread (Write).Start ("B");

  }

  static void Read() {

    while (true) {

\_rw.EnterReadLock();

      foreach (int i in \_items) Thread.Sleep (10);

\_rw.ExitReadLock();

    }

  }

  static void Write (object threadID) {

    while (true) {

      int newNumber = GetRandNum (100);

\_rw.EnterWriteLock();

      \_items.Add (newNumber);

\_rw.ExitWriteLock();

      Console.WriteLine ("Thread " + threadID + " added " + newNumber);

      Thread.Sleep (100);

    }

  }

  static int GetRandNum (int max) { lock (\_rand) return \_rand.Next(max); }

}

Here’s the result:

Thread B added 61

Thread A added 83

Thread B added 55

Thread A added 33

...

## Aborting Threads

You can end a thread forcibly via the **Abort** method:

class Abort {

  static void Main() {

    Thread t = new Thread (delegate() { while(true); } );   // Spin forever

    t.Start();

    Thread.Sleep (1000);        // Let it run for a second...

    t.Abort();                  // then abort it.

  }

}

## Ending Processes

Another way in which a thread can end is when the parent process terminates. One example of this is when a worker thread’s **IsBackground** property is set to true, and the main thread finishes while the worker is still running. The background thread is unable to keep the application alive, and so the process terminates, taking the background thread with it.

When a thread terminates because of its parent process, it stops dead, and no finally blocks are executed.

The same situation arises when a user terminates an unresponsive application via the Windows Task Manager, or a process is killed programmatically via **Process.Kill**.

# PARALLEL PROGRAMMING

In this section, we cover the multithreading APIs new to Framework 4.0 for leveraging multicore processors:

* Parallel LINQ or [PLINQ](http://www.albahari.com/threading/part5.aspx#_PLINQ)
* The [**Parallel**](http://www.albahari.com/threading/part5.aspx#_The_Parallel_Class) class
* The [task parallelism](http://www.albahari.com/threading/part5.aspx#_Task_Parallelism) constructs
* The [concurrent collections](http://www.albahari.com/threading/part5.aspx#_Concurrent_Collections)
* [SpinLock and SpinWait](http://www.albahari.com/threading/part5.aspx#_SpinLock_and_SpinWait)

These APIs are collectively known (loosely) as PFX (Parallel Framework). The [**Parallel**](http://www.albahari.com/threading/part5.aspx#_The_Parallel_Class) class together with the [task parallelism constructs](http://www.albahari.com/threading/part5.aspx#_Task_Parallelism) is called the Task Parallel Library or TPL.

## PFX Concepts

There are two strategies for partitioning work among threads: data parallelism and task parallelism.

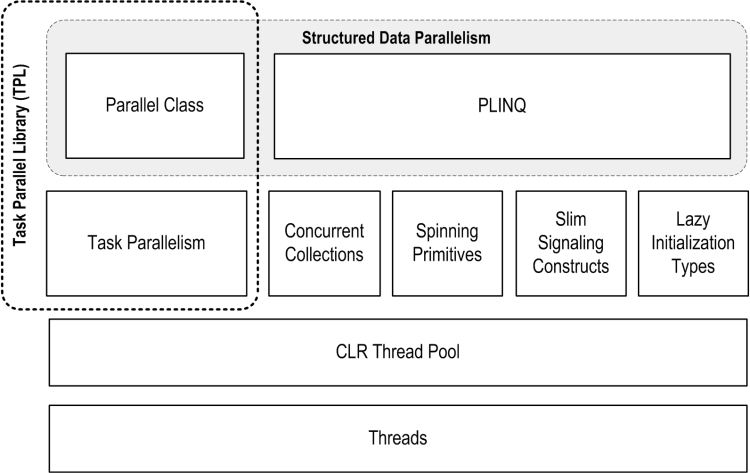
When a set of tasks must be performed on many data values, we can parallelize by having each thread perform the (same) set of tasks on a subset of values. This is called data parallelism because we are partitioning the databetween threads. In contrast, with task parallelism we partition the tasks; in other words, we have each thread perform a different task.

In general, data parallelism is easier and scales better to highly parallel hardware, because it reduces or eliminates shared data (thereby reducing contention and thread-safety issues). Also, data parallelism leverages the fact that there are often more data values than discrete tasks, increasing the parallelism potential.

Data parallelism is also conducive to structured parallelism, which means that parallel work units start and finish in the same place in your program. In contrast, task parallelism tends to be unstructured, meaning that parallel work units may start and finish in places scattered across your program. Structured parallelism is simpler and less error-prone and allows you to farm the difficult job of partitioning and thread coordination (and even result collation) out to libraries.

## PFX Components

PFX comprises two layers of functionality. The higher layer consists of two structured data parallelism APIs: [PLINQ](http://www.albahari.com/threading/part5.aspx" \l "_PLINQ)and the [**Parallel**](http://www.albahari.com/threading/part5.aspx#_The_Parallel_Class) class. The lower layer contains the task parallelism classes — plus a set of additional constructs to help with parallel programming activities.



PLINQ offers the richest functionality: it automates all the steps of parallelization — including partitioning the work into tasks, executing those tasks on threads, and collating the results into a single output sequence. It’s called declarative — because you simply declare that you want to parallelize your work (which you structure as a LINQ query), and let the Framework take care of the implementation details. In contrast, the other approaches are imperative, in that you need to explicitly write code to partition or collate. In the case of the **Parallel** class, you must collate results yourself; with the task parallelism constructs, you must partition the work yourself, too:

|  |  |  |
| --- | --- | --- |
|  | **Partitions work** | **Collates results** |
| [PLINQ](http://www.albahari.com/threading/part5.aspx#_PLINQ) | Yes | Yes |
| The [**Parallel**](http://www.albahari.com/threading/part5.aspx#_The_Parallel_Class) class | Yes | No |
| PFX’s [task parallelism](http://www.albahari.com/threading/part5.aspx#_Task_Parallelism) | No | No |

The [concurrent collections](http://www.albahari.com/threading/part5.aspx#_Concurrent_Collections) and [spinning primitives](http://www.albahari.com/threading/part5.aspx#_SpinLock_and_SpinWait) help you with lower-level parallel programming activities. These are important because PFX has been designed to work not only with today’s hardware, but also with future generations of processors with far more cores. If you want to move a pile of chopped wood and you have 32 workers to do the job, the biggest challenge is moving the wood without the workers getting in each other's way. It’s the same with dividing an algorithm among 32 cores: if ordinary locks are used to protect common resources, the resultant [blocking](http://www.albahari.com/threading/part2.aspx#_Blocking) may mean that only a fraction of those cores are ever actually busy at once. The concurrent collections are tuned specifically for highly concurrent access, with the focus on minimizing or eliminating blocking. PLINQ and the **Parallel** class themselves rely on the concurrent collections and on spinning primitives for efficient management of work.

## When to Use PFX

The primary use case for PFX is parallel programming: leveraging multicore processors to speed up computationally intensive code.

A challenge in leveraging multicores is Amdahl's law, which states that the maximum performance improvement from parallelization is governed by the portion of the code that must execute sequentially. For instance, if only two-thirds of an algorithm’s execution time is parallelizable, you can never exceed a threefold performance gain — even with an infinite number of cores.

So, before proceeding, it’s worth verifying that the bottleneck is in parallelizable code. It’s also worth considering whether your code needs to be computationally intensive — optimization is often the easiest and most effective approach. There’s a trade-off, though, in that some optimization techniques can make it harder to parallelize code.

The easiest gains come with what’s called embarrassingly parallel problems — where a job can be divided easily into tasks that execute efficiently on their own (structured parallelism is very well suited to such problems). Examples include many image processing tasks, ray tracing, and brute force approaches in mathematics or cryptography. An example of a nonembarrassingly parallel problem is implementing an optimized version of the quicksort algorithm — a good result takes some thought and may require unstructured parallelism.

## PLINQ

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.

// Calculate prime numbers using a simple (unoptimized) algorithm.

//

// NB: All code listings in this chapter are available as interactive code snippets in [LINQPad](http://www.linqpad.net).

// To activate these samples, click **Download More Samples** in LINQPad's Samples tab in the

// bottom left, and select **C-SHARP 4.0 in a Nutshell: More Chapters**.

IEnumerable<int> numbers = Enumerable.Range (3, 100000-3);

var parallelQuery =

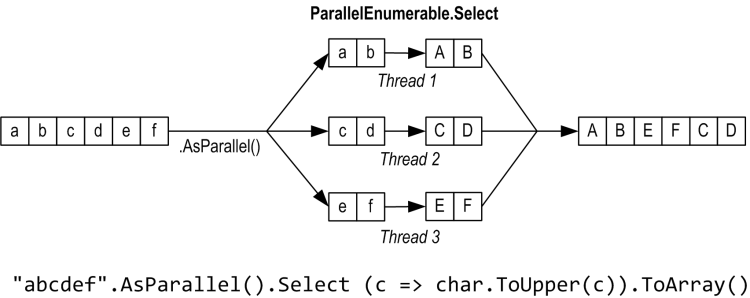
  from n in numbers.AsParallel()

  where Enumerable.Range (2, (int) Math.Sqrt (n)).All (i => n % i > 0)

  select n;

int[] primes = parallelQuery.ToArray();

**AsParallel** is an extension method in **System.Linq.ParallelEnumerable**. It wraps the input in a sequence based on **ParallelQuery<TSource>**, which causes the LINQ query operators that you subsequently call to bind to an alternate set of extension methods defined in **ParallelEnumerable**. These provide parallel implementations of each of the standard query operators. Essentially, they work by partitioning the input sequence into chunks that execute on different threads, collating the results back into a single output sequence for consumption:



## The Parallel Class

PFX provides a basic form of structured parallelism via three static methods in the **Parallel** class:

**Parallel.Invoke** Executes an array of delegates in parallel

**Parallel.For** Performs the parallel equivalent of a C-SHARP **for** loop

**Parallel.ForEach** Performs the parallel equivalent of a C-SHARP **foreach** loop

All three methods block until all work is complete. As with [PLINQ](http://www.albahari.com/threading/part5.aspx#_PLINQ), after an unhandled exception, remaining workers are stopped after their current iteration and the exception (or exceptions) are thrown back to the caller — wrapped in an **[AggregateException](http://www.albahari.com/threading/part5.aspx" \l "_Working_with_AggregateException)**.

### Parallel.Invoke

**Parallel.Invoke** executes an array of **Action** delegates in parallel, and then waits for them to complete. The simplest version of the method is defined as follows:

public static void Invoke (params Action[] actions);

Here’s how we can use **Parallel.Invoke** to download two web pages at once:

Parallel.Invoke (

 () => new WebClient().DownloadFile ("http://www.linqpad.net", "lp.html"),

 () => new WebClient().DownloadFile ("http://www.jaoo.dk", "jaoo.html"));

On the surface, this seems like a convenient shortcut for creating and waiting on two [**Task**](http://www.albahari.com/threading/part5.aspx#_Task_Parallelism) objects (or [asynchronous delegates](http://www.albahari.com/threading/#_Asynchronous_delegates)). But there’s an important difference: **Parallel.Invoke** still works efficiently if you pass in an array of a million delegates. This is because it partitions large numbers of elements into batches which it assigns to a handful of underlying **Task**s — rather than creating a separate **Task** for each delegate.

As with all of **Parallel**’s methods, you’re on your own when it comes to collating the results. This means you need to keep [thread safety](http://www.albahari.com/threading/part2.aspx#_Thread_Safety) in mind. The following, for instance, is thread-unsafe:

var data = new List<string>();

Parallel.Invoke (

 () => data.Add (new WebClient().DownloadString ("http://www.foo.com")),

 () => data.Add (new WebClient().DownloadString ("http://www.far.com")));

[Locking](http://www.albahari.com/threading/part2.aspx#_Locking) around adding to the list would resolve this, although locking would create a bottleneck if you had a much larger array of quickly executing delegates. A better solution is to use a thread-safe collection such as [ConcurrentBag](http://www.albahari.com/threading/part5.aspx" \l "_ConcurrentBagT) would be ideal in this case.

### Parallel.For and Parallel.ForEach

**Parallel.For** and **Parallel.ForEach** perform the equivalent of a C-SHARP **for** and **foreach** loop, but with each iteration executing in parallel instead of sequentially. Here are their (simplest) signatures:

public static ParallelLoopResult For (

  int fromInclusive, int toExclusive, Action<int> body)

public static ParallelLoopResult ForEach<TSource> (

  IEnumerable<TSource> source, Action<TSource> body)

The following sequential **for** loop:

for (int i = 0; i < 100; i++)

  Foo (i);

is parallelized like this:

Parallel.For (0, 100, i => Foo (i));

or more simply:

Parallel.For (0, 100, Foo);

And the following sequential **foreach**:

foreach (char c in "Hello, world")

  Foo (c);

is parallelized like this:

Parallel.ForEach ("Hello, world", Foo);

To give a practical example, if we import the **System.Security.Cryptography** namespace, we can generate six public/private key-pair strings in parallel as follows:

var keyPairs = new string[6];

Parallel.For (0, keyPairs.Length, i => keyPairs[i] = RSA.Create().ToXmlString (true));

As with **[Parallel.Invoke](http://www.albahari.com/threading/part5.aspx" \l "_Parallel.Invoke)**, we can feed **Parallel.For** and **Parallel.ForEach** a large number of work items and they’ll be efficiently partitioned onto a few tasks.

### ParallelLoopState: Breaking early out of loops

Because the loop body in a parallel **For** or **ForEach** is a delegate, you can’t exit the loop early with a **break** statement. Instead, you must call **Break** or **Stop** on a **ParallelLoopState** object:

public class ParallelLoopState {

  public void Break();

  public void Stop();

  public bool IsExceptional { get; }

  public bool IsStopped { get; }

  public long? LowestBreakIteration { get; }

  public bool ShouldExitCurrentIteration { get; }

}

Obtaining a **ParallelLoopState** is easy: all versions of **For** and **ForEach** are overloaded to accept loop bodies of type **Action<TSource,ParallelLoopState>**. So, to parallelize this:

foreach (char c in "Hello, world")

  if (c == ',')

break;

  else

    Console.Write (c);

do this:

Parallel.ForEach ("Hello, world", (c, loopState) => {

  if (c == ',')

loopState.Break();

  else

    Console.Write (c);

});

Hlloe

# Task

Task parallelism is the lowest-level approach to parallelization with PFX. The classes for working at this level are defined in the **System.Threading.Tasks** namespace and comprise the following:

|  |  |
| --- | --- |
| Class | Purpose |
| [**Task**](http://www.albahari.com/threading/part5.aspx#_Creating_and_Starting_Tasks) | For managing a unit for work |
| [**Task<TResult>**](http://www.albahari.com/threading/part5.aspx#_Creating_and_Starting_Tasks) | For managing a unit for work with a return value |
| [**TaskFactory**](http://www.albahari.com/threading/part5.aspx#_TaskFactory) | For creating tasks |
| [**TaskFactory<TResult>**](http://www.albahari.com/threading/part5.aspx#_TaskFactory) | For creating tasks and continuations with the same return type |
| [**TaskScheduler**](http://www.albahari.com/threading/part5.aspx#_Task_Schedulers_and_UIs) | For managing the scheduling of tasks |
| [TaskCompletionSource](http://www.albahari.com/threading/part5.aspx#_TaskCompletionSource) | For manually controlling a task’s workflow |

Essentially, a task is a lightweight object for managing a parallelizable unit of work. A task avoids the overhead of starting a dedicated thread by using the CLR’s [thread pool](http://www.albahari.com/threading/#_Thread_Pooling): this is the same thread pool used by **ThreadPool.QueueUserWorkItem**, tweaked in CLR 4.0 to work more efficiently with **Task**s (and more efficiently in general).

## Task

static void Main() {

  Task<string> task = Task.Factory.StartNew<string>

    ( () => DownloadString ("http://www.linqpad.net"), “TaskName” );

  RunSomeOtherMethod();

  string result = task.Result;

}

static string DownloadString (string uri) {

  using (var wc = new System.Net.WebClient())

    return wc.DownloadString (uri);

}

## TaskCreationOptions

* **LongRunning** suggests to the scheduler to dedicate a thread to the task. This is beneficial for long-running tasks because they might otherwise “hog” the queue, and force short-running tasks to wait an unreasonable amount of time before being scheduled. **LongRunning** is also good for [blocking](http://www.albahari.com/threading/part2.aspx#_Blocking) tasks.
* **PreferFairness** tells the scheduler to try to ensure that tasks are scheduled in the order they were started. It may ordinarily do otherwise, because it internally optimizes the scheduling of tasks using local work-stealing queues. This optimization is of practical benefit with very small (fine-grained) tasks.
* **AttachedToParent** is for creating child tasks.

## Child tasks

When one task starts another, you can optionally establish a parent-child relationship by specifying **TaskCreationOptions.AttachedToParent**:

Task parent = Task.Factory.StartNew (() => {

  Console.WriteLine ("I am a parent");

  Task.Factory.StartNew (() => {       // Detached task

    Console.WriteLine ("I am detached");

  });

  Task.Factory.StartNew (() => {       // Child task

    Console.WriteLine ("I am a child");

  }, TaskCreationOptions.AttachedToParent);

});

A child task is special in that when you wait for the parent task to complete, it waits for any children as well. This can be particularly useful when a child task is a [continuation](http://www.albahari.com/threading/part5.aspx#_Continuations), as we’ll see shortly.

## Waiting on Tasks

You can explicitly wait for a task to complete in two ways:

* Calling its **Wait** method (optionally with a timeout)
* Accessing its **Result** property (in the case of **Task<TResult>**)
* **Task.WaitAll** - waits for all the specified tasks to finish
* **Task.WaitAny** - waits for just one task to finish

**WaitAll** is similar to waiting out each task in turn, but is more efficient in that it requires (at most) just one context switch. Also, if one or more of the tasks throw an unhandled exception, **WaitAll** still waits out every task — and then rethrows a single **[AggregateException](http://www.albahari.com/threading/part5.aspx" \l "_Working_with_AggregateException)** that accumulates the exceptions from each faulted task. It’s equivalent to doing this:

// Assume t1, t2 and t3 are tasks:

var exceptions = new List<Exception>();

try { t1.Wait(); } catch (AggregateException ex) { exceptions.Add (ex); }

try { t2.Wait(); } catch (AggregateException ex) { exceptions.Add (ex); }

try { t3.Wait(); } catch (AggregateException ex) { exceptions.Add (ex); }

if (exceptions.Count > 0) throw new AggregateException (exceptions);

Calling **WaitAny** is equivalent to waiting on a **[ManualResetEventSlim](http://www.albahari.com/threading/part2.aspx" \l "_ManualResetEvent)** that’s signaled by each task as it finishes.

## Exception-Handling Tasks

When you wait for a task to complete (by calling its **Wait** method or accessing its **Result** property), any unhandled exceptions are conveniently rethrown to the caller, wrapped in an **[AggregateException](http://www.albahari.com/threading/part5.aspx" \l "_Working_with_AggregateException)** object. This usually avoids the need to write code within task blocks to handle unexpected exceptions; instead we can do this:

int x = 0;

Task<int> calc = Task.Factory.StartNew (() => 7 / x);

try {

  Console.WriteLine (calc.Result);

}

catch (AggregateException aex) {

  Console.Write (aex.InnerException.Message);  // Attempted to divide by 0

}

## UnobservedTaskException

The static **TaskScheduler.UnobservedTaskException** event provides a final last resort for dealing with unhandled task exceptions. By handling this event, you can intercept task exceptions that would otherwise end the application — and provide your own logic for dealing with them.

## Canceling Tasks

You can optionally pass in a [cancellation token](http://www.albahari.com/threading/part3.aspx#_Cancellation_Tokens) when starting a task. This lets you cancel tasks via the cooperative cancellation pattern [described previously](http://www.albahari.com/threading/part3.aspx#_Cancellation_Tokens):

var cancelSource = new CancellationTokenSource();

CancellationToken token = cancelSource.Token;

Task task = Task.Factory.StartNew (() => {

  // Do some stuff...

  token.ThrowIfCancellationRequested();  // Check for cancellation request

  // Do some stuff...

}, token);

...

cancelSource.Cancel();

To detect a canceled task, catch an **[AggregateException](http://www.albahari.com/threading/part5.aspx" \l "_Working_with_AggregateException)** and check the inner exception as follows:

try {

  task.Wait();

}

catch (AggregateException ex) {

  if (ex.InnerException is OperationCanceledException)

    Console.Write ("Task canceled!");

}

If the task is canceled before it has started, it won’t get scheduled — an **OperationCanceledException** will instead be thrown on the task immediately.

## Continuations

Sometimes it’s useful to start a task right after another one completes (or fails). The **ContinueWith** method on the **Task** class does exactly this:

Task task1 = Task.Factory.StartNew (() => Console.Write ("antecedant.."));

Task task2 = task1.ContinueWith (ant => Console.Write ("..continuation"));

As soon as **task1** finishes, fails, or is canceled, **task2** automatically starts. (If **task1** had completed before the second line of code ran, **task2** would be scheduled to execute right away.) The **ant** argument passed to the continuation’s lambda expression is a reference to the antecedent task.

Our example demonstrated the simplest kind of continuation, and is functionally similar to the following:

Task task = Task.Factory.StartNew (() => {

  Console.Write ("antecedent..");

  Console.Write ("..continuation");

});

The continuation-based approach, however, is more flexible in that you could first wait on **task1**, and then later wait on **task2**. This is particularly useful if **task1** returns data.

### Continuations and Task<TResult>

Task.Factory.StartNew<int> (() => 8)

  .ContinueWith (ant => ant.Result \* 2)

  .ContinueWith (ant => Math.Sqrt (ant.Result))

  .ContinueWith (ant => Console.WriteLine (ant.Result));   // 4

Our example is somewhat contrived for simplicity; in real life, these lambda expressions would call computationally intensive functions.

### Continuations and child tasks

A powerful feature of continuations is that they kick off only when all [child tasks](http://www.albahari.com/threading/part5.aspx#_Child_tasks) have completed. At that point, any exceptions thrown by the children are marshaled to the continuation.

In the following example, we start three child tasks, each throwing a **NullReferenceException**. We then catch all of them in one fell swoop via a continuation on the parent:

TaskCreationOptions atp = TaskCreationOptions.AttachedToParent;

Task.Factory.StartNew (() => {

  Task.Factory.StartNew (() => { throw null; }, atp);

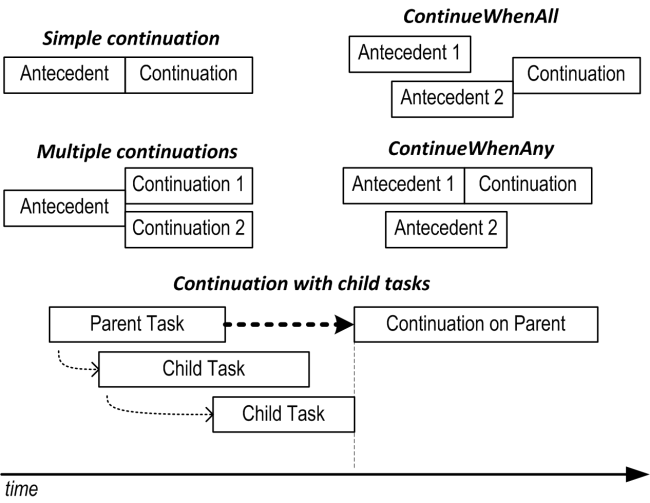
  Task.Factory.StartNew (() => { throw null; }, atp);

  Task.Factory.StartNew (() => { throw null; }, atp);

})

.ContinueWith (p => Console.WriteLine (p.Exception),

                    TaskContinuationOptions.OnlyOnFaulted);



### Conditional continuations

NotOnRanToCompletion = 0x10000,

NotOnFaulted = 0x20000,

NotOnCanceled = 0x40000,

OnlyOnRanToCompletion = NotOnFaulted | NotOnCanceled,

OnlyOnFaulted = NotOnRanToCompletion | NotOnCanceled,

OnlyOnCanceled = NotOnRanToCompletion | NotOnFaulted

It’s essential to grasp that when a continuation doesn’t execute by virtue of these flags, the continuation is not forgotten or abandoned — it’s canceled. This means that any continuations on the continuation itself will then run — unless you predicate them with **NotOnCanceled**. For example, consider this:

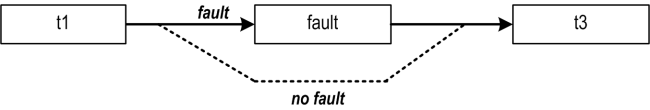
Task t1 = Task.Factory.StartNew (...);

Task fault = t1.ContinueWith (ant => Console.WriteLine ("fault"),

                              TaskContinuationOptions.OnlyOnFaulted);

Task t3 = fault.ContinueWith (ant => Console.WriteLine ("t3"));

As it stands, **t3** will always get scheduled — even if **t1** doesn’t throw an exception. This is because if **t1** succeeds, the **fault** task will be canceled, and with no continuation restrictions placed on **t3**, **t3** will then execute unconditionally.



If we want **t3** to execute only if **fault** actually runs, we must instead do this:

Task t3 = fault.ContinueWith (ant => Console.WriteLine ("t3"),

                              TaskContinuationOptions.NotOnCanceled);

### Continuations with multiple antecedents

Another useful feature of continuations is that you can schedule them to execute based on the completion of multiple antecedents. **ContinueWhenAll** schedules execution when all antecedents have completed; **ContinueWhenAny** schedules execution when one antecedent completes.

var task1 = Task.Factory.StartNew (() => Console.Write ("X"));

var task2 = Task.Factory.StartNew (() => Console.Write ("Y"));

var continuation = Task.Factory.ContinueWhenAll (

  new[] { task1, task2 }, tasks => Console.WriteLine ("Done"));

This writes “Done” after writing “XY” or “YX”. The **tasks** argument in the lambda expression gives you access to the array of completed tasks, which is useful when the antecedents return data.

The following example adds together numbers returned from two antecedent tasks:

// task1 and task2 would call complex functions in real life:

Task<int> task1 = Task.Factory.StartNew (() => 123);

Task<int> task2 = Task.Factory.StartNew (() => 456);

Task<int> task3 = Task<int>.Factory.ContinueWhenAll (

  new[] { task1, task2 }, tasks => tasks.Sum (t => t.Result));

Console.WriteLine (task3.Result);           // 579

### Multiple continuations on a single antecedent

Calling **ContinueWith** more than once on the same task creates multiple continuations on a single antecedent. When the antecedent finishes, all continuations will start together (unless you specify **TaskContinuationOptions.ExecuteSynchronously**, in which case the continuations will execute sequentially).

The following waits for one second, and then writes either “XY” or “YX”:

var t = Task.Factory.StartNew (() => Thread.Sleep (1000));

t.ContinueWith (ant => Console.Write ("X"));

t.ContinueWith (ant => Console.Write ("Y"));

## Task Schedulers and UIs

A task scheduler allocates tasks to threads. All tasks are associated with a task scheduler, which is represented by the abstract **TaskScheduler** class. The Framework provides two concrete implementations: the default scheduler that works in tandem with the [CLR thread pool](http://www.albahari.com/threading/#_Thread_Pooling), and the synchronization context scheduler. The latter is designed (primarily) to help you with the threading model of WPF and Windows Forms, which requires that UI elements and controls are accessed [only from the thread that created them](http://www.albahari.com/threading/part2.aspx#_Rich_Client_Applications). For example, suppose we wanted to fetch some data from a web service in the background, and then update a WPF label called **lblResult** with its result. We can divide this into two tasks:

1. Call a method to get data from the web service (antecedent task).
2. Update **lblResult** with the results ([continuation](http://www.albahari.com/threading/part5.aspx#_Continuations) task).

If, for a [continuation task](http://www.albahari.com/threading/part5.aspx#_Continuations), we specify the synchronization context scheduler obtained when the window was constructed, we can safely update **lblResult**:

public partial class MyWindow : Window {

  TaskScheduler \_uiScheduler;   // Declare this as a field so we can use

                                // it throughout our class.

  public MyWindow() {

    InitializeComponent();

    // Get the UI scheduler for the thread that created the form:

    \_uiScheduler = TaskScheduler.FromCurrentSynchronizationContext();

    Task.Factory.StartNew<string> (SomeComplexWebService)

      .ContinueWith (ant => lblResult.Content = ant.Result, \_uiScheduler);

  }

  string SomeComplexWebService() { ... }

}

## TaskFactory

When you call **Task.Factory**, you’re calling a static property on **Task** that returns a default **TaskFactory** object. The purpose of a task factory is to create tasks — specifically, three kinds of tasks:

* “Ordinary” tasks (via **StartNew**)
* Continuations with multiple antecedents (via **ContinueWhenAll** and **ContinueWhenAny**)
* Tasks that wrap methods that follow the asynchronous programming model (via **FromAsync**)

### Reuse task factory

var factory = new TaskFactory (

  TaskCreationOptions.LongRunning | TaskCreationOptions.AttachedToParent,

  TaskContinuationOptions.None);

Creating tasks is then simply a matter of calling **StartNew** on the factory:

Task task1 = factory.StartNew (Method1);

Task task2 = factory.StartNew (Method2);

...