C# - Ultimate Guide - Beginner to Advanced | Master class

Section 32 – Tasks

Drawbacks of Threading

Complexity

Threading can be significantly more complex to manage than tasks. You have to deal with low-level thread creation, synchronization primitives (e.g., locks, mutexes) which can lead to more error-prone code.

Scalability

Threads are heavyweight resources, and creating too many of them can lead to excessive memory usage and decreased performance.

Cancellation

Implementation of cancellation that allows you to gracefully terminate ongoing operations is possible in threads; but it is too complex and inefficient.

Asynchronous Programming

Threads don't support asynchronous programming out-of-box.

When you try to implement asynchronous I/O operations using threads, they tend to block the thread until the I/O operation completes. So it leads to overload on system resources.

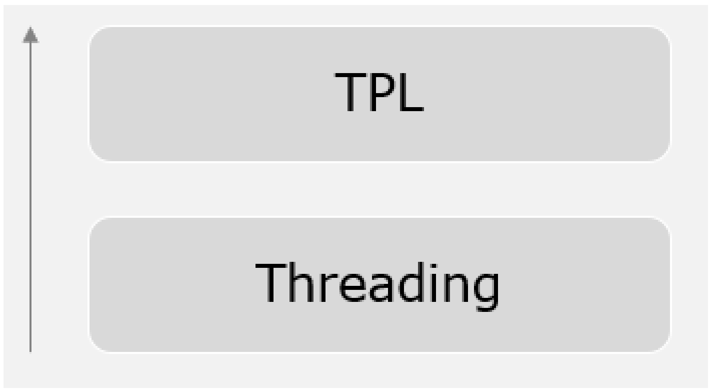
Parallel Programming

Parallel programming allows you to execute the code parallelly on multiple CPU-cores, which is not supported by the threads.

Introduction to Tasks & TPL

The Task Parallel Library (TPL) is a set of APIs and runtime features, which provides a higher-level abstraction to implement concurrency, parallelism and asynchronous programming, which works based on threading.

TPL internally works based on "Thread Pool".

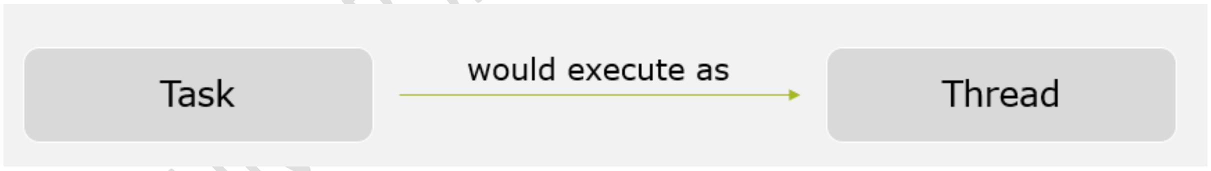


Tasks

The fundamental building block of TPL is the "Task".

A Task represents an operation (method) that can be executed concurrently & parallelly with other tasks.

Tasks can represent both CPU-bound and I/O-bound operations.



How TPL Works Internally?

Task Creation

When you create a task, it gets scheduled by the Task Scheduler.

Task Scheduler

The 'Task Scheduler' is responsible for managing the execution of tasks. It determines how tasks are executed and which threads they run on.

Thread Pool

The Task Scheduler utilizes the Thread Pool to allocate and manage threads. It avoids the overhead of creating new threads for each task by reusing existing Thread Pool threads.

Concurrency Control

The Task Scheduler ensures proper coordination and scheduling of tasks to maximize CPU utilization and minimize contention for resources.

Features and Advantages of TPL

Abstraction

TPL abstracts away low-level details of thread management, making it easier to work with concurrency.

Efficient Resource Utilization

TPL efficiently manages threads via the Task Scheduler, which reuses threads from the Thread Pool, reducing resource overhead.

Cancellation

TPL supports graceful cancellation of tasks using CancellationToken, allowing the user to cancel desired tasks.

Asynchronous Programming

Tasks are well-suited for asynchronous programming, allowing non-blocking I/O operations for improved responsiveness.

Parallelism

TPL seamlessly integrates with parallel programming constructs like Parallel.ForEach and PLINQ (Parallel LINQ) to parallelize operations on collections.

Task Continuations

You can define task continuations that specify what to do when a task completes, fails, or is canceled.

Use Cases for TPL

Multithreaded computations, where CPU-bound tasks can be parallelized.

Asynchronous I/O operations, such as reading/writing files, making network requests, or database interactions.

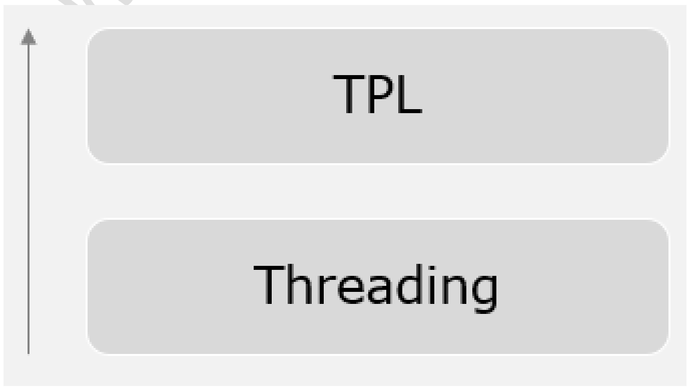
Parallel processing of collections using Parallel.ForEach or PLINQ.

Responsive user interfaces that require non-blocking operations.

'Task' Class

The "Task" class (System.Threading.Tasks.Task) represents a task (a building block of TPL) which represents an operation (method) that can be executed concurrently & parallelly with other tasks.

Tasks are internally managed by "Task Scheduler" in TPL.



Properties of 'Task' class

public bool IsCompleted { get; }

public TaskStatus Status { get; }

public bool IsFaulted { get; }

public bool IsCanceled { get; }

public AggregateException Exception { get; }

public int Id { get; }

public TResult Result { get; }

Methods of 'Task' class – Part 1

public void Wait()

public bool Wait(int millisecondsTimeout)

public static void WaitAll(params Task[] tasks)

public static int WaitAny(params Task[] tasks)

public Task ContinueWith(Action<Task> continuationAction)

Methods of 'Task' class – Part 2

public static Task WhenAll(IEnumerable<Task> tasks)

public static Task<Task> WhenAny(IEnumerable<Task> tasks)

public static Task<TResult> FromResult(TResult result)

public static Task Delay(int millisecondsDelay)

public static Task FromException(Exception exception)

public static Task Run(Action action)

public static Task<TResult> Run(Func<TResult> function)

Task.Run

public static Task Run(Action action)

It is used to create a task. It submits the given method reference to the 'Task Scheduler', which leads to create & run the task.

It creates an returns an object of "Task" type.

‘Action’ delegate: The method / lambda expression should receive NO arguments and NO return value.

Stopwatch

'Stopwatch' class

The "Stopwatch" class (System.Diagnostics.Stopwatch) provides functionality for measuring time intervals with high precision.

It is used measure the execution time of a block of code or the performance of a specific operation.

Properties of 'Stopwatch' class

bool IsRunning { get; }

Property that indicates whether the stopwatch is currently running.

long ElapsedMilliseconds { get; }

Property that returns the total elapsed time in milliseconds

Methods of 'Stopwatch' class

void Start()

Method that starts or resumes the stopwatch.

void Stop()

Method that stops or pauses the stopwatch.

void Reset()

Method that resets the stopwatch, clearing any recorded time.

void Restart()

Method that combines the functions of both Reset() and Start(), resetting the stopwatch and immediately starting a new measurement.

Task.Factory.StartNew( )

public Task StartNew(Action action)

It is used to create a task. It submits the given method reference to the 'Task Scheduler', which leads to create & run the task.

It creates an returns an object of "Task" type.

'Action' delegate: The method / lambda expression should receive NO arguments and NO return value.

public Task StartNew(Action action,

CancellationToken cancellationToken,

TaskCreationOptions creationOptions,

TaskScheduler scheduler)

It also creates a new task and creates an object of "Task" type and returns the same, much like StartNew(Action action) method.

This overload of the method can specify custom value for CancellationToken, TaskCreationOptions and custom TaskScheduler.

Task.Wait( )

public void Wait()

This method is used to block the current thread until the task on which the method is called completes its execution.

It is similar to "Thread.Join()" method.

Task<TResult>

public class Task<TResult> : Task

It is a generic class that represents an asynchronous operation much like "Task" class; but the Task<TResult> class can produce a result of type TResult.

It inherits the "Task" class; so it contains all properties & methods of "Task" class.

Creating Task<TResult>

class System.Threading.Tasks.Task

public static Task<TResult> Run(Func<TResult> function)

It is used to create a task. It submits the given method reference to the 'Task Scheduler', which leads to create & run the task.

It creates an returns an object of "Task<TResult>" type.

The method / lambda expression should receive NO arguments; but can return a value of "TResult" type.

class System.Threading.Tasks.TaskFactory

public Task<TResult> StartNew(Func<TResult> function)

It is used to create a task. It submits the given method reference to the 'Task Scheduler', which leads to create & run the task.

It creates an returns an object of "Task<TResult>" type.

The method / lambda expression should receive NO arguments; but can return a value of "TResult" type.

Task.WaitAll( )

public static void WaitAll(params Task[] tasks)

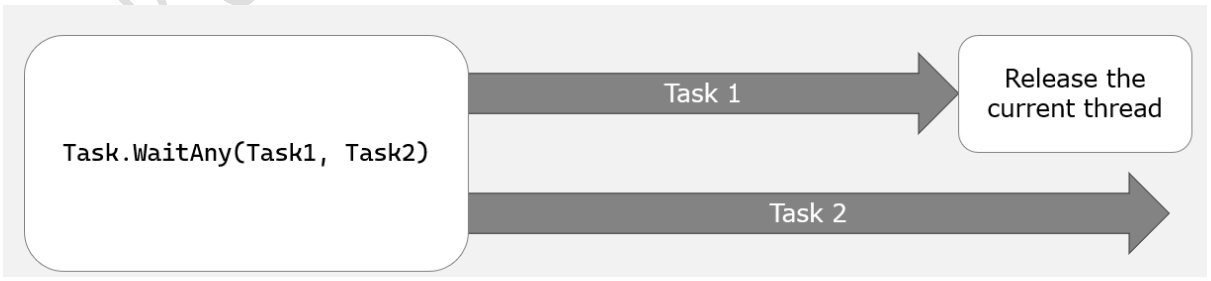
This method is used to block the current thread, waiting for the completion of multiple tasks before proceeding with further execution.

It is logically equivalent to calling "Task.Wait()" method on each task.

Task.WaitAny( )

public static int WaitAny(params Task[] tasks)

This method is used to block the current thread, waiting for the completion of any one of the specified tasks before proceeding.



Task.Delay( )

public static Task Delay(int millisecondsDelay)

This method is used make a delay or pause in the execution of tasks without blocking the calling thread.

It is similar to Thread.Sleep() method in Threading.

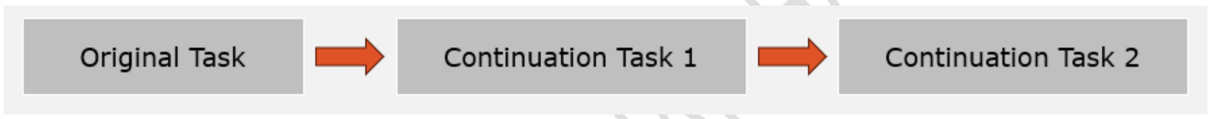
It doesn't pause the current task; but instead, it creates and returns a new "Task" object.

public Task ContinueWith(Action<Task> continuationTask)

This method schedules a "continuation task" to run when the "original task" is completed (either successfully or with an exception).

It creates an returns an object of "Task" class that represents the continuation task.

The continuation task can read result or exception of the original task.



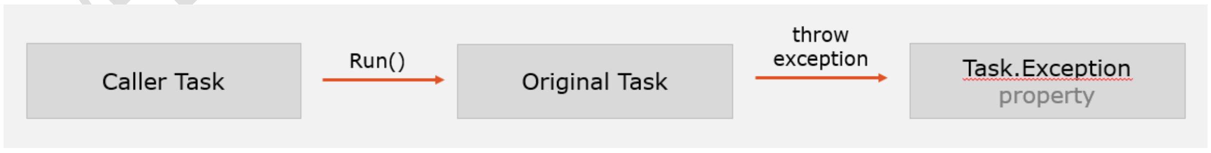
Task Exception Handling

Properties of 'Task' class

public AggregateException Exception { get; }

Gets the exception that caused the task to fault (if any).

It is null if the task completed successfully or wasn't faulted

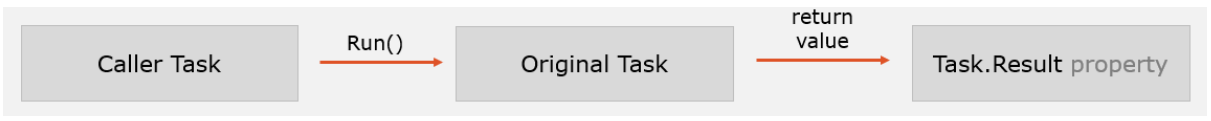


Properties of 'Task<T>' class

public TResult Result { get; }

Gets the result of the task that was returned by the task method.

If the task is not completed, it automatically blocks the current thread until the task gets completed.



Properties of 'Task' class

public TaskStatus Status { get; }

Gets the current status of the task; any one of the following values.

1. WaitingForActivation

The task is created but hasn't started executing yet.

It is waiting to be scheduled by the task scheduler.

2. WaitingToRun

The task is scheduled to run but is waiting for its turn to be executed by the task scheduler.

public TaskStatus Status { get; }

Gets the current status of the task; any one of the following values.

3. Running

The task is currently executing its code. It's actively running.

4. RanToCompletion

The task has completed successfully without any exceptions.

It has reached to end of the task method successfully.

public TaskStatus Status { get; }

Gets the current status of the task; any one of the following values.

5. Faulted

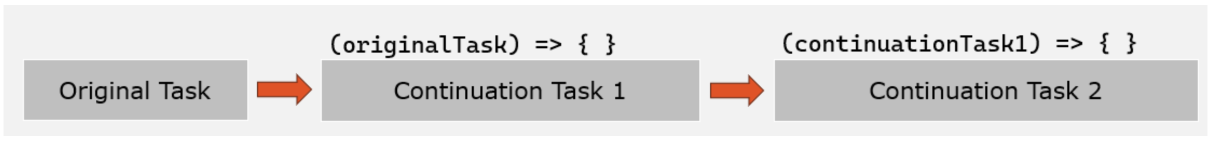
The task completed with an unhandled exception.

This indicates that an error occurred during the execution of the task.

6. Canceled

The task was canceled using the CancellationToken.

Task Continuation Chain

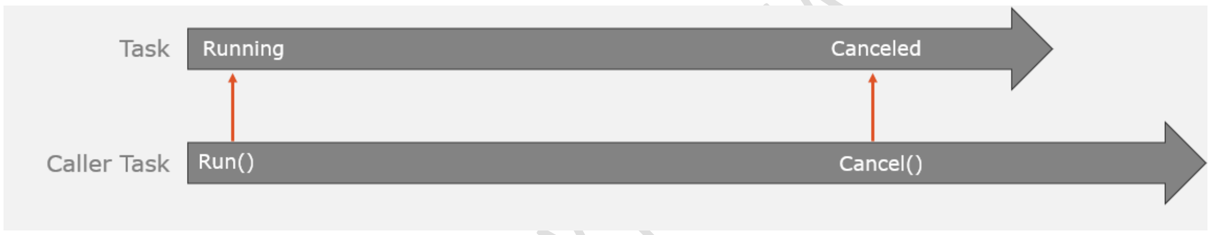


Task Cancellation

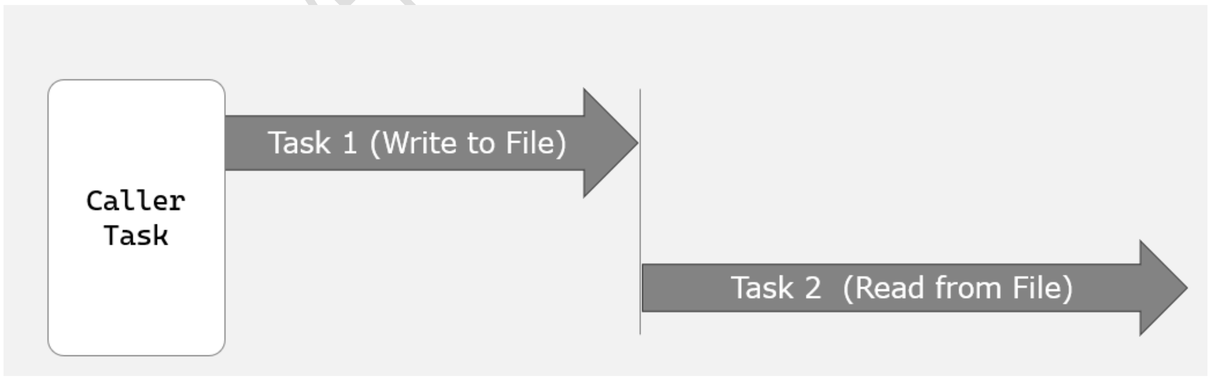
Task cancellation is a process that allows you to gracefully cancel the execution of one or more tasks.

Initially, the caller task creates a "cancellation token" and associates the same with the task.

Subsequently, the caller task can cancel the task using the same "cancellation token".



File I/O Operations with TPL


Caller Task
Task 1 (Write to File)
Task 2 (Read from File)