SZAKDOLGOZAT

Morsiani Renato

Debrecen
2009

Debreceni Egyetem

Informatikai Kar

PROCESSZORÜTEMEZÉS TÖBBPROCESSZOROS RENDSZEREKBEN

Szakdolgozat

Témavezető
Dr. Fazekas Gábor
PhD. egyetemi docens

Készítette
Morsiani Renato
Programtervező informatikus (Bsc)

Debrecen
2009

Tartalomjegyzék

Bevezetés

Az ütemezési teóriák szűk keresztmetszetet alkotó erőforrásoknak (pl. processzorok, gépek, robotok, stb.), tevékenységek között való optimális elosztásával foglalkozik. Célja egy vagy több teljesítményjellemző optimalizálása. Az ütemezések vizsgálatának története egészen ötven évvel ezelőttig nyúlik vissza, Johnson (1954) és Bellman (1956) alapokat teremtő munkásságáig. Azóta az ágazat jelentős fejlődésen ment keresztül. Ennek eredményeképp számos különféle ütemezési modell és optimalizációs technika került kifejlesztésre, melyeket az iparban, kommunikációban és egyéb területeken alkalmaznak sikerrel. Mára az ütemezési teóriák feladatköre jelentős, általánosan elismert, gyorsan fejlődő területe a kutatásnak.

Napjainkban a többmagos rendszerek megjelenésével már nem csak a nagyvállalatok többprocesszoros kiszolgáló parkjaiban találkozunk a hatékony ütemezés kérdésével, hanem akár a saját íróasztalunkon is.

Dolgozatom célja ezen szerteágazó és jelentős méretű terület főbb irányvonalainak, illetve néhány eredményének bemutatása.

Ennek első lépesként az első részben a klasszikus multiprocesszoros ütemezési eljárások számítástudományi és elméleti vetületeit fejtem ki részletesebben. Az egyszerű ütemezéstől kezdve a preemptív és precedencia megszorításokkal terhelt környezeteken keresztül a valós idejűség kérdéséig eljutva, egységes jelölésrendszer alkalmazásával téve ezt.

Ebből kiindulva a második részben a valós idejű rendszerek problémaköre kerül terítékre. Az ilyen fajta rendszerek napjainkban meglehetősen jelentős szerepet töltenek be, ehhez elég csak az atomerőművek rendszereit vezérlő számítógépekre, vagy a jövő intelligens autópályáit irányító bonyolult mechanizmusokra gondolnunk. Meglehetősen nehéz feladat olyan hatékony algoritmusokat találni, amelyek a szoros és sok esetben nagyon szigorú időbeli elvárásoknak eleget tudnak tenni.

A klasszikus ütemezési terület és a valós idejű rendszerek témaköre is meglehetősen jól fejlett és erős megoldásokkal bír egyprocesszoros rendszerek tekintetében. Multiprocesszoros rendszerek esetben azonban a kérdés hirtelen sokkal összetettebbé válik. A több végrehajtó egység adta végtelen sok lehetőséget ki kell tudnunk használni.

1. Elméleti modellek

1.1 Alapfogalmak

Tegyük fel, hogy *n* feladat (job, task) J_j (j=1,…, n) végrehajtása szükséges *m* párhuzamos gépen vagy processzoron M_i (i = 1,…,*m*). Minden gép legfeljebb egy feladat kezelésére képes egy időben. Minden feladat végrehajtható a gépek bármelyikén.

Az *ütemezés* minden feladatra vonatkozólag, egy vagy több időintervallum elosztása egy vagy több végrehajtó egység között. Az ilyen ütemezéseket úgynevezett *Gantt grafikon*okon ábrázolhatjuk. Ezek a grafikonok lehetnek gép-orientáltak vagy feladat-orientáltak. Az alábbiakban az előbbi módon kerülnek ábrázolásra az ütemezések.

A szóban forgó ütemezési probléma célja, bizonyos feltételeknek eleget tevő ütemezés megtalálása.

Egy ütemezés *megvalósítható (feasible)*, ha:

- Nincs két időintervallum, mely egyazon gépen átfedné egymást.
- Nincs két időintervallum kiosztva egyazon feladat számára.
- Megfelel bizonyos probléma specifikus osztályozásoknak, amelyek az alábbiakban lesznek definiálva.

Egy ütemezés *optimális*, ha minimalizálja a megadott optimalitási feltételeket (lásd lentebb).

A probléma jellemezhető egy három-mezős osztályozással:

Az első mező – $\alpha=\alpha_1\alpha_2$ – részletezi a gépi környezetet. p_{ij} jelölje az időt, amely J_j végrehajtásához szükséges az M_i gépen. Az α_1 három lehetséges értéke:

- *P (identikus gépek):*

  $p_{ij} = p_j$

  J_j végrehajtási ideje megegyezik minden M_i-n.
- *Q (uniform gépek):*

  $p_{ij} = p_j / s_i$

  J_j végrehajtási ideje az M_i-n megegyezik a J_j végrehajtási igényének (p_j) és az M_i gép s_i sebességének hányadosával.
- *R (össze nem függő gépek):*

  p_{ij} tetszőleges.

Ha α_2 pozitív egész szám, akkor *m* konstans és megegyezik α_2-vel. Ha α_2 üres, akkor *m* változó.

A második mező – β – jelöl bizonyos feladat jellemzőket. Ha β üres, az a következőket jelenti:

- minden p_{ij} (vagy p_j) tetszőleges, nem negatív egész szám
- nincs *precedencia megszorítás* a feladatok között
- nincs *preemtivitás* (többfeladatos időszeletes ütemezés) engedélyezve
- minden feladat végrehajtásra elérhetővé válik a 0. időpillanatban

A harmadik mező – γ – felel meg a választott *optimalitási feltételnek*. Minden megvalósítható ütemezés definiál *befejeződési időt (completion time)*, C_j-t a J_j-re vonatkozóan (j=1,…,n). Kétféle kritérium minimalizásáról lehet szó:

- *maximális befejeződési idő* (más néven szűk keresztmetszet - bottleneck):
  C_{max}=max{C_1,…,C_n}

- *teljes befejeződési idő*:
  $\sum C_j = C_1 + \cdots + C_n$
  Ha szükséges súlyozottan is tekinthetjük ezeket az értékeket. Ennek jelölése: $\sum w_j C_j$

Az γ optimális értékének jelölése: γ^*

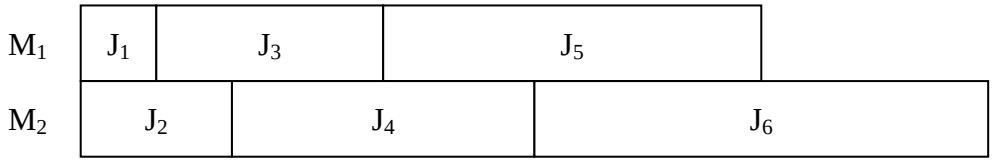
Példák:

P2 | | $\sum C_j$

Probléma: teljes befejeződési idő minimalizálása két identikus gépen.

n = 6; $p_j = j$ (j=1,…,6)

Optimális ütemezés:



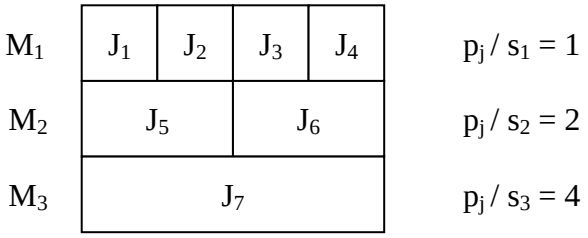
$\sum C_j^* = 34$

Q3 | | C_{max}

Probléma: maximális befejeződési idő minimalizálása három uniform gép esetén

$s_1 = 4$, $s_2 = 2$, $s_3 = 1$; $n = 7$; $p_j=4$ (j=1,…,7)

Optimális ütemezés:



$C_{max}^* = 4$

R | | C_{max}

Probléma: a maximális befejeződési idő minimalizálása m össze nem függő gépen

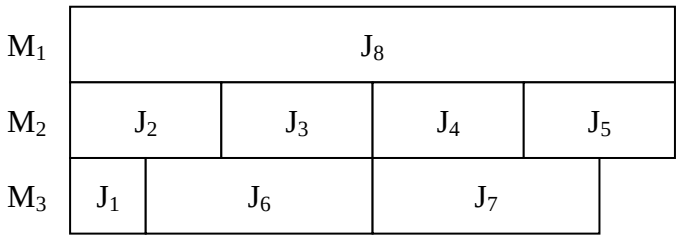
$m = 3$; $n = 8$

$p_{11} = 1$, $p_{1j} = 1$ (j=2,…,7), $p_{18} = 8$,

$p_{21} = 1$, $p_{2j} = 2$ (j=2,…,7), $p_{28} = 9$,

$p_{31} = 1$, $p_{3j} = 3$ (j=2,…,7), $p_{38} = 9$.

Optimális ütemezés:



$C_{max}^* = 8$

1.2 Alapvető módszerek

A P | | $\sum C_j$, és általánosabb változata az R | | $\sum C_j$ problémák az egyedüli non preemptív problémák, amelyek bizonyítottan megoldhatóak polinomiális időben. A P2 | | C_{max} és a P2 | | $\sum w_i C_j$ problémák NP-nehezek. Minden probléma, amely identikus gépekkel kapcsolatos és NP-nehéz, NP-nehéz marad akkor is, ha a gépeket uniform gépekre cseréljük.

A *Shortest Processing Time* (legkisebb végrehajtási idő) szabály megoldja a P | | $\sum C_j$ problémát O (n log n) időben, a következő módon:

Feltételezzük, hogy $n = \ell m$ (ha nem így van, akkor 0 végrehajtási idejű többlet feladatok hozzáadása szükséges). Az alábbiaknak megfelelően újraszámozzuk a feladatokat:

$$p_1 \leq \ldots \leq p_n$$

Ütemezzük a feladatokat a következőképpen:

$J_{(k-1)m+1}$, $J_{(k-1)m+2}$, …, J_{km} az m gép k-dik pozíciójába. (k=1,…,ℓ)

Ezen algoritmus általánosítható úgy, hogy a Q| |∑ C_j problémát is megoldja O (n log n) időben.

Az R| | ∑ C_j probléma legáltalánosabb esete formalizálható és megoldható, mint egy ***lineáris szállítási feladat*** O (n^3) időben. Valamint tovább alakítható hozzárendelési feladattá is.

Legyen:

$$x_{ijk} = \begin{cases} 1\ , ha\ J_j\ az\ M_i\ gép\ ütemezésének\ k-dik\ pozíciójában\ áll \\ 0 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad , egyébként \end{cases}$$

A minimalizálandó probléma:

$$\sum_{i=1}^{m}\sum_{j=1}^{n}\sum_{k=1}^{n} kp_{ij}\ x_{ijk}$$

Feltételrendszer:

$\sum_{i=1}^{m}\sum_{k=1}^{n} x_{ijk} = 1\ (j = 1, \ldots, n)$ – Egy job pontosan egy helyen.

$\sum_{j=1}^{n} x_{ijk} \leq 1\ \ (i = 1, \ldots, m; k = 1, \ldots, n)$ – Egy job legfeljebb egyszer.

$x_{ijk} \leq 0\ (i = 1, \ldots, m; j = 1, \ldots, n; k = 1, \ldots, n)$ – Nem negativitási feltétel.

Így a ∑ C_j minimalizációja polinomiális időt igényel, még m össze nem függő gép esetén is. Ezzel szemben a C_{max} minimalizációja NP-nehéz, még két identikus gépen is.

Nem hagyható figyelmen kívül, hogy e problémák optimalizációs algoritmusai leszámlálható természetűek. Egy általános ***dinamikus programozási séma*** széles körben alkalmazható. A P| |C_{max} probléma esetén e séma a következő:

Legyen:

$$B_j(t_1, \ldots, t_m) = \begin{cases} igaz, ha\ J_1, \ldots, J_j\ ütemezhető\ M_1, \ldots, M_m;\ \ M_i\ foglalt\ 0 - t_i (i = 1, \ldots, m) \\ hamis \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad , egyébként \end{cases}$$

$$B_0(t_1, \ldots, t_m) = \begin{cases} \ \ igaz,\ \ ha\ t_i = 0\ (i = 1, \ldots, m) \\ hamis, \qquad\qquad\qquad egyébként \end{cases}$$

Ekkor a rekurzív egyenlet:

$B_j(t_1, \dots, t_m) = \bigvee_{i=1}^{m} B_{j-1}(t_1, \dots, t_{i-1}, t_i - p_j, t_{i+1}, \dots, t_m)$

Legyen C a C_{max}^* optimális érték felső korlátja. Számoljuk ki j=0,1,…,n esetén a B_j=(t_1,…,t_m) értéket, ahol t_i=0,1,…,c (i=1,…,m), és határozzuk meg:

$$C_{max}^* = \min\{\max\{t_1,\dots,t_m\} \mid B_n(t_1,\dots,t_m) = \text{true}\}$$

Ez a folyamat megoldja P| |C_{max} O (nC^m) időben. A C nagy értékei esetén a *branch-and-bound* módszer jól alkalmazható.

A P| |C_{max} probléma NP-nehéz volta igazolttá teszi gyors közelítő algoritmusok alkalmazását. Ezen típusú megközelítés egyik legkorábbi eredménye a P| |C_{max} probléma *list scheduling* (LS) *(listaütemezés)* eszközzel történő megoldásával kapcsolatos. Ebben az esetben a feladatok egy priorizált listája adott, és minden lépésben az első elérhető gép kiválasztásra kerül a listában elsőként elérhetővé váló feladat végrehajtására.
Ennek egyik típusa a *longest processing time* (LPT) *(leghosszabb végrehajtási idő)* szabályt alkalmazza. Eszerint a lista a nem növekvő p_j-nek megfelelő sorrendben tartalmazza a feladatokat.

1.3 Preemptivitás

Egy újabb szempontból is módosítjuk az előzőekben felvázolt multiprocesszoros ütemezési modellt. Feltesszük, hogy korlátlan mértékű preemptivitás engedélyezett. Ez az jelenti, hogy bármely feladat végrehajtása tetszőleges gyakorisággal megszakítható és újraindítható egy másik végrehajtó egységen azonnal, vagy egy későbbi időben bármely végrehajtó egységen.
Jelölés: pmtn.
Belátható, hogy a P| pmtn| $\sum C_j$ probléma esetében a preemptivitásnak semmilyen előnye nincs a végeredmény szempontjából. Így a nem preemptív SPT szabály is képes megoldani a problémát O (n log n) időben.

Az SPT szabály egy preemptív változata képes megoldani O(n log n+mn) időben a Q| pmtn| $\sum C_j$ problémát az alábbi módon:

- Rendezzük a feladatokat az SPT szabálynak megfelelően
- Rendre ütemezzük a feladatokat preemptív módón, úgy, hogy az egyes feladatok befejeződési ideje minimális legyen.

Más szavakkal, ütemezzük az n feladatot az M_1 leggyorsabb gépre, amíg az be nem fejeződik a $t_1 = p_n / s_1$ időpillanatban. Ezután ütemezzük az $(n – 1)$ feladatot először az M_2 gépre t_1 ideig, majd az M_1 gépre t_1 időpillanattól befejeződésig, azaz $t_2 \geq t_1$ időpillanatig. Az $(n – 2)$ feladatot ütemezzük először az M_3 gépre t_1 ideig, majd az M_2 gépre $t_2 – t_1$ ideig, végül az M_1 gépre a t_2 időpillanattól befejeződésig, azaz $t_3 \geq t_2$ időpillanatig, és így tovább.

Az eredményként kapott ütemezés legfeljebb $(m-1)(n-\frac{m}{2})$ váltást (preemption) tartalmaz.

Q| pmtn| $\sum C_j$

m = 3; $s_1 = 3$, $s_2 = 2$, $s_3 = 1$; n =4;

$p_1 = 3$, $p_2 = p_3 = 8$, $p_4 = 10$

Optimális ütemezés:

M_1	J_1	J_2	J_3	J_4
M_2	J_2	J_3	J_4	
M_3	J_3	J_4		

$\sum C_j^* = 14$

A bizonyítottan NP-nehéz R| pmtn| $\sum C_j$ probléma a multiprocesszoros ütemezés tárgykörének egyik legérdekesebb problémája.

A P| pmtn| C_{max} és a Q| pmtn| C_{max} problémák megkülönböztetettek, mivel mindkét esetben létezik egy egyszerű zárt kifejezés, amely egyértelmű alsó korlátot ad C_{max}^*-ra.

A P| pmtn| C_{max} esetén:

$$C_{max}^* = max\left\{p_1, \dots, p_n, \frac{p_1 + \cdots + p_n}{m}\right\}$$

A *körülzárás (wrap-around)* szabály megoldja a problémát O(n) időben:

- Rendre töltsük fel a gépeket, bármilyen sorrendben ütemezve a feladatokat.

- Amikor az egyes feladatok esetén a fenti időlimitet elérjük, vágjuk ketté a job-ot.

Ez esetben legfeljebb (m – 1) váltás (preemption) lesz az ütemezésben.

P| pmtn| C_{max}

m = 3; n = 6; $p_j = j$ (j = 1,…,6) ⟶ $C^*_{max} = max\left\{6, \frac{21}{3}\right\} = 7$

Optimális ütemezés:

M_1	J_1	J_2	J_3	J_4
M_2	J_4		J_5	
M_3	J_5	J_6		

A Q| pmtn| C_{max} esetén:

$$C^*_{max} = max\left\{\frac{p_1}{s_1}, \frac{p_1 + p_2}{s_1 + s_2}, \dots, \frac{p_1 + \cdots + p_n}{s_1 + \cdots + s_m}\right\}$$

ahol $s_1 \geq \dots \geq s_m$ és $p_1 \geq \dots \geq p_n$. Egy bonyolult algoritmus O(n) időben megoldja ezt a problémát. Ez esetben legfeljebb 2(m – 1) váltás (preemption) lesz.

Az alábbiakban bemutatunk egy algoritmust, amely optimális megoldást ad a Q| pmtn| C_{max} problémára:

- Adott egy részleges ütemezés a t időpillanatig.
- *level* $p_j(t)$: A J_j job végrehajtási igényének azon részét jelenti, amely nem lett végrehajtva a t időpillanat előtt.
- A t időpillanatban egy assign(t) metódus kerül meghívásra, amely hozzárendeli a faladatokat a végrehajtó gépekhez.
- A gépek ezzel a hozzárendeléssel működnek egy bizonyos s (> t) időpillanatig. Ekkor új hozzárendelés történik, és a folyamat elölről kezdődik.
- *együttes végrehajtás: r* feladat együttes végrehajtása T ideig azt jelenti, hogy minden feladatot felváltva T / r ideig hajtunk végre minden kijelölt gépen.

Az algoritmus:

1. t := 0;
2. WHILE létezik job pozitív *level* taggal DO
   BEGIN
3. Assign(t);
4. t_1 := min{$s > t$ | egy job az s időpillanatban befejeződik};
5. t_2 := min{$s > t$ | létezik i, j job, melyekre: $p_i(t) > p_j(t)$ és $p_i(s) = p_j(s)$};

6. $t := \min\{t_1, t_2\}$;
END
7. Konstruáljuk meg az ütemezést.

Assign (**t**):
1. $J := \{i \mid p_i(t) > 0\}$;
2. $M := \{M_1, \ldots, M_m\}$;
3. WHILE $J \neq \emptyset$ és $M \neq \emptyset$ DO
BEGIN
4. Keressük meg a legnagyobb *level* taggal rendelkező feladatok halmazát ($I \subseteq J$);
5. $r := \min\{|M|, |I|\}$;
6. Rendeljük az *I*-beli feladatokat *együttes végrehajtás*ra az *r* leggyorsabb gépen;
7. $J := J \setminus I$;
8. Távolítsuk el az *r* leggyorsabb gépet *M*-ből
END

Az R| pmtn| C_{max} probléma megadható egy *lineáris programozási* feladatként is:

Legyen x_{ij} = a J_j által eltöltött idő az M_i gépen.

A minimalizálandó probléma: C_{max}.

A Feltételrendszer:

$\sum_{i=1}^{m} {x_{ij}}/{p_{ij}} = 1 \ (j = 1, \ldots, n)$ – minden job legyen befejezve.

$\sum_{i=1}^{m} x_{ij} \leq C_{max} \ (j = 1, \ldots, n)$ – egyik job végrehajtási ideje sem lehet nagyobb C_{max}-nál.

$\sum_{j=1}^{n} x_{ij} \leq C_{max} \ (i = 1, \ldots, m)$ – egyik gép sem dolgozhat több ideig, mint C_{max}.

$x_{ij} \leq 0 \ (i = 1, \ldots, m; j = 1, \ldots, n)$ – Nem negativitási feltétel.

Khachian bebizonyította, hogy a lineáris programozási feladatok megoldhatóak polinomiális időben. A kapott eredmény (x_{ij}^*) egy megoldható ütemezés, amelyet így szintén polinomiális időben megkonstruálhatunk. Ekkor a váltások (preemption) száma nem lesz több $\frac{7}{2}m^2$-nél.

1.3.1 A release time és due time fogalma

Kiegészíthetjük a (preemptív) ütemezési modellünket azzal a feltételezéssel, hogy J_j végrehajtásra kész állapotba csak egy bizonyos egész értékű időpontban kerülhet. Ez az időpont az *elindítási idő* (*release time*) – r_j (j=1…,n).

Ezen új tényező kapcsán háromféle algoritmust különböztethetünk meg:

- *On-line:* bármely időpontban, csak az éppen végrehajtásra kész folyamatokról kapott információk szükségesek.

- *Közelítőleg (Nearly) On-line:* A fentiek mellett a következő elindítási időket is ismernünk kell.
- *Off-line:* Minden információ már előzetesen ismert.

A P| pmtn, r_j| C_j és Q| pmtn, r_j| C_j problémák esetében nem létezik on-line algoritmus, még két identikus gép esetében sem. A P| pmtn, r_j| C_{max} probléma megoldható O(nm) időben egy on-line algoritmussal. A Q| pmtn, r_j| C_{max} probléma pedig $O(n^2)$ időben egy közelítőleg on-line algoritmussal.

További kiegészítésképp feltesszük, hogy a fentiek mellett a J_j feladatra szánt végrehajtási idő korlátozott. Be kell fejeződnie nem később, mint egy megadott *lejárati idő* (*due time*) – d_j (j=1,…,n) és $r_j \leq d_j$. Ekkor a C_{max} minimalizálásának célkitűzését felváltja egy olyan megvalósítható preemptív ütemezés létezésének keresése, amely tiszteletben tartja az elindítási és lejárati időket. Bebizonyították, hogy nem létezik közelítőleg on-line algoritmus, még két identikus gép esetében sem.
Azonban off-line algoritmusok léteznek (részletesebb algoritmus lásd: 1.3.1):

- P|pmtn, r_j,d_j| megoldható $O(n^3)$ időben egy hálózati folyam (*network flow*) számítással.
- Q|pmtn, r_j,d_j| pedig $O(n^6)$ időben egy általánosított *network flow model* segítségével.

Egyéb optimalitási feltételek is bevezethetőek ebben a problémakörben, amelyek függnek a lejárati időtől (due time).
Minden J_j job-hoz megadhatjuk az alábbiakat:

$$L_j = C_j - d_j \qquad késés$$
$$F_j = C_j - r_j \qquad áthaladási\ idő$$
$$E_j = max\{0, d_j - C_j\} \qquad koraiság$$
$$T_j = max\{0, C_j - d_j\} \qquad pontatlanság$$
$$D_j = |C_j - d_j| \qquad abszolút\ eltérés$$
$$S_j = (C_j - d_j)^2 \qquad négyzetes\ eltérés$$
$$U_j = \begin{cases} 0, ha\ C_j \leq d_j \\ 1, egyébként \end{cases} \qquad egységnyi\ büntetés$$

Ezekből előállíthatóak az új optimalitási feltételek az eddigiek mintájára (Pl.: L_{max}, $\sum T_j$, $\frac{1}{n}\sum_{j=1}^{n} F_j$). Az alábbiakban néhány, ezen új feltételekkel kapcsolatos, problémát mutatunk be.

$P \mid p_j=1, r_j, d_j \mid L_{max}$

Feltesszük, hogy a feladatok az $r_1 \leq r_2 \leq ... \leq r_n$ sorrendnek megfelelően vannak indexelve. Ha az összes release time egész, akkor a probléma egyszerűen megoldható. Egy bizonyítottan optimális ütemezést kaphatunk, ha az elérhető job-okat a lejárati idejük nem csökkenő sorrendjében rendezzük. Ha egy adott t időpillanatban nem minden végrehajtó egység foglalt, és van olyan J_j job, melyre: $r_j \leq t$, akkor ezen job-ok közül a legkisebb lejárati idővel rendelkezőt fogjuk ütemezni.

Algoritmus:

K: t időpillanatban foglalt gépek számlálója

m: az összes gép száma

M: t időpillanatban elérhető, de nem ütemezett job-ok halmaza

j: ütemezett job-ok számlálója

1. $j := 1$;
2. WHILE $j \leq n$ DO
   BEGIN
3. $t := r_j$; $M := \{J_i \mid J_i$ nem ütemezett; $r_i \leq t\}$; $K := 1$;
4. WHILE $M \neq \emptyset$ DO
   BEGIN
5. A legkisebb lejárati idejű J_i job megkeresése M-ben ;
6. $M := M \setminus \{J_i\}$;
7. J_i ütemezése a t időpillanatra;
8. $j := j + 1$;
9. IF $K + 1 \leq m$ THEN $K := K + 1$
   ELSE
   BEGIN
10. $t := t + 1$;
11. $K := 1$;
12. $M := M \cup \{J_i \mid J_i$ nincs ütemezve; $r_i \leq t\}$
   END
   END
   END

Az algoritmus O(n log n) időben lefut. Ez a módszer nem ad optimális megoldást abban az esetben, ha az elindítási idők tetszőleges racionális számok lehetnek.

$P \mid p_j=1, r_j, d_j \mid \sum w_j U_j$

Egy optimális ütemezést kaphatunk erre a problémára a korai job-ok egy S nevű halmaza által. A késő job-ok ütemezhetőek a folyamat végén, tetszőleges sorrendben. A korai job-okat az alábbi módon ütemezhetjük:

- Rendezzük a feladatokat azok lejárati idejük szerint nem csökkenő sorrendbe.
- Ütemezzük a feladatokat ebben a sorrendben (azaz az első *m* feladatot a 0. időpillanatra, a következő *m* feladatot az 1. időpillanatra *S*-ben… és így tovább).

A korai job-ok optimális *S* halmazának előállításhoz azokat nem csökkenő lejárati idejüknek megfelelő sorrendben adjuk hozzá a halmazhoz (és próbáljuk meg ütemezni őket). Amint egy feladat késővé válik, töröljük az *S*-ből a legkisebb w_j értékkel rendelkező job-ot.

Az alábbi algoritmus megadja a korai feladatok optimális *S* halmazát.

1. $S := \emptyset$;
2. FOR $i := 1$ TO n DO
3. IF i késik, mikor ütemezésre kerülne a legkorábbi üres időpillanatba egy gépen.
   THEN
   BEGIN
4. Keressük meg az i^* feladatot, melyre: $w_{i^*} = min_{i \in S} w_i$;
5. IF $w_{i^*} < w_i$ THEN cseréljük ki i^*-ot i-re az ütemezésben és *S*-ben
   END
6. ELSE adjuk hozzá *i*-t *S*-hez és ütemezzük a legkorábbi szabad időpillanatra.

Ez az algoritmus lefut O(n log n) időben megfelelő adatszerkezetek használata esetén.

1.3.2 Preemptivitás és a release/due time

P | pmtn, r_j, d_j | C_{max} és P | pmtn, r_j, d_j | L_{max}

Egy olyan preemptív ütemezést keresünk, melyben a maximális késés ($max_{j=1}^{n}\{C_j - d_j\}$) minimális. Adott egy küszöb érték (L). A kérdés: létezik-e olyan ütemezés, amelyre igaz az alábbi:

$$max_{j=1}^{n} L_j = max_{j=1}^{n}\{C_j - d_j\} \leq L$$

Ez akkor és csak akkor áll fenn, ha $C_j \leq d_j^L = L + d_j \quad j = 1, \dots, n.$

Ez alapján minden faladatnak be kell fejeződnie a módosított lejárati idő (d_j^L) előtt, és nem kezdődhet el r_j előtt. Azaz J_j-t végre kell hajtani a hozzá tartozó $[r_j,\ d_j^L]$ intervallumban. Ezeket az intervallumokat *időablak*oknak (*time window*) hívjuk.

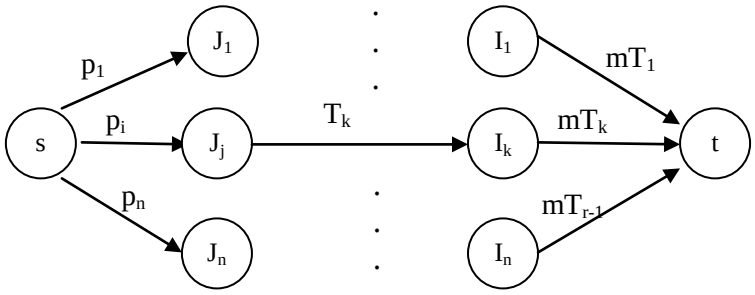
A probléma visszavezethető egy maximális hálózati folyam problémára az alábbiak szerint.

Legyen $t_1 < t_2 < \dots < t_r$ a különböző r_j és d_j értékek rendezett sorozata. Ekkor az intervallumok:

$$I_k = [t_k, t_{k+1}]\ , melynek\ hossza\ \ T_k = t_{k+1} - t_k \qquad k = 1, \dots, r-1$$

Hozzárendelünk egy-egy feladat-csúcspontot minden J_j job-hoz és egy-egy intervallum-csúcspontot minden I_k intervallumhoz. Továbbá hozzáadunk két többlet csúcspontot *s* és *t* néven. A hálózatot az alábbiak szerint definiáljuk:

- Minden feladat-csúcshoz vezet egy él *s*-ből, és ezek kapacitása rendre p_j.
- Minden intervallum-csúcsból vezet egy él *t*-be, ezek kapacitása: mT_k.
- J_j és I_k között létezik él, ha J_j végrehajtható I_k-ban (azaz $r_j \leq t_k$ és $t_{k+1} \leq d_j$). Ezen élek kapacitása: T_k.



Akkor és csak akkor létezik az összes időablakot tiszteletben tartó ütemezés, ha a maximális folyam a fenti hálózatban $\sum_{j=1}^{n} p_j$ értékű. Ebben az esetben a (J_j, I_k) élhez tartozó folyam (x_{jk}) megegyezik azzal az időperiódussal, amelyben a J_j job végrehajtás alatt állt az I_k időintervallumon belül.

Teljesülnie kell az alábbiaknak:

$\sum_{k=1}^{r-1} x_{jk} = p_j \qquad j = 1, \dots, n$ – minden feladat be legyen fejezve.

$\sum_{j=1}^{n} x_{jk} \leq mT_k \quad k = 1, \dots, r-1$ – ne lépjük túl a rendszer kapacitását egy intervallumban sem.

$x_{jk} \leq T_k \;\; minden\ (J_i, I_k)\ él\ esetén$ – ne lépjük túl egyik időablak hosszát sem.

Ha létezik olyan maximális folyam, amely kielégíti a fenti feltételeket, akkor egy megvalósítható ütemezés állítható elő: az *m* identikus gépen részfeladatokat (J_{ik}) ütemezünk $x_{ik} > 0$ végrehajtási idővel az I_k intervallumokba.

Mivel a hálózat legfeljebb O(n) csúcsból áll, a maximális folyam probléma megoldható $O(n^3)$ időben. Az így előállt P | pmtn, r_j, d_j | C_{max} probléma megoldása: $C_{max} \leq T_k$.

Ahhoz, hogy megoldjuk a P | pmtn, r_j, d_j | L_{max} problémát, bináris keresést kell végrehajtanunk minden *L* értékre vonatkozólag. Ekkor – $n\ max_{j=1}^{n} p_j \leq L_{max} \leq n\ max_{j=1}^{n} p_j$.

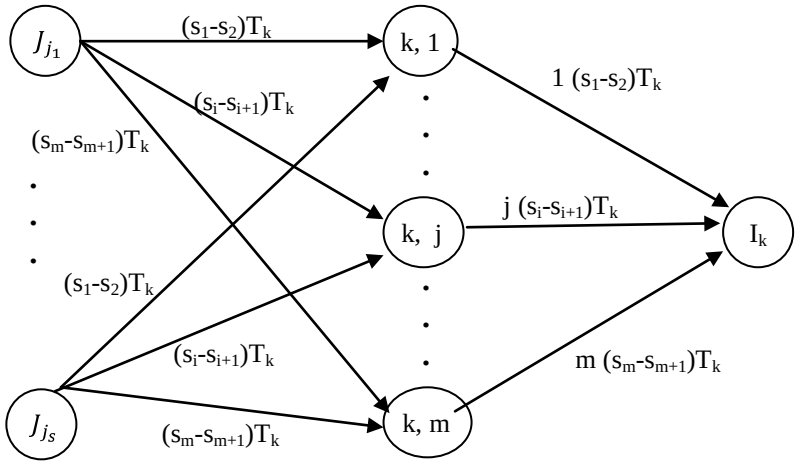
Q | pmtn, r_j, d_j | C_{max} és Q | pmtn, r_j, d_j | L_{max}

Ezen problémák megoldását a P | pmtn, r_j, d_j | C_{max} probléma megoldásához hasonlóan maximális hálózati folyam esetére vezethetjük vissza. A feltételek, az intervallumok és az időablakok meghatározása ugyanaz, mint az említett identikus gépekre vonatkozó probléma

esetén. Ezen felül feltesszük, hogy a végrehajtó egységek sebességeik nem növekvő sorrendjében vannak indexelve: $s_1 \geq s_2 \geq \ldots \geq s_m$.

Az alábbi módon egészítsük ki a P | pmtn, r_j, d_j | C_{max} problémához tartozó gráfot:

- Ez esetben I_k legyen tetszőleges közbenső csomópont, amely elődjeinek halmaza: $J_{j_1}, J_{j_2}, \ldots, J_{j_s}$.
- Ezután kicseréljük az $I_k, J_{j_1}, J_{j_2}, \ldots, J_{j_s}$ által meghatározott alhálózatot a lentebb ábrázolt kiterjesztett hálózatra. Valamint definiáljuk: $s_{m+1} = 0$.
- Megtartjuk az eredeti gráf *s*, *t*, J_j csúcsait, az ezekhez tartozó éleket és azok kapacitásait is.



Ha ebben a kiterjesztett hálózatban létezik *s*-ből *t*-be menő folyam $\sum_{j=1}^{n} p_j$ értékkel, akkor létezik megvalósítható ütemezés (ez fordítva is igaz).

Ez a maximális folyam probléma megoldható $O(mn^3)$ időben. Labetoulle, Lawler, Lenstra, és Rinnoy Kan azonban kifejlesztett a Q | pmtn, r_j, d_j | C_{max} probléma megoldására egy hatékony, O(n log n + mn) bonyolultságú, algoritmust.

Ahhoz, hogy megoldjuk a Q| pmtn, r_j, d_j | L_{max} problémát, bináris keresést kell végrehajtanunk minden *L* értékre vonatkozólag. Ekkor $-n\ max_{j=1}^{n} p_j \leq L_{max} \leq n\ max_{j=1}^{n} p_j$.

R | pmtn, d_j | L_{max} és R | pmtn, r_j, d_j | L_{max}

A problémák egy-egy lineáris programozási feladat megalkotásával oldhatóak meg, amely minimalizálja L_{max} értékét.

Feltesszük, hogy a végrehajtandó feladatok a nem csökkenő lejárati idejüknek megfelelően vannak számozva, azaz $d_1 \leq d_2 \leq \ldots \leq d_n$.

Legyen $x_{ij}^{(1)}$ azon idő teljes mennyisége, amelyet az M_i gép a J_j feladat megoldásával tölt az $I_1 = [0,\ d_1+L_{max}]$ időintervallumban. Továbbá, legyen $x_{ij}^{(k)}$ azon idő teljes mennyisége, amelyet az M_i gép a J_j feladat megoldásával tölt az $I_k = [d_{k-1}+L_{max},\ d_k+L_{max}]$ időintervallumban ($k = 2,\ldots n$). Ekkor a megoldandó feladat:

Minimalizálandó probléma: L_{max}

Feltételrendszer:

$$\sum_{i=1}^{m}\sum_{k=1}^{j}\frac{x_{ij}^{(k)}}{p_{ij}} = 1, \qquad j = 1,\ldots,n$$

$$\sum_{i=1}^{m} x_{ij}^{(1)} \le d_1 + L_{max}, \qquad j = 1,\ldots,n$$

$$\sum_{i=1}^{m} x_{ij}^{(k)} \le d_k - d_{k-1}, \qquad j = k,\ldots,n;\ \ k = 2,\ldots,n$$

$$\sum_{j=1}^{n} x_{ij}^{(1)} \le d_1 + L_{max}, \qquad i = 1,\ldots,m$$

$$\sum_{j=k}^{n} x_{ij}^{(k)} \le d_k - d_{k-1}, \qquad i = 1,\ldots,m;\ \ k = 2,\ldots,n$$

$$x_{ij}^{(k)} \ge 0, \qquad i = 1,\ldots,m;\ \ j,k = 1,\ldots,n$$

Ha az optimális L_{max} adott, akkor az optimális ütemezés megkapható, ha minden I_k időintervallumhoz ($k = 1,\ldots,n$) megkonstruáljuk a neki megfelelő ütemezést az $\left(x_{ij}^{(k)}\right)$ mátrix adatai alapján.

Hasonló módon oldható meg az R | pmtn, r_j, d_j | L_{max} probléma is. Ebben az esetben a $[t_k,\ t_{k+1}]$ intervallumokat vesszük figyelembe ($k = 1,\ldots,r-1$), ahol $t_1 < t_2 <\ldots< t_r$ az összes $d_j + L_{max}$ és r_j értékek rendezett sorozata. Ekkor az $x_{ij}^{(k)}$változó a J_j job M_i gépen történő végrehajtási idejét jelenti a $[t_k,\ t_{k+1}]$ időintervallumban.

1.4 Precedencia megszorítások

A P| | C_{max} probléma polinomiális időben való optimális megoldásához további egyszerűsítő feltételek szükségeltetnek. Feltesszük, hogy a feladatoknak egységnyi végrehajtási idejük van (p_j=1). Ez a feltétel a továbbiakban lehetővé teszi a feladatok közötti precedencia megszorítások vizsgálatát.

A precedencia megszorítások két típusát különböztethetjük meg:

- *prec* (korlát nélküli precedencia megszorítások):
  Adott egy irányított, aciklikus gráf, *G*, melynek csúcspontjai: 1,…,*n*.

Ha *G* tartalmaz egy irányított utat *j*-ből *k*-ba, azt $J_j \rightarrow J_k$ –val jelöljük, és elvárjuk, hogy J_j hamarabb befejeződjön, mint J_k elkezdődhetne

- *tree* (fa jellegű precedencia megszorítások)
  G egy gyökérből kiinduló fa szerkezet, melynek minden csúcspontjából legfeljebb egy él indul ki.

Az egyik legrégebbi eredmény ebben a probléma kategóriában a P| tree, $p_j = 1| C_{max}$ feladat megoldása O(n) időben. Bizonyított, hogy ez a módszer, változtatásokkal, képes ezen kívül a P| tree, $p_j = 1| \sum C_j$ problémát is megoldani.

Hu algoritmusa (Hu's algorithm) a kritikus útvonal ütemezéshez (critical path scheduling) kapcsolódik. Ez a critical path method (CPM) matematikai probléma egy változata, amely egy több feladatból álló projekt megoldásához szükséges időt minimalizálja egy gráf segítségével. A módszer meghatározza mely feladatok végezhetőek párhuzamosan, és melyik az a kritikus útvonal, amelynek késleltetése, sérülése az egész projekt befejezéséhez szükséges időt növelné. A módszer kiegészíthető azzal, hogy erőforrásokat rendelünk az egyes feladatokhoz és meghatározzuk ezek befolyását az adott feladat végrehajtási idejére.

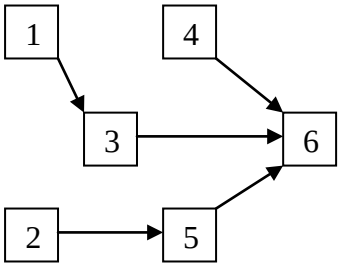
A fentebb említett ütemezési feladatban a módszert az alábbi módon alkalmazzuk:

- Definiáljuk ℓ_j-t, J_j szintjét, mint a csúcsok számát, melyek a *j* csúcstól a fa gyökeréig húzódó egyetlen útvonalon helyezkednek el.
- Alkalmazzunk listaütemezést (lásd: 1.2) arra a listára, amely tartalmazza a job-okat a nem növekvő ℓ_j által meghatározott sorrendben.

P| tree, $p_j = 1| C_{max}$

m = 2; n = 6;

G:



j	1	2	3	4	5	6
ℓ_j	3	3	2	2	2	1

Optimális ütemezés:

M_1	J_1	J_3	J_5	J_6
M_2	J_2	J_4		

$C_{max}^* = 4$

A másik ilyen eredmény a P2| prec, $p_j = 1$| C_{max} megoldása polinomiális idő alatt. Egy $O(n^3)$ bonyolultságú algoritmus a következő:

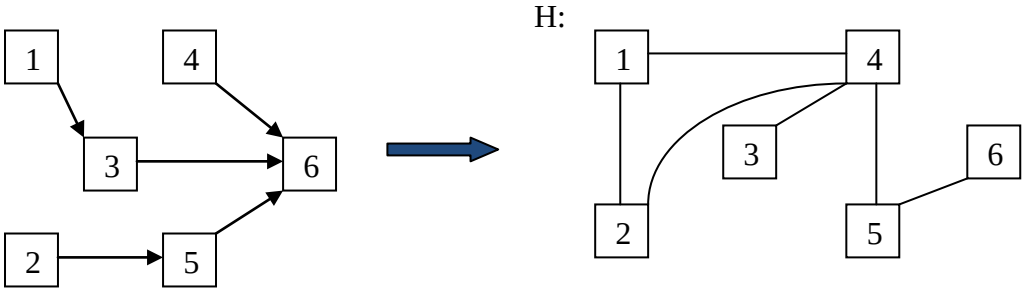
- Konstruáljunk egy irányítatlan gráfot, *H*-t, melynek csúcsai: 1,..,*n*.
- A *j* és *k* csúcs között csak akkor van él, ha sem $J_j \rightarrow J_k$ sem $J_k \rightarrow J_j$ nem áll fenn.
- Származtassunk egy optimális ütemezést *H* maximális számosságú párosításából (maximum cardinality matching). Egy párosítás *H* olyan éleinek halmaza, amelyek nem tartalmaznak közös csúcspontot.

A probléma továbbra is $O(n^3)$ időben megoldható, ha minden feladatot végre kell hajtanunk annak elindítási ideje (release time) és lejárati ideje (due time) között.

P2| prec, $p_j = 1$| C_{max}

n = 6;

G: H:



Optimális ütemezés:

M_1	J_1	J_3	J_5
M_2	J_2	J_4	J_6

$C_{max}^* = 3$

Minden egyéb $m \geq 3$ esetén a Pm| prec, $p_j = 1$| C_{max} probléma bonyolultsága nyitott kérdés. Azonban a P| prec, $p_j = 1$| C_{max} probléma NP-nehéz. Bebizonyították, hogy nincs polinomiális közelítő algoritmus erre a problémára, amely elérhet 4/3-nál (C_{max} / C_{max}^*) jobb legrosszabb-eset korlátot, mígnem *P=NP*. A critical path scheduling (CP) esetén ez az alábbi képlet szerint alakul:

$$C_{max}(CP)/C_{max}^* \leq \begin{cases} \frac{4}{3} & , ha\ m = 2 \\ 2 - \frac{1}{m-1} & , ha\ m \geq 3 \end{cases}$$

1.4.1 Precedencia és a release/due time

Az elindulási idők (r_j, d_j) a precedencia megszorításokkal *konzisztens*nek nevezzük, ha $r_i + p_i \leq r_j$ minden $J_i \rightarrow J_j$ esetén. Hasonlóképp a lejárati idők esetén a konzisztencia fennáll, ha $d_i \leq d_j - p_j$ minden $J_i \rightarrow J_j$ esetén.

P| tree, $p_j = 1$| L_{max}

A problémát megoldó módszer két lépésből áll. Az első lépésben módosítjuk a feladatok lejárati idejét úgy, hogy azok konzisztensek legyenek a precedencia megszorításokkal.

A második lépésben ütemezzük a feladatokat azok módosított lejárati idejének nem csökkenő sorrendjében.

A lejárati időt módosító eljárás során az eredeti d_i értékeket kicseréljük a $\{d_i, d_j - 1\}$ értékre minden $J_i \rightarrow J_j$ esetén. Ezt egy szisztematikus úton tesszük meg a fa gyökerétől a levélelemekig haladva. Miután módosítottuk a J_i lejárati idejét, töröljük az annak megfelelő csúcspontot a fából. Jelentse IP(i) az i csúcspont (azaz J_i) közvetlen elődeinek halmazát.

Algoritmus:

1. $T := \{i \mid i$ nem rendelkezik utóddal$\}$;
2. WHILE $T \neq \emptyset$ DO
   BEGIN
3. Keressük meg i-t a T-ben;
4. FOR ALL $j \in IP(i)$ DO
      BEGIN
5. $d_j := \min\{d_j, d_i - 1\}$;
6. $T := T \cup \{j\}$
      END;
7. $T := T \setminus \{i\}$
   END

A lejárati időt módosító algoritmus O(n) időben végrehajtható. A módosított lejárati időket d_i'-vel jelöljük, továbbá $d_i' < d_j'$ minden $J_i \rightarrow J_j$ esetén.

A második lépésben a feladatokat azok módosított lejárati idejének nem csökkenő sorrendjében ütemezzük. Minden feladatot a legkorábbi elérhető indulási időre ütemezzük. A legkorábbi elérhető indulási idő azt az időpillanatot jelenti, amelyben kevesebb, mint m folyamat van indulásra ütemezve és az aktuális folyamat összes elődje befejeződött. Ez a lépés O(n) időben végrehajtható, így a teljes probléma O(n log n) időben megoldható.

P2| prec, $p_j = 1$| L_{max}

Ezt a problémát szintén a lejárati idők módosításával oldhatjuk meg. Bár az itt alkalmazott módszer összetett. A módosított lejárati idők kiszámítása a leszármazottaktól az elődök felé halad. Legyen S(i) a J_i job nem szükségszerűen közvetlen leszármazottainak halmaza. Tegyük fel, hogy a módosított lejárati idők kiszámítása megtörtént minden $j \in S(i)$ esetére. Ezután definiálunk minden d valós érték, és i job paraméterekkel ellátott függvényt:

$$g(i,d) = |\{k | k \in S(i), d'_k \leq d\}|$$

A $g(i,d)$ függvény megadja az i csúcs összes olyan k leszármazottjának számát, amelyre teljesül, hogy $d'_k \leq d$.

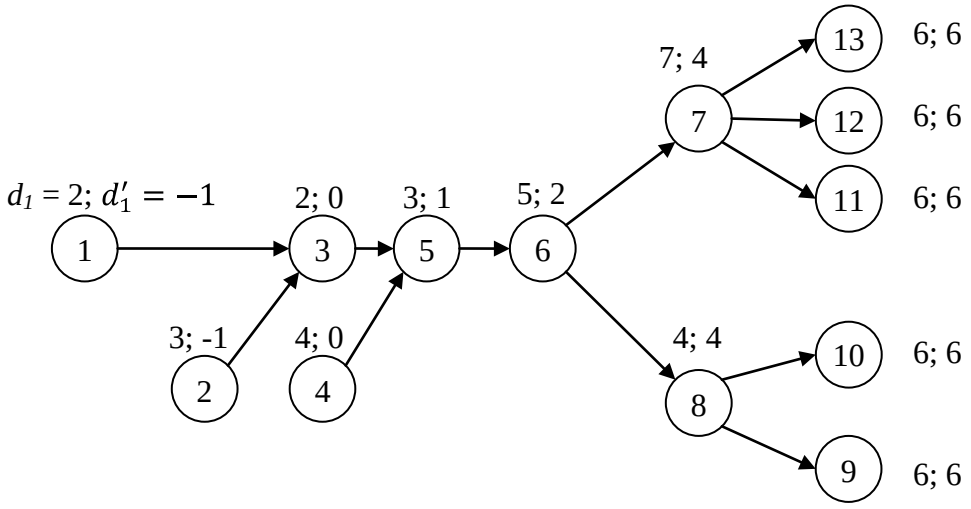
Ha $j \in S(i)$ és i minden leszármazottjának be kellett fejeződnie azok módosított lejárati ideje előtt, akkor az i feladatnak be kell fejeződni az alábbi időpillanat előtt:

$$d'_j - \left\lceil \frac{g(i,d'_j)}{2} \right\rceil$$

Ez a d'_i alábbi definíciójához vezet:

$$d'_i = min\left\{d_i, min\left\{d'_j - \left\lceil \frac{g(i,d'_j)}{2} \right\rceil \mid j \in S(i)\right\}\right\}$$

Példa:



Az i = 9, 10, 11, 12, 13 utód nélküli csúcs esetén $d_i = d'_i = 6$.

A további csúcspontok esetén: $d'_8 = min\left\{4, 6 - \left\lceil \frac{2}{2} \right\rceil\right\} = 4$; $d'_7 = min\left\{7, 6 - \left\lceil \frac{3}{2} \right\rceil\right\} = 4$;

$d'_6 = min\left\{5, 4 - \left\lceil \frac{2}{2} \right\rceil, 6 - \left\lceil \frac{7}{2} \right\rceil\right\} = 2$; … stb.

Ha a feladatok topologikus sorrendben vannak rendezve, akkor a módosított lejárati idők kiszámíthatóak, ha a csúcsokat fordított topologikus sorrendben járjuk be. Ez O(n^2) lépést

igényel, amely egy $O(n^3)$ időbeli megoldáshoz vezet. A folyamat bonyolultságát javíthatjuk $O(n^2)$-re, ha létrehozunk egy L listát, amelyben a feladatok az aktuális lejárati idejüknek (módosított vagy eredeti) megfelelően vannak rendezve. L(i) jelenti a lista i-edik helyén elhelyezkedő folyamatot. Tegyük fel, hogy az i job minden utódja módosítva lett és éppen a d_i' értéket akarjuk meghatározni. Ekkor az L listában keresünk. Ha $k = L(j) \in S(i)$, akkor a $g(i, d_k')$ függvény értéke az i csúcs utódjainak száma lesz, amelyek a listában j-vel egyező, vagy annál kisebb pozícióban helyezkednek el. Így a lista használatával hatékonyabban számolhatunk.

Az eredményként kapott listát felhasználva konstruálhatunk egy optimális ütemezést, oly módon, hogy a feladatokat azok módosított lejárati idejüknek nem csökkenő sorrendjében ütemezzük. Minden job a lehetséges legkorábbi időpillanatra kerül ütemezésre a két végrehajtó egység valamelyikén.

A teljes $P2|\ prec,\ p_j = 1|\ L_{max}$ probléma megoldásának bonyolultsága $O(n^2)$ (vagy $O(n^{\log 7})$, ha az utód halmazok is meghatározandóak).

Az eredeti lejárati időket tekintetbe vevő ütemezésben nem lesznek késő feladatok, akkor és csak akkor, ha a módosított lejárati időket figyelembe véve sem léteznek ilyenek.

A *release time* és *due time* témaköre egyúttal átvezet bennünket a valós idejű rendszerek problémakörébe, melyet részletesebben a következő fejezetben mutatunk be.

2. Valós idejű rendszerek

A valós idejű rendszerek bonyolultsága a nagyon egyszerű mikro-kontrollerektől egészen a nagy bonyolultságú, komplex és elosztott rendszerekig terjedhet. Ezen rendszerek létfontosságúak az olyan ipari infrastruktúrák esetén, mint például parancsvezérlés, folyamatvezérlés, légi irányítás, űrhajók szenzorjai, közlekedésirányítás, feladat kritikus számítások, robotika, nagy sebességű és multimédiás rendszerek, intelligens autópálya rendszerek… stb. Ezekben az esetekben az időnek alapvető fontosságú szerepe van. A későn kapott jó válasz ugyanolyan rossz, mint a hibás válasz.

A szakirodalomban az ilyen rendszerek definíciója a következő:

„*Olyan rendszerek, amelyekben a rendszer helyessége nem csak a számítás logikai eredményétől függ, hanem az időpillanattól is, amelyben az előáll*”.

2.1 Mit értünk valós idejű rendszerek alatt?

A fizikai világban a valós idejű rendszerek célja az, hogy fizikai hatást fejtsenek ki egy kiválasztott időkereten belül. Tipikus esetben egy ilyen rendszer tartalmaz *vezérlő rendszert* (*controlling system*), és *vezérelt rendszert* (*controlled system*). A vezérlő rendszer kapcsolatban áll környezetével, az ott rendelkezésre álló információkat, vagy azok egy részét szerzi meg. Egy valós idejű számítógépet, amely vezérel egy folyamatot vagy eszközt, az érzékelők valamilyen periodikus időközönként leolvasási adatokkal látják el. A számítógépnek ezekre kötelessége válaszolnia valamilyen jel elküldésével. Lehetséges, hogy váratlan vagy rendszertelen események történnek, és ezekre is válaszolni kell. Minden esetben lesz egy időkorlát, amelyen belül szolgáltatni kell valamilyen választ. A számítógép válaszadási képessége függ attól, hogy a szükséges számítások adott időn belüli elvégzésére milyen kapacitással rendelkezik. Ha több esemény következik be egymáshoz közeli időpillanatban, a számítógépnek ütemeznie kell a számításokat, annak érdekében, hogy minden kérés megkapja a megfelelő választ a maga időkorlátján belül. Minden folyamat, amely megjelenik a valós idejű rendszerben, rendelkezik bizonyos időbeli megszorításokkal, ezeket a megszorításokat figyelembe kell venni a folyamatok ütemezése során.

- *Elindítási idő (release /ready/ time)*: Az időpillanat, amelyben a folyamat készen áll a feldolgozásra.
- *Határidő (deadline)*: Az idő, amelyre a folyamat végrehajtásának be kellene fejeződnie, miután végrehajtásra kész állapotba került.

- *Minimális késleltetés (minimum delay)*: Az a minimális időmennyiség, amelynek el kell telnie a folyamat futásra kész állapotba kerülése, és annak tényleges végrehajtása között.
- *Maximális késleltetés (maximum delay)*: Az a maximális időmennyiség, amely eltelhet a folyamat futásra kész állapotba kerülése, és annak tényleges végrehajtása között.
- *Legrosszabb végrehajtási idő (worst case execution time)*: Az a maximális időmennyiség, amely a folyamat befejezésére rendelkezésre áll, annak végrehajtásra kész állapotba kerülése után. Ezen fogalom legrosszabb válaszidő (worst case response time) néven is ismert.
- *Futási idő (run time)*: A folyamat végrehajtásra kész állapotba kerülése után, annak végrehajtására szánt megszakításmentes idő mennyisége.
- *Súly/Prioritás (Weight/Priority)*: A folyamat relatív sürgőssége.

Nagymértékű és változó terhelésű rendszerek esetén, szükség lehet egynél több végrehajtó egységre a rendszeren belül. Ha a folyamatoknak függőségei, előfeltételei vannak akkor a befejeződési idők kiszámítása egy multiprocesszoros rendszerben magától értetődően bonyolultabb művelet, mint egyprocesszoros rendszer esetén.

Az alkalmazás természetéből fakadóan pedig sok esetben elosztott számítási műveletekre van szükség, több – egymáshoz kommunikációs vonallal kapcsolt – végrehajtó egység segítségével. Ez a kommunikációs költség és idő tovább bonyolíthatja a problémát.

2.1.1 Formális definíció

Ahogy az előbbi részből látható, az időtényező alapvető fontosságú a valós idejű rendszerek esetében. Ha az időbeli szigorú megszorítások nem teljesülnek, akkor rendszerhiba (system failure) következik be. Így alapvető szükségességű, hogy az időbeli megszorítások betartását garantáljuk. Az időbeli viselkedés garanciájához a rendszernek *előre megjósolható* (prediktábilis, *predictable*) működésűnek kell lennie. Ez azt jelenti, hogy ha egy folyamat aktiválódott, akkor bizonyosan meg kell tudni határozni annak befejeződési idejét. Továbbá kívánatos, hogy a rendszer magas fokú kihasználtságot tudjon elérni eme feltételek betartása mellett.

Szükségszerű, hogy a vezérlő rendszer által érzékelt környezeti állapot konzisztens legyen az aktuális környezeti állapottal, különben a kifejtett hatás káros lehet. Ezért a környezet periodikus monitorozása és az érzékelt információk időben történő feldolgozása lényeges.

Egy normális valós idejű alkalmazás több folyamatból épül fel, amelyek mindegyike eltérő szintű kritikussággal rendelkezhet az alábbiak szerint.

- *Gyenge valós idejű folyamat (soft real-time task)*: Bár a hiányzó, nem teljesített határidők nem kívánatosak egy valós idejű rendszerben, a gyenge valós idejű folyamatoknak lehetőségük van kihagyni néhányat, miközben a rendszer működése helyes marad (ez azonban bizonyos fokú büntetést vonhat maga után).
- *Erős valós idejű folyamat (hard real-time task)*: Az ilyen folyamatok nem hagyhatnak el egyetlen határidőt sem, különben ez nem kívánatos vagy végzetes eredményhez vezethet a rendszer számára.
- *Szilárd valós idejű folyamat (firm real-time task)*: Az ilyen folyamatok minél hamarabb befejeződnek az előírt határidejük előtt, annál több jutalmat kapnak.

Formálisan a következőképp adhatunk meg egy valós idejű rendszert:
Legyen egy rendszer, amely folyamatok egy halmazából áll, $T = \{T_1, T_2, \dots, T_n\}$, ahol minden folyamat legrosszabb végrehajtási ideje C_i. A rendszert valós idejűnek nevezzük, ha létezik legalább egy folyamat ($T_i \in T$), amely az alábbi kategóriák egyikébe esik:

- A T_i folyamat *erős valós idejű folyamat*. Azaz a T_i folyamat végrehajtásának be kellene fejeződni egy magadott D_i határidőn belül. Azaz $C_i \leq D_i$.
- A T_i folyamat *gyenge valós idejű folyamat*. Azaz minél később fejeződik be a T_i folyamat végrehajtása annak megadott D_i határideje után, annál nagyobb büntetést kap. Egy $P(T_i)$ büntetés függvényt definiálunk a folyamathoz:
  - Ha $C_i \leq D_i$, akkor $P(T_i) = 0$.
  - Egyébként $P(T_i) > 0$.
  - A büntetés függvény értéke $C_i - D_i$ függvényében növekszik.
- A T_i folyamat *szilárd valós idejű folyamat*. Azaz minél hamarabb fejeződik be a T_i folyamat végrehajtása annak megadott D_i határideje előtt, annál nagyobb jutalmat kap. Egy $R(T_i)$ jutalom függvényt definiálunk a folyamathoz:
  - Ha $C_i \geq D_i$, akkor $R(T_i) = 0$.
  - Egyébként $P(T_i) > 0$.
  - A jutalom függvény értéke $D_i - C_i$ függvényében növekszik.

A T valós idejű folyamatok halmaza állhat a fenti három kategóriába eső folyamatok tetszőleges kombinációjából is.

Legyen T_S a T-beli gyenge valós idejű folyamatok halmaza $T_S = \{T_{S,1}, T_{S,2}, \dots, T_{S,3}\}$. A büntetés függvényre pedig igaz: $P(T) = \sum_{i=1}^{l} P(T_{S,i})$. Továbbá T_F a T-beli szilárd valós idejű folyamatok halmaza $T_F = \{T_{F,1}, T_{F,2}, \dots, T_{F,3}\}$. A jutalom függvényre pedig igaz legyen: $R(T) = \sum_{i=1}^{l} R(T_{F,i})$.

2.2 Valós idejű ütemezés modellje

Folyamatok egy adott halmazára vonatkozólag az általános ütemezési probléma olyan sorrend meghatározásával kapcsolatos, amely megadja, hogy bizonyos megszorítások mellett mely folyamatok kerüljenek végrehajtásra. Valós idejű rendszerek esetén az alábbi célokat kell figyelembe venni:

- Időbeli megszorítások betartása.
- Osztott erőforrásokhoz való egyidejű hozzáférés megakadályozása.
- A megszorítások betartása mellett minél nagyobb fokú kihasználtság biztosítása.
- Preemptivitás okozta környezetváltások (context switch) redukálása.
- Elosztott valós idejű rendszerek esetén a kommunikációs költség redukálása.
- Erős/gyenge/szilárd valós idejűség figyelembe vétele, amely lehetőséget adhat dinamikus ütemezési vezérelvek alkalmazására.
- Megbízhatóság, biztonság, védelem.

A rendszernek leginkább megfelelő ütemezést meghatározza továbbá a rendszer és a rajta megjelenő folyamatok jellege:

- Gyenge / Erős / Szilárd valós idejű folyamatok
- Periodikus / Aperiodikus / Szórványos folyamatok
    - A *periodikus folyamatok* bizonyos fix rendszerességgel aktiválódnak. Általában egy-egy folyamat a megadott *P* perióduson belül egyszer kell, hogy lefusson.
    - Az *aperiodikus folyamatok* rendszertelenül és ismeretlen gyakorisággal aktiválódnak. Az egyedüli korlátot a *D* határidő jelenti.
    - A *szórványos folyamatok* rendszertelenül, de korlátozott gyakorisággal aktiválódhatnak. Ebben az esetben a határidő mellett adott az a minimum időintervallum, amely két sikeres aktiválás között eltelhet.
- Preemptív / Nem preemptív folyamatok

- Több / Egyprocesszoros rendszer
- Fix / Dinamikus prioritású folyamatok
- Rugalmas / Statikus rendszer
    - A rendszer működéséhez szükséges információk már annak elindítása előtt is rendelkezésre állnak (statikus), illetve működés közben is létrejöhetnek vagy módosulhatnak (dinamikus).
- Egymástól független / függő folyamatok

A valós idejű rendszerek folyamatai, feldolgozási környezetei és optimalitási kritériumai formalizálhatóak az 1. részben leírtaknak megfelelően (r_j, d_j, C_j, prioritás, precedenciák, késés, átlagos áthaladási idő…stb.). A valós idejű rendszerek esetén kiemelt fontosságúak az áthaladási idő, késés, pontatlanság valamint ezek átlagos vagy súlyozott átlagos értéke. Az ütemezhetőség fogalma kiegészül a következőképp: Egy folyamat ütemezhető, ha annak minden példánya befejeződik nem később, mint azok határideje. Továbbá egy folyamat halmaz ütemezhető, ha annak összes tartalmazott folyamata ütemezhető.

A megadott valós idejű rendszer tulajdonságai, paraméterei és alkalmazott ütemezési stratégiája alapján meghatározhatóak azok a feltételek, amelyek ütemezési algoritmusának megvalósításhoz elégségesek. Egy olyan rendszerben, amely egynél több processzort tartalmaz, nemcsak az ütemezési algoritmusról kell döntenünk, hanem az allokáló algoritmust is meg kell határoznunk. Ez utóbbinak feladata, hogy a rendelkezésre álló végrehajtó egységekhez hozzárendelje a folyamatokat. Multiprocesszoros rendszerek esetén ez a két algoritmus együttesen határozza meg egy-egy ütemezés megvalósíthatóságának szükséges feltételeit.

A továbbiakban két optimális valós idejű algoritmust tekintünk, amelyek bár elsősorban egyprocesszoros alkalmazásra születtek, viszont számos aspektusban kapcsolódnak a multiprocesszoros rendszerekhez. Egy optimális valós idejű algoritmusra igaz, hogy ha nem képes teljesíteni bizonyos időbeli megszorításokat, akkor nem létezik olyan másik ütemezés, amely képes lenne azok teljesítésére. A most következő két algoritmus esetén az alábbi feltételezésekkel élünk:

- Nincs olyan folyamat, amelynek lenne nem preemptív része, és a preemptivitás költsége elhanyagolható.

- Csak a végrehajtási igények lényegesek; memória, I/O, és más erőforrás szükségletek elhanyagolhatóak.
- Minden folyamat független; nincsenek precedencia megszorítások.

2.2.1 Rate-Monotonic (RM) ütemezés

Az RM ütemezés a legszélesebb körben tanulmányozott és gyakorlatban alkalmazott algoritmus. A fenti három megszorításon felül feltesszük, hogy minden folyamat periodikus és a T_i folyamat prioritása nagyobb, mint a T_j folyamaté, ahol $i < j$. Az RM ütemezés egy típusa a statikus prioritásos, prioritás vezérlet algoritmusoknak abban az értelemben, hogy minden kérelem prioritása ismert már annak megérkezése előtt. Ezt a prioritást a folyamatok periódusa határozza meg. Egy periodikus folyamat *példányok* (*instance*) végtelen sorozatából áll, amely példányok határideje lehet nagyobb, kisebb vagy egyenlő az aktuális példány elindítási idejével. Továbbá minden folyamat minden példányának végrehajtási ideje megegyezik.

Egy periodikus T_i folyamatot az alábbi három tényező határoz meg:

- P_i, a kérelem periódus ideje.
- C_i, a végrehajtási idő.
- D_i, a folyamat határideje.

Egy n periodikus folyamatból álló halmaz *kihasználási tényezője* (*utilization factor*) a $\sum_{i=1}^{n} \frac{C_i}{P_i}$ összeg. Ha $\sum_{i=1}^{n} \frac{C_i}{P_i} \leq n(2^{1/n} - 1)$, ahol n az ütemezendő folyamatok száma, akkor az RM algoritmus képes ütemezni az összes folyamatot úgy, hogy azok mindegyike teljesítse saját határidejét *(Liu – Layland feltétel).* Ez viszont egy elégséges, de nem szükséges feltétel. Létezhetnek olyan, RM által ütemezhető, folyamathalmazok, amelyekre nem áll fenn ez az egyenlőtlenség.

Egy szükséges és elégséges feltétel az alábbi:

Feltesszük, hogy a T_i folyamat végrehajtása befejeződik a t időpillanatban.

$$W_i(t) = \sum_{j=1}^{i} C_j \left\lceil \frac{t}{P_j} \right\rceil = t - \text{üresjárat}$$

$$L_i(t) = \frac{W_i(t)}{t}$$

$$L = min_{0<t<P_i}\{L_i(t)\}$$

A T_i folyamat megvalósíthatóan ütemezhető akkor és csak akkor, ha $L_i(t) \leq 1$. Ebben az esetben a $T_1, T_2, \ldots, T_{i-1}$ is megvalósíthatóan ütemezhető.

Az RM ütemezés hátránya, hogy a folyamatok prioritása azok periódusai által definiáltatott. Sokszor meg kell változtatnunk egy-egy folyamat prioritását annak érdekében, hogy a kritikus folyamatok biztosan sikerrel befejeződjenek. A kérdés az, hogyan oldható meg, hogy a kritikus folyamatok teljesítsék határidejüket a legrosszabb esetben is.

Erre megoldást kínál az alábbi két módszer valamelyike:

- Megnyújtjuk a nem kritikus folyamatok periódusidejét egy *k* tényezővel.
- Lerövidítjük a kritikus folyamatok periódus idejét egy *k* tényezővel.

Idáig feltettük, hogy egy-egy folyamat relatív határideje megegyezik annak periódusával. Ha ezt a feltételezést elhagyjuk, akkor az RM algoritmus nem lesz többé optimális. Ha $D_i \leq P_i$, akkor ugyanazon folyamat legfeljebb egy példányban fog futni bármely időpillanatban. Viszont ha $D_i > P_i$, akkor lehetséges, hogy egyazon folyamat több példányban is jelen lesz egyszerre. Az utóbbi esetben ellenőrzés alatt kell tartanunk a példányok számát annak érdekében, hogy a szükséges válaszidőt fenntarthassuk.

Az RM algoritmus időbonyolultsága $O((N + a)^2)$, ahol N az összes kérések száma az n periodikus folyamatot tartalmazó rendszerben, és a az aperiodikus folyamatok száma.

2.2.2 Earliest Deadline First (EDF) ütemezés

Az EDF ütemezési algoritmus egy optimális prioritás vezérelt algoritmus, amelyben a magasabb prioritás ahhoz a kéréshez kerül hozzárendelésre, amely a legkorábban lejáró határidővel rendelkezik. Továbbá a magasabb prioritású kérés megszakíthat egy nála kisebb prioritással rendelkezőt. Ezen algoritmus esetén dinamikus prioritásról van szó, amely azt jelenti, hogy egy kérés prioritása akkor válik ismerté, amikor az beérkezik. Az RM algoritmussal analóg módon szokták *Deadline-monotonic* algoritmusnak is nevezni. Feltételezzük, hogy minden időpillanatban, amelyben egy új, végrehajtásra kész folyamat érkezik, azt beszúrjuk a korábban végrehajtásra kész állapotba került folyamatok, határidők szerint rendezett listájába. Amennyiben rendezett listákat használunk, akkor az EDF algoritmus időbonyolultsága az RM algoritmuséval megegyező.

Az EDF algoritmus esetén megtesszük ugyanazokat a feltevéseket, amelyeket az RM esetén feltettünk, kivéve azt, hogy a folyamatoknak nem kell periodikusaknak lenniük.

Ha minden folyamat periodikus, és relatív határidejük megegyezik periódusukkal, akkor az EDF képes megvalósíthatóan ütemezni őket akkor és csak akkor, ha $\sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1$. Nincs egyszerű ütemezhetőségi próba arra az esetre vonatkozólag, ahol a relatív határidők nem minden esetben azonosak a periódusokkal. Az alábbi egy erre az esetre vonatkozó próba:

$U = \sum_{i=1}^{n} \frac{C_i}{P_i}$;

$D_{max} = max_{1 \leq i \leq n}\{D_i\}$;

$P = lkt(P_1, \ldots, P_n)$, ahol *lkt* a legkisebb közös többszörös függvény.

Legyen *h(t)* a *t*-nél kisebb abszolút határidővel rendelkező összes folyamat végrehajtási idejének összege.

Egy *n* folyamatból álló halmaz *nem* EDF-megvalósítható akkor és csak akkor, ha

- $U < 1$ vagy
- létezik $t < min\left\{P + D_{max},\ \frac{U}{1-U} max_{1 \leq i \leq n}\{P_i - D_i\}\right\}$, amelyre: *h(t)* > 1

Létezik egy másik dinamikus prioritásos ütemező algoritmus, a Least laxity first (LLF) algoritmus. Ez periodikus valós idejű rendszerek esetén optimális is. Egy folyamat pontatlansága vagy lazasága (laxity) a határidő és a hátralévő végrehajtási idő különbségét jelenti. Más szavakkal a pontatlanság az a maximális időtartam, amelyet a folyamat várhat, határidejének betartása mellett. Az algoritmus a legnagyobb prioritást annak az aktív folyamatnak adja, amelynek pontatlansága a legkisebb. Ezután a legnagyobb prioritású folyamat végrehajtásra kerül. Miközben egy folyamat végrehajtás alatt áll, az megszakítható egy olyan folyamat által, melynek pontatlansága az aktuálisan futó folyamaté alá csökkent. Probléma akkor merül fel, ha két folyamat közel azonos pontatlansággal rendelkezik. Ez nagymértékben megnövelheti a környezet váltások számát, ezzel rontva a teljesítményt. Az EDF-hez hasonlóan ez a módszer is a folyamatok (pontatlanság szerint) rendezett listájába való beszúrást alkalmazza. Időbonyolultsága azonos az EDF időbonyolultságával.

2.3 Multiprocesszoros ütemező algoritmusok

A valós idejű rendszerek ütemezése sokat tanulmányozott terület, elsősorban egyprocesszoros rendszereken. Olyan gépeken, amelyekben pontosan egy megosztott processzor áll rendelkezésre, és minden folyamatnak ezen kell végrehajtódnia. Multiprocesszoros platformok esetében több processzor áll rendelkezésre, amelyeknek bármelyikén végrehajtásra kerülhetnek ezek a folyamatok. A *P-Fair ütemezés* (proportionate fairness) az

egyike azon kevés ismert optimális módszereknek, amelyek folyamatok multiprocesszoros rendszerben való ütemezésével kapcsolatosak. Azonban a folyamatok processzorokhoz való optimális hozzárendelése majdnem minden gyakorlati esetben NP-nehéznek bizonyul. Így különféle *heurisztikák*at kell alkalmaznunk. Ezek a heurisztikák nem képesek garantálni, hogy találunk olyan hozzárendelést, amely lehetővé teszi, hogy minden folyamat megvalósíthatóan ütemezhető legyen. Az egyetlen lehetőségünk az, hogy a hozzárendelés után ellenőrizzük a megvalósíthatóságot, és ha a hozzárendelés nem bizonyul megfelelőnek, akkor megpróbáljuk módosítani azt.

Amikor egy ilyen hozzárendelés megoldhatóságát ellenőrizzük, számításba kell vennünk a kommunikációs költségeket. Ha a kommunikáló folyamatok egyazon végrehajtó egységhez vannak rendelve, akkor ez a költség zérus. Mikor azonban különböző processzorokhoz vannak rendelve, akkor a kommunikációs költség pozitív lesz.

A következő feltételezéseket is meg kell tennünk multiprocesszoros ütemező algoritmusok tervezésekor:

- Folyamat preemptivitás megengedett
    - Így valamely processzoron futó folyamat megszakítható annak befejeződése előtt, és egy későbbi időben végrehajtása folytatható.
- Folyamat migráció megengedett
    - Így az a folyamat, melynek végrehajtását megszakítottuk egy bizonyos processzoron, folytatható lesz egy másikon.
- Folyamat párhuzamosság tiltott
    - Bármely folyamat egy időben legfeljebb egy végrehajtó egységen futhat.

A többprocesszoros ütemezési technikák két nagyobb kategóriába sorolhatóak:

- Globális ütemezési algoritmusok (Global scheduling algorithms)
- Szétválasztásos ütemezési algoritmusok (Partitioning scheduling algorithms)

2.3.1 Globális ütemezési algoritmusok

A globális ütemező algoritmusok egy – a processzorok között megosztott – várakozási sorban tárolják azokat a folyamatokat, amelyek beérkeztek, de végrehajtásuk még nem ért véget. Tegyük fel, hogy *m* processzor áll rendelkezésre. Minden időegységben a sor *m* legnagyobb prioritású folyamata kerül kiválasztásra és végrehajtásra az *m* processzoron, migráció és

preemptivitás használata mellett (ha szükséges). Ehhez az egy processzoron optimális RM ütemezés is alkalmazható, bár ebben az esetben már egyáltalán nem optimális.

A *focused addressing and bidding* algoritmus a globális ütemezési módszerek egy gyakorlati példája. Az algoritmus alapötlete a következő. Minden processzor karbantart egy státusz táblázatot, amely jelöli, hogy mely folyamatok lettek már futtatva. Ezen felül karbantartanak egy-egy táblázatot a többi processzor fennmaradó, kihasználatlan számítási kapacitásáról. Az időtengely úgynevezett ablakokra (fix hosszúságú intervallumok) van osztva. Minden processzor rendszeresen elküldi társainak a következő időablakának éppen szabad hányadáról szóló információt.
A másik oldalról nézve, egy túlterhet processzor ellenőrzi a fennmaradó szabad kapacitásokról szóló információit és kiválasztja azt a processzort, amelyen a legnagyobb valószínűséggel lesz sikeres a túlterhelés miatt veszélyeztetett folyamat végrehajtása. Átadja a folyamatot az így kiválasztott egységnek. Azonban lehetséges, hogy a választás alapjául szolgáló információ már nem aktuális, és a kiválasztott processzor már nem rendelkezik azzal a szabad kapacitással, amely a folyamat végrehajtásához szükséges. Annak érdekében, hogy ezt a problémát elkerülje, ellenőriz más, gyengén terhelt processzort is. A válaszokat a kiválasztott cél processzor kapja meg. Így ha ő maga már nem képes a sikeres végrehajtásra, akkor ellenőrizni tudja, hogy mely más processzor lehet alkalmas rá, és továbbítja annak a folyamatot.

A globális ütemezések egyik legjelentősebb problémája az úgynevezett *Dhall effektus*, egy ütemezési dilemma, amelyben néhány folyamathalmaz ütemezhetetlennek bizonyul annak ellenére, hogy kihasználási tényezőjük (RM) alacsony. Ez megjelenik akkor is, ha a jól ismert RM módszert közvetlenül alkalmazzuk globális ütemezésre.
Legyen egy folyamathalmaz $T = \{(P_1 = 1, C_1 = 2\epsilon), (P_2 = 1, C_2 = 2\epsilon), \dots, (P_m = 1, C_m = 2\epsilon), (P_{m+1} = 1 + \epsilon, C_{m+1} = 1)\}$, RM-ütemezhető kellene, hogy legyen m processzoron. Ebben az esetben a T_{m+1} rendelkezik majd a legkisebb prioritással, és csak az összes többi után kerülhet végrehajtásra. A folyamathalmaz azonban nem ütemezhető, és ha $\epsilon \to 0$, akkor a folyamathalmaz processzor kihasználási tényezője 1 lesz, függetlenül a processzorok számtól. Eközben ahogy m nő, úgy csökken a rendszer kihasználtsága a 0 irányába. Azaz bekövetkezik a Dhall effektus.

Megfigyelhetjük azonban, hogy ha a T_{m+1}-hez valamiképp nagyobb prioritást rendelhetnénk, úgy a folyamathalmaz ütemezhetővé válna. Ahhoz, hogy ezt elérjük az egyes folyamatok prioritásának a periódusuk és végrehajtási idejük különbségét választjuk meg. Ekkor a T_{m+1} kapja a legnagyobb prioritást, és a folyamathalmaz akkor is ütemezhető marad, ha valamely folyamatának végrehajtási ideje kissé megváltozik. Ezen alapszik a következőkben bemutatott eljárás is.

2.3.1.1 adaptiveTkC

A fenti megfigyelés alapján kapunk egy újfajta prioritás hozzárendelési sémát, a *TkC* nevű sémát, amelyben a tetszőleges T_i folyamat prioritását a periódus és a végrehajtási idő súlyozott különbsége adja meg. Képletben: $P_i - k \cdot C_i$, ahol k egy úgynevezett *globális szabadsági tényező (slack factor).* Ha k = 0, akkor az eredeti RM prioritás hozzárendelést kapjuk vissza. A fő kérdés az, hogy mennyi legyen k értéke.
Ha $k \leq 0$, a Dhall effektust tapasztaljuk. Még akkor is, ha $0 < k \leq 1$ a Dhall effektushoz hasonló folyamatot tapasztalhatunk.
Legyen egy folyamathalmaz $T = \left\{(P_1 = 1, C_1 = \epsilon), (P_2 = 1, C_2 = \epsilon), \dots, (P_m = 1, C_m = \epsilon), \left(P_{m+1} = \frac{1}{\epsilon^2} + \epsilon, C_{m+1} = \frac{1}{\epsilon^2}(1 - \frac{\epsilon}{2})\right)\right\}$, ahol $\frac{1}{\epsilon}$ pozitív egész, ütemezhető kellene, hogy legyen m processzoron TkC használatával ($0 < k \leq 1$). A folyamathalmaz azonban nem ütemezhető, és ha $\epsilon \to 0$, akkor a folyamathalmaz processzor kihasználási tényezője 1 lesz.

Olyan k érték választása esetén, amely kissé nagyobb 1-nél szintén a Dhall effektushoz hasonló folyamatot tapasztalunk. Legyen egy folyamathalmaz $T = \left\{\left(P_1 = 1, C_1 = \frac{k-1}{k} + \epsilon\right), \left(P_2 = 1, C_2 = \frac{k-1}{k} + \epsilon\right), \dots, \left(P_m = 1, C_m = \frac{k-1}{k} + \epsilon\right), \left(P_{m+1} = \frac{1}{\epsilon^2} + \frac{k-1}{k} + \epsilon, C_{m+1} = \frac{1}{\epsilon^2}(1 - \frac{k-1}{k} - \frac{\epsilon}{2})\right)\right\}$, ahol $\frac{1}{\epsilon}$ pozitív egész, ütemezhető kellene, hogy legyen m processzoron TkC használatával ($1 \leq k$). A folyamathalmaz azonban nem ütemezhető, és ha $\epsilon \to 0$, akkor a folyamathalmaz processzor kihasználási tényezője 1 lesz.

Másrészt egy túl nagy k érték választása is rossz ötlet, mivel így a prioritások úgy kerülnek kiosztásra, hogy a leghosszabb végrehajtási idejű folyamat kapja a legnagyobb prioritást.

Legyen a folyamathalmaz $T = \left\{\left(P_1 = 1, C_1 = \frac{1}{1+k} + \epsilon\right), \left(P_2 = 1, C_2 = \frac{1}{1+k} + \epsilon\right), \ldots, \left(P_m = 1, C_m = \frac{1}{1+k} + \epsilon\right), \left(P_{m+1} = \frac{1}{1+k} + \epsilon, C_{m+1} = \epsilon^2\right)\right\}$, ahol $k \to \infty$, ütemezhető kellene, hogy legyen *m* processzoron TkC használatával. A folyamathalmaz azonban nem ütemezhető, és ha $\epsilon \to 0$, akkor a folyamathalmaz processzor kihasználási tényezője 1 lesz, függetlenül a processzorok számától.

Ahhoz, hogy a TkC hasznos legyen, szükségünk van egy jó *k* értékre. A vizsgált és nem ütemezhetőnek minősülő, alacsony kihasználtságot produkáló esetek rendelkeznek két közös tulajdonsággal:

- A folyamatok száma egyel nagyobb a processzorok számánál.
- Az összes legmagasabb prioritású folyamatnak megegyezik a periódusa és végrehajtási ideje.

Mivel ezek a folyamathalmazok a legrosszabb eseteket szimbolizálják, így ezekre az esetekre érdemes a *k* értékét optimalizálnunk (*k* ezen értékét alkalmazó eljárás neve: adaptiveTkC).

Legyen $m \geq 2$ processzor, és $n = m + 1$ folyamat. Az *m* legnagyobb prioritású folyamat periódusa és végrehajtási ideje megegyezik. A folyamatok a $P_i - k \cdot C_i$ képlet adta periódusok szerint rendezettek. A következő *k* érték maximalizálja a rendszer kihasználtságának legkisebb értékét:

$$k = \frac{1}{2} \cdot \frac{m - 1 + \sqrt{5m^2 - 6m + 1}}{m}$$

A szóban forgó kihasználtság:

$$U_s = 2\frac{m}{3m - 1 + \sqrt{5m^2 - 6m + 1}}$$

Az így kapott rendszer kihasználtság alsó határa sosem lesz kisebb, mint $\lim_{m\to\infty} U_s > 0{,}38$, ami azt jelenti, hogy sikerült elkerülni a Dhall effektust. Kiterjedt tapasztalati és kísérleti vizsgálatok szerint a módszer ugyanilyen sikerrel alkalmazható általános folyamathalmazok esetében is.

Felmerülhet a kérdés, hogy érdemes-e globális ütemezést alkalmaznunk miközben a várható rendszerkihasználtság mindössze 38%. Azonban figyelembe kell vennünk azt a tényt is, hogy szétválasztásos ütemezés esetén ez az érték 41%. Egy átlagos környezetben az adaptiveTkC képes akár jobb teljesítményt is nyújtani szétválasztásos társainál.

2.3.2 Szétválasztásos ütemezési algoritmusok

Az ilyen típusú algoritmusok szétválasztják a folyamatok halmazát, úgy, hogy minden folyamat, amely egy egységbe került, ugyanazon processzorhoz kerül hozzárendelésre. A folyamatok migrációja nem megengedett, így a multiprocesszoros ütemezési feladatot visszavezettük több egyprocesszoros ütemezési problémára.
Tehát folyamat osztályok halmazát definiáljuk. A folyamatok, amelyek azonos osztályba tartoznak, garantáltan teljesítik az RM ütemezhetőség feltételeit egy processzoron. Egyenként hozzárendeljük a folyamatokat a megfelelő folyamatosztályokhoz. Ezután, ezzel a hozzárendeléssel, lefuttatjuk az RM ütemező algoritmust minden processzoron.

Azonban az optimális processzor-folyamat hozzárendelések megtalálása NP-nehéz feladat. Így a folyamatokat általában nem optimális heurisztikák segítségével kell szétválasztani. További nehézség, hogy az ütemezhetőség eldöntésére használható szükséges és elégséges RM feltétel (lásd: 2.2.1) nem elég gyorsan ellenőrizhető és adat-függő, melynek következtében kiszámítása akár – a folyamatok számától függően – exponenciális időbonyolultságú is lehet. A gyorsan kiszámítható Liu-Layland feltétel pedig amellett, hogy szintén függ a folyamatok számától, csak elégséges, de nem szükséges feltételt ad meg, így létezhetnek olyan RM-ütemezhető folyamat halmazok, amelyeket nem ismerhetünk fel a segítségével.

A hozzárendeléshez leggyakrabban alkalmazott egyik heurisztika az úgynevezett *hátizsák probléma* (bin-packing problem) heurisztikus megoldása. Ebben a processzorokat fix méretű tárolókként, a folyamatokat pedig változó méretű tárgyakként tekintjük. A feladat célja: az összes tárgy elhelyezése a legkevesebb számú tároló felhasználásával. Más szavakkal: minél nagyobb legyen a rendelkezésre álló számítási kapacitás kihasználtsága, miközben teljesülnek a valós idejűség megszorításai. Erre több egyszerű módszer is létezik:

- First Fit: az első megfelelő méretű helyre kerül be a folyamat.
- Best Fit: az összes közül a leginkább megfelelő méretű helyre kerül a folyamat.
- Last Fit: az utolsó megfelelő méretű helyre kerül a folyamat.
- Next Fit: az utolsó helyre kerül a folyamat, ha szükséges új sort kezd.

A fenti módszereknek léteznek hatékonyságot növelő, átrendezést alkalmazó változataik is (Pl.: First Fit Decreasing, amely a folyamatokat kihasználási tényezőjük (*utilization factor*) alapján rendezi a hozzárendelés megkezdése előtt).

További hatékonyság növekedés érhető el, ha az úgynevezett *kompatibilis folyamatokat* egyazon processzorhoz rendeljük. A kompatibilis folyamatok olyan folyamatok, amelyek relatíve magas processzorkihasználtságot érnek el abban az esetben, ha ugyanazon processzoron futnak. Például az egymáshoz közeli periódussal rendelkező folyamatok ilyenek, valamint vizsgálatok szerint a kettő hatványaihoz közelálló periódussal rendelkezőek is kompatibilisnek tekinthetőek.

Az ütemezhetőség vizsgálatára pedig számos hatékonyan számolható, és kellően pontos módszert fejlesztettek már ki. A hatékony számolhatóság azért nagyon fontos, mert ezek az ütemezhetőségi vizsgálatok minden processzoron többször is lefutnak, így ha a vizsgálat bonyolult, akkor a teljes megoldás időbonyolultsága gyorsan és nagymértékben növekszik. Ezen vizsgálatok és a bin-packing heurisztikák használatával hatékonyan működő, RM ütemezéshez közelítő módszereket kaphatunk. Ezekből tekintjük most a legjobbaknak vélteket.

2.3.2.1 RM-FFDU

Először is bemutatjuk az ütemezhetőségi feltételt, amelyet az RM-FFDU alkalmazni fog, majd a használt processzor-folyamat hozzárendelés stratégiáját.

Legyen $\sum = \{T_i = (C_i, P_i) | i = 1,2, \dots, n-1\}$ folyamatoknak egy $(n-1)$ tagú halmaza. Ezen folyamatok kihasználási tényezője legyen rendre $\{u_1, u_2, \dots, u_{n-1}\}$. Feltesszük, hogy ezek a folyamatok megvalósíthatóan ütemezhetőek az RM algoritmussal. Egy újonnan érkező folyamat $T_n = (C_n, P_n)$ megvalósíthatóan RM-ütemezhető együttesen a korábbi $(n-1)$ folyamattal, ha teljesül az alábbi egyenlőtlenség:

$$\frac{C_n}{P_n} \leq 2\left[\prod_{i=1}^{n-1}(1+u_i)\right]^{-1} - 1$$

Ez a feltétel magában foglalja a Liu-Layland feltételt és időbonyolultsága tisztán O(n). Hatékonyságban erősebb, mivel mindig olyan processzor kihasználtságot jelez, amely nem kisebb, mint a Liu-Layland feltétel által megadott érték. Továbbá időbonyolultságát tekintve jobb, mint a szükséges és elégséges RM feltétel, mivel a folyamatok számától csak lineárisan függ.

A nagyobb processzor kihasználást az új feltétel (*Utilization Oriented – UO* feltétel) úgy éri el, hogy figyelembe veszi a folyamatok kihasználási tényezőinek relatív értékét.

A folyamatok processzorokhoz való rendeléséhez az ismert bin-packing heurisztikák jól alkalmazhatóak. A számos algoritmus közül a First fit a leggyakrabban alkalmazott és legszélesebb körben tanulmányozott módszer. A First fit egyszerű, on-line és közel optimális teljesítmény nyújt. A csökkenő rendezést alkalmazó híres First Fit Decreasing módszer nyilvánvalóan off-line. Ismert, hogy ha az adatok elő feldolgozás alá kerülnek, majd ugyanazt a heurisztikát alkalmazzuk, akkor a hatékonyság növelhető. Davari és Dhall FFDUF algoritmusa egy példa a First fit heurisztikának kihasználási tényezőjük alapján rendezett folyamatok esetén való alkalmazására. Ezen okoktól vezérelve ebben az esetben is ez a heurisztika került alkalmazásra.

Az algoritmus: (Input: Folyamat halmaz; Output: m)

1. Rendezzük a folyamatokat kihasználási tényezőik nem növekvő sorrendjébe.
2. $i := 1$; $m := 1$;
3. $j := 1$; WHILE ($u_i > 2/\prod_{l=1}^{k_j}(u_{j,l}+1)$) DO
4. $j := j + 1$;
5. $k_j := k_j + 1$;
6. $U_j := U_j + 1$; // a T_i folyamat hozzárendelése P_j-hez
7. IF ($j > m$) THEN $m := j$;
8. $i := i + 1$;
9. IF ($i > n$) THEN Exit;
10. ELSE Goto 3;

A különbség az RM-FFDU és az FFD heurisztika között az, hogy más-más feltételt használnak a processzorra vonatkozólag, ugyanez a különbség fennáll az RM-FFDU és az FFDUF között is. Az algoritmus végén az m értéke jelöli, hogy az RM-FFDU hány processzorra képes ütemezni a folyamatokat. A k_j értéke a P_j processzorhoz rendelt folyamatok számát adja. Az RM-FFDU algoritmus időbonyolultsága $O(n \log n)$.

Bár eme algoritmus az FFD heurisztikát alkalmazza, annak hatékonyság növelő módszerei nem alkalmazhatóak közvetlen módon itt, minthogy a processzor kihasználtsága egy változó kell, hogy legyen, míg a „tárolók” mérete fix marad vagy egységnyi.

2.3.2.2 R-BOUND-MP

Ez a módszer szintén a folyamatok periódusáról szóló információkat használja fel annak érdekében, hogy jó ütemezhetőségi vizsgálatot végezhessen.

A T-BOUND módszer (amely az R-BOUND alapja) először is átalakítja az eredeti folyamathalmazt, majd ezen új halmazon végzi el vizsgálatait. Az átalakító algoritmust *Scale Task Set* algoritmusnak hívják.

Legyen T az eredeti folyamathalmaz: $T = \{T_i = (C_i, P_i) | i = 1,2, \dots, n\}$

Scale Task Set (Input: *T*; Output: *T'*):

1. BEGIN
2. FOR $i = 1$ TO $n - 1$ DO
3. $P_i' = P_i \cdot 2^{log\left\lfloor \frac{P_n}{P_i} \right\rfloor}$;
4. $C_i' = C_i \cdot 2^{log\left\lfloor \frac{P_n}{P_i} \right\rfloor}$;
5. END FOR
6. Rendezzük a *T'* folyamatait periódusidejük növekvő sorrendjébe!
7. RETURN *T';*
8. END

Bizonyított, hogy ha az eredményül kapott *T'* RM-ütemezhető, akkor az eredeti *T* is az. Viszont ha a *T'* nem ütemezhetőnek bizonyul, abból nem következtethetünk *T* ütemezhetőségére vonatkozólag semmire.

Ezek után a T-BOUND az alábbi ütemezhetőségi vizsgálatot alkalmazza:

Legyen egy *n* elemű *T* folyamathalmaz, amely teljesíti, hogy:

- a folyamatok periódusaik növekvő sorrendjében vannak,
- bármely két folyamat periódusának aránya kisebb, mint 2.

Ekkor a *T* RM-ütemezhető, ha teljesül:

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq \sum_{i=1}^{n-1} \left\lceil \frac{T_{i+1}}{T_i} \right\rceil + 2\frac{T_1}{T_n} - n$$

Mivel a Scale Task Set algoritmus által elvégzett átalakítások után, az eredményül kapott folyamathalmaz teljesíti az előfeltételeket, így amennyiben a fenti egyenlőtlenség is igaznak bizonyul rá, akkor az eredeti folyamathalmaz is RM-ütemezhető.

Az R-BOUND a T-BOUND ütemezhetőségi feltételének legkisebb felsőhatárát használja az ütemezhetőség eldöntésére.

Legyen egy *n* elemű *T* folyamathalmaz, továbbá $r = P_n/P_1$, ahol P_1 és P_n a legkisebb és legnagyobb periódust jelentik, és $r < 2$. Ekkor a $B(r, n)$ lesz a T-BOUND feltétele alapján meghatározott processzor kihasználtság értéke.

$$B(r,n) = \sum_{i=1}^{n-1} \left\lceil \frac{P_{i+1}}{P_i} \right\rceil + \frac{2}{r} - n$$

Definiáljuk $B_{min}(r, n)$-t a $B(r, n)$ minimumaként, azaz a T-BOUND használatával elért processzor kihasználás értékének legkisebb felsőhatáraként.

Tekintsünk egy *n* elemű *T* folyamathalmazt, amely teljesíti, hogy:

- a folyamatok periódusaik növekvő sorrendjében vannak, ahol a legkisebb és legnagyobb periódus fix és ismert,
- bármely két folyamat periódusának aránya kisebb, mint 2.

A processzor kihasználtság értéke eme folyamathalmaz esetén minimális, ha két egymást követő folyamat periódusának aránya konstans. Ekkor $B_{min}(r, n)$:

$$B_{min}(r,n) = (n-1)\left(r^{1/n-1} - 1\right) + 2/r - 1$$

Végezetül az R-BOUND-MP a fentiek összességét a következőképp alkalmazza:

- A Scale Task Set algoritmus használatával átalakítja az eredeti folyamathalmazt.
- Az átalakított folyamatokat a First fit bin-packing módszer használatával hozzárendeli a processzorokhoz.
- A First fit módszer ebben az esetben pedig az R-BOUND feltételt használja annak eldöntésére, hogy egy processzor képes-e új folyamatot fogadni.
- Végezetül az eredeti folyamathalmaz folyamatait rendre hozzárendeli azon processzorhoz, amelyhez átalakított megfelelőjük rendelődött.

Mivel a T-BOUND ütemezhetőségi feltétele elvárja, hogy a folyamatok periódusaik szerint növekvőleg rendezve legyenek, így a hozzárendelésre használt módszer nem rendezheti át azokat. Ez azt jelenti, hogy a rendezést alkalmazó (például FFD) bin-packing algoritmusok ebben az esetben nem használhatóak.

A tapasztalati eredmények azt mutatják, hogy eme multiprocesszor ütemezési megközelítés képes elérni akár 96%-os processzor kihasználtságot nagyszámú folyamat esetén, amellett, hogy hatékonyan számítható.

Az R-BOUND használható továbbá magasabb multiprocesszoros ütemezhetőség elérésére, amikor az RM algoritmust aperiodikus folyamatok periodikus szerver (speciális periodikus folyamat, amely a beérkező aperiodikus folyamatok kezelésére szolgál) beillesztésével próbáljuk meg kezelni, vagy amikor átmeneti működési hibákat kell a rendszernek elviselnie.

2.3.3 A globális és szétválasztásos módszerek összevetése

A globális stratégiák számos hátrányos tulajdonsággal rendelkeznek szétválasztásos társaikhoz képest. Utóbbiak esetén kisebb ütemezési többletköltséggel kell számolni, mint a globális módszerben, mivel a folyamatokat nem kell átvinni egyik processzorról a másikra. Továbbá a szétválasztásos módszerek egyprocesszoros problémákká redukálják a feladatot, így jól ismert és fejlesztett algoritmusokat alkalmazhatunk minden processzoron.
Ugyanakkor a szétválasztásos módszer két negatív következménnyel is rendelkezik. Először is, az optimális processzor-folyamat hozzárendelések megtalálása NP-teljes feladat. Így a folyamatokat általában nem optimális heurisztikák segítségével kell szétválasztani. Másodszor léteznek olyan folyamat rendszerek, amelyek akkor és csak akkor ütemezhetőek, ha folyamataik nem kerülnek szétválasztásra.
A szakirodalomban a két módszer közül a szétválasztásos kapta a legnagyobb figyelmet. Ennek fő oka, hogy ez a módszer könnyen használható arra, hogy garantáljuk a futás idejű teljesítményt. Egy új folyamat valamely processzorhoz rendelése során egyprocesszoros ütemezhetőségi tesztet alkalmazva elérhető, hogy futási időben minden folyamat teljesíthesse határidejét. Míg a szétválasztás optimális megoldásának elérése kiszámíthatóság szempontjából elérhetetlennek látszik, számos hatékony heurisztikus módszert dolgoztak ki. Ezek mindegyike bin-packing módszereken alapul, teljesítmény garanciákat biztosít és meglehetősen jó átlagteljesítmény elérésére képes, miközben polinomiális időben kiszámítható.
A globális módszer látványosan kevesebb figyelmet kapott, elsősorban a következő korlátai következtében. Először is, nem létezett hozzá kellően hatékony ütemezhetőségi teszt. Az egyedüli ismert szükséges és elégséges ütemezhetőségi teszt kiszámítása exponenciális időt igényel. Másodszor, nem létezett optimális prioritás hozzárendelési séma a globális módszerhez. Továbbá az alacsony kihasználási tényezővel rendelkező folyamathalmazok globálisan ütemezhetetlenek RM segítségével (Dhall effektus).

Ezen jellemzőket figyelembe véve sokszor alkalmaznak *hibrid szétválasztásos/globális* stratégiákat a gyakorlatban (pl.: RM-FFDU+adaptiveTkC). Egy ilyen hibrid megoldás használatának oka, hogy a globális módszer segítségével növelhetjük a rendszer kihasználtságát anélkül, hogy veszélyeztetnénk a szétválasztásosan ütemezett folyamatoknak biztosított garanciákat. Továbbá a hibrid megoldás átlagos környezetben képes jobb teljesítményt nyújtani a csak szétválasztásos ütemezési módoknál.
Az RM-FFDU+adaptiveTkC a következőképpen működik:

- Ahány folyamatot csak lehet, szétválasztásosan ütemezünk az RM-FFDU segítségével (lokális prioritások hozzárendelése mellett).
- A fennmaradó folyamatokhoz (ha vannak) az adaptiveTkC séma segítségével rendelünk prioritásokat.
- Minden processzor rendelkezik egy lokális várakozási listával a szétválasztásos folyamatoknak, és van egy közös várakozási lista a globálisan ütemezett folyamatok számára.
- Minden processzor a saját lokális listájából futtat folyamatot, amennyiben az nem üres. Egyébként pedig a globális listából választ.

A két fő módszert egymással szembe állító irodalmak mindegyike elhanyagolhatónak tekinti a preemptivitás és a folyamatmigráció költségét. Azonban e tényezők jelentős befolyással bírhatnak a valós világban működő rendszereken.
A szétválasztásos módszer esetén nem történhet migráció a különböző processzorok között. A globális eljárás egyik legnagyobb hátulütőjének pedig pontosan e migráció létét tartják, okozott költségei miatt. Ez a költség valóban jelentős lehet, ha a processzor-folyamat hozzárendelés tetszőlegesen következik be. Legrosszabb esetben megtörténhet az is, hogy az m legnagyobb prioritású folyamat, megszakítása után, más processzorokhoz kerül hozzárendelésre ismét, annak ellenére, hogy ugyanazon m folyamatról van szó, amely a megszakítás előtt is futott. Ez természetesen nagymértékű felesleges migrációs költséget indukál. Így érdemes olyan processzor-folyamat hozzárendelési módszert alkalmazni, amely tekintettel van erre az eshetőségre is, ezzel minimalizálva a megszakítások (preemptivitás, migráció) számát. Egy ilyen eljárás alapötlete, hogy megpróbálja meghatározni az éppen futtatásra kijelölt folyamatok között azokat, amelyek már eddig is végrehajtás alatt álltak.

Ezután megpróbálja a talált folyamatokat azon processzorokhoz rendelni ismét, amelyekhez korábban voltak.

Az alábbi egy ilyen megoldást leíró algoritmus (*preemption-aware*):

Input:

- Legyen T_{before} azon folyamatok halmaza, amelyek épp futtatva voltak a megszakítás pillanatában az *m* processzoron.
- Minden p_i processzoron az ilyen folyamat a $T_{i,j,before}$ folyamat lesz.
- Legyen $T_{highest}$ azon folyamatok halmaza, amelyek futásra lettek kijelölve.

Output:

- Minden p_i processzoron a $T_{i,j,after}$ folyamat lesz kijelölve futásra.

1. $E = \{p_i \mid (T_{i,j,before} \neq nemLétező) \wedge (T_{i,j,before} \in T_{highest})\}$;
2. FOR EACH $p_i \in E$
3. $T_{i,j,before}$ eltávolítása $T_{highest}$-ből;
4. $T_{i,j,after}$:= $T_{i,j,before}$;
5. FOR EACH $p_i \notin E$
6. IF $T_{highest} \neq ø$
7. egy tetszőleges T_j választása $T_{highest}$-ből;
8. T_j eltávolítása $T_{highest}$-ből;
9. $T_{i,j,after}$:= T_j;
10. ELSE
11. $T_{i,j,after}$:= *nemLétező*;

A preemptivitás mértékét tekintve a preemption-aware algoritmust alkalmazó globális módszer jobban teljesít, mint a szétválasztásos módszerek. Ennek oka, hogy szétválasztásos esetben egy beérkező magasabb prioritású folyamat minden esetben megszakítja az alacsonyabb prioritású futó folyamatot. Globális esetben a beérkező folyamatnak lehetősége van más (akár éppen szabad) processzorhoz kerülnie.

Hasonló jelenség tapasztalható, ha más globális módszerekkel vetjük össze a preemption-awaret alkalmazó eljárást. Ekkor egy magasabb prioritású folyamat véget érése esetén az alacsonyabb prioritású folyamatok processzor hozzárendelései változatlanok maradhatnak.

Annak ellenére, hogy korábban a globális megközelítést a szétválasztásosnál rosszabbnak tekintették, bizonyítást nyert, hogy jobb prioritás hozzárendelés és preemptivitás kezelés megléte esetén ez a hiedelem nem szükségszerűen igaz. Ugyanakkor a szétválasztásos

módszer adta futási idejű garanciák, és számos esetben hatékonyabb ütemezhetőség sem vitatható. Bár a két módszert ötvöző hibrid megoldás is meglehetősen életképesnek bizonyulhat a valós idejű rendszerekben.

2.3.4 FUZZY algoritmus

Az úgynevezett Fuzzy logika alkalmazási területe *periodikus folyamatok*nak, *gyenge valós idejű* multiprocesszoros rendszerekben való ütemezésével kapcsolatos.

A legtöbb kutatás a valós idejű rendszerek ütemezési megszorításait pontosaknak, jól meghatározottaknak tekinti. Azonban számos környezet létezik, amelyekben eme paraméterek értéke bizonytalan, határozatlan. Ezen bizonytalanság okán alkalmazható a fuzzy logika annak eldöntésére, hogy a rendszerben megjelenő kérések milyen sorrendben kerüljenek végrehajtásra. Továbbá ennek eredményeképpen redukálható lehet az elvesztett kérések száma, és növelhető a rendszer kihasználtsága is. Ebben a megközelítésben a rendszer ütemezési paramétereit bizonytalan változóként (fuzzy variable) kezeljük.

A globális és szétválasztásos ütemezési algoritmusok elsősorban az időbeli megszorításokra koncentrálnak, azonban számos más implicit megszorítás is jelen lehet egy-egy környezetben. Ilyen tényező például a környezet teljes körű ismeretének hiánya vagy bizonytalansága, a kapott információk időben korlátos érvényessége valamint egyéb erőforrás megszorítások. Ezen megszorításokat gyakran csak részben tudjuk kielégíteni, ennek fokát előnnyel hasznosíthatjuk paraméterként az ütemezés döntési folyamataiban. Az így kapott paraméterek szintén modellezhetőek a fuzzy megközelítésben.

2.3.4.1 Fuzzy Következtetési Rendszer (FIS)

A Fuzzy logika (elmosódott, vagy „homálylogika”) a Boole logika egyfajta kiterjesztése. Kezeli a *részleges igazság* koncepcióját, amely jelzi, hogy egy tétel vagy előfeltétel milyen mértékben igaz. Míg a klasszikus logika szerint minden kifejezhető bináris kifejezésekkel (0 vagy 1, igaz vagy hamis), addig a fuzzy logika felcseréli eme igazságértékeket az igazság fokának értékével. Ez a látásmód hasonlít ahhoz, ahogy az ember képes mérlegelni és dönteni bizonytalan környezetekben is.

A Fuzzy Következtetési Rendszer (Fuzzy Inference System) elméleti felépítése meglehetősen egyszerű. Áll egy bemeneti, egy feldolgozó és egy kimeneti szakaszból.

A *bemeneti szakasz*ban a bejövő információk feltérképezése történik, mint például:

- hivatkozások gyakorisága
- hivatkozások aktualitása
- …stb.

Teszi mindezt a megfelelő hozzátartozási függvényértékek és igazságértékek megállapítása érdekében.

A *feldolgozó szakasz*ban alkalmazásra kerülnek a szükséges (bemenő adatoktól függő) szabályok és előállnak a megfelelő eredmények. Végezetül ezeket az eredményeket egységes formába hozzuk.

A *kimenő szakasz* pedig ezeket az egységesített eredményeket valamilyen meghatározott specifikus kimeneti értékké konvertálja.

A *hozzátartozási függvény* a fuzzy (elmosódott) halmazok esetén hasonló a klasszikus halmazoknál alkalmazott indikátor függvényhez. Ez egy görbét határoz meg, amely azt hivatott jelölni, hogy a bemeneti névtér egy-egy pontja milyen hozzátartozási értékkel áll kapcsolatban. Ezt az értéket nevezik az „igazság fokának" (degree of truth), amely a 0 és 1 közötti valós szám lehet. A legáltalánosabb ábrázolása a hozzátartozási függvényeknek háromszöges, de trapéz vagy haranggörbe alakzatokat is használnak. A bemeneti névteret gyakran a tárgyalt univerzumnak is nevezik.

A feldolgozó szakasz másik elnevezése a *következtetető motor*. Ez logikai szabályok egy gyűjteményén alapszik, amelyek általában „HA-AKKOR" formájú állításokként vannak jelen. A „HA" részt előzménynek (antecedent), az „AKKOR" részt következménynek (consequent) nevezik. Egy tipikus fuzzy következtető rendszer ilyen szabályok tucatjait tartalmazza, melyeket egy úgynevezett *ismeretbázis*ban (knowledgebase) tárol.

Például: HA a *pontatlanság kritikus* AKKOR *prioritás* legyen *nagyon magas.* Itt a *pontatlanság* és a *prioritás* nyelvi változók, a *kritikus* és a *nagyon magas* pedig nyelvi kifejezések. Minden nyelvi kifejezés egy-egy hozzátartozási függvénnyel áll kapcsolatban.

Egy következtető rendszer megpróbálja feldolgozni a bemenő információkat, és rendelkezésére álló ismeretbázisának segítségével előállítani valamilyen kimenetet. Ezt a műveletet öt lépésben végzi el:

1. Bemenet „elmosódottá tétele” (fuzzifying)
    - Hozzátartozási függvények segítségével meghatározza, hogy egy-egy bemenet milyen mértékben tartozik a megfelelő fuzzy halmazhoz.
2. Fuzzy operátorok alkalmazása
    - A bemenetek elmosódottá tétele által ismerté válik, hogy a különböző szabályok előzmény tagjainak egy-egy része milyen mértékben kerül kielégítésre.
    - Ha egy szabály előzmény része több tagból áll, akkor fuzzy operátor kerül alkalmazásra, amely a teljes előzmény eredményét reprezentáló értéket állít elő.
3. Következtető módszerek alkalmazása
    - A szabályok előzmény értékének függvényében módosításra kerül a kimeneti fuzzy halmaz.
4. Minden kimenet összeállítása
    - A különböző szabályok kimenetét reprezentáló fuzzy halmazok egyesítése egyetlen fuzzy halmazzá.
5. Kimenetek „élesítése” (defuzzifying)
    - Az egyesített kimeneti fuzzy halmaz leképezése egyetlen értékre.

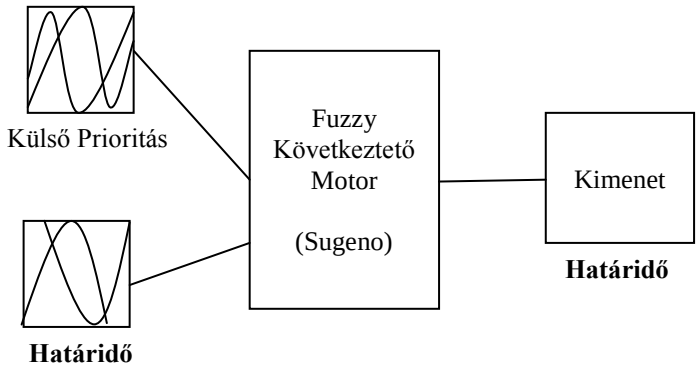
Két egyszerű következtető rendszer létezik. Az elsőt Mamdani fuzzy következtető módszerének nevezik, melyet Ebrahim Mamdani tervezett 1975-ben. A másik ilyen módszer pedig az 1985-ben bemutatott Takagi-Sugeno-Kang metódus. Ez a két megoldás számos szempontból hasonlít egymásra, mint például a bementi információk elmosódottá tételében és a fuzzy operátorok meghatározásában.

A legfőbb különbség Mamdani és Sugeno rendszere között, hogy az utóbbi által igényelt kimeneti hozzátartozási függvények lineárisak vagy konstansok. Mamdani rendszere viszont fuzzy halmazt vár el eme függvények esetén.
A Sugeno módszer előnyei:

- Hatékonyan számítható – ami a valós idejű rendszereknél alapvető fontosságú.
- Matematikai analízis szempontjából jól felszerelt.
- Jól optimalizálható.

2.3.4.2 A Modell

Az alábbi ábrán a következtető rendszer látható.



A modellben a bemeneti szint két nyelvi változót tartalmaz. Az első ilyen a külső prioritás, amely a külvilág által a folyamathoz rendelt prioritás, ami statikus. Egy lehetséges értéke lehet például a periódusidő, ahogy a Rate Monotonic algoritmus esetében. A másik bemeneti változó lehet a határidő, ahogy az ábrán látható. Ezt egyszerűen kicserélhetjük a pontatlanságra, várakozási időre vagy más, ütemező algoritmussal kapcsolatos, mérőszámra. A különböző paraméterek más-más reakciót válthatnak ki a rendszer részéről. Az egyedüli dolog, amit figyelembe kell venni, hogy a bemeneti változók megváltoztatása a kapcsolódó hozzátartozási függvényekben is ennek megfelelő változást idéz elő.

Attól függően, hogy a pontatlanságot vagy a határidőt tekintjük, továbbá, hogy globális vagy szétválasztásos módszert alkalmazunk, három különböző algoritmust különböztethetünk meg:

- FGEDF (Fuzzy Global EDF)
- FGMLF (Fuzzy Global MLF)
- FPEDF (Fuzzy Partitioned EDF)
- FPMLF (Fuzzy Partitioned MLF)

A rendszer kimenete egyfajta prioritás, amelyet paraméterként használhatunk fel a későbbi döntéshozás során.

A Fuzzy szabályok megpróbálják kezelni ezeket a bemenő paramétereket, amelyeket a valós világ határozhat meg a rendszer működése folyamán. Néhány ilyen szabály lehet:

- HA (*Külső prioritás* magas) ÉS (*Pontatlanság* kritikus) AKKOR (*Prioritás* nagyon magas)
- HA (*Külső prioritás* normális) ÉS (*Pontatlanság* kritikus) AKKOR (*Prioritás* magas)

- HA (*Külső prioritás* nagyon alacsony) ÉS (*Pontatlanság* kritikus) AKKOR (*Prioritás* normális)
- HA (*Külső prioritás* magas) ÉS (*Pontatlanság* megfelelő) AKKOR (*Prioritás* normális)
- HA (*Külső prioritás* nagyon alacsony) ÉS (*Pontatlanság* megfelelő) AKKOR (*Prioritás* nagyon alacsony)

A fuzzy következtető rendszerben a szabályok száma közvetlen hatással van az időbonyolultságra. Tehát minél kevesebb a szabály, annál jobb a teljesítmény.

Az FGEDF algoritmus:

Ciklus

A rendszer minden CPU-jára vonatkozólag tegyük az alábbiakat:

1. Minden *T* futásra kész folyamat (amely még nem fut) esetén töltsük be annak külső prioritását és határidejét a következtető modulba. Tekintsük az így kapott kimenetet a *T* folyamat prioritásának.
2. Futassuk a folyamatot az általa elérhető legnagyobb prioritáson mindaddig, amíg valamilyen ütemezési esemény be nem következik (folyamat befejeződése, új folyamat érkezése).
3. Frissítsük a rendszer állapotát (határidők, stb.).

Ciklus Vége

Az FGMLF algoritmus megegyezik a fenti algoritmussal, azzal az eltéréssel, hogy a határidőt pontatlanságra cseréljük.

Az FPEDF algoritmus:

Ciklus

1. Minden *T* futásra kész folyamat (amely még nem futott egy másik CPU-n sem) esetén töltsük be annak külső prioritását és határidejét a következtető modulba. Tekintsük az így kapott kimenetet a *T* folyamat prioritásának.
2. Futassuk a folyamatot az általa elérhető legnagyobb prioritáson mindaddig, amíg valamilyen ütemezési esemény be nem következik (folyamat befejeződése, új folyamat érkezése).
3. Frissítsük a rendszer állapotát (határidők, stb.).

Ciklus Vége

Az FPMLF algoritmus megegyezik a fenti algoritmussal, azzal az eltéréssel, hogy a határidőt pontatlanságra cseréljük.

Kísérletek szerint a határidők használata paraméterként valós idejű multiprocesszoros rendszerekben ígéretesebb, mintha a pontatlanságot használnánk. Továbbá a szétválasztásos módszer jobban teljesít globális társánál fuzzy valós idejűség mellett.

Összefoglalás

Dolgozatomban a többprocesszoros ütemezés tárgykörének két nagyobb területét fejtettem ki részletesen. Az első rész a klasszikus modellekről szólt. Ebben a részben megadtam a többprocesszoros ütemezési problémák specifikálásának módjait, céljait, illetve a lényegesebb alapfogalmak definícióját. A többprocesszoros rendszerek mindhárom csoportjára vonatkozólag láthattunk ütemezési eljárásokat. Ezek az eljárások az esetek többségében az optimalitásra törekedtek és nehezen kiszámíthatónak bizonyultak. Kifejtettem miként lehet a preemptivitás, a precedencia és a release/due time problémákat kezelni. Természetesen számos egyéb algoritmus, módszer létezik a témakörre vonatkozólag, melyekre nem tértem ki részletesen. Ezek egyike a Shop (Open Shop, Flow Shop) ütemezések tárgyköre, melyek egy másabb jellegű interpretációját adják ennek a problémának.

A második részben a valós idejű rendszerekkel foglalkoztam. Vázoltam működésük módját, leírásuk módozatait valamint két optimális ütemezési eljárást, melyek a többprocesszoros rendszerek esetén is meghatározó szerepet töltenek be. A multiprocesszoros ütemezésre vonatkozólag részletesebben megvizsgáltam a két fő irányvonal (globális és szétválasztásos) alapötletét, heurisztikáit, előnyeit/hátrányait. Az összevetésben láthattuk, hogy a két irányvonal képes egymást felülmúlni annak ellenére, hogy kezdetben a szétválasztásos módszert tekintették erősebbnek. Továbbá egy köztes hibrid megoldás is bemutatásra került, amely igyekszik a kétfajta működési elv előnyeit kiaknázni. A heurisztikák és algoritmusok tárgyalásánál főképpen a fix prioritásos módszereket tekintettem, de bemutattam az egyik legérdekesebb dinamikus prioritás alapú, fuzzy logikát alkalmazó technikát.

A dolgozatban bemutatott algoritmusokon túl számos egyéb módszer is létezik a többprocesszoros ütemezés megvalósítására. Ilyenek például a legmodernebb genetikus és neurogenetikus algoritmusok, vagy a rajintelligencián alapuló módszerek. Előbbiek a természetben végbemenő evolúciós folyamatok mintájára működnek, utóbbiak pedig a rajokban élő élőlények viselkedését veszik alapul. Szintén külön odafigyelést érdemlő témakör még a precedencia megszorítások megjelenése a valós idejű rendszerekben, illetve az aperiodikus és szórványos folyamatok kezelése, de tárgyalásuk nagyon messzire vezet az eredeti multiprocesszoros ütemezés témakörétől. Így eme módszerekre összetettségük miatt a dolgozat keretein belül részletesen nem tértem ki.

Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani témavezetőmnek, Dr. Fazekas Gábor egyetemi docensnek a dolgozat megírásához nyújtott hasznos tanácsaiért, támogatásáért illetve az általa biztosított szakirodalomért. Valamint szeretném megköszönni családomnak a bíztatást, melyet a dolgozat megírása során adtak.

Irodalomjegyzék

[1] J.K. Lenstra and A.H.G. Rinnooy Kan, "*An introduction to multiprocessor scheduling*", Technical Report, CWI, Amsterdam, 1981.

[2] Sahni, S, "*Preemptive Scheduling with due dates*", Operation Res. v.27, 1979.

[3] Brucker, Peter, "*Scheduling Algorithms*", 5th ed., 2007.

[4] Mohammadi, A. and Akl, S.G., "*Scheduling algorithms for real-time systems*", Technical Report No. 2005-499, School of Computing, Queen's University, Kingston, Ontario, July 2005.

[5] S. Baruah, N. Cohen, G. Plaxton, and D. Varvel, "*Proportionate progress: A notion of fairness in resource allocation*" Algorithmica , Volume 15, Number 6, pp. 600-625, June, 1996.

[6] A. Burns, *"Scheduling hard real-time systems: A review,"* Software Engineering Journal, Number 5, May, 1991.

[7] C. M. Krishna and K. G. Shin, *"Real-Time Systems,"* MIT Press and McGraw-Hill Company, 1997.

[8] B. Andersson and J. Jonsson. *"Some insights on fixed-priority preemptive non-partitioned multiprocessor scheduling"* In *Proc. of the IEEE Real-Time Systems Symposium – Workin- Progress Session*, Orlando, Florida, November 27–30, 2000.

[9] Y. Oh and S. H. Son. *"Fixed-priority scheduling of periodic tasks on multiprocessor systems"* Technical Report 95-16, Department of Computer Science, University of Virginia, March 1995.

[10] S. Lauzac, R. Melhem, and D. Moss´e. *"An efficient RMS admission control and its application to multiprocessor scheduling"* In *Proc. of the IEEE Int'l Parallel Processing Symposium*, pages 511–518, Orlando, Florida, March 1998.

[11] C. L. Liu and J.W. Layland. *"Scheduling algorithms for multiprogramming in a hard-real-time environment" Journal of the Association for ComputingMachinery*, 20(1):46–61, January 1973.

[12] B. Andersson and J. Jonsson. *"Fixed-priority preemptive multiprocessor scheduling: To partition or not to partition"* In *Proc. of the IEEE Int'l Conference on Real-Time Computing Systems and Applications*, Cheju Island, Korea, December 12–14, 2000.

[13] Sabeghi, M., Deldari, H., and Khajouei, S. *"A fuzzy algorithm for scheduling periodic tasks on multiprocessor soft real-time systems"* In *Proceedings of the 17th IASTED international Conference on Modelling and Simulation* 2006.

[14] Mamdani E.H., Assilian S., *"An experiment in linguistic synthesis with a fuzzy logic controller"*, International Journal of Man-Machine Studies, Vol. 7, No. 1, pp. 1-13, 1975.

[15] *"Multiprocessor Scheduling: Theory and Applications"*, Edited by E. Levner 2007.

[16] http://en.wikipedia.org/wiki

[17] http://portal.acm.org

[18] http://www.computer.org