eleks

Multithreading/Multitasking. Task Parallel Library. Patterns.

by Oleksandr Kravchuk, JR .NET Developer

What is the Multithreading?

An ability that allows you to run several sections of code simultaneously.

Or pretend like. //For 1 CPU core

- So, why does modern OS supports threads?

Because with this approach "RESET" button is pressed less often

On an Operating System level

Process

Executing instance of a program. Virtual memory and no direct communication. Threads container

Insides:

- PID
- Memory (Code and Data, Stack, Heap, Shared Memory...)
- File Descriptors
- Registers
- Kernel State (Process State, Priority, Statistics)

Thread

Basic unit to which the operating system allocates processor time. Executes within the context of a process and shares the same resources allotted to the process by the kernel.

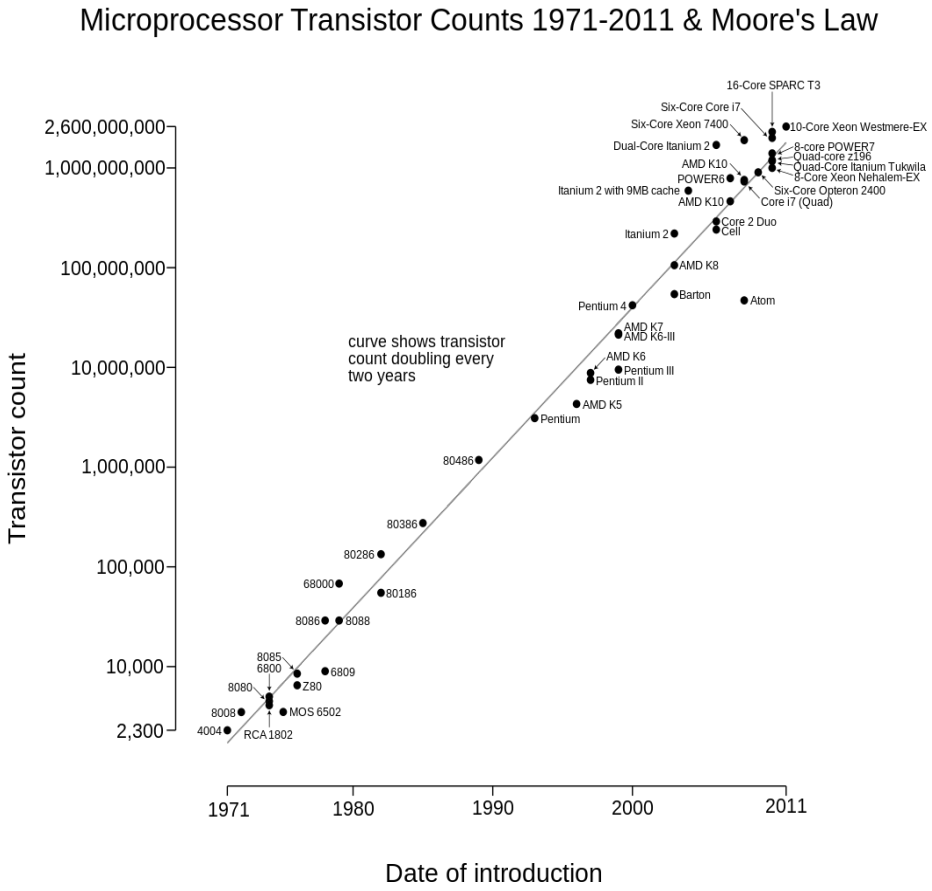
Insides:

- Thread Kernel Object
- Thread Environment Block (TEB)
- Stacks (User-mode and Kernel-mode)

Thread in numbers

- Kernel State (Kernel Object)
  - 700 bytes for x86
  - 1240 bytes for x64
- Thread environment block
  - 1 memory page (4 Kb)
- User-mode stack
  - 1+ Mb
- Kernel-mode stack
  - 12 Kb for x86
  - 24 Kb for x64

Note: Also, whenever a thread is created in a process, all unmanaged DLLs loaded in that process have their `DllMain` method called. Similarly, whenever a thread dies.



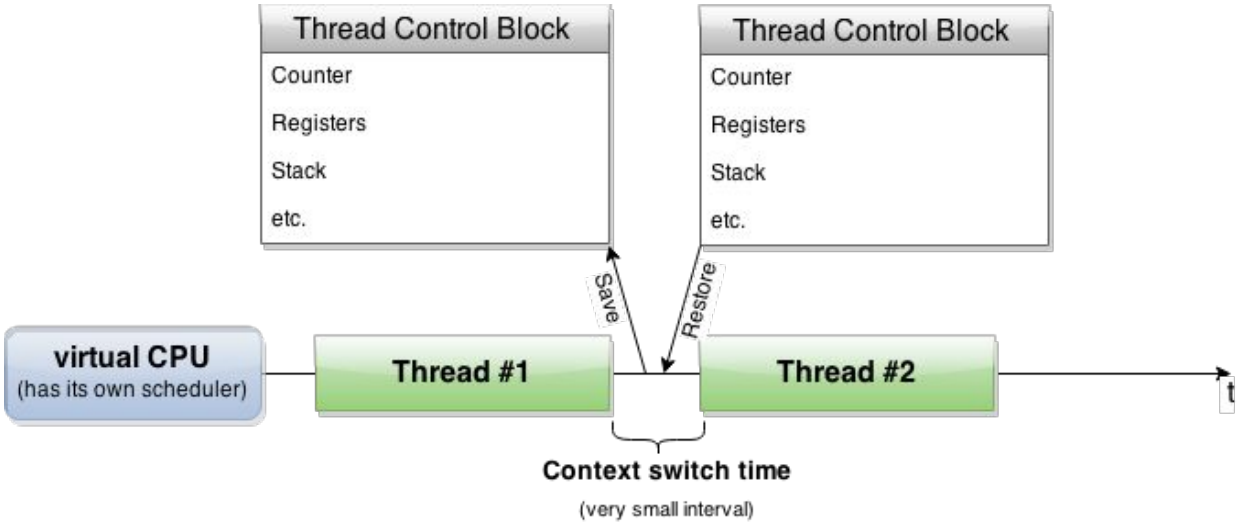
Hardware trends

CPU development:

- Single-core
- Multi-socket motherboards
- Single-core with Hyper-threading
- Multi-core
- Multi-core with Hyper-threading

Context switches

- Kernel-level scheduler responsibility
- Schedule applies to threads, not to processes
- Relies on the priority (process priority + thread priority)



Process and Thread Priority Relations

Relative Thread Priority	Process priority Class					
	Idle	Below Normal	Normal	Above Normal	High	Real-time
Time-Critical	15	15	15	15	15	31
Highest	6	8	10	10	12	26
Above Normal	5	7	9	9	11	25
Normal	4	6	8	8	10	24
Below Normal	3	5	7	7	9	23
Lowest	2	4	6	6	8	22
Idle	1	1	1	1	1	16

Where should I use Threads?

- Client-side GUI applications where responsiveness is important.
- Client-side and server-side applications where non-sequentially execution is possible. For performance improvements

Thread usage example.

Briefly about Thread class

- Return type is void
- Constructors:

- `Thread(ThreadStart)`
- `Thread(ParameterizedThreadStart)`
- `Thread(ThreadStart, Int32)`
- `Thread(ParameterizedThreadStart, Int32)`

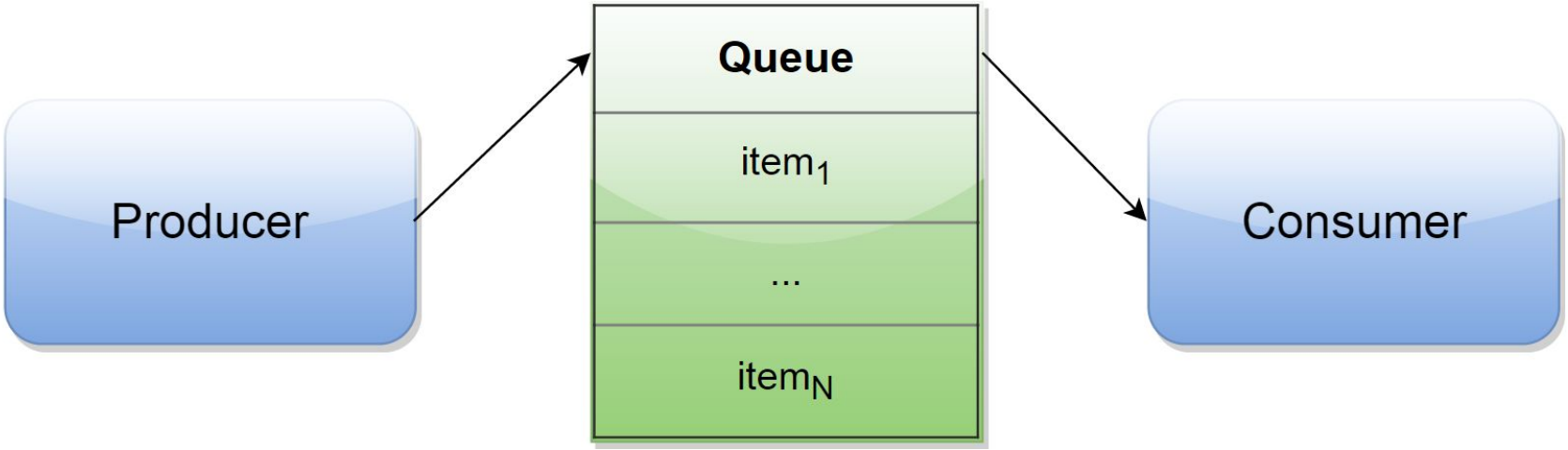
, where ThreadStart and ParameterizedThreadStart are the delegates, lambdas, closures, Action<T>, Func<T>, etc. Also, you may limit thread stack size By passing second parameter.

- Start() method to run the thread
- Use IsAlive property to wait for the thread start
- Join() method to wait till thread ends
- Use closures to simplify value return

- Set thread IsBackground property to true for immediately suspension when parent foreground thread ends
- Exceptions can be caught only on the same thread

Producer/Consumer Pattern

- BlockingCollection<T> as queue
- Variable number of producer/consumer threads

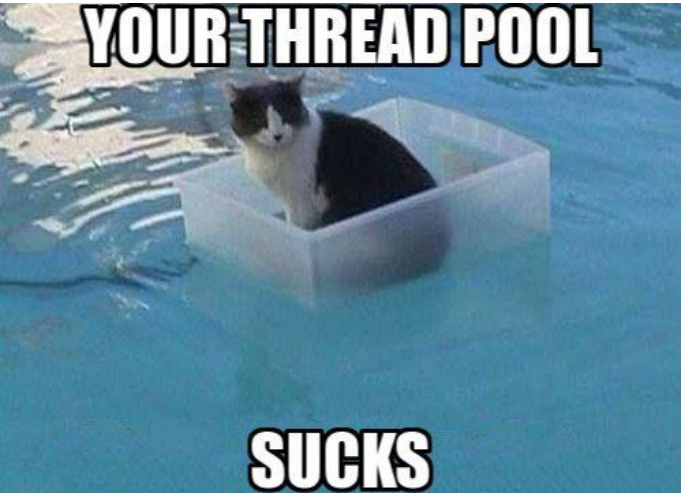


P/C Pattern implementation

```
public class Producer : IDisposable {
    private volatile bool _isRunning;
    private Thread _commandGetThread;
    private object _commandGetterLocker = new object();
    private int _sleepInterval;
    private Consumer _executor;

    public Producer(Consumer executor, int sleepInterval) {
        … //Set defaults
        _isRunning = true;
        _commandGetThread = new Thread(CommandRequestSend);
        _commandGetThread.Start();
    }

    private void CommandRequestSend() {
        while (_isRunning) {
            lock (_commandGetterLocker) {
                … //GetCommands code goes here
                _executor.EnqueueCommands(webCommands);
            }
            Thread.Sleep(_sleepInterval);
        }
    }

    public void Dispose() { … } //use Join() instead of Abort()
}
```

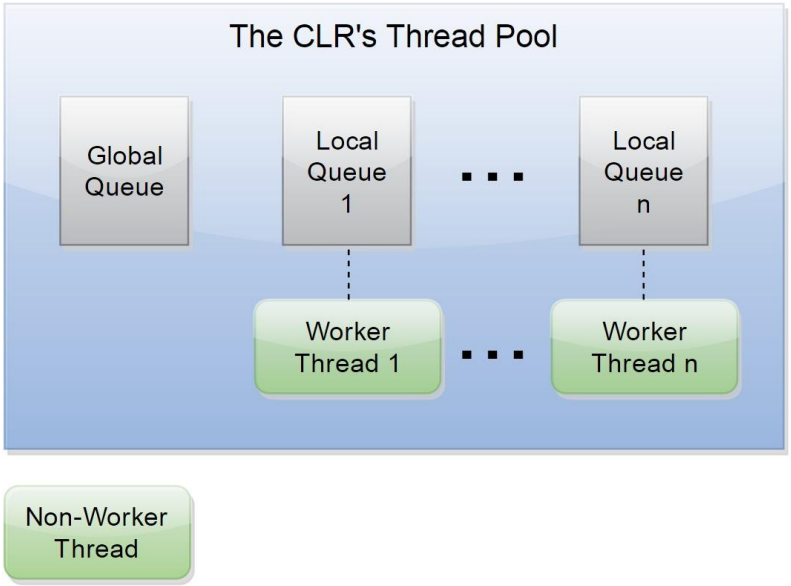
```
public class Consumer : IDisposable {
    private volatile bool _isRunning;
    private object locker = new object();
    private Thread[] executants;
    private ICommandRepository _commandsRepo = new CommandListRepository();

    public Consumer(int executorsCount) {
        _isRunning = true;
        executants = new Thread[executorsCount];
        for (int i = 0; i < executorsCount; i++)
            (executants[i] = new Thread(Execute)).Start();
    }

    public void EnqueueTask(List<BLCommand> commands) {
        lock (locker) {
            _commandsRepo.AddCommands(commands);
            Monitor.PulseAll(locker);
        }
    }

    void Execute() {
        while (_isRunning) {
            lock (locker) {
                while (_commandsRepo.IsEmpty()) Monitor.Wait(locker);
                commandClient = _commandsRepo.GetCommand();
            }
            if (commandClient == null) return;

            … //Execute Command Code (better wrap with try-catch)
        }
    }

    public void Dispose() { … } //enque null in each thread and join
}
```

CLR ThreadPool



- Class ThreadPool was introduced in .Net Framework 3.5. Later, Task approach will use it in 4.0 version
- ThreadPool works on CLR level. It has highly intelligent algorithm for thread management.
- Only busy threads in pool
- To perform asynchronous operation: just call

How the Thread Pool Manages Its Threads?



- What is ideal thread number?
- How queues are scheduled?
- What is Work-Stealing?
- How CLR manages thread number?

Thread Pool usage example.

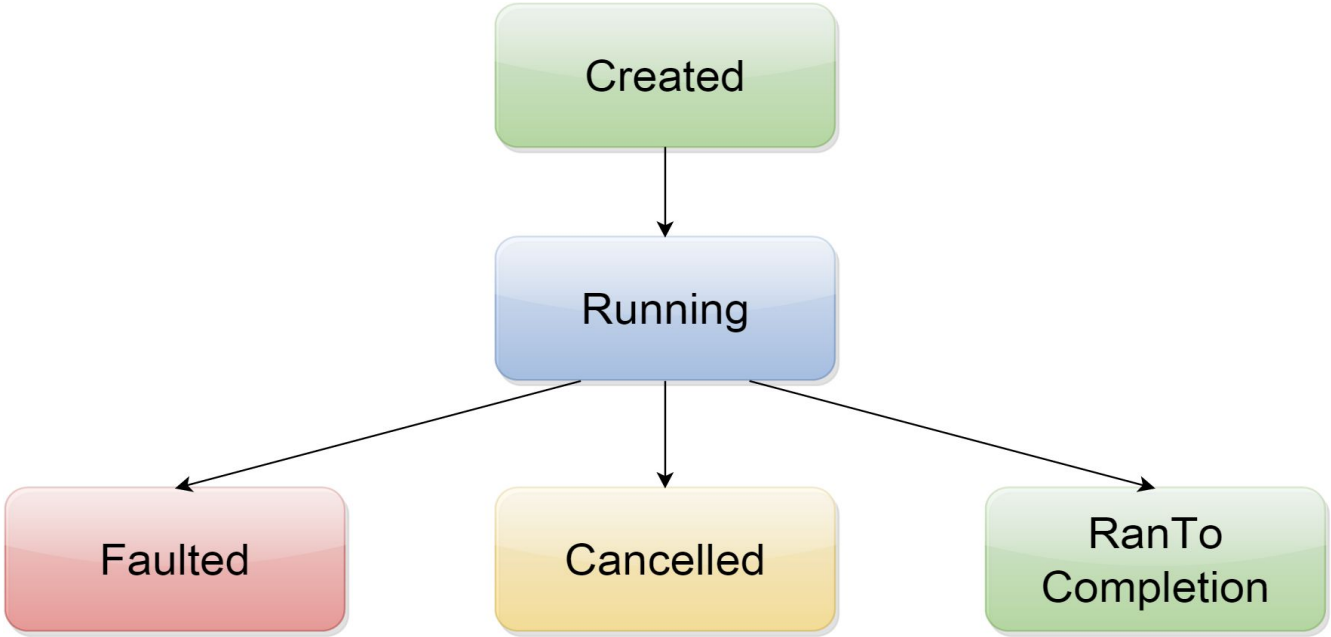
Tasks concept

- Return value from asynchronous operation. Just call task.Result
- You know, when operation completes
- Task class for void and Task<T> generic for T object return
- No-headache with exception handling. Throws AggregateException with inner exceptions tree that corresponds to Tasks tree
- Task start does not guarantee execution in separate thread!

```
ThreadPool.QueueUserWorkItem(SomeLongTermFunction);
```

```
var task = new Task(SomeLongTermFunction);
task.Start();
```

Tasks states



*Also, task can be in waiting (for activation, to run, for children's completion) states

Waiting

- task.Wait() instead of while(!task.IsCompleted)
- Task.WaitAny() for response processing with best perform
- Task.WaitAll() if you need all results

Cancelling

1. Create CancellationTokenSource object and pass its Token property to task constructor
2. Start task and call Cancel() method on CancellationTokenSource object
3. Task will stop and throw AggregateException

Continuations

In order to write scalable software, you must not have your threads block.
Calling task.Wait() will pause current thread until Result property became available

Its better for performance to start next task immediately after previous.
For this case, there are .ContinueWith() extension for task.

Usage sample:

```
var task = new Task(SomeLongTermFunction, cancelToken.Token);
task.ContinueWith(parentTask => AnotherLongTermFunction(),
    TaskContinuationOptions.NotOnFaulted);
task.Start();
```

Tasks are very flexible

Factories

To create a bunch of tasks that return void, then you will construct a TaskFactory.
If you want to create a bunch of tasks that have a specific return type, then you will construct a TaskFactory<TResult>

```
var factory =
   new TaskFactory<int>(TaskScheduler
      .FromCurrentSynchronizationContext());
factory.StartNew(() => GetFibonacciNumber(1));
factory.StartNew(() => GetFibonacciNumber(2));
factory.StartNew(() => GetFibonacciNumber(3));
```

Schedulers

TaskScheduler object is responsible for executing scheduled tasks and also exposes task information to the Visual Studio debugger
The FCL ships with two TaskScheduler-derived types:

- the thread pool task scheduler
- synchronization context task scheduler.

By default, all applications use the thread pool task scheduler.

The Parallel class

To simplify writing code for parallel execution, there are:

Parallel.For(fromInclusive, toConclusive, index => method(index));

Parallel.ForEach(IEnumerable, item => method(item));

Parallel.Invoke(method0(), method1(), method2()...);

They all have overloaded versions that takes ParallelOption object as parameter.
ParallelOption contains such settings:
- MaxThreadNumbers
- CancellationToken
- TaskScheduler

Tasks interaction in Parallel

Also, there is possibilty in TPL that allows interaction among parallel parts of algorythm.
Use localInit and localFinal parameters.
Code sample:

```
var files = Directory.EnumerateFiles(path, searchPattern, searchOption);
var masterTotal = 0;
var result = Parallel.ForEach<String, int>(
   files,
   () => { return 0; /* Set taskLocalTotal initial value to 0*/ },
   (file, loopState, index, taskLocalTotal) =>
      {
         // body: Invoked once per work item
         // Get this file's size and add it to this task's running total
         var fileLength = 0;
         FileStream fs = null;
         try
         {
             fs = File.OpenRead(file);
             fileLength = (int) fs.Length;
         }
         catch (IOException) { /* Ignore any files we can't access */ }
         finally
         {
            if (fs != null) fs.Dispose();
         }
```

Not every algorithm could be parallel

PLINQ

- Parallel Language Integrated Query – set of extensions that allows parallel processing of ParallelQuery<T> collection.
- To transform IEnumerable<T> into ParallelQuery<T> - just call AsParallel() on it (AsSequential() for vice versa)
- Supports almost the same functionality, as the ordinar LINQ.
- Also, offers some additional ParallelEnumerable methods that you can call to control how the query is processed:
  - `WithCancellation(CancellationToken)`
  - `WithDegreeOfParalelism(Int32)`
  - `WithExecutionMode(ParallelExecutionMode)`
  - `WithMergeOptions(ParallelMergeOption)`

Parallel and PLINQ usage example.

Timers

Allows you to perform a Periodic Compute-Bound Operation.

To many timers in .Net:

1. **System.Threading.Timer**
2. **System.Windows.Forms.Timer**
3. **System.Windows.Threading.DispatcherTimer** (Silverlight and WPF)
4. **Windows.UI.Xaml's DispatcherTimer** (Windows Store Apps)
5. **System.Timers.Timer.** Obsolete class. Wrapper for System.Threading.Timer.

System.Threading.Timer usage example

```
private static Timer s_timer;
public static void Main()
{
   Console.WriteLine("Checking status every 2 seconds");

   // Create the Timer ensuring that it never fires. This ensures
   // that s_timer refers to it BEFORE Status is invoked by a
   // thread pool thread
   s_timer = new Timer(Status, null, Timeout.Infinite,
Timeout.Infinite);

   // Now that s_timer is assigned to, we can let the timer fire
   // knowing that calling Change in Status will not throw a
   // NullReferenceException
   s_timer.Change(0, Timeout.Infinite);
   Console.ReadLine(); // Prevent the process from terminating
}

// This method's signature must match the TimerCallback delegate
private static void Status(Object state)
{
   // This method is executed by a thread pool thread
   Console.WriteLine("In Status at {0}", DateTime.Now);
   Thread.Sleep(1000); // Simulates other work (1 second)

   // Just before returning, have the Timer fire again in 2
seconds
   s_timer.Change(2000, Timeout.Infinite);

   // When this method returns, the thread goes back
   // to the pool and waits for another work item
}
```

Async/Await

- Object should have GetAwaiter() method implemented to be available for await
- Async method without awaits inside will be executed synchronously
- Compiler will create continuations for code after await
- There are a lot of async functions in FCL that can be easily found by suffix “Async”

- Exception can be catched from main thread only if async method is awaited
- Using await with a Task, the first inner exception is thrown instead of an AggregateException
- “await” keyword inside of a catch{} and a finally {} blocks are supported from C# 6.0

Async/Await example.

Asynchronous Programming Patterns

- Asynchronous Programming Model (APM)
- Event-based Asynchronous Pattern (EAP)
- Task-based Asynchronous Pattern (TAP)

APM to TAP conversion:

```
await Task.Factory.FromAsync(
stream.BeginRead, stream.EndRead, null);
```

Inspired by Technology.
Driven by Value.

Have a question? Write to eleksinfo@eleks.com

Find us at eleks.com


Продам гараж: + 38066 123 45 12