DISTRIBUOVANÉ PROGRAMOVÁNÍ (NTI/DPG)

Co se nikam nevešlo

Ing. Igor Kopetschke – TUL, NTI

http://www.nti.tul.cz

TECHNICKÁ UNIVERZITA V LIBERCI

Zámky (Lock)

- Pro exkluzivitu přístupu do kritických sekcí jsme zatím používali **monitor** a **synchronized**
- Od verze 5 Java podporuje tzv. **explicitní zámky**
- Součástí balíku **java.util.concurrent.locks**
- Umožňují mnohem větší kontrolu a flexibilitu
- Negativem je složitější kód
- Obecně tyto zámky implementují rozhraní **Lock**
- Výhodné při větším množství jednodušších procesů či vláken
- Rozhraní **Lock** předepisuje následující metody

Zámky (Lock)

- Rozhraní Lock předepisuje mj. tyto základní metody

- **`void lock()`**
  - Požádá o zámek. Není-li dostupný, zařadí vlákno mezi čekající
- **`void unlock()`**
  - Uvolní zámek
- **`boolean tryLock()`**
  - Požádá o zámek, v případě neúspěchu na něj nečeká a vrátí **false**
  - Další podoba metody definuje čas čekání

Zámky (Lock)

- Pro standardní použití se užívá implementační třída **ReentrantLock**
- Vlákno vlastnící zámek o něj může žádat opětovně
- Zámek ovšem registruje počet žádostí z vlákna ..
- A vyžaduje aby vlákno provedlo stejný počet uvolnění
- Až potom je zámek skutečně uvolněn pro ostatní vlákna
- Uvolnění vlákna je nutno zabezpečit tak, aby jej nevyřadilo vyhození výjimky
- Vhodným místem je tedy blok **finally**

Zámky (Lock)

- **Výhody** :
  - lepší kontrola nad uzamykáním prostředků a kritických sekcí
  - řešení překrývání kritických sekcí. Například když nastane situace, kdy uvolnění jedné KS je možné až po vstupu do jiné kritické sekce
  - Férové přidělení zámku - **boolean** v konstruktoru
  - Je-li **true**, zámek je přidělen vláknu, které požádalo jako první
  - Je-li **false**, přidělení zámku je náhodné
- **Doporučení**:
  - Instanci zámku jako **final** – ochrana před podstrčením jiné instance

Zámky (ReadWriteLock)

- **Dvojitý zámek**:
  - V praxi často situace, kdy současné čtení nevadí, problém nastane pouze při současném **read / write**
  - Řešení nabízí implementace rozhraní **ReadWriteLock** a její metody
  - `Lock readLock()` – zámek pro vlákna, které chtějí číst
  - `Lock writeLock()` – zámek pro vlákna, které chtějí zapisovat
  - Čtecí zámek připustí **více** vláken najednou
  - Zápisový zámek pouze **jedno** vlákno
  - Implementační třída **ReentrantReadWriteLock**

Vláknově bezpečné objekty a kolekce

- Používají se v paralelním prostředí
- Zabezpečení **atomicity** a ochrana konzistence dat
- Atomické objekty
  - **java.util.concurrent.atomic**
  - AtomicInteger, AtomicIntegerArray, AtomicLong, AtomicLongArray, AtomicBoolean …
- Thread-safe kolekce
  - Původní třídy **Vector** a **Hashtable** – synchronizované, ale pomalé, dnes **zavržené**
  - třída **Collections** a její statické metody
    - `synchronizedCollection(Collection c)`
    - `synchronizedMap(Map m)`
    - `synchronizedSet(Set s)`
    - `synchronizedList(List l)`
  - Metody vrací kontejnery s vláknově zabezpečenými metodami, není zabezpečen **iterátor**

Vláknově bezpečné objekty a kolekce

- Použití kolekcí z balíku java.util.concurrent
- **ConcurrentLinkedQueue<E>**
  - Fronta synchronizující čtení a zápis
  - Neomezená velikost – pomalá metody size() – počítá prvky
  - na rozdíl od **Vector** neblokuje celou kolekci, iterátor není citlivý na paralelní změnu fronty
  - Iterátor je slabě konzistentní – pracují s obsahem platným v době aktivace a nemusí nutně respektovat následné změny
  - Iterátor nevyhodí výjimku ConcurrentModificationException
  - Podporuje více producentů a konzumentů
- Alternativou **LinkedBlockingQueue<E>**
  - implementace BlockingQueue
  - umožňuje čekat na položku (priorita, čas)

Vláknově bezpečné objekty a kolekce

- **ConcurrentHashMap<K,V>**
  - Synchronní přístup pro neomezený počet read/write vláken
  - Na rozdíl od Hashtable **neblokuje** celou kolekci
  - Počet write vláken je možno nastavit jako **concurrencyLevel**
  - Malá hodnota – zdržování, velká hodnota – vysoká zátěž
  - Pro write operace používá atomické operace
- **CopyOnWriteArrayList<E>**
  - vláknově bezpečný kontejner
  - metody, které provádí změnu, vytváří kopii pole
  - užitečné pouze tehdy, je-li kontejner často iterován a počet iteračních vláken výrazně převyšuje počet write vláken
  - pokud vlákno změní obsah zrovna iterovaného kontejneru, nic se neprojeví, změna se promítne pouze v kopii objektu, procházené pole je nezměněno

.. A to je pro dnešek vše

DĚKUJI ZA POZORNOST