Java tutorial

Párhuzamosság

- A program egyszerre több mindent is csinálhat
- Lehetőségek:
  - Számítással egyidőben IO
  - Több processzor: számítások egyidőben
  - Egy processzor több programot futtat (process)
    - többfelhasználós rendszer
    - időosztásos technika
  - Egy program több végrehajtási szálból áll (thread)

Cél

- Hatékonyság növelése: ha több processzor van
- Több felhasználó kiszolgálása
  - egy időben
  - interakció
- *Program logikai tagolása*
  - az egyik szál a felhasználói felülettel foglalkozik
  - a másik szál a hálózaton keresztül kommunikál valakivel

Alkalmazási területek

- Számításilag nehéz problémák
  - pl. időjárás-előrejelzés
- Valós idejű alkalmazások
- Operációs rendszerek
- Folyamatszabályozási feladatok
- Szimulációk
  - pl. ágensek
- Elosztott rendszerek
  - pl. repülőtéri helyfoglalás

Párhuzamossági modellek

- Megosztott (shared)
  - Több processzor, ugyanaz a memóriaterület
- Elosztott (distributed)
  - Több processzor, mindnek saját memória
  - Kommunikációs csatornák, üzenetküldés
- Mi van, ha csak egy processzor van?

Ha egy processzor van...

- A processzor kapcsolgat a különböző folyamatok között
- Mindegyiket csak kis ideig futtatja
- A párhuzamosság látszata
- A processzoridő jó kihasználása
  - Blokkolt folyamatok nem tartják fel a többit
  - A váltogatás is időigényes!

Párhuzamosság egy folyamaton belül

- Végrehajtási szálak (thread)
- *Ilyenekkel fogunk foglalkozni*
- Pehelysúlyú (Lightweight, kevés költségű)
- Leginkább a feladat logikai darabolásából
  - De elképzelhető, hogy különböző processzorokra kerülnek a szálak
- „Megosztott” jelleg: közös memória

Szálak Java-ban

- Beépített nyelvi támogatás: java.lang
- Nyelvi fogalom: (végrehajtási) szál, thread
- Támogató osztály: java.lang.Thread

Vigyázat!

- Nagyon nehéz párhuzamos programot írni!
- Az ember már nem látja át
- Sok probléma
  - kommunikáció
  - szinkronizáció
  - ütemezés
  - interferencia

Java tutorial

Végrehajtási szálak létrehozása

- A főprogram egy végrehajtási szál
- További végrehajtási szálak hozhatók létre
- Egy **`Thread`** osztályba tartozó objektumot létre kell hozni
- Az objektum **`start()`** metódusával indítjuk a végrehajtási szálat
- A szál programja az objektum **`run()`** metódusában van

Példa

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

- Hát ez még semmi különöset sem csinál, mert üres a **`run()`**

Példa

```
class Hello {
  public static void main(String args[]){
    (new Thread()).start();
  }
}
```

- Hát ez még semmi különöset sem csinál, mert üres a `run()`

Példa a run() felüldefiniálására

```
class Hello {
  public static void main(String args[]){
    (new MyThread()).start();
  }
}
class MyThread extends Thread {
  public void run(){
    while(true) System.out.println("Hi!");
  }
}
```

Példa a run() felüldefiniálására

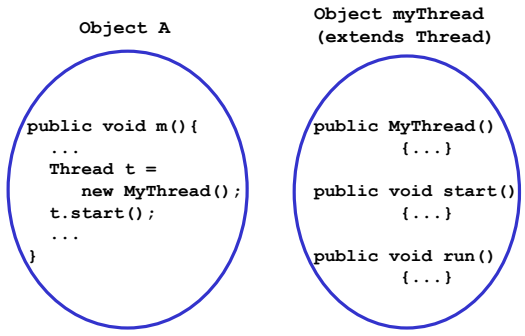
```
class Hello {
  public static void main(String args[]){
    (new Thread(){
            public void run(){
              while(true)
                System.out.println("Hi!");
            }
          }).start();
  }
}
```

Az elindítás

- Nem elég létrehozni egy Thread objektumot
  - A Thread objektum nem a végrehajtási szál
  - Csak egy eszköz, aminek segítségével különböző dolgokat csinálhatunk egy végrehajtási szállal
- Meg kell hívni a start() metódusát
- Ez automatikusan elindítja a run() metódust
- Ezt a run()-t kell felüldefiniálni, megadni a szál programját

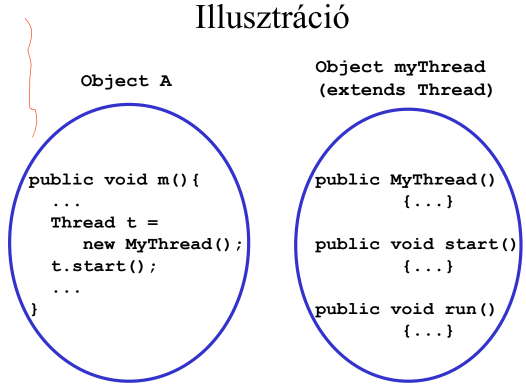
Java tutorial

Illusztráció

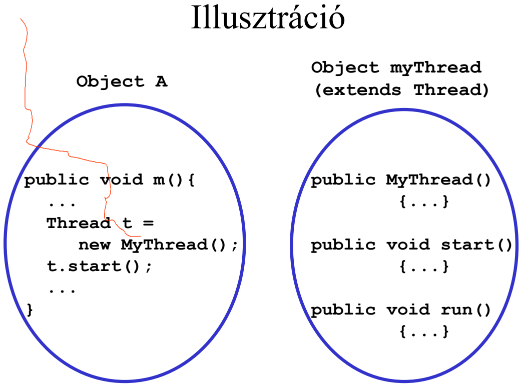


Java tutorial

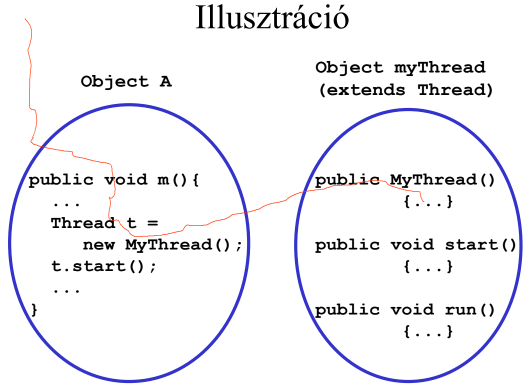
Illusztráció



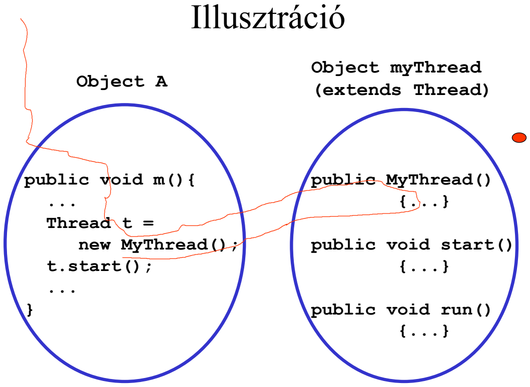
Illusztráció



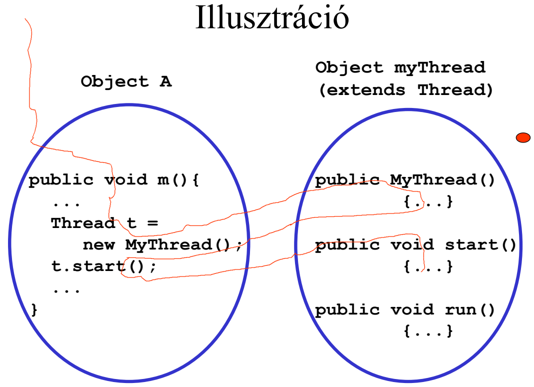
Illusztráció



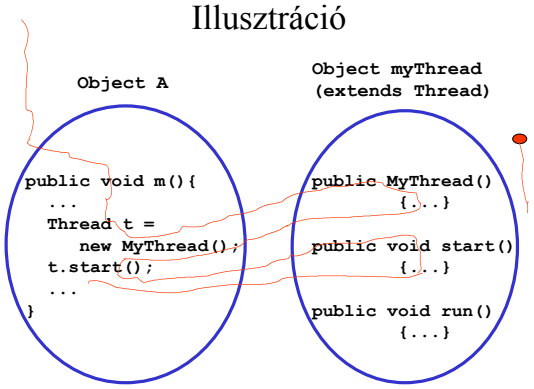
Illusztráció



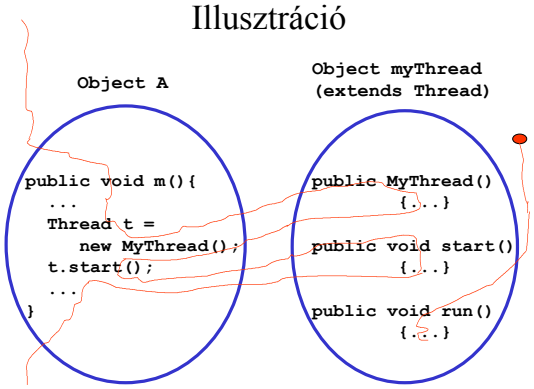
Illusztráció



Illusztráció



Illusztráció



Java tutorial

Kérdés

```
class Hello {
  public static void main(String args[]){
    (new Thread()).start();
  }
}
```

- Hány szálon fut ez a program?
  Hány végrehajtási szál van?

Mit csinál ez a program?

```
class Hello {
  public static void main(String args[]){
    (new MyThread()).start();
    while(true) System.out.println("Bye");
  }
}
class MyThread extends Thread {
  public void run(){
    while(true) System.out.println("Hi!");
  }
}
```

Feladat

- Próbáljuk ki, hogy mit csinál az előző program, de...
- ... legyen három szál
  - hozzunk létre két példányt a MyThread osztályból
  - inicializáljuk a két példányt különböző stringekkel
  - ezeket a stringeket kell kiírniuk

Mit kéne csinálnia?

- Definiálatlan
- Ütemezéstől függ
- A nyelv definíciója nem tesz megkötést az ütemezésre
- Különböző platformokon / virtuális gépeken különbözőképpen működhet
- A virtuális gép meghatároz(hat)ja az ütemezési stratégiát
- De azon belül is sok lehetőség van
- Sok mindentől függ (pl. hőmérséklettől)

Ütemezés különböző platformokon

- Solaris alatt: egy szál addig fut, amíg csak lehet (hatékonyabb)
- NT alatt: időosztásos ütemezés (igazságosabb)

- Összefoglalva: írjunk olyan programokat, amelyek működése nem érzékeny az ütemezésre

Pártatlanság (fairness)

- Ha azt akarjuk, hogy minden szál „egyszerre”, „párhuzamosan” fusson
- Ha egy szál már „elég sokat” dolgozott, adjon lehetőséget más szálaknak is
- `yield()`
- Ezen metódus meghívásával lehet lemondani a vezérlésről
- A Thread osztály statikus metódusa

Mit csinál ez a program?

```
class Hello {
  public static void main(String args[]){
    (new MyThread()).start();
    while(true){
      System.out.println("Bye");
      Thread.yield();
    }
  }
}
...
```

Feladat

- Írjuk be a yield()-eket a szálainkba!

Java tutorial

A másik út...

- Java-ban egyszeres öröklődés
- Végrehajtási szálnál leszármaztatás a Thread osztályból
  - „elhasználja” azt az egy lehetőséget
- Megoldás: ne kelljen leszármaztatni
- Megvalósítjuk a `Runnable` interfészt, ami előírja a `run()` metódust
- Egy Thread objektum létrehozásánál a konstruktornak átadunk egy futtatható objektumot

Példa a run() megadására

```
class Hello {
  public static void main(String args[]){
    (new MyThread()).start();
  }
}
class MyThread extends Thread {
  public void run(){
    while(true) System.out.println("Hi!");
  }
}
```

Példa a run() megadására

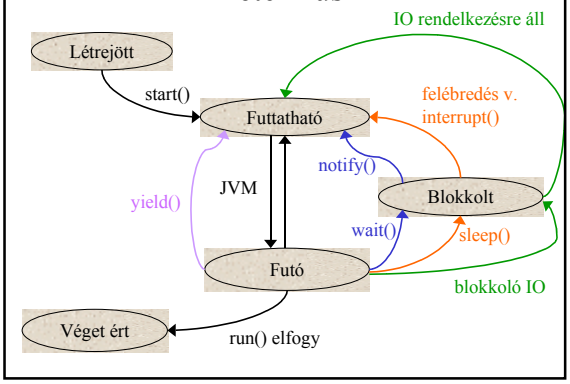
```
class Hello {
  public static void main(String args[]){
    (new Thread(new MyRunnable())).start();
  }
}
class MyRunnable implements Runnable {
  public void run(){
    while(true) System.out.println("Hi!");
  }
}
```

Tipikus példa

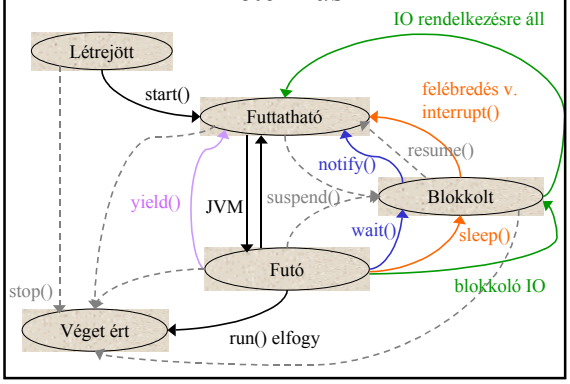
```
class MyApplet extends Applet
               implements Runnable {
  public void run(){
    // animáció megjelenítése
  }

  public void start(){
    (new Thread(this)).start();
  }

  ...
}
```

Java tutorial

Életciklus



Életciklus




```
import java.util.*; import java.io.*;
class SleepDemo extends Thread {

   public void run(){
      while(true){
         try { sleep(1000); }
             catch (InterruptedException ie){}
         System.out.println(new Date());
      }
   }

   public static void main(String[] args) {
      (new SleepDemo()).start();
      while(true){
         System.err.println();
      }
   }

}
```

```
import java.util.*; import java.io.*;
class BlokkolóDemo extends Thread {

   public void run(){
      while(true){
         try { System.in.read(); }
             catch (IOException ie){}
         System.out.println(new Date());
      }
   }

   public static void main(String[] args) {
      (new BlokkolóDemo()).start();
      while(true){
         System.err.println();
      }
   }

}
```

Java tutorial

Feladat

- A mezőn (ami a képernyő megfelelője) egy nyúl és egy róka bóklászik. A nyulat mi irányítjuk a szabványos bemenetről az 'q', 'w', 'e', 'a', 's', 'd', 'y', 'x' és 'c' billentyűk segítségével. A róka a nyúl felé igyekszik. Az állatok a 8 velük szomszédos mezőre léphetnek át egy lépésben. A róka kb. kétszer olyan gyakran lép, mint a nyúl, viszont minden irányváltoztatás előtt meg kell állnia.

Szál leállítása

- A stop() metódus nem javasolt.
- Bízzuk rá a szálra, hogy mikor akar megállni.
- Ha a run() egy ciklus, akkor szabjunk neki feltételt.
  - Gyakran egy sleep() is van a ciklusban.
  - A feltétel egy flag-et figyelhet, amit kívülről átbillenthetünk.
- Feladat: állítsuk le a nyuszit, ha megfogta a róka!

Példa

```
class MyApplet extends Applet
                implements Runnable {
  private boolean fut = false;
  public void start(){...}
  public void stop(){...}
  public void run(){...}
  ...
}
```

```
public void start(){
    fut = true;
    (new Thread(this)).start();
}
                          public void stop(){
                              fut = false;
                          }
public void run(){
    while(fut){
       ...    // animáció egy lépése
       try{ sleep(100); }
       catch(InterruptedExeption e){...}
    }
}
```

Java tutorial

Az első probléma: interferencia

- Két vagy több szál, noha külön-külön jók, együtt mégis butaságot csinálnak: a || b ≠ ab V ba
  - felülírják egymás eredményeit
  - inkonzisztenciát okoznak
- Például két szál ugyanazon az adaton dolgozik egyidejűleg

```
class Számla {
  int egyenleg;
  public void rátesz(int összeg){
    int újEgyenleg;
    újEgyenleg = egyenleg+összeg;
    egyenleg = újEgyenleg;
  }
  ...
}
```

Interferencia ellen: szinkronizáció

Az adatokhoz való hozzáférés szerializálása

Kölcsönös kizárás

Kritikus szakaszok védelme

- Szemafor - mint vonatoknál
  - P és V művelet
  - veszélyes programozás, rossz minőségű kód
- **Monitor**
  - adatok + műveletek kölcsönös kizárással
  - jól illeszkedik az objektum-elvű szemlélethez
- Író-olvasó
  - J-ban nincs olyan nyelvi elem, mint pl. Ada protected

Példa: „thread-safe” számla

```
class Számla {
  private int egyenleg;
  public synchronized void rátesz(int összeg){
    int újEgyenleg;
    újEgyenleg = egyenleg+összeg;
    egyenleg = újEgyenleg;
  }
  ...
}
```

Java tutorial

A synchronized kulcsszó

- Metódusok elé írhatjuk (de pl. interfészekben nem!)
- Kölcsönös kizárás arra a metódusra, sőt...
- Kulcs (lock) + várakozási sor
  - A kulcs azé az objektumé, amelyiké a metódus
  - Ugyanaz a kulcs az összes szinkronizált metódusához
  1. Mielőtt egy szál beléphetne egy szinkronizált metódusba, meg kell szereznie a kulcsot
  2. Vár rá a várakozási sorban
  3. Kilépéskor visszaadja a kulcsot
- A nyulas-rókás feladatban hol van szükség szinkronizációra? Írjuk bele!

```
class Számla {
  private int egyenleg;
  public synchronized void rátesz(int összeg){
    int újEgyenleg;
    újEgyenleg = egyenleg+összeg;
    egyenleg = újEgyenleg;
  }
  public synchronized void kivesz(int összeg)
  throws SzámlaTúllépésException {
    if( egyenleg < összeg )
       throw new SzámlaTúllépésException();
    else egyenleg -= összeg;
  }
  ...
}
```

Szinkronizált blokkok

- A synchronized kulcsszó védhet blokk utasítást is
- Ilyenkor meg kell adni, hogy melyik objektum kulcsán szinkronizáljon

```
synchronized(obj){...}
```

- Metódus szinkronizációjával egyenértékű

```
public void rátesz(int összeg){
    synchronized(this){ ... }
}
```

- Ha a számla objektum rátesz metódusa nem szinkr.

```
...
synchronized(számla){ számla.rátesz(100); }
...
```

Szinkronizált blokkok (2)

- Sokszor úgy használjuk, hogy a monitor szemléletet megtörjük
- Nem az adat műveleteire biztosítjuk a kölcsönös kizárást, hanem az adathoz hozzáférni igyekvő kódba tesszük
- A kritikus szakasz utasításhoz hasonlít
- Létjogosultság: ha nem egy objektumban vannak azok az adatok, amelyekhez szerializált hozzáférést akarunk garantálni
  - Erőforrások kezelése, tranzakciók
  - Később látunk példát...

Statikus szinkronizált metódusok

```
class A {
  static synchronized void m(...){...}
}
```

- Milyen objektum kulcsán szinkronizálunk?
- Önelemzés: az osztályok futási idejű reprezentációja a virtuális gépben egy `Class` osztályú objektum - ezen
- Kérdés: ezek szerint futhatnak egyidőben szinkronizált statikus és példánymetódusok?

Amikor nem kell szinkronizálni...

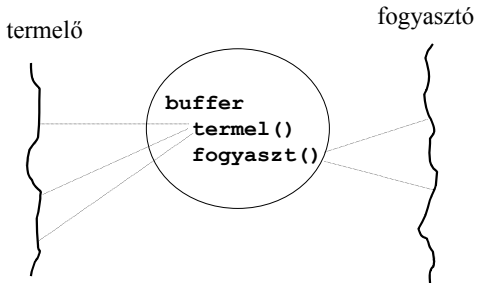
- Atomi műveletek Java-ban: értékadás volatile változóknak
  - a **`volatile`** egy módosítószó
  - egyesek szerint még ez sem kell primitív típusú változók esetén, kivéve a **`long`** és a **`double`** változókat
- Ha a jobboldalon nem szerepel a változó
- Ha már több változó összefügg (típusinvariáns), akkor szerializáljuk a hozzáférést synchronized-dal
- Feladat: ez alapján a nyulászatból kivehető egy pár szinkronizálás...

Java tutorial

wait - notify

- Szignálokra hasonlít
- Egy feltétel teljesüléséig blokkolhatja magát egy szál
- A feltétel (potenciális) bekövetkezését jelezheti egy másik szál
- Alapfeladat: termelő - fogyasztó (korlátos) bufferen keresztül kommunikálnak
  - egy termelő, egy fogyasztó
  - több termelő, több fogyasztó

Termelő-fogyasztó



A termelő szál definíciója

```
public class Termelő extends Thread {
  Buffer buffer;
  public Termelő( Buffer buffer ){
    this.buffer = buffer;
  }
  public void run(){
    while(true){
      char ch = (char) System.in.read();
      buffer.termel(new Character(ch));
    }
  }
}
```

A termelő szál definíciója

```
public class Termelő extends Thread {
  Buffer buffer;
  public Termelő( Buffer buffer ){
    this.buffer = buffer;
  }
  public void run(){
    while(true){
      try{ char ch = (char) System.in.read();
           buffer.termel(new Character(ch));
      } catch ( IOException e ){}
    }
  }
}
```

Korlátlan buffer, egy fogyasztó

```
public class Buffer extends Sor {
  public synchronized Object fogyaszt(){
    if( üres() )  wait();
    return kivesz();
  }
  public synchronized void termel(Object o){
    betesz(o);
    notify();
  }
}
```

Korlátlan buffer, egy fogyasztó

```
public class Buffer extends Sor {
  public synchronized Object fogyaszt(){
    if( üres() )
       try{ wait(); }
       catch(InterruptedException e){}
    return kivesz();
  }
  public synchronized void termel(Object o){
    betesz(o);
    notify();
  }
}
```

Korlátlan buffer, egy fogyasztó

```
public class Buffer extends Sor {
  public synchronized Object fogyaszt(){
    if( üres() )
       try{ wait(); }
       catch(InterruptedException e){return null;}
    return kivesz();
  }
  public synchronized void termel(Object o){
    betesz(o);
    notify();
  }
}
```

Java tutorial

Működés

- Minden objektumhoz tartozik a sima kulcshoz tartozó várakozási soron kívül egy másik, az ún. *wait-várakozási sor*
- A wait() hatására a szál bekerül ebbe
- A notify() hatására az egyik várakozó kikerül belőle
- A wait() és notify() hívások csak olyan kódrészben szerepelhetnek, amelyek ugyanazon az objektumon szinkronizáltak

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

- A szál megszerzi az objektum kulcsát, ehhez, ha kell, sorban áll egy ideig (synchronized)
- A wait() hatására elengedi a kulcsot, és bekerül a wait-várakozási sorba
- Egy másik szál megkaparinthatja a kulcsot (kezdődik a synchronized)
- A notify() metódussal felébreszthet egy wait-es alvót, aki bekerül a kulcsos várakozási sorba
- A synchronized végén elengedi a kulcsot
- A felébredt alvónak (is) lehetősége van megszerezni a kulcssot és továbbmenni
- A synchronized blokkja végén ő is elengedi a kulcsot

Korlátlan buffer, több fogyasztó

```
public class Buffer extends Sor {
  public synchronized Object fogyaszt(){
    while( üres() )
       try{ wait(); }
       catch(InterruptedException e){}
    return kivesz();
  }
  public synchronized void termel(Object o){
    betesz(o);
    notifyAll();
  }
}
```

Nincs szükség busy waiting-re

Feladatok

- Fejezzük be a termelő-fogyasztó feladatot a fogyasztó implementálásával
- Hogyan valósítanánk meg a szemaforokat Java-ban?
- És az író-olvasó problémát?
- Javítsunk a nyulas-rókás programunkon! A képernyőre akkor kell kirajzolni a mezőt, ha valami változás történt...

Java tutorial

Szálak kommunikációja

- Aszinkron leginkább
- Szinkron kommunikáció (pl. Ada randevú) helyett csak wait-notify szinkronizáció van
- Megosztott és elosztott szemlélet egyaránt
- Megosztott: közös memórián keresztül
- Elosztott: üzenetküldés csatornákkal (Piped csatornák)
- Termelő-fogyasztó

Közös memórián keresztül

- A kommunikációra használt objektumokra referenciákat lehet átadni, eljuttatni a szálakba
  - pl. a róka a nyúlról így szerez információt
- Végletes esetben magát a futtatható objektumot is megoszthatjuk:

```
class A implements Runnable {
  ... // adatok, műveletekkel
  public void run(){ ... }
}
```

```
A a = new A();
Thread t1 = new Thread(a);
Thread t2 = new Thread(a);
```

Piped csatornaosztályok

- PipedInputStream, PipedOutputStream PipedReader, PipedWriter
- Egy bemenetit és egy kimenetit összekapcsolunk
- A cső egyik végére az egyik szál ír, a cső másik végéről a másik szál olvassa
- Vigyázat: az olvasás blokkoló művelet!
  - available(), ready() - bár ez utóbbival rosszak a tapasztalatok

```
PipedReader r = new PipedReader();
PipedWriter w = new PipedWriter(r);
(new TermelőSzál(w)).start();
(new FogyasztóSzál(w)).start();

public class TermelőSzál extends Thread {
  ...
  Writer w;
  public TermelőSzál( Writer w ){ this.w = w; }
  public void run(){
      while(fut){
         char adat = ...  // termel
         w.write(adat); w.flush();
      }
  }
}
```

Java tutorial

Holtpont (deadlock)

- Néhány folyamat véglegesen blokkolódik, arra várnak, hogy egy másik, szintén a holtpontos halmazban levő folyamat csináljon valamit
- Az interferencia tökéletes kiküszöbölése :-)
- Túl sok a szinkronizáció
- Gyakran erőforrás-kezelés vagy tranzakciók közben
- Példa: étkező filozófusok (dining philosophers)

Mit lehet tenni?

- Nincs univerzális megoldás, a programozó dolga a feladathoz illeszkedő megoldás kidolgozása
- Detektálás, megszüntetés (pl. timeout), előrejelzés, megelőzés
- Megelőzés: erőforrások sorrendbe állítása, szimmetria megtörése, központi irányítás, véletlenszerű várakoztatás, stb.
  - pl. a filozófusok...

```
class A {
  synchronized void m1(){...}
  synchronized void m2(B b){... b.m1() ...}
}
class B {
  synchronized void m1(){...}
  synchronized void m2(A a){... a.m1() ...}
}

      A a = new A(); B b = new B();
```

Egyik szálban `a.m2(b);`
Másik szálban `b.m2(a);`

```
class A {
  void m1(){...}
  void m2(B b){... b.m1() ...}
}
class B {
  void m1(){...}
  void m2(A a){... a.m1() ...}
}

        A a = new A(); B b = new B();
          Object o = new Object();
```

Egyik szálban `synchronized(o) {a.m2(b);}`
Másik szálban `synchronized(o) {b.m2(a);}`

Kiéheztetés (starvation, livelock)

- A rendszer nem áll le, a folyamatok mennek, de van olyan köztük, amelyik nem tudja azt csinálni, amit szeretne
- Pl. több fogyasztó közül az egyik el van hanyagolva. Vagy az egyik filozófus sosem eszik.
- Amikor a folyamatra kerül a vezérlés,ő kezd futni, de akkor épp nem teljesül a továbbhaladási feltétel
- Könnyen bekövetkezhet wait-notify mellett
- Kivédés/megelőzés még nehezebb, mint a holtpont esetén

Prioritás

- A szálakhoz prioritás van rendelve
- Prioritási szintek: 1 (MIN_PRIORITY) és 10 (MAX_PRIORITY) között
  - Rendszerszintű szálak prioritása 11
- A Java a prioritási szintek között preemptív ütemezést kér: a magasabb prioritású futhat
  - Sajnos a JVM-nek nem kötelező ezt betartania
  - Ne ezen múljon a programunk helyessége
- Egy szinten belül definiálatlan
  - SUN „Green" szálak: kooperatív
  - SUN „Natív" szálak: preemptív

Prioritás (folyt.)

- Ha egy szál futtathatóvá válik (pl. felébred vagy elindul), és egy alacsonyabb prioritású fut éppen, akkor az átadja a vezérlést
- Prioritási szint lekérdezése és beállítása:
  ```
  int getPriority()
  void setPriority(int priority)
  ```
- Szál létrehozásakor a létrehozott örökli a létrehozó prioritását, ha mást nem adunk meg
- Általános stratégia: sokat dolgozó szálnak kisebb prioritást érdemes adni

Szálcsoportok

- A szálak szálcsoportokba sorolhatók
  - Szál létrehozásakor
- java.lang.ThreadGroup
- Logikailag összetartozó szálak lehetnek egy csoportban
- A csoport maximális prioritása beállítható, a csoportbeli szálaké ezt nem fogja meghaladni
- A szálak lekérdezhetnek információkat a saját csoportjukról
- A szálcsoportok hierarchiába szervezhetők

Démonok

- Szálak lehetnek démonok
- Ha a nem démon szálak véget érnek, akkor a program is véget ér, és a démon szálak a háttérben futnak tovább
- Az elindítás, azaz a start() előtt be kell állítani
  ```
  boolean isDaemon()
  void setDaemon(boolean on)
  ```

Még néhány művelet

- join()
  A futó szál blokkolódik, amíg az a szál véget nem ér, amelyiknek a join-ját meghívta
  - időkorlát adható meg
- interrupt()
  Egy blokkolt állapotú szálat (sleep, wait) felébreszt. A felébredt szálban a blokkoló utasítás helyén InterruptedException lép fel
- stop(), suspend() és resume()
  Elavultak, mert veszélyesek (helyesség, holtpont)