

C# Threading

Hans-Wolfgang Loidl

<H.W.Loidl@hw.ac.uk>

School of Mathematical and Computer Sciences,
Heriot-Watt University, Edinburgh



Semester 1 — 2018/19

⁰Based on: "An Introduction to programming with C# Threads"

By Andrew Birrell, Microsoft, 2005

Examples from "Programming C# 5.0", Jesse Liberty, O'Reilly. Chapter 20.

Processes and Threads

- Traditionally, a process in an operating system consists of an execution environment and a single thread of execution (single activity).
- However, concurrency can be required in many programs (e.g in GUIs) for various reasons.
- The solution was to improve the notion of a process to contain an execution environment and one or more threads of execution.

Processes and Threads (cont'd)

- **An execution environment is a collection of kernel resources locally managed, which threads have access to. It consists of:**
 - ▶ **An address space.**
 - ▶ **Threads synchronization and communication resources**
 - ▶ **Higher-level resources such as file access.**

Processes and Threads (cont'd)

- Threads represent activities which can be created and destroyed dynamically as required and several of them can be running on a single execution environment.
- The aim of using multiple threads in a single environment is:
 - ▶ To maximise the concurrency of execution between operations, enabling the *overlap of computation* with input and output.
 - ▶ E.g. one thread can execute a client request while another thread serving another request (optimising server performance).

Currency and Parallelism

- In some applications concurrency is a natural way of structuring your program:
 - ▶ In GUIs separate threads handle separate events
- Concurrency is also useful operating slow devices including e.g. disks and printers.
 - ▶ IO operations are implemented as a separate thread while the program is progressing through other threads.
- Concurrency is required to exploit multi-processor machines.
 - ▶ Allowing processes to use the available processors rather than one.

Sources of Concurrency

- **Concurrency aides user interaction:**
 - ▶ Program could be processing a user request in the background and at the same time responding to user interactions by updating GUI.
- **Concurrency aides performance:**
 - ▶ A web server is multi-threaded to be able to handle multiple user requests concurrently.

Thread Primitives

- Thread Creation.
- Mutual Exclusion.
- Event waiting.
- Waking up a thread.
- The above primitives are supported by C#'s `System.Threading` namespace and C# lock statement.

Thread Creation

- **A thread is constructed in C# by:**
 - ▶ **Creating a Thread object.**
 - ▶ **Passing to it a ThreadStart delegate.**
 - ▶ **Calling the start method of the created thread.**
- **Creating and starting a thread is called forking.**

Thread Creation Example

```
1 Thread t = new Thread(new ThreadStart(func.A));  
2  
3 t.start();  
4  
5 func.B();  
6  
7 t.join();
```

Main thread executing

- The code above executes functions `func.A()` and `func.B()` concurrently.
- Initially, only the main thread is executing.
- In Line 3, Thread `t` is created and started.
- While Thread `t` is executing `func.A()`, the main thread is executing `func.B()`
- Execution completes when both method calls have completed.

Thread Creation Example

```
1 Thread t = new Thread(new ThreadStart(func.A));  
2  
3 t.start();  
4  
5 func.B();  
6  
7 t.join();
```

Main thread executing

Thread t started, executing func.A()

main exec func.B(), t exec func.A()

- The code above executes functions `func.A()` and `func.B()` concurrently.
- Initially, only the main thread is executing.
- In Line 3, Thread `t` is created and started.
- While Thread `t` is executing `func.A()`, the main thread is executing `func.B()`
- Execution completes when both method calls have completed.

Thread Creation Example

```
1 Thread t = new Thread(new ThreadStart(func.A));  
2  
3 t.start();  
4  
5 func.B();  
6  
7 t.join();
```

Main thread executing

Thread `t` started, executing `func.A()`

main exec `func.B()`, `t` exec `func.A()`

Waiting for both threads to complete

- The code above executes functions `func.A()` and `func.B()` concurrently.
- Initially, only the main thread is executing.
- In Line 3, Thread `t` is created and started.
- While Thread `t` is executing `func.A()`, the main thread is executing `func.B()`
- Execution completes when both method calls have completed.

Mutual Exclusion

- **Mutual exclusion is required to control threads access to a shared resource.**
- **We need to be able to specify a region of code that only one thread can execute at any time.**
- **Sometimes called critical section.**

Mutual Exclusion in C#

```
1 lock(expression)
2     statement
```

- Mutual exclusion is supported in C# by class Monitor and the lock statement.
- The lock argument can be any C# object.
- By default, C# objects are unlocked.
- The lock statement
 - ▶ locks the object passed as its argument,
 - ▶ executes the statements,
 - ▶ then unlocks the object.
- If another thread attempts to access the locked object, the second thread is blocked until the lock releases the object.

Example: Swap

```
1 public void Swap() {  
2     lock (this) {  
3         Console.WriteLine("Swap enter: x={0}, y={1}",  
4                             this.x, this.y);  
5         int z = this.x;  
6         this.x = this.y;  
7         this.y = z;  
8         Console.WriteLine("Swap leave: x={0}, y={1}",  
9                             this.x, this.y);  
10    }  
11 }
```

⁰Examples from "Programming C# 3.0", Jesse Liberty, O'Reilly. Chapter 20.

Example: Swap (cont'd)

```
1 public void DoTest() {  
2     Thread t1 = new Thread(new ThreadStart(Swap));  
3     Thread t2 = new Thread(new ThreadStart(Swap));  
4     t1.Start();  
5     t2.Start();  
6     t1.Join();  
7     t2.Join();  
8 }
```

Waiting for a Condition

- Locking an object is a simple scheduling policy.
- The shared memory accessed inside the lock statement is the scheduled resource.
 - ▶ More complicated scheduling is sometimes required.
 - ▶ Blocking a thread until a condition is true.
 - ▶ Supported in C# using the *Wait, Pulse and PulseAll* functions of class `Monitor`.

Waiting for a Condition (cont'd)

- A thread must hold the lock to be able to call the *Wait* function.
- The *Wait* call unlocks the object and blocks the thread.
- The *Pulse* function awakens at least one thread blocked on the locked object.
- The *PulseAll* awakens all threads currently waiting on the locked object.
- When a thread is awoken after calling *Wait* and blocking, it re-locks the object and return.

Example: Increment/Decrement

```
1 public void Decrementer() {
2     try {
3         // synchronise this area
4         Monitor.Enter(this);
5         if (counter < 1) {
6             Console.WriteLine("In Decrementer. Counter: {1}",
7                 Thread.CurrentThread.Name, counter);
8             Monitor.Wait(this);
9         }
10
11        while (counter > 0) {
12            long temp = counter;
13            temp--;
14            Thread.Sleep(1);
15            counter = temp;
16            Console.WriteLine("In Decrementer. Counter: {1}",
17                Thread.CurrentThread.Name, counter);
18        } finally {
19            Monitor.Exit(this);
20        }
21    }
22 }
```

Example: Increment/Decrement (cont'd)

```
1 public void Incrementer() {
2     try {
3         // synchronise this area
4         Monitor.Enter(this);
5
6         while (counter < 10) {
7             long temp = counter;
8             temp++;
9             Thread.Sleep(1);
10            counter = temp;
11            Console.WriteLine("In Incrementer.{1}.",
12                               Thread.CurrentThread.Name, counter);
13        }
14        Monitor.Pulse(this);
15    } finally {
16        Console.WriteLine("Exiting...",
17                           Thread.CurrentThread.Name);
18        Monitor.Exit(this);
19    }
```

Example: Increment/Decrement (cont'd)

```
1 public void DoTest() {
2     Thread[] myThreads = {
3         new Thread( new ThreadStart(Decrementer)),
4         new Thread( new ThreadStart(Incrementer)) };
5
6     int n = 1;
7     foreach (Thread myThread in myThreads) {
8         myThread.IsBackground = true;
9         myThread.Name = "Thread"+n.ToString();
10        Console.WriteLine("Starting thread {0}",
11            myThread.Name);
12        myThread.Start();
13        n++;
14        Thread.Sleep(500);
15    }
16    foreach (Thread myThread in myThreads) {
17        myThread.Join();
18    }
19    Console.WriteLine("All my threads are done");
```

Example explained

- 2 threads are created: one for incrementing another for decrementing a global counter
- A monitor is used to ensure that reading and writing of the counter is done atomically
- `Monitor.Enter/Exit` are used for entering/leaving an atomic block (critical section).
- The decrementer first checks whether the value can be decremented.
- `Monitor.Pulse` is used to inform the waiting thread of a status change.

Thread Interruption

- Interrupting a thread is sometimes required to get the thread out from a wait.
- This can be achieved in C# by using the `interrupt` function of the `Thread` class.
- A thread `t` in a wait state can be interrupted by another thread by calling `t.interrupt()`.
 - ▶ `t` will then resume execution by relocking the object (maybe after waiting for the lock to become unlocked).
- Interrupts complicate programs and should be avoided if possible.

Race Conditions

Example:

- Thread A opens a file
- Thread B writes to the file
 - ▶ \implies The program is successful, *if* A is fast enough to open the file, before B starts writing.

Deadlocks

- Thread A locks object M1
- Thread B locks object M2
- Thread A blocks trying to lock M2
- Thread B blocks trying to lock M1
- \implies None of the 2 threads can make progress

Avoiding Deadlocks

- **Maintain a partial order for acquiring locks in the program.**
- **For any pair of objects M1, M2, each thread that needs to have both objects locked simultaneously should lock the objects in the same order.**
- **E.g. M1 is always locked before M2.**
- **⇒ This avoids deadlocks caused by locks.**

Deadlocks caused by waits

Example:

- Thread A acquires resource 1
- Thread B acquires resource 2
- Thread A wants 2, so it calls `Wait` to wait for 2
- Thread B wants 1, so it calls `Wait` to wait for 1
- \implies Again, partial order can be used to avoid the deadlock.

Other Potential Problems

- ***Starvation:*** When locking objects or using `Monitor.Wait()` on an object, there is a risk that the object will never make progress.
- Program complexity.

Background Worker

- `BackgroundWorker` is a helper class in the `System.ComponentModel` namespace for managing a worker thread.
- To use it you need to
 - ▶ Instantiate `BackgroundWorker` and handle the `DoWork` event.
 - ▶ Call `RunWorkerAsync`, optionally with an object argument.
- Any argument passed to `RunWorkerAsync` will be forwarded to `DoWork`'s event handler, via the event argument's `Argument` property.
- For more info on monitoring progress, cancellation of work etc, follow the link below.

⁰See this section in “Threading in C#”, by Joe Albahari

Background Worker Example

```
1 class Program {
2     static BackgroundWorker _bw = new BackgroundWorker();
3
4     static void Main() {
5         _bw.DoWork += bw_DoWork;    // register the method
6         // to be called
7         _bw.RunWorkerAsync ("Message to worker"); // run
8         // the method asynchronously
9         Console.ReadLine();
10    }
11
12    static void bw_DoWork (object sender, DoWorkEventArgs
13        e) {
14        // This is called on the worker thread
15        Console.WriteLine (e.Argument);    // writes "
16        // Message to worker"
17        // Perform time-consuming task...
18    }
19 }
```

The `async` & `await` constructs

The `async` & `await` constructs provide language support to implement *asynchronous methods* without the need to generate threads explicitly:

- A method can have the modifier `async` to indicate that it is an asynchronous methods
- The return type of the method is then of the form `Task<TResult>`, i.e. the method returns a handle to the computation that is producing a result
- The `await` keyword is used to wait for the result that is being generated by an asynchronous method
- *While the asynchronous method waits for the result, control returns to the caller of the `async` method.*

⁰See this MSDN article on “Threading and Asynchronous Programming”

Example of `async/await`

Asynchronous file reading (main interface):

```
1 public async Task ProcessRead(string filePath) {  
2     try {  
3         string text = await ReadTextAsync(filePath);  
4         Console.WriteLine(text);  
5     } catch (Exception ex) {  
6         Console.WriteLine(ex.Message);  
7     }  
8 }
```

Example of `async/await`

Asynchronous file reading (low-level implementation):

```
1 private async Task<string> ReadTextAsync(string
    filePath) {
2     using (FileStream sourceStream =
3         new FileStream(filePath,
4             FileMode.Open, FileAccess.Read,
5                 FileShare.Read,
6                 bufferSize: 4096, useAsync: true)) {
7         StringBuilder sb = new StringBuilder();
8         byte[] buffer = new byte[0x1000];
9         int numRead;
10        while ((numRead = await sourceStream.ReadAsync(
11            buffer, 0, buffer.Length)) != 0) {
12            string text = Encoding.Unicode.GetString(
13                buffer, 0, numRead);
14            sb.Append(text);
15        }
16        return sb.ToString();
17    }
```


Example of `async/await`

A tester function, calling an asynchronous method several times:

```
1 public async Task DoIt(params string[] strs){
2     Task t;
3     List<Task> tasks = new List<Task>();
4     foreach (string str in strs) {
5         t = ProcessRead(str);
6         tasks.Add(t);
7     }
8     await Task.WhenAll(tasks);
9 }
```

⁰See Asynchronous Programming with Async and Await (C# and Visual Basic)

Resources

Sample sources and background reading:

- `threads2.cs`: incrementer/decrementer
- `threads4.cs`: incrementer/decrementer with marks
- `mulT.cs`: expanded multi-threading example
- `BgWorker.cs`: background worker example
- `asyncFiles.cs`: async example
- See this screencast on LinkedIn Learning on “Async Programming in C#”
- See this section in “Threading in C#”, by Joe Albahari
- See this MSDN article on “Threading and Asynchronous Programming”
- See Asynchronous Programming with Async and Await (C# and Visual Basic)

Summary

Technologies for *non-blocking* behaviour of your code:

- **Threads** are the most powerful mechanism, allowing for independent strands of computation
 - ▶ Independent threads also allow the usage of *parallelism* to make your program run faster (e.g. one thread per core)
 - ▶ Managing threads can be difficult and common pitfalls are deadlocks, race conditions, and starvation
- A **BackgroundWorker** task achieves asynchronous behaviour without explicitly generating threads.
 - ▶ The task will run along-side the main application.
 - ▶ When the task blocks on some operation, the caller can take over and continue with other parts of the program.
- The **async/await** constructs allow you to compose your own asynchronous methods
 - ▶ Simpler than threads or BackgroundWorker, but still single-threaded, and not suitable for parallel execution.