## **Threading**

Modern applications often need to perform multiple tasks concurrently to remain responsive or to utilize multi-core processors effectively. In C#, the primary mechanism for achieving this is through **threading**.

### **Creating and Starting a Thread**

Every client program (like Console, WPF, or Windows Forms applications) starts with a single thread, known as the "main" thread, automatically created by the operating system. To introduce concurrency, you create additional threads.

You can explicitly create a new thread by instantiating a System.Threading.Thread object and calling its Start() method. The simplest Thread constructor takes a ThreadStart delegate, which points to a parameterless method where the new thread's execution will begin.

Here's a classic example:

| using System; using System.Threading;  class Program {  static void Main()  {  Thread t = new Thread(WriteY); // Create a new thread, instructing it to run WriteY()  t.Start(); // Start the new thread's execution   // Meanwhile, the main thread continues its own work  for (int i = 0; i < 1000; i++) Console.Write("x");  }   static void WriteY()  {  for (int i = 0; i < 1000; i++) Console.Write("y");  } } |
| --- |

**Typical Output:** You'll observe an interleaved output of 'x's and 'y's (e.g., xxxxxyyyyyxxxxx...). This demonstrates the simultaneous execution. On a single-core machine, the operating system rapidly switches between threads (preemption) to simulate parallelism. On multi-core machines, these threads can truly execute in parallel.

A thread's IsAlive property indicates if it is currently running. A thread terminates when its entry delegate finishes execution and, once ended, cannot be restarted. For debugging purposes, you can assign a Name to a thread (once per thread), which is invaluable in debugging tools like Visual Studio. The static Thread.CurrentThread property allows you to get a reference to the thread currently executing your code.

### **Thread Synchronization: Join and Sleep**

Controlling the flow and interaction between threads is fundamental:

* **Join():** This method allows one thread to wait for another thread to complete its execution.

| Thread t = new Thread(Go); t.Start(); t.Join(); // Main thread waits until 't' finishes Console.WriteLine("Thread t has ended!");  void Go() { /\* ... long running work ... \*/ } |
| --- |

* Join() can also include a timeout, returning true if the thread finished or false if the timeout occurred.
* **Sleep():** This static method pauses the *current* thread for a specified duration. The thread immediately yields its processor time slice.

| Thread.Sleep(TimeSpan.FromHours(1)); // Pause for an hour Thread.Sleep(500); // Pause for 500 milliseconds |
| --- |

* Thread.Sleep(0) is a special case: it voluntarily relinquishes the current thread's time slice immediately to allow other waiting threads to run. Thread.Yield() is similar but only yields to threads running on the *same processor*. While useful for advanced performance tuning or diagnosing thread safety issues, these should be used judiciously.

### **Thread State and Blocking**

A thread is considered **blocked** when its execution is paused for reasons such as Sleep or Join. A blocked thread consumes no processor time until its blocking condition is met. You can inspect a thread's state using its ThreadState property (though it's primarily for diagnostics and not suitable for synchronization due to race conditions).

When a thread blocks or unblocks, the operating system performs a **context switch**, which incurs a small performance overhead.

### **I/O-bound vs. Compute-bound Operations**

Understanding the nature of an operation helps in choosing the right concurrency strategy:

* **I/O-bound:** An operation that spends most of its time waiting for an external event (e.g., waiting for data from a disk, network, or user input). Thread.Sleep is also considered I/O-bound in this context.
* **Compute-bound:** An operation that spends most of its time actively performing CPU-intensive calculations.

### **Blocking vs. Spinning**

When an I/O-bound operation needs to wait, it can either **block** or **spin**:

* **Blocking:** The thread pauses, consumes no CPU time, and waits synchronously (e.g., Console.ReadLine, Thread.Sleep, Thread.Join). This is generally efficient for I/O-bound tasks as the OS can schedule other work.
* **Spinning:** The thread remains active and continuously checks a condition in a loop.

| while (DateTime.Now < nextStartTime); // Continuous spinning |
| --- |

* Continuous spinning is generally **highly wasteful** of CPU time, as the OS believes the thread is doing important work and allocates resources. While very brief spinning (SpinLock, SpinWait) can be effective to avoid context switch overhead for extremely short waits, for anything longer, blocking is preferred. Blocking also has a cost (each thread consumes ~1MB of memory), which can be problematic for applications handling thousands of concurrent I/O operations. In such cases, **asynchronous callback-based approaches** are far more efficient, as they release the thread entirely while waiting.

### **Local Versus Shared State**

Each thread in C# is assigned its own memory stack. This ensures that **local variables** declared within a method are unique to each thread executing that method.

| new Thread(Go).Start(); // Call Go() on a new thread Go(); // Call Go() on the main thread  void Go() {  for (int cycles = 0; cycles < 5; cycles++) Console.Write('?'); } // Output: ?????????? (10 question marks, 5 from each thread's 'cycles' variable) |
| --- |

However, threads can **share data** if they have a common reference to the same object or variable. This often occurs with:

* **Shared Fields:** When multiple threads access the same instance field or static field.
* **Captured Variables:** When lambda expressions capture local variables from their enclosing scope, and that lambda is then executed by different threads.

| bool \_done = false; new Thread(Go).Start(); Go();  void Go() {  // Both threads share the same \_done variable  if (!\_done) { \_done = true; Console.WriteLine("Done"); } } // Output: "Done" (most likely only once) |
| --- |

This highlights a critical concept: **thread safety**. In the above example, the output is *nondeterministic*. It's possible, though less likely in the first version, for "Done" to be printed twice if both threads simultaneously evaluate if (!\_done) to true before one can set \_done = true. This type of intermittent error is notorious in multithreaded programming due to shared writable state.

### **Locking and Thread Safety**

To prevent the issues arising from shared mutable state, we implement **thread safety** using **locks**. C# provides the lock statement for this purpose:

| class ThreadSafe {  static bool \_done;  static readonly object \_locker = new object(); // A dedicated object for locking   static void Main()  {  new Thread(Go).Start();  Go();  }   static void Go()  {  lock (\_locker) // Only one thread can enter this block at a time for \_locker  {  if (!\_done) { Console.WriteLine("Done"); \_done = true; }  }  } } |
| --- |

When two threads contend for the same lock (\_locker in this case), one thread will wait (block) until the lock becomes available. This ensures that only one thread can execute the protected code block at a time, guaranteeing "Done" is printed only once. Code protected in this manner is called **thread-safe**.

Even simple operations like x++ are not thread-safe without a lock, as they involve multiple underlying CPU operations (read, increment, write) that can be interleaved by threads. While locking is powerful, it's not a panacea; it introduces complexity and risks like deadlocks if not used carefully. Often, it's better to avoid shared state altogether if possible.

### **Passing Data to a Thread**

You often need to pass arguments to a thread's startup method:

* **Lambda Expressions (Recommended):** This is the most flexible and convenient approach.

| Thread t = new Thread(() => Print("Hello from t!")); t.Start(); void Print(string message) => Console.WriteLine(message); |
| --- |

* You can pass any number of arguments and even wrap multi-statement logic within the lambda.
* **ParameterizedThreadStart Delegate:** An alternative is to pass a single object argument to the Thread.Start() method, which aligns with the ParameterizedThreadStart delegate:

| Thread t = new Thread(Print); t.Start("Hello from t!"); void Print(object messageObj) {  string message = (string)messageObj; // Requires casting  Console.WriteLine(message); } |
| --- |

**Caution with Captured Variables in Loops:**

When using lambda expressions to start threads within a loop, be extremely careful about variables captured from the loop's scope.

| for (int i = 0; i < 10; i++)  new Thread(() => Console.Write(i)).Start(); // Problematic! // Output is nondeterministic (e.g., 0223557799). // 'i' refers to the same memory location throughout the loop's lifetime. |
| --- |

The solution is to capture the loop variable into a *temporary variable* that is local to each iteration:

| for (int i = 0; i < 10; i++) {  int temp = i; // 'temp' is new for each iteration  new Thread(() => Console.Write(temp)).Start(); } // Output: 0123456789 (order still undefined, but all digits appear once) |
| --- |

This ensures each thread captures a distinct copy of the variable's value at that iteration.

### **Exception Handling in Threads**

A crucial point: try/catch/finally blocks *in effect when a thread is created are irrelevant to exceptions occurring on the new thread*. Each thread has its own independent execution path and its own exception handling stack.

| try {  new Thread(Go).Start(); // Exception in Go() will NOT be caught here } catch (Exception ex) {  Console.WriteLine("Exception!"); // Never reached } void Go() { throw null; } // Throws NullReferenceException |
| --- |

The new thread will experience an unhandled NullReferenceException, which, if not caught, will typically cause the entire application to shut down.

**Remedy:** Place exception handlers *inside* the thread's entry method (Go in this case):

| new Thread(Go).Start(); void Go() {  try  {  // ... code that might throw ...  throw null;  }  catch (Exception ex)  {  // Log the exception, display a user-friendly message, or signal other threads.  // Do NOT ignore exceptions in production code.  } } |
| --- |

Just as with your main thread, it is vital to have comprehensive exception handling on all custom worker thread entry methods in production applications to prevent application crashes and allow for robust error reporting.

### **Foreground Versus Background Threads**

Threads in C# can be either **foreground** or **background** threads:

* **Foreground Threads:** Keep the application alive as long as *any* foreground thread is running. The application will not exit until all foreground threads have completed. (Threads you create explicitly are foreground by default).
* **Background Threads:** Do *not* keep the application alive. When all foreground threads finish, the application terminates, and any still-running background threads are abruptly aborted.

You can query or change a thread's status using its IsBackground property.

| static void Main(string[] args) {  Thread worker = new Thread(() => Console.ReadLine()); // Worker waits for input  if (args.Length > 0) worker.IsBackground = true; // Make background if argument passed  worker.Start();  // Main thread exits immediately } |
| --- |

If run without arguments, the worker thread (foreground) will keep the application running. If run with an argument, worker becomes background, and the application exits as soon as Main completes, terminating ReadLine.

**Caution:** When a background thread is abruptly terminated, any finally or using blocks in its execution stack are circumvented. If cleanup (e.g., deleting temp files) is critical, you must explicitly wait for background threads to finish before application exit (e.g., using Join or signaling constructs, often with a timeout).

### **Thread Priority**

A thread's Priority property (an enum with values Lowest, BelowNormal, Normal, AboveNormal, Highest) determines its share of execution time relative to other active threads in the operating system.

Elevating a thread's priority can starve other threads, potentially impacting overall system responsiveness. For higher priority than threads in *other processes*, you also need to elevate the process priority using Process.PriorityClass in System.Diagnostics. This is useful for non-UI processes needing low latency but can significantly slow down the entire computer if misused by compute-intensive applications.

### **Signaling**

**Signaling** is the mechanism where one or more threads wait until they receive notification(s) from other threads. The simplest signaling construct is ManualResetEvent (part of a broader set of signaling constructs covered in detail in Chapter 21).

* **ManualResetEvent:** You create it in a false (nonsignaled/closed) state.
  + WaitOne(): Blocks the current thread until the event is signaled.
  + Set(): "Opens" the signal, releasing any waiting threads. The signal remains open until explicitly Reset().
  + Reset(): "Closes" the signal, causing subsequent WaitOne() calls to block.

| var signal = new ManualResetEvent(false); // Initially closed  new Thread(() => {  Console.WriteLine("Waiting for signal...");  signal.WaitOne(); // Blocks until signal.Set() is called  signal.Dispose(); // Release the event  Console.WriteLine("Got signal!"); }).Start();  Thread.Sleep(2000); // Simulate some work on the main thread signal.Set(); // Signal the waiting thread |
| --- |

### **The Thread Pool**

Creating a new thread incurs a small overhead (e.g., for setting up its memory stack). The **thread pool** mitigates this by maintaining a pool of pre-created, recyclable threads. This is essential for efficient parallel programming and fine-grained concurrency, as it avoids the overhead of repeated thread startup for short-lived operations.

Key characteristics of pooled threads:

* You **cannot set their Name** (making debugging slightly harder, though you can add descriptions in Visual Studio).
* Pooled threads are **always background threads**.
* **Blocking pooled threads can degrade performance** as it ties up a valuable pool resource.

You can determine if you are on a pooled thread using Thread.CurrentThread.IsThreadPoolThread.

**Entering the Thread Pool:**

* **Task.Run() (Recommended for modern C#):** The easiest way to explicitly run something on a pooled thread. (More on Tasks in subsequent sections).

| Task.Run(() => Console.WriteLine("Hello from the thread pool")); |
| --- |

* **ThreadPool.QueueUserWorkItem() (Legacy):**

| ThreadPool.QueueUserWorkItem(notUsed => Console.WriteLine("Hello from QueueUserWorkItem")); |
| --- |

* **Implicit Usage:** The thread pool is implicitly used by many .NET features: ASP.NET Core, System.Timers.Timer, System.Threading.Timer, parallel programming constructs, and the legacy BackgroundWorker class.

**Hygiene in the Thread Pool:**

The CLR's thread pool is sophisticated. It tries to prevent CPU oversubscription (more active threads than CPU cores, leading to expensive context switches and cache invalidation). It does this by:

* Starting with a number of concurrent tasks equal to the number of hardware cores.
* Dynamically tuning the level of concurrency using a **hill-climbing algorithm**, constantly adjusting the workload to optimize throughput.

This strategy works best when:

* Work items are **short-running** (ideally &lt; 100 ms), allowing the CLR frequent opportunities to adjust.
* **Blocking jobs do not dominate the pool**. Blocking makes the CLR falsely believe the CPU is busy. While the CLR attempts to compensate by injecting more threads, this can introduce latency and make the pool vulnerable to oversubscription later.

Maintaining "good hygiene" in the thread pool (avoiding excessive blocking and keeping work items short) is vital for maximizing CPU utilization, especially when using parallel programming APIs.