Pázmány Péter Katolikus Egyetem
Információs Technológiai és Bionikai Kar

Szálkezelés I.

Java programozás

5. gyakorlat

Multitasking

- Miért?
  - lassú IO műveletek alatt is lehessen feladatokat végezni
  - tudjon több program egyszerre futni
    - háttérzene, chat ablak a háttérben stb.
  - egy program tudjon egyszerre több feladatot is elvégezni
    - pl.: egy szerver tudjon több klienssel is kommunikálni, egy böngészőben lehessen több fület nyitni
  - algoritmusok párhuzamosítása a gyorsabb futás érdekében

Multitasking

- Egy átlagos személyi számítógépben a processzor általában néhány (2-8) magot tartalmaz, amelyek mindegyike egy valós végrehajtási szálat futtathat.
  - +HyperThreading: egy fizikai processzormag az operációs rendszer felé két logikai magként jelenik meg, a különböző feladatokat a processzor hardveresen párhuzamosítja, így egy órajel alatt a fizikai magban éppen nem használt részek is feladatot végezhetnek
- Az operációs rendszer általában néhány száz folyamatot futtat látszólag párhuzamosan, azonban a valóságban nem lehet mindet egyszerre futtatni.
- Erre a problémára nyújt megoldást a multitasking, amely során a folyamatok versengenek a processzormagokért, a látszólagosan egyszerre futó programok valójában felváltva, nagyon gyorsan váltakozva futnak.

Process - Folyamat

- Az operációs rendszer szolgáltatásai és a felhasználói programok mind egy-egy (néha több) folyamatként (processzként) futnak.
- Minden folyamat egy saját futási környezettel rendelkezik, ami
  - privát memóriablokkokat,
  - saját hívási vermet (call stack),
  - látszólag kizárólagos regiszterhasználatot jelent.
- A multitasking operációs rendszer sűrűn váltogat a processzek között (task-switching), ilyenkor a processzor regisztereinek tartalmát lementi és visszatölti a következő processzhez tartozó állapotot (context-switching). Ezt a műveletet az OS megfelelően ütemezi (pl. időosztással), többmagos rendszer esetén a rendszer ezt egyszerre több magon tudja elvégezni.
- A folyamatok egymás létezéséről többnyire nem tudnak (egy erőforrás elérhetetlensége is csak közvetetten utal arra, hogy nem egyedül futnak).
- Egymással inter-process communication (IPC) protokollokon keresztül kommunikálnak, amit az operációs rendszer biztosít.

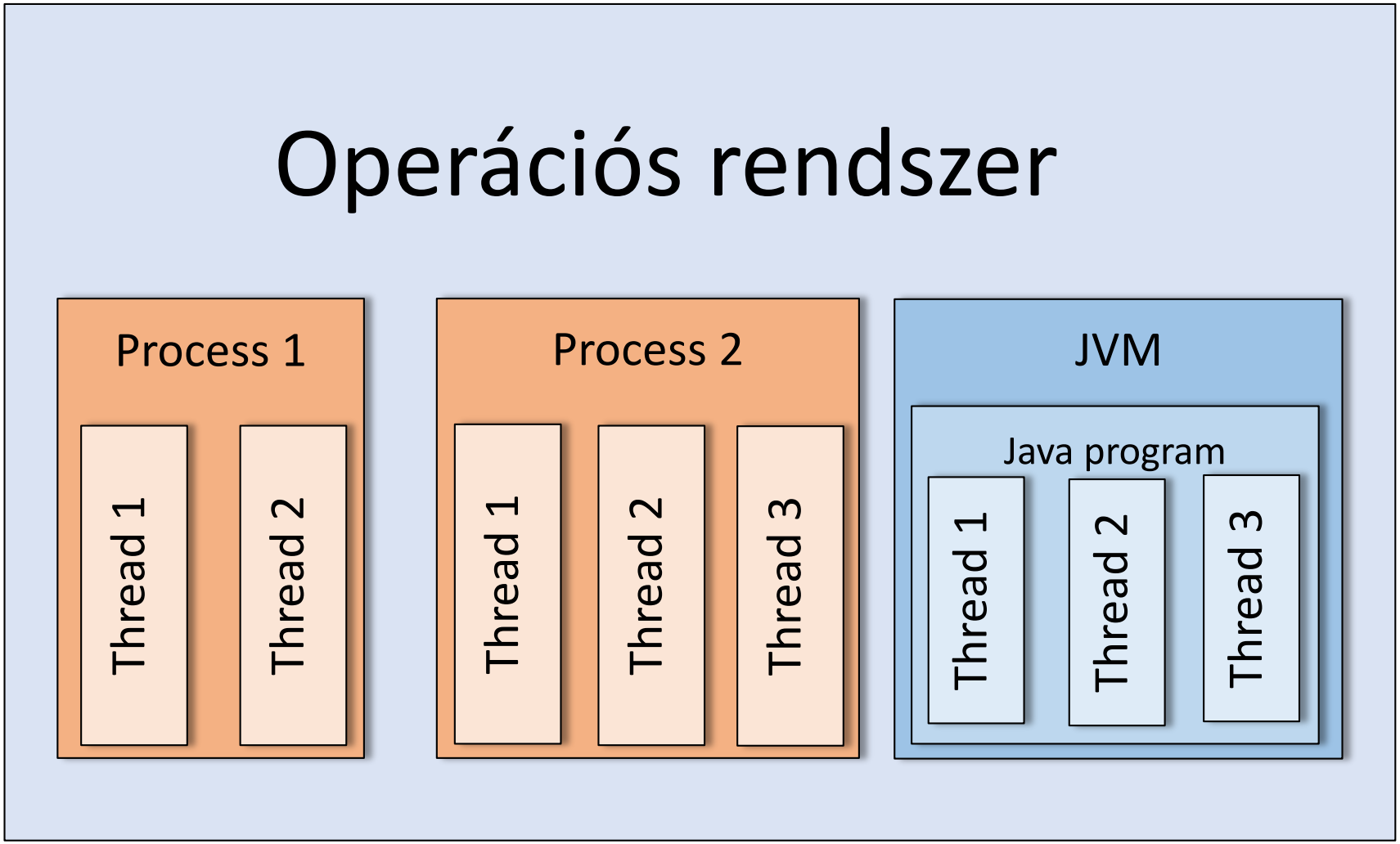
Thread - Szál

- Egy processzen belül is szükség lehet több munkafolyamat párhuzamos futtatására.
- Erre az operációs rendszer szál (thread) formájában nyújt beépített támogatást.
- A szálak a processzhez nagyon hasonló módon futnak, de minden szál egy-egy folyamathoz tartozik, így azok a szálak, amelyek egy folyamathoz tartoznak, mind közösen használhatják a processz által lefoglalt memóriaterületet és erőforrásokat.

Többszálú programozás Java-ban

- A Java virtuális gépen futó alkalmazásunk is egy ”virtuális” processz, majdnem egyenértékű egy natív (ténylegesen a processzoron futó) folyamattal (kommunikálhat más processzekkel, használhat erőforrásokat), hiszen a virtuális gép natív processzként fut.
- A Java processz is szálakat tartalmaz, amelyek párhuzamos futtatását a JVM vagy az operációs rendszer ütemezi.
- A JVM elvárt működése szabványban van leírva, ez nem követeli meg, hogy a különböző JVM implementációk hogyan ütemezzenek, vagy ütemezőt tartalmazzanak-e egyáltalán, ezért semmilyen feltételezésünk nem lehet a szálak végrehajtási sorrendjéről, ahogy arról sem, hogy egy szál mennyi időre kap futási jogot.
- Többnyire jó közelítés, hogy egyszerre futnak.

Java többszálúság



Többszálú program

- Bizonyos problémákat lehetetlen, vagy nem hatékony egy szálon megoldani.
- Például egy lassabb periféria (pl. nyomtató) válaszára várakozunk, egyetlen szál esetén a válaszra tétlenül várakozni időpazarlás (hiszen közben nem tudunk mással foglalkozni), ekkor kevésbé hatékony megoldásokra gondolhatunk először:
  - Polling: a programunk bizonyos időközönként megszakítja a feladatvégzését, majd ellenőrzi a kívánt állapot beálltát (pl. a nyomtató ismét elérhető)
  - Busy waiting: a pollinghoz hasonló, azonban nincs egyéb feladatvégzés. Ha a tétlen várakozást nem tudjuk megvalósítani, a programnak egyfolytában le kell kérdeznie az állapotot, amíg az a kívánt állapotba nem kerül, ezáltal a processzort folyamatosan a lekérdezéssel terheljük
- Blocking eljárások (amelyek hosszabb ideig dolgoznak, mielőtt visszatérnének az eredménnyel) zökkenőmentes futtatása lehetetlen szálak nélkül (később láthatunk rá példát hálózatkezelésnél).

Szálkezelés eddig

- Eddigi programjainkban is volt szálkezelés tudtunkon kívül.
- Egyetlen szál futott, a `main()` függvény tartalmával.
- Az általunk írt kód tehát eddig ezen az egy szálon futott.

Újabb szálak létrehozása

- Javában újabb szálakat speciális osztályok segítségével indíthatunk.
- Ahhoz, hogy egy új szálban indítható osztályt kapjunk, az osztálynak meg kell valósítania a `Runnable` interface-t, amely interface összesen egy `run()` metódussal rendelkezik. Az osztályunkban megvalósított `run()` metódus fogja tartalmazni az általunk indított szál feladatait.
- A szál elindításához példányosítanunk kell az interface-t megvalósító osztályt, majd azt egy `Thread` objektum konstruktorának paraméterül átadjuk, végül az új `Thread` példány `start()` (és nem a `run()`) metódusát hívjuk meg. Ekkor a vezérlés kettéágazik, a `main()` függvény is fut tovább, és vele párhuzamosan egy új szálban a `run()` tartalma is elkezd végrehajtódni.
- Egy szál futása befejeződik, ha a `run()` metódus véget ér.

Újabb szálak létrehozása (folyt.)

- Mivel a `Thread` osztály is megvalósítja a `Runnable` interface-t, így új szálban indítható osztályt úgy is létrehozhatunk, hogy a `Thread` osztályból öröklünk, azonban a `run()` metódust felül kell definiálnunk, mivel az alapértelmezetten a neki átadott `Runnable` példány `run()` metódusát hívja meg (ha nem adtunk át ilyet, akkor nem csinál semmit).
- *Miért jó, hogy van külön `Thread` osztály és `Runnable` interfész?*
- Habár nehézkesebbnek tűnik a `Runnable` interface használata, mégis ez az ajánlott, mivel így bármiből öröklődhet a szálat elindító osztály (`Thread` használatakor az egyszeres öröklődés miatt semmi másból nem öröklődhet).

- Vizsgáljuk meg a kiadott kódban (BasicThread) a `Runnable` használatát, majd írjunk egy új, `Thread`-ből öröklő osztályt!

A szálak nem objektumok

- Fontos, hogy a szálak nem azonosak az őket elindító objektummal.
- Az osztály csak a végrehajtandó kódot, az objektum pedig a változóit adja a szálhoz.
- Mivel ez a kód meghívhatja más objektumok metódusait, így egy szál bőven több, mint a `run()` metódusban található kód, előfordulhat, hogy egy objektum metódusát több szálból is meghívjuk.
- Illetve az objektum változóit is módosíthatjuk más szálakból, függvényhívásokkal vagy ha láthatóak az adott szálat futtató objektum számára.

A szálak vezérlése

- A szál és az azt létrehozó objektum közötti különbségtétel azért fontos, mert emiatt a szálak futását nem az objektumokon keresztül befolyásolhatjuk, hanem a `Thread` osztály statikus metódusaival.
  - Van néhány kivétel, de ezekre később visszatérünk.
  - Emiatt egy adott szálat vezérlő utasítást mindig az a szál kap meg, aki az utasítást végrehajtó kódot futtatja (tehát a szál ráfut egy szálvezérlő utasításra).
  - A vezérlőutasítások az ütemezésre hatnak
    - Például várakoztatnak, altatnak
  - Pl.: `Thread.sleep(1000)` utasítás az aktuális szálat altatja egy másodpercig.
- Nézzünk meg a `BadExampleForWaiting` csomagot!
  - Láthatunk benne egy kerülendő példát, aminek folyamán valamilyen erőforrásra várunk.
  - Mi vele a hiba?
  - Amikor várunk arra, hogy adat jöjjön, folyamatosan kérdezgetjük, hogy még mindig üres-e a sorunk.
  - Ez felesleges és megterheli a processzort, elveszi az időt a többi hasznos futó folyamat elől.
  - Ez a korábban említett busy waiting.

A szálak vezérlése

- Nézzünk meg a `BetterExampleForWaiting` csomagot!
  - Itt egy polling-ot valósítunk meg
- Ha megnézzük, hogy mennyire használják a programok a processzort, akkor szépen láthatjuk a különbséget.
- De ez se igazán szép megoldás, a Java szálkezelése ennél sokkal jobb módszereket is lehetővé tesz:
  - Képesek vagyunk egy szálat akkor felébreszteni, ha valamilyen esemény történik. Ezeket később fogjuk tanulni.

Néhány vezérlő utasítás

- `start()`: elindítja a szál `run()` metódusát.
- ~~`stop()`~~: már nem javasolt a használata (csak a kompatibilitás miatt létezik még mindig, kvázi tilos használni)
- `sleep(long millis)`: ezredmásodpercben meghatározott ideig vár.
- `sleep(long millis, int nanos)`: milliszekundum plusz nanoszekundum ideig vár. (Elméletben…)
- Nézzük meg a `SleepSample` csomagot!

A szálak második anyagrészében

- `interrupt()`, `join()`, `yield()`, `wait()`, `notify()`, `notifyAll()`

Versengő szálak

- Előfordulhat, hogy két szál ugyanazt az objektumot használja egyszerre (vagy azon keresztül azonos erőforrást).
- Az esetek többségében ez nem okoz problémát, de ha egy összetett adatot módosít a két szál, akkor könnyen inkonzisztens állapotba juthat az objektum.
- Például egy komplex számot tároló objektumot az egyik szál írja, a másik szál olvassa.

Komplex osztály (nem szálbiztos)

```
public class Complex {

  private double re;
  private double im;

  public void set(double re, double im) {
        this.re = re;
        this.im = im;
  }

  public double[] get() {
        double[] result = {re, im};
        return result;
  }

}
```

Futtatás

- Írjuk meg a kódot, próbáljuk ki, mi történik (ComplexExample)!
- Az egyik szálból hívva a valós, a másikból hívva a képzetes érték 0 legyen!
- Azokra a kiírt értékekre kell figyelni, amelyekben mind a valós, mind a képzetes rész nem 0, ugyanis ilyen értékeket elvileg nem írunk bele…
- Ha ilyet látunk, írjunk ki hibaüzenetet!
- Mit látunk?

Komplex osztály

- Hogyan okozhat hibát az osztály konkurens használata?
  - Megtörténhet, hogy az egyik szál meghívja a `set()` függvényt, ami sikeresen felülírja a re adattagot.
  - De pont akkor történik a szálcsere, amikor a valós rész már átíródott, a képzetes még nem.
  - Ekkor rossz értéket tartalmaz az objektum (se nem a korábbi, se nem a későbbi szám értékét), ha meghívjuk a `get()` függvényt, akkor ezt a hibás értéket kapjuk meg.
- Ehhez hasonló hiba történhet akkor is, ha az olvasás szakad félbe egy írás kedvéért, vagy ha az egyik szál akkor írja felül a komplex számot, amikor a másik még csak a felét módosította.

Java szinkronizáció

- A versengő szálak okozta hibák elkerülésére a Java a monitor módszert alkalmazza.
  - Ez annyit jelent, hogy egy szál bizonyos erőforrások (kódrészlet, objektum) használati jogát magához ragadhatja, megakadályozva ezzel, hogy más szálak az adott erőforrást párhuzamosan használhassák.
    - Levédhetjük metódusainkat úgy, hogy csak egy szál tudja őket meghívni egyszerre (ekkor bármely védett metódus hívása blokkolja az összes védett metódus használatát az adott példányon).
    - Levédhetünk egy objektumot egy adott kódszakaszra.
  - Javában minden objektum monitor, de a monitor birtoklása nem feltétlenül jelenti az objektumhoz való hozzáférés kizárólagos jogát.
    - Hiszen monitor megszerzése nélkül is hozzáférhetünk az objektumhoz (mint eddig is).
    - A monitor olyan mint egy jelző lámpa: ha figyelembe veszed, akkor biztos nem mész át az úton egy autóval egyszerre, de ha nem figyeled akkor át tudsz menni bármikor.

Java szinkronizáció

A monitor megszerzése történhet védett metódusokon keresztül vagy szinkronizált blokk segítségével.

- Egy blokk a **`synchronized`** kulcsszóval tehető szinkronizálttá
- Mindig egy objektumra (és nem változóra) szinkronizálunk
  - Ekkor próbáljuk megszerezni a monitort
- Ha valaki birtokolja a monitort, akkor a szál egy várakozási sorba kerül
- Amikor felszabadul a monitor, a sorban valamelyik szál megkapja
- A szinkronizált blokk belsejében csak a monitor birtokában tartózkodhatunk
- A védett metódusok funkcionalitásban nem különböznek egy, a példányra (vagy statikus metódus esetében az osztályra) szinkronizált blokktól.

Szálbiztos Komplex osztály

```
public class ComplexThreadSafe {

  private double re;
  private double im;

  public void set(double re, double im)  {
    // A metódus magához ragadja a példányváltozó monitorját
    synchronized (this) {
      this.re = re;
      this.im = im;
    } // Itt adja vissza a monitort
  }

  public double[] get() {
    // A metódus magához ragadja a példányváltozó monitorját
    synchronized (this) {
      double[] result = {re, im};
      return result;
    }  // Itt adja vissza a monitort
  }

}
```

Szálbiztos Komplex osztály

```
public class ComplexThreadSafe {

  private double re;
  private double im;

  public synchronized void set(double re, double im) {
    this.re = re;
    this.im = im;
  }

  public synchronized double[] get() {
    double[] result = { re, im };
    return result;
  }
}
```

- Próbáljuk ki, mi történik ha lecseréljük a Complex osztályt a ComplexThreadSafe-re!

Atomi műveletek

- Azt gondolnánk, hogy egyetlen primitív típusú változó esetén ez a probléma nem fordulhat elő.
  - Sajnos egyetlen változó módosítása is több elemi lépésből állhat, ha nem csak kiolvasásról vagy értékadásról van szó.
  - A legegyszerűbb ilyen eset egy változó inkrementálása, mely a változó lekéréséből, növeléséből és visszaírásából áll. Látható, hogy még ilyen eleminek tűnő művelet is elromolhat, ha pont ezalatt történik szálváltás.
  - Például: `++i`
    - Itt ugye először az `i` értékét be kell olvasni, hozzáadni egyet, majd visszaírni az eredményt.
    - Ha két szál úgy van ütemezve, hogy először mindkettő beolvassa az `i` (régi) értékét, majd mindkettő megnöveli eggyel és visszaírja, akkor az `i` új értéke végső soron csak eggyel lett megnövelve.
    - Emiatt az egész műveletet védeni kell
- A `java.util.concurrent.atomic` csomagban található néhány beépített osztály, amelyek műveleteit szálbiztosra írták meg:
  - Ezek teljes egészében végrehajtódnak, mielőtt az objektumon más művelet végrehajtódna: atomi műveletek
  - `AtomicInteger`, `AtomicBoolean`, `AtomicReference`

Volatile

- Általános esetben a szálak cache-elhetik (kimásolják, és a másolattal dolgoznak egy ideig, és csak később frissítik a főmemóriát) a változó értékét, és nem írják rögtön a főmemóriába, ha az megváltozott. Ez inkonzisztenciához vezethet, amikor az átírás megtörténik, illetve más szálak esetleg nem értesülnek a változásról.
  - A **`volatile`** kulcsszóval ellátott változók értékeinek módosulásakor azonnal a főmemóriába íródik az érték, ezzel elkerüljük a bajt.
  - Ez csak írásnál vagy olvasásnál megoldás, frissítésnél (amikor írunk és olvasunk is) ugyanúgy versengés van
- Ha csak egy-egy változót akarunk védeni, és azt nem kell frissítenünk, akkor érdemes a **`volatile`**-t használni. Ám abban az esetben, ha sok változóról van szó, vagy frissíteni kell, akkor a **`synchronized`** lehet a jobb választás.
  - A **`volatile`** mindig a főmemóriához nyúl, ami lassú lehet a **`synchronized`**-hoz képest (itt a lock előtt és után frissül a memória)
- Nézzük meg a `volatile` csomag tartalmát! További olvasnivalók:
  - https://dzone.com/articles/java-volatile-keyword-0
  - https://vanillajava.blogspot.hu/2012/01/demonstrating-when-volatile-is-required.html

Synchronized versus volatile

Technika	Előny	Hátrány
synchronized	Csak a lock előtt és után frissül a főmemória Nincs konkurencia	Nincs konkurencia
volatile	Megengedi a konkurenciát	Minden memóriaelérésnél frissül a főmemória Megengedi a konkurenciát

Még néhány példa szinkronizációra

- Nézzük meg a SyncSample csomagot! (2 program)
- Nézzük meg a PortEmulatort!
- Nézzük meg a szálak ütemezését úgy, hogy egy változót két szálról egyszerre módosítunk (egyikről növeljük, másikról csökkentjük) (Counter).

Deadlock, Livelock

- *Deadlock*: ez a jelenség akkor alakul ki, amikor két szál az idők végezetéig egymásra vár.
  - Ketten állnak a széles folyosón, mindkettő arra vár, hogy a másik oldalra lépjen. Nem történik semmi.
  - Adott két szál (1,2). Az egyik lefoglalta A-t, a másik B-t. Mindkettő le akarja foglalni mindkét erőforrást. Addig nem fogják a sajátjukat elereszteni, amíg meg nem szerezték a másikat.
- *Livelock*: hasonlít a deadlockhoz, azzal a különbséggel, hogy míg ott a szálak várnak egymásra, itt azt a helyzetet kell elképzelni, hogy ha az egyik szál csinál valamit, arra reagál a másik szál, erre reagál az első. Előfordulhat, hogy az örökkévalóságig reagálnak egymásra, így ezen szálak nem tudnak mást csinálni.
  - Két járókelő áll egymással szemben a széles utcán. Mindketten kilépnek jobbra. Aztán balra. Aztán jobbra. Aztán balra...
  - Két szál (I,II) le akarja foglalni A-t és B-t. I lefoglalta A-t, II pedig B-t.
    I le akarja foglalni B-t, de nem tudja, ezért elengedi A-t. II ugyanígy tesz, csak B-vel. Ezután mindkettő kezdi elölről a procedúrát.

Starving

- Akkor alakul ki, ha egy szál nem tud hozzáférni egy megosztott erőforráshoz, mert azt más szálak uralják.
  - Valaki áll egy ajtóban. Ki akar menni a folyosóra, de nem tud, mert ott folyamatosan rohannak az emberek.
  - Egy szál le akarja foglalni a nyomtatót. A nyomtatót éppen valaki más lefoglalta és még nem szabadította fel.
  - Ha alacsony a szál prioritása (erről bővebben később…), akkor gyakran előfordulhat:
    - A diákok sorban állnak a menzán, de folyamatosan jönnek közben a tanárok is, akik mindig beállnak a sor elejére, így a diákok nem jutnak előbbre.

Gyakorló feladat G06F01

- Készíts egy konkurens sort, ahol *általában* egyszerre lehet kivenni és berakni elemeket.
  - Először készíts egy sort, ami két verem (in, out) segítségével van implementálva.
  - Majd oldd meg a szinkronizációt:
    - Két lock-ot használj: in, out.
      - Elem behelyezésekor az in-re szinkronizálj.
      - Elem kivételekor:
        - Ha az out nem üres, akkor az out-ra.
        - Ha az out üres, akkor szükséges az in-re és az out-ra is szinkronizálni.
  - Ha készen vagy a konkurens adatszerkezettel, teszteld két szál segítségével:
    - Az egyik növekvő sorrendbe egy adott intervallumból egész számokat rakjon a sorba.
    - A másik vegye ki a számokat és írja egy fájlba, ha a kapott fájlban helyes a sorrend nagy valószínűséggel jól működik az elkészített adatszerkezet. (A szálnak akkor van vége, ha az író szál egy osztott változón keresztül ezt jelzi.)

Gyakorló feladat G06F02

- Készíts egy útkereszteződést szimuláló programot!
- Legyen több Car szálad, ami egy rendszámmal (100.000 és 999.999 közötti véletlen sorszámmal) azonosított autót reprezentál, ami szeretne átmenni a kereszteződésen.
  - Az áthaladás úgy történik, hogy:
    - Az autó megy előre 3 mp-et a kereszteződésig.
    - Ha nincs a kereszteződésben másik autó, akkor áthalad rajta (2 mp alatt)
    - Ha van autó a kereszteződésben, akkor pontosan addig vár, amíg az a másik ki nem ér belőle.
    - Utána megy tovább 3 mp-et, ami után az autó elköszön.
  - Áthaladás közben mindig logoljon a konzolra:
    - Pl. „A(z) 173974 rendszámú autó közeledik a kereszteződés felé”
    - Pl. „A(z) 372800 rendszámú autó megérkezett a kereszteződés elé”
    - Pl. „A(z) 507218 rendszámú autó elhagyta a kereszteződést”
- Legyen egy Intersection (útkereszteződés) objektumod, ennek segítségével oldják meg azt az autók, hogy mindig csak egy tartózkodik a kereszteződésben, akkor is, ha egyszerre ér oda kettő autó.
- Ne használj sleep-et vagy busy waitinget, amíg várakozik az autó a kereszteződésnél, induljon el azonnal, ha a másik kihaladt onnan.
- Készíts példaprogramot, ami megmutatja, hogy jól működik a kereszteződésed akkor is, amikor sok autó érkezik egyszerre a kereszteződéshez, de akkor is, amikor kis különbséggel (fél mp) érkeznek oda.

Gyakorló feladat G06F03

- A feladat egy horgászat szimulálása:
  - Öt barát elindul horgászni
  - Mivel kora reggel indulnak, mindenki véletlenszerűen otthon felejti a horgászengedélyét
  - A céljuk, hogy 100 kg halat fogjanak összesen
  - Csak egy doboz csalit visznek magukkal, egyszerre csak egy ember tud csalit kivenni a dobozból, és a csalizás 2 másodpercig tart
  - Csak akkor kell csalizni, amikor elkezdenek horgászni és amikor megfogták a halat
  - Jó a halak étvágya, folyamatosak a kapások, ezért a horgászok a bedobás után 3-5 másodperccel megpróbálják megakasztani a halat
  - A horgászoknak 80% esélyük van a hal megfogására, a kifogott halak 1 és 3 kg közötti tömegűek véletlenszerűen
  - A kifogott halat visszaengedik, tehát a szimuláció szempontjából csak a tömegük számít, ezt közösen számolják
  - A horgászat megkezdése után 20 másodperccel megjelenik a halőr, minden horgásztól egyenként elkéri az engedélyét, és megszámolja, hogy összesen mennyi büntetést kell fizetniük (10000 forint a büntetés emberenként)
  - A szimuláció minden lépését írasd ki a konzolra (pl. Józsi elkezdett csalizni, Béla egy 2 kilós halat fogott, …). A végén a halak tömegét és a büntetést is írasd ki!

Gyakorló feladat G06F04

- Torta sütés
- Olga, Mása és Irina eperkrémes csoki tortát sütnek Andrej születésnapjára. Olga készíti a piskóta alapot, Mása a tejszínes eperkrémet, Irina pedig a csokoládé bevonatot.
- A sütés előtt elmennek bevásárolni, de egy (véletlenszerűen választott) hozzávalót sajnos elfelejtenek megvenni, így Irinának vissza kell mennie érte a boltba, ami 30 percig tart.
- A sütés során minden hozzávalóból csak egy dobozzal áll rendelkezésre, így egyszerre csak egyik lány kezében lehet. Minden hozzávalót 1-3 percre foglalnak le a kiméréshez (ez is sorsolással dőljön el).
- A piskótához szükség van tojásra, lisztre, cukorra, sütőporra, valamint vajra. A krémhez tejszín eper, cukor, és zselatin kell. Míg a bevonat kakaóból, cukorból, vízből és vajból készül.
- A konyhában csak egy mixer van, ezen is meg kell osztozniuk. Mixer kell a piskóta és a krém készítéséhez. A mixert minden esetben 3-5 percig használják. Az egyszerűség kedvéért a mixert csak akkor foglalhatja le mindenki, ha már kimérte a hozzávalókat.

Gyakorló feladat G06F04

- Tehát a piskóta elkészítéséhez Olga először kiméri a hozzávalókat, majd a végén a mixerrel összekeveri őket. Ezután 30 percig tart a sütés.
- A krémet Mása hasonló módon készíti, a megfelelő hozzávalók kiadagolása után használja a mixert, ezzel a krém elkészült.
- Végül Irina a bevonatot egy főző edényben készíti, így neki nem kell várnia a mixerre. A főzés 20 percet vesz igénybe.
- A végén együtt rakják össze a tortát, amint az alap, a krém, és a bevonat is elkészült.
- A program minden lépést írjon ki a konzolra. Pl.:
  - A bevásárlásnál kimaradt a cukor, Irina elment cukorért.
  - Olga kimérte a cukrot.
  - A piskóta alap elkészült.
  - Stb.
  - A torta kész van.

Gyakorló feladat G06F05

- Egy iskolában programozás géptermi ZH-t írnak
  - Ez ún. ZH üzemmódban történik, mindenki egy szerverről tölti le a feladatokat, és oda tölti fel a megoldását
    - A szerver 100 kérést (kapcsolatot) tud egyszerre kipróbálni
    - Ha egy kapcsolat úgy jön létre, hogy már legalább (rajta kívűl) 100 kapcsolat létrejött, akkor a kapcsolat ideje kétszer olyan hosszú lesz
  - Négy labort használnak fel a lebonyolításra
    - Minden laborhoz tartozik egy-egy switch, ezen keresztül tudnak a szerverrel kommunikálni
    - Egy teremben 30-an írnak
    - Minden switch egyszerre maximum 15 kapcsolatot tud fenntartani normál sebesség mellett
      - Minden olyan kapcsolat, amelyik úgy indult, hogy már volt legalább 15 létrejött kapcsolat a switchen keresztül, létrejön ugyan, de kétszer olyan lassú lesz
      - Minden olyan kapcsolat, amelyik úgy indul, hogy van legalább 25, már létrejött kapcsolat, nem jön létre
        - 3 másodperc után újra próbálkozik
      - A kapcsolatok idejének szorzói összeszorzódnak (tehát, ha pl. mind a 120-an próbálnak kapcsolatot létesíteni, az utolsó 20-nak négyszer annyi ideig fog tartani a kérés)

Gyakorló feladat G06F05

- A ZH menete a következő
  - Az elején mindenki megkapja a feladatlapot (papíron is), és elkezdni az tanulmányozni
    - Kb. a kétharmada ezelőtt letölti a feladatokat és a környezetet a szerverről is, a maradék csak utána
  - A felügyelő tanár harsányan közli mindenkivel, hogy aki közvetlenül a szerverre mer dolgozni, azt utólag buktatja még számítástechnikából
    - Tehát egy-két ember kivételével mindenki a saját gépén dolgozik, és csak időnként ment a szerverre
  - Mielőtt beadná, mindenki még egyszer feltölt mindent a szerverre
    - Van, aki korábban, félidő környékén beadja (mert mindennel gyorsan készen van)
- Írd meg a ZH szimulációját!
  - Minden gép/diák egy szál legyen
  - Az egész ZH 120 másodpercig tart
    - Pontosabban eddig kezdeményezhetnek fel- és letöltést a diákok
    - A feladatlap elolvasása az elején kb. 10 másodpercig tart
    - A környezet letöltése kb. 5 másodpercig tart terheletlen hálózaton
    - Egy mentés kb. 5 másodpercig tart terheletlen hálózaton
    - A legtöbben kb. 20 másodpercenként mentenek
      - Akik a szerveren dolgoznak, ők kb. 5 másodpercenként
  - Minden kapcsolat kezdetét és végét írd ki konzolra (időbélyeggel)!
  - Ne alakuljon ki versengés, semmilyen ütemezés esetén!
  - A házi feladat határideje: 2016. 03. 24. 8:00