Feladatok (task) együttműködése

dr. Kovácsházy Tamás

5. anyagrész,

Kölcsönös kizárás, szinkronizáció, kommunikáció

Méréstechnika és
Információs Rendszerek
Tanszék

Feladatok együttműködése

- Kérdések:
  - Erőforrások használata, közös erőforrások?
  - Feladatok kommunikációja?
  - Feladatok szinkronizációja?
  - Architektúra függő kérdések?
    - Mit és hogyan használunk a feladatok megoldására?
    - Mik a következmények?

Párhuzamos végrehajthatóság

- Bernstein feltétele:
  - Pi és Pj két darabja egy programnak.
  - Pi összes bemeneti változója Ii, és az összes kimeneti változója Oi, ugyan ez Pj-re Ij és Oj.
  - A két program párhuzamosan végrehajtható (vagyis független):

$$Ij \cap Oi = 0$$

$$Ii \cap Oj = 0$$

$$Oi \cap Oj = 0$$

Együttműködés lehetőségei

- Közös memórián keresztül (RAM v. PRAM modell):
  - Szálak esetén (közös memória).
- Üzenetekkel:
  - Részletesen beszélünk róla később.

RAM modell

- Klasszikus Random Access Memory (egy végrehajtó egység).
- RAM-modell szerint működik, azaz:
  - Tárolórekeszekből áll,
  - Egy dimenzióban, rekeszenként címezhető, csak rekeszenként, írás és olvasás műveletekkel érhető el,
  - Az írás a teljes rekesztartalmat felülírja az előző tartalomtól független új értékkel,
  - Az olvasás nem változtatja meg a rekesz tartalmát, tehát tetszőleges számú, egymást követő olvasás az olvasásokat megelőzően utoljára beírt értéket adja vissza.

PRAM modell

- Parallel Random Access Memory (sok végrehajtó egység).
- Több végrehajtó egység írhatja és olvashatja párhuzamosan.
- Változások a RAM modellhez képest:
  - Az olvasás-olvasás ütközésekor mindkét olvasás ugyanazt az eredményt adja, és ez megegyezik a rekesz tartalmával,
  - Az olvasás-írás ütközésekor a rekesz tartalma felülíródik a beírni szándékozott adattal, az olvasás eredménye vagy a rekesz régi, vagy az új tartalma lesz (versenyhelyzet), más érték nem lehet,
  - Az írás-írás ütközésekor valamelyik művelet hatása érvényesül, a két beírni szándékozott érték valamelyike írja felül a rekesz tartalmát (versenyhelyzet), harmadik érték nem alakulhat ki.
- A gyakorlatban ezt használjuk…

Erőforrás

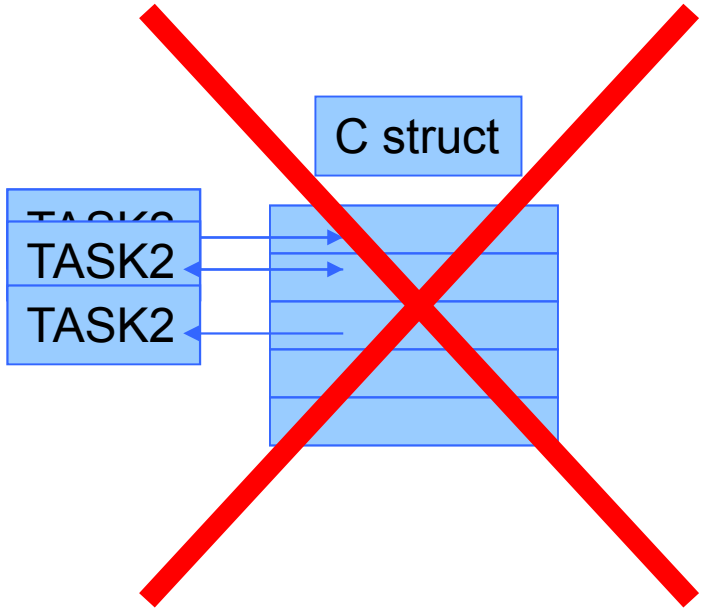
- Erőforrás (resource)
  - Minden olyan eszköz, amire a párhuzamos programnak futása közben szüksége van.
  - A legfontosabb erőforrás a végrehajtó egység.
  - Memória és annak tartalma (tárolt adatstruktúrák).
  - Perifériák.
  - Stb.

Közös erőforrás

- Közös erőforrás (shared resource)
  - Egy időintervallumban több, párhuzamosan futó feladatnak lehet rá szüksége.
  - Az erőforráson osztoznak a feladatok.
  - Többnyire egy időben egy vagy maximum megadott számú feladat tudja helyesen használni (írás és olvasás).
    - Egy felhasználó: Printer, Asszinkron soros port (UART), összetett adattípusokból létrehozott változók (string, tömb, struktúra, objektum).
    - Több párhuzamos felhasználó: SCSI vagy SATA NCQ HDD (N parancs optimalizált párhuzamos végrehajtására képesek).
  - ***A rendszertervezőnek és programozónak a legfontosabb feladata, hogy felismerje a közös erőforrásokat, és biztosítsa azok helyes használatát.***
  - Az OS szolgáltatásokat nyújt a probléma megoldására, de a megoldás a programozó kezében van!

Példa

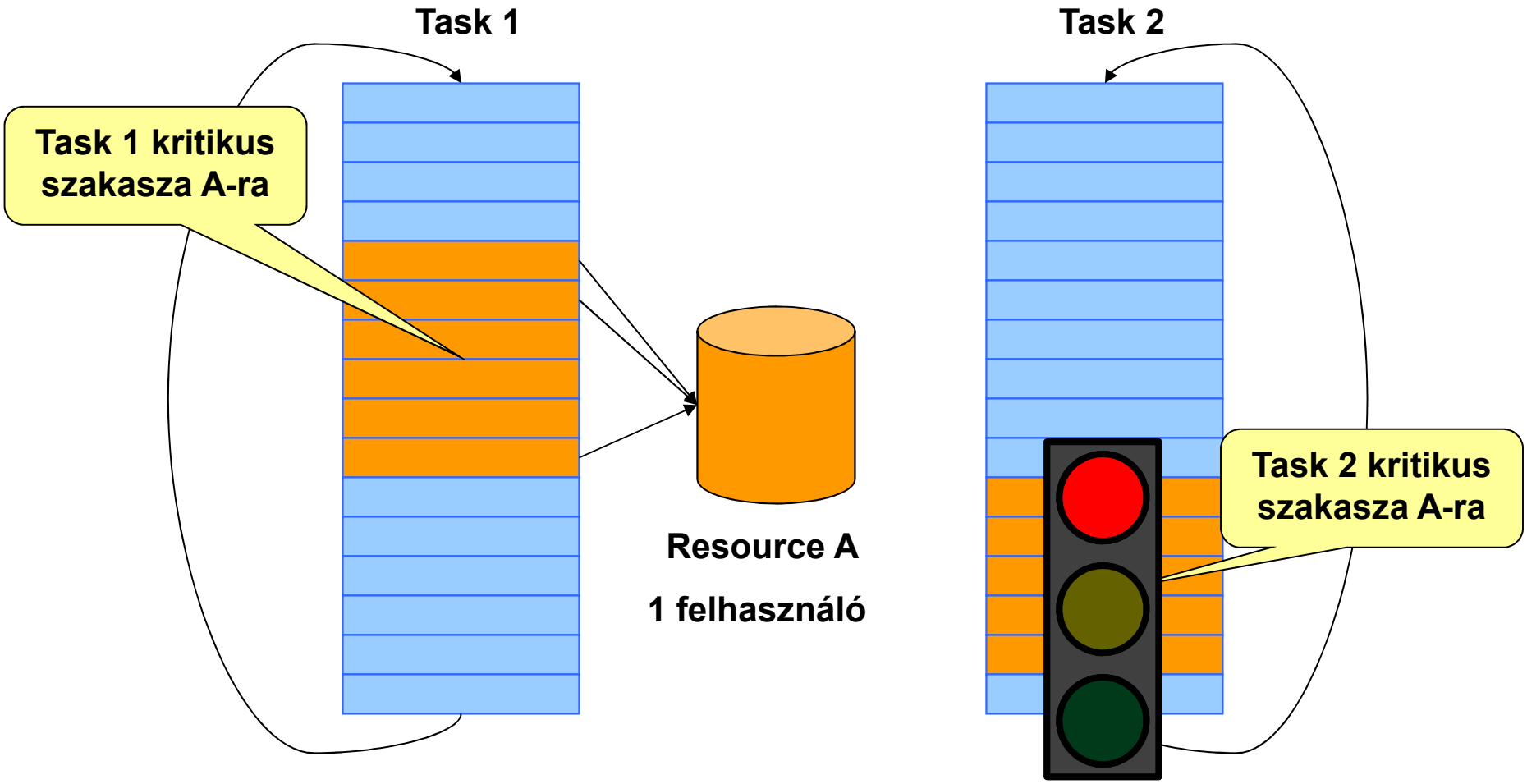
- Összetett adattípus:
  - C struktúra vagy tömb típusú változó.
- Közös erőforrás, ha több részfeladat használja:
  - Task1 írja.
  - Task2 olvassa.
  - Task1 fut először.
  - Task2 futtatására vált a rendszer a írás közben (pl. preemptív a rendszer).
  - Inkonzisztens állapotban olvassa ki a változót Task2.
  - Súlyos hiba!



A probléma megoldása

- Kölcsönös kizárás (mutual exclusion)
  - Annak biztosítása, hogy a közös erőforrást egy időben csak annyi magában szekvenciális feladat használja, amely mellett a helyes működése garantálható.
  - A kölcsönös kizárást meg kell oldanunk a programban.
  - Többnyire a használt erőforrást lock-oljuk (elzárjuk).
    - Nem engedjük hozzáférni a többi részfeladatot.
    - A kérdés az, hogy azt hogyan tudjuk megoldani, és milyen részletességgel kell megoldanunk azt.
- Kritikus szakasz (critical section)
  - A magában szekvenciális feladatok azon kódrészletei, amely során a kölcsönös kizárást egy bizonyos közös erőforrásra biztosítjuk.
  - A kritikus szakasz a kérdéses közös erőforráshoz tartozik.
  - A kritikus szakaszt a hozzá tartozó erőforrásra atomi műveletként (nem megszakítható módon) kell végrehajtanunk.

Szemléltetés



Atomi művelet

- Atomi művelet (atomic operation)
  - Nem megszakítható művelet, amelyet a processzor egyetlen utasításként hajt végre.
  - Egyprocesszoros rendszerben bármilyen műveletsor atomivá tehető a műveletsor elején az IT teljes tiltásával, majd a műveletsor végén annak engedélyezésével.
  - TAS, RMW, speciális CPU utasítások az IT tiltás/engedélyezés elkerülésére.
    - Test and Set, Read-Modify-Write, stb.
    - Elemi adattípusra (8/16/32/64 bit).
    - A modern processzoroknak vannak ilyen utasításai.
  - A közös erőforrások lock-olást, a kritikus szakasz megvalósítását atomi műveletekre vezetjük vissza.

Közös erőforrások védelme

- Mik férhetnek hozzá a közös erőforrásokhoz?
  - ISR (Interrupt service routine).
  - Feladat (folyamat vagy szál).
  - DMA.
- Lehetőségek:
  - IT tiltása és engedélyezése. (speciális esetben lehetséges).
  - Ütemező tiltása és engedélyezése. (speciális esetben lehetséges).
  - Locking (erőforrás specifikus lefoglalás majd feloldás).
- Más, kutatási fázisban lévő megoldások:
  - Software/hardware transactional memory (STM/HTM).
  - Software-isolated processes (MS Singularity).
- A lock megoldás sokak szerint nem jó, de jobb mint bármi eddig használt megoldás, vagyis ez az elfogadott megoldás.

Közös erőforrások védelme

- Mik férhetnek hozzá a közös erőforrásokhoz?
  - ISR (Interrupt service routi
  - Feladat (folya
  - DMA.
- Lehetőség
  - IT tilt
  - Üteme
    lehetsé
  - Locking
- Más, kutatási
  - Software/hardware transact ory (STM/HTM).
  - Software-isolated processes (MS Singularity).
- A lock megoldás sokak szerint nem jó, de jobb mint bármi eddig használt megoldás, vagyis ez az elfogadott megoldás. Ismerős helyzet…

„Democracy is the worst form of Government except all those other forms that have been tried from time to time. „

Sir Winston Churchill

Hardware Transactional Memory

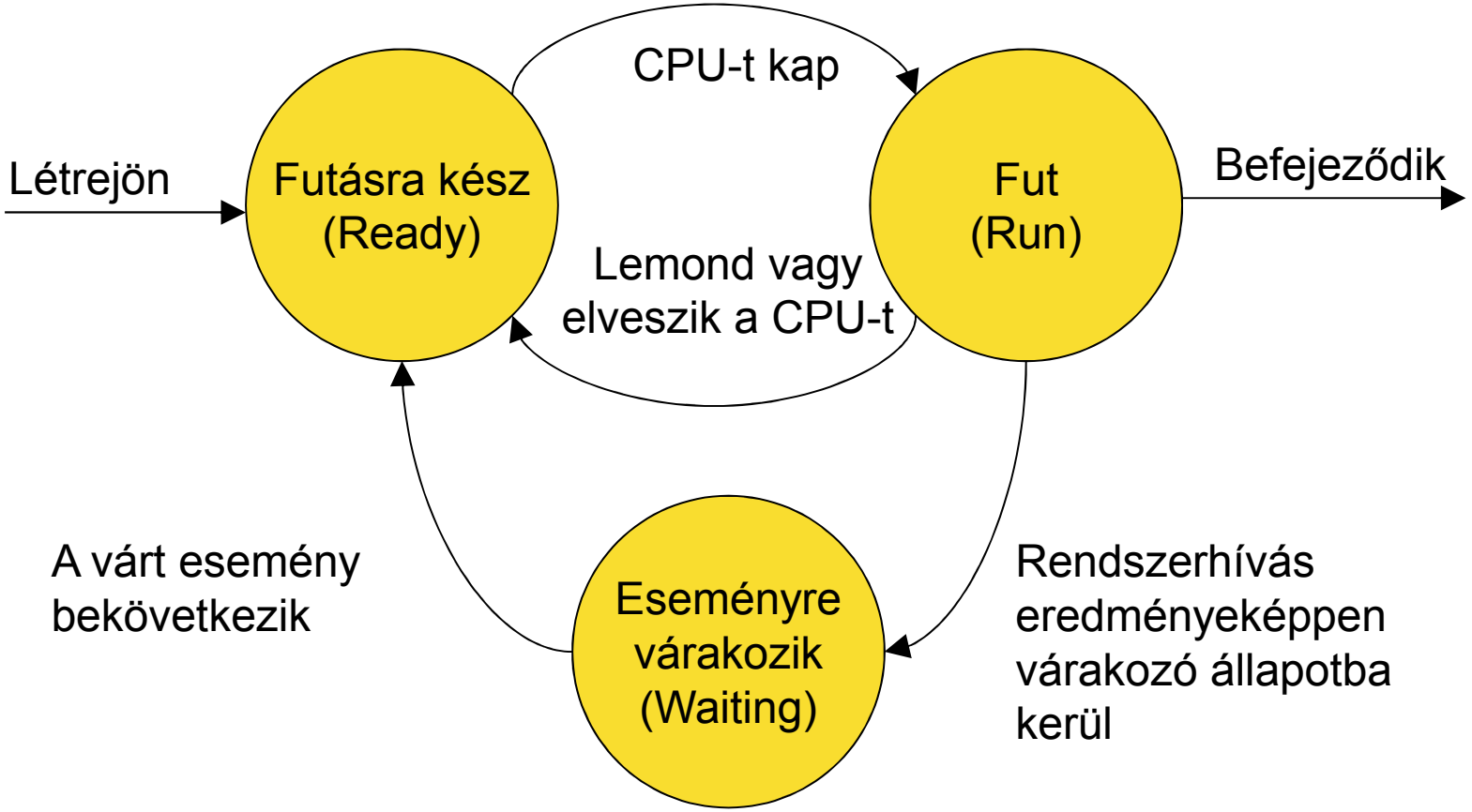
- A HTM azért perspektivikus…
- Első kísérlet az implementációra egy nagyobb gyártótól
  - SUN Rock processor
    - Új utasítások: chkpt, commit
    - Új státusz regiszter: cps
    - A CPU végül nem készült el
- IBM BlueGene/Q szuperszámítógép
- Intel Haswell (2013 környékén fog megjelenni)
  - Transactional Synchronization Extensions (TSX)
  - 2 működési mód
    - Az egyik visszafelé kompatibilis, a másik már nem

Újrahívhatóság (reentrancy)

- A közös erőforrás problémájának egyfajta kiterjesztett esete egy függvényen/objektumon belül is felléphet, amennyiben ezt a függvényt (metódust) egyszerre többen is meghívhatják.
- Hogyan fordulhat ez elő?
  - Ugyanazt a függvényt hívjuk egy taszkból is és egy megszakítás rutinból is.
  - Az ütemezés preemptív, és ugyanazt a függvényt hívjuk két taszkból is.
- Az újrahívhatóság feltételei:
  - Hosszú lista...
  - Azt kell vizsgálni, hogy az újrahívott függvény/metódusban használt változók, perifériák, függvények/metódusok, stb. közös erőforrásnak minősülnek, és ha azok, akkor azokat közös erőforrásként megfelelően kezeli-e a függvény?
- Példák:
  - PC BIOS hívások nem újrahívhatóak!
  - Preemptív operációs rendszer rendszerhívásai mindig újrahívhatók, de az API függvényeinek lehetnek újrahívható (thread safe) és nem újrahívható (gyorsabb a kölcsönös kizárás megvalósításának elmaradása miatt) verziói is.
  - Egyéb programkönyvtárak? Mi a helyzet a kedvenc JPEG kezelő könyvtáraddal?

Várakozás közös erőforrásra

- A más feladat által használt közös erőforrásra is „eseményre várakozik” állapotban várakozik a feladat.
- Az OS nyújt a közös erőforrások védelmére szolgáltatásokat.



Lock-olás és az ütemező, szabad erőforrás

- A közös erőforrást használni kívánó feladat meg próbálja szerezni az erőforrást egy OS hívással.
  - Az erőforrás szabad (nem használt):
    - Az erőforrást az OS lefoglalja a feladat számára.
    - Visszatér a feladathoz, vagyis az „fut” vagy „futásra kész” állapotba kerül (az OS-től függ), majd CPU megkapása után a hívásból visszatérve fut tovább.
  - Használja az erőforrást.
  - A használat végén felszabadítja azt egy OS hívással.
    - Lásd következő fólia.

Lock-olás és az ütemező, foglalt erőforrás

- A közös erőforrást használni kívánó feladat meg próbálja szerezni az erőforrást egy OS hívással.
  - Az erőforrás foglalt (használatban van):
    - A feladat az adott erőforrásra váró feladatok várakozási sorába (tipikusan FIFO), „eseményre vár” állapotba kerül.
    - Fut az ütemező a következő futó feladat kiválasztására.
  - Ha egy másik feladat később felszabadítja az erőforrást:
    - Az erőforrás felszabadítása során az erőforrásra váró feladatok közül kiválasztásra kerül az erőforrást megkapó feladat (tipikusan FIFO az ütemezés)
    - Annak számára lefoglalja az erőforrást.
    - Majd „futásra kész” állapotba helyezi azt.
    - Fut az ütemező, ami új futó feladatot talál

Részletesség

- A lock-olás részletessége (Fine or course grained locking)
  - A kölcsönös kizárás megvalósítása erőforrás használattal jár (CPU), minimalizálni kell a használatát.
    - Túl sok rendszerhívás jelentős overhead-del jár.
  - Viszont a túl nagy egységekben végzett kölcsönös kizárás is erőforrás pazarlással jár.
    - A rendszerben „nehezebb” futásra kész feladatot találni.
  - Pl. Periféria egy buszon (félvezető hőmérő I^2C buszon)
    - Működés: mérés indítás, 200ms mérési idő, mérési eredmény kiolvasása.
    - A teljes mérés idejére megvalósított kölcsönös kizárás a buszra: Hosszú ideig nem érhető el a busz más célra sem.
    - A mérés indításra és a mérési eredmény kiolvasására külön-külön valósítsuk meg a kölcsönös kizárást a buszon: Sok OS hívás, mivel a kölcsönös kizárás OS hívást jelent.

Hibák 1.

- Versenyhelyzet (race condition):
  - A párhuzamos program lefutása során a közös erőforrás helytelen használata miatt a közös erőforrás vagy az azt használó program nem megfelelő (hibás) állapotba kerül.
  - Ilyen volt a korábbi összetett adattípusos példa.
  - A hibás állapot részletei erősen függenek azt használó szekvenciális részfeladatok lefutási sorrendjétől.
- Kiéheztetés (starvation):
  - Ha a párhuzamos rendszer hibás működése miatt egy feladat soha nem jut hozzá a működéséhez szükséges erőforrásokhoz, akkor ki van éheztetve (nem tud futni).
  - Nem csak a CPU-ra, de más közös erőforrásokra is felmerülhet.
    - CPU-ra “Futásra kész” állapotban vár, az összes többi erőforrásra „Eseményre várakozik” állapotban vár

Hibák 2.

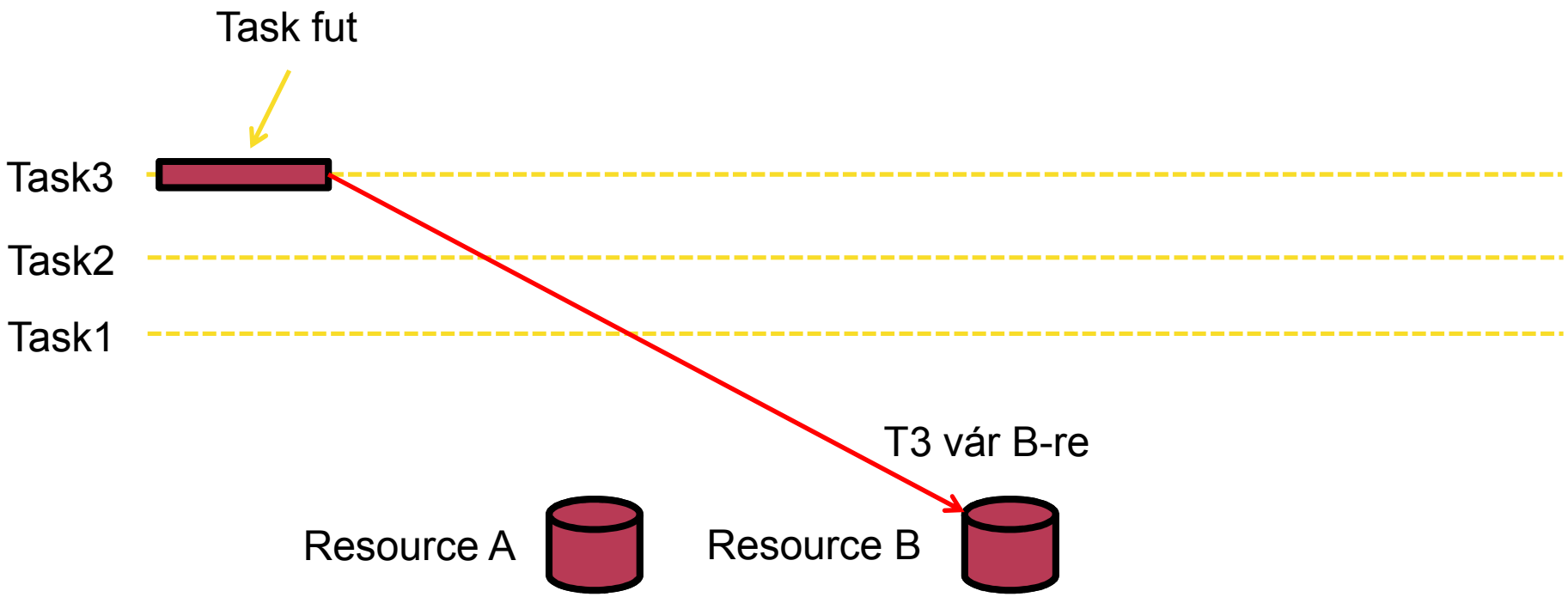
- Holtpont:
  - A közös erőforrások hibás beállítása vagy használata miatt a rendszerben a részfeladatok egymásra várnak.
  - Nincs futásra kész folyamat.
  - Nem jöhet létre belső esemény.
  - A rendszer nem tud előrelépni.
- Livelock:
  - Példa: „Két kedves ember összetalálkozik az ajtóban.”
  - Többnyire a hibás holtpont feloldás eredménye.
  - A rendszer folyamatosan dolgozik, de nem lép előre.

Hibák 3. Prioritás inverzió

- Prioritás inverzió (priority inversion):
  - Prioritásos ütemezőkben fordulhat elő, de az erőforrás használattal is összefügg.
  - A legegyszerűbb esetének előfordulásához kell:
    - 3 feladat, különböző statikus prioritással,
    - Egy közös erőforrás, amelyet a 3 feladat közül a legmagasabb és a legalacsonyabb is használni kíván.
    - A közepes prioritású feladatnak CPU intenzívnek kell lennie.
  - Kedvezőbb esetben csak a rendszer teljesítménye csökkent, a válaszidők nőnek. Valós idejű rendszer????
  - Rosszabb esetben kiéheztetés, vagy akár holtpont is lehet a prioritás inverzió eredménye.
  - Klasszikus példa: Mars Pathfinder 1997...

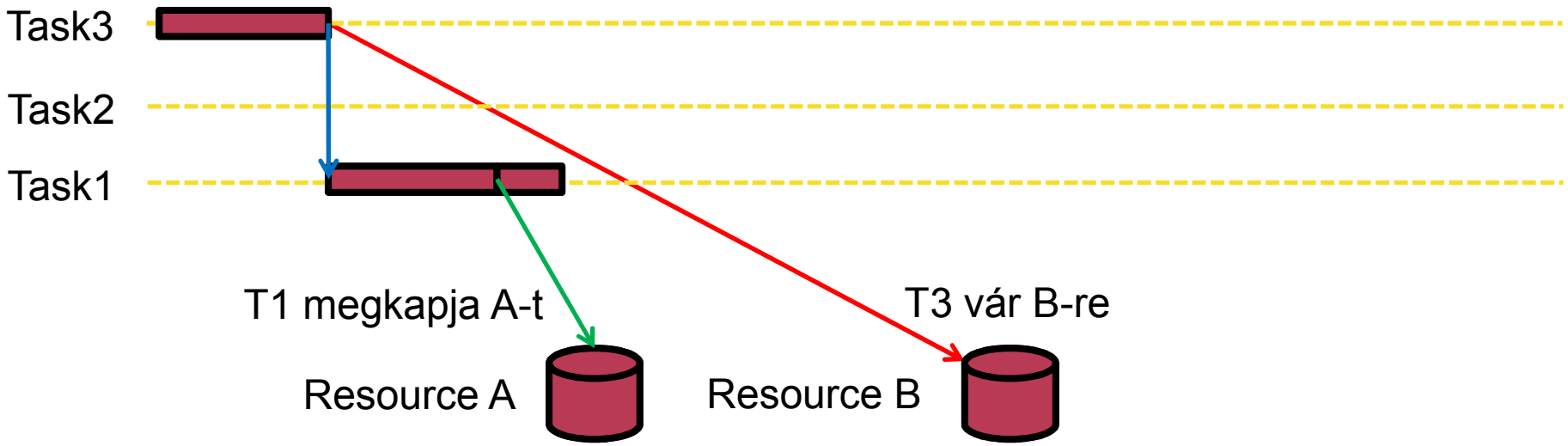
Prioritás inverzió példa 1.

- Lépések sorozata:
  - Task3 magas prioritású feladat valamilyen eseményre vár (de nem A erőforrás felszabadulására). Pl. B erőforrásra…



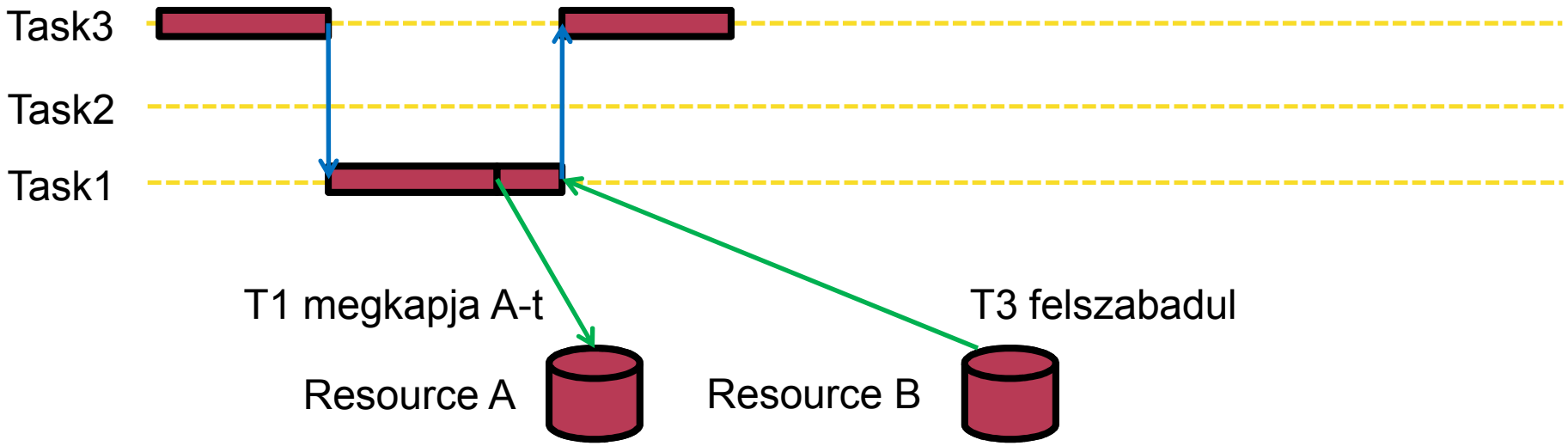
Prioritás inverzió példa 2.

- Lépések sorozata:
  - Task1 alacsony prioritású feladat fut, és megszerzi az A erőforrást, és azt használva fut tovább.



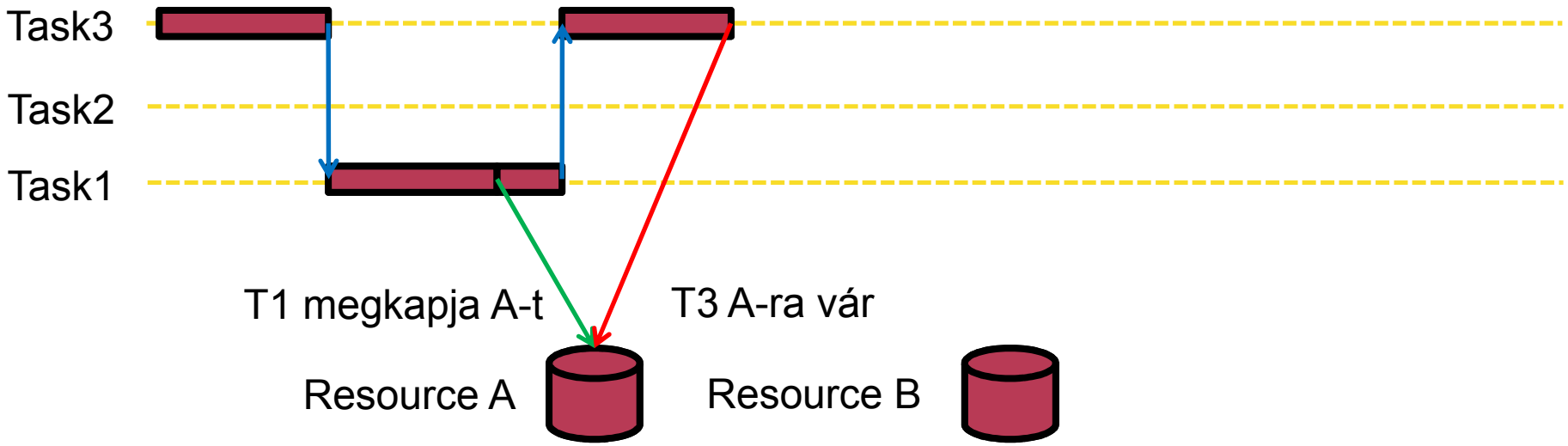
Prioritás inverzió példa 3.

- Lépések sorozata:
  - Task3 magas prioritású feladat által várt esemény megérkezik (B felszabadul), futni kezd (preemptív ütemezés).



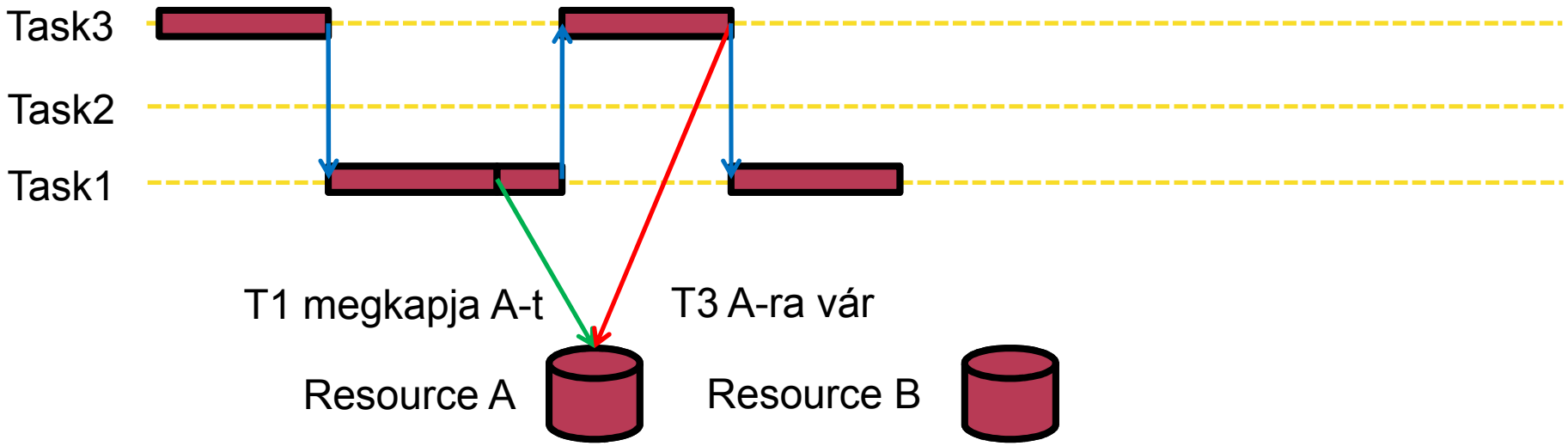
Prioritás inverzió példa 4.

- Lépések sorozata:
  - Task3 fut, majd használni kívánja az A erőforrást. Mivel A foglalt, ezért várakozni kezd (lényegében T1-re vár).



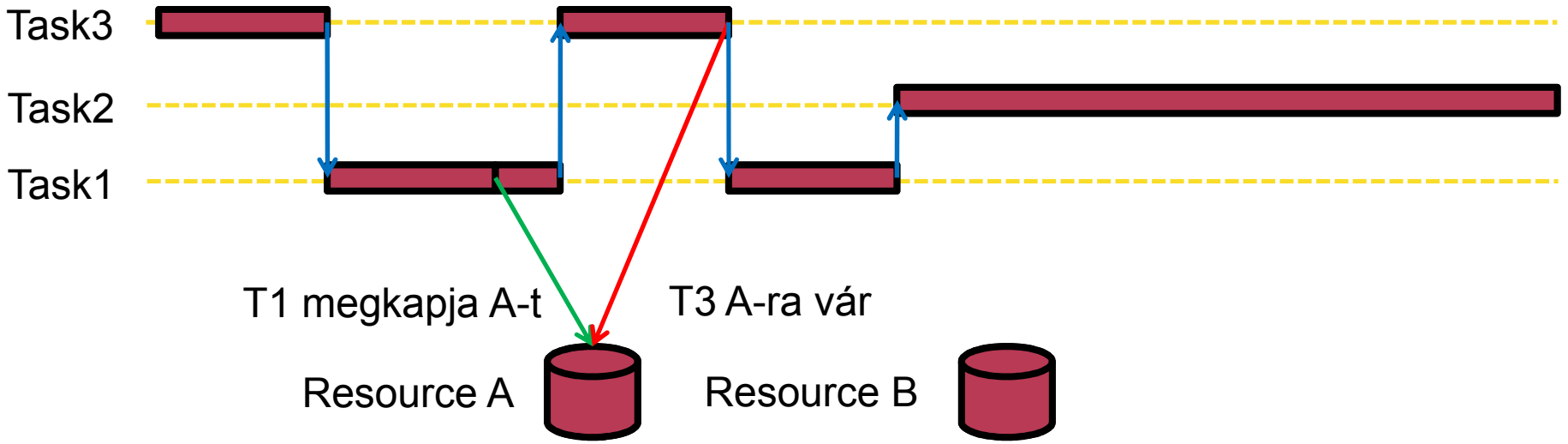
Prioritás inverzió példa 5.

- Lépések sorozata:
  - Task1 ismét tud futni (nincs magasabb prioritású feladat), használja A-t.



Prioritás inverzió példa 6.

- Lépések sorozata:
  - Task2 közepes prioritású feladat futásra kész állapotba kerül, és mivel magasabb prioritású, futó állapotba kerül, és hosszú (akár végtelen) ideig CPU intenzív feladatokat hajt végre (I/O burst nélkül).



Prioritás inverzió példa 7.

- Eredmény:
  - Task3 nem tud továbblépni, hiszen eseményre vár (A erőforrás felszabadulására).
  - Task1 nem tud továbblépni, hiszen CPU-ra vár.
  - Task2 (TaskX) nem foglalkozik az A erőforrással, és amíg intenzíven használja/használják a CPU-t:
    - Task1 nem tudja befejezni a munkáját és felszabadítani az A erőforrást.
    - Amíg A erőforrás nem szabadul fel, Task3 nem tud futni, és végrehajtani a magas prioritású feladatát.
- Task3 magas prioritású feladatot alacsonyabb prioritásúak nem hagynak futni... ☹, 👎, 💣, ☠
- Ez egy egyszerű eset, ennél összetettebb módon is előállhat ez a helyzet!

Megoldás

- Prioritás öröklés (Priority inheritance, PI):
  - Az alacsony prioritású feladat megörökli az általa kölcsönös kizárással feltartott feladat prioritását a kritikus szakaszából való kilépéséig.
  - Csak részben oldja meg a problémát.
- Prioritás plafon (Priority ceiling, PC):
  - Majdnem ugyan az, de az adott közös erőforrást használó legnagyobb prioritású feladat prioritását örökli meg (ami lehet nagyobb mint az éppen feltartott feladat).
  - Az adott erőforrást máskor használó többi feladat sem tud futni (ha esetleg azok is CPU intenzívvé „válnak”).
  - Az alacsony prioritású feladat akadályoztatás nélkül le tud futni.
- Modern beágyazott OS-ekben választhatóak...
  - None, PI, PC (ha preemptív ütemezőt választunk)

Megoldás

- Pri
  - Így javították meg a Mars Pathfinder-t is. Lényegében a prioritás öröklést beállították , és újrafordították az OS-t, aztán letöltötték a patch-et (diff) MARS-on lévő HW-ba...
- Miért nem használták egyből? : Nem tudták hogy van ilyen jelenség, a VxWorks-ben meg nem ez volt a default (overhead-je nagyobb).
- Miért nem fedezték fel tesztelés közben? : Mert soha nem teszteltek valóshoz hasonló terhelési körülmények között. A teljes adatgyűjtés soha nem ment a földön, csak részrendszerek (gratulálok…).
  - futni.
- Modern beágyazott OS-ekben vála…k...
  - None, PI, PC (ha preemptív ütemez… használtunk)

Prioritás Inverzió más szempontból

- Sokak szerint a prioritás inverzió alapvetően egy rendszertervezési hiba eredménye:
  - Nem speciális protokollokkal (PI, PC) kell megoldani, hanem jól meg kell tervezni a rendszert (prioritások kiosztása, kölcsönös kizárás, kölcsönös kizárás időtartama, stb.).
  - "Against Priority Inheritance" by Victor Yodaiken
    - A RTLinux kitalálója és fő fejlesztője…
    - http://www.linuxdevices.com/articles/AT7168794919.html

Hogyan lock-olunk

- Passzív várakozás az OS szolgáltatásainak felhasználásával.
- Aktív várakozás.

Lock feloldására várakozás passzívan

- Sleeplock, blocking call, etc.
  - Ütemező által karbantartott várakozási sorok (beszéltünk róla).
    - Ha az erőforrás nem lock-olt.
      - Megkapja az erőforrást a feladat lezárva és fut tovább.
    - Ha az erőforrás lock-olt.
      - A feladat megy az erőforráshoz tartozó várakozási sorba, a futásra kész feladatok közül egy futó állapotba kerül.
      - Ha az erőforrás felszabadul, akkor az erőforráshoz tartozó sor elején álló megkapja az erőforrást lezárva, és futásra kész állapotba kerül.
  - Erőforrás-takarékos, de van overhead-je.
    - OS fut, ütemezés és kontextus váltás.
  - Utána csak futásra kész sorba kerül a részfeladat, pontos időzítés nehezen megoldható (a futás kezdete érdekes). Alsó korlátot ad.
  - A processzor aludhat, ha nincs feladat:
    - Aztán HW IT-re felébred (belső esemény nem történhet).

Lock feloldására várakozás aktívan

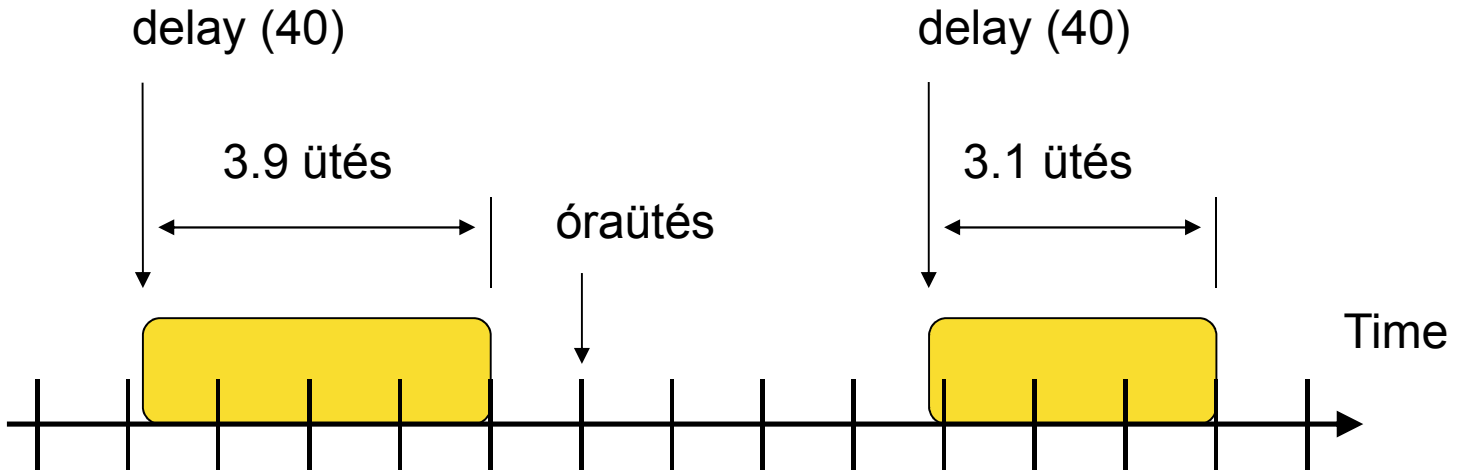
- Livelock, spinlock, busy wait, spinning:
  - Aktív várakozás az erőforrás felszabadulására és megszerzésére (CPU erőforrás pazarlás).
    - Ha aktívan vár egy részfeladat, akkor a többi részfeladat hogyan tudja „előidézni” a várakozást megszüntető változást (eseményt) a rendszerben?
    - Nem tudnak futni, az aktívan várakozó fut!
    - Külső HW megszakítás és/vagy több CPU esetén ez nem probléma.
  - Fogyasztás is nő, hiszen a CPU folyamatosan fut, nem tud aludni (ha nincs éppen feladat).
  - Compiler optimalizáció kiszedheti/eltávolíthatja a kódot.
  - Függ a CPU sebességétől (ha adott időt akarunk várni):
    - A kód hordozhatósága rossz. Az ütemező befolyásolja, alsó korlát csak.
    - Mérjük meg : Linux Bogomips : "bogus" MIPS.
    - Mi van, ha a CPU változtatja az órajelét?

Akkor hogyan várakozzunk?

- Rövididejű várakozáshoz a spinlock elkerülhetetlen:
  - ***Garantáltan rövid idejű*** kölcsönös kizárási problémák kezelésére.
  - Periféria kezelés során, kb. n* 1μs vagy az alatti időtartamú időzítésre.
- HW Timer is használható adott időtartamú várakozásra, bár többnyire limitált számú Timer periféria van a MCU-ban.
  - IT overhead megjelenik.
  - Az IT késleltetésnél csak legalább egy nagyságrenddel nagyobb várakozás indokolható az overhead miatt.
  - Ez kombinálható az aktív várakozással („kevésbé aktív várakozás”).
- Nehéz jó kompromisszumot találni.
  - Sleeplock (ütemező) – Dedikált Timer IT – spinlock ?
  - A pontos határok sok szemponttól függenek.
- Az ütemező nem használ a feladatok ütemezése során spinlock jellegű hívásokat (egyértelműen sleeplock alapú).
- Maga az OS kernel viszont gyakran használhat (Pl. SMP Linux).

Rokon probléma: Adott idejű várakozás

- Az OS-ben van egy beépített szoftver timer szolgáltatás (a timeslice v. system tick-ből levezetve)
- Milyen pontos az OS SW timer?
  - pl. delay(N), ahol N a várakozás időtartama ms vagy μs-ban.
  - Hívás és futásra kész állapotba kerülés között eltelt időnek a felső korlátját adja meg többnyire.
    - Ezek után az ütemezőtől függ a tényleges várakozás ideje. Ebben az értelemben alsó korlát.
  - Az OS rendszeróra óraütésnek a felbontásával dolgozik.
  - Példa: 10 ms-es óraütés, 40 ms várakozás



Rokon probléma: Időmérés

- Ha az OS timer felbontása nem elég (1-10-20ms).
- Időintervallum mérés két esemény között:
  - Többnyire μs-os, vagy akár ns-os felbontás kell!
  - Szabadon futó Timer periféria értékének lekérdezése, és különbség képzés.
    - Felbontás programozható.
    - Túlcsordulás esetén IT.
  - Timestamp counter (pl. Pentium Timestamp Counter):
    - A nagyobb processzorokon megtalálható.
      - Adott felbontás (CPUclk, vagy CPUclk/2^n).
      - Pl. 64-bit regiszter az x86-os CPU-kon a Pentium család megjelenése óta.
      - Túlcsordulás nem kerül jelzésre, kis valószínűséggel történik meg a nagy bitszám miatt (4 GHz CPU esetén 53375.99 nap, 145.8 szökőév)
  - Események közötti idő, stb. mérhető vele.
  - Időmérés többprocesszoros/elosztott rendszerben?
    - Ne menjünk bele, külön tárgy lenne (nyitott kutatási téma)…

Eszközök RAM/PRAM modell esetén

- Kölcsönös kizárás megoldására:
  - Lock bit,
  - Szemafor,
  - Kritikus szakasz objektum,
  - Mutex,
  - Stb. (minden OS-ben megvan a megoldás).
- Minden OS-ban hasonló eszközöket fogunk találni.
  - Persze kicsit más lesz a nevük és működésük, lesznek különböző verziók, stb., a dokumentáció olvasása nem kerülhető el…
- Tipikus hibák kivédésére és a kölcsönös kizárási probléma megoldására:
  - Monitor.

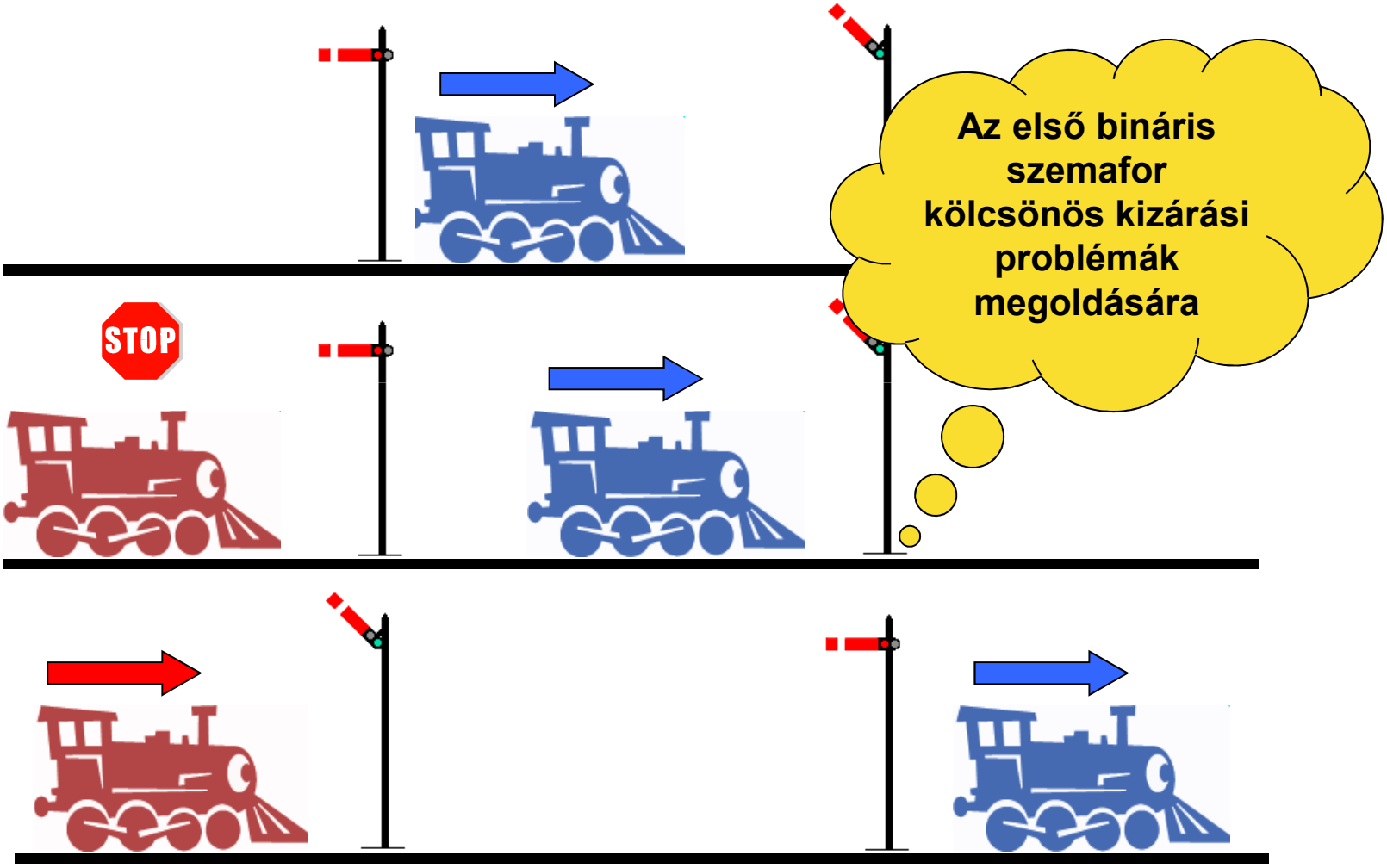
Lock bit

- Legegyszerűbb forma.
  - A védendő erőforráshoz tartozik egy logikai változó (Boolean).
- Lock bit jelentése:
  - Lock bit FALSE ⇨ nem használt az erőforrás.
  - Lock bit TRUE ⇨ használt az erőforrás.
- Belépés művelet:
  - Tesztelés, ha
    - Lock bit == FALSE
      - Lock bit = TRUE (az erőforrás lock-olása)
      - Megyünk tovább (belépünk a kritikus szakaszba)
    - Lock bit == TRUE
      - Aktív várakozás, amíg nem lesz FALSE, és utána Lock-oljuk
- Kilépés művelet:
  - Lock bit = FALSE

Atomi utasítások

- A lock bit alkalmazása során egy súlyos hiba jelenhet meg:
  - IT érkezik a lock bit tesztelése során:
    - A következő utasítás már nem fog lefutni (az ISR-re kerül a vezérlés), vagyis a kritikus szakaszba lépést nem jelezzük, pedig már abban vagyunk.
    - Az IT-ben vagy az utána futó más részfeladatokban az erőforrás szabadnak látszik.
    - A védett erőforrás inkonzisztens állapotba kerülhet.
- Megoldás:
  - IT tiltás a teszt előtt, IT engedélyezés a beállítás után (egy CPU esetén).
  - Test and Set (TAS), vagy hasonló atomikus gépi utasítás.

Szemafor (Semaphore)



Szemafor (EWD ötlete, 60-as évek)

- Bináris és counter típusú szemafor
  - Bináris: egy feladat a kritikus szakaszban.
  - Counter típusú: több feladat a kritikus szakaszban, vagy N darabos erőforrás készletből M darab lefoglalása.
- OS hívás, a bináris szemafor egy magas szintű lock bit.
  - Implementációtól függően:
    - Aktívan várhat (ma már nem jellemző).
    - Várakozó állapotba helyezi a részfeladatot.
- E.W. Dijkstra (EWD 1036?) találta ki a 60-as évek közepén.
- Két művelet értelmezett rajta:
  - Belépés: P(), Wait(), Pend(), …
  - Kilépés: V(), Signal(), Post(), …
  - Sokféle elnevezés, a P()-ről vita van, hogy mit is jelent, a V() a holland kilépés szóból ered.

Példakód...

```
P() {
    while (value <= 0) // Spinlock, not used today
        ; // no operation
    value--;
}

V() {
    value++;
}
```

// Nincs meg az inicializálás és megszüntetés

// Nem atomikus a „test and set” a P()-ben

// P(n) és V(n) jellegű belépés/kilépés nem támogatott...

Szemafor (EWD ötlete, 60-as évek)

- A counter típusú esetén a belépés és kilépés lehet egy számmal paraméterezett (hány egység lép be vagy ki).
- Ha 1-nél több erőforrásra van szükségünk, akkor azokat vagy egyben mind megkapjuk, vagy a töredékeket nem foglaljuk le (más feladatnak szüksége lehet rájuk).
  - Ezért paraméterezhető ebben az esetben a belépés és a kilépés a szükséges erőforrások számával.
  - Az egyenként (pl. for ciklussal lefoglalva őket) N darab erőforrás lefoglalása könnyen versenyhelyzethez, vagy akár holtponthoz vezethet.

Kritikus szakasz objektum és Mutex

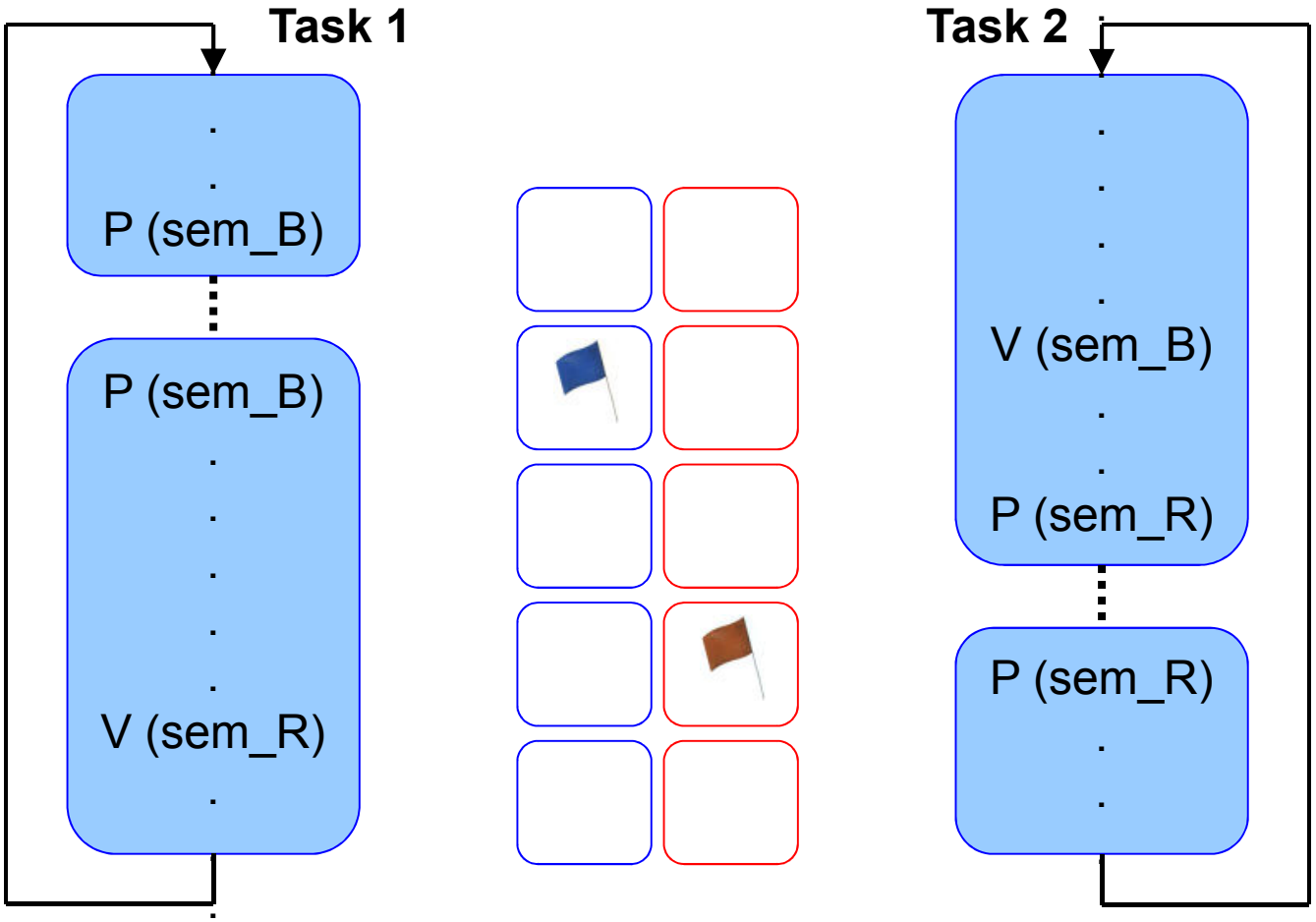
- Lényegében bináris szemafor szerűen működnek.
- Kritikus szakasz objektum.
  - Használata:
    - Létre kell hozni a CriticalSection objektumot.
    - Enter() metódussal lépünk be a kritikus szakaszba.
      - Blokkol (sleeplock), ha már valaki benne van a kritikus szakaszban.
    - Leave() metódussal lépünk ki a kritikus szakaszból.
    - Ha szükséges a CriticalSection objektum megszüntethető.
- Mutex: Mutual Exclusion rövidítése.
  - Használata:
    - Acquire()/WaitOne() függvény vagy metódus.
    - Release() függvény vagy metódus.
  - Multiple read single write mutex (Readers–writer lock)

Miért lock-olunk még?

- Kölcsönös kizárást lezártuk.
- Részfeladatok közötti szinkronizáció visszavezetése kölcsönös kizárásra:
  - Nincs védendő objektum igazából, részfeladatok együttműködése a cél.
- Pl. Randevú (rendezvous).
  - Két vagy több feladat összehangolt végrehajtása
  - Pl. a Coroutine-nál megvalósított termelő-fogyasztó probléma megoldható 2 bináris szemaforral is.
  - Ebben az esetben az operációs rendszer által nyújtott szolgáltatásokat használunk, és a feladatok passzívan várnak egymásra.
  - A szemaforok alapesetben foglaltként vannak inicializálva ebben az esetben.
- Memórián keresztül történő kommunikáció.
  - A kommunikációra használt memória közös erőforrás.

Kétoldalú randevú bináris szemaforral

Kétoldalú randevú szemaforral, B és R foglaltként inicializálva
(bilateral rendezvous)



Kommunikáció: Védett memóriaterület

- Szálak közötti kommunikáció során:
  - Folyamatok így nem tudnak kommunikálni (MMU).
  - A UNIX SystemV shared memory nem valódi osztott memória, valójában OS szolgáltatás!
    - ***Folyamatok közötti kommunikációt tesz lehetővé.***
- Tömb, Struktúra, Objektum.
- Egy vagy kétirányú kommunikáció
  - Egyirányú: A küldő írja, a vevő/vevők kiolvassák belőle.
  - Kétirányú: Minden fél írja és olvassa.
- A kölcsönös kizárást lock-bit, szemafor, mutex, kritikus szakasz objektum, stb. oldja meg.

Lock-olás során elkövetett tipikus hibák

- Belépés/Kilépés elmaradása.
- Többszöri be- vagy kilépés.
- Más erőforrás lefoglalása.
- Az erőforrás indokolatlanul hosszan történő lezárása.
  - Minimális időre kell törekedni.
- Prioritás inverzió.
  - Foglalkoztunk vele.
- Deadlock és livelock.
  - Külön foglalkozunk vele.

Monitor (hibák elkerülésére)

- Lokalizáljuk a lock-olással kapcsolatos feladatokat a közös erőforrást körülvevő API-val
  - A lock-olás nem szétszórva történik a programban, hanem egyetlen, a közös erőforráshoz szorosan tartozó programrészletben.
  - A megvalósítás lehet automatikus, például nyelvi szinten (pl. JAVA, C#).
    - A compiler valósítja meg a kölcsönös kizárást biztosító konstrukciókat (pl. szemafor vagy mutex tényleges alkalmazásával).
  - Kézzel is készíthetünk hasonló konstrukciót, pl. egy védett objektumot hozhatunk létre, amely a nyilvános metódusaiban elvégzi a lock-olást, és elrejti a közös erőforrást.

Monitor fejlődése

- Hoare és Mesa szemantika (működés/jelentés).
  - Charles Antony Richard Hoare.
    - Ötlet és részletes elméleti alapok kidolgozása.
  - Mesa programozási nyelv.
    - Xerox PARC (Ethernet, grafikus felület, lézerprinter, stb.).

Hoare és Mesa szemantika

- Hoare szemantika:
  - Azonnal az erőforrást megszerző részfeladat fut.
    - Erőforrás igényes és nehéz megvalósítani.
    - Nem kompatibilis a preemptív ütemezőkkel.
- Mesa szemantika:
  - A futásra kész részfeladatok közé kerül, és később az ütemező futtatja.
  - Vannak vele gondok, de azok megoldhatóak (most nem megyünk bele).
  - "notifyAll" vagy "broadcast" üzenetek küldése is lehetséges.
    - Minden adott eseményre váró futásra kész állapotba kerül.

JAVA példa

- A „synchronized” blokk

```
synchronized (object) {
            // az adott „object”-re biztosítva van
            // a kölcsönös kizárás a blokkon belül
        }
```

- A „synchronized” kulcsszó

```
synchronized void myMethod() {
            // A metódushoz tartozó objektumra
            // biztosít kölcsönös kizárást
}
```

C# esetén : lock block hasonló…