Budapesti Műszaki és Gazdaságtudományi Egyetem

Java technológia

Szálak

Sipos Róbert

siposr@hit.bme.hu

2014. 02. 27.

Java és operációs rendszer szálak I.

- A Java lehetővé teszi programok **több szálon** (thread) történő végrehajtását.
- A Java szálakat a Java VM folyamata által **az operációs rendszer szintjén létrehozott gyermek folyamatok, vagy (jellemzőbben) szálak** testesítik meg.
- Az 1.3-as verzió előtt Linux rendszerekben a szálakat tisztán Javaban valósították meg (a Unix/Linux eltérések miatt, a portolást megkönnyítendő), ezt a megoldást **green threads**nek hívták. A továbbiakban a green threads specifikumaival nem foglalkozunk.

Java és operációs rendszer szálak II.

- A Java szálakhoz **prioritás**t rendelhetünk, amelyet a Java továbbít az operációs rendszer ütemezőjének, ennek hatékonysága változó.
- Mivel a Java szálak operációs rendszer szálakként vannak megvalósítva, az operációs rendszer és a hardware lehetőségeitől függően **konkurrensen**, vagy **párhuzamosan** kerülnek végrehajtásra.
- SMP (Symmetric Multi-Processing) gépen, megfelelő operációs rendszer támogatás mellett a Java szálak külön processzoron is futhatnak (az operációs rendszer ütemezőjétől függően).

Szálak a Java VM-ben

- A Java VM-ben kétféle szál létezhet, **démon** (daemon) és **nem-démon** (non-daemon) szál.
- Kezdetben egy nem-démon szál jön létre, amely elkezdi végrehajtani a megadott osztály main metódusát.
- A Java VM akkor áll le, amikor az utolsó nem-démon szál futása is befejeződött.
- A démon szálak arra valók, hogy a háttérben szolgáltatásokat illetve egyéb tevékenységeket valósíthassunk meg anélkül, hogy ez a tevékenység a Java VM leállását megakadályozná.

Szálak létrehozása I.

- A szálakat a **Thread osztály** és leszármazottainak példányai testesítik meg.
- Egy Thread példány elindításakor a **run** metódusát hajtja végre egy önálló szálon. A Thread run metódusa nem csinál semmit.
- Értelmes tevékenységet végző szálat két módon hozhatunk létre:
  - a Thread osztályból leszármaztatunk egy új osztályt, melyben a run metódust felüldefiniáljuk,
  - egy, a **Runnable interfész**t megvalósító osztály egy példányát adjuk át a Thread osztály konstruktorának.
- A Runnable interfész egyetlen run metódust tartalmaz, használatára általában akkor van szükség, ha a run metódust tartalmazó osztály nem származhat a Thread osztályból.

Szálak létrehozása II.

```
public class MyThread extends Thread {
   public void run () {
      System.out.println ("Ezt a szál írja ki.");
   }
   public static void main (String[] args) {
      new MyThread ().start ();
   }
}
```

- vagy

```
public class MyThread implements Runnable {
   public void run () {
      System.out.println ("Ezt a szál írja ki.");
   }
   public static void main (String[] args) {
      new Thread (new MyThread ()).start ();
   }
}
```

Példa

```
public class MyThread extends Thread {









   public void run () {
      System.out.println ("???. szál.");
   }


   public static void main (String[] args) {
      for (int i = 1; i <= 4; i++) {

         new MyThread ().start ();

      }
   }
}
```

Példa

```
public class MyThread extends Thread {


   private int n;

   public MyThread(int n) { this.n = n; }


   public void run () {
      System.out.println (n + ". szál.");
   }


   public static void main (String[] args) {
      for (int i = 1; i <= 4; i++) {

         new MyThread (i).start ();

      }
   }
}
```

Szinkronizáció I.

- Többszálú programozás esetén szükség van a szálak **szinkronizáció**jára amennyiben azok közös adatokon dolgoznak.
- A szinkronizációnak több célja is lehet:
  - **kölcsönös kizárás** (mutual exclusion), szemafor (semaphore), kritikus szakasz (critical section),
  - **randevú** (rendezvous), az egyik szál felfüggeszti futását amíg a másik be nem fejeződik,
  - **várakozás** egy másik szálon történő valamely esemény bekövetkeztére (wait),
  - stb.

Szinkronizáció II.

- A Java szálmodelljében a szinkronizáció az ún. **monitor**ok segítségével valósul meg.
- Minden objektumhoz (példányhoz) tartozik egy monitor.
- A monitort **egyszerre egy szál birtokolhatja**.
- Ha egyszerre több szál is igényt tart a monitorra, a többi szál várakozni kényszerül, futása felfüggesztődik.
- Ha a monitort birtokló szál elengedi a monitort, a monitort igénylő többi szál verseng a monitorért. A szálak közül az ütemező választja ki azt, amelyik a monitort megkapja.

Szinkronizáció III.

- A szinkronizáció a `synchronized` kulcsszó használatával valósítható meg.

```
public class Test {
   Object o = new Object ();

   public static void main (String[] args) {
      synchronized (o) {       // egyszerre csak egy szál tartózkodhat
         ...                                    // ebben a blokkban
      }
   }
}
```

- A fenti konstrukció neve **synchronized blokk**.
- A blokkba való belépéskor a blokkot végrehajtó szál megszerzi a megadott objektum monitorát, a blokkból való kilépéskor elengedi azt.

Szinkronizáció IV.

- Sokszor arra van szükség, hogy egész metódusokat szinkronizáljunk:

```
public class Test {

   public synchronized void method () {     // egyszerre csak egy szál
      ...                                   // tartózkodhat ebben a metódusban
   }
}
```

- A fenti esetben a metódust hívó szál a metódust tartalmazó objektum monitorát birtokolja a metódus végrehajtásának idejére.
- Ha osztálymetódust szinkronizálunk, a hívó szálnak egy Classból származó osztálynak, a metódust tartalmazó osztályt leíró példányának monitorát kell megszereznie.

Szinkronizáció V.

- Példa szinkronizált osztálymetódusra:

```
public class Test {

   public static synchronized void method () {  // egyszerre csak egy
      ...                                       // szál tartózkodhat ebben a metódusban
   }
}
```

- A példában használt monitor az alábbi objektumhoz tartozik:

```
Test.class                                                    // .class operátor
```

- ami megegyezik azzal, hogy:

```
new Test ().getClass ()                  // az Object osztály getClass metódusa
```

Szinkronizáció VI.

- Nem feltétlenül szükséges a szinkronizáció akkor, ha **atomi műveletek**et végzünk.
- Minden primitív típusú mező írása illetve olvasása atomi művelet, kivéve a long és a double típusokat.
- Az atomicitás azt garantálja, hogy ha két szál hajtja végre a műveletet konkurrensen, szinkronizáció nélkül, a többi szál nem láthat inkonzisztens állapotot a mezőben.

```
public class Test {
   public void setA (int a) { this.a = a; }                    // atomi
   public synchronized void setL (long l) { this.l = l; } // nem atomi

   int  a;
   long l;
}
```

Szinkronizáció VII.

- Az atomicitás csak az inkonzisztencia ellen véd, de azt nem garantálja, hogy a többi szál mikor látja a módosított értéket (cache).

```
public void Test extends Thread {

   public void run () {
      while (!stopNow) { ... }
   }

   public void stopThread () {
      stopNow = true;
   }

   private boolean stopNow = false;
}
```

- **Vigyázat! A fenti példában a szál lehet, hogy soha nem áll le!**

Szinkronizáció VIII.

- Kézenfekvő (és helyes) megoldás a mezőhöz történő hozzáférések szinkronizációja:

```
public void Test extends Thread {

   public void run () {
      while (!isStopNow ()) { ... }
   }

   public synchronized void stopThread () { stopNow = true; }

   private synchronized boolean isStopNow () { return stopNow; }

   private boolean stopNow = false;
}
```

Szinkronizáció IX.

- Alternatív (és szintén helyes) megoldás a **volatile** módosító használata:

```
public void Test extends Thread {

   public void run () {
      while (!stopNow) { ... }
   }

   public void stopThread () {
      stopNow = true;
   }

   private volatile boolean stopNow = false;
}
```

- A volatile módosító garantálja, hogy a mező írása után az új érték azonnal láthatóvá válik a többi szál számára.

Az Object osztály szolgáltatásai I.

- Az Object osztály néhány metódusa az adott objektum monitorával kapcsolatos néhány funkciót biztosít:
  - **wait ()**: Felfüggeszti a hívó szál futását.
  - **wait (long timeout)**: Legfeljebb timeout ms-re felfüggeszti a hívó szál futását.
  - **wait (long timeout, int nanos)**: Mint az előző, de ns-t is megadhatunk.
  - **notify ()**: Az objektum monitorán waittel várakozó szálak közül egyet felébreszt (véletlenszerűen választ).
  - **notifyAll ()**: Mint a notify, de minden várakozó szálat felébreszt.
- Ezek a metódusok csak **az objektum monitorának birtokában** hívhatók.

Az Object osztály szolgáltatásai II.

- A wait metódus hatására a hívó szál lemond a monitorról, és várakozik, amíg egy másik szál meg nem hívja az objektum notify, vagy notifyAll metódusát.
- A notify, illetve notifyAll hatására felébredő szálak ugyanúgy versenyeznek a monitor birtoklásáért, mint a többi, a monitort szinkronizáció céljából birtokolni kívánó szál.
- Ha a szál visszaszerezte a monitort, folytatja a futását.
- Ha a szál a wait metódus időkorlátos változataival várakozik, akkor legkésőbb a megadott idő elteltével automatikusan felébred, és megkísérli visszaszerezni a monitort.

Az Object osztály szolgáltatásai III.

- Példa a wait és a notify használatára:

```
public void WaitNotifyTest {

   public synchronized varok () {
      try {
         wait ();                              // felfüggesztem a futásomat
      } catch (InterruptedException e) {}
   }

   public synchronized mehetsz () {
      notifyAll ();                          // minden várakozót felébresztek
   }
}
```

- A varok metodust hívó szál addig várakozik, amíg valaki meg nem hívja a mehetsz metódust.
- A szinkronizáció a WaitNotifyTest példány monitorára történik.

A Thread osztály szolgáltatásai I.

- A Thread osztályban az alábbi metódusok szolgálják a szálak vezérlését:
  - **start ()**: A szál végrehajtása elkezdődik.
  - **stop ()**: A szál végrehajtása megszakad. Ez a metódus **elavult**, mert használata veszélyes. (A szál által birtokolt monitorok felszabadulnak, a többi szál inkonzisztens állapotot láthat.)
  - **suspend ()**, **resume ()**: A szál végrehajtásának ideiglenes megszakítása és folytatása. Ez a két metódus **elavult**, mert használatuk veszélyes. (Mivel a suspend nem szabadítja fel a szál által birtokolt monitorokat, deadlock következik be, ha például a resume-ot kiadó szál az első szál által birtokolt monitorra vár.)

A Thread osztály szolgáltatásai II.

- A Thread osztály további metódusai:
  - **yield ()**: A szál egy rövid időre felfüggeszti futását, hogy a többi szál is futhasson. (A jelenlegi, preemptív ütemezők esetén nincs jelentősége.)
  - **join ()**: A hívó szál várakozik, amíg ez a szál be nem fejezi futását. Ezt a metódust randevú megvalósítására használhatjuk.
  - **join (long millis)**, **join (long millis, int nanos)**: Mint a wait időkorlátos változatai.
  - **sleep (long millis)**, **sleep (long millis, int nanos)**: A szál felfüggeszti futását a megadott időtartamra. A szál által birtokolt monitorokat nem engedi el.

A Thread osztály szolgáltatásai III.

- A Thread osztály további metódusai:
  - **interrupt ()**: Ha a szál wait, join, vagy sleep metódusban várakozik, megszakítja a várakozást, és a szál futása folytatódik. A wait, join, illetve sleep metódus ilyenkor **InterruptedException**t dob.
  - **setDaemon (boolean daemon)**, **isDaemon ()**: A szál démon tulajdonságának beállítására és lekérdezésére szolgál.
  - **setPriority (int priority)**, **getPriority ()**: A szál prioritásának beállítására és lekérdezésére szolgál.
  - static Thread **currentThread ()**: A futó szál lekérdezésére szolgál.
  - static boolean **holdsLock (Object obj)**: Akkor add vissza true-t, ha a futó szál birtokolja a megadott objektum monitorát.

A Thread osztály szolgáltatásai IV.

- Példa a join használatára:

```
public class MyThread extends Thread {
    ...
}

public class JoinTest {

   public void metodus () {
      MyThread mt = new MyThread ();
      mt.start ();                              // elindítom a másik szálat
      ...                                         // addig csinálok valamit
      try {
         mt.join ();                  // megvárom, amíg a másik befejeződik
      } catch (InterruptedException e) {}
   }
}
```

Szálak és kivételek

- Ha egy **szálban kivétel keletkezik** és ezt a szálban **nem kezeljük le**, a szál leáll, és a kivétel egy hibaüzenet formájában a standard errorra íródik.
- Ha egy szálat a stop metódussal leállítunk, egy **ThreadDeath** kivétel keletkezik, amelyet lekezelve elvégezhetjük a szál utáni takarítást.
- Mivel a ThreadDeath gyakorlatilag bárhol keletkezhet a szálban, használata rendkívül kényelmetlen (sőt, szinte lehetetlen tökéletesen megcsinálni).

Szálak csoportosítása

- A **ThreadGroup** osztály segítségével a szálakat csoportokba rendezhetjük.
- A ThreadGroupok **fát alkothatnak**.
- A csoportban lévő szálak bizonyos tulajdonságait egyszerre állíthatjuk (pl. prioritás, démon, stb.).
- A régi biztonsági modellhez készült, ma már nem nagyon van rá szükség.

Lokális változók szálakban

- A **ThreadLocal** osztály segítségével olyan változókat hozhatunk létre, amelyeknek minden szálhoz tartozik egy-egy külön példánya.
- A get és set metódusokkal az érték lekérdezhető és beállítható (Java 5-től kezdve generikus típus: ThreadLocal<T>).

```
public class ThreadLocalTest {

   public ThreadLocalTest () {
      tl = new ThreadLocal ();
      tl.set (new Integer (0));
   }

   public void metodus () {                    // megnöveli a tárolt értéket
      int i = ((Integer) tl.get ()).intValue ();
      tl.set (new Integer (i+1));
   }

   private ThreadLocal tl;
}
```