MATHEMA

Multithreading / Asynchronous Programming in .NET

Diego Correa

Mathema GmbH

About me

Diego Correa

Software Developer, from Paraguay, center of Southamerica.
I arrived in Germany a year ago.

Emphasis on the following programming languages: .NET/C#, Java, Javascript

Session content

- Benefits of Multithreading and Async
- Process, Program, Application
- States and a Lifetime of a Thread
- Synchronization of Threads
- Multithreading and Async Programming

Benefits

Responsiveness

- For example, in a web browser allow user interaction in one thread while a video is being loaded in another thread.

Resource Sharing

- The benefit of sharing code and data is that it allows an application to have several threads of activity within the same address space. **Keep in mind sync strategies!**.

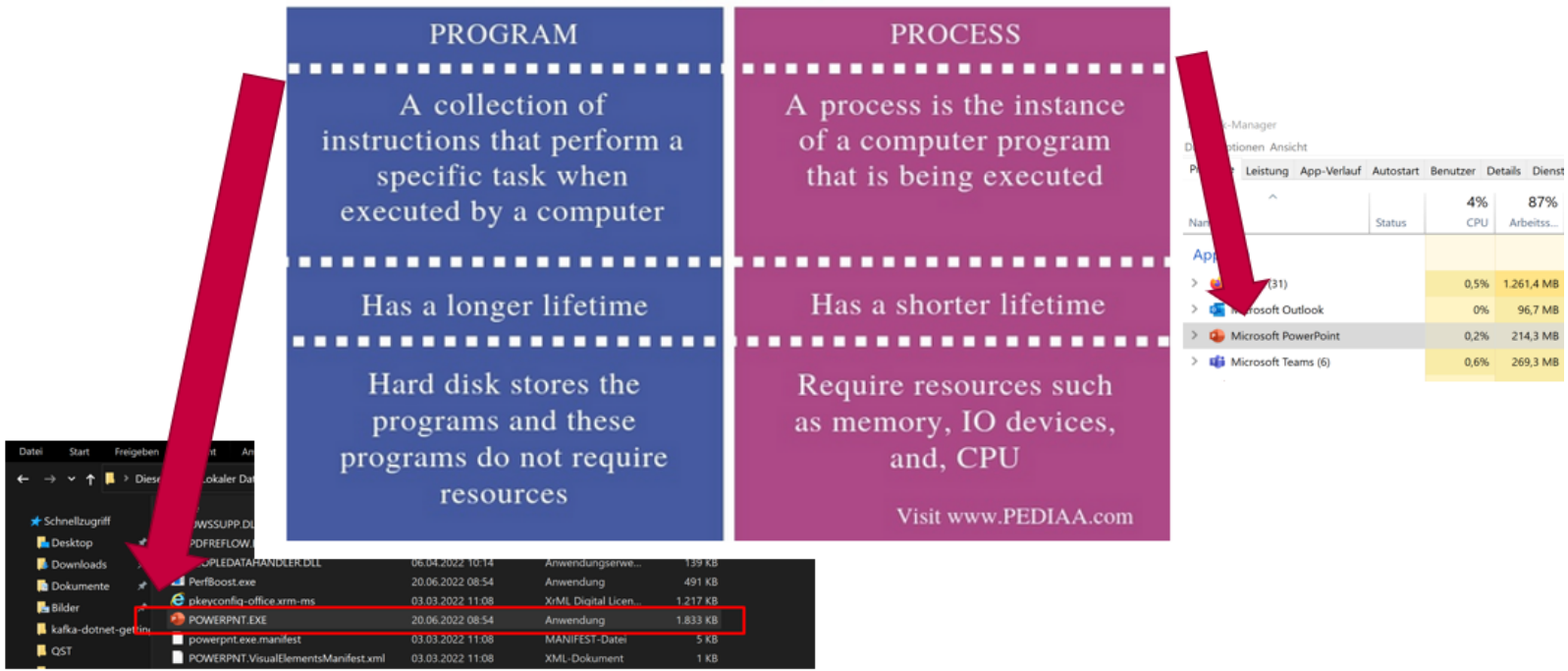
Economy

- The context switching of working with threads is cheaper than working with processes because they are lighter. In Solaris, for example, creating a process is ***30 time slower*** than creating threads and context switching is ***5 times slower***.

Scalability

- Increase in case of multiprocessor architecture.

Program and Process



Program and Application

Program	Application
All programs are not applications	All applications are programs
Do not require applications to operate	Applications require programs to operate

Application: higher-level concept that encompasses one or more programs or processes working together to provide a specific functionality or service to users.

Tasks and Threads

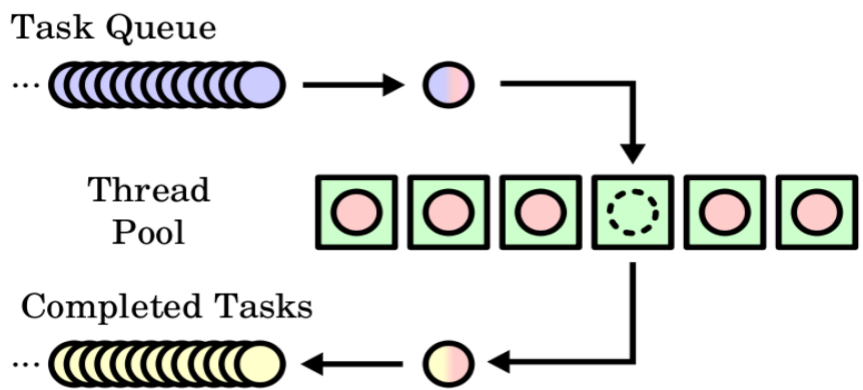
A **task** is a future or promise, a **thread** is a way to fullfilling that promise.

A **task** uses the threadpool, which saves resources as creating threads can be expensive.

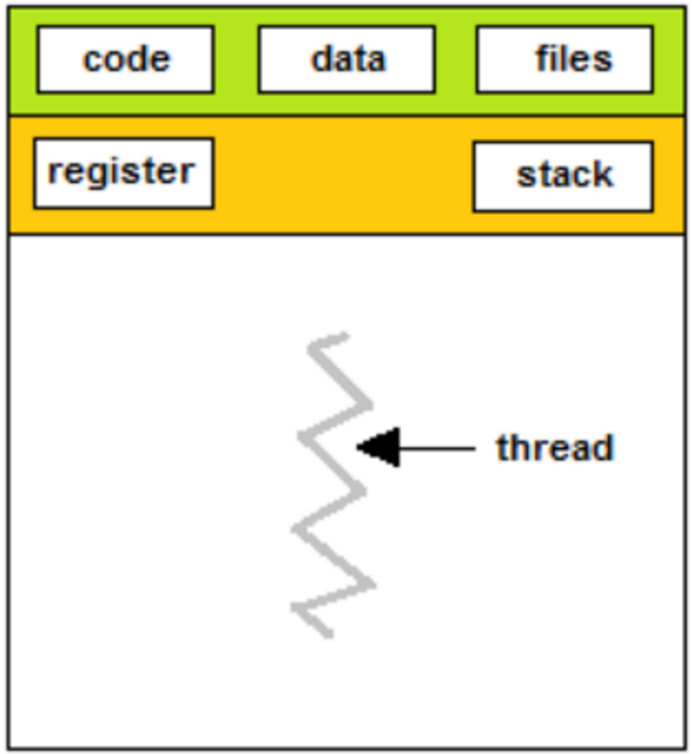
A **thread** is not a .NET construct, they are built into your operating system.

A **task** returns a result and a cancellation, a **thread** not.

Threadpool: pre-created thread structure in memory managed by CLR

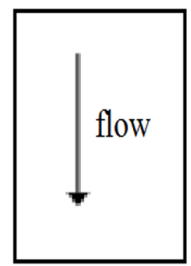


Sequential Execution of Process



single-threaded process

Are unique in that they don't share data and information; they're isolated execution entities. In short, a process has its own stack, memory, and data



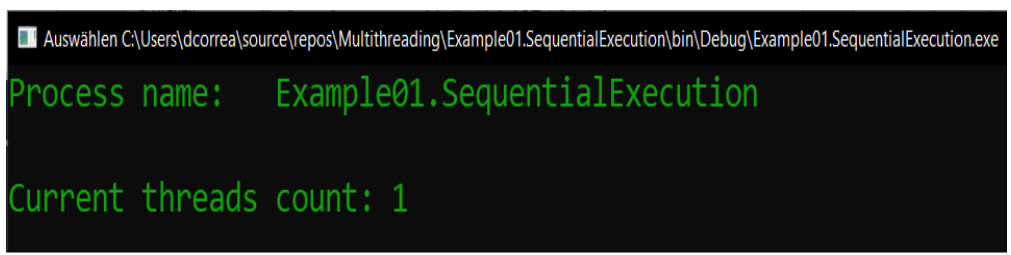
Single Thread Program

By default every program has single thread

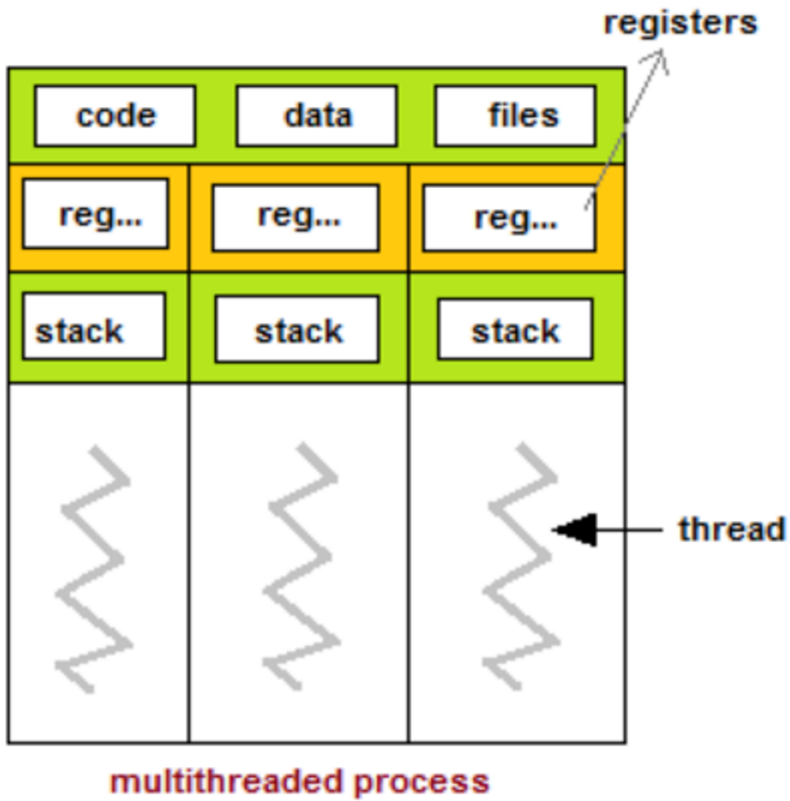
One flow direction

DEMO: SequentialExecution

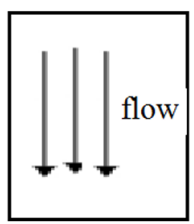
```
static void Main(string[] args)
{
    var currentProcess = System.Diagnostics.Process.GetCurrentProcess();
    Console.WriteLine("Process name:\t" + currentProcess.ProcessName);
    Console.WriteLine("\nCurrent threads count: " + GetThreadsCount());
}
```



Concept of Thread inside a Process



- Own program counter.
- Own stack.
- Own set of registers where the program counter mainly keeps track of which instruction to execute next, a set of register mainly hold its current working variables, and a stack mainly contains the history of execution.

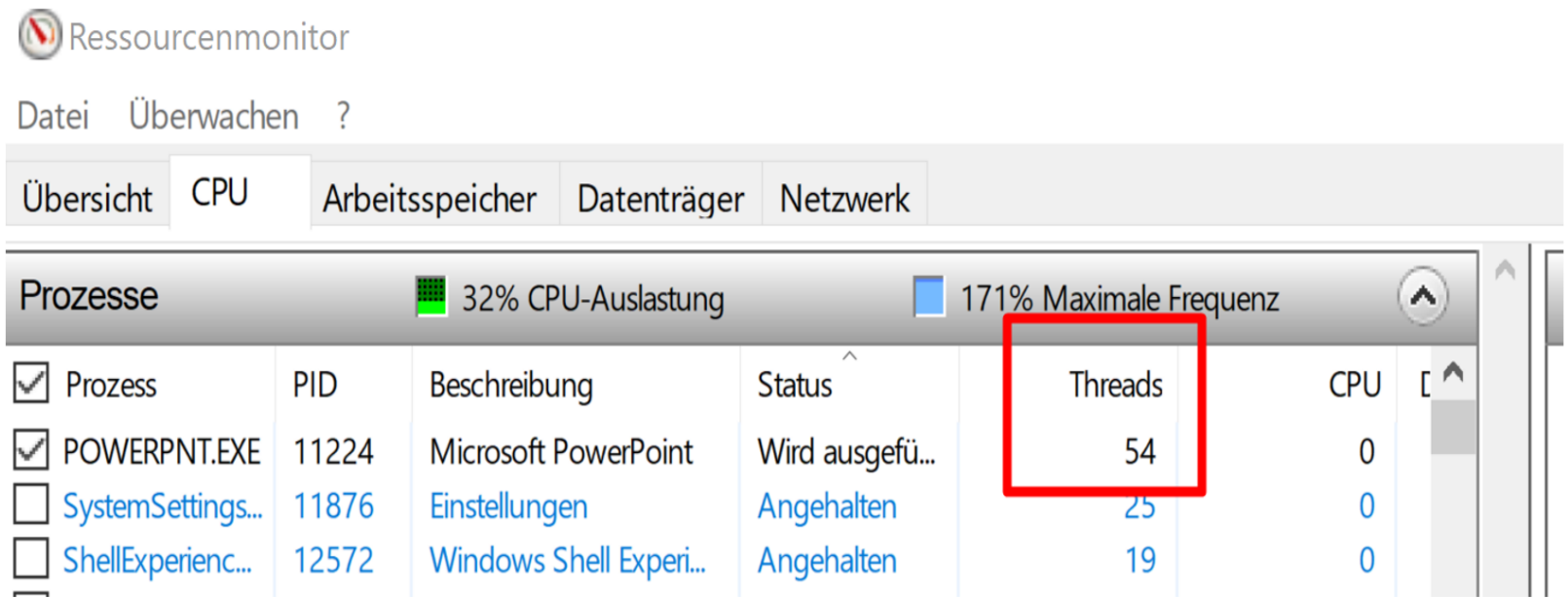


Multiple Thread Program

More the one flow of a program,executing in specified fashion

Concept of Thread

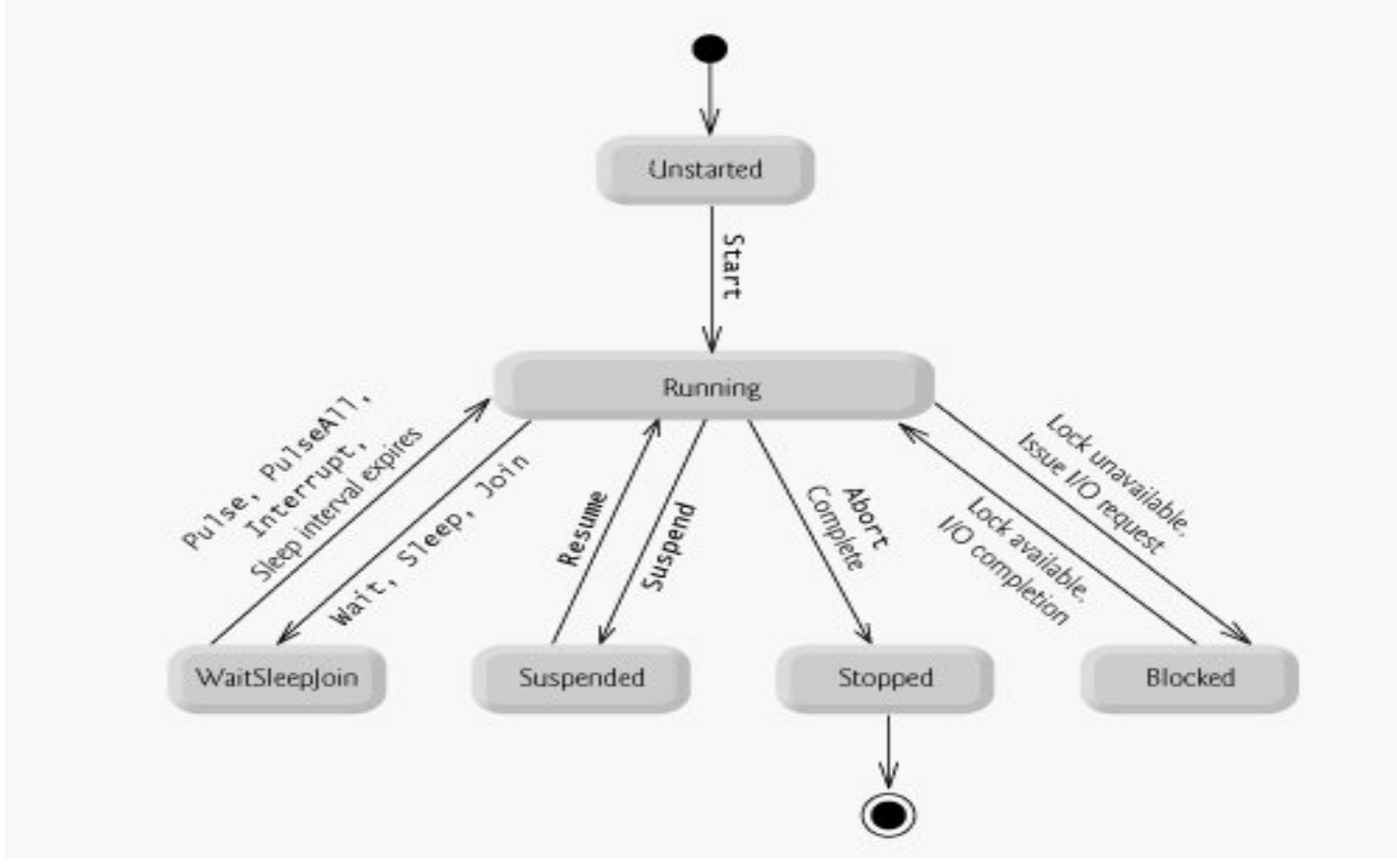
A different name for threads is subprocess, because it is part of a whole process



Concept of Thread and Time-Slicing

DEMO: ConcurrentExecution

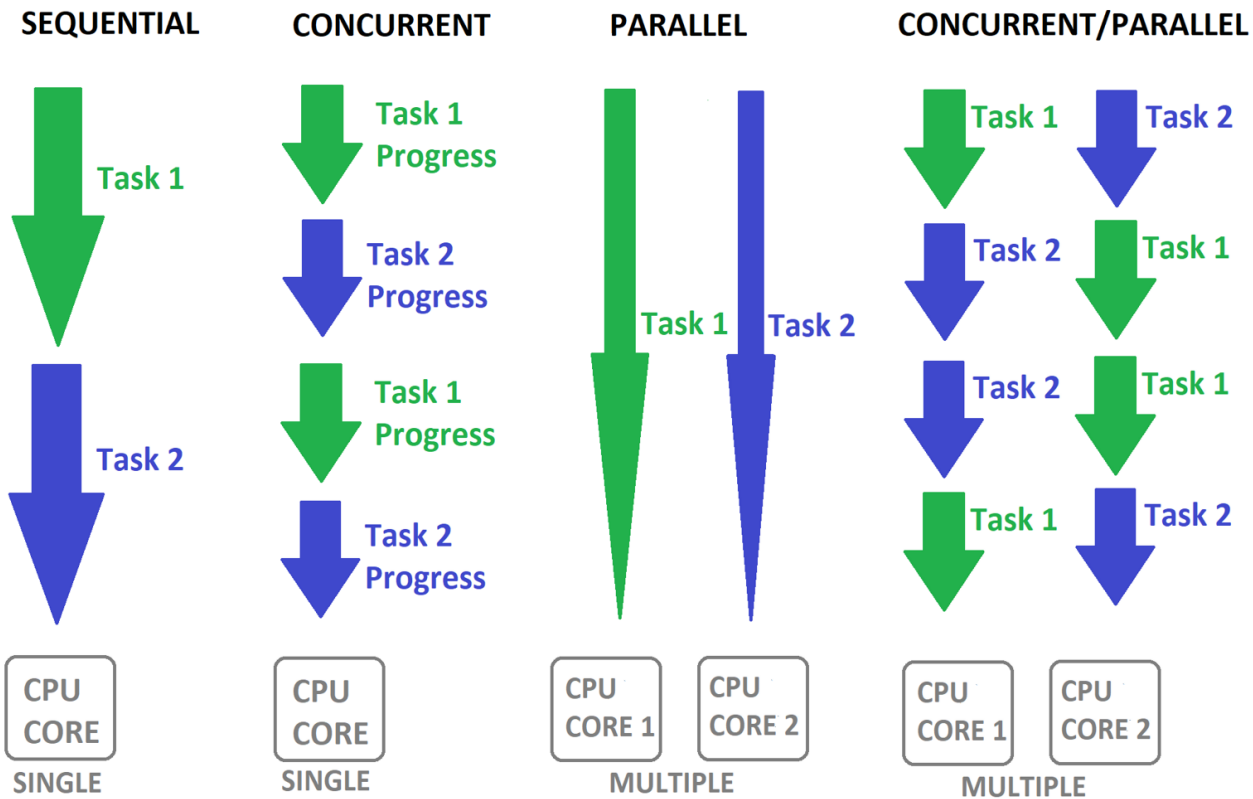
States and Lifetime of Thread



States and Lifetime of Thread

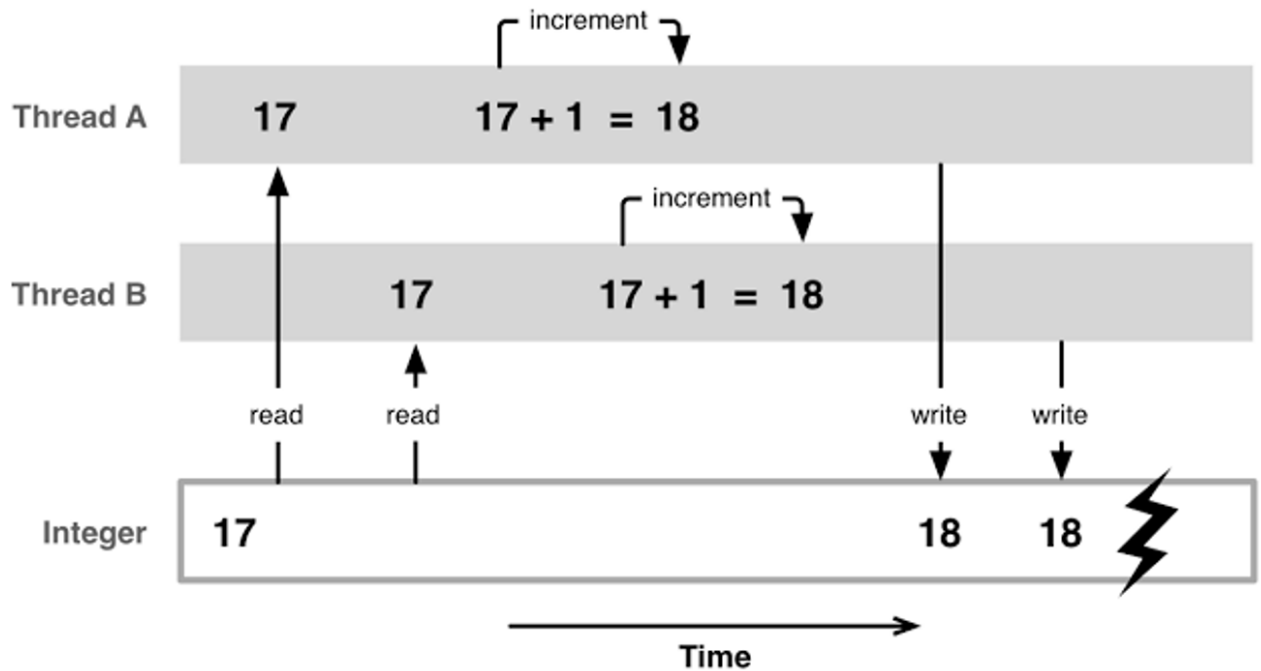
DEMO: ThreadStates

Types of Process/Thread Execution



Race Conditions

A race condition occurs when two or more threads can access shared data and they try to change it at the same time.



DEMO: RaceCondition

Synchronization of Threads

Mutual Exclusion: Prevents simultaneous access to a shared resource. Used in concurrent programming with a critical section, a piece of code in which processes or threads access a shared resource.

Sync Object	Purpose	Cross-process	Best Use Cases	Overhead
lock, Monitor	Ensures just one thread can access a resource (or section of code) at a time	-	Access a shared variable	20 ns
			With Monitor more control	
Mutex		Yes	Interprocess sync	1000 ns
Semaphore	Ensures not more than a specified number of concurrent threads can access a resource	Yes	Limited capacity, number of db connections	1000 ns
SemaphoreSlim (since v.4.0)		-		200 ns

DEMO: Synchronization

Background and Foreground Threads

Foreground Thread

- Keep running event after the application exits.
- Ability to prevent the application from terminating.
- The CLR will not shut down the application until all Foreground Threads have stopped.

Background Thread

- Will quit if our main application quits. In short, if our main application quits, the background thread will also quit.
- Are views by the CLR and if all Foreground Threads have terminated, any and all Background Threads are automatically stopped when the application quits.

Background and Foreground Threads

DEMO: BackgroundVsForegroundThreads

Multithreading and Asynchronous Programming

Multithreading

Maximize the multi-core processors.

In C#, the System.Threading namespace.

Not uses ThreadPool.

CPU-Bound operations.

Consumes more memory resources.

Explicit thread creation

```
public static void Main(string[] args)
{
    new Thread(SomethingToDo).Start();
    Console.WriteLine("Main method continues whil
}

public static void SomethingToDo()
{
    Thread.Sleep(1000);
}
```

Asynchronous Programming

Non-blocking approach.

In C#, the System.Threading.Tasks namespace.

Uses ThreadPool.

CPU and IO-Bound operations.

Lighter, reuses threads.

Use of async/await

```
public static async Task Main(string[] args)
{
    await SomethingToDoAsync();
    Console.WriteLine("Main method continues whil
}

public static async Task SomethingToDoAsync()
{
    await Task.Delay(1000);
}
```

DEMO: SyncMultithreadingVsAsyncProgramming

Conclusions

Why Task? Why not Thread?

Here are the reasons for using tasks instead of threads in multithreaded application:

- **Use of system resources more efficiently and extensively:** Tasks are queued to the ThreadPool behind the scenes, which has been updated with algorithms that determine and adjust the number of threads, as well as load balancing to improve performance.
- **More programmatic control than threads or work items:** TPL supports waiting, cancellation, continuations, robust exception handling, and so on.

For writing multi-threaded, asynchronous, and parallel programming in .NET, TPL is the preferred API.

Alternatives to Threads

- Coroutines, C# 8.0 and above with async and yield and also in NuGet Package/Unity Game Platform
- Consumer-Producer and Message Passing Pattern like System.Threading.Channels (since .NET Core 2.1) or Actor Model (Akka .NET)

Questions

?

End of Session

THANK YOU FOR YOUR ATTENTION

Email: diego.correa@mathema.de

Repo with Slides + Demos: https://github.com/MATHEMA-GmbH/MultithreadingAsyncProgramming

Mathema GmbH:
Schillerstr. 14
90409 Nürnberg
Tel: +49 (0)911 376745-0
www.mathema.de