Vlákna v Jave

(základy paralelného programovania)

Róbert Novotný
robert.novotny@upjs.sk

Načo je dobré paralelné programovanie?

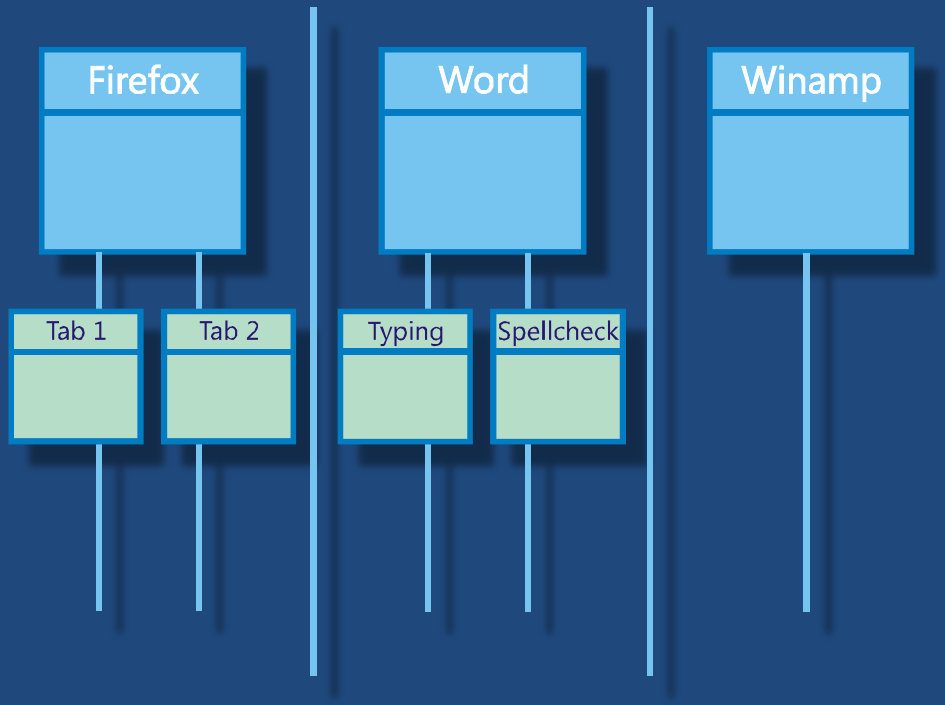
- algoritmus je definovaný ako postupnosť krokov, ktorá sa vykonáva postupne (sekvenčne, sériovo)
- softvér je implementáciou algoritmu, kde sa jednotlivé inštrukcie tiež vykonávajú sekvenčne

Čo keby sa niektoré inštrukcie dali vykonávať súčasne?

- *píšem text a zároveň kontrolujem pravopis*
- *prehliadam web a zároveň sa mi prehráva zvuk*

Paralelné vykonávanie inštrukcií

- v súčasných operačných systémoch sa to už deje
- máme totiž **procesy** – paralelne bežiace inštancie programov
  - koľko procesov beží vo vašich notebookoch?
- v rámci procesu môže paralelne bežať niekoľko vlákien (**threads**)



Každý proces pozostáva z minimálne jedného vlákna.

Java a vlákna

- práca s vláknami bola braná do úvahy už pri návrhu Java platformy
  - C: rozšírenie cez viacero knižníc; C#: na úrovni Javy; PHP a Python: neuvažuje sa
- základné veci zabudované priamo do jazyka
  - vlákna sú objekty
  - základné koncepty vo viacerých štandardných balíčkoch
- Thread synchronized lock mutex monitor Runnable queue wait-notify thread-safety critical section executor callable

Java a vlákna

- ako spustiť viacero paralelných úloh?
- viacerými spôsobmi (toť tradícia Javy)

1. vytvoriť **úlohu**, teda triedu implementujúcu java.lang.Runnable
2. do metódy run() uviesť kód, ktorý bude bežať paralelne
3. inštanciu spustiť **exekútorom**

Trieda implementujúca *Runnable*

```
public class NumberTask implements Runnable {
  private int number;

  public NumberTask(int number) {
    this.number = number;
  }

  public void run() {
    for(int i = 0; i < 3; i++) {
      System.out.println(number);
    }
  }
}
```

- trikrát vypíše na konzolu jedno číslo

Exekútor nevymáha majetok

- exekútor je „služba", ktorej vieme zaslať viacero úloh
- postará sa o ich paralelný beh
- zvyčajne každú úlohu spustí v samostatnom vlákne

Aký je rozdiel medzi vláknom a úlohou?

- **úloha** predstavuje kód (algoritmus), ktorý sa má vykonať paralelne
- **vlákna** predstavujú paralelné behy úloh

Paralelný beh úloh v Jave

```
public static void main(String[] args) {
  ExecutorService executor
    = Executors.newCachedThreadPool();
  NumberTask task1 = new NumberTask(24);
  NumberTask task2 = new NumberTask(12);
  System.out.println("Odosielam úlohu.");
  executor.submit(task1);
  System.out.println("Odosielam druhú úlohu.");
  executor.submit(task2);
  System.out.println("Úlohy odoslané.");
  executor.shutdown();
}
```

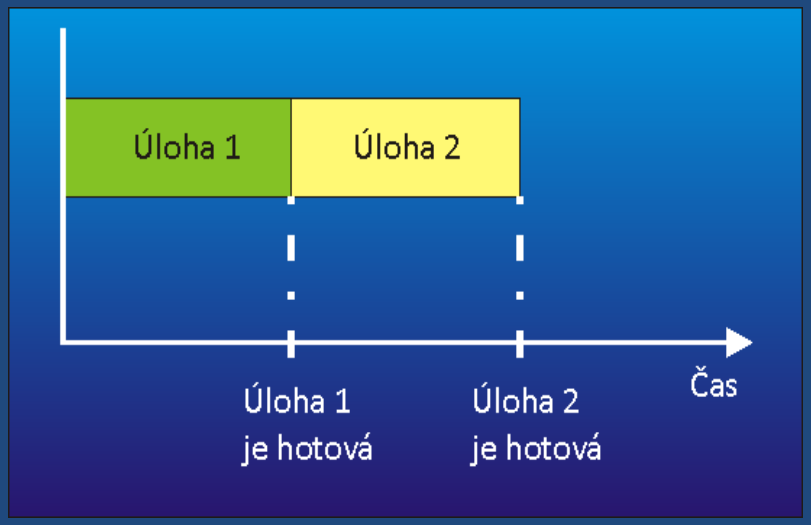
Paralelný beh úloh v Jave

- po odoslaní úlohy cez submit() sa úloha **ihneď** spustí paralelne v inom vlákne
- po vykonaní všetkých úloh treba exekútora zastaviť (metóda shutDown())
  - inak pobeží na pozadí jedno vlákno, ktoré spravuje úlohy a aplikácia neskončí

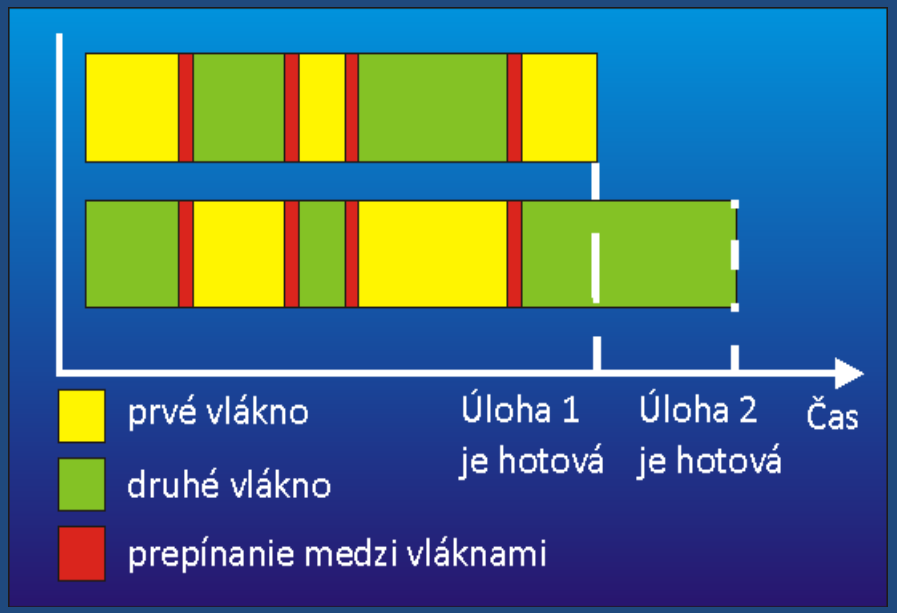
```
Odosielam úlohu.
24
24
Odosielam druhú úlohu.
12
Úlohy odoslané.
24
12
12
...
```

Procesy vs. vlákna

Bežný jednoúlohový systém



Geniálny viacúlohový systém

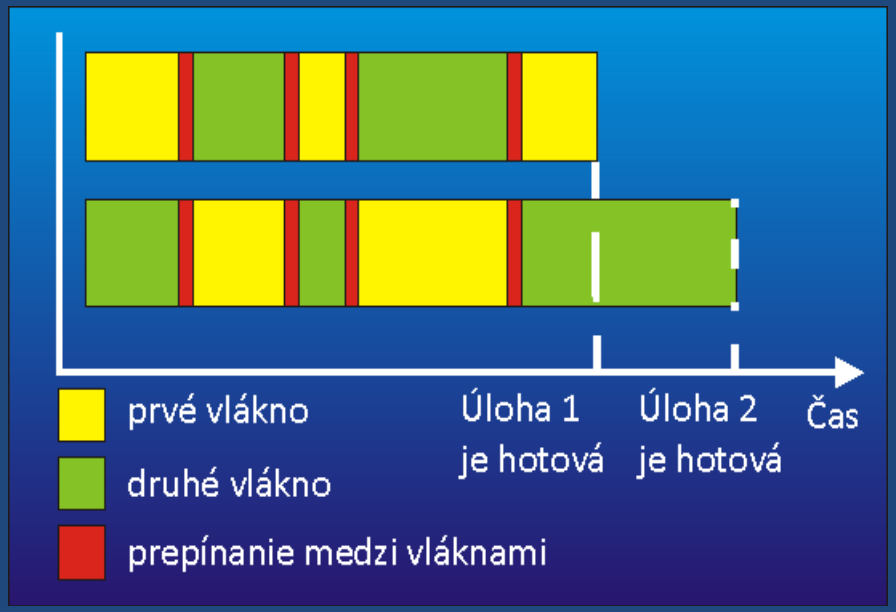


odkedy používam viacúlohový systém, moja bielizeň je voňavejšia!

Procesy vs. vlákna

- na jednom procesore môže bežať v skutočnosti paralelne len jedno vlákno
  - na dvoch dve atď
- procesor budí ilúziu paralelnosti tým, že prepína vlákna medzi sebou

Geniálny viacúlohový systém



- time-slicing – procesorový čas je rozdelený medzi jednotlivé vlákna

Paralelný beh úloh v Jave

- prepínanie medzi vláknami je nedeterministické = náhodné
- výpis čísel je pri každom spustení programu v inom poradí, môže sa prestriedavať...
- jedna z tragédií: nedá sa spoliehať na poradie!

```
Odosielam úlohu.
24
24
Odosielam druhú úlohu.
12
Úlohy odoslané.
24
12
12
...
```

```
Odosielam úlohu.
24
Odosielam druhú úlohu.
12
12
Úlohy odoslané.
24
12
24
24
...
```

Thread – klasická trieda pre vlákna

- úloha (= inštancia *Runnable*) bola zaslaná exekútorovi, ktorý ju spustil v paralelnom vlákne
- exekútor je elegantný, ľahko sa s ním pracuje
- **staršia možnosť**: úlohu podhodíme priamo vláknu, t. j. inštancii java.lang.Thread
- na inštancii zavoláme metódu start()

```
public static void main(String[] args) {
  Runnable úloha = new TicTacTask();
  Thread vlákno = new Thread(úloha);
  vlákno.start();

  //aby sme videli, že kód beží paralelne:
  while(true) {
    System.out.println("Hlavné vlákno beží");
  }
}
```

Procesy vs. vlákna

- ak to preženieme s vláknami, môže sa stať, že aplikácia strávi čas prepínaním medzi nimi a neostane čas na skutočnú prácu
- ak sa máte hrať s dvoma deťmi, je to (pomerne) ľahké
- ak sa máte hrať s celou materskou škôlkou, strávite veľa času manažovaním detí, aby nepadali z okien, nebili sa...

Pozastavenie vykonávania úlohy (*sleep*)

- čo ak chceme, aby sa úloha v istom momente na chvíľu pozastavila?
  - príklad: vypíš číslo, čakaj, vypíš ďalšie
- vlákno vykonávajúce úlohu môžeme uspať (*sleep*)

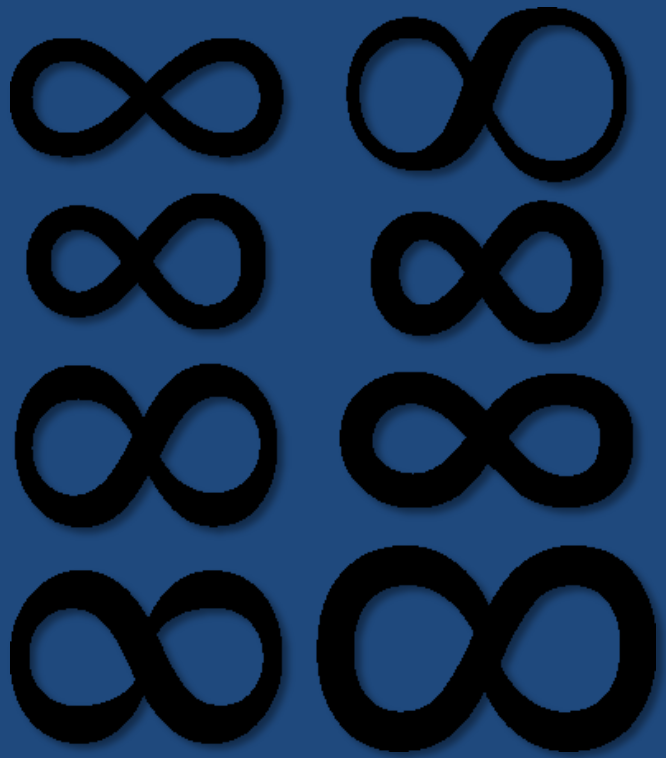
v metóde run() môžeme použiť

```
try {
    TimeUnit.SECONDS.sleep(1);
} catch (InterruptedException e) { }
```

- trieda *TimeUnit* má rôzne časové jednotky (*SECONDS*, *MILLISECONDS*...), v parametri `sleep()` uvedieme dĺžku spánku
- výnimku budeme zatiaľ ignorovať

Donekonečna bežiace vlákna

- naše vlákna bežali len krátku dobu
- zadali sme im úlohu, vykonali ju a skončili
- sú úlohy, ktoré bežia neustále:
  - kontrola pravopisu
  - prekresľovanie okna
- implementácia je jednoduchá:
  - kód v metóde `run()` pobeží v nekonečnom cykle
  - while(true) {...}



Ukážka kódu vlákna

```
public class TicTacTask implements Runnable {
    public void run() {
        boolean isTic = false;
        while(true) {
            String ticOrTac = isTic ? "tik" : "tak";
            System.out.println(ticOrTac);
            isTic = !isTic;

            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) { }
        }
    }
}
```

Každú sekundu vypíše „tik“ alebo „tak“.

- cyklíme sa do nekonečna: zabezpečíme priebežný beh vlákna
- po vypísaní textu „zaspíme“ na sekundu
- výnimku (ne)odchytíme, niekto totiž môže spiace vlákno prebudiť skôr než vyprší limit

Paralelný beh úloh v Jave

```
public static void main(String[] args) {
  ExecutorService executor
    = Executors.newCachedThreadPool();
  TicTacTask task = new TicTacTask();
  System.out.println("Odosielam úlohu.");
  executor.submit(task);
  // ... tu sa deje milión vecí ...
  executor.shutdown();
}
```

- `shutdown()` uzatvorí exekútor po dobehnutí všetkých úloh
  - nové úlohy už nebudú prijímané
- lenže naša úloha nikdy nedobehne!
- táto aplikácia teda pobeží donekonečna

Ukončenie behu úlohy

- shutdown() uzatvorí exekútor po dobehnutí úloh
- shutdownNow() sa pokúsi prerušiť beh vlákien, v ktorých bežia úlohy

```
executor.shutdownNow();
```

- ak namiesto **shutdown()** vložíme do aplikácie **shutdownNow()**, nič sa nestane!

Vlákno v Jave sa nedá odstreliť!

Ukončenie behu úlohy

- vlákno bežiacej úlohy nie je možné odstreliť
- kedysi bola možnosť (metóda Thread#stop()),
  - objekty však mohli zostať v nekonzistentnom stave
- úlohe možno navrhnúť, aby sa ukončila
- shutdownNow() navrhne všetkým odoslaným úlohám v exekútore ukončenie
  - ak úloha spí (*sleep*), vyvolá sa *InterruptedException*, ktorú odchytíme a ukončíme beh
  - ak úloha nikdy nezaspí, musí aktívne volať statickú metódu Thread.currentThread().isInterrupted(), ktorá zistí, či nastalo prerušenie

Ukončenie behu úlohy

- metóda *shutdownNow()* nastaví na vlákne úlohy príznak „mal by si sa ukončiť“
- statické volanie Thread.currentThread().isInterrupted() zistí hodnotu príznaku a vynuluje ho
- bežné volanie *isInterrupted()* len zistí hodnotu príznaku
- volanie *interrupted()* zistí hodnotu príznaku a vynuluje ju

Ukážka ukončenia behu úlohy

```
public class HeartBeatTask implements Runnable {
  public void run() {
    while(true) {
      System.out.println("Stále žijem.");
      if(Thread.interrupted()) {
        System.out.println("Koniec.");
        return;
      }
    }
  }
}
```

```
public static void main(String[] args)
  throws Exception {

  ExecutorService executor = ...
  HeartBeatTask task = new HeartBeatTask();
  executor.submit(task);

  TimeUnit.SECONDS.sleep(2);
  executor.shutdownNow();
}
```

spustíme "tlčúce" vlákno, uspíme hlavné vlákno na dve sekundy a potom tlčúce vlákno ukončíme

Odporúčanie pre písanie úloh

- každá nekonečne bežiaca úloha musí mať vo vnútri cyklu
  - buď kontrolu cez isInterrupted() / interrupted()
  - alebo mať možnosť sa uspať cez TimeUnit.XXX.sleep()
- v opačnom prípade je neodstreliteľná
- samozrejme, každé vlákno dobehne po (násilnom) ukončení aplikácie

Poznámky k *shutdown/shutdownNow*

- shutdown() ani shutdownNow() neblokujú beh hlavného vlákna
- po zavolaní týchto metód sa teda nečaká na ukončenie úloh
- ak chceme zastaviť exekútor a počkať na dobehnutie / odstrelenie úloh:

```
executor.shutdownNow();
executor.awaitTermination(1, TimeUnit.SECONDS)
```

- zavoláme **shutdown()** / **shutdownNow()**
- **awaitTermination()** počká daný čas
  - vráti **true**, ak sa exekútor zatvoril pred uplynutím lehoty

blokujeme hlavné vlákno 1 sekundu

Procesy vs. vlákna

- vieme, že každý proces pozostáva z minimálne jedného vlákna
- vlákna zdieľajú spoločnú haldu (miesto, kde nažívajú objekty), procesy túto možnosť nemajú
  - to je výhoda: pohodlná výmena dát
  - to je najväčšia kliatba: nesprávne použitie vedie k nesmierne ťažkému ladeniu
- ale každé vlákno má svoje vlastné lokálne premenné, do ktorých iné vlákna nevidia

Zdieľanie dát medzi vláknami

- vlákna si môžu zdieľať dáta (keďže majú spoločný prístup k halde)
- môže nastať viacero problémov:
  - *interferencia*: viacero vlákien pristupuje naraz k rovnakým dátam
  - *nekonzistencia*: jedno vlákno vidí dáta jedným spôsobom, iné vlákno vidí tie isté dáta inak
  - *uviaznutia*: vlákno pracuje nad zdieľaným dátami a tým zablokuje ostatné vlákna

Spolupráca medzi vláknami 2

Niektoré problémy rieši synchronizácia!

Príklad, keď sa veci po..kazia

- Majme dvoch stravníkov (= dve vlákna) pristupujúcich k rovnakej jedálni
- Napriek zdanlivej paralelnosti sa v danej chvíli vykonáva len jediná inštrukcia (= riadok)

```
class Jedáleň {
  int početObedov;

  public obslúž(Stravník s) {
    if(početObedov > 0) {
       početObedov = početObedov--;
    }
  }
}
```

- Niektoré „jednoduché“ operácie môžu byť postupnosťou viacerých operácií, aj keď to nie je zjavné:

```
početObedov--  => získaj hodnotu, zníž o 1, ulož hodnotu
```

Príklad, keď sa veci po..kazia

- Majme na začiatku päť obedov:

Stravník Ignác	Stravník Gejza
jedáleň.obslúž()	
if(početObedov > 0)	
	jedáleň.obslúž()
	if(početObedov > 0)
získaj počet obedov (5)	
	získaj počet obedov (5)
	zníž počet o jedna (4)
zníž počet obedov (4)	
ulož novú hodnotu (4)	
	ulož novú hodnotu (4)

- Lenže na konci máme mať len tri obedy!

Príklad, keď sa veci po..kazia

- predošlý príklad ukázal 2 veci:
  - zmena dát jedným vláknom sa nemusí prejaviť v druhom vlákne (nekonzistencia)
    - ešte extrémnejšia situácia: každé vlákno beží na samostatnom jadre s nezávislými cache pamäťami. Zmena premennej sa nemusí prejaviť v pamäti RAM, ktorú vidia všetky vlákna.
  - operácie vykonávané vláknami sa môžu prekrývať (interferencia)
    - môžu (ale nemusia! a to je horor!) nastať konflikty, ktoré sú zdanlivo nevinné alebo neodladiteľné

Riešenie: kritická sekcia

- vlákno si môže zdieľané dáta uzamknúť pre seba, vykonať s nimi, čo treba a potom ich uvoľniť.
- ďalší negatívny príklad:

1. mám rezeň na tanieri
2. vezmem príbor
3. slintám
4. napichnem rezeň
5. žujem

po treťom kroku ma plánovač úloh vypne a spustí druhé vlákno

1. mám rezeň na tanieri
2. vezmem príbor
3. slintám
4. napichnem rezeň
5. žujem

- zrazu zistím, že v prvom vlákne nemám čo napichnúť!

Riešenie: kritická sekcia

1. vezmem rezeň
2. zamknem sa s ním do kuchyne
3. vezmem príbor
4. zjem
5. vyjdem z kuchyne

kritická sekcia

- kritická sekcia je úsek kódu, ku ktorému môže pristupovať najviac jedno vlákno
- celá filozofia sa nazýva monitor
- výlučný prístup jedným vláknom zabezpečíme zámkom na objekt, ktorý budeme modifikovať (tu: rezeň)
- mutex lock (mutual exclusion, vzájomné vylúčenie) – zámok vzájomného vylúčenia

Synchronizované metódy

- Metóda môže byť synchronizovaná
- Celá metóda je potom kritickou sekciou, kde sa uzamkne celá inštancia triedy.

```
class Jedáleň {
  int početObedov;

  public synchronized obslúž(Stravník s) {
    if(početObedov > 0) {
       početObedov = početObedov--;
    }
  }
}
```

- Synchronizovanú metódu na danej inštancii jedálne môže vykonávať najviac jedno vlákno.
- Ak chce iné vlákno vykonávať tú istú metódu, plánovač úloh ho zaradí do fronty.
- Analógia: horda ľudí pobehujúca pred kuchyňou a búchajúca na jej dvere.

Synchronizované sekcie kódu

- Kritická sekcia by mala byť čo najkratšia.
- Dlhé kritické sekcie = dlhšia doba vykonávania = dlhý pobyt v kuchyni = dlhé čakanie = nervóznejší ostatní ľudia.
- Okrem toho: niekedy chceme uzamknúť aj iný objekt než seba

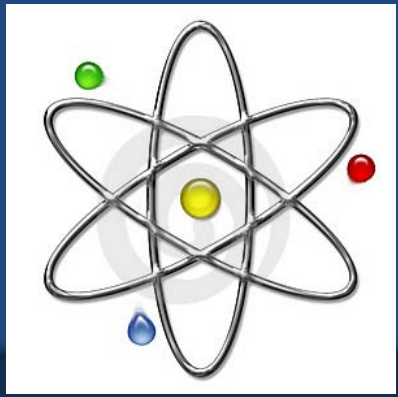
- blok synchronized umožňuje určiť kritickú sekciu
- a umožňuje určiť objekt, ktorý sa uzamkne

```
class Jedáleň {
  int početObedov;

  public obslúž(Stravník s) {
    synchronized(this) {
      if(početObedov > 0) {
         početObedov = početObedov--;
      }
    }
  }
}
```

Atomické operácie

- Niektoré operácie sú sami o sebe atomické: teda buď sa vykonajú celé, alebo sa nevykonajú vôbec.
- Priradenie do premennej a získanie jej hodnoty je atomické:
  - okrem primitívov typu double a long
- ak je inštančná premenná označená ako volatile, priradenie a čítanie je *vždy* atomické
- takáto premenná je navyše vždy viditeľná pre všetky vlákna
  - napr. optimalizácia ukladania hodnoty do cache je vypnutá, hodnota sa vždy číta z hlavnej pamäte



Synchronizované sekcie kódu

- Kedy synchronizovať?
- Ak pracujem s dátami v inštančných premenných
  - prístup k lokálnym premenným v metóde nemusí byť synchronizovaný
- Ak pracujem s objektami, ktoré môžu byť zdieľané

- Trieda, ktorá pracuje s dátami bezpečne, sa nazýva thread-safe.
- Ak správne používame thread-safe triedu, tiež sme thread-safe.
- Takto môžeme zdola vybudovať bezpečnú aplikáciu
- Náznak svetového trendu: inštančné premenné sú zlé ;-)

Thread-safe triedy a zvyšok

- Ako viem, či je trieda thread-safe?
- Mám to explicitne napísané v dokumentácii
- Ak to v dokumentácii nie je, predpokladám, že trieda nie je thread-safe
- Častý príklad: kolekcie (zoznamy, množiny...)
  - štandardné implementácie nie sú bezpečné
    - niekto mi môže zmeniť zoznam, keď cez neho prechádzam
    - dve vlákna môžu vkladať do rovnakého poľa v útrobách ArrayListu – ten sa rozpadne
  - prístup k nim treba synchronizovať, ale jestvujú aj iné možnosti

Thread-safe triedy a zvyšok

- riešenie z doby kamennej (Java 1.0):
  - zoznamu zodpovedá trieda java.util.Vector
  - pre mapu máme java.util.Hashtable
  - pre množinu máme... nič
    - môžeme si nasimulovať vektorom

- tieto triedy sú thread-safe
  - bežná otázka na interview: *aký je hlavný rozdiel medzi ArrayList-om a Vectorom*?
- lenže tak, že všetky metódy sú *synchronized*
  - to je to nesmierne pomalé
  - rádovo 10-krát pomalšie oproti iným riešeniam
  - dlho sa totiž čaká na uvoľnenie zámku
- tieto triedy sú zastaralé, nie je dôvod ich používať, jestvujú oveľa lepšie spôsoby

Thread-safe triedy a zvyšok

- trieda java.util.Collections má statické metódy
- vedia obaliť kolekciu a vrátiť jej thread-safe verziu
  - Collections.synchronizedList(*zoznam*)
  - Collections.synchronizedSet(*množina*)
  - Collections.synchronizedMap(*mapa*)
  - a pre zvyšné interfejsy kolekcií
- problém:
  - s takýmito kolekciami nemusíme pracovať v kritickej sekcii
  - ale: prechádzať ich musíme v kritickej sekcii
    - inak nedefinovaný stav

Thread-safe kolekcie a zvyšok

- synchronizácia platí len pre jednotlivé operácie

```
Set<String> mená = Arrays.asList("Ringo", "John",
                                 "George", "Paul");
Set<String> synchroMená = Collections.synchronizedSet(mená);
for(int i = 0; i < synchroMená.size(); i++) {
  System.out.println(synchroMená.get(i));
}
```

- podmienka *i < synchroMená.size()* vo *for* cykle sa overuje v každej iterácii
- čo ak zistíme, že veľkosť je 5 a iné vlákno nám po overení podmienky vymaže posledný prvok? => NullPointerException
- iterovanie musí byť v kritickej sekcii!

Thread-safe kolekcie a zvyšok

- iterovanie musí byť v kritickej sekcii!

```
synchronized(synchroMená) {
  for(int i = 0; i < mená.size(); i++) {
    System.out.println(mená.get(i));
  }
}
```

```
synchronized(synchroMená) {
  for(String meno : mená) {
    System.out.println(mená.get(i));
  }
}
```

- problém: kým jedno vlákno iteruje, ostatné musia čakať...

Thread-safe kolekcie a zvyšok

- čo sa stane, ak by sme zabudli na kritickú sekciu?
  - skrátený for cyklus používa v skutočnosti iterátory (metóda iterator() vracajúca java.util.Iterator)
  - tie sú *fail-fast* – ak sa zmení kolekcia, cez ktorú iterujú, vyhodia výnimku ConcurrentModificationException
- ak iterujeme klasickým for cyklom, môžu sa diať divné veci!
  - zrada!
  - môžu sa vracať záhadné null hodnoty
  - celkovo sa to môže správať záhadne a nepredvídateľne

Thread-safe kolekcie a zvyšok

- synchronizované kolekcie sú len obmedzene thread-safe
- jednotlivé operácie sú thread-safe
- postupnosti operácii však už nie!

```
// vkladajme prvok do mapy, len keď v nej nejestvuje
Map mapa = Collections.synchronizedMap(new HashMap());
if (!mapa.containsKey(key)) {
      mapa.put(key, value);
}
```

- takýto kód musíme mať v kritickej sekcii
- budí sa mylný dojem, že stačí mi takto synchronizovať a mám pokoj, ale s týmto mi nepomôže nik!

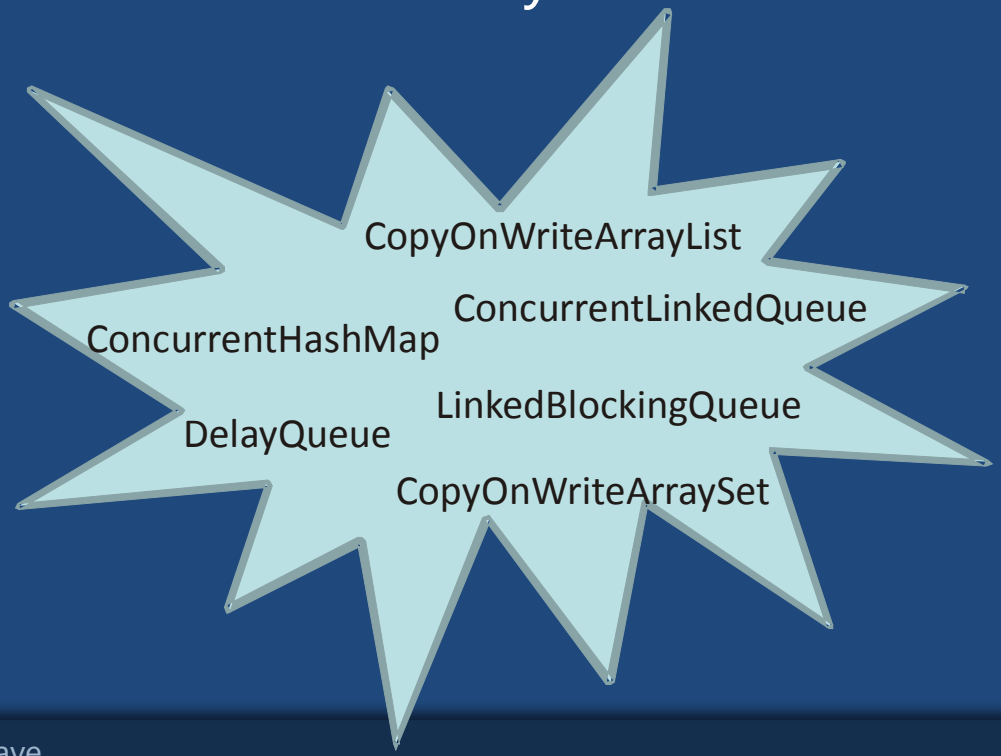
čo ak po overení podmienky a ešte pred vložením prvku iné vlákno vloží prvok s rovnakým kľúčom?

Nevýhody synchronizovaných kolekcií

- zdanlivo sa s nimi pracuje jednoducho
- škálovateľnosť (= efektivita pri narastajúcom počte súčasných použití) klesá
- synchronizované kolekcie, vektory a hašovacie tabuľky podporujú prácu len s jediným zámkom:
  - v danej chvíli môže s kolekciou narábať jediné vlákno
  - ostatné musia čakať
- dokonca smutný protipríklad:
  - použitím synchronizovanej mapy chceme implementovať cache, do ktorej ukladáme často používané dáta, aby sme urýchlili prístup
  - môže sa stať, že vlákna budú čakať na získanie dát z cache dlhšie než by trvalo ich nové vytvorenie

Konkurentné kolekcie

- našťastie máme konkurentné kolekcie
- konkurentné kolekcie majú operácie implementované nad špeciálnymi algoritmami, ktoré podporujú paralelný prístup
- sú prirodzene a automaticky thread-safe



Konkurentné kolekcie

- zoznam: java.util.concurrent.CopyOnWriteArrayList
  - užitočný, ak zoznamom prechádzame oveľa častejšie než ho modifikujeme
  - pracuje nad poľom – zmeny vytvárajú na pozadí nové pole
  - iterovanie: na začiatku sa vytvorí kópia poľa, nad ktorou sa iteruje. Ak sa počas iterovania pole zmení, iteráciu to neovplyvní
- množina: java.util.concurrent.CopyOnWriteArraySet
  - analogický k zoznamu
- mapa: java.util.concurrent.ConcurrentHashMap
  - získavanie prvkov zvyčajne neblokuje
  - operácie získavania a modifikovania sa môžu prekrývať
  - získavanie zodpovedá stavu po poslednej ukončenej modifikácii
  - pracuje nad viacerými hašovacími tabuľkami, ktoré sú interne uzamykané jednotlivo

Konkurentné kolekcie

- front pevnej dĺžky: java.util.concurrent.ArrayBlockingQueue
  - kolekcia FIFO (prvý vložený prvok bude aj prvým vybratým)
  - prázdny front => vyberanie prvkov čaká na naplnenie zoznamu
  - plný front => vkladanie prvkov čaká na vyprázdnenie
  - dĺžka frontu je nemenná, nikdy sa nezväčšuje
  - dobré pre problém *producent-konzument*
- všeobecný front: java.util.concurrent.ConcurrentLinkedQueue
  - typická trieda pre kolekciu zdieľanú viacerými vláknami
  - FIFO
  - neobmedzená dĺžka
- k dispozícii aj ďalšie fronty:
  - prioritné fronty, synchrónne (výbery spárované s vkladaniami)...

Koordinácia vlákien

- behy úloh vo vláknach možno koordinovať
- základné operácie:
  - interrupt – navrhne vláknu úlohy, aby sa ukončilo
  - sleep – vlákno sa uspí na nejaký čas
  - yield – vlákno sa dobrovoľne vzdá procesorového času v prospech ostatných vlákien
  - join – vlákno sa spustí až po dobehnutí iného vlákna
  - wait-notify – vlákno sa uspí a čaká, kým ho niekto neupozorní
- nesprávne použitie môže vyvolať mnoho ťažko laditeľných problémov: deadlock, livelock...

Prerušenie behu vlákna – *interrupt*

- ak úloha beží v exekútore, volanie *shutdownNow()* preruší všetky vlákna
- ak úloha beží vo vlákne, zavolaním interrupt() na inštancii *Thread*u navrhnem vláknu ukončenie
- platia tie isté odporúčania ako pri shutdownNow()

```
public static void main(String[] args)
  throws Exception {

  TicTacTask task = new TicTacTask();
  Thread ticTacThread = new Thread(task);
  ticTacThread.start();

  TimeUnit.SECONDS.sleep(2);
  ticTacThread.interrupt();
}
```

Uspanie vlákna – *sleep*

- v úlohe môžem zavolať *TimeUnit.[časováJednotka].sleep()*
- ak mám inštanciu vlákna, môžem na nej zavolať sleep() priamo.
- v parametri udám milisekundy

```
public static void main(String[] args)
  throws Exception {

  TicTacTask task = new TicTacTask();
  Thread ticTacThread = new Thread(task);
  ticTacThread.start();
  // uspím hlavné vlákno
  TimeUnit.SECONDS.sleep(2);
 // uspím tik-tak vlákno na 4 sekundy
  ticTacThread.sleep(4000)
}
```

Vzdanie sa procesorového času – *yield*

- úloha sa môže na chvíľu dobrovoľne vzdať procesorového času
- galatne uvoľní procesor iným vláknam
- na rozdiel od sleep(), kde zadávame čas, tuto je na zvážení procesora, ako dlho nechá vlákno čakať
- v úlohe zavoláme Thread.currentThread().yield()
- ak máme inštanciu Thread-u, môžeme yield() volať priamo na nej
- praktické použitie veľmi zriedkavé

Spustenie vlákna po dobehnutí iného vlákna

- vlákno sa môže spustiť po dobehnutí iného vlákna
- metóda t.join() – aktuálne vlákno sa spustí po dobehnutí vlákna *t*
- funguje len na Threadoch, ale vieme nasimulovať aj v exekútorovi
- hlavný bežec začne bežať po dobehnutí prvého vlákna:

```
public static void main(String[] args)
  throws InterruptedException {
  Thread prvéVlákno = ...
  prvéVlákno.start();

  prvéVlákno.join();
  for (int i = 0; i < 10; i++) {
    System.out.println("Hlavný bežec beží "
      + i + ". kolo.");
  }
}
```

Ďalšie problémy pri koordinácii vlákien

- problém obedujúcich filozofov (Dijkstra, 1965)
- pri stole sedí päť filozofov, ktorí
  - buď osamote uvažujú
  - alebo jedia špagety
- filozofovia majú po ľavici a pravici 1 vidličku
- špagety jedia 2 vidličkami
- použiť môžu len vidličky po ľavici a pravici (žiadne naťahovanie cez stôl!)
- filozofovia medzi sebou nekecajú

Obedujúci filozofovia a deadlock

- čo ak každý filozof schytí vidličku po ľavici?
- každý filozof začne čakať na vidličku po pravici, aby mohol jesť
- lenže tie sú obsadené!
- Descartes čaká, kým mu Voltaire uvoľní vidličku, lenže ten čaká na Platóna... Socrates čaká na Descarta

Uviaznutie! Deadlock!

- filozofovia zvádzajú súboj o obsadenie vidličky, lenže víťaz závisí od načasovania vlákien. To je nedeterministické:

Race Condition!

Podmienky pre výskyt deadlocku

„Ak sa na križovatke stretnú dve súpravy, obe sú povinné zastaviť. Súprava môže pokračovať v jazde až po tom, čo druhá súprava opustí križovatku.“

-- nelogický železničný zákon v Kansase

1. **vzájomné vylúčenie**: prostriedok (dáta) môže využívať najviac jedno vlákno (vidličku drží len 1 filozof)
2. **drží a čaká**: vlákno drží jeden prostriedok a čaká na iný (filozof drží jednu vidličku a čaká na druhú)
3. **vylúčenie preemptívnosti**: vlákno sa musí vzdať prostriedku dobrovoľne. Nie je k dispozícii spôsob, ktorým je možné odobrať vláknu prostriedok. (filozofovi nemôže nik odňať vidličku)
4. **cyklické čakanie**: dve či viac vlákien tvoria cyklickú postupnosť, kde jedno vlákno čaká na prostriedok držaný nasledujúcim vláknom v postupnosti (Descartes čaká na Voltaira, ... čaká na Descartesa)

Oprava deadlocku a livelock

- zavedie sa časový limit
- ak filozof drží ľavú vidličku a do piatich minút sa neuvoľní pravá vidlička, musí ľavú vidličku položiť
- potom *musí* ďalších päť minút uvažovať
- zbavíme sa uviaznutia
- čo ak všetci *naraz* prídu k stolu, *naraz* zdvihnú ľavé vidličky, po piatich minútach ich položia a potom čakajú päť minút, potom opäť *naraz* zdvihnú ľavé vidličky...

Livelock! Vlákna pracujú, ale systém prešľapuje na mieste.

Obedujúci filozofovia

- čo ak máme dvoch filozofov sediacich oproti (napr. Descartesa a Konfucia), ktorí rýchlo jedia a rýchlo uvažujú?
- každý vezme dve vidličky
- najedia sa a odložia ich
- ale predtým než sa dostanú k vidličkám ostatní, ich opäť chytia

Starvation! Vyhladovanie!

- ostatní traja filozofovia smutne sedia a hladujú
- problém je s načasovaním: ak procesor uprednostňuje niektoré vlákna, ostatné môžu čakať veľmi dlho

Riešenie obedujúcich filozofov

- zavedenie čašníka
  - filozof musí požiadať čašníka o povolenie zobrať si vidličku
  - najprv si berie ľavú a potom pravú
- očíslovanie vidličiek a filozofov
  - filozof najprv berie vidličku s nižším číslom a potom s vyšším číslom
  - odkladá najprv vyššiu vidličku, až potom nižšiu
  - ak sa všetci vrhnú na jedenie naraz, piaty filozof bude musieť čakať
- a mnohé iné riešenia (prístup Chandy-Misra...)

Typická úloha: producent-konzument

- Majme sklad (s obmedzenou kapacitou)
  - dodávateľ doň vkladá palety s tovarom
  - odberatelia ich vyberajú
- Dodávateľ je producent – vkladá do kontajnera položky, ak je voľné miesto
- Odberateľ je konzument – vyberá z kontajnera položky, ak tam nejaké sú.
- dve paralelne bežiace úlohy pracujú nad zdieľaným frontom s pevnou dĺžkou
- Ako zabezpečiť konzistentnosť dát?

Podpora producenta-konzumenta v Jave

- v Jave existuje priama podpora: interfejs java.util.concurrent.BlockingQueue
- kolekcia podporujúca front presne spĺňajúci požiadavky nášho skladu
- implementácia java.util.concurrent.LinkedBlockingQueue
  - umožňuje špecifikovať kapacitu
  - ak je front plný, ďalšie vkladanie čaká dovtedy, kým sa neuvoľní miesto
  - ak je front prázdny, vyberanie čaká dovtedy, kým sa neuvoľní miesto
  - navyše je to kolekcia, môžeme používať klasické metódy (add, remove, iterácie...)
- nemusíme používať žiadnu synchronizáciu

Producent v Jave

```
class Producent implements Runnable {

   private final BlockingQueue<String> front;

   Producer(BlockingQueue<String> front) {
     this.front = front;
   }

   public void run() {
     try {
       while(true) {
         front.put( ... );
       }
     } catch (InterruptedException ex) {
        // niekto nás prerušil
     }
   }
}
```

Konzument v Jave

```
class Konzument implements Runnable {

   private final BlockingQueue<String> front;

   Konzument(BlockingQueue<String> front) {
     this.front = front;
   }

   public void run() {
     try {
       while(true) {
         String s = front.take( ... );
       }
     } catch (InterruptedException ex) {
        // niekto nás prerušil
     }
   }
}
```

Príklad s dvoma konzumentami

```
public class Tester {
  public static void main(String[] args) {
     BlockingQueue front = new LinkedBlockingQueue();

     Producent p = new Producent(front);

     Konzument c1 = new Konzument(front);
     Konzument c2 = new Konzument(front);

     new Thread(p).start();
     new Thread(c1).start();
     new Thread(c2).start();
   }
 }
```

Viac o exekútoroch

- exekútor ExecutorService podporuje 2 typy úloh:
  - objekt typu Runnable (ak nepotrebujeme výsledok)
  - objekt typu Callable (metóda vracia výsledok)
- vykonávateľovi zašleme úlohu
  - metóda submit()
- úloha sa hneď začne vykonávať paralelne a niekedy v budúcnosti vráti výsledok
- submit() vracia inštanciu *budúceho výsledku* Future
- z inštancie *Future* vieme získať výsledok metódou get() Jej zavolanie počká na dobehnutie úlohy.

Paralelný beh úloh v Jave

- príklad: úloha vygeneruje päť náhodných čísiel, pred každým počká sekundu a vráti ich súčet.

```
public class Task implements Callable<Integer> {
  public Integer call() throws Exception {
    Random random = new Random();

    int sum = 0;
    for (int i = 0; i < 5; i++) {
      System.out.println("Suma je zatiaľ " + sum);
      TimeUnit.SECONDS.sleep(1);
      sum = sum + random.nextInt(50);
    }

    return sum;
  }
}
```

Paralelný beh úloh v Jave

- Odošleme úlohu cez submit()
- Volanie get() na budúcom výsledku čaká na dobehnutie – blokuje hlavné vlákno, kým nie je k dispozícii výsledok

```
public static void main(String[] args) {
  try {
    ExecutorService executor = Executors.newFixedThreadPool(1);
    WaitTask task = new WaitTask();
    System.out.println("Odosielam úlohu.");
    Future<Integer> result = executor.submit(task);
    System.out.println("Úloha odoslaná.");
    System.out.println("Výsledok: " + result.get());
    executor.shutdown();
  } catch (ExecutionException e) {
    System.out.println("Chyba pri získavaní výsledku.");
  } catch (InterruptedException e) {
    System.out.println("Čakajúce vlákno bolo prerušené.");
  }
}
```

Rôzne druhy exekútorov

- exekútor oddeľuje úlohu od spôsobu jej vykonania
- typická implementácia: thread pool - zásoba vlákien

Príklad: zákazníci a pokladne v *[vložte obľúbený obchod]*

- *úloha*: zákazník, ktorý chce platiť (*task*)
- *odbavovateľ*: pokladňa (beží a odbavuje zákazníkov)
- máme zásobu otvorených pokladní (*thread pool*)
- niektoré sú voľné, niektoré nie
- ak je zákazníkov veľa, otvoria sa nové pokladne (ale nemáme ich neobmedzený počet)
- ak je zákazníkov málo, pokladne zívajú prázdnotou, prípadne sa zatvoria

Ako získať konkrétneho exekútora

- statické metódy na triede java.util.concurrent.Executors

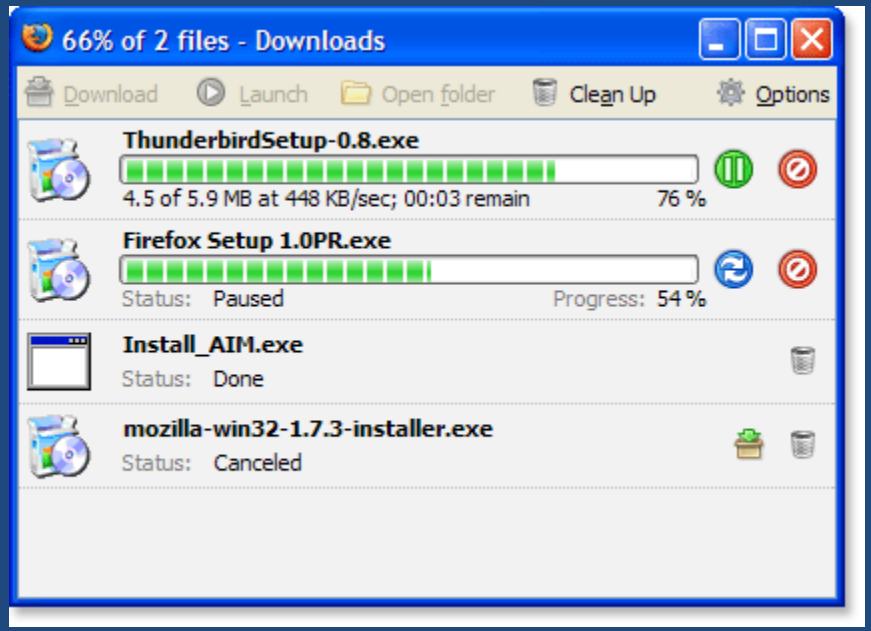
newFixedThreadPool(int X)	nový exekútor s pevným počtom vlákien *obchod s X pokladňami, ktoré sú vždy otvorené a nikdy sa nezatvoria*
newSingleThreadPool()	nový exekútor s jediným vláknom
newCachedThreadPool()	nové vlákna vytvára podľa potreby, vlákna ležiace ladom po istom čase zatvorí, recykluje odpočívajúce vlákna
newScheduledThreadPool()	pool, ktorý umožňuje plánované spúšťanie, opakované späšťanie, či odkladať spustenie o nejaký čas

- ďalšie užitočné metódy:

callable(Runnable r)	prevedie inštanciu Runnable na Callable, ktorá vracia null, keď dobehne
callable(Runnable r, V *result*)	to isté, po dobehnutí vráti *výsledok*

Ďalšie úlohy pre synchronizáciu vlákien

- chceme sťahovať veľa súborov z internetu paralelne
- po skončení sťahovania ukončiť aplikáciu
- Čiže:
  - z hlavného vlákna chceme naraz spustiť veľa vlákien
  - a po dobehnutí všetkých vlákien vykonať nejakú činnosť.



Paralelný beh a ukončenie vlákien I.

- vytvoríme veľa vlákien
- hlavné vlákno *join()*-neme na každé z vlákien

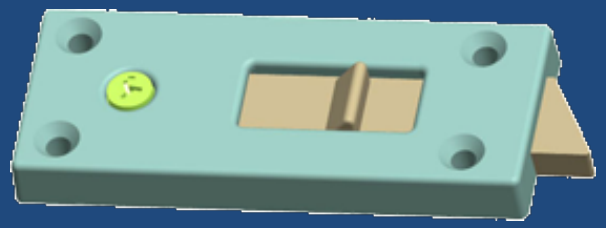
```
public static void main(String[] args) {
  for(int i = 0; i < 17; i++) {
     Thread t = new Thread(…);
     // spustíme vlákno
     t.start();
     // hlavné vlákno počká, tým t neskončí
     t.join();
  }
  System.out.println("Vlákna dobehli.");
}
```

Paralelný beh a ukončenie vlákien II.

- ak používame úlohy implementujúce *Runnable*, k joinu nemáme prístup
- úloha môže po dobehnutí znížiť globálne počítadlo o 1
- počítadlo má na začiatku takú hodnotu, koľko úloh pobeží paralelne
- hlavné vlákno môže aktívne kontrolovať, či má počítadlo hodnotu 0
- alebo použijeme *wait-notify* z problému producent-konzument. Hlavné vlákno bude spať, úlohy ho budú zobúdzať

Paralelný beh a ukončenie vlákien III

- trieda java.util.concurrent.CountDownLatch
- viacpolohová západka s odpočítavaním
- na začiatku nastavíme na danú hodnotu
- každé vlákno po dobehnutí zníži hodnotu o 1
- hlavné vlákno zatiaľ čaká, kým hodnota nedosiahne nulu
- latch je *thread-safe*, nemusíme používať kritickú sekciu



Paralelný beh a ukončenie vlákien

```
public class Task implements Callable<Integer> {
  private CountDownLatch latch;
  private int[] data;

  public Task(CountDownLatch latch, int[] data) {
    this.latch = latch;
    this.data = data;
  }

  public Integer call() throws Exception {
    int min = Integer.MAXINT;
    for (int i = 0; i < 5; i++) {
      if(data[i] < min)
        min;
    }
    latch.countDown();
    return min;
  }
}
```

po dobehnutí úlohy znížime počítadlo

Paralelný beh a ukončenie vlákien

```
public static void main(String[] args)
throws InterruptedException
{
   int[] pole1 = {1, 6, 2, 9, 5};
   int[] pole1 = {3, 4, 1, 4, 18};
   int početÚloh = 2;

   CountDownLatch stopSignal = new CountDownLatch(početÚloh);
   ExecutorService executor
     = Executors.newFixedThreadPool(početÚloh);
   executor.submit(new Task(stopSignal, pole1));
   executor.submit(new Task(stopSignal, pole2));

   stopSignal.await();
}
```

čakáme, kým počítadlo nedosiahne nulu = kým nedobehnú úlohy

Paralelný beh a ukončenie vlákien IV

```
executor.invokeAll(Collection<Callable<T>> úlohy)
```

- exekútorovi podhodíme kolekciu úloh, ktoré vykoná
- metóda blokuje vlákno, ktoré ju zavolalo, kým nedobehnú všetky úlohy
  - úloha môže skončiť normálne
  - alebo hodiť výnimku
- po zavolaní **invokeAll()** teda čakáme, až kým neskončia všetky úlohy

Paralelný beh a ukončenie vlákien

```
// sťahujeme v piatich vláknach
Executor executor
   = Executors.newFixedThreadPool(5);
// pripravíme si zoznam úloh
Collection<Callable<Object>> tasks
   = new LinkedList<Callable<Object>>();
// stiahneme osem skladieb albumu
for(int i = 0; i < 8; i++) {
  Callable<Object> task
     = new Mp3Download("track" + i + ".mp3");
  tasks.add(task);
}
executor.invokeAll(tasks);
// výpis prebehne po dobehnutí všetkých sťahovaní
System.out.println("Album stiahnutý");
```

Paralelný beh a ukončenie vlákien IV

```
executor.invokeAny(Collection<Callable<T>> úlohy)
```

- metóda blokuje vlákno, ktoré ju zavolalo, kým nedobehne aspoň jedna úloha
  - úloha môže skončiť normálne
  - alebo hodiť výnimku
- následne sú ostatné úlohy ukončené
- po zavolaní **invokeAll()** teda čakáme, až kým neskončí aspoň 1 úloha

Prevod medzi *Runnable* a *Callable*

- *Runnable* nevracia výsledok, má metódu *run()*
- *Callable* vracia výsledok, má metódu *call()*
- ak má inštancia Runnable vrátiť po dobehnutí *null*

```
Runnable task = new VýpisSprávyTask();
Callable<Object> callableTask = Executors.callable(task)
```

- ak má vrátiť implicitnú hodnotu:

```
Callable<String> c = Executors.callable(task, "Hotovo");
// po dobehnutí vráti metóda get() reťazec Hotovo
```

Zhrnutie

- vláknové programovanie je na prvý pohľad hrozné
  - strašne zle sa ladí
  - treba myslieť súčasne na veľa vecí
  - treba pri ňom rozmýšľať – princíp "natrieskam-dajak bude" nefunguje, pretože aplikácia chvíľu beží a o chvíľu vytuhne
  - nástroje nám s ním neveľmi pomôžu
- na druhej strane sa mu nevyhneme
  - frekvencia jadier sa už zvyšovať nebude
  - budú sa len pridávať jadrá
- Java poskytuje množstvo vysokoúrovňových tried, ktoré prácu uľahčujú a abstrahujú od detailov