Beágyazott információs rendszerek

2. Ütemezés (folyt.)

2020. október 1.

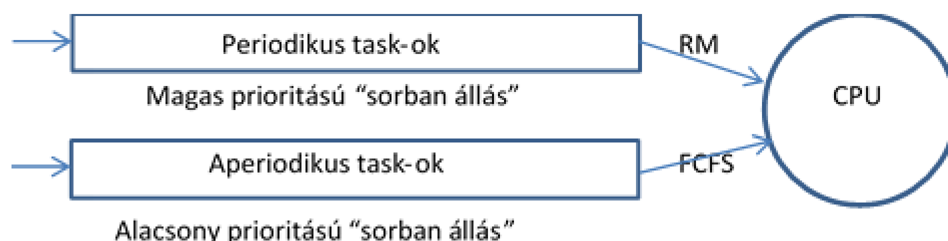
Emlékeztető

Periodikus és aperiodikus taszkok együttes kezelése:

Az alábbiakban ismertetett módszerek esetében a következő előzetes feltételezésekkel élünk:

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

A háttérbeni ütemezés (Background Scheduling) módszere:



A módszer előnye egyszerűsége, hátránya pedig az, hogy az aperiodikus taszkok válaszideje nagyon nagy lehet. (FCFS=First-Come-First-Served.)

Ha az aperiodikus taszkok esetén a válaszidő kritikus, akkor az ún. **szerver-módszerek** alkalmazása jobb eredményt adhat.

A **szerver-módszer** az aperiodikus taszkok végrehajtásához **szeparáltan** biztosít processzor időt. Ennek eszköze a **szerver-taszk**, amelyet a periodikus taszkokkal együtt ütemezünk.

1. Polling Server (PS): Az aperiodikus kérések teljesítése külön ún. **szerver taszk** (S) segítségével, a szerver kapacitás (T_S, C_S) terhére, **független** ütemezési stratégiával történik. **Példa:**

Ha nincsen aperiodikus kérés, amikor a szerver futására sor kerülhetne, akkor a PS **felfüggeszti** magát, kapacitása **nem örződik** meg!

Legyen T_S=5, C_S=2.

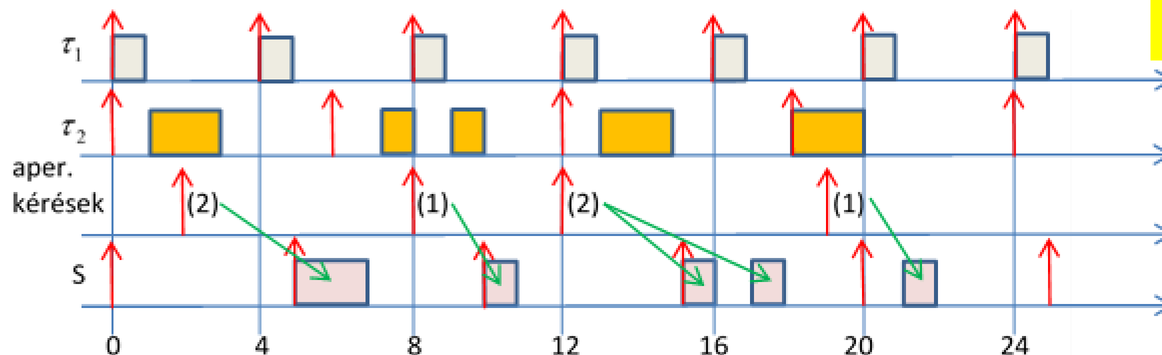
	C	T
τ_1	1	4
τ_2	2	6

A szerver taszk (RM szerint) a középső prioritásra kerül.

A taszkok egyidejű indítását, azaz azonos kezdőfázist feltételezve az ütemezés a következőképpen alakul:

Legyen T_S=5, C_S=2.

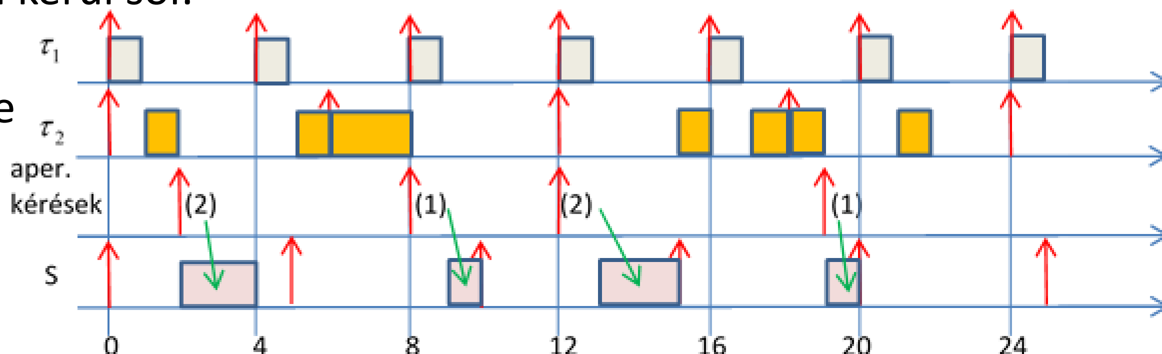
	C	T
τ_1	1	4
τ_2	2	6



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

2. Deferrable Server (DS):

Az aperiodikus kérések teljesítése külön ún. **szerver taszk** (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** taszk futása halasztódik, kapacitását a periódus végéig megőrzi. **Példa:** Az előző példa adataival…

Ezzel a módszerrel az aperiodikus taszkokra **sokkal jobb válaszidők** érhetők el.
(A szerver taszk ütemezése az előzővel azonos módon, RM stratégiával történt.)

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

Példa: Legyen T_S=5, C_S=1. Az ezen kívül ütemezendő task-ok adatai:

	C	T
τ_1	4	10
τ_2	8	20

Emlékeztető

$T_S=5$, $C_S=1$.

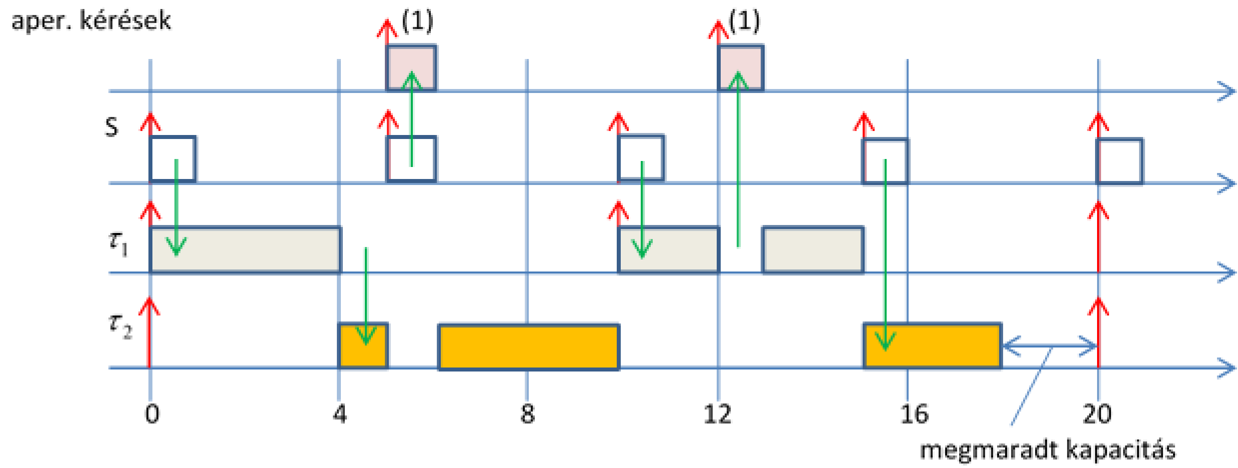
	C	T
τ_1	4	10
τ_2	8	20

A szerver task (RM szerint) a legmagasabb prioritásra kerül.

a processzor kihasználtsági tényező:

$$\mu = \frac{1}{5} + \frac{4}{10} + \frac{8}{20} = 1$$

A taszkok egyszerre indulnak.



Mivel nincsen előzetesen aperiodikus kérés, **az első ütemben** megjelenő szerver kapacitást felhasználja a $\boldsymbol{\tau_1}$ taszk.

Ennek következtében $\boldsymbol{\tau_2}$ taszk korábban indulhat, vagyis a szerver kapacitás ide kerül.

A második ütemben megjelenő szerver kapacitást közvetlenül felhasználjuk.

A **harmadik ütemben** megjelenő szerver kapacitást a $\boldsymbol{\tau_1}$ taszk használja fel, amit a második aperiodikus kérés kiszolgálására visszacserél.

A **negyedik ütemben** érkező szerver kapacitást a τ_2 taszk hasznosítja.

Ezzel együtt kétperiódusnyi szerver kapacitás "halmozódik fel" a végrehajtásánál, ami mozgósítható lenne, ha lenne további **aperiodikus** kérés.

Példa: Legyen T_S=5, C_S=1. A szerver taszk (RM szerint) a legmagasabb prioritásra ke

Emlékeztető

Az ütemezendő taszkok:

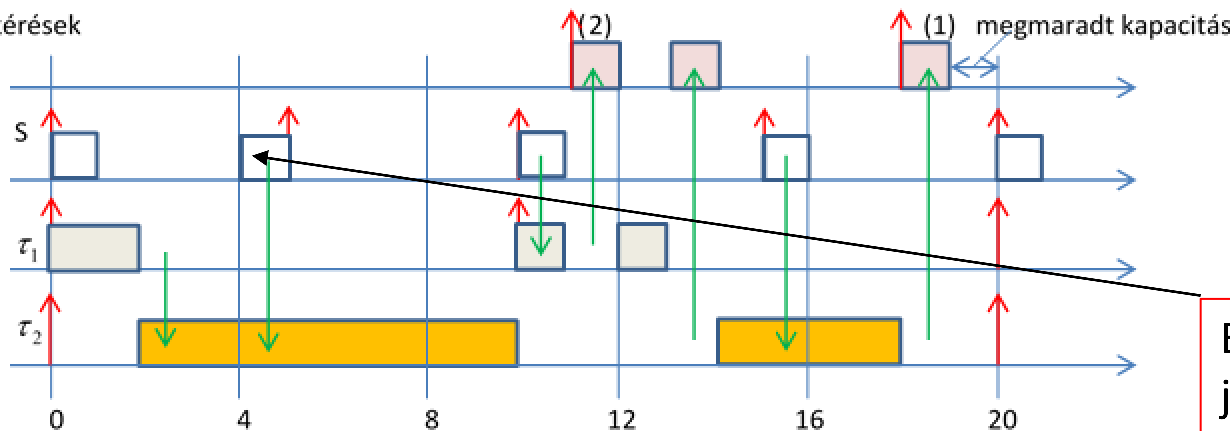
A task-ok egyszerre indulnak:

	C	T
τ_1	2	10
τ_2	12	20

A processzor kihasználtsági tényező:

$$\mu = \frac{1}{5} + \frac{2}{10} + \frac{12}{20} = 1.$$

A kapacitás másik taszkhoz történő áthelyezése azzal is jár, hogy az áthelyezett kapacitás a befogadó taszk prioritásán használható fel.



Ez a blokk helyesen eggyel jobbra helyezendő!

Lásd: a **11.** időpillanatban kért **2** időegységnyi kapacitás első fele a $\boldsymbol{\tau_1}$ taszknál lelhető fel, a második fele pedig a $\boldsymbol{\tau_2}$ taszknál. Az első fél futását követően a $\boldsymbol{\tau_1}$ taszk fut tovább, majd csak annak lefutása után áll rendelkezésre a $\boldsymbol{\tau_2}$ taszk prioritásán elérhető második fél. Itt egy periódusnyi szerver kapacitás "halmozódik fel" a $\boldsymbol{\tau_2}$ 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 taszk **periódusnyira** jelenik meg a szerver kapacitás.

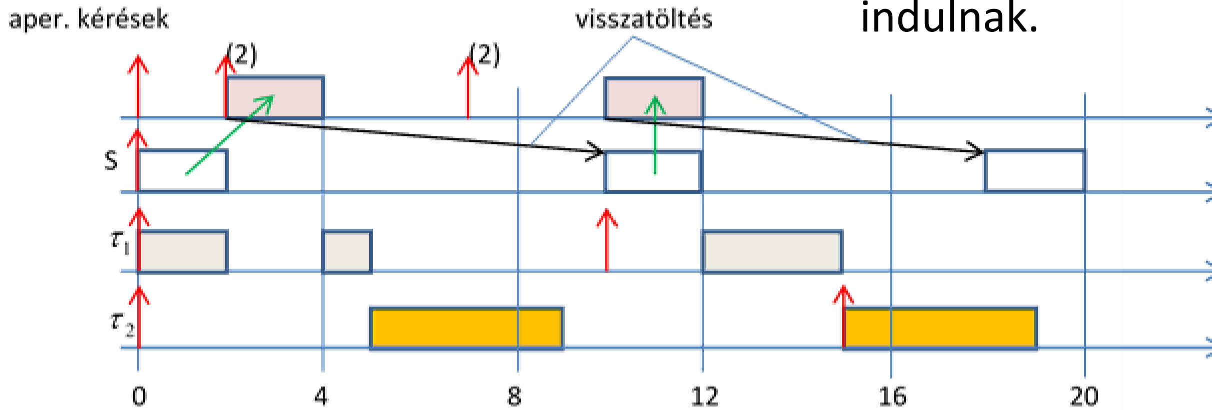
Példa: T_S=8, C_S=2. Az ezen kívül ütemezendő task-ok adatait az alábbi táblázat tartalmazza:

	C	T
τ_1	3	10
τ_2	4	15

T_S=8, C_S=2. A szerver taszknak a legmagasabb prioritása. A taszkok egyidejűleg indulnak.

Emlékeztető

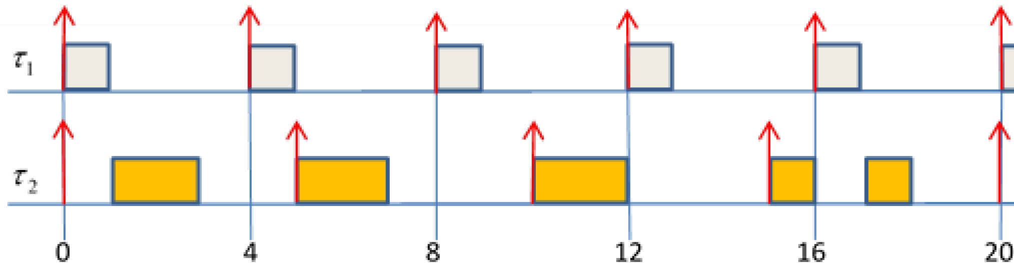
	C	T
τ_1	3	10
τ_2	4	15



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.

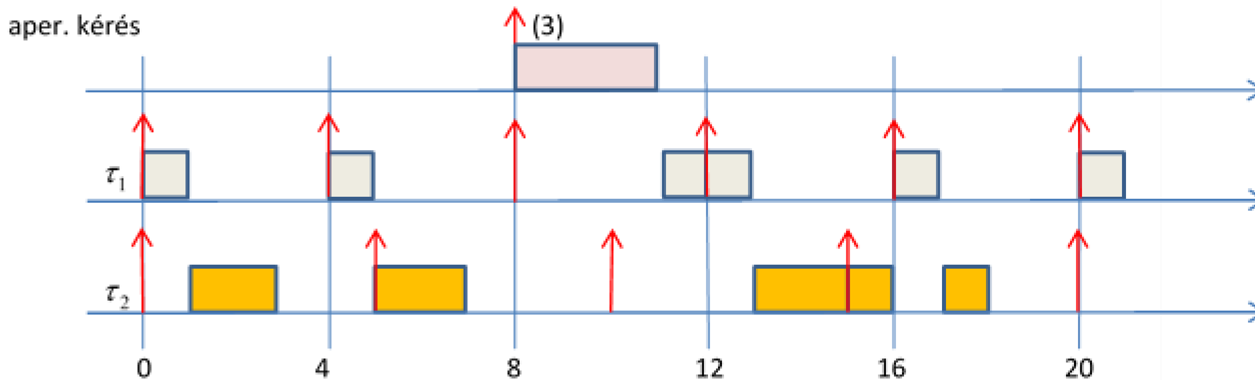
Példa: A normál ütemezés **RM** szerint:

	C	T
τ_1	1	4
τ_2	2	5



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:



Emlékeztető

Dual Priority Scheduling: Három prioritási szint van: **alacsony, közepes** és **magas**. Kezdetben **a kemény valós idejű** taszkok az **alacsony** prioritáson futnak!

A **puha valós idejű** taszkok és az aperiodikus taszkok a **közepes** prioritási szintre kerülnek.

A **kemény valós idejű** taszkok 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. Ezek **fix prioritású** szerverek. Az **EDF** ütemezési stratégiára is alapozhatók szerverek. Ezek **dinamikus prioritású** szerverek.

Total Bandwidth Server (TBS): A módszer lényege, hogy minden **aperiodikus** kéréshez egy lehetséges **korábbi határidőt** rendelünk oly módon, hogy az aperiodikus terhelés teljes processzor használata **sosem halad meg** egy előre specifikált $\boldsymbol{\mu_S}$ értéket.

A módszer elnevezése arra vezethető vissza, hogy minden esetben, amikor egy aperiodikus kérés érkezik, amennyiben lehetséges, a **szerver teljes sávszélességét** ($\boldsymbol{\mu_S}$) a kéréshez rendeljük. Ha egy k-adik aperiodikus kérés érkezik a $t = r_k$ időpontban, akkor a kéréshez a következő határidőt rendeljük:

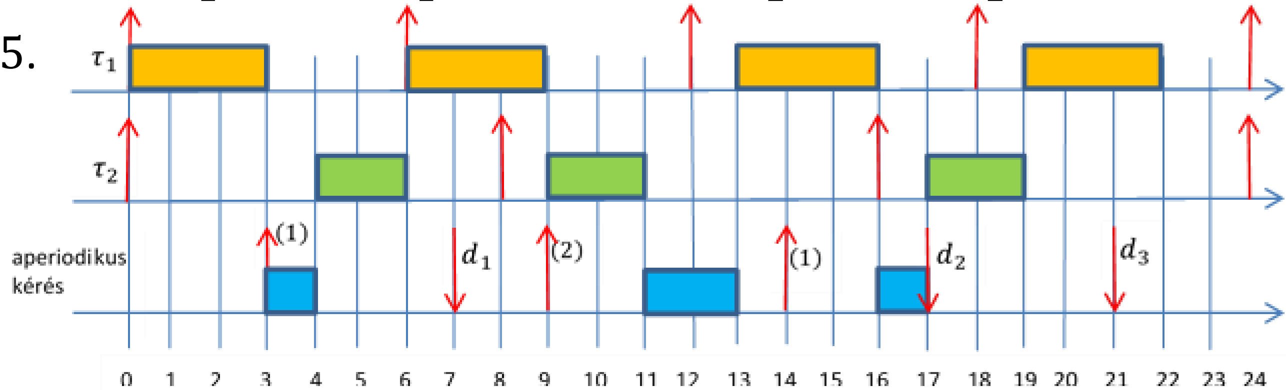
$$d_k = max(r_k, d_{k-1}) + \frac{C_k}{\mu_S}$$

ahol C_k a kérés végrehajtási ideje. Definíció szerint $d_0 = 0$. Az előző kéréshez rendelt sávszélesség a d_{k-1} határidőn keresztül jut érvényre. Valahányszor egy ilyen határidő hozzárendelés megtörténik, a kérést beillesztjük a futtatandó taszkok várakozó sorába, és ezáltal az ugyanúgy ütemezésre kerül az EDF algoritmus szerint, mint a periodikus taszkok.

A megvalósítás többlet processzoridő igénye gyakorlatilag elhanyagolható.

Példa: Két periodikus taszkunk van: $T_1 = 6ms, C_1 = 3ms$, illetve $T_2 = 8ms, C_2 = 2ms$.

Ezzel $\mu_P = 0.75$ és $\mu_S = 0.25$.



Az első aperiodikus kérés $\boldsymbol{t = 3ms}$ időpontban érkezik,

amihez határidőként

$\boldsymbol{d_1} = r_1 + C_{a1}/\mu_S = (3 + 1/0.25)ms = \boldsymbol{7ms}$-ot rendelünk.

Mivel ez a legközelebbi határidő, az aperiodikus kérés azonnal végrehajtódik. A 2. kéréshez, amelyik $\boldsymbol{t = 9ms}$ időpontban érkezik, $d_2 = r_2 + C_{a2}/\mu_S = (9 + 2/0.25)ms = \boldsymbol{17ms}$-ot rendelünk. Ez a kérés azonban nem hajtódik végre azonnal, mert a τ_2 taszknak közelebbi a határideje: $\boldsymbol{16\ ms}$. Végül a harmadik aperiodikus kérés $\boldsymbol{t = 14ms}$ időpontban érkezik, amely $d_3 = max(r_3, d_2) + C_{a3}/\mu_S = (17 + 1/0.25)ms = \boldsymbol{21ms}$ határidőt kap.

A harmadik aperiodikus kérés nem hajtódik végre azonnal, mert a τ_1 taszknak közelebbi a határideje: $\boldsymbol{18\ ms}$.

Bizonyítható, hogy ha a periodikus taszkok processzor kihasználtsági tényezője $\boldsymbol{\mu_P}$, a **Total Bandwidth Server**-é pedig $\boldsymbol{\mu_S}$, akkor ez a taszk készlet az EDF algoritmussal akkor és csak akkor ütemezhető, ha $\mu_P + \mu_S \leq 1.$ **Bizonyítás:**

Minden $[t_1, t_2]$ intervallumban, ha $\boldsymbol{C_a}$ azon aperiodikus kérések összes számítási ideje, amelyek $\boldsymbol{t_1}$-ben vagy azt követően érkeztek, és kiszolgálásra kerültek $\boldsymbol{t_2}$ vagy azt megelőző határidőre, akkor

$C_a \leq (t_2 - t_1)\mu_S$, mert

$$C_a = \sum_{k=k_1}^{k_2} C_{ak} = \mu_S \sum_{k=k_1}^{k_2} \left(d_k - max(r_k, d_{k-1})\right) \leq \mu_S \left(d_{k_2} - max\left(r_{k_1}, d_{k_1-1}\right)\right) \leq \mu_S(t_2 - t_1).$$

Ezt követően a bizonyítás a tisztán periodikus eset bizonyítását követi.

Példák további dinamikus prioritású szerverekre felsorolásszerűen:

Dynamic Priority Exchange Server

Dynamic Sporadic Server

Earliest Deadline Late Server

+ különféle módosításaik

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

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

elégséges ütemezhetőségi feltétel, de ez nem szükséges, **pesszimisztikus**.

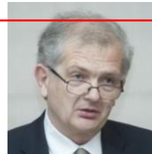
Kevésbé pesszimisztikus, ha egyidejű indítást feltételezve minden task-ra megvizsgáljuk a $C_i + I_i \leq D_i$ feltétel teljesülését.

Itt $I_i = \sum_{k=1}^{n} \left\lceil \frac{D_i}{T_k} \right\rceil C_k$.

Ez a feltétel is **elégséges**, de **nem szükséges**.

A **szükséges és elégséges** feltétel:

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



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 taszk processzor igénye a $t+L$ időpontig vagy azt megelőzően befejezendő feladatokhoz szükséges processzor idő.

Olyan periodikus taszkok 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 taszk-készlet **akkor és csak akkor** ütemezhető EDF algoritmussal, ha **minden** $L > 0$ esetében

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

Bizonyítás: Egyrészt, mivel $\mu = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$ ezért

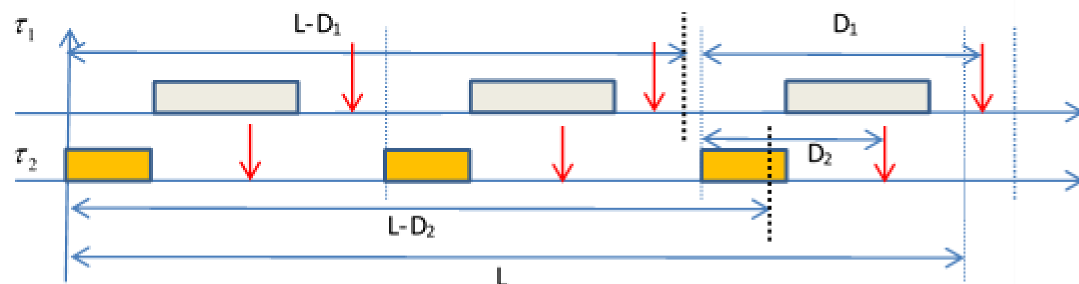
$$L \geq \mu L = \sum_{k=1}^{n} \left(\frac{L}{T_k} \right) C_k \geq \sum_{k=1}^{n} \left\lfloor \frac{L}{T_k} \right\rfloor C_k$$

Másrészt, ha $\mu > 1$, akkor van olyan $L > 0$, amelyre (*) nem áll fent, ugyanis például L-et a $T_1, T_2, \ldots, T_n$ szorzatára 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.

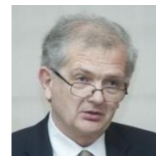
Ehhez tekintsük a következő ábrán két taszk esetét, melyek az egyszerűség kedvéért legyenek azonos periodicitásúak, de eltérő határidejűek:

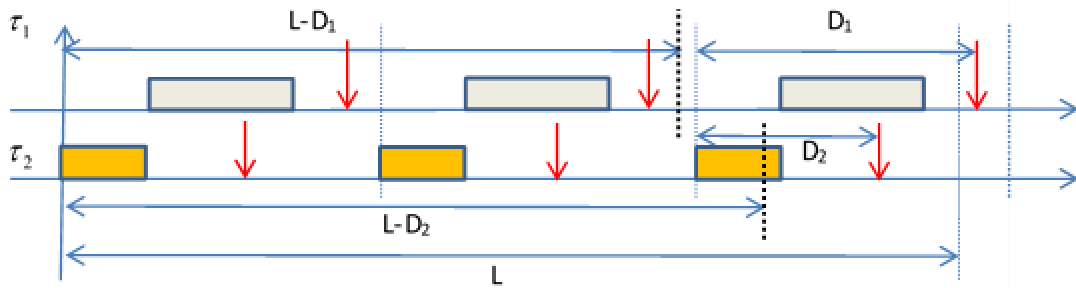


?

$$C_1(0,L) = \left\lfloor \frac{L}{T_1} \right\rfloor C_1$$

$$C_2(0,L) = \left(\left\lfloor \frac{L}{T_2} \right\rfloor + 1 \right) C_2$$





$$C_1(0,L) = \left\lfloor \frac{L}{T_1} \right\rfloor C_1$$

$$C_2(0,L) = \left(\left\lfloor \frac{L}{T_2} \right\rfloor + 1 \right) C_2$$

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

Kiegészítések a válaszidő képletéhez:

1. Kooperatív ütemezés:

Egy taszk futásának adott pontján szempont lehet a taszk futás **mielőbbi** befejezése.
Ennek eszköze **a preempció**/futás megszakítás **tiltása** a taszk 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 **többletfutást** igénylő hibakezelés:
C_i^f extra számítási idő minden taszk esetében: Egyetlen hibára:

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

Figyeljük meg: hep_i !

F hibára:

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

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

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

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 taszkot futó állapotba helyez, akkor **először** a processzor regisztereiben lévő tartalmakat **menteni kell**, majd a processzor regisztereibe **bele kell írni** a taszk **futtatási környezetét** megadó értékeket, és csak utána futtatható a kód.

A **válaszidő** tehát növelendő a taszk környezet "kapcsolási" (**context switch**) idejével.

A taszk futását megszakító magasabb prioritású taszkok futtatásakor is **váltani kell a futtatási környezetet**, ezért a magasabb prioritású taszkok számítási idejéhez hozzá kell adni **átkapcsolás** és a **visszakapcsolás** időigényét.

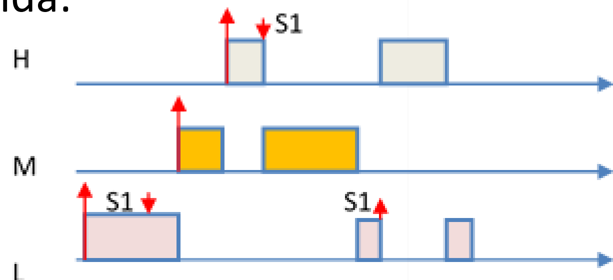
Ütemezés nem független task-ok esetén

Az ún. time-sharing rendszerek kivételével, ahol egymástól független felhasználók osztoznak a számítógép processzor kapacitásán, az alkalmazások túlnyomó többsége azzal jellemezhető, hogy a taszkok futása egymástól nem teljesen független:

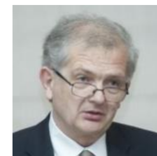
a taszkok egymással kommunikálnak,
egymással adatot cserélnek,
egymás számítási eredményeire várnak,
közös erőforrást használnak, stb.

ezért előfordulhat, hogy magasabb prioritású futását alacsonyabb prioritású akadályozza (blokkolja).

Példa:

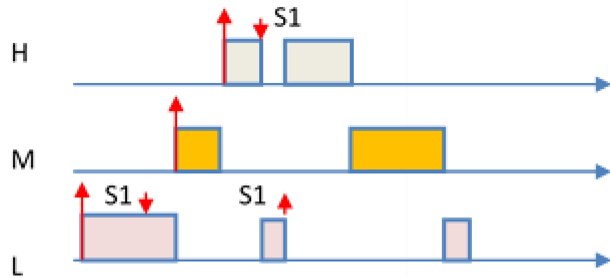


A jelenséget ***prioritás inverziónak*** nevezzük, mert látszólag az M és a H taszkok prioritásai felcserélődnek.



A prioritás öröklés algoritmus (Priority Inheritance Protocol, PIP):

A prioritás inverzió elkerülése úgy lehetséges, hogy a H taszk **kritikus szakaszba** lépési szándékának megjelenésekor az L taszk ideiglenesen **„megörökli”** a H taszk prioritását (dinamikus prioritás), hogy mielőbb fejezze be a kritikus szakaszbeli teendőit, majd ezt követően **visszatér** az eredeti (statikus) prioritási rend.

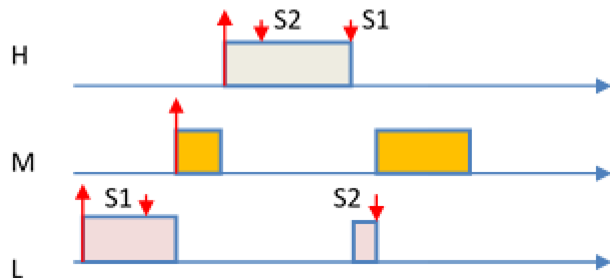


A H task válaszideje lényegesen csökken, a **blokkolási idő** a legkedvezőtlenebb esetben az L task kritikus szakaszban töltött idejével egyenlő.

A **blokkolási idő** (B_i) figyelembevétele válaszidő számításnál:

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

Több közös erőforrás/kritikus szakasz egyidejű működtetésénél felmerülhet a **holtpont** (deadlock) problémája, azaz a **kölcsönös egymásra várás** esete, ami – a szemaforok konkrét implementációjától függően – a program lefagyását is eredményezheti.



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 előbb be kell fejeznie a kritikus szakaszban lévő kódrészének futtatását.