PROGRAMOVÉ STRUKTURY: VLÁKNA V JAVĚ

Třída Thread, rozhraní Runnable, kritická sekce, stavy vláken, balík java.util.concurrent

Java vlákna

- Vlákno v Javě = *thread*
- Paralelně proveditelné jednotky jsou objekty s metodou `run()`
- Kód metody `run()` může být prováděn souběžně s jinými takovými metodami a s metodou `main()`
- Metoda `main()` je vláknem
- Metodu `run()` spouštíme nepřímo voláním metody `start()`

Java vlákna (2)

- Definice tříd, jejichž objekty mohou mít paralelně prováděné objekty
  - jako podtřídu třídy `Thread`
    - je součástí `java.lang` balíku
    - je potomkem třídy `Object`
  - implementací rozhraní `runnable`
    - má definovánu pouze metodu `run()`
- Upřednostnit rozhraní `runnable`, pokud přepisujeme pouze metodu `run()`
- Vlákno `t` se spustí až po provedení příkazu `t.start()`

Java vlákna (3)

- **Z třídy `Thread` odvodíme potomka s metodou `run()`**

```
class MyThread extends Thread {
    public void run() {…}
…
}
```

- **Vytvoříme instanci této třídy**

```
MyThread t = new MyThread();
…
```

Java vlákna (4)

- Konstrukce třídy implementující rozhraní `runnable`

```
class MyR implements Runnable {
    public void run() {…}
…
}
```

- Konstrukce objektu této třídy (s metodou `run()`)

```
MyR m = new MyR();
```

- Vytvoření vlákna na tomto objektu

```
Thread t = new Thread(m);
```

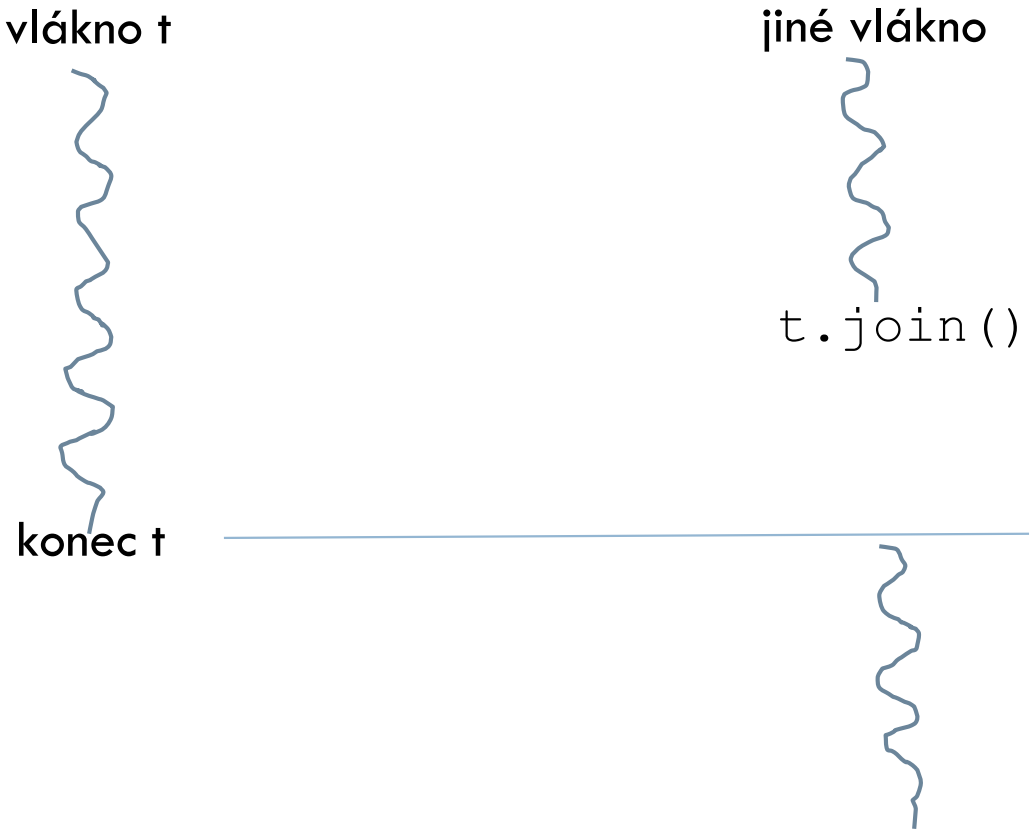
- Je zde použit konstruktor

```
Thread(Runnable threadOb)
```

Metody třídy Thread

- `final void setName(String jmeno)`
- `final String getName()`
- `final void setPriority(int priorita)`
- `final int getPriority()`
- `final boolean isAlive()`
- `final void join()`
- `static void sleep(long milisekundy)`
- `void run()`
- `void start()`

Znázornění efektu join



První ukázkové programy

Popis programu	Varianta Runnable	Varianta Thread
Main startuje jedno vlákno „potomek“, které počítá do pěti, main přitom tiskne tečky.	1R-Vlakna	1T-Vlakna
To samé, vlákno získá název až v okamžiku svého spuštění.	2R-Vlakna	2T-Vlakna
Main startuje 3 identická vlákna počítající do dvou, sám tiskne tečky a počká, až vlákna skončí.	3R-Vlakna	3T-Vlakna
To samé, main netiskne tečky a čeká na dokončení vláken voláním join().	4R-Vlakna	4T-Vlakna
Main startuje dvě vlákna s různou prioritou, která závodí v počítání do 500 a čeká na jejich dokončení voláním join().	5R-Vlakna	5T-Vlakna

Thread vs. Runnable

1T-Vlakno

```
class MyThread extends Thread {
  MyThread(String name) {
    super(name);
  }
  public void run() {
    System.out.println(getName() + " startuje.");
    …
  }
}
class Vlakno {
  public static void main(String args[]) {
    MyThread mt = new MyThread("potomek");
    mt.start();
    …
  }
}
```

1R-Vlakno

```
class MyThread implements Runnable {
  String thrdName;
  MyThread(String name) {
    thrdName = name;
  }
  public void run() {
    System.out.println(thrdName + " startuje.");
     …
  }
}
class Vlakno {
  public static void main(String args[]) {
    MyThread mt = new MyThread("potomek");
    Thread newThrd = new Thread(mt);
    newThrd.start();
    …
  }
}
```

Identifikace ukončení činnosti vláken

- Nejčastěji k zastavení dojde doběhnutím metody `run()`
- Stav lze testovat metodou `isAlive()` vracející `true`
  - provedeno new a není dead
  - modifikace předchozích programů?
- Čekáním na skončení jiného vlákna voláním `join()`
- Existuje alternativa čekání na skončení vlákna, informující, že se čeká na jeho konec
- Existuje alternativa pro timeout volání `join()`
  - `t.join(500)` – počká 500 ms, pak pokračuje dál
  - `t.join(0)` – nekonečné čekání jako `t.join()`

Volání join() vlákna t


```
final void join()
      throws InterruptedException

// rodič vytvoří a spustí vlákno t
Thread t = new Thread(m);
t.start();
// rodič něco dělá
t.join(); // rodič čeká na skončení t
// rodič pokračuje po skončení t
```

Volání interrupt() vlákna t


```
Thread t = new Thread(m);
t.start();
// rodič něco dělá
t.interrupt();
```

- Rodič oznamuje t, že na něj čeká a předčasně t přeruší. Pokud nespí, nastaví mu příznak, který lze testovat metodami
  - `boolean interrupted()` - shodí příznak
  - `boolean isinterrupted()` - neshodí příznak

```
t.join();
// rodič pokračuje po skončení t
```

Priorita vláken

- Pravděpodobnost častosti získání času procesoru
  - vysoká priorita = hodně času procesoru
  - nízká priorita = málo času procesoru
- Implicitně potomek získá stejnou prioritu jako jeho rodič
- `final void setPriority(int priorita)`
  - `Min_Priority` – hodnota 1, konstanta `Thread`
  - `Norm_Priority` – hodnota 5, konstanta `Thread`
  - `Max_Priority` – hodnota 10, konstanta `Thread`
- `final int getPriority()`
- Závisí na JVM, nemusí ji respektovat

Rychlostní závislost výpočtu

- ukázat program 6T-RZ

Kritická sekce

- Řeší problém sdílení zdrojů formou vzájemného vyloučení současného přístupu
- Metoda označená `synchronized`
  - uzamkne objekt, pro který je volána
- Jiná vlákna volající tuto metodu musí počkat ve frontě
  - zamezení interference vláken způsobující nekonzistentnosti paměti
- Po opuštění `synchronized` metody se objekt odemkne

Kritická sekce (2)

- Objekt může mít současně synchronizační i nesynchronizační metody
- Nesynchronizační metody nevyžadují zámek
- Lze provádět nesynchronized metody i na zamknutém objektu
- Každý objekt Javy je vybaven zámkem, které musí vlákno vlastnit, chce-li provést `synchronized` metodu na objektu

Kritická sekce (3)


```
Class Queue {
   …
   public synchronized int vyber(){…}
   …
   public synchronized void uloz(int co){…}
   …
}
```

Kritická sekce (4)

- Synchronizovaný příkaz

```
synchronized (výraz s hodnotou objekt) příkaz
```

- Zamkne přístup k objektu (je zadán výrazem) pro následný úsek programu
- Systém musí objekt vybavit frontou pro metody, které chtějí s ním v příkazu pracovat

Komunikace mezi vlákny

- Řeší situaci, kdy metoda vlákna potřebuje přístup k dočasně nepřístupnému zdroji

- Může čekat v nějakém cyklu
  - neefektivní využití objektu, nad nímž pracuje
- Může se zřeknout kontroly nad objektem a jiným vláknům umožnit ho používat, ale musí jim to dát na vědomí

Komunikace mezi vlákny (2)

- Metody zděděné z třídy `Object`
  - aplikovatelné pouze na `synchronized`
    - metody
    - příkazy

```
final void wait() throws InterruptedExeption
final void wait(long ms) throws …
final void wait(long ms, int nano) throws …
final void notify()
final void notifyAll()
```

Komunikace mezi vlákny (3)

- Voláním `wait()` přejde vlákno do stavu blokovaný
  - uvolní zámek objektu
  - musí být uvnitř `try` bloku
- Voláním `notify()` oživí vlákno z čela fronty na objekt
- Voláním `notifyAll()` oživí všechna vlákna nárokující si přístup k objektu
  - ta pak o přístup normálně soutěží
    - na základě priority nebo
    - plánovacího algoritmu JVM
- Obě metody mohou být volány z vláken
  - které vlastní zámek (`synchronized` metod a příkazů)
  - a jsou děděny z pratřídy `Object`

Komunikace mezi vlákny (4)

- Po volání `wait()` se vlákno zablokuje a nelze ho naplánovat, dokud nenastane některá z alternativ:

1. jiné vlákno nezavolá `notify()` pro tento objekt (vlákno se tak stane runnable)
2. jiné vlákno nezavolá `notifyAll()` pro tento objekt
3. jiné vlákno nezavolá `interrupt()` pro tento objekt a vyhodí výjimku
4. uplyne specifikovaný čas při volání `wait()`

Poznámky k synchronized

- Konstruktor nemůže být `synchronized`
  - hlásí se syntaktická chyba
  - nemělo by to ani smysl
- Jaký má efekt volání `static synchronized` metody?
  - je asociována s třídou
  - volající vlákno zabere zámek třídy (ta je považována také za objekt) a má pak výlučný přístup ke statickým položkám třídy
  - tento zámek nesouvisí se zámky instancí této třídy

Poznámky k synchronized (2)

- Vlákno nemůže zabrat zámek, který vlastní již jiné vlákno
- Vlákno může opakovaně zabrat zámek, který již samo vlastní
  - reentrantní synchronizace
  - nastává tehdy, když `synchronized` kód vyvolá metodu, která také obsahuje `synchronized` kód
    - oba používají tentýž zámek

Poznámky k synchronized (3)

- Atomické akce jako např. read a write proměnných deklarovaných jako `volatile` (= nestálé) nevyžadují synchronizaci.
- Vlákno si pak nesmí tvořit jejich kopii
  - z optimalizačního důvodu to normálně dělat může
- Pokud hodnota proměnné neovlivňuje stav jiných proměnných včetně sebe, pak se nemusí synchronizovat.
- Jejich použití je časově úspornější.
- Balík `java.util.concurrent` poskytuje i metody, které jsou atomické.

Semafor jako ADT v Javě


```
class Semafor {
  private int count;
  public Semafor(int initialCount) {
    count = initialCount;            // když je 1, je to binární semafor
  }
  public synchronized void cekej( ) {
    try {
      while (count <= 0 ) wait( );   // musí být nedělitelné
      count--;                       // nad instancí semaforu
    } catch (InterruptedException e) { }
  }
  public synchronized void uvolni( ) {
    count++;
    notify( );
  }
}
```

Odstranění rychlostní závislosti

- ukázat program 7T-Monitor

Rekapitulace

- Každé vlákno je instancí třídy `java.lang.Thread` nebo jejího potomka
- Třída `Thread` má metody:
  - `run()` – je vždy přepsána v potomku `Thread`
  - `start()` – spustí vlákno, metoda `run()` není přímo spustitelná
  - `yield()` – odevzdání zbytku přiděleného času a zařazení do fronty na procesor
  - `sleep(ms)` – zablokování vlákna na daný čas
  - `isAlive()` – běží-li vlákno, vrací `true`, jinak `false`
  - `join()`
  - `getPriority()`
  - `setPriority()`
  - … a cca 20 dalších

Rekapitulace (2)

- Objekt má metody, které `Thread` dědí:
- `final void notify()`
  - oživí vlákno z čela fronty na čekající na objekt
- `final void notifyAll()`
  - oživí všechna vlákna nárokující si přístup k objektu
- `final void wait() throws InterruptedExeption`
  - vlákno čeká, až jiné zavolá `notify()` či `notifyAll()`
- `final void wait(long)throws …`
  - vlákno čeká na `notify()` či `notifyAll()` nebo vypršení specifikovaného času

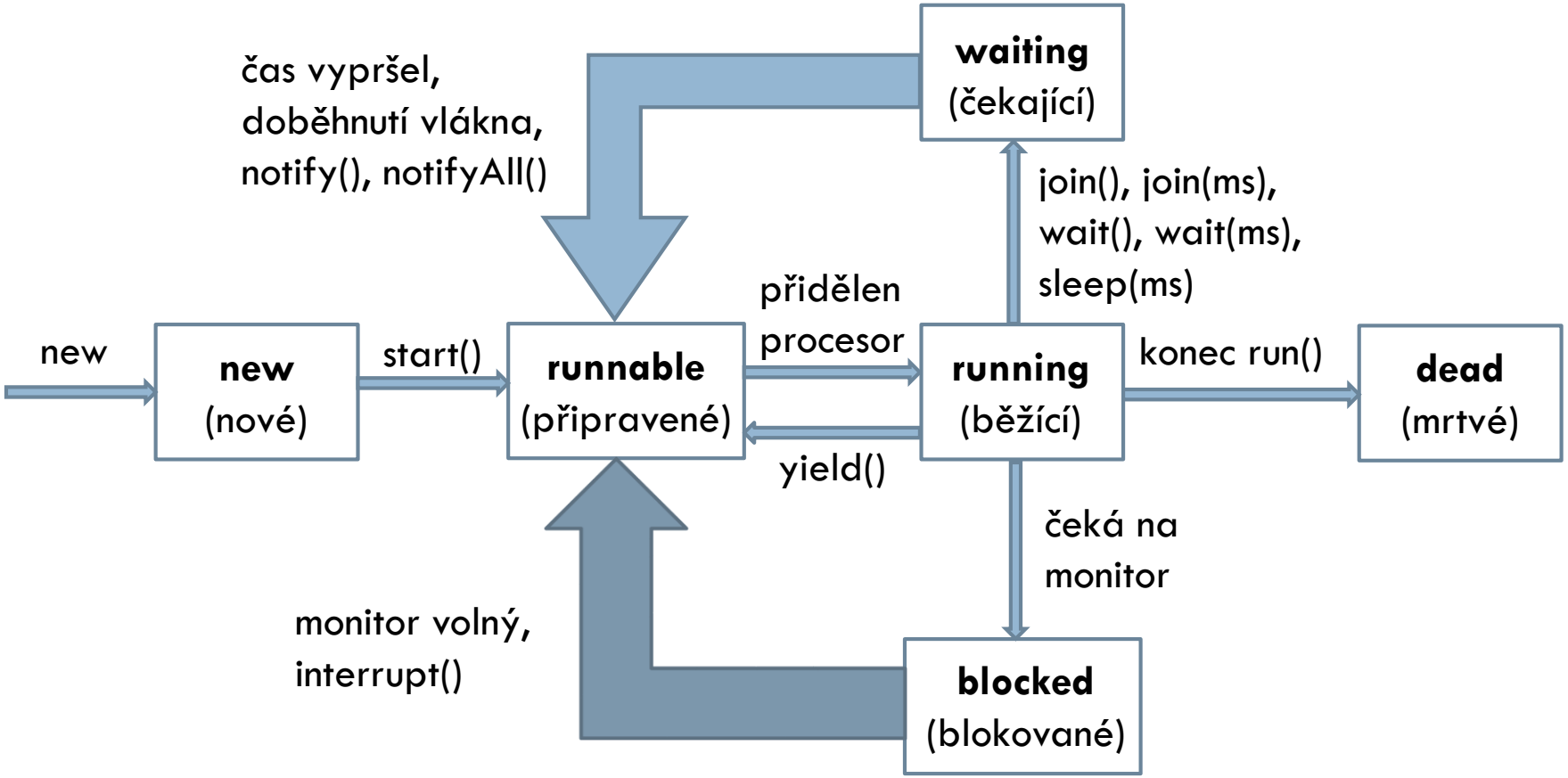
Stavy vláken

- Nové
  - ještě nezačalo běžet
- Připravené
  - nemá přidělený procesor
  - plánovač vybere z fronty připravených vlákno s nejvyšší prioritou
- Běžící
  - má přidělený procesor

Stavy vláken (2)

- Blokované
  - čeká ve frontě na monitor
- Čekající
  - provedlo volání `wait()`, `join()`, ...
- Časově čekající
  - provedlo volání `wait(ms)`, `sleep(ms)`, `join(ms)`, ...
- Mrtvé
  - už doběhlo

Přechody mezi stavy vlákna Javy



Příklad producent a konzument

- ukázat příklad 8T-PC

Synchronized příkazy


```
class Konto {
  private int konto;                    private Object zamek = new Object();
  public int stav() {return konto;}     public Konto(int i) { konto = i;}
  public void vyber(int kolik) {
    int lokal;
    synchronized (zamek) {
      try { lokal = konto;  Thread.sleep(100);  konto = lokal - kolik;
      } catch (InterruptedException e) {}
    }
  }
  public void vloz(int kolik) {
    int lokal;
    synchronized (zamek) {
      try { lokal = konto;  Thread.sleep(300);  konto = lokal + kolik;
      } catch (InterruptedException e) {}
    }
  }
}
```

Zavržené metody

- `final void suspend()`
  - pozastavení vlákna, kterému pošleme suspend
- `final void resume()`
  - obnovení vlákna, kterému zašleme resume
- `final void stop()`
  - ukončení vlákna, kterému zašleme stop
- Důvod zavržení
  - nebezpečné konstrukce, které snadno způsobí deadlock, když se aplikují na objekt, který je právě v monitoru

Zavržené metody (2)

- Lze je nahradit bezpečnějšími konstrukcemi s `wait()` a `notify()`
- Zavedeme `boolean` proměnnou se jménem `susFlag` inicializovanou `false`
- V metodě `run()` suspendovaného vlákna `synchronized` příkaz tvaru:

```
synchronized(this) {
  while (susFlag) {
    wait();
  }
}
```

Zavržené metody (3)

- Příkaz `suspend()` nahradíme voláním metody `mojeSuspend()` tvaru:

```
void mojeSuspend() {
  susFlag = true;
}
```

- Příkaz `resume()` nahradíme voláním metody:

```
synchronized void mojeResume() {
  susFlag = false;
  notify();
}
```

Vlákna typu démon

- Uživatelská vlákna – nejsou démonem
- Vlákna typu démon (daemon)
  - vytvoříme voláním `setDaemon()` ze třídy `Thread`
  - volat ještě před spuštěním vlákna metodou `start()`
  - je to natrvalo
  - metoda `boolean isDaemon()` testuje charakter vlákna
- Ukončí se, jakmile žádné uživatelské vlákno neběží
- Používá se u aplikací
  - prováděných na pozadí
  - nepotřebujících po sobě uklízet

Vlákna typu démon (2)

- JVM spustí jedno uživatelské vlákno – `main()`
  - ostatní vytvářená vlákna mají typ a prioritu toho vlákna, ze kterého jsou spouštěna
    - pokud to nezměníme voláním `setPriority()` či `setDaemon()`
- JVM provádí výpočet, dokud nenastane:
  - volání metody `exit()` ze třídy `Runtime`
  - všechna uživatelská vlákna jsou ve stavu „mrtvý“, když:
    - dokončila metody `run()`
    - vyhodila výjimku
- JVM brání uživatelským vláknům v ukončení běhu
- JVM nebrání vláknům typu démon skončit

Skupiny vláken

- Každé vlákno je členem skupiny, což dovoluje manipulovat skupinou jako jedním objektem
  - lze všechny odstartovat jediným voláním metody `start()`
- Skupiny lze implementovat pomocí třídy `ThreadGroup` z `java.lang`
- JVM začne výpočet vytvořením skupiny `main`
- Nespecifikuje-li se jiná, jsou všechna vytvářená vlákna členy skupiny `main`

Skupiny vláken (2)

- Členství ve skupině je natrvalo
- Vlákno lze začlenit do skupiny

```
ThreadGroup jinaSkupina =
  myThread.getThreadGroup();
```

- Vrací jméno skupiny, ke které patří `myThread`

Balík java.util.concurrent

- Vestavěná primitiva Javy nestačí k pohodlné synchronizaci, protože
  - neumožňují couvnout po pokusu o získání zámku, který je zabrán
  - neumožňují couvnout po vypršení času, po který je vlákno ochotno čekat na uvolnění zámku
    - nedovolují provést alternativní činnost
  - nelze změnit sémantiku uzamčení s ohledem na
    - reentrantnost
    - ochranu proti čtení versus psaní

Balík java.util.concurrent (2)

- Vestavěná primitiva Javy nestačí k pohodlné synchronizaci, protože:
  - neřídí přístup k synchronizaci, každá metoda může použít blok `synchronized` na libovolný objekt

```
synchronized (referenceNaObjekt) {
  // kritická sekce
}
```

  - nelze získat zámek v jedné metodě a uvolnit ho v jiné

Balík java.util.concurrent (3)

- Poskytované třídy a rozhraní
- Iterface Executor

```
public interface Executor {
   void execute(Runnable r);
}
```

- Executor může být jednoduchý interface
  - dovoluje vytvářet systém pro plánování, řízení a exekuci množin vláken

Balík java.util.concurrent (4)

- Paralelní kolekce
  - implementující Queue, List, Map
- Atomické proměnné
  - třídy pro bezpečnější komunikaci s proměnnými (primitivních typů i referenčních) efektivněji než pomocí synchronizace
- Synchronizační třídy
  - semafory, bariéry, závory (latches) a výměníky (exchangers)
- Zámky
  - jejich implementace dovoluje specifikovat timeout při pokusu získat zámek a dělat něco jiného, když není volný
- Nanosekundová granularita
  - jemnější čas

Třída ReentrantLock

- Konstruktory
  - `ReentrantLock()`
  - `ReentrantLock(boolean fair)`
    - hodnota `true` zajišťuje férové chování, nepředbíhá
- Metody třídy
  - `int getHoldCount()`
    - kolikrát drží zámek dané vlákno
  - `int getQueueLength()`
    - kolik vláken chce tento zámek
  - `protected Thread getOwner()`
    - vrátí vlákno, které vlastní zámek, nebo `null`

Třída ReentrantLock (2)

- Metody třídy (pokračování)
  - `boolean hasQueuedThread(Thread thread)`
    - čeká zadané vlákno na tento zámek?
  - `void lock()`
    - získání zámku, není-li volný, musí čekat
  - `void unlock()`
    - uvolnění zámku
  - `boolean tryLock()`
  - `boolean tryLock(long t-o, TimeUnit unit)`
    - je-li zámek volný (během timeoutu), získá zámek, jinak nic
  - cca. 20 dalších metod

Třída Semaphore

- Konstruktory
  - `Semaphore(int povoleni)`
  - `Semaphore(int povoleni, boolean f)`
    - `povoleni` udávají počet zdrojů
    - `f` určuje fér chování při hodnotě `true`
- Získání povolení
  - `void acquire()` – pro jedno povolení
  - `void acquire(int povoleni)`
- Tyto metody blokují vlákno, dokud počet povolení (zdrojů) není k dispozici nebo dokud čekající vlákno není přerušeno vyhozením `InterruptedException`

Třída Semaphore (2)

- Získaní povolení (pokračování)
  - `void acquireUninterruptibly()` – pro jedno povolení
  - `void acquireUninterruptibly(int povoleni)`
- Metody pozastavují vlákno, které čeká nepřerušitelně až do získání potřebného počtu povolení
- Případné požadované přerušení se projeví až po získání povolení

Třída Semaphore (3)

- Získání povolení (pokračování)
  - `boolean tryAcquire()`
  - `boolean tryAcquire(int povoleni)`
  - `boolean tryAcquire(long timeout, TimeUnit unit)`
  - `boolean tryAcquire(int povoleni, long timeout, TimeUnit unit)`
- Získání povolení, je-li zjištěn potřebný počet volných
- Nezablokování exekuce vlákna

Třída Semaphore (4)

- Uvolnění povolení
  - `void release()` – pro jedno povolení
  - `void release(int povoleni)`
- Uvolní zadaný počet povolení

- Př. 10 sekund čekání na jedno povolení:

```
boolean z = tryAcquire(10, TimeUnit.SECONDS)
```

Třída CyclicBarrier

- Dovoluje čekání množiny vláken na sebe navzájem před pokračováním výpočtu
- Nazývá se cyklická, protože může být znovu použita po uvolnění čekajících vláken
- Obvykle je použita, když úloha je rozdělena na podúlohy takové, že každá z nich může být prováděna separátně

Třída CyclicBarrier (2)

- Konstruktory
  - `CyclicBarrier(int ucastnici)`
  - `CyclicBarrier(int ucastnici, Runnable akce)`
- `ucastnici` určují počet podúloh = vláken
- `akce` se provede
  - po spojení všech vláken
  - před jejich další exekucí

Třída CyclicBarrier (3)

- Metody cyklické bariéry
  - `int await()`
    - čeká, až všichni účastníci vyvolají `await()` na této bariéře
  - `int await(long timeout, TimeUnit unit)`
    - čeká, až všichni účastníci vyvolají `await()`, maximálně do vypršení timeoutu
  - `int getNumberWaiting()`
  - `int getParties()`
    - počet účastníků, kteří prolomí bariéru
  - `boolean isBroken()`
    - bariéra porušená timeoutem, přerušením, resetem, výjimkou
  - `void reset()`
    - resetuje porušenou bariéru

Třída CountDownLatch

- Česky závora
- Synchronizační prostředek, který dovoluje vláknu/vláknům čekat až se dokončí operace v jiných vláknech
- Inicializuje se se zadaným čítačem, který je součástí konstruktoru a funguje obdobně jako počet účastníků v `CyclicBarrier` konstruktoru
- Určuje, kolikrát musí být vyvolána metoda `countDown()`
- Po dosažení zadaného počtu jsou všechna vlákna čekající v důsledku vyvolání metody `await()` uvolněna k exekuci

Třída CountDownLatch (2)

- Konstruktor
  - `CountDownLatch(int pocet)`
    - nastavujeme závoru, na kolik vláken bude čekat
- Metody závory
  - `void await()`
    - čekání volajícího vlákna až do vynulování čítače
  - `boolean await(long timeout, TimeUnit unit)`
    - čeká se maximálně do vypršení timeoutu
  - `void countDown()`
    - dekremenatace čítače, při nule uvolní čekající vlákna
  - `long getCount()`
    - hodnota čítače
  - `String toString()`
    - identifikace závory s čítačem

Třída Exchanger<V>

- Česky výměník
- Umožňuje komunikaci vláken předáváním objektu `V`
- K předání objektů je volána metoda `exchange()`, která je obousměrná, vlákna si navzájem předají data
  - k výměně dojde, když obě vlákna již dosáhla místo, kde volají metodu `exchange()`
- Typickým příkladem použití je úloha producenta konzumenta
  - nekomunikují plněním a vyprazdňováním jednoho (kruhového) bufferu
  - vymění si buffery celé (prázdný za plný a opačně).

Třída Exchanger<V> (2)

- Konstruktor
  - `Exchanger()`
    - nový výměník pro objekt typu V
- Metody výměníku
  - `V exchange(V object)`
    - čeká na jiné vlákno, se kterým chce provést výměnu
    - až se vlákna sejdou, je výměna provedena
  - `V exchange(V object, long timeout, TimeUnit unit)`
    - výměna se provede maximálně do vypršení timeoutu