Beágyazott információs rendszerek

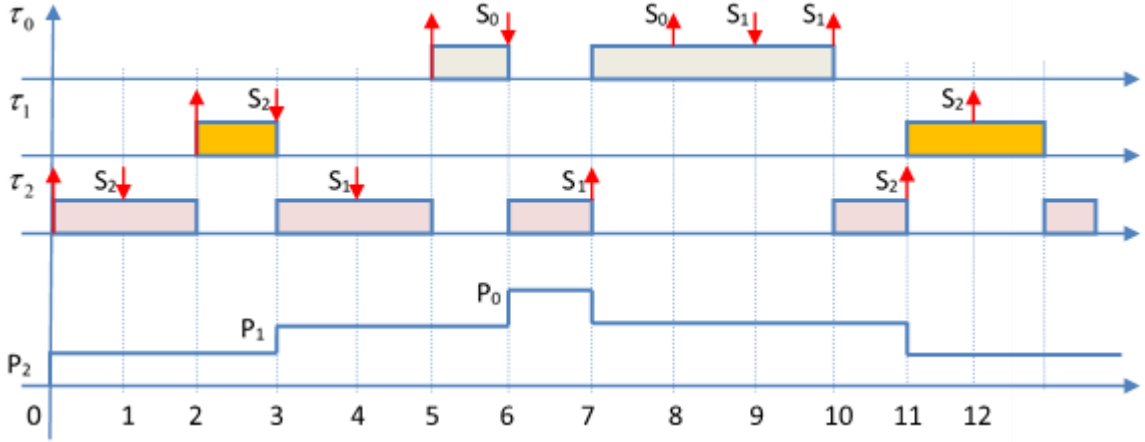
4. Időmérés, időszolgáltatás, óra-szinkronizáció

2020. október 14.

Példa: A futtatandó taszkok csökkenő prioritású sorrendben: $\boldsymbol{\tau_0, \tau_1, \tau_2}$.

A prioritásaik: P_0, P_1 és P_2. Az erőforrásokat S_0, S_1 és S_2 szemaforok őrzik. Prioritás plafonjaik:

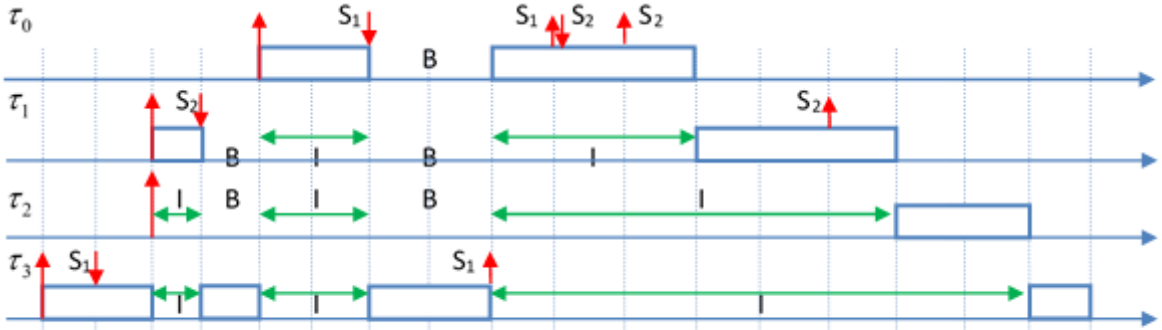
$C(S_0) = P_0,\ C(S_1) = P_0,\ C(S_2) = P_1.$



A PCP hatására $\boldsymbol{\tau_0}$ annak ellenére blokkolódik az S_0 szemaforral védett erőforrás használata előtt, hogy az erőforrás szabad!
Ennek az a kiváltó oka, hogy a $\boldsymbol{\tau_2}$ task a $\boldsymbol{\tau_0}$-val azonos prioritás plafonú S_1 szemaforral védett kritikus szakaszban tartózkodik.

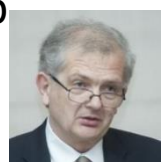
Példa: A futtatandó taszkok csökkenő prioritású sorrendben: $\boldsymbol{\tau_0, \tau_1, \tau_2, \tau_3}$. A prioritásaik: P_0, P_1, P_2 és P_3. Az erőforrásokat S_1 és S_2 szemaforok őrzik. Prioritás plafonjaik:

$C(S_1) = P_0,\ C(S_2) = P_0.$



Az ábrán nyomon követhető a protokoll működése.
I az interferencia intervallumokat, B-vel pedig a blokkolási intervallumokat jelöli.

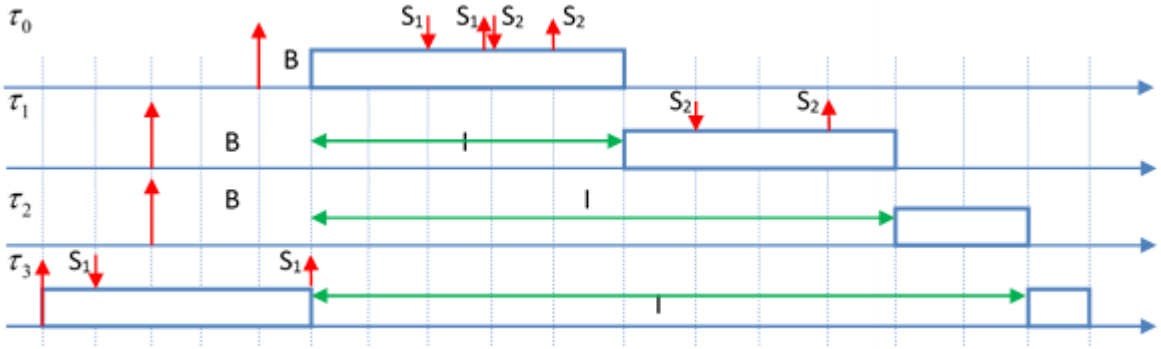
Ezeknek az összege adja az adott taszk tényleges blokkolási idejét, aminek a maximuma a legkedvezőtlenebb esetben a $\boldsymbol{\tau_3}$ taszk kritikus szakaszának processzoridő igényével egyezik meg.



Emlékeztető

Azonnali prioritás felső-határ (plafon) protokoll (Immediate Priority Ceiling Protocol, IPCP):

A protokoll lényege, hogy a taszkok a kritikus szakaszba lépéskor **azonnal** a kritikus szakaszt védő szemafor **prioritás plafonjának megfelelő** dinamikus **prioritást** kapnak!



Ennek értelmében az ábrán a τ_3 taszk a kritikus szakaszba lépve **azonnal** P_0 prioritást kap, és egészen a **kritikus szakasz elhagyásáig** azon marad.

Az **IPCP** protokoll könnyebben implementálható, mint a PCP, látható módon kevesebb a taszk-váltás, és ennek következtében a futtatási környezet-váltás.

A szemaforokat nem kell implementálni, mert mindig szabad állapotúak!

Érdemes megfigyelni, hogy ebben a példában - az **IPCP** alkalmazása esetén - a legnagyobb prioritású taszk válaszideje egy időegységgel csökkent.

Az IPCP elnevezése a **POSIX** szabványban **Priority Protect Protocol,** a **Real-Time Java-ban** pedig **Priority Ceiling Emulation.**

3. Memória menedzsment

A beágyazott rendszerek jelentős részénél **nem számíthatunk** arra, hogy az eszköz időről-időre alaphelyzetbe kerül (reset-elődik), és a programfutások **káros mellékhatásai** ezzel **eliminálódnak**. Ezért minden esetben **úgy kell terveznünk**, hogy az alkalmazás futásával párhuzamosan az **erőforrások** teljesítőképessége **ne degradálódjon**.

Emlékeztető

- **Statikus memória allokáció:**
minden fixen kiosztva. **Előny:** egy csomó hibaforrás kizárva.
Hátrány: nem alkalmazható rekurzió és semmi olyasmi, ami az újrahívhatóságot igényli.

- **Verem (stack) alapú menedzsment:** Sok program esetében fordítási időben nem mondható meg a szükséges **stack** méret. Nem tudjuk ugyanis, hogy például (közel) egy időben hány megszakítás-kiszolgálás válik szükségessé. Ilyenkor tesztelés szükséges. Ehhez adott mintával fel kell tölteni az előre beállított méretű **stack** területet, majd a teszt-futtatás után rákeresni, hogy a program meddig használta, azaz meddig írta felül a betöltött mintázatot.

Ez az ún. **watermark** meghatározás. Sok RTOS támogatja. Az ellenőrzést célszerű lehet összekötni a watchdog timer indításával.

Ökölszabály: a **stack** méretét 50%-kal nagyobbra kell választani, mint a tesztelések során tapasztalt legnagyobb (worst case) igény.

- **Halom (heap) alapú menedzsment:** A C a $malloc( )$ és $free( )$ függvényekkel kezeli, ami a programozóra nagy felelősséget hárít.

Az egyik legnehezebb probléma, amelyet az alkalmazói program szintjén nem is lehet kezelni, a memória **feldarabolódás/tördelődés** problémája (**fragmentation**).

Ez azáltal jön létre, hogy a felszabadított blokkoknál kisebbek kérése esetén **olyan** (kicsi) **memória darabok maradnak**, amelyek sosem kerülnek felhasználásra.
Ilyenkor egyrészt nincs garancia arra, hogy nem fogy el a memória a töredék darabok miatt, másrészt a nyilvántartott szabad memóriadarabok száma nő, aminek következtében nő a memória-keresés végrehajtási ideje.

A másik probléma a memória "**zárvány**" (vagy más szóval elfolyás (**leakage**)),

Emlékeztető

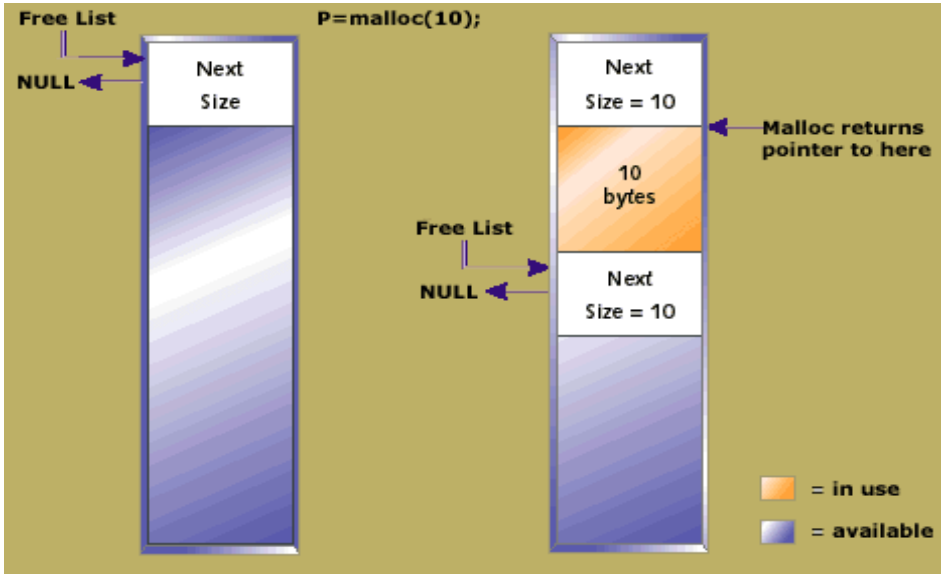
amely a következők miatt jöhet létre: a kódolás egy adott pontján **a programozó elbizonytalanodhat**, vajon egy adott memória blokkra szükség van-e még?

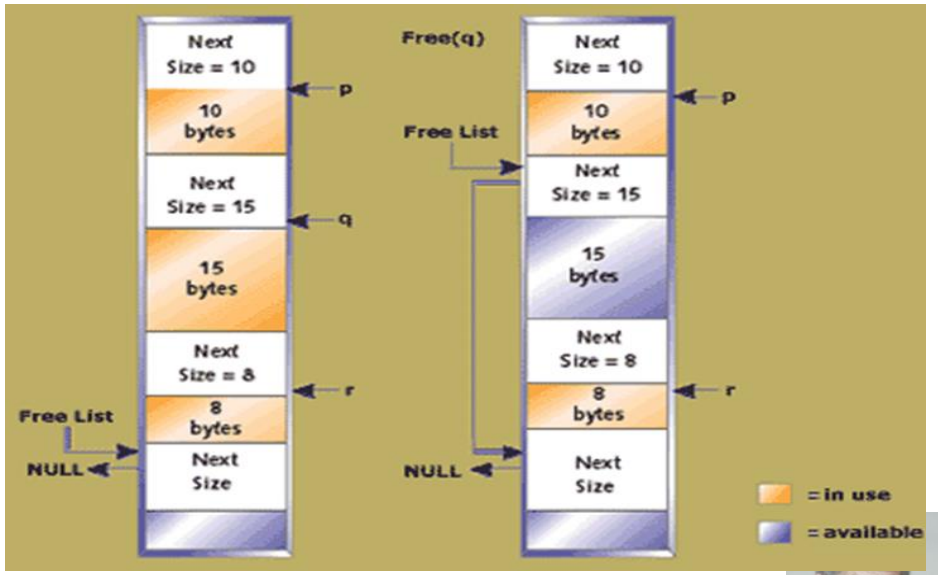
Ha **felszabadítja**, de továbbra is használja, például egy, az ugyanarra a blokkra mutató második pointer segítségével, akkor a program jól működhet mindaddig, amíg az adott memória területet a program egy másik része le nem foglalja.

Ezt követően a program két része felül fogja írni egymás adatait!

Ha nem szabadítja fel, például azon az alapon, hogy még szükség lehet rá, akkor előfordulhat, hogy soha többet nem lesz rá lehetősége, mert a rámutató pointerek időközben érvényüket veszítették, vagy másra használta fel őket.

Ettől maga program még jó marad, de ha rendszeresen meghívjuk ezt a program-részletet, akkor a zárványok száma **állandóan nőni fog**, aminek következtében a program **futási ideje megnő.**





4. Időmérés, időszolgáltatás, óra-szinkronizáció

Időmérés eszközei és módszerei:

(1) ***Időmérés elektronikus számlálóval:***

A forrás által generált ún. "kapuidő" maga a mérendő időtartam.

A mérés kezdetekor nullázott számláló a kapuidő alatt beérkezett impulzusokat számlálja.

$T_x \cong \frac{N}{f_0}$, ahol *N* a számláló tartalma, f_o pedig az órajel frekvencia.

Az időmérés (worst-case) relatív hibája: $\left|\frac{\Delta T_x}{T_x}\right| \cong \left|\frac{1}{N}\right| + \left|\frac{\Delta f_0}{f_0}\right|$

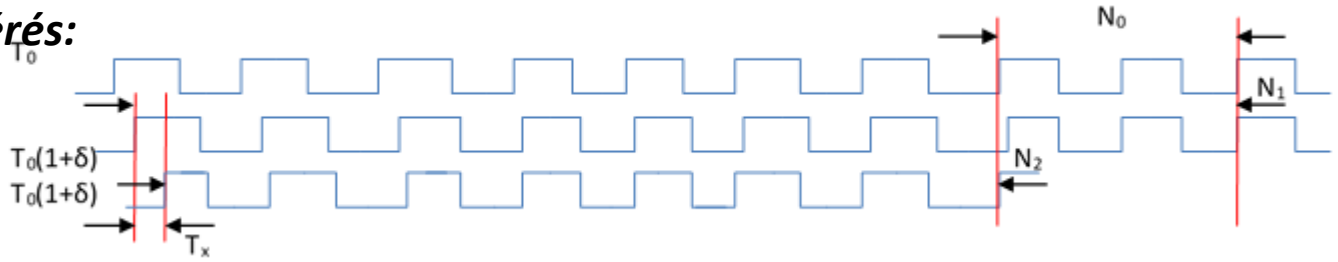
Ezt az összefüggést a teljes differenciál felírásából kiindulva származtatjuk:

$$dT_x = \frac{\partial T_x}{\partial N} dN + \frac{\partial T_x}{\partial f_0} df_0 = \frac{1}{f_0} dN - \frac{N}{f_0^2} df_0$$, amit elosztva $T_x = \frac{N}{f_0}$-szel $\frac{dT_x}{T_x} = \frac{dN}{N} - \frac{df_0}{f_0}$

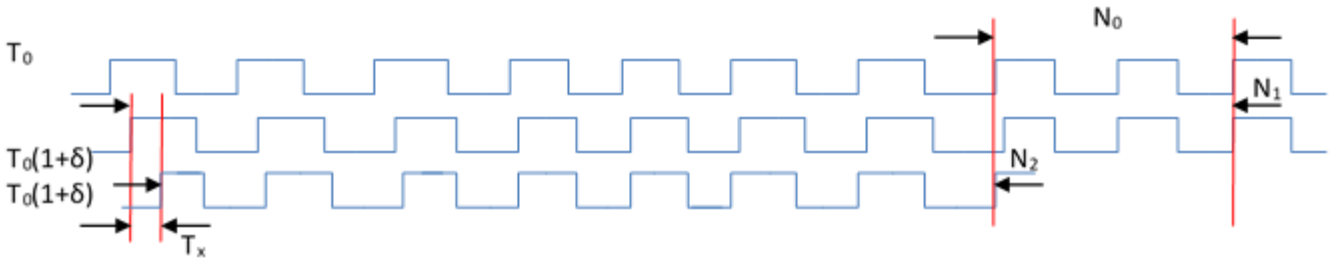
Mivel a megváltozások előjelét nem ismerjük, ezért legtöbbször a relatív megváltozások abszolút értékét írjuk fel a legkedvezőtlenebb esetre.

(2) ***Kettős nóniuszos időmérés:***

A mérendő időtartam kezdete és vége egy-egy $T_0(1+\delta)$ periódusidejű, kvarcpontosságú órát indít.



Ezek jelét egy szabadon futó T_0 periódusidejű, kvarcpontosságú óra jelével hasonlítjuk össze, figyelve a felfutó élek egybeesését.



A mérendő időtartam **kezdetétől** az első koincidenciáig eltelt idő $N_1 T_0(1+\delta)$,

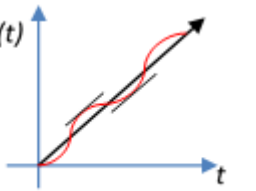
a mérendő időtartam **végétől** az első koincidenciáig eltelt idő $N_2 T_0(1+\delta)$,

a két koincidencia között eltelt idő pedig $N_0 T_0$. Ezzel $T_x = T_0[\pm N_0 + (N_1 - N_2)(1+\delta)]$

ahol az N_0 előtti előjelet a két koincidencia időbeni sorrendje határozza meg.

Ha $\boldsymbol{T_0 = 5\ nsec}$ és $\boldsymbol{\delta = 0.004}$, akkor a legkisebb, még mérhető időtartam $\boldsymbol{20\ psec}$!

Az órák, mint a valós idő adott pontosságú forrásai:



Az idő forrását **órának** nevezzük. A k-jelű óra a valós idő egy $C_k(t)$ függvénye.

Referencia óra: a teljesen pontos óra.

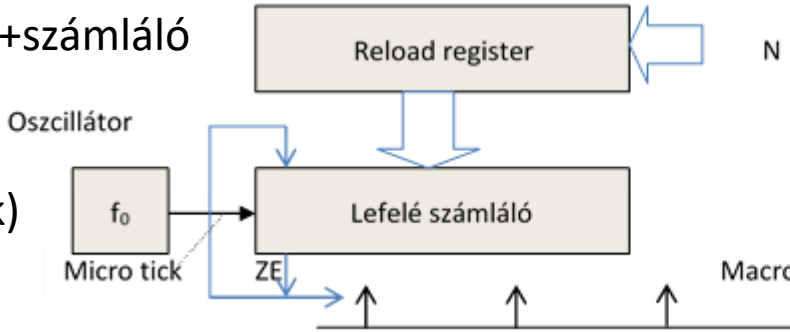
Ha a k-jelű teljesen pontos, akkor $C_k(t) = t; \forall t$

Helyes óra: a k-jelű óra helyes (correct) t_0-ban, ha $C_k\ t_0) = t_0$

Pontos óra: a k-jelű óra pontos (accurate) t_0-ban, ha $\partial C_k(t)/\partial t = 1; t = t_0$

Ha egy óra pontatlan egy adott időpillanatban, akkor azt mondjuk, hogy csúszik.

A fizikai óra: Oszcillátor+számláló felbontóképessége g (g: granularity), mikro-óraütés (microtick)



A kvarc-oszcillátor jelét egy lefelé számláló leosztja, és a nulla (ZE) elérésekor kiad egy impulzust.

A fizikai referencia óra: jele $\boldsymbol{C}$, felbontóképessége g^C. Pl.: 10^{15} óraütés/sec $\rightarrow g^C = 10^{-15}$ sec.

Értéke a nemzetközi idő szabvány szerinti abszolút idő.

Időbélyeg: $\boldsymbol{C}(e)$: az *e* esemény abszolút időbélyege.

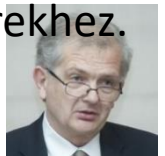
Óra drift: *A k*-jelű fizikai óra két, önkényesen kiválasztott óraütése között eltelt időt a referencia órával megmérjük, és a vizsgált óra által mutatott időkülönbséget viszonyítjuk ennek teljesen pontos értékéhez:

$$drift_k(t_i, t_{i+1}) = \frac{C_k(t_{i+1}) - C_k(t_i)}{C(t_{i+1}) - C(t_i)}$$

Mivel a ***drift*** ideális értéke 1, ezért szokás definiálni a ***drift-mértéket***: $\delta_k = \rho_k = |drift_k - 1|$ formában, ami specifikációs adat az órára, egyhez képest nagyon kis érték (10^{-2}...10^{-7} sec/sec). Előfordul, hogy a szóhasználat ezt nevezi ***drift***-nek, ami a nagyságrendi eltérés miatt nem okoz félreértést. A ***drift mérték*** maga a ***drift*** egytől való eltérésének előjelét nem hordozza. Ha nincs szinkronizáció, akkor az órák a ***drift*** következtében eltérő ütemben haladnak, „elmásznak”.

Ennek súlyos következményei lehetnek! **Példa:** Öböl háború, Dhahran, 1991. február 25.

Egy **Patriot** rendszer elvétett egy **Scud** rakétát.
Egy fizikai óra mintegy **100 óráig** (>4 nap) szinkronizálás nélkül maradt, ez alatt - kvantálási hiba következtében - összeszedett **0.3433 sec** késést, ami **687 méteres** követési hibát okozott a célkövető számításaiban, és ez által a mintegy **1.7 km/sec** sebességgel haladó rakéta kikerült célkövető látóköréből. Következmény: **28 halott, 98 sebesült**! A hiba hátterében az állt, hogy korábban a **Patriot** rendszereket rövidebb működési idő feltételezésével, lényegesen lassabb eszközök ellen fejlesztették, és az Öböl háború idején fejlesztették tovább **Scud** rendszerekhez.



A konkrét tragédiát okozó hibát már február elején felfedezték, február 16-án **a módosított szoftvert ki is adták**, de az nem jutott el az érintett **Patriot** rendszerbe.

Óra ofszet: Tekintsünk két órát azonos felbontóképességgel: $ofszet_{j,k}(t) = \left|C_j(t) - C_k(t)\right|$

***Együttfutás* (precision):** Tekintsünk n órát! Az együttfutás: $\Pi(t) = \underbrace{max}_{\forall 1 \le j,k \le n} \left[ofszet_{j,k}(t)\right]$

A drift miatt ez az idő múlásával nő, ezért kell szinkronizálni.

Ez az ún. "**belső szinkronizáció**", mert az órákat egymáshoz igyekszünk szinkronizálni.

***Pontosság* (accuracy):** a k jelű óra ofszetje a referencia órához képest:

$ofszet_{k,ref}(t) = \left|C_k(t) - C_{ref}(t)\right|$

A drift miatt ez az idő múlásával nő, ezért kell szinkronizálni.

Ez az ún. **"külső szinkronizáció",** mert az órákat a referencia órához igyekszünk szinkronizálni.

Példa: Igaz-e a következő állítás?

Ha minden óra a vizsgált halmazban kívülről szinkronizált ***A* pontossággal**, akkor az óra-együttes belülről is szinkronizált **2*A* együttfutással**.

Az állítás igaz!

Fordítva nyilván nem.

Az idő mérése elosztott rendszerekben

Az eddigiektől eltérően jellegzetesen különböző órákkal; az elosztott rendszerben mindenkinek saját órája van.

Globális idő: az univerzális referencia idő "gyengített" változata.

Tegyük fel, hogy a csomópontokban lévő C_k órák g^k felbontással ketyegnek.

Belülről szinkronizáltak Π együttfutással, azaz tetszőleges j és k párra $\left|C_j(t) - C_k(t)\right| < \Pi$ $\forall t$-re. A **globális idő** az univerzális referencia idővel azonos pontosságú, de durvább felbontású óraként fogható fel, melynek ütései az ún. **makro-ütések**.

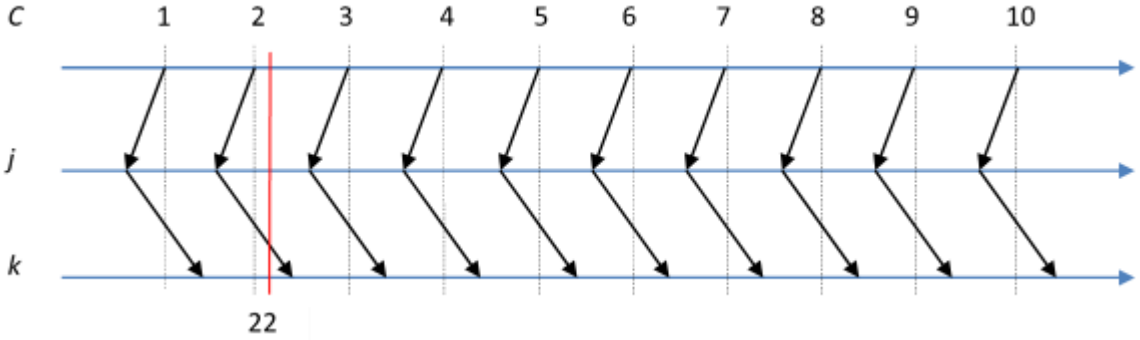
Mikor használható értelmesen a globális idő?

Ha a globális idő felbontása $g > \Pi$, vagyis a szinkronizációs hiba kisebb, mint a felbontóképesség!

Ez egyben azt is jelenti, hogy egy ***e*** esemény időbélyegei a ***j*** és ***k*** csomópontok globális idő értékeivel legfeljebb egyetlen értékben különböznek.

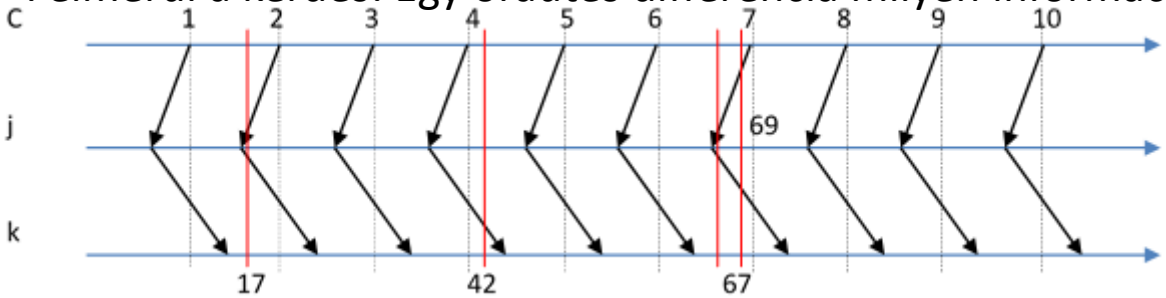
$|C_j(e) - C_k(e)| \leq 1$. Ez a legjobb, amit elérhetünk, mert mindig előfordulhat az a szituáció, hogy először a ***j*** óra üt, majd bekövetkezik az ***e*** esemény, majd üt a ***k*** óra is. Ilyenkor a két óra egy óraütés differenciával bélyegzi az ***e*** eseményt.

Példa (minden makro-ütés tíz mikro-ütésnek felel meg):



Az e: 22 mikro-ütésnél lévő eseményt j: 2-nek, k: 1-nek jelzi

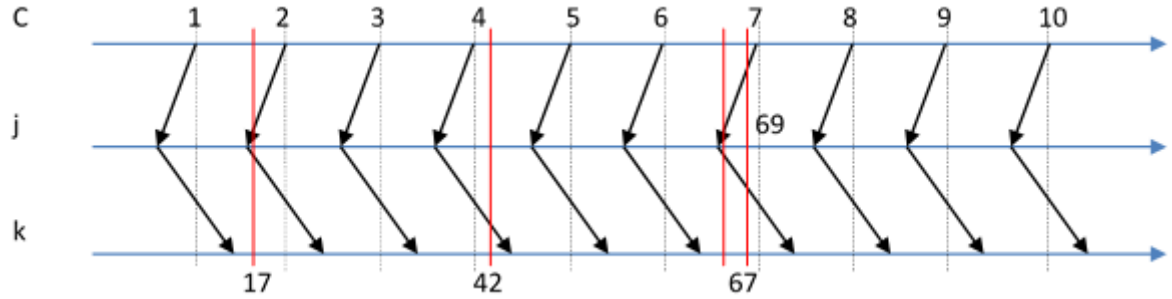
Felmerül a kérdés: Egy óraütés differencia milyen információt hordoz?



Az e: 17 mikro-óraütésnél j: 2, k: 1.

Az e: 42 mikro-óraütésnél j: 4, k: 3.

Ha az e: **42** és az e: **17** események időkülönbségét a C_k és C_j órák különbségeként mérjük, akkor a mérés **1**-et ad a globális időbélyegben annak ellenére, hogy a tényleges különbség **25** mikro-ütés.



Az e: 67 mikro-óraütésnél j: 7, k: 6.
Az e: 69 mikro-óraütésnél j: 7, k: 6.

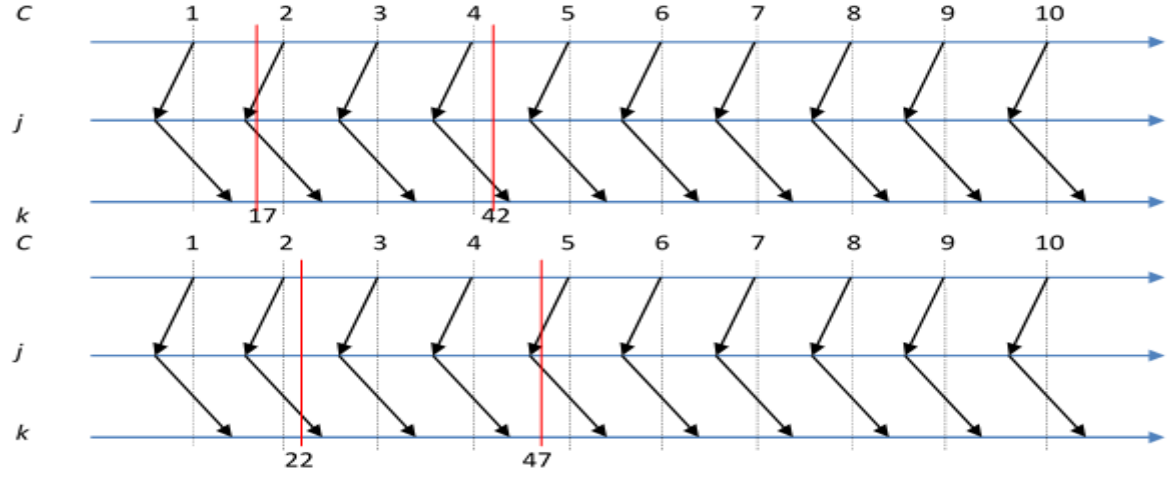
Ha az $\boldsymbol{e}$: **69** és az $\boldsymbol{e}$: **67** események időkülönbségét a C_j és C_k órák különbségeként mérjük, akkor a mérés **1**-et ad a globális időbélyegben annak ellenére, hogy a tényleges különbség **2** mikro-ütés.

Probléma: Az időbeni sorrendet a második esetben nem tudjuk egyértelműen megállapítani a makro-ütések alapján! Az $\boldsymbol{e}$: **67** mikro-óraütésnél $\boldsymbol{j}$: **7**, az $\boldsymbol{e}$: **69** mikro-óraütésnél $\boldsymbol{k}$: **6**!

Állítás: Ha két makro óraütés a differencia, akkor már meg tudjuk mondani az időbeni sorrendet, mert a szinkronizálási és a digitalizálási hiba mindig kisebb, mint 2.

Idő-intervallum mérése: $(d_m - 2g) < d_v < (d_m + 2g)$

ahol d_v az intervallum tényleges értéke, d_m pedig a mért érték.



Az e: 42 és az e: 17 események $C_k - C_j$ időkülönbségét mérve a mérés 1-et ad. (25 mikro-ütés.)

Az e: 47 és az e: 22 események $C_j - C_k$ időkülönbségét mérve a mérés 4-et ad.
(Ez is 25 mikro-ütés!)

Óra rendszerek típusai:

- **Központi óra rendszerek** *(central clock systems)*:
  - egy pontos óra szolgáltatja az időt a teljes rendszer számára,
  - hibatűréshez készenléti (standby) redundanciát alkalmaznak,
  - pontos módszer (ns-en, ms-en belül), költséges
  - a kommunikációs igény alacsony (egy üzenet frissítésenként),
  - a GPS (Global Positioning System) jó példa erre (4 órajelet sugárzó műhold, ns pontosság).
- **Központilag felügyelt óra rendszerek** *(centrally controlled clock systems)*:
  - egy (pontosnak elfogadott) master óra lekérdezi a slave órákat,
  - megmérik az óra eltéréseket és a master korrekciót ír elő a slave számára,
  - ha a master óra meghibásodik, akkor (választási algoritmussal) új master-t választanak,
  - az átviteli időket és a késleltetéseket becsülni kell, mert lényegesen befolyásolják a mért óra eltéréseket,
  - a kommunikációs terhelés erősebb, mint előbb.
- **Elosztott óra rendszerek** *(distributed clock systems)*
  - az óra szempontjából az összes csomópont homogén, ugyanazt az algoritmust futtatja,
  - minden csomópont frissíti az óráját, miután megkapta, és helyesség szempontjából ellenőrizte/becsülte a más órák által kapott időt,
  - a hibatűrés protokoll alapú. Ha egy csomópont kiesik, az nem befolyásolja a többi csomópont működését; észlelik a hibát és figyelmen kívül hagyják a meghibásodott csomópontot,
  - a kommunikációs igény viszonylag nagy, különösen akkor, ha alattomos hibák (pl. bizánci hibák) esetén is a robusztusság követelmény.

Idő normáliák (standardok)

- **Nemzetközi Atomi Idő** *(Temps Atomique Internationale, TAI)*.
  Alapja egy ún. atomóra: Cesium-133 atom által (specifikált módon) kisugárzott frekvencia 9 192 631 770-ed része 1 sec. A TAI által biztosított időskála kronoszkópikus, azaz folytonos.
- **Univerzális idő** vagy **Egyezményes koordinált világidő** (*Universal Time Coordinated, UTC*).
  A Föld és a Nap mozgásából, azaz asztronómiai megfigyelésekből vezették le.
  - 1972-ben lépett a GMT (Greenwich Mean Time) helyébe azzal, hogy a másodperc a TAI szerint értendő.
  - A Föld mozgása enyhén szabálytalan, ezért alkalmanként beszúrnak egy szökő másodpercet.
  - 1958. január elsején a TAI és az UTC (egy megegyezés alapján) ugyanazt mutatta. Azóta az UTC mintegy 40 másodperces eltérést "szedett fel".
  - Az USA mérésügyi hivatalának (National Institute of Standards and Technology: NIST) rövidhullámú rádióadója (hívójele: WWV) folyamatosan ad frekvencia és időjelet 2.5, 5., 10, 15 és 20 MHz frekvencián. A jelek időbeni pontossága ± 1msec, véletlen atmoszférikus ingadozások miatt ± 10msec. (Geostacionárius műholdról ± 0.5msec.)
- **Idő formátum**: legelterjedtebb: **Network Time Protocol** (NTP)
  Ez a formátum 8 bájtot használ, amelyből 4 az UTC másodperceket, 4 pedig a másodperc törtrészét tárolja, az utóbbit 232 psec felbontásban.
  1972 január elsején 00:00:00-kor 2 272 060 800.0 került a 8-bájtos számlálóba, ami az 1900. január elseje 00:00:00-tól eltelt másodpercek száma volt.
  Ez az ábrázolási mód 2036-ig „jó" (136 év a körülfordulási ciklusa).

Példa: Az óraszinkronizáció szükségessége/jelentősége: UNIX **make** program

Nagy programok forrásai fel vannak osztva részekre (pl 100 file).

Csak azokat kell újrafordítani, amelyekhez tartozó forrás megváltozott.

Ha a forrás időbélyege későbbi, például **input.c** (timestamp **2151**), és **input.o** (timestamp **2150**), akkor újra kell fordítani a forrás file-t.

De ha az editor és a compiler különböző gépen fut, akkor az időbélyegek értelmezésével baj lehet, ha az órák nincsenek szinkronban!

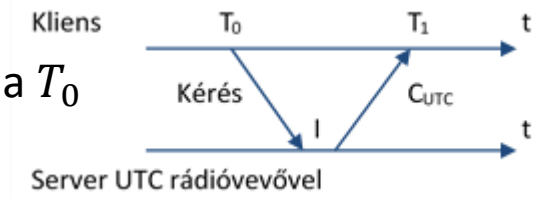
Ha azt tapasztaljuk, hogy a forrás időbélyege korábbi, mint a lefordítotté, azaz **output.c** (timestamp **2143**), és **output.o** (timestamp **2144**), akkor nem fordítunk.

De ha ennek az az oka, hogy az editort futtató gép órája késik két időegységet, akkor baj van!

Órák szinkronizálása

Berkeley algoritmus: Aktív időszerver: rendszeresen lekérdezi a csomópontok óráját, átlagolja azokat, majd visszaküldi.

Cristian algoritmus: A szinkronizálást a kliens kezdeményezi a T_0 időpillanatban egy UTC rádióvevővel rendelkező szervernél.



A kérés megérkezésekor, az interrupt kiszolgálását (I) követően a szerver lekérdezi az UTC rádiót, majd a lekérdezett C_{UTC} értéket megküldjük a kliensnek. A T_1 időpillanatban megérkező óra adatot korrigálni kell az üzenettovábbítás idejével. Ha az üzenettovábbítás ideje mindkét irányban közel azonos, akkor a szükséges korrekció közelítése: $\sim \frac{T_1 - T_0 - I}{2}$.