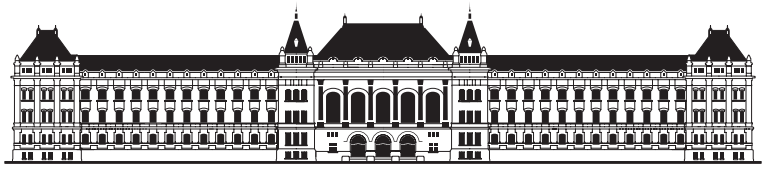

Budapesti Műszaki és Gazdaságtudományi Egyetem
Méréstechnika és Információs Rendszerek Tanszék
Hibatűrő Rendszerek Kutatócsoport

Rendszerintegráció és -felügyelet laboratórium
2013/2014 tavaszi félév

Munkafolyamatok megvalósítása Java nyelven

Mérési segédlet
v1.2

Izsó Benedek
Ujhelyi Zoltán
Szárnyas Gábor
(szarnyas@mit.bme.hu)
2014. március 30.

1. Bevezető

A labor során a méréseket végző hallgató a gyakorlatban is megismerkedik a rendszerintegráció és rendszerfelügyelet során használatos módszerekkel és eszközökkel. Végigköveti egy elosztott alkalmazás megvalósításának és felügyeletének legfontosabb lépéseit, ipari környezetben használt integrációs köztesréteg (middleware) technológiák és felügyeleti eszközök használatával. A mérések a következő témakörökhöz kapcsolódnak:

1. Munkafolyamatok megvalósítása Java nyelven
2. Rendszerfelügyelet komplexesemény-feldolgozással
3. Megbízható üzenetküldés IBM WebSphere MQ alapon
4. Kommunikáció JMS és JMX technológia segítségével
5. OSGi szolgáltatások fejlesztése
6. Aktor modell konkurens alkalmazások készítésére: Akka
7. Felügyeleti adatok vizuális elemzése

A jelen mérés során a hallgató megvalósítja az előre megrajzolt munkafolyamatot a Java nyelv beépített eszközeivel: szálak segítségével lehetséges egy elosztott folyamat végrehajtása, míg a végrehajtás állapota egy Swing alapú grafikus felületen tekinthető meg.

2. Többszálú alkalmazások fejlesztése Java nyelven

Napjainkban az alkalmazások jelentős része több feladat párhuzamos végrehajtására képes. Ennek oka részben az így elérhető jobb teljesítmény, másrészt így lehet biztosítani, hogy az alkalmazások grafikus felhasználói felülete akkor is reagáljon a felhasználónak, amikor egy hosszabb feladatot hajt végre.

A párhuzamos végrehajtás támogatására a különböző operációs rendszerek, illetve programozási környezetek többféle lehetőséget biztosítanak. Lehetséges ugyanazon alkalmazást több példányban, más néven több *processzben* elindítani, vagy egy alkalmazást több *szál* segítségével futtatni.

Az egyes processzek alapvetően önállóak, memóriaterületük és állapotuk független a többitől, ezért a processzek közötti váltás viszonylag drága művelet. Processzek közötti információmegosztásra csak dedikált interprocessz kommunikációs mechanizmusok használhatóak. Alkalmazáson belüli párhuzamosításra ezért gyakran szálakat használunk, amelyek megosztják egymás között a memória címtartományukat, ezáltal lehetővé téve a gyorsabb kommunikációt.

2.1. Szálak Java környezetben

Java nyelven a szálak készítésére egy egyszerű API használható: a külön szálon futtatandó kódot egy Runnable interfészen belül kell megírni, ennek egy példányát kell átadni az új `Thread` példánynak.[1]

[1]Szükség esetén lehetőség van a `Thread` osztályból leszármaztatva közvetlenül egy saját szál implementációt létrehozni. Ugyanakkor logikailag tisztább a `Runnable` használata (nem módosítani akarjuk a `Thread` osztály viselkedését), valamint könnyebb újrahasznosítást tesz lehetővé (pl. használható az `Executor` keretrendszerben is).

A következő kódrészlet egy egyszerű szál futtatását mutatja be. A `PrimeRun` osztály prímszámok számítására használható, míg a `PrimeTest` osztály futtatja a szálat a megadott 143-as paraméterrel.

```
public class PrimeRun implements Runnable {
	long minPrime;
	long[] foundPrimes;

	PrimeRun(long minPrime) {
		this.minPrime = minPrime;
	}

	@Override
	public void run() {
		// compute primes larger than minPrime
		// TODO insert code here foundPrimes=...
	}
}

public class PrimeTest {

	public static void main() {
		Thread thread = new Thread(new PrimeRun(143));
		thread.start();
	}
}
```

2.1.1. Adatcsere szálak között

A szálak közötti adatcsere hasonló módon történik, mint szálakon belül: a `Runnable` objektum attribútumai elérhetőek más szálakból is. Amire figyelni kell, hogy a nyelv teljesítményokokból nem biztosít kizárólagos hozzáférést az attribútumokhoz, kivéve, ha a fejlesztő ezt külön megadja. Ezen megkötés nélkül a különböző szálakon futó utasítássorok átlapolódása miatt *inkonzisztens memóriakép* (memory consistency error) alakulhat ki, vagy *zavarás* (thread interference) következhet be.

Zavaró szálak esetén nem konzisztens módon lehetséges, hogy bizonyos utasítások hatása teljesen figyelmen kívül marad (vagy esetleg tökéletesen működik). Például ha ugyanarra a változóra két szál meghívja a `++*,␣ill.␣\verb*–*` utasításokat, az olvasások és írások átlapolódása miatt lehetséges, hogy a `d` változó értéke változatlan marad (helyes viselkedés), eggyel csökken (növelés „nem hajtódik végre” [2]) vagy eggyel nő (csökkentés „nem hajtódik végre” [2]).

A metódusok szinkronizációja lehetővé tesz egy egyszerű szinkronizációs stratégiát: ha minden írási és olvasási művelet szinkronizált metóduson (`synchronized` kulcsszó) keresztül történik, akkor sem inkonzisztens memóriakép, sem zavarás nem alakulhat ki. Ugyanakkor gondokat okozhat az élőség kapcsán, másrészt teljesítményszempontból sem optimális, sokszor felesleges várakozásokat iktat a rendszerbe.

A teljesítményen sokat javít a *szinkronizált utasítások* (synchronized statements) használata, ami sokkal kisebb léptékű, így sokkal kisebb mértékű blokkolás érhető el. Szinkronizált

[2]Ténylegesen végrehajtódik, csak az eredményt a párhuzamos végrehajtás miatt elveszíti a rendszer.

utasítások használata esetén meg kell adni egy zár objektumot is – erre szinkronizált metódusok esetén nem volt szükség, akkor maga az objektum alkotta a zárat. Fontos, hogy ilyen esetekben a rendszer nem garantálja, hogy nem történik zavaró utasítás-átlapolódás.

A következő kódrészlet egy egyszerű példát mutat arra, hogy mikor lehet hasznos több kisebb zárat használni: mivel az egyes változókhoz tartozó inkrementáló utasításnak semmi köze a másik változóhoz, ezért értelmes függetlenül zárolni őket.

```
public class MsLunch {
      private long c1 = 0;
      private long c2 = 0;
      private Object lock1 = new Object();
      private Object lock2 = new Object();

      public void inc1() {
            synchronized (lock1) {
                  c1++;
            }
      }

      public void inc2() {
            synchronized (lock2) {
                  c2++;
            }
      }
}
```

További egyszerű szinkronizációs lehetőséget biztosít az attribútumok megjelölése `volatile` kulcsszóval. Az így megjelölt attribútumok elemi írási és olvasási műveletei garantáltan atomiak (nem szakíthatóak meg), ezen felül a műveletek között előbb-történt reláció áll fenn (szálak közt is). Más szóval, az így megjelölt változók változásai azonnal látszódnak a többi szál számára is. A módszer nem garantálja az átlapolódás teljes hiányát, ugyanakkor lényegesen egyszerűsíti a konzisztens memóriakép előállítását. Többek között olyan változókat lehet érdemes megjelölni a kulcsszóval, amelyek szálak közti jelzésre szolgálnak.

2.1.2. Párhuzamos kollekciók

A Java Collections API legnagyobb része egyszálú működésre lett kitalálva. Ha több szál között kell megosztani egy kollekciót, akkor vagy gondoskodni kell a megfelelő szinkronizációról, vagy olyan osztályokat kell választani, amit eleve párhuzamos elérésre terveztek. Ilyen kollekciók a `BlockingQueue` vagy a `ConcurrentMap` interfészek megvalósításai.

A `BlockingQueue` objektumok olyan sort definiálnak, amely teli sorhoz adásnál, illetve üres sorból olvasásnál blokkolják az őt meghívó szálat, esetleg *időtúllépés* (timeout) jön létre. Így az osztály jól felhasználható szálak közötti kommunikációra: a termelő ír a sorba, amely szükség esetén blokkolja az olvasást, míg a fogyasztó addig blokkolódik, amíg a termelő valamit nem tesz be a sorba. Az interfész implementációi támogatják a több termelő-több fogyasztó felállást is.

A `ConcurrentMap` objektumok a gyakran használt `Map` objektumok szálbiztos megfelelőik: mivel az írási és olvasási műveletek között garantálják az előbb-történt reláció teljesülését.

2.1.3. Egyéb feladatok szálak vezérlése kapcsán

Abban az esetben, ha egy szál nem-blokkoló hívással ellenőrzi, hogy valami esemény bekövetkezett-e (pl. egy másik szál beállított-e egy változót), akkor ez a szál képes lehet a processzort lényegében teljes időben lefoglalni. Hogy ezt elkerüljük, az ilyen szálak futás közben szüneteltetik magukat egy időre. Ehhez a `Thread` osztály statikus `sleep()` metódusát kell használni, amelynek paraméterül azt az időt kell megadni, amíg a szál várakozzon. Több feladatnál akár néhány másodperces várakozás sem okoz lényeges problémát, ugyanakkor jelentősen csökkenti a gép terhelését.

Egy szál akkor fejezi be a futást, ha a hozzá tartozó `Runnable` osztály befejezte a `run()` metódusának végrehajtását. Ha a szál ciklikusan fut, szükséges lehet kívülről leállítani. Ennek javasolt módja a szál megszakítása.

Egy másik szálat úgy lehet megszakítani, hogy a megfelelő `Thread` objektum `interrupt()` metódusát meghívjuk (ez nem statikus). A szálnak erre figyelnie kell: bizonyos metódusok, mint pl. a `Thread.sleep()` megszakítás esetén `InterruptedException` kivételt dob. Amennyiben nem hív meg a szál ilyen metódust, akkor a `Thread.interrupted()` statikus metódus segítségével lehet megvizsgálni, hogy érkezett-e megszakítás. Noha tetszőleges kódot lehet írni ezen megszakítások kezelésére, tipikusan a szál leállítása az elvárt viselkedés.

Amennyiben szükséges, a `Thread` objektum `join()` metódusát meghívva várakozni lehet addig, amíg a megfelelő szál futása be nem fejeződik. Ez szinkronizálásban segít; kilépéskor pedig az összes szálra meghívva a metódust meggyőződhetünk róla, hogy minden szál befejeződött (és felszabadította a lefoglalt erőforrásait).

2.2. Grafikus felhasználói felületek fejlesztése Java nyelven

Az egyik leggyakrabban használt Java grafikus toolkit a beépített Swing keretrendszer. A rendszer rugalmas megoldást jelent a különböző platformok grafikus felületének megalkotásához.

A rendszer a grafikus felületet egymásba ágyazott komponensek segítségével építi fel, a hierarchia tetején egy ablakot reprezentáló elemmel (`JFrame`, `JDialog` vagy `JApplet`). A komponensek elrendezéséhez layout algoritmusok használhatóak, amelyek a rendelkezésre álló hely függvényében méretezik a beágyazott komponenseket.

A Swing keretrendszer használatát mennyiségi okokból nem részletezzük, szükség esetén érdemes a 4. szakaszban hivatkozott Swing tutorialt alaposabban átnézni.

2.3. Többszálú grafikus alkalmazások

Annak ellenére, hogy a Java nyelv támogatja a többszálú alkalmazások fejlesztését, a Swing keretrendszer fő kódja alapvetően egyszálú: miután egy Swing komponens létrejött, minden kód, ami közvetlenül a komponens állapotát írja vagy olvassa, egy dedikált eseménykezelő szálon fut (event-dispatching thread)[3]. Ennek az az oka, hogy a grafikus felület eseményvezérelt viselkedése nehezen egyeztethető össze a többszálú alkalmazások fejlesztésénél szükséges szinkronizációval.

Ugyanakkor gyakori feladat, hogy az alkalmazásban háttérszálakat kell használni, ugyanis ellenkező esetben a hosszabb feladatok elvégzése az alkalmazás felületének blokkolásával járna. A Java 6-os változatában ennek megkönnyítésére bevezették a `SwingWorker`

[3]Van néhány kivétel a szabály alól, ezek a következő leírást tartalmazzák a Javadoc fejkommentben: *"This method is thread safe, although most Swing methods are not."*

osztályt, amely egy osztályban gyűjti össze a feladatot ténylegesen elvégző kódot, valamint az eredményt (és állapotot) a felületen megjelenítő kódot.

A `SwingWorker` működése kapcsán három szál jut szóhoz:

1. Az *aktuális szál*, amelyik ténylegesen meghívja a `SwingWorker`t implementáló osztályunkat. A feladat háttérben indítása az `execute()` függvénnyel lehetséges, ami gyakorlatilag az általunk felüldefiniált `doInBackground()` metódus háttérben történő futtatását oldja meg. A háttérművelet eredménye a `get()` metódussal kérhető le. Az `execute()` és `get()` futtatása nem jár szálváltással, azaz a hívó kód szálján (aktuális szál) futnak.

2. A *munkaszál* (worker thread), a háttérben végzi a munkát. Az osztályunkban implementált `doInBackground()` metódus fut ezen a szálon, itt végződik el minden időigényes munka.

3. *Eseménykezelő szál*, amely a megjelenítés frissítését végzi. A `process()` és `done()` metódusok segítségével lehet frissíteni a grafikus felületen megjelenített adatokat. A `process()` metódus akkor fut le, amikor a `doInBackground()` metódusból `publish()` hívás történik, a `done()` a `doInBackground()` lefutása után.

A következő példa megmutatja, hogyan lehet a `SwingWorker` osztályt felhasználni. A `PrimeNumbersTask` egy olyan szálat hoz létre, amely prímszámok listáját adja vissza, méghozzá annyit belőlük, amennyit a konstruktor paramétereként megkap. Mivel a rendszer meghívja a `publish()` metódust minden egyes kiszámolt számmal, így a process metódusban a számolás során folyamatosan frissül a szövegmező tartalma a kiszámolt értékekkel.

```
class PrimeNumbersTask extends SwingWorker<List<Integer>, Integer> {
      int numbersToFind;

      PrimeNumbersTask(JTextArea textArea, int numbersToFind) {
            // initialize
            this.numbersToFind = numbersToFind;
      }

      @Override
      public List<Integer> doInBackground() {
            List<Integer> numbers = new ArrayList<>();
            while (!enough && !isCancelled()) {
                  number = nextPrimeNumber();
                  numbers.add(number);
                  publish(number);
                  setProgress(100 * numbers.size() / numbersToFind);
            }
            return numbers;
      }
      @Override
      protected void process(List<Integer> chunks) {
            for (int number : chunks) {
                  textArea.append(number + "\n");
            }
      }
```

```
}

public void runTask() throws InterruptedException, ExecutionException {

      final JProgressBar progressBar = new JProgressBar(0, 100);
      PrimeNumbersTask task = new PrimeNumbersTask(textArea, 100);
      task.addPropertyChangeListener(new PropertyChangeListener() {
            public void propertyChange(PropertyChangeEvent evt) {
                  if ("progress".equals(evt.getPropertyName())) {
                        progressBar.setValue((Integer)
                           evt.getNewValue());
                  }
            }
      });

      task.execute();
      System.out.println(task.get()); // prints all prime numbers we
         have got
}
```

3. A mérés elvégzése

3.1. Mérési feladatok

A mérés előkészítése során el kell készíteni egy üzleti folyamat modellt, amely munkafolyamatot kell a félév során a különböző bemutatott technikákkal megvalósítani. A folyamat meglétét az első mérés előtt ellenőrizzük.

Szintén érdemes lehet az üzleti logikát megvalósító Java osztályokat előzetesen elkészíteni (noha lehetséges csak a mérés során elkészíteni): a munkafolyamat minden csomópontjához tartozzon egy osztály, ami a bemenő objektumon elvégzi a megfelelő műveletet, és visszaadja az új objektumot (akár új típussal) a kimenetre. A mérés kódját Java 7 nyelven kell implementálni.

Ha az üzleti logika osztályai megvannak, akkor el lehet készíteni minden egyes munkafolyamat csomóponthoz egy ablakot, ami jelzi, hogy a folyamat milyen állapotban van, lehetőséget adva a hibadetektálásra. Minden ablakon legyen egy gomb is, amivel vezérelni lehet a futtatást (pl. következő lépésre ugrás). A későbbi mérések szempontjából lényeges, hogy minden csomópontnak külön ablaka legyen!

Ezt követi a kommunikációért felelős szálak megalkotása. Minden csomópont külön szálon fut, és a folyamat állapota legyen nyomonkövethető. Elfogadható megoldás egy közös vezérlő szál létrehozása, amelyen keresztül kommunikálnak a munkafolyamat elemei.

Ezekből az elemekből szükséges felépíteni az előzetesen modellezett üzleti folyamatot. Érdemes lehet a tesztelhetőség érdekében az implementációt részlegesen elkészíteni (például az első két csomópont), és a tapasztalatok alapján építeni tovább a folyamatot.

3.2. A mérés kiértékelése

A mérés után szükséges, hogy bemutassuk a munkafolyamatot futás közben, valamint egy mérési jegyzőkönyv készítése.

A mérési jegyzőkönyvnek tartalmaznia kell a munkafolyamat leírását (képpel), megvalósításának magas szintű, architekturális leírását, valamint a lényeges implementációs részeket kiemelve. A forráskódot exportált Eclipse projektként kell mellékelni, a jegyzőkönyvben legfeljebb csak a különösen érdekes részeket kell kiemelni.

Az értékelés részben a kódminőség, részben a jegyzőkönyv alapján történik. A kódminőséghez hozzátartozik a megfelelő hibakezelés (a kivételek kiírása a normál kimenetre nem számít hibakezelésnek), a forráskód olvashatóság (beleértve tördelést is – ajánlott az automatikus kódtördelés használata).

4. További segédanyagok

A részletes, pontos referenciák használata szükséges a mérés sikeres teljesítéséhez:

- Java Concurrency Tutorial: `http://download.oracle.com/javase/tutorial/essential/concurrency/index.html`
- Exploring Java: Chapter 6, Thread: `http://oreilly.com/catalog/expjava/excerpt/index.html`
- A Visual Guide to Swing Components (Java Look and Feel): `http://download.oracle.com/javase/tutorial/ui/features/components.html`
- Using Swing Components Tutorial: `http://download.oracle.com/javase/tutorial/uiswing/components/index.html`
- Java Concurrency in Swing Tutorial: `http://download.oracle.com/javase/tutorial/uiswing/concurrency/index.html`
- Javadoc:
  - `http://download.oracle.com/javase/6/docs/api/java/util/concurrent/BlockingQueue.html`
  - `http://download.oracle.com/javase/6/docs/api/javax/swing/SwingWorker.html`
  - `http://download.oracle.com/javase/6/docs/api/java/lang/Thread.html`
  - `http://download.oracle.com/javase/6/docs/api/java/util/concurrent/ConcurrentMap.html`

5. Ellenőrző kérdések

1. Mire való a `synchronized` kulcsszó?
2. Mi a lényeges különbség a szinkronizált metódusok és blokkok használata között?
3. Mire való a `volatile` kulcsszó, és hogyan kell használni?
4. Mi a leglényegesebb különbség a `Runnable` és a `SwingWorker` interfész használata között?

5. Milyen szálon fut a `SwingWorker` osztály segítségével megírt háttértaszk?
6. Hogyan lehet egy szálat biztonságosan leállítani (`Thread.stop()` nem használható)?
7. Mit biztosít a `BlockingQueue`?