Operációs rendszerek: Memóriakezelés

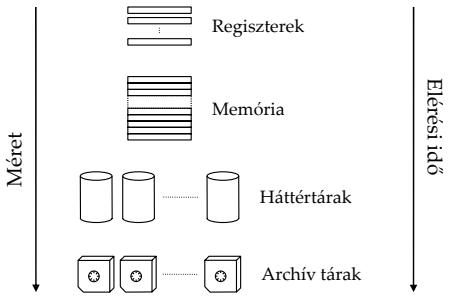
Balogh Ádám
Lőrentey Károly
Eötvös Loránd Tudományegyetem
Informatikai Kar
Algoritmusok és Alkalmazásaik Tanszék

Tartalomjegyzék

A tár fogalma

- Adatokat tárol
  - Programkódok
  - Programok adatai
  - *Neumann-elv: mindkettő tárolható ugyanabban a tárban, vegyesen*
- Adatok elérhetősége:
  - Elérés módja: írás, olvasás vagy végrehajtás
  - Elérés kulcsa: címzés, asszociatív lekérdezés
  - Egyszerre elérhető adatok mennyisége: egy byte-tól több megabyte-ig
- További jellemzők:
  - Méret
  - Elérési sebesség

A tárak fajtái



Adatcsere a tárak között

- Látható: minél gyakrabban használunk egy adatot, annál magasabb szinten célszerű tárolni
- Szintek között esetlegesen gyorsítótárak
- Adatcserét végezheti:
  - Felhasználó (pl. archív tárak – háttértárak)
  - Felhasználói program (pl. regiszterek – memória)
  - Operációs rendszer (pl. memória – háttértárak részben)
  - Hardver (pl. bizonyos gyorsítótárak)

A memóriakezelés feladatai

- Adatok elhelyezése a memóriában
  - Programok betöltése
  - Adatterületek biztosítása programok számára
- Memória megosztása különböző folyamatok között

Programok betöltése (1)

- Feladat: végrehajtható programot be kell olvasni a háttértárról a memóriába, és ott el kell indítani
- Végrehajtható program meghatározott címekkel dolgozik ⇨ programot mindig ugyanoda kell tölteni
- Memória megosztása: minden folyamat más címtartományhoz férhet hozzá ⇨ programjaik is különböző címeken vannak
- Ez ebben a formában ellentmondás!

Programok betöltése (2)

- Megoldások:
  - Egyszerre egy program a memóriában, többi a háttértárakon: nem hatékony
    - Javítás: futó program a memória alján, a többi feljebb: még így sem hatékony (sok mozgatás)
  - Átcímzés: programbetöltő (rendszerszolgáltatás) átírja a beolvasott program címeit, mielőtt elindítaná
    - Nehéz megtalálni a címeket (pl. relatív címzés esetén) ⇨ nem hatékony
  - Bázisregiszter használata: programok nem fix címekre hivatkoznak, hanem egy bázisregisztertől számított relatív címekre
    - Csak a bázisregisztert kell átírni a program elindítása előtt

Felhasználói adatterületek

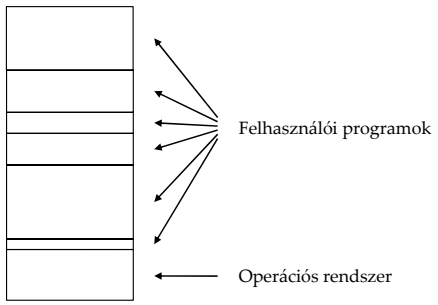
- Adatok keletkezése a memóriában:
  - Felhasználói program állítja elő
  - Háttértárról töltődik be
- Mindkét esetben kell egy szabad terület, ahol elfér
  - Operációs rendszer feladata ennek a területnek a biztosítása, és kezdőcímének közlése a felhasználói programmal
  - Szükséges rendszerszolgáltatás: memóriaigénylés

Memória felosztása

- Sok módszer ismert:
  - Partícionálás
    - Fix számú és méretű partíció
    - Dinamikusan változó számú és méretű partíciók
  - Virtuális memóriakezelés
    - Lapozás
    - Szegmentálás
    - Hibrid megoldások
- Egyes módszereken belül is sok különböző megvalósítás, sok különböző algoritmus létezik

Tartalomjegyzék

Partícionálás



Rögzített partíciók (1)

- Rendszergazda definiálja a partíciók számát, és az egyes partíciók méretét
- Legelső partícióban az operációs rendszer kódja fut
- Minden folyamat olyan partícióba kerül, amelybe befér a programja
- Ha egy program nem fér be egyik szabad partícióba sem, a háttértáron várakozik

Rögzített partíciók (2)

- 1. algoritmus:
  - Ha egy partíció szabaddá válik, megkeressük azt a legnagyobb folyamatot, amelyik befér
  - Probléma: nem igazságos, a partíció méretét alulról közelítő folyamatok előnyt élveznek
- 2. algoritmus:
  - Az 1. finomítása
  - Amikor kiválasztjuk a folyamatot, akkor minden más folyamatnál, amely szintén befért volna, növekszik egy számláló
  - Ha a számláló egy bizonyos érték fölött van, a folyamat előnyt élvez

Rögzített partíciók (3)

- Az algoritmus tovább javítható:
  - Minden program méretét felkerekítjük a legkisebb olyan partíció méretére, amelybe beférne, így csak akkor kerül hátrányba egy folyamat valamely partíciónál, ha létezik kisebb is, amelybe még befér

Dinamikus partíciók (1)

- Csakis az operációs rendszernek van előre rögzített méretű partíciója
- Minden folyamat létrejöttekor az operációs rendszer keres neki egy helyet a memóriában, ahol elfér, és létrehoz számára egy partíciót
- Ha egy folyamat véget ér, megszűnik a partíciója is
- Ha nincs olyan összefüggő szabad terület, ahol elférne, a háttértáron várakozik

Dinamikus partíciók (2)

- Operációs rendszer feladata:
  - Partíciók nyilvántartása
  - Szabad területek nyilvántartása (pl. láncolt lis-tában), szomszédos darabok összeolvasztása
  - Megfelelő méretű szabad terület kiválasztása egy új folyamat számára
- Problémák:
  - Pazarlás: kis folyamatok elfoglalják a nagy szabad területeket a nagyok elől
  - Elaprózódás: sok kis szabad terület marad, amelyekben nem férnek el folyamatok, de összesen sok területet foglalnak el

Dinamikus partíciók (3)

- 1. algoritmus: **First Fit**
  - Első megfelelő méretű partíciót választjuk a szabadlistából
  - Előnye: gyors és egyszerű
  - Hátránya: memória eleje elaprózódik, egyre növekvő keresési idő
- 2. algoritmus: **Next Fit**
  - Az előző módosítása: nem az elejéről indulunk, hanem a legutóbb létrejött folyamat számára talált partíciótól
  - Előnye: gyors és egyszerű, memóriát egyenletesen osztja el, egyenlő keresési idők

Dinamikus partíciók (4)

- 2. algoritmus (folytatás)
  - Hátránya: nem nagyon törődik a pazarlás és az elaprózódás kérdéseivel
- 3. algoritmus: **Best Fit**
  - Végignézzük az összes szabad területet, és a legkisebb olyat választjuk, amelyikbe befér
  - Előnye: nem pazarol
  - Hátránya: lassú, és a túl kicsi fennmaradó részek a memória elaprózódásához vezetnek
- 3. algoritmus javítása:
  - Adunk egy alsó korlátot a fennmaradó rész méretére

Dinamikus partíciók (5)

- 4. algoritmus: **Worst Fit**
  - A lehető legnagyobb szabad területet választjuk
  - Előny: véd a memória elaprózódása ellen
  - Hátrány: lassú és pazarló
- 4. algoritmus javításai:
  - Szabad területek kupac adatszerkezetben
  - Egy maximum a folyamat méretétől függően
- Minden algoritmusnál célszerűbb bájtok helyett blokkokban foglalni a memóriát, mert a néhány bájtos részek partíciónak alkalmatlanok, és csak lassítják a keresést

Tartalomjegyzék

Virtuális memória (1)

- Alapproblémák:
  - Előfordulhat, hogy egy program teljes egészében egyáltalán nem fér be
  - Mozgatás a háttértárra és vissza sok erőforrást igényel
  - Programok rövid idő alatt csak kis részét használják a tárterületüknek
  - Biztonsági probléma: nehéz elérni, hogy a programok ne nyúlhassanak ki a partícióból
  - Nem mindig tudhatjuk a program indulásakor, hogy mennyi memóriára lesz szükség, azt dinamikusan kell lefoglalni
- Ötlet: programok által látott memóriaterület különbözzön a fizikailag létezőtől!

Virtuális memória (2)

- Követelmények:
  - Minden program egy saját memóriaterületet lásson, mintha az egész memória az övé volna
  - Bármely címre lehessen hivatkozni a területen belül, és az adatok permanensen tárolódjanak ott (mint a fizikai memóriában)
  - Program ne vegyen észre semmit a megvalósítás módjáról
  - Virtuális memória elérésének hatékonysága ne legyen sokkal rosszabb, mint a fizikaié

Virtuális memória (3)

- Megvalósítás:
  - Mind a virtuális, mind a fizikai memóriát felosztjuk, és a részeket egymáshoz rendeljük
  - Egyes virtuális memóriabeli területekhez nem rendelünk területet a fizikai memóriából
    - Egyáltalán nem rendelünk hozzá területet: üres memóriaterület
    - Háttértáron tároljuk az adatait
  - Ha hivatkozás történik egy olyan címre, amelyhez nem tartozik fizikai memóriaterület: kivétel keletkezik:
    - Kivételkezelő feladata a hozzárendelések megváltoztatása úgy, hogy tartozzon hozzá

Virtuális memória (4)

- Megvalósítás (folyatás):
  - Nem egy virtuális címtér van, hanem minden folyamatnak saját
  - Más folyamatok adatait tartalmazó fizikai memóriabeli területekhez nem tartozik terület a folyamat virtuális memóriájában
  - Több folyamat virtuális memóriájába is leképezett fizikai memóriaterületek:
    - operációs rendszer kódja és adatai
    - kommunikációs adatok

Virtuális memória (5)

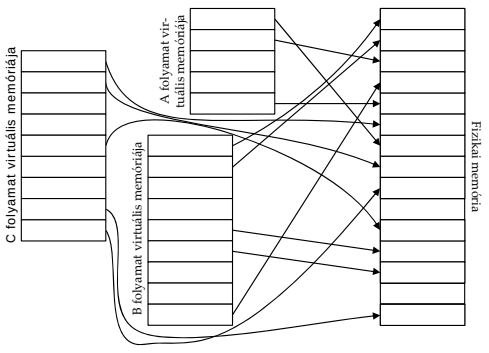
- Leképezés:
  - Az összetartozó virtuális és fizikai területek címeit tárolni kell
    - bájtonként: rugalmas, de a táblázat mérete elfoglalná a fizikai memória nagy részét
    - rögzített méretű darabonként (lapozás): lásd később
    - változó méretű darabonként (szegmentálás): lásd később

Tartalomjegyzék

Lapozás (1)

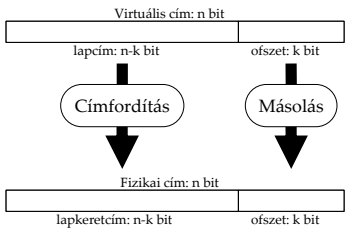
- Mind a virtuális, mind a fizikai memóriát egyenlő méretű darabokra osztjuk fel:
  - Virtuális memóriában lapok
  - Fizikai memóriában lapkeretek
  - Szemléletes elnevezés: van egy csomó lapunk, de csak akkor tudunk dolgozni velük, ha keretbe tesszük őket; a keretek száma azonban kisebb, mint a lapoké
- Egy lapon belüli címek a fizikai memóriában is egy lapkereten belül lesznek

Lapozás (2)



Lapozás (3)

Lapok mérete: általában 2^k:



Címfordítás (1)

- Lapok és lapkeretek egymáshoz rendelését egy táblázat tartalmazza: laptábla
- Minden folyamatnak külön laptáblája van
- Laptábla a lapcím szerint van indexelve
- Mezői tartalmazzák, hogy a laphoz tartozik-e lapkeret a fizikai memóriában, és ha igen, mi ott a címe
- Hivatkozás során ez a lapkeretcím a táblázatból a lapcím helyére másolódik

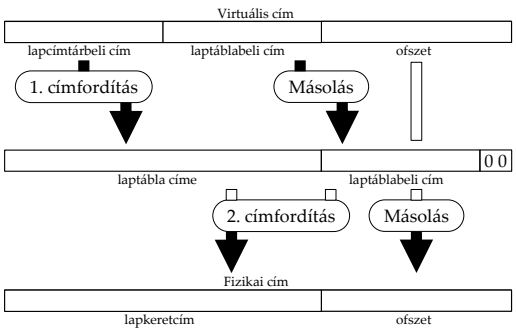
Címfordítás (2)

- Ha nem tartozik hozzá lapkeret: laphiba kivétel történik, és a kivételkezelő feladata a lapot valamelyik lapkeretben elhelyezni
- A laphiba kivétel mindig hiba (fault), azaz a végén a kivételt okozó utasítás ismételten végrehajtásra kerül
- Kettős memóriahivatkozás kiküszöbölése: gyakori lapcím-lapkeretcím párosok egy gyorsítótárban is szerepelnek

Lapméret

- Nagy lapméret:
  - Kis laptáblázatok
  - Belső elaprózódás: folyamatok a lapméretre való kerekítés miatt több memóriát kapnak, mint amennyire szükségük van
- Kis lapméret:
  - Kevés memória veszik kárba a kerekítés miatt
  - Nagy laptáblázatok, amik elfoglalják a fizikai memória nagy részét
- Megoldás: két szintű laptáblázatok

Két szintű laptáblázat (1)



Két szintű laptáblázat (2)

- Felső szint: lapcímtár
  - Lapcímtár állandóan a memóriában van
  - Bejegyzései a lapcím felső része által vannak indexelve, és a megfelelő laptáblák címét tartalmazzák
- Alsó szint: laptáblák
  - Nem mindig vannak a memóriában
  - Ha nincsen benn a hivatkozott laptábla, akkor laphiba keletkezik
  - Kivételkezelő feladata a laptábla betöltése

Laptábla szerkezete (1)

- Optimális megoldás: egy laptábla egy lapot foglaljon el
  - Ekkor a nem jelenlévő laptábla miatti kivétel kezelése nem különbözik a nem jelenlévő lap miatti kivétel kezelésével
  - Ha 32 bites címeket használunk, akkor 4 kilobyte-os lapok esetén kétszintű laptáblát használva bejegyzésenként 32 bittel teljesül ez a feltétel
  - Lapkeret címe viszont csak 20 bit, így fennmarad 12 bit egyéb információk tárolására

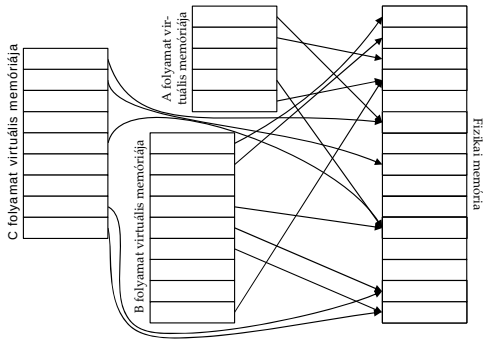
Laptábla szerkezete (2)

- További információk a bejegyzésekben:
  - Jelen van-e a lap (tartozik-e hozzá lapkeret)
  - Védelmi információk (milyen védelmi szintről hozzáférhető a lap)
  - Hozzáférés módja (írás, olvasás, végrehajtás)
  - Gyorsítótár információk (I/O területek esetén fontos lehet)
  - Hivatkoztak-e már a lapra (amióta a fizikai memóriában van)
  - Írtak-e már a lapra (amióta a fizikai memóriában van)

Közös lapok (1)

- Folyamatok nincsenek teljesen elszigetelve egymástól:
  - Operációs rendszer kódja és adatai mindegyik folyamat laptáblájába be vannak képezve
  - Két azonos programot futtató folyamatban a program kódja elég, ha egyszer szerepel
  - Folyamatközi kommunikáció gyakran hatékonyabb közös memóriaterületen, mint rendszerszolgáltatások segítségével

Közös lapok (2)



A fork rendszerhívás

- Láttuk: *fork* létrehoz egy másolatot a folyamatról, de gyakran rögtön felülírjuk ezt a másolatot ⇨ nem hatékony
- Hatékony megvalósítás:
  - Folyamat összes lapját csak olvashatóra állítjuk
  - Másik folyamat lapjait ugyanazokra a lapkeretekre képezzük, szintén csak olvashatóan
  - Ha valamelyik folyamat írni próbál egyik lapjára, kivétel történik, ekkor másolunk ténylegesen

Laphiba kezelése

- Hiányzó lapot be kell tölteni a háttértárról, vagy létre kell hozni egy üres lapot
- Ha van üres lapkeret ⇨ oda betehető a lap
- Ha nincs üres lapkeret ⇨ ki kell dobni egy lapot
  - Kérdés: melyiket dobjuk ki
  - Válasz: sok különböző algoritmus létezik
  - Laphiba kezelése sok erőforrást igényel ⇨ cél, hogy minél kevesebb legyen belőle

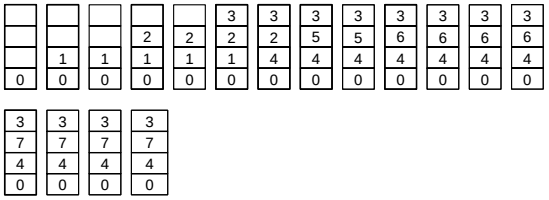
Tartalomjegyzék

Az optimális algoritmus (1)

- Minél kevesebb laphiba: mindig azt a lapot kell kidobni, amelyre a legkésőbb lesz szükség
- Ekkor minden két laphiba között maximális lesz az időkülönbség ⇨ laphibák száma minimális

Az optimális algoritmus (2)

8 lap és 4 lapkeret, hivatkozási sorrend:
0 1 0 2 0 3 4 5 0 6 4 0 3 7 4 7 0
A memória az egyes hivatkozások után:

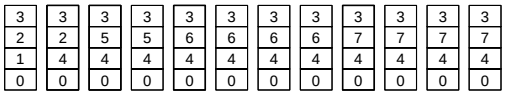


Az LRU algoritmus (1)

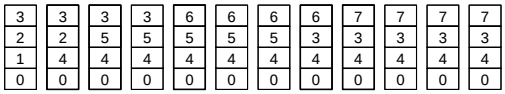
- Nem tudjuk előre, hogy melyik lapra fognak a legkésőbb hivatkozni
- Amit tudunk: programok lokalitásának elve: egy program rövid időn belül kis memóriaterületet használ, ugyanazokra a lapokra sokszor hivatkozik
- Következmény: amire már régen hivatkozott, arra a közeljövőben nem is valószínű, hogy hivatkozik ⇨ azt kell kidobni, amire a legrégebben hivatkozott (LRU=least recently used)

Az LRU algoritmus (2)

8 lap és 4 lapkeret, hivatkozási sorrend:
0 1 0 2 0 3 4 5 0 6 4 0 3 7 4 7 0
Optimális algoritmus:



LRU algoritmus:



Az LRU algoritmus (3)

- Probléma: hogyan követjük nyomon a hivatkozásokat
  - Operációs rendszer: nagyon lelassulna
  - Hardver: nagyon bonyolult lenne
- További probléma: hol tároljuk a hivatkozások idejét
  - Laptábla méretét nagyon megnövelné
- Megoldás: LRU-t is csak közelítjük

A FIFO algoritmus (1)

- A FIFO az LRU legegyszerűbb közelítése: azt dobjuk ki, amelyik a legrégebb óta benn van
- Megvalósítása:
  - sor adatszerkezettel
  - körben járó lapkeretmutatóval (számlálóval)

A FIFO algoritmus (2)

8 lap és 4 lapkeret, hivatkozási sorrend:
0 1 0 2 0 3 4 5 0 6 4 0 3 7 4 7 0
LRU algoritmus:

3	3	3	3	6	6	6	6	7	7	7	7	
2	2	5	5	5	5	5	3	3	3	3	3	
1	4	4	4	4	4	4	4	4	4	4	4	
0	0	0	0	0	0	0	0	0	0	0	0	

FIFO algoritmus:

3	3	3	3	6	6	6	6	6	6	6	0
2	2	2	0	0	0	0	0	0	4	4	4
1	1	5	5	5	5	5	5	7	7	7	7
0	4	4	4	4	4	4	3	3	3	3	3

A FIFO algoritmus (3)

- Probléma: program egyszerre több laphalmazzal is dolgozik, és ezeknek nem üres a metszetük
  - LRU ezt figyelembe veszi, FIFO nem ⇨ FIFO általában több laphibát okoz
- Bélády-anomália: előfordulhat, hogy több lapkerettel több laphiba történik

A Bélády-anomália

5 lap, hivatkozási sorrend:
0 1 2 3 0 1 4 0 1 2 3 4
3 lapkeret:

		2	2	2	1	1	1	1	1	3	3
	1	1	1	0	0	0	0	0	2	2	2
0	0	0	3	3	3	4	4	4	4	4	4

4 lapkeret:

			3	3	3	3	3	3	2	2	2
		2	2	2	2	2	1	1	1	1	
	1	1	1	1	1	1	0	0	0	0	4
0	0	0	0	0	0	4	4	4	4	3	3

Az NFU algoritmus (1)

- Láttuk: laptáblákban gyakran van egy bit, ami jelzi, hogy írtak-e már a lapra, amióta a fizikai memóriában van (A, mint *accessed* vagy R, mint *referenced*)
- Ötlet: bizonyos időközönként vizsgáljuk meg ezeket, adjuk egy számlálóhoz, nullázzuk le, és laphiba esetén azt dobjuk ki, ahol ez a számláló a legkisebb
- Probléma: ha egyszer hosszú időn keresztül hivatkoztak egy lapra, utána nem, az még sokáig előnyt élvez

Az NFU algoritmus (2)

- Javítás: a számlálót toljuk el jobbra, és a legmagasabb helyiértékre írjuk az új bitet!

A	Számláló								Számláló							
1	1	0	0	1	1	1	0	0	1	1	0	0	1	1	1	0

- Benn lévő lapok számlálói laphibakor:

1	0	0	1	1	1	0	0
1	1	0	0	0	1	1	1
0	0	0	1	1	1	0	0
0	1	0	0	1	1	1	0
1	1	0	1	1	1	1	1
0	0	0	0	1	1	0	1
0	0	1	1	1	1	0	1
0	0	0	0	1	1	1	1

⇦ Ezt kell kidobni

Az NFU algoritmus (3)

- Algoritmus neve: NFU (*not frequently used*)
- Belátható: mentes az anomáliától
  - Verem típusú: ugyanazon hivatkozási sorrend esetén a nagyobb memória mindig tartalmazza a kisebb összes lapját
- Problémák:
  - Túl hosszú számláló nem fér el
  - Nullázások és eltolások: költséges műveletek

A második esély (1)

- FIFO alapproblémája: legrégebben benn lévő lapot akkor is kidobja, ha arra nemrég hivatkoztak
- Javítás: vizsgáljuk meg, hogy hivatkoztak-e rá az utóbbi időben, és ha igen, kapjon még egy esélyt:
  - sor első (vagy a mutatott lapkeret) lapjának hivatkozási bitje 1, akkor nullázzuk, és a sor végére tesszük (mutatót léptetjük ciklikusan)
  - ha 0, akkor kidobjuk, és újat a sor végére tesszük (mutatót a következőre állítjuk ciklikusan)

A második esély (2)

8 lap és 4 lapkeret, hivatkozási sorrend:
0 1 2 3 4 1 2 0 1 2
FIFO algoritmus:

3	3	3	3	3	3	2
2	2	2	2	2	1	1
1	1	1	1	0	0	0
0	4	4	4	4	4	4

Második esély algoritmus:

3^A	3^A	3	3	0^A	0^A	0^A
2^A	2^A	2	2^A	2^A	2^A	2^A
1^A	1^A	1^A	1^A	1^A	1^A	1^A
0^A	4^A	4^A	4^A	4^A	4^A	4^A

Módosított lapok

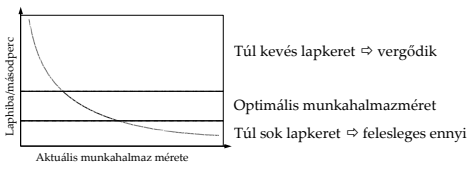
- Lapcserék költsége különbözik: ha olyan lapot dobunk ki, amit csak olvastak, nem kell kiírni a lemezre
- Laptáblában gyakran van egy bit, ami jelzi, hogy módosították-e a lapot (D, mint *dirty* vagy M, mint *modified*)
- Ezt soha nem nullázzuk!!!
- Eddigi algoritmusok kiegészíthetők: olyat próbálunk kidobni, amire nem írtak
  - De: továbbra is az számít elsődlegesen, hogy mikor használták utoljára!

Globális és lokális lapozás

- Globális: nem tekintjük a folyamatokat
- Lokális: fizikai memóriát felosztjuk a folyamatok között, és folyamatonként alkalmazzuk a lapozási algoritmusokat
  - Folyamat által rövid időn belül használt lapok: szükséges munkahalmaz (working set)
  - Folyamatnak jutó lapkeretek: aktuális munkahalmaz
  - Ha a szükséges nagyobb, mint az aktuális: túl sok laphiba: vergődés
  - Ha az aktuális nagyobb, mint a szükséges: pazarlunk

A munkahalmaz mérete

- Feladat: az igényelt és az aktuális munkahalmaz mérete hasonló legyen
- Megoldás: a laphibák gyakoriságáról (*page fault frequency*) statisztikát kell vezetni:



Többszintű lapozás

- Láttuk: lokális algoritmusok eleve két szintűek:
  - Alsó szinten lapcserék
  - Felső szinten munkahalmaz méretének állítása
- Másik szintezés:
  - Alsó szinten lokális lapcserék, de nem közvetlenül a háttértárra, hanem egy közös tárolóba
  - Felső szinten globális lapcsere ebből a tárolóból
  - Ekkor laphiba kivétel történik olyankor is, ha a lap a közös tárolóban van

Előlapozás

- Eddig arról volt szó, hogy a rendszer akkor tölt be egy lapot, ha laphiba történik
  - Kivételek feldolgozása erőforrásigényes
  - Háttértárról célszerű nagyobb összefüggő adatmennyiséget beolvasni
- Előlapozás: betöltünk olyan lapokat is, amikre eddig nem történt hivatkozás
  - Program első lapjai új program indításakor
  - Statisztikák: melyik lap után általában mely lapokra történt hivatkozás

Tartalomjegyzék

Szegmentálás (1)

- Szegmens: egy tetszőleges méretű lineáris címtér
  - Minden szegmensnek egyedi azonosítója van, ami független a fizikai memóriában elfoglalt helyétől
  - Szegmensazonosítók és fizikai memóriabeli kezdőcímek egymáshoz rendelése a szegmenstáblában van
- Szegmentált memória: több lineáris címtérből álló virtuális memória
- Egy program különböző adatszerkezetei külön szegmensekben helyezhetők el

Szegmentálás (2)

- Előnyei a lapozáshoz képest:
  - Rugalmasság: ha egy adatszerkezet mérete változik, nem változik a többinek a címe
  - Szegmentálás jobban illeszkedik a program szerkezetéhez: munkahalmaz jobban definiálható
  - Különböző típusúak lehetnek: csak olvasható/írható-olvasható adat, csak végrehajtható/olvasható-végrehajtható kód
  - Könnyebben megvalósítható, strukturáltabb adatmegosztás folyamatok között

Szegmentálás (3)

- Hátrányai a lapozáshoz képest:
  - Hosszabb címek a plusz szegmensazonosító miatt
  - Bonyolultabb algoritmusok a változó méret miatt
  - Védelem nehezebb: mi van, ha egy rendszerhívásnak egy a rendszer területén lévő adatszerkezet címét adjuk át?
    - Lapozás: címtér elvágása rendszer-felhasználó részre ⇨ egy összehasonlítás

Szegmentálás megvalósítása

- Egyesíteni kell:
  - A dinamikus partíciók módszerénél látott elhelyezési algoritmusokat ⇨ szegmenselhelyezési algoritmusok
    - Több szegmens, mint partíció ⇨ memória jobban töredezik ⇨ néha elkerülhetetlen a tömörítés
  - A lapozásnál látott cserélő algoritmusokat ⇨ szegmenscserélési algoritmusok
    - Nem mindig fér be a szegmens az előző helyére ⇨ néha többet is ki kell dobni
    - Nagy szegmens kidobása költséges lehet
    - Kis szegmenst gyakran nem éri meg kidobni

A szegmentálástáblázat (1)

- Szegmensek leírói: deszkriptorok
- Deszkriptor tartalma:
  - Szegmens fizikai címe
  - Szegmens típusa (kód vagy adat)
  - Benn van-e a szegmens a fizikai memóriában
  - Hivatkoztak-e a szegmensre, amióta benn van
  - Szegmens elérési módja (írás, olvasás vagy végrehajtás)
  - Szegmens védelmi szintje
- Deszkriptortábla indexei: szelektorok

A szegmentálástáblázat (2)

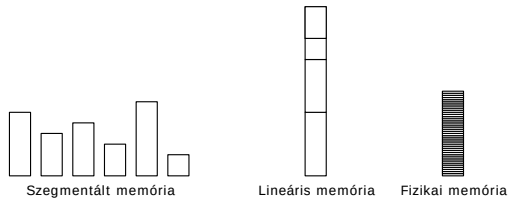
- Szegmensregiszterek: egy szelektort tartalmaznak
  - Lehet alapértelmezett adat- és kódszegmens
  - Előnye: nem kell minden hivatkozásnál explicit megadni a szelektort ⇨ rövidebb címek
  - Nem alapértelmezettre való hivatkozás: regiszter azonosítója általában rövidebb, mint egy szelektor
- Szegmensregiszter árnyékregisztere: szelektorhoz tartozó deszkriptor van benne
  - Célja: többszörös memóriaelérés elkerülése (mint lapozásnál a TLB)

Szegmentálás és védelem

- Minden folyamatnak külön deszkriptortáb-la ⇨ lokális és globális deszkriptortáblák
  - Globális: rendszer szegmensei és lokális táblák deszkriptorai
  - Lokális: felhasználói szegmensek deszkriptorai
- Hamis szelektorok problémája: rendszerhívás egy rendszerszegmens szelektorát kapja
  - Szelektorokban is állítható a védelmi szint ⇨ rendszerhívás felhasználói szintre állítja ezt, és így próbálja meg elérni az adott területet

Szegmentálás lapozással (1)

- Általában: szegmensméret >> lapméret
- Ötlet: szegmensek eleje és vége laphatárra igazodjon



Szegmentálás lapozással (2)

- Szegmensek határainak változtatása: csak a lineáris memóriában kell mozgatni az adatokat, ami a laptáblák átírását jelenti
- Cserélő algoritmusok két szinten alkalmazhatók:
  - Alsó szinten lapcserék
  - Felső szinten szegmenscserék

Tartalomjegyzék

A Linux memóriakezelése

- Virtuális memóriakezelés:
  - Lineáris, lapozott memóriamodell
- Memória felosztása:
  - Nem lapozható terület
    - Hardver biztosítja, vagy az operációs rendszer statikusan kezeli a laptáblákat (identitás leképezéssel)
    - Kernel teljes egészében itt foglal helyet ⇨ önmagát nem lapozza
  - Lapozható terület
    - Felhasználói programok

Egy folyamat területei (1)

- A program kódja és statikus adatai
  - Kezdetben nem tartozik hozzá lapkeret
  - Igény szerint lapozódik be, közvetlenül a végrehajtott állományból
  - Egy program kódja csak egyszer van benn a fizikai memóriában
- Dinamikusan foglalt területek
  - Kezdetben szintén nem tartozik hozzá lapkeret
  - Igény szerint jönnek létre üres (kinullázott) lapkeretek

Egy folyamat területei (2)

- A programhoz dinamikusan (futásidőben) szerkesztett könyvtárak
  - Kezelésük a program kódjához és statikus adataihoz hasonlóan történik
- Memóriaterület-leíró
  - Folyamatleíró része
  - Egy láncolt lista a folyamat memóriaterületeiről
  - Elemek egy-egy terület típusát, helyét, méretét és lehetséges műveleteit tartalmazzák
  - Lista kezelése dinamikusan történik új terület foglalásakor illetve régi felszabadításakor

Gyorsítótárak

- Lapgyorsítótár
  - Lefutott programok lapkeretei ide kerülnek
  - Célja: ha rövid időn belül valamelyik folyamat ismét elindítja ugyanezt a programot, nem kell újra a háttértárról betölteni
- Puffergyorsítótár
  - Háttértárakról és más beviteli eszközökről olvasott blokkok itt őrződnek meg
  - Célja: ne kelljen őket újra a megfelelő eszközről beolvasni
  - Ha az adatok közben megváltoznak a háttértáron: rendszer érvényteleníti

Laptáblák

- Absztrakt szinten három szintű laptáblák
- Makrók az adott architektúra laptábláira képezik le őket
- Előny: architektúra-függetlenség
- Hátrány: hatékonyság nem optimális

Lapok fajtái (1)

- Gyorsítótárak lapjai
  - Ha kevés a szabad lap, először ezektől szabadul meg
  - Nem kell a háttértárra írni (hatékonyság)
  - Minden folyamatot egyformán érint (globális algoritmus)
- Osztott memóriaterületek lapjai
  - Óvatosan kell kidobni: előbb minden érintett folyamatnál nem jelenlevőre kell állítani
  - Ciklikusan kerülnek kidobásra

Lapok fajtái (2)

- Egyéb lapok
  - Egy-egy folyamathoz tartoznak ⇨ kidobásuk csak az adott folyamatot érinti
  - Lapcserélő algoritmus (LRU közelítés)
    - Bizonyos időközönként vizsgálja a lapok hivatkozási bitjét
    - Ha hivatkoztak rá ⇨ hárommal növel egy számlálót
    - Ha nem ⇨ eggyel csökkenti azt
    - Számláló értéke 0 és 20 között lehet, kezdetben 3
    - Ha eléri a 0-t ⇨ azonnal kidobja
    - Vizsgálat után nullázza a hivatkozási bitet

Lapcserék

- Egy speciális rendszerfolyamat figyeli a szabad lapok számát
  - Ha egy adott szint alá csökken, kidob 3 lapot
  - Ezután vár, és folytatja a tevékenységet, amíg a szabad lapok száma el nem éri a kívánt szintet
  - Ha a működés ellenére egy kritikus szint alá csökken a szabad lapok száma, akkor ezentúl 3 helyett 6 lapot dob ki, és két kidobás között feleannyi ideig vár
- Előlapozás: mindig két lap kerül betöltésre: a hivatkozott, és az azt követő

Tartalomjegyzék

A VMS memóriakezelése

- Virtuális memóriakezelés:
  - Lineáris, lapozott memóriamodell
- Memória felosztása:
  - Nem lapozható terület
    - Csak a kernel bizonyos részei (pl. zárak)
  - Lapozható terület
    - Felhasználói programok
    - Kernel nagy része

Egy folyamat területei

- Felhasználói program kódja (P0)
- Felhasználói program adatai (P1)
- Rendszer (S0)
  - Minden folyamatnál azonos
  - Közös laptábla
  - Folyamatok laptáblái is itt vannak
- Folyamatok laptáblái: PHD-ben (folyamatleíró változó méretű részében)
  - Egyszintűek ⇨ méretük nagyon nagy is lehet ⇨ lapozhatók

Lapgazdálkodás (1)

- Munkahalmaz:
  - Mérete adott határok között dinamikusan változik
    - Méret határai a folyamat tulajdonosától függnek
    - Változás függ
      - a rendelkezésre álló szabad lapok számától
      - a folyamat által időegység alatt generált laphibák számától
- Lapkeret felszabadítása: szabadlistára kerül
  - Tartalma megőrződik (azaz: szabadlista egyben gyorsítótár is)
  - Könnyű laphiba: innen töltődik be vagy üres lap

Lapgazdálkodás (2)

- Lapcserélési algoritmus: FIFO
  - Ez a súlyos laphibák szempontjából különösen hatékony LRU közelítés!
  - Munkahalmaz méretének változtatásakor a könnyű laphibák is számítanak ⇨ anomáliák megjelenhetnek a növeléssel, de a további növelésekkel eltűnnek
- Laphiba kezelése:
  - Amíg a lap betöltődik, más folyamat fut (aktuális folyamat állapota: PFW)
  - Az adott lapot követő néhány másik lap is betöltődik

Lapgazdálkodás (3)

- Szabadlista:
  - Újra felhasználható lapok: kinullázás után tetszőleges folyamatnak adhatók
  - Módosított lapok: előbb ki kell írni a háttértárra, majd áttenni a másik listába
- Ha a módosított-lista túl nagy, elindul a kiírás (SWAPPER rendszerfolyamat egy szála)
- Ha ez a lista kritikusan nagy lesz, a rendszer megtiltja a további lapkeretek kiosztását: folyamatok RWMPB állapotba

Vészhelyzetek kezelése

- Láttuk: folyamatok munkahalmazai függenek a szabadlista méretétől, azaz ha a szabadlista mérete túlzott mértékben lecsökken, a rendszer csökkenti az összes munkahalmaz méretét
- Ha a munkahalmazok már minimálisak: egész folyamatok kidobása:
  - Összes lap a csereállományba
  - PHD is megszűnik
  - Visszatöltődéskor az összes kiírt lap egyszerre kerül vissza

Egyéb érdekességek

- Minden új folyamat úgy jön létre, mint egy vészhelyzet miatt teljes egészében kidobott folyamat
- Munkahalmazok méreteinek szabályozását, módosított lapok kiírását és a vészhelyzeti működést a SWAPPER rendszerfolyamat végzi
  - Prioritása 16
  - Ha efölötti prioritású (valós idejű) programot írunk, ezt figyelembe kell vennünk!