Operációs rendszerek feladatai

Számolási példák és algoritmusok

Operációs rendszerek (VIMIA219)

Készítették: Darvas Dániel, Horányi Gergő, Jámbor Attila, Micskei Zoltán, Szabó Tamás

Utolsó módosítás: 2012. június 6.

Verzió: 1.5.7

Budapesti Műszaki és Gazdaságtudományi Egyetem
Méréstechnika és Információs Rendszerek Tanszék

Tartalom

1 Ütemezési algoritmusok[1]

Az ütemezés feladata kiválasztani a rendszerben lévő futásra kész feladatok közül a futót. A feladatokat többnyire feladatsorokban tároljuk.

Mivel az ütemezőnek többféle, egymásnak akár ellentmondó szempontnak is meg kéne felelnie, ezért sokféle ütemezési algoritmust dolgoztak. Ezeket a következő főbb csoportokba sorolhatjuk:

- *Nem preemptív ütemező*: A futó feladattól nem vehető el a CPU, neki kell lemondania arról vagy eseményre kell várnia.
- *Preemptív ütemező*: A futó feladattól az OS elveheti a futási jogot.

1.1 Ütemezési algoritmusok jellemzése

Az alább felsorolt metrikák az ütemezési algoritmusok jellemzésére szolgálnak.

- *CPU kihasználtság*: A hasznos munkával töltött idő aránya az összes időhöz képest.

$$\frac{\sum t_{CPU,hasznos\ munka}}{\sum t_{CPU}} \cdot 100\ [\%]$$

  A CPU kihasználtságát a kontextus váltáshoz, illetve az ütemezéshez szükséges idő ronthatja le. Kontextus váltás akkor történik, ha egy a processzoron egy másik folyamat kezd el futni (beleértve az 1. folyamat futásának kezdetét). Ütemezés akkor történik, amikor az ütemező algoritmusnak kell futnia. Ez utóbbival csak a RR ütemezés esetében számolunk.

- *Átbocsátó képesség*: Adott időegység alatt elvégzett feladatok száma.

$$\frac{elvégzett\ munkák\ száma}{idő}\ \left[\frac{1}{s}\right]$$

- *Várakozási idő*: Az összes idő, amit a feladat várakozással töltött. A munka jellegéből és a végrehajtás részleteiből adódóan statisztikai ingadozás lehet, ezért átlagos várakozási időt számolunk leggyakrabban.

$$t_{várakozás} = t_{futásra\ kész} + t_{más,nem\ futó\ állapotok}\ [s]$$

- *Körülfordulási idő:* Egy feladatra vonatkozóan a rendszerbe helyezéstől a teljesítésig eltelt idő. A munka jellegéből és a végrehajtás részleteiből adódóan statisztikai ingadozás lehet, ezért átlagos körülfordulási időt számolunk leggyakrabban.

$$t_{körülfordulás} = t_{CPU,végrehajtás} + t_{várakozás}\ [s]$$

- *Válaszidő*: A feladat megkezdésétől az első kimenetek produkálásáig eltelt idő. A munka jellegéből és a végrehajtás részleteiből adódóan statisztikai ingadozás lehet, ezért átlagos válaszidőt számolunk leggyakrabban.

1.2 Klasszikus ütemezési algoritmusok

A következő alapvető ütemezési algoritmusokkal foglalkoztunk az előadás során

[1] Jelen segédlet csak a legfontosabb fogalmakra, magyarázatokra szorítkozik. Az előadás diasoraiban az elméleti részről részletes leírás található.

- *FCFS (First Came First Served)*: A legrégebben várakozó feladatot választja ki végrehajtásra. Nem preemptív algoritmus. A várakozási sor azonos végéről veszi ki mindig a feladatokat.
- *RR (Round Robin)*: Az időosztásos rendszerek számára találták ki. Minden egyes feladat egy meghatározott időszeletig futhat, utána az ütemező elveszi a futási jogot a feladattól. Amikor az ütemező elveszi a futási jogot egy feladattól, akkor egy FIFO (First In First Out) listába helyezi be azt. Jellegéből adódóan preemptív algoritmus. Ha egy folyamat az időszelet lejárta előtt fejezi be futását, akkor az ütemező algoritmusnak le kell futnia (és ez a CPU kihasználtság metrika értékét csökkenti).
- Prioritásos ütemezők:
    - *SJF (Shortest Job First)*: A legrövidebb löketidejű feladatot választja ki a futásra készek közül. Nem preemptív algoritmus.
    - *SRTF (Shortest Remaining Time First)*: Az SJF preemptív változata, amely, ha új folyamat válik futásra késszé, akkor megvizsgálja, hogy az aktuálisan végrehajtott vagy az új folyamat löketideje a kisebb és azt választja ki futtatásra.

A következőkben bemutatunk három egyszerű példát, ahol egy adott kiindulási állapotból megvizsgáljuk, hogy milyen eredményt adnak a különböző ütemezési algoritmusok.

Az ütemezési algoritmusok lefutását szemléltető ábrákon minden esetben alul található meg a várakozási sor tartalma.

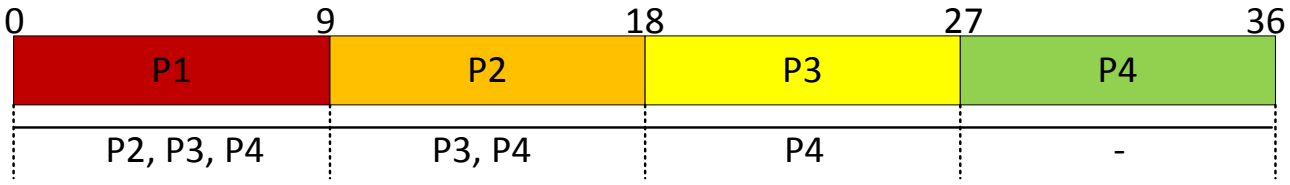
1.2.1 Számolási feladat I.

Egy rendszerbe az alábbi feladatok érkeznek be.

Feladat neve	Beérkezési idő (ms)	Löketidő (ms)
P1	0	9
P2	0	9
P3	0	9
P4	0	9

1.2.1.1 FCFS algoritmus

Az ütemezési algoritmus lefutása:



Az ütemezéssel kapcsolatos metrikák értéke:

Metrika	Érték
CPU kihasználtság[2]	4 context switch, (36,4 – 0,4) / 36,4 = 98,9 %
Körülfordulási idők átlaga	(9+18+27+36) / 4 = 22,5 ms
Várakozási idők átlaga	(0+9+18+27) / 4 = 13,5 ms
Válaszidők átlaga	(0+9+18+27) / 4 = 13,5 ms

[2] Számoljunk a továbbiakban mindig 0.1 ms ütemezési és kontextus váltás (cs – context switch) idővel.

1.2.1.2 SJF algoritmus

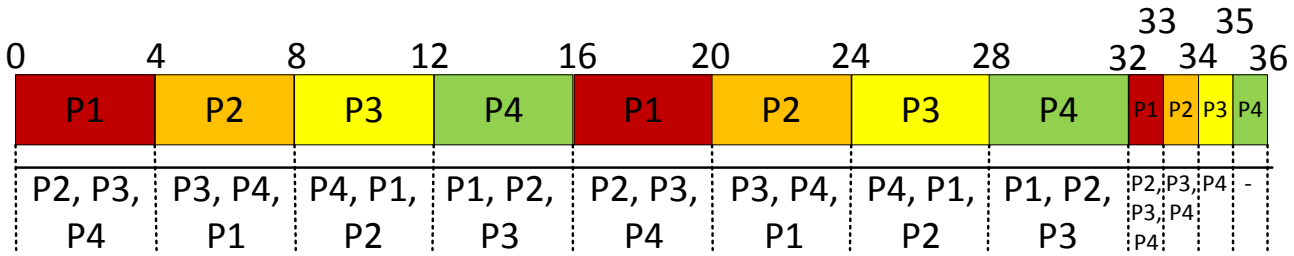
Ebben a konkrét feladatban az ütemezési algoritmus lefutása ugyanaz, mint az FCFS esetén, ebből adódik az is, hogy az algoritmust jellemző paraméterek is ugyan azok lesznek.

1.2.1.3 SRTF algoritmus

Ebben a konkrét feladatban az ütemezési algoritmus lefutása ugyanaz, mint az FCFS esetén, ebből adódik az is, hogy az algoritmust jellemző paraméterek is ugyan azok lesznek.

1.2.1.4 RR algoritmus

Az ütemezési algoritmus lefutása (4 ms-os időszelettel számolva):



Az ütemezéssel kapcsolatos metrikák értéke:

Metrika	Érték
CPU kihasználtság	12 context switch + 3 ütemezés, (37,5 – 1,5) / 37,5 = 96 %
Körülfordulási idők átlaga	(33+34+35+36) / 4 = 34,5 ms
Várakozási idők átlaga	(24+25+26+27) / 4 = 25,5 ms
Válaszidők átlaga	(0+4+8+12) / 4 = 6 ms

A kihasználtság kiszámításához az alábbi kontextus váltási időpontokat vettük figyelembe: 0 ms, 4 ms, 8 ms, 12 ms, 16 ms, 20 ms, 24 ms, 28 ms, 32 ms, 33 ms, 34 ms, 35 ms. Ütemező algoritmus futásának időpontjai: 33 ms, 34 ms, 35 ms.

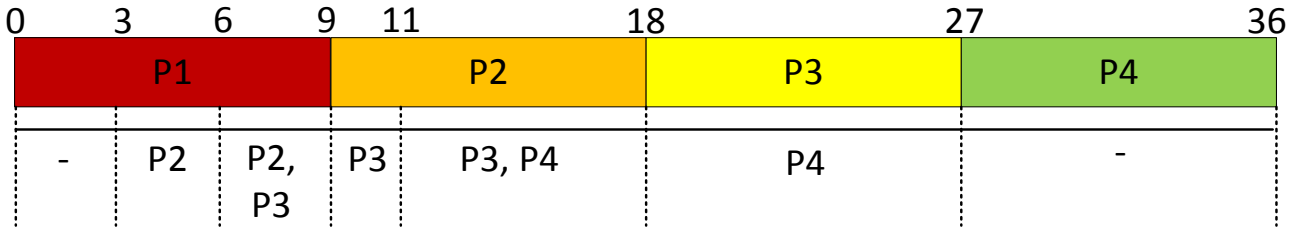
1.2.2 Számolási feladat II.

Egy rendszerbe az alábbi feladatok érkeznek be.

Folyamat neve	Beérkezési idő (ms)	Löketidő (ms)
P1	0	9
P2	3	9
P3	6	9
P4	11	9

1.2.2.1 FCFS algoritmus

Az ütemezési algoritmus lefutása:



Az ütemezéssel kapcsolatos metrikák értéke:

Metrika	Érték
CPU kihasználtság	(36,4 – 0,4) / 36,4 = 98,9 %
Körülfordulási idők átlaga	(9+15+21+25) / 4 = 17,5 ms
Várakozási idők átlaga	(0+6+12+16) / 4 = 8,5 ms
Válaszidők átlaga	(0+6+12+16) / 4 = 8,5 ms

1.2.2.2 SJF algoritmus

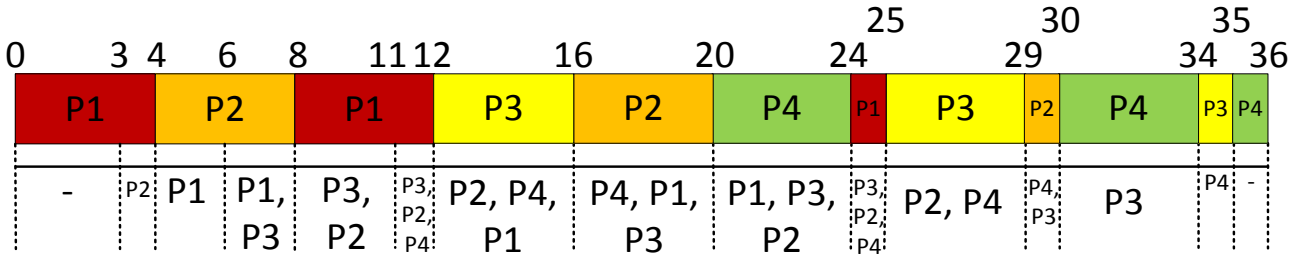
Ebben a konkrét feladatban az ütemezési algoritmus lefutása ugyanaz, mint az FCFS esetén, ebből adódik az is, hogy az algoritmust jellemző paraméterek is ugyan azok lesznek.

1.2.2.3 SRTF algoritmus

Ebben a konkrét feladatban az ütemezési algoritmus lefutása ugyanaz, mint az FCFS esetén, ebből adódik az is, hogy az algoritmust jellemző paraméterek is ugyan azok lesznek.

1.2.2.4 RR algoritmus

Az ütemezési algoritmus lefutása (időszelet 4 ms):[3]



Az ütemezéssel kapcsolatos metrikák értéke:

Metrika	Érték
CPU kihasználtság	12 context switch + 3 ütemezés, (37,5 – 1,5) / 37,5 = 96 %
Körülfordulási idők átlaga	(25+27+29+25) / 4 = 26,5 ms
Várakozási idők átlaga	(16+18+20+16) / 4 = 17,5 ms
Válaszidők átlaga	(0+1+6+9) / 4 = 4 ms

[3] Itt érdemes megfigyelni például a 8-adik óraütésnél lezajlódó ütemezést. Mivel FIFO listát tartunk karban, a P1 már korábban bekerül a listába (amikor a 4-edik óraütésnél elveszi tőle a futási jogot az ütemező) mint a P3, ezért kapja meg megint ő a futási jogot és csak 12-ben fogja megkapni P3 a futás jogát.

A kihasználtság kiszámításához az alábbi kontextus váltási időpontokat vettük figyelembe: 0 ms, 4 ms, 8 ms, 12 ms, 16 ms, 20 ms, 24 ms, 25 ms, 29 ms, 30 ms, 34 ms, 35 ms. Ütemező algoritmus futásának időpontjai: 25 ms, 30 ms, 35 ms.

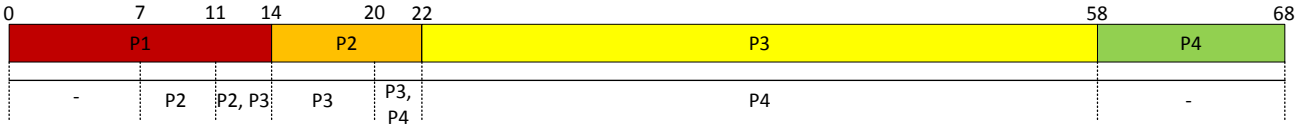
1.2.3 Számolási feladat III.

Egy rendszerbe az alábbi feladatok érkeznek be.

Folyamat neve	Beérkezési idő (ms)	Löketidő (ms)
P1	0	14
P2	7	8
P3	11	36
P4	20	10

1.2.3.1 FCFS algoritmus

Az ütemezési algoritmus lefutása:

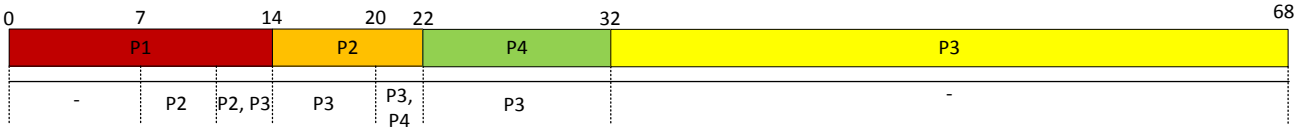


Az ütemezéssel kapcsolatos metrikák értéke:

Metrika	Érték
CPU kihasználtság	4 context switch, (68,4 – 0,4) / 68,4 = 99,42 %
Körülfordulási idők átlaga	(14+15+47+48) / 4 = 31 ms
Várakozási idők átlaga	(0+7+11+38) / 4 = 14 ms
Válaszidők átlaga	(0+7+11+38) / 4 = 14 ms

1.2.3.2 SJF algoritmus

Az ütemezési algoritmus lefutása:



Az ütemezéssel kapcsolatos metrikák értéke:

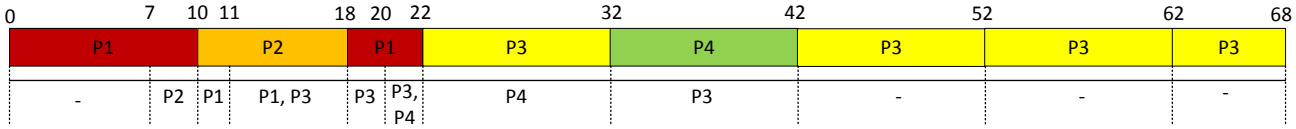
Metrika	Érték
CPU kihasználtság	4 context switch, (68,4 – 0,4) / 68,4 = 99,42 %
Körülfordulási idők átlaga	(14+15+57+12) / 4 = 24,5 ms
Várakozási idők átlaga	(0+7+21+2) / 4 = 7,5 ms
Válaszidők átlaga	(0+7+21+2) / 4 = 7,5 ms

1.2.3.3 SRTF algoritmus

Ebben a konkrét feladatban az ütemezési algoritmus lefutása ugyanaz, mint az SJF esetén, ebből adódik az is, hogy az algoritmust jellemző paraméterek is ugyan azok lesznek.

1.2.3.4 RR algoritmus

Az ütemezési algoritmus lefutása (időszelet 10 ms):



Az ütemezéssel kapcsolatos metrikák értéke:

Metrika	Érték
CPU kihasználtság	6 context switch + 3 ütemezés, (68,9 – 0,9) / 68,9 = 98,69 %
Körülfordulási idők átlaga	(22+11+57+22) / 4 = 28 ms
Várakozási idők átlaga	(8+3+21+12) / 4 = 11 ms
Válaszidők átlaga	(0+3+11+12) / 4 = 6,5 ms

A kihasználtság kiszámításához az alábbi kontextus váltási időpontokat vettük figyelembe: 0 ms, 10 ms, 18 ms, 22 ms, 32 ms, 42 ms. Ütemező algoritmus futásának időpontjai: 18 ms, 22 ms, 42 ms. A legutolsó ütemezési időpont egybeesik az időkeret leteltével, de ilyen esetben is számolunk az ütemező algoritmus futásával (a valóságban ezek nyilván már implementáció-függő részletek).

1.2.4 Példa vizsgafeladat

Egy rendszerben az alábbi feladatok érkeznek be az alábbi sorrendben:

Feladat	Beérkezési idő (ms)	CPU löket(ms)
P1	0	3
P2	3	10
P3	3	3
P4	6	6
P5	8	3

A determinisztikus modellezés módszerével vizsgálja meg, hogy a körbefordulási idő szempontjából az RR ütemezési algoritmus 4 vagy 10 ms időszelet mellet kedvezőbb-e a fenti feladatkészlet eseten? Az ütemezés időigényét hanyagolja el számításai során! Számolja ki a két esetre az átlagos körbefordulási időt! Magyarázza meg az eredményeket! Hogyan kell megválasztani az RR algoritmus eseten az időszeletet?

Megoldás

I. eset: 4 ms időszelet

Időtartam	FUT	Futást kezdte	Futást befejezi	Ready sor tartalma
0-3	P1	0	3	-
3-6	P2	3	-	P3(3)
6-7	P2	3	7	P3(3), P4(6)
7-8	P3	7	-	P4(6), P2(6)
8-10	P3	7	10	P4(6), P2(6), P5(3)
10-14	P4	10	14	P2(6), P5(3)
14-18	P2	14	18	P5(3), P4(2)
18-21	P5	18	21	P4(2), P2(2)
21-23	P4	21	23	P2(2)
23-25	P2	23	25	-

(A ready sor tartalma oszlopnál a feladat neve után a hátralévő löketidőt jelzi a zárójeles kifejezés.)

Átlagos körbefordulási idő: az egyes feladatok által a rendszerbe belépéstől a kilépésig eltöltött idő átlaga, a mértékegysége ms.

t_{ta}=((3-0)+(25-3)+(10-3)+(23-6)+(21-8))/5=(3+22+7+17+13)/5=62/5=12.4 ms

II. eset: 10 ms időszelet:

Időtartam	FUT	Futást kezdte	Futást befejezi	Ready sor tartalma
0-3	P1	0	3	-
3-6	P2	3	-	P3(3)
6-8	P2	3	-	P3(3), P4(6)
9-13	P2	3	13	P3(3), P4(6), P5(3)
13-16	P3	13	16	P4(6), P5(3)
16-22	P4	16	22	P5(3)
22-25	P5	22	25	-

t_{ta}=((3-0)+(13-3)+(16-3)+(22-6)+(25-8))/25=(3+10+13+16+17)/5=59/5=11.8 ms

A körülfordulási idő kismertekben nő a kisebb időszelet eseten, de az RR algoritmus elsősorban a válaszidő szempontjából kedvezőbb (interaktív feladatok). Az átlagos CPU löket (3+10+3+6+3)/5=25/5=5ms, ennek a 80%-ra kell beállítani az időszeletet a tapasztalati összefüggés alapján az ideális működéshez, ami pont a választott 4 ms. A 10 ms eseten az RR algoritmus lényegében átmegy a FIFO algoritmusba ennél a feladat készletnél, és ezért a P2 feltartja a többi feladat lefutását (konvoj hatás).

1.2.5 Önálló feladatok

1.2.5.1 Önálló feladat I.

Egy rendszerben az alábbi feladatok találhatók:

Feladat	Beérkezési idő (ms)	CPU löket (ms)
P1	0	3
P2	3	24
P3	3	3
P4	5	6
P5	8	3

A FIFO és az SRTF ütemező algoritmusok közül melyik nyújtja a kisebb átlagos körülfordulási időt az adott terhelésnél? Adja meg a körülfordulási idő definícióját és mértékegységét is! A megoldásban mutassa be, hogyan jutott az eredményre!

1.2.5.2 Önálló feladat II.

Egy rendszerben az alábbi feladatok találhatók:

Feladat	Beérkezési idő (ms)	CPU löket (ms)
P1	0	6
P2	3	21
P3	3	3
P4	5	6

Az SJF (3 pont) és Round-robin (5 pont) ütemező algoritmusok közül melyik nyújtja a kisebb átlagos várakozási időt a megadott terhelésnél? Válaszában adja meg pontosan, hogy melyik feladat mikor fut. A Round-robin esetén az időszelet 4 ms. Adja meg a várakozási idő definícióját és mértékegységét is! A megoldásban mutassa be, hogyan jutott az eredményre! Melyik ütemezőt használjuk a gyakorlatban, röviden indokolja a válaszát (2 pont)?

1.2.5.3 Önálló feladat III.

Egy rendszerben az alábbi feladatok érkeznek be az alábbi sorrendben:

Feladat	Beérkezési idő (ms)	CPU löket (ms)
P1	0	3
P2	3	10
P3	3	3
P4	6	6
P5	8	3

- A determinisztikus modellezés módszerével vizsgálja meg, hogy a körbefordulási idő szempontjából az RR ütemezési algoritmus 4 vagy 10 ms időszelet mellet kedvezőbb-e a fenti feladatkészlet esetén? Az ütemezés időigényét hanyagolja el számításai során!
- Gannt diagrammon ábrázolja a feladatok lefutását! (2-2 pont)
- Számolja ki a két esetre az átlagos körbefordulási időt! (2-2 pont)
- Magyarázza meg az eredményeket! Hogyan kell megválasztani az RR algoritmus esetén az időszeletet? (2 pont)

1.3 Tradicionális Unix ütemező

A tradicionális Unix ütemező algoritmus az Unix rendszerekben használt ütemezés tipikus példája. A következőkben leírtak a tradicionális ütemező egy konkrét implementációjának felelnek meg. A különböző megvalósítások ettől kisebb mértékben eltérhetnek.

1.3.1 A tradicionális ütemező működése

A tradicionális Unix ütemező prioritásos ütemező. Minden folyamathoz tartozik egy-egy prioritás. A folyamatok ütemezése azonban eltér attól függően, hogy milyen módban futnak.

Felhasználói mód	Kernel mód
preemptív azonos prioritás esetén round robin	nem preemptív
dinamikusan változó prioritású folyamatok	rögzített prioritású folyamatok

Ez alapján kernel módban egy futó folyamatot nem lehet kényszeríteni a CPU használat lemondására nagyobb prioritású folyamatok számára sem.

1.3.1.1 Prioritások

A folyamatok **prioritása** 0 és 127 közötti. Az alacsonyabb számérték magasabb prioritást jelöl. Tipikusan az 50-nél kisebb prioritások (0..49) a kernel módú folyamatok prioritásai, a felhasználói módú folyamatok prioritása pedig legalább 50. Ez alapján a legalacsonyabb prioritású kernel módú folyamat is hamarabb kap kiszolgálást, mint a legnagyobb prioritású felhasználói módú folyamat.

Kernel módú folyamatok prioritása

Kernel módban a folyamatok prioritása csak attól függ, hogy a folyamat milyen eseményre várakozik. *Például lemez I/O-ra várakozás esetén* `p_pri`*=20, terminál I/O-ra várakozás esetén* `p_pri`*=28.*

Felhasználói módú folyamatok prioritása

Felhasználói módban a folyamatok prioritása két dologtól függ: a felhasználó által adott külső prioritástól (ún. nice számtól) és a folyamat korábbi CPU használatától. A nice szám a felhasználó által megadható érték. A kisebb nice szám fontosabb folyamatot jelöl.

Leírók

A prioritások kiszámításához az ütemező az alábbi paramétereket tárolja a folyamatokról:

- **p_pri**: aktuális ütemezési prioritás
- **p_usrpri**: felhasználói módban érvényes prioritás
- **p_cpu**: a CPU használat mértékére vonatkozó szám
- **p_nice**: a felhasználó által adott nice szám

A `p_pri` paraméter alapján ütemezi az ütemező a folyamatot. A `p_usrpri` tárolja a felhasználói módú prioritást, amikor a folyamat kernel módban fut. Felhasználói módban futó program esetén `p_usrpri=p_pri`.

Prioritások karbantartása

Ezeket az adatokat az ütemező különböző periodicitással tartja karban.

- Minden óraütésnél az ütemező megnöveli a futó folyamat `p_cpu` értékét eggyel.
- Minden 100. óraütésnél az ütemező korrigálja a `p_cpu` és `p_usrpri` értékeket az alábbiak szerint:

```
p_cpu = p_cpu * KF, ahol KF egy korrekciós faktor (lásd később)
p_usrpri = 50 + p_cpu/4 + 2*p_nice
```

A korrekciós faktor a futásra kész folyamatok számától függ. $KF = \frac{2 \cdot FK}{2 \cdot FK + 1}$, ahol FK a futásra kész folyamatok száma. A `p_cpu` számításánál a kerekítés általános szabályai alkalmazandók.

Prioritási szintek

A folyamatok leírói láncolt listákban tárolja az ütemező. Azonban nincs külön lista mind a 128 prioritási szinthez, ehelyett a prioritási szintek négyesével össze vannak fogva. Külön listában szerepelnek azonban a kernel és felhasználói módú folyamatok. Emiatt a legalacsonyabb prioritású kernelfolyamatok láncolt listájába csak két prioritási szint tartozik (48–49), míg a legalacsonyabb prioritású felhasználói módú folyamatok listájában 6 prioritási szint szerepel (122..127).

Azaz egy listában az alábbi prioritású folyamatok szerepelnek: 0..3, 4..7, … 44..47, 48–49, 50..53, 54..57, …, 118..121, 122..127. Az azonos listában szereplő folyamatok azonos prioritásúnak tekintendők. *Például az 58-as és 60-as prioritású folyamatok azonos prioritásúnak számítanak (mindkettő az 58..61 listában van), de az 57-es prioritású folyamat magasabb prioritási szintű az 58-as és 60-as folyamatnál egyaránt.*

1.3.1.2 Futó folyamat váltása

Az ütemező **minden óraütésnél** megvizsgálja, hogy van-e az aktuálisan futó folyamatnál magasabb prioritású futásra kész folyamat. Ha igen, újraütemez. Emellett **minden 10. óraütésnél** akkor is vált futó folyamatot az ütemező, ha van az aktuálisan futóval azonos prioritású futásra kész folyamat (round robin).

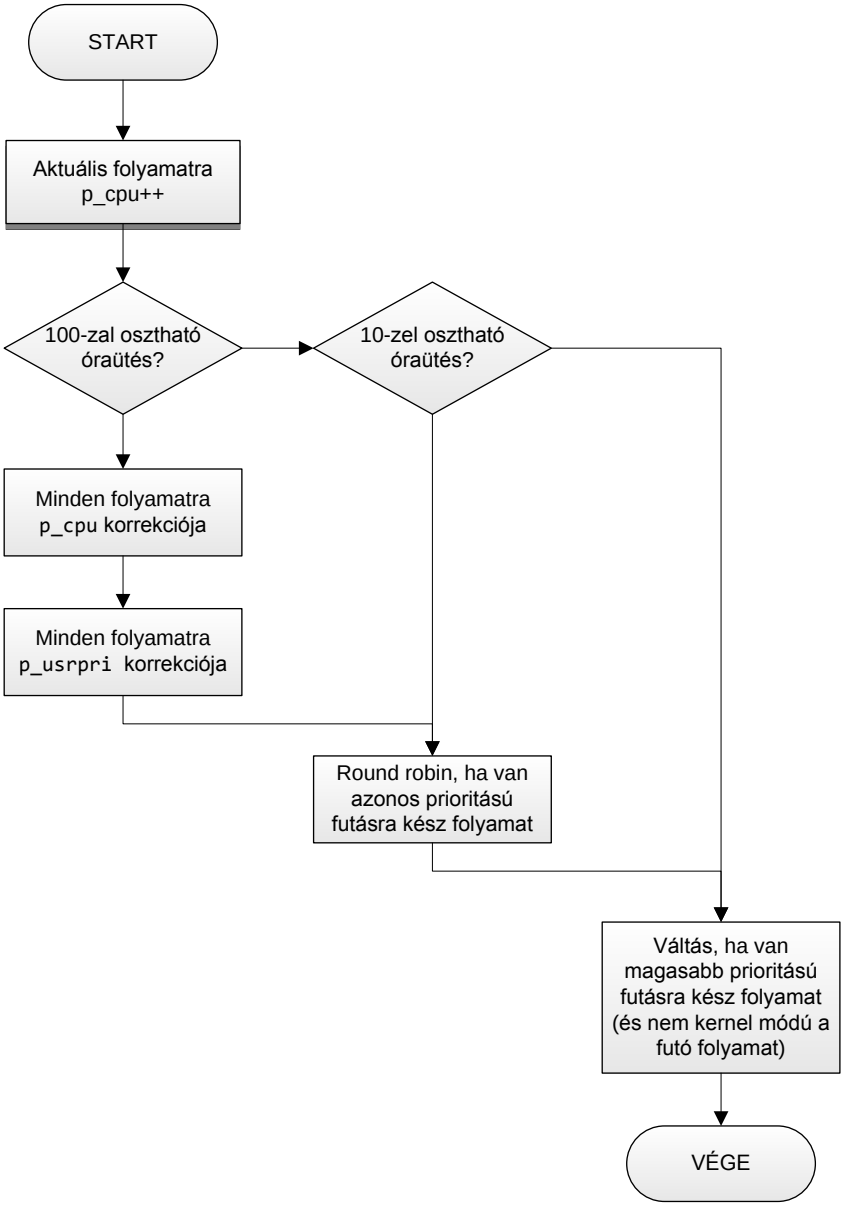
Az itt vázolt algoritmushoz képest számos eltérés mutatkozhat konkrét implementációkban. Egyes esetekben a kernel módú folyamatok prioritása nem 50-től kezdődik, hanem más értéktől. Bizonyos megvalósításoknál a korrekciós faktor értéke konstans. Megint más esetekben a minden 10. és 100. óraütésnél történő újraütemezések és korrekció más periodicitással történnek. Egyes implementációk pedig nem választják szét a kernel és felhasználói módú folyamatok láncolt listáit.

1.3.1.3 A működés összefoglalva

	futó folyamatra	**minden folyamatra**
minden óraütésre	`p_cpu++`	ha van magasabb prioritású futásra kész folyamat, akkor váltás (és a futó folyamat nem kernel módú)
minden 10. óraütésre	round robin ütemezés, ha van a futó folyamattal megegyező prioritású futásra kész folyamat	

minden 100. óraütésre		korrekciós faktor kiszámítása KF=(2*KF)/(2*KF+1)
		`p_cpu*=KF`
		`p_usrpri=50+p_cpu/4+2*p_nice`

1.3.1.4 A feladatok sorrendisége egy óraütés esetén



1.3.2 Példafeladat

Adott három folyamat: *A*, *B* és *C.* Minden folyamat csak felhasználói módban fut és mindegyik futásra kész. Kezdetben mindegyik folyamatra `p_pri` = 50. Az *A* és *B* folyamatra `p_nice` = 0, a *C* folyamatra `p_nice` = 10. Mindegyik folyamat a 0. óraütésnél indul, azaz kezdetben mindegyikre `p_cpu` = 0. A folyamatok beérkezési sorrendje: *A*, *B*, *C*.

óraütés	A folyamat		B folyamat		C folyamat		futó folyamat	
	p_pri = p_usrpri	p_cpu	p_pri = p_usrpri	p_cpu	p_pri = p_usrpri	p_cpu	óraütés előtt	óraütés után
kezdés	50	0	50	0	50	0		A

Mivel az A folyamat érkezett be elsőként, ezért ez kezd futni. A futó folyamatra minden óraütésnél p_cpu++.

1	50	1	50	0	50	0	A	A
2	50	2	50	0	50	0	A	A
...								
9	50	9	50	0	50	0	A	A

A 10. óraütésnél mivel van futásra kész folyamat az aktuálisan futó folyamattal azonos prioritással, ezért váltás lesz. A 10. óraütéstől a futó folyamat B lesz.

10	50	10	50	0	50	0	A	B
...								
19	50	10	50	9	50	0	B	B

A 20. óraütésnél mivel van futásra kész folyamat az aktuálisan futó folyamattal azonos prioritással, ezért váltás lesz. A 20. óraütéstől a futó folyamat C lesz.

20	50	10	50	10	50	0	B	C
...								
30	50	10	50	10	50	10	C	A
...								
40	50	20	50	10	50	10	A	B
...								
50	50	20	50	20	50	10	B	C
...								
60	50	20	50	20	50	20	C	A
...								
70	50	30	50	20	50	20	A	B
...								
80	50	30	50	30	50	20	B	C
...								
90	50	30	50	30	50	30	C	A
...								
99	50	39	50	30	50	30	A	A

A 100. óraütésnél a folyamatok p_cpu és p_pri értékeinek karbantartása következik.
*A korrekciós faktor KF=2*2/(2*2+1)=4/5=0,8, mivel két futásra kész folyamat van a rendszerben.*
*A folyamatra: p_cpu új értéke: p_cpu*KF = 40 * 0,8 = 32*
*p_pri=p_usrpri új értéke: 50+p_cpu/4+2*p_nice = 50+32/4+2*0 = 58*
*B folyamatra: p_cpu új értéke: p_cpu*KF = 30 * 0,8 = 24*
*p_pri=p_usrpri új értéke: 50+p_cpu/4+2*p_nice = 50+24/4+2*0 = 56*
*C folyamatra: p_cpu új értéke: p_cpu*KF = 30 * 0,8 = 24*
*p_pri=p_usrpri új értéke: 50+p_cpu/4+2*p_nice = 50+24/4+2*10 = 76*
A B folyamat magasabb prioritású az A-nál (a B a 54..57 sorban van, az A az 58..61 sorban), így a B fog futni.
(Megjegyeznő, hogy ha azonos prioritási sorban lennének, akkor is a B futna innentől, mivel a 100. óraütésnél round robin újraütemezés is van.)

100	58	32	56	24	76	24	A	B
101	58	32	56	25	76	24	B	B
...								
110	58	32	56	34	76	24	B	B
...								
120	58	32	56	44	76	24	B	B
...								
130	58	32	56	54	76	24	B	B
...								

A 200. óraütésnél esedékes p_pri újraszámításig csak a B folyamat fog futni.

1.3.3 Önálló feladat

Egy UNIX rendszerben öt egyforma ütemezési adattal (p_pri = 60, p_cpu=0, nice=5) rendelkező folyamat van futásra kész állapotban t=0 időpontban. A lenti táblázatban írja le a folyamatok ütemezését! Eredményét az ütemezési algoritmus tömör leírásával, képletekkel és számítási részletekkel is igazolja! Vegye figyelembe a tanult korrekciós faktort, valamint mindhárom UNIX ütemezési tevékenységet!

	A folyamat		**B folyamat**		**C folyamat**		**D folyamat**		**E folyamat**		**Az óraütés…**		
óraütés	**p_pri**	**p_cpu**	**p_pri**	**p_cpu**	**p_pri**	**p_cpu**	**p_pri**	**p_cpu**	**p_pri**	**p_cpu**	**alatt fut**	**után fut**	**óraütés**
kiindulás	60	0	60	0	60	0	60	0	60	0			kiindulás
1		1									**A**		1
2													2
3													3
9													9
10													10
20													20
21													21
90													90
91													91
99													99
100													100
101													101

Algoritmus (3 pont), képletek (4 pont), számítás (3 pont):

2 Feladatok együttműködése

Ha a rendszerünkben több feladat is van, amik egymással együttműködnek (pl. az egyik a másiktól vár eredményt, ugyanazt az erőforrást akarják ketten is használni, stb.), akkor az további megoldandó problémákat jelent az operációs rendszer számára. A *közös erőforrások* (shared resource) kezelése során többféle hiba is felléphet, ha nem gondoskodunk azok helyes kezeléséről. Ezek közül a legfontosabbak:

- versenyhelyzet (race condition),
- kiéheztetés (starvation),
- livelock,
- holtpont (deadlock),
- prioritás inverzió (priority inversion).

2.1 Holtpont

A holtpont definíciója a következő. „Egy rendszer feladatainak egy H részhalmaza holtponton van, ha a H halmazba tartozó valamennyi feladat olyan eseményre vár, amelyet csak egy másik, H halmazbeli feladat tudna előállítani." Tehát bizonyos feladatok nem tudnak továbblépni, a várakozó feladatok között egy kör alakul ki. Holtpont kezelésére többféle módszer közül választhatunk.

- *Strucc algoritmus*: nem foglalkozunk a problémával.
- *Holtpont észlelése és feloldása*: adott időnként megvizsgáljuk az erőforrás foglalásokat, és ha holtpontot veszünk észre, akkor azt megpróbáljuk megszüntetni (pl. felszámolunk egyes feladatokat).
- *Holtpont megelőzése*: tervezési időben gondoskodunk arról, hogy ne alakulhasson ki holtpont. Például kikötjük, hogy minden feladatnak egyszerre kell lefoglalnia az összes szükséges erőforrást.
- *Holtpont elkerülés*: az erőforrás foglalások előtt ellenőrizzük, hogy a foglalás hatására kialakulhat-e holtpont, és ha igen, nem engedélyezzük a foglalást.

A holtpont elkerülésre az egyik lehetséges módszer a bankár algoritmus használata, ezt nézzük most meg részletesebben.

2.1.1 Bankár algoritmus

2.1.1.1 Az algoritmus működése

Az algoritmus által használt adatstruktúrák és változók:

- **N** (egész szám): feladatok száma,
- **M** (egész szám): erőforrástípusok száma (mindegyik típusú erőforrásból több példányunk lehet),
- **MAXr** (M elemű vektor): az egyes erőforrástípusokból hány példány áll rendelkezésre a rendszerben.
- **MAX** ($N \times M$ mátrix): az egyes feladatoknak mennyi a maximális igénye az egyes erőforrástípusokból, az algoritmus futása során nem változik,
- **FOGLAL** ($N \times M$ mátrix): egy adott pillanatban az egyes feladatok hány példányt foglaltak le az egyes erőforrástípusokból,
- **SZABAD** (M elemű vektor): aktuálisan mennyi szabad példány van még az egyes erőforrástípusokból,
- **MÉG** ($N \times M$ mátrix): aktuálisan legfeljebb mekkora igényt nyújthatnak még be az egyes feladatok, értéke $\boldsymbol{MAX - FOGLAL}$,

- **KÉR** ($N \times M$ mátrix): azt tárolja, hogy az egyes feladatok hány példányt igényelnek jelenleg (ha a foglalások több lépésben történnek, vagy nem akarják a maximális igénynek megfelelő erőforráspéldányt elkérni)

Az algoritmus alapgondolata, hogy a beérkezett kérésekhez megnézi, hogy az a legrosszabb esetben is teljesíthető-e, biztos, hogy nem alakulhat ki holtpont. Az algoritmus a holtpont lehetőségét mutatja ki, a végrehajtás pontos sorrendje dönti el, hogy ténylegesen kialakul-e a holtpont. Az algoritmus főbb lépései:

1. Adott kérés ellenőrzése (van-e egyáltalán ennyi szabad erőforrás).
2. Nyilvántartás átállítása az új állapotra (kérés rögzítése a szabad és foglalt mátrixokba).
3. Biztonságosság vizsgálata (kialakulhat-e az új állapotban holtpont):
    a. Kezdőértékek beállítása
    b. Továbblépésre esélyes feladatok keresése: olyan feladat, akinek a jelenleg szabad erőforrásokkal kielégíthető az igénye. Ha van ilyen, akkor az általa legfoglalt erőforrásokat felszabadíthatjuk. Ezt addig kell folytatni, amíg találunk ilyen feladatot.
    c. Kiértékelés: ha az előző lépésben nem jelöltük valamelyik feladatot továbblépésre esélyesnek, akkor az holtpontra juthat, tehát a rendszer nem biztonságos.
4. Ha nem biztonságos a rendszer, akkor az állapot visszaállítása.

Az algoritmus pontos leírása megtalálható a magyar nyelvű tankönyv 2.2.7.7 fejezetében.

2.1.1.2 Mintapélda

Feladat

Egy rendszerben 4 erőforrásosztály van (A, B, C és D), az egyes osztályokba rendre 10, 11, 7 és 10 erőforrás tartozik. A rendszerben 5 folyamat verseng az erőforrásokért, a következő aktuális foglalással és maximális igénnyel:

	maximális igény				**aktuális foglalás**			
	A	B	C	D	A	B	C	D
P1	2	2	5	4	0	2	3	3
P2	7	7	3	4	3	1	2	2
P3	5	6	6	4	2	2	0	2
P4	4	1	2	3	2	1	2	2
P5	6	3	1	1	1	3	0	0

A rendszer a bankár algoritmust alkalmazza a holtpont elkerülésére. Biztonságos állapotban van-e jelenleg a rendszer? Ha igen, mutassa meg, a folyamatok hogyan tudják befejezni a működésüket, ha nem, hogyan alakulhat ki holtpont.

Megoldás

Első lépésként állítsuk be azoknak a változóknak az értékét, amit tudunk.

$$N = 5$$

$$M = 4$$

$$MAX = \begin{bmatrix} 2 & 2 & 5 & 4 \\ 7 & 7 & 3 & 4 \\ 5 & 6 & 6 & 4 \\ 4 & 1 & 2 & 3 \\ 6 & 3 & 1 & 1 \end{bmatrix}$$

$$FOGLAL = \begin{bmatrix} 0 & 2 & 3 & 3 \\ 3 & 1 & 2 & 2 \\ 2 & 2 & 0 & 2 \\ 2 & 1 & 2 & 2 \\ 1 & 3 & 0 & 0 \end{bmatrix}$$

$$MAXr = [10 \quad 11 \quad 7 \quad 10]$$

Számoljuk ki szabad erőforrásokat: MAXr-ből levonjuk a FOGLAL mátrix oszlopainak összegzésével kapott vektort.

$$SZABAD = [10 \quad 11 \quad 7 \quad 10] - [8 \quad 9 \quad 7 \quad 9] = [2 \quad 2 \quad 0 \quad 1]$$

Számoljuk ki a még kért erőforrásokat:

$$MÉG = MAX - FOGLAL = \begin{bmatrix} 2 & 2 & 5 & 4 \\ 7 & 7 & 3 & 4 \\ 5 & 6 & 6 & 4 \\ 4 & 1 & 2 & 3 \\ 6 & 3 & 1 & 1 \end{bmatrix} - \begin{bmatrix} 0 & 2 & 3 & 3 \\ 3 & 1 & 2 & 2 \\ 2 & 2 & 0 & 2 \\ 2 & 1 & 2 & 2 \\ 1 & 3 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 2 & 1 \\ 4 & 6 & 2 & 2 \\ 3 & 4 & 6 & 2 \\ 2 & 0 & 0 & 1 \\ 5 & 0 & 1 & 1 \end{bmatrix}$$

Az algoritmusnak most csak a 3. pontját futtatjuk, tehát ellenőrizni kell, hogy biztonságos-e a jelenlegi állapot.

a) *Kezdőértékek beállítása*: egy N elemű bool tömbben (LEFUT) gyűjtjük, hogy melyik feladat tudhat lefutni, kezdetben ennek az összes értéke hamis.

b) *Továbblépésre esélyes feladat keresése:* meg kell nézni, hogy van-e olyan feladat, ami legfeljebb annyi erőforrást kér, ami jelenleg szabad ($MÉG[i] \leq SZABAD$).

Ez egy ciklus, a következő iterációkkal:

- Kezdetben P4 az egyetlen ilyen. Ha P4 lefut, akkor visszaadja az általa foglalt erőforrásokat.

$$SZABAD = [2 \quad 2 \quad 0 \quad 1] + [2 \quad 1 \quad 2 \quad 2] = [4 \quad 3 \quad 2 \quad 3]$$

$$LEFUT[4] = igaz$$

- A következő iterációban P1 továbbléphet:

$$SZABAD = [4 \quad 3 \quad 2 \quad 3] + [0 \quad 2 \quad 3 \quad 3] = [4 \quad 5 \quad 5 \quad 6]$$

$$LEFUT[1] = igaz$$

- A következő iterációban nincs már olyan feladat, ami lefuthat (se P2, se P3, se P5 esetén nem teljesül, hogy $MÉG[i] \leq SZABAD$).

c) *Kiértékelés*: mivel LEFUT nem minden eleme igaz, ezért az állapot nem biztonságos. Tehát lehetséges, hogy kialakulhat a kezdeti állapotból holtpont (de az algoritmus nem bizonyítja, hogy ki is fog alakulni a holtpont!).

2.1.1.3 További feladatok

Ezeket a feladatokat érdemes megoldani önállóan, és csak utána ellenőrizni a végeredményt.

Feladat H1

Egy rendszerben három erőforrásosztály van (A, B, C), az egyes osztályokba rendre 9, 19 és 29 erőforrás tartozik. A rendszerben 6 folyamat verseng az erőforrásokért, a következő foglalással és új igénnyel:

	új kérés			**aktuális foglalás**		
	A	B	C	A	B	C
P1	1	10	4	2	8	2
P2	3	5	7	2	1	3
P3	1	1	1	1	1	1
P4	5	6	3	0	1	0
P5	6	6	7	2	2	2
P6	5	8	2	0	1	5

A rendszer a bankár algoritmust alkalmazza a holtpont elkerülésére. Biztonságos állapotban van-e jelenleg a rendszer? Ha igen, mutassa meg, a folyamatok hogyan tudják befejezni a működésüket, ha nem, hogyan alakulhat ki holtpont.[4]

[4] Végeredmény: az állapot nem biztonságos. P3, P2, P4, P6 lefutása után [5 9 25] szabad erőforrás marad, ami sem P1-nek, sem P5-nek nem elég. Figyelem: a feladatban MÉG van megadva és nem MAX. Ebből MAX = MÉG + FOGLAL alapján számolható MAX értéke.

3 Memóriakezelés

A központi tár szervezése, kezelése az operációs rendszerek tervezését, implementálását és teljesítményét befolyásoló egyik legfontosabb tényező.

A programhoz általában lineáris, folytonos címtartomány tartozik, amely 0-tól kezdődően a program maximális címéig terjed. Ezt logikai címtartománynak nevezzük, ugyanis a mai rendszerekben a programok végrehajtása gyakorlatilag soha nem a 0. fizikai címtől kezdődik, hanem a logikai címeket az operációs rendszer képezi le fizikai címekre. Ez kényelmes a programozó számára, és számos teljesítménynövelő eljárás alkalmazását teszi lehetővé (pl. virtuális memória használata).

3.1 Memóriafoglalási stratégiák (allokáció)[5]

3.1.1 A program címeinek kötése

A logikai és a fizikai címek közötti megfeleltetés a program számára transzparens módon történik. A két tartomány közötti megfeleltetés a program életciklusának különböző fázisaiban is történhet:

- *fordítás közben (compile time)*: A fordító program rendel fizikai címet a program logikai címtartományához. Ez a technika igen merev.
- *szerkesztés közben (link time)*: A program több egymástól független lefordított modulból áll, amelyek más-más logikai címekre hivatkoznak. A kapcsolatszerkesztő program feladata, hogy az összes modult egymás után elhelyezze a fizikai tárban, majd feloldja a modulok közötti hivatkozásokat.
- *betöltés közben (load time)*: A fordítás során egy úgynevezett áthelyezhető kódot állítunk elő. A betöltő program feladata, hogy a program címeit módosítsa az aktuális címkiosztás szerint.
- *futás közben*: A program mindvégig logikai címekkel dolgozik. A konkrét fizikai címeket az operációs rendszer határozza meg az egyes utasítások végrehajtása során.

3.1.2 Stratégiák a memóriaterületek lefoglalására

A logikai címek leképzése során az operációs rendszer feladata, hogy a központi memóriában elegendő helyet jelöljön ki a program számára. A szükséges tárterület lefoglalható folytonos módon, vagy valamilyen stratégia szerint kisebb részekben. Jelen segédletben mi csak az előbbivel foglalkozunk, vagyis azzal az esettel, amikor a program számára lefoglalt memóriaterület összefüggő.

Ha egy program véget ér, akkor a számára lefoglalt területet felszabadítjuk a memóriában. Abban az esetben, ha ennek helyén egy új program számára helyet foglalunk, akkor az minden bizonnyal nem fogja teljes egészében kitölteni a felszabadult helyet. Ha ekkor a fennmaradó hely túlságosan kicsi, az operációs rendszer ott már nem tud helyet foglalni egy újabb program számára, az a terület tulajdonképpen elveszik az operációs rendszer számára. Ezt a jelentséget külső tördelődésnek nevezzük.

A külső tördelődés csökkentésére különböző stratégiákat dolgoztak ki a programokhoz rendelendő tárterület kiválasztására. Az operációs rendszer a betöltendő program számára a szabad területek közül az alábbiak szerint választhat:

- *Első megfelelő (First fit)*: A rendelkezésre álló szabad területek közül a legelső elegendő méretűt foglaljuk le.

[5] Az összefoglaló elkészítéséhez az alábbi könyvet használtuk fel: Kiss István, Kondorosi Károly: Operációs rendszerek, 2004, Műegyetemi Kiadó

- *Következő megfelelő (Next Fit)*: Az Első megfelelő stratégiával szemben itt a keresést nem a tár elejéről kezdjük, hanem az után a terület után, amit legutoljára foglaltunk.
- *Legjobban megfelelő (Best fit)*: A legkisebbet foglaljuk le azon szabad területek közül, amelyek legalább akkorák, mint a lefoglalandó terület.
- *Legrosszabban illeszkedő (Worst fit)*: Az elérhető legnagyobb szabad területet allokáljuk. A spekuláció az, hogy a maradék terület még talán elegendő lesz egy újabb foglalás számára.

3.1.3 Allokációs mintapélda

Az alábbiakban a különböző allokációs stratégiák működését hasonlítjuk össze.

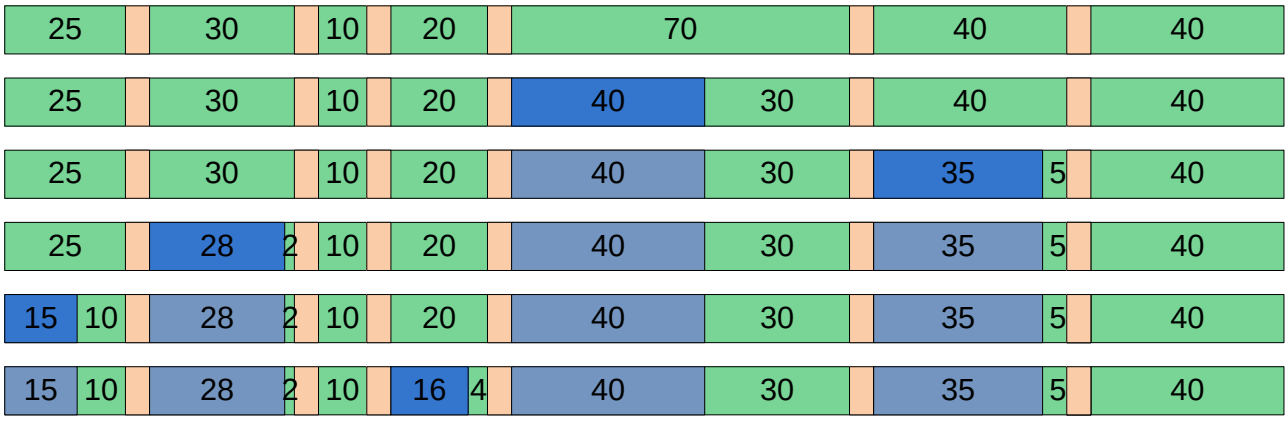
Feladat:

Mutassa be a változó méretű memória partíciók lefoglalásánál használt a) first fit, b) next fit, c) best fit, d) worst fit algoritmusokat. Egy rendszerben az adott pillanatban 25k, 30k, 10k, 20k, 70k, 40k és 40k méretű szabad területek vannak. Hogyan fog a fenti négy algoritmus sorrendben 40k, 35k, 28k, 15k, 16k méretű memória igényeknek helyet foglalni?

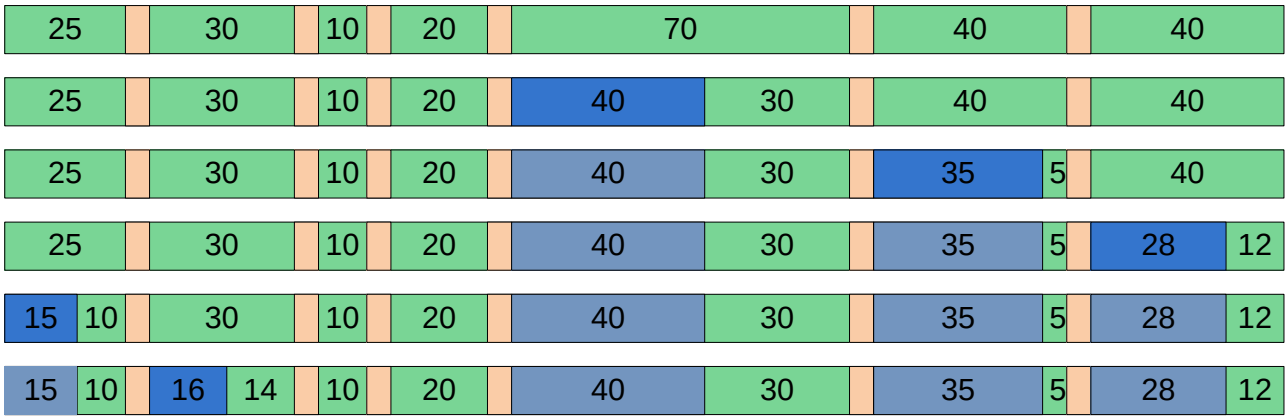
Megoldás:

Az alábbiakban a feladat megoldása látható a 4 algoritmusra. Zölddel a szabad területek vannak jelölve, kékkel pedig a lefoglalt területek. Minden egyes sorban 1-1 foglalást teljesítünk, amit élénkebb kék színnel emeltünk ki.

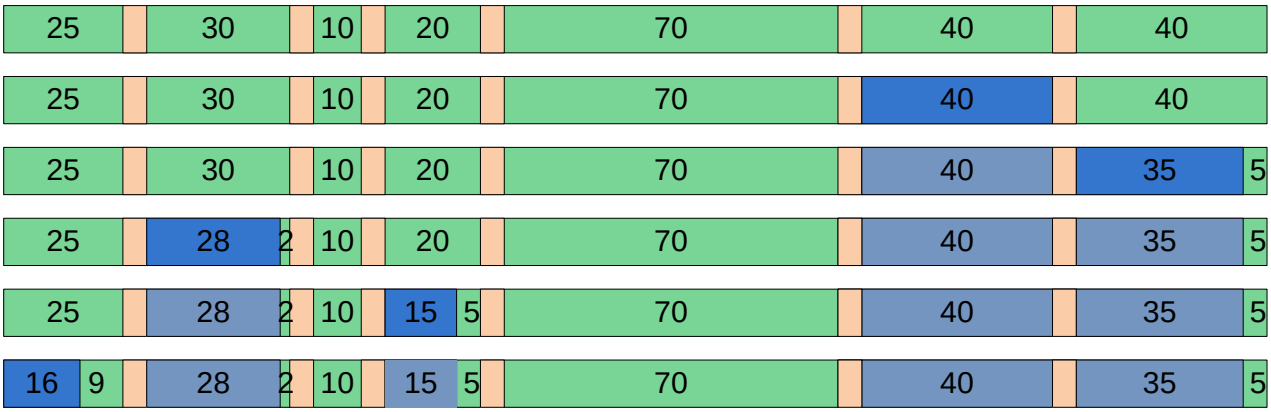
a) first fit



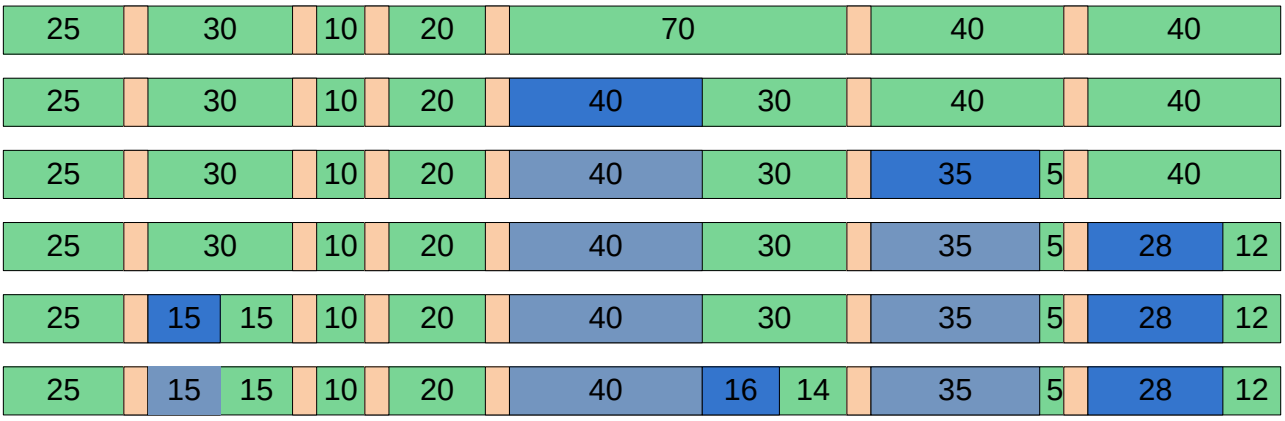
b) next fit



c) best fit



d) worst fit



3.1.4 További feladatok

3.1.4.1 Feladat H1

Hogyan változna a mintapélda végeredménye, ha az eddigi kérések mellé még érkezne egy 30k, 40k, vagy 50k méretű kérés?

3.1.4.2 Feladat H2

Mutassa be a változó méretű memória partíciók lefoglalásánál használt a) first fit, b) next fit, c) best fit, d) worst fit algoritmusokat. Egy rendszerben az adott pillanatban 50k, 30k, 200k, 16k és 30k méretű szabad területek vannak. Hogyan fog a fenti négy algoritmus sorrendben 20k, 30k, 10k, 100k, 60k méretű memória igényeknek helyet foglalni?

3.1.5 Példa vizsgafeladat

Mutassa be a megadott adatokkal, példákon keresztül a változó méretű memória partíciók lefoglalásánál használt first fit, next fit, best fit, és worst fit algoritmusokat.

Egy rendszerben az adott pillanatban 24 kbyte, 64 kbyte, 12 kbyte, 80 kbyte, 16 kbyte, 48 kbyte és 40 kbyte méretű szabad, összefüggő memória területek vannak. A rendszerben a *memória 4 kbyte-os blokkokban kerül nyilvántartásra*, ennél kisebb méretű töredék igény esetén a teljes blokk lefoglalásra kerül. Hogyan fog a fenti négy algoritmus sorrendben 65 kbyte, 22 kbyte, 48, kbyte, 12 kbyte, 62 kbyte méretű memória igénynek helyet foglalni?

Indokolja a tapasztalatokat és tegyen javaslatot az eredmények javítására!

Megoldás

First fit:

Foglalás	Memóriaterület						
	24k	**64k**	**12k**	**80k**	**16k**	**48k**	**40k**
65k				12k (80-68)			
22k	0k (24-24)						
48k		16k (64-48)					
12k		4k (16-12)					
62k	Nem	Nem	Nem	Nem	Nem	Nem	Nem

Next fit:

Foglalás	Memóriaterület						
	24k	**64k**	**12k**	**80k**	**16k**	**48k**	**40k**
65k				12k (80-68)			
22k						24k (48-24)	
48k		16k (64-48)					
12k		4k (16-12)					
62k	Nem	Nem	Nem	Nem	Nem	Nem	Nem

Best fit:

Foglalás	Memóriaterület						
	24k	**64k**	**12k**	**80k**	**16k**	**48k**	**40k**
65k				12k (80-68)			
22k	0k (24-24)						
48k						0k (48-48)	
12k			0k (12-12)				
62k		0k (64-64)					

Worst fit:

Foglalás	Memóriaterület						
	24k	**64k**	**12k**	**80k**	**16k**	**48k**	**40k**
65k				12k (80-68)			
22k		40k (64-24)					
48k						0k(48-48)	
12k		28k (40-12)					Vagy ez...
62k	Nem	Nem	Nem	Nem	Nem	Nem	Nem

Indoklás: Ebben az esetben a best fit képes egyedül allokálni a memóriát. Ugyanakkor ez az algoritmus a legnagyobb számítási komplexitású (kb. a worst fittel azonos a komplexitása). A worst fit a tapasztalatok szerinte a legrosszabb (átlagosan a teljes allokálható memória fele elveszik), míg a másik

három átlagosan hasonló teljesítményű (a teljes allokálható memória 1/3-a veszik el). Ez itt nem így van, a best fit egy kicsit jobb.

Módszerek az eredmények javítására:

1. Szemétgyűjtés (garbage collection) alkalmazása, vagyis a memória allokáció futás idejű átrendezése nem kielégíthető igények esetén. Hátránya, hogy erőforrás igényes.
2. Lapszervezés használata. Hátránya, hogy kell hozza MMU támogatás.

3.2 Lapcsere algoritmusok

A mai modern operációs rendszerek működésének alapvető tulajdonsága, hogy az éppen aktuálisan felhasznált adatokat a memóriában tárolja. Nagyon fontos tehát, hogy az operációs rendszer miként menedzseli a memória tartalmát. Erre a problémára egy megoldás a virtuális tárkezelés módszere.

Fontos megállapítani, hogy egy folyamat működéséhez nem szükséges a teljes folyamatot a fizikai memóriában tárolni, hiszen vannak olyan részei, amelyek soha nem hajtódnak végre (pl.: hibakezelés). Az is tény, hogy egyes kódrészletek akár több folyamathoz is tartozhatnak, ezeket felesleges a fizikai tárban duplikálni. A virtuális tárkezelés módszere mindezeket figyelembe veszi és egy átfogó megoldást jelent.

A módszerrel szemben a következő elvárások támaszthatóak:

- Lehessen a rendelkezésre álló fizikai memóriánál nagyobb memóriaigényű folyamatokat futtatni.
- A folyamatok csak a ténylegesen szükséges adatokat tartsák a fizikai memóriában.
- Az operációs rendszer mindig csak a szükséges kódrészleteket töltse be.
- A folyamatok osztozhassanak a közös kódrészeken valamint a közös adatszerkezeteken.

3.2.1 Lapszervezés

A folyamat által látott folytonos címteret meg kell különböztetni a valós címtértől. A kettő közti konverziót a lapszervezésű memóriakezelés esetén a *laptábla* segítségével végzi el az operációs rendszer. Ez a tábla lehetővé teszi, hogy egy lap csak akkor kerüljön be a fizikai memóriába, amikor ténylegesen szükséges (tehát amikor egy folyamat hivatkozik rá). Ennek megvalósítására a laptábla minden bejegyzéséhez tartozik egy *valid/invalid bit*, amely azt tárolja, hogy az adott bejegyzéshez tartozó memóriatartalom be van-e töltve a fizikai memóriába. Ha egy olyan lapra hivatkozunk, amely *invalid*, tehát még nincs a memóriában, akkor egy laphiba (*page fault*) generálódik az MMU (*memory management unit*) által. Ez nem hibás működést jelent, hanem csak annyit, hogy azt a lapot be kell tölteni. Ez úgy valósítható meg, hogy a *fault* hatására a folyamat működése megszakad, az operációs rendszer behozza a fizikai memóriába a hivatkozott lapot, majd visszaadja a vezérlést a folyamatnak. Ez a módszer lehetővé teszi, hogy az egész lapozás a folyamatok számára transzparensen történjen meg, hiszen a lapozás során a folyamat passzívan várakozik, és az egész műveletről nem is értesül.

3.2.2 Lapozási stratégiák

Alapvetően két stratégia különböztethető meg. Ha a lapozás csak laphibák esetén történik meg, akkor azt *igény szerinti lapozásnak* nevezik. Ilyenkor csak a hivatkozott lap töltődik be a memóriába. Ennek előnye, hogy csak a ténylegesen szükséges lapok töltődnek be, azonban könnyű azt is belátni, hogy ez lassú. Új lap behozása lényegesen lassabb, mint egy már betöltött lap elérése, hiszen a háttértár jóval lassabb sebességű, mint a memória.

A másik stratégia az *előretekintő lapozás*, amely során az operációs rendszer valamilyen módon megjósolja, hogy milyen lapokra lesz szükség, és ezeket betölti a fizikai memóriába. Amennyiben a jóslás sikeres, akkor ez gyorsulást jelent, de amennyiben rosszul jósolt az operációs rendszer, akkor

feleslegesen pazarolta az erőforrásokat. A módszer alkalmazása azt is igényli, hogy legyenek szabad erőforrások (pl.: processzoridő) a rendszerben.

3.2.3 Lapcsere stratégiák

A virtuális tárkezelési módszer során is elkerülhetetlenül elérkezik az a pillanat, hogy a fizikai memória megtelik, azaz nincs üres hely benne, ahova behozható lenne az új memórialap. Ilyenkor lépnek életbe a lapcserélési stratégiák, amelyek meghatározzák, hogy melyik lap helyére töltődjön majd be az új memóriatartalom.

A lapcserélés megvalósítható globálisan (ilyenkor az összes lap közül választódik egy áldozat) és lokálisan (ilyenkor csak a folyamat lapjai között keres az algoritmus) is.

3.2.3.1 Optimális algoritmus

Ahogy a neve mutatja, ez az algoritmus optimális, tehát a „legjobb". Az egyetlen probléma vele, hogy nem realizálható. Arra azonban kitűnő, hogy a többi algoritmus hatékonyságát ehhez hasonlítsuk.

A működési elve nagyon egyszerű, amikor egy új lapot kell behozni, akkor azt egy olyan helyére teszi, amelyikre a legkésőbb lesz újra szükség, tehát amelyikre a legkésőbb hivatkoznak újra. Nyilvánvaló, hogy ez azt igényli, hogy a rendszer korlátlan hosszan előre lásson az időben, ami nyilván lehetetlen.

3.2.3.2 Legrégebbi lap algoritmus (*First-In-First-Out*)

Az algoritmus rendkívül egyszerűen megvalósítható (ez a legnagyobb előnye). A behozott lapokat a behozás sorrendje szerint sorba rendezi az algoritmus, majd ha szükséges, azt cseréli le, amelyet a legrégebben töltődött be (tehát amelyik az utolsó a sorban). Ezzel az a probléma, hogy olyan lapokat is lecserél, amelyeket gyakran használ a rendszer.

Bélády anomália

Az algoritmuson megfigyelhető a Bélády anomália is, amely azt jelenti, hogy egyes esetekben annak ellenére, hogy a rendszer nagyobb memóriaterületet kap, mégis több laphibát generál. Ez látható később a számítási példáknál is.

3.2.3.3 Újabb esély algoritmus (*Second Chance*)

Az előző algoritmus továbbfejlesztése. A működése annyiban különbözik, hogy minden lap mellé egy biten (*referenced v. used bit*) tárolódik, hogy volt-e a közelmúltban használva. Amikor a FIFO szerint egy olyan lapot kellene törölni, amelyen ez a bit be van állítva, olyankor ez a lap bennmaradhat a memóriában, de a bit értékét hamisra kell állítani, valamint a sor elejére kerül. Az algoritmus ezután megvizsgálja a következő lapot, egészen addig, amíg nem talál egy olyan lapot, amely törölhető a memóriából. Ez nem lesz egy végtelen ciklus, hiszen ha mindig törli a bit értékét, és legrosszabb esetben (azaz ha az összes lap esetén be volt állítva a bit) az eredeti utolsó lapot fogja törölni.

Amennyibe egy olyan lapra hivatkozik a rendszer, amely már a memóriában volt, olyankor a bit értékét igazra kell állítani. A memóriába behozáskor is igaz lesz ez az érték.

Ez az algoritmus még mindig nagyon egyszerűen megvalósítható. A lapcserélést a behozás sorrendje határozza meg, de már figyelembe veszi a használatot is.

3.2.3.4 Legrégebben nem használt algoritmus (*Least Recently Used*)

Az algoritmus egy aránylag bonyolult megoldás, de jól megközelíti az optimális megoldást. Az algoritmus csak hátrafele néz, azonban ez a lokalitás elve miatt a valóságban hatékony lehet. A lapcserélés áldozatát az alapján választja ki, hogy melyik volt legutoljára használva. Több különféle konkrét megvalósítása is létezik:

- Számlálós megvalósítás: minden laphoz egy időbélyeget ment el a rendszer, amely azt tárolja, hogy mikor volt utoljára használva.

- Láncolt listás megvalósítás: A legutoljára használt lap mindig egy láncolt lista végére kerül és így az áldozat mindig kiválasztható a lista elejéről.

Sok esetben csak egy közelítését alkalmazzák.

3.2.3.5 Legkevésbé használt algoritmus (*Least Frequenty Used*)

Az algoritmus alapja, hogy a közelmúltban gyakran használt lapokat a lokalitás miatt nagy valószínűséggel újra fogjuk használni, így ezeket nem érdemes lecserélni.

Minden lap esetében számláljuk, hogy hány hivatkozás történt rá. A lapcserélés áldozata az a lap lesz, amelyikre a legkevesebb hivatkozás történt. Az algoritmus nem felejt, tehát a számláló sose nullázódik le. Ez problémát jelenthet, hiszen ha egy lap csak a folyamat betöltődéséhez szükséges, de ott nagyon intenzíven van használva, akkor később is benn fog maradni a memóriában.

Egy másik probléma az algoritmussal, hogy a frissen behozott lapokat fogja lecserélni. Erre az a megoldás, hogy ideig ezeket a lapokat be kell fagyasztani a memóriába.

3.2.3.6 Utóbbi időben nem használt algoritmus (*Not Recently Used*)

Az algoritmus használja a módosított (*M bit*) valamint a hivatkozott (*R bit*) biteket is. A két bit értéke alapján egy prioritási sorrend határozható meg:

- 0. szint: R = 0 és M = 0 (*legalacsonyabb szint*)
- 1. szint: R = 0 és M = 1
- 2. szint: R = 1 és M = 0
- 3. szint: R = 1 és M = 1 (*legmagasabb szint*)

Az algoritmus mindig a legalacsonyabb prioritású szintről választ egy áldozatot a cseréléshez (ha több lehetőség is van, akkor implementációfüggő módon teszi ezt). A hivatkozott bit értékét az operációs rendszer adott időközönként törölheti, azonban a módosított bit értékét mindig meg kell őrizni.

3.2.4 Példafeladatok

3.2.4.1 Feladat: Optimális algoritmus

Adott az alábbi laphivatkozási sorrend: 8, 3, 2, 9, 3, 3, 9, 6, 8, 2, 7, 1. Vizsgálja meg és írja le az Optimális lapcserélési stratégia működését 3 méretű memóriánál és írja le, miért van lehetőség arra, hogy az optimális algoritmust egyáltalán megközelíthetjük a tényleges megvalósításoknál!

Az algoritmus működésének magyarázata a 3.2.3.1-es pontban olvasható. A memória tartalma az alábbi módon alakul:

Lap-hivatkozások	**8**	**3**	**2**	**9**	**3**	**3**	**9**	**6**	**8**	**2**	**7**	**1**
	8	8	8	8	8	8	8	8	8	2	7	1
	-	3	3	3	3	3	3	3	3	3	3	3
	-	-	2	9	9	9	9	6	6	6	6	6

Színmagyarázat:

- Laphiba – ezek a hibák úgy keletkeztek, hogy volt még hely a memóriában és nem kellett cserélni
- Lapcsere – újonnan behozott lapok, telített memóriánál, cseréléssel

Összesen 8 laphiba történt.

Az optimális algoritmust nem lehet megvalósítani, hiszen korlátlan távolságra előretekint, tehát jósol. A lokalitás elve teszi lehetővé, hogy mégis lehet olyan algoritmust készíteni, amely megközelíti az optimális algoritmus hatékonyságát. A lokalitás elve azt mondja ki, hogy a folyamatok nagy valószínűséggel csak az éppen aktuális utasítás környékére fognak hivatkozni.

3.2.4.2 Feladat: FIFO algoritmus

Adott az alábbi laphivatkozási sorrend: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5. Vizsgálja meg és írja le a FIFO lapcserélési stratégia működését 3 és 4 memóriakeret esetén és magyarázza meg a tapasztaltakat!

Az algoritmus magyarázata a 3.2.3.2-es pontban olvasható.

1. eset: 3 lap méretű memória esetén az alábbi módon alakul a memória tartalma:

Lap-hivatkozások	**1**	**2**	**3**	**4**	**1**	**2**	**5**	**1**	**2**	**3**	**4**	**5**
	1	2	3	4	1	2	5	5	5	3	4	4
	-	1	2	3	4	1	2	2	2	5	3	3
	-	-	1	2	3	4	1	1	1	2	5	5

Színmagyarázat a 3.2.4.1-es feladatnál található.

Összesen 9 laphiba történt.

2. eset: 4 lap méretű memória esetén:

Lap-hivatkozások	**1**	**2**	**3**	**4**	**1**	**2**	**5**	**1**	**2**	**3**	**4**	**5**
	1	2	3	4	4	4	5	1	2	3	4	5
	-	1	2	3	3	3	4	5	1	2	3	4
	-	-	1	2	2	2	3	4	5	1	2	3
	-	-	-	1	1	1	2	3	4	5	1	2

A színmagyarázat megegyezik a korábbival. Megfigyelhető, hogy itt 10 laphiba történt. Ez több mint a kisebb memória esetén tapasztalt laphibák száma, amely némileg meglepő lehet. Ezt a jelenséget *Bélády anomáliának* nevezik.

A táblázatban a lapok pozíciója nem a memóriában ténylegesen elfoglalt fizikai helyet jelöli, hanem a logikai sorrendet hivatott megjeleníteni.

3.2.4.3 Feladat: Újabb esély algoritmus

Adott az alábbi laphivatkozási sorrend: 1, 2, 3, 3, 4, 5, 3, 6, 5, 2. Vizsgálja meg az Újabb esély (Second Chance) lapcserélési stratégia működését 4 méretű memóriánál! Jelölje minden lap esetén az algoritmushoz szükséges bitek értékét is!

Az algoritmus működése a 3.2.3.3-as pontban olvasható.

A memória tartalmának alakulása:

Lap-hivatkozások	1	2	3	3	4					5	3	6	5		2
	1 (R)	2 (R)	3 (R)	3 (R)	4 (R)	1	2	3	4	5 (R)	5 (R)	6 (R)	6 (R)	3	2 (R)
	-	1 (R)	2 (R)	2 (R)	3 (R)	4 (R)	1	2	3	4	4	5 (R)	5 (R)	6 (R)	3
	-	-	1 (R)	1 (R)	2 (R)	3 (R)	4 (R)	1	2	3	3 (R)	4	4	5 (R)	6 (R)
	-	-	-	-	1 (R)	2 (R)	3 (R)	4 (R)	1	2	2	3 (R)	3 (R)	4	5 (R)

Színmagyarázat a 3.2.4.1-es feladatnál található. A *referenced* (hivatkozott) bit értékét az R betűk jelölik. Ahol egy oszlop fejléce üres, ott az előző hivatkozás feldolgozása történik még. Nem kötelező ezeket kifejteni, azonban így jobban megérthető az algoritmus működése.

7 laphiba történt.

A táblázatban a lapok pozíciója nem a memóriában ténylegesen elfoglalt fizikai helyet jelöli, hanem a logikai sorrendet hivatott megjeleníteni.

3.2.4.4 Feladat: Legrégebben nem használt algoritmus

Adott az alábbi laphivatkozási sorrend: 9, 5, 8, 3, 2, 6, 3, 7, 7, 9, 7, 3, 7, 2, 4. Vizsgálja meg és írja le a Legrégebben nem használt (LRU) lapcserélési stratégia működését 4 méretű memóriánál! Jelölje minden lap esetén az algoritmushoz szükséges mérőszám pillanatnyi értékét is!

Az algoritmus működése a 3.2.3.4-as pontban olvasható.

A memória tartalmának alakulása:

Lap-hivatkozások	9	5	8	3	2	6	3	7	7	9	7	3	7	2	4
	9 (0)	9 (1)	9 (2)	9 (3)	2 (0)	2 (1)	2 (2)	2 (3)	2 (4)	9 (0)	9 (1)	9 (2)	9 (3)	9 (4)	4 (0)
	-	5 (0)	5 (1)	5 (2)	5 (3)	6 (0)	6 (1)	6 (2)	6 (3)	6 (4)	6 (5)	6 (6)	6 (7)	2 (0)	2 (1)
	-	-	8 (0)	8 (1)	8 (2)	8 (3)	8 (4)	7 (0)	7 (0)	7 (1)	7 (0)	7 (1)	7 (0)	7 (1)	7 (2)
	-	-	-	3 (0)	3 (1)	3 (2)	3 (0)	3 (1)	3 (2)	3 (3)	3 (4)	3 (0)	3 (1)	3 (2)	3 (3)

Színmagyarázat a 3.2.4.1-es feladatnál található. Minden lap esetén zárójelben jelölve van a legutolsó hivatkozás óta eltelt idő.

10 laphiba történt.

3.2.4.5 Feladat: Legkevésbé használt algoritmus

Adott az alábbi laphivatkozási sorrend: 7, 1, 6, 3, 0, 0, 3, 3, 1, 9, 5, 5, 5. Vizsgálja meg és írja le a Legkevésbé használt (LFU) lapcserélési stratégia működését 4 méretű memóriánál! Jelölje minden lap esetén az algoritmushoz szükséges mérőszám és a szükséges bit(ek) pillanatnyi értékét is! Az algoritmus 3 időegységig fagyassza be a lapokat!

Az algoritmus működése a 3.2.3.5-ös pontban olvasható.

A memória tartalmának alakulása:

Lap-hivatkozások	7	1	6	3	0	0	3	3	1	9	5	5	5
	7 (1) (F)	7 (1) (F)	7 (1) (F)	7 (1)	0 (1) (F)	0 (2) (F)	0 (2) (F)	0 (2)	0 (2)	0 (2)	5 (1) (F)	5 (2) (F)	5 (3) (F)
	-	1 (1) (F)	1 (1) (F)	1 (1) (F)	1 (1)	1 (1)	1 (1)	1 (1)	1 (2)	1 (2)	1 (2)	1 (2)	1 (2)
	-	-	6 (1) (F)	6 (1) (F)	6 (1) (F)	6 (1)	6 (1)	6 (1)	6 (1)	9 (1) (F)	9 (1) (F)	9 (1) (F)	9 (1)
	-	-	-	3 (1) (F)	3 (1) (F)	3 (1) (F)	3 (2)	3 (3)	3 (3)	3 (3)	3 (3)	3 (3)	3 (3)

Színmagyarázat a 3.2.4.1-es feladatnál található. Minden lap esetében zárójelben látható a hivatkozások száma, valamint az (F) jel a befagyasztást (lock bit) jelenti.

7 laphiba történt.

3.2.4.6 Feladat: Utóbbi időben nem használt algoritmus

Adott az alábbi laphivatkozási sorrend: 7, 9, 2, 1, 5, 5, 8, 5, 9, 6, 2, 8, 0. Az aláhúzás azt jelzi, hogy ott módosítás is történt. Vizsgálja meg és írja le az Utóbbi időben nem használt (NRU) lapcserélési stratégia működését 3 méretű memóriánál! Jelölje minden lap esetén az algoritmus működéséhez szükséges bitek pillanatnyi értékét is! A hivatkozott bitek 3 időegységenként automatikusan törlődnek az operációs rendszer által.

Az algoritmus működése a 3.2.3.6-os pontban olvasható.

A memória tartalmának alakulása:

Lap-hivatkozások	7	9	2		1	5	5		8	5	9		6	2	8		0
	7 (R) (M)	7 (R) (M)	7 (R) (M)	7 (M)	7 (M)	7 (M)	7 (M)	7 (M)	7 (M)	7 (M)	9 (R)	9	6 (R) (M)	6 (R) (M)	6 (R) (M)	6 (M)	6 (M)
	-	9 (R)	9 (R)	9	1 (R)	1 (R)	1 (R)	1	8 (R) (M)	8 (R) (M)	8 (R) (M)	8 (M)	8 (M)	2 (R)	2 (R)	2	0 (R)
	-	-	2 (R)	2	2	5 (R)	5 (R) (M)	5 (M)	5 (M)	5 (R) (M)	5 (R) (M)	5 (M)	5 (M)	5 (M)	8 (R)	8	8

Színmagyarázat a 3.2.4.1-es feladatnál található. A módosított (M) és hivatkozott (R) bitek minden lap esetében jelölve vannak. Az üres fejlécű oszlopok a 3 időegységenkénti hivatkozott bit törlésének eredményét jelölik.

11 laphiba történt.

3.2.4.7 Feladat: Példa vizsgafeladat

Egy igény szerinti lapozást használó rendszerben 4 fizikai memórialap áll egy folyamat rendelkezésére. A futás folyamán sorban a következő virtuális lapokra történik hivatkozás:

1. sorozat: 0, 1, 2, 3, 0, 1, 2, 3, 4, 1, 2, 3, 4, 1

2. sorozat: 0, 1, 2, 3, 4, 1, 0, 1, 2, 3, 1, 4, 0, 1

Hány laphiba következik be a rendszerben a következő lapcsere algoritmusok esetén a sorozatokban, ha kezdetben a 4 lap üres minden sorozat indításakor?

- Legrégebbi lap (FIFO) algoritmus alkalmazásánál
- Újabb esély (SC) algoritmus alkalmazásánál

Röviden magyarázza meg az eredményeket!

Megoldás:

A feladatban négy különböző esetet kell megvizsgálni, mindkét sorozatra két különböző algoritmust kell lefuttatni.

I. FIFO 4 fizikai memória kerettel és 1. sorozat:

Lapok	**0**	**1**	**2**	**3**	**0**	**1**	**2**	**3**	**4**	**1**	**2**	**3**	**4**	**1**
FIFO0	0	1	2	3	3	3	3	3	4	4	4	4	4	4
FIFO1		0	1	2	2	2	2	2	3	3	3	3	3	3
FIFO2			0	1	1	1	1	1	2	2	2	2	2	2
FIFO3				0	0	0	0	0	1	1	1	1	1	1
Laphiba	**I**	**I**	**I**	**I**					**I**					

Laphibák száma: 5

II. FIFO 4 fizikai memória kerettel és 2. sorozat:

Lapok	**0**	**1**	**2**	**3**	**4**	**1**	**0**	**1**	**2**	**3**	**1**	**4**	**0**	**1**
FIFO0	0	1	2	3	4	4	0	1	2	3	3	4	0	1
FIFO1		0	1	2	3	3	4	0	1	2	2	3	4	0
FIFO2			0	1	2	2	3	4	0	1	1	2	3	4
FIFO3				0	1	1	2	3	4	0	0	1	2	3
Laphiba	**I**	**I**	**I**	**I**	**I**		**I**	**I**	**I**	**I**		**I**	**I**	**I**

Laphibák száma: 12

III. SC 4 fizikai memória kerettel és 1. sorozat:

Lapok	**0**	**1**	**2**	**3**	**0**	**1**	**2**	**3**	**4**	**1**	**2**	**3**	**4**	**1**
FIFO0	0,Y	1,Y	2,Y	3,Y	3,Y	3,Y	3,Y	3,Y	4,Y	4,Y	4,Y	4,Y	4,Y	4,Y
FIFO1		0,Y	1,Y	2,Y	2,Y	2,Y	2,Y	2,Y	3,N	3,N	3,N	3,Y	3,Y	3,Y
FIFO2			0,Y	1,Y	1,Y	1,Y	1,Y	1,Y	2,N	2,N	2,Y	2,Y	2,Y	2,Y
FIFO3				0,Y	0,Y	0,Y	0,Y	0,Y	1,N	1,Y	1,Y	1,Y	1,Y	1,Y
Laphiba	**I**	**I**	**I**	**I**					**I**					

Laphibák száma: 5

IV. SC 4 fizikai memória kerettel és 2. sorozat:

Lapok	**0**	**1**	**2**	**3**	**4**	**1**	**0**	**1**	**2**	**3**	**1**	**4**	**0**	**1**
FIFO0	0,Y	1,Y	2,Y	3,Y	4,Y	4,Y	0,Y	0,Y	2,Y	3,Y	3,Y	4,Y	0,Y	0,Y
FIFO1		0,Y	1,Y	2,Y	3,N	3,N	1,N	1,Y	0,Y	2,N	2,N	1,N	4,Y	4,Y
FIFO2			0,Y	1,Y	2,N	2,N	4,Y	4,Y	1,Y	0,N	0,N	3,Y	1,N	1,Y
FIFO3				0,Y	1,N	1,Y	3,N	3,N	4,Y	1,N	1,Y	2,N	3,Y	3,Y
Laphiba	**I**	**I**	**I**	**I**	**I**		**I**		**I**	**I**		**I**	**I**	

Laphibák száma: 10

A kapott eredmények értékelésénél többek között azt kell megvizsgálni, hogy miért ezt az eredményt kaptuk, valamint, hogy a különböző algoritmusok a vártnak megfelelően teljesítettek-e egymással szemben.

1. Az első sorozat első részének a munkahalmaza belefér a rendelkezésre álló fizikai memória keretek számába, aztán a munkahalmaz változik, és ezért van egy laphiba (0 lecserélődik 4-re), utána ismét belefér a munkahalmaz a rendelkezésre álló fizikai memória keretekbe.

2. A második sorozat munkahalmaza nem fér bele a rendelkezésre álló fizikai memória keretekbe, de az 1. lapra gyakran hivatkozik a folyamat.

3. A 2. sorozatban 2 alkalommal megtakaríthattunk egy lapcserét, mivel az SC algoritmus ezekben az esetekben sikeresen bent tartotta a gyakran használt 1-es lapot.

4. Az SC algoritmus statisztikailag jobban viselkedik (alacsonyabb a laphiba gyakoriság) mint a FIFO, viszont implementációja bonyolultabb, és HW támogatást is igényel (used/referenced) bit.

4 További információ

[1] Operációs rendszerek tantárgy előadásai, http://www.mit.bme.hu/oktatas/targyak/vimia219
[2] Kóczy Annamária, Kondorosi Károly (szerk): Operációs rendszerek mérnöki megközelítésben, Panem, 2000.
[3] Jun Suzuki: Special Topics on Concurrent and Distributed Systems, Lecture Note 7
University of Massachusetts Boston, Department of Computer Science
www.cs.umb.edu/~jxs/courses/2007/697/notes/note07.ppt
[4] Avi Silberschatz, Peter Baer Galvin, Greg Gagne: Operating System Concepts, Eight Edition, 2008,
Előadásfóliák, 9. fejezet
http://codex.cs.yale.edu/avi/os-book/OS8/os8c/slide-dir/index.html