10. Gyakorlat

legendi@inf.elte.hu

2010. április 20.

Threading

"Concurrency is hard and boring. Unfortunately, my favoured technique of ignoring it and hoping it will go away doesn't look like it's going to bear fruit."

Párhuzamosság: több részfeladat egyidejűleg történő végrehajtása.
Miért?

- A feladat logikai szerkezete
- A program több, fizikailag is független eszközön fut
- Hatékonyság
  (v.ö. Amdahl's law `http://en.wikipedia.org/wiki/Amdahl's_law`)

Elég régóta foglalkoztatja az embereket. *Látszat párhuzamosságról* is hallani még (op. rendszerek, multitasking: egyszerre egy folyamatot hajt végre, de adott időtartam alatt akár többet is), de a *valódi párhuzamosság* is már mindennapos (pl. többmagos, többprocesszoros gépekben).

Párhuzamosság szintjei

- Utasítások
- Taskok
- Folyamatok (processes)
- **Szálak (threads)**

Viselkedésük alapján lehetnek:

- Függetlenek
- Versengők
- Együttműködők

Alapproblémák

- Kommunikáció: kommunikációs közeg: socket, signal handler, fájl, osztott memória, etc.
- Szinkronizáció: folyamatok összehangolása, szinkron – aszinkron

Alapdefiníciók

Szinkronizáció olyan folyamat, amellyel meghatározható a folyamatokban szereplő utasítások relatív sorrendje

Kölcsönös kizárás osztott változók biztonságos használatához

Kritikus szakasz program azon része, ahol egy időben csak egyetlen folyamat tartózkodhat

Atomi művelet bármilyen közbeeső állapota nem látható a többi folyamat számára

Miért kell ez az egész? Pl. x++, 64 bites JVM , longon ábrázolva 2 regiszterben van tárolva → 2 olvasás + 2 írás

Szálak létrehozása

Két lehetőség:

- `Thread` osztályból származtatva: a run() metódust kell felüldefiniálni, majd a start() segítségével indítható az új szál. Megjegyzés: start() függvényt **nem bántod**, csak ha hívod a super.start() függvényt is! Példa:

```
package gyak10;

class TestThread extends Thread {
    @Override
    public void run() {
        System.out.println("TestThread");
    }
}

public class Create1 {
    public static void main(String[] args) {
        TestThread test = new TestThread();
        test.start();
    }
}
```

Névtelen osztállyal ugyanez:

```
new Thread() {
    @Override
    public void run() {
        System.out.println("TestThread");
    }
}.start();
```

- `Runnable` interfész implementálása: ha a származtatás nem lehetséges (pl. a fő osztály egy JFrame, Applet, etc.). Egyetlen függvénye van: run(), melyet meg kell valósítani. Indítani úgy lehet, ha egy Thread objektumnak megadod paraméterként, és arra meghívjuk a start() eljárást:

```
package gyak10;

class TestRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("TestRunnable");
    }
}

public class Create2 {
    public static void main(String[] args) {
        Thread thread = new Thread( new TestRunnable() );
        thread.start();
    }
}
```

Ugyanez névtelen osztállyal:

```
new Thread( new Runnable() {
    @Override
    public void run() {
        System.out.println("TestRunnable");
    }
}).start();
```

Szálak függvényei

- start(): indítás

- stop(): megállítás (deprecated). Megjegyzés: utána érdemes a Thread referenciát null értékre állítani.
- suspend(), resume(): felfüggesztés, majd újraindítás (deprecated)
- join(): másik szál befejezésének megvárása
- sleep(<ms>): adott időnyi várakozás
- yield(): well, ehh...
- getName(): konstruktorban beállítható név lekérdezése (később már nem változtatható)
- getThreadGroup(): konstruktorban beállítható csoport (később már nem változtatható). Egyszerre egyhez tartozhat, hierarchiába szervezhető (egy csoport más csoportokat is tartalmazhat).
- setDaemon(): daemon szál készítése (akkor terminál, ha minden más, nem daemon szál is már terminált).
- setPriority(<prior>): prior lehet 1-10, fontosságot jelöl. OS függő, hogy pontosan milyen hatása van. Időosztásos (*time slicing*) rendszereken nincs gond vele, egyébként egy "önző" szál teljesen befoglalhatja a CPU-t.

Részletesen

`http://java.sun.com/javase/6/docs/api/java/lang/Thread.html`

Megjegyzés Sok deprecated függvény, mert könnyen deadlockhoz vezethetnek (pl. erőforrás lefoglalásának megsznüntetése). Mindig van kerülőút, pl. szál leállítására:

```
private volatile isRunning = true;

public void stopRunning() {
    isRunning = false;
}

@Override
public void run() {
    while ( isRunning ) { ... }
}
```

Felmerülő problémák

Azon túl, hogy megbízhatóság ...

Holtpont kölcsönösen egymásra várakoznak a folyamatok, és egyik sem tud tovább haladni

Kiéheztetés több folyamat azonos erőforrást használ, és valamelyik ritkán fér csak hozzá

Versenyhelyzetek amikor egy számítás helyessége függ a végrehajtó folyamatok sorrendjétől (pl. *c*heck-then-act blokkok)

Nemdetirminisztikus végrehajtás kétszer ugyanazt a viselkedést produkálni lehetetlen, debuggolás esélytelen

A szinkronizációt ezen problémák elkerülésével kell megoldani.

Kölcsönös kizárás

Javaban ún. *szinkronizációs burok* van:

```
synchronized ( resource ) {
    ...
}
```

Ez garantálja, hogy az azonos lockhoz tartozó blokkokban egyszerre egy szál lehet csak (gond - kódblokkot védünk, nem erőforrást). A `synchronized` használható példány-, és osztályfüggvény módosítószavaként, ekkor a jelentése:

```
public synchronized void f() {
    ...
}

// Ekvivalens:
public void f() {
    synchronized ( this )  {
        ...
    }
}
```

Illetve osztályfüggvények esetén:

```
class MyClass {
    public static synchronized void s() {
        ....
```

```
    }

    // Ekvivalens:
    public static void s() {
        synchronized ( MyClass.class ) {
            ...
        }
    }
}
```

Megjegyzés ha csak egy szál változtathat egy változót, a többi csak olvassa, akkor jöhet jól a `volatile` kulcsszó, amely garantálja, hogy a szálak nem cache-elik az adott változó értékét, mindig a frissítik (ld. a stop() kiváltására írt példát feljebb!).

Megjegyzés Immutable osztályokhoz nem kell szinkronizálni!

Szinkronizáció üzenetekkel

Feltételes beváráshoz: *Object* osztályban definiált *wait()*, *notify()* és *notifyAll()* függvények. A wait hívásának hatására a szál elengedi a lockot és blokkolódik, amíg egy másik szál nem jelzi számára, hogy az adott feltétel teljesül (notify()).

Használatához *mindig* egy monitor szükséges, különben futásidejű hibát kapunk!

```
synchronized (monitor) {
    monitor.wait();
}

synchronized (monitor) {
    monitor.notify();
}
```

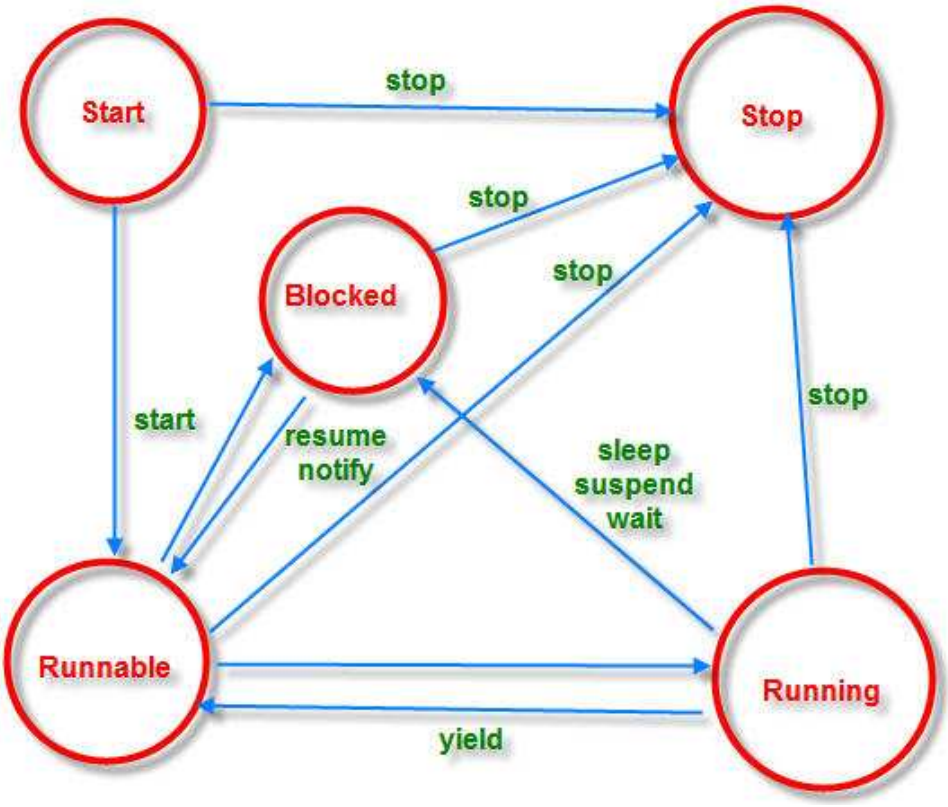
Deadlockra példa

```
package gyak10;

public class Deadlock {
    public static void main(String[] args) {
        final Object res1 = new Object();
        final Object res2 = new Object();

        new Thread() {
```

```
            @Override
            public void run() {
                synchronized (res1) {
                    System.out.println("1 - Got res1");
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    synchronized (res2) {
                        System.out.println("1 - Got res2");
                    }

                }
            }
        }.start();

        new Thread() {
            @Override
            public void run() {
                synchronized (res2) {
                    System.out.println("2 - Got res2");
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    synchronized (res1) {
                        System.out.println("2 - Got res1");
                    }

                }
            }
        }.start();
    }
}
```

Szállak állapotai

Ld. 1 ábra.



1. ábra. Szállak állapotai

Kollekciók

Szinkronizált vs. nem szinkronizált adatszerkezetek (pl. Vector vs. ArrayList). Az iterátorok *fail-fast* iterátorok: ha bejárás közben módosítják az adatszerkezetet, reccsen egy `java.util.ConcurrentModificationException` kivétellel:

```
package gyak10;

import java.util.ArrayList;

public class FailFast {
    public static void main(final String[] args)
            throws InterruptedException {
        final ArrayList<String> list = new ArrayList<String>();
        for (int i=0; i<100; ++i) list.add("" + i);

        final Thread reader = new Thread() {
            @Override
```

```
            public void run() {
                try {
                    for (final String act : list) {
                        System.out.println(act);
                            Thread.sleep(100);
                    }
                } catch (final InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };

        reader.start();
        Thread.sleep(500);

        list.remove(50);
    }
}
```

Szinkronizált adatszerkezetek készítése wrapperekkel, példa listára, másra hasonlóan:

```
final List<T> list = Collections.synchronizedList(new ArrayList<T>(...));
```

Feladatok

1. Készíts egy 2 szállal működő programot, amelyek neve térjen el! A szálak tízszer egymás után írják ki a képernyőre a nevüket, majd várjanak egy keveset (0-5 másodpercet, véletlenszerűen).

2. Készíts 5 szálat, amelyek a következő prioritás-szintekkel futnak: 3, 4, 5, 6, 7 (ez szerepeljen a szálak nevében is!). A szálak egy végtelen ciklusban írják ki a nevüket. Elemezd az eredményt!

3. Készíts egy 5 szállal dolgozó programot, amelyek ugyanazt a közös változót kiírják, majd csökkentik (100-ról 0-ra). Figyelj a szinkronizációra, és a végén ellenőrizd le, hogy valóban helyes outputot kaptál-e!

4. Készíts egy 3 szálú alkalmazást! Legyen egy termelő, és két fogyasztó szálunk. Az termelő szál induljon el, és töltsön fel egy kollekciót 10 db véletlen számmal (a másik két szál indulás után várjon)! Ezután jelezzen a másik két szálnak (`wait()`, `notify()`), hogy elkezdhetik a számok feldolgozását: adják össze őket. Az eredeti szál várja be a feldolgozást, majd írja ki a részösszegek összegét!

5. Kérdezd le egy új szálban az összes futó szálat, és írd ki azok neveit! Értékeld a látottakat egy grafikus alkalmazás indítása eseten (ehhez rekurzívan be kell járni a `getParent()`-eket)!

6. Készíts 2 szálat! Az első állítsa elő az első tíz hatványát a kettes számnak (majd várjon egy másodpercet), a másik legyen egy daemon szál, amely a nevét írja ki, majd vár egy másodpercet egy végtelen ciklusban.

7. Készíts 5 szálat, amelyek egy saját csoportban vannak! A szálak egy véletlen számot választanak az 1-100 intervallumból, és fél másodpercenként növelnek egy saját számlálót ezzel az értékkel, amíg az meg nem haladja az 1000-et.

   A szálak elindítása után a fő szálban várjunk 10 másodpercet, majd listázzuk ki az aktív szálakat, a maximális prioritást a szálak között, és írjunk ki egy listát a szálakról és azok tulajdonságairól! Ezután függesszük fel az összes szálat, írjuk ki a szülő `ThreadGroup` nevét, majd újra indítsuk el az összes szálat a saját csoportunkban. A végén pedig várjuk be az összes szálat!

8. Készíts egy egyszerű grafikus alkalmazást, amely egyetlen panelt tartalmaz, az aktuális idővel. A panelen található információt másodpercenként frissítsd! Az osztály definíciója nézzen ki a következőképpen:

   ```
   public class ... extends JFrame implements Runnable {
       ...
   }
   ```

9. Egészítsd ki az előző feladatot úgy, hogy ha ráklikkel a felhasználó a `Label`-re, akkor szüneteltesse a frissítést a program. Ha újra ráklikkel, folytassa a számlálást!

Java Concurrency - 1.6

A `java.util.concurrent.*`, `java.util.concurrent.atomic.*`, `java.util.concurrent.lock.*` csomagok változatos, hatékony eszközöket nyújtanak:

- Barrier, Semaphor, FutureTask, ...
- Adatszerkezetek: ConcurrentHashMap, BlockingQueue, ...
- Lockok, pl. ReentrantLock, ...
- Atomi változók: AtomicLong, AtomicReference, ...

+/- Feladat

A feladatokat a `legendi@inf.elte.hu` címre küldjétek, **következő szombat éjfélig**! A subject a következőképp nézzen ki:

```
csop<csoportszám>_gyak<gyakorlat száma>_<EHA-kód>_<feladatok száma>
```

Például:

```
csop1_gyak10_LERIAAT_1
csop1_gyak10_LERIAAT_12
```

Mellékelni csak a Java forrásfájlokat mellékeljétek (semmiképp ne teljes Eclipse/NetBeans projecteket), esetleg rarolva, zipelve, és egy levelet küldjetek (több levél esetén az utolsó mellékleteit értékelem)! További fontos kritériumok:

- A konvenciókra figyeljetek plz!
- Ékezetes karaktereket **ne** használjatok! Főleg azonosítók esetében ne! A Java ugyan ezt megengedi, ugyanakkor a különböző környezetekbe való konvertáláskor ($latin2 \leftrightarrow UTF-8 \leftrightarrow Cp1250$) összetörnek a karakterek! Az ilyen forrásokat fordítani, következésképp értékelni sem tudom.

1. Feladat

Egészítsd ki a 8. gyakorlaton készített port scanner alkalmazást, hogy 2. parancssori argumentumként meg lehessen mondani hány szállal hajtsa végre párhuzamosan a vizsgálatot!

2. Feladat

Készítsünk egy saját WWW keresőmotort, amely egy cím → URL kereső adatbázist képesz készíteni! A program 5 szál használatával keressen az interneten, minden szám max. 50 weboldalt járjon végig, és írják ki egy közös fájlba soronként a weboldal címét (a title meta-tag értékét), valamint az éppen vizsgált URL-t. Parancssori argumentumként kapjon egy URL címet, amelyet végigolvasva további URL címeket keressenek a szálak.