Beágyazott információs rendszerek

2. Ütemezés (folyt.)

2020. október 7.

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.

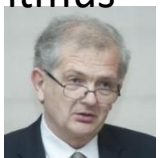
Total Bandwidth Server (TBS): A módszer lényege, hogy minden **aperiodikus** kéréshez egy lehetséges **korábbi határidőt** rendelünk oly módon, hogy az aperiodikus terhelés teljes processzor használata **sosem halad meg** egy előre specifikált μ_S értéket.

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

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

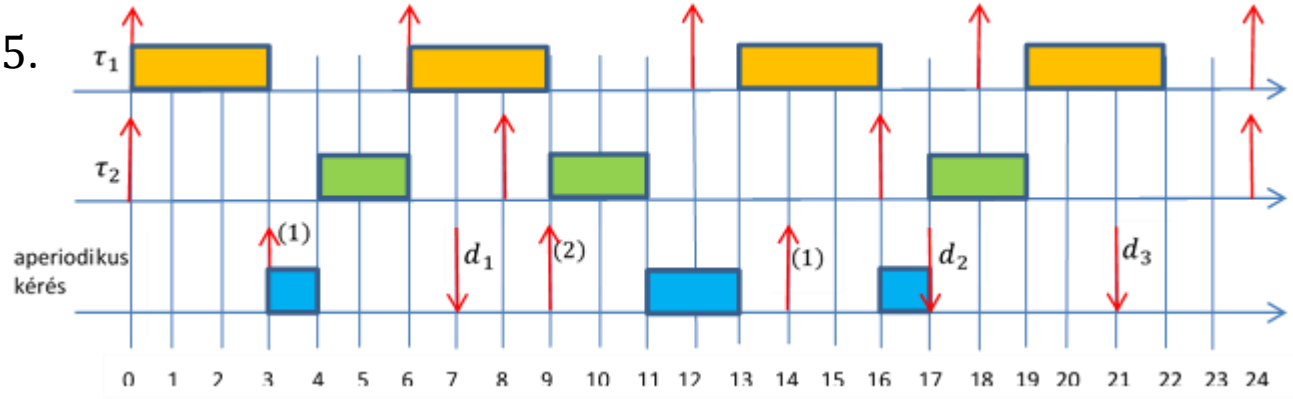
ahol C_k a kérés végrehajtási ideje. Definíció szerint $d_0 = 0$.
Az előző kéréshez rendelt sávszélesség a d_{k-1} határidőn keresztül jut érvényre. Valahányszor egy ilyen határidő hozzárendelés megtörténik, a kérést beillesztjük a futtatandó taszkok várakozó sorába, és ezáltal az ugyanúgy ütemezésre 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 $T_2 = 8ms, C_2 =$

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 számításideje, amelyek $\boldsymbol{t_1}$-ben vagy azt követően érkeztek, és kiszolgálásra kerültek $\boldsymbol{t_2}$ vagy azt megelőző határidőre, akkor

Emlékeztető

$C_a \leq (t_2 - t_1)\mu_S$, 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**.

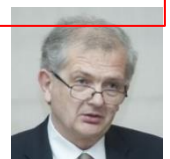
Kevésbé pesszimisztikus, ha egyidejű indítást feltételezve minden task-ra megvizsgáljuk a $C_i + I_i \leq D_i$ feltétel teljesülését.

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:

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



Emlékeztető

Ha az EDF algoritmust $D_i < T_i$ mellett használjuk, akkor közvetlenül a processzor k 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

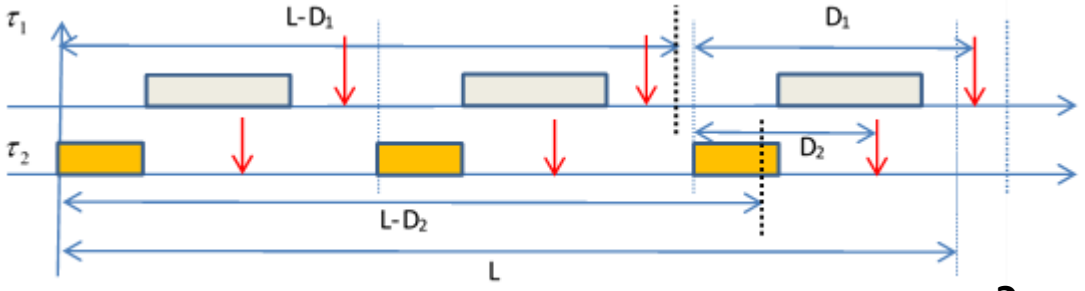
$$L \geq \mu L = \sum_{k=1}^{n} \left(\frac{L}{T_k} \right) C_k \geq \sum_{k=1}^{n} \left\lfloor \frac{L}{T_k} \right\rfloor C_k$$

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.

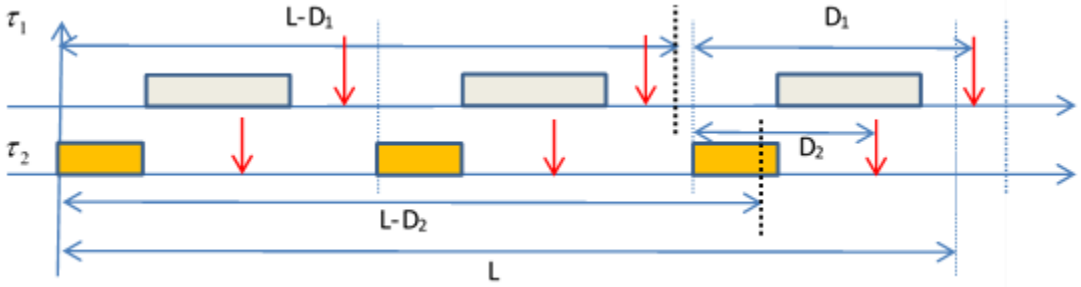
Ehhez tekintsük a következő ábrán két taszk esetét, melyek az egyszerűség kedvéért legyenek azonos periodicitásúak, de eltérő határidejűek:



?

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

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



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

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

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 $\forall L > 0 \;\; L \geq \sum_{k=1}^{n} \left(\left\lfloor \frac{L - D_k}{T_k} \right\rfloor + 1 \right) C_k$

Emlékeztető

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:

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

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

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

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**.

Emlékeztető

Ha az ütemező egy taszkot futó állapotba helyez, akkor **először** a processzor regisz 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.

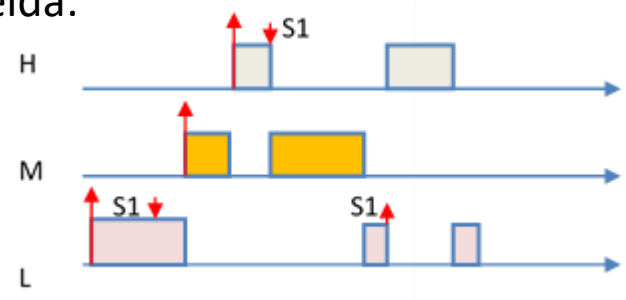
Ütemezés nem független taszkok esetén

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:

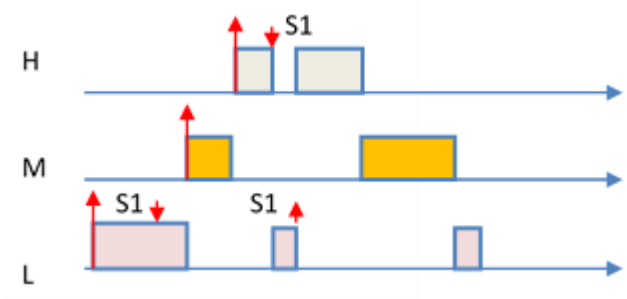


A jelenséget ***prioritás inverziónak*** nevezzük, mert látszólag az M és a H taszkok prioritásai felcserélődnek.

Emlékeztető

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.

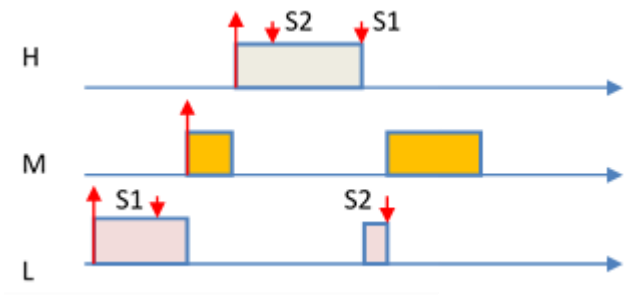


A H task válaszideje lényegesen csökken, a **blokkolási idő** a legkedvezőtlenebb esetben az L task kritikus szakaszban töltött idejével egyenlő.

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.

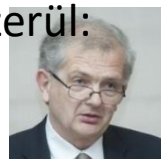


Az L taszk az **S1** szemaforral védett **kritikus szakaszba** kerül.

Az L taszk a kritikus szakaszon belül egy további, az **S2** szemaforral védett erőforráshoz fog fordulni.

Ezt az erőforrást a H taszk – az ábrán látható időviszonyok mellett – ugyancsak használja.

Amikor a H taszk az S1 szemaforral védett erőforráshoz fordul, akkor blokkolódni kényszerül: az L taszknak előbb be kell fejeznie a kritikus szakaszban lévő kódrészének futtatását.



Azonban az **S2** szemaforhoz fordulva kialakul az **egymásra várás**, az ún. **holtpont** (deadlock).

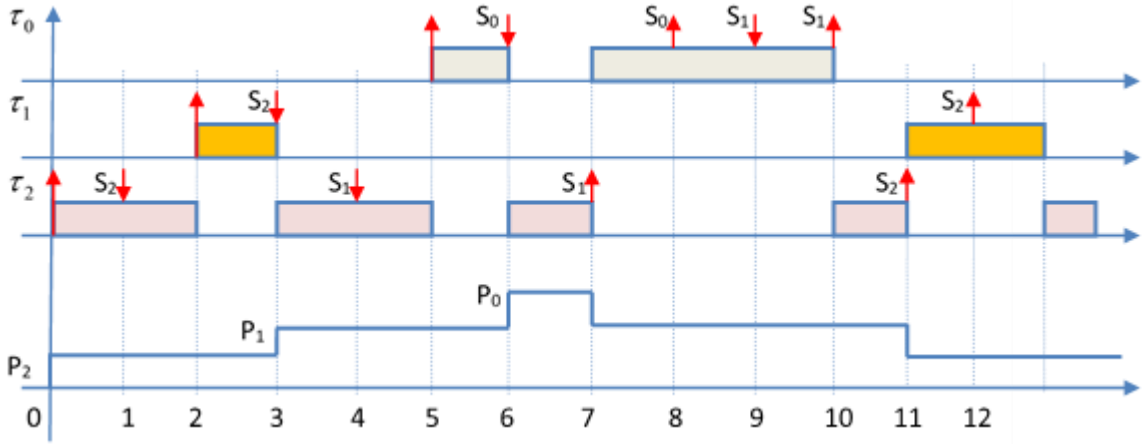
Ennek megakadályozására dolgozták ki a prioritás felső-határ/plafon (ceiling) protokollokat.

Prioritás felső-határ (plafon) protokoll (Priority Ceiling Protocol, PCP):

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

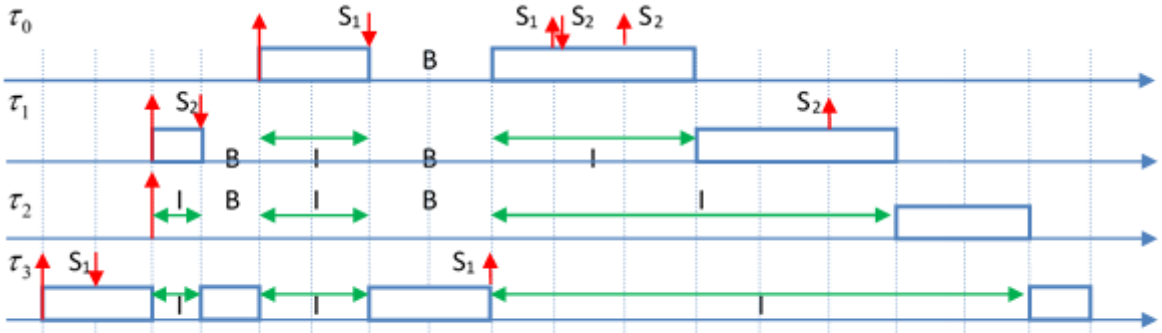
Az **első erőforrás** lefoglalása engedélyezett.
A protokoll hatása az, hogy egy **második erőforrás** lefoglalása csak akkor lehetséges, ha nincsen magasabb prioritású taszk, amely mind a két erőforrást használja.
Ebből következik, hogy a leghosszabb idő, amivel egy taszk blokkolható, egyenlő az alacsonyabb prioritású taszkokban a leghosszabb kritikus szakasz végrehajtási idejével.

Példa: A futtatandó taszkok csökkenő prioritású sorrendben: $\boldsymbol{\tau_0, \tau_1, \tau_2}$. A prioritásaik: P_0, P_1 és P_2. Az erőforrásokat S_0, S_1 és S_2 szemaforok őrzik. Prioritás plafonjaik: $C(S_0) = P_0$, $C(S_1) = P_0$, $C(S_2) = P_1$.



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

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

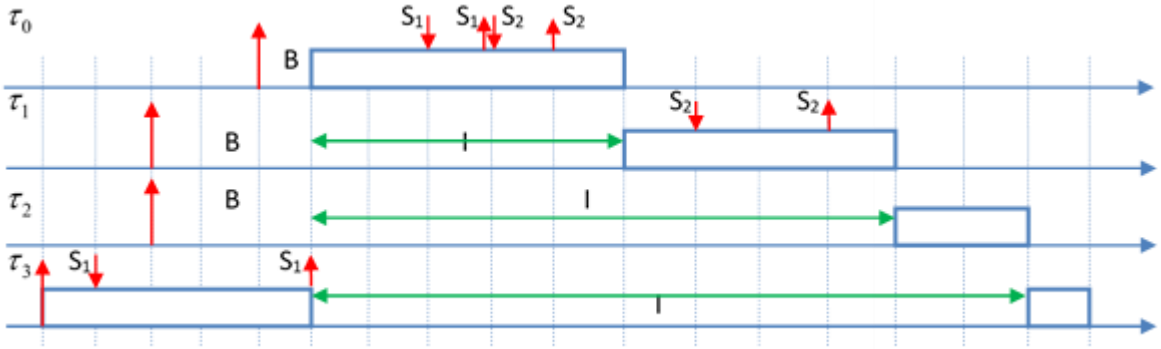


Az ábrán nyomon követhető a protokoll működése.
I az interferencia intervallumokat, B-vel pedig a blokkolási intervallumokat jelöli.

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

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

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



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

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

A szemaforokat nem kell implementálni, mert mindig szabad állapotúak!

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

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

3. Memória menedzsment

A beágyazott rendszerek jelentős részénél **nem számíthatunk** arra, hogy az eszköz időről-időre alaphelyzetbe kerül (reset-elődik), és a programfutások **káros mellékhatásai** ezzel **eliminálódnak**.

Ezért minden esetben **úgy kell terveznünk**, hogy az alkalmazás futásával párhuzamosan az **erőforrások** teljesítőképessége **ne degradálódjon**.

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

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

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

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

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

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

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

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

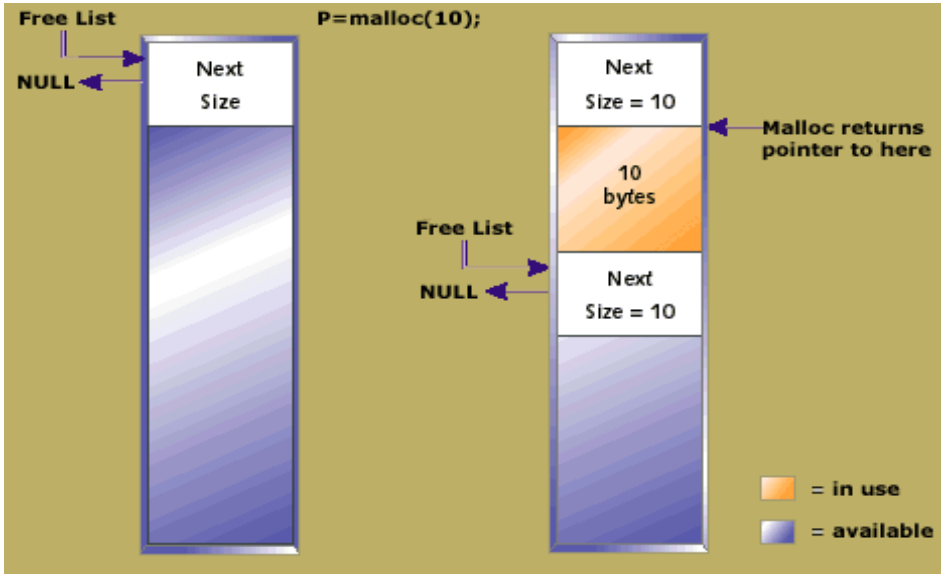
amely a következők miatt jöhet létre: a kódolás egy adott pontján **a programozó elbizonytalanodhat**, vajon egy adott memória blokkra szükség van-e még?

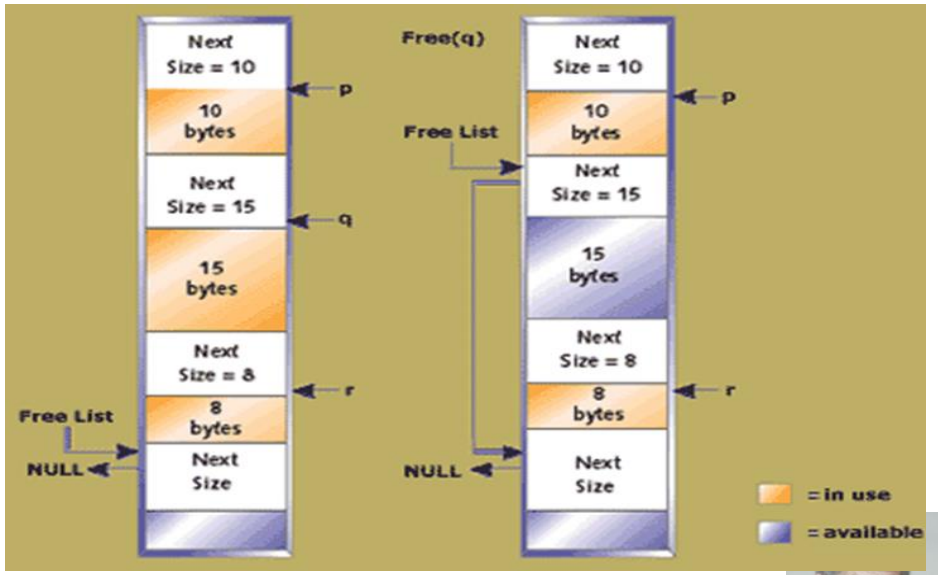
Ha **felszabadítja**, de továbbra is használja, például egy, az ugyanarra a blokkra mutató második pointer segítségével, akkor a program jól működhet mindaddig, amíg az adott memória területet a program egy másik része le nem foglalja.

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.**





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. Lényegében 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).

3) az inicializálást követően a $salloc( )$ tiltva van.

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

Multitasking: Minden taszknak 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 taszknak 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 taszk által foglalt memóriát a másiknak kell felszabadítania.

(4) ha a taszkok között memória tartalmakat mozgatunk, akkor jó tudni, hogy aktuálisan melyik taszk birtokolja a memóriát.

(5) közös **heap** esetén is javasolható a taszkonkénti 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.

(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.