Beágyazott információs rendszerek

2. Ütemezés (folyt.)

2020. szeptember 17.

Mennyiségek, változók valós idejű rendszerekben

Ahhoz, hogy az RT képre alapozott számításaink pontosak legyenek, be kell tartanunk következő feltételt: $\left[C(t_{felhasználás}) - C(t_{megfigyelés})\right] \leq d_{pontosság}$

Egy periodikusan frissített RT képet ***parametrikusnak***, vagy ***fázis-érzéketlennek*** hívunk, ha

$$d_{pontosság} > \left(d_{frissítés} + WCET_{\text{üzenet továbbítás}}\right).$$

A parametrikus RT kép a vevő oldalon bármikor felhasználható anélkül, hogy a beérkezés és a felhasználás fázisviszonyait mérlegelni kellene: még a pontossági időn belül megjön a frissítés.

Egy periodikusan frissített RT képet ***fázis-érzékenynek*** hívunk, ha

$$WCET_{\text{üzenet továbbítás}} < d_{pontosság} < \left(d_{frissítés} + WCET_{\text{üzenet továbbítás}}\right).$$

Ilyenkor nem biztos, hogy a pontossági időn belül megjön a frissítés, ezért a frissítés és a felhasználás idejére oda kell figyelni.

A fázis érzékenységet megfelelő mintavételi frekvenciával, vagy állapotbecslés alkalmazásával kerülhetjük el.

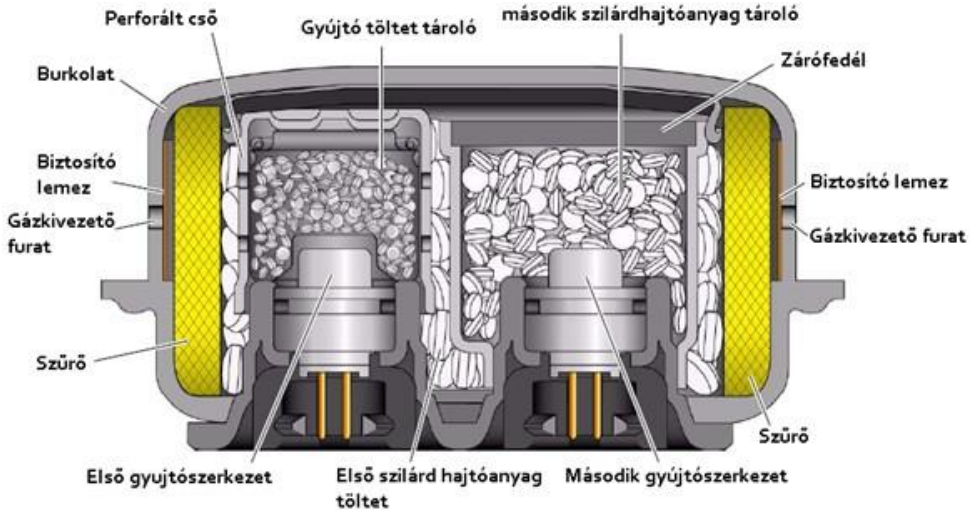
Idempotencia: Ugyanazt az üzenetet többször átküldve az eredmény ugyanaz!

Példa: szelep-állás 45° (állapot üzenet) ↔ szelep-állás változás 5° (esemény üzenet).

Emlékeztető

Kemény és puha valós idejű rendszerek:

- **Kemény valós idejű rendszer** (hard real-time system (HRT)):
  katasztrofális következményekkel jár, ha nem tartjuk az időkorlátot (pl. légzsák vezérlése).
- puha valós idejű rendszer (soft real-time system (SRT), on-line system):
  az eredmény értékes az időkorláton túl is, de az idővel degradálódik (pl. banki rendszerek).



(Kormány)légzsák kioldó szerkezet

Bankautomata, ATM

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, hogy legyen.
SRT esetén a válaszidő **másodperc** nagyságrendű, az időkorlát túllépése nem okoz katasztrófát.

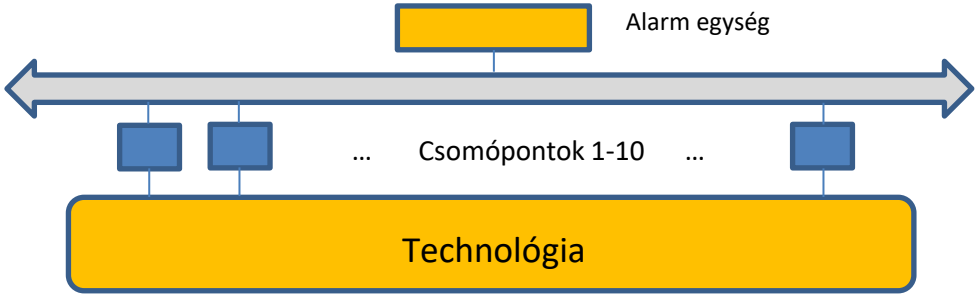
- *viselkedés csúcsterhelés esetén* (peak-load performance):

HRT esetén jól definiált kell, hogy legyen.

„Ha esik, ha fúj!”

Eseményvezérelt (ET) és idővezérelt (TT) rendszerek:

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**-on belül el kell jutnia az alarm egységhez.

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. intermessage 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.

Idővezérelt eset: TT/CAN protokoll szerint.

A csomópontok rendszeresen beküldik az állapotjelző biteket az alarm egységnek.

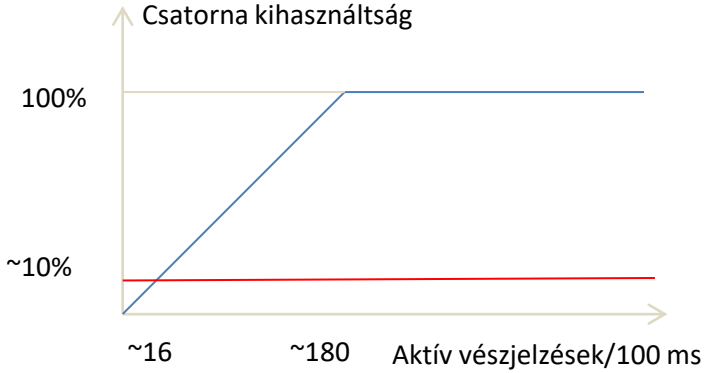
Ez **40** bitenként egy-egy üzenet.

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**.

Emlékeztető

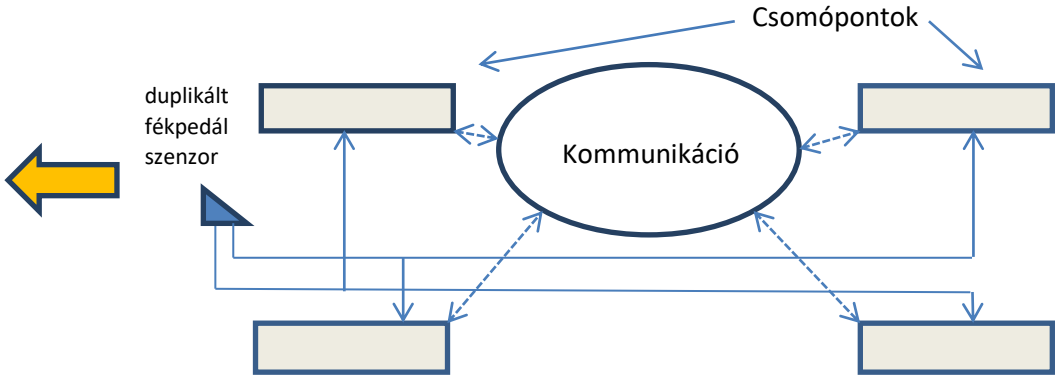
88 bites üzenetekben gondolkodva **10 000/88 ~ 110** juthat át a határidőn belül.

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



Megegyezés protokollok jelentősége

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



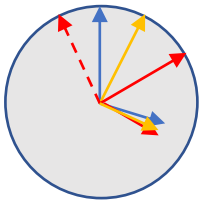
A példa szerint a biztonság érdekében **duplikált fékpedál szenzort** alkalmazunk.
Az egyes kerekek fékjeihez önálló vezérlő csomópontok tartoznak.
A csomópontok egymást tájékoztatják arról, hogy mi az ő véleményük a szenzor értékéről, és kiszámítják a fékerőt.

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 befejezés, elosztott választás, stb.
Hibák fellépése esetén is megállapodásra kellene jutni! **Ez nem mindig sikerül!**

Megegyezés **bizánci típusú hibák** esetén: **Példa:** Órák szinkronizálása:

Emlékeztető



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.

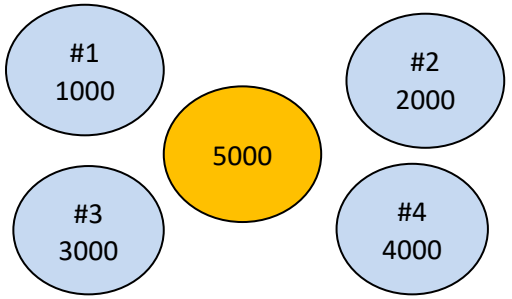
Ezt a hibafajtát nevezzük **bizánci típusú hibának**.

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 **3*k*+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:



Menetközben kiderül, hogy az egyik generális hazudós („szoftver hiba”). Az ellenségnek **5000** katonája van.

Az ábrán látható elrendezésben 4 hadtest generálisa megegyezésre törekszik az egyszerre harcba küldhető katonák számát illetően.
A (formális) szövetségesek egymással hibamentesen kommunikálnak: mindenki megküldi a katonái számát.
A **#3** számú generális/csomópont hazudós (szoftver hibás):
x, y, z a ténylegestől különböző, egymástól potenciálisan eltérő értéket küld a helyes (**3000**) helyett.

Az egyes csomópontokban az alábbi adatok állnak rendelkezésre:
#1: (1K, 2K, xK, 4K),
#2: (1K, 2K, yK, 4K),
#3: (1K ,2K ,3K, 4K),
#4: (1K, 2K, zK, 4K).

Annak érdekében, hogy az értékek helyes voltát ellenőrizni tudják, az információs vektoraikat körbeküldik a kommunikációs csatornáikon keresztül.

A csatornák az előzőek szerint fognak viselkedni, tehát a hazudós csomópont a körbeküldött vektor-elemeket illetően is hazudós.

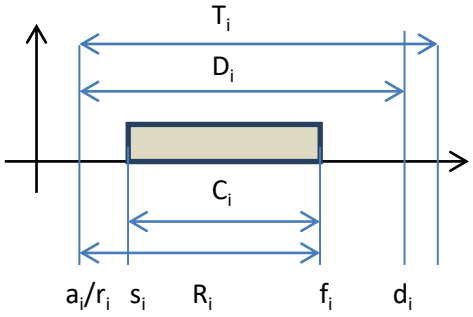
A körbeküldést követően az egyes csomópontokban a következő információ áll rendelkezésre (ezer katonában):

#1: $\begin{bmatrix} 1 & 2 & y & 4 \\ a & b & c & d \\ 1 & 2 & z & 4 \end{bmatrix}$ **#2**: $\begin{bmatrix} 1 & 2 & x & 4 \\ e & f & g & h \\ 1 & 2 & z & 4 \end{bmatrix}$ **#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. A **#3**-as generálistól esetleges értékek érkeznek.

A nem hazudós generálisok következtetése, hogy $[\mathbf{1K} \quad \mathbf{2K} \quad \mathbf{ismeretlen} \quad \mathbf{4K}]$, azaz lesz legalább **7 ezer** katona, akire számítani lehet a támadásnál.

2. Ütemezés



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

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

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. Az ütemezéssel kapcsolatos döntések **tervezési időben** történnek.

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 adunk 10 ms-ot a második funkciónak,
12.5 Hz-es periodicitással, azaz 80 ms-onként adunk 10 ms-ot a harmadik funkciónak,
6.25 Hz-es periodicitással, azaz 160 ms-onként adunk10 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!

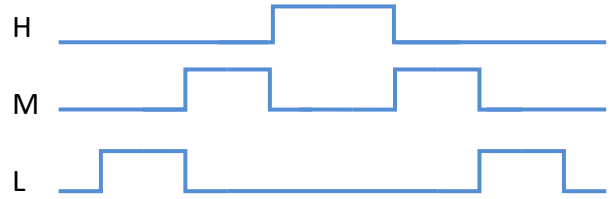
Ezt a tulajdonságot használták ki a világ első digitális szűrőkkel működő valós idejű 1/3 oktáv elemzőjének, **a Brüel & Kjaer 2131** tervezői is 1977-ben!

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.
A megvalósított berendezésben f_m=66.667 kHz. A legmagasabb frekvenciasáv és az összes többi kiszolgálása a mintavételi idő **felét-felét** veszi igénybe.

2. **Időosztásos (time-shared)/körforgó (round-robin) ütemezés:**
A futtatható task-ok egy FIFO-ba kerülnek, és a legelöl álló task fog futni **maximum egy időszelet ideig.** Az időszelet általában néhányszor **10 ms**, ami a taszk-októl független paraméter.
Ha az adott taszk nem fut le az időszelet alatt, akkor futása megszakad, és a FIFO végére kerül.

3. **Prioritásos ütemezés:** A futtatható taszk-ok közül az fut, amelyiknek legnagyobb a prioritása.
A prioritás hozzárendelés történhet tervezési és futási időben egyaránt.

Illusztráció:



A három taszk rendre alacsony (**L**=low), közepes (**M**=medium) és magas (**H**=high) prioritású.
Ezeket a prioritásokat tervezési időben osztottuk ki.
Az ábrán mindhárom taszk azonnal futni kezd, amint futtathatóvá válik.

Az ábrán látható esetben a legalacsonyabb prioritású taszk válaszideje $R_L=C_L+C_M+C_H$.

Ha a középső és/vagy a magas prioritású taszk 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 taszk-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 taszk-ok futása akadályozza az alacsonyabb prioritású taszk-ok végrehajtását.

A $\forall k \in hp_i$ azokat a taszk-okat jelöli ki, amelyek prioritása nagyobb, mint *i* (*hp*=higher priority).
A $\lceil \quad \rceil$ zárójel a felső-egész képzés operátora. $\lceil 1.02 \rceil = 2, \lceil 2.0 \rceil = 2.$
A módszer a válaszidő lehető legkedvezőtlenebb értékét adja meg.
(Worst-case response time.)

Mivel a 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 módszer neve: Deadline Monotonic Analysis (**DMA**).

Feltételezi, hogy a task-okhoz aszerint rendelünk prioritást, hogy mekkora a D_i határidejük.

A módszer alkalmazásánál feltételezzük, hogy $\boldsymbol{D_i \leq T_i}$.

A módszer periodikus task-ok mellett ún. sporadikus task-okra is alkalmazható.

Periodikus taszk: ismert és fix $\boldsymbol{T_i}$ periodusidővel jellemezhető.

Sporadikus taszk: a kérések nem periodikusak, de **ismert** és fix egy olyan $\boldsymbol{T_i}$ időérték, ami minimálisan eltelik két kérés között.

Aperiodikus taszk: a kérések nem periodikusak, és **nincs** egy olyan ismert és fix $\boldsymbol{T_i}$ időérték, ami minimálisan eltelik két kérés között. Egy kérést követően azonnal jöhet egy következő kérés.

Ebben az esetben a **DMA** módszer nem alkalmazható.

Példa: Egy 4 taszkot 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 taszkok sorrendje a prioritási sorrend. Ha a határidők megegyeznek, akkor másodlagos szempontok alapján döntünk a prioritásról.

Határozzuk meg a 3-as taszk worst-case válaszidejét az iteratív eljárás segítségével!

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:

38<50, tehát a 3-as task legkedvezőtlenebb esetben is teljesíti az előírt határidőt.

Vegyük észre, hogy a 4-es taszk adatait az eljárás során nem használtuk fel, a számításhoz felesleges volt megadnunk.
Vegyük azt is észre, hogy taszkokat egymástól függetleneknek képzeltük el.

Egymástól nem független, azaz például egymással kommunikáló, egymásnak adatot továbbító taszkok esetében előfordul(hat), hogy magasabb prioritású taszk alacsonyabb által szolgáltatott adatra várni kénytelen.

Ez a várakozási idő értelemszerűen a mindenkori és a worst-case válaszidejét egyaránt módosítani fogja.

Ütemezhetőség, ütemezhetőségi tesztek:

Szükséges teszt: nem ütemezhető, ha a szükséges feltétel nem teljesül.
Elégséges teszt: biztosan ütemezhető, ha az elégséges feltétel teljesül.
Egzakt teszt: 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 taszkok 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 $\boldsymbol{\mu \leq 1}$ nem teljesül, akkor a task-ok nem ütemezhetőek. A $\boldsymbol{\mu \leq 1}$ tehát szükséges feltétel. n a taszkok száma.

Ha $\boldsymbol{N}$ processzorunk van, akkor $\boldsymbol{\mu \leq N}$ a feltétel.

Válaszidő számítás periodikus és sporadikus taszk-ok esetén:

Példa: Egy 4 taszkot és egy megszakítást (i_1) kiszolgáló rendszer adatai a következők:

Taszk	T	C	D
i_1	10	0.5	3
τ_1	3	0.5	3
τ_2	6	0.75	6
τ_3	14	1.25	14
τ_4	50	5	50

(az idők pl. ms-ban értendők)
Határozzuk meg a τ_4 taszk 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

10.75<50, tehát a határidő minden esetben teljesül!

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. (Volvo 1995-től, az S80-asnál.)