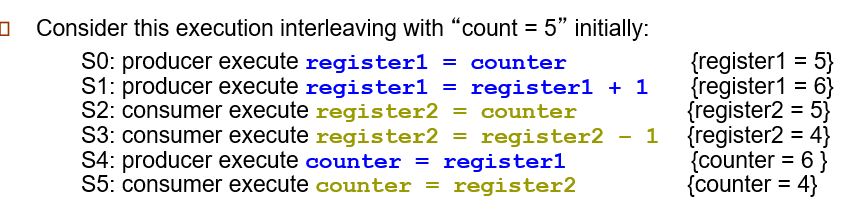
**Chapter 5**

Folyamat szinkronizáció – 8. előadás

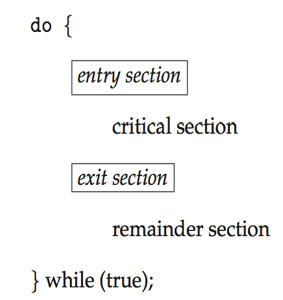
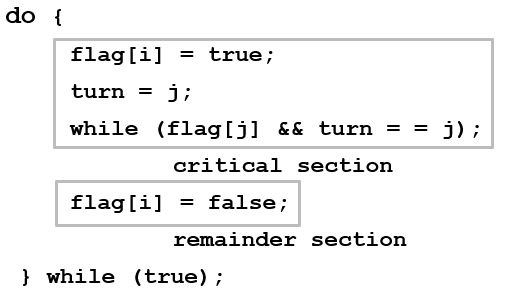
Alapvetések eddig

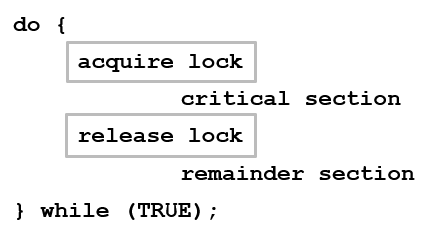
* Producer – Consumer koncepció
  + Működhet, ha időben nem közel történik két dolog
* Folyamatok mehetnek konkurensen
  + Meg lehet szakítani, részben elvégezve a folyamatot
* Konkurens elérés osztott adathoz adat inkonzisztenciához vezethet
  + Későbbiekben erre lesz példa mi is ez az inkonzisztencia
  + Módok kellenek arra, hogy elkerüljük ezt

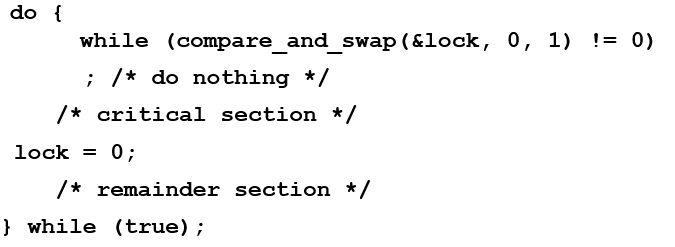
Race Condition - Versenyhelyzet

* Példa az adat inkonzisztenciára, versenyhelyzetre
  + Mivel a producer és consumer bármikor megszakítható, ezért hiba lehet, ahogy itt is, keresztül húzza egymást a producer és consumer
  + Ha mondjuk így egy tömb x.-dik elemét keressük, és egy 10 elemű tömbnél a counter 15, az még rosszabb, mert nem vesszük észre a hibát, és kiírhat valami memória szemetet
* Kell valami megoldás, a versenyhelyzet elkerülésére 🡪 Kritikus szakasz

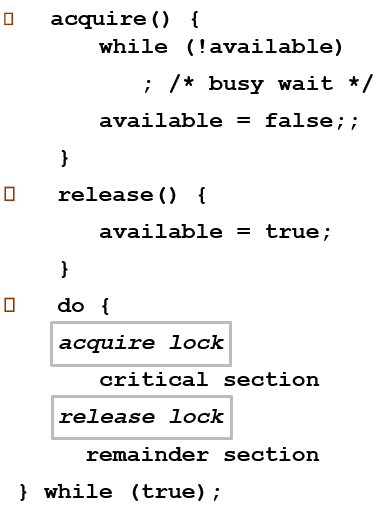
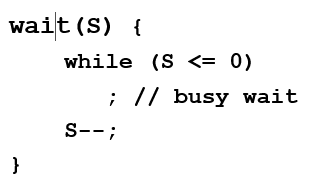
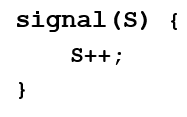
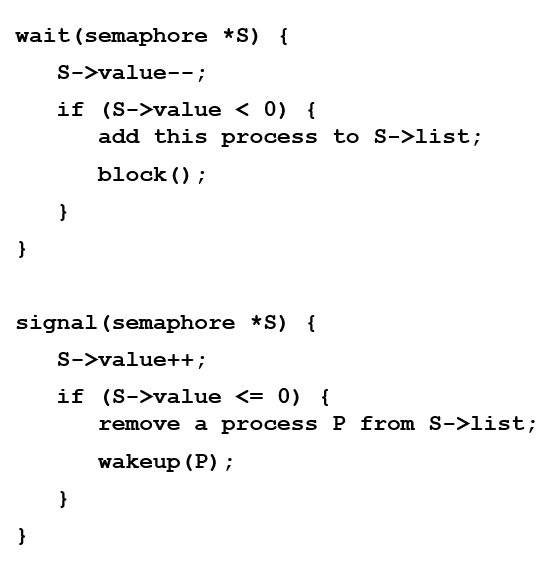
Kritikus szakasz – Critical Selection

* A megoldás a versenyhelyzetre
* Bevezetünk egy fogalmat, a kritikus szakaszt 🡪 Minden folyamatnak van ilyen része
  + Közös változók változtatása, fájlírás, egy táblázat frissítése stb.
  + Amikor egy folyamat a kritikus szakaszban van, akkor más folyamat nem lehet a kritikus szakaszában
* Alapelképzelés
* Hogyan védjük a kritikus szakaszt?
  + Kölcsönös Kizárás
    - Egyszerre csak egy folyamat lehet kritikus szakaszában
  + Haladás
    - Ne lehessen egy folyamat kritikus szakaszát végtelenségig halasztani
    - Ha nincs egy folyamat se kritikus szakaszba, szerzünk egyet, ami abba akar menni
  + Ütemezett várakozás
    - Bármely program véges időn belül be tudjon lépni a kritikus szakaszába
    - Van egy limit egy folyamat mennyiszer kérheti a kritikus szakaszba való belépést
* Kritikus szakasz kezelése OS-ben
  + A kernel módban történik a kritikus szakasz
  + Kétféle hozzáállás
    - Nem preemptív
      * Ha folyamat Kernel módban van, nem lehet tőle elvenni a gépidőt
      * Versenyhelyzet lényegében nem fordulhat elő
      * Ilyen nem nagyon van mert a user türelmetlen
    - Preemptív
      * Nem számít milyen módban van, ha ütemező el akarja, elveszi tőle a gépidőt
      * Összes modern OS
      * Gyorsabb, reszponzívabb 🡪 De megszakadhat bármikor
        + Mi legyen, ha kritikus szakaszban veszi el?
  + Peterson megoldása
    - Load és Store atomi utasítások, nem lehet megszakítani
    - Két közös változó folyamatok között
      * Int turn
        + Megmondja kinek a „köre” a kritikus szakaszba lépni
      * Boolean flag[i] 🡪alapból hamis
        + Megmondja egy folyamat készen áll-e belépni a kritikus szakaszba
        + Flag[i]=true 🡪 i.-dik folyamat készen áll
    - Teljesül a
      * kölcsönös kizárás
      * Haladás
      * Ütemezett várakozás

Hardveres szinkronizáció

* Locking/Zárolás
  + Belép kritikus szakaszba 🡪 Megkéri a zárat 🡪 Végez 🡪 Elengedi a zárat
  + Hardveres támogatás kritikus szakasz implementációjához
  + Atomi utasítás 🡪 Nem lehet megszakítani, mert hardverszintű
    - Letiltja a CPU az ütemezőt
  + 2 folyamatra biztosan működik
* Compare and Swap
  + Ugyanaz, mint a Lock, csak van benne egy if, hogy olyan állapotban van-e a zár, mint vártam
  + 2 folyamatra biztosan működik
* Megoldás több folyamatra 🡪 Waiting
  + Van egy waiting segéd görbe
  + Egy folyamat kritikus szakaszban 🡪 Letárolja az összes dolgát 🡪 Exit section   
    🡪 Végigmegy folyamatosan egy tömbön, amiben azt tároljuk melyik folyamat várakozik éppen, hogy kritikus szakaszba lépjen
    - Ha van, kijelöli a következőt
    - Ha nincs, kinyitja a zárat, és a következő folyamat amelyik majd egyszer kritikusba akar lépni beléphet azonnal
* Eddig voltak Hardveres Lockok, innentől szoftveres

Szoftveres szinkronizáció Megoldások

* A hardveres megoldások komplikáltak, és nem minden programozó számára elérhetőek
* Mutex Lock
  + Kritikus szakasz
    - Kezdete 🡪 Acquire lock
    - Vége 🡪 Release Lock
  + Egy boolean változó, hogy nyitott-e a zár
  + Probléma🡪 Összes folyamat be akar lépni 🡪 Busy waiting
    - Gépidő égetés
    - Spinlock
* Semaphore
  + Egy szofisztikáltabb mutex lock
  + Egy erőforráshoz több folyamat
  + S 🡪 Egész szám mennyi folyamat lehet egyszerre kritikus szakaszban
    - Ha nulla, akkor maximumon van jelenleg futó kritikus folyamatok száma
  + Két atomi utasítás
    - Wait
      * Meg akarja várni amíg erőforrást kap
    - Signal
      * Le akarja adni az erőforrást
  + Kétféle semaphore
    - Bináris semaphor
      * S =0 vagy 1 🡪 Lényegében egy Mutex Lock
    - Számoló semaphor
      * S értéke tetszőleges
  + Sok problémát megold
  + Garantálni kell, hogy egy folyamat nem hajthatja végre egyszerre a wait-et és signal-t
    - Wait és Signal a kritikus szakaszban
    - Így viszont keletkezhet Busy waiting
      * Kevés folyamatnál nem gáz
    - Alkalmazások sok időt tölthetnek a kritikus szakaszban, szóval ezt we no likey
  + Semaphore implementáció Busy Waiting nélkül
    - Minden semaphore-nak van egy waiting sora (queue)
      * Ez tartalmaz egy értéket
      * És egy mutatót a sor következő rekordjára
    - Block művelet
      * A folyamatot, ami meghívta a megfelelő waiting queue-ba teszi
    - Wakeup művelet
      * Kivesz egy folyamatot a waiting queue-ból, beteszi ready-be

Deadlock/Holtpont és Starvation/Éhezés

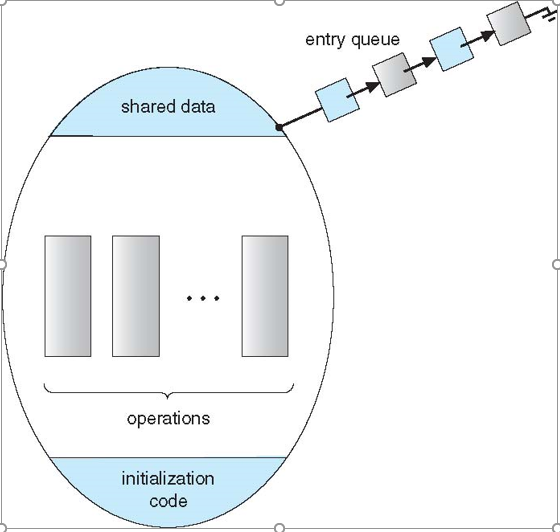
* Deadlock
  + Kettő vagy több folyamat határozatlan ideig vár egy eseményre, ami az egyik másik várakozó folyamat által keletkezhetne, tehát nem jutunk sehova
* Starvation
  + Soha nem tud kijutni a semaphore sorból ahova beragadt
* Megoldások 🡪 Prioritások

Klasszikus szinkronizáció problémák

* Bounded Buffer
* Readers Writers Problem

Dining Philosphers Problem – A kajáló filozófusok problémája

* 5-en ülnek egy körasztal körül
* Két állapota van a filozófusoknak 🡪 Esznek, vagy gondolkodnak
* Nem foglalkoznak a mellettük ülővel
* Időnként enni akarnak, amihez fel kell venni a jobb és bal oldalukról egy-egy evőpálcikát
* Megosztott adat
  + Egy tál rizs az asztal közepén
  + 5 darab evőpálcika
* Először ha enni akarnak felveszik a jobb oldalukról a pálcikát, majd a bal oldalról
  + Ha nincs egy oldalon pálcika várnak, hogy a mellettük ülő befejezze az evést
* Probléma 🡪 Mind az öten egyszerre akarnak venni
  + Mind felveszik az egyik oldalukról a pálcikát és várnak, hogy a másik oldalukon ülő befejezze a kajálást, mert ott nincs pálcika
  + Mind az öten egymásra várnak, és soha nem fognak előrébb jutni 🡪 Deadlock
* Megoldások
  + Max 4 filozófus ülhessen az asztalnál
  + Csak akkor vegyenek fel pálcikát ha egyszerre mind a kettő elérhető
    - Felvevés kritikus szakasz
  + Több evőpálcika

Monitorok

* Egy magas szintű absztraktció, ami lehetővé teszi kényelmesen és hatékonyan a folyamatok szinkornizációját
* Absztrakt adattípus
  + Belső változók csak a belső kódnak elérhetőek 🡪 Class lényegében
* Egyszerre csak egy folyamat lehet aktív a monitorban

