19



Introducing Multithreading

|  |
| --- |
| T |

wo significant trends of the past decade have had an enormous effect on the field of software development. First, the continued decrease in the cost of performing computations is no longer driven by increases in clock speed and transistor density, as illustrated by Figure 19.1. Rather, the cost of computation is now falling because it is economical to make hardware that has multiple CPUs.

|  |
| --- |
|  |
| Figure 19.1: Clock Speeds over Time |

\*\*\*COMP: Image taken from - <https://raw.githubusercontent.com/karlrupp/microprocessor-trend-data/master/42yrs/42-years-processor-trend.png> under creative license as documented at <https://github.com/karlrupp/microprocessor-trend-data>.

Figure is located at .\PDFs\Fig.19.01.pdf

Second, computations now routinely involve enormous latency. Latency is, simply put, the amount of time required to obtain a desired result. There are two principal causes of latency. Processor-bound latency occurs when the computational task is complex; if a computation requires performing 12 billion arithmetic operations and the total processing power available is only 6 billion operations per second, at least 2 seconds of processor-bound latency will be incurred between asking for the result and obtaining it. I/O-bound latency, by contrast, is latency incurred by the need to obtain data from an external source such as a disk drive, web server, and so on. Any computation that requires fetching data from a web server physically located far from the client machine will incur latency equivalent to millions of processor cycles.

These two trends together create an enormous challenge for modern software developers. Given that machines have more computing power than ever, how are we to make effective use of that power to deliver results to the user quickly and without compromising on the user experience? How do we avoid creating frustrating user interfaces that freeze up when a high-latency operation is triggered? Moreover, how do we go about splitting CPU-bound work among multiple processors to decrease the time required for the computation?

The standard technique for engineering software that keeps the user interface responsive and CPU utilization high is to write multithreaded programs that do multiple computations in parallel. Unfortunately, multithreading logic is notoriously difficult to get right; we spend the next four chapters exploring what makes multithreading difficult and learning how to use higher-level abstractions and new language features to ease that burden.

The first higher-level abstractions was the Parallel Extensions library that was released with .NET 4.0[[1]](#footnote-2). It includes the Task Parallel Library (TPL), discussed in this chapter, and the and Parallel LINQ (PLINQ), which is discussed in Chapter 21. The second higher level abstraction is the Task-based Asynchronous Pattern (TAP) and its accompanying language support in C# 5.0 and later.

Although I strongly encourage you to use these higher-level abstractions, I also cover some of the lower-level threading APIs from previous versions of the .NET runtime at the end of this chapter. Additional multithreading patterns prior to C# 5.0 are available for download at http://IntelliTect.com/EssentialCSharp along with the chapters from Essential C# 3.0. Thus, if you want to fully understand the resources from multithreaded programming without the later features, you still have access to that material.

I begin this chapter with a few beginner topics in case you are new to multithreading.

\*\*\*COMP: Insert 20mindmap

\*\*\*PRODUCTION: Please redraw to match mindmaps from other chapters.

|  |
| --- |
|  |

Multithreading Basics

|  |  |
| --- | --- |
|  | |
|  | Beginner Topic  Multithreading Jargon  There is a lot of confusing jargon associated with multithreading, so let’s define a few terms.  A CPU (central processing unit) or core[[2]](#footnote-3) is the unit of hardware that actually executes a given program. Every machine has at least one CPU, though today multiple CPU machines are common. Many modern CPUs support simultaneous multithreading (which Intel trademarks as Hyper-Threading), a mode whereby a single CPU can appear as multiple “virtual” CPUs.  A process is a currently executing instance of a given program; the fundamental purpose of the operating system is to manage processes. Each process contains one or more threads. A process may be accessed programmatically by an instance of the Process class in the System.Diagnostics namespace.  C# programming at the level of statements and expressions is fundamentally about describing flow of control, and thus far in this book we’ve made the implicit assumption that a given program has only a single point of control. You can imagine the point of control as being a cursor that enters the text of your program at the Main method when you start it up, and then moves around the program as the various conditions, loops, method calls, and so on, are executed. A thread is this point of control. The System.Threading namespace contains the API for manipulating a thread, specifically, the System.Threading.Thread class.  A single-threaded program is one in which there is only one thread in the process. A multithreaded program has two or more threads in the process.  A piece of code is said to be thread safe if it behaves correctly when used in a multithreaded program. The threading model of a piece of code is the set of requirements that the code places upon its caller in exchange for guaranteeing thread safety. For example, the threading model of many classes is “static methods may be called from any thread, but instance methods may be called only from the thread that allocated the instance.”  A task is a unit of potentially high-latency work that produces a resultant value or desired side effect. The distinction between tasks and threads is as follows: A task represents a job that needs to be performed, whereas a thread represents the worker that does the job. A task is useful only for its side effects and is represented by an instance of the Task class. A task used to produce a value of a given type is represented by the Task<T> class, which derives from the non-generic Task type. These can be found in the System.Threading.Tasks namespace.  A thread pool is a collection of threads, along with logic for determining how to assign work to those threads. When your program has a task to perform, it can delegate a worker thread from the pool, assign the thread to perform the task, and then de-allocate it when the work completes, thereby making it available the next time additional work is requested. |
|  | |

|  |  |
| --- | --- |
|  | |
|  | Beginner Topic  The Why and How of Multithreading  There are two principal scenarios for multithreading: enabling multitasking and dealing with latency.  Users think nothing of running dozens of processes at the same time. They might have presentations and spreadsheets open for editing while at the same time they are browsing documents on the Internet, listening to music, receiving instant messages and email arrival notifications, and watching the little clock in the corner. Each of these processes must continue to do its job even though it is not the only task the machine has to attend to. This kind of multitasking is usually implemented at the process level, but there are situations in which you want to do this sort of multitasking within a single process.  For the purposes of this book, however, we will mostly be considering multithreading as a technique for dealing with latency. For example, to import a large file while simultaneously allowing a user to click Cancel, a developer creates an additional thread to perform the import. By performing the import on a different thread, the user can request cancellation instead of freezing the user interface until the import completes.  If enough cores are available that each thread can be assigned a core, each thread essentially gets its own little machine. However, often, there are more threads than cores. Even the relatively common multicore machines of today still have only a handful of cores, while each process could quite possibly run dozens of threads.  To overcome the discrepancy between the numerous threads and the handful of cores, an operating system simulates multiple threads running concurrently by time slicing. The operating system switches execution from one thread to the next so quickly that it appears the threads are executing simultaneously. The time that the processor executes a thread before switching to another is the time slice or quantum. The act of changing which thread is executing on a core is called a context switch.  The effect is like that of a fiber-optic telephone line in which the fiber-optic line represents the processor and each conversation represents a thread. A (single-mode) fiber-optic telephone line can send only one signal at a time, but many people can hold simultaneous conversations over the line. The fiber-optic channel is fast enough to switch between conversations so quickly that each conversation appears uninterrupted. Similarly, each thread of a multithreaded process appears  If two operations are running in parallel, via either true multicore parallelism or simulated parallelism using time slicing, they are said to be concurrent. To implement such concurrency, you invoke it asynchronously, such that both the execution and the completion of the invoked operation are separate from the control flow that invoked it. Concurrency, therefore, occurs when work dispatched asynchronously executes in parallel with the current control flow. Parallel programming is the act of taking a single problem and splitting it into pieces, whereby you asynchronously initiate the process of each piece such that the pieces can all be processed concurrently. |
|  | |

|  |  |
| --- | --- |
|  | |
|  | Beginner Topic  Performance Considerations  A thread that is servicing an I/O-bound operation can essentially be ignored by the operating system until the result is available from the I/O subsystem; switching away from an I/O-bound thread to a processor-bound thread results in more efficient processor utilization because the CPU is not idle while waiting for the I/O operation to complete.  However, context switching is not free; the current internal state of the CPU must be saved to memory, and the state associated with the new thread must be loaded. Similarly, if thread A is doing lots of work with one piece of memory, and thread B is doing lots of work with another piece of memory, context switching between them will likely mean that all of the data that was loaded into the cache from thread A will get replaced with the data from thread B (or vice versa). If there are too many threads, the switching overhead can begin to noticeably affect performance. Adding more threads will likely decrease performance further, to the point where the processor spends more time switching from one thread to another than it does accomplishing the work of each thread.  Even if we ignore the cost of context switching, time slicing itself can have a huge impact on performance. Suppose, for example, that you have two processor-bound high-latency tasks, each working out the average of two lists of 1 billion numbers each. Suppose the processor can perform 1 billion operations per second. If the two tasks are each associated with a thread, and the two threads each have their own core, obviously we can get both results in 1 second.  If, however, we have a single processor that the two threads share, time slicing will perform a few hundred thousand operations on one thread, then switch to the other thread, then switch back, and so on. Each task will consume a total of 1 second of processor time, and the results of both will therefore be available after 2 seconds, leading to an average completion time of 2 seconds. (Again, we are ignoring the cost of context switching.)  If we assigned those two tasks to a single thread that performed the first task and did not even start the second until after the first was completed, the result of the first task would be obtained in 1 second and the result of the subsequent task would be obtained 1 second after that, leading to an average time of 1.5 seconds (a task completes in either 1 or 2 seconds and therefore, on average, completes in 1.5 seconds).   |  | | --- | | Guidelines  DO NOT fall into the common error of believing that more threads always make code faster.  DO carefully measure performance when attempting to speed up processor-bound problems through multithreading. | |
|  | |

\*\*\*COMP: End Beginner Topic after Guidelines

|  |  |
| --- | --- |
|  | |
|  | Beginner Topic  Threading Problems  We’ve said several times that writing multithreaded programs is complex and difficult, but we have not said why. In a nutshell, the problem is that many of our reasonable assumptions that are true of single-threaded programs are violated in multithreaded programs. The issues include a lack of atomicity, race conditions, complex memory models, and deadlocks.  Most Operations Are Not Atomic  An atomic operation is one that always is observed to be either not started or already completed. Its state is never externally visible as “in progress.” Consider, for example, this code fragment:  if (bankAccounts.Checking.Balance >= 1000.00m)  {  bankAccounts.Checking.Balance -= 1000.00m;  bankAccounts.Savings.Balance += 1000.00m;  }  This operation—checking for available funds and then conditionally debiting one account and crediting another—needs to be atomic. In other words, for it to execute correctly, we must ensure that there is never a moment when the operation can be observed to be partially completed. Imagine, for example, that two threads are running in this code concurrently. It is possible that both threads verify that there are sufficient funds in the account, and then both threads do a transfer of funds, even if there are only sufficient funds in the account to do the transfer once. And, in fact, the situation is considerably worse: There are no operations in this code fragment that are atomic! Even operations like compound addition/subtraction or reading and writing a property of decimal type are non-atomic operations in C#. As such, they can all be observed to be “partially complete” in multithreaded scenarios—only partially incremented or decremented. The observation of inconsistent state due to partially completed non-atomic operations is a special case of a more general problem, called a race condition.  Uncertainty Caused by Race Conditions  As we discussed earlier, concurrency is often simulated by time slicing. In the absence of special thread synchronization (which we discuss in detail the next chapter), the operating system can switch contexts between any two threads at any time of its choosing. As a consequence, when two threads are accessing the same object, which thread wins the race and gets to run first is unpredictable. If there are two threads running in the code fragment given previously, for example, it is possible that one thread might win the race and get all the way to the end before the second thread gets a chance to run. It is also possible that the context switch might happen after the first thread does the balance check, and the second thread might then win the race to get all the way to the end first.  The behavior of code that contains race conditions depends on the timing of context switches. This dependency introduces uncertainty concerning program execution. The order in which one instruction will execute relative to an instruction in a different thread is unknown. The worst of it is that code containing race conditions often will behave correctly 99.9 percent of the time, and then one time in a thousand a different thread wins the race due to an accident of timing. This unpredictability is what makes multithreaded programming so difficult.  Because such race conditions are difficult to replicate in the laboratory, much of the quality assurance of multithreaded code depends on long-running stress tests, specially designed code analysis tools, and a significant investment in code analysis and code review by experts. Perhaps more important than any of these is the discipline of keeping things as simple as possible. Often, in the name of hypothetical performance, a developer will try to avoid the simple approach of using a lock and go for lower-level primitives such as interlocked operations and volatiles, which makes it much more likely that their code is wrong. “Keep it simple” is possibly one of the most important guidelines of good multithreaded programming.  Chapter 20 is about techniques for dealing with race conditions.  Memory Models Are Complex  The existence of race conditions, where two points of control can “race” through a piece of code at unpredictable and inconsistent speeds, is bad enough, but it gets worse. Consider two threads that are running on two different processors but are accessing the same fields of some object. Modern processors do not actually access main memory every time you use a variable. Rather, they make a local copy in special cache memory on the processor; these caches are then periodically synchronized with main memory. This means that two threads that read from and write to the same location on two different processors can, in fact, be failing to observe each other’s updates to that memory or observing inconsistent results. Essentially what we have here is a race condition that depends on when processors choose to synchronize their caches.  Locking Leads to Deadlocks  Clearly there must exist mechanisms to make non-atomic operations into atomic operations, to instruct the operating system to schedule threads so as to avoid races, and to ensure that processor caches are synchronized when necessary. The primary mechanism used to solve all these problems in C# programs is the lock statement. This statement allows the developer to identify a section of code as “critical” code that only one thread may be in at one time; if multiple threads try to enter the critical section, the operating system will suspend[[3]](#footnote-4) all but one. The operating system also ensures that processor caches are synchronized properly upon encountering a lock.  However, locks introduce problems of their own (along with performance overhead). Most notably, if the order of lock acquisition between threads varies, a deadlock could occur such that threads freeze, each waiting for the other to release its lock.  For example, consider Figure 19.2.  \*\*\*COMP: Insert 19fig02   |  | | --- | | Macintosh HD:Users:annapopick:Desktop:Freelance:Pearson Freelance:Pearson_InProgress:9781509303588_Michaelis:Michaelis_Author:Michaelis_Word_AllEdits:Michaelis_Art:Michaelis_NumberedFigures:19fig02.pdf | | Figure 19.2: Deadlock Timeline |   At this point, each thread is waiting on the other thread before proceeding, so each thread is blocked, leading to an overall deadlock in the execution of that code.  We discuss various locking techniques in detail in Chapter 20.   |  | | --- | | Guidelines  DO NOT make an unwarranted assumption that any operation that is seemingly atomic in single threaded code will be atomic in multithreaded code.  DO NOT assume that all threads will observe all side effects of operations on shared memory in a consistent order.  DO ensure that code that concurrently acquires multiple locks always acquires them in the same order.  AVOID all race conditions—that is, conditions where program behavior depends on how the operating system chooses to schedule threads. | |
|  | |

\*\*\*COMP: End Beginner Topic after Guidelines\*\*\*COMP: Insert “Begin 4.0” tab

Asynchronous Tasks

Multithreaded programming includes the following complexities:

1. Monitoring an asynchronous operation state for completion: This includes determining when an asynchronous operation has completed, preferably not by polling the thread’s state or by blocking and waiting.

2. Thread pooling: This avoids the significant cost of starting and tearing down threads. In addition, thread pooling avoids the creation of too many threads, such that the system spends more time switching threads than running them.

3. Avoiding deadlocks: This involves preventing the occurrence of deadlocks while attempting to protect the data from simultaneous access by two different threads.

4. Providing atomicity across operations and synchronizing data access: Adding synchronization around groups of operations ensures that operations execute as a single unit and that they are appropriately interrupted by another thread. Locking is provided so that two different threads do not access the data simultaneously.

Furthermore, anytime a method is long running, multithreaded programming will probably be required—that is, invoking the long-running method asynchronously. Unfortunately, multi-threaded programming prior to C# 5.0 required using with a relatively low-level System.Threading.Thread class that had little in the way of patterns to avoid any of the these complexities.

However, as developers wrote more multithreaded code, a common set of scenarios and programming patterns for handling those scenarios emerges. And, to simplify the programming model, C# 5.0 codified the patterns, introducing a new threading type System.Threading.Tasks.Task, which greatly enhanced the programmability of one such pattern—TAP[[4]](#footnote-5)—by leveraging the TPL[[5]](#footnote-6) from .NET 4.0 and enhancing the C# language with new constructs to support it. This and the following section delve into the details of the TPL on its own and then the TPL with the async/await contextual keywords that simplify TAP programming.

Why the TPL?

Creating a thread is a relatively expensive operation, and each thread consumes a large amount (1 megabyte, by default, on Windows for example) of virtual memory. It is likely more efficient to use a thread pool to allocate threads when needed, assign asynchronous work to the thread, run the work to completion, and then reuse the thread for subsequent asynchronous work rather than destroying the thread when the work is complete and creating a new one later.

In.NET Framework 4 and later, instead of creating an operating system thread each time asynchronous work is started, the TPL creates a Task and tells the task scheduler that there is asynchronous work to perform. A task scheduler might use many different strategies to fulfill this purpose, but by default it requests a worker thread from the thread pool. The thread pool might decide that it is more efficient to run the task later, after some currently executing tasks have completed, or it might decide to schedule the task’s worker thread to a particular processor. The thread pool determines whether it is more efficient to create an entirely new thread or to reuse an existing thread that previously finished executing.

By abstracting the concept of asynchronous work into the Task object, the TPL provides an object that represents asynchronous work and provides an object-oriented API for interacting with that work. Moreover, by providing an object that represents the unit of work, the TPL enables programmatically building up workflows by composing small tasks into larger ones, as we’ll see.

A task is an object that encapsulates work that executes asynchronously. This should sound familiar: A delegate is also an object that represents code. The difference between a task and a delegate is that delegates are synchronous and tasks are asynchronous. Executing a delegate, say, an Action, immediately transfers the point of control of the current thread to the delegate’s code; control does not return to the caller until the delegate is finished. By contrast, starting a task almost immediately returns control to the caller, no matter how much work the task must perform. The task executes asynchronously, typically on another thread (though, as we will in Chapter 20, it is possible and even beneficial to execute tasks asynchronously with only one thread). A task essentially transforms a delegate from a synchronous to an asynchronous execution pattern.

Introducing Asynchronous Tasks

You know when a delegate is done executing on the current thread because the caller cannot do anything until the delegate is done. But how do you know when a task is done, and how do you get the result, if there is one? Consider the example of turning a synchronous delegate into an asynchronous task. The worker thread will write periods to the console, while the main thread writes hyphens.

Starting the task obtains a thread from the thread pool, creating a second point of control, and executes the delegate on that thread. The point of control on the main thread continues normally after the call to start the task (Task.Run()).

Listing 19.1: Invoking an Asynchronous Task

using System;

using System.Threading.Tasks;

public class Program

{

public static void Main()

{

const int repetitions = 10000;

// Use Task.Factory.StartNew<string>() for

// TPL prior to .NET 4.5

Task task = Task.Run(() =>

{

for(int count = 0;

count < repetitions; count++)

{

Console.Write('-');

}

});

for(int count = 0; count < repetitions; count++)

{

Console.Write('+');

}

// Wait until the Task completes

task.Wait();

}

}

The code that is to run in a new thread is defined in the delegate (of type Action in this case) passed to the Task.Run() method. This delegate (in the form of a lambda expression) prints out dashes to the console repeatedly. The loop that follows the starting of the task is almost identical, except that it displays plus signs.

Notice that following the call to Task.Run() the Action passed as the argument immediately starts executing. The Task is said to be “hot,” meaning that it has already been triggered to start executing—as opposed to a “cold” task, which needs to be explicitly started before the asynchronous work begins.

Although a Task can also be instantiated in a cold state via the Task constructor, doing so is generally appropriate only as an implementation detail internal to an API that returns an already running (hot) Task, one triggered by a call to Task.Start().

Notice that the exact state of a hot task is indeterminate immediately following the call to Run(). The behavior is determined by a combination of the operating system,its load, and the accompanying task library. The combination determines whether Run() chooses to execute the task’s worker thread immediately or delay it until additional resources are available. In fact, it is possible that the hot task is already finished by the time the code on the calling thread gets its turn to execute again. The call to Wait() forces the main thread to wait until all the work assigned to the task has completed executing.

In this scenario, we have a single task, but it is also possible for many tasks to be running asynchronously. It is common to have a set of tasks where you want to wait for all of them to complete, or for any one of them to complete, before continuing execution of the current thread. The Task.WaitAll() and Task.WaitAny() methods do so.

So far, we’ve seen how a task can take an Action and run it asynchronously. But what if the work executed in the task returns a result? We can use the Task<T> type to run a Func<T> asynchronously. When executing a delegate synchronously, we know that control will not return until the result is available. When executing a Task<T> asynchronously, we can poll it from one thread to see if it is done, and fetch the result when it is.[[6]](#footnote-7) Listing 19.2 demonstrates how to do so in a console application. Note that this sample uses a PiCalculator.Calculate() method that we will delve into further in the section “Executing Loop Iterations in Parallel.”

Listing 19.2: Polling a Task<T>

using System;

using System.Threading.Tasks;

public class Program

{

public static void Main()

{

// Use Task.Factory.StartNew<string>() for

// TPL prior to .NET 4.5

Task<string> task =

Task.Run<string>(

() => PiCalculator.Calculate(100));

foreach(

char busySymbol in Utility.BusySymbols())

{

if(task.IsCompleted)

{

Console.Write('\b');

break;

}

Console.Write(busySymbol);

}

Console.WriteLine();

Console.WriteLine(task.Result);

System.Diagnostics.Trace.Assert(

task.IsCompleted);

}

}

public class PiCalculator

{

public static string Calculate(int digits = 100)

{

// ...

}

}

public class Utility

{

public static IEnumerable<char> BusySymbols()

{

string busySymbols = @"-\|/-\|/";

int next = 0;

while(true)

{

yield return busySymbols[next];

next = (next + 1) % busySymbols.Length;

yield return '\b';

}

}

}

This listing shows that the data type of the task is Task<string>. The generic type includes a Result property from which to retrieve the value returned by the Func<string> that the Task<string> executes.

Note that Listing 19.2 does not make a call to Wait(). Instead, reading from the Result property automatically causes the current thread to block until the result is available, if it isn’t already; in this case we know that it will already be complete when the result is fetched.

In addition to the IsCompleted and Result properties on Task<T>, several others are worth noting:

* The IsCompleted property is set to true when a task completes, whether it completed normally or faulted (i.e., ended because it threw an exception). More detailed information on the status of a task can be obtained by reading the Status property, which returns a value of type TaskStatus. Possible values are Created, WaitingForActivation, WaitingToRun, Running, WaitingForChildrenToComplete, RanToCompletion, Canceled, and Faulted. IsCompleted is true whenever the Status is RanToCompletion, Canceled, or Faulted. Of course, if the task is running on another thread and you read the status as running, the status could change to completed at any time, including immediately after you read the value of the property. The same is true of many other states—even Created could potentially change if a different thread starts it. Only RanToCompletion, Canceled, and Faulted can be considered final states that no longer can be transitioned.

A task can be uniquely identified by the value of the Id property. The static Task.CurrentId property provides the identifier for the currently executing Task (i.e., the task that is executing the Task.CurrentId call). These properties are especially useful when debugging.

You can use the AsyncState to associate additional data with a task. For example, imagine a List<T> whose values will be computed by various tasks. Each task could contain the index of the value in the AsyncState property. This way, when the task completes, the code can index into the list using the AsyncState (first casting it to an int).[[7]](#footnote-8)

We discuss other useful properties later in this chapter under “Canceling a Task.”

Task Continuation

We’ve talked several times about the control flow of a program without ever saying what the most fundamental nature of control flow is: Control flow determines what happens next. When you have a simple control flow like Console.WriteLine(x.ToString());, the control flow tells you that when ToString completes normally, the next thing that will happen is a call to WriteLine with the value returned as the argument. The concept of “what happens next” is called continuation; each point in a control flow has a continuation. In our example, the continuation of ToString is WriteLine (and the continuation of WriteLine is whatever code runs in the next statement). The idea of continuation is so elementary to C# programming that most programmers don’t even think about it; it’s part of the invisible air that they breathe. The act of C# programming is the act of constructing continuation upon continuation until the control flow of the entire program is complete.

Notice that the continuation of a given piece of code in a normal C# program will be executed immediately upon the completion of that code. When ToString() returns, the point of control on the current thread immediately does a synchronous call to WriteLine. Notice also that there are actually two possible continuations of a given piece of code: the normal continuation and the exceptional continuation that will be executed if the current piece of code throws an exception.

Asynchronous method calls, such as starting a Task, add an additional dimension to the control flow. With an asynchronous Task invocation, the control flow goes immediately to the statement after the Task.Start(), while at the same time, it begins executing within the body of the Task delegate. In other words, what happens next when asynchrony is involved is multidimensional. Unlike with exceptions where the continuation is just a different path, with asynchrony continuation is an additional, parallel path.

Asynchronous tasks also allow composition of larger tasks out of smaller tasks by describing asynchronous continuations. Just as with regular control flow, a task can have different continuations to handle error situations, and tasks can be melded together by manipulating their continuations. There are several techniques for doing so, the most explicit of which is the ContinueWith() method (see Listing 19.3 and its corresponding output, Output 19.1).

Listing 19.3: Calling Task.ContinueWith()

using System;

using System.Threading.Tasks;

public class Program

{

public static void Main()

{

Console.WriteLine("Before");

// Use Task.Factory.StartNew<string>() for

// TPL prior to .NET 4.5

Task taskA =

Task.Run( () =>

Console.WriteLine("Starting..."))

.ContinueWith(antecedent =>

Console.WriteLine("Continuing A..."));

Task taskB = taskA.ContinueWith( antecedent =>

Console.WriteLine("Continuing B..."));

Task taskC = taskA.ContinueWith( antecedent =>

Console.WriteLine("Continuing C..."));

Task.WaitAll(taskB, taskC);

Console.WriteLine("Finished!");

}

}

|  |
| --- |
| Output 19.1 |
| Before  Starting...  Continuing A...  Continuing C...  Continuing B...  Finished! |

The ContinueWith() method enables “chaining” two tasks together, such that when the predecessor task—the antecedent task—completes, the second task—the continuation task—is automatically started asynchronously. In Listing 19.5, for example, Console.WriteLine("Starting...") is the antecedent task body and Console.WriteLine("Continuing A...") is its continuation task body. The continuation task takes a Task as its argument (antecedent), thereby allowing the continuation task’s code to access the antecedent task’s completion state. When the antecedent task is completed, the continuation task starts automatically, asynchronously executing the second delegate and passing the just-completed antecedent task as an argument to that delegate. Furthermore, since the ContinueWith() method returns a Task as well, that Task can be used as the antecedent of yet another Task, and so on, forming a continuation chain of Tasks that can be arbitrarily long.

If you call ContinueWith() twice on the same antecedent task (as Listing 19.5 shows with taskB and taskC representing continuation tasks for taskA), the antecedent task (taskA) has two continuation tasks, and when the antecedent task completes, both continuation tasks will be executed asynchronously. Notice that the order of execution of the continuation tasks from a single antecedent is indeterminate at compile time. Output 19.1 happens to show taskC executing before taskB, but in a second execution of the program, the order might be reversed. However, taskA will always execute before taskB and taskC because the latter are continuation tasks of taskA and therefore can’t start before taskA completes. Similarly, the Console.WriteLine("Starting...") delegate will always execute to completion before taskA (Console.WriteLine("Continuing A...")) because the latter is a continuation task of the former. Furthermore, Finished! will always appear last because of the call to Task.WaitAll(taskB, taskC) that blocks the control flow from continuing until both taskB and taskC complete.

Many different overloads of ContinueWith() are possible, and some of them take a TaskContinuationOptions value to tweak the behavior of the continuation chain. These values are flags, so they can be combined using the logical OR operator (|). A brief description of some of the possible flag values appears in Table 19.1; see the online documentation[[8]](#footnote-9) for more details.

|  |  |
| --- | --- |
| Table 19.1: List of Available TaskContinuationOptions Enums | |
| Enum | Description |
| None | This is the default behavior. The continuation task will be executed when the antecedent task completes, regardless of its task status. |
| PreferFairness | If two tasks were both asynchronously started, one before the other, there is no guarantee that the one that was started first actually gets to run first. This flag asks the task scheduler to try to increase the likelihood that the first task started is the first task to execute—something that is particularly relevant when the two tasks you describe are created from different thread pool threads. |
| LongRunning | This tells the task scheduler that the task is likely to be an I/O-bound high-latency task. The scheduler can then allow other queued work to be processed rather than starved because of the long-running task. This option should be used sparingly. |
| AttachedToParent | This specifies that a task should attempt to attach to a parent task within the task hierarchy. |
| DenyChildAttach (.NET 4.5) | This throws an exception if creation of a child task is attempted. If code within the continuation tries to use AttachedToParent, it will behave as if there was no parent. |
| NotOnRanToCompletion\* | This specifies that the continuation task should not be scheduled if its antecedent ran to completion. This option is not valid for multitask continuations. |
| NotOnFaulted\* | This specifies that the continuation task should not be scheduled if its antecedent threw an unhandled exception. This option is not valid for multitask continuations. |
| OnlyOnCanceled\* | This specifies that the continuation task should be scheduled only if its antecedent was canceled. This option is not valid for multitask continuations. |
| NotOnCanceled\* | This specifies that the continuation task should not be scheduled if its antecedent was canceled. This option is not valid for multitask continuations. |
| OnlyOnFaulted\* | This specifies that the continuation task should be scheduled only if its antecedent threw an unhandled exception. This option is not valid for multitask continuations. |
| OnlyOnRanToCompletion\* | This specifies that the continuation task should be scheduled only if its antecedent ran to completion. This option is not valid for multitask continuations. |
| ExecuteSynchronously | This specifies that the continuation task should be executed synchronously. With this option specified, the continuation the schedule will attempt to execute the work on is the same thread that causes the antecedent task to transition into its final state. If the antecedent is already complete when the continuation is created, the continuation will run on the thread creating the continuation. |
| HideScheduler (.NET 4.5) | This prevents the ambient scheduler from being seen as the current scheduler in the created task. This means that operations like Run/StartNew and ContinueWith that are performed in the created task will see TaskScheduler.Default (null) as the current scheduler. This is useful when continuation should run on a particular scheduler, but the continuation is calling out to additional code that should not schedule work on the same scheduler. |
| LazyCancellation (.NET 4.5) | This causes the continuation to delay monitoring the supplied cancellation token for a cancellation request until the antecedent has completed. Consider tasks t1, t2, and t3, where the latter is a continuation of the former. If t2 is canceled before t1 completes, it is possible that t3 could start before t1 completes. Setting LazyCancellation avoids this. |
| RunContinuationsAsynchronously (.NET 4.6) | When a task is created with the RunContinuationsAsynchronously option, that tells the task that it should force its continuations to run asynchronously. Even if the task is itself a continuation, this option does not affect how that task is run, only how continuations from it are run. A continuation task can be created with both TaskContinuationOptions.ExecuteSynchronously and TaskContinuationOptions.RunContinuationsAsynchronously. The former causes the continuation to execute synchronously when its antecedent completes, and causes the continuation’s continuations to run asynchronously when the continuation completes. |

In Table 19.1, the items denoted with a star (\*) indicate under which conditions the continuation task will be executed; thus they are particularly useful for creating continuations that act like event handlers for the antecedent task’s behavior. Listing 19.6 demonstrates how an antecedent task can be given multiple continuations that execute conditionally, depending on how the antecedent task completed.

Listing 19.6: Registering for Notifications of Task Behavior with ContinueWith()

using System;

using System.Threading.Tasks;

using System.Diagnostics;

using AddisonWesley.Michaelis.EssentialCSharp.Shared;

public class Program

{

public static void Main()

{

// Use Task.Factory.StartNew<string>() for

// TPL prior to .NET 4.5

Task<string> task =

Task.Run<string>(

() => PiCalculator.Calculate(10));

Task faultedTask = task.ContinueWith(

(antecedentTask) =>

{

Trace.Assert(antecedentTask.IsFaulted);

Console.WriteLine(

"Task State: Faulted");

},

TaskContinuationOptions.OnlyOnFaulted);

Task canceledTask = task.ContinueWith(

(antecedentTask) =>

{

Trace.Assert(antecedentTask.IsCanceled);

Console.WriteLine(

"Task State: Canceled");

},

TaskContinuationOptions.OnlyOnCanceled);

Task completedTask = task.ContinueWith(

(antecedentTask) =>

{

Trace.Assert(antecedentTask.IsCompleted);

Console.WriteLine(

"Task State: Completed");

}, TaskContinuationOptions.

OnlyOnRanToCompletion);

completedTask.Wait();

}

}

In this listing, we effectively register listeners for events on the antecedent’s task so that when the task completes normally or abnormally, the particular “listening” task will begin executing. This is a powerful capability, particularly if the original task is a fire-and-forget task—that is, a task that we start, hook up to continuation tasks, and then never refer to again.

In Listing 19.6, notice that the final Wait() call is on completedTask, not on task—the original antecedent task created with Task.Run(). Although each delegate’s antecedentTask is a reference to the antecedent task (task), from outside the delegate listeners we can effectively discard the reference to the original task. We can then rely solely on the continuation tasks that begin executing asynchronously without any need for follow-up code that checks the status of the original task.

In this case, we call completedTask.Wait() so that the main thread does not exit the program before the completed output appears (see Output 19.3).

|  |
| --- |
| Output 19.3 |
| Task State: Completed. |

In this case, invoking completedTask.Wait() is somewhat contrived because we know that the original task will complete successfully. However, invoking Wait() on canceledTask or faultedTask will result in an exception. Those continuation tasks run only if the antecedent task is canceled or throws an exception; given that will not happen in this program, those tasks will never be scheduled to run, and waiting for them to complete would throw an exception. The continuation options in Listing 19.3 happen to be mutually exclusive, so when the antecedent task runs to completion and the task associated with completedTask executes, the task scheduler automatically cancels the tasks associated with canceledTask and faultedTask. The canceled tasks end with their state set to Canceled. Therefore, calling Wait() (or any other invocation that would cause the current thread to wait for a task completion) on either of these tasks will throw an exception indicating that they are canceled.

A less contrived approach might be to call Task.WaitAny(completedTask, canceledTask, faultedTask), which will throw an AggregateException that then needs to be handled.

Unhandled Exception Handling on Task with AggregateException

When calling a method synchronously, we can wrap it in a try block with a catch clause to identify to the compiler which code we want to execute when an exception occurs. This does not work with an asynchronous call, however. We cannot simply wrap a try block around a call to Start() to catch an exception, because control immediately returns from the call, and control will then leave the try block, possibly long before the exception occurs on the worker thread. One solution is to wrap the body of the task delegate with a try/catch block. Exceptions thrown on and subsequently caught by the worker thread will consequently not present problems, as a try block will work normally on the worker thread. This is not the case, however, for unhandled exceptions—those that the worker thread does not catch.

Generally (starting with version 2.0[[9]](#footnote-10) of the CLR), unhandled exceptions on any thread are treated as fatal, trigger the operating system error reporting dialog, and cause the application to terminate abnormally. All exceptions on all threads must be caught, and if they are not, the application is not allowed to continue to run. (For some advanced techniques for dealing with unhandled exceptions, see the upcoming Advanced Topic titled “Dealing with Unhandled Exceptions on a Thread.”) Fortunately, this is not the case, however, for unhandled exceptions in an asynchronously running task. Rather, the task scheduler inserts a catchall exception handler around the delegate so that if the task throws an otherwise unhandled exception, the catchall handler will catch it and record the details of the exception in the task, avoiding any trigger of the CLR automatically terminating the process.

As we saw in Listing 19.6, one technique for dealing with a faulted task is to explicitly create a continuation task that is the fault handler for that task; the task scheduler will automatically schedule the continuation when it detects that the antecedent task threw an unhandled exception. If no such handler is present, however, and Wait() (or an attempt to get the Result) executes on a faulted task, an AggregateException will be thrown (see Listing 19.7 and Output 19.4).

Listing 19.5: Handling a Task’s Unhandled Exception

using System;

using System.Threading.Tasks;

public class Program

{

public static void Main()

{

// Use Task.Factory.StartNew<string>() for

// TPL prior to .NET 4.5

Task task = Task.Run(() =>

{

throw new InvalidOperationException();

});

try

{

task.Wait();

}

catch(AggregateException exception)

{

exception.Handle(eachException =>

{

Console.WriteLine(

$"ERROR: { eachException.Message }");

return true;

});

}

}

}

|  |
| --- |
| Output 19.3 |
| ERROR: Operation is not valid due to the current state of the object. |

The aggregate exception is so called because it may contain many exceptions collected from one or more faulted tasks. Imagine, for example, asynchronously executing ten tasks in parallel and five of them throwing exceptions. To report all five exceptions and have them handled in a single catch block, the framework uses the AggregateException as a means of collecting the exceptions and reporting them as a single exception. Furthermore, since it is unknown at compile time whether a worker task will throw one or more exceptions, an unhandled faulted task will always throw an AggregateException. Listing 19.5 and Output 19.3 demonstrate this behavior. Even though the unhandled exception thrown on the worker thread was of type InvalidOperationException, the type of the exception caught on the main thread is still an AggregateException. Also, as expected, to catch the exception requires an AggregateException catch block.

A list of the exceptions contained within an AggregateException is available from the InnerExceptions property. As a result, you can iterate over this property to examine each exception and determine the appropriate course of action. Alternatively, and as shown in Listing 19.5, you can use the AggregateException.Handle() method, specifying an expression to execute against each individual exception contained within the AggregateException. One important characteristic of the Handle() method to consider, however, is that it is a predicate. As such, the predicate should return true for any exceptions that the Handle() delegate successfully addresses. If any exception handling invocation returns false for an exception, the Handle() method will throw a new AggregateException that contains the composite list of such corresponding exceptions.

You can also observe the state of a faulted task without causing the exception to be rethrown on the current thread by simply looking at the Exception property of the task. Listing 19.8 demonstrates this approach by waiting for the completion of a fault continuation of a task[[10]](#footnote-11) that we know will throw an exception.

Listing 19.6: Observing Unhandled Exceptions on a Task Using ContinueWith()

using System;

using System.Diagnostics;

using System.Threading.Tasks;

public class Program

{

public static void Main()

{

bool parentTaskFaulted = false;

Task task = new Task(() =>

{

throw new InvalidOperationException();

});

Task continuationTask = task.ContinueWith(

(antecedentTask) =>

{

parentTaskFaulted =

antecedentTask.IsFaulted;

}, TaskContinuationOptions.OnlyOnFaulted);

task.Start();

continuationTask.Wait();

Trace.Assert(parentTaskFaulted);

Trace.Assert(task.IsFaulted);

task.Exception!.Handle(eachException =>

{

Console.WriteLine(

$"ERROR: { eachException.Message }");

return true;

});

}

}

Notice that to retrieve the unhandled exception on the original task, we use the Exception property (and dereferencing with the null forgiving operator because we know the value will not be null). The result is output identical to Output 19.3.

If an exception that occurs within a task goes entirely unobserved—that is, (1) it isn’t caught from within the task; (2) the completion of the task is never observed, via Wait(), Result, or accessing the Exception property, for example; and (c) the faulted ContinueWith() is never observed—then the exception is likely to go unhandled entirely, resulting in a process-wide unhandled exception. In .NET 4.0, such a faulted task would get rethrown by the finalizer thread and likely crash the process. In contrast, in .NET 4.5, the crashing has been suppressed (although the CLR can be configured for the crashing behavior if preferred).

In either case, you can register for an unhandled task exception via the TaskScheduler.UnobservedTaskException event.

|  |  |
| --- | --- |
|  | |
|  | AdVanced Topic  Dealing with Unhandled Exceptions on a Thread  As we discussed earlier, an unhandled exception on any thread by default causes the application to shut down. An unhandled exception is a fatal, unexpected bug, and the exception may have occurred because a crucial data structure is corrupt. You therefore have no idea what the program could possibly be doing, so the safest thing to do is to shut down the whole thing immediately.  Ideally, no programs would ever throw unhandled exceptions on any thread; programs that do so have bugs, and the best course of action is to find and fix the bug before the software is shipped to customers. However, rather than shutting down an application as soon as possible when an unhandled exception occurs, it is often desirable to save any working data and/or log the exception for error reporting and future debugging. This requires a mechanism to register notifications of unhandled exceptions.  With both the Microsoft .NET Framework and .NET Core 2.0 (or later), every AppDomain provides such a mechanism, and to observe the unhandled exceptions that occur in an AppDomain, you must add a handler to the UnhandledException event. The UnhandledException event will fire for all unhandled exceptions on threads within the application domain, whether it is the main thread or a worker thread. Note that the purpose of this mechanism is notification; it does not permit the application to recover from the unhandled exception and continue executing. After the event handlers run, the application will display the operating system's error reporting dialog, and then the application will exit. (For console applications, the exception details will also appear on the console.)  In Listing 19.7, we show how to create a second thread that throws an exception, which is then handled by the application domain’s unhandled exception event handler. For demonstration purposes, to ensure that thread timing issues do not come into play, we insert some artificial delays using Thread.Sleep. Output 19.4 shows the results.  Listing 19.7: Registering for Unhandled Exceptions  using System;  using System.Diagnostics;  using System.Threading;  public static class Program  {  public static Stopwatch \_Clock = new Stopwatch();  public static void Main()  {  try  {  \_Clock.Start();  // Register a callback to receive notifications  // of any unhandled exception  AppDomain.CurrentDomain.UnhandledException +=  (s, e) =>  {  Message("Event handler starting");  Delay(4000);  };  Thread thread = new Thread(() =>  {  Message("Throwing exception.");  throw new Exception();  });  thread.Start();  Delay(2000);  }  finally  {  Message("Finally block running.");  }  }  static void Delay(int i)  {  Message($"Sleeping for {i} ms");  Thread.Sleep(i);  Message("Awake");  }  static void Message(string text)  {  Console.WriteLine("{0}:{1:0000}:{2}",  Thread.CurrentThread.ManagedThreadId,  \_Clock.ElapsedMilliseconds, text);  }  }   |  | | --- | | Output 19.4 | | 3:0047:Throwing exception.  3:0052:Unhandled exception handler starting.  3:0055:Sleeping for 4000 ms  1:0058:Sleeping for 2000 ms  1:2059:Awake  1:2060:Finally block running.  3:4059:Awake  Unhandled Exception: System.Exception: Exception of type 'System.  Exception' was thrown. |   As you can see in Output 19.4, the new thread is assigned thread ID 3 and the main thread is assigned thread ID 1. The operating system schedules thread 3 to run for a while; it throws an unhandled exception, the event handler is invoked, and it goes to sleep. Soon thereafter, the operating system realizes that thread 1 can be scheduled, but its code immediately puts it to sleep. Thread 1 wakes up first and runs the finally block, and then 2 seconds later thread 3 wakes up, and the unhandled exception finally crashes the process.  This sequence of events—the event handler executing, and the process crashing after it is finished—is typical but not guaranteed. The moment there is an unhandled exception in your program, all bets are off; the program is now in an unknown and potentially very unstable state, so its behavior can be unpredictable. In this case, as you can see, the CLR allows the main thread to continue running and executing its finally block, even though it knows that by the time control gets to the finally block, another thread is in the AppDomain’s unhandled exception event handler.  To emphasize this fact, try changing the delays so that the main thread sleeps longer than the event handler. In that scenario, the finally block will never execute! The process will be destroyed by the unhandled exception before thread 1 wakes up. You can also get different results depending on whether the exception-throwing thread is or is not created by the thread pool. The best practice, therefore, is to avoid all possible unhandled exceptions, whether they occur in worker threads or in the main thread.  How does this pertain to tasks? What if there are unfinished tasks hanging around the system when you want to shut it down? We look at task cancellation in the next section.   |  | | --- | | Guidelines  AVOID writing programs that produce unhandled exceptions on any thread.  CONSIDER registering an unhandled exception event handler for debugging, logging, and emergency shutdown purposes.  DO cancel unfinished tasks rather than allowing them to run during application shutdown. | |
|  | |

\*\*\*COMP: End Advanced Topic after Guidelines

Canceling a Task

Earlier in this chapter, we described why it’s a bad idea to rudely abort a thread so as to cancel a task being performed by that thread. The TPL uses cooperative cancellation, a far more polite, robust, and reliable technique for safely canceling a task that is no longer needed. A task that supports cancellation monitors a CancellationToken object (found in the System.Threading namespace) by periodically polling it to see if a cancellation request has been issued. Listing 19.8 demonstrates both the cancellation request and the response to the request. Output 19.5 shows the results.

Listing 19.8: Canceling a Task Using CancellationToken

using System;

using System.Threading;

using System.Threading.Tasks;

using AddisonWesley.Michaelis.EssentialCSharp.Shared;

public class Program

{

public static void Main()

{

string stars =

"\*".PadRight(Console.WindowWidth-1, '\*');

Console.WriteLine("Push ENTER to exit.");

CancellationTokenSource cancellationTokenSource=

new CancellationTokenSource();

// Use Task.Factory.StartNew<string>() for

// TPL prior to .NET 4.5

Task task = Task.Run(

() =>

WritePi(cancellationTokenSource.Token),

cancellationTokenSource.Token);

// Wait for the user's input

Console.ReadLine();

cancellationTokenSource.Cancel();

Console.WriteLine(stars);

task.Wait();

Console.WriteLine();

}

private static void WritePi(

CancellationToken cancellationToken)

{

const int batchSize = 1;

string piSection = string.Empty;

int i = 0;

while(!cancellationToken.IsCancellationRequested

|| i == int.MaxValue)

{

piSection = PiCalculator.Calculate(

batchSize, (i++) \* batchSize);

Console.Write(piSection);

}

}

}

|  |
| --- |
| Output 19.5 |
| Push ENTER to exit.  3.141592653589793238462643383279502884197169399375105820974944592307816  40628620899862803482534211706798214808651328230664709384460955058223172  5359408128481117450  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  2 |

After starting the task, a Console.Read() blocks the main thread. At the same time, the task continues to execute, calculating the next digit of pi and printing it out. Once the user presses Enter, the execution encounters a call to CancellationTokenSource.Cancel(). In Listing 19.8, we split the call to task.Cancel() from the call to task.Wait() and print out a line of asterisks in between. The purpose of this step is to show that quite possibly an additional iteration will occur before the cancellation token is observed—hence the additional 2 in Output 19.5 following the stars. The 2 appears because the CancellationTokenSource.Cancel() doesn’t rudely stop the task from executing. The task keeps on running until it checks the token, and politely shuts down when it sees that the owner of the token is requesting cancellation of the task.

The Cancel() call effectively sets the IsCancellationRequested property on all cancellation tokens copied from CancellationTokenSource.Token. There are a few things to note, however:

* A CancellationToken, not a CancellationTokenSource, is given to the asynchronous task. A CancellationToken enables polling for a cancellation request; the CancellationTokenSource provides the token and signals it when it is canceled (see Figure 19.3). By passing the CancellationToken rather than the CancellationTokenSource, we don’t have to worry about thread synchronization issues on the CancellationTokenSource because the latter remains accessible to only the original thread.

A CancellationToken is a struct, so it is copied by value. The value returned by CancellationTokenSource.Token produces a copy of the token. The fact that CancellationToken is a value type and a copy is created results in thread safe access to CancellationTokenSource.Token—it is available only from within the WritePi() method.

\*\*\*COMP: Insert 19fig03

|  |
| --- |
| Macintosh HD:Users:annapopick:Desktop:Freelance:Pearson Freelance:Pearson_InProgress:9781509303588_Michaelis:Michaelis_Author:Michaelis_Word_AllEdits:Michaelis_Art:Michaelis_NumberedFigures:19fig03.pdf |
| Figure 19.3: CancellationTokenSource and CancellationToken Class Diagrams |

To monitor the IsCancellationRequested property, a copy of the CancellationToken (retrieved from CancellationTokenSource.Token) is passed to the task. In Listing 19.9, we then occasionally check the IsCancellationRequested property on the CancellationToken parameter; in this case, we check after each digit calculation. If IsCancellationRequested returns true, the while loop exits. Unlike a thread abort, which would throw an exception at essentially a random point, we exit the loop using normal control flow. We guarantee that the code is responsive to cancellation requests by polling frequently.

One other point to note about the CancellationToken is the overloaded Register() method. Via this method, you can register an action that will be invoked whenever the token is canceled. In other words, calling the Register() method subscribes to a listener delegate on the corresponding CancellationTokenSource’s Cancel().

Given that canceling before completing is the expected behavior in this program, the code in Listing 19.9 does not throw a System.Threading.Tasks.TaskCanceledException. As a consequence, task.Status will return TaskStatus.RanToCompletion—providing no indication that the work of the task was, in fact, canceled. In this example, there is no need for such an indication; however, the TPL does include the capability to do this. If the cancel call were disruptive in some way—preventing a valid result from returning, for example—throwing a TaskCanceledException (which derives from System.OperationCanceledException) would be the TPL pattern for reporting it. Instead of throwing the exception explicitly, CancellationToken includes a ThrowIfCancellationRequested() method to report the exception more easily, assuming an instance of CancellationToken is available.

If you attempt to call Wait() (or obtain the Result) on a task that threw TaskCanceledException, the behavior is the same as if any other exception had been thrown in the task: The call will throw an AggregateException. The exception is a means of communicating that the state of execution following the task is potentially incomplete. Unlike a successfully completed task in which all expected work executed successfully, a canceled task potentially has partially completed work—the state of the work is untrusted.

This example demonstrates how a long-running processor-bound operation (calculating pi almost indefinitely) can monitor for a cancellation request and respond if one occurs. There are some cases, however, when cancellation can occur without explicitly coding for it within the target task. For example, the Parallel class discussed Chapter 21, offers such a behavior by default.

|  |
| --- |
| Guidelines  DO cancel unfinished tasks rather than allowing them to run during application shutdown. |

\*\*\*COMP: Insert “Begin 5.0” tab

Task.Run(): A Shortcut and Simplification to Task.Factory.StartNew()

In .NET 4.0, the general practice for obtaining a task was to call Task.Factory.StartNew(). In .NET 4.5, a simpler calling structure was provided in Task.Run(). Like Task.Run(), Task.Factory.StartNew() could be used in C# 4.0 scenarios to invoke CPU-intensive methods that require an additional thread to be created.

Given .NET 4.5, Task.Run() should be used by default unless it proves insufficient. For example, if you need to control the task with TaskCreationOptions, if you need to specify an alternative scheduler, or if, for performance reasons, you want to pass in object state, you should consider using Task.Factory.StartNew(). Only in rare cases, where you need to separate creation from scheduling, should constructor instantiation followed by a call to Start() be considered.

Listing 19.9 provides an example of using Task.Factory.StartNew().

Listing 19.9: Using Task.Factory.StartNew()

public Task<string> CalculatePiAsync(int digits)

{

return Task.Factory.StartNew<string>(

() => CalculatePi(digits));

}

private string CalculatePi(int digits)

{

// ...

}

\*\*\*COMP: Insert “End 5.0” tab

Long-Running Tasks

The thread pool assumes that work items will be processor bound and relatively short-lived; it makes these assumptions to effectively throttle the number of threads created. This prevents both overallocation of expensive thread resources and oversubscription of processors that would lead to excessive context switching and time slicing.

But what if the developer knows that a task will be long-running and, therefore, will hold on to an underlying thread resource for a long time? In this case, the developer can notify the scheduler that the task is unlikely to complete its work anytime soon. This has two effects. First, it hints to the scheduler that perhaps a dedicated thread ought to be created specifically for this task rather than attempting to use a thread from the thread pool. Second, it hints to the scheduler that perhaps this would be a good time to allow more tasks to be scheduled than there are processors to handle them. This will cause more time slicing to happen, which is a good thing. We do not want one long-running task to hog an entire processor and prevent shorter-running tasks from using it. The short-running tasks will be able to use their time slice to finish a large percentage of their work, and the long-running task is unlikely to notice the relatively slight delays caused by sharing a processor with other tasks. To accomplish this, use the TaskCreationOptions.LongRunning option when calling StartNew(), as shown in Listing 19.10. (Task.Run() does not support a TaskCreationOptions parameter.)

Listing 19.10: Cooperatively Executing Long-Running Tasks

using System.Threading.Tasks;

// ...

Task task = Task.Factory.StartNew(

() =>

WritePi(cancellationTokenSource.Token),

TaskCreationOptions.LongRunning);

// ...

|  |
| --- |
| Guidelines  DO inform the task factory that a newly created task is likely to be long-running so that it can manage it appropriately.  DO use TaskCreationOptions.LongRunning sparingly. |

Tasks Are Disposable

Note that Task also supports IDisposable. This is necessary because Task may allocate a WaitHandle when waiting for it to complete; since WaitHandle supports IDisposable, Task also supports IDisposable in accordance with best practices. However, note that the preceding code samples do not include a Dispose() call, nor do they rely on such a call implicitly via the using statement. The listings instead rely on an automatic WaitHandle finalizer invocation when the program exits.

This approach leads to two notable results. First, the handles live longer and hence consume more resources than they ought to. Second, the garbage collector is slightly less efficient because finalized objects survive into the next generation. However, both of these concerns are inconsequential in the Task case unless an extraordinarily large number of tasks are being finalized. Therefore, even though technically speaking all code should be disposing of tasks, you needn’t bother to do so unless performance metrics require it and it’s easy—that is, if you’re certain that Tasks have completed and no other code is using them.

\*\*\*COMP: Insert “Begin 5.0” tab

Working with System.Threading

The Parallel Extensions library is extraordinarily useful because it allows you to manipulate a higher-level abstraction, the task, rather than working directly with threads. However, you might need to work with code written before the TPL and PLINQ were available (prior to .NET 4.0), or you might have a programming problem not directly addressed by them. To do this you will leverage the Thread class and related API in System.Threading. System.Threading.Thread represents a point of control in the program, wrapping operating system threads while the namespace provides additional managed APIs to manage those threads.

One common method in Thread is Sleep(), however, in spite of its convenience, it should be avoided. Thread.Sleep() puts the current thread to sleep, essentially telling the operating system not to schedule any time slices to this thread until (at least) the given amount of time has passed. This might sound like a sensible thing to do, but it is a “bad code smell” that indicates the design of the program could probably be better. Putting a thread to sleep is a bad programming practice because the whole point of allocating an expensive resource like a thread is to get work out of that resource. (You wouldn’t pay an employee to sleep, so do not pay the price of allocating an expensive thread only to put it to sleep for millions or billions of processor cycles.) That said, there are a couple valid use cases.

The first is to put a thread to sleep with a time delay of zero to indicate to the operating system “the current thread is politely giving up the rest of its quantum to another thread if there is one that can use it” The polite thread will then be scheduled normally, without any further delay. The second is in test code to simulate a thread that is working on some high-latency operation without actually having to burn a processor doing some pointless arithmetic. Other uses in production code should be reviewed carefully to ensure that there is not a better way to obtain the desired effect.

Another type in System.Threading is the ThreadPool, which is designed to limit an excess number of threads to negatively impact performance. Threads are expensive resources, thread context switching is not free, and running two jobs in simulated parallelism via time slicing can be significantly slower than running them one after the other. And, while the thread pool does its job well, it does not include providing services to deal with long-running jobs or jobs that need to be synchronized with the main thread or with one another. What we really need to do is build a higher-level abstraction that can use threads and thread pools as an implementation detail - that abstraction. And, since TPL provides that abstraction, ThreadPool can essentially be deprecated entirely in favor of the TPL based APIs.

For more details on other techniques for managing worker threads that were commonly used prior to .NET 4, see the Essential C# 3.0 multithreading chapters at https://IntelliTect.com/EssentialCSharp.

|  |
| --- |
| Guidelines  AVOID calling Thread.Sleep() in production code.  DO use tasks and related APIs in favor of System.Theading classes such as Thread and ThreadPool. |

For more information on both System.Threading.ThreadPool and System.Threading.Thread's Sleep() method see XXXXXX.

Summary

At the beginning of this chapter, we briefly glossed over some of the difficult problems that developers often face when writing multithreaded programs: atomicity problems, deadlocks, and other race conditions that introduce uncertainty and bad behavior into multithreaded programs. We then delved into the Task Parallel Library (TPL) with it Task-based Asynchronous Pattern (TAP), a new API for creating and scheduling units of work represented by Task objects and how the simplify multithreaded programming by automatically rewriting your programs to manage the continuation “wiring” that composes larger tasks out of smaller tasks. In Chapter 20 and 21 we introduce additional high-level abstractions that cover additional scenarios and simplify TAP even further.

In Chapter 22, we switch over to cover how to avoid atomicity problems with synchronized access to shared resources without introducing deadlocks.

\*\*\*COMP: Insert “End 4.0” margin note

[19 1](#_Toc38470376)

[Introducing Multithreading 1](#_Toc38470377)

[Multithreading Basics 3](#_Toc38470378)

[Beginner Topic 3](#_Toc38470379)

[Multithreading Jargon 3](#_Toc38470380)

[Beginner Topic 6](#_Toc38470381)

[The Why and How of Multithreading 6](#_Toc38470382)

[Beginner Topic 8](#_Toc38470383)

[Performance Considerations 8](#_Toc38470384)

[Beginner Topic 9](#_Toc38470385)

[Threading Problems 9](#_Toc38470386)

[Most Operations Are Not Atomic 9](#_Toc38470387)

[Uncertainty Caused by Race Conditions 10](#_Toc38470388)

[Memory Models Are Complex 11](#_Toc38470389)

[Locking Leads to Deadlocks 11](#_Toc38470390)

[Asynchronous Tasks 12](#_Toc38470391)

[Why the TPL? 13](#_Toc38470392)

[Introducing Asynchronous Tasks 14](#_Toc38470393)

[Task Continuation 19](#_Toc38470394)

[Unhandled Exception Handling on Task with AggregateException 26](#_Toc38470395)

[AdVanced Topic 29](#_Toc38470396)

[Dealing with Unhandled Exceptions on a Thread 29](#_Toc38470397)

[Canceling a Task 33](#_Toc38470398)

[Task.Run(): A Shortcut and Simplification to Task.Factory.StartNew() 37](#_Toc38470399)

[Long-Running Tasks 37](#_Toc38470400)

[Tasks Are Disposable 38](#_Toc38470401)

[Working with System.Threading 39](#_Toc38470402)

[Summary 40](#_Toc38470403)

1. [↑](#footnote-ref-2)
2. . Technically, we ought to say that CPU always refers to the physical chip, and core may refer to a physical or virtual CPU. This distinction is unimportant for the purposes of this book, so we use these terms interchangeably. [↑](#footnote-ref-3)
3. . Either by putting the thread to sleep, spinning the thread, or spinning the thread before putting it back into sleep mode and repeating. [↑](#footnote-ref-4)
4. As a reminder TAP is Task-based Asynchronous Pattern. [↑](#footnote-ref-5)
5. As a reminder TPL is the Task Parallel Library [↑](#footnote-ref-6)
6. . Exercise caution when using this polling technique. When creating a task from a delegate, as we have here, the task will be scheduled to run on a worker thread from the thread pool. As a consequence, the current thread will loop until the work is complete on the worker thread. This technique works, but it might consume CPU resources unnecessarily. Such a polling technique is dangerously broken if, instead of scheduling the task to run on a worker thread, you schedule the task to execute in the future on the current thread. Since the current thread is in a loop polling the task, it will loop forever because the task will not complete until the current thread exits the loop. [↑](#footnote-ref-7)
7. . Be careful when using tasks to asynchronously mutate collections. The tasks might be running on worker threads, and the collection might not be thread safe. It is safer to fill in the collection from the main thread after the tasks are completed. [↑](#footnote-ref-8)
8. . Microsoft .NET Docs, https://docs.microsoft.com/dotnet/api/system.threading.tasks.taskcontinuationoptions [↑](#footnote-ref-9)
9. . In version 1.0 of the CLR, an unhandled exception on a worker thread terminated the thread but not the application. As a result, it was possible for a buggy program to have all its worker threads die, but the main thread would continue to run, even though the program was no longer doing any work. This is a confusing situation for users to be in; it is better to signal to the user that the application is in a bad state and terminate it before it can do any more harm. [↑](#footnote-ref-10)
10. . As we discussed earlier, waiting for a fault continuation to complete is a strange thing to do because most of the time it will never be scheduled to run in the first place. This code is provided for illustrative purpoes only. [↑](#footnote-ref-11)