Alkalmazott beágyazott rendszerek

Óravázlatok (2-12.)

Összeállította:

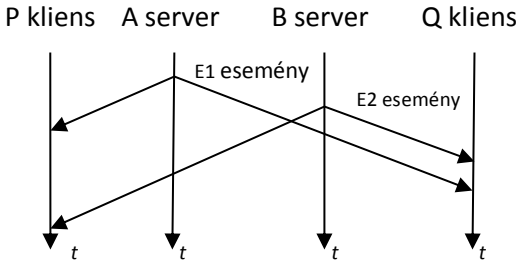
Péceli Gábor

2017.

1. Bevezetés, ráhangolódás (folyt.)

Példák az időviszonyok sajátosságaira beágyazott rendszerekben:

- **Relativisztikus hatás**: a kommunikáció időviszonyai események tényleges sorrendjét a vétel helyén megváltoztathatják. (Lásd az alábbi illusztrációt!)



Az ábrán az látható, hogy a Q kliens esetében az E2 eseményről szóló híradás megelőzi az időben korábbi E1 eseményről érkező híradást. Az események sorrendjétől függő döntések esetén ebből baj lehet. Ha az E1 és az E2 események nem függetlenek egymástól, akkor jogos felvetés, hogy az E2 eseményről szóló híradás megérkezését követően a híradásokat figyelembe vevő döntéseinkkel várakozzunk addig, amíg minden olyan eseményről szóló híradás, amely az E2 eseménnyel egyidejűleg vagy azt megelőzően történt – a legkedvezőtlenebb esetben is – beérkezik a Q klienshez. Ezt a várakozási időt akció késleltetési időnek (action delay) nevezzük. A szükséges akció késleltetési időt akkor tudjuk meghatározni, ha van minimális (alsó) és maximális (felső) korlátunk az üzenettovábbítási időre, azaz a d üzenettovábbítási időre fennáll:

$$d_{\min} \le d \le d_{\max} .$$

Ha a szóban forgó csomópontok számára ismert a globális idő, azaz az üzenettovábbítás során van értelme időbélyeget is küldeni, akkor a kliens a vétel időpontja és a küldés időbélyege alapján meg tudja határozni, hogy a vett üzenet továbbítása mennyi időt vett igénybe, és a legkedvezőtlenebb viszonyokat feltételezve meddig kell még várakoznia ahhoz, hogy minden egyidejűleg vagy megelőzően küldött üzenet megérkezzen. Ez az időpont természetesen az üzenetküldés időpontjának és a $d_{\max}$ értéknek összege lesz. Ilyenkor az akció késleltetési idő $d_{\max}$.

Ha a szóban forgó csomópontok számára a globális idő nem ismert vagy legalábbis lokális óráik nem szinkronizáltak, akkor az időbélyeg közvetlenül nem használható, az akció késletetés mértékének meghatározásánál csak az üzenet beérkezéséből tudunk kiindulni. Mivel a tényleges üzenettovábbítási időt nem tudjuk mérni, ezért a legkedvezőbb $d_{\min}$ időt feltételezve még mindenképpen várunk $d_{\max}$ - $d_{\min}$ időt az akció beindításáig. Mivel elképzelhető, hogy az üzenet a legkedvezőtlenebb $d_{\max}$ idő alatt érkezett, ezért az üzenet küldéséhez képest a legkedvezőtlenebb esetben $2d_{\max}$ - $d_{\min}$ várakozásra kerül sor, ilyenkor tehát az akció késleltetési idő $2d_{\max}$ - $d_{\min}$.

Megjegyzések:

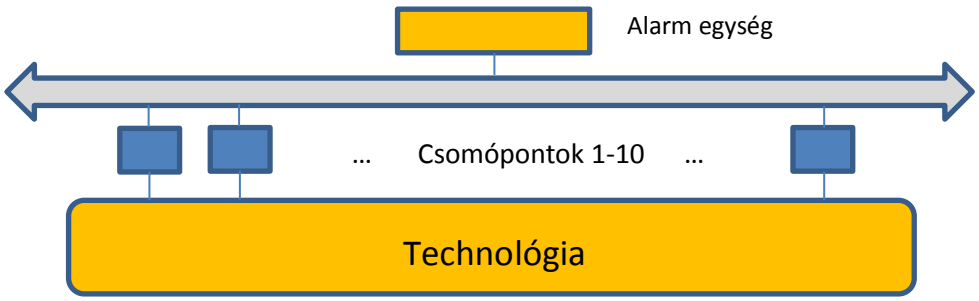
1. Látható, hogy a második esetben, nagy $d_{\max}$ - $d_{\min}$ érték esetén, lényegesen kedvezőtlenebb helyzettel állunk szemben. Célszerű az üzenetküldési időt ~állandó értéken tartani.
2. Bizonyos kommunikációs protokollok esetén a $d_{\max}$ - $d_{\min}$ különbség nagy lehet: például tokenvezérelt busz esetén, ha a token körüljárási idő mondjuk 10 *ms*, maga az üzenettovábbítás pedig mindig 1 *ms*, akkor $d_{\max}$ = 11*ms* lesz, míg $d_{\min}$ = 1 *ms*, hiszen a legkedvezőtlenebb esetben az üzenettovábbítás kezdeményezését közvetlenül megelőzi a token továbbítása az adott csomópontról/készülékről, tehát 10 *ms*-ig várni kell, és utána a tokent visszakapva már 1 *ms* alatt a címzetthez jut, de ez összességében $d_{\max}$ = 11*ms* időtartamot eredményez.

3. Az akció késleltetési idő kivárásával olyan helyzetet teremtünk, hogy egy adott csomópontra érkező üzenet kapcsán elmondható, hogy minden, a küldésével egyidejű, ill. korábbi üzenet megérkezett (ill. sosem fog megérkezni). Ezt a relációt állandóságnak (permanence), magát az üzenetet pedig permanensnek nevezzük.
4. Az állandóság/akció késleltetési idő ki nem várása súlyos következményekkel járhat vissza nem fordítható akciók kezdeményezése esetén. Gondoljunk fegyver elsütésére, pilóta katapultálására.
5. Beágyazott rendszerek esetében a csomópontok közötti kommunikációban – sok esetben – maguk a fizikai/technológiai berendezések is részt vesznek. Egy beavatkozás eredményeként ezen berendezések működésében beállt változásokat érzékelőkkel detektáljuk. A beavatkozó és az érzékelő közötti fizikai folyamatok ilyenkor kommunikációs csatornaként (is) működnek. Ezek valójában rejtett csatornák, amelyek időviszonyairól az akció késletetés meghatározása során nem szabad megfeledkeznünk.
6. Valós idejű rendszerekben az akció késletetés mértéke problémát okozhat az üzenetekben küldött információ felhasználhatóságát illetően: egy mérési adat hamar elévülhet (nem lesz elegendően pontos), ha a forrása egy időben gyorsan változó folyamat.

- **Eseményvezérelt (event triggered, ET) és idővezérelt (time triggered, TT) rendszerek:**

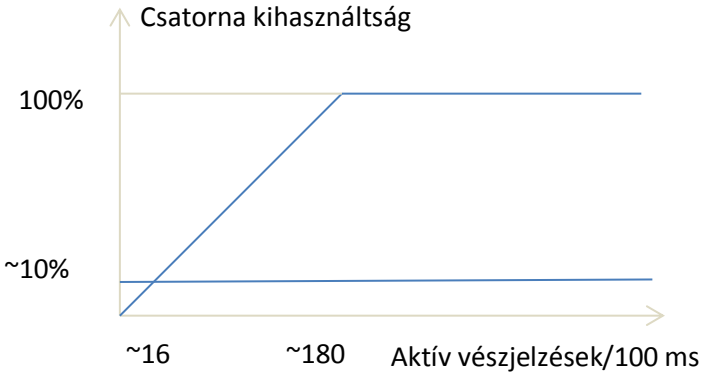
Az eseményvezérelt rendszerek a kiváltó események/kérések hatására hajtják végre az eseményhez rendelt programot. Ezzel a megközelítéssel kedvező válaszidők érhetőek el, de a közel egyidejű események számának növekedésével a rendszer kapacitása/átbocsátóképessége/teljesítménye elégtelenné válik, és ebből adódóan a határidők betartása ellehetetlenül. Az idővezérelt rendszerek esetében minden megoldandó feladathoz tervezési időben egy különálló időszeletet rendelünk, ezáltal a feladat-végrehajtás előzetesen ismert válaszidő mellett garantálható.

Példa: Egy technológiai folyamatot 10 csomópont felügyel. Mindegyik csomópont 40 bináris jelet (vészjelzés, például határérték átlépés információ) figyel. A 10 csomópont egymással buszon kommunikál. Ugyanide csatlakozik egy vészjelző (alarm) egység. A buszon a jelátviteli sebesség 100 kbit/s. A vészjelzésnek 100 ms-en belül el kell jutnia az alarm egységhez.



1. Eseményvezérelt eset: ET/CAN protokoll szerint. A legkisebb átvihető üzenethossz a bájt. A protokoll szabályai szerint felépülő üzenet teljes hossza: 44 bit overhead, 1 bájt üzenet, amit 4 bit ún. inermessage gap követ. Ez összesen 56 bit. A 100 kbit/s azt jelenti, hogy az előírt 100 msec-en belül 10 000 bit jut át. 56 bites üzenetekben gondolkodva 10 000/56 ~ 180 juthat át a specifikált határidőn belül. Mivel 180 < 400, ezért egyidejűleg valamennyi jelzés átküldésére nincsen lehetőség, az átvitel csatorna ~180 egyidejű üzenet esetén telítődik.
2. Idővezérelt eset: TT/CAN protokoll szerint. A csomópontok rendszeresen beküldik az állapotjelző biteket az alarm egységnek. Ez 40 bitenként egy-egy üzenet beküldésével megoldható. A protokoll szabályai szerint felépülő üzenet teljes hossza: 44 bit overhead, 40 bit (5 bájt) üzenet, amit 4 bit ún. inermessage gap követ. Ez összesen 88 bit. A 100 kbit/s azt jelenti, hogy az előírt 100 msec-en belül 10 000 bit jut át. 88 bites üzenetekben gondolkodva 10 000/88 ~ 110 juthat át a specifikált határidőn

belül. Mivel 110 > 10, ezért valamennyi állapotjelző bit átjut az alarm egységhez, ráadásul állandó, ~ 10%-os csatorna kihasználás mellett.



- **Kemény és puha valós idejű rendszerek:**

- kemény valós idejű rendszer (hard real-time system (HRT)): katasztrofális következményekkel jár, ha nem tartjuk az időkorlátot (pl. járművek vezérlése).
- puha valós idejű rendszer (soft real-time system (SRT), online system): az eredmény értékes az időkorláton túl is, de az idővel degradálódik (pl. banki/tranzakciós rendszerek).

HRT és SRT jellemzése különböző szempontok szerint:

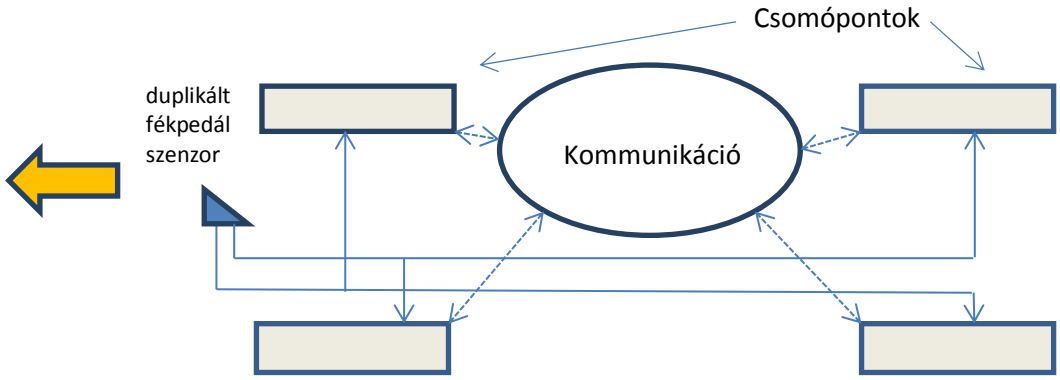
- *válaszidő* (response time): HRT esetében msec, vagy annál kevesebb (pl. légzsák), az emberi beavatkozás lehetősége kizárt, a rendszer autonóm működésű és biztonságos kell legyen. SRT esetén a válaszidő másodperc nagyságrendű, az időkorlát túllépése nem okoz katasztrófát.
- *viselkedés csúcsterhelés esetén* (peak-load performance): HRT esetén jól definiált kell legyen. Tervezéskor biztosítani kell, hogy a számítógépes rendszer minden szituációban az időkorláton belül teljesítse feladatát, hiszen a HRT rendszerek éppen azáltal valósítják meg a velük szemben megfogalmazott elvárásokat, hogy még a ritkán előforduló csúcsterhelések idején is jósolható módon viselkednek. Az SRT rendszereket átlagos teljesítmény-jellemzőkre tervezzük, a ritkán előforduló csúcsterhelések következményeit - gazdaságossági megfontolásból - elviseljük.
- *az ütem vezérlése* (control of pace): A HRT rendszernek minden körülmények között szinkronban kell lennie környezetének (irányított objektum, ill. az emberi operátor) állapotával. Az SRT rendszerek befolyásolják környezetüket, ha nem képesek eleget tenni feladatuknak (egy tranzakciós rendszer például megnöveli a válaszidejét).
- *biztonság* (safety): A biztonság kritikusságának mértékétől függően sokféle feladat merülhet fel tervezési időben. Autonóm hibadetektálási mechanizmusokat kell kidolgozni, amelyek valamilyen "talpra állítási" (recovery) akciót indítanak az adott alkalmazás által diktált időviszonyok mellett.
- *az adatfájlok mérete* (size of data files): HRT rendszerek kisméretű adatfájlokon dolgoznak, amelyek valós idejű adatbázist alkotnak. Ezek jellemzője az adatintegritás rövid idejűsége, mert az idő múlásával az adatok jelentős része aktualitását veszíti. Az SRT rendszerekben éppen ellenkezőleg a hosszú idejű adatintegritás fontos.
- *a redundancia típusa* (redundancy type): SRT rendszerekben (pl. tranzakciós rendszerek) hiba esetén a számításokat "visszagörgetik" a legutolsó ellenőrzési ponthoz, amikor még biztosan helyes volt a működés és onnan kezdik a "talpra állítást". HRT rendszerek esetén ez a stratégia csak korlátozottan használható mert: (1) az időkorlát tartása nehéz, mert a visszagörgetéshez szükséges idő nem, vagy

nehezen jósolható, (2) a környezetet befolyásoló "utasítás" nem tehető meg nem történtté, (3) az ellenőrzési pontnál érvényes adatok az idő múlásával érvényüket veszítik.

- *adat integritás* (data integrity): HRT: rövid idejű, SRT: hosszú idejű.
- *hibadetektálás* (error detection): HRT: autonóm, SRT: felhasználó által segített.

- Megegyezés protokollok jelentősége

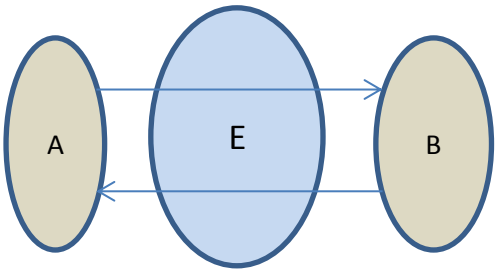
Példa: elektronikus fékvezérlés (brake-by-wire):



A példa szerint a biztonság érdekében duplikált fékpedál szenzort alkalmazunk. Az egyes kerekek fékjeihez önálló vezérlő csomópontok tartoznak. A csomópontok egymást tájékoztatják arról, hogy mi az ő véleményük a szenzor értékéről, és kiszámítják a fékerőt. Ha megsérül egy csomópont, akkor automatikusan szabadonfutó lesz, ilyenkor nincsen fékhatás. A többi három, amikor észleli, hogy egy kiesett, automatikusan újraszámítja a fékerőt, és biztonságosan fékez.

Elosztott rendszerekben sokféle kérdésben szükséges futási idejű megállapodás: idő szinkronizáció, elosztott állapotok konzisztenciája, elosztott kölcsönös kizárás, elosztott tranzakciós megállapodás, elosztott befejezés, elosztott választás, stb. Közös probléma, hogy hibák fellépése esetén is megállapodásra kellene jutni. Ez nem mindig sikerül:

Példa: Két hadsereg problémája: A szövetséges A és B hadseregnek együttesen több katonája van, mint az E ellenségnek, de egyenként kevesebb. Megállapodásra kell jutni a támadás időpontjáról. Ehhez kommunikálni kell, például hírnököt (H) küldeni, akit azonban elfoghat az E ellenség, tehát a kommunikáció nem hibamentes.



Ha A parancsnoka H hírnököt küld B parancsnokának, hogy holnap délután 4-kor támadjunk, akkor a nem hibamentes csatorna miatt kell visszaigazolás. (Ettől függetlenül az is lehetséges, hogy B parancsnoka is küld hírnököt más időpont javaslattal.)

A probléma nyilvánvaló:

- Ha H nem tér vissza A-hoz, mi a konklúzió?
- Ha H a visszaúton esik fogságba, akkor B elindul adott valószínűséggel, de A nem fog, mert nem kapott visszaigazolást.
- Ha H az odaúton esik fogságba, akkor A van veszélyben, ha egyedül cselekszik.
- Ha H vissza is tér A-hoz, van valószínűsége, hogy B nem támad, mert nem tudja visszaért-e a hírnök. Ezt elkerülendő B elküldheti a saját hírnökét A-hoz, annak ellenőrzésére, hogy a visszaigazolás odaért-e.

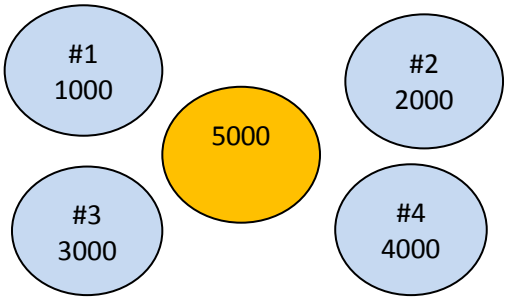
Ha újabb hírnököket küldünk, akkor nő annak valószínűsége, hogy a visszaigazolás átjut, de ez nem oldja meg alapvetően a problémát, mert mindig van véges valószínűsége, hogy a hírnököt elfogják.

Lehetetlenségi tétel (Impossibility Result): Formálisan bizonyítható, hogy nem garantálható, hogy két vagy több elosztott egység megegyezésre/megállapodásra jut véges idő alatt egy aszinkron kommunikációs közegen keresztül, ha a közeg veszteséges vagy valamelyik egység kiesik. Amit tehetünk: a megegyezés valószínűségét növeljük.

Megegyezés bizánci típusú hibák esetén:

Példa: Órák szinkronizálása: Az A óra 4.00-t mutat, a B óra 4.05-t mutat, a C óra az A-nak 3.55-t, a B-nek 4.10-et. Ezt a hibafajtát nevezzük bizánci típusú hibának. Ilyenkor nem jön létre a megállapodás, mert az A óra és a B óra is arra a megállapításra jut, hogy az általa mutatott érték a másik két óra által mutatott érték számtani közepe, tehát nincs indok megváltoztatni. A bizánci típusú hibás csomópont kiszűrése akkor lehetséges, ha legalább $3k+1$ csomópont vesz részt a szinkronizációban, ahol a k a bizánci típusú hibás csomópontok számát jelöli. Esetünkben egy hibátlanul működő további óra-csomópont (D) szükséges a hibás csomópont kiszűréséhez.

Példa: A bizánci generálisok problémája: Az alábbi ábrán látható elrendezésben 4 hadtest generálisa megegyezésre törekszik az egyszerre harcba küldhető katonák számát illetően, de menetközben kiderül, hogy az egyik generális hazudós („szoftver hiba"). Az ellenségnek 5000 katonája van.



A (formális) szövetségesek egymással hibamentesen kommunikálnak: mindenki megküldi a katonái számát. Az egyes csomópontokban az alábbi adatok állnak rendelkezésre:

#1: (1K, 2K, xK, 4K), #2: (1K, 2K, yK, 4K), #3: (1,2,3,4), #4: (1K, 2K, zK, 4K), ahol x, y, z a ténylegestől különböző, egymástól potenciálisan eltérő érték, mert a #3 számú generális/csomópont hazudós (szoftver hibás). Nyilván ezzel a #1, #2 és #4-es csomópontokban nincsenek tisztában, mindenki csak az értékeket ismeri.

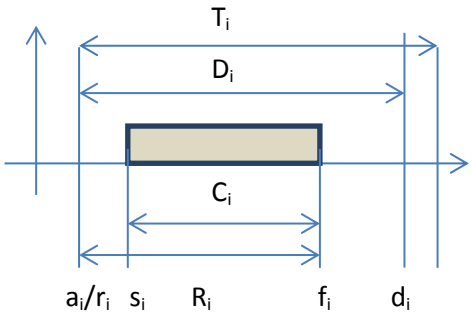
Annak érdekében, hogy az értékek helyes voltát ellenőrizni tudják, az információs vektoraikat körbeküldik a kommunikációs csatornáikon keresztül, amelyek az előzőek szerint fognak viselkedni, tehát a hazudós csomópont a körbeküldött vektor-elemeket illetően is hazudós. A körbeküldést követően az egyes csomópontokban a következő információ áll rendelkezésre (ezer katonában):

$$\#1:\begin{bmatrix}1 & 2 & y & 4\\ a & b & c & d\\ 1 & 2 & z & 4\end{bmatrix}\quad \#2:\begin{bmatrix}1 & 2 & x & 4\\ e & f & g & h\\ 1 & 2 & z & 4\end{bmatrix}\quad \#4:\begin{bmatrix}1 & 2 & x & 4\\ 1 & 2 & z & 4\\ i & j & k & l\end{bmatrix}$$

Mindhárom – nem hazudós – generális a három információs vektor esetében két helyről ugyanazt az információt kapja, kivéve a #3-as generális esetében. Következtetésük, hogy $[1 \quad 2 \quad ismeretlen \quad 4]$, azaz lesz legalább 7 ezer katona, akire számítani lehet a támadásnál.

2. Ütemezés

Probléma: a processzor(ok)nak többféle időzítés mellett többféle feladatot (task) kell ellátniuk. Egy i-edik feladathoz (task-hoz) köthető időviszonyok az alábbiak szerint értelmezhetőek:



Itt a_i vagy r_i az érkezési idő (arrival/release/request time), s_i a végrehajtás kezdésének ideje (start time), f_i a végrehajtás befejezésének ideje (finishing time), d_i a végrehajtás határideje (deadline), T_i a periódusidő (period time), D_i=d_i-a_i a kérés időpontjához képesti határidő, C_i a számítási idő (computation time), R_i=f_i-a_i a válaszidő.

1. Ciklikus ütemezés: a legegyszerűbb, tervezési időben fix időszeletek osztunk ki periodikus kérések kiszolgálására, és ezt ciklikusan ismételjük. A kiosztást tipikusan óra-vezérelt módon oldjuk meg, ezért óra vezérelt vagy idő-vezérelt ütemezésnek is nevezzük. Többféle változata van, de közös tulajdonságuk, hogy az ütemezéssel kapcsolatos döntések tervezési időben történnek, és ezáltal a futási időben jelentkező overhead alacsony. Ugyancsak jellemzőjük, hogy a HRT task-ok paraméterei ismertek és fixek.

Példa: 10 ms-os időszeletet kap minden feladat (kis keret). Négy funkciót úgy valósítunk meg: 50 Hz-es periodicitással, azaz 20 ms-onként adunk 10 ms-ot az első funkciónak, 25 Hz-es periodicitással, azaz 40 ms-onként 10 ms-ot a második funkciónak, 12.5 Hz-es periodicitással, azaz 80 ms-onként 10 ms-ot a harmadik funkciónak, és végül 6.25 Hz-es periodicitással, azaz 160 ms-onként 10 ms-ot a negyedik funkciónak.

Természetesen az időszeletek kiosztása variálható, de érdemben csak tervezési időben, tehát az ütemezés meglehetősen kötött/merev lesz.

Megjegyzés: A fenti példában az első funkció a processzoridő felét, a második a negyedét, a harmadik a nyolcadát, stb. használta fel. Érdemes felidézni azt az eredményt, hogy

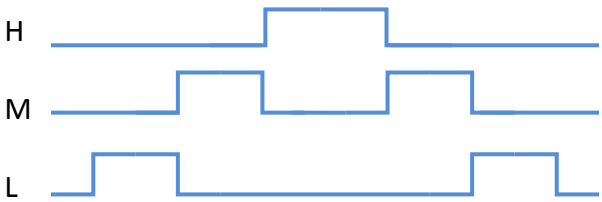
$$\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\ldots\rightarrow 1,$$

azaz a funkciók száma növelhető a végtelenségig, ha az igényelt processzoridő rendre az előző felére csökken! Ezt a tulajdonságot használták ki a világ első digitális szűrőkkel működő valósidejű 1/3 oktáv elemzőjének, a Brüel & Kjaer 2131 tervezői is 1977-ben! Ez a berendezés 1.6 Hz és 20 kHz tartományban, összesen 42 sávban képes 1/3 oktávos analízisre, illetve 2 Hz és 16 kHz sávközépi frekvenciákkal 14 sávban oktáv analízisre. Mivel az oktáv analízis olyan sávszűrőket alkalmaz, amelyek 3 dB-es sávhatárainak aránya 1:2, ezért felvethető, hogy amennyiben a 16 kHz sávközepű digitális sávszűrő f_m mintavételi frekvenciával működik, akkor – kellő mértékű sávkorlátozás esetén – az eggyel alatta levő, 8 kHz sávközepű digitális sávszűrőnek elegendő $f_m/2$ mintavételi frekvenciával működnie, és így tovább. Elvileg akármilyen kis frekvenciákig elmehetnénk, hiszen a fenti összeg csak határértékben éri el az 1-et. A megvalósított berendezésben f_m=66.667 kHz. A legmagasabb frekvenciasáv és az összes többi kiszolgálása a mintavételi idő felét-felét veszi igénybe. Ebben az ütemben két digitális szűrő blokk működik. Az egyik egy hatod-fokú sávszűrő hardver, amelyet $1/2f_m$ idő alatt háromszor használunk – rendre más paraméterekkel – a három egyharmad oktávos szűrő egy-egy újabb kimeneti értékének meghatározására, a másik egy negyedfokú alul-áteresztő hardver, amelyet $1/2f_m$ idő alatt háromszor használunk – rendre más paraméterekkel – egy tizenketted-fokú alul-áteresztő megvalósítására, amellyel a mintavételi tétel alkalmazhatóságának érdekében sávkorlátozást végzünk. Az A/D átalakítóból érkező mintát először a sávkorlátozó alul-áteresztő szűrőre vezetjük, majd ennek kimentéről levehető mintát a sávszűrők kapják. A mintavételi idő második időszeletében valamelyik alacsonyabb frekvenciás sávkorlátozó, majd sávszűrő működtetése történik.

2. Időosztásos (time-shared)/körforgó (round-robin) ütemezés: A futtatható task-ok egy FIFO-ba (First-In First-Out) kerülnek, és a legelöl álló task fog futni maximum egy időszelet ideig. Az időszelet általában

néhányszor 10 ms, ami a task-októl független paraméter. Ha az adott task nem fut le az időszelet alatt, akkor futása megszakad, és a FIFO végére kerül.

3. Prioritásos ütemezés: A futtatható task-ok közül az fut, amelyiknek legnagyobb a prioritása. A prioritás hozzárendelés történhet tervezési és futási időben egyaránt. A működést a következő ábra illusztrálja. A három task rendre alacsony (L=low), közepes (M=medium) és magas (H=high) prioritású. Ezeket a prioritásokat tervezési időben osztottuk ki. Az ábrán mindhárom task azonnal futni kezd, amint futtathatóvá válik.



Az ábrán látható esetben a legalacsonyabb prioritású task válaszideje $R_L=C_L+C_M+C_H$. Ha a középső és/vagy a magas prioritású task periodikusan kér, akkor az időviszonyok függvényében elképzelhető, hogy az R_L idő alatt többször is lefut. A válaszidő számítást a legkedvezőtlenebb esetre, az i-edik task-ra vonatkoztatva, a következő képlettel tudjuk elvégezni:

$$R_i = C_i + I_i = C_i + \sum_{\forall k \in hp_i} \left\lceil \frac{R_i}{T_k} \right\rceil C_k ,$$

ahol I_i az ún. interferencia idő, azaz az az időtartam, amíg a magasabb prioritású task-ok futása akadályozza az alacsonyabb prioritású task-ok végrehajtását. A $\forall k \in hp_i$ azokat a task-okat jelöli ki, amelyek prioritása nagyobb, mint i (hp=higher priority). A $\lceil \ \rceil$ zárójel a felső-egész képzés operátora. $\lceil 1.02 \rceil = 2$, $\lceil 2.0 \rceil = 2$. Mivel a fenti képletben a baloldalon szereplő R_i a jobboldalon is szerepel egy erősen nemlineáris függvény argumentumában, ezért iteratív eljárás alkalmazására kényszerülünk:

$$R_i^{n+1} = C_i + I_i = C_i + \sum_{\forall k \in hp_i} \left\lceil \frac{R_i^n}{T_k} \right\rceil C_k$$

Az iterációt addig folytatjuk, amíg: egy n_0 érték mellett $R_i^{n_0+1} = R_i^{n_0}$. A bemutatott eljárás Deadline Monotonic Analysis (DMA) néven szerepel a szakirodalomban, és azt feltételezi, hogy a task-okhoz aszerint rendelünk prioritást, hogy mekkora a D_i határidejük. A módszer alkalmazásánál feltételezzük, hogy $D_i \le T_i$. A módszer periodikus task-ok mellett ún. sporadikus task-okra is alkalmazható.

Periodikus task: ismert és fix T_i periodusidővel jellemezhető.
Sporadikus task: a kérések nem periodikusak, de ismert és fix egy olyan T_i időérték, ami minimálisan eltelik két kérés között.
Aperiodikus task: a kérések nem periodikusak, és nincs egy olyan ismert és fix T_i időérték, ami minimálisan eltelik két kérés között, tehát egy kérést követően azonnal megjelenhet egy következő kérés. Értelemszerűen ebben az esetben a DMA módszer nem alkalmazható.

Fontos megjegyezni, hogy a DMA módszer nem a válaszidőt, hanem annak a lehető legkedvezőtlenebb értékét adja meg. (Worst-case response time.)

Példa: Egy 4 task-ot kiszolgáló rendszer adatai a következők (az idők pl. ms-ban értendők):

Task	T	C	D
1	250	5	10

2	10	2	10
3	330	25	50
4	1000	29	1000

A task-ok sorrendje a prioritási sorrend. Ha a határidők megegyeznek, akkor másodlagos szempontok alapján döntünk a prioritásról. A példában az első task számítási ideje nagyobb, kisebb a "lazasága", ezért jogos lehet az előbbre sorolása. Határozzuk meg a 3-as task worst-case válaszidejét az iteratív eljárás segítségével! Az iteratív eljárás táblázatos formában:

Lépés	R^n	I	R^{n+1}
1	0	0	25
2	25	5+3*2	36
3	36	5+4*2	38
4	38	5+4*2	38

Megjegyzések:

1. 38<50, tehát a 3-as task legkedvezőtlenebb esetben is teljesíti az előírt határidőt.
2. Vegyük észre, hogy a 4-es task adatait az eljárás során nem használtuk fel, a számításhoz felesleges volt megadnunk.
3. Vegyük azt is észre, hogy task-okat egymástól függetleneknek képzeltük el. Egymástól nem független, azaz például egymással kommunikáló, egymásnak adatot továbbító task-ok esetében előfordul(hat), hogy magasabb prioritású task alacsonyabb által szolgáltatott adatra várni kénytelen. Ez a várakozási idő értelemszerűen a mindenkori és a worst-case válaszidejét egyaránt módosítani fogja.

Példa: Egy 4 task-ot és egy megszakítást (i_1) kiszolgáló rendszer adatai a következők (az idők pl. ms-ban értendők):

Task	T	C	D
i_1	10	0.5	3
t_1	3	0.5	3
t_2	6	0.75	6
t_3	14	1.25	14
t_4	50	5	50

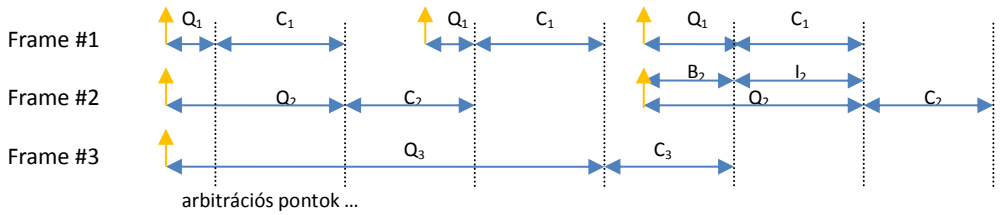
Határozzuk meg a t_4 task worst-case válaszidejét az iteratív eljárás segítségével! Az iteratív eljárás táblázatos formában:

Lépés	R^n	I	R^{n+1}
1	0	0	5
2	5	0.5+1.0+0.75+1.25	8.5
3	8.5	0.5+1.5+1.50+1.25	9.75
4	9.75	0.5+2.0+1.50+1.25	10.25
5	10.25	1.0+2.0+1.50+1.25	10.75
6	10.75	1.0+2.0+1.50+1.25	10.75

Látható, hogy 10.75<50, tehát a határidő minden esetben teljesül.
Megjegyzés: az ismertetett DMA analízis technikákat autógyárak intenzíven használják worst-case válaszidő analízis céljából, hogy a terméket optimalizálják a szükséges órajel frekvenciák/sávszélességek és az ehhez kapcsolódó zavarérzékenységek csökkentésével. (A Volvo már 1995-től használ ilyet, legelőször a S80-as típusnál.)

Példa: A DMA analízis egy módosított formája használható nem preemptív, azaz az éppen futó task-ot nem megszakító működés esetén is. Erre példaként szolgáljon a prioritásos CAN bus válaszidő analízise.

A CAN (Control Area Network, ISO 11898, Bosch) buszon történő kommunikáció jellegzetességeit az alábbi ábra mutatja be. Itt három üzenet továbbítását kell megoldanunk prioritásos rendben.



Az ábrán látható szaggatott vonalak az ún. arbitrációs pontokat jelölik az időtengely mentén. Ezekben az időpontokban történik annak vizsgálata, hogy melyik üzenet (frame) továbbítására kerül sor. A prioritási sorrend felülről csökkenő. Az egyidejű kérések vizsgálatáig mindhárom üzenet várakozik. A vizsgálatot követően a frame#1 átvitelére kerül sor. C_1 a kommunikáció ideje, megfeleltethető a számítási időnek azzal, hogy itt minden frame esetén ugyanaz az érték. Ezt követi a frame#2 átvitele. Közben a magas prioritáson újabb kérés érkezik, ami a vizsgálatig várakozik, majd sor kerül az üzenet továbbítására. Más kérés nem lévén ezt követi a frame#3 átvitele. Közben az egyidejű magas és közepes prioritású kérés várakozik. A közepes prioritás itteni várakozása két részre bontható: az egyik a B_2 ún. blokkolódás, ami alatt alacsonyabb prioritású üzenet átvitele folyik, a másik az I_2 ún. interferencia, ami alatt magasabb prioritású üzenet továbbítására kerül sor. Mindezek alapján a válaszidő számítása a következőképpen történik:

$$R_i = C_i + Q_i \text{, ahol } Q_i = B_i + \sum_{\forall k \in hp_i} \left\lceil \frac{Q_i}{T_k} \right\rceil C_k .$$

B_i a leghosszabb üzenet-átviteli idő egy tetszőleges alacsonyabb prioritású frame részéről. Az ábra alapján látható, hogy mivel itt is a legkedvezőtlenebb esetet vizsgáljuk, ezért a leghosszabb blokkolás idő két arbitráció között eltelt idő lehet. Ez az az eset, amikor a kérés éppen az arbitrációt követően érkezett.

Üzenet	T [ms]	C[ms]
1	3	1.35
2	6	1.35
3	10	1.35
4	30	1.35
5	40	1.35
6	40	1.35
7	100	1.35

Az üzenetek periodikusak és prioritásuk felülről csökkenő. A küldésükre vonatkozó kérés érkezése aszinkron, tehát tetszőleges kezdőfázissal érkezhetnek. A 7. üzenet fékezéssel kapcsolatos információt hordoz, 100 ms alatt a rendeltetési helyére kell kerüljön. Az iteratív eljárás a várakozási időre vonatkozóan:

Lépés	Q^n	I							B	Q^{n+1}
		1	2	3	4	5	6	Összeg		
1	0	-	-	-	-	-	-	0	1.35	1.35
2	1.35	1	1	1	1	1	1	8.1	1.35	9.45
3	9.45	4	2	1	1	1	1	13.5	1.35	14.85
4	14.85	5	3	2	1	1	1	17.55	1.35	18.9
5	18.9	7	4	2	1	1	1	21.6	1.35	22.95
6	22.95	8	4	3	1	1	1	24.3	1.35	25.65
7	25.65	9	5	3	1	1	1	27	1.35	28.35
8	28.35	10	5	3	1	1	1	28.35	1.35	29.7
9	29.7	10	5	3	1	1	1	28.35	1.35	29.7

A worst-case várakozási idő tehát 29.7 ms, amivel a worst-case válaszidő: 29.7ms+1.35ms=31.05 ms. Ez kisebb, mint a megadott 100 ms, tehát teljesül a specifikált határidő.

Megjegyzés: az ismertetett DMA analízis technikákat autógyárak intenzíven használják worst-case válaszidő analízis céljából, hogy a terméket optimalizálják a szükséges órajel frekvenciák/sávszélességek és az ehhez kapcsolódó zavarérzékenységek csökkentésével. (A Volvo már 1995-től használ ilyet, legelőször a S80-as típusnál.)

2. Ütemezés (folytatás)

Ütemezhetőség, ütemezhetőségi tesztek:

- *szükséges:* nem ütemezhető, ha a szükséges feltétel nem teljesül.
- *elégséges:* biztosan ütemezhető, ha az elégséges feltétel teljesül.
- *egzakt:* szükséges és elégséges, és a teszt az ütemezés létezését is megmutatja. Az egzakt ütemezhetőségi tesztek komplexitásuk alapján az NP-teljes problémák osztályába tartoznak, ezért számítástechnikailag kezelhetetlenek, ezekkel a továbbiakban nem foglalkozunk.

Periodikus task-ok esetén a szükséges feltételek között elsőként az ún. processzor-kihasználtsági tényező említhető, ami az időegységre vetített processzor-idő igények összege:

$$\mu = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

Egyprocesszoros rendszerben, ha $\mu \le 1$ nem teljesül, akkor a task-ok nem ütemezhetőek, tehát a $\mu \le 1$ szükséges feltétel.

Ütemezési stratégiák:

Rate-monotonic (RM) (1973): Periodikus, egymástól független task-ok esetére, akkor, ha $D_i=T_i$ és C_i ismert és konstans. A prioritás hozzárendelés úgy történik, hogy a legnagyobb prioritást a legkisebb periódusidejű task kapja. Az eljárás preemptív. Feltételezzük, hogy a task-ok közötti átkapcsolás ideje elhanyagolható. Az RM algoritmusra elégséges teszt adható. A megadott feltételek teljesülése mellett, ha

$$\mu = \sum_{i=1}^{n} \frac{C_i}{T_i} \le n\left(2^{\frac{1}{n}} - 1\right) \xrightarrow[n\to\infty]{} \ln 2 \sim 0.7,$$

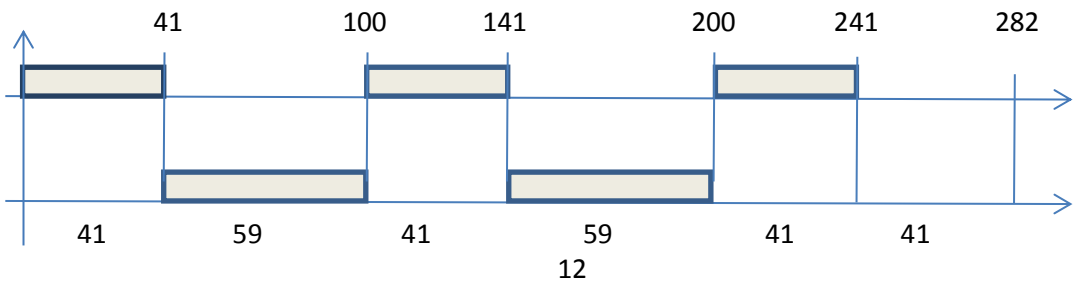
teljesül (n az ütemezendő task-ok száma), akkor biztos van ütemezés. Nagyobb processzor kihasználtság mellett is elképzelhető, hogy az RM stratégiával ütemezhetőek a task-ok, de erre nincsen garancia. Véletlenszerűen választott T_i és C_i esetén a szimulációk $\mu = 0.88$ értékig sikerrel jártak. Ha a periódusidők egy alapérték egész-számú többszörösei, akkor bizonyítható, hogy $\mu = 1$ elérhető.

Példa: A példa azt illusztrálja, hogy milyen periódusidő és számítási idő viszonyok esetén jutunk el az ütemezhetőség határára. Ha n=2, akkor

$\frac{T_2}{T_1} = \sqrt{2}$, $C_1 = T_2 - T_1$, akkor $\frac{C_1}{T_1} = \frac{T_2 - T_1}{T_1} = \frac{C_2}{T_2}$ választással: $\mu = 2\left(\sqrt{2} - 1\right)$, illetve tetszőleges i esetén, ha

$\frac{T_{i+1}}{T_1} = 2^{\frac{1}{n}}$, $C_i = T_{i+1} - T_i$, akkor $\mu = n\left(\frac{T_{i+1}}{T_i} - 1\right) = n\left(2^{\frac{1}{n}} - 1\right)$.

Egy két task-ból álló rendszer esetén legyen $T_1 = 100$, $C_1 = 41$, $T_1 = 141$, $C_1 = 59$, mind *ms* dimenziójú. A processzor-kihasználtsági tényező $\frac{41}{100} + \frac{59}{141} = 0.41 + 0.4184 = 0.8248$, azaz lényegében a képlettel kapható érték. Az ütemezés időviszonyai egyidejű kezdést feltételezve:



Látható, hogy a számítási idők minimális növelése esetén az ütemezés az RM stratégia esetén ellehetetlenül. Ugyanakkor 241 és 282 között nincsen ütemezhető feladat, tehát a processzor-kihasználtság nem növelhető.

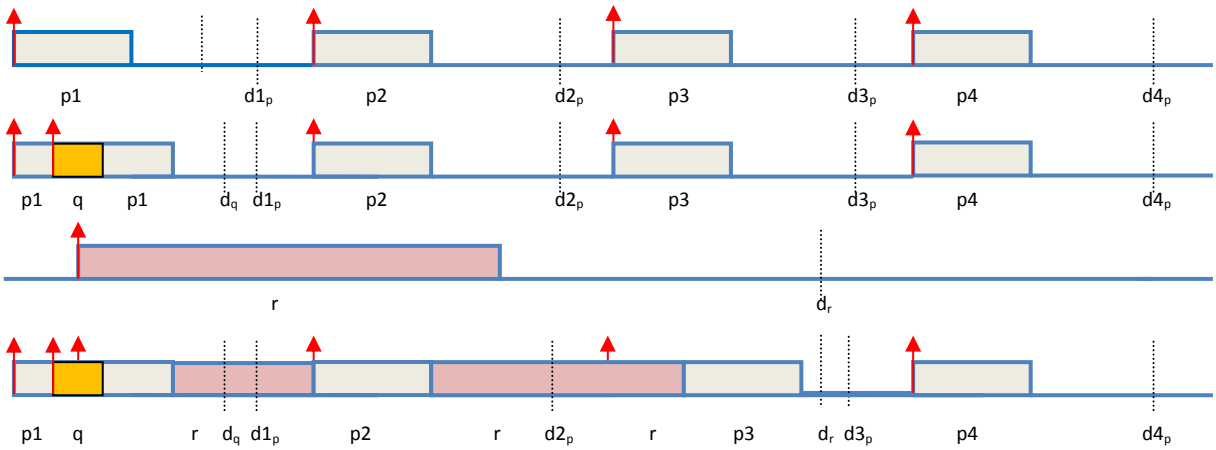
Megjegyzés:

1. Az RM eljárás alkalmazása esetén a legkedvezőtlenebb esetet a task-ok induláskor egyidejű kezdése jelenti. Azt mondjuk, hogy ilyenkor a kezdőfázis nulla. Nullától különböző kezdőfázis ütemezhetőségi szempontból kedvező.
2. Az RM eljárás alkalmazása esetén, ha csak a szükséges feltétel teljesül, az elégséges nem, akkor az ütemezhetőségi vizsgálatot a periódusidők legkisebb közös többszörösére kell elvégezni.

Példa: Ütemezhető-e RM algoritmussal az a 8 task-ból álló rendszer, amelynél a periódusidők rendre 10, 20, 30, 40, 50, 60, 70, 80, a számítási idő pedig rendre 1, 2, 3, 4, 5, 6, 7, 8? Mivel $8\left(2^{\frac{1}{8}}-1\right)=0.7240608$, és $\mu=0.8$, ezért első benyomásunk az lehet, hogy az elégséges feltétel nem teljesül, ezért nincsen garancia az ütemezhetőségre. De ha észrevesszük, hogy a periódusidő egészszámú többszörösök, amikor $\mu=1$ mellett is garantált az ütemezés, akkor a kérdésre egyértelmű igennel válaszolunk. Ha viszont a periódusidők rendre 10, 21, 32, 43, 54, 65, 76, 87, a számítási idők pedig rendre 1, 2.1, 3.2, 4.3, 5.4, 6.5, 7.6, 8.7, amikor $\mu=0.8$ éppen úgy, mint az előző esetben, akkor már csak az általános esetre vonatkozó elégséges feltétel alapján tudunk válaszolni. Az ütemezhetőségről ebben a második esetben előzetesen csak annyit tudunk mondani, hogy lehet, hogy igen, de lehet, hogy nem.

Megjegyzés: Az előzetesen kiértékelhető tesztek jelentőségét alátámasztja az a tény, hogy egyértelmű teszteredmény hiányában a vizsgálatot a periódusidők legkisebb közös többszörösére kell elvégezni, ami a megadott értékek esetén igencsak nagy szám.

Earliest Deadline First (EDF) stratégia: Periodikus, egymástól független task-ok esetére, akkor, ha $D_i \le T_i$ és C_i ismert és konstans. A prioritás hozzárendelés úgy történik, hogy futás közben a processzort (és ezzel a legnagyobb prioritást) az a task kapja, amelyiknek legközelebbi a határideje. Az eljárás preemptív. Itt is feltételezzük, hogy a task-ok közötti átkapcsolás ideje elhanyagolható. Az EDF algoritmusra elégséges teszt adható: A megadott feltételeknek eleget tevő task együttes ütemezhető, ha $\mu \le 1$, azaz a 100%-os processzor-kihasználtság elérhető. A működést az alábbi ábra illusztrálja:



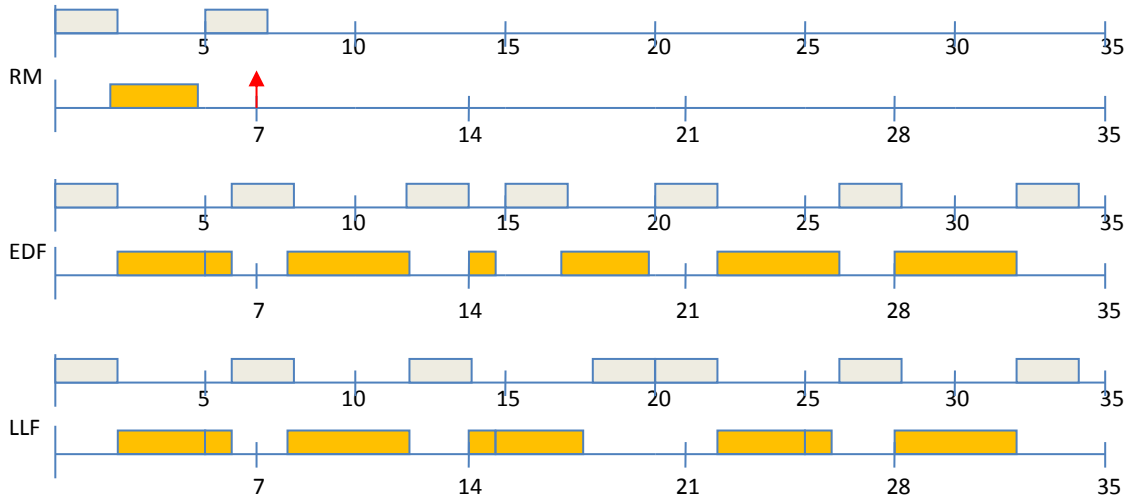
Az első sorban a p task periodikus kéréseit, futásait (p…) és a kapcsolódó határidőket (d…p) láthatjuk. A második sorban megjelenik a q task kérése a p1 futás alatt. Mivel q határideje korábbi, mint p1-é, ezért q fut le előbb. A harmadik sorban az r task kérése és határideje látszik. A negyedik sor összegzi a három task futását: q és p1 lefutása után az r task fut, hiszen nincsen az övénél korábbi határidejű. A p2 bejelentkezéskor ő lesz a legkorábbi határidejű, tehát lefut, majd r futása folytatódik. A p3-as futás határideje későbbi, mint r határideje, ezért először r futása fejeződik be, majd határidő előtt lefut p3 is.

Least Laxity First (LLF) stratégia: Az EDF-hez hasonló. Az alkalmazási feltételek azonosak, de a processzort (azaz a legnagyobb prioritást) nem a legközelebbi határidejű, hanem a legkisebb „lazasággal" (laxity-vel) rendelkező task kapja meg. Ez a vizsgálati időpontban a határidő és a még hátralévő számítási idő különbsége. A megadott feltételeknek eleget tevő task együttes ütemezhető, ha $\mu \leq 1$, azaz a 100%-os processzor-kihasználtság elérhető.
Megjegyzés: Az EDF és az LLF stratégia aperiodikus task-ok esetén is alkalmazható, de mivel a processzor-kihasználtsági tényező aperiodikus taskok esetében csak eltérő módon értelmezhető, ezért a fentiekben megadott elégséges feltétel nem alkalmazható.

Példa: Az RM és az EDF algoritmusok összehasonlítása. Két task-unk van. A periódus idejük és a határidejük megegyezik. T_1=5 ms, C_1=2 ms, T_2=7 ms, C_2=4 ms. A processzor-kihasználtsági tényező:

$$\frac{2}{5}+\frac{4}{7}=0.4+0.57=0.97.$$

A szükséges feltétel az ütemezhetőséghez teljesül, de az elégségesség csak az EDF esetén. Induláskor egyidejű kérést feltételezve a RM eljárás, az EDF eljárás és a LLF eljárás:



Látható, hogy a RM eljárás esetében a második task 7 ms-nál lekési a határidőt, az EDF és az LLF eljárással pedig ütemezhetőek lesznek a task-ok. Mind az EDF, mind az LLF eljárásnál természetesen adódó szabály, hogy azonos határidő, ill. laxity esetén a kevesebb task-váltást eredményező választással élünk. A task váltások ugyanis processzor-időt vesznek igénybe, hiszen az éppen futó task futtatási környezetét (regiszter-tartalmak) menteni kell a task-hoz rendelt, és a memóriában található Task Control Bock-ba (TCB), míg váltás keretében a futtatandó task futási környezetét pedig a memóriából a processzor regisztereibe kell tölteni. A regiszterek feltöltésére, ill. tartalmuk kimásolására a processzorok általában rendelkeznek gyors mechanizmusokkal, de értelemszerűen ezeknek is van időigényük.

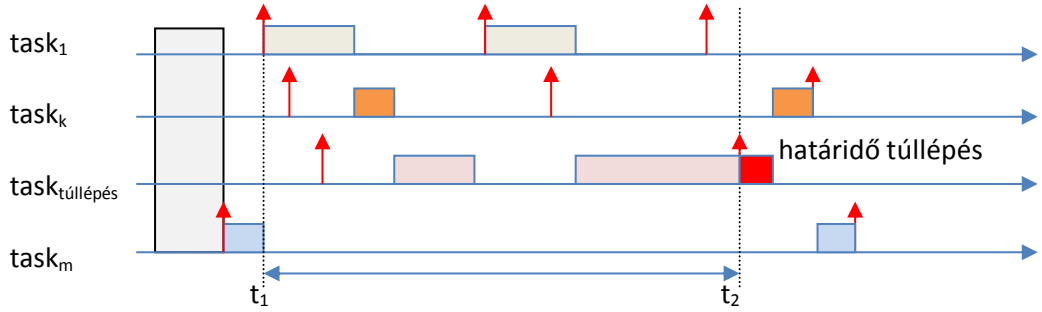
Az EDF ütemezhetőség bizonyítása
A bizonyítást periodikus task-ok és D_i=T_i esetére mutatjuk be. Az állítás a következő: egy periodikus task-készlet EDF-fel akkor és csak akkor ütemezhető, ha

$$\mu=\sum_{i=1}^{n}\frac{C_i}{T_i}\leq 1.$$

A bizonyítás: *csak akkor* rész: Azt mutatjuk meg, hogy $\mu>1$ esetében a task-készlet nem ütemezhető. Ehhez definiáljuk a $T=T_1T_2...T_n$ időtartamot, azaz a periódusidők közös többszörösét. Ezalatt az idő alatt a task-ok által igényelt processzor idő a következőképpen számítható:

$$\sum_{i=1}^{n}\frac{T}{T_i}C_i=\mu T.$$

Ha $\mu > 1$, akkor az igényelt processzoridő meghaladja a hozzáférhető processzor-időt, tehát a task-készlethez nem létezik ütemezés.
A bizonyítás: *ha* rész: Az elégségességet ellentmondással bizonyítjuk. Tegyük fel, hogy $\mu < 1$, de a task-készlet mégsem ütemezhető. A bizonyítás gondolatmenetének megértését az alábbi ábra segíti.



Az ábrán periódikus task-ok ütemezését látjuk EDF stratégia szerint. Ha feltételezésünk szerint a task-készlet nem ütemezhető, akkor kell legyen olyan task, amelyik lekési a határidőt. Legyen t_2 az az időpont, amikor a határidő túllépés bekövetkezik, és $[t_1,t_2]$ pedig a leghosszabb folyamatos processzor-használat a határidő-túllépés előtt úgy, hogy a $[t_1,t_2]$-ben csak t_2 előtti vagy azzal egyező határidejű kérések végrehajtására került sor. t_1 valamelyik periodikus kéréssel egybeeső időpont. Legyen $C_P(t_1,t_2)$ a periodikus task-ok által a $[t_1,t_2]$-ben kért teljes számítási idő, ami a következő módon számítható:

$$C_P(t_1,t_2) = \sum_{r_k \le t_1, d_k \le t_2} C_k = \sum_{i=1}^{n} \left\lfloor \frac{t_2 - t_1}{T_i} \right\rfloor C_i ,$$

ahol $\lfloor ... \rfloor$ az alsó-egészt kijelölő függvény. (Vegyük észre, hogy a legfelső sorban a harmadik kérés teljesítésére az algoritmus szabályai szerint nem kerül sor, ezért helytálló az alsó-egész hozzárendelés.) Ha ezt majoráljuk az alábbiak szerint:

$$C_P(t_1,t_2) = \sum_{r_k \le t_1, d_k \le t_2} C_k = \sum_{i=1}^{n} \left\lfloor \frac{t_2 - t_1}{T_i} \right\rfloor C_i \le \sum_{i=1}^{n} \frac{t_2 - t_1}{T_i} C_i = (t_2 - t_1)\mu ,$$

akkor, mivel t_2-ben túlléptük a határidőt, a $C_P(t_1,t_2)$ időnek nagyobbnak kell lennie, mint a rendelkezésre álló processzor-idő, azaz $(t_2\text{-}t_1)$. Ezzel

$$(t_2 - t_1) < C_P(t_1,t_2) \le (t_2 - t_1)\mu ,$$

amiből $\mu > 1$ következik, ami pedig ellentmondás, vagyis a kiinduláskor megfogalmazott állítás hamis.

Periodikus és aperiodikus task-ok együttes kezelése: elsősorban kemény valós idejű (kemény határidejű) rendszerekre koncentrálunk, de a puha valós idejű (puha határidejű) rendszerek ütemezését is kezeljük.

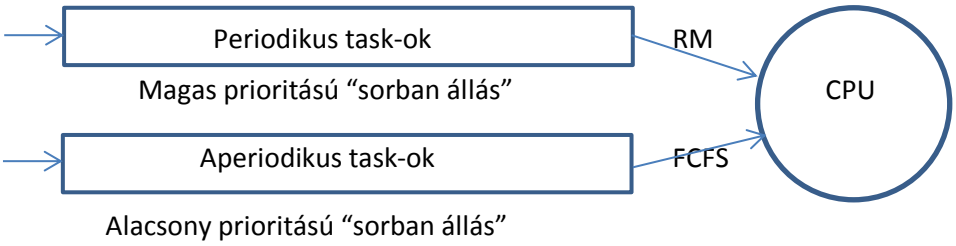
Kemény és puha határidejű taskok együttes kezelésénél két szabály alkalmazására kerül sor:

1. szabály: Minden task ütemezhető kell legyen átlagos végrehajtási és érkezési idő feltételezésével.
2. szabály: Minden kemény határidejű (kemény valós idejű) task ütemezhető kell legyen valamennyi task legkedvezőtlenebb végrehajtási (worst-case execution) és érkezési (worst-case arrival) idejének feltételezése mellett.

Az alábbiakban ismertetett módszerek esetében a következő előzetes feltételezésekkel élünk:

1. A periodikus task-ok ütemezése RM algoritmus szerint történik.
2. A periodikus task-ok egyidejűleg (nulla kezdőfázissal) indulnak és $D_i=T_i$.
3. Az aperiodikus kérések érkezési ideje ismeretlen.
4. Sporadikus task-ok esetén $D_i=T_i$.

A háttérbeni ütemezés (Background Scheduling) módszere:

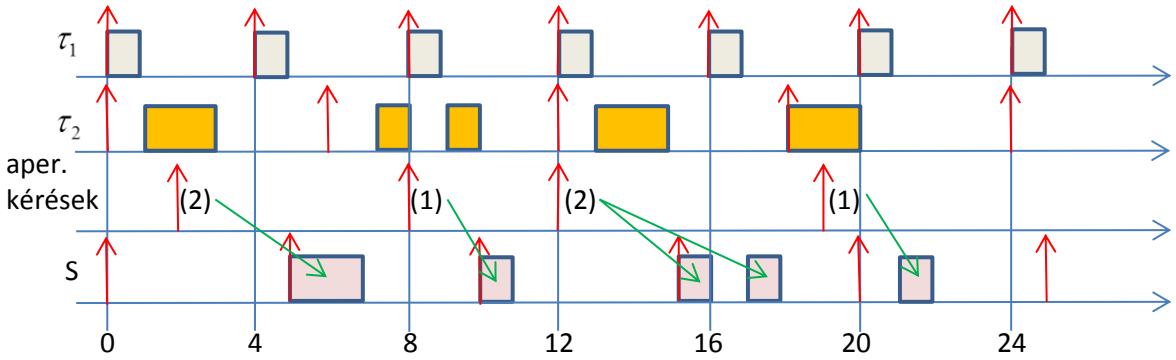


A módszer előnye egyszerűsége, hátránya pedig az, hogy az aperiodikus taks-ok válaszideje nagyon nagy lehet. (FCFS=First-Come-First-Served.)
Ha az aperiodikus task-ok esetén a válaszidő kritikus, akkor az ún. server-módszerek alkalmazása jobb eredményt adhatnak. A server-módszer az aperiodikus task-ok végrehajtásához szeparáltan biztosít processzor időt. Ennek eszköze a server-task, amelyet a periodikus task-okkal együtt ütemezünk.
Polling Server (PS): Az aperiodikus kérések teljesítése külön ún. szerver task (S) segítségével, a szerver kapacitás (T_S,C_S) terhére, független ütemezési stratégiával történik. Ha nincsen aperiodikus kérés, amikor a szerver futására sor kerülhetne, akkor a PS felfüggeszti magát, kapacitása nem őrződik meg.

Példa: Legyen T_S=5, C_S=2. Az ezen kívül ütemezendő task-ok adatait az alábbi táblázat tartalmazza:

	C	T
τ_1	1	4
τ_2	2	6

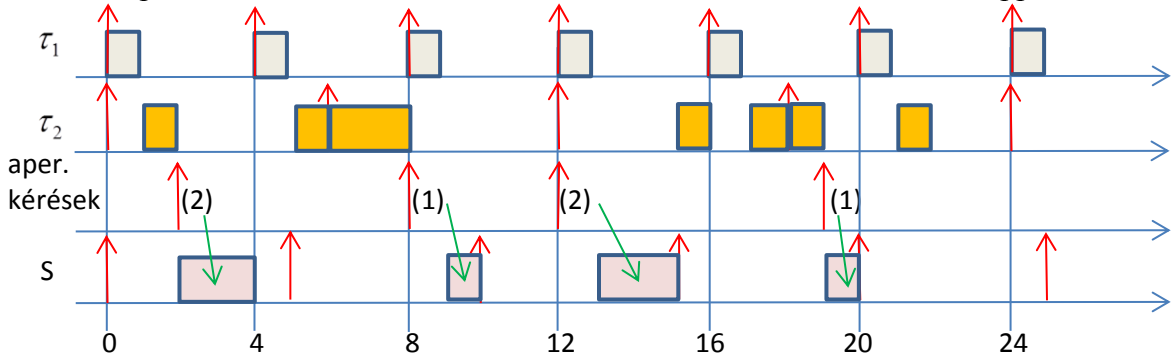
A szerver task (RM szerint) a középső prioritásra kerül. A task-ok egyidejű indítását, azaz azonos kezdőfázist feltételezve az ütemezés a következőképpen alakul:



Látható, hogy a legkedvezőtlenebb esetben az aperiodikus kérések teljesítésére – a magasabb prioritású task-ok által okozott interferenciát nem számítva – csak egy teljes szerver task periódus elteltével kerül sor.

Deferrable Server (DS): Az aperiodikus kérések teljesítése külön ún. szerver task (S) segítségével, a szerver kapacitás (T_S,C_S) terhére, független ütemezési stratégiával történik. Ha nincsen aperiodikus kérés, amikor a szerver futására sor kerülhetne, akkor a DS task futása halasztódik, kapacitását a periódus végéig megőrzi. Ezzel a módszerrel az aperiodikus task-okra sokkal jobb válaszidők érhetők el.

Példa: Az előző példa adataival, és futtatási feltételeivel az ütemezés a következőképpen alakul:



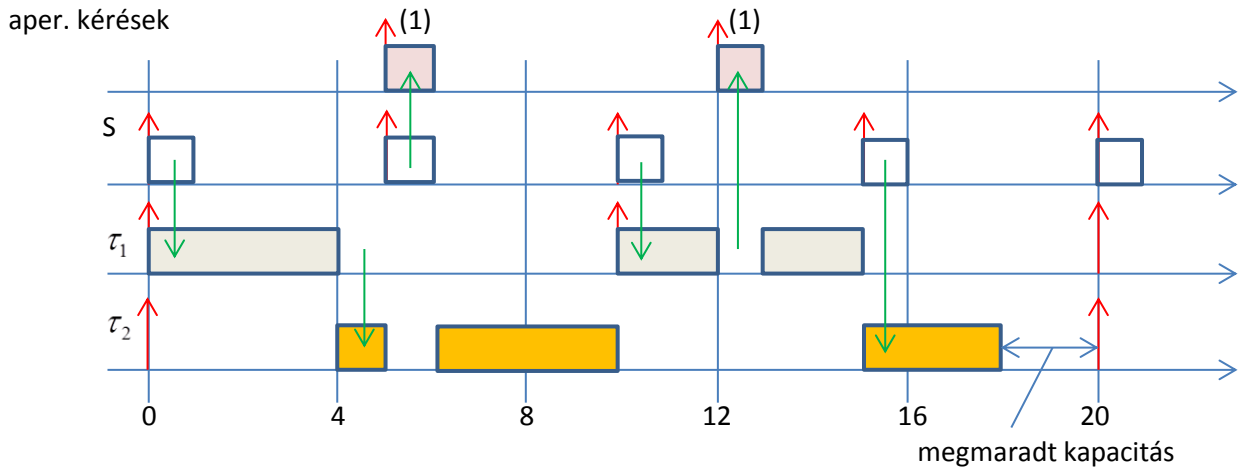
Látható, hogy a szerver task prioritási szintjétől is függően az aperiodikus kérések teljesítése lényegesen jobb válaszidők mellett történik. (A példában a szerver task ütemezése az előzővel azonos módon, RM stratégiával történt.)

Priority Exchange Server (PE): Olyan, mint a DS, magas prioritáson futó szervert használ, de másképpen őrzi a kapacitást: alacsonyabb prioritású periodikus task kapacitásával cseréli ki.

Példa: Legyen T_S=5, C_S=1. Az ezen kívül ütemezendő task-ok adatait az alábbi táblázat tartalmazza:

	C	T
τ_1	4	10
τ_2	8	20

A szerver task (RM szerint) a legmagasabb prioritásra kerül. Vegyük észre, hogy a processzor kihasználtsági tényező: $\mu = \frac{1}{5} + \frac{4}{10} + \frac{8}{20} = 1$. A task-ok egyidejű indítását, azaz azonos kezdőfázist feltételezve az ütemezés a következőképpen alakul:

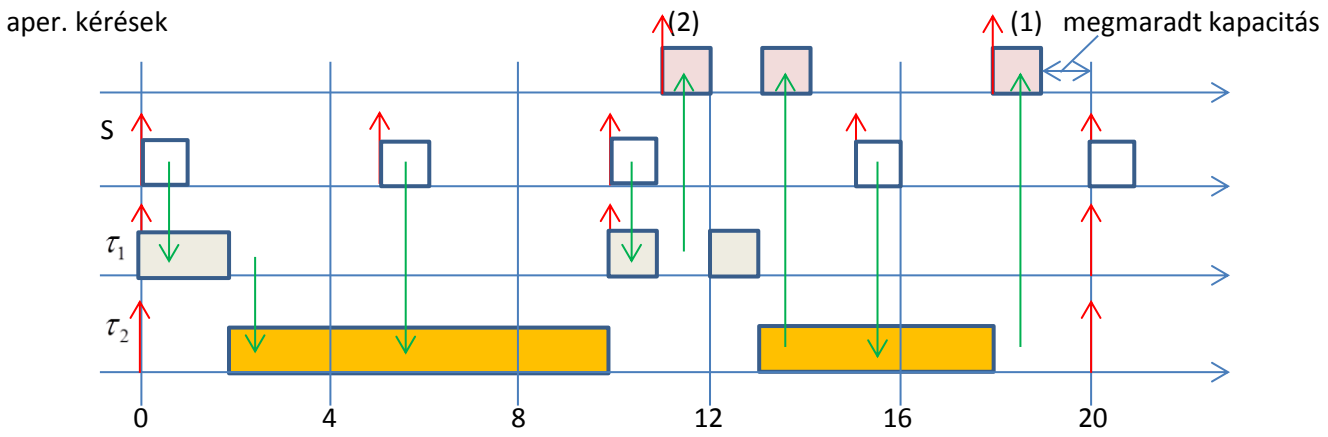


Mivel nincsen előzetesen aperiodikus kérés, az első ütemben megjelenő szerver kapacitást felhasználja a τ_1 task. Ennek következtében τ_2 task korábban indulhat, vagyis a szerver kapacitás ide kerül. A második ütemben megjelenő szerver kapacitást közvetlenül felhasználjuk. A harmadik ütemben megjelenő szerver kapacitást a τ_1 task használja fel, amit a második aperiodikus kérés kiszolgálására visszacserél. A negyedik ütemben érkező szerver kapacitást a τ_2 task hasznosítja. Ezzel együtt két periódusnyi szerver kapacitás "halmozódik fel" a τ_2 végrehajtásánál, ami mozgósítható lenne, ha lenne további aperiodikus kérés.

Példa: Legyen T_S=5, C_S=1. Az ezen kívül ütemezendő task-ok adatait az alábbi táblázat tartalmazza:

	C	T
τ_1	2	10
τ_2	12	20

A szerver task (RM szerint) a legmagasabb prioritásra kerül. Vegyük észre, hogy a processzor kihasználtsági tényező: $\mu = \frac{1}{5} + \frac{2}{10} + \frac{12}{20} = 1$. A task-ok egyidejű indítását, azaz azonos kezdőfázist feltételezve az ütemezés a következőképpen alakul:



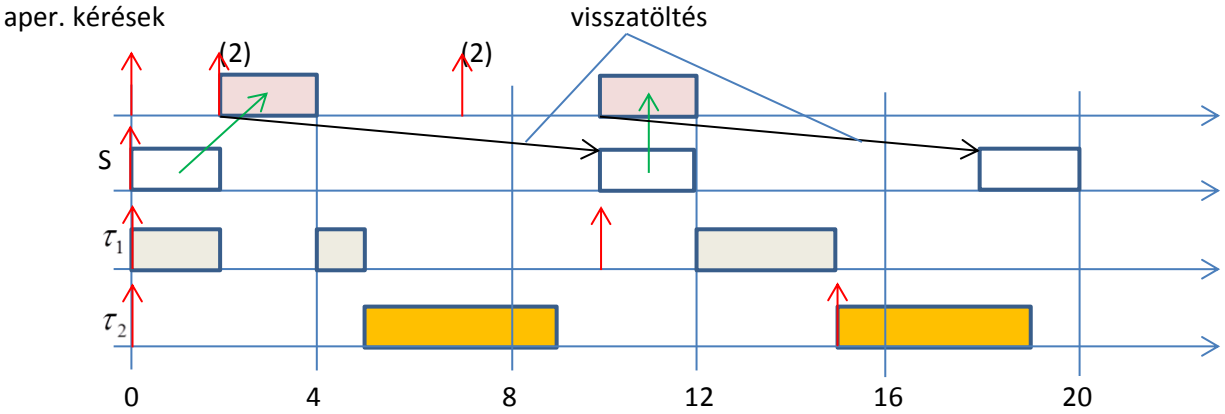
Az ábrán nyomon követhetjük a szerver kapacitások felhasználásának módozatait, és azt is megfigyelhetjük, a kapacitás másik task-hoz történő áthelyezése azzal is jár, hogy az áthelyezett kapacitás a befogadó task prioritásán használható fel. Lásd: a 11. időpillanatban kért 2 időegységnyi idő első fele a τ_1 task-nál lelhető fel, a második fele pedig a τ_2 task-nál. Az első fél futását követően a τ_1 task fut tovább, majd csak annak lefutása után áll rendelkezésre a τ_2 task prioritásán elérhető második fél. Itt egy periódusnyi szerver kapacitás "halmozódik fel" a τ_2 végrehajtásánál, ami mozgósítható lenne, ha lenne további aperiodikus kérés.

Sporadic Server (SS): Olyan, mint a DS, megőrzi kapacitását, de másképpen tölti vissza: nem a periódus elején, hanem a felhasználást követően. A felhasználás kezdetétől egy szerver task periódusnyira jelenik meg a szerver kapacitás.

Példa: Legyen T_S=8, C_S=2. Az ezen kívül ütemezendő task-ok adatait az alábbi táblázat tartalmazza:

	C	T
τ_1	3	10
τ_2	4	15

A szerver task (RM szerint) a legmagasabb prioritásra kerül. Vegyük észre, hogy a processzor kihasználtsági tényező: $\mu = \frac{1}{5} + \frac{2}{10} + \frac{12}{20} = 1$. A task-ok egyidejű indítását, azaz azonos kezdőfázist feltételezve az ütemezés a következőképpen alakul:

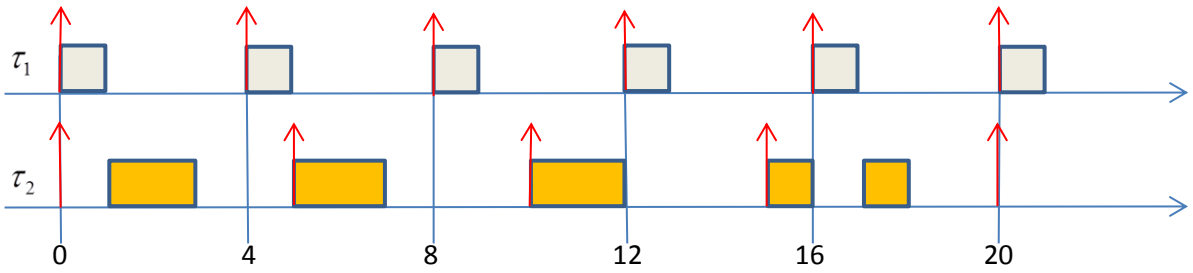


Slack stealing: Az egyes task-ok végrehajtása között fellelhető szabadidőt, "lazaságot" használjuk fel. Sokkal jobb válaszidőt ad, mint a DS, a PE vagy a SS eljárás. A számítási, megvalósítási komplexitást, és a memóriaigényt illetően a leginkább ráfordítás igényes eljárás.

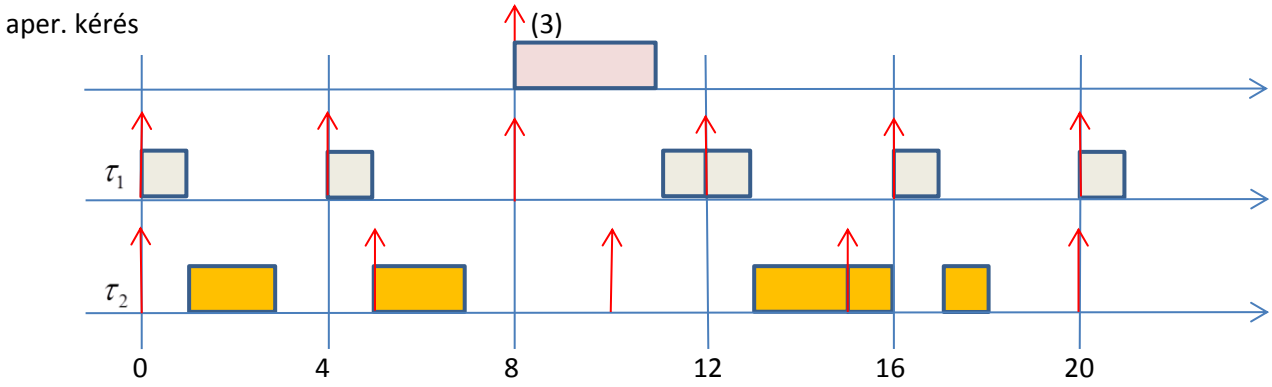
Példa:

	C	T
τ_1	1	4
τ_2	2	5

A normál ütemezés RM szerint:



Aperiodikus kérés érkezését követően kiszámításra kerül, hogy mennyi tartalék/"lazaság" van a rendszerben, és azt megkapja az aperiodikus task a legnagyobb prioritással az alábbiak szerint:



Dual Priority Scheduling: Három prioritási szint van: alacsony, közepes és magas. Kezdetben a kemény valós idejű task-ok az alacsony prioritáson futnak. A puha valós idejű task-ok és az aperiodikus task-ok a közepes prioritási szintre kerülnek. A kemény valós idejű taskok a határidő előtt $X_i = D_i - R_i$, ún. promóciós idővel átkerülnek a magas prioritásra, hogy a határidőt be tudják tartani. ($R_i = B_i + C_i + I_i$). Az alacsony, közepes és magas szintek értelemszerűen önmagukon belül további prioritási szintekre bonthatók.

Megjegyzés: A fentiekben bemutatott szerver megoldások rendre a RM ütemezési stratégiát követve működnek. Hasonló megoldások származtathatóak az EDF ütemezési stratégiára alapozva, de ezek bemutatásától itt eltekintünk. Míg az előzőeket fix prioritású, az utóbbiakat dinamikus prioritású szervereknek nevezzük.

2. Ütemezés (folytatás)

Ütemezhetőség $D_i<T_i$ esetben: Az eddigi vizsgálatok és állítások szinte kivétel nélkül a $D_i=T_i$ esethez tartoztak. Ha a határidő kisebb, mint a periódusidő, akkor a prioritás hozzárendelés történhet a határidők alapján. Ennek jellegzetes formája a *Deadline Monotonic* (DM) algoritmus. Ehhez természetesen a

$$\sum_{i=1}^{n}\frac{C_i}{D_i}\le n\left(2^{\frac{1}{n}}-1\right)$$

elégséges ütemezhetőségi feltétel, de nem szükséges, pesszimisztikus. Kevésbé pesszimisztikus, ha egyidejű indítást feltételezve (mivel s processzor igény szempontjából ez a legkedvezőtlenebb) minden task-ra megvizsgáljuk a $C_i+I_i\le D_i$ feltétel teljesülését. Itt $I_i=\sum_{\forall k\in hp_i}\left\lceil\frac{D_i}{T_k}\right\rceil C_k$. Ez a feltétel is elégséges, de nem szükséges. A szükséges és elégséges feltételt a már korábban megtárgyalt válaszidő kifejezés teljesülése adja:

$$R_i=C_i+I_i=C_i+\sum_{\forall k\in hp_i}\left\lceil\frac{R_i}{T_k}\right\rceil C_k<D_i.$$

Ha az EDF algoritmust $D_i<T_i$ mellett használjuk, akkor közvetlenül a processzor kihasználtsági tényezőt nem tudjuk használni. Helyette az ún. processzor-igény módszer (processor demand approach) ajánlható. Ezt először a $D_i=T_i$ esetre mutatjuk be. Általában egy tetszőleges [t,t+L] intervallumban egy τ_i task processzor igénye a t+L időpontig vagy azt megelőzően befejezendő feladatokhoz szükséges processzor idő. Olyan periodikus task-ok esetében, amelyek t=0 időpontban kezdenek futni, és amelyekre $D_i=T_i$, tetszőleges [0,L] intervallumban a teljes processzor idő

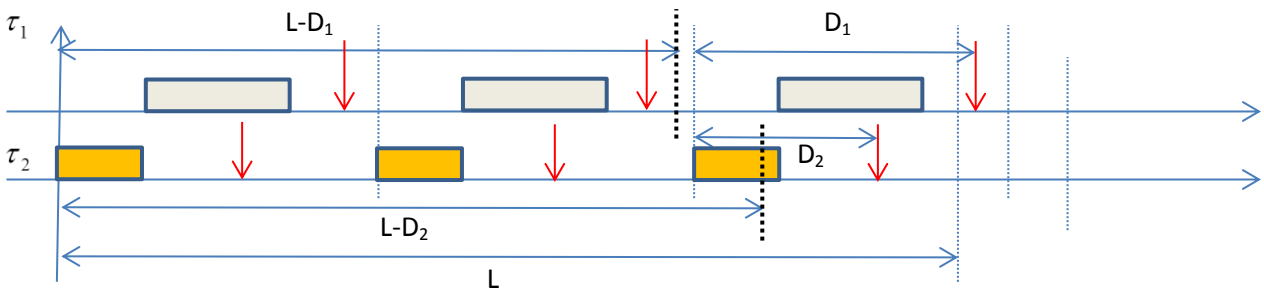
$$C_p(0,L)=\sum_{k=1}^{n}\left\lfloor\frac{L}{T_k}\right\rfloor C_k.$$

Állítás: Egy periodikus task-készlet akkor és csak akkor ütemezhető EDF algoritmussal, ha minden L>0 esetében

$$L\ge\sum_{k=1}^{n}\left\lfloor\frac{L}{T_k}\right\rfloor C_k. \qquad (*)$$

Bizonyítás: Egyrészt, mivel $\mu=\sum_{i=1}^{n}\frac{C_i}{T_i}\le 1$, ezért $L\ge\mu L=\sum_{k=1}^{n}\left(\frac{L}{T_k}\right)C_k\ge\sum_{k=1}^{n}\left\lfloor\frac{L}{T_k}\right\rfloor C_k$. Másrészt, ha $\mu>1$, akkor van olyan L>0, amelyre (*) nem áll fent, ugyanis például L-et a T_1, T_2, …, T_n legkisebb közös többszörösére választva: $L<\mu L=\sum_{k=1}^{n}\left(\frac{L}{T_k}\right)C_k=\sum_{k=1}^{n}\left\lfloor\frac{L}{T_k}\right\rfloor C_k$.

Ha $D_i<T_i$, akkor a $C_p(0,L)$ számítása a fentiektől eltérő módon történik. Ehhez tekintsük a következő ábrán két task esetét, melyek az egyszerűség kedvéért legyenek azonos periodicitásúak, de eltérő határidejűek:



Az ábra alapján a τ_1 task processzor idő igénye, figyelembe véve, hogy a harmadik periódus határideje már kívül esik az L hosszúságú intervallumon a $C_1(0,L)=\left\lfloor \frac{L}{T_1} \right\rfloor C_1$ összefüggéssel adható meg, míg ugyanez a τ_2 task esetében $C_2(0,L)=\left(\left\lfloor \frac{L}{T_2} \right\rfloor +1\right)C_2$ összefüggéssel adható meg, mert itt a harmadik periódus határideje az L hosszúságú intervallumon belül esik. Az ábra segítségével könnyen belátható, hogy a két eset együtt kezelhető, ha a következő módon számolunk:

$$C_i(0,L)=\left(\left\lfloor \frac{L-D_i}{T_i} \right\rfloor +1\right)C_i .$$

Ennek felhasználásával:

Állítás: Egy periodikus task-készlet akkor és csak akkor ütemezhető az EDF algoritmussal, ha minden L>0 esetén

$$L \geq \sum_{k=1}^{n}\left(\left\lfloor \frac{L-D_k}{T_k} \right\rfloor +1\right)C_k .$$

Összefoglalva:

	$D_i=T_i$	**$D_i<T_i$**
statikus prioritás	**RM** Processzor kihasználtsági megközelítés $\mu \leq n\left(2^{\frac{1}{n}}-1\right)$	**DM** Válaszidő megközelítés $\forall i-re \quad R_i=C_i+\sum_{\forall k\in hp_i}\left\lceil \frac{R_i}{T_k} \right\rceil C_k \leq D_i$
dinamikus prioritás	**EDF** Processzor kihasználtsági megközelítés $\mu \leq 1$	**EDF** Processzor-igény megközelítés $\forall L>0 \quad L \geq \sum_{k=1}^{n}\left(\left\lfloor \frac{L-D_k}{T_k} \right\rfloor +1\right)C_k$

Kiegészítések a válaszidő képletéhez:

1. Kooperatív ütemezés: A task futásának adott pontján szempont lehet a task futás mielőbbi befejezése. Ennek eszköze a preempció/futás megszakítás tiltása a task futásának a végéig. Ha ennek időtartama F_i, akkor a válaszidő $R_i=R_i+F_i$ formában írható, ahol

   $$R_i^{'}=B_i+C_i-F_i+\sum_{\forall k\in hp_i}\left\lceil \frac{R_i^{'}}{T_k} \right\rceil C_k .$$

   Ilyenkor az utolsó szakasz, ha fut, akkor a legmagasabb prioritáson fut.

2. Hibatűrés: exception handler, recovery block, általában többlet futást igénylő hibakezelés: C_i^f extra számítási idő minden task esetében.
   Egyetlen hibára:

   $$R_i=B_i+C_i+\sum_{\forall k\in hp_i}\left\lceil \frac{R_i}{T_k} \right\rceil C_k+\underbrace{\max}_{k\in hep_i}C_k^f .$$

   Mivel nem tudjuk, hogy a vizsgált és a magasabb prioritású task-ok melyike hibásodik meg, ezért a leghosszabb futási idejű hibakezelő programot választjuk. (*hep=higher or equal priority*)
   F hibára:

   $$R_i=B_i+C_i+\sum_{\forall k\in hp_i}\left\lceil \frac{R_i}{T_k} \right\rceil C_k+\underbrace{\max}_{k\in hep_i}\left(FC_k^f\right).$$

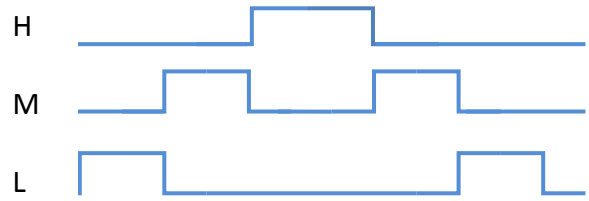
Ha T_f jelöli két hiba előfordulás közötti legrövidebb időt (inter arrival time):

$$R_i = B_i + C_i + \sum_{\forall k \in hp_i} \left\lceil \frac{R_i}{T_k} \right\rceil C_k + \underbrace{\max}_{k \in hep_i} \left(\left\lceil \frac{R_i}{T_f} \right\rceil C_k^f \right).$$

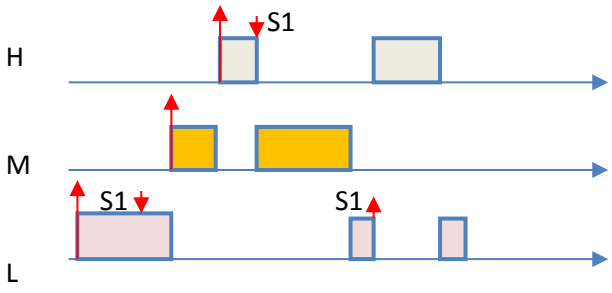
3. Az óra handler és az átkapcsolások többletidő-igénye:
   - Az ütemező sok esetben óra interrupt-ra indul (tick scheduling), ilyenkor a kérés beérkezése és az óraütés között eltelt idővel a válaszidő megnövelendő. Ha a beérkezés időpontja külön nem mérhető, akkor két óraütés között eltelt idővel növelendő a válaszidő: ez a legrosszabb eset.
   - Ha az ütemező egy task-ot futó állapotba helyez, akkor először a processzor regisztereiben lévő tartalmakat menteni kell, majd a processzor regisztereibe bele kell írni a task futtatási környezetét megadó értékeket, és csak utána futtatható a kód. A válaszidő tehát növelendő a task környezet "kapcsolási" (context switch) idejével. A task futását megszakító magasabb prioritású taskok futtatásakor is váltani kell a futtatási környezetet, ezért a magasabb prioritású task-ok számítási idejéhez hozzá kell adni átkapcsolás és a visszakapcsolás időigényét.

Ütemezés nem független task-ok esetén

Az ún. time-sharing rendszerek kivételével, ahol egymástól független felhasználók osztoznak a számítógép processzor kapacitásán, az alkalmazások túlnyomó többsége azzal jellemezhető, hogy a task-ok futása egymástól nem teljesen független, a task-ok egymással kommunikálnak, egymással adatot cserélnek, egymás számítási eredményeire várnak, közös erőforrást használnak, ezért előfordulhat, hogy magasabb prioritású futását alacsonyabb prioritású akadályozza (blokkolja). Idézzük fel az előző óra prioritásos ütemezést illusztráló ábráját! Ha a vázolt szituációban a L task olyan erőforrást használ, amit később a H task is használni szeretne, akkor előfordulhat, hogy várakoznia kell mindaddig, amíg az erőforrás újból szabaddá nem válik.



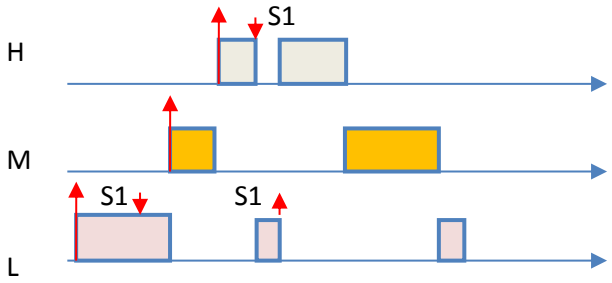
Ezt illusztrálja a következő ábra:



Az ábrán látható, hogy az L task az S1 szemaforral védett közös erőforrás használatába kezd (a szemafort „foglaltra"/"pirosra" állítja, ún. kritikus szakaszba lép), de az M task a futását megszakítja. Az M task futása a H task kérésének megérkezése után megszakad, majd a H fut, de szeretné használni az L task által használatba vett közös erőforrást. Mivel az nem lehetséges addig, amíg az L a közös erőforráson elvégzendő műveletekkel nem végez, ezért futása megszakad, azt mondjuk, hogy blokkolódik. A blokkolódás addig tart, amíg L újra sorra nem kerül, és fel nem szabadítja a közös erőforrást (az S1 szemafor „szabadra"/"zöldre" állításával). Látható, hogy a H task végrehajtása jelentős késedelmet szenvedhet, mert az L task csak azután

jut processzorhoz, miután az M lefut. A jelenséget ***prioritás inverziónak*** nevezzük, mert látszólag az M és a H taskok prioritásai felcserélődnek.

A prioritás öröklés algoritmus (Priority Inheritance Protocol, PIP)**:** A prioritás inverzió elkerülése úgy lehetséges, hogy a H task kritikus szakaszba lépési szándékának megjelenésekor az L task ideiglenesen „megörökli" a H task prioritását (dinamikus prioritás), hogy mielőbb fejezze be a kritikus szakaszbeli teendőit, majd ezt követően visszatér az eredeti (statikus) prioritási rend. Ilyenkor a futás a következőképpen alakul:

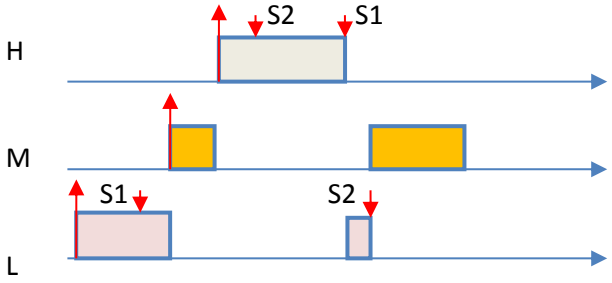


Látható, hogy a H task válaszideje lényegesen csökken, a blokkolási idő a legkedvezőtlenebb esetben az L task kritikus szakaszban töltött idejével egyenlő.

A blokkolási idő (B_i) figyelembevétele válaszidő számításnál:

$$R_i = C_i + B_i + I_i = C_i + B_i + \sum_{\forall k \in hp_i} \left\lceil \frac{R_i}{T_k} \right\rceil C_k$$

A prioritás öröklés algoritmus több erőforrás esetén kiegészítésre/módosításra szorul, amit a következő ábra illusztrál:



Az L task az S1 szemaforral védett kritikus szakaszba kerül. Az L task a kritikus szakaszon belül egy további, az S2 szemaforral védett erőforráshoz fog fordulni. Ezt az erőforrást a H task – az ábrán látható időviszonyok mellett – ugyancsak használja. Amikor a H task az S1 szemaforral védett erőforráshoz fordul, akkor blokkolódni kényszerül: az L tasknak - örökölve a H prioritást - előbb be kell fejeznie a kritikus szakaszban lévő kódrészének futtatását. Azonban az S2 szemaforhoz fordulva kialakul az egymásra várás, az ún. holtpont (deadlock). Ennek megakadályozására dolgozták ki a prioritás felső-határ/plafon (ceiling) protokollokat.

Prioritás felső-határ (plafon) protokoll (Priority Ceiling Protocol, PCP): A közös erőforrások kezelése kölcsönös kizárással/kritikus szakasszal történik. Ennek megvalósítására szemaforokat használunk, amelyek jelzik az erőforrás szabad vagy foglalt állapotát. Szabad állapotú erőforrás használata a kéréskor azonnal lehetséges, a foglalt állapotú erőforrás a kérő task-ot blokkolja. A blokkolt task az erőforrás szabaddá válásakor „ébred fel", és válik futtathatóvá. Futni akkor fog, amikor az ütemező futó állapotba helyezi.

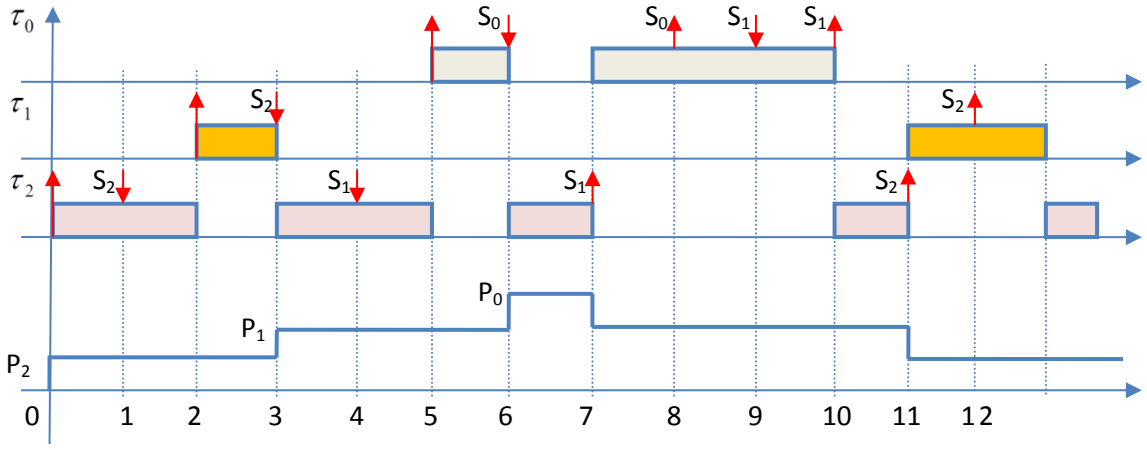
A PCP protokoll:

- Prioritásos rendszert működtetünk, és feltesszük, hogy éppen a τ_i jelű task fut.
- Minden S_k szemafornak van $C(S_k)$ prioritás plafonja, ami egyenlő a futása során az S_k szemafort foglalt állapotba helyezni képes task-ok között a legmagasabb prioritással rendelkező task prioritásával.
- Jelölje S^* a legnagyobb $C(S^*)$ prioritás plafonú szemafort a τ_i-től különböző task-ok által foglaltra állított szemaforok közül.
- Ahhoz, hogy egy S_k szemafor által védett kritikus szakaszba lépjünk, a τ_i task prioritása (P_i) magasabb kell legyen $C(S^*)$-nál. Ha $P_i \leq C(S^*)$, akkor a τ_i task felfüggeszti a futását, blokkolódik.
- A τ_i task blokkolódása esetén az ő prioritását a szemafort foglaltra állító τ_k task megörökli.
- Amikor a τ_k task a τ_i task blokkolódását előidéző szemafor foglaltságát megszünteti, akkor örökölt prioritását elveszíti, az ütemező a taskok ütemezését ennek megfelelően módosítja.

Megjegyzés:

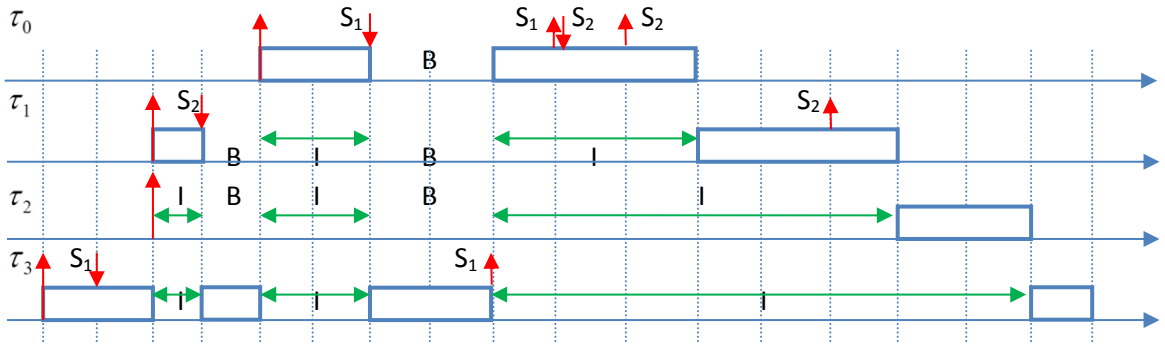
Az első erőforrás lefoglalása engedélyezett. A protokoll hatása az, hogy egy második erőforrás lefoglalása csak akkor lehetséges, ha nincsen magasabb prioritású task, amely mind a két erőforrást használja. Ebből következik, hogy a leghosszabb idő, amivel egy task blokkolható, egyenlő az alacsonyabb prioritású task-okban a leghosszabb kritikus szakasz végrehajtási idejével. Ezt az időt írjuk a worst-case válaszidő számítás képletében szereplő B_i helyére.

Példa: A futtatandó task-ok csökkenő prioritású sorrendben: τ_0, τ_1, τ_2. A prioritásaik: P_0, P_1 és P_2. Az erőforrásokat S_0, S_1 és S_2 szemaforok őrzik. Prioritás plafonjaik: $C(S_0) = P_0$, $C(S_1) = P_0$, $C(S_2) = P_1$.



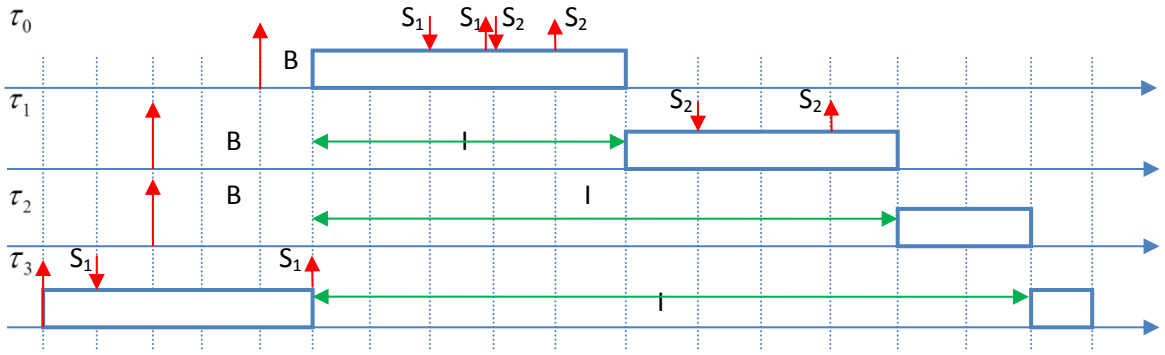
Figyeljük meg az ábrán, hogy a protokoll hatására τ_0 annak ellenére blokkolódik az S_0 szemaforral védett erőforrás használata előtt, hogy az erőforrás szabad. Ennek az a kiváltó oka, hogy a τ_2 task a τ_0-val azonos prioritás plafonú S_1 szemaforral védett kritikus szakaszban tartózkodik.

Példa: A futtatandó task-ok csökkenő prioritású sorrendben: $\tau_0, \tau_1, \tau_2, \tau_3$. 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-vel az interferencia intervallumokat, B-vel pedig a blokkolási intervallumokat jelöltük. Ezeknek az összege adja az adott task tényleges blokkolási idejét, aminek a maximuma a legkedvezőtlenebb esetben a τ_3 task kritikus szakaszának processzoridő igényével egyezik meg.

Azonnali prioritás felső-határ (plafon) protokoll (Immediate Priority Ceiling Protocol, IPCP): A protokoll lényege, hogy a taskok a kritikus szakaszba lépéskor azonnal a kritikus szakaszt védő szemafor prioritás plafonjának megfelelő dinamikus prioritást kapnak. Ennek értelmében az alábbi ábrán a τ_3 task 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 task-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 task válaszideje egy időegységgel csökkent.



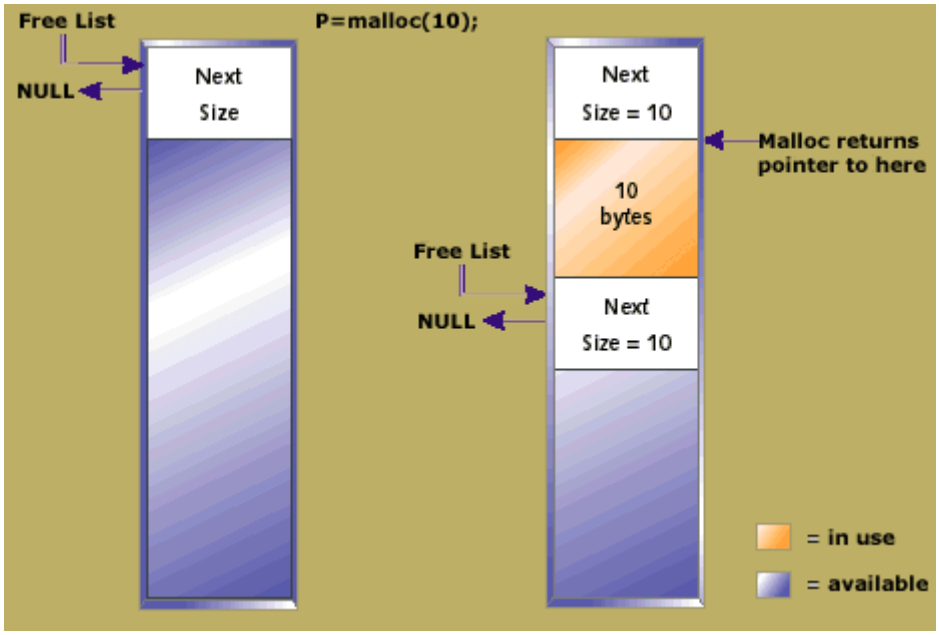
Az IPCP elnevezése a POSIX szabványban Priority Protect Protocol, a Real-Time Java-ban pedig Priority Ceiling Emulation.

3. Memória menedzsment

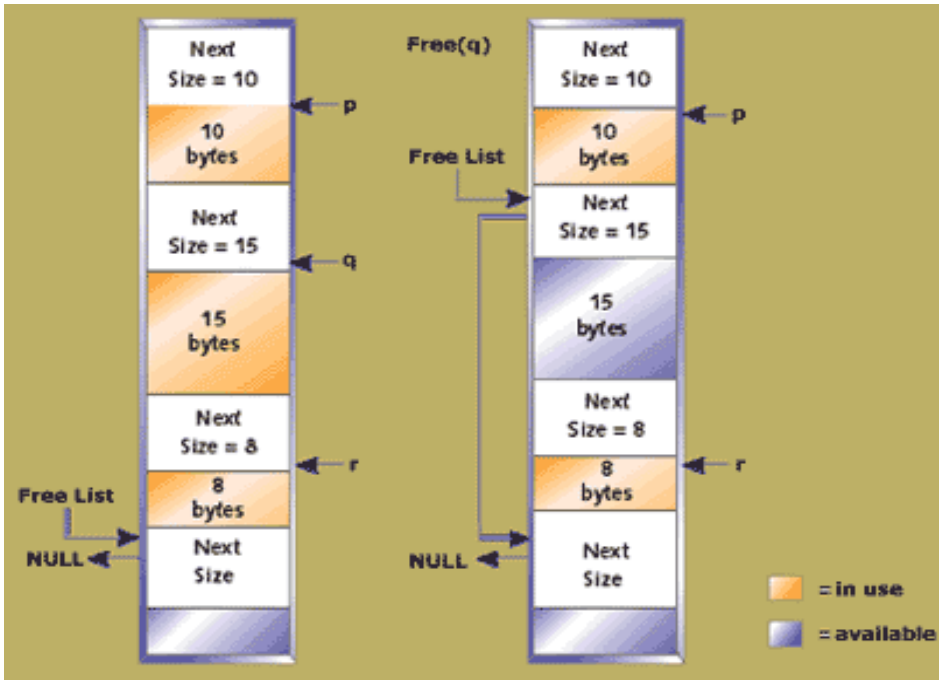
A nem független task-ok ütemezési kérdései kapcsán szembesültünk az erőforrások kezelésének néhány problémájával. Ebben a fejezetben az erőforrások közül a memóriára vonatkozó néhány kérdést tárgyaljuk a beágyazott rendszerek szempontjából. Előre bocsátva: a beágyazott rendszerek jelentős részénél nem 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.

- **Statikus memória allokáció:** minden fixen kiosztva. Előny: egy csomó hibaforrás kizárva. Viszont 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*)), 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.



Az ábrán az első 10 byte foglalása és annak adminisztrálása látható. (Ezt és a következő ábrát Niall Murphy (Panelsoft): Memory Management c. előadása tartalmazza, ami több helyen, így például az Embedded Systems Conference Europe 2000-en hangzott el.)



Az ábra bal oldalán 10, 15 és 8 byte foglalása és annak adminisztrálása látható. Az ábra jobb oldalán a 15 byte-os blokk felszabadítása és annak következményei láthatók.

Példa*:* UNIX alkalmazásokban mérték, hogy az allokációk 90%-ában 6-féle méret, 99.9%-ában pedig 141-féle méret fordult elő. Beágyazott rendszerekben nincsenek file-ok, kevés a szöveg-kezelés, valószínűleg ennél jobb a helyzet.

Példák felszabadítási stratégiákra: (1) a felszabadított tartomány címe a Free List elejére teendő, ezáltal a végrehajtási idő rögzített hosszúságú lesz. (2) a felszabadított tartományokat cím szerinti sorrendbe állítani - a végrehajtási idő ilyenkor a lista hosszával változik. Rendezett listákban a felszabadított blokkok gyorsabban összevonhatók - ami segít a feldarabolódás elkerülésében.

Példák foglalási stratégiákra: (1) first fit (gyors), (2) best fit (kimerítő keresés)

Megjegyzés: Az idő múlásával mind a felszabadításnál, mind a foglalásnál a (2) szerinti változat futási ideje nő: egy idő után már "szinte" csak ez fut.

Konklúzió: Nagy megbízhatóság esetén beágyazott rendszerekben nem használható a heap alapú menedzsment. UNIX alkalmazásokban, körültekintő tervezés esetén, a töredezés csak 1% szintű veszteséget jelent a tapasztaltok szerint, de nincs igazán garancia.

Javaslat: korlátozott heap használat: statikus allokáció: (1) csak az inicializáláskor használjuk a malloc( ) függvényt és nincs felszabadítás. (2) célszerű saját programot írni: ezzel a blokk header elkerülhető (pl. salloc( ) függvény (statikus allokáció)). (3) az inicializálást követően a salloc( ) tiltva van.

Javaslat: dinamikus allokáció, de fix blokk mérettel. (partícióknak is nevezik).

- **Multitasking:** Minden task-nak saját stack-je kell legyen, heap lehet saját, vagy nem saját függetlenül attól, hogy statikus, partíció jellegű, vagy általános allokációs módszert használtunk. (1) ha minden task-nak saját heap-je van, akkor a méretbeállítás problémás. (2) ha közös a heap, akkor a hozzáférésnél biztosítandó a kölcsönös kizárás. (3) ha közös a heap, akkor lehetséges, hogy az egyik task által foglalt memóriát a másiknak kell felszabadítania. (4) ha a taskok között memória tartalmakat mozgatunk, akkor jó tudni, hogy aktuálisan melyik task birtokolja a memóriát. (4) közös heap esetén is javasolható a task-onkénti statisztika készítése a rendszer működésének jobb megértése érdekében.
- **Átvett könyvtárak memória használata:** Problémák: (1) memóriát a könyvtári programnak kell foglalnia. (2) memóriát felszabadítani az alkalmazás tud. (3) a könyvtári programhoz is rendelhetünk

statikus memóriát, de ilyenkor nem lesz újrahívható, bár ez sokszor kell. (4) mindezekre a könyvtár írójának kellene gondolnia: esetleg saját könyvtári rutinok felkínálása a memória felszabadítására (ún. Pluggable memory management).

- **Automatikus szemétgyűjtés:** (automatic garbage collection): a Java, LISP, Smalltalk nyelvekben van ilyen. Két alapvető mechanizmus: (1) a pointerek objektumként megszüntethetik magukat, ha nincs rájuk szükség. (2) az egész memóriát átnézzük, hogy van-e az adott memória blokkra hivatkozó pointer benne. Ha nincs, akkor a blokk felszabadítható. Megjegyzés: a C++-ban létrehozható ún. smart pointer, amely segíti a szemétgyűjtés megvalósítását.

4. Időmérés, időszolgáltatás, óra-szinkronizáció

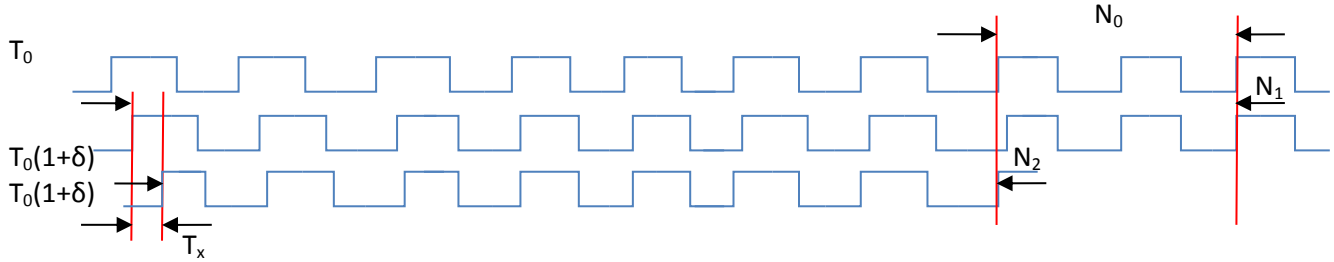
Időmérés eszközei és módszerei:
(1) *Időmérés elektronikus számlálóval:* Precíz óragenerátor jelének számlálása a megmérendő ideig az alábbi ábra szerint:

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. A közelítő egyenlőség arra utal, hogy N mindig egész, míg $T_x f_0$ nem feltétlenül az. Ebből fakad a mérés ún. kvantálási hibája. A mérés elvben is csak legfeljebb akkor pontos, ha T_x az $\frac{1}{f_0}$ egészszámú többszöröse. Az időmérés (worst-case) relatív hibája az alábbi összefüggéssel adható meg:

$$\left|\frac{\Delta T_x}{T_x}\right| \cong \left|\frac{1}{N}\right| + \left|\frac{\Delta f_0}{f_0}\right|,$$

azaz a pontos méréshez pontos és a mérendő időhöz képest nagy frekvenciájú óra szükséges, hogy N értéke kellően nagy legyen. 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 $\frac{dT_x}{T_x} = \frac{dN}{N} - \frac{df_0}{f_0}$ értéket kapunk differenciális megváltozások esetére. Természetesen N csak diszkrét értékeket vehet fel, ezért megváltozása csak ± 1 egészszámú többszöröse lehet. Bár a képlet szerint N és f_0 relatív megváltozása egymást kompenzáló hatású tud lenni, 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

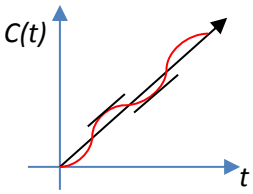


Ezek jelét egy szabadon futó T_0 periódusidejű, kvarcpontosságú óra jelével hasonlítjuk össze, figyelve a felfutó élek egybeesését. A mérendő időtartam kezdetétől az első koincidenciáig eltelt idő $N_1T_0(1+\delta)$, a mérendő időtartam végétől az első koincidenciáig eltelt idő $N_2T_0(1+\delta)$, a két koincidencia között eltelt idő pedig N_0T_0. Mindezek alapján

$$T_x = T_0\left[\pm N_0 + (N_1 - N_2)(1+\delta)\right],$$

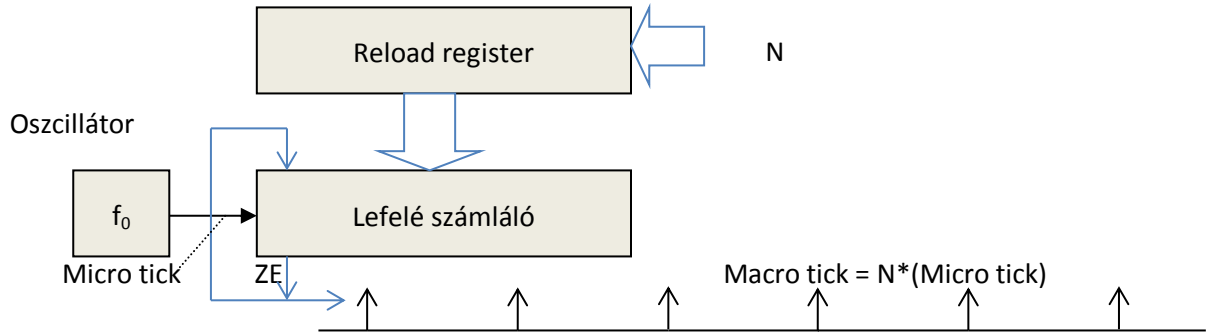
ahol az N_0 előtti előjelet a két koincidencia időbeni sorrendje határozza meg. Ha T_0=5 nsec és δ=0.004, akkor a legkisebb, még mérhető időtartam 20psec. Megjegyzés: a kvarcpontosságú, de indítható óra, valamint a koincidencia megvalósítása nehéz feladat.

4. Időmérés, időszolgáltatás, óra-szinkronizáció (folytatás)



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 abban az időpontban.

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



Az ábrán látható elrendezésben a nagyfrekvenciás kvarc-oszcillátor jelét egy lefelé számláló leosztja, és a nulla (ZE) elérésekor kiad egy impulzust. Ez adja a fizikai óra alapütemét, és ennek hatására töltődik a számlálóba a reload regiszter tartalma (*N*). A fizikai óra alapütemét az oszcillátorból származó mikro óraütéshez képest makro óraütésnek nevezhetjük. Ennek gyakorisága *N* állításával lehetséges. Mivel tipikusan az így előállított órának csak a makro óraütései férhetők hozzá, ezért ezt a szóhasználatot a továbbiakban nem követjük, helyette ennek gyakoriságára/felbontóképességére utalunk.

A fizikai referencia óra: jele *C*, felbontóképessége g^C. Pl.: 10^{15} óraütés/sec $\rightarrow$ $g^C = 10^{-15}$ sec. Értéke a nemzetközi idő szabvány szerinti abszolút idő.

Időbélyeg: $C(e)$: az *e* esemény abszolút időbélyege.

Óra drift: *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, tipikusan 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. Általában feltételezhető, hogy a *drift* egy körüli, a *drift-mérték* nulla körüli számérték.) 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, mert egy óra mintegy 100 óráig 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 a 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:

$$\Pi(t) = \underbrace{\max}_{\forall 1 \le j,k \le n} \left(ofszet_{j,k}(t) \right).$$

Megjegyzés: 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:

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

Megjegyzés: 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.

Időtartam mérése két órával

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

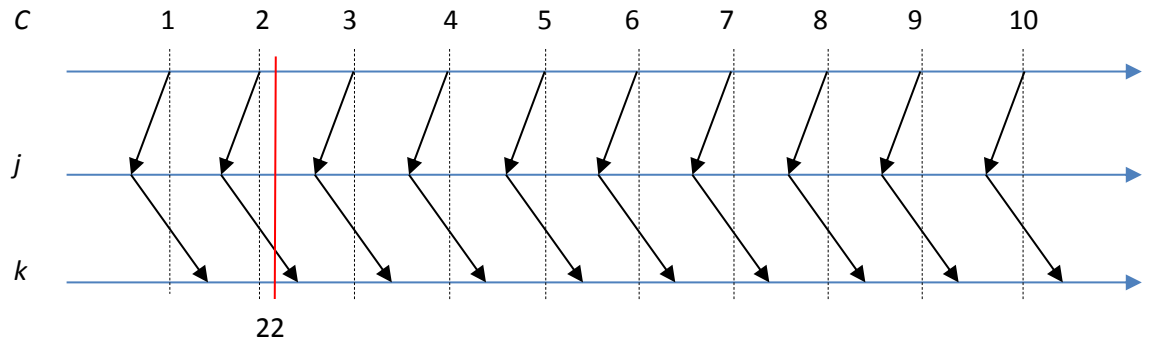
$$\left| C_j(t) - C_k(t) \right| < \Pi, \ \forall t - re.$$

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? Akkor, ha a globális idő felbontása g>Π, 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.

$$\left| C_j(e) - C_k(e) \right| \le 1.$$

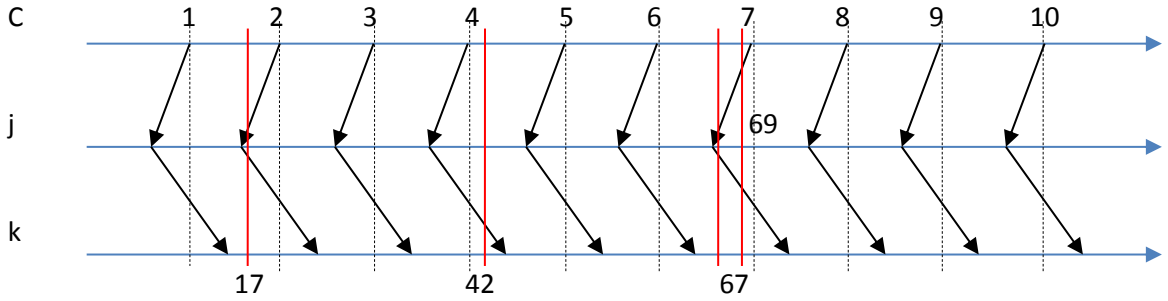
Ez a legjobb, amit elérhetünk, mert mindig előfordulhat az a szituáció, hogy először a *j* óra üt, majd bekövetkezik az *e* esemény, majd üt a *k* óra is. Ilyenkor a két óra egy óraütés differenciával bélyegzi az *e* eseményt.

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



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

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 *e*:69 és az *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.

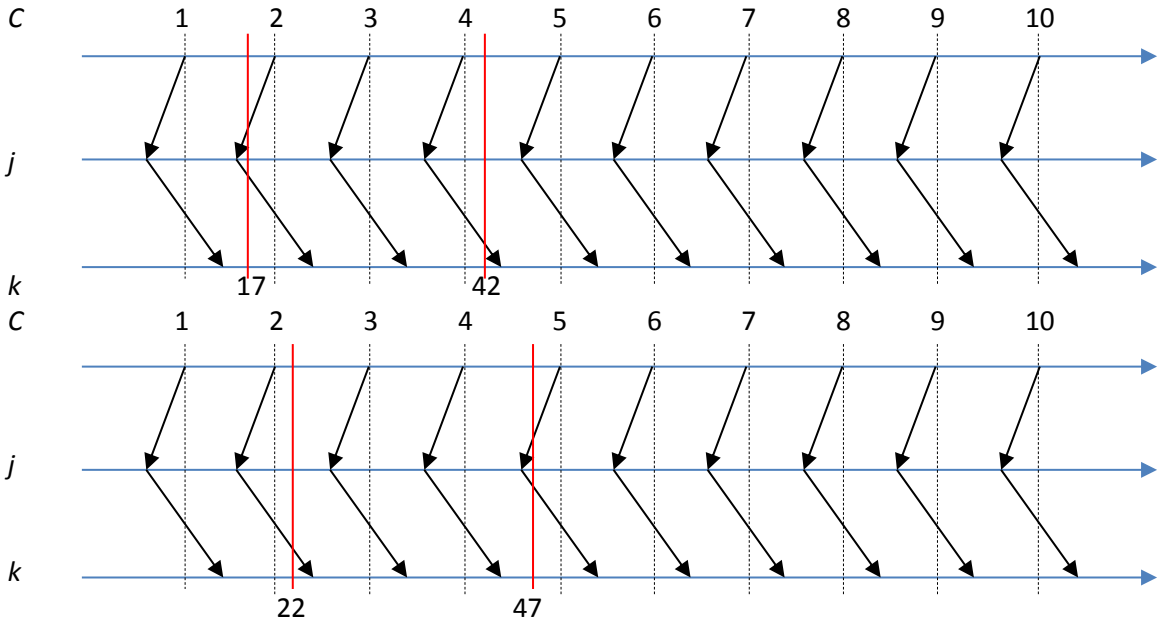
Probléma: Az időbeni sorrendet a második esetben nem tudjuk egyértelműen megállapítani a makro-ütések alapján. Az *e*:67 mikro-óraütésnél j:7, az *e*:69 mikro-óraütésnél k:6!

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:

$$\boxed{(d_m - 2g) < d_v < (d_m + 2)},$$

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



A felső ábrán, 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

alsó ábrán, ha az *e*:47 és az *e*:22 események időkülönbségét a C_j és C_k órák különbségeként mérjük, akkor a mérés 4-et ad a globális időbélyegben annak ellenére, hogy a tényleges különbség 25 mikro-ütés.

Óra rendszerek típusai:

- Központi óra rendszerek *(central clock systems)*:
  - egy pontos óra szolgáltatja az időt a teljes rendszer számára, a többi órát a rendszer a normális működés alatt figyelmen kívül hagyja,
  - hibatűréshez készenléti (standby) redundanciát alkalmaznak,
  - pontos módszer (ns-en, ms-en belül), költséges
  - speciális, a processzorba integrált hardvert igényel; a központi óra állítja ezt a hardvert a megfelelő értékre; ezt minden végrehajtó folyamat olvasni tudja,
  - a kommunikációs igény alacsony (egy üzenet frissítésenként),
  - a GPS (Global Positioning System) jó példa erre (4 órajelet sugárzó műhold, amellyel néhány ns pontossággal lehet szinkronizálni).
- 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 valamilyen 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, lásd később) esetén is a robusztusság követelmény.

Idő normáliák (standardok)

Elosztott, valós idejű rendszerekben kettőt használnak elterjedten:

- 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 30 másodperccel eltérést "szedett fel". Mivel ezeket a Bureau Internationale de l'Heure - szükség szerint - szökő másodpercekkel korrigálja, a tényleges eltérés mindig ismert és kismértékű. Megjegyzés: a szökő másodperc beillesztése a naptári év váltás pillanatában veszélyes: az 1996. január 1-én 00:00:00-kor egy másodperccel visszaállított óra még egyszer léptette a napot megadó számlálót és ezért a következő másodpercben az óra január 2-át mutatott.
  - 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 ± 1 msec, véletlen atmoszférikus ingadozások miatt ± 10 msec. (Geostacionárius műholdról ± 0.5 msec.)

- 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).

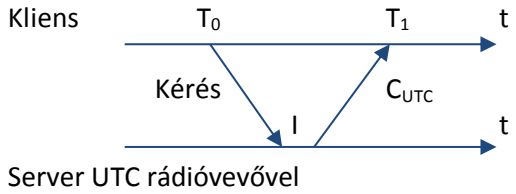
Példa: A óraszinkronizáció szükségessége/jelentősége: UNIX **make** program: nagy programok forrásai fel vannak osztva részekre (pl 100 file). Csak azokat kell újrafordítani, amelyekhez tartozó forrás megváltozott. Ha a forrás időbélyege későbbi, például input.c (timestamp 2151), és input.o (timestamp 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, hogyaz editort futtató gép órája késik két időegységet, akkor baj van!

Órák szinkronizálása: Berkeley algoritmus

Az időszerver aktív: rendszeresen lekérdezi a csomópontok óráját, átlagolja azokat, majd visszaküldi.

Órák szinkronizálása: 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} 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: $\sim \frac{T_1 - T_0 - I}{2}$.



Megjegyzés: Problémát okozhat, ha a C_{UTC}+korrekció < T_1, azaz a kliens óráját vissza kell állítani, mert siet. Ha kliens óra éppen egymást követő eseményekhez rendel időbélyeget, akkor előfordulhat, hogy visszaállítását követően későbbi eseményhez korábbi időbélyeget rendel, és ezzel az események időbeni sorrendjét látszólag megfordítja. Ha ez a veszély fennáll, akkor az órát nem szabad visszaállítani, csak „lassítani" addig, amíg futása szinkronba nem kerül az *UTC* rádió órájával.

Órák szinkronizálása: Master-slave algoritmusok

1. A master óra (i-jelű) szinkronizálást kezdeményez T_1 -ben. Az órából kiolvasott érték hibája e_1. ($T_1 = C_i(T_1) + e_1$). A slave a j csomópontban van. A T_1 -ben elküldött üzenet μ_i^j ideig utazik és T_2-ben érkezik meg. Ekkor $C_j(T_2)$ olvasható ki, amivel $T2 = C_j(T_2) + e_2$.
2. A slave kiszámolja a különbséget:

$$\boxed{d_1 = C_j(T_2) - C_i(T_1)}$$

A vétel és az adás idejének összehasonlításával:

$$C_i(T_1) + e_1 + \mu_i^j = C_j(T_2) + e_2 \rightarrow d_1 = C_j(T_2) - C_i(T_1) = \mu_i^j + (e_1 - e_2)$$

Ha a slave és a master órájának "átlagos" eltérését ε_j jelöli, akkor

$$\varepsilon_j = (e_1 - e_2) + E_j^1,$$

ahol E_j^1 zajt modellez. (Az $(e_1 - e_2)$ pillanatértékek különbsége.) Ezzel

$$d_1 = C_j(T_2) - C_i(T_1) = \mu_i^j + (e_1 - e_2) = \mu_i^j + \varepsilon_j - E_j^1$$

3. A slave elküldi órája állását a masternak $T_3 = C_j(T_3) + e_3$ időben. Az üzenet $T_4 = C_i(T_4) + e_4$ időben érkezik μ_j^i utazási időt követően. Ekkor a master kiszámítja a

$$\boxed{d_2 = C_i(T_4) - C_j(T_3)}$$

különbséget. A valóságos idők összevetésével:

$$C_j(T_3) + e_3 + \mu_j^i = C_i(T_4) + e_4 \rightarrow d_2 = C_i(T_4) - C_j(T_3) = \mu_j^i + (e_3 - e_4).$$

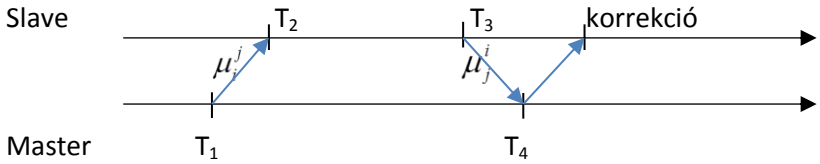
A slave és a master órájának ε_j eltérését most $-\varepsilon_j = (e_3 - e_4) + E_j^2$

formában írhatjuk, ahol E_j^2 ugyancsak zajt modellez. Ezzel $d_2 = \mu_j^i - \varepsilon_j - E_j^2$.

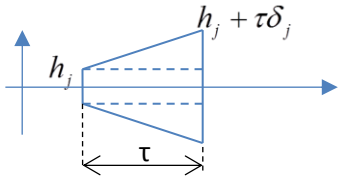
A két különbség különbsége adja az órák "átlagos" eltérését, azaz a szükséges (slave) korrekció értékét:

$$\boxed{(d_1 - d_2)/2 = \varepsilon_j + (\mu_i^j - \mu_j^i)/2 - (E_j^1 - E_j^2)/2 = \varepsilon_j + h_j},$$

ahol h_j a kommunikációs idők eltéréséből és az órák kvantálási hibáinak eltéréséből adódó hiba. Ennek a véletlen komponensét átlagolással csökkenthetjük.



A szinkronizálást követően megmaradó hiba a drift miatt változni/nőni fog. Ennek tartományát mutatja az alábbi ábra:



Az ábra alapján látható, hogy a korrekció után megmaradó hiba és a drift miatt τ idő multán, a legkedvezőtlenebb esetben az órák egymástól $2\left(\tau \underbrace{\max}_{j} \delta_j + \underbrace{\max}_{j} |h_j|\right)$ távolságra kerülhetnek.

Példa: Tempo algoritmus: master-slave szinkronizáció az elosztott Berkeley Unixban.

Master oldal:

Az alapalgoritmust N-szer lefuttatjuk:

for k=1 to N

do

Inicializálás:

do

$T_A \leftarrow C_i(now)$

$\forall j \neq i$ Send T_A to j

endo

A slave-ektől kapott adatok feldolgozása:

$\forall j \neq i$:

do

$d_2^j \leftarrow C_i(now) - T_B$

$\Delta_j(k) \leftarrow \left(d_1^j - d_2^j\right)/2$

endo

endo ; *rendelkezésre áll N differencia* $\forall j - re$

$\forall j \neq i$:

do

$\Delta_j = (1/N)\sum_{k=1}^{N} \Delta_j(k)$

Send (Δ_j) to j

endo

Slave oldal:

do

$d_1^j \leftarrow C_j(now) - T_A$

$T_B \leftarrow C_j(now)$

Send (T_B, d_1^j) to i

endo

do

$C_j(t) \leftarrow C_j(t) - \Delta_j$

endo

Megjegyzés: (1) A közölt változatban a korrekció pontossága a master és a slave között a mérések többszöri megismétlésével és az eredmények átlagolásával javult. (2) Amennyiben a master és a slave kommunikációjának további részletei ismertek (például LAN környezetben), akkor ennek felhasználásával a korrekció tovább finomítható. (3) Ha *n* processzor órájának frissítése a feladat, és minden slave *p*-szer lekérdezésre kerül, akkor a master-slave szinkronizáció kommunikáció igénye $(2p+1)n$-nel jellemezhető τ időközönként.

Órák szinkronizálása: Elosztott óra szinkronizálási algoritmusok

I. Maximális hiba minimalizálása

Minden óra tudja, hogy helyes $[C_i(t_0) = t_0]$ egy adott intervallumban:

$$[C_i(t) - E_i(t), C_i(t) + E_i(t)].$$

$E_i(t)$ összetevői: ε_i alaphiba, vagy maradék-hiba a ρ_i reset időpontban.

μ_i^j a késleltetés az i óra olvasása és a j óra frissítése között.

A δ_i drift következtében a késleltetés okozta $\delta_i \mu_i^j$ hiba.

A kommunikáció időtartama és a drift okozta hibát egyaránt a $E_i(t)$ megnövelésével vesszük figyelembe: $(1+\delta_i)\mu_i^j$.

Maga az algoritmus:

Ha kérés érkezik $j \neq i$ *-től*

do

$E_i(t) \leftarrow \varepsilon_i + (C_i(t) - \rho_i)\delta_i$

Send $(C_i(t), E_i(t))$ to *j*.

endo

Egyik szabály (az *i*-edik óra szemszögéből).

Legalább egyszer τ *időközönként*

$\forall j \neq i$: Request$(C_j(t), E_j(t))$;

for $j \neq i$ do begin

Receive$(C_j(t), E_j(t))$;

if $(C_j(t), E_j(t))$ is consistent with $(C_i(t), E_i(t))$

then if $E_j(t) + (1+\delta_i)\mu_j^i \leq E_i(t)$

then begin

$C_i(t) \leftarrow C_j(t)$

$\varepsilon_i \leftarrow E_j(t) + (1+\delta_i)\mu_j^i$

$\rho_i \leftarrow C_j(t)$

end

else ignore it

end

endo

Másik szabály

II. Intervallumok metszése

Az első szabály ugyanaz, mint előbb. A folytatás:

Legalább egyszer τ *időközönként*

$\forall j \neq i$: Request$\left(C_j(t), E_j(t)\right)$;

$\forall j \neq i$: Receive$\left(C_j(t), E_j(t)\right)$;

$\forall j \neq i$: $L_j(t) \leftarrow \left(C_j(t) - E_j(t)\right)$; baloldali intervallum határ

$\forall j \neq i$: $R_j(t) \leftarrow \left(C_j(t) + E_j(t)\right) + \left(1+\delta_i\right)\mu_j^i$; jobboldali intervallum határ

$\alpha \leftarrow \max\left(L_j\right)$; $\beta \leftarrow \min\left(R_j\right)$

if $\alpha < \beta$

then

$$\varepsilon_i \leftarrow \frac{1}{2}(\beta - \alpha);$$

$$C_i(t) \leftarrow \frac{1}{2}(\alpha + \beta);$$

$$\rho_i \leftarrow \frac{1}{2}(\alpha + \beta)$$

end

else ignore them all

endo

Megjegyzés: (1) Az intervallumok metszése módszer pontosabb, de kevésbé robusztus. (2) Ha *n* processzor órájának frissítése a feladat, 2(*n*-1)*n* az elosztott óra szinkronizálás kommunikáció igénye τ időközönként.

Megjegyzések az óraszinkronizálás témaköréhez

1. Kétarcú/kétszínű (bizánci) hiba

Az A óra 4:00-t mutat, a B óra 4:05-öt mutat, a C óra az A órának 3:55-öt, a B-nek pedig 4:10-et. A C óra "bizánci" hibás. Ezek az órák nem tudják szinkronizálni magukat. Bizonyítható, hogy a szinkronizálhatóság feltétele, hogy $N \geq$ (3k+1) óra legyen, ahol k a "bizánci" hibás órák száma.

2. *A szinkronizációs üzenet jittere*

A *jitter*: $d_{\max} - d_{\min}$

- alkalmazói programból történő szinkronizáció esetén: 500 μs …5ms
- operációs rendszer kernelből: 10 μs …100 μs
- kommunikációs vezérlő hardveréből: <10 μs .

Bizonyítható, hogy *N* óra esetén, ha *e* értékű jittert (latency jittert) tételezünk fel a kommunikációban, akkor teljesen pontos órák esetén sem érhető el jobb együttfutás, mint

$$\Pi = e\left(1 - \frac{1}{N}\right).$$

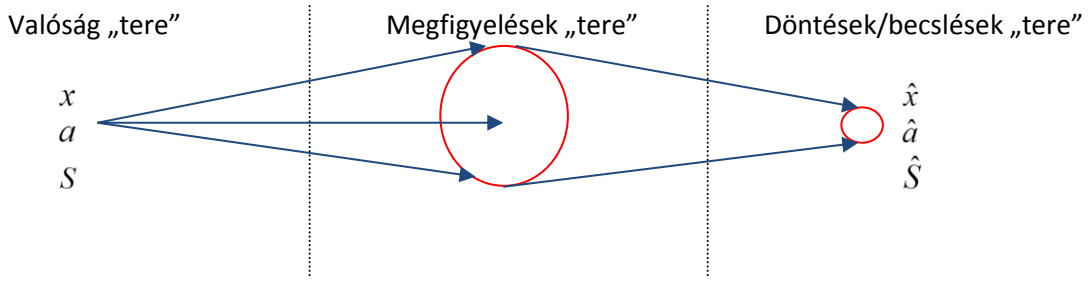
3. Hibatűrő átlagolás algoritmus:

N órából *k* bizánci típusú. Első lépésként minden órához (a hozzátartozó csomópontban) összegyűjtjük a saját óra és a többi csomópont órájának a különbségét. (A saját órára ez nulla.) A különbségek közül a *k* legkisebbet és a *k* legnagyobbat elhagyjuk és a maradék *N*-2*k* értéket átlagoljuk, majd a korrekciót erre az átlagra alapozva végezzük el. Valójában azzal a feltételezéssel élünk, hogy a hibás órák jobban eltérnek a jóktól, így azokat elhagyva a többi képes kell legyen arra, hogy jó eredményt szolgáltasson.

5. A befogadó környezet modellezése:

A mérési eljárás: a megismerési folyamat része, amelynek során a rendelkezésünkre álló ismereteinket pontosítjuk, ill. bővítjük. Az alábbi ábra a folyamat interpretálását segíti. A mérés során a valóság jelenségeit szeretnénk megragadni. Ezt a „megragadást" előszeretettel végezzük olyan jellemzőkre építve, amelyek valamilyen értelemben stabilitást mutatnak. Ilyen jellemzőkhöz (is) absztrakció révén jutunk. Kiemelt szerephez jutnak

- az állapotváltozók (x), amelyek változásai a kölcsönhatások révén fellépő energia-folyamatokhoz köthetők (feszültség, nyomás, hőmérséklet, sebesség, stb.)
- a paraméterek (a), amelyek a kölcsönhatások intenzitásviszonyait ragadják meg, és
- a struktúrák (S), amelyek a rendszer-komponensek kapcsolatait írják le.

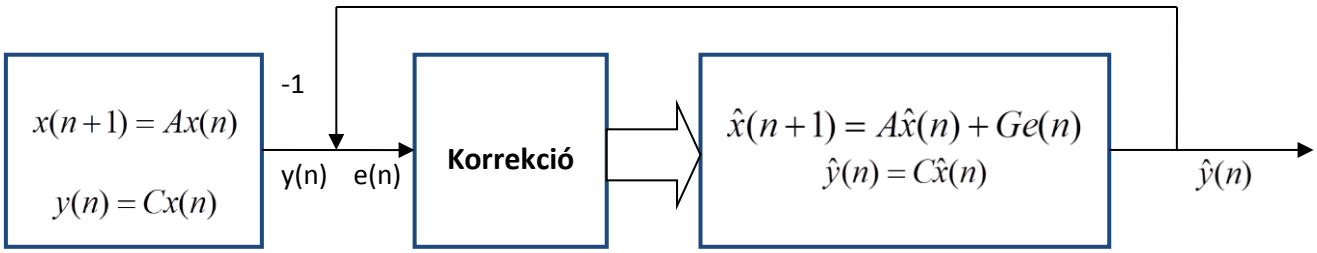


A valóság „tere" egy olyan absztrakció, amelyben a vizsgált jellemzők konkrét értékei a tér egy pontjának felelnek meg. A mérés előtt a pont koordinátáit nem ismerjük. A mérések során egy-egy ilyen pont koordinátáinak meghatározására (megmérésére) törekszünk, ami – ismert módon – csak közelítőleg lehetséges (a mérés hibával terhelt). További nehézség, hogy a mérendő mennyiséghez sok esetben nem férünk közvetlenül hozzá, ezért többnyire csak valamilyen leképzéséből tudunk kiindulni. Ezt a leképzést nevezzük megfigyelésnek. A mérendő és a megfigyelt érték közötti út a mérési/jelátviteli csatorna.

Megfigyelés determinisztikus csatorna esetén: az alábbi ábra illusztratív példaként egy időben diszkrét megfigyelőt mutat be. A megfigyelt "valóságot" autonóm rendszerként képzeljük el, és diszkrét modellel írjuk le. A „valóságot" és a megfigyelést leíró állapot, ill. megfigyelési egyenletek:

$$x(n+1) = Ax(n), \tag{1}$$

$$y(n) = Cx(n), \tag{2}$$



ahol az $x(n)$ állapotvektor N dimenziós, az A állapotátmenet mátrix $N*N$ dimenziós, az $y(n)$ megfigyelés $M \leq N$ dimenziós vektor, a C megfigyelési mátrix pedig $M*N$ dimenziós. Célunk az $x(n)$ állapotvektor becslése. Ennek eszköze a megfigyelő, amely a „valóság" másolata igyekszik lenni azáltal, hogy egy korrekciós/tanuló/adaptáló mechanizmus eredményeképpen – praktikusan egy számítógépes program által megvalósult módon - követi azt. A követés bekövetkeztével a mérés „eredménye" $\hat{x}(n)$ a megfigyelőből olvasható ki. A megfigyelőben megvalósuló „másolat" állapot, ill. megfigyelési egyenletei:

$$\hat{x}(n+1) = A\hat{x}(n) + Ge(n), \tag{3}$$

$$\hat{y}(n) = C\hat{x}(n), \tag{4}$$

ahol a *G* korrekciós mátrix *N*M* dimenziós, $e(n) = y(n) - \hat{y}(n)$. A *G* mátrixot úgy tervezzük meg, hogy $\hat{x}(n) \to x(n)$. (1) és (3) különbségét képezve:

$$x(n+1) - \hat{x}(n+1) = Ax(n) - A\hat{x}(n) - Ge(n) = (A - GC)(x(n) - \hat{x}(n)). \quad (5)$$

Bevezetve az $\varepsilon(n+1) = x(n+1) - \hat{x}(n+1)$, valamint az $F = A - GC$ jelöléseket, az ún. hibarendszer állapotátmenet mátrixa:

$$\varepsilon(n+1) = F\varepsilon(n). \quad (6)$$

A *G* korrekciós mátrixot úgy kell megtervezni, hogy $\varepsilon(n) \xrightarrow[n\to\infty]{} 0$, aminek érdekében célszerűen $\varepsilon(n+1) < \varepsilon(n)$, $\forall$ *n*-re, azaz *F* csökkenti $\varepsilon(n)$ hosszát minden lépésben, vagyis idegen szóval „kontraktív”.

Megjegyzések:

1. Az $\varepsilon(n)$ hibavektorral kapcsolatos egyenlőtlenség értelemszerűen a vektor hosszára (normájára) értelmezendő, skalár esetben pedig a hiba abszolút értékére.
2. A hiba eltűnéséhez természetesen nem kell megkövetelnünk a csökkenés monotonitását, csak a hibarendszer stabilitását, azaz külső gerjesztés nélküli esetben a nullához konvergálását. Ez interpretálható úgy is, hogy a hibarendszer a belső energiáját a stabil állapot elérése érdekében leadja, idegen szóval disszipálja. Ha ez a disszipáció az iteráció minden lépésében fennáll, akkor a hibavektor hosszának csökkenése monoton folyamat lesz.

Esetek:

1. $F = A - GC = 0$. Ebben az esetben $G = AC^{-1}$. Ez akkor lehetséges, ha *C* négyzetes, azaz a megfigyelés éppen annyi komponensű, mint maga az állapotvektor. Így aztán nem is csoda, hogy iteráció nélkül, egyetlen lépésben meg tudjuk határozni az állapotvektor értékét. Ez azt jelenti, hogy a megfigyelő, ezen belül a „másolat”, egyetlen lépés után követni képes a megfigyelt (fizikai) rendszert.

2. $F^N = (A - GC)^N = 0$. Ebben az esetben a hibarendszer *N* lépésben konvergál:

$$x(N) - \hat{x}(N) = (A - GC)^N (x(0) - \hat{x}(0)) = 0 \quad (7)$$

   Az $F^N = 0$ tulajdonságú mátrixok, az ún. nemderogatórius nilpotens mátrixok, amelyek sajátja, hogy valamennyi sajátértékük nulla. Az ilyen tulajdonságú állapotátmenet mátrixszal jellemezhető rendszerek véges impulzusválaszúak (ún. *FIR* rendszerek), hiszen a kezdeti hiba véges lépésben eltűnik. (Megjegyzés: ha $F^M = 0$, ahol *M*<*N*, akkor *F* ún. derogatórius nilpotens mátrix, ilyenkor a konvergencia kevesebb, mint *N* lépésben bekövetkezik.)

3. Ha $F^N = (A - GC)^N \neq 0$, akkor a stabilra tervezett hibarendszer állapotvektorának hossza exponenciális jelleggel fog csökkenni. Egy ilyen hibarendszer akkor lesz stabil, ha összes sajátértéke az egységsugarú körön belül helyezkedik el. Az ilyen tulajdonságú állapotátmenet mátrixszal jellemezhető rendszerek végtelen impulzusválaszúak (ún. *IIR* rendszerek), mert a kezdeti hiba csak végtelen lépésben tűnik el.

5. Mennyiségek, változók valós idejű rendszerekben

Példák:

1. Példa: Adott $A=\begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix}$; $C=\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}$. Hogyan állítsuk be *G*-t? $G=AC^{-1}=A=\begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix}$

2. Példa: Adott $A=\begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix}$; $C=\begin{bmatrix}1 & 1\end{bmatrix}$. Hogyan állítsuk be *G*-t? $G=\begin{bmatrix}g_0\\ g_1\end{bmatrix}=?$

$GC=\begin{bmatrix}g_0\\ g_1\end{bmatrix}\begin{bmatrix}1 & 1\end{bmatrix}=\begin{bmatrix}g_0 & g_0\\ g_1 & g_1\end{bmatrix}$. $[A-GC]=\begin{bmatrix}1-g_0 & -g_0\\ -g_1 & -1-1g_1\end{bmatrix}$. $[A-GC]^2=0$ alapján határozzuk meg *G*-t:

$$\begin{bmatrix}1-g_0 & -g_0\\ -g_1 & -1-g_1\end{bmatrix}\begin{bmatrix}1-g_0 & -g_0\\ -g_1 & -1-g_1\end{bmatrix}=\begin{bmatrix}1-2g_0+g_0^2+g_0g_1 & -g_0+g_0^2+g_0+g_0g_1\\ -g_1+g_1^2+g_1+g_0g_1 & 1+2g_1+g_1^2+g_0g_1\end{bmatrix}=\begin{bmatrix}0 & 0\\ 0 & 0\end{bmatrix}.$$

A mellékátló kifejezéseit a főátló kifejezéseibe behelyettesítve kapjuk: $1-2g_0=0$, illetve $1+2g_1=0$, amiből: $g_0=0.5$ és $g_1=-0.5$. Ellenőrzésképpen:

$$\begin{bmatrix}0.5 & -0.5\\ 0.5 & -0.5\end{bmatrix}\begin{bmatrix}0.5 & -0.5\\ 0.5 & -0.5\end{bmatrix}=\begin{bmatrix}0 & 0\\ 0 & 0\end{bmatrix}.$$

3. Példa: Határozzuk meg $[A-GC]$ sajátértékeit a *2. Példa* eredményének felhasználásával:

$$\det[\lambda I-A+GC]=0=\det\begin{bmatrix}\lambda-0.5 & 0.5\\ -0.5 & \lambda+0.5\end{bmatrix}=(\lambda-0.5)(\lambda+0.5)+0.25=\lambda^2-0.25+0.25=0.$$

Mindkét sajátérték nulla.

Megjegyzés:

1. Ez a tulajdonság általánosan igaz véges lépésben konvergálni képes rendszerek esetében.
2. Az ilyen rendszerek átviteli függvénye olyan (elfajuló) racionális törtfüggvény, amelynek valamennyi pólusa az origóban van:

$$H(z)=a_1z^{-1}+a_2z^{-2}+...+a_Nz^{-N}=\frac{a_N+a_{N-1}z+a_{N-2}z^2+...+a_1z^{N-1}}{z^N} \qquad (8)$$

   Ezek az ún. véges impulzusválaszú (*FIR*) szűrők. (8) időtartománybeli megfelelője:

$$y(n)=a_1x(n-1)+a_2x(n-2)+...+a_Nx(n-N), \qquad (9)$$

   ahol a valós idejű kiszámíthatóság miatt csak $x(n)$ korábbi mintái szerepelhetnek.
3. A *3. példában* a sajátértékekre vonatkozó feltétel felhasználható a g_0 és a g_1 értékek meghatározására:

$$\det[\lambda I-A+GC]=0=\det\begin{bmatrix}\lambda-1+g_0 & g_0\\ g_1 & \lambda+1+g_1\end{bmatrix}=\lambda^2+\lambda(g_0+g_1)+g_0-g_1-1=\lambda^2=0$$

Ebből: $g_0+g_1=0$, ill. $g_0-g_1=1$, amiből: $g_0=0.5$ és $g_1=-0.5$.

Megfigyelés zajos csatorna esetén: Ebben az esetben nem $\varepsilon(n)\xrightarrow[n\to\infty]{}0$ az elvárásunk, hanem $E[\varepsilon(n)\varepsilon^T(n)]\xrightarrow[n\to\infty]{}\min$ legyen. Ezzel a hibarendszer (6) állapotegyenletét az

$$E[\varepsilon(n+1)\varepsilon^T(n+1)]=FE[\varepsilon(n)\varepsilon^T(n)]F^T \qquad (10)$$

összefüggés váltja fel. Ez a hiba-mátrix központi szerepet kap a híres Kalman prediktor, ill. szűrő esetében. (R.E. Kalman világhírű, magyar származású tudós, 2016 nyarán hunyt el.)

Megjegyzések:

1. A megfigyelő elrendezés mindkét modellje (lásd az előző előadás ábráját) „gerjeszthető” egy közös gerjesztéssel. Mivel a modellek lineárisak, a szuperpozíció értelmében a megfigyelő konvergenciája változatlanul megvalósul.
2. A 2. ábra szerinti megfigyelőt Luenberger megfigyelőnek nevezzük. Luenberger szerint majdnem minden rendszer megfigyelő. A megfigyelő tulajdonság feltétele, hogy a megfigyelő legyen „gyorsabb”, mint a megfigyelt rendszer, különben nem képes követni a változásokat.
3. Egy ellenállás- vagy impedancia-mérő híd ismeretlen elemet tartalmazó hídága a valóság fizikai modellje, a kiegyenlítő elemet tartalmazó ága pedig a megfigyelőben felépülő, beállítható/hangolható modell. A hídágak osztópontján megjelenő feszültségek különbsége vezérli a hangolást, és a végén a két feszültség megegyezik, a beállítható elemről leolvasott érték segítségével meghatározható az ismeretlen. Ez az áramkör, a hangolást végző operátor részvételével megvalósítja a megfigyelőt.

Legkisebb négyzetes hibájú (LS) becslők: nincs előzetes ismeretünk sem a mérendő paraméterről, sem a csatorna karakterisztikáról (a zajról). Tegyük fel, hogy a megfigyelési egyenlet lineáris: $z = Ua + n$. Feltételezzük, hogy az a paraméter $\hat{a}$ értéket vesz fel, és felállítjuk a megfigyelés modelljét: $U\hat{a}$. A megfigyelést ezzel összevetve keressük $\hat{a}$ legjobb beállítását négyzetes hibafüggvény feltételezésével:

$$C(a,\hat{a}) = (z - U\hat{a})^T (z - U\hat{a}) = z^T z - z^T U\hat{a} - \hat{a}^T U^T z + \hat{a}^T U^T U\hat{a} = z^T z - 2\,\hat{a}^T U^T z + \hat{a}^T U^T U\hat{a} \tag{11}$$

melynek szélsőértékét (minimumát) keressük: $\left.\frac{\partial C(a,\hat{a})}{\partial \hat{a}}\right|_{\hat{a}=\hat{a}_{LS}} = 0$ feltétel vizsgálatával. (11) deriválásával $-2U^T z + 2U^T U\hat{a} = 0$, amivel:

$$\boxed{\hat{a}_{LS} = [U^T U]^{-1} U^T z} \tag{12}$$

Megjegyzések:

1. A derivált helyességét egyszerűen leellenőrizhetjük, ha a (11) összefüggésben kijelölt mátrix-, ill. vektor-szorzásokat kifejtjük, és azt követően a deriválást komponensenként végezzük el.
2. Általánosított/súlyozott négyzetes kritériumot is használhatunk, ha bevezetünk egy Q négyzetes súlyozó mátrixot:

$$C(a,\hat{a}) = (z - U\hat{a})^T Q (z - U\hat{a}), \tag{13}$$

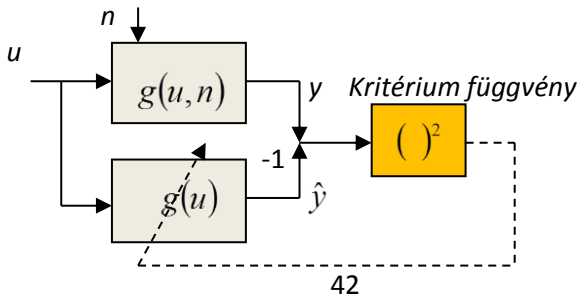
amivel

$$\boxed{\hat{a}_{LS} = [U^T Q U]^{-1} U^T Q z}. \tag{14}$$

Modellillesztés

A legkisebb négyzetes hibájú becslők esetén nincs előzetes ismeretünk, valójában modellt illesztettünk. A modellillesztés problémája meglehetősen szerteágazó. Egyik klasszikus válfaja a regresszió számítás.

Regresszió-számítás: függő és független változók közötti közvetlen, determinisztikus kapcsolat meghatározása, a modellillesztés egy speciális esete. Az alábbi ábrán látható elrendezésben a modellezendő $y = g(u,n)$ függvény kétfajta független változóval rendelkezik: az egyiket $u(n)$ jelöli, amelyet ismerünk és „kézben tudunk tartani”, a másik, amelyiket $n(n)$ jelöli, amely ismeretlen, kézben nem tartható, tipikusan zajfolyamatnak elképzelt/modellezett folyamat.



Megjegyzések:

1. A továbbiakban az argumentumként szereplő kis „*n*” nagyon gyakran az iterációs lépést azonosítja vagy diszkrét időindex, amely ekvivalens módon tényleges indexként is megjelenik időnként. Ennek megfelelően $u(n)=u_n$, ill. $y(n)=y_n$ egyenértékűek.
2. A kis „*n*” kettős használata senkit se zavarjon, a különbség egyértelmű: argumentumként, ill. indexként diszkrét („idő”) index, önállóan pedig zajfolyamatként interpretáljuk.

A modellezéshez egy általunk kézben tartott, tipikusan paraméterek segítségével módosítható („hangolható”) $\hat{y}=\hat{g}(u)$ függvényt használunk. A cél egy olyan „beállítás” elérése, amely valamilyen értelemben optimális. Tipikusan négyzetes kritériumot használunk:

$$\varepsilon = E\left\{(y-\hat{y})^T(y-\hat{y})\right\} \tag{15}$$

Lineáris regresszió: Az illesztendő függvény a $\hat{g}(u)=a_0+a_1u$ **skalár** lineáris függvény, melynek paraméterei úgy választandók meg, hogy $E\left\{(y-\hat{g}(u)^2\right\}$ minimális legyen. Legyen ismert $\mu_u,\mu_y,\sigma_u,\sigma_y,\rho$, ahol az utóbbi a normalizált kereszt-kovariancia függvény: $\rho=\dfrac{E\left\{(u-\mu_u)(y-\mu_y\right\}}{\sigma_u\sigma_y}$. Minimalizálandó az

$$\varepsilon = E\left\{(y-a_0-a_1u)^2\right\}=E\left\{y^2\right\}+a_0^2+a_1^2E\left\{u^2\right\}-2a_0E\left\{y\right\}-2a_1E\left\{uy\right\}-2a_0a_1E\left\{u\right\} \tag{16}$$

összefüggés a_0 és a_1 szerint:

$$\frac{\partial\varepsilon}{\partial a_0}=2a_0-2\mu_y+2a_1\mu_u=0, \text{ ahonnan } a_0=\mu_y-a_1\mu_u, \text{ amit} \tag{17}$$

$\dfrac{\partial\varepsilon}{\partial a_1}=2a_1(\sigma_u^2+\mu_u^2)-2(\rho\sigma_u\sigma_y+\mu_u\mu_y)+2a_0\mu_u=0$ kifejezésbe behelyettesítve

$$\boxed{\hat{a}_0=\mu_y-\mu_u\rho\frac{\sigma_y}{\sigma_u},\ \hat{a}_1=\rho\frac{\sigma_y}{\sigma_u}} \tag{18}$$

Megjegyzések:

1. A (18) összefüggés származtatásakor felhasználtuk, hogy $E\left\{(u-\mu_u)^2\right\}=\sigma_u^2=E\left\{u^2\right\}-\mu_u^2$, valamint $E\left\{(u-\mu_u)(y-\mu_y)\right\}=E\left\{uy\right\}-\mu_u\mu_y$.
2. A lineáris regresszió feladatának egyfajta általánosítása az ún. polinomiális regresszió:

$$\hat{g}(u)=\sum_{k=0}^{N}a_ku^k, \tag{19}$$

amelynek fontos tulajdonsága, hogy paramétereiben lineáris. A paramétereiben lineáris modelleket azért kedveljük, mert négyzetes hibakritérium esetén a szélsőérték-keresés lineáris egyenletrendszer megoldására vezet, mivel a négyzetes kifejezések paraméterek szerinti deriválása lineáris összefüggést eredményez.

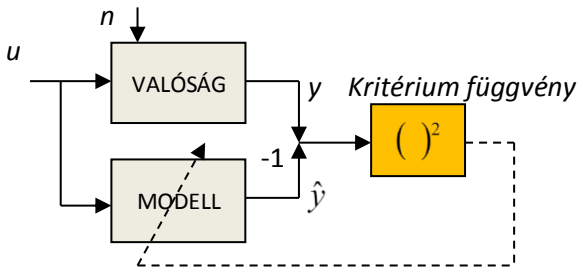
Lineáris regresszió mérési adatok alapján: a fentieket végigvihetjük akkor is, ha nincsen előzetes információnk. Ilyenkor $y_n=a_0+a_1u_n+w_n$, ahol w_n *az additív zaj,* n=0,1, …,N-1, $z=Ua+n$, mint eddig. A legkisebb négyzetes hibájú becslő összefüggéseit használva:

$$z=\begin{bmatrix}y_0\\y_1\\\vdots\\y_{N-1}\end{bmatrix}=\begin{bmatrix}1&u_0\\1&u_1\\\vdots&\vdots\\1&u_{N-1}\end{bmatrix}\begin{bmatrix}a_0\\a_1\end{bmatrix}+\begin{bmatrix}w_0\\w_1\\\vdots\\w_{N-1}\end{bmatrix},\ [U^TU]=\begin{bmatrix}N&\sum_{n=0}^{N-1}u_n\\\sum_{n=0}^{N-1}u_n&\sum_{n=0}^{N-1}u_n^2\end{bmatrix},\ U^Tz=\begin{bmatrix}\sum_{n=0}^{N-1}y_n\\\sum_{n=0}^{N-1}u_ny_n\end{bmatrix}.$$

$$\begin{bmatrix}\hat{a}_0\\ \hat{a}_1\end{bmatrix}=\frac{1}{\frac{1}{N}\sum_{n=0}^{N-1}u_n^2-\left(\frac{1}{N}\sum_{n=0}^{N-1}u_n\right)^2}\begin{bmatrix}\frac{1}{N}\sum_{n=0}^{N-1}u_n^2 & -\frac{1}{N}\sum_{n=0}^{N-1}u_n\\ -\frac{1}{N}\sum_{n=0}^{N-1}u_n & 1\end{bmatrix}\begin{bmatrix}\frac{1}{N}\sum_{n=0}^{N-1}y_n\\ \frac{1}{N}\sum_{n=0}^{N-1}u_n y_n\end{bmatrix}.$$

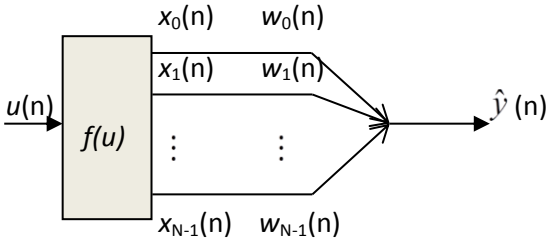
Megjegyzés: Ebben az eredményben a (82) kifejezésben szereplő statisztikai jellemzők becslőinek összetevőit azonosíthatjuk, és átalakításokkal - tipikusan az átlagtól való eltérések felírásával - ezeket a kifejezéseket egymásnak teljesen megfeleltethetjük. Tegyék meg!

A regressziós séma általánosítása: Az alábbi ábrán a modell-illesztést a regressziós sémát követő módon mutatjuk be.



Az u bemenetre adott y választ szeretnénk valamilyen kritérium szerint (az ábrán négyzetes értelemben) legjobban megközelíteni a modell $\hat{y}$ válaszával. Érdekes összevetni ezt a sémát a megfigyelő sémával (lásd előző előadás). A nagyfokú hasonlóság egyértelmű: mindkét esetben modellillesztést végzünk. A megfigyelő séma esetén a paramétereket ismerjük, és az állapotokat becsüljük, míg a regressziós sémában a modellünk „állapotát” kézben tartjuk, és a paramétereket keressük. Mindkét séma „párhuzamos” abban az értelemben, hogy a „bemenő jelet” illetően párhuzamosan kapcsolódnak.

Adaptív lineáris kombinátor: Az általánosított regressziós séma kapcsán az egyik gyakran használt modell-családot az alábbi ábra mutatja be. A modell két részből áll: az egyik egy rögzített függvény, a másik egy változtatható paraméterekkel súlyozó lineáris kombinátor.



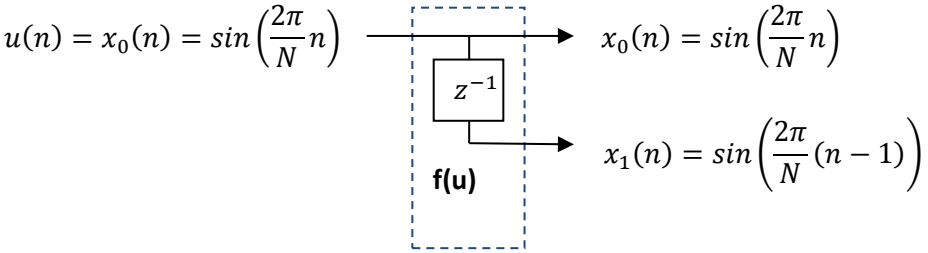
Ebben az $u(n)$ diszkrét értéksorozatból egy $X^T(n)=\begin{bmatrix}x_o(n) & x_1(n) & \dots & x_{N-1}(n)\end{bmatrix}$ értéksorozatot állítunk elő először, majd ezen értékek lineáris kombinációjaként állítjuk elő az $\hat{y}(n)$ értéket. Az optimumkeresés során a $W^T(n)=\begin{bmatrix}w_0(n) & w_1(n) & \dots & w_{N-1}(n)\end{bmatrix}$ paraméterek legkedvezőbb, minimális négyzetes hibát eredményező beállítására törekszünk. Minimalizáljuk az

$$\varepsilon(n)=E\left\{[y(n)-X^T(n)W(n)]^T[y(n)-X^T(n)W(n)]\right\}=$$
$$=E\left\{y^T(n)y(n)\right\}-2W^T(n)E\left\{X(n)y(n)\right\}+W^T(n)E\left\{X(n)X^T(n)\right\}W(n). \qquad (20)$$

Vezessük be a $E\{X(n)y(n)\}=P$, és a $E\left\{X(n)X^T(n)\right\}=R$ jelölést! Ezzel a szélsőérték keresés $\frac{\partial\varepsilon(n)}{\partial W(n)}=-2P+2RW(n)=0$, amiből az optimális beállítás: $\boxed{W^*=R^{-1}P}$ (21)

A (21) összefüggés az ún. Wiener-Hopf egyenlet.

Példa: Legyen $X^T(n)=\left[\sin(2\pi n/N) \quad \sin(2\pi(n-1)/N)\right]$, azaz egy szinuszos hullámforma két egymás utáni mintája.



A regressziós vektor és a paraméter vektor ebben a példában kétdimenziós. Itt most *N* azt jelöli, hogy a szinuszos hullámforma egy periódusa hány mintából áll. $y(n)=2\cos(2\pi n/N)$. Hogyan válasszuk meg a

$$W^T(n)=\left[w_0(n) \quad w_1(n)\right] \tag{22}$$

paramétereket ahhoz, hogy a közelítés négyzetes hibája minimális legyen? A *R* és a *P* mátrixok a szinuszos, ill. koszinuszos hullámformák teljes (*N*>2) periódusra történő átlagolásával származtathatók:

$$R=\begin{bmatrix} 0.5 & 0.5\cos\frac{2\pi}{N} \\ 0.5\cos\frac{2\pi}{N} & 0.5 \end{bmatrix},\ P=\begin{bmatrix} 0 \\ -\sin\frac{2\pi}{N} \end{bmatrix}. \tag{23}$$

$$E\left\{\sin^2(2\pi/N)\right\}=E\left\{\sin^2(2\pi(n-1)/N)\right\}=0.5,\ E\left\{\sin(2\pi n/N)\sin(2\pi(n-1)/N)\right\}=\frac{\cos(\frac{2\pi}{N})}{2},$$

$$E\left\{2\sin(2\pi n/N)\cos(2\pi n/N)\right\}=0,\qquad E\left\{2\sin(2\pi(n-1)/N)\cos(2\pi n/N)\right\}=-\sin\frac{2\pi}{N}.$$

$$R^{-1}=\frac{1}{0.25\sin^2\frac{2\pi}{N}}\begin{bmatrix} 0.5 & -0.5\cos\frac{2\pi}{N} \\ -0.5\cos\frac{2\pi}{N} & 0.5 \end{bmatrix},\ \boxed{W^*=R^{-1}P=\begin{bmatrix} \frac{2}{\tan(2\pi/N)} \\ -\frac{2}{\sin(2\pi/N)} \end{bmatrix}} \tag{24}$$

Megjegyzések:

1. $X^T(n)W^*=2\frac{\sin(2\pi n/N)}{\tan(2\pi/N)}-2\frac{\sin(2\pi(n-1)/N)}{\sin(2\pi/N)}=2\cos(2\pi n/N)$.

2. Mivel szinuszos minták lineáris kombinációjával hiba nélkül elő lehet állítani koszinuszos hullámformák mintáit, ezért a példa szerinti esetben $\varepsilon_{\min}=0$, azaz a hibafelület paraboloid legalsó pontja érinti a paraméterek síkját.

Út az adaptív eljárásokhoz: (20) és (21) alapján: $W^*=R^{-1}P$, $\nabla(n)=2(RW(n)-P)$. Ez utóbbi mindkét oldalát megszorozva az $\frac{1}{2}R^{-1}$ mátrixszal: $\boxed{W^*=W(n)-\frac{1}{2}R^{-1}\nabla(n)}$. (25)

Feltételezve, hogy nincs tökéletes ismeretünk az *R* mátrixról, és ebből adódóan a gradiensről, (25) átírható egy iteratív formára: $W(n+1)=W(n)-\frac{1}{2}\hat{R}^{-1}\hat{\nabla}(n)$, illetve a $0<\mu<1$ „bátorsági” tényező bevezetésével, visszaírva a „tökéletes” *R* mátrixot és gradienst

$$\boxed{W(n+1)=W(n)-\mu R^{-1}\nabla(n)}. \tag{26}$$

Megjegyzések:

1. Ha pontosan ismerjük az R mátrixot és gradienst, akkor $\mu = \frac{1}{2}$ egylépéses konvergenciát biztosít tetszőleges W(n) kezdőpontból.
2. Mivel $\nabla(n) = 2R[W(n) - W^*]$, ezért ezt a (26) összefüggésbe behelyettesítve, és az egyenlet mindkét oldalából levonva W^* értékét:
   $W(n+1) - W^* = (1-2\mu)(W(n) - W^*) = V(n+1) = (1-2\mu)^{n+1} V(0)$, vagyis a kezdeti hiba exponenciális jelleggel csökken, ha $\mu \neq \frac{1}{2}$. Ha $0 < \mu < 0.5$, akkor monoton csökkenő hibával, ellenkező esetben pedig monoton csökkenő amplitúdójú, de lengő jellegű hibával közelítjük meg.
3. A modell-illesztés gradiens módszereit a szerint különböztetjük meg, hogy a (26) szerinti összefüggés alkalmazásához milyen előzetes ismeretek állnak rendelkezésünkre.

Az adaptív lineáris kombinátor működését leíró egyenletek, amennyiben az R és a P mátrixok ismertek:

$$\boxed{W(n+1) = W(n) - \mu R^{-1} \nabla(n)}, \text{ ill. } \boxed{V(n+1) = (1-2\mu) V(n)}. \qquad (27)$$

Az előadás idejében került sor a tárgy nagyzárthelyi dolgozatának megírására.

5. Mennyiségek, változók valós idejű rendszerekben (folytatás)

A replikátum determinizmusa

Ha a megbízhatóságot aktív redundanciával, azaz fizikai többszörözéssel javítjuk, akkor a párhuzamosan működő egységeknél meg kell követeljük, hogy (1) a kívülről látható RAM állapotuk ugyanaz legyen, és (2) a kimenetek azonosak legyenek, maximum d időbeni eltéréssel. A d értékét a rendszer dinamikai tulajdonságai alapján határozhatjuk meg: kell maradjon idő a hibás vagy hiányzó adat pótlására a replikátumból.

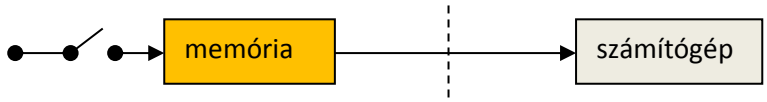
Példa: Három csatornás repülés-irányító rendszer többségi szavazással. Mindegyik csatorna önálló szenzorokkal és számítógépekkel rendelkezik, hogy az Ún. közös-módusú hibák valószínűségét minimalizáljuk. A "felszállás kezdete" eseményt követően egy előírt időn belül a vezérlő rendszernek ellenőriznie kell, hogy a repülőgép elérte-e a felszálláshoz szükséges sebességet. Ha igen, akkor kezdeményezi az emelkedést és a motorokat tovább gyorsítja. Ha nem, akkor a felszállási folyamat megszakítandó, és a motorokat le kell állítani. Az alábbi táblázat egy olyan helyzetet ír le, ahol a replikátum determinizmusa feltétel nem teljesül, és a hibás csatorna érvényesül a döntésben:

Csatorna	Döntés	Akció
1. csatorna	Felszállás	Motor gyorsítása
2. csatorna	Megszakítás	Motor leállítása
3. csatorna	Megszakítás	Motor gyorsítása

A táblázat szerinti első két csatorna helyesen működik, csak nem teljesül a replikátum determinizmusa feltétel. Véletlen hatások eredményeképpen (eltérés a szenzor kalibrációban, digitalizálási hiba, a sebességmérés időpontjában kicsi eltérés) a két csatorna eltérő következtetésre jut. A harmadik csatorna hibásan működik, mert megszakítást dönt, és gyorsítja a motort. Az akcióra vonatkozó többségi szavazás a hibás csatorna által javasolt eredményt hoz a replikátum determinizmusára vonatkozó feltétel teljesülésének hiányában.

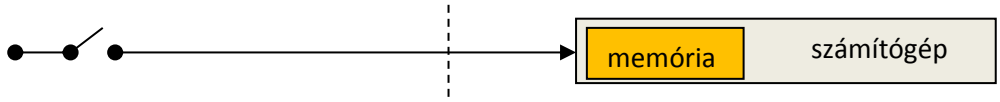
Mintavételezés és lekérdezés:

Mintavételezés (sampling) szóhasználattal élünk, ha az adatot a szenzor egységnél írjuk memóriába:



A mintavételezés megóvja a rendszert, hogy több esemény érkezzen, mint a specifikációban rögzített. A memória a számítógép befolyásolhatósági tartományán kívül helyezkedik el. A számítógép leállása, újraindulása esetén a memóriatartalom nem vész el.

A lekérdezés (polling) szóhasználattal élünk, ha az adatot a számítógép memóriájába helyezzük:

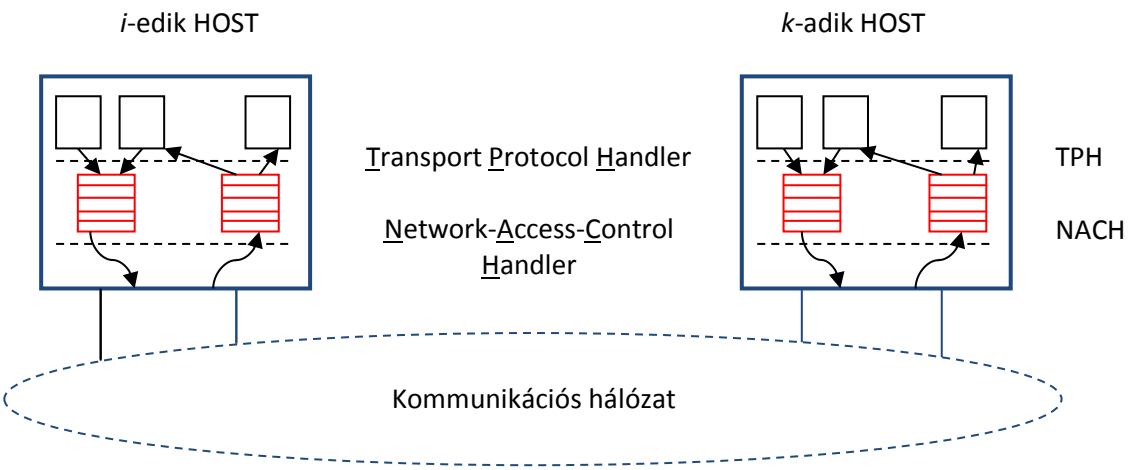


Funkcionális szempontból a két megoldás nem tér el egymástól, de hiba esetén a mintavételezés robusztusabb.

Megjegyzés: Az interrupt mechanizmus a pollingot bemutató ábrával jellemezhető. Súlyos problémája, hogy külső eszköz befolyásolhatósági tartományába helyezi a számítógépet, ezért fokozott körültekintéssel kell alkalmazni, mert hiba esetén oly mértékben túlterhelheti a processzort, hogy az képtelen lesz feladatait (időre) ellátni.

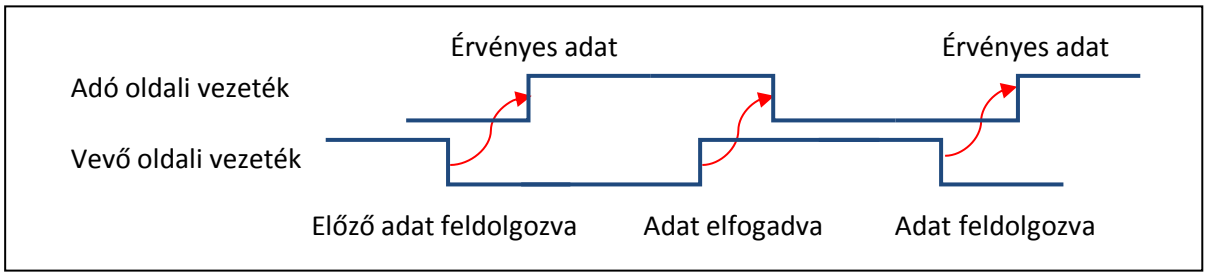
6. Valós idejű kommunikáció

Az általános séma:



Megjegyzés: Általában bonyolult mechanizmusok, várólisták jellemzőek. Valós idejű követelmények nehezen teljesíthetőek.

Az időviszonyok kritikus volta a fizikai szinten is jól azonosítható. Aszinkron kommunikáció esetén szinkronizálás kell, ez a "handshaking". A kétvezetékes handshake:



A kommunikáció sebesség- és időviszonyait az adó és a vevő sebessége és egyéb feladatai együtt határozzák meg, hiszen az adat vevő oldali "feldolgozásáig" újabb adat továbbításában az adó nem gondolkodhat.

Követelmények:

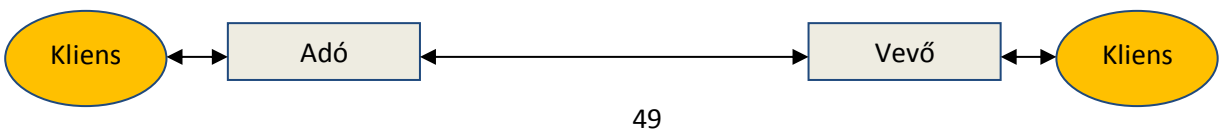
1. Lehetőleg kis protokoll késleltetés (protocol latency) és jitter (latency jitter). (Latency árnyaltabb jelentése: lappangás, homály, elrejtettség).
2. Komponálhatóság: segíteni kell az időbeni követelmények teljesülését: HOST ↔ CNI (Communication Network Intrface), időszakos tűzfal szerep, HOST önálló tesztelhetősége.
3. Flexibilitás: gépkocsi funkciók időbeni működése extrákkal, extrák nélkül …
4. Hibadetektálás: Jósolható és hibatűrő kommunikáció kell. End-to-end nyugtázás. Egy szelep zárjon automatikusan, ha az állítását lehetővé tevő vezeték elszakad, de erről menjen értesítés a központnak.
5. Struktúra: pont-pont kapcsolat kezelhetetlen bonyolultságú kábelezéssel jár, helyette busz és gyűrű.

Az adatáramlás szabályozása (flow control):

Explicit forgalomszabályozás:

Példa: PAR (Positive Acknowledgement or Retransmission) protokollok: Több változat van, de ezek közösek az alábbiakban:

(1) Az adóoldali kliens kezdeményez. (2) A vevő jogosult késleltetni. (3) A hibát az adó detektálja. (4) Hibajavítás időbeni redundanciával.



Adó oldali program:

(1) Az ismételt küldések számlálóját nullázzuk.
(2) Indítjuk a visszaigazoláshoz rendelt time-out számlálót.
(3) Indítjuk az üzenetet.
(4) A time-out-on belül fogadjuk a visszaigazolást.
(5) Értesítjük a klienst a sikeres adattovábbításról.

Ha nincs visszaigazolás a time-out-on belül:

(a) Ellenőrizzük az ismételt küldések számlálóját, hogy elérte-e a maximumát.
(b) Ha igen, akkor megszakít minden tevékenységet, és hibát jelez a kliensnek.
(c) Ha nem, akkor inkrementálja az ismételt küldések számlálóját, és visszatér a fenti (2) ponthoz.

Vevő oldali program:

(1) Üzenet érkezésekor ellenőrzi, hogy ez az üzenet érkezett-e már korábban.
(2) Ha nem, akkor visszaigazol, és értesíti a kliensét.
(3) Ha igen, akkor csak visszaigazol. (Ilyenkor az előző visszaigazolás time-out időn túl érkezhetett az adóhoz, ha egyáltalán megérkezett.

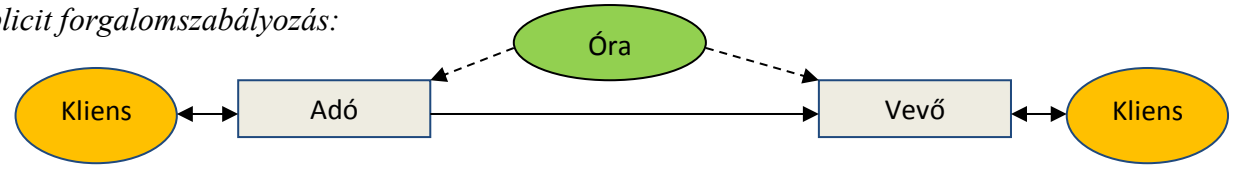
Megjegyzés:
Az adó oldalon a vétel visszaigazolása és a vevőoldalon az adat elfogadás időpontja között jelentős eltérés lehet.

Példa: Token vezérelt buszon az üzenettovábbítás ideje 1 *ms*, a token körüljárás ideje 10 *ms*.
A beállítandó time-out: 10+1+10+1=22 *ms*, hiszen worst-case esetben, ha éppen elment a token 10 *ms*-ot kell várni, erre jön az üzenettovábbítás 1 *ms*-a, majd a visszaigazoláskor ugyanez ismétlődhet. A d_{min}=1 *ms*, a d_{max}= (ismétlések száma)*time-out+10 *ms*+1 *ms*. Ha kétszer ismétlünk (azaz háromszor próbálkozunk), akkor d_{max}= 55 *ms*.

Ezekkel néhány jellemző a következőképpen alakul:
- jitter= d_{max}- d_{min}=54 *ms*.
- Akció késleltetés, ha van globális óra: d_{max} = 55 *ms*.
- Akció késleltetés, ha nincs globális óra: 2* d_{max} - d_{min}= 109 *ms*.
- A hibadetektálás késleltetése: 3*time-out: 66 *ms*.

A PAR protokoll és a számpélda azt illusztrálja, hogy az ún. explicit forgalomszabályozás valós idejű alkalmazásokban kedvezőtlen lehet a nagymértékű jitter, akció késleltetés és hibadetektálás késletetés miatt.

Implicit forgalomszabályozás:



A kommunikáció idővezérelt. Az adó és a vevő is rendelkezik egy tervezési időben elkészült időrendi táblázattal (“vasúti menetrend”). Ebben egyértelmű az adás és egyidejűleg a vétel időpontja/időintervalluma. Az adó az óraütés vezérlésére „kitolja” az üzenetet, a vevő pedig “behúzza” (push-pull jellegű működés). Ez a logika sok esetben jobban illeszkedik a valós idejű követelményekhez. A hibadetektálás például a vevő által azonnal lehetővé válik, ha a várt adat nem érkezik meg. (Az adó részéről ez egy ún. fail-silent üzemmódban létet jelent, azaz hibás állapotát azzal jelzi, hogy nem küld üzenetet.)

Globális időalap kell. Az adó csak meghatározott időpontokban ad, nincs handshake, a hibadetektálás a vevő dolga: tudja, hogy mikor kell/kellett volna üzenetnek érkeznie. A hibatűrős aktív redundanciával valósul meg: *k* fizikai üzenet kópia, ha legalább egy megérkezik, addig sikeres. A csatorna egyirányú, ami többszereplős esetben előnyös.

Az idővezérelt architektúra (Time Triggered Architecture, TTA) és az idővezérelt protokollok (Time-Triggered Protocols, TTP)

(Az idővezérelt architektúráról és protokollról részletes leírás található a tantárgy tanszéki honlapján. Az alábbiak csak néhány kiemelt jellemzőt foglalnak össze, ill. olyan részleteket, amelyek az említett dokumentumban nem szerepelnek.)

Hard real-time (HRT) rendszerek implementálására szolgál. Két változata van: a TTP/C, amely hibatűrő HRT rendszerekhez készült, és a TTP/A, amely olcsó ipari alkalmazások esetén jön számításba (pl. terepi busz (field bus) alkalmazásokban). A rendszer hibatűrő egységekből (FTU: Fault Tolerent Unit) felépülő fürt (cluster). Minden FTU cluster egy, kettő, vagy több csomópontból áll, amelyeket a kommunikációs hálózat köt össze. Minden csomópont két részrendszerből, a host számítógépből és a kommunikációs vezérlőből áll. A kommunikációs hálózat interfész (CNI) a csomóponton belüli interfész a host és a kommunikációs vezérlő között. A CNI egy dual-portos RAM memória (DPRAM). Az adat integritást a Non-Blocking Write (NBW) Protocol biztosítja (lásd később). A kommunikációs vezérlő lokális memóriája tartalmazza az üzeneteket leíró listát (Message Description List: MEDL), amely meghatározza, hogy mely időpontban küldhet a csomópont üzenetet, ill. mely időpontban várhat más csomópontból. A MEDL méretét a fürt-ciklus mérete határozza meg. A TTP vezérlő független hardverként ún. Bus Guardian egységeket is tartalmaz, amelyek figyelik a vezérlő busz-hozzáférési mintáit, és leállítják a vezérlő működését, ha a szabályos hozzáférési minták időzítése megsérül.

Fontos tulajdonságok:

(1) A TTP egy időosztásos-többszörös-hozzáférésű (time-division-multiple-access: TDMA) protokoll.

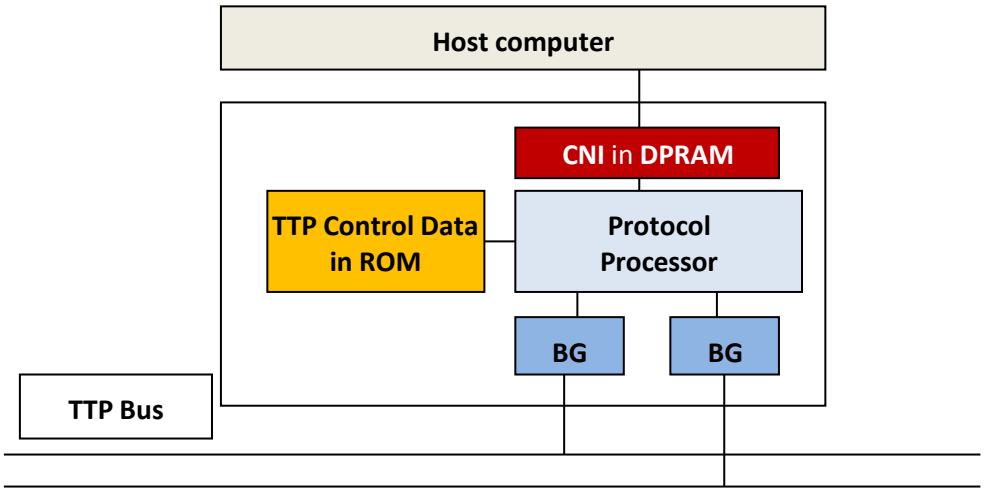
(2) A komponálhatóságot szolgálja, hogy a kommunikációs vezérlő autonóm, amelyet a MEDL és a globális óra vezérel. A host számítógépek hibája nem tudja befolyásolni a kommunikációs rendszert, mert vezérlő jel nem megy át a CNI-n és a MEDL nem férhető hozzá a host felől.

(3) A kommunikáció módja tervezési időben dől el (olyan, mint a vasúti menetrend), mindenki előre tudja mikor kap, ill. mikor küld üzenetet. Ha hiányzik/elmarad az üzenet, akkor azonnal detektálható a hiba.

(4) Az üzenet azonosítása (naming): az üzenet és küldőjének neve nem kell, hogy része legyen az üzenetnek, a MEDL-ből kinyerhető. Ugyanannak az RT változónak más és más nevet adhatunk az egyes host-ok szoftverében.

(5) Visszaigazolás: előzetesen tudjuk, minden helyesen működő vevő veszi a helyesen működő adó üzenetét. Amint egy vevő visszaigazol egy üzenetet, arra lehet következtetni, hogy az üzenet kiküldése helyesen történt és azt minden helyesen működő vevő megkapta.

(6) Hiba esetén "hallgatás" az időtartományban: a TTP feltételezi, hogy a csomópontok támogatják a "fail silence" absztrakciót az időtartományban, ami azt jelenti, hogy egy csomópont vagy küld üzenetet a helyes időpontban, vagy nem küld semmit. A csomópontnak ezt a tulajdonságát a TTP vezérlőn belül a bus guardian valósítja meg. Az amplitúdó tartományban a hibakezelés a host felelőssége, a TTP csak CRC-t biztosít.



A CNI felépítése:

A CNI az idővezérelt architektúra legfontosabb interfésze, mert ez az egyetlen interfész, amely a host szoftvere által látható. A Status Registereket a TTP vezérlő írja, a Control Registereket pedig a host.

Status Registers	Control Registers
(S1) Global Internal Time	(C1) Watchdog
(S2) Node Time	(C2) Timeout Register
(S3) Message Description List	(C3) Mode Change Request
(S4) Membership	(C4) Reconfiguration Request
(S5) Status Information	(C5) External Rate Correction

S1: A fürt közös órája két bájton. S2: a vezérlő saját órája. S3: MEDL Pointer. S4: annyi bitből áll, ahány csomópont van a fürtben. Ha egy bit "TRUE", akkor működött az illető csomópont a legutolsó kommunikációs időszeletben, ha "FALSE", akkor nem működött. C1: A host periodikusan frissíti, a vezérlő ellenőrzi. Ha elmarad a frissítés, akkor a vezérlő – hibát sejtve - leállítja az üzenetküldést. C2: A host írja, lejártakor megszakítást okoz. Például a host a fürt órájához szinkronizálhatja magát egy előírt későbbi időben. C3: Például új ütemezésre lehet áttérni ennek segítségével. C4: Meghibásodás esetén szerepcsere kezdeményezhető. C5: külső óra szinkronizálást (pl. GPS) tesz lehetővé.

A Message Description List (MEDL) felépítése

Node Time	Address	D	L	I	A
Mikor	*Mit: Az üzenet címe*	*irány*	*hossz*		

I: azt adja meg, hogy inicializálással kapcsolatos üzenet, vagy normál üzenet. A: egy további paraméter-mező, amely a változtatásokkal (mode changes) kapcsolatos információkat tartalmaz.

A hibatűrő egységek (Fault-Tolerant Units) rendeltetése egy csomópont hibájának a maszkolása. Ha a csomópont a fail-silent absztrakciót valósítja meg, akkor a csomópontok duplikálása elegendő egyszeres csomópont-hiba tolerálására. Ha a csomópont nem valósítja meg a fail-silent absztrakciót, de lehet érték-hibája a CNI-nél, akkor háromszoros moduláris redundancia (TMR: triple modular redundancy) valósítandó meg. Ez „háromból kettő" szavazással maszkolja az érték-hibát. Ha a csomópont hiba esetén fellépő viselkedéséről semmit sem tudunk, azaz akár bizánci típusú hiba is felléphet, akkor négy csomópont tudja maszkolni a hibát.

Teljesítőképesség határok TT rendszerekben:

Tegyük fel, hogy az egy-egy üzenet továbbítására szánt keretek 20 µs időtartamúak, és 80%-os a sávszélesség kihasználás, tehát 5 µs az ún. inter-frame-gap. Ez a 25 µs gyakoriság 40 000 üzenet/sec üzenettovábbítási sebességet tesz lehetővé. Ha 10 csomópontot foglal magába a fürt(klaszter), akkor ez csomópontonként 4kHz-es mintavételi frekvenciát jelent. Természetesen a 20 µs alatt átvihető adatmennyiség a sávszélesség függvénye.

Példa: 5Mbit/s sávszélesség esetén $5*10^{6}*20*10^{-6}$ = 100 bit (~12 byte) vihető át.
Példa: 1Gbit/s sávszélesség esetén $1*10^{9}*20*10^{-6}$ = 20 000 bit (2500 byte) vihető át.

Szinkronizáció ET és TT rendszerek között

A csomópont host gépe eseményvezérelt (ET) módon működik, a hálózat pedig idővezérelt (TT). Ez utóbbi azt jelenti, hogy a hálózati kommunikáció interfésze (CNI) nem blokkolható következmények nélkül. A hálózat felőli írást vizsgáljuk.
NBW: Non-blocking Write Protocol: Egy író (CNI), több olvasó (a host taskjai) van a rendszerben. A CNI blokkolás nélkül (hand-shake nélkül) átírja a DPRAM tartalmát, ami azonban egybeeshet egy olvasással → inkonzisztencia léphet fel. Ha az olvasó észleli az interferenciát, akkor megpróbálja újra mindaddig, amíg meg nem kapja a konzisztens verziót. Az olvasási próbálkozások száma korlátos kell legyen. A protokollnak szüksége van egy együttfutó vezérlő mezőre (Concurrency Control Field: CCF), amelyhez a hozzáférést kölcsönösen kizáró módon, hardverrel kell garantálni. Ezt nullára kell inicializálni és az író által inkrementálni az írás megkezdése előtt, majd a végén is. Az olvasó az olvasás művelet előtt olvassa a CCF-et és ha páratlan, akkor azonnal ismétel, ha nem, akkor az olvasás végén ellenőrzi, hogy változott-e a CCF, azaz történt-e írás időközben, ha igen, akkor újból próbálkozik.

Inicializálás: CCF:=0

Írás:

```
Start:  CCF_old:=CCF;
        CCF:=CCF_old+1;
        <írás>
        CCF:=CCF_old+2;
```

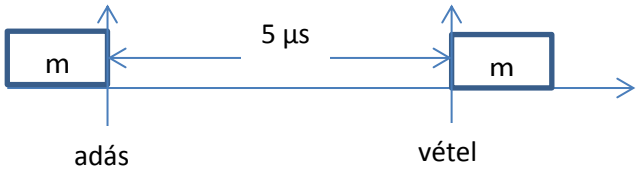
Olvasás:

```
Start:  CCF_begin:=CCF;
        if CCF_begin=odd then goto Start;
        <olvasás>
        CCF_end:=CCF;
        if CCF_end ≠ CCF_begin then goto Start;
```

Az olvasási próbálkozások száma korlátos, ha az írások közötti idő lényegesen nagyobb, mint maga az írás, vagy az olvasás.

Kommunikációs közeg jellemzők:

A csatorna jellemzők: (1) sávszélesség 10kbit-sec -> 1Mbit/sec pl. autóban, vezetéken, Gbit/sec üvegszálon. (2) terjedési sebesség/késleltetés: 300 000 km/sec, 1 láb/nsec. Kábelben ennek 2/3-a. Pl.: 5 µsec kell 1km megtételéhez. (3) csatorna bit hosszúsága: azon bitek száma, amelyek a terjedési késleltetés alatt átérnek. Pl. 100 Mbit/s sávszélesség mellett 200 m hosszú kábelen a bit hosszúság 100 bit, mert a terjedési késleltetés ezen a hosszon 1 µsec. (4) adat hatékonyság: buszon történő kommunikáció esetén ki kell várni minimálisan egy terjedési késleltetésnyi időt ahhoz, hogy az adó újból adhasson (a buszon lévő tartalmat kimerevítjük addig, amíg a vevő nem olvassa be onnan az adatot. Az adat hatékonyság < $m/(m+bl)$, ahol m az üzenethossz, bl pedig a csatorna bithosszúsága. Pl. 1 km hosszú buszon, 100 Mbit/s sávszélesség mellett bl = 500, ha az üzenet 100 bit, akkor az adat hatékonyság: 100/(100+500) = 16.6%.



Megjegyzések a kommunikáció témaköréhez:

1. A fizikai réteg szintjén a kommunikáció:

(1) aszinkron: ha szinkronizációra alkalmas jelszint-átmenet csak az üzenet elején van. Pl. az UART (Universal Asznchronous Receiver Transmitter) ilyen: általában rövid üzenet, pl. 10bit, így olcsó oszcillátor is elég (pl. 10^{-2}sec/sec).
(2) szinkron: menet közben is szinkronizál, mert vannak erre használható szintátmenetei.
Megjegyzés: Figyeljük meg, hogy az aszinkron és szinkron itt mást jelent, mint általában!

Példák:

NRZ kód (non-return-to-zero)***:*** nem szinkronizáló
11010001: "1" magas szint, "0" alacsony szint
Manchester kód: szinkronizáló
11010001: "1" felfutó él, "0" lefutó él két órajel között "félúton": az éppen következő bit dönti el, hogy az "órajel" idejében vissza kell-e futni a másik szintre, vagy sem. Hátránya, hogy 1/2 bit-cellával jellemezhető: kétszer is változhat a jel, míg más kódokban legfeljebb egyszer.
Módosított frekvenciamodulációs kód: szinkronizáló
11010001: órajel és adatjel pozíciók helyezkednek el egymást váltva. "1": jelváltás történik, "0": nem történik jelváltás az adatjel pozícióban. Ha több, mint két "0" van egymás után, akkor az órajel pozícióban jelváltás lesz.

2. A protokoll tervezés alapvető konfliktusai

A kiegyensúlyozott protokoll tervezés törekszik számos szempont összeegyeztetésére. Vannak azonban olyanok, amelyek nem egyeztethetők össze. Az alábbiak ezek közül mutatnak be néhányat buszon történő kommunikáció esetében.

Külső vezérlés ↔ komponálhatóság

Képzeljünk el egy elosztott valós-idejű rendszert. Minden csomóponthoz tartozik egy host számítógépe, aminek van agy kommunikációs hálózat interfésze (Communication Network Interface: CNI). Az időtartománybeli komponálhatóság megkívánja, hogy:

- a CNI teljesen specifikált legyen az időtartományban;
- a rendszerbe további csomópontok integrálása semmilyen változást nem idéz elő az egyes CNI-k időbeni tulajdonságait illetően,
- minden host időbeni tulajdonságai a CNI-től függetlenül tesztelhetők.

Ha az időbeni tulajdonságok nincsenek benne a CNI specifikációjában, akkor, például azért, mert az üzenetküldés időpontja *külső* és *ismeretlen* információ a kommunikációs rendszer számára, nem lehetséges a komponálhatóság biztosítása az időtartományban. Ha a CNI időbeni tulajdonságai teljes mértékben specifikáltak, akkor az alacsonyszintű komponálhatóság elérhető. (Az alkalmazói program szintjén ettől függetlenül lehetnek olyan jósolhatatlan kimenetű kölcsönhatások, amelyek az alacsonyszintű kommunikációs interfészben nem detektálhatóak, és amelyek a magas szintű komponálhatóságot kizárják.)

Az eseményvezérelt rendszerek esetében az időbeni vezérlés jelei külső forrásból származnak, ezért a csomópontok host számítógépeiben az alacsonyszintű komponálhatóság nem biztosítható.

Példa: Ha a csomópontok mindegyike bármikor versenyezhet az egyetlen kommunikációs csatorna birtoklásáért, akkor képtelenség elkerülni az ütközések következtében fellépő átviteli késletetést bármilyen okos közeg-hozzáférési protokollt alkalmazunk is. A járulékos átviteli/kommunikációs késleltetések érvényteleníthetik a valós idejű képek időbeni pontosságát.

Flexibilitás ↔ hibadetektálás

A flexibilitás azt jelenti, hogy a csomópont viselkedése nincs előzetesen korlátozva. Replikátum nélküli architektúrában a hiba detektálása csak akkor lehetséges, ha az aktuális viselkedés összevethető a várt viselkedésre vonatkozó előzetes (a priori) ismerettel. Ha ilyen ismeret nem áll rendelkezésre, akkor a hálózat nem védhető meg a hibás csomópontjától.

Példa: Ha egy eseményvezérelt rendszerben periodikus működést nem tételezhetünk fel, ha nincs az üzenetküldés gyakoriságának valamilyen korlátja, akkor nem kerülhető el, hogy egy (esetlegesen hibás) csomópont ne sajátítsa ki a hálózatot.

Példa: Ha egy csomópontot nem kényszerítünk arra, hogy rendszeres időközönként "szívdobbanás" jellegű üzeneteket küldjön, akkor nem lehetséges a csomópont hibáját detektálni (valamilyen korlátozott késleltetéssel).

Sporadikus adat ↔ periodikus adat

Egy valós idejű protokoll lehet hatékony periodikus adatra, és sporadikus adatra, de egyidejűleg mindkettőre nem. Periodikus adat továbbítása (Pl. szabályozási hurkok koordinálására használt adatok esetében) minimális kommunikációs késleltetéssel (latency jitter) kell megtörténjen. Mivel a periodikus adat továbbításának ideje előzetesen ismert, ezért konfliktusmentes ütemezések készíthető off-line. A sporadikus adatokat kérésre, és minimális késleltetéssel kell továbbítani. Ha a külső kérés időpontja egybeesik a periodikus adat továbbítási időpontjával, akkor el kell dönteni, hogy melyiket késleltetjük. Bármelyik esetben a kommunikációs késletetés nő, azaz mindkét cél egyidejű kielégítése nem lehetséges.

Egy pontról történő vezérlés ↔ hibatűrés

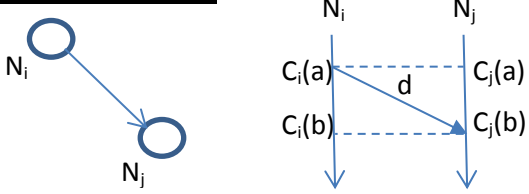
Minden protokoll, amelyet egy pontról vezérelünk, egy meghibásodási ponttal rendelkezik. Ez egyértelmű egy központi master-en alapuló kommunikáció protokoll esetében. De ilyen módon viselkedik minden időpillanatban egy token-passing rendszer is: ha token-t birtokló csomópont elromlik, akkor nincs tovább kommunikáció mindaddig, amíg nem detektáljuk a token elvesztését járulékos time-out mechanizmussal, és helyre nem állítjuk a token-t. Ez időt vesz igénybe, és megszakítja a valós-idejű kommunikációt. Bizonyos értelemben a token helyreállítás nemtriviális problémája kapcsolódik a központi master átkapcsolása egy standby masterre problémához egy multi-master protokollban.

Valószínűségi hozzáférés ↔ a replikátum determinizmusa

Ha közeg-hozzáférés valószínűségi mechanizmusokkal (pl. a konfliktus feloldás véletlen számok alkalmazásával) történik, akkor aktív redundancia igénye esetén nem garantálható, hogy a replikátum a versengő csomópontok közül ugyanazt hozza ki győztesnek. A replikátum determinizmusa nélkül a replikátum eltérő helyes eredményre juthat, ez azonban a rendszer egészében inkonzisztenciát eredményez.

3. Időszinkronizáció vezetéknélküli hálózatokban

Egyirányú szinkronizáció:

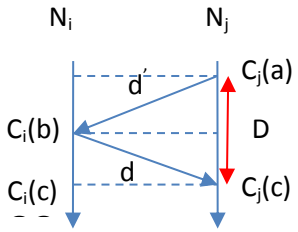


Az N_j csomópont nem ismeri d-t, csak azt, hogy az N_i csomópont órája $C_i(a)$ értéket azt megelőzően mutatott, hogy az N_j csomópont órája $C_j(b)$-t. Ahhoz, hogy szinkronizálni tudjunk vagy $C_j(a)$ vagy $C_i(b)$ értékét meg kell becsüljük. Ha ismert $d_{min} \leq d \leq d_{max}$, akkor

$$\hat{C}_j(a) \approx C_j(b) - \frac{d_{min}+d_{max}}{2} \text{ vagy } \hat{C}_i(b) \approx C_i(a) + \frac{d_{min}+d_{max}}{2}.$$

Ezek ismeretében az N_j csomópont óráját vagy $\hat{C}_j(a) - C_i(a)$ értékkel vagy $C_j(b) - \hat{C}_i(b)$ értékkel kell késleltetnünk/visszaállítanunk. Ha a kommunikáció jittere (d_{max}-d_{min}) nagy, akkor az így végrehajtott szinkronizáció pontatlan lesz, hiszen például $C_j(a)$ alsó határa $C_j(b) - d_{max}$, felső határa pedig $C_j(b) - d_{min}$ értékkel adható meg, ami ilyenkor széles tartomány.

Kétirányú (Oda-vissza, round trip) szinkronizáció:



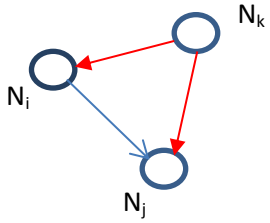
Az N_j csomópont tudja, hogy $0 \leq d \leq D$. $D = C_j(c) - C_j(a)$. Ha $d_{min} \leq d \leq d_{max}$, akkor $\max(D - d_{max}, d_{min})$ és $\min(d_{max}, D - d_{min})$ adják d korlátait. Az itt számítható becslő:

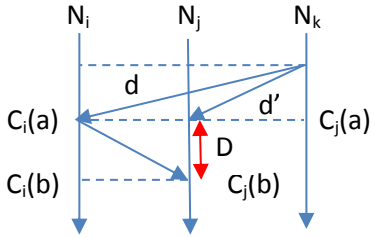
$\hat{C}_j(b) \approx C_j(c) - \frac{D}{2}$, aminek alsó határa $C_j(c) - (D - d_{min})$, felső határa pedig $C_j(c) - d_{min}$.

Itt az N_j csomópont óráját a $\hat{C}_j(b) - C_i(b)$ értékkel kell késleltetni/visszaállítani. Ezzel a megoldással jobb minőségű szinkronizáció érhető el. A worst-case szinkronizációs hiba: $\frac{D}{2} - d_{min}$, ami az ábra alapján is könnyen belátható. A módszer pontossága javítható az ún. *valószínűségi idő szinkronizációval*, amely esetében a vétel után az N_j csomópont ellenőrzi, hogy a $\frac{D}{2} - d_{min} <$ egy specifikált küszöbnél. Ha nem, akkor ismétel.

Anoním (Reference broadcasting) szinkronizáció:

Az egyirányú és a kétirányú előnyeit ötvözi.





A szinkronizáció kezdeményezője az N_k csomópont. Azt használjuk ki, hogy miközben a kommunikáció ideje változó, a broadcasting jellegből adódóan $d \approx d'$. Ezzel

$$\hat{C}_i(b) \approx C_i(a) + D,$$

aminek ismeretében az N_j csomópont óráját $C_j(b) - \hat{C}_i(b)$ értékkel kell késleltetnünk/visszaállítanunk. Fontos sajátosság, hogy az N_j csomópont szinkronizálása úgy valósul meg, hogy eközben a rádiójának nem kell adnia.

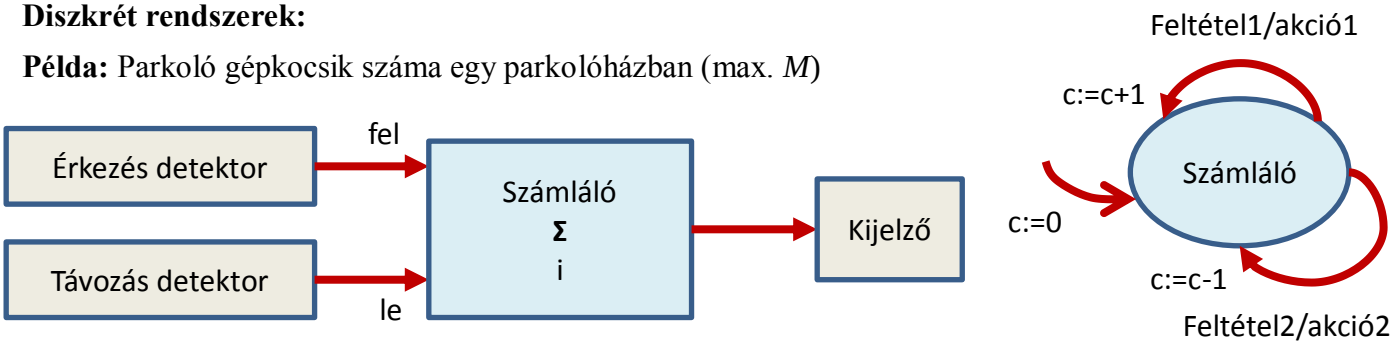
8. Esettanulmányok

8.1. Hibrid rendszerek

Szakaszosan folytonos, általában dinamikus rendszerek, amelyeknél a szakaszhatáron markáns állapotváltozás következik be. Ebből a szempontból tekinthetők olyan diszkrét rendszereknek, amelyek állapotátmenetei közötti idő értelmezett, és ezalatt történik is valami.

Diszkrét rendszerek:

Példa: Parkoló gépkocsik száma egy parkolóházban (max. *M*)



Feltétel1/akció1: $fel \wedge \neg le \wedge c < M \,/\, c+1$ Feltétel2/akció2: $le \wedge \neg fel \wedge c > 0 \,/\, c-1$

Példa: *Termosztát hiszterézissel:*

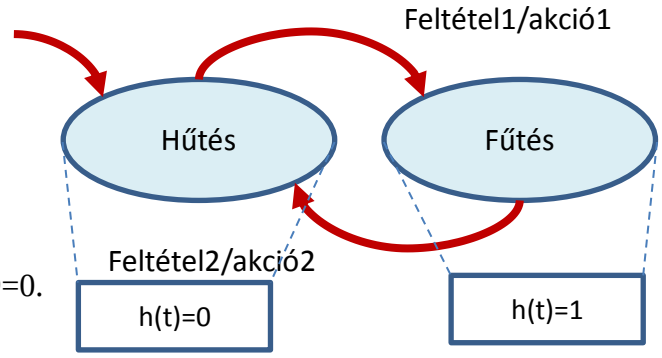
Feltétel1/akció1: Hőmérséklet ≤ 18 fok/fűtés_be

Feltétel2/akció2: Hőmérséklet ≥ 22 fok/fűtés_ki

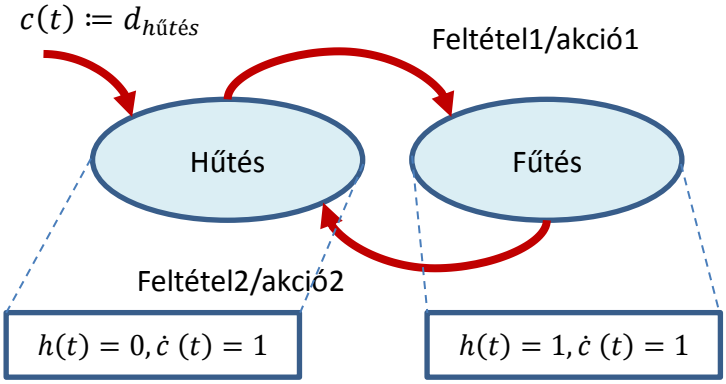
A rendszer bemenete: a környezeti hőmérséklet

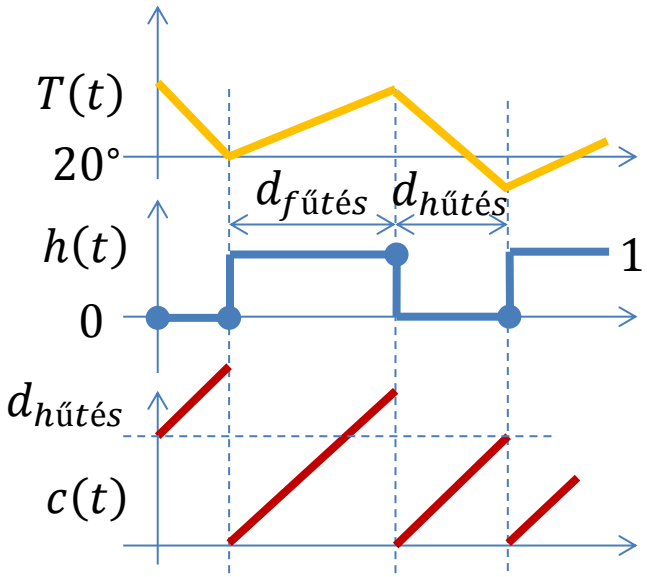
A rendszer kimenete: Fűtés_be, Fűtés_ki parancsok:

az ennek megfeletethető időfüggvények: *h(t)*=1, *h(t)*=0.



Példa: *Termosztát hiszterézis helyett időzítéssel:* ehhez az ún. **időzített automata** modellt alkalmazzuk, amelyik a legegyszerűbb nemtriviális hibrid rendszer. Ezek az automaták az állapotaik mögött (adott időtartamig) mérik az idő múlását: $\forall t \in d_m$ (valamilyen időtartam) $\dot{c}(t) = a$, azaz változik az óra értéke az idő múlásával.





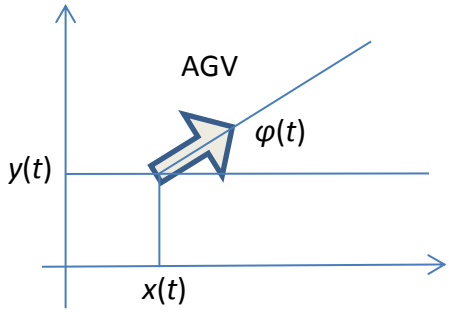
Feltétel1/akció1: $T(t) \leq 20 \wedge c(t) \geq d_{hűtés} / c(t) = 0$.

Feltétel2/akció2: $T(t) \geq 20 \wedge c(t) \geq d_{fűtés} / c(t) = 0$.

Megjegyzés: (1) *h(t)* és *c(t)* tekinthetők az állapotfinomítás eszközeinek. Szokás (üzem)módról beszélni. (Modal systems). (2) Az idődiagramon vázolt esetben a *T*>20 foknál. Abban az esetben, ha kevesebb, akkor a rendszer azonnal fűteni kezd: az óra indítási/kezdeti értékek tehát ilyen megfontolás(ok) mentén születhetnek.

Példa: *Önjáró targonca.* (Automated Guided Vehicle, AGV)

Két szabadságfokú jármű, felfestett csík követésére képes. Minden *t* időpontban a hossztengelye mentén *v*(*t*) sebességgel mozog, azzal hogy $0 \le v(t) \le 10km/h$. A súlypontja körül fordulni is tud $\omega(t)$ szögsebességgel, azzal hogy $-\pi \le \omega(t) \le \pi\, rad/sec$.



$$\dot{x}(t) = v(t)\cos(\varphi(t))$$
$$\dot{y}(t) = v(t)\sin(\varphi(t))$$
$$\dot{\varphi}(t) = \omega(t)$$

Kétszintű szabályozás: a targonca mindig 10 *km/h* sebességgel halad. Négy működési módja van: **balra, jobbra, egyenesen, megállás.**

Minden működési módhoz külön differenciálegyenlet tartozik.

egyenesen:

$$\dot{x}(t) = 10\cos(\varphi(t))$$
$$\dot{y}(t) = 10\sin(\varphi(t))$$
$$\dot{\varphi}(t) = 0$$

balra:

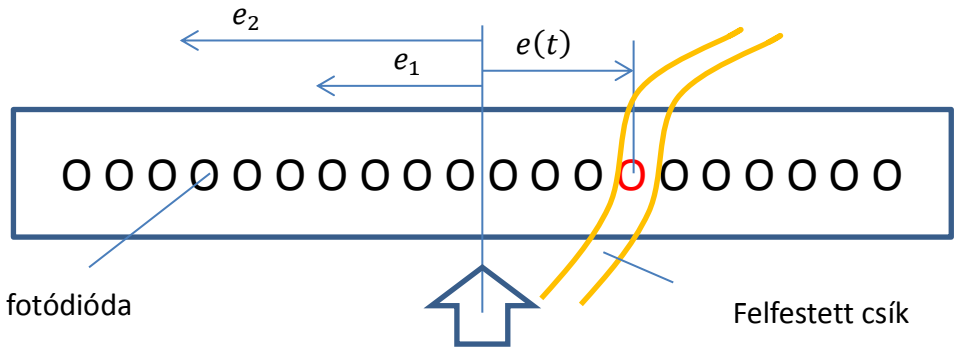
$$\dot{x}(t) = 10\cos(\varphi(t))$$
$$\dot{y}(t) = 10\sin(\varphi(t))$$
$$\dot{\varphi}(t) = \pi$$

jobbra:

$$\dot{x}(t) = 10\cos(\varphi(t))$$
$$\dot{y}(t) = 10\sin(\varphi(t))$$
$$\dot{\varphi}(t) = -\pi$$

megállás:

$$\dot{x}(t) = 0$$
$$\dot{y}(t) = 0$$
$$\dot{\varphi}(t) = 0$$



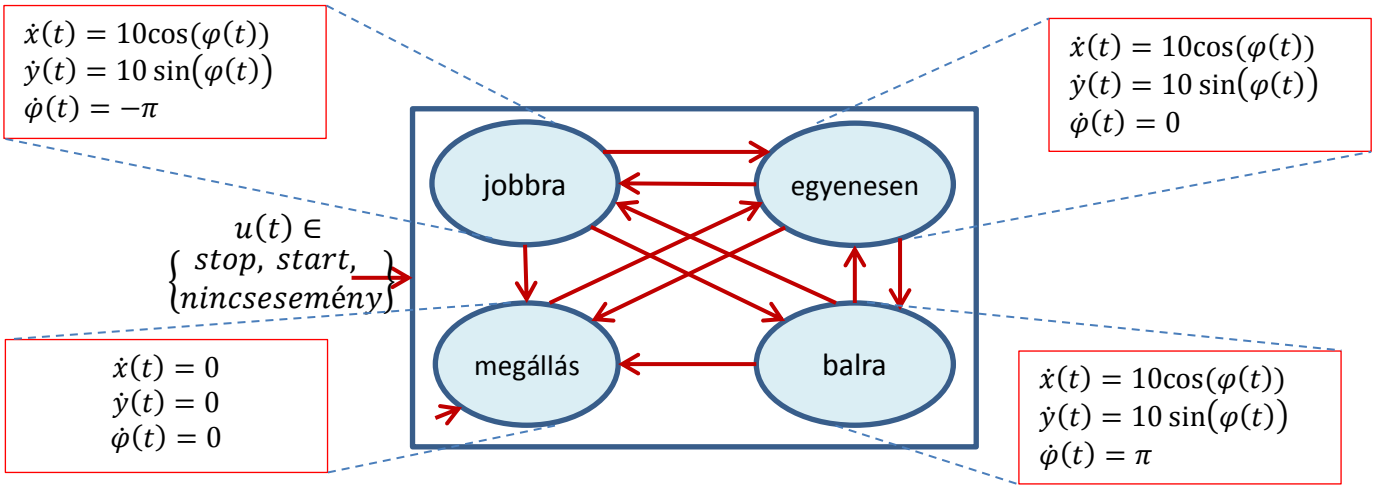
A targonca érzékelője: a haladási irányra merőleges fotódióda sor, kimenőjele $e(t) = f(x(t), y(t))$. Ha $e(t) > 0$, akkor balra tér el, ha $e(t) < 0$, akkor jobbra tér el.

A targonca vezérlése: ha $|e(t)| < e_1$, akkor egyenesen haladjon tovább;
$0 < e_2 < e(t)$, túlságosan eltér balra, forduljon jobbra;
$0 > -e_2 > e(t)$, túlságosan eltér jobbra, forduljon balra.

A bemeneti események halmaza: $u(t) \in \{stop, start, nincs\_esemény\}$. Mivel a *stop* és a *start* pillanatszerű események, a *nincs_esemény* a köztes időkre adja meg *u*(*t*) értelmezését.

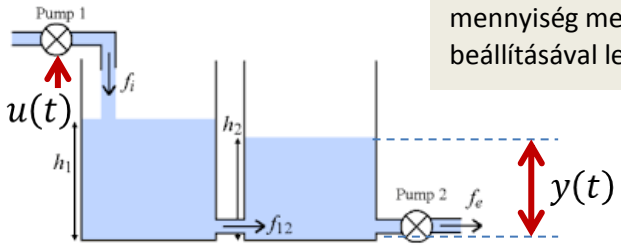
Állapotátmenetet generáló feltételek:

$$indulj\_el = \{(v(t), x(t), y(t), \varphi(t)) | u(t) = start\}$$
$$menj\_egyenesen = \{(v(t), x(t), y(t), \varphi(t)) | u(t) \neq stop, |e(t)| < e_1\}$$
$$menj\_jobbra = \{(v(t), x(t), y(t), \varphi(t)) | u(t) \neq stop, e_2 < e(t)\}$$
$$menj\_balra = \{(v(t), x(t), y(t), \varphi(t)) | u(t) \neq stop, -e_2 > e(t)\}$$
$$állj\_meg = \{(v(t), x(t), y(t), \varphi(t)) | u(t) = stop\}$$



8.2. Nemkonvencionális modellezés és szabályozás – hatékony implementáció

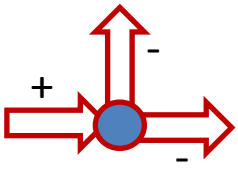
1. Példa: Kvalitatív modellezés és szabályozás I.
Feladat: ***Olyan szabályzó tervezése, amely a második tartály y(t) szintjét előírt értéken tartja.***



Ez a szivattyúzandó *u(t)* mennyiség megfelelő beállításával lehetséges.

A kvantitatív modell problémái:
a) A fizikai korlátok nincsenek beépítve;
b) Az összefüggéseket linearizáljuk;
c) A numerikus értékek nem pontosak és időben változnak,

Kvalitatív okoskodás (Qualitative Reasoning): Csak a mennyiségek irányultságát vesszük figyelembe, az értékkészlet: $\{-,0,+\}$. Alapvető fizikai kényszereket betartunk!



Ha egy csomóponti elágazásnál két ágon kifolyik az "anyag", akkor a harmadikon befolyik.
Egy "Q" mennyiség kvalitatív értéke egy "a" értékre vonatkoztatva: $[Q]_a$
Egy "Q" mennyiség megváltozásának kvalitatív értéke a kvalitatív derivált: $[\delta Q]_a$, $[\delta^2 Q]_a$, …

Műveletek: $(invert\ A)$: Megfordítja az előjelet.
$(vote\ A_1,\ A_2,\ \ldots,\ A_n)$: Értéke a többségi előjel.

A 2. tartály szintjének kvalitatív szabályozása: L_2 jelöli a második tartály szinthibáját:

$[L_2] = +$: magasabb, mint kellene.	$[\delta U] = +$: a szivattyúzás növelendő.
$[L_2] = 0$: megegyezik.	$[\delta U] = 0$: a szivattyúzás mértéke megfelelő.
$[L_2] = -$: alacsonyabb, mint kellene.	$[\delta U] = -$: a szivattyúzás mértéke csökkentendő.

$[\delta U] = +$: rögzített értékű növekmény: ΔU.
A kvalitatív értékek csak a mintavételi időpontokban léteznek. A mintavételi időpontok között nincsen detektálás.

$$[L_2]_{(k)} = \left[aktuális\ szint_{(k)} - megkívánt\ szint_{(k)}\right]$$

Egy igen egyszerű szabályzó:

$$Q1 \overset{def}{=} [\delta U]_{(k)} = (invert[L_2])_{(k)}$$

Megjegyzés: Ha a ΔU nagyobb érték, akkor nő a túllövés és az oszcilláció, de gyorsan reagál. Ha ΔU kisebb érték, akkor csökken a túllövés és az oszcilláció, de lassabb a működés.

Javított szabályzók:

Figyelembe vett mennyiségek:

A 2. tartály szinthibája:	+,0,-	} 3*3*3=27 eset.
A 2. tartály szintváltozási sebessége:	+,0,-	
Az első tartály szintváltozási sebessége:	+,0,-	

$$Q2 \overset{def}{=} [\delta U]_{(k)} = \Big(invert\Big(vote\big(vote\big([L_2]_{(k)}, [\delta L_2]_{(k)}\big), [\delta L_1]_{(k)}\big)\Big)\Big)_{(k)}$$

$$Q3 \overset{def}{=} [\delta U]_{(k)} = \Big(invert\Big(vote\big([L_2]_{(k)}, [\delta L_2]_{(k)}, [\delta L_1]_{(k)}\big)\Big)\Big)_{(k)}$$

A $[\delta L_1]$ meghatározása $\delta L_1 = \big(L_{2(k)} - L_{2(k-1)}\big) - \big(L_{2(k-1)} - L_{2(k-2)}\big)$ alapján mérési adatokból történik.

A 27 lehetséges kvalitatív érték kombináció esetére a három szabályzó javaslatát az alábbi táblázat foglalja össze:

	$[L_2]$	$[\delta L_2]$	$[\delta L_1]$		$Q2$	$Q3$
1	+	+	+	-	-	-
2	+	+	0	-	-	-
3	+	+	-	-	0	-
4	+	0	+	-	-	-
5	+	0	0	-	-	-
…						
20	-	+	0	+	0	0
…						
27	-	-	-	+	+	+

Megjegyzés: (1) A feladatra empirikusan kidolgozott szabályrendszer nem tudta kezelni a "A 2. tartály a kívánt szint felett állandó értéket mutat, az 1. tartály szintje esik." (2) A mintavételezési idő és a ΔU érték megválasztása kritikus tervezői döntés.

2. Példa: Kvalitatív modellezés és szabályozás II.

Feladat: ***A fordított inga kvalitatív modellezése nemdeterminisztikus automatával.***

Olyan rendszerek esetében, amikor az *x(k)* állapotvektorról csak egy [*x(k)*] kvantált érték ismert.

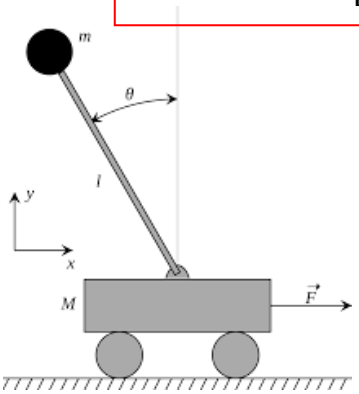
Ok/létjogosultság: szög és szögsebesség mérés pontatlansága.

Linearizált modell $\theta = 0$ környezetében:

$$\dot{x}(t) = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & -\frac{mg}{M} & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & \frac{(m+M)g}{Ml} & 0 \end{bmatrix} x(t) + \begin{bmatrix} 0 \\ \frac{1}{m} \\ 0 \\ -\frac{1}{Ml} \end{bmatrix} u(t)$$

$$x(t) = \begin{bmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \end{bmatrix}$$

$$u(t) = F$$

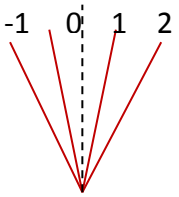


$$M = 1kg,\ m = 0.1kg,\ l = 0.5m,\ g = 9.81\frac{m}{s^2}$$

A mérési érzéketlenség: 0.0175 rad a θ-ra, és 0.0175/20ms a $\dot{\theta}$-ra.
Nem stabilizálható a rúd, ha $|x_3| > 0.21\ rad$ (12°), és $|x_4| > 0.87$.

A szögre (3-as index) és a szögsebességre (4-es index) a tartomány határok az alábbi ábra szerint:

$$g_{3,-1} = -0.210, g_{3,0} = -0.0175, g_{3,1} = 0.0175, g_{3,2} = 0.210$$
$$g_{4,-1} = -0.870, g_{4,0} = -0.0175, g_{4,1} = 0{,}0175, g_{4,2} = 0.870$$



A két középső tartományban tartózkodást 0-val, a baloldaliban - 1-gyel, a jobboldaliban +1-gyel jelölve, a következő kvalitatív állapotok definiálhatók:

$$z_1 = \begin{bmatrix} -1 \\ -1 \end{bmatrix}, z_2 = \begin{bmatrix} -1 \\ 0 \end{bmatrix}, z_3 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, z_4 = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, z_5 = \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$
$$z_6 = \begin{bmatrix} 0 \\ 1 \end{bmatrix},\ z_7 = \begin{bmatrix} 1 \\ -1 \end{bmatrix},\ z_8 = \begin{bmatrix} 1 \\ 0 \end{bmatrix},\ z_9 = \begin{bmatrix} 1 \\ 1 \end{bmatrix},\ z_{10} = kívül,$$

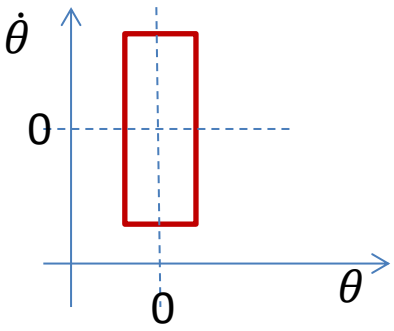
A bemeneten a "rántás" kvalitatív értékei:

$$u(k) = 10 \Leftrightarrow v(k) = 1, \qquad u(k) = 0 \Leftrightarrow v(k) = 0, \qquad u(k) = -10 \Leftrightarrow v(k) = -1$$

A kvalitatív állapotokhoz megfelelő beavatkozást rendelve a rúd stabilizálható:

$z(k)$	z_1	z_2	z_3	z_4	z_5	z_6	z_7	z_8	z_8
$u(k)$	-1	0	0	-1	0	1	0	0	1

A kvalitatív szabályzó: $[u(k)] = f([x(k)])$



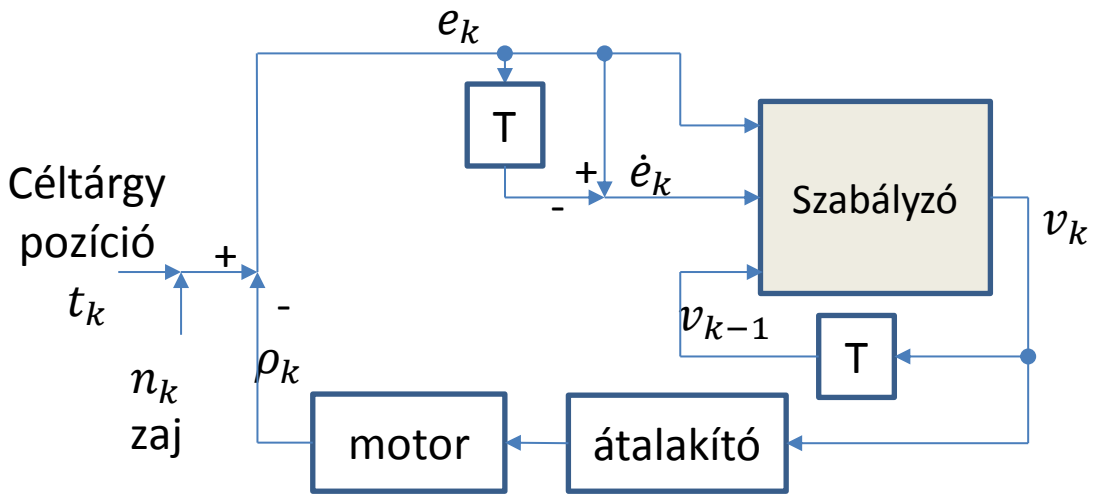
Megjegyzés: A T mintavételezési idő és az F érték megválasztása kritikus tervezői döntés.

Az ábrán a mozgás trajektóriájának idealizált változata látható. A zaj-zavar hatások következtében a helygörbe nem teljesen önmagába visszatérő jellegű.

3. Példa: ***Adaptív célkövető rendszer fuzzy modellezéssel/szabályozással***

A célkövető két forgatómechanizmussal rendelkezik: az egyik azimut (0 … 180 fok) irányú, a másik emelkedés (0 … 90 fok) irányú. Az azimut az a szög, amit a kijelölt irány vízszintes vetülete a déli vagy az északi iránnyal bezár.

Szenzor: minden olyan eszköz alkalmas, amely kellő pontossággal képes a célra mutatni: Laser, videokamera, nagy nyereségű antenna.



Jelölések:

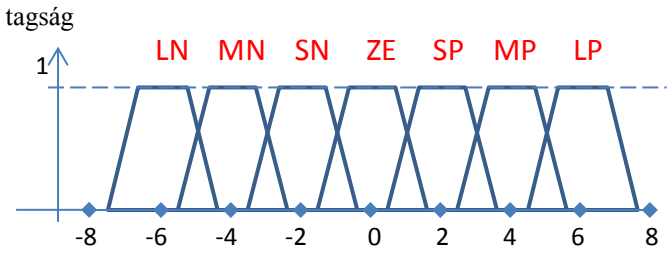
t_k	céltárgy pozíció;	n_k	megfigyelési zaj;	ρ_k	célkövető pozíció
e_k	követési hiba;	$\dot{e}_k$	köv. hibaváltozás;	v_k	becsült szögsebesség
T	mintavételi idő.				

$$\rho_k = \rho_{k-1} + Tv_{k-1} + hiba$$

hiba = pozicionálási bizonytalanság

Fuzzy szabályzó: A becsült szögsebesség tartománya: [-6,6]. (Ez egy tervezői döntés, ezzel rögzül a skálázás.) Mivel $|v_k| \leq \frac{9.0}{T} fok/sec$ azimut irányban, és $|v_k| \leq \frac{4.5}{T} fok/sec$ emelkedés irányban, ezért az egyes csatornák erősítései: $1.5/T$ és $0.75/T$.

Maga a szabályzó heurisztikus szintállító szabályokat tartalmaz az e_k, $\dot{e}_k$ és v_{k-1} értékei alapján. Hét fuzzy szint értéket definiálunk tagsági függvény megadásával:



LN=Large Negative
MN=Medium Negative
SN=Small Negative
ZE=Zero
SP=Small Positive
MP=Medium Positive
LP=Large Positive

Minden bemenethez egy hételemű vektort rendelünk:

1 →	(0	0	0	0.7	0.7	0	0)
-4 →	(0	1	0	0	0	0	0)
3.8→	(0	0	0	0	0.1	1	0)

A figyelembe vett értékekhez ún. fuzzy-asszociatív-memória (FAM) szabályokat rendelünk: Például: az *i*-edik szabály:

$$IF\ e_k = MP \wedge \dot{e}_k = SN \wedge v_{k-1} = ZE\ THEN\ v_k = SP$$

Rövidített formában: (*MP,SN,ZE;SP*). Az *i*-edik FAM szabály skalár értéke: $w_i = \min(tagsági\ értékek)$.

Példa: $e_k = 2.6,\ \dot{e}_k = -2.0, v_{k-1} = 1.8$. Az ezekhez rendelt hételemű vektorok:

LN	MN	SN	ZE	SP	MP	LP
0	0	0	0	1	0.4	0
0	0	1	0	0	0	0
0	0	0	0.1	1	0	0

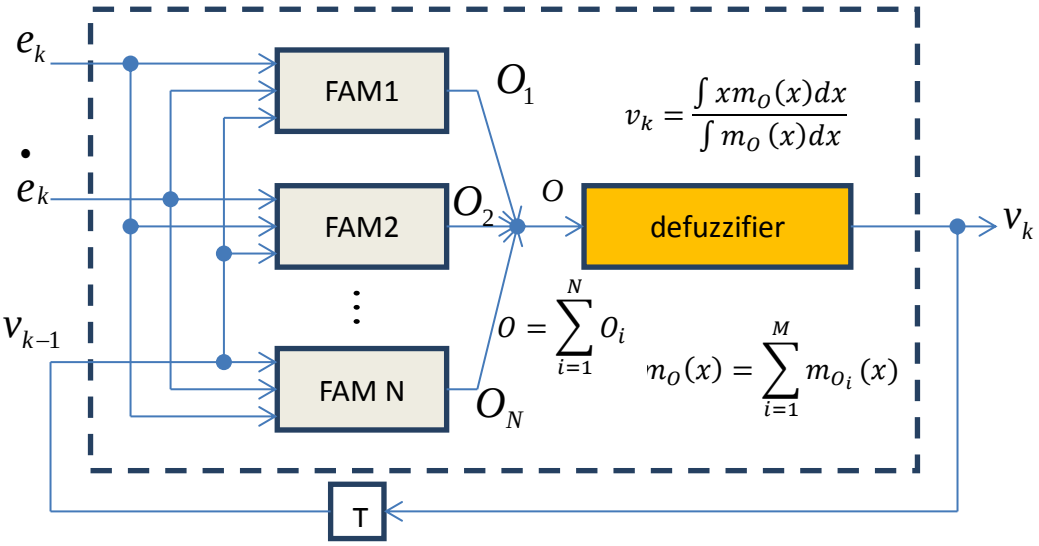
A szabályhoz kapcsolódó tagsági értékek:

$$m_{MP}(e_k) = 0.4$$
$$m_{SN}(\dot{e}_k) = 1$$
$$m_{ZE}(v_{k-1}) = 0.1$$

Az i-edik szabály skalár értéke:

$$w_i = \min(0.4, 1, 0.1) = 0.1$$

A szabályzó kialakítása:



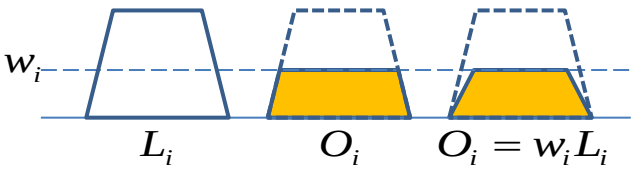
A kimeneti fuzzy halmaz alakja a FAM szabály kódolásától függ:

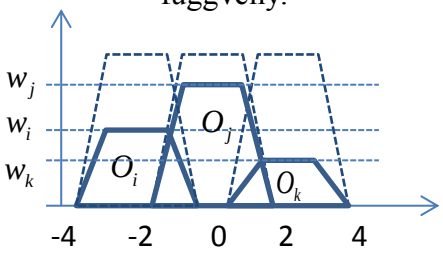
Korreláció-szorzat kódolás:
$m_{O_i}(x) = w_i m_{L_i}(x)$

Korreláció-minimum kódolás:
$m_{O_i}(x) = \\ = \min(w_i, m_{L_i}(x))$.

Itt $m_{L_i}(x)$ az i-edik FAM szabály kimenetéhez kapcsolódó tagsági függvény.

A kimeneti fuzzy halmaz származtatásának két lehetséges változata:





A ***defuzzifier*** numerikus értéket rendel az egyes FAM szabályok kimeneti fuzzy halmazainak összegéhez. Ez az összegzett halmaz a súlyozott trapézok, mint függvények pontonkénti összeadásával jön létre. Hasonlítható a valószínűség-számítás sűrűségfüggvényeihez azzal a különbséggel, hogy itt a görbe alatti terület nem egy. A ***defuzzifier*** által végzett számítás, melynek képlete az ábrán látható, a v_k értéket centroidként állítja elő: röviden ***fuzzy centroidnak*** nevezzük.

A fuzzy szabályzó implementációja: A FAM$_i$ szabály: (*MP,SN,ZE;SP*). A k-adik időpillanatban: $e_k = 2.6$, $\dot{e}_k = -2.0, v_{k-1} = 1.8$.

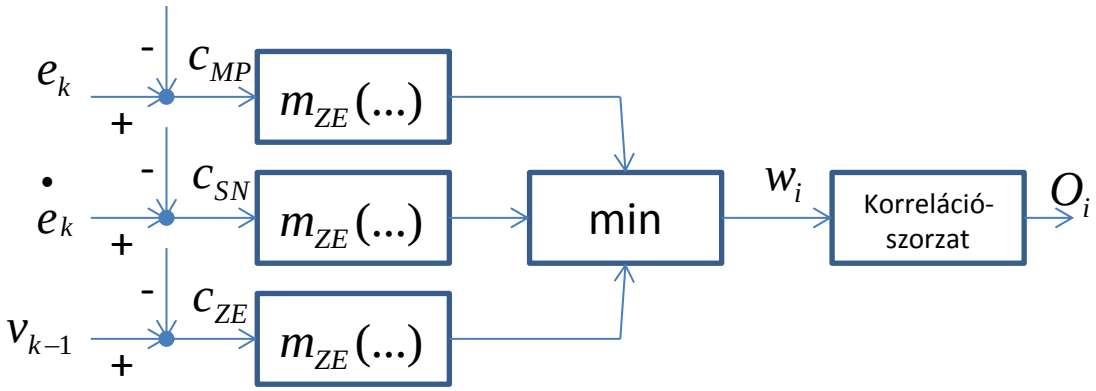
$$w_i = \min\left(m_{MP}(e_k), m_{SN}(\dot{e}_k), m_{ZE}(v_{k-1}) \right) = \min(0.4,1,0.1) = 0.1$$

Mivel a feladatban minden fuzzy halmaz alakja azonos: Pl.: $m_{SP}(x) = m_{ZE}(x-2)$. Általában $m_{L_i}(x) = m_{ZE}(x - c_{L_i})$, ahol c_{L_i} az adott tagsági függvény centroidja.

$$w_i = \min\left(m_{ZE}(e_k - c_{MP}), m_{ZE}(\dot{e}_k - c_{SN}), m_{ZE}(v_{k-1} - c_{ZE}) \right)$$

$$w_i = \min\left(m_{ZE}(-1.4), m_{ZE}(0), m_{ZE}(1.8)\right) = \min(0.4,1,0,1) = 0.1$$

Korreláció-szorzat kódolás esetén: $m_{O_i}(x) = w_i m_{ZE}(x - c_i)$, ezzel az i-edik FAM szabályimplementációja az alábbi ábra szerint történhet:



7. Beágyazott operációs rendszerek

7.1. Beágyazott rendszerek szoftver vonatkozásai: Tipikus szoftver architektúrák

Szempontok: számítási kapacitás, memóriaméret (RAM, ROM), fejleszthetőség, továbbfejleszthetőség, reakcióidő külső, aszinkron eseményesetén, védelem (memória), rekurzió, függvények újrahívásának támogatása, processzor kihasználtsága.
Tulajdonságok, amik alapján minősítjük az egyes megoldásokat:
- maximális válaszidő,
- hardverkezelés megvalósítása,
- taskok közötti kommunikáció megvalósítása,
- tervezhetőség,
- alkalmazási kör

Szoftver architektúrák osztályozása: periodikus, prioritásos, eseményvezérelt, idővezérelt
Gyakorlati megvalósítás szempontjából:
- ciklikus programszervezés
- IT-vel kiegészített ciklikus programszervezés
- ütemezett függvények módszere
- RTOS

Ciklikus programszervezés
- körforgó
- súlyozott körforgó
- idővezérelt körforgó
- szigorúan idővezérelt

Egyszerű ciklikus programszervezés: A processzor végtelen ciklusban pörög, akkor is fut, amikor senki sem igényel kiszolgálást.

```
void main() {
while (TRUE){
if (DeviceA_Needs_Service()) {Service_A};
if (DeviceB_Needs_Service()) {Service_B};
if (DeviceC_Needs_Service()) {Service_C};
...
}
}
```

Tulajdonságok:
- maximális válaszidő: tA+ tB+ tC+ ..., azaz a maximális ciklusidő.
- hardverkezelés: lekérdezéssel (polling)
- taszkok közötti kommunikáció: megosztott változókkal (nem preemptív, így nem gond!)
- fejleszthetőség: rossz
- HRT viselkedés: lassú (pl. nyomtatási taszk) (ettől még lehet RT)
- processzor kihasználtság:100% (ez NEM jó!)
- alkalmazási kör: ahol a rendszer időállandója nagyobb a ciklus futásánál (gyors és ritka események)

Súlyozott körforgó programszervezés: a gyakoribb taszkok a cikluson belül ismétlődhetnek

```
void main() {
while (TRUE){
if (DeviceA_Needs_Service()) {Service_A};
if (DeviceB_Needs_Service()) {Service_B};
if (DeviceA_Needs_Service()) {Service_A};
```

```
if (DeviceC_Needs_Service()) {Service_C};
if (DeviceA_Needs_Service()) {Service_A};
...
}
}
```

Tulajdonságok:

- max. válaszidő: tA+ tB+ tA+ tC+ tA+.., de gyakoribb taszkokra kisebb, mint a maximális ciklusidő
- hardverkezelés: lekérdezéssel (polling)
- taszkok közötti kommunikáció: megosztott változókkal (nem preemptív, így nem gond!)
- fejleszthetőség: rossz
- processzor kihasználtság: továbbra is 100%
- egyéb tulajdonság: prioritás jellegű viselkedés, de NEM preemptív

Idővezérelt körforgó programszervezés: A ciklus határokat egy timer adja (csak a határokat!). Timer IT-nként egyszer vagy többször lefut a ciklus. Egy cikluson belül lehet súlyozott körforgó.

Tulajdonságok:

- max. válaszidő: ciklus periódusideje
- hardverkezelés: lekérdezéssel (polling)
- taszkok közötti kommunikáció: megosztott változókkal
- fejleszthetőség: rossz
- processzor kihasználtság:<100%, van standby

Szigorúan idővezérelt programszervezés (time-triggered protokoll): Minden taszk futása előre meghatározott időben indul. Adminisztrálás: egy táblázatban az idők és a függvény referenciák (hiper-ciklusonként), mikro futtatórendszer figyeli az időket, és indítja a „taszkokat”.

Tulajdonságok:

- max. válaszidő: adott taszk ütemezett gyakorisága (+ a taszk futási ideje)
- hardverkezelés: lekérdezéssel(polling)
- taszkok közötti kommunikáció: megosztott változókkal
- fejleszthetősé : rossz
- processzor kihasználtság:<100% van standby
- HRT viselkedés: OK, alkalmazása biztonságkritikus rendszerekben tipikus

IT-vel kiegészített ciklikus programszervezés: nem lekérdezéssel, hanem megszakítással jelzünk.

```
FLAG A, B;
void interrupt A_Handler() { Handle_HW_A(); A=TRUE; }
void interrupt B_Handler() { Handle_HW_B(); B=TRUE; }
void interrupt C_Handler() { Handle_HW_C(); C=TRUE; }
void main() {
while (TRUE){
 if A {A=FALSE; Service_A(); }
 if B { B=FALSE; Service_B(); }
 if C { C=FALSE; Service_C(); }
...
}
}
```

Tulajdonságok:

- max. válaszidő: tA+ tB+ tC+ ...(+ IT) csak a jelzés gyorsul, a kiszolgálás nem
- hardverkezelés: megszakítással, prioritás lehetősége
- taszkok közötti kommunikáció: megosztott változókkal taszk-taszk között: nem gond. IT-taszk között: kiürítés (megosztott változók problémája)

- fejleszthetőség: IT szempontjából jó, de taszkok hozzáadásával megváltoznak a viszonyok
- alkalmazási kör: ha a taszkok futási ideje kb. azonos. Ez a legelterjedtebb.

Ütemezett függvények módszere

```
void interrupt A_Handler() { Handle_HW_A(); PutFunction(Service_A); }
void interrupt B_Handler() { Handle_HW_B(); PutFunction(Service_B); }
void interrupt C_Handler() { Handle_HW_C(); PutFunction(Service_C); }
void Service_A();
void Service_B();void Service_C();
void main() {
while (TRUE){
while (IsFunctionQueueEmpty());
CallFirstFromQueue();
}
}
```

Tulajdonságok:

- max. válaszidő: leghosszabb taszk futási ideje+ i. taszk futási ideje
- hardverkezelés: megszakítással
- taszkok közötti kommunikáció: megosztott változókkal task-task között: nem gond (nem preemptív). IT-taszk között: kiürítés (megosztott változók problémája)
- fejleszthetőség: jó
- queue-ból való kiemelés sorrendje lehet: (1) FIFO, (2) prioritás alapján
- hátrány: továbbra sem preemptív
- processzor kihasználtság:100%

Kérdés: hogyan kell kiegészíteni, hogy ne legyen 100%?

Valósidejű operációs rendszerre épített szoftver

```
void interrupt A_Handler() { Handle_HW_A(); Signal_A(); }
void interrupt B_Handler() { Handle_HW_B(); Signal_B(); }
void Service_A();
void Service_B();
void task_A(void) {
while (TRUE){
Wait_for_Signal_A(); Service_A();
}
}
void task_B(void) {
while (TRUE){
Wait_for_Signal_B(); Service_B();
}
}
```

Tulajdonságok:

- max. válaszidő: op. rendszer. jellemző adata(~10 usec) (+ a taszk futási ideje) alacsonyabb prioritású taszk esetén: nagyobb prioritású taszkok idejének összege
- hardverkezelés: megszakítással
- taszkok közötti kommunikáció: RTOS kommunikációs függvényekkel. Ez egyben szinkronizáció is.
- fejleszthetőség: nagyon jó
- HRT viselkedés: jó
- processzor kihasználtság:<100% (Mikor? Ha idle alatt sleep!)
- alkalmazási kör: bárhol alkalmazható

- hátrány: operációs rendszer plusz kódot és időt jelent

Alapfogalmak:

beágyazott OS:	kis erőforrásigény (uC-en is elfut)
valósidejű OS:	külső eseményre adott véges, determinisztikus válaszidő
taszk:	összefüggő tevékenységek sorozata
job:	taszkok részfeladatai
process:	ütemezési egység, saját memóriaterülete van (taszkokat így implementáljuk)
thread:	ütemezési egység, nincs saját memóriája
kernel:	OS magja
skálázhatóság:	OS szolgáltatásai fordítási időben ki/bekapcsolhatók, forráskóddal elérhetőség

Kernel feladatai:

- párhuzamos programozói környezet biztosítása,
- ütemezés,
- taszkok közötti kommunikáció biztosítása,
- megszakítások kezelése
- időzítés,
- memória kezelés

Skálázással bejöhet még:

- perifériák kezelése, rendszerprogramok (API)
- kommunikációs csatornák kezelése
- virtuális memória management, file rendszer stb.

7.2. Asztali és beágyazott operációs rendszerek összehasonlítása

(A valós-idejű operációs rendszerekről (RTOS) részletes leírás található a tantárgy tanszéki honlapján. Az alábbiak csak néhány kiemelt jellemzőt foglalnak össze, ill. olyan részleteket, amelyek az említett dokumentumban nem szerepelnek.)

a. Az **asztali operációs rendszerek** nem alkalmasak beágyazott rendszerekhez, mert:

- szolgáltatásai feleslegesen széleskörűek;
- nem modulárisak, nem hibatűrők, nem konfigurálhatóak, nem módosíthatóak;
- túl nagy tárigényűek;
- energiafogyasztásra nem optimalizáltak;
- nem küldetés-kritikus alkalmazásokra tervezték őket;
- az időzítési bizonytalanságok túl nagyok.

b. Szükség van **konfigurálhatóság**ra:

- egyetlen RTOS nem elégít ki minden igényt;
- a fel nem használt funkciók/adatok okozta overhead nem tolerálható;
- sok olyan beágyazott rendszer van, amelynek nincsen diszkje, billentyűzete, képernyője, egere.

A **konfigurálás** tipikus **eszközei**:

- a felesleges funkciók eltávolítása (például linker segítségével);
- feltételes fordítás alkalmazásával (#if és #ifdef parancsok);

Megjegyzés: A verifikáció nehézkes olyan rendszerekben, amelyek nagy számban tartalmaznak konfigurálással származtatott operációs rendszereket:

- minden konfigurálással származtatott operációs rendszert alaposan tesztelni kell;
- pl. az eCos (a Red Hat open source RT operációs rendszere) 100 és 200 közötti konfigurációs ponttal rendelkezik.

c. A beágyazott operációs rendszerek **eszközmeghajtóit a taskok kezelik**, nem pedig integrált meghajtók:

- a jósolhatóságot javítja, ha mindent az ütemező kezel;
- praktikusan nincs olyan eszköz, amelyet az operációs rendszer minden változata támogatna, legfeljebb a rendszer időzítő.

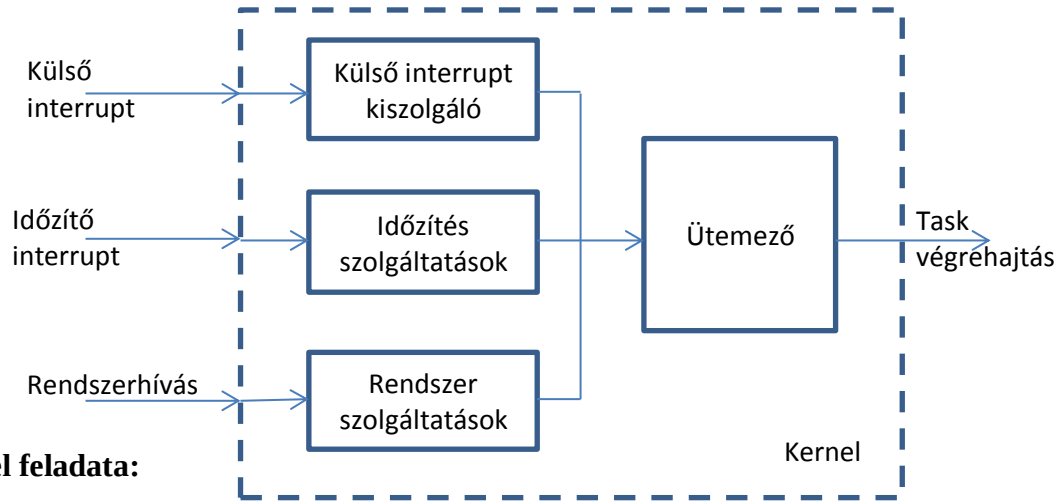
Beágyazott RTOS	Standard OS
alkalmazói szoftver	alkalmazói szoftver
middleware-ek	middleware-ek
eszközmeghajtók	operációs rendszer
real-time kernel	eszközmeghajtók

d. A beágyazott operációs rendszerekben **megszakítást bármely taszk használhat**:
- A standard OS-ben súlyos megbízhatatlansági forrás lenne;
- A beágyazott programokról feltételezzük, hogy teszteltek;
- Megengedhető, hogy megszakítás közvetlenül indítson vagy megállítson taszk-okat (azáltal, hogy a megszakítás táblázatba a taszkok kezdőcímeit írjuk). Ez hatékonyabb és jósolhatóbb, mint OS szolgáltatásokon keresztül.

Megjegyzés:
- Azonban a komponálhatóság sérül: ha egy taszk futását egy megszakításhoz kötjük, akkor nehéz lehet egy másik taszk hozzáadása, amelyet ugyanahhoz az eseményhez kötve kell elindítani.
- Ha a valós-idejű feldolgozás szempont, akkor a megszakítások kiszolgálási idejét figyelembe kell venni. Ebben az esetben a megszakításokat is az ütemezőnek kell kezelnie.

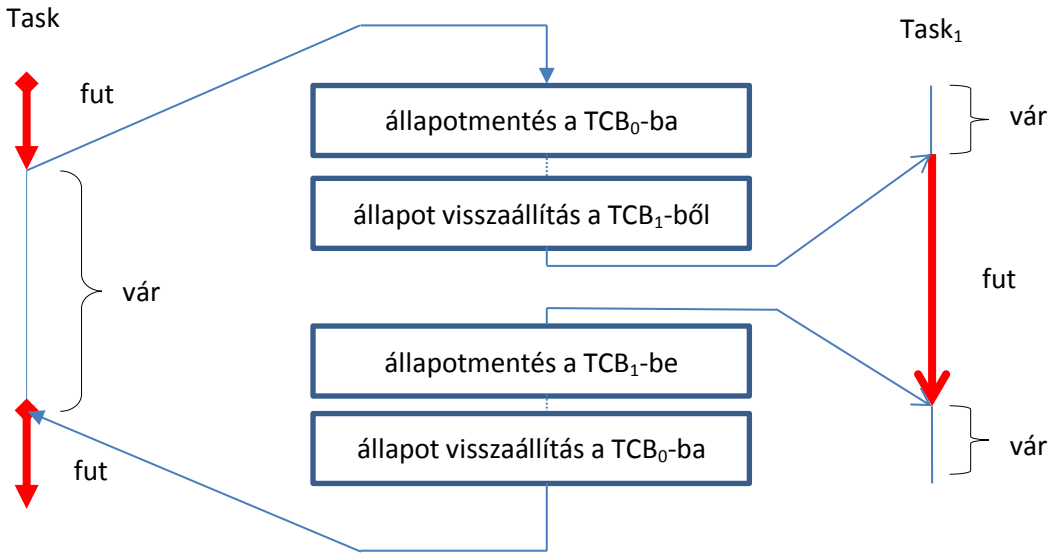
e. A beágyazott operációs rendszerekben **védelmi mechanizmusok nem szükségesek** minden esetben:
- A beágyazott rendszereket tipikusan egy adott célra tervezik, teszteletlen programot ritkán futtatnak, a szoftvert megbízhatónak tekintik.
- Nincs szükség privilégizált I/O utasításokra, a task-ok el tudják intézni a rájuk vonatkozó I/O műveleteket;
- Természetesen biztonsági szempontok védelmi mechanizmusokat szükségessé tehetnek.

f. A **valós idejű operációs rendszerek (RTOS)** valós idejű rendszerek létrehozását támogatják. Követelmény:
- Az időbeni viselkedés jósolható: minden operációs rendszer szolgáltatás esetében a végrehajtási idő maximuma ismert kell legyen. Az RTOS determinisztikus viselkedésű, majdnem minden tevékenységet az ütemező felügyel.
- Az RTOS intézi az időzítést és az ütemezést: ennek érdekében jó, ha ismeri a taszk-ok határidejeit, és nagy felbontású időzítő szolgáltatásokat kell biztosítson.
- Az RTOS legyen gyors (praktikus megfontolásból).
- Az RTOS folyamat-menedzsment szolgáltatásai:



A kernel feladata:

- A konkurens (kvázi-parallel) feladatok végrehajtása task-ok vagy szálak (threads) formájában:
- a task állapotok kézbentartásával és a task-ok sorba állításával,
- a task preempciók végrehajtásával (gyors context switching) és gyors IT kezeléssel,
- A CPU ütemezése (a határidők garantálása, a task várakozások minimalizálása, a számítási teljesítmény méltányos szétosztása);
- A task-ok szinkronizálása (kritikus szakaszok, szemaforok, monitorok, kölcsönös kizárás);
- A task-ok közötti kommunikáció (bufferelés);
- A valós-idejű óra belső referenciaként történő támogatása.



g. **Standard operációs rendszerek valós-idejű kiterjesztései**

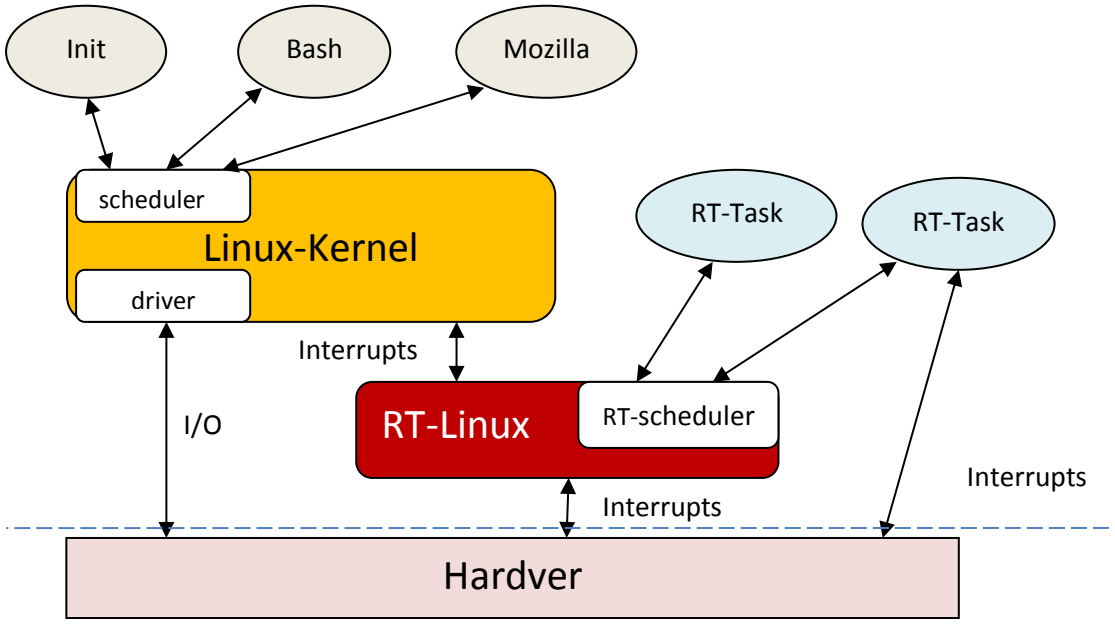
A real-time kernel futtat minden RT task-ot, a standard OS pedig egyetlen task-ként hajtódik végre:

RT-task 1	RT-task 2	non-RT task 1	non-RT task 2
eszközmeghajtó	eszközmeghajtó	Standard-OS	
real-time kernel			

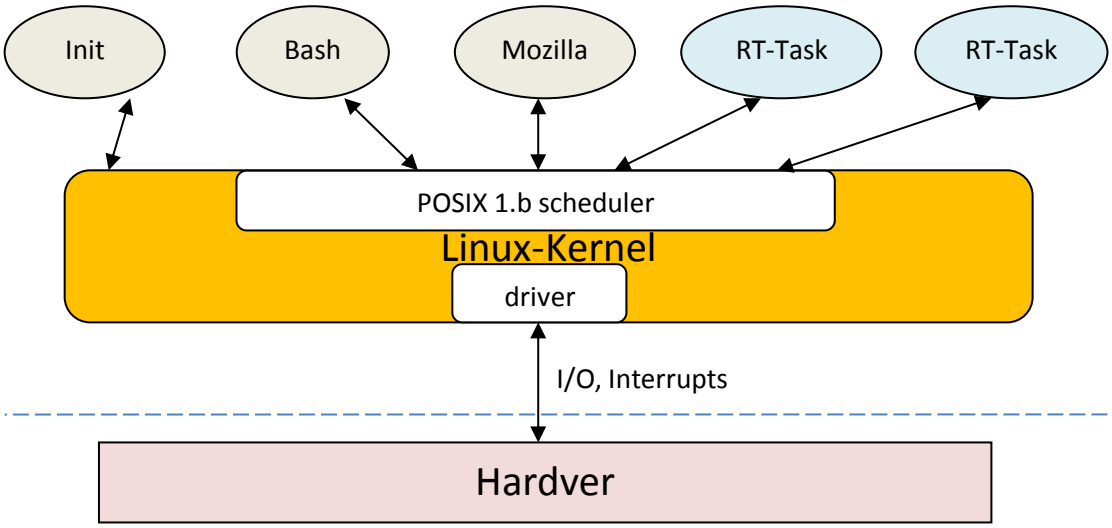
Megjegyzések:

- A standard OS összeomlása nem befolyásolja az RT-task-ok futását;
- Az RT-task nem tudja használni a standard OS szolgáltatásokat: a várakozásokat alulmúló elrendezés.

Példa: RT Linux



Példa: Posix 1.b RT-extensions to Linux



A szokványos Linux scheduler lecserélhető a POSIX scheduler-re, amely RT task-ok számára prioritást biztosít. A standard OS hívások mellett speciális RT hívások is vannak. A programozhatóság egyszerű, de nincs garancia a határidők teljesülésére. (POSIX: "**P**ortable **O**perating **S**ystem **I**nterface for uni**X**".)

7.3. Virtualizáció beágyazott rendszerekben

Vitualizáció: a szoftver hordozhatóságot szolgálja, azaz fusson különféle hardvereken. A virtuális gép (VM: Virtual Machine) olyan szoftver környezetet biztosít az adott szoftver számára, mintha tényleges hardveren futna az alábbi struktúrában:

Alkalmazás
Operációs rendszer
Hypervisor
Processzor

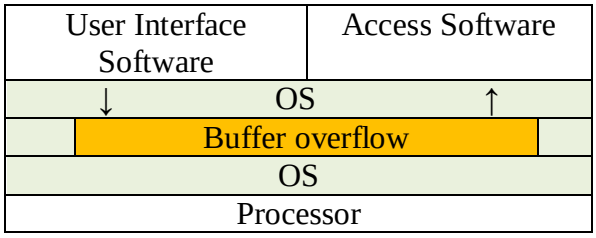
Az a szoftver réteg, amelyik a virtuális környezetet biztosítja az ún. virtuális gép monitor (VMM) vagy hypervisor. Három fő funkciója van:
- az eredeti géppel azonos szoftver környezetet biztosít;
- legfeljebb lassabb a futása;
- teljes mértékben felügyeli a rendszer erőforrásait.

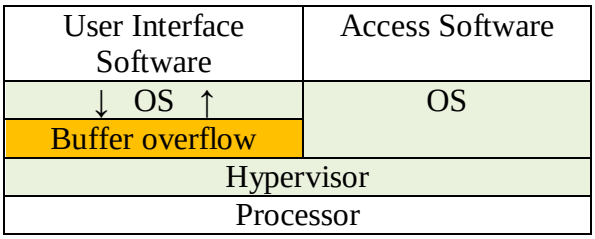
A VM utasítások többsége közvetlenül végrehajtható a hardveren, egy részük interpreter-rel valósul meg. Ezek között vannak:
- a vezérlés-érzékeny utasítások, amelyek módosítják a privilegizált gép-állapotokat, ezért interferálnak a hypervisor erőforrások feletti felügyeletével.
- a viselkedés-érzékeny utasítások, amelyek hozzáférnek (olvassák) a privilegizált gép-állapotokat.

Konkurens operációs rendszerek virtuális gépen

User Interface Software	Access Software
Standard OS	RTOS
Hypervisor	
Processor	

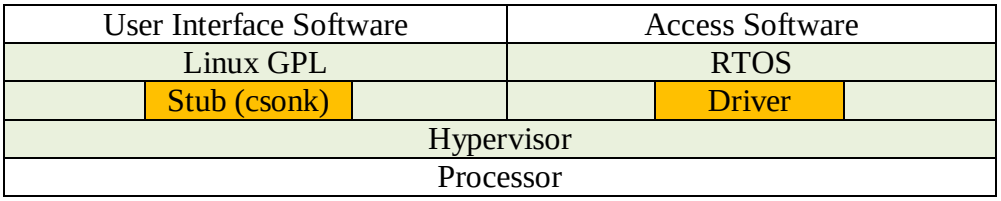
A biztonság növelése virtualizáció alkalmazásával





Az egyik alkalmazás okozta hiba nem terjed át a másik alkalmazásra, mert annak saját operációs rendszere van.

Licensz elválasztás virtualizáció alkalmazásával



GPL: General Public Licence. A Linux ezen licensz szabályai szerint férhető hozzá szabadon. Ezért minden hozzá készített szoftver is szabad hozzáférésű. Ahol ez probléma, ott a Linux és célalkalmazás külön virtuális gépeken fut, csak egy ún. csonkot (vagy proxy-t) készítenek, amely hyperhivásokon keresztül éri el a nem szabad hozzáférésű drivert, és azon keresztül a kapcsolódó hardvert.

A virtualizáció korlátai beágyazott rendszerekben:

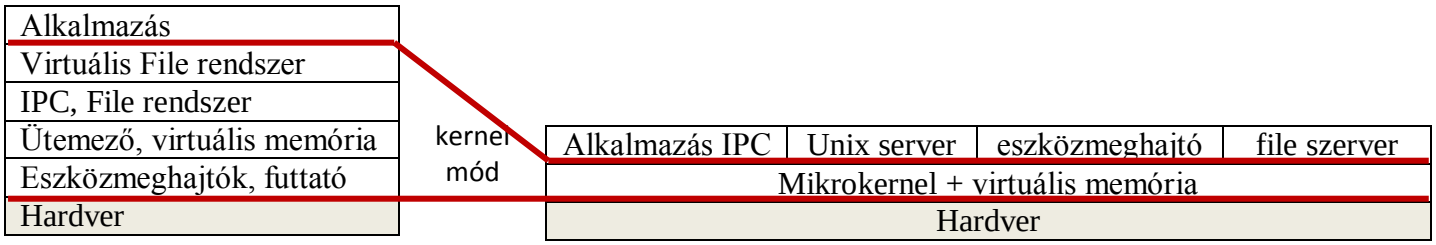
- Az alkalmazások egyre növekvő komplexitása mellett a több operációs-rendszer futtatás nagyon nagyméretű kódot eredményez, ami önmagában hibaforrás lehet, nagy memóriát igényel, többet fogyaszt.
- Az egyes alrendszerek szoros együttműködése szükséges, ehhez nem passzol a szeparáltság.
- Az egyes alrendszerek hatékony kommunikációja igény, amit a virtuális gép modell nem támogat.
- Az egyes alrendszerek közös erőforrásokon osztoznak, amit nehézkes megszervezni, ha több operációs rendszer fut párhuzamosan.
- A virtualizáció következménye, hogy az ütemezés két szinten történik: (1) Hypervisor és a VM között, (2) minden VM-en futó operációs rendszeren belül.
- A kritikus biztonsági követelmények teljesülését a virtuálizáció egymaga nem támogatja. A kritikus kódrészeket (ún. trusted computing base, TCB) privilegizált módban kell futtatni a processzoron. A hypervisor is része a TCB-nek. Az ilyen kódnak bizonyítottan helyesnek kell lennie. A virtualizáció növeli az ilyen kód méretét.

Milyen támogató szoftverre van szükségük a beágyazott rendszereknek?

- támogassa a virtualizáció minden előnyét;
- támogassa az erősen kölcsönhatásban lévő, közepes komplexitású komponensek erős egymásba-ágyazását annak érdekében, hogy a hibás állapotból helyreállni képes, robusztus rendszereket hozzunk létre;
- támogassa a nagy sávszélességű, kis késleltetésű kommunikációt, amely konfigurálható, rendszerszintű biztonsági politikával párosul;
- globális ütemezési stratégia érvényesül a különféle alrendszerek task-jaira;
- lehessen úgy alrendszereket létrehozni, hogy nagyon kicsi a TCB-jük.

A mikrokernel (microvisor) technológia: a beágyazott rendszerekhez jobban illeszkedő virtualizációs technológia

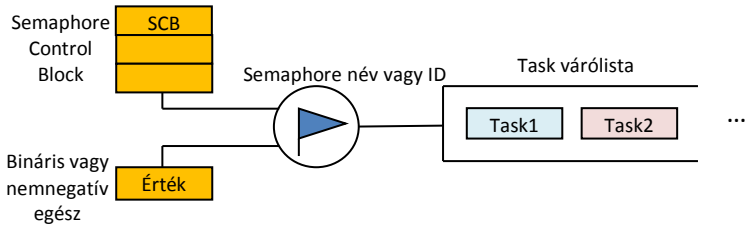
A mikrokernel (microvisor) egy minimális privilegizált szoftver réteg, amely csak általános mechanizmusokat biztosít. Az aktuális rendszerszolgáltatások és stratégiák a felhasználói módban futó komponenseken valósulnak meg. A mikrokernel elve: Egy mikrokernelen belül csak olyan koncepciónak van létjogosultsága, amelyet ha kiviszünk a kernelből, és ezáltal versengő implementációkat engedünk meg, az a megkívánt rendszer funkcionalitás megvalósulását megakadályozná. A mikrokernel nem nyújt semmilyen szolgáltatást, csak mechanizmusokat biztosít szolgáltatások implementálásához. A hagyományos (monolitikus) operációs rendszer és a mikrokernel alapú rendszerek struktúrájának eltérését az alábbi ábra mutatja:



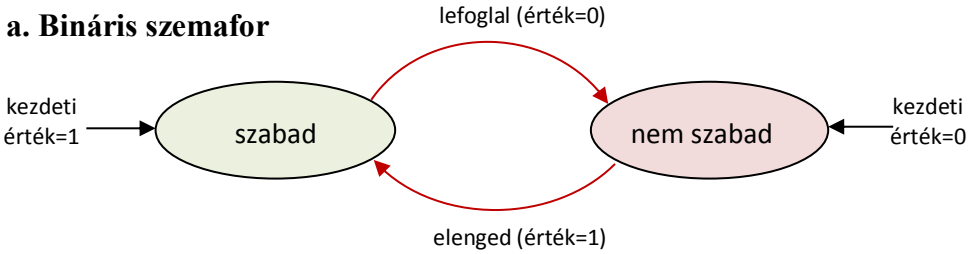
A hagyományos struktúra vertikális jellegű, a mikrokernel pedig horizontális. Az utóbbinál nincs érdemi különbség az alkalmazás és a rendszerszolgáltatás között: ezek mind felhasználói módban futnak. Minden ilyen task beágyazódik a kernel által létrehozott hardver memória mezőjébe. Ezen kívül más részeket csak kernel mechanizmusok révén befolyásolhat, tipikusan üzenetek küldésével. Ezek az ún. message-passing mechanizmusok (Inter Process Communication, IPC).
Részletesebb leírások a mikrokernelekről: pl.
https://gdmissionsystems.com/cyber/products/trusted-computing-cross-domain/microvisor-products/

7.4. Szinkronizáció RTOS szolgáltatások segítségével (Korábbi tanulmányok ismétlése)

Szemaforok



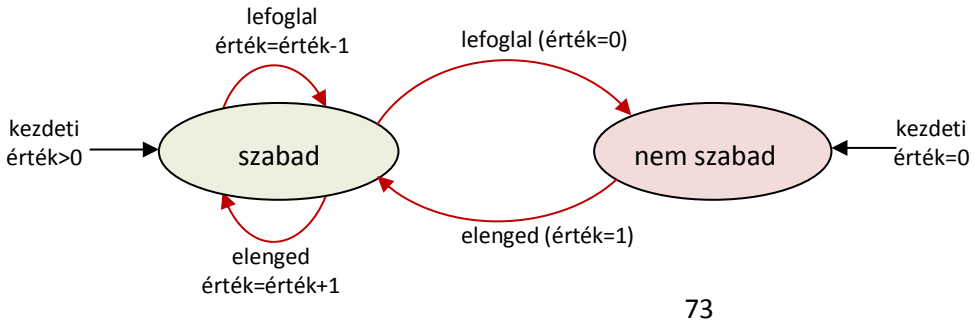
a. Bináris szemafor



szabad: available
nem szabad: unavailable
lefoglal: acquire
elenged: release

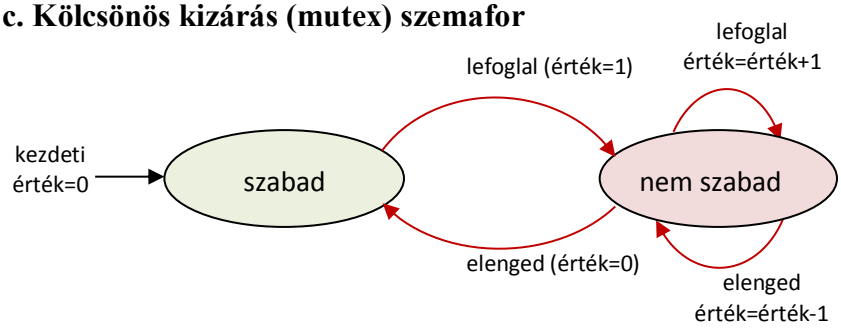
Globális erőforrás: akármelyik task felszabadíthatja, akkor is, ha nem foglalta előzetesen.
A létrehozáskor lehet *szabad* vagy *nem szabad*.

b. Számláló szemafor



Globális erőforrás: akármelyik task állíthatja, akkor is, ha nem foglalta előzetesen.
Lehet korlátozott számú érték (token) → tipikusan a kezdeti érték.
Lehet (gyakorlatilag) korlátlan számú token → előjel nélküli egész vagy előjel nélküli hosszú egész.

c. Kölcsönös kizárás (mutex) szemafor



szabad: unlocked
nem szabad: locked

Mutex tulajdonlás (ownership): Egy task tulajdonhoz jut a foglalás révén, és elveszíti azt elengedéskor. Másik task nem engedheti el, mint a bináris szemafor esetében.

Szemafor műveletek

Create	binary, counting, mutex
Delete	ID megadása → a várólista felszabadítása. Csak szabad szemafor törölhető.
Acquire	alternatívák: take, sm_p, pend, lock: foglal
Release	alternatívák: give, sm_v, post, unlock: elenged

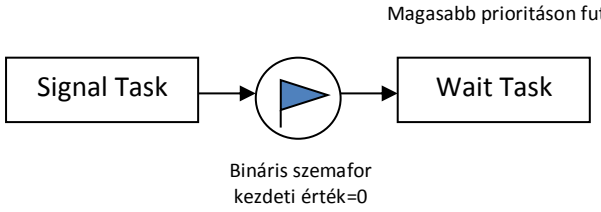
A foglalás következményei:

- wait forever → a task blokkolva addig, amíg nem szabadítjuk fel
- wait with a timeout → a task blokkolva addig, amíg nem szabadítjuk fel vagy lejár a time-out
- do not wait → megkéri a szemafor token-t, de ha nem szabad, akkor nem blokkolódik

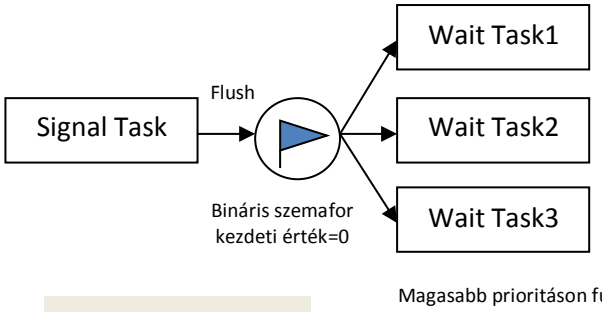
Flush	felszabadít minden task-ot, amely egy szemaforra vár.
Show info	a szemafor általános információit mutatja
Show blocked tasks	azoknak a task-oknak az ID-jét adja, amelyek várnak egy szemaforra.

Tipikus szemaforhasználat

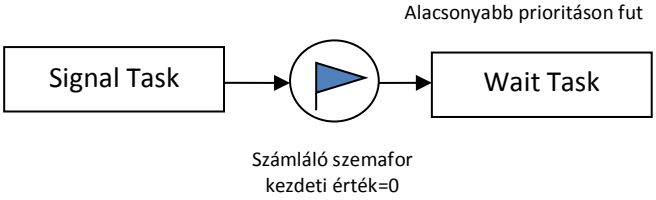
Wait-and-Signal szinkronizáció:
Vezérlés átadás: a Wait Task fut először → elakad → a Signal Task elfut a felszabadításig.



Multiple-Task Wait-and-Signal szinkronizáció:

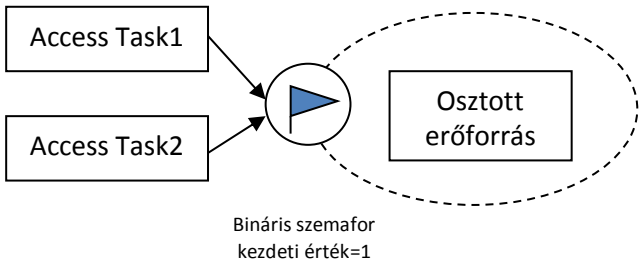


Credit-Tracking szinkronizáció:



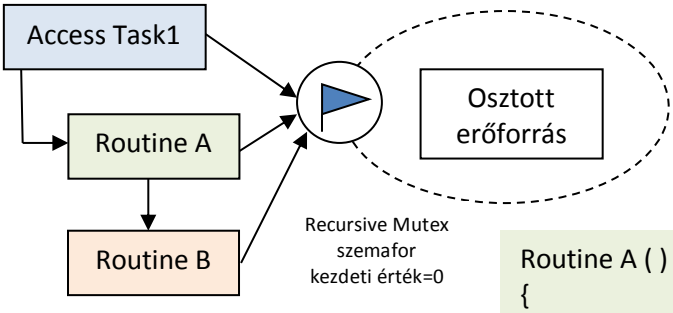
Tipikus példa:
burst-ös jel, IT-vel

Single Shared-Resource-Access szinkronizáció:



Probléma: tévedésből bárki felszabadíthatja a bináris szemafort, ezért mutex szemafor javasolható.

Recursive Shared-Resource-Access szinkronizáció:



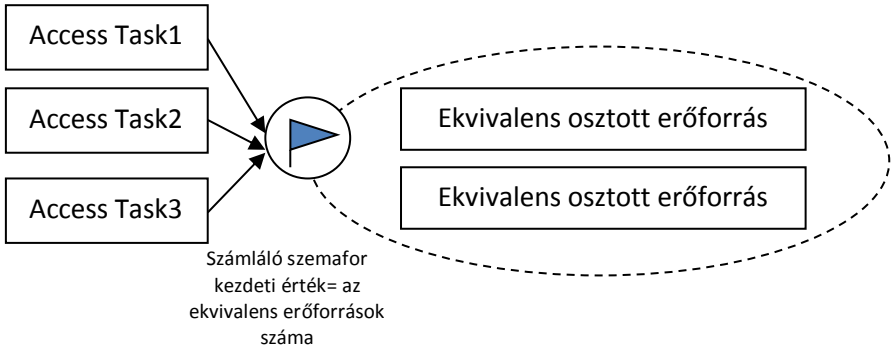
Rekurzív hozzáférés igény esetére kidolgozott, ún. Recursive Mutex segítségével történik. Fontos vonatkozás, hogy a Mutex szemafor az Access Task tulajdonába kerül.

```
Access Task ( )
{
        Acquire Mutex
        Access shared resource
        Call Routine A
        Release Mutex
}
```

```
Routine A ( )
{
        Acquire Mutex
        Access shared resource
        Call Routine B
        Release Mutex
}
```

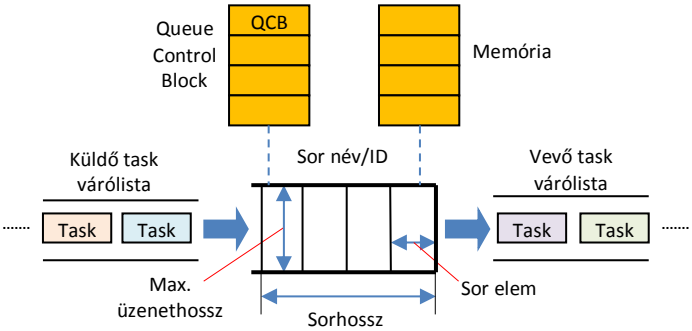
```
Routine B ( )
{
        Acquire Mutex
        Access shared resource
        Release Mutex
}
```

Multiple Shared-Resource-Access szinkronizáció:

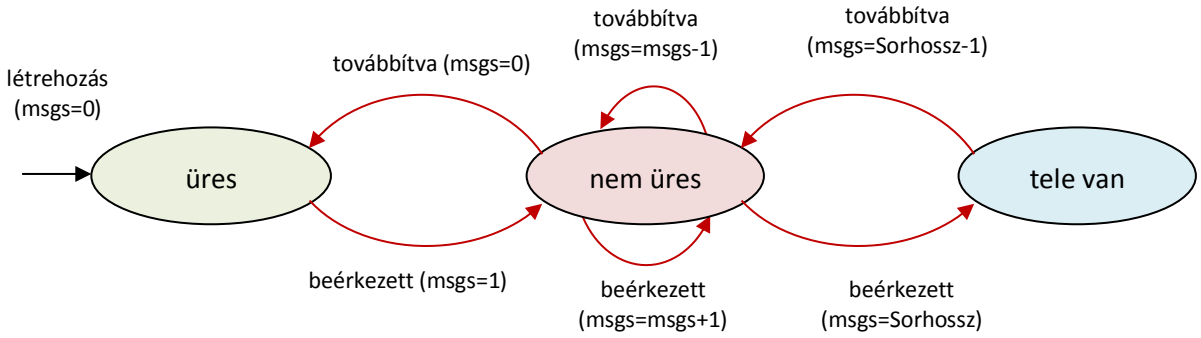


Megjegyzés: Analógia: sorbaállás több pénztár esetén.

Üzenetsorok (Message Queues): a task-ok üzenetváltását segítő mechanizmus



A taskok elküldik üzeneteiket az üzenetsorba. Ha az üzenetsor megtelt, akkor várólista képződik a fogadó oldalon. Ha az üzenetsor üres, akkor az üzenetre várók várólistát képeznek a vételi oldalon. A működés véges állapotú automatával (Finite State Machine: **FSM**) írható le.



Megjegyzés: Az implementációtól függően az üzenet akár három példányban is megjelenhet: (1) a küldő task memóriaterületén, (2) az üzenetsor memóriaterületén, (3) a vevő task memóriaterületén. Ha ez az üzenet mérete miatt gondot okoz, akkor alkalmazható olyan implementáció, hogy csak az üzenet pointerét és az üzenethosszát továbbítjuk, maga az üzenet egyetlen példányban van a memóriában.

Üzenetsor műveletek

Create	Létrehozás
Delete	Megszüntetés
Send	Küldés egy üzenetsornak
Receive	Vétel egy üzenetsorból
Broadcast	Küldés mindenhová

Az elküldött üzenetek továbbítása First-In, First-Out (FIFO) logika alapján történhet, de sürgős üzenetek esetén Last-In, First-Out (LIFO) stratégia is felvethető.

A küldés lehetséges következményei:

- block forever → a task blokkolva addig, amíg az üzenetsor tele van
- block with a timeout → a task blokkolva addig, amíg az üzenetsor tele van vagy lejár a time-out
- not block → a task akkor sem blokkolódik, ha az üzenetsor tele van (egyes implementációk interrupt-tal történő küldést is megengednek: az interrupt nem blokkolódik, legfeljebb hibaüzenetet küld.)

Az üzenetsor információs műveletei:

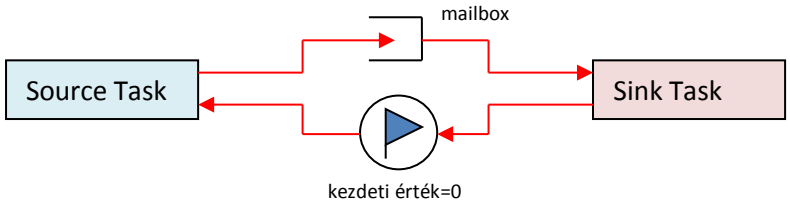
Show queue info
Show queue's task-waiting list

Tipikus üzenetsor használatok

Non-interlocked, One-Way Data Communication

Laza csatolási forma. Ha a küldés interrupt-tal történik, akkor többnyire ezt használják.

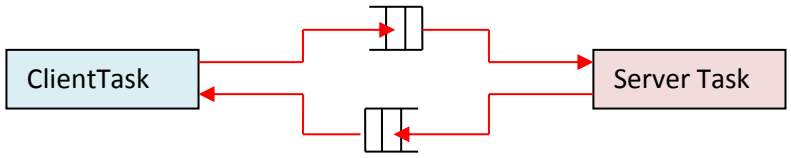
Interlocked, One-Way Data Communication



Az üzenet elküldése után a küldő a bináris szemaforra vár. A szemafor akkor engedi tovább, ha a vevő engedi.

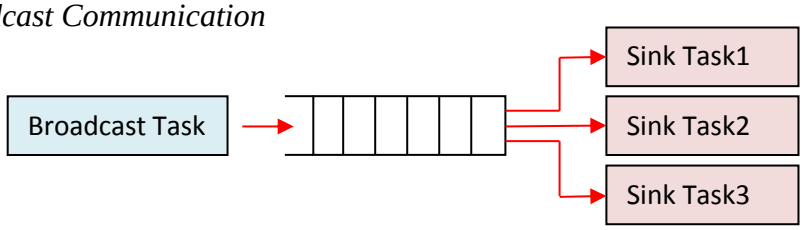
A mailbox egy üzenet fogadására képes üzenetsor.

Interlocked, Two-Way Data Communication



Általában a Server Task prioritása a magasabb, hogy a Client Task kérései hamar kiszolgálásra kerüljenek.

Broadcast Communication



További Kernel objektumok

Pipe: Lényegében egy FIFO memória, amely strukturálatlan adatok továbbítását teszi lehetővé. Nem különálló üzenetekből áll, hanem egy bitsorozat. Működése az üzenetsorhoz hasonlóan írható le.

Esemény regiszter: A *Task Control Block* (TCB) része, bitjei adott események bekövetkezésének jelzésére szolgálnak.

Signal vagy **Event:** Szoftver interrupt. A futó task futása megszakad, és a Signal-hoz/Event-hez rendelt program fut le.

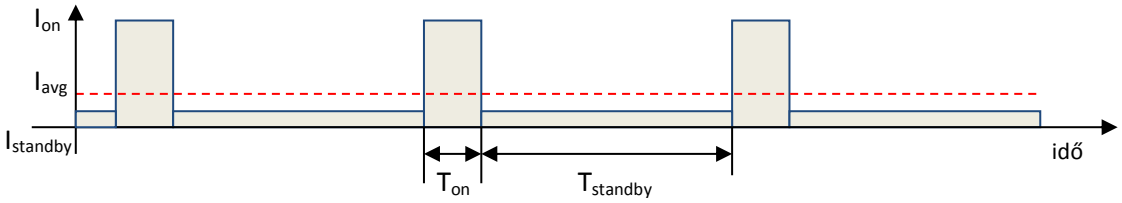
Conditional Variables: Megosztott erőforrásokhoz rendelt, ezek révén tudható meg, hogy az adott erőforrás milyen állapotban van.

8. Esettanulmányok (folyt.): 8.3. Energiaviszonyok – Tervezési szempontok

Példa: két ceruzaelem „képessége”, szenzorhálózati alkalmazásban (mikrovezérlő+rádió+szenzorok):

2 db AA elem (3000 mAh), üzemidő: min 1 év (8760 óra), P_{on}=150 mW (I_{on}=50mA), $I_{standby}$=50µA.

áramfelvételtel

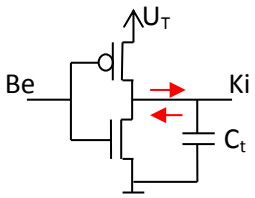


$$I_{avg} = \frac{I_{on}T_{on}}{T_{on}+T_{standby}} + \frac{I_{standby}T_{standby}}{T_{on}+T_{standby}} = I_{on}\lambda + I_{standby}(1-\lambda), \quad I_{avg\,max} = \frac{3000mAh}{8760h} = 342\mu A$$

$$\lambda = \frac{I_{avg\,max}-I_{standby}}{I_{on}-I_{standby}} = 0.0058 \approx 0.6\% \qquad \approx 8\frac{perc}{nap}.$$

Példa: Ha óránként végzünk mérést, akkor minden órában 20 másodperc üzemidő lehetséges.

CMOS áramkörök teljesítményviszonyai és energiaigénye:



A föld és a tápfeszültség között elhelyezkedő két tranzisztor felváltva zárt, és nyitott kapcsolóként üzemel. A szivárgást és az átkapcsoláskor rövid ideig tartó rövidzárási áramot leszámítva csak a kapacitív terhelést töltő/kisütő áram jelent terhelést a tápforrás számára. A szivárgást elhanyagolva a fogyasztott teljesítmény:

$$P \sim \alpha C_t U_T^2 f,$$

ahol U_T a tápfeszültség, α az ún. kapcsolási aktivitás, C_t a terhelő kapacitás, f pedig az órajel frekvencia. Az áramkör késleltetése:

$$\tau \sim C_t \frac{U_T}{(U_T - U_k)^2}, \qquad \text{ahol } U_k \text{ a küszöbfeszültség, } U_k \ll U_T.$$

Megállapítható, hogy:

- a tápfeszültség csökkentése a teljesítményt négyzetesen csökkenti;
- a késleltetés a tápfeszültség reciprokával nő,
- a maximális órajel-frekvencia csökkentés csak lineárisan csökkenti a teljesítményt.

Az energiafogyasztás optimalizálásának lehetőségei (Dynamic Voltage Scaling: DVS):

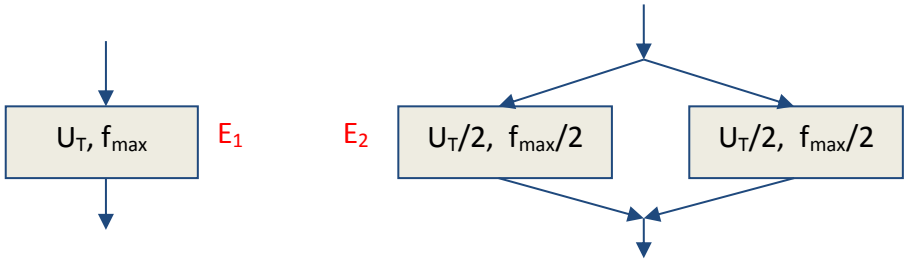
$$P \sim \alpha C_t U_T^2 f\ , \quad E \sim \alpha C_t U_T^2 f t = \alpha C_t U_T^2 (ciklusok\ száma).$$

Egy adott task energiaigényének csökkentése érdekében:

- csökkentsük a tápfeszültséget;
- csökkentsük a kapcsolási aktivitást;
- csökkentsük a terhelő kapacitást;
- csökkentsük a ciklusok számát.

A szivárgás csökkentésének leghatékonyabb módja a tápfeszültség kikapcsolása azoknál az áramköröknél, amelyeket éppen nem használunk (Power Supply Gating).

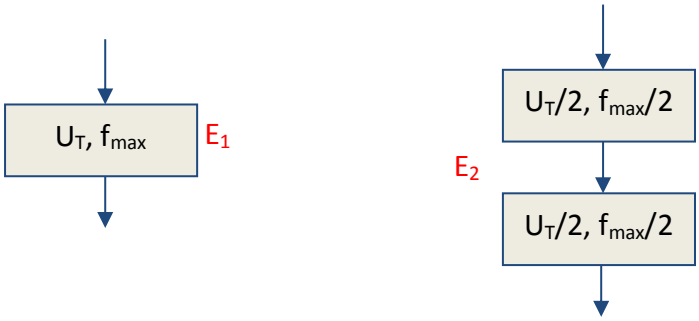
Párhuzamos kialakítás: Kétszerezett hardver felére csökkentett tápfeszültséggel és órajel frekvenciával.



$$E_1 \sim U_T^2 (ciklusok\ száma), \qquad E_2 = E_1/4.$$

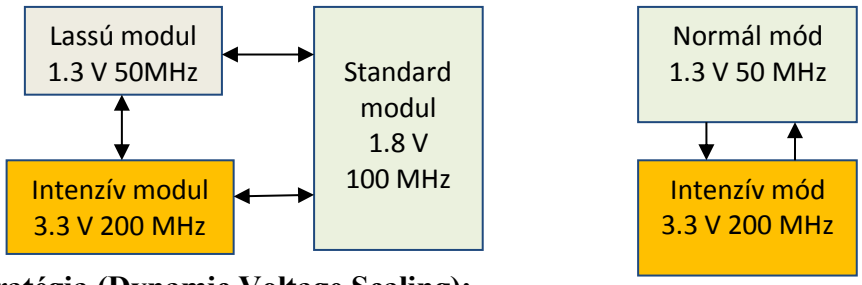
A műveletek száma és a késleltetés változatlan, az energiaigény negyedére csökkent.

Csővezeték kialakítás (Pipelining): Kétszerezett hardver felére csökkentett tápfeszültséggel és órajel frekvenciával.

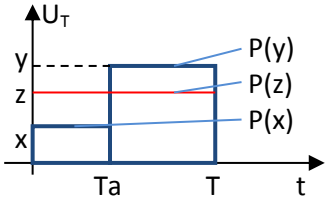


A műveletek száma változatlan, az energiaigény a negyedére csökkent.

Nem minden komponens azonos sebesség igényű és az igény időben változhat:



Optimális stratégia (Dynamic Voltage Scaling):



A eset: *Ta* ideig *x* feszültséggel, (1-*a*)*T* ideig *y* feszültséggel működtetjük az áramkört.
Az energiafogyasztás: $\boxed{T(aP(x)+(1-a)P(y)).}$
B eset: végrehajtás $z = ax + (1-a)y$ feszültséggel *T* ideig.
Az energiafogyasztás: $\boxed{TP(z).}$

Mivel a teljesítmény a tápfeszültség négyzetes függvénye, ezért, $P(z) < aP(x) + (1-a)P(y)$, vagyis célszerű konstans feszültséggel működtetni az áramkört. (A lineáris kombináció a parabola húrját adja meg, ami a parabola felett helyezkedi el.)

Példa:

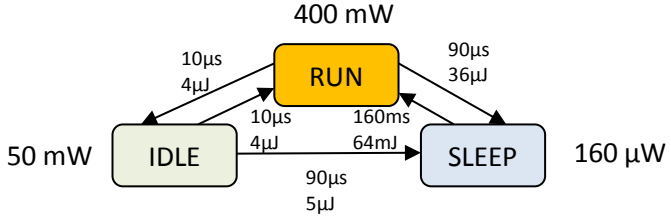
U_T[V]	5.0	4.0	2.5
Energia ciklusonként [nJ]	40	25	10
f_{max} [MHz]	50	40	25
ciklusidő [ns]	20	25	40

Egy task végrehajtása 10^9 ciklus végrehajtását igényli. Ehhez 25 másodperc áll rendelkezésre.

a. Leggyorsabb végrehajtás: 10^9 ciklus @ 50 MHz.
Ennek energiaigénye: $E_a = 10^9 * 40 * 10^{-9} = \mathbf{40}$ [J], idő igénye: $10^9 * 20 * 10^{-9} = \mathbf{20}s$.
b. Végrehajtás két feszültségen: 0.75*10^9 ciklus @ 50 MHz + 0.25*10^9 ciklus @ 25 MHz.
Ennek energiaigénye: $E_a = 0.75 * 10^9 * 40 * 10^{-9} + 0.25 * 10^9 * 10 * 10^{-9} = \mathbf{32.5}$ [J],
idő igénye: $0.75 * 10^9 * 20 * 10^{-9} + 0.25 * 10^9 * 40 * 10^{-9} = \mathbf{25}s$.
c. Végrehajtás optimális feszültségen: 10^9 ciklus @ 40 MHz.
Ennek energiaigénye: $E_a = 10^9 * 25 * 10^{-9} = \mathbf{25}$ [J], idő igénye: $10^9 * 25 * 10^{-9} = \mathbf{25}s$.

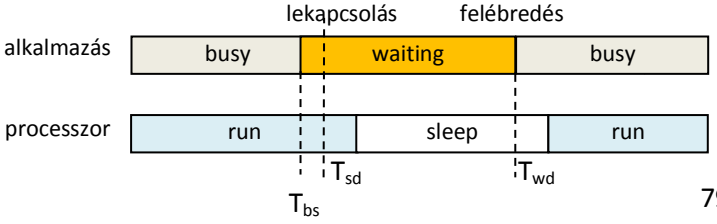
Megjegyzés: Értelemszerűen kell legyen idő-tartalék a taszk végrehajtásánál.

Dinamikus teljesítmény menedzsment (Dynamic Power Management (DPM)):



Hardver támogatást igényel.
Példa: **StrongARM SA1100**
IDLE: egy szoftver rutin megállíthatja a processzort, ha nincs használatban, de IT-t fogad.
SLEEP: minden aktivitást a chipen leállít.

Példa: a teljesítménymenedzsment időviszonyai:



T_{bs}: time before shutdown
T_{sd}: shutdown delay
T_{wd}: wakeup delay

Megjegyzés: A lekapcsolás csak hosszú használaton kívüli idők felmerülése esetén indokolt.

Példa: Dinamikus teljesítmény menedzsment a működési frekvencia állításával:

Tételezzük fel, hogy egy CMOS processzor $P(f)$ teljesítményfelvétele f frekvencián:

$$P(f) = \left[10\left(\frac{f}{100MHz}\right)^3 + 20\right] mWatt$$

A teljesítményfelvétel csökkentésére a végrehajtási frekvencia csökkentése használható. A maximum, ill. a minimum frekvencia értékek:

$$f_{max} = 1000MHz / f_{min} = 50MHz$$

A frekvencia-változtatás elhanyagolható ráfordítás igényű, a processzor $50MHz$ és $1000MHz$ között tetszőleges frekvencián működni tud. A teljesítményfelvétel csökkentése érdekében a processzor *sleep* (alvó/kikapcsolt) állapotba is vezérelhető. Alvó állapotban a processzor fogyasztása elhanyagolható. A processzor alvó állapotból futó állapotba vezérlése azonban energiát igényel, aminek értéke: $3 \times 10^{-5} Joule$. (Futó állapotból alvó állapotba vezérlés energiaigénye elhanyagolható.) A ki/bekapcsolás időigénye ugyancsak elhanyagolható, pillanatszerűnek tekinthető.

A processzor három task-ot hajt végre:

	érkezési idő	határidő	ciklusok száma
τ_1	0	2ms	100000
τ_2	2ms	6ms	100000
τ_3	6ms	7ms	80000

A feladat szerint a processzor futó állapotban kell legyen a nulla és a 7ms időpillanatokban.

1. feladat:

C ciklus végrehajtásának energiaigénye $\frac{CP(f)}{f}$, hiszen $C = tf$, és $E = Pt$. Ha túl alacsony a frekvencia, akkor a hosszú idejű végrehajtás, ha túl nagy a frekvencia, akkor pedig a növekvő teljesítményfelvétel növeli az energiaigényt. Létezik $50MHz$ és $1000MHz$ között egy olyan f_{krit} kritikus frekvencia, amely mellett tetszőleges számú *C* ciklus energia felvétele minimális. Mekkora ez a frekvencia az adott processzor esetében?

Megoldás:

Keressük $\frac{P(f)}{f}$ minimumát. A fenti képlethez igazodva vezessük be: $f_n = \frac{f}{100MHz}$ normalizált frekvenciát! $\frac{P(f_n)}{f_n}$ első deriváltja $20f_n - \frac{20}{f_n^2}$, ami akkor nulla, ha $f_n = 1$. Ezzel $f_{krit} = 100MHz$.

2. feladat:

A processzor *idle* (tétlen) állapotban f_{min} frekvenciával jár, és ekkor t másodperc alatt $P(f_{min}) \times t$. Megtérülési időnek (*break-even time*) nevezzük annak az intervallumnak a hosszát, amelynek elteltével érdemes a processzort tétlen állapotból alvó állapotba kapcsolni. Mekkora a processzor megtérülési ideje?

Megoldás:

Akkor érdemes alvó állapotba kapcsolni, ha az elérhető energia megtakarítás már fedezi a visszakapcsoláshoz szükséges $3 \times 10^{-5} + 0\ Joule$ többlet ráfordítást:

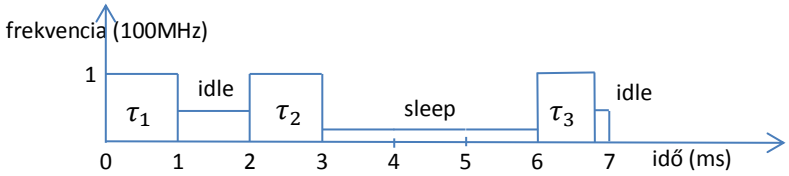
$$\text{Energia(tétlen, } f_{min}) - \text{Energia(alvó)} \geq \text{Energia(alvóból futó állapotba)}$$

$$P(f_{min}) \times t_{megtérülési} - 0 \geq 3 \times 10^{-5} Joule$$

$$t_{megtérülési} \geq \frac{3 \times 10^{-5} Joule}{10^{-3} \times (10 \times 0.5^3 + 20) Watt} = 1.412ms.$$

3. feladat:
Az ún. munkaterhelést megőrző ütemezést (*workload-conserving schedule*) úgy definiáljuk, mint egy olyan ütemezés, amely mindig végrehajt valamely task-ot hacsak a futásra kész task-ok listája nem üres. Készítsünk ilyen ütemezést, amely minimalizálja az energiafogyasztást és egyidejűleg betartja a három task-ra vonatkozó ütemezési előírásokat. Ehhez a task végrehajtások során használjuk az f_{krit} frekvenciát. Mekkora az energiafelhasználás ilyenkor?

Megoldás:



Mivel $P(f_{krit}) = 30mWatt$, és $P(f_{min}) = 21.25mWatt$, így az ábra szerinti ütemezéshez tartozó energia-fogyasztás:

$$(30 \times 1 + 21.25 \times 1 + 30 \times 1 + 3 \times 10^1 + 30 \times 0.8 + 21.25 \times 0.2)\mu Joule = 0.1395\, mJoule.$$

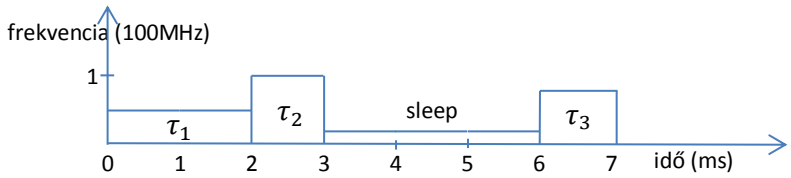
Vegyük észre, hogy az [1,2] intervallum rövidebb, mint a megtérülési idő, ezért nem merül fel az alvó állapotba kapcsolás.

4. feladat:

Lehet-e olyan munkaterhelést megőrző ütemezést készíteni, ami ennél is kisebb energiafogyasztással jár, miközben betartjuk a három task-ra vonatkozó ütemezési előírásokat?

Megoldás:

Igen, mert kihasználható a teljesítményfelvétel konvex jellege: lassítható τ_1 és τ_3 végrehajtása oly mértékben, hogy ne kelljen *idle* állapotba kapcsolni. Ezzel ugyanis annak ellenére, hogy a kritikus frekvenciánál alacsonyabbat alkalmazunk, mégis kevesebb energiát fogyasztunk. Az ábra szerinti ütemezéshez tartozó energiafogyasztás:



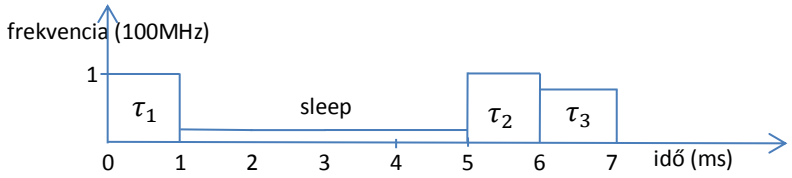
$$(21.25 \times 2 + 30 \times 1 + 3 \times 10^1 + (10 \times 0.8^3 + 20) \times 1)\mu Joule = 0.12762 mJoule.$$

5. feladat:

Lehet-e olyan ütemezést készíteni, akár a munkaterhelést megőrző stratégia feladásával, amely még ennél is kisebb energiafogyasztással jár?

Megoldás:

Igen. A megoldás lényege, hogy egy blokkba gyűjtjük azokat az időszakokat, ahol a processzor nem fut, mert ezáltal érdemes lesz azonnal alvó állapotba küldeni, mihelyst az lehetséges. Ezen kívül a τ_1 task végrehajtásához a *kritikus frekvenciát* használjuk. Ez a példa azt illusztrálja, hogy a munkaterhelést megőrző stratégiák nem feltétlenül a legjobbak. Az ábra szerinti ütemezéshez tartozó energiafogyasztás:



$$(30 \times 1 + 3 \times 10^1 + 30 \times 1 + (10 \times 0.8^3 + 20) \times 1)\mu Joule = 0.115120 mJoule.$$

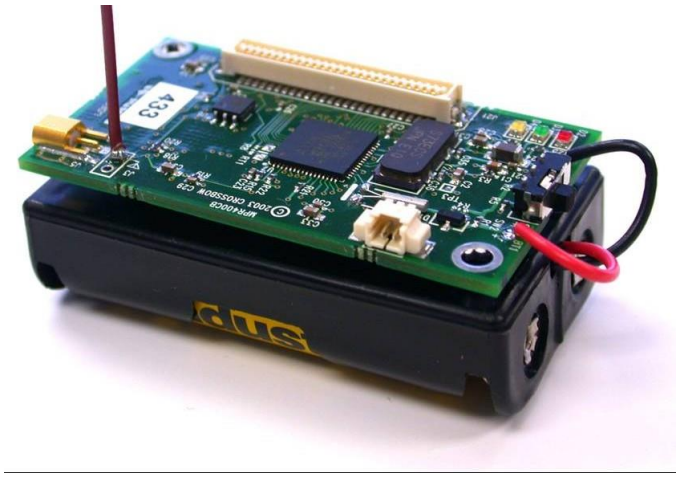
8. Esettanulmányok (folyt.)

Ezen címszó keretében olyan témakörök rövid bemutatására kerül sor, amelyek a beágyazott rendszerek megvalósításával kapcsolatos részletproblémákat és megoldási alternatívákat tárnak fel.

8.4. Szenzorhálózatok

(A szenzorhálózatokról részletes fólia-sorozat található a tantárgy tanszéki honlapján. Az alábbiak csak néhány kiemelt jellemzőt foglalnak össze.)

A szenzorhálózati csomópontok jellegzetes megjelenési formája a Berkeley Mica2 mote, amely az alábbi fényképen látható. Mérete a nyomtatott áramköri lap alatt elhelyezett 2*AA elem alapján becsülhető. Felépítése és hardver jellemzői a Szenzorhálózatok I. című dokumentumban leírtak alapján ismerhető meg. Ugyanitt olvasható az eszköz szóba jövő alkalmazásainak listája. Az alkalmazások jellemzője a térbeli kiterjedés, és a szükséges csomópontok nagy száma. A működés során lényeges az energiatakarékosság.



A TinyOS operációs rendszer
(A TinyOS operációs rendszerről részletes fólia-sorozat található a tantárgy tanszéki honlapján. Az alábbiak csak néhány kiemelt jellemzőt foglalnak össze.)

Miért van rá szükség?
A tradicionális operációs rendszerekkel nehézségek vannak szenzorhálózatok esetében, mert a többszálas architektúra nemigen használható kellő hatékonysággal, nagy a memóriaigény, az energiafelhasználás minimalizálását nem támogatják.
A vezeték nélküli szenzorhálózatok esetében lényeges (1) a konkurens végrehajtás, (2) az energiafelhasználás hatékonysága, (3) kis memóriaigény (small memory footprint), és (4) a sokrétű felhasználás támogatottsága.

Főbb jellegzetességei:

A TinyOS nyílt hozzáférésű operációs rendszer, amelykifejezetten vezeték nélküli szenzorhálózati alkalmazásokhoz készült. Komponens alapú, NesC (Networked embedded system C) nyelven íródott a University of California, Berkeley és az Intel Research együttműködésében.

A komponens alapú architektúra lehetővé teszi a gyakori változtatásokat, és eközben a kódméret minimális szinten tartható. A végrehajtás eseményvezérelt, és ebből adódóan nagymértékben konkurens. Energia hatékony, mert a processzor - amint lehetséges – *sleep* állapotba kerül. Kicsi a "lábnyoma", mert FIFO alapú, nem megszakítható ütemezést alkalmaz.

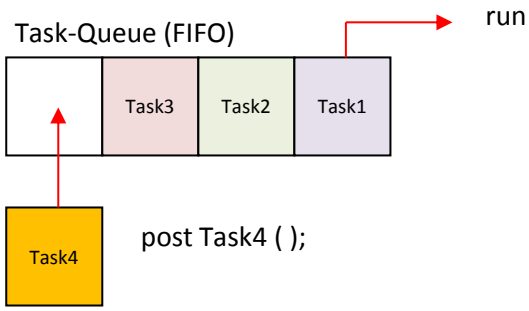
Statikus memória allokációt használ, a memória követelmények fordítási időben dőlnek el. A lokális változók mentése a stack-re történik.

Energia hatékony, kétszintű ütemezést használ: (1) Hossza futó task-ok és események okozta interruptok, (2) Sleep üzemmód hacsak nincs task a sorbaállási sorban, ébresztés eseményre. A task-ok idő-flexibilis háttér job-ok, egymáshoz viszonyítva atomikusak, azaz egymás nem szakítják meg, futásokat csak interruptként megjelenő események szakíthatják meg. Az események időkritikus, rövidebb idejű programrészek, LIFO (Last-in First-Out) logika szerint kerülnek feldolgozásra, kezdeményezhetik task-ok késleltetett futását.

A programok komponensekből épülnek fel, minden komponens specifikál egy interfészt, és ezek segítségével kerül sor a "huzalozásra", aminek eredménye a konfiguráció.

A komponenseknek kétirányú interfészeket használnak (use), ill. biztosítanak (provide). A komponensek parancsokat (command) hívnak és implementálnak, és eseményeket (events) jeleznek és kezelnek. A komponensek a használt (used) interfészeken érkező eseményeket kezelik, és a parancsokat implementáló interfészeket biztosítanak (provide).

A komponensek hierarchiája: A parancsok "lefelé" haladnak, nem blokkoló kérések, a vezérlés a hívóhoz kerül vissza. Az események "felfelé" haladnak, taskot helyeznek el a várólistán (function queue scheduling), alacsonyabb szintű parancsot hívnak. A vezérlés a jelzést adóhoz kerül vissza.



Ciklusok elkerülésére azáltal kerül sor, hogy az események hívhatnak parancsokat, de a parancsok nem tudnak eseményt kezdeményezni.

Kommunikáció szenzorhálózatokban

(A szenzorhálózatokon belüli kommunikációról részletes fólia-sorozat található a tantárgy tanszéki honlapján. Az alábbiak csak néhány kiemelt jellemzőt foglalnak össze.)

Szabványos megoldások: tipikusan az ISM (Industrial, Scientific, Medical) 2.4 GHz-es sávban, szórt spektrummal: ZigBee/IEEE 802.15.4, IEEE 802.11b (Wi-Fi) WLAN (Wireless Local Area Network), Bluetooth WPAN (Wireless Personal Area Network).

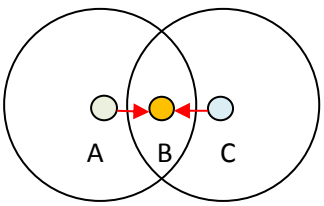
Közeghozzáférés: Statikus/Dinamikus.

Szenzorhálózatokban tipikus a dinamikus (igény szerinti) csatorna-hozzáférési jog kiosztás, ezen belül is a CSMA: Carrier Sense Multiple Access.

Az ütközés elkerülés módja: adás előtt belehallgat a csatornába, ha nem érzékel adást, akkor adni kezd, ha adást érzékel, akkor vár.

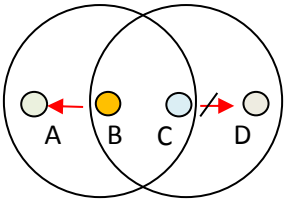
CSMA problémák:

Rejtett terminál problémája:



- A ad B-nek
- C nem hallja A-t!
- C is ad B-nek
- B egyik adást sem tudja venni

Látható terminál problémája:



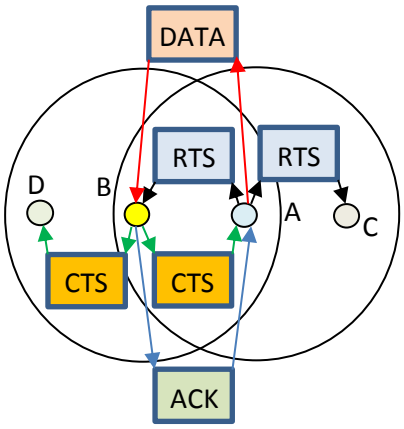
- B ad A-nak
- C is szeretne adni D-nek!
- C hallja B-t
- C nem ad, bár nem okozna ütközést

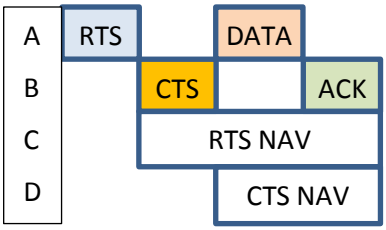
CSMA módosítások

CSMA foglaltság jelzéssel: két csatornát használunk, az egyiket az adat továbbítására, a másikat a foglaltság jelzésére. A vevő a foglaltság csatornán folyamatosan jelez. Adás előtt az adó ellenőrzi mind az adatcsatornát, mind a foglaltság-csatornát. A csomópontnak egyszerre kell adni és venni, ami költséges. Az egyidejű két csatorna nagyobb sávszélességet köt le.

Request To Send/Clear To Send (RTS/CTS): Két fázisban működik: (1) Handshake, (2) adattovábbítás. Az alapgondolat: az ütközés a vevőnél történik. Kizárja a rejtett terminál problémát. Hosszabb üzenetek esetén előnyös, egyébként nagy az overhead.

Működése:





- Az „A” adó RTS üzenetet küld (“B”-nek)
- A „B” vevő CTS üzenettel válaszol
- Az adó a CTS vétele után továbbítja az adatcsomagot

A többi csomópont RTS, CTS vétele után nem adhat!
(NAV = Network Allocation Vector)

Routing (Adásvonal vezetés, útvonalválasztás)

A szenzorhálózat ad-hoc. A csomópontok véletlen eloszlásúak, a kapcsolatok véletlenszerűen jönnek létre, nem megbízhatóak (fading), a csomópontok lehetnek mobilak, és lehetnek sokan.

Tipikus feladatok:

- Egy forrás → sok (akár minden csomópont) cél. Pl. egy központi csomópont utasításokat terjeszt a hálózatban.
- Sok forrás → egy cél. Pl. adatgyűjtés és továbbítás a központba.
- Egy forrás → egy cél. Pl. adatcsere csomópontok között.

Adatküldési modellek:

- *Idővezérelt:* a szenzorok működése és az adatküldés idővezérelt. Tipikus alkalmazás: előre eltervezett adatgyűjtés. Energiatakarékos működéshez előnyös. Alvás → szinkronizált ébredés.
- *Eseményvezérelt:* a szenzorok működését környezeti események kezdeményezik. Időkritikus alkalmazásoknál célszerű. Energiatakarékos üzem nehezebben valósítható meg.
- *Lekérdezéses:* a szenzorok a központ lekérdező parancsára aktiválódnak.

Tipikus hálózati struktúrák:

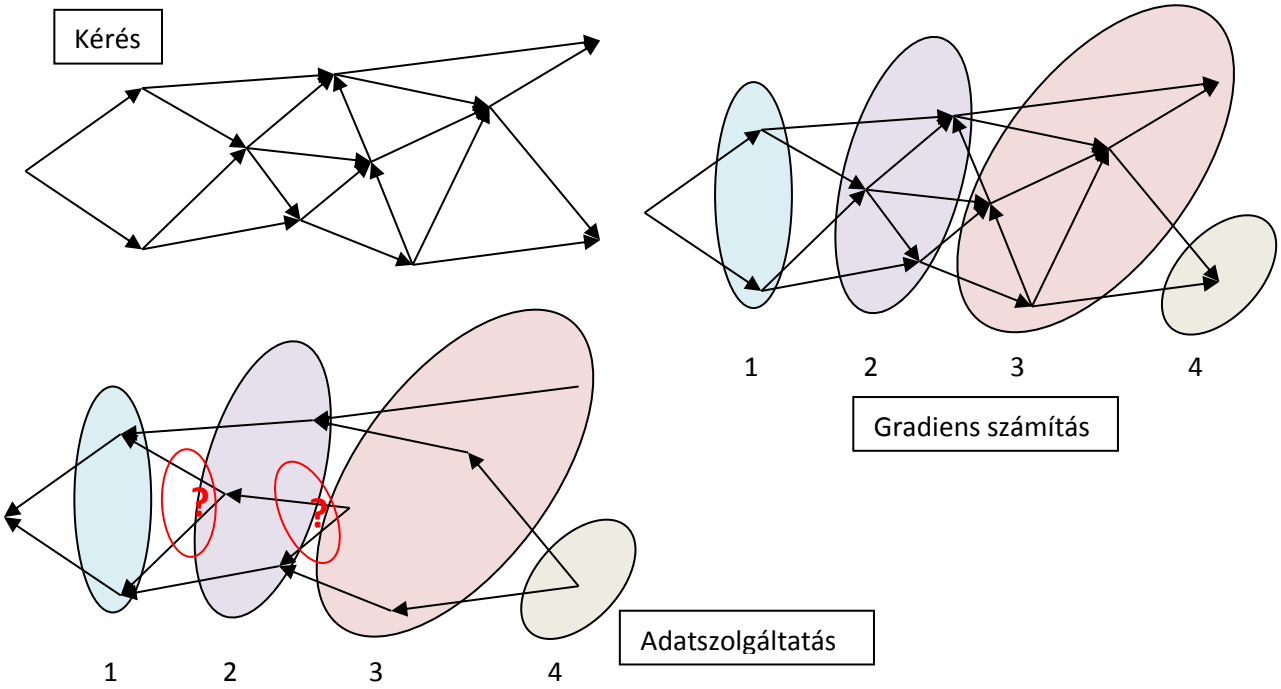
- *Egyszintű (Flat):* Egyenrangú csomópontok, nehezen skálázható.
- *Hierarchikus:* Csoportok alakulnak: csoporton belüli és csoportok közötti kommunikáció működik. A csoportok között vezérlő csomópontok tartják a kapcsolatot. A vezérlők kitüntetett képességűek. A vezérlő szerep dinamikusan változhat.

Egyszintű: *Elárasztásos adásvonal vezetés/útvonalválasztás (Flood routing):*

- *Üzenetszórásos* (broadcast) üzenettovábbítás.
- Minden üzenet első vételekor a vevő megjegyzi az üzenetet vagy ha nem szól akkor csak az azonosítóját, majd szétsugározza az üzenetet.
- *Tipikus alkalmazás:* egy forrás → sok cél (parancsot kap kvázi egyidejűleg).
- *Előnye:* egyszerű, hibatűrő a nagy redundancia miatt.
- *Hátránya:* rengeteg (feleslegesnek bizonyuló) üzenet és energiafogyasztás, továbbá ütközések (rejtett terminál.)
- *Módosítások:*
  - a vevő csak *p* valószínűséggel terjeszt tovább. A *p* topológia-függő.
  - az ütközések elkerülése érdekében: a vétel után késleltetett továbbítás, véletlen várakozási idő.

Egyszintű: *Gradiens-alapú adásvonal vezetés/útvonalválasztás(Gradient Based Routing (GBR)):*

- *Három fázis:* (1) Kérés, (2) Gradiens számítás, (3) adatszolgáltatás.
- *Tipikus alkalmazás:* sok forrás → egy cél (adatgyűjtés).
  (1) Kérés: a központ kérést küld a hálózatba: terjesztés elárasztással.
  (2) Gradiens számítás: a kérés terjesztése közben „gradiens mérés” → A “gradiens” a legrövidebb “távolság” a központtól: Legkevesebb hány lépésben küldhető meg a kért adat a központnak.
  (3) Adatszolgáltatás: adat továbbítás a legrövidebb távolságú úton, eközben aggregáció lehetséges.



GBR változatok: Több lehetséges útvonalból melyiket válasszuk?

- *sztochasztikus:* véletlen választás
- *energia-alapú* kiegészítés: a kevés energiájú csomópont megemeli a saját „gradiens” értékét, így másfelé tereli a forgalmat.

Hierarchikus: csak említés szintjén, a név alapján visszakereshető az interneten:

Low Energy Adaptive Clustering Hierarchy (LEACH): Hierarchikus, dinamikusan létrejövő klaszterekre alapozva.

Geographic and Energy Aware Routing (GEAR): Elhelyezkedés alapú, üzenet csak a célzott régió felé halad.

8.5. Néhány megjegyzés a biztonságkritikus rendszerek témában

(Az alábbiak kivonatok Dr. Majzik István (BME MIT) a "Valósidejű és biztonságkritikus rendszerek" című tárgyhoz készített előadásvázlatából.)

Biztonsági követelmények rendszere

- Kockázatelemzés: Tolerable Hazard Rate (THR): eltűrhető veszélygyakoriság, eltűrhető veszély ráta
  - Folyamatos üzem esetén a veszélyt okozó hibajelenség gyakorisága óránként;
  - Nem folytonos üzem esetén a veszélyt okozó hibajelenség valószínűsége a funkció meghívásakor
- Kategóriákba sorolás: Safety Integrity Level (SIL) – Biztonságintegritási szint

SIL	Biztonságkritikus funkció hibája/óra
1	10^{-6}<THR<10^{-5}
2	10^{-7}<THR<10^{-6}
3	10^{-8}<THR<10^{-7}
4	10^{-9}<THR<10^{-8}

1 év = 8760 óra. SIL4 feltételezésével: $10^8/8760 \cong 11415$ év hiba nélkül. Ha 15 év az élettartam, akkor ~750 berendezésből egyben lesz hiba, mert 15*750=11250.

A szolgáltatásbiztonság alapjellemzői:

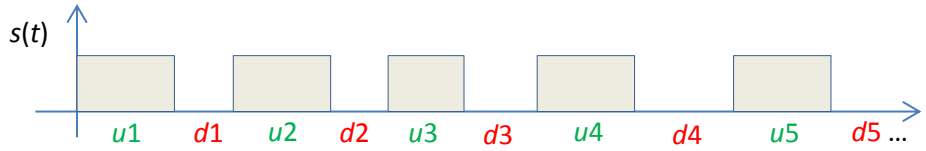
- Megbízhatóság: a rendszer folyamatos szolgáltatást nyújt;
- Rendelkezésre állás: a rendszer (javítva) használatra kész;
- Biztonság(osság): nincs káreset/baleset;
- Bizalmasság: nincs jogosulatlan információközlés;
- Karbantarthatóság: javítás és fejlesztés lehetősége;

A szolgáltatásbiztonság további jellemzői:

- Tesztelhetőség: tesztelés lehetősége;
- Teljesítőképesség; teljesítmény és megbízhatóság.

Megbízhatósági mértékek:

- Állapot particionálás: $s(t)$ rendszerállapot: hibás (D), hibamentes (U) állapotpartíció.



Várható értékek:

- Első hiba bekövetkezése: MTFF=E{u1} Mean Time to First Failure
- Hibamentes működési idő: MUT=E{ui} Mean Up Time
- Ugyanez: MTTF Mean Time To Failure
- Hibás állapot ideje: MDT=E{di} Mean Down Time
- Ugyanez: MTTR Mean Time To Repair
- Hibák közötti idő: MTBF=MUT+MDT Mean Time Between Failures

Valószínűségi időfüggvények:

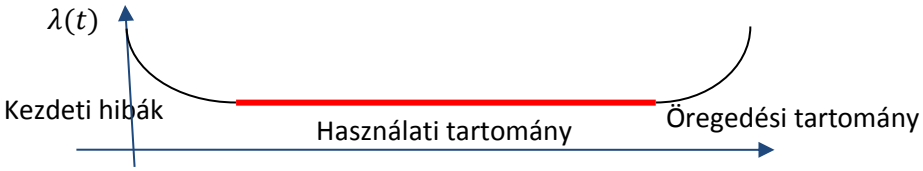
- Rendelkezésre állás: a(t)=P{s(t)∈ U} közben meghibásodhat (idővel csökken)
- Megbízhatóság: r(t)=P{s(t') ∈ U} $\forall t' < t$, nem hibásodhat meg
- Készenlét: K=$lim_{t\to\infty}$a(t) rendszeresen javított rendszer esetén (idővel nullára csökken)

- Készenlét: K=A=MTTF/(MTTF+MTTR)

Komponens jellemzők:

- Meghibásodási tényező: λ(t) milyen valószínűséggel hibásodik meg *t* környezetében?
  $\lambda(t)\Delta t = P\{s(t+\Delta t) \in D | s(t) \in U\}, \Delta t \to 0.$
  részletezve (lásd az eloszlás és sűrűségfüggvények, valamint deriváltjaik kapcsolatát):
  $\lambda(t) = -\frac{1}{r(t)}\frac{dr(t)}{dt}$, amivel $r(t) = e^{-\int_0^t \lambda(t)dt}$.

Elektronikai alkatrészek kádgörbéje:



A használati tartományban $\lambda(t) = \lambda$.
Exponenciális eloszlást feltételezve:

$$r(t) = e^{-\lambda t}$$

$$MTFF = E\{u1\} = \int_0^\infty r(t)dt = \frac{1}{\lambda}$$

Megjegyzés: A kezdeti hibákat gyártás utáni teszttel szűrik ki.

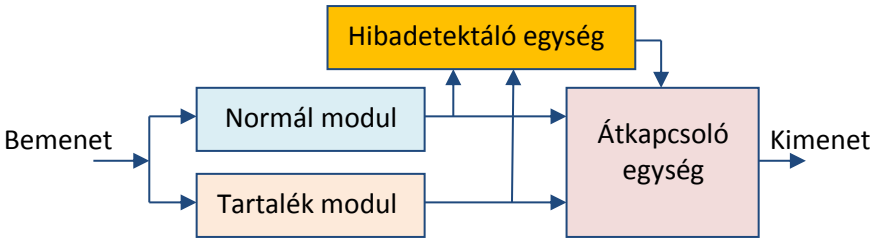
Hibatűrő működés
Redundancia: (1) Hardver, (2) Szoftver, (3) Információ, (4) Idő.
Redundancia típusai: hideg tartalék, langyos tartalék, meleg tartalék, lásd az alábbi táblázatot:

Redundancia/tulajdonság	Hideg tartalék	Langyos tartalék	Meleg tartalék
Alapelv	Csak hiba esetén aktiválva	Csökkentett terheléssel működik	Ugyanúgy működik, mint az elsődleges
Előnye	Nem hibásodik meg a passzív komponens	Kisebb meghibásodási tényező	Gyorsan átveheti az elsődleges helyét
Hátránya	Lassan veszi át az elsődleges helyét	Közepes sebességű feladat átvétel	Azonos meghibásodási tényező
Példa	Kikapcsolt tartalék számítógép	Naplózó számítógép belép elsődlegesként	Árnyék számítógép

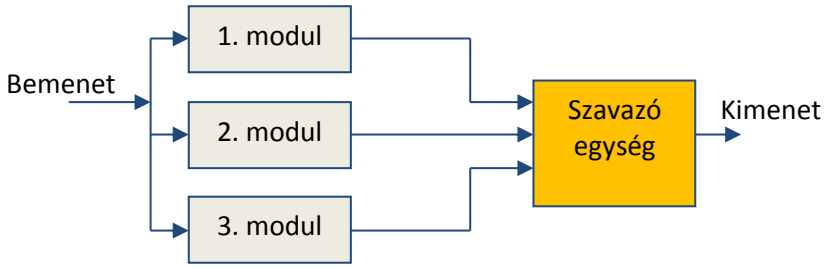
Milyen redundancia használandó?

- Hardver tervezési hibák (<1%): hardver redundancia, eltérő tervezésű.
- Hardver állandósult működési hibák (~10%): hardver redundancia, pl. tartalék processzor.
- Szoftver tervezési hibák (~10-20%): szoftver redundancia, eltérő tervezésű.
- Hardver időleges működési hibák (~70-80%): idő-redundancia (pl. utasítás újravégrehajtás), információ redundancia (pl. hibajavítás), szoftver redundancia (pl. állapotmentés és helyreállítás).

Állandósult hardver hibák kezelése:
Kettőzés: alapesetben csak hibadetektálás, a hibatűréshez diagnosztikai támogatás és átkapcsolás kell.



TMR: Triple-modular redundancy: Hiba maszkolása többségi szavazással. A szavazó kritikus, de egyszerű.

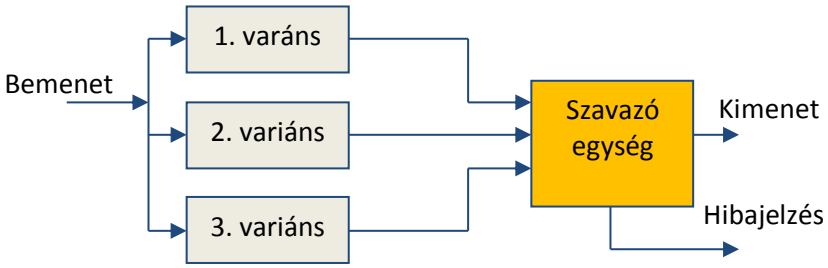


NMR: N-modular redundancy: Hiba maszkolás többségi szavazással. A missziós idő túlélése nagyobb esélyű, utána javítás lehetséges. Repülőgép fedélzeti eszközök: 4MR, 5MR, esetenként 7MR.

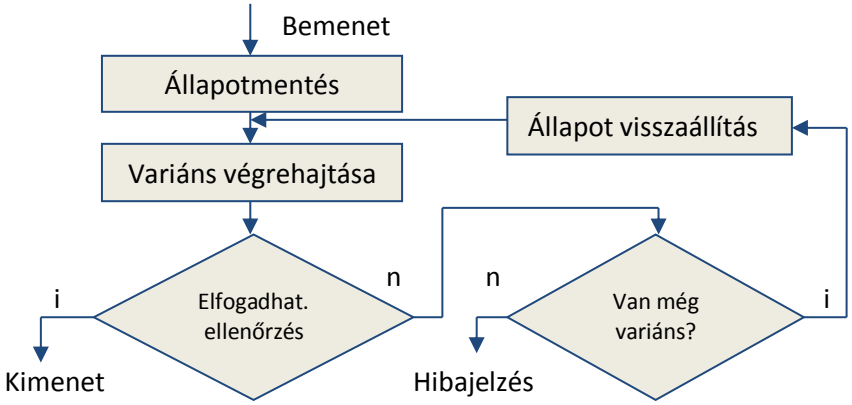
Szoftver hibák kezelése:

Variánsok alkalmazása: azonos specifikáció; de eltérő algoritmus, adatstruktúra; más fejlesztési környezet, programnyelv; elszigetelt fejlesztés.

N-verziós programozás: aktív redundancia, a variánsok párhuzamos végrehajtása, többségi szavazás. Ha a variánsok kimeneteire elfogadhatósági tartományt is adunk, akkor a szavazó azt is ellenőrzi. A szavazó maga ún. egyszeres hibapont, azaz ha elromlik, akkor a funkció kiesik, de a szavazó egyszerű, ezért kisebb a kockázat.



Javító blokkok technikája: passzív redundancia, csak hibaesetén aktiválódik. A variánsok kimenetének elfogadhatóságát ellenőrizzük, ha erre nincs lehetőség, akkor a módszer nem alkalmazható. Ha hiba lép fel, akkor tartalék variáns soros végrehajtására kerül sor.

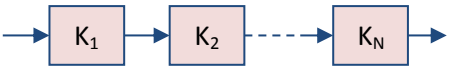


Összehasonlítás:

Tulajdonság/típus	N-verziós programozás	Javító blokkok
Ellenőrzés	Szavazás, relatív	Elfogadhatóság, abszolút
Végrehajtás	Párhuzamos	Soros
Időigény	Leghosszabb variáns v. time-out	Hibák számától függ
Redundancia aktiválása	Mindig	Csak hiba esetén
Tolerált hibák	[(N-1)/2]	N-1
Hibakezelés	Maszkolás	Helyreállítás

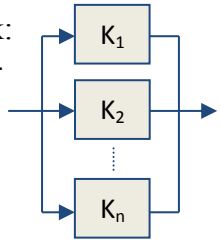
Megbízhatósági blokkdiagram (Reliability Block Diagram)

1. Soros rendszer: a komponensek sorba kapcsolódnak:



A rendszer akkor hibátlan, ha valamennyi komponens az.
A rendszer megbízhatósága a komponensek megbízhatóságának szorzata:
$r_R(t) = \prod_{i=1}^{N} r_i\,(t)$. Ha a komponensek meghibásodási tényezője λ_i, akkor a rendszer $MTFF = \frac{1}{\sum_{i=1}^{N} \lambda_i}$.

2. Párhuzamos rendszer: A komponensek párhuzamosan kapcsolódnak:

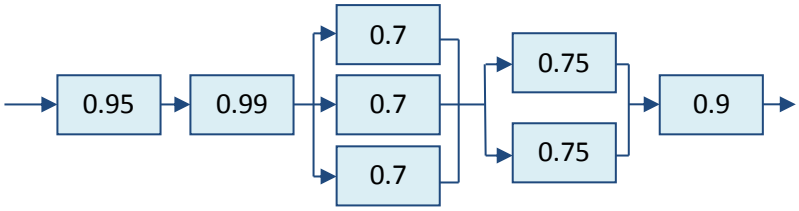


A rendszer akkor hibás, ha valamennyi komponens hibás. A hiba valószínűsége: (1- megbízhatóság). $1 - r_R(t) = \prod_{i=1}^{N}(1 - r_i\,(t))$. Ha a komponensek megbízhatósága azonos: $r_K(t)$, akkor

$$r_R(t) = 1 - (1 - r_K(t))^N$$

Ha a komponensek meghibásodási tényezője λ, akkor a rendszer $MTFF = \frac{1}{\lambda}\sum_{i=1}^{N}\frac{1}{i}$.

3. Összetett rendszer: részenként számítható:



A rendszer készenlét a komponensek készenléti adataiból:

$$K_R = 0.95 \cdot 0.99 \cdot [1 - (1 - 0.7)^3] \cdot [1 - (1 - 0.75)^2] \cdot 0.9 = 0.95 \cdot 0.99 \cdot 0.973 \cdot 0.9375 \cdot 0.9 = 0.77$$

4. N-ből M hibás komponens esete: N egyforma komponens, M vagy több komponens hiba esetén a rendszer is hibás. A rendszer megbízhatósága (a komponensek megbízhatósága egyforma: *r*):

$$r_R = \sum_{i=0}^{M-1} P\{\text{éppen i hiba van}\} = \sum_{i=0}^{M-1} \binom{N}{i} (1-r)^i r^{N-i}$$

Ideális többségi szavazás (TMR): N=3, M=2 esetén:

$$r_R = \sum_{i=0}^{1} \binom{3}{i} (1-r)^i\, r^{3-i} = \binom{3}{0} (1-r)^0 r^3 + \binom{3}{1} (1-r)^1 r^2 = 3r^2 - 2r^2.$$

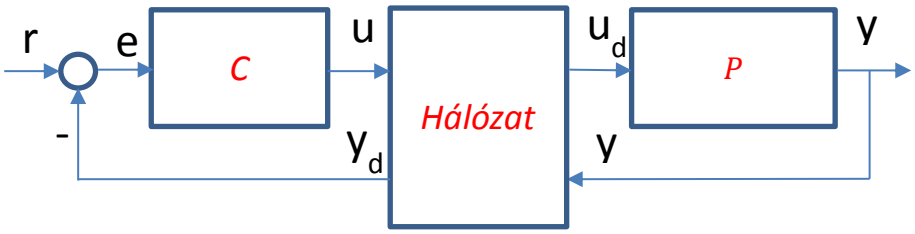
Exponenciális eloszlást feltételezve: $r(t) = e^{-\lambda t}$ alkalmazásával:

$MTFF = \int_0^\infty r_R(t)dt = \int_0^\infty (3r^2 - 2r^3)dt = \frac{3}{2\lambda} - \frac{2}{3\lambda} = \frac{5}{6\lambda}$, ami kisebb, mintha csak egy komponens lenne.

8. Esettanulmányok (folyt.)

8.6. Szabályozás hálózaton keresztül

A *C* szabályozó (Controller) és a *P* szakasz (Plant) egymástól távol helyezkednek el, kommunikációjuk számítógép-hálózaton keresztül történik. Az *r* az alapjel, az *e* az ellenőrző jel, az *u* a beavatkozó jel, és *y* a kimeneti jel. Az u_d a beavatkozó jel hálózat által késleltetett értéke, az y_d pedig a kimenőjel hálózat által késleltetett értéke.

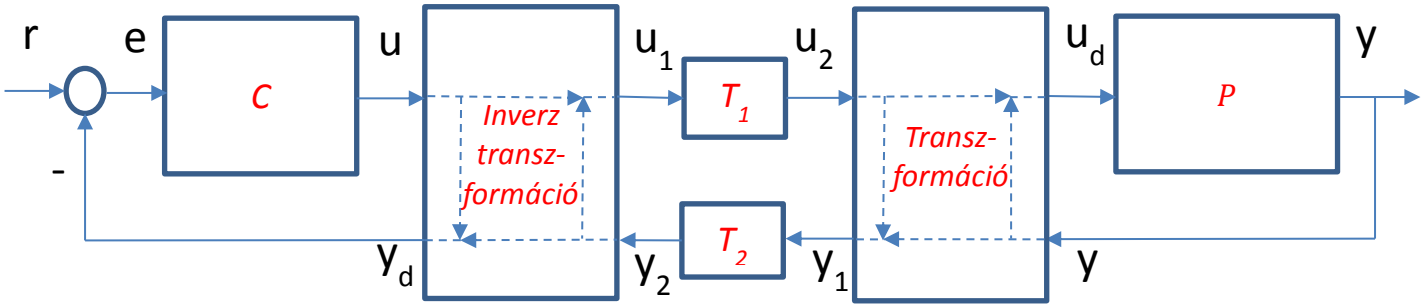


A hálózat késleltetése tetszőleges mértékű lehet, de az alábbiakban konstans értékűnek feltételezzük. A késleltetés megjelenése a hurokban nyilvánvaló stabilitási problémák forrása tud lenni. Ennek elkerülésére az alábbiakban az egyes részelemek passzivitásának kikényszerítését alkalmazzuk. A rendszer egy eleme, például a *P* szakasz passzív, ha minden $t > 0$ esetre fennáll:

$$\int_0^t P_{in}(\tau)d\tau = \int_0^t u_d(\tau)y(\tau)d\tau \geq 0$$

Itt feltételeztük, hogy $t = 0$-ban a szakasz energiamentes. Az alábbiakban bemutatjuk, hogy a hálózat és a szakasz, valamint a szabályozó és a hálózat közötti alkalmas jel-transzformációval egy passzív *P* szakasz az őt kiegészítő hálózattal együtt passzívvá tehető, és alkalmas szabályozóval kiegészítve a zárt kör stabilitása tetszőleges (de véges) mértékű késleltetés mellett is biztosítható.

$$u_2(t) = u_1(t - T_1)$$



$$\begin{bmatrix} u_1(t) \\ y_2(t) \end{bmatrix} = T^{-1} \begin{bmatrix} u(t) \\ y_d(t) \end{bmatrix} \qquad y_2(t) = y_1(t - T_2) \qquad \begin{bmatrix} u_d(t) \\ y(t) \end{bmatrix} = T \begin{bmatrix} u_2(t) \\ y_1(t) \end{bmatrix}$$

A hálózatban tárolt energia: $V_N(t) = \int_0^t [(u_1^T(\tau)u_1(\tau) + y_1^T(\tau)y_1(\tau) - u_2^T(\tau)u_2(\tau) - y_2^T(\tau)y_2(\tau)]d\tau$

ahol megengedjük vektorok alkalmazását, és amibe a késleltetők kimeneti jelét behelyettesítve

$$V_N(t) = \int_{t-T_1}^t u_1^T(\tau)u_1(\tau)d\tau + \int_{t-T_2}^t y_1^T(\tau)y_1(\tau)d\tau \geq 0$$

értéket kapunk, hiszen a hálózatba bevitt és az onnan kivitt energiák különbsége éppen a T_1 és T_2 időtartamokhoz rendelhető energiamennyiség. Mivel ez pozitív, ezért az előző kifejezés átírható

$$\int_0^t [(u_1^T(\tau)u_1(\tau) - y_2^T(\tau)y_2(\tau)]d\tau \geq \int_0^t [u_2^T(\tau)u_2(\tau) - y_1^T(\tau)y_1(\tau)]d\tau$$

formába, azaz a P szakasz transzformált belső energiája, és a hálózat belső energiája nagyobb, mint a P szakasz transzformált belső energiája. Keresett az a transzformáció, amely a P szakasz belső energiáját a megadott formába transzformálja:

$$\begin{bmatrix} u_d(t) \\ y(t) \end{bmatrix} = \begin{bmatrix} t_{00} & t_{01} \\ t_{10} & t_{11} \end{bmatrix} \begin{bmatrix} u_2(t) \\ y_1(t) \end{bmatrix} \qquad \int_0^t (u_d^T(\tau)y(\tau)\, d\tau = \int_0^t [u_2^T(\tau)u_2(\tau) - y_1^T(\tau)y_1(\tau)]d\tau$$

Keresett az a transzformáció, amely a P szakasz és a hálózat belső energiáját a megadott formába transzformálja:

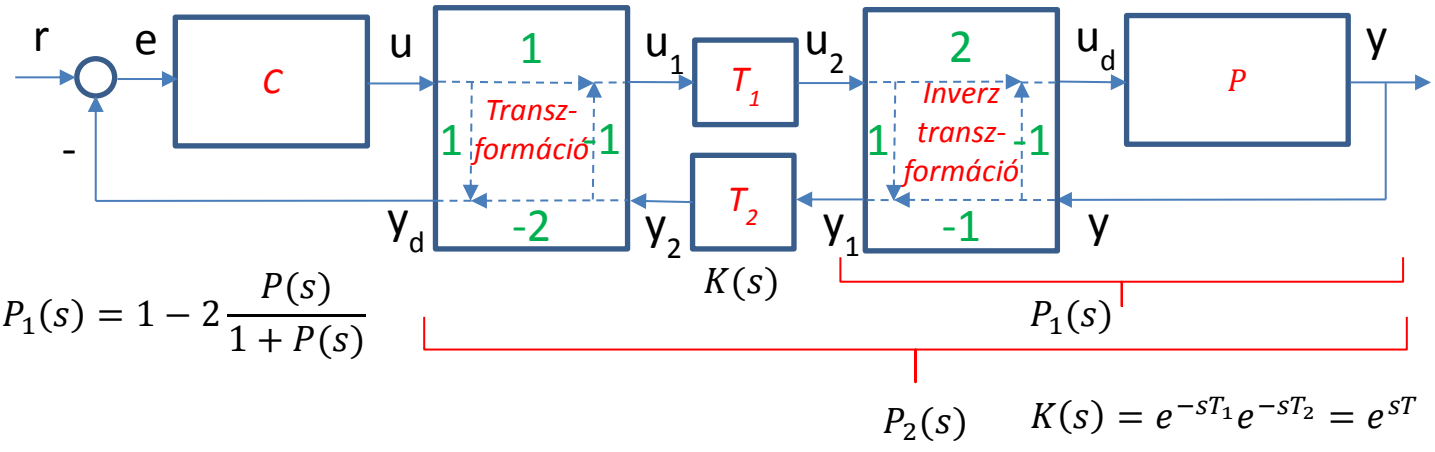
$$\begin{bmatrix} u(t) \\ y_d(t) \end{bmatrix} = \begin{bmatrix} t_{00} & t_{01} \\ t_{10} & t_{11} \end{bmatrix} \begin{bmatrix} u_1(t) \\ y_2(t) \end{bmatrix} \qquad \int_0^t u^T(\tau)y_d(\tau)\, d\tau = \int_0^t [u_1^T(\tau)u_1(\tau) - y_2^T(\tau)y_2(\tau)]d\tau$$

Egy lehetséges megoldás:

$$\begin{bmatrix} u_d(t) \\ y(t) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} u_2(t) \\ y_1(t) \end{bmatrix} \qquad \begin{bmatrix} u(t) \\ y_d(t) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} u_1(t) \\ y_2(t) \end{bmatrix}$$

A hozzátartozó implementáció:

$$\begin{bmatrix} u_d(t) \\ y_1(t) \end{bmatrix} = \begin{bmatrix} 2 & -1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} u_2(t) \\ y(t) \end{bmatrix} \qquad \begin{bmatrix} u_1(t) \\ y_d(t) \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 1 & -2 \end{bmatrix} \begin{bmatrix} u(t) \\ y_2(t) \end{bmatrix}$$



$$P_1(s) = 1 - 2\frac{P(s)}{1 + P(s)} \qquad K(s) = e^{-sT_1}e^{-sT_2} = e^{sT}$$

$$P_2(s) = \frac{1 - K(s) + P(s)(1 + K(s))}{1 + K(s) + P(s)\big(1 - K(s)\big)} = \frac{1 + P(s) - \big(1 - P(s)\big)K(s)}{1 + P(s) + \big(1 - P(s)\big)K(s)}$$

A továbbiakban ez kerül $P(s)$ helyére a szabályozó tervezés során, de ha P passzív, akkor P_2 is az!

9. Összefoglalás

A tárgy súlypontjainak és legfontosabb üzeneteinek áttekintése