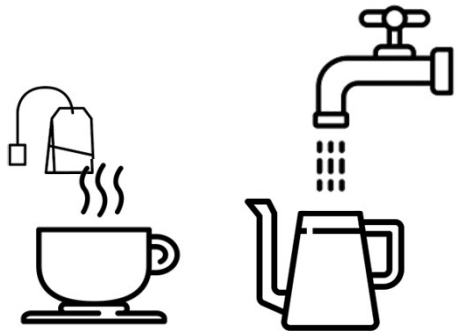
Asynchronus programming

Concurrency, Parallellism, Threads and Tasks in C#

www.everyloop.com

Synchronous programming



GetKettle();

FillKettle();

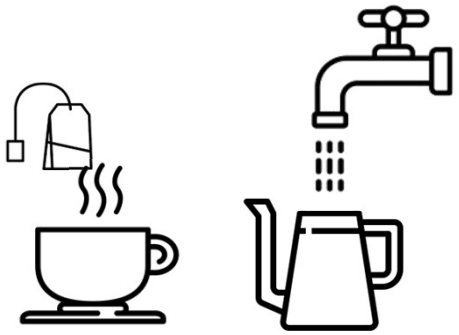
BoilWater();

GetCup();

AddTea();

PourWater();

Asynchronous programming



GetKettle();

FillKettle();

BoilWaterAsync();

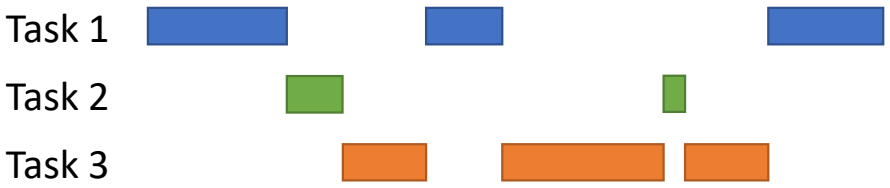
GetCup();

AddTea();

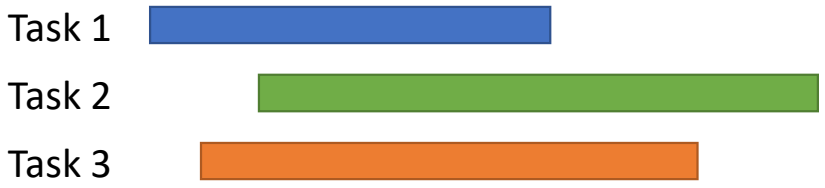
Await boiling water

PourWater();

Concurrency



Parallellism

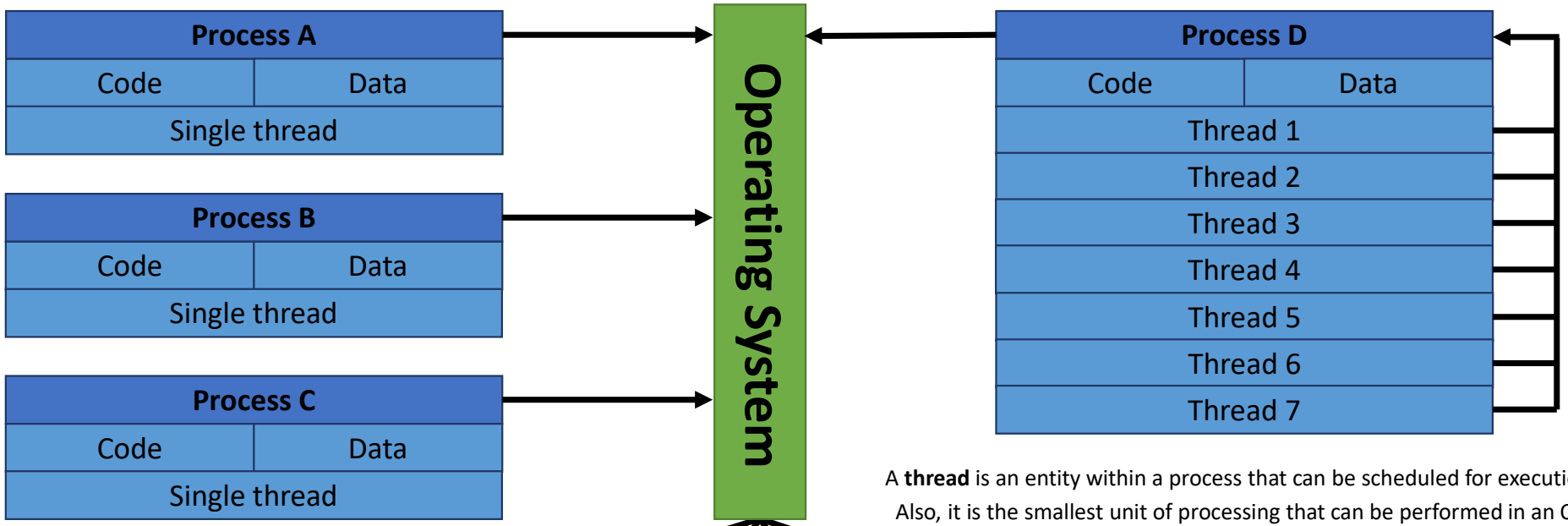


Multitasking

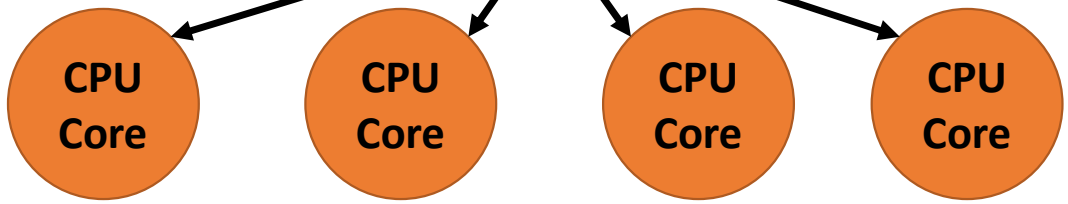
The concurrent execution of multiple tasks (also known as processes)

Multithreading

the ability of a processor to execute multiple threads concurrently



A **thread** is an entity within a process that can be scheduled for execution. Also, it is the smallest unit of processing that can be performed in an OS.



Multiprocessing

the use of two or more central processing units (CPUs) within a single computer system

Threads, tasks and threadpool

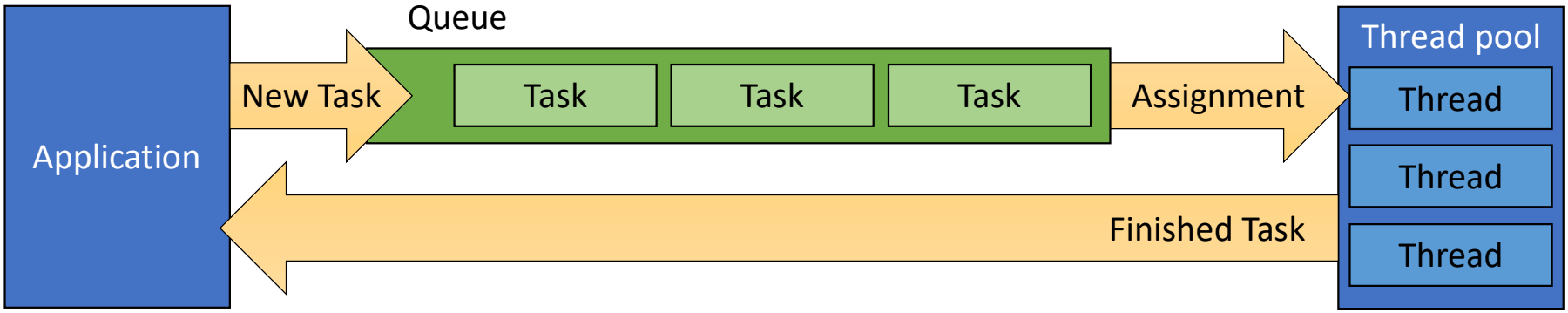
In modern C# the .NET classes Thread and Threadpool should not be used directly in most cases.

Thread represents an actual OS-level thread, with its own stack and kernel resources. While Thread allows the highest degree of control, it can be quite tricky to juggle the low level aspects of multi threading manually.

Another problem with Thread is that OS threads are costly. Each thread consumes a non-trivial amount of memory for its stack, and adds additional CPU overhead.

ThreadPool is a wrapper around a pool of threads maintained by the CLR, that allows for low level control of the pool.

A thread pool maintains multiple threads waiting for tasks to be allocated for concurrent execution. The use of a thread pool avoids the overhead of creating too many threads and is instead reusing the threads in the pool to work through the queued tasks.



Task Parallell Library (TPL) Starting with the .NET Framework 4, the TPL is the preferred way to write multithreaded and parallel code.

https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl

Task and Task<TResult>

A future (or promise) is a type that represents some operation that will complete in the future.
The modern future types in .NET are Task and Task<TResult>

The **Task** class represents a single operation that does not return a value and that usually executes asynchronously

The **Task<TResult>** class represents a single operation that returns a value and that usually executes asynchronously

Task.Run() queues the specified work to run on the ThreadPool and returns a task or Task<TResult> handle for that work.

The **Status** property gets the current status of the task, such as WaitingToRun, Canceld, Running or RanToCompletion.

The **Wait()** method makes the program wait synchronously for the task to complete execution.

The **Result** property gets the result value of a Task<TResult>

The **ContinueWith()** method creates a continuation that executes asynchronously when the target Task completes.

Examples:

```
Task task = Task.Run(() => {
   Console.WriteLine("Running task!");
});

if (task.Status == TaskStatus.Running)
   Console.WriteLine("Task is still running!");
});

task.Wait();

Task<int> task2 = Task.Run(() => {
   int result = 0;
   for (int i = 0; i < 100; i++) {
      result += i;
   }
   return result;
});

task2.ContinueWith(t => Console.WriteLine(t.Result));
```

Thread safety

Multithreading solves problems with throughput and responsiveness, but in doing so it introduces new problems.

A race condition is a bug that occurs when the outcome of a program depends on which of two or more threads reaches a particular block of code first. Running the program many times produces different results, and the result of any given run cannot be predicted.

A deadlock occurs when each of two threads tries to lock a resource the other has already locked. Neither thread can make any further progress.

Thread-safe collections

The .NET Framework 4 introduces the System.Collections.Concurrent namespace, which includes several collection classes that are both thread-safe and scalable. Multiple threads can safely and efficiently add or remove items from these collections, without requiring additional synchronization in user code. When you write new code, use the concurrent collection classes whenever multiple threads will write to the collection concurrently.

ConcurrentDictionary<TKey, TValue>

ConcurrentStack<T>

ConcurrentQueue<T>

ConcurrentBag<T>

BlockingCollection<T>

Async and Await

Modern asynchronous .NET applications use two keywords: async and await.

The async keyword is added to a method declaration, and its primary purpose is to enable the await keyword within that method (the keywords were introduced as a pair for backward-compatibility reasons).

An async method should return Task<T> if it returns a value, or Task if it does not return a value.

Avoid async void!
It is possible to have an async method return void, but you should only do this if you're writing an async event handler.
A regular async method without a return value should return Task, not void.

An async method begins executing synchronously, just like any other method.
Within an async method, the await keyword performs an asynchronous wait on its argument.
First, it checks whether the operation is already complete; if it is, it continues executing (synchronously).
Otherwise, it will pause the async method and return an incomplete task.
When that operation completes some time later, the async method will resume executing.

Once you start using async, it's best to allow it to grow through your code.
If you call an async method, you should (eventually) await the task it returns.