6. modul – Párhuzamosság 1.

Handout

Osztott rendszerek GY

Eötvös Loránd Tudományegyetem, 2020

Elmélet

Terminológia

- **Program:** Utasítások sorozata
- **Futási környezet:** Programok futtatásához használt adatszerkezet, tartalmazza többek között a program saját memóriáját és az utasítás-számlálót.
- **Folyamat:** Egy program és egy ahhoz kapcsolodó futási környezet együttese, melyet az operációs rendszer menedzsel.
- **Szál:** Program és egy ahhoz kapcsolódó futási környezet együttese, egy folyamaton belül.

További kifejezések

- Többszálú alkalmazás: Egynél több szálat futtató alkalmazás.
- Konkurens alkalmazás: Többszálú alkalmazás, ahol minden szál *halad* (mindig valamely utasítása előbb-utóbb végrehajtásra kerül). Megvalósítása: multitasking.
- Párhuzamos alkalmazás: *egyszerre* történő végrehajtás (pl. processzormagokon, számítógépeken).

További kifejezések

Alkalmazások

- Aszinkron input/output események:
  - Grafikus alkalmazások: Felhasználói felület és háttérfolyamatok
  - Többszálú szerverek
- Párhuzamosítás: Munkamegosztás processzormagok között
- Virtualizáció
- ...

Kapcsolódó témakörök

- Előadás: Folyamatok
- Előadás: Szinkronizáció
- Előadás: Konzisztencia
- Oracle: konkurens programozás (angol)

Gyakorlat

I. Szálak példányosítása

- Java nyelvben a szálakat a `Thread osztály` példányai reprezentálják.
- A példányosítás csupán létrehozza a reprezentációt, ezzel a szál még nem kezdi meg a futását: az alábbi megoldásoknak tehát még nem lesz látható eredménye.

1. megoldás

Szál létrehozásához implementáljuk a Runnable interfész `run` metódusát.

```
private static class MyRunnable implements Runnable {
  @Override
  public void run(){
    System.out.println("hello");
  }
}

public static void main(String[] args){
  Runnable r = new MyRunnable();
  Thread t = new Thread(r);
}
```

2. megoldás

A `Runnable` objektumokat nagyon ritkán szoktuk újrafelhasználni, ezért nincs igazán értelme nevet adni a saját osztályunknak. Nevesített osztály helyett implementáljuk az interfészt névtelen osztállyal.

```
1  public static void main(String[] args){
2    Runnable r = new Runnable(){
3      @Override
4      public void run(){
5        System.out.println("hello");
6      }
7    };
8    Thread t = new Thread(r);
9  }
```

3. megoldás

A Java 8 új, kényelmesebb szintaxist vezetett be azon objektumok számára, melyek függvénykifejezésként (lambda) viselkednek. A `Runnable` interfészt implementáló osztályok objektumai is függvénykifejezésként viselkednek, ezért itt is használhatjuk az új, rövidebb szintaxist: `() -> { System.out.println("hello"); }` egy paraméter és visszatérési érték nélküli függvénykifejezés, melynek mellékhatása a `System.out.println("hello")` utasítás végrehajtása. A fordító egy a korábbival megegyező hasonló névtelen `Runnable` osztályra fordítja le ezt a jelölést.

```
1 public static void main(String[] args){
2   Runnable r = () -> { System.out.println("hello"); };
3   Thread t = new Thread(r);
4 }
```

2. Szálak indítása

A szálak saját memóriával és utasításszámlálóval rendelkeznek, valamint hozzáférnek egy közös memóriaterülethez is. A JVM úgy hajtja végre a szálak utasításait, mintha azok egymással egy időben futnának.

Egy-egy szálat a `Thread` osztály `start` metódusával indíthatunk el.

```
1 public static void main(String[] args){
2   Runnable r = () -> { System.out.println("hello"); };
3   Thread t = new Thread(r);
4   t.start();
5   System.out.println("world");
6 }
```

Lényeges, hogy ebben a példában nem egy, hanem két szál indul el:

- minden Java program rendelkezik egy ún. főszállal. Azok a programok, amelyekben nincs – pl. a `Thread` osztály segítségével történő – szálkezelés, egyszálú programok.
- a `t` referencia által hivatkozott objektum is egy szálat reprezentál, mely a `start` metódus hívásának pillanatában kezdi meg futását. Ekkor a főszál és a `t` szál együtt futnak.

3. Nem-determinisztikus végrehajtás

A Java a szál saját utasításait mindig a `run` metódusban leírt vezérlésnek megfelelő sorrendben hajtja végre. Semmilyen garanciát nem ad azonban arra nézve, hogy az egymással párhuzamosan futó szálak utasításai egymáshoz képest milyen sorrendben fognak végrehajtódni.

A korábbi példában nem tudjuk előre megmondani, hogy a gyerekszál `System.out.println("hello");` utasítása, vagy a főszál `System.out.println("world");` utasítása kerül korábban végrehajtásra.

Ennek megfelelően a futtatás után a sztenderd kimeneten egyaránt megjelenhet kimenetként a

```
hello
world
```

felirat, és a

```
world
hello
```

felirat.

A konkrét sorrendet a JVM belső működése befolyásolja, ennek megfelelően tekinthetjük véletlenszerűnek. Ezt a jelenséget, ahol egy program egy véletlen faktortól függően több lehetséges végrehajtási úttal is rendelkezik, **nem-determinisztikus végrehajtásnak** nevezzük.

4. Versenyhelyzet, kölcsönös kizárás

Noha a nem-determinisztikus programoknak számos felhasználása van (pl. kriptográfia, játékfejlesztés), a többszálúságból fakadó nem-determinisztikusságnak káros mellékhatásai lehetnek és ezért jellemzően kerülnünk el.

Ilyen klasszikus problémakör az ún. **versenyhelyzet** (race condition) problémák köre.

- Versenyhelyzet során több, párhuzamosan futó szál próbál hozzáférni egy azon erőforráshoz, és a hozzáférésük sorrendje nem-determinisztikus.
- A korábbi példában a főszál és a `t` gyerekszál egymással "versengve" próbálnak hozzáférni a `System.out` sztenderd kimenethez, és nem tudjuk melyikük lesz a "győztes".

Kitérő:

- Érdemes megfigyelnünk, hogy akár tetszőleges futást követően is csak a szavak sorrendje változó: a karakterek mindig a megfelelő (determinisztikus) sorrendben követik egymást.
- Ez annak köszönhető, hogy a `println` metódus nem engedi, hogy a két szál egyszerre hajtsa végre: a később érkező szálnak várnak kell, amíg a hamarabb érkező szál végrehajtja a `println` metódust.
- Ezt a tulajdonságot – az erőforrás levédését *átlapoló* hozzáférésekkel szemben – **kölcsönös kizárásnak** nevezzük, és később tanulunk ennek a megvalósításáról is.

Számos versenyhelyzet-probléma létezik, melyekre különböző megoldások születtek.

Vizsgáljunk meg most néhány eszközt, amellyel garantálhatjuk, hogy a `"hello"` kiíratása determinisztikusan megelőzze a `"world"` kiíratását.

1. megoldás: Altatás (nem-determinisztikus)

A statikus `Thread.sleep` metódus a paraméterként miliszekundumban megadott ideig blokkolja az utasítást végrehajtó szálat. Blokkoljuk a `"world"` szót kiíró főszálat kellően hosszú ideig ahhoz, hogy előtte biztosan végrehajtódjon a `"hello"` szót kiíró gyerekszál.

```
1  public static void main(String[] args)
2      throws InterruptedException {
3
4    Runnable r = () -> { System.out.println("hello"); };
5    Thread t = new Thread(r);
6    t.start();
7
8    Thread.sleep(500);
9    System.out.println("world");
10 }
```

Felmerül a kérdés, hogy pontosan mennyit kell várnia a főszálnak?

- Számba véve a futás időt befolyásoló tényezőket (hardware környezet, operációs rendszer, JVM, felhasználó, stb.), be kell látnunk, hogy erre a kérdésre nem tudunk kielégítő választ adni: a futás idő nem-determinisztikus.
- 500 miliszekundum valószínűleg elég a jelenlegi körülmények között, de mi történik, ha egy korlátozott teljesítményű virtuális gépen futtatjuk a programot? Mi történik, ha a program továbbfejlesztése során a gyerekszálba hosszú, esetleg felhasználói interakciót is tartalmazó műveletek kerülnek be?

Az altatás tehát nem eliminálja a nem-determinisztikusságot: csak látszatmegoldás, és ezért produkciós kódban szinte **mindig kerülendő**.

2. megoldás: Tevékeny várakozás

A következő egyszerű megoldással biztosíthatjuk, hogy a főszál ne írjon ki semmit a gyerekszál befejeződéséig:

- vezessünk be egy logikai változót annak jelölésére, hogy befejeződött-e a gyerekszál
- amikor a gyerekszál véget ér, állítsa a változót igazra.
- a főszálat ún. **tevékeny várakozással** blokkoljuk: végtelen ciklusban olvassa a változót, és amikor a változó igazra vált, kilép a ciklusból és elvégzi a kiíratást

```java
public volatile static boolean b = false;

public static void main(String[] args) {
  Runnable r = () ->
    { try {
        Thread.sleep(500);
      } catch(InterruptedException e){
      }
      System.out.println("hello");
      b = true;
    };

  Thread t = new Thread(r);
  t.start();

  while(!b){
  }
  System.out.println("world");
}
```

Ez a megoldás minden körülmény között garantálja, hogy a főszál csak azután írja ki a `"world"` szót, miután a gyerekszál kiírta a `"hello"` szót.

- Ennek illusztrálására a gyerekszál futásidejét 500 miliszekundummal kitoltuk: mivel a program már nem függ az egyes műveletek futásidejétől, az altatás sem befolyásolta károsan az eredményt
- A `volatile` kulcsszóról később tanulunk.

A tevékeny várakozás determinisztikus és helyes eredményt ad, azonban **a végtelen ciklus annak ellenére pörgeti a CPU-t, hogy semmilyen hasznos munkát nem végez**.

- Tevékeny várakozás helyett **érdemes inkább a Java által kínált lehetőségre támaszkodnunk** a blokkoláshoz: a *szinkronizációs primitívekre* és az ezeket alkalmazó könyvtárakra (pl. `java.util.concurrency`).
- Ezek egyszerűen csak félrerakják a feladatot későbbre, és ezért nem fogyasztanak felesleges erőforrásokat.

3. megoldás: `Thread.join`

A jelen problémának egy egyszerű és elegáns megoldását adja, ha kihasználjuk a `Thread` osztály lehetőségeit: a `t.join` metódus mindaddig blokkolja a metódust meghívó szálat (jelen esetben a főszálat), amíg a `t` szál véget nem ér.

```
1  public static void main(String[] args)
2      throws InterruptedException {
3    Runnable r = () ->
4      { try { Thread.sleep(500);
5        } catch(InterruptedException e){
6        }
7        System.out.println("hello");
8      };
9    Thread t = new Thread(r);
10   t.start();
11   t.join();
12   System.out.println("world");
13 }
```

4. megoldás: Zárak

Az esetek többségében érdemes a mások által implemenált magasszintű megoldásokat használnunk.

- A Java ilyen megoldásokat tartalmazó beépített csomagja a `java.util.concurrency`.

Az alábbiakban a tevékeny várakozásnál bemutatott algoritmust a `CountDownLatch` osztály segítségével valósítottuk meg.

- Ez egy számlálót tárol, aminek kezdeti értékét a konstruktorban állíthatjuk be (`1`).
- Az `await` metódus mindaddig blokkol, amíg a számláló értéke el nem éri a `0`-t.
- A `countDown` metódus egyel csökkenti a számlálót.

```java
import java.util.concurrency.CountDownLatch;
...
static CountDownLatch latch = new CountDownLatch(1);
public static void main(String[] args)
    throws InterruptedException {
  Runnable r = () ->
    { try { Thread.sleep(500);
      } catch(InterruptedException e){
      }
      System.out.println("hello");
      latch.countDown();
    };
  Thread t = new Thread(r);
  t.start();
  latch.await();

  System.out.println("world");
}
```