10- Paralelní programování

Asynchronní programování

- Asynchronní volání neblokují chod volajícího
- Cílem je responzivita aplikace (například aby UI nezamrzlo při stahování dat z databáze)

Paralelní programování

- Rozdělení práce do několika vláken
- Cílem je výkon (rychlost) aplikace
- Např. Quicksort

Thread

- Třída Thread reprezentuje vlákno (tok)
- Aplikace může běžet ve více paralelních vláknech
- Start() – spuštění vlákna
- Join() – čekání na dokončení vlákna
- Sleep(milisekundy) – pozastavení běhu vlákna
- Abort() – zabití vlákna

Task

- Spouštíme asynchronní operaci a čekáme na její výsledek
- V C# jde o třídu
- Vytvoření instance této třídy vytvoří novou úlohu
- Třída je generická
- Obsahuje informace o úloze
- Úloh může běžet více naráz
  - `Task task = new Task(() => { });` vytvoří úlohu
  - `task.Start();` spustí úlohu
  - `Task.Run(() => { });` vytvoří a spustí úlohu
  - `Task.Factory.StartNew(() => { });` je podobné jako `Task.Run(() => { });`, pouze drobné rozdíly
  - `Task.WaitAll(task);` čeká na dokončení úloh(y), blokuje aktuální vlákno
  - `Parallel.Invoke(() => {úloha1}, () => {úloha2}, ...);` vytvoří a spustí úlohy paralelně a počká na jejich dokončení
  - `Task.WhenAll(task);` reprezentuje úlohu, která se dokončí, až když skončí všechny úlohy na vstupním parametru

Parallel.For

- Paralelní průchody cyklem
- `Parallel.For(odInkluzivni, doExkluzivni, (iteracniPromenna) => { });`
- `Parallel.ForEach(kolekce, (iteracniPromenna) => { });`

Synchronizace vláken

- Synchronizace vláken – koordinovaný přístup ke sdíleným prostředkům
- Při zpracování dat vláknem by k těmto datům neměla mít ostatní vlákna přístup
    - Problém nastává, když vlákno s daty provádí neatomickou operaci a jiné vlákno začne s těmito daty také pracovat
    - (data ještě nejsou plně zpracovaná a připravená k další akci)

```
static void Main(string[] args)
{
    for (int i = 0; i < 3; i++) new Thread(Method).Start();
}

static object baton = new object();

static void Method()
{
    var id = Environment.CurrentManagedThreadId;
    WriteLine(id + " se snaží dostat do chráněné sekce");
    lock(baton) {
        WriteLine(id + " se nachází ve chráněné sekci");
        Thread.Sleep(10);
        WriteLine(id + " opouští chráněnou sekci");
    }
    Thread.Sleep(10);
    WriteLine(id + " je na konci metody");
}

/* Výstup může vypadat:

4 se snaží dostat do chráněné sekce
5 se snaží dostat do chráněné sekce
6 se snaží dostat do chráněné sekce
4 se nachází ve chráněné sekci
4 opouští chráněnou sekci
5 se nachází ve chráněné sekci
4 je na konci metody
5 opouští chráněnou sekci
6 se nachází ve chráněné sekci
5 je na konci metody
6 opouští chráněnou sekci
6 je na konci metody

*/
```

Async, await

- Klíčová slova usnadňující vytváření asynchronního kódu
- async označí metodu jako asynchronní

- aby byla metoda asynchronní, musí zároveň v jejím těle obsahovat await (vyčkání na dokončení metody)
- Asynchronní metody vrací Task<T>

Příklady

- Sečtení všech čísel v poli
- Pomocí Thread

```
static void Main(string[] args)
{
    CPUs = Environment.ProcessorCount; // Vrací počet logických procesorů, které mohou být využity CLR
    portionSize = numbers.Length / CPUs; // Rozdělení pole pro jednotlivá vlákna
    sumPortions = new long[CPUs]; // Pole pro (mezi)výsledky
    long sum = 0;

    // Vytvoření pole vláken a jejich spuštění
    Thread[] threads = new Thread[CPUs];
    for (int i = 0; i < CPUs; i++) {
        threads[i] = new Thread(SumPortion);
        threads[i].Start(i);
    }
    // Počkat, až všechna vlákna skončí
    // .Join() blokuje volající vlákno, dokud dané vlákno neskončí
    for (int i = 0; i < CPUs; i++) threads[i].Join();
    // Sečíst (mezi)výsledky jednotlivých vláken
    for (int i = 0; i < CPUs; i++) sum += sumPortions[i];

    WriteLine(sum);
}

static int CPUs;
static int portionSize;
static long[] sumPortions;

static void SumPortion(object _portionNumber)
{
    int portionNumber = (int)_portionNumber; // Explicitní přetypování
    var start = portionSize * portionNumber;
    var end = (portionNumber == CPUs - 1) ? (numbers.Length) : (portionSize * portionNumber + portionSize);
    // $"Vlákno {Environment.CurrentManagedThreadId}" pracuje od {start} do {end - 1}
    long sum = 0;
    for (int i = start; i < end; i++) sum += numbers[i];
    sumPortions[portionNumber] = sum;
}
```

- Pomocí Task

```
static void Main(string[] args)
{
    int CPUs = Environment.ProcessorCount;
    int portionSize = numbers.Length / CPUs;
    long[] sumPortions = new long[CPUs];
    long sum = 0;

    Task[] tasks = new Task[CPUs];
    for (int i = 0; i < CPUs; i++) {
        var tid = i; // Pro jistotu
        tasks[tid] = Task.Run(() => {

            var start = portionSize * tid;
            var end = (tid == CPUs - 1) ? (numbers.Length) : (portionSize * tid + portionSize);

            long sum = 0;
            for (int j = start; j < end; j++) sum += numbers[j];
            sumPortions[tid] = sum;

        });
    }

    Task.WaitAll(tasks); // Počká, až se dokončí všechny Tasky v poli

    for (int i = 0; i < CPUs; i++) sum += sumPortions[i]; // Sečíst (mezi)výsledky

    WriteLine(sum);
}
```

- Pomocí Parallel.For

```
static void Main(string[] args)
{
    int CPUs = Environment.ProcessorCount;
    int portionSize = numbers.Length / CPUs;
    long[] sumPortions = new long[CPUs];
    long sum = 0;

    Parallel.For(0, CPUs, (i) => {

        var tid = i;
        var start = portionSize * tid;
        var end = (tid == CPUs - 1) ? (numbers.Length) : (portionSize * tid + portionSize);

        long sum = 0;
        for (int j = start; j < end; j++) sum += numbers[j];
        sumPortions[tid] = sum;

    });

    for (int i = 0; i < CPUs; i++) sum += sumPortions[i];

    WriteLine(sum);
}
```