**Vlákna, Paralerní programování, Asynchroní metody, Concurrent design patterns**

**Paralelní vs Sekvenčí vs Distribuované programování**

* Programy využívají **většinou jak sekvenční tak paralelní programování** zároveň

**Sekvenční**

* Styl programování na který jsme "nejvíce" zvyklý
* Program běží na jednom hlavním vlákně
* Operace se provádí **sekvenčně** (tzn. jedna za druhou)
* **Další příkaz se neprovádí dokud se zcela nedokončí předešlý!**
* Jednoduchá implementace
  + přemýšlíme sekvenčně
    - jak chceme aby se příkazy vykonávaly za sebou
* Hlavní nevýhodou je **pomalost**
  + v jednu určitou chvíli se vykonává **pouze jedna instrukce**
* Jako řešení bylo představeno **paralelní programování**

**Paralelní**

* **Hlavní myšlenka** **vykonávání více operací naráz**
* Využívá více vláken procesoru na kterých provádí výpočty souběžně
* **Výhoda** -> rychlost díky provádění **více operací zároveň**
* Složitější na implementaci
  + Musíme zohledňovat principy paralelního programování (problém s integritou dat..)
    - Toto řeší concurrent design patterns
  + Problém musíme rozdělit do více podproblémů, které mohou běžet souběžně
* **Pozor!**
  + Pokud procesor nemá tolik fyzických procesorů, kolik v programu vytvoříme vláken - neběží souběžně. OS mezi vlákny velice rychle přepíná tzn. vytváří iluzi, že běží více výpočtů naráz
  + Také vždy neplatí, že pokud vytovříme 4 vlákna, rozdělí se mezi 4 fyzické procesory, většinou tomu tak ale je.
* **Příklady využití** 
  + TCP Klient -> Vytvoříme si vlákna pro naslouchání a odesílání dat
  + TCP Server -> Pro každého klienta si vytvoříme samostatné vlákno
  + Ajax
    - Udělá request na server
    - mezitím webovka stále běží na **hl. vlákně**
    - mezitím co **vedlejší** čeká na odpověď ze serveru s daty
      * aby mohlo upravit obsah stránky

**Distribuované**

* **Více autonomních počítačů, které se tváří jako jeden systém**
* **Hlavní myšlenka** rozdělení problému mezi více výpočetních jednotek
* **Výhoda**+ -> Díky využití více počítačů máme přístup k více paměti a procesorům (tzn. možnost pracovat s těžšími problémy)
* **Nevýhoda-** -> Těžká implementace, musíme si vytvořit komunikační protokol na míru k problému
* Využívá jak sekvenční, tak paralelní postupy
* Nesdílí mezi sebou paměť ! Pouze posílají mezi sebou data !!
* Počítače mezi sebou většinou komunikují pomocí nějáké komunikační sítě
* **Příklad využití**
  + **Řešení časově/paměťově složitého algoritmu (bruteforce..)**

**Proces**

* Proces je něco jako program, celek kódu, který vykonává úrčitou činnost
* **Procesy mezi sebou nesdílí paměť!**
* **Větší celek než vlákno!**
* Proces **může mít své podprocesy**, které s ním souvisí, ale pracují nezávisle na hlavním procesu

**Vlákna**

* Něco jako "podprogram", sada instrukcí, která běží nezávisle na jiných vláknech v procesu.
* **Sdílí paměť s ostatními vlákny!**
* Většinou máme 1 hlavní vlákno => které **vytváří a spravuje ostatní vlákna**
* Každé vlákno má svůj vlastní zásobník -> aby nedocházelo přepisování dat mezi vlákny
* V C# reprezentováno třídou Thread (System.Threading)
  + Thread.isAlive -> Zjistí zda vlákno žije
  + Thread.Start() -> Spustí vlákno
  + Thread.Join() -> Zablokuje volající vlákno do té doby, než toto vlákno skončí
  + Thread.Sleep() -> Uspí konkrétní vlákno na určený čas

**Vytvoření a spuštění vlákna**

Thread t1 = new Thread(func);

t1.Start();

**Nebezpečí vláken**

* **Pokud více vláken přistupuje k jedné části kódu současně a modifikují proměnné, může nastat problém**:
  + **Postup:**
    - Mámě proměnnou typu **int** s názvem **číslo** a hodnotou nastavenou na **0**
    - Vlákno 1, přečte proměnnou číslo (hodnota proměnné = 0)
    - Vlákno 2, přečte proměnnou číslo (hodnota proměnné = 0)
    - Vlákno 1, přičte k proměnné číslo 1 (hodnota proměnné pro vlákno 1 = 1)
    - Vlákno 2, odečte od proměnné číslo 1 (hodnota proměnné pro vlákno 2 = -1)
    - Vlákno 1, uloží proměnnou číslo (proměnná nyní nabývá hodnoty 1)
    - Vlákno 2, uloží proměnnou číslo (proměnná nyní nabývá hodnoty -1)
  + **Tímto byla narušena integrita dat**
    - Po těchto operacích by proměnná měla nabývat hodnoty 0, ale nabývá hodnoty -1.

A screen shot of a computer program

Description automatically generated with medium confidence

**A black background with white text

Description automatically generated with low confidence**

**Asynchronní metody**

* **Svým voláním neblokují běžíčí vlákno**
* V C# jsou tyto metody označeny keywordem Async
* Při volání metody běží paralelně, nezávisle na volajícím vlákně
  + Pokud chceme na dokončení metody počkat, **použijeme keyword Await**.
    - (Na metodu musíme mít nějákou referenci, abychom toto mohli udělat..)
* Keyword Await lze použít pouze v Async metodě.
* **Pro jejich využití nemusíme vytvářet vlákna** a proto usnadňují paralelní programování.
* Na pozadí jsou implementovány pomocí vláken.

**Příklad v C#**

**Concurrent design patterns**

**Read/Write Lock**

* Řeší problém s integritou dat => 2 vlákna nebudou pracovat se stejnou proměnnou zároveň
* Uzamkne část kódu, pokud do ní vstoupí vlákno
  + Uzamykání chvilku trvá..
* Do této části kódu nesmí vstoupit žádné jiné vlákno
* Po ukončení práce uzamykatele se kód odemkne a dovnitř se vpustí další vlákno buďto:
  + Náhodně
  + Vlákna se řadí do fronty
* V C# implementován pomocí keywordu lock(object)
  + Část kódu se uzamyká na základě nějákého objektu

**Monitor**

* Ekvivalentní locku (lock je compiler keyword pro monitor...), trochu jiná syntax + metody ->
  + Monitor.enter(): Uzamyká objekt
  + Monitor.exit():
  + Monitor.wait(): Čeká než přijde pulse z jiného threadu
  + Monitor.pulse(): Pulsne jeden konkrétní thread
  + Monitor.pulseAll(): Pulsne všechny thready v procesu

**Thread Pool**

* Hlavní myšlenkou je vytvoření a udržování více vláken najednou, kterým jsou postupně přidělovány tasky
* Výhodou je eliminace času který by byl potřeba na vytvoření/terminaci vlákna
* Důležitým parametrem je zvolení optimálního počtu threadů v poolu

**Mutex**

* Lock, který může pracovat skrze více procesů -> není vázán na jeden. (tzv. computer-wide mutex vs. application-wide lock)