Vlákna a paralelní programování

Proces je program běžící v rámci operačního systému, který má vyhrazené systémové prostředky, zejména zásobníkovou paměť (pro lokální proměnné a volání funkcí), procesorový čas (daný prioritou procesu a algoritmem výběru), a datovou paměť pro objekty (haldu).

Vlákno (*Thread*) je na víceúlohovém OS část procesu. Proces se skládá z hlavního vlákna a nepovinně z několika pracovních vláken. Vlákna se chovají jako procesy, ale sdílí datovou paměť a mohou se mezi sebou synchronizovat programově, nejen rozhodnutím OS jako proces. Vláknu lze předat parametr typu `object`, viz kód se dvěma souběžnými (*concurrent*) pracovními vlákny:

```
public static void Main(string[] args)
{
      Thread t1 = new Thread(WriteThread);
      Thread t2 = new Thread(WriteThread);
      t1.Start("+");
      t2.Start("-");
      Console.ReadKey();
}

public static void WriteThread(object o) {
      for(int i=0; i<100; ++i) Console.Write(o);
}
```

Úloha (*Task*) je abstrakce nad vláknem. Vytvoření a provoz vlákna je relativně náročná operace (OS musí vyhradit paměť zásobníku a zajistit přepínání kontextu při střídání vláken na jednom procesoru). Z pohledu programátora se úloha chová jako vlákno, ale OS na pozadí provádí optimalizaci, zejména:

- využití návrhového vzoru **Fond** (*Pool*), kdy se prostředky přiřazené ukončenému vláknu neuvolní, ale recyklují pro budoucí použití jiným vláknem
- vyžití návrhového vzoru **Líná Inicializace** (*Lazy Init*), kdy se vlákno vytvoří pouze tehdy, je-li skutečně zapotřebí

Úlohu můžeme spustit nejsnáze metodou `.Start()` podobně, jako vlákno, nebo statickou metodou `Task.Run()`, viz kód níže. Oba způsoby vyžadují objekt `Action`, který má parametr metodu void bez parametrů, je tedy touto metodou implicitně volatelný. Spuštění metody s parametrem jako úlohy zajistíme nejsnáze lambda výrazem void bez parametrů:

```
public static void Main(string[] args) {
      new Task(()=>WriteTask("+")).Start();
      Task.Run(()=>WriteTask("-"));
      Console.ReadKey(true);
}
static void WriteTask(string s) {
      for(int i=0; i<100; ++i) Console.Write(s);
}
```

Úlohově řízené asynchronní programování (*Task-based Asynchronous Pattern*) je návrhový vzor pro paralelní programování implementovaný v jazyce C# od v.5 klíčovými slovy `async` – `await`:

- `async` modifikátor metody znamená, že bude implicitně (tj. bez nutnosti volání `Task.Run`) vytvořena a spuštěna jako úloha, která poběží paralelně. Pokud asynchronní metoda vrací hodnotu, není tato k dispozici ihned, ale jako příslib (*promise*). Deterministický program však musí mít kontrolu nad tím, kdy se objeví
- `await` se objevuje na pravé straně přiřazení u asynchronní metody, zajišťuje odložení vykonání příkazu přiřazení do návratu asynchronní metody. Návrat asynchronní metody je speciální typ, který se genericky zapisuje jako `Task<T>`, kde `T` je návratová hodnota metody. `await` je možno volat pouze v asynchronní metodě.

```
public static void Main(string[] args) {
      AsyncWriter("+");
      Writer("-");
      Console.ReadKey(true);
}

static async void AsyncWriter(string o) {
      await Task.Delay(N);
      for(int i=0; i<100; ++i) Console.Write(o);
}
static void Writer(string o) {
      for(int i=0; i<100; ++i) Console.Write(o);
}
```

V uvedeném kódu se pustí nejdříve metoda `AsyncWriter`, která je díky `async` spuštěna asynchronně. Její kód se začíná vykonávat synchronně, dokud nenarazí na příkaz `await`. Chování v této chvíli se pak liší v závislosti na `N`:

- `N=0` Nevzniká žádná prodleva, kód se spustí synchronně (nejdřív `+`, poté `-`)
- `N>0` Prodleva způsobí vznik nového vlákna, ve kterém se vykoná zbytek metody, hlavní vlákno mezi tím spustí metodu `Writer` (`+` a `-` se vypisují na střídačku)
- `N>150` (cca) Prodleva je tak velká, že se při čekání stihne vykonat metoda `Writer`. `Console.ReadKey` běží jako blokující vlákno, metoda `AsyncAwait` tedy pravděpodobně poběží v hlavním vlákně, tj. nevytvoří se vlákno navíc, jako v předchozím případě, efekt je stejný jako při volání `Thread.Sleep(150)`

Stavy vlákna

Vlákno je nejčastěji v následujících stavech:

- `Unstarted` – vytvořené, ale dosud nespuštěné
- `Running` – běžící
- `Stopped` – ukončené
- `WaitSleepJoin` – zablokované, což může nastat čekáním (*Wait*) na odemčení zdrojů (konstrukce `lock`), uspáním (*Sleep*) pomocí `Thread.Sleep()` nebo čekáním na dokončení pracovních vláken (volání `thread.Join()` z volajícího vlákna)

Dále je vlákno možné přerušit voláním `thread.Interrupt()`, což v něm vyvolá vyjímku `ThreadInterruptedException` (vlákna mají vlastní zásobník, ošetření této vyjímky je tedy nutno provést ve vlákně, kde nastala).

Nedoporučuje se používat metody `thread.Suspend()` a `thread.Abort()`, které způsobí pozastavení (stav `Suspended`) a násilné ukončení (stav `Aborted`) vlákna: obojí data může zanechat v nekonzistentním stavu.

Asynchronní čtení z konzole

Objekt `Console` se při čtení zamkne, což způsobí „visící konzoli“ (při čtení je nutno stisknout jeden enter navíc). Aby se tak nestalo, veškeré čtení musí běžet asynchronně. Mějme aplikaci, kde uživatel má na zadání hesla 2 sekundy: napojení úlohy je možné metodou `task.Wait()`, která může mít parametr udávající maximální počet milisekund, které je ochotno volající vlákno počkat na dokončení. Pokud to volané vlákno nestihne, `Wait` vrací false:

```
static string pass;
public static void Main(string[] args) {
      Task t = new Task(new Action(getPass));
      t.Start();
      if(t.Wait(2000)) Console.WriteLine(pass);
      else {
            Console.WriteLine("too late");
      }
      // Console.ReadKey(true); způsobí visící konzoli
}

public static async void getPass() {
      pass = Console.ReadLine();
}
```

Vzájemnou synchronizaci úloh lépe zvládá objekt `AutoResetEvent`, který má dvě hlavní metody:

- `WaitOne()` vzdává se vykonávání ve prospěch volaného vlákna, dokud od něj nedostane signál
- `Set()` posílá signál pozastavenému vláknu