

Asynchronní programování, vlákna, atd.

Ing. Michal Radecký, Ph.D.



EVROPSKÁ UNIE
Evropské strukturální a investiční fondy
Operační program Výzkum, vývoj a vzdělávání



MINISTERSTVO ŠKOLSTVÍ,
MLÁDEŽE A TĚLOVÝCHOVY

Vlákná (Threads)

- Vlákno je část kódů (algoritmu), který může být vykonáván nezávisle na dalších.
- Každé vlákno je vykonáváno v rámci procesu aplikace, což definuje jeho běhové izolované prostředí.
 - Single-thread aplikace – v rámci procesu běží pouze jedno exkluzivní vlákno
 - Multi-thread aplikace – v rámci procesu běží více vláken, které si sdílí zdroje (především paměť – sdílená data (shared state data))
- Vlákna jsou vytvářena nepřímo (servisní činnosti – garbage collection, finalization) nebo přímo – programově.
- V případě, že procesor nepodporuje více vláken, nebo jich nepodporuje dostatečné množství, pracují tato vlákna „za sebou“ a v rámci procesu se přepíná jejich běh.
- **System.Threading.Thread**

<https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread?view=net-5.0>

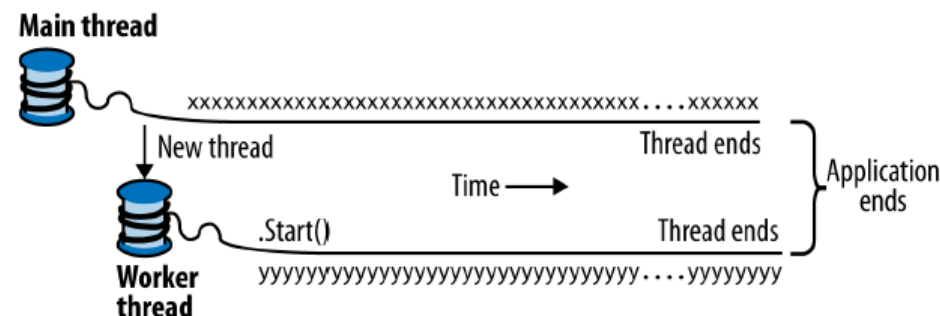
Vytvoření a spuštění vlákna

```
static void Main(string[] args)
{
    Thread t = new Thread(WriteY);
    t.Start();

    new Thread(() =>
    {
        for (int i = 0; i < 1000; i++)
        {
            Console.Write("z");
        }
    }).Start();

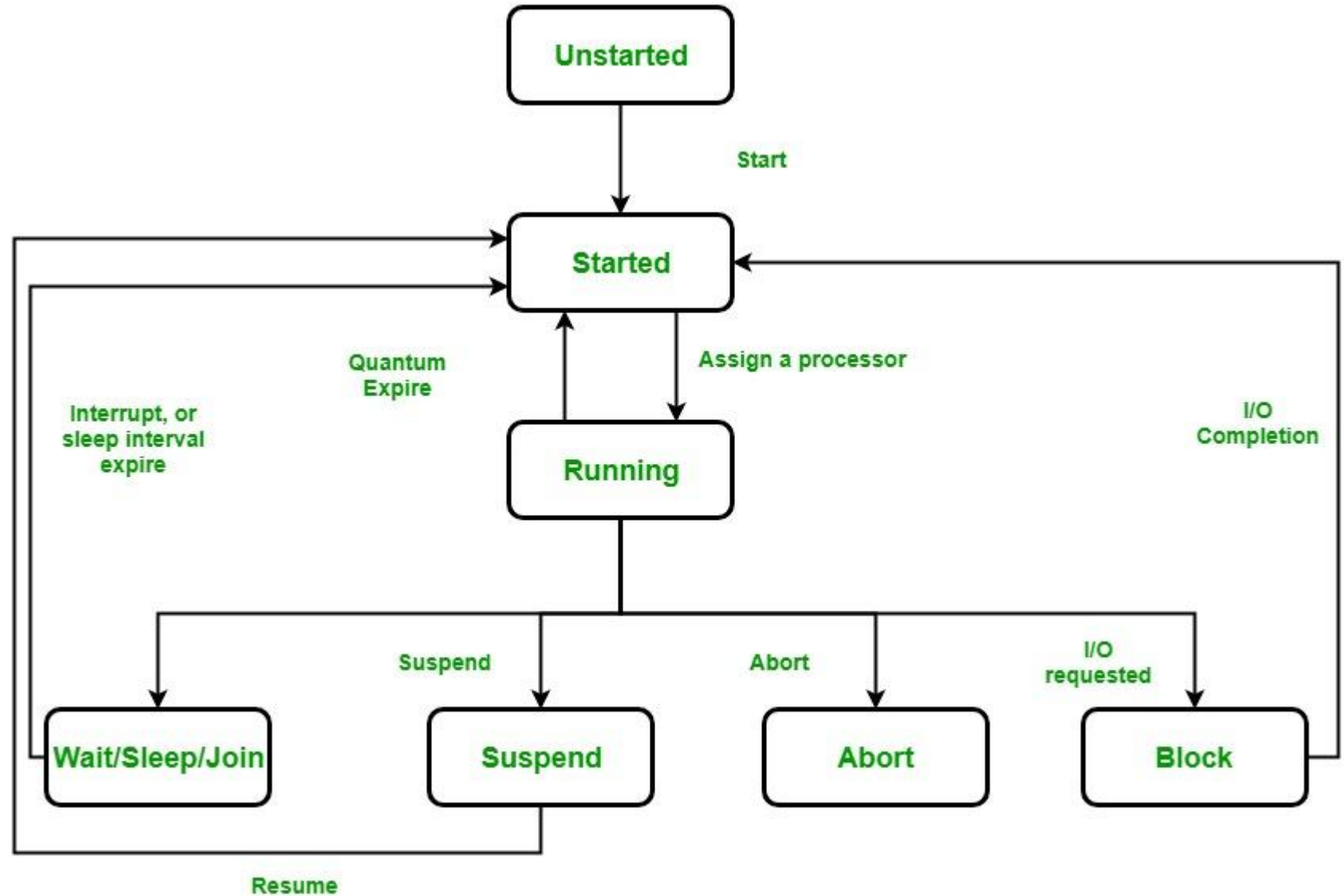
    //do in main thread
    for (int i = 0; i < 1000; i++)
    {
        Console.Write("x");
        Thread.Sleep(10);
    }
}

static void WriteY()
{
    for (int i = 0; i < 1000; i++)
    {
        Console.Write("y");
        Thread.Sleep(100);
    }
}
```



- Využívá se delegát, vždy metoda void, může/nemusí být static
- Delegát bez parametrů – ThreadStart()
- Delegát s parametry
 - ParameterizedThreadStart(object o), Start(o)
 - Násobné parametry s typy pomocí Lambda syntaxe
- Stavové informace o vlákně Thread.CurrentThread
- Delegáti nemají návratovou hodnotu, je třeba řešit pomocí callback metod (delegátů)

Stavy vláken



Ovládání vláken

- `Thread.Sleep(100)` – „uspání“ aktuálního vlákna na definovanou dobu
- `t.Join()` – čeká v aktuálním vláknu na dokončení jiného konkrétního vlákna
- `t.Join(100)` – čeká na ukončení vlákna, nebo na timeout (vrací `true` pokud vlákno opravdu skončilo)
- `t.Interrupt()` – přerušení konkrétního vlákna, jen ve stavu nečinnosti (`WaitSleepJoin`), v kódu vlákna je vyvolána výjimka `ThreadInterruptedException`
- `t.spinWait(10)` – prázdná smyčka s definovaným počtem iterací, vhodné při čekání na zdroje, vlákno je pořád běžící
- `t.Abort()` – ukončení konkrétního vlákna, možné problémy s konzistencí běhu

Zachytávání výjimek

- Nemá smysl zachytávat výjimky způsobené během kódu vlákna vně vlákna
- Výjimky zachytávat v samotném kódu, které vlákno vykonává

```
public static void Main()
{
    new Thread(Go).Start();
}

static void Go()
{
    try
    {
        ...
        throw null; // NullReferenceException
        ...
    }
    catch (Exception ex)
    {
    }
}
```

Vlákna Foreground a Background

- Vytvářená vlákna jsou v základu vždy Foreground (na popředí)
- Dokud nějaké takové vlákno běží, proces/aplikace běží také
- Pokud běží pouze Background vlákna, je aplikace ukončena vč. vláken

```
Thread worker = new Thread(() => Console.ReadLine());
```

```
if (args.Length > 0) worker.IsBackground = true;  
worker.Start();
```

- Možnost nastavování priorit konkrétního vlákna `t.Priority`

Práce s daty mezi vlákny

- Každé vlákno má vlastní „memory stack“, lokální proměnné jsou oddělené
- Sdílená data mohou být jako
 - atributy třídy metody vlákna (pak se sdílí data tohoto objektu)
 - static proměnné (sdíleno mezi všemi vlákny)
 - lokální proměnné využitě v lambda výrazu/anonymní delegát (stejně sdílení jako atributy třídy)
- Problém: výstup může být neurčitý, ovlivněný postupem zpracování jednotlivých vláken, záleží na posloupnosti příkazů, apod.

Práce s daty mezi vlákny

```
class ThreadTest
{
    static bool _done; // Static fields are shared between all threads in the same
                        // application domain.

    static void Main()
    {
        new Thread(Go).Start();
        Go();
    }

    static void Go()
    {
        if (!_done) { _done = true; Console.WriteLine("Done"); }
    }
}
```

Zamykání a Thread Safety

- V případě sdílení dat mezi vlákny může docházet k problémům
- Řešením jsou tzv. zámky, příkaz `lock`
- Zámek definuje část kódu, který může v jeden okamžik zpracovávat pouze jedno vlákno
- Pro zamykání se využívá sdílená proměnná – jakýkoliv referencovatelný objekt
- Kód, který díky zámkům eliminuje neurčitost výstupu/zpracování se označuje jako Thread Safe
- Nesprávné zamykání může vést k tzv. deadlocku

Zamykání a Thread Safety

```
static bool _done;  
static readonly object _locker = new object();  
  
static void Main()  
{  
    new Thread(Go).Start();  
    Go();  
}  
  
static void Go()  
{  
    lock(_locker)  
    {  
        if (!_done) { Console.WriteLine("Done"); _done = true; }  
    }  
}
```

- Obdobně je možné používat třídu Monitor

<https://docs.microsoft.com/en-us/dotnet/api/system.threading.monitor?view=net-5.0>

Vlákná v UI aplikacích

- Aplikace s uživatelským rozhraním (UI) jsou citlivé na správné využívání vláken
- Pokud spouštíme např. časově a výkonově náročný výpočet, dojde k blokaci hlavního vlákna – to je ale zároveň zodpovědné za interakci v rámci UI (zpracování událostí)
- Řešením je spouštět takovéto části kódu v samostatném vlákně a následně aktualizovat UI
- Protože prvky UI „patří“ hlavnímu vlákně, nelze k nim přistupovat přímo, ale je třeba předat požadavek vlákně, které tyto prvky vytvořily, tzv. marshalling
- Přístupy se liší dle technologií
 - WPF – `BeginInvoke` / `Invoke` na objektu `Dispatcher`
 - UWA – `RunAsync` / `Invoke` na objektu `Dispatcher`
 - Windows Forms – `BeginInvoke` / `Invoke` na prvku UI

Vlákna v UI aplikacích

```
partial class MyWindow : Window
{
    public MyWindow()
    {
        InitializeComponent();
        new Thread(Work).Start();
    }
    void Work()
    {
        Thread.Sleep(5000); // Simulate time-consuming task
        UpdateMessage("The answer");
    }
    void UpdateMessage(string message)
    {
        Action action = () => txtMessage.Text = message;
        Dispatcher.BeginInvoke(action);
    }
}
```

ThreadPool

- Koncept spravovaný CLR (Common Language Runtime) umožňující efektivnější práci s vlákny – snížení režie, recyklace předpřipravených vláken
- Vhodné pro „paralelní“ zpracování jednoduchých krátkodobých úloh (do 100-200ms)
- Vlákna v ThreadPoolu jsou vždy typu Background
- Blokována vlákna snižují výkonnost ThreadPoolu
- ThreadPool implicitně využívají
 - WCF, Remoting, ASP.NET, ASMX Web Services application servers
 - System.Timers.Timer, System.Threading.Timer
 - paralelní LINQ/PLINQ, třída Parallel, concurrent collections, atd.
 - třída BackgroundWorker
 - asynchronní delegáti
- **System.Threading.Tasks.Task**
<https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task?view=net-5.0>

Thread vs. Task

- **Thread** je low-level koncept pro realizaci „paralelního“ běhu části kódu
- Programátor má větší kontrolu nad samotným kódem, může explicitně ovlivňovat chování vláken
- Nelze jednoduše „počkat“ na výsledek a dále jej použít (sdílená proměnná, výjimky)
- Nelze jednoduše definovat „workflow“ na základě ukončení vláken

- **Task** je abstrakce běhu části kódu založená na konceptu „příslibu výsledku v budoucnu“
- Task nemusí nutně znamenat vytváření nových vláken, ale principiálně jde o uvolnění vlákna pro děláni něčeho jiného (např. čtení z disku) a návrat zpět (různé úlohy mohou běžet na jednom vlákně, např. WPF)
 - řízeno `SynchronizationContext`
- Součást Task Parallel Library
- Základ pro moderní asynchronní programování `Task<T>`
- Menší přímá kontrola nad vykonáváním kódu
- Efektivnější provádění kódu a asynchronního volání
- Vytváření vláken standardně v rámci konceptu `ThreadPool`
- Snadná implementace s využitím `Action` delegáta

```
Task.Run(() => Console.WriteLine("Ahoj"));
```

Práce s třídou Task

```
Task task = Task.Run(() =>
{
    Thread.Sleep(2000);
    Console.WriteLine("Foo");
});
```

```
Console.WriteLine(task.IsCompleted); // False
task.Wait(); // Blocks until task is complete
```

```
Task taskA = new Task( () => Console.WriteLine("Hello from taskA.")).Start();
```

- Metoda `Task.Run` vrací objekt `Task`, se kterým je možné dále pracovat – ovládat, sledovat stav, získat výsledek, zachytávat výjimky, atd.
- Je možné vytvářet a pracovat s nově a přímo vytvořeným objektem třídy `Task`
- Pokud je potřeba spustit více dlouhotrvajících úloh, není vhodné standardní vytvoření na `ThreadPoolu`, ale použít metodu `Task.Factory.StartNew` a parametr `TaskCreationOptions.LongRunning`. `Factory` slouží i pro vytváření úloh s vyšší kontrolou programátora.

Práce s výsledkem

```
Task<int> primeNumberTask = Task.Run(() =>  
    Enumerable.Range(2, 3000000).Count(n =>  
        Enumerable.Range(2, (int)Math.Sqrt(n) - 1).All(i => n % i > 0))));
```

```
Console.WriteLine("Task running...");  
Console.WriteLine("The answer is " + primeNumberTask.Result);  
    //Wait after result is available  
    //bad approach, problem with deadlock
```

```
Console.WriteLine("The answer is " + await primeNumberTask);  
    //Asynchronous wait after result is available
```

Navazování úloh (continuations)

```
Task<int> primeNumberTask = Task.Run(() =>
    Enumerable.Range(2, 3000000).Count(n =>
        Enumerable.Range(2, (int)Math.Sqrt(n) - 1).All(i => n % i > 0)));

var awaiter = primeNumberTask.GetAwaiter();
awaiter.OnCompleted(() =>
{
    int result = awaiter.GetResult();
    Console.WriteLine(result); // Writes result
});

primeNumberTask.ContinueWith(antecedent =>
{
    int result = antecedent.Result;
    Console.WriteLine(result);
});
```

- Princip umožňující „když skončíš, pokračuj něčím dále“

Ovládání úloh Tasks

- `Task.Delay(100)` – pozastaví zpracování kódu úlohy – asynchronní analogie k `Thread.Sleep(100)`
- `t.Wait()` – „čeká“, až se daná úloha dokončí
- `Task.WaitAll(task,...)` – „čeká“, až se všechny specifikované úlohy dokončí
- `Task.WaitAny(task,...)` – „čeká“, až je alespoň jeden ze specifikovaných ploch dokončený
- `Task.WhenAll(task,...)` – vrací nový Task, který bude dokončen, když budou dokončeny všechny specifikované úlohy
- `Task.WhenAny(task,...)` – vrací nový Task, který bude ukončen, když bude dokončen alespoň jedna specifikovaná úloha

Zachytávání výjimek

- Oproti vláknům se výjimky propagují mimo tělo „vlákna“
- Výjimka se propaguje na úroveň příkazu `Wait` nebo zpracování výsledku `Result`

```
Task task = Task.Run(() => { throw null; });
try
{
    task.Wait();
}
catch (AggregateException aex)
{
    if (aex.InnerException is NullReferenceException)
        Console.WriteLine("Null!");
    else
        throw;
}
```

Asynchronní programování

- Synchronní – kód je spuštěn před návratem na místo volání, blokuující
- Asynchronní – kód je spuštěn po návratu na místo volání, neblokuující
- Asynchronní programování se snaží implementovat déle trvající části kódu přímo asynchronně – vyšší efektivita, jednoduchost. Kdežto u synchronního programování se implementuje standardně a následně se využívají asynchronní principy běhu (Thread, Task).
- Typické asynchronní koncepty
 - `Thread.Start`
 - `Task.Run`
 - metody jakožto pokračování zpracování úloh
- Realizace pomocí `async` a `await` (od C# 5)

await

- Klíčové slovo (operátor) zjednodušující implementaci navazující úlohy (continuation)

```
var result = await expression;  
statement(s);
```

```
var awaiter = expression.GetAwaiter();  
awaiter.OnCompleted (() =>  
{  
    var result = awaiter.GetResult();  
    statement(s);  
});
```

- Co realizuje
 - Zajistí čekání na dokončení asynchronní operace – **asynchronní čekání**
 - Zajistí zpracování návratové hodnoty v rámci objektu Task
 - Pokud Task obsahuje výjimku, znovu ji vyhodí

async

- Klíčové slovo (deklarace metody), které překladači říká, že metoda obsahuje alespoň jedno asynchronní volání
- Může být aplikováno pouze na metody (lambda výrazy), které vrací `void` nebo `Task`, `Task<T>`
- Překladač zajistí zabalení výsledku a označení objektu `Task` jako dokončený až po ukončení dané metody
- Možnost čekat na výsledek dvěma způsoby
 - asynchronně (`await`)
 - synchronně (`Task.Result`/`Task.Wait()`) – problém s deadlockem (s ohledem na to, jaké vlákno se blokuje, např. WPF)
- V praxi by se neměl kombinovat asynchronní a synchronizační přístup (`Wait` – blokuje), s ohledem na provoz na stejných vláknech, apod.

Asynchronní programování - tutoriál

- How to use Async/Await/Task in C#
<https://youtu.be/3GhKdDCvtKE>
- C# Async/Await/Task Explained (Deep Dive)
<https://youtu.be/il9gl8MH17s>

Další zdroje

- <https://profinit.eu/blog/ponorme-se-do-async-await-v-jazyce-c-1-cast/>
- <https://profinit.eu/blog/ponorme-se-do-async-await-v-jazyce-c-2-cast/>
- <https://profinit.eu/blog/ponorme-se-do-async-await-v-jazyce-c-3-cast/>

Zdroje

- <https://docs.microsoft.com/cs-cz/dotnet/>
- <https://devblogs.microsoft.com/>
- <https://www.codeguru.com/csharp/>
- YAMIKANI FUKIZI, Kenneth, Jason DE OLIVEIRA a Michel BRUCHET. *Learn ASP.NET Core 3: Develop modern web applications*. Second edition. Packt Publishing, 2019. ISBN 978-1789610130.
- ALBAHARI, Joseph. *C# 10.0 in a Nutshell: The Definitive Reference*. O'Reilly Media; 1st edition, 2022. ISBN 978-1098121952.
- ALBAHARI, Joseph. *C# 10 and .NET 6 – Modern Cross-Platform Development: Build apps, websites, and services with ASP.NET Core 6, Blazor, and EF Core 6 using Visual Studio 2022 and Visual Studio Code*. 6th edition. Packt Publishing, 2021. ISBN 978-1801077361.