Paralelní programování

Asynchronní a paralelní programování. Thread, Task, zámek, chráněná sekce, použití paralelismu, Async/await.

Asynchronní a paralelní programování

Kód se standardně vykonává synchronně. To zjednodušeně znamená řádek po řádku. Aplikace, respektive **proces** navíc standardně běží v jednom (hlavním) vlákně. Zde vzniká problém, pokud je například volaná metoda příliš složitá. Její vykonávání tak může zaseknout celé vlákno a uživatel neví, co se s programem děje.

Tento problém lze řešit buď asynchronním vykonáváním či paralelizací.

Kód se vykonává asynchronně v asynchronních podprogramech. Jedná se vlastně o **úlohy** (task). Ty kód vykonávají synchronně v rámci sebe samé (pokud nespouští další úlohy). Vně (z místa, kde byla daná úloha spuštěna) se pak kód vykonává dál nehledě na to, zda se úloha dokončila, či nikoli.

Paralelizovaný kód běží ve vlastním **vlákně** (thread). Vlákna mohou běžet současně – pokud jsou skutečná vlákna, respektive jádra procesoru k dispozici. Pokud nejsou – operační systém mezi vlákny přepíná. Toto přepínání se dá částečně řídit prioritou jednotlivých vláken.

Thread

Thread je v jazyce C Sharp třída reprezentující vlákno. Vytvořením instance třídy vytvoříme nové vlákno v rámci procesu.

Základními vlastnostmi třídy jsou:

- IsAlive – Boolean; určuje, zda vlákno (ne)běží.
- Name – string; usnadňuje ladění, umožňuje vlákno pojmenovat.

Základními metodami třídy jsou:

- Sleep – statická metoda, která uspí aktuálně běžící vlákno
- Join – zablokuje aktuální vlákno, dokud se metoda nedokončí.
- Start – spustí vlákno.
- Abort – zastaví vlákno, čímž v něm vyvolá výjimku.

```
// The ThreadProc method is called when the thread starts.
// It loops ten times, writing to the console and yielding
// the rest of its time slice each time, and then ends.
public static void ThreadProc() {
    for (int i = 0; i < 10; i++) {
        Console.WriteLine("ThreadProc: {0}", i);
        // Yield the rest of the time slice.
        Thread.Sleep(0);
    }
}

public static void Main() {
    Console.WriteLine("Main thread: Start a second thread.");
    // The constructor for the Thread class requires a ThreadStart
    // delegate that represents the method to be executed on the
    // thread.  C# simplifies the creation of this delegate.
    Thread t = new Thread(new ThreadStart(ThreadProc));

    // Start ThreadProc.  Note that on a uniprocessor, the new
    // thread does not get any processor time until the main thread
    // is preempted or yields.  Uncomment the Thread.Sleep that
    // follows t.Start() to see the difference.
    t.Start();
    //Thread.Sleep(0);

    for (int i = 0; i < 4; i++) {
        Console.WriteLine("Main thread: Do some work.");
        Thread.Sleep(0);
    }

    Console.WriteLine("Main thread: Call Join(), to wait until ThreadProc ends.");
    t.Join();
    Console.WriteLine("Main thread: ThreadProc.Join has returned.  Press Enter to e
    Console.ReadLine();
}
```

Task

Task je v jazyce C Sharp třída reprezentující úlohu. Vytvořením instance třídy (pomocí tovární metody Run) vytvoříme a spustíme novou úlohu.

Třída Task je generická. Typ je typem pro výsledek, jež akce prováděná v rámci úlohy vrací.

Instance vytvořená standardně přes konstruktor, je neběžící úlohou. Spustit jí lze metodou Start. Pokud chceme úlohu spustit hned po vytvoření, lze použít zmíněnou metodu Run.

Na ukončení úlohy lze počkat pomocí metody Wait.

I úlohy mohou běžet paralelně. Je to však plně v režii operačního systému. Pokud chceme provádět paralelní akce, lze použít knihovnu Task.Parallel.

Zámek

Problém s vlákny je ten, že mohou přistupovat k datům, která jsou právě zpracovávaná. To může vést k nežádoucím výsledkům, či dokonce pádu aplikace nebo ztrátě dat.

Problém nastává ve chvíli, kdy operační systém přepne vlákno ve chvíli, kdy jiné vlákno začne vykonávat kritický kód, ale nedokončí ho data nejsou plně zpracována a připravena k další akci.

Vznikla tak konstrukce známá jako zámek. Jedná se o konstrukci s parametrem, jímž může být libovolný objekt, kterým danou část kódu zamkneme. Tato část kódu se pak vždy dokončí, pokud jí dané vlákno začne vykonávat.

Tato konstrukce je ve skutečnosti tzv. syntaktickým cukrem. Stejné funkce dosáhneme použitím třídy Monitor (která ve skutečnosti použita je.

Příklad:

```
static int count = 0;
static void ThreadJob1()
{
  for (int i=0; i < 5; i++)
  {
    Monitor.Enter(countLock);
    int tmp = count;
    Console.WriteLine ("Read count={0}", tmp);
    Thread.Sleep(20);
    tmp++;
    Console.WriteLine ("Incremented tmp to {0}", tmp);
    Thread.Sleep(10);
    count = tmp;
    Console.WriteLine ("count={0}", tmp);
    Monitor.Exit(countLock);
    Thread.Sleep(40);
    }
}
```

Použití zámku:

```
static void ThreadJob()
  {
    for (int i=0; i < 5; i++)
    {
      lock (countLock)
      {
        int tmp = count;
        Console.WriteLine ("Read count={0}", tmp);
        Thread.Sleep(20);
        tmp++;
        Console.WriteLine ("Incremented tmp to {0}", tmp);
        Thread.Sleep(10);
        count = tmp;
        Console.WriteLine ("Written count={0}", tmp);
      }
      Thread.Sleep(40);
    }
  }
```

Dále ještě existuje třída Mutex, která umožňuje limitovat počet metod, které mohou k datům přistoupit.

Použití paralelismu

Paralelismus je vhodně použít na výpočty pracující s velkým množstvím vzájemně nezávislých dat. Samozřejmě je k tomu také potřeba disponovat hardwarem, který umožní reálné vykonávání více výpočtů.

Jediný smysl paralelizace na jedno-jádrovém procesoru měli historické osobní počítače, které větším počtem jader nedisponovali, ale operační systém nabízel multitasking.

Dnes jsou však jedno–čipy používány účelově a nemá smysl na nich spouštět operace paralelně. Přepínání vláken je totiž doprovázeno režií. Výstup bychom tak obdrželi, ještě později, než kdybychom prováděli operace sekvenčně.

Naopak pokud disponujeme velkým počtem jader, je režie velmi nízká cena, za počet akcí, které proběhnou najednou.

Async/await

Async a await jsou klíčová slova v jazyce C Sharp usnadňující vytváření asynchronního kódu.

Klíčové slovo async je použito v hlavičce metody, jež má být asynchronní. Uvádí se za modifikátorem přístupu. Aby se metoda skutečně vykonala asynchronně, musí v ní být použito klíčové slovo await. Dobrou konvencí je, aby asynchronní metoda vracela Task. V případě, že nevrací žádná data bude Task neparametrický.

Klíčové slovo await je použito těsně před volání asynchronní metody. Při jeho použití se vyčká na dokončení metody.