Java Concurrency

Kozsik Tamás

kto@elte.hu
http://kto.web.elte.hu/

Eötvös Loránd Tudományegyetem
Programozási Nyelvek és Fordítóprogramok Tanszék

2010.

Tartalom

Tartalom

Tartalom

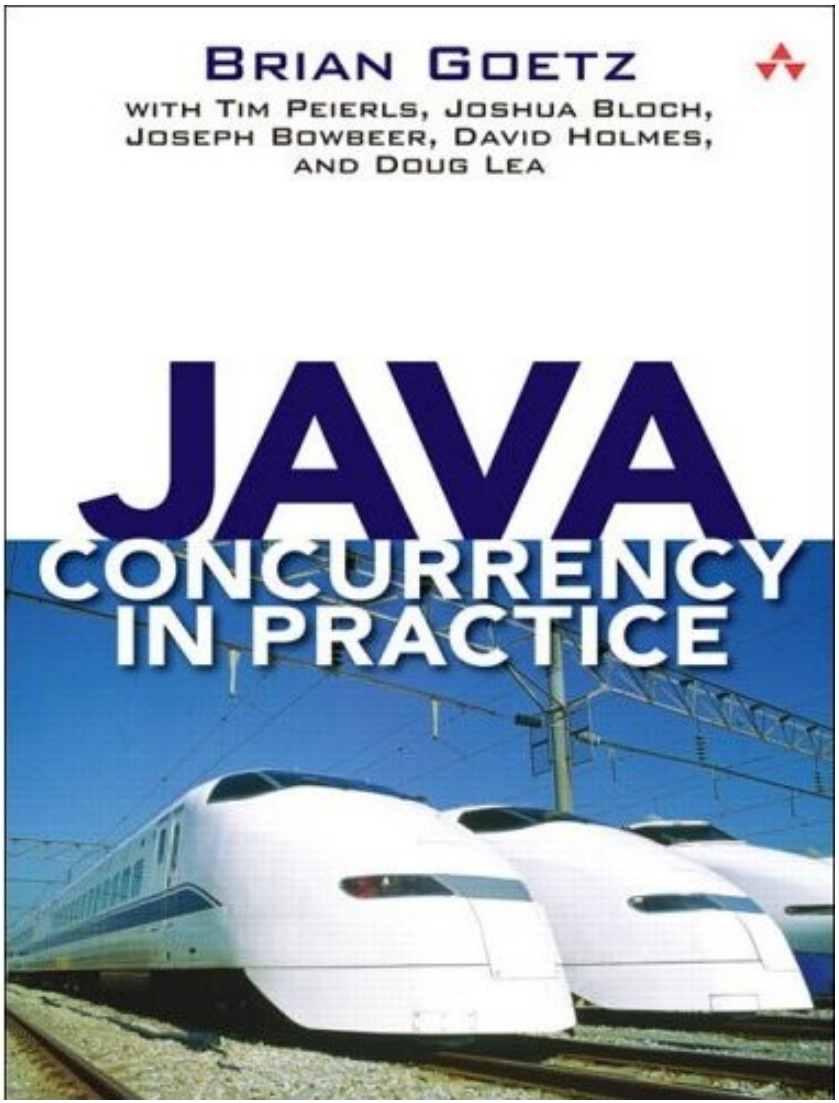
Tartalom

Tartalom

Tartalom

Tartalom

Egy hasznos könyv



A példák letölthetők
(creative commons)

Miért kell többszálúság?

- Eredendően konkurrens feladatok megoldása
- Hatékonyság növelése
  - Kiszolgálási idő
  - Válaszidő
  - Áteresztőképesség
  - Skálázódás

Szálkezelés alapok

- `Thread`, `Runnable`
- Szálak életciklusa
- Ütemezés (preemptív, `yield`)
- Blokkoló műveletek, `sleep`
- Kommunikáció közös változókon keresztül
- `synchronized` metódus és blokk
- `volatile` változók
- `wait-notify`
- Stream-alapú kommunikáció (`Piped`-streamek)

Java Concurrency API

- `java.util.concurrent`
- `java.util.concurrent.locks`
- `java.util.concurrent.atomic`

- Megjelenés: JDK 1.5
- Néhány újdonság: JDK 1.6
- Fork-join modell: JDK 1.7
- Párhuzamos streamek: JDK 1.8

Alapelvek

- A helyes működést nem áldozzuk fel
- Az egyszerűséget és karbantarthatóságot csak okkal áldozzuk fel
  - Nem optimalizálunk idő előtt
  - Egymásnak ellentmondó szempontok
  - Profilozás

Helyesség

Hibás az a program, amely szinkronizáció nélkül oszt meg módosuló állapotot szálak között.

- Ne osszunk meg a változót szálak között
- A változó legyen módosíthatatlan
- Használjunk szinkronizációt

Szinkronizáció

- Atomicitás/kizárás (JCIP 2.2)
- Láthatóság (JCIP 3.1)

Szálbiztosság

- Szálbiztos osztály: nem hibásabb konkurrens szituációban, mint egyszálú környezetben
- Szálbiztos objektum használata nem igényel további szinkronizációt
- Állapotmentes objektum szálbiztos
- Módosíthatatlan objektum szálbiztos
- Race condition (JCIP 2.3) kiküszöbölendő
- Atomi check-then-act (zárolás, `java.util.concurrent.atomic`)

Atomi csomagolás

- `java.util.concurrent.atomic`
- Példa (JCIP 2.4)
- Ellenpélda: invariáns összes változója... (JCIP 2.5)

Zárolás

- Megosztott adathoz való minden hozzáférés ugyanazzal a lockkal
- Invariánsban szereplő összes változót ugyanazzal a lockkal
- Ne tartson indokolatlanul hosszú ideig (JCIP 2.6 és 2.8)

Hogyan lehet?

```
A = B = 0
```

```
r2 = A
B = 1
```

```
r1 = B
A = 2
```

```
r2 == 2  &  r1 == 1
```

Meglepő eredmények

- Kód átrendezése
- Cache-elés (fordító és processzor)
- `long` és `double` esetén nem atomi az olvasás/írás
- Word tearing nem engedett

Java Language Memory Model

- Többszálú Java program jelentése
- Optimalizációk hatásai
- Egyszálú program – értelemszerű (szálon belüli sorrendiség)
- Több szálból használt változók elérése
  - olvasás–írás konfliktus
  - írás–írás konfliktus

Egy program lehetséges (legális) végrehajtásai

- Események sorrendisége: happens-before reláció
- Írt és olvasott értékek viszonya, konzisztencia
- Oksági viszony

A happens-before reláció

- Programszövegbeli sorrendiség (részbenrendezés) – po
- Szinkronizációs események sorrendje adott (teljes rendezés)
- Synchronizes-with reláció (részbenrendezés) – sw
- $\mathrm{hb} = (\mathrm{po} \cup \mathrm{sw})^+$

- Szekvenciális konzisztencia – túl korlátozó

„Helyes” program nem tartalmaz data race-t

Data race egy adott végrehajtásban

Ha konfliktusban álló változóelérések nincsenek happens-before relációban

- Jól szinkronizált a program $\Leftrightarrow$ szekvenciálisan konzisztens végrehajtás nem tartalmaz data race-t
- Ha nincs data race, minden végrehajtás szekvenciálisan konzisztensnek tűnik
- Java 5: Nem jól szinkronizált programnak is kell jelentést adni (biztonság)

Biztonság

```
A = B = 0
```

```
r2 = A          r1 = B
B = r2          A = r1
```

```
r2 == 42  &  r1 == 42
```

- Ezt nem szeretnénk
- Rosszul szinkronizált programban se legyen légbőlkapott érték
- Security mechanizmusok kikerülése nem kívánatos

Volatile

- Olvasás és írás: szinkronizációs események
- Láthatóság biztosítása
- Konzisztens (nem elévült) érték minden szálból
- `long` és `double` esetén nem lesz „légbőlkapott” érték
- A `-server` kapcsoló...
- Példa: egy szálból írjuk (ne legyen race condition), több szálból olvashatjuk

Példa

```
import java.applet.Applet;
import java.awt.Graphics;
public class Animation extends Applet implements Runnable {
  private volatile boolean running = false;
  public void start(){
    running = true;
    new Thread(this).start();
  }
  public void stop(){ running = false; }
  private int x = 0;
  public void run(){
    while( running ){
      x = x>100 ? 0 : x+1;
      repaint();
      try{ Thread.sleep(20); }catch(InterruptedException e){}
    }
  }
  public void paint( Graphics g ){ g.drawLine(0,30,x,30); }
}
```

Típusinvariáns megőrzése

- Belső állapot ne szökjön ki (JCIP 3.6)
  - Paraméter „idegen” műveletnek
- Biztonságos létrehozás
  - Ne szökjön ki a `this` (JCIP 3.7 és 3.8)
- Biztonságos nyilvánosságra hozás (JCIP 3.14 és 3.15)
  - Módosíthatatlan mező
  - Módosíthatatlan objektum

Módosíthatatlan mező nyilvánosságra hozása

```
class FinalFieldExample {
        final int x;
        int y;
        static FinalFieldExample f;
        public FinalFieldExample() {
                x = 3;
                y = 4;
        }
        static void writer() {
                f = new FinalFieldExample();
        }
        static void reader() {
                if (f != null) {
                  int i = f.x; // guaranteed to see 3
                  int j = f.y; // could see 0
                }
        }
}
```

Módosíthatatlan mező nyilvánosságra hozása

```
class FinalFieldExample {
        final int x = 3;
        int y = 4;
        static FinalFieldExample f;
        static void writer() {
                f = new FinalFieldExample();
        }
        static void reader() {
                if (f != null) {
                  int i = f.x; // guaranteed to see 3
                  int j = f.y; // could see 0
                }
        }
}
```

Módosíthatatlan objektum

Módosíthatatlan

- Helyesen konstruált (nem szökik ki a `this`)
- Az állapot nem módosulhat létrehozás után
- A mezői `final`ok (mondjuk...)

- Szálbiztos
- A nyilvánosságra hozása sem igényel szinkronizációt
- Nem olyan költséges, mint gondolnánk (JCIP 3.12 és 3.13)

Final változó lehet elévült?

```
class Trivial {
        int x;
        Trivial(){
                System.out.println(x);
                x = 1;
        }
        public static void main( String[] args ){
                System.out.println( new Trivial().x );
        }
}
```

Final változó lehet elévült?

```
class CompileError {
        final int x;
        CompileError(){
                System.out.println(x);
                x = 1;
        }
        public static void main( String[] args ){
                System.out.println( new CompileError().x );
        }
}
```

Final változó lehet elévült?

```
class EscapingThis {
        static EscapingThis o;
        final int x;
        EscapingThis(){
                o = this;
                System.out.println(o.x);
                x = 1;
        }
        public static void main( String[] args ){
                System.out.println( new EscapingThis().x );
        }
}
```

Biztonságos nyilvánosságra hozás

- Adott egy helyesen (biztonságosan) konstruált objektum;
- A referencia és az objektum állapota egyszerre váljon nyilvánossá
  - Statikus inicializátor
  - `volatile` mező vagy `AtomicReference`
  - Egy helyesen konstruált objektum `final` mezője
  - Zárolással megfelelően védett mező

Szálra korlátozás

- Lokális változók használata (JCIP 3.9)
- `ThreadLocal` (JCIP 3.10)
- Ad-hoc megoldások: veszélyes (pl. Swing használata)

Objektumra korlátozás

- Szálbiztos osztály definiálásához az állapot
- Például: Java monitor (JCIP 4.4 és 4.5)
- `deepCopy`: nem elég az `unmodifiableMap` vagy sekély másolat (copy konstr.)
- Hatékonyság: mi van, ha nagyon sok jármű van?
- Szemantikus kérdés: konzisztens pillanatfelvétel történik

Konkurrens használatra tervezett adatszerkezettel

- Például `ConcurrentHashMap`pel (JCIP 4.6 és 4.7)
  - Módosíthatatlan `Point`: szálbiztosan használható
  - Szemantikus kérdés: nem feltétlenül konzisztens, de frissülő nézet (v.ö. JCIP 4.8)
  - Másik megoldás: nyilvánossá tett módosítható pontokkal (JCIP 4.11 és 4.12)
- Vagy `CopyOnWriteArrayList`tel
- Vagy az `Atomic` osztályokkal

Szálbiztos osztály kiterjesztése

- Származtatás (JCIP 4.13): nehezebb megőrizni a szinkronizációs működést
- Kliensoldali zárolás (JCIP 4.14 és 4.15)
- Becsomagolás/monitor (JCIP 4.16)

Szinkronizált adatszerkezetek

- Eleve szinkronizált: `Vector` és `Hashtable`
- Szinkronizációs burokba zárt: `Collections.synchronizedXxx`

- Kliensoldali zárolás összetett tevékenység esetén - gyakori
- Külső iterátor és `ConcurrentModificationException`
  - Zárolás és belső iterátor: túl hosszú lehet
  - Másolat létrehozása?
  - Rejtett iterálás: `toString` (JCIP 5.6), `hashCode`, `equals`, `containsAll`, `removeAll`, `retainAll`

ConcurrentHashMap

- `Map`
- Finomabb zárolás, nagyobb egyidejűséget enged, jól skálázódik
- `ConcurrentMap`: atomi `putIfAbsent`, `remove`, `replace` check-then-act műveletek
- Iterálás: gyengén konzisztens
  - Nincs `ConcurrentModificationException`
  - Vagy a létrehozás pillanatában létező állapot, vagy esetleg módosuló nézet
- `size` és `isEmpty` csak becslés
- Nem jó akkor, ha az egész adatszerkezetet kívánjuk zárolni

CopyOnWriteArrayList és CopyOnWriteArraySet

- `List`, illetve `Set`
- Hatékony, ha sok az iterálás és kevés a módosítás
- A változtató műveletek új tömböt hoznak létre benne
- Az iterálás triviálisan szálbiztos (pillanatfelvétel)
- Változtatás iterátoron keresztül: `UnsupportedOperationException`

ConcurrentSkipListMap és ConcurrentSkipListSet

- A JDK 6 hozta be
- Hatékony (átlagosan logaritmikus műveletek, nagy egyidejűség)
- Gyengén konzisztens iterálás
- `ConcurrentMap` műveletek
- (`Concurrent`)`NavigableMap` és `NavigableSet`
  - Szintén JDK 6
  - `SortedMap` és `SortedSet` leszármazottjai
  - Keresés: megadott értékhez legközelebbi kisebb/nagyobb
- Tömeges műveltek nem atomiak
- `size` nem konstans idejű

BlockingQueue

- `Queue`: közvetlen elérést nem támogat
- Többféle betevő/kivevő műveletek
  - Hagyományos (kivétel)
  - Speciális értéket használ
  - Blokkolódik
  - Időkorlátosan blokkolódik
- Megvalósítások
  - Tömbbel (korlátos méretű)
  - Láncoltan
  - Prioritással (nem FIFO)
  - CSP/Ada randevú: `SynchronousQueue` (kérhető FIFO)
- Specializáció: BlockingDeque (JDK 6)

Termelő-fogyasztó feladat

- Könnyen megvalósítható `BlockingQueue` segítségével
- Persze másképp, pl. `wait-notify` segítségével is
- `Executor`: thread pool + work queue
- Korlátozott méretű buffer: memória védelme
- Példa: JCIP 5.8 és 5.9
- Work stealing: minden fogyasztónak egy `BlockingDeque`

Szinkronizáló osztályok

- Amivel meg lehet várakoztatni folyamatokat
- Például `BlockingQueue`
- Latch, Semaphore, Barrier

Latch

Várakozás egy eseményre...

- Amíg zárva van, feltartja a folyamatokat
- Ha elér a végállapotba, kinyílik
- Ezután minden folyamat (tovább)mehet

- `CountDownLatch`
- `FutureTask` – egy kis ízelítő...
  - Inicializáció egy elvégzendő számítással: `Runnable` vagy `Callable`
  - `Future`: aszinkron számítás eredménye, bevárása: `get`
  - `Runnable` is
  - JCIP 5.12 és 5.13

Szemafor

- `Semaphore`
- Bináris szemafor
- Általánosított szemafor
- Nem csak az a szál `release`-elhet, amelyik `acquire`-t mondott

- Resource pool megvalósítása: blokkol, ha kiürült
- Korlátozott méretű adatszerkezet (JCIP 5.14)

Barrier

Várakozás a többi szálra...

- Akkor mehet tovább az összes várakozó, ha mindenki „odaért”
- Lépésekre bontott számítás, ahol egy lépés konkurrensen hajtódik végre: `CyclicBarrier` (JCIP 5.15)
  - Létrehozáskor lerögzített számú folyamatnak
  - Mindig újraindul
  - `await` egyedi érkezési azonosítót ad (vezetőválasztáshoz jó)
  - Opcionális barrier action létrehozásnál
- Két folyamat szikronizált adatcseréje: `Exchanger`

Feladat

- Az elvégzendő munka egy egysége
- Konkurrens végrehajtás szálakban
- A feladatok méretének megválasztása kritikus
  - Feladat = szál költséges lehet
  - Túl kicsi feladatok: versengés, ütemezési overhead
  - Túl nagy feladatok: kismértékű konkurrencia
- Feladatok és szálak explicit összekapcsolása: skálázási problémák

Executor

java.lang.Runnable

```
void run()
```

java.util.concurrent.Callable<V>

```
V call() throws java.lang.Exception
```

java.util.concurrent.Future<V>

```
V get() throws InterruptedExc., ExecutionExc.
```

...

implementáció: `FutureTask<V>(Callable<V>)`

java.util.concurrent.Executor

```
void execute(Runnable) throws RejectedExecutionExc.
```

Megvalósítás

- `Executors` factory osztály
  - `newCachedThreadPool()`
  - `newFixedThreadPool(int nThreads)`
  - `newSingleThreadExecutor()`
- Példák (JCIP 6.1, 6.2, 6.4)
- Saját megvalósítás is lehet (JCIP 6.5, 6.6)

Életcikluskezelés

java.util.concurrent.Executor

```
void execute(Runnable) throws RejectedExecutionExc.
```

java.util.concurrent.ExecutorService extends Executor

```
void shutDown()
List<Runnable> shutDownNow()
boolean isShutDown()
boolean isTerminated()
boolean awaitTermination(long, TimeUnit) throws InterruptedExc.
```

- Példa (JCIP 6.8)
- Az eddig megismert megvalósítások is ilyenek...
- `AbstractExecutorService` és `ThreadPoolExecutor`

Feladat kiosztása

java.util.concurrent.Executor

```
void execute(Runnable) throws RejectedExecutionExc.
```

java.util.concurrent.ExecutorService extends Executor

```
<T>        Future<T> submit    (Callable<T>)
           Future<?> submit    (Runnable)
<T>        Future<T> submit    (Runnable, T)
<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>>)
<T>               T invokeAny(Collection<? extends Callable<T>>)
```

Időzítés

- `Timer` és `TimerTask` – nem jól viseli az elszállt feladatokat
- `ScheduledThreadPoolExecutor`
  - Konstruktorral
  - `Executors.newScheduledThreadPool`lal
- `DelayQueue<E extends Delayed>`
  - `BlockingQueue`
  - Lejárt elemek vehetők ki

Kötegelt feladatok

- Sok feladat végrehajtása
- Eredmények beérkezési sorrendben

java.util.concurrent.CompletionService<V>

```
Future<V> submit(Callable<V>)
Future<V> take() throws InterruptedException
Future<V> poll()
...
```

- Megvalósítás: `ExecutorCompletionService<V>`
  - Konstruktorának átadandó egy `Executor`

Párhuzamos számítások: ForkJoinTask

- Tiszta számítás vagy izolált adat manipulálása
- Nagy tömegben használható
  - fine grained
  - 1000 számítási lépés
- `ForkJoinPool`ban futnak
- `fork`, `join`, `Future`, `invokeAll`
- Egy taszkból elforkolt taszkok ugyanabban a poolban
- Ne blokkolódjon (szinkronizáció, IO)
- Alosztályokból származtatunk
  (`RecursiveAction`, `RecursiveTask`, `CountedCompleter`)

ForkJoinPool

- egy `ExecutorService`
- jellemzően `ForkJoinTask`ok futtatására
- work stealing
- használható a `commonPool()`
- taszk indítása: `ForkJoinTask fjt`
  - egy másik `ForkJoinTask`ból: `fjt.fork()`
  - kívülről: `submit(fjt)`

Tipikus vizsgafeladat

- Valósítsd meg a mergesort algoritmust szekvenciális és párhuzamos változatban.
  - Elemezd ki a gyorsulást!
  - Hogyan szabályozod a párhuzamosság mértékét?
- Egy lehetséges szignatúra a szekvenciális változathoz:

```
int[] mergesort( int[] array )
```

Miért van rá szükség?

- Felhasználó visszavonja a kérését
- Időkorlátos tevékenység ideje lejár
- Egy másik komponens megoldást talált
- Hiba történt valamelyik komponensben
- Az alkalmazás/szolgáltatás leáll

- Példa (JCIP 7.1, 7.2, 7.3)

Folyamatok félbeszakítása

- Szál: `interrupt`
- Feladat: `cancel`
- Executor: `shutdown`

- Időkorlátos végrehajtás

Interrupt

java.lang.Thread

```
public void interrupt()
public boolean isInterrupted()
public static boolean interrupted()
```

- Példa (7.5)
- Feladat megszakítása őrizze meg a szálnak a megszakítást
- Feladat: félbeszakítható `factoring.Computation`
  - Főprogram időkorláttal
  - Főprogram leállítja felhasználói kérésre

Future.cancel

- Feladat félbeszakításához

java.util.concurrent.Future

```
boolean cancel(boolean mayInterruptIfRunning)
boolean isCancelled()
...
```

- `cancel(false)`: ne futtasd, ha még nem fut
- `cancel(true)`: ... és szakítsd félbe a szálat, ha fut

- Standard Executorokban való futtatásnál rendben van a `cancel(true)`
- Egyébként csak akkor, ha ismerjük a futtató szál reakcióját
- Időkorlátos futtatás (JCIP 7.10)

Szolgáltatás leállítása

- Életciklus metódusok
- Termelő-fogyasztó leállítása: a termelőket is és a fogyasztókat is
- Példa (JCIP 7.13, 7.14 és 7.15)

Haladási problémák

- Holtpont/deadlock
- Livelock
- Kiéheztetés
- Priority inversion
  - Priority inheritance
  - Priority ceiling

Hatékonysági kérdések

- Milyen gyorsan
- Milyen sokat
- Skálázódás

Általános javaslatok

- Kerüljük az idő előtti optimalizálást
- Mérjünk találgatás helyett
- Értsük meg az elméleti korlátokat (pl. Amdahl)
- Tesztelés, code review, statikus programanalízis

Amdahl törvénye

- Egy adott program gyorsíthatósága korlátozott
- A szekvenciális részének mérete korlátozza
- Legyen a szekvenciális végrehajtási idő $a + b$
  - ebből „a” a nem párhuzamosítható rész
  - jelölje α a nem párhuzamosítható rész arányát: $\alpha = \frac{a}{a+b} \in (0, 1]$
- Ekkor n processzoron a végrehajtási idő legalább $a + b/n$
- A gyorsulás

$$S(n) \leq \frac{a+b}{a+\frac{b}{n}} = \frac{a+b}{(a+b)\left(\alpha+\frac{1-\alpha}{n}\right)} = \frac{1}{\alpha+\frac{1-\alpha}{n}} \leq \frac{1}{\alpha} = \frac{a+b}{a}$$

Gustafson törvénye

- Több processzorral több/nagyobb feladatot meg tudunk oldani
- Ha elég sok feladatunk és elég processzorunk van, akármekkora gyorsulás elérhető
- Tegyük fel, hogy a megoldandó probléma mérete a processzorszám lineáris függvénye
- Legyen n processzoron a párhuzamos végrehajtási idő $a + b$
  - ebből „a” a szekvenciálisan futó rész
  - jelölje α a szekvenciálisan futó rész arányát: $\alpha = \frac{a}{a+b} \in (0, 1]$
- Feltételezzük, hogy α nem nő, ha a processzorszámot növeljük
- A gyorsulás

$$S(n) \geq \frac{a + n \cdot b}{a + b} = \frac{(a + b)\big(\alpha + n \cdot (1 - \alpha)\big)}{a + b} = \alpha + n \cdot (1 - \alpha)$$

Költségek és optimalizációk

- Kontextusváltás
  - CPU-idő (többezer órajel) és több cache miss
  - Minimális időszelet
  - Túl gyakori blokkolódásnál gyakran vész el az időszelet
- Szinkronizáció
  - Contended és uncontended
    - Az uncontendedre van optimalizálva
    - Blokkolás vagy spinning
  - Kioptimalizálható zárolás, ha a lock garantáltan uncontended
  - Escape analízis (JCIP 11.3), lock elision (IBM VM, HotSpot 7)
  - Lock coarsening (JCIP 11.3)
- Object pooling – rossz ötlet
  - `new Object()` kb. 10 gépi utasítás
  - Még egyszálú programban sem éri meg

Contended zárolást optimalizáljuk!

- A zárolás idejét csökkentsük (JCIP 11.4 és 11.5)
- A zárolások gyakoriságát csökkentsük (több lockra bontsuk)
  - Lock splitting (JCIP 11.6 és 11.7)
  - Lock striping (JCIP 11.8)
- A kölcsönös kizárást gyengébbel helyettesítsük
  - Konkurrens használatra tervezett adatszerkezetek
  - Módosíthatatlan objektumok
  - `Atomic` változók
  - `ReadWriteLock`

Explicit lockok

java.util.concurrent.locks.Lock

- Feltétel nélküli
- Nem blokkoló
- Időhöz kötött
- Megszakítható

... zárolások

Megvalósítások

- Zárolási szemantika
- Ütemezés
- Sorrendiség
- Hatékonyság?

ReentrantLock

- A beépített zároláshoz hasonló
- Több szituációban használható
  - Haladási problémák elkerülése: nem blokkoló és időhöz kötött (JCIP 13.3 és 13.4)
  - Félbeszakítható várakozás: megszakítható (JCIP 13.5)
  - Nem blokkszerkezetes kód (pl. hand-over-hand locking)
  - FIFO vs. tolakodó
- Veszélyesebb konstrukció (JCIP 13.2)
- Hatékonyság
  - HotSpot 5-ig sokkal jobb volt
  - HotSpot 6-tól kb. egyformák
  - A jövőben a beépített zárolás tovább javulhat
- Thread dump
  - HotSpot 5-ig csak beépített
  - HotSpot 6-tól explicit is, de nincs információ az aktivációs rekordról

Író-olvasó szinkronizáció

- `ReadWriteLock` interfész
  - `readLock()`
  - `writeLock()`
- Kérdések
  - Írás után ki jön? (Olvasók, író, FIFO)
  - Tolakodhatnak az olvasók?
  - Reentráns az olvasó vagy az író lock?
  - Downgrading: íróból lehet olvasó lock?
  - Upgrading: olvasóból lehet író lock?

Megvalósítás: ReentrantReadWriteLock

- Reentráns olvasó és író lock
- Lehet FIFO és tolakodó
- Downgrading OK, upgrading deadlockol
- Java 5-ben az olvasó lock nem ismerte a záróló szálat, Java 6-ban már igen

A wait-notify mechanizmus

- Minden objektumhoz wait-set
- Várakozás szignálig
- Zárolás szükséges
- `wait` előtt ellenőrizni a várakozási feltételt
- `wait`-et ciklusba szervezni
- `notify` és `notifyAll`
- Példa (JCIP 14.6)

Egyéb szinkronizációs eszközök

- `Condition`
- `AbstractQueuedSynchronizer`
- CAS
  - Nem blokkoló adatszerkezetek