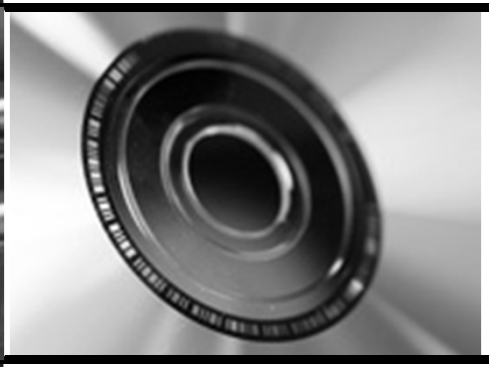
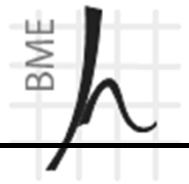




A JAVA VIRTUÁLIS GÉP

Garbage Collection, JIT Compiler, Többszálúság

Varga Endre Sándor
doktorjelölt
BME Híradástechnikai Tanszék
vendre@hit.bme.hu

2011. november 8.,
Budapest



Mi a JVM?

- A legnépszerűbb és legrugalmasabb virtuális gép
- Alapból a Java nyelvhez készült
- Fontos elemei
  - Garbage Collector
  - JIT Compiler
- Ezekről lesz szó

Garbage Collection alapok, JVM heap struktúra, serial, parallel és concurrent Garbage Collection

GARBAGE COLLECTION

Bevezetés

- A **szemétgyűjtő (garbage collector)** a Java VM memóriakezelésének kulcsfontosságú eleme.
- A memóriakezeléssel kapcsolatos feladatok zömét leveszi a felhasználó válláról, megkönnyítve a biztonságos, memóriaszivárgástól mentes programok készítését.
- A szemétgyűjtő a felhasználói kódtól függetlenül, jobbára észrevétlenül működik.
- Elvileg nem kell, hogy egy Java programozó tisztában legyen működésének részleteivel, azonban ezen ismeretek révén “szemétgyűjtőbarát” programozási stílust alakíthatunk ki.

A memória szerkezete I.

- Egy felhasználói program szemszögéből (vezérlésáramlásos paradigmát feltételezve) a memória szerkezetét általában három részből állónak tekinthetjük:
  - **programkód**,
  - **stack** (verem),
  - **heap** (kupac).
- A programkód mérete jellemzően változatlan, a mi szempontunkból mindenesetre érdektelen.
- A stack és a heap alkotják a program által adatok tárolására használatos memóriatartományt.
- A fenti tartományok architekturális okokból történő további felosztásával (szegmentálás, virtuális tárkezelés) egyelőre nem foglalkozunk.

A memória szerkezete II.

- A stack és a heap szerepe és működése nagyban különbözik.
- A stack feladata a program szegmentált szerkezetéhez (eljárások, függvények, stb.), illetve a szegmensek közötti hívások lebonyolításához szükséges adatok tárolása.
  - (visszatérési címek),
  - paraméterek,
  - lokális változók.
- A heap ezzel szemben a program szerkezetéhez nem kapcsolódó adatok tárolására szolgál.
  - dinamikus adatszerkezetek.

A memória szerkezete III.

- A stack, nevének megfelelően, verem jellegű, tehát a push és a pop műveleteket használjuk rajta.
  - A push egy elemet helyez a stack tetejére, a pop leveszi a legfelső elemet.
  - A stack lényegében egy LIFO adatszerkezet.
- A stack verem szervezését az indokolja, hogy amikor egy szegmens egy másikat hív meg, a hívással kapcsolatos adatokat a hívás idejére fogadja, majd a hívás végeztével azok kényelmesen “eltakaríthatók”.
- Ez a struktúra egyszerűvé teszi hívások “láncainak”, illetve a rekurziónak a kezelését is.

A memória szerkezete IV.

- Egy függvényhívás jellemzően az alábbi lépésekből áll:
  - Visszatérési cím a stackre.
  - Paraméterek értékei a stackre.
  - Ugrás a függvény kódjára.
  - Lokális változók létrehozása a stacken.
  - A függvény kódjának futása.
  - A paraméterek és lokális változók levétele a stackről.
  - Ugrás a visszatérési címre.
  - A visszatérési érték és a visszatérési cím levétele a stackről.
- A lokális változók kialakítása, illetve a levétel művelet egyszerűen a stack tetejét jelző, ún. stack pointer átállításával történik.

A memória szerkezete V.

- Az stack egy híváshoz tartozó elemeit **stack frame**-nek hívjuk.
- A fentiekből következik, hogy a szegmensek paraméterei, illetve a lokális változók által elfoglalt memória kezelése “automatikus”, a stack működéséből következik, és csak annak helyes megvalósításán múlik.
- A heap működése lényegében más, mérete vagy fix, vagy nagyobb lépésekben változik, az adatok elhelyezése valamilyen allokációs stratégia mentén történik.
- Abban az esetben, amikor a stack és a heap egyazon véges méretű memóriaterületen kap helyet (mikroszámítógépek, szegmentált memóriakezelés, stb.), a szokásos megoldás az, hogy a stack a terület egyik, a heap a másik végéről indulva növekszik, a memória pedig akkor fogy el, amikor összeérnek. A modern operációs rendszerekben már általában nem ez a helyzet, a stack és a heap egymástól független.

A heap I.

- A heap használata az alábbi három lépést jelenti:
  - memóriafoglalás (allokáció),
  - a foglalt terület használata,
  - memóriafelszabadítás (deallokáció).
- Az allokáció során kell találni egy, a kért méretű, vagy annál nagyobb üres területet, majd fel kell jegyezni, hogy az adott terület foglalt.
  - Erre számos módszer van, first-fit, best-fit algoritmusok, az üres területek nyilvántartása méret szerint, lookaside listák, melyekkel fix egységekben allokálható a terület, stb.

A heap II.

- A felszabadítás elvileg egyszerű, mert csak a terület foglaltságát kell törölni a listából, azonban az egymást követő foglalások és felszabadítások a heap fragmentációját okozzák, így a szabad memória mennyiségénél jóval kisebb egybefüggő területet sem lehet már foglalni.
- A fragmentáció elkerülésére egyrészt megfelelő allokációs algoritmust kell használni, másrészt szükség lehet a heap időszakos karbantartására is.
- Ha egy allokáció ezután sem teljesíthető, a heap területét meg kell növelni, vagy jelezni kell a hibát (elfogyott a memória).

A heap III.

- Az olyan, széles körben elterjedt és használt nyelvekben, mint a C és a C++, a memória allokáció és deallokáció a programozó kezében van.
- Ennek előnye, hogy explicite rendelkezhetünk a szükségtelenné váló memóriaterületek felszabadításáról, ennek elmulasztása viszont elkerülhetetlenül memóriaszivárgáshoz vezet.

```
void fuggveny () {
   char *szoveg = (char *) malloc (200);
   // ... használom
   free (szoveg);
}
```

- Ha az utolsó sor (a `free`) hiányzik, a lefoglalt 200 byte soha nem szabadul fel, noha a pointer lokális változó, és mint ilyen, a függvényből való visszatéréskor “megszűnik”.

Objektumok elérhetősége

- A továbbiakban visszatérünk az objektum-orientált szemlélethez, és nem lefoglalt memóriaterületekről, hanem objektumokról beszélünk.
- **A szemétgyűjtés elve az, hogy a deallokációt automatikusan végezzük el, amikor az adott objektumra már nincs szükség.**
- A szemétgyűjtés két alapvető kérdése, hogy
  - mikor mondhatjuk, hogy nincs szükség egy adott objektumra, illetve
  - hogyan találjuk meg a szükségtelenné vált objektumokat.
- Az első kérdésre a rövid válasz az, hogy azok az objektumok szükségtelenek, amelyeket már nem lehet referenciákon keresztül elérni.

Referenciaszámláló szemétgyűjtő I.

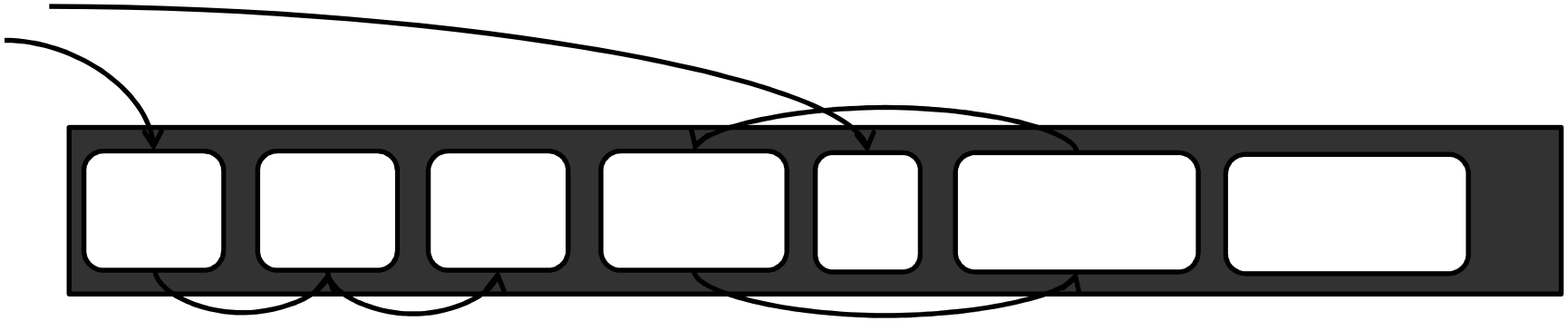
- A hosszabb válasz az, hogy ennek megállapítására több stratégia létezik.
- Ezek közül a legegyeszerűbb a **referenciaszámláló szemétgyűjtő** (reference counting collector) működési elve.
- A szemétgyűjtő nyilvántartja az egyes objektumokra mutató referenciák számát. Amennyiben ez a szám eléri a nullát, az objektum szükségtelennek minősül, és felszámolható.
- A megvalósítás azon alapul, hogy a fordító instrumentálja a kódot, és minden referencia értékadáskor növeli, illetve csökkenti a megfelelő objektumra mutató referenciák számát.

Referenciaszámláló szemétgyűjtő II.

- A módszer előnye, hogy egyszerű, és bizonyos körülmények mellett utólag is megvalósítható hasonló viselkedés.
  - A C++ library `string` osztálya úgy viselkedik, mintha egy referenciaszámláló szemétgyűjtő kezelné, ezt az értékadás operátor felüldefiniálásával és a destruktorok felhasználásával éri el.
- Több hátránya is van:
  - Együttműködést igényel a fordítótól.
  - Az értékadások állandó figyelése jelentősen lassítja a kód futását.
  - Kívülről elérhetetlen, ciklikus struktúrákat nem tud felszámolni.

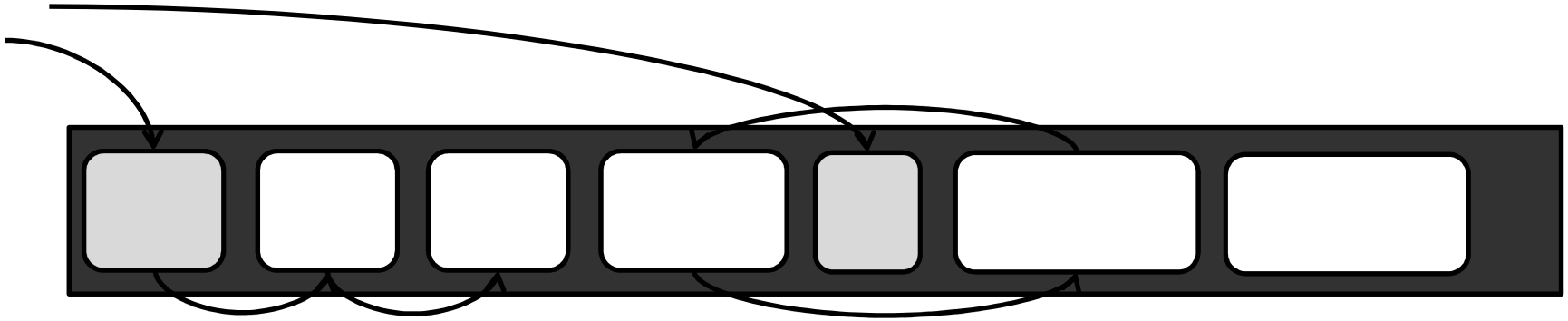
Alapok – GC

- A Garbage Collection lényege az „elérhetetlen” objektumok által foglalt memória felszabadítása
- GC Root: stabil referenciák, az elérhető objektumok halmazát ezekből érjük el



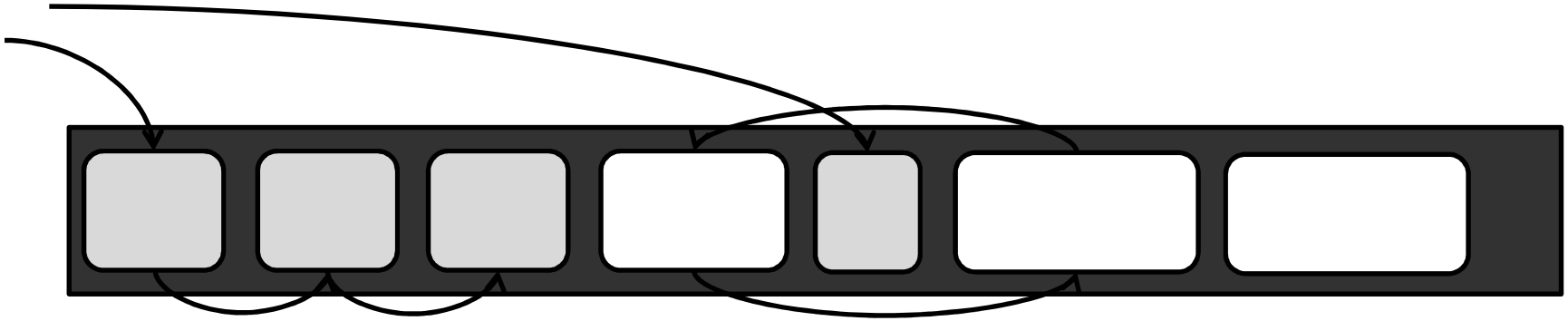
Alapok – GC

- Első körben fel kell deríteni a GC root-ok által mutatott objektumokat:



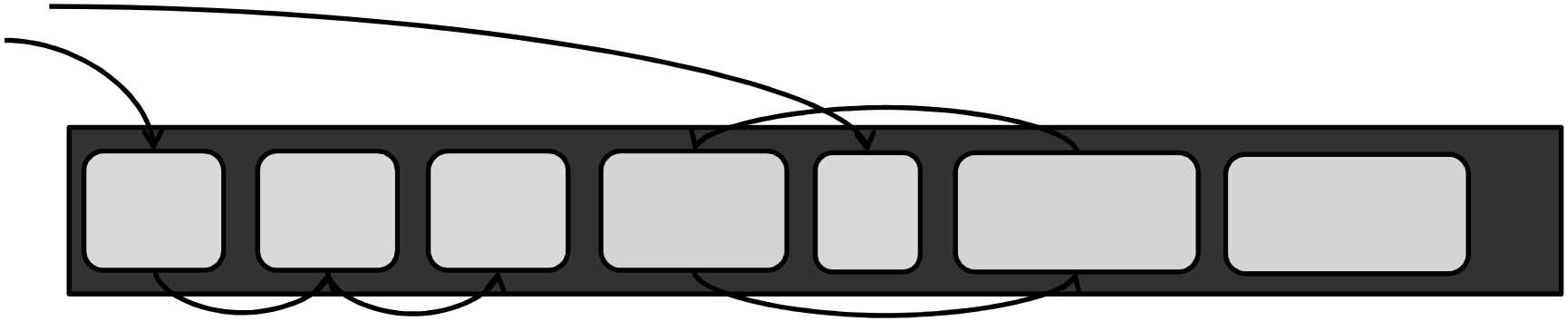
Alapok – GC

- Második lépésben felderíteni a GC root-ok által közvetlenül mutatott objektumok által hivatkozott többi objektumot



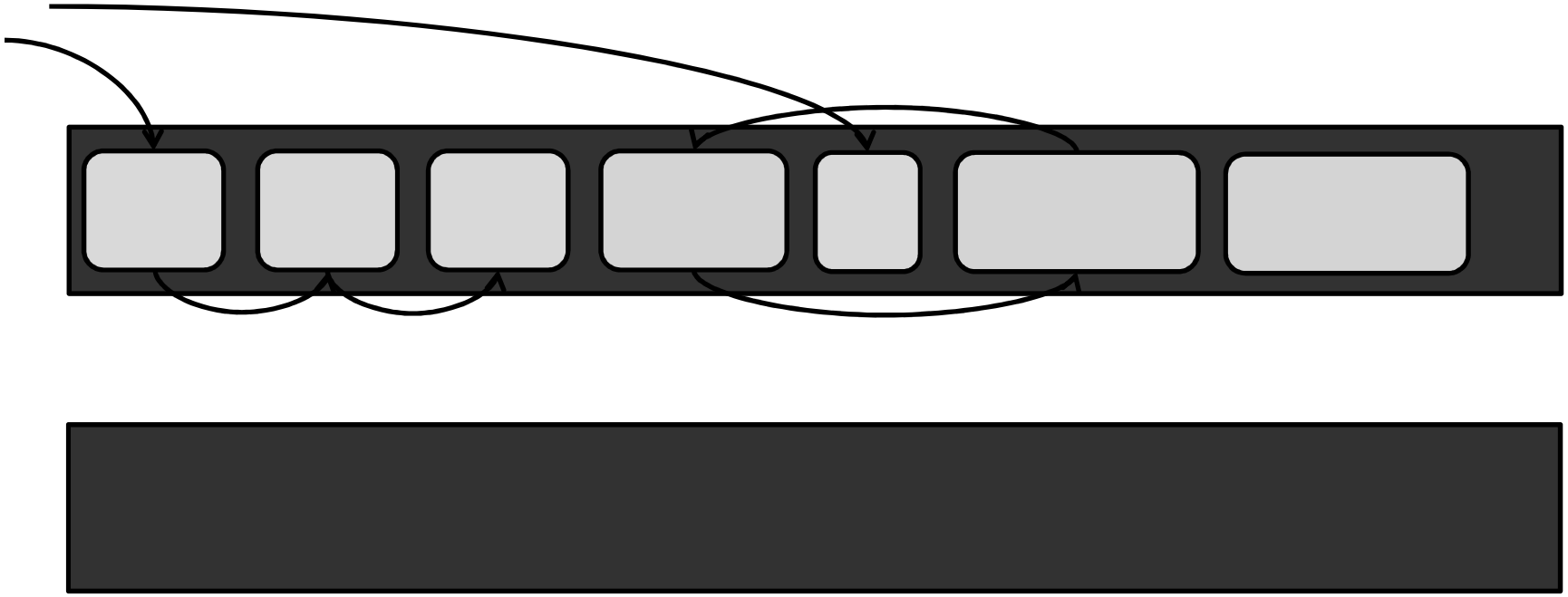
Alapok – GC

- Végül ami marad, azok a nem használt objektumok



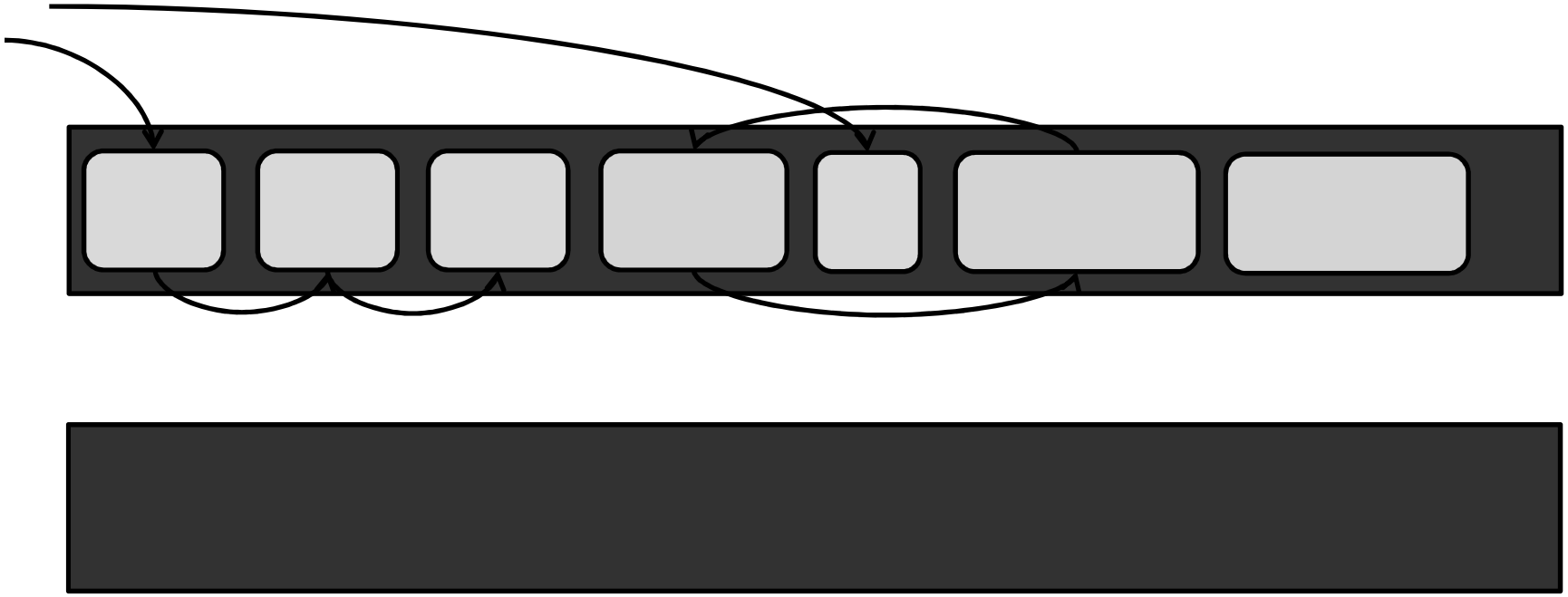
Alapok – Copying GC

- El kell dönteni, mi legyen a „fölösleges” és a megmaradó objektumokkal



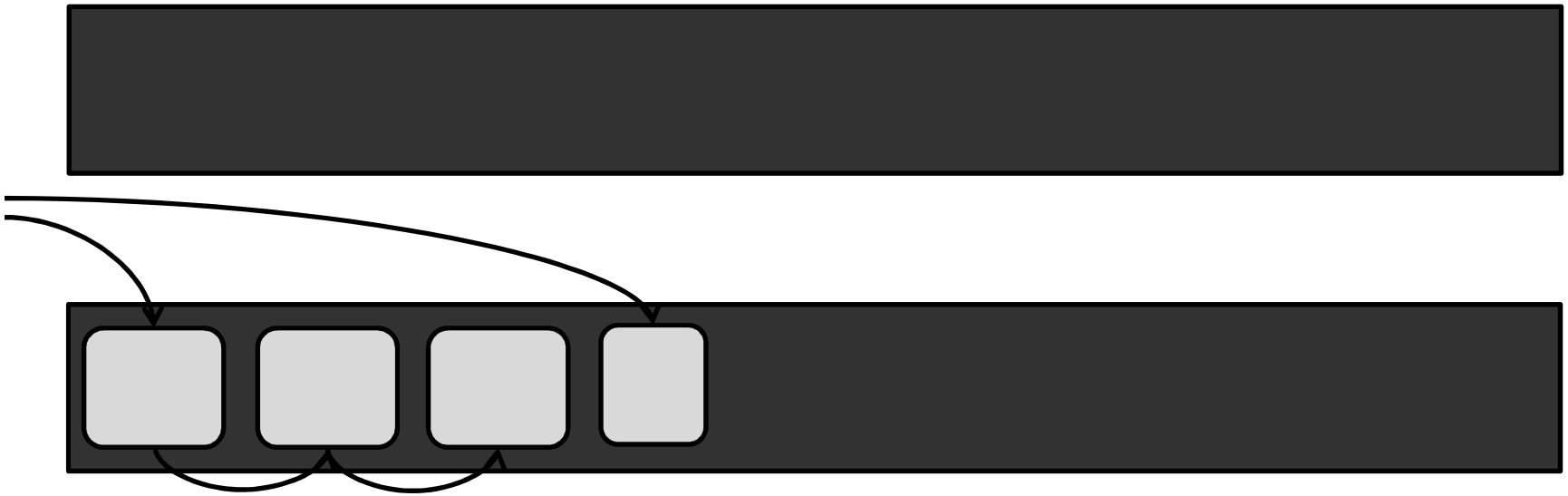
Alapok – Copying GC

- Ha két heap-et tartunk fent, akkor a GC végén elég csak egyikből a másikba másolni a megmaradó elemeket



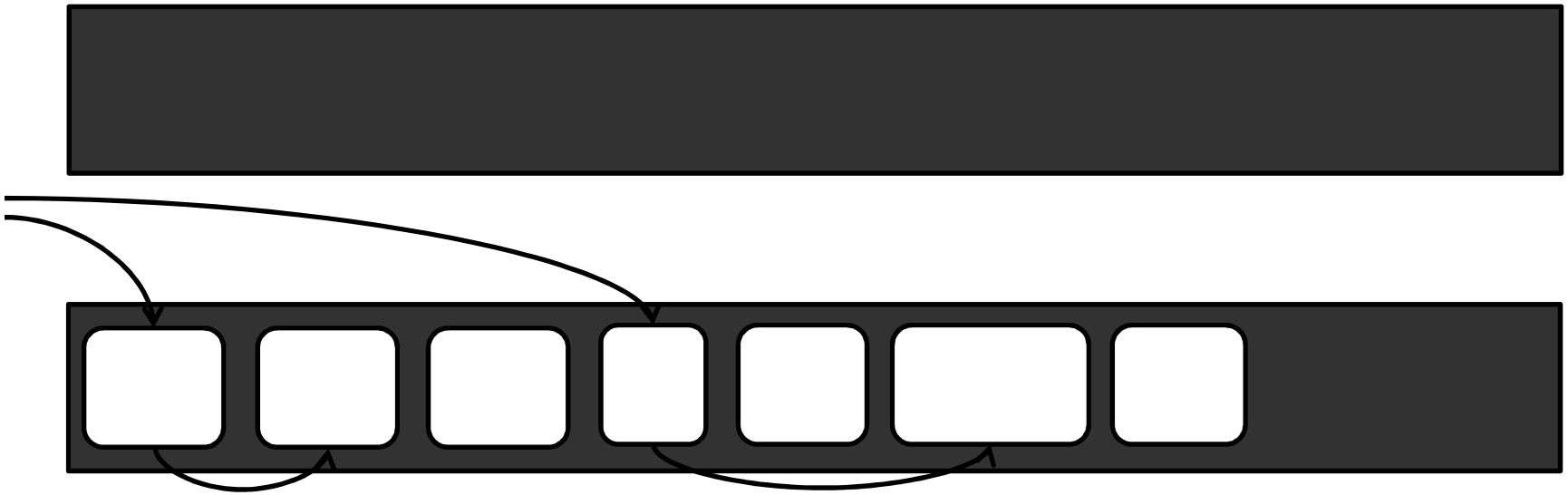
Alapok – Copying GC

- Ha két heap-et tartunk fent, akkor a GC végén elég csak egyikből a másikba másolni a megmaradó elemeket



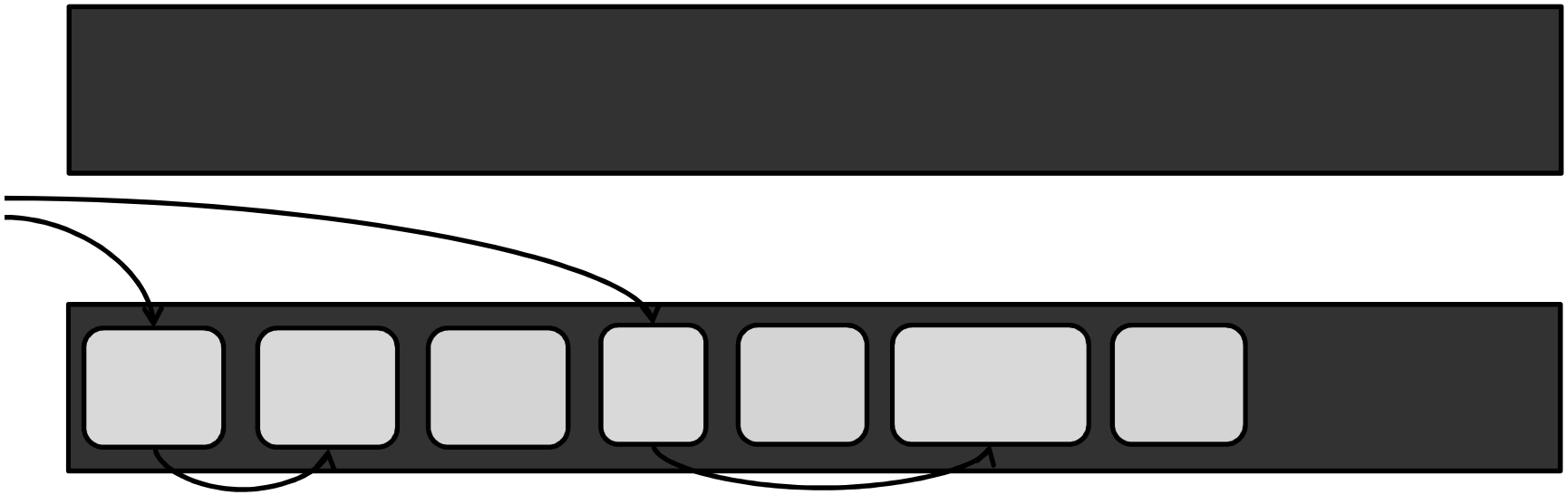
Alapok – Copying GC

- Az újabb GC menet az új heap-en történik, és onnan másolunk az előző heap-re



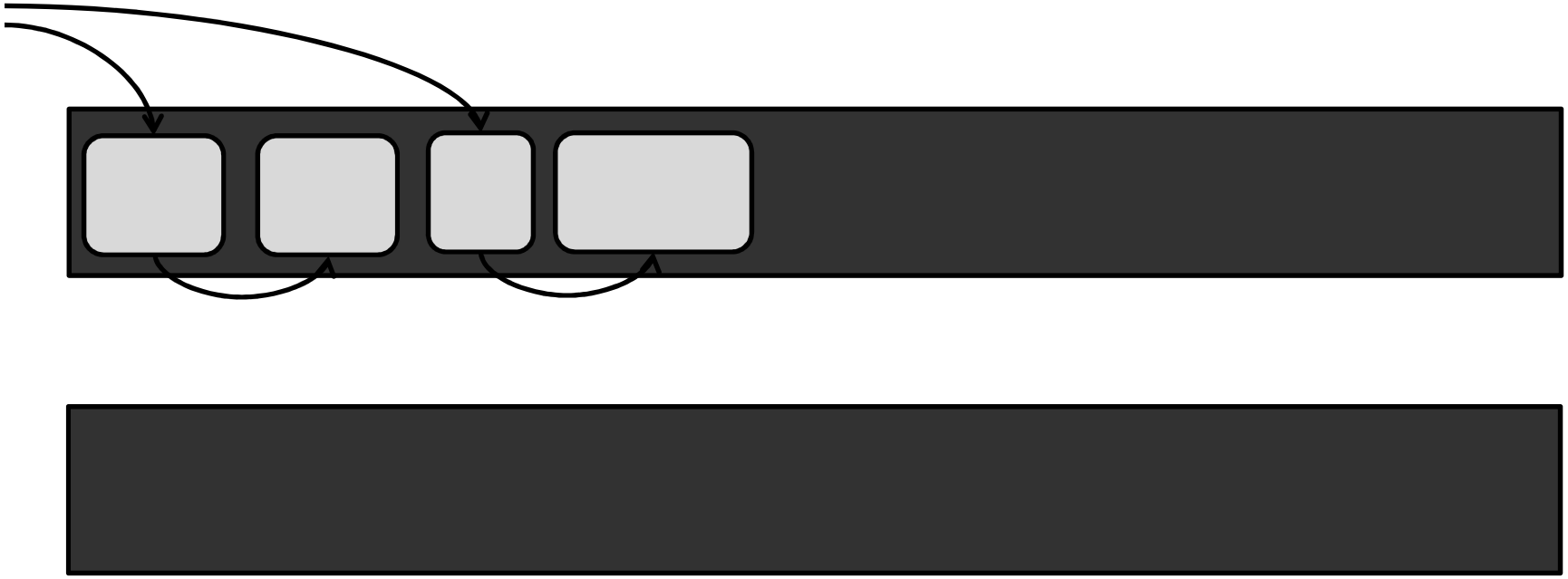
Alapok – Copying GC

- Az újabb GC menet az új heap-en történik, és onnan másolunk az előző heap-re



Alapok – Copying GC

- Az újabb GC menet az új heap-en történik, és onnan másolunk az előző heap-re

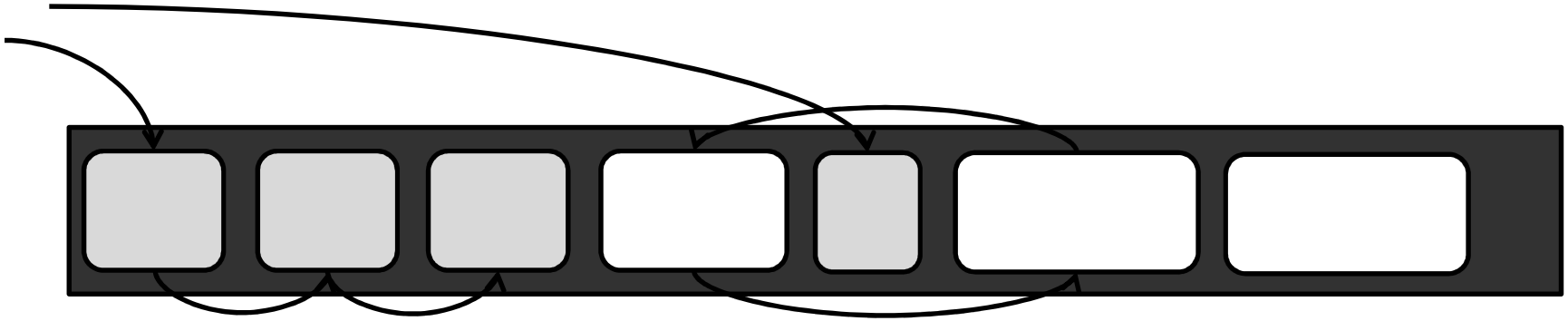


Copying GC – előnyök, hátrányok

- Előnyei:
  - Nagy, töredezetlen allokálatlan blokkok maradnak egy-egy GC végén
  - Ha kevés a túlélő, akkor nagyon gyors – csak kevés objektumot kell másolni
- Hátrányai
  - Két ugyanakkora méretű heap-et igényel – pazarlás sok esetben
  - Sok túlélő (survivor) esetén sokat kell másolni

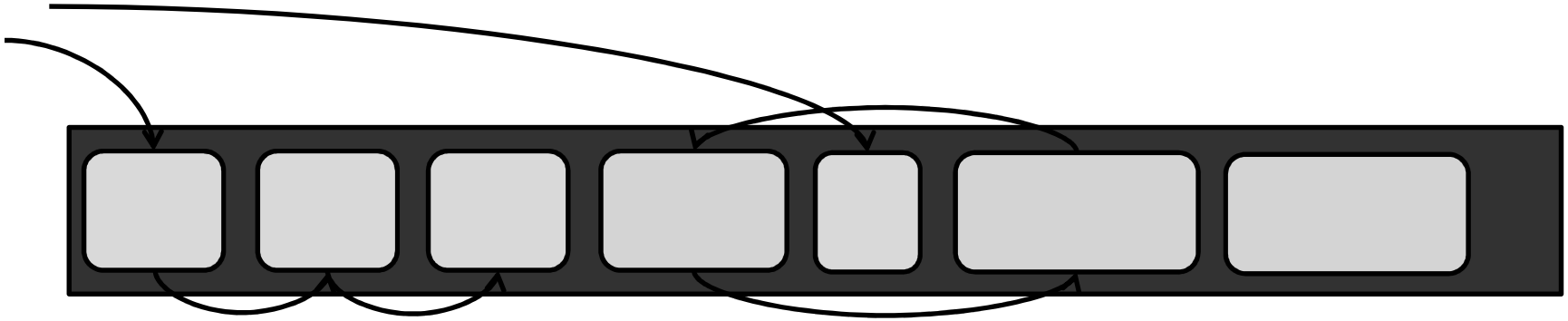
Mark-Sweep-Compact GC

- Másik népszerű GC elv
- Első körben ugyanúgy felderítjük az elérhető objektumokat
  - Mark



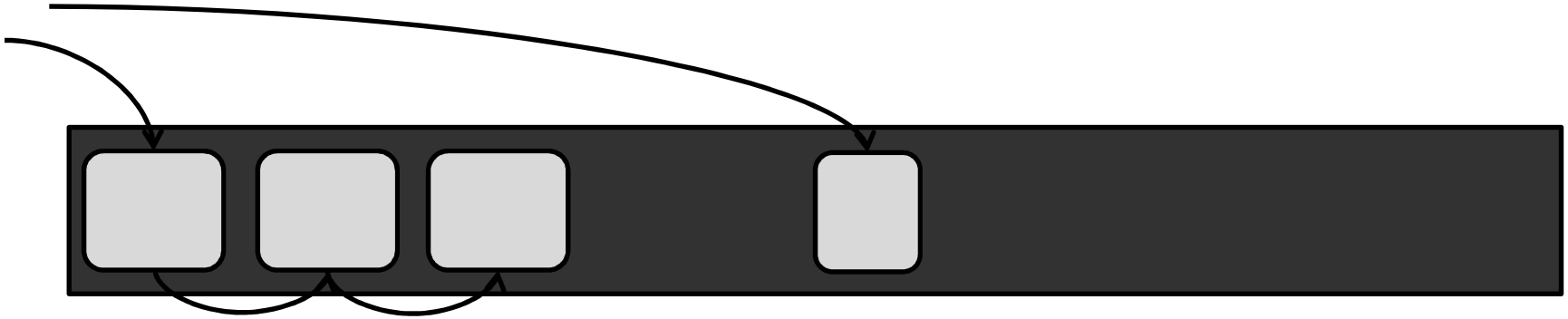
Mark-Sweep-Compact GC

- Második körben eltávolítjuk a nem használtakat
  - Sweep



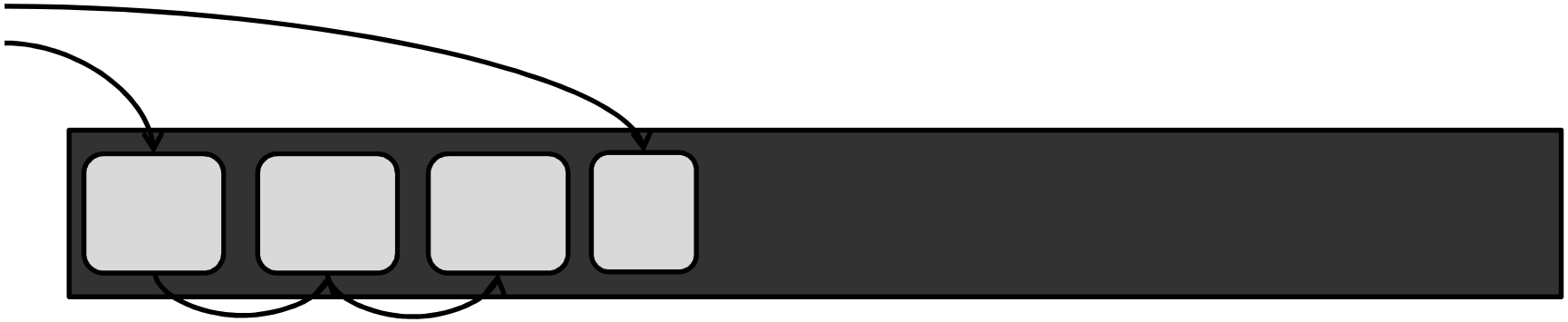
Mark-Sweep-Compact GC

- Második körben eltávolítjuk a nem használtakat
  - Sweep



Mark-Sweep-Compact GC

- Harmadik körben „tömörítjük” a megmaradtakat
  - Compact



MSC előnyök – hátrányok

- Előnyei
  - Csak egy heap-et igényel
  - Ha sok a túlélő akkor kevés másolást (mozgatást) igényel
- Hátrányai
  - A mozgatás költséges lehet
  - Ha nem mozgatunk mindig, akkor a heap töredezik
  - A túlélők mellett a „kihaltakat” is követni kell

MSC vs Copying

- Összefoglalva az alábbi (kissé egyszerűsített) megállapításokat tehetjük

Túlélők száma	Mark-Sweep Compact	Copying
Sok	Hatékony	Kevésbé hatékony
Kevés	Kevésbé hatékony	Hatékony

MSC vs Copying

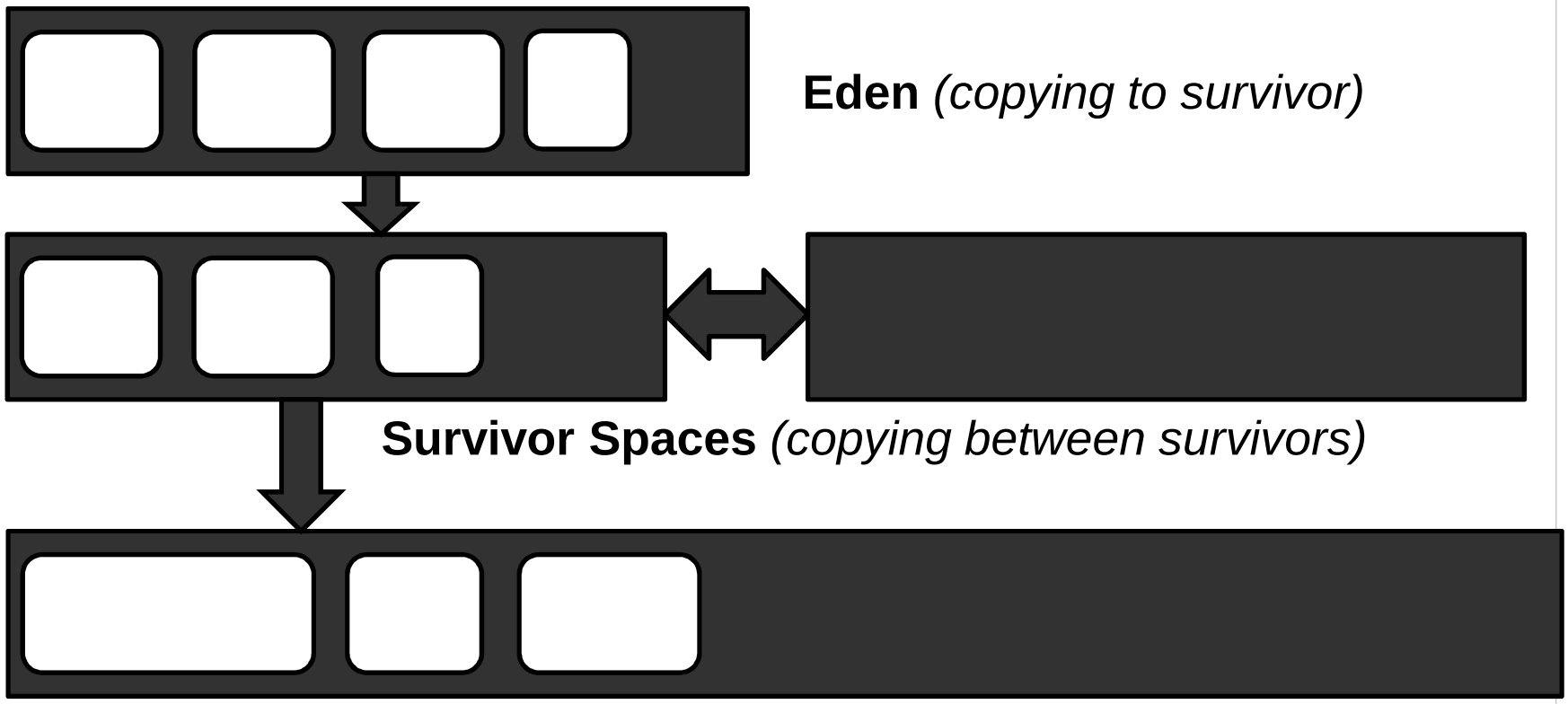
- Összefoglalva az alábbi (kissé egyszerűsített) megállapításokat tehetjük

Túlélők száma	Mark-Sweep Compact	Copying
Sok	Hatékony	Kevésbé hatékony
Kevés	Kevésbé hatékony	Hatékony

- Egy fontos gyakorlati megfigyelés:
  - A frissen „készült” objektumok általában rövid életűek
    - kevés a túlélő
  - A régóta „élő” objektumok általában megmaradnak hosszú ideig
    - sok a túlélő

Generational GC

- **Osszuk fel a heap-et generációkra!**
  - Fiatalok – Copying; Idősek – Mark-Sweep-Compact

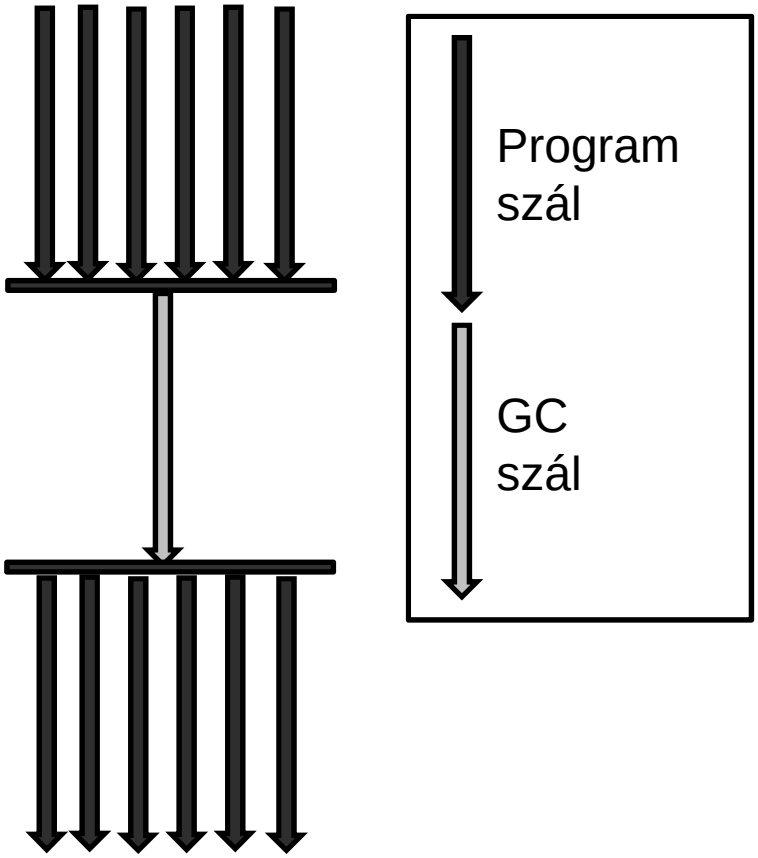


Serial vs Parallel vs Concurrent

- A GC algoritmusok esetén (hasonlóan a hálózatokhoz) két fontos minőségi jellemzőt különböztetünk meg
  - Átbocsátóképesség (Througput)
  - Válaszidő (Latency)

Serial GC

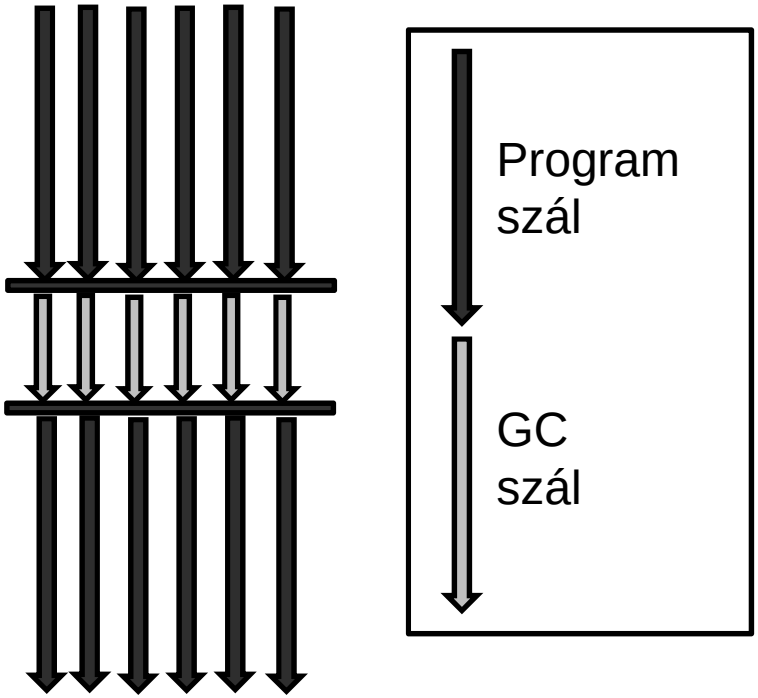
- GC eseménykor a program futása felfüggesztésre kerül, majd a GC algoritmus fut le:



- Egyszerű GC algoritmus
- Magas válaszidő
- Alacsony átbocsátás

Parallel GC

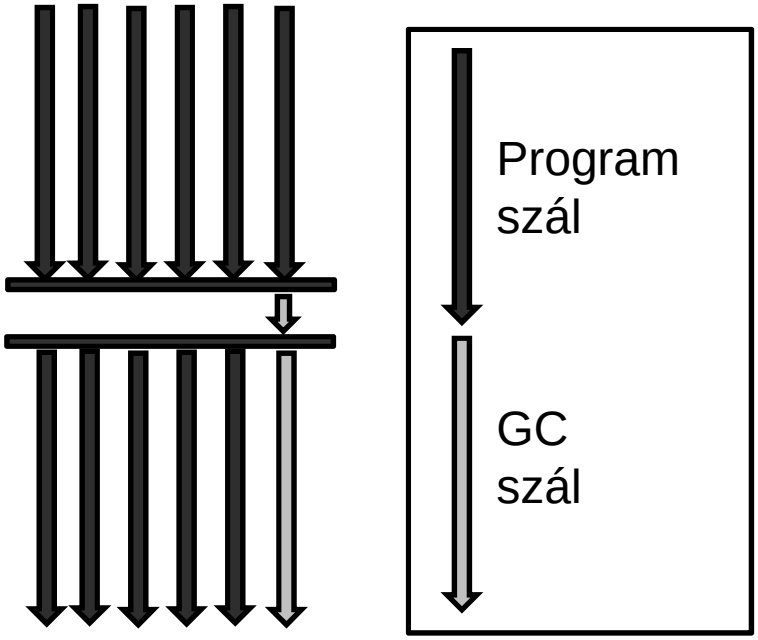
- GC eseménykor a program futása felfüggesztésre kerül, és a GC algoritmus **párhuzamosítva** fut le:



- Bonyolultabb GC algoritmus
- Közepes válaszidő
- Magas átbocsátás

Concurrent GC

- GC eseménykor a program futása **rövid** felfüggesztésre **kerül, és a GC algoritmus a programmal párhuzamosan** fut le:



- Bonyolultabb GC algoritmus
- Rövid válaszidő
- Közepes átbocsátás

Concurrent és Parallel

- Az előzőek alapján látszik, hogy a Concurrent illetve parallel fogalmak **ortogonálisak (függetlenek)**
  - Lehetséges ezek kombinációját is használni
  - Ennek részleteit nem tárgyaljuk
- Mikor melyik, ha választani kell?
  - Batch jellegű, offline feladat (szimuláció, bonyolult számítások) – Parallel
  - Interaktív jellegű, online feladat (weboldal, tőzsdei alkalmazás) – Concurrent

GC kapcsolók

- A különböző GC változatok használata

Young generation	Old generation	Kapcsoló
Serial	Serial	-XX:+UseSerialGC
Parallel	Serial	-XX:+UseParallelGC
Parallel	Parallel	-XX:+UseParallelOldGC
Serial	Concurrent	-XX:+UseConcMarkSweepGC -XX:-UseParNewGC
Parallel	Concurrent	-XX:+UseConcMarkSweepGC -XX:+UseParNewGC
G1		-XX:+UseG1GC

- + rengeteg egyéb tuningolási lehetőség...

Tévhitek I.

- Néhány tévhit és kétes értékű jótanács széles körben elterjedt a szemétgyűjtővel kapcsolatban. Ezeket elkerülve, a szemétgyűjtő működésének ismeretében segíthetjük annak munkáját.
- *“Új objektumok létrehozása drága.”*
  - Nem igaz, sőt, felszámolásuk is nagyon olcsó. Ha egy objektumot újból használni akarunk, sokszor érdemesebb újat létrehozni belőle, hogy az általa létrehozott objektumokkal együtt a fiatal generációba kerüljön, így a szemétgyűjtőnek nem kell nyilvántartani az idősebből a fiatalabb generációba mutató referenciákat.
- *“Ne dobjuk ki az újrafelhasználható objektumokat, tartsuk őket egy készletben, és vegyünk onnan, ha szükséges (object pooling).”*
  - Nem jó stratégia, kivéve, ha nagyon idő-, vagy erőforrásigényes a létrehozásuk, lásd fentebb.

Tévhitek II.

- *“A szükségtelenné vált objektumokat tartalmazó referenciákat állítsuk nullra, így a szemétgyűjtő hamarabb felszámolja őket.”*
  - Nem igaz. A nullra állításra néha valóban szükség van, például egy stack esetében, de általában semmi kedvező hatása nincs, ha a változóink érvényességi körét helyesen választjuk meg.
  - A változók érvényességi köre ne legyen nagyobb, mint okvetlenül szükséges (ne használjuk mezőt lokális változó helyett), így amint lehet, az általuk mutatott objektumok elérhetetlenné fognak válni.
- *“A* `System.gc()` *metódussal hatékonyan ütemezhetjük a szemétgyűjtést.”*
  - A `System.gc()` tudtára adja a szemétgyűjtőnek, hogy “valamikor mostanában jó lenne begyűjteni a szemetet”. Azonnali szemétgyűjtésről szó sincs.
  - A `System.gc()` mindig “nagy” szemétgyűjtést indít, ami drágább, és általában nincs is rá szükség, elég a fiatal generáció körében “tisztogatni”.

Tévhitek III.

- *“A Permanent Generation (PermGen) elemei állandóak”*
  - Nem igaz. Ha így lenne, akkor nem lenne szükséges őket a heap-en tárolni.
  - **A PermGen azoknak az objektumoknak a helye, amiket a virtuális gép hoz létre közvetlenül (pl: Class osztályok)**

Interpreter a HotSpot VM-ben, JIT compiler

INTERPRETER ÉS JIT COMPILER

Interpreter vs JIT compiler

- A JVM virtuális gép, vagyis platformfüggetlen bytecode-ot futtat
- Ezt naiv módon interpretálással tehetjük meg
  - Beolvassuk a következő bytecode-ot
  - egy switch-ben a bytecode alapján kiválasztjuk mit kell tenni
  - végrehajtjuk az utasítást
- A JVM interpretere **template alapú**
  - Nincs, switch, az utasítások helyére kész gépi kódú blokkok helyettesítődnek be
  - **még ez is lassú**
- Fordítani kell tehát a bytecode-ot és optimalizálni

JIT Compiler

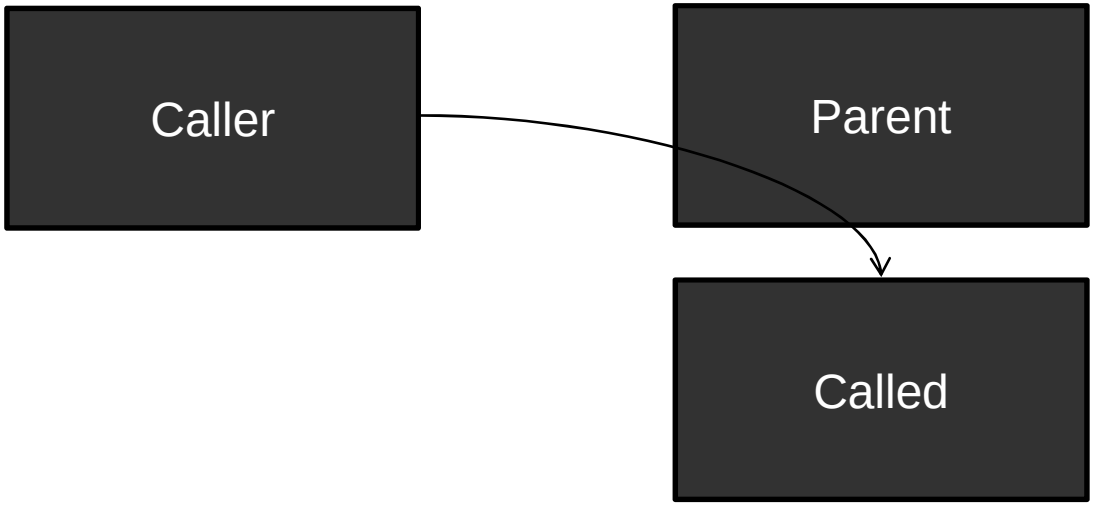
- Just-In-Time
  - csak akkor fordít amikor szükség van rá
- Megfigyelés: a programok az idejük 90%-át a kód 10%-ában töltik
  - Ezt a 10%-ot hívjuk hotspot-nak
  - Elég csak azt lefordítani, ami tényleg számít
- Kezdjük interpretálással a végrehajtást!
  - Mérjük ki futás közben, melyek a teljesítménykritikus részek
  - Optimalizáljuk menet közben

JIT compiler – eszközök, fogalmak

- Dependency
  - valamilyen feltételezés az adott kódrészletről
  - pl: az adott if utasítás feltétele mindig false
  - pl: az adott interfésznek (szülőosztálynak) csak egy megvalósítását használjuk
- On-stack replacement
  - Az a folyamat amelynek során az adott kódrészletet lefordított kódra cseréljük le
- Deoptimization
  - Amikor egy dependency sérül (a feltételezés hamissá válik) a VM visszavált interpretált módra

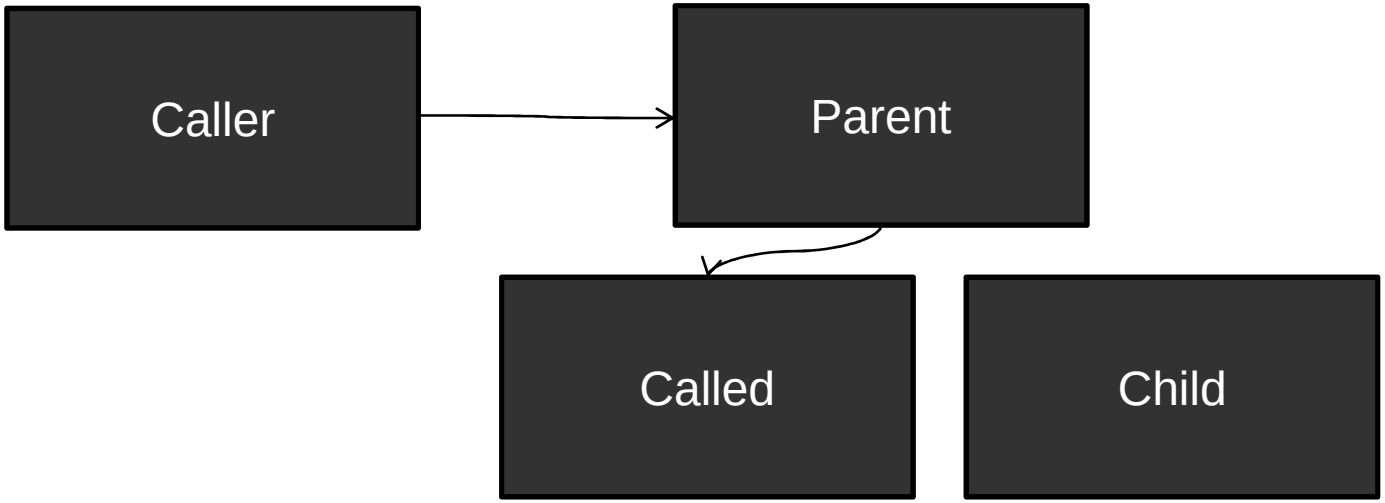
Inlining

- Amikor egy metódust sokat hívunk, akkor annak teste egy az egyben bemásolható a hívás pontjába
  - ezt hívjuk inlining-nak
- Ha egy Caller osztály egy Parent típuson keresztül hívja a Called osztályt és csak az az egy leszármazottja van Parent-nek, akkor inline-olható a hívás



Inlining

- Ha Parent típusnak van másik alosztálya is, akkor az inline-ing nem működik a polimorfizmus miatt
- Ilyenkor vagy nem lehet inlining optimalizálást végrehajtani, vagy ha menet közben töltődik be egy új alosztály, akkor az inline hívást deoptimalizálni kell



Egyéb módszerek

- Inline hívások esetén, ha a hívó lekezel egy kivételt amit a hívott dob, akkor az egy egyszerű GOTO (jump) utasítássá konvertálódhat
- Egyes könyvtárhívások közvetlenül gépi kódra fordulnak, nincs köztes interpretált futtatás
  - **compiler intrinsic**
  - System.arraycopy, array.size, stb.
- If esetén ritkán végrehajtódott ágakat „eldobhat”
  - ha mégis arra az ágra futunk, akkor deoptimalizálás történik
- Stb.

Client vs Server VM

- A JVM két fő compilert használ
  - C1, vagy „client”
  - C2, vagy „server” (vagy „opto”)
- Client
  - gyors indításra optimalizált
  - -client kapcsolóval érhető el
  - **ez az alapértelmezett**
- Server
  - „A” HotSpot compiler
  - Lassabb indulás, de agresszívebb optimalizálás
  - -server kapcsolóval érhető el

Locking, ligthweight locking, biased locking

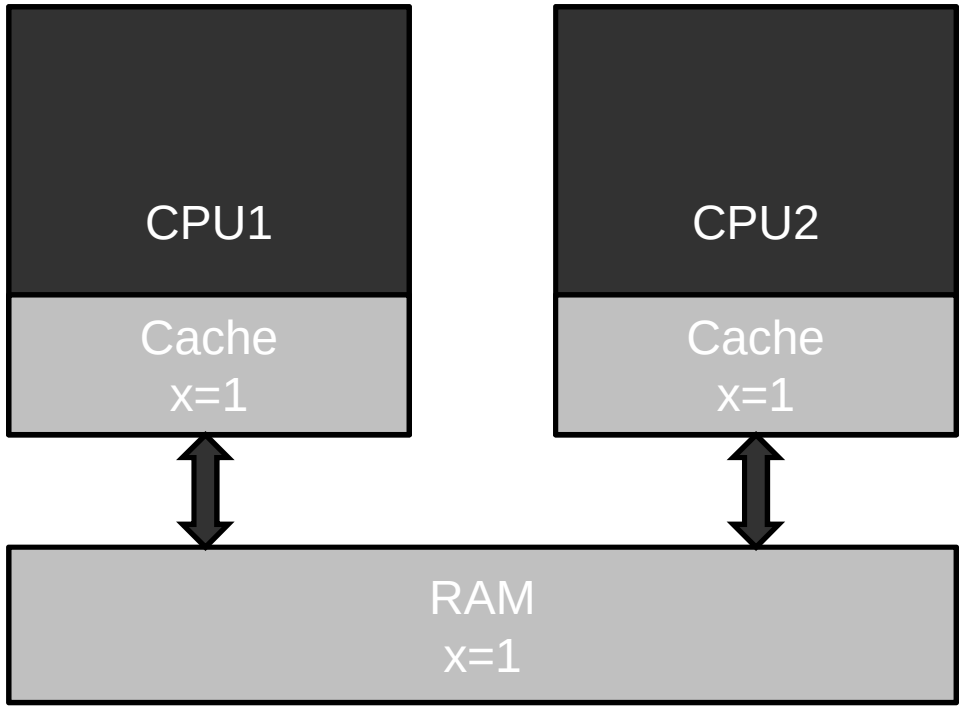
TÖBBSZÁLÚSÁG

Láthatóság

- A memóriáról általában úgy gondolkodunk, ha valamit beleírunk, akkor az ott is van, és a következő olvasás azt olvassa ki, amit tartalmaz
- A gyakorlatban ez egyáltalán nincs így!
- Azt, hogy egy írás látható-e egy olvasás számára, **láthatóságnak (visibility)** nevezzük

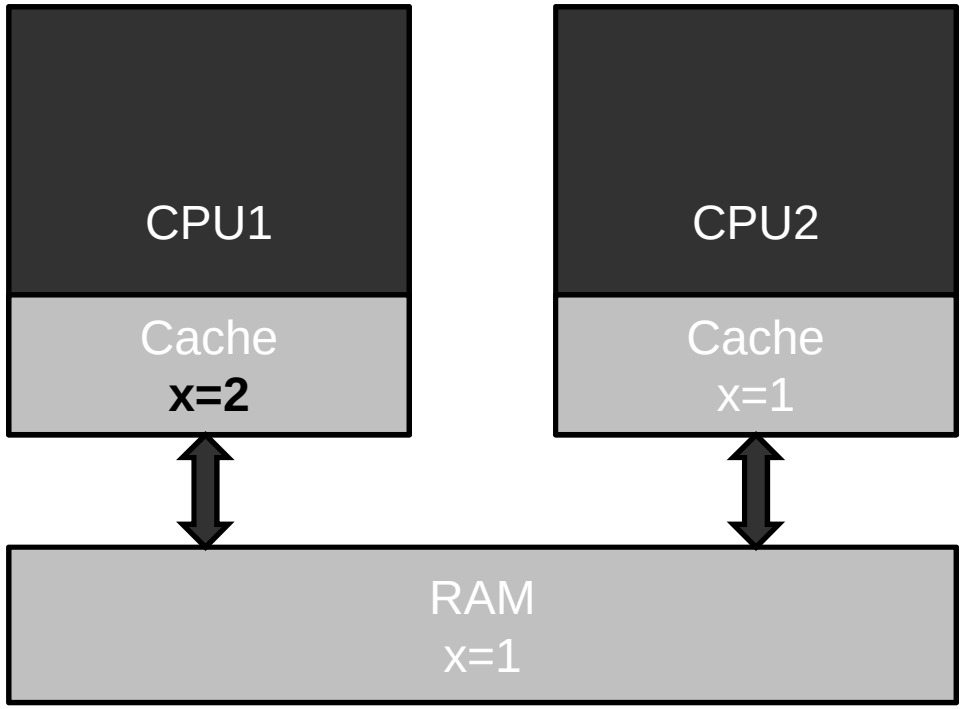
CPU cache inkonzisztencia

- Az egyik láthatóságot korlátozó tényező a cache



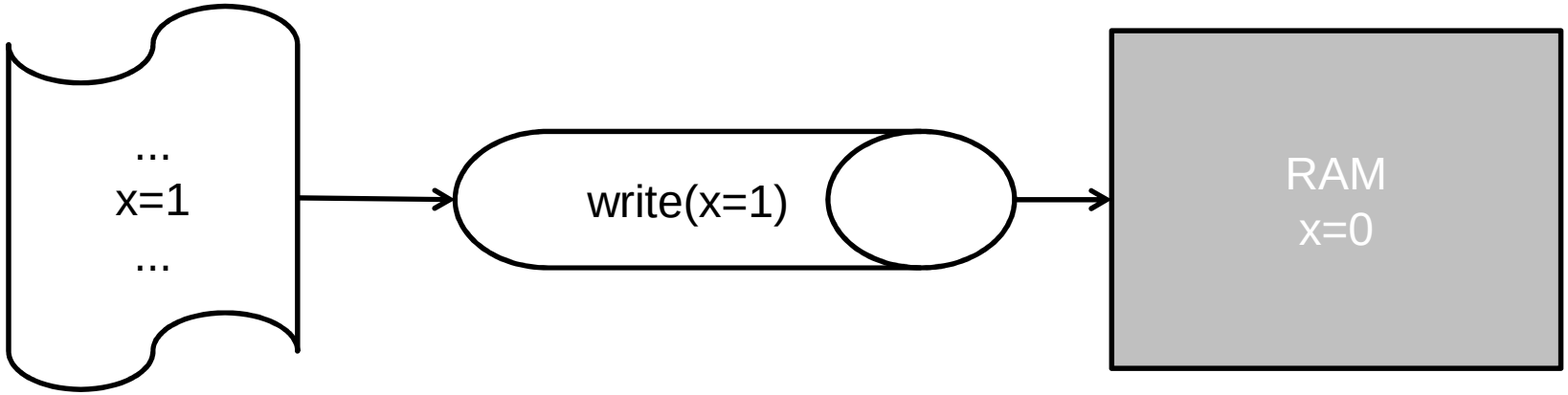
CPU cache inkonzisztencia

- A memóriaírás alapból nem látszik a RAM, sem a többi cache számára
- **Szinkronizálni kell!**



Out-of-order writes

- A cache inkonzisztencia csak több CPU (vagy mag) esetén léphet fel
- Vagyis egy CPU esetén nincs gond a láthatósággal?
  - A compiler külön utasítás hiányában **újrarendezheti** az utasításokat, így azok sorrendje nem garantált
  - A CPU nem hajtja azonnal végre az írásokat, hanem **buffereli** azokat



Memory barrier

- Ahhoz, hogy az írások és olvasások egymáshoz képest megfelelően sorrendezve legyenek, szinkronizálni kell
- Ennek az egyik módja a Memory Barrier
  - Intel utasítások: LFENCE, SFENCE, MFENCE
  - JVM-ben: LOCK ADD [ESP], 0
- Másik módja: CPU lock módjának bekapcsolása
  - Intel esetén: LOCK prefix, #LOCK vonalat állítja 1-be
  - Ez szolgál atomi műveletek megvalósítására is
- JVM-ben azokat a változókat, melyekre az írások és olvasások globálisan sorrendezettek kell legyenek, a **volatile** kulccszóval jelöljük

Locking

- Java-ban a beépített konstrukció a **synchronized** blokk
  - Atomi műveletek megvalósítására alkalmas
  - Ez egyben Memory Barrier is
  - Minden írás egy unlock(M) előtt látható minden olvasás számára egy lock(M) után
- Lock-olni költséges
  - Cache-eket össze kell hangolni
  - Egyéb optimalizációk is felfüggesztésre kerülnek
- Jó, hogyha elkerüljük amikor lehet

Lightweight locking

- Megfigyelés: aki elkér egy lock-ot (lock(M)) az lesz nagy valószínűséggel a következő felhasználója a lock-nak
  - Legyen a lock olcsó, ha már megszereztük
- Lightweight locking
  - Alapból nem az operációs rendszer (és CPU) „nehézsúlyú” módszereit használjuk
  - Ez a fajta locking nem hatékony, ha sok különböző felhasználója van ugyanannak a lock-nak (**lock contention**)
  - Ilyenkor deoptimalizálunk: a lock-ot konvertáljuk „nehézsúlyú” lock-ká
    - **lock inflation**

Biased locking

- Megfigyelés: nagyon gyakran ugyanaz a szál használja a lock-ot
- **Biased locking**
  - Adjuk a szálnak a lock-ot teljes egészében, és ezt jelöljük
  - Ilyenkor egy újabb lock-nak gyakorlatilag nulla a költsége
  - Ha másik szál éri el ugyanazt a lock-ot, akkor vonjuk vissza az egészet, és állítsunk be mindent úgy, mintha az eredeti szál lighweight locking-ot használt volna
    - **bias revokation**

Locking összefoglalás

- Egy beépített lock élete a JVM-ben
  - Unlocked -> Biased -> Lightweight locked -> Inflated
- Ahogy nő az adott lock különböző szálak által történő használatának gyakorisága **(lock contention)**, úgy váltunk egyre „nehézsúlyúbb” megvalósításra
- **Ezt csak virtuális gépen lehet hatékonyan megtenni, ahol a kód menet közben kerül fordításra!**
  - Többek között ezért nagyon gyors a Java többmagos környezetben, gyakran C kóddal egyenértékű teljesítménnyel

Kérdések?

KÖSZÖNÖM A FIGYELMET!

Híradástechnikai Tanszék

Varga Endre Sándor
doktorjelölt
BME Híradástechnikai Tanszék
vendre@hit.bme.hu