Contents

[Use the ValueTask<TResult> 1](#_Toc171447242)

[Combine multiple tasks 1](#_Toc171447243)

[Use CancellationToken 1](#_Toc171447244)

[Limit the number of concurrent tasks 2](#_Toc171447245)

[Custom awaiters 2](#_Toc171447246)

[Async streams 3](#_Toc171447247)

[Asynchronous disposal 3](#_Toc171447248)

[Best practices 3](#_Toc171447249)

[Task.WhenAll 3](#_Toc171447250)

[WhenAll vs WaitAll 4](#_Toc171447251)

[Task vs ValueTask 6](#_Toc171447252)

# Task vs Thread

**Thread** is a lower-level concept: if you're directly starting a thread, you *know* it will be a separate thread, rather than executing on the thread pool etc.

**Task** is more than just an abstraction of "where to run some code" though - it's really just "the promise of a result in the future". So as some different examples:

* Task.Delay doesn't need any actual CPU time; it's just like setting a timer to go off in the future
* A task returned by WebClient.DownloadStringTaskAsync won't take much CPU time locally; it's representing a result which is likely to spend most of its time in network latency or remote work (at the web server)
* A task returned by Task.Run() really *is* saying "I want you to execute this code separately"; the exact thread on which that code executes depends on a number of factors.

Note that the Task<T> abstraction is pivotal to the async support in C# 5.

In general, I'd recommend that you use the higher level abstraction wherever you can: in modern C# code you should rarely need to explicitly start your own thread.

# Use the ValueTask<TResult>

…struct for high-performance scenarios where the result is often available synchronously. This can help reduce memory allocations and improve performance:

public async ValueTask<int> CalculateResultAsync(int input)

{

if (input < 0)

{

return -1;

}

int result = await Task.Run(() => PerformComplexCalculation(input));

return result;

}

# Combine multiple tasks

…using Task.WhenAll or Task.WhenAny to perform parallel operations, improving the overall efficiency of your application:

public async Task ProcessDataAsync()

{

Task<string> fetchDataTask = FetchDataAsync();

Task<string> readDataTask = ReadDataAsync();

await Task.WhenAll(fetchDataTask, readDataTask);

string fetchedData = fetchDataTask.Result;

string readData = readDataTask.Result;

// Process the data

}

# Use CancellationToken

…to cancel long-running tasks gracefully, enabling better resource management and preventing unnecessary work:

public async Task<string> FetchDataAsync(CancellationToken cancellationToken)

{

using (var httpClient = new HttpClient())

{

cancellationToken.ThrowIfCancellationRequested();

string result = await httpClient.GetStringAsync("https://example.com/data").ConfigureAwait(false);

return result;

}

}

public async Task ProcessDataAsync()

{

var cancellationTokenSource = new CancellationTokenSource();

try

{

string data = await FetchDataAsync(cancellationTokenSource.Token);

// Process the data

}

catch (OperationCanceledException)

{

// Handle the cancellation

}

cancellationTokenSource.Dispose();

}

# Limit the number of concurrent tasks

…when dealing with loops and async methods, to avoid excessive resource usage. This can be achieved using techniques such as SemaphoreSlim:

public async Task ProcessMultipleFilesAsync(IEnumerable<string> filePaths)

{

var semaphore = new SemaphoreSlim(4); // Limit to 4 concurrent tasks

var tasks = filePaths.Select(async filePath =>

{

await semaphore.WaitAsync();

try

{

await ProcessFileAsync(filePath);

}

finally

{

semaphore.Release();

}

});

await Task.WhenAll(tasks);

}

# Custom awaiters

It’s possible to create custom awaiters by implementing the INotifyCompletion or ICriticalNotifyCompletion interfaces and providing a GetAwaiter method for a specific type. This allows you to use the await keyword with custom types, enabling advanced scenarios and optimizations.

# Async streams

In C# 8.0, you can use the IAsyncEnumerable<T> interface and await foreach to work with asynchronous streams. This allows you to asynchronously enumerate and process collections of items that are produced or fetched asynchronously.

# Asynchronous disposal

C# 8.0 also introduced the IAsyncDisposable interface, which provides an async version of the Dispose method called DisposeAsync. This enables you to perform asynchronous cleanup operations when disposing of resources.

Typically, IAsyncDisposable is used with classes or objects that have async methods and also hold onto unmanaged resources such as file handles, network connections, or database connections.

# Best practices

* Avoid using async void methods, as they can’t be awaited and can lead to unhandled exceptions. Instead, use async Task methods for event handlers, which provide proper exception handling.
* Use ConfigureAwait(false) when possible to avoid capturing the context, especially in library code or when optimizing performance. This reduces the risk of deadlocks and improves efficiency.
* Use Task.Run for CPU-bound operations that can benefit from parallelism, effectively offloading the work to a separate thread and preventing the main thread from being blocked.
* Limit the number of concurrent tasks when using async methods in loops, using techniques such as SemaphoreSlim or Task.WhenAll with a limited number of tasks to avoid excessive resource usage.

# Task.WhenAll

Task.WhenAll is a method provided by the Task class that takes an IEnumerable<Task> or a set of Task instances as parameters and returns a new Task that completes once all input tasks have completed.

Potential issues:

* exceptions thrown by individual tasks are wrapped in an AggregateException. It’s crucial to handle this AggregateException and to unwrap it to understand the actual exceptions that occurred in the tasks. Otherwise, you may miss critical exceptions or experience unexpected behavior.

try

{

await Task.WhenAll(task1, task2, task3);

}

catch (AggregateException ae)

{

// Handle the individual exceptions using Unwrap or InnerExceptions

foreach (var innerException in ae.InnerExceptions)

{

// Handle each exception as necessary

}

}

* Thread Starvation: If you’re not careful about how many tasks you’re waiting for concurrently, your application may experience thread starvation, especially in cases where your tasks have a high degree of parallelism. Consider using SemaphoreSlim or other concurrency control methods to limit the number of concurrent tasks.
* Resource Exhaustion: Running an excessive number of tasks concurrently can lead to memory pressure, file handle exhaustion, or network limitations.

Task.WaitAll

# WhenAll vs WaitAll

**In genereal**, Task.WaitAll is a method that blocks the main thread until all tasks complete, whereas Task.WhenAll is an asynchronous operation, enabling you to initiate multiple tasks without blocking the main thread. Task.WhenAll returns a task that represents the collective completion of those tasks.

**Impact on Main Thread Execution:**

- Task.WaitAll: Blocks the main thread until all specified tasks are completed, potentially causing the main thread to wait and not perform other work.

- Task.WhenAll: Doesn’t block the main thread, enabling it to continue executing other tasks or operations while concurrently monitoring the completion of the specified tasks. This is particularly useful for tasks involving I/O or network calls.

**Return Types and Completion Handling:**

- Task.WaitAll: Doesn’t return a value; it’s a void method. Used primarily when ensuring that a set of tasks have finished before proceeding further in your code.

- Task.WhenAll: Returns a new task that represents the completion of all tasks provided in an enumerable collection. Suitable for asynchronous workflows and easier to work with when awaiting the combined completion of multiple tasks.

**Exception Handling:**

- Task.WaitAll: When any of the tasks being waited for throw exceptions during their execution, Task.WaitAll consolidates these exceptions into a System.AggregateException and throws it. You’ll receive a single aggregate exception containing all individual exceptions if multiple tasks encounter exceptions.

In the code below, we have two tasks: task1 and task2. Task1 throws a NullReferenceException, while task2 throws an InvalidTimeZoneException during execution. Both of these exceptions are captured and can be observed in the error output.

Task[] tasks = new Task[2];  
  
tasks[0] = Task.Run(() => throw new NullReferenceException());  
tasks[1] = Task.Run(() => throw new InvalidTimeZoneException());  
  
Task.WaitAll(tasks);  
  
  
// Unhandled exception. System.AggregateException: One or more errors occurred. (Object reference not set to an instance of an object.) (Exception of type 'System.InvalidTimeZoneException' was thrown.)  
// ---> System.NullReferenceException: Object reference not set to an instance of an object.  
// at Program.<>c.<<Main>$>b\_\_0\_0()  
// at System.Threading.Tasks.Task`1.InnerInvoke()  
// at System.Threading.ExecutionContext.RunFromThreadPoolDispatchLoop(Thread threadPoolThread, ExecutionContext executionContext, ContextCallback callback, Object state)  
// ---> (Inner Exception #1) System.InvalidTimeZoneException: Exception of type 'System.InvalidTimeZoneException' was thrown.  
// at Program.<>c.<<Main>$>b\_\_0\_1()  
// at System.Threading.Tasks.Task`1.InnerInvoke()  
// at System.Threading.ExecutionContext.RunFromThreadPoolDispatchLoop(Thread threadPoolThread, ExecutionContext executionContext, ContextCallback callback, Object state)

*-***Task.WhenAll:** In contrast, if any of the tasks you’re waiting for throw exceptions during their execution with Task.WhenAll, it will **unwrap the AggregateException and return only the first exception** it encounters.

In the following code, there are two tasks: task1 and task2. Task1 throws a NullReferenceException, and task2 throws an InvalidTimeZoneException during execution. When handling exceptions, the code unwraps the AggregateException and returns only the first exception encountered.

Task[] tasks = new Task[2];  
  
tasks[0] = Task.Run(() => throw new NullReferenceException());  
tasks[1] = Task.Run(() => throw new InvalidTimeZoneException());  
  
await Task.WhenAll(tasks);  
  
// Unhandled exception. System.NullReferenceException: Object reference not set to an instance of an object.  
// at Program.<>c.<<Main>$>b\_\_0\_0() in  
// at System.Threading.Tasks.Task`1.InnerInvoke()  
// at System.Threading.ExecutionContext.RunFromThreadPoolDispatchLoop(Thread threadPoolThread, ExecutionContext executionContext, ContextCallback callback, Object state)  
// --- End of stack trace from previous location ---  
// at System.Threading.ExecutionContext.RunFromThreadPoolDispatchLoop(Thread threadPoolThread, ExecutionContext executionContext, ContextCallback callback, Object state)  
// at System.Threading.Tasks.Task.ExecuteWithThreadLocal(Task& currentTaskSlot, Thread threadPoolThread)  
// --- End of stack trace from previous location ---  
// at Program.<Main>$(String[] args) in  
// at Program.<Main>(String[] args)

# Task vs ValueTask

ValueTask лучше юзать, когда в методе чаще ожидается синхронная работа. Например данные берутся из кэша, а если в кэше нет, то делается асинхронный запрос в сервис. Тогда будет меньше аллокаций в хипе.

В остальных случаях лучше Task.

public async ValueTask<int> PerformOperationAsync()

{

if (cache.TryGetFromCache( out int result))

return result;

result = await ComputeResultAsync();

cache.StoreInCache(result);

return result;

}

Но важно не переполнить стек (если много параллельных вызовов). Короче, надо всегда смотреть.

<https://www.youtube.com/watch?v=mEhkelf0K6g>