A memória fogalma

• A memória (tár) egy számítógépben az adatokat tárolja
• Neumann elv: programok kódja és adatai ugyanabban a memóriában tárolhatók
• Mai számítógépek szinte kivétel nélkül binárisak Ö tárak is bináris számok formájában tárolják az adatokat

Tárolt adatok fajtái

• ???
  • Folyamatos (pl. számítások részeredményei)
  • Alkalmankénti (pl. biztonsági mentések)
• ???
  • Keletkezésük után nem változnak
  • Folyamatosan változnak
• Adatok tartóssága
  • Tartós (permanens)
  • Átmeneti (tranziens)
• Adatok elérésének kulcsa
  • Adat helye
  • Adat tartalmának meghatározott része

Csak olvasható memóriák

• Kis mennyiségű adatok tárolásához: ROM
  • Egyesek és nullák helye fizikailag rögzített
  • Áramkör speciális égetési eljárással alakul ki
  • Gyors elérés, nagyobb méretben azonban drága
  • Firmware kódok és adatok tárolása
• Nagy mennyiségű adatok tárolásához: optikai tárolás (CD-ROM, DVD-ROM)
  • Anyag benyomódásai és kiemelkedései jelölik a nullák és az egyesek helyét
  • Anyag speciális eljárással egyszer alakítható ki
  • Mechanikus szerkezet miatt lassú elérés, de olcsó gyártás
  • Telepítendő programcsomagok, multimédia

Egyszer írható memóriák

• Néha a felhasználó is szeretne házilag adatokat elhelyezni csak olvasható memóriában
  • Saját firmware (esetleg saját készülékhez)
  • Saját szerzemények tárolása
  • Megoldás: egyszer írható, többször olvasható memóra
• ROM egyszer írható megfelelője: PROM
• CD-ROM és DVD-ROM egyszer írható megfelelője: CD-R és DVD±R
  • Figyelem! Csak véglegesnek szánt adatokat írjunk CD-R-re, mert a felesleges CD-R és DVD±R veszélyes hulladék. Átmeneti tároláshoz használjunk CD-RW-t illetve DVD±RW-t,

ami nem sokkal drágább!

Ritkán írható memóriák

• Előfordul, hogy a véglegesnek szánt adatokat mégis meg kell változtatni
  - Programhibák javítása
  - Új protokollok, interfészek támogatása
  - Saját művünk továbbfejlesztése
• EPROM: ultraibolya lámpával törölhető PROM
• EEPROM: szoftveresen törölhető, de írás lassú
• CD-RW, DVD±RW: törölhető, és törlés után újraírható
  - Módosítás nem lehetséges az ISO9660 szabvány szerint

Írható és olvasható memóriák

• Nagyon sokfélék lehetnek
  - Processzor regiszterei (állandó használat)
  - RAM (általában DRAM) (nagyon gyakori használat)
  - Háttértárak (változó gyakoriságú használat)
• Merevlemezek
•Flash-memóriák (nem felejtő)
  - Archív tárak (ritka használat)
• Mágnesszalagok
• CD-RW, DVD±RW, DVD-RAM lemezek UDF
formátummal
• Méretük nagyságrendileg különbözik
• Adatok elérési ideje szintén
• A kettő nagyjából egyenesen arányos

Tartós és átmeneti tárolás

• Látjuk: amihez gyakrabban férünk hozzá, azt gyorsabb memóriában kell tárolni
  - Ezek viszont kisebbek, így mindent nem lehet ott
• Megoldás
  - Lassabb, nagyobb memóriákban tárolt adatokról másolatot készítünk a gyorsabb, kisebb memóriákba
  - Ha az adatot módosítottuk, a munka végén visszaírjuk a lassabb, nagyobb memóriákba
  - Azaz: lassabb, nagyobb memóriákban (háttér- és archív tárak) tartósan tároljuk az adatot (ezek áramforrás nélkül sem felejtik el tartalmukat)
  - Kisebb, gyorsabb memóriákban (regiszterek, RAM) átmenetileg, a munka idejére (állandó áramforrás kell, hogy ne felejtsék el tartalmukat)

Gyorsítótárak (cache)

• Az egyes szintek között külön gyorsítótárak is lehetnek
  - Regiszterek és a RAM között (általában SRAM)

• Processzoron belül
• Processzor és RAM közötti sínen
•Lehetnek általános és speciális célú gyorsítótárak
    - RAM és háttértár között
• Háttértárba beépítve
• RAM egy része erre a célra fenntartott (szoftver)
• Gyorsítótárak célja: az előbb vázolt elvet, miszerint minél gyakrabban használunk egy adatot, annál gyorsabb memóriában tároljuk alkalmazva a a hierarchiát finomítja

Adatok elérésének kulcsa

• Adat fizikai helye
    - Regiszter neve
    - Memóriacím
• Adat tartalmának meghatározott része:
asszociatív elérés
    - Állománynév
    - Gyorsítótárban az eredeti memóriabeli cím
• Fontos még az egyszerre elérhető adatok mennyisége is
    - Bájt
    - Gépi szó
    - Blokk

Asszociatív gyorsítótárak

• El kell tárolni az eredeti címet is, kiolvasásnál össze kell hasonlítani
• Közvetlen leképezésű cache
    - Cím alsó bitjei képezik a cache-beli címet
    - Minden adat helye egyértelmű
• Kiolvasáskor egyetlen összehasonlítás kell (olcsó)
•Egyértelmű az is, hogy mi kerül ki a cache-ből
• Cache-kihasználtsága nem tökéletes
• Teljesen asszociatív cache
    - Cache-beli cím tetszőleges
• Kiolvasáskor az összeset (párhuzamosan) meg kell vizsgálni (drága)
• Bonyolult meghatározni, hogy melyik cím-adat pár
kerüljön ki a cache-ből, ha az már betelt
• Csoport-asszociatív cache
    - Hibrid megoldás
    - Cím alsó bitjei nem címet, csak egy csoportot határoznak meg
    - Egy csoporton belül több cím is lehetséges
•Ha n cím van, akkor n utas csoport-asszociatív cache
•Ez alapján a közvetlen leképezésű 1 utas, a teljesen

asszociatív meg annyi utas, amekkora maga a cache
• Kiolvasáskor n párhuzamos összehasonlítás
• Csak csoporton belül kell eldönteni, hogy melyik a törlendő
• Általában nem bájtokat, gépi szavakat, hanem szomszédos szavakból álló úgynevezett cache vonalat tárolnak (pl. 4 gépi szó, azaz 32 bájt)

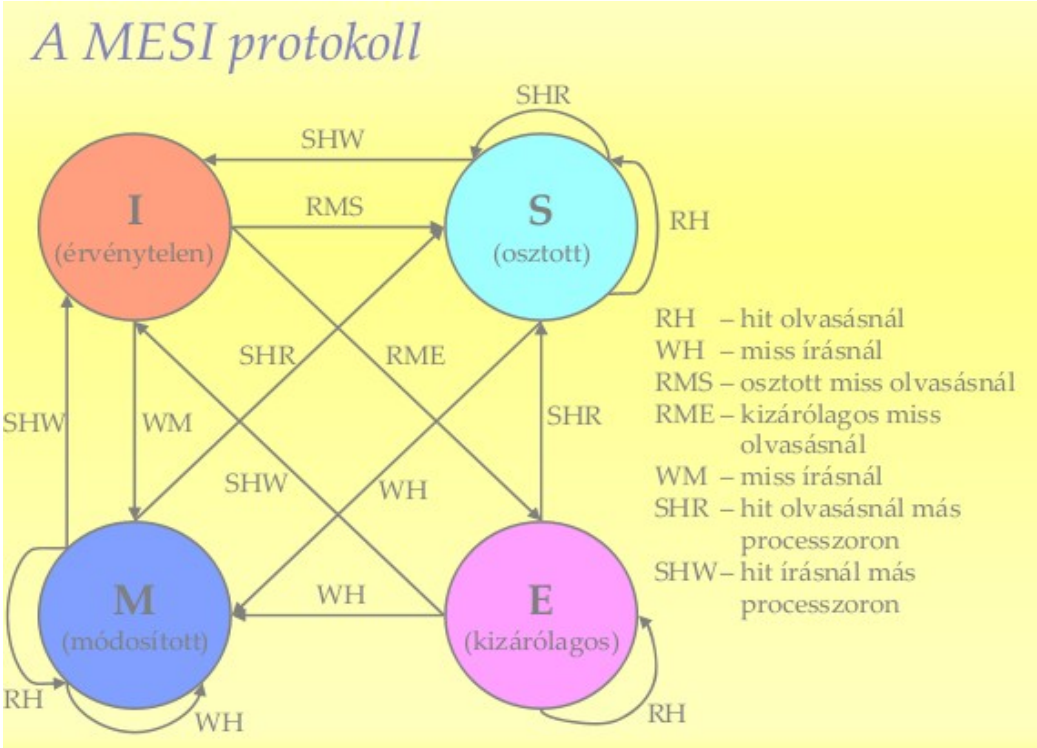
A gyorsítótárak működése

• Olvasás
  - A processzor memóriakezelő egysége (MMU) nem közvetlenül a RAM-hoz, hanem a cache-hez fordul
  - Ha a cache-ben van a tárolt érték (cache-hit), akkor onnan töltődik be (gyors)
  - Ha nincs ott (cache-miss), akkor a RAM-ból töltődik be (lassú), és közben a cache-be is bekerül

• Írás
  - Write-through cache: beíródik a RAM-ba (lassú), és közben a cache-ben is bekerül vagy módosul
  - Write-behind cache: csak a cache-be íródik be (gyors), a RAM-ba csak akkor, ha a cache-ből kikerül

A gyorsítótárak problémái

• RAM-ba képzett bemeneti eszközök
  - Processzor nem tudja megkülönbözteti a RAM-tól
  - Ha a cache-be egyszer bekerült egy érték, mindig onnnan olvas, hiába jönne más a bemeneti eszközről
  - Megoldás: bizonyos címekre tiltani kell a cache használatát
• RAM-ba képzett kimeneti eszközök
  - Write-behind cache esetén nem kerül ki az adat az eszközre időben (vagy egyáltalán nem)
  - Megoldás: bizonyos címekre csak write-through cache engedélyezhető
• Többprocesszoros rendszerek
  - Ha közös adatokkal dolgoznak, akkor gondoskodni kell az adatok konzisztenciájáról
• Write-through cache alkalmazása önmagában nem elég: ha egy másik processzor a saját cache-ében eltárolt egy régi adatot, akkor azt fogja használni
• Cache tiltása a közösen használt memóriaterületen nagyon elrontaná a hatékonyságot
•Megoldás: ha egy processzor módosít egy adatot, akkor nem elég azt kiírnia a központi memóriába, hanem a többi processzor cache-ében is érvényteleníteni kell
•Hardver támogatás szükséges hozzá
  - Szabvány: MESI protokoll



Lemezgyorsítótárak

• Eddig a processzor és a RAM közötti gyorsítótárakról volt szó
• Lemezbe épített gyorsítótár
  - Működhet write-behind elven, de biztosítani kell,

hogy áramszünet esetén legyen energia a
tartalmának kiírására
• Szoftver által a RAM-ban megvalósított gyorsítótár
  - Csakis ideiglenes adatoknál késleltetheti a kiírást
  - Közös háttértáras osztott rendszer esetén hasonló protokollt kell megvalósítani, mint a MESI

Memóriakezelési feladatok

Adatok mozgatása a memóriák között

• Processzor-RAM gyorsítótáraknál és a lemezbe épített gyorsítótáraknál a hardver átlátszatlanul mozgatja az adatokat
• Memória és a processzor között a programok utasítási mozgatnak
• Háttértár és a memória között a felhasználó illetve programjai mozgatnak adatokat (programbetöltés, állományok megnyitása stb.)
• Archív tár és a háttértár között a felhasználó mozgatja az adatokat (mentés, visszatöltés, telepítés)

Operációs rendszer memóriakezelési feladatai

• Adatokat el kell helyeznie a központi memóriában (RAM)
  - Programokat tölt be
  - Adatterületeket biztosít programok számára

• Fel kell szabadítania a feleslegessé váló adatok területeit
• A szabad területet nyilván kell tartania
• Meg kell osztania a memóriát a különböző
folyamatok között

Programok betöltése

• Feladat: végrehajtható programot (adatainak kezdőértékével együtt) be kell olvasni a háttértárról a memóriába, és ott el kell indítani
• Végrehajtható program meghatározott címekkel
dolgozik
    - Ugróutasítások meghatározott címekre ugranak
    - Adatok rögzített címeken vannak
    - Mindig ugyanoda kellene tölteni
• Memória megosztása: minden folyamat más címtartományhoz férhet hozzá, így nem biztos, hogy mindig ugyanoda tudjuk tölteni
• Lehetséges egoldások:
    - Egyszerre csak egy program legyen a memóriában,
többi a háttértárakon: nem hatékony
• Javítás: futó program a memória alján, a többi feljebb: még így sem hatékony (sok mozgatás)
    - A rendszer programbetöltéskor átírja a beolvasott
program hivatkozott címeit, mielőtt elindítaná
• Nehéz megtalálni a címeket (pl. ha relatív címzést is használunk) Ö nem hatékony
    - Mindenhol báziscímzést használunk
• Csak a bázisregisztert kell átírni a program elindítása előtt
• Bázisregiszter használata lehet implicit (nem kell feltüntetni a címzésnél)

Felhasználói adatterületek

• A memóriában új adatok is keletkezhetnek
    - Felhasználói program állítja elő
    - Háttértárról töltődik be
    - Felhasználó adja meg (interakció)
    - Hálózatról töltődik le
• Kell egy szabad terület, ahol elfér
    - Operációs rendszer feladata ennek a területnek a biztosítása
    - Szükséges rendszerszolgáltatás: memóriaigénylés
    - A kijelölt terület kezdőcímével kell visszatérnia
    - Párja: felszabadítás

Memória felosztása

• Sok módszer ismert:
    - Partícionálás (régen)
• Fix számú és méretű partíció

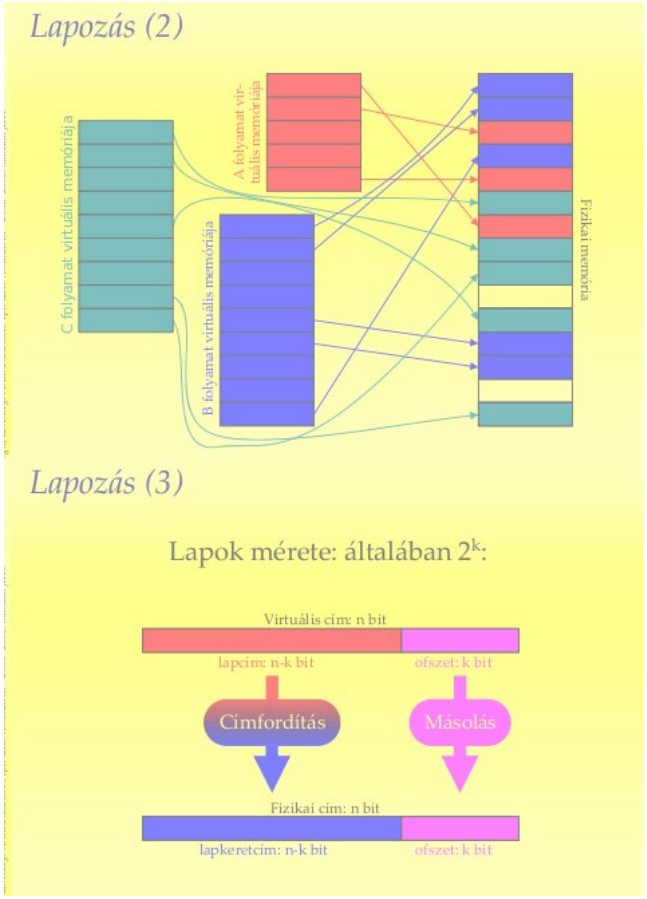
• Dinamikusan változó számú és méretű partíciók
    - Virtuális memóriakezelés (modern rendszerekben)
• Lapozás
• Szegmentálás
• Hibrid megoldások
• Egyes módszereken belül is sok különböző
megvalósítás, sok különböző algoritmus létezik

Virtuális memória

• Alapproblémák:
    - Előfordulhat, hogy egy program teljes egészében egyáltalán nem fér be a központi tárba
    - Mozgatás a háttértárra és vissza sok erőforrást igényel
    - Programok lokalitásának elve: egy program rövid idő alatt csak kis részét használják a tárterületüknek
    - Biztonsági probléma: nehéz elérni, hogy a folyamatok ne nyúlhassanak ki a partíciójukból
    - Nem mindig tudhatjuk a program indulásakor, hogy maximálisan mennyi memóriára lesz szüksége, azt dinamikusan kell lefoglalni
• Ötlet: programok által látott memóriaterület különbözzön a fizikailag létezőtől!
• 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
    - Program ne vegyen észre semmit a megvalósítás módjából
    - Virtuális memória elérésének hatékonysága ne legyen sokkal rosszabb, mint a fizikaié

Lapozás

• 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
• Felhasználói programok csak a lapokat látják,
mintha az lenne a fizikai memória



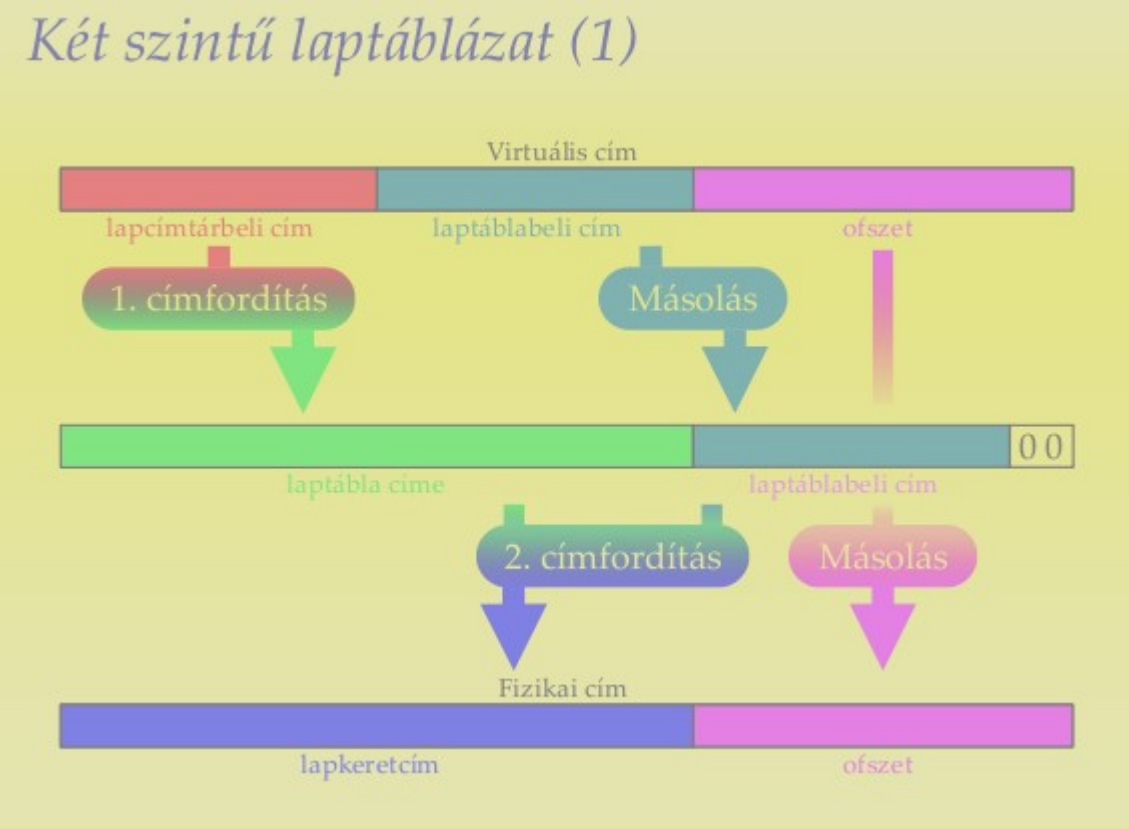
Címfordítás

• 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
• 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: TLB

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
  - Laptábla lapozása bonyolult
• Megoldás: két szintű laptáblázatok



Két szintű laptáblázat

• 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

• 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

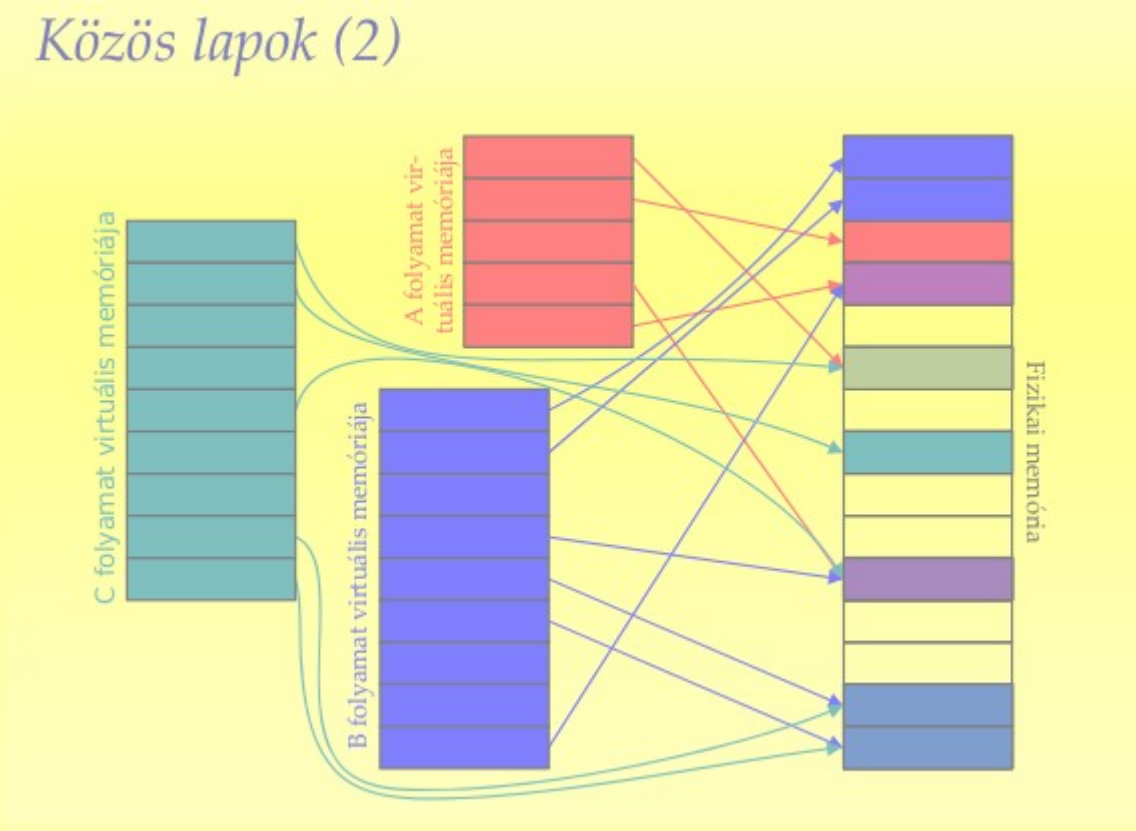
- Pentiumban ezt a módszert alkalmazzák

• 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)
- Gyorsítótár információk (cache-elt lap, ha igen, write-through vagy write-behind)
- Hivatkoztak-e már a lapra (amióta a fizikai memóriában van)
- Írtak-e már a lapra (amióta a fizikai memóriában van)

• Ha nem, és megvan a háttértáron, akkor nem kell újra kiírni

Közös lapok

• 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



A fork rendszerhívás

• Láttuk: fork létrehoz egy másolatot a folyamatról, de gyakran rögtön felülírjuk ezt a másolatot Ö 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 Ö oda betehető a lap
• Ha nincs üres lapkeret Ö ki kell dobni egy lapot
- Kérdés: melyiket dobjuk ki
- Válasz: sok különböző algoritmus létezik
- Laphiba kezelése sok erőforrást igényel Ö cél, hogy minél kevesebb legyen belőle

Lapcserélési algoritmusok

Az optimális algoritmus

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

Az LRU algoritmus

• 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 Ö azt kell kidobni, amire a legrégebben hivatkozott (LRU=least recently used)
• 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

• A FIFO az LRU legegyszerűbb közelítése: azt dobjuk ki, amelyik a legrégebb óta benn van
• Megvalósítása: sor adatszerkezettel
• Probléma: program egyszerre több laphalmazzal is dolgozik, és ezeknek nem üres a metszetük
- LRU ezt figyelembe veszi, FIFO nem Ö FIFO általában több laphibát okoz
• Javítás: vizsgáljuk meg, hogy hivatkoztak-e rá az utóbbi időben, és ha igen, kapjon még egy esélyt:
- sor első lapjának hivatkozási bitje 1, akkor nullázzuk, és a sor végére tesszük
- ha 0, akkor kidobjuk, és újat a sor végére tesszük

- vizsgálat után a hivatkozási bitet nullázzuk

Javítási módszerek

• Eddigi algoritmusok kiegészíthetők: olyat próbálunk kidobni, amire nem írtak
  - De: továbbra is az számít elsődlegesen, hogy mikor használták utoljára!
• 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

Globális és lokális lapozás

• Globális: nem tekintjük a folyamatokat
• Lokális: fizikai memóriát felosztjuk a folyamatok
között, és folyamatonként alkalmazzuk a
lapozási algoritmusokat
  - Memória felosztásánál figyelembe kell venni az igényeket
  - Igények felmérése: statiszikát kell vezetni arról, hogy melyik folyamat hány laphibát okozott az utóbbi időben, és ennek megfelelően adni neki vagy elvenni tőle lapot
• Gyakorlatban alapvetően lokális algoritmusokat alkalmaznak

Szegmentálás

• 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

A szegmentálás előnyei és hátrányai a lapozáshoz képest

- Rugalmasság: ha egy adatszerkezet mérete változik, nem változik a többinek a címe
- Szegmentálás jobban illeszkedik a program szerkezetéhez
- Különböző típusúak lehetnek: adat és kódszegmensek
- Könnyebben megvalósítható, strukturáltabb adatmegosztás folyamatok között
- Hosszabb címek a plusz szegmensazonosító miatt
- Bonyolultabb algoritmusok a változó méret miatt

Szegmentálás megvalósítása

• A lapozásnál látott cserélő algoritmusokat tovább kell fejleszteni → 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
- Szegmensek között lyukak keletkezhetnek, így sokkal bonyolultabb a memória karbantartása

A szegmentálástáblázat

• 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

• 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
  - Így nem kell mindig a deszkriptortáblához nyúlni
  - Pentium: szegmensregiszterbe szekektor töltésekor a deszkriptor betöltődik a megfelelő árnyékregiszterbe

Szegmentálás és védelem

• Minden folyamatnak külön lokális
deszkriptortábla és egy globális deszkriptortábla
  - Globális: rendszer szegmensei és lokális táblák deszkriptorai
  - Lokális: felhasználói szegmensek deszkriptorai
• Közös adatterület kétféleképpen biztosítható
  - Közös szegmens a GDT-ben
  - Közös fizikai memóriaterületre leképzett szegmensek a saját LDT-kben

Szegmentálás lapozással

• Általában: szegmensméret >> lapméret
• Ötlet: szegmensek eleje és vége laphatárra igazodjon
• Ekkor szegmentálhatunk és lapozhatunk is

Lemezegységek felépítése

• Logikai felépítés: blokkok egydimenziós tömbje
    • 1 blokk általában 256 vagy 512 bájt
    • Le kell képezni a fizikai felépítésre
• Fizikai felépítés: cilinder, sáv, szektor
    • a lemez közepe felé haladva a sávon belüli szektorok

száma csökkenhet (pl. C1541)

    • CD, DVD lemezeken nincsenek sávok; a szektorok fizikailag is egy spirális alakba „feltekert”

Fizikai és logikai formázás

• Fizikai formázás
    • A sávok és szektorok kialakítása az adathordozó felületen
    • Általában már a gyárban elvégzik, de szükség esetén a felhasználó megismételheti (pl. szektorméret-változtatás céljából)
    • A szektoroknak „láthatatlan” fej- és láblécük van, benne a szektor számával és a hibajavító kóddal
• Logikai formázás
    • A felhasználás előkészítése
    • Két lépésből áll:

1. A nyers lemezterület felszeletelése elkülönített részekre (partíciókra)
2. Az egyes partíciókon a felhasználáshoz szükséges adatszerkezetek felépítése (a fájlrendszer létrehozása)

Boot blokk

• A lemez első blokkját a rendszer speciális célra tartja fenn, neve boot blokk
• Egy rövid programot tartalmaz, mely a memóriába olvassa az operációs rendszer magját, és ráadva a vezérlést elindítja a rendszert
• A rendszer bekapcsolásakor a ROM-ba huzalozott rendszerindító rutin olvassa be és indítja el a boot blokkot
• A rendszerindítás általában többszintű, bonyolult folyamat

Bevitel/kivitel ütemezés (1)

• Alacsonyszintű IO kérések szerkezete:
    • kérés fajtája (olvasás vagy írás)
    • a kért blokk száma (vagy fizikai cím)
    • pufferterület címe a memóriában
    • mozgatandó bájtok száma
• Általában egy lemezegységet egyszerre több folyamat is használni akar
    • Több IO kérés is kiszolgálásra várakozik
    • Melyiket hajtsuk végre először?

• Az író olvasó fej mozgatása sokáig tart
    • Fejmozgási idő (seek time)
    • Egy cilinderen belül nem kell mozogni
• Ha már a megfelelő cilinderen állunk, meg kell várni, míg a megfelelő szektor a fej alá pörög
    • Elfordulási idő (rotational latency)
• Nem mindegy, hogy milyen sorrendben olvassuk, írjuk a blokkokat
• Az IO ütemező feladata a kérések kiszolgálási sorrendjének „jó” megválasztása
    • Fejmozgások, elfordulási idő minimalizálása
    • Átlagos válaszidő csökkentése, sávszélesség növelése
    • Cserébe nő a CPU igény (overhead)

Sorrendi ütemezés (FCFS)

• First Came, First Served
    • A kéréseket egyszerűen a beérkezés sorrendjében szolgáljuk ki
    • Hasonló a FIFO ütemezéshez
• Nem törődik a fej mozgásával
    • Hosszú válaszidő, kis sávszélesség
    • Cserébe a válaszidő szórása kicsi
    • Igazságos ütemezés, kiéheztetés nem fordulhat elő

Lusta ütemezés (SSTF)

• Shortest Seek Time First
    • A kéréseket a kiszolgáláshoz szükséges fejmozdulás sorrendjében szolgálja ki
    • Lásd SJF ütemezés
• A fejmozdulások töredékükre csökkennek
    • Sávszélesség látványosan megnő
    • Válaszidő szórása nagy
    • Fennáll a kiéheztetés veszélye, főleg hosszú várakozás sor esetén

Lift ütemezés (SCAN)

• Pásztázó algoritmus
    • A fej a diszk egyik szélétől a másikig ide-oda pásztáz
    • Az éppen útba eső kérést szolgálja ki
    • A liftek működéséhez hasonlít
• Jó kompromisszum
    • Sávszélesség nagy
    • Kiéheztetés szinte kizárva
    • Várakozási idő szórása viszonylag nagy
• Javítás: a visszafelé haladás közben ne szolgáljunk ki kéréseket (C-SCAN)

Előlegező ütemezés

• Anticipatory I/O scheduling
• A lift ütemezés heurisztikus javítása
• Gyakran egymás utáni blokkokat olvasnak a folyamataink (szekvenciális olvasás)
• Egy ilyen olvasás után várakozzunk egy kicsit mielőtt más kéréseket kiszolgálunk
    • Egy kicsit: néhány milliszekundum
    • Ha tudjuk, hogy a folyamat nem szekvenciálisan olvas, ne várakozzunk
    • Más információ híján várakozzunk mindig
• Párhuzamos szekvenciális olvasások esetén megspórolhatjuk az ide-oda pásztázást

• Melyik ütemező algoritmust válasszuk?
• Sorrendi ütemezés
    • Triviális implementáció, nincs is ütemező
    • Ha egyszerre csak egy kérés van, csak egyféleképpen lehet ütemezni
    • Egyfelhasználós rendszerek
• Lusta ütemezés
    • Egyszerű, természetes, hatékony
    • Túl nagy terhelés esetén fellép a kiéheztetés
• Lift ütemezés
    • Nagy IO terhelésű rendszerek
    • Fair ütemezés, megelőzi a kiéheztetést

• Az elfordulási idő is késlekedést okoz
    • Modern lemezegységeken a fejmozdulás ideje alig haladja meg az elfordulás miatti késedelmet
    • A fenti algoritmusok csak a fejmozdulási időt veszik figyelembe

Beépített ütemezés

- A lemezegység saját ütemezőt tartalmazhat (pl. SCSI-2 tagged queuing)
- Az adott modellhez illeszkedő, elfordulási időt is figyelembe vevő ütemezés
- Az OS a kéréseket ömlesztve továbbítja a lemezvezérlőnek, rábízza az ütemezést
- Az OS feladatszintű IO ütemezést is végez

(lapozás vs. egyéb, írás vs. olvasás, stb.)
Kiszolgálási idő csökkentése
• Ügyes szervezéssel hatékonyabbá tehetjük rendszerünket
    • Az összetartozó adatok legyenek egymás mellett a lemezen
    • A sávszélesség a lemez szélén a legnagyobb
    • A leggyakrabban használt adatok legyenek a lemez közepén, vagy tároljuk őket több példányban

- Olvassunk/írjunk egyszerre több blokkot
- A szabad memóriát használjuk fel lemez- gyorsítótárnak

Adattömörítéssel csökkentsük az IO

Partíciók

• A partícionálással a lemezt független szeletekre osztjuk
• A partíciók az alkalmazások és az OS magasabb rétegei számára általában a lemezegységekhez hasonló eszközként látszanak
• Az egyes partíciókat különböző célokra használhatjuk

- Nyers partíciók (pl. adatbáziskezelőknek)
- Virtuális memóriaterület (swap)
- Fájlrendszer

RAID

• Redundant Array of Inexpensive Disks
(olcsó lemezegységek redundáns tömbje)

- Ha egy diszk átlagosan 100 000 üzemóra (kb. 11 év) után mondja fel a szolgálatot, akkor egy 100 diszkből álló rendszerből kb. 42 naponta egy diszk kiesik!
- Megoldás: az adatainkat tároljuk egyszerre több diszken
- A redundancia megvalósítását rejtsük egy virtuális lemezegység mögé (nincs szükség új interfészre)
- Különböző RAID szintek közül választhatunk (RAID-0-6)

• Szoftverből és hardverből is megvalósítható

- Hardver-RAID esetén általában egész diszkeket kötünk össze, az OS szemszögéből az eredmény egy szokásos lemezegységnek látszik
- Szoftver-RAID-et az OS valósítja meg, így partíciók felett is működhet
- A hardver megvalósítás drágább, de hatékonyabb

RAID 0 (Striping)

• Néhány diszk tárterületének összefűzésével megsokszorozhatjuk az összefüggő(nek látszó) tárkapacitást
• A logikai diszk blokkjait általában felváltva osztjuk szét a fizikai diszkek szektorai között (striping)
• Az IO műveletek párhuzamosításával nő a teljesítmény
• Nincs redundancia!
• Az adatvesztés esélye nem csökken, hanem nő
• Általában a blokknál nagyobb egységeket kezelünk (stripe), de akár bitszintű szétosztás is lehetséges

RAID 1 (tükrözés)

• Minden adatot két független diszken tárolunk

• A tárolókapacitás a felére csökken
• Olvasási teljesítmény nőhet, írás nem változik, vagy kissé csökken
• Diszkhibából eredő adatvesztés esélye jelentősen csökken
• Egyszerű, de drága megoldás
    • Nagyon kicsi processzorigény
    • 1 GiB adat tárolásához 2 Gib diszkterület szükséges

RAID 2 (ECC)

• A gépi memóriánál megszokott hibajavító kódok használata (ECC memória)
• Az adatbitek mellett néhány extra bitet is tárolunk (lásd Hamming kódok)
• A bájt bitjeit és a hibajavító biteket tároljuk különböző diszkeken
• Az egyik diszk meghibásodása esetén a paritásbitekből helyreállítható a hiányzó bit
• pl. 4 diszk kapacitáshoz elég 7 fizikai diszk
• A gyakorlatban ritkán használt

RAID 3 (paritásbitek)

• A memóriával ellentétben a P
lemezegységek jelzik, ha hiba történik
• Nincs szükség a teljes hibajavító kódra, elég egyszerűen a paritásbitet tárolni (XOR)
• Az ismert pozíciójú hibás bit ebből helyreállítható
• Előnyök:
    • Olcsó: n diszk kapacitáshoz elég n+1 diszk
    • Egy blokk írása/olvasása szétosztódik a diszkek között, tehát felgyorsul
• Hátrányok:
    • Magasabb CPU igény
    • I/O műveletekben az összes diszk részt vesz, a párhuzamos teljesítmény romlik

RAID 4 (paritásblokkok)

• A RAID 0 megoldást egészítsük ki egy P paritásdiszkkel
• Nincs szükség a bájtok felszabdalására, a paritás független blokkokra is számolható
• Egy diszk kiesése esetén a paritásdiszk és a többi diszk blokkjaiból helyreállíthatók
az adatok
• Előnyök:
    • A RAID 3-hoz hasonlóan olcsó
    • Egy blokk beolvasásához elég egyetlen diszk, így a független olvasások párhuzamosíthatóak
• Hátrányok:
    • Az egyedi olvasásműveletek sebessége csökken
    • Az írások nem párhuzamosíthatóak (a paritásdiszket minden írás használja)
    • A diszkek igénybevétele nem egyforma

RAID 5 (elosztott paritásblokkok)

• A RAID 4 javítása: a paritásblokkokat keverjük az adatblokkok közé
• Például egy 5 diszkből álló tömbben az n. blokkhoz tartozó paritásblokkot tároljuk az (n mod 5)+1. diszken, a többi diszk n. blokkjai tárolják az adatokat
• A diszkek igénybevétele kiegyenlítődik
• Az írások némileg párhuzamosíthatók, azonban még mindig jóval lassabbak az egyszerű tükrözésnél

RAID 6 (P+Q) P Q

• Paritásblokk (P) mellett hibajavító kódok (Q, Reed-Solomon)
• n+2 diszk költséggel n diszknyi kapacitást nyújt, és bármely két diszk kiesését elviseli
• Matematikai háttér: Galois-terek
• Jelentős, a RAID 5-nél is magasabb CPU igény
• Elvileg általánosítható kettőnél több diszk kiesésére, a gyakorlatban általában nem éri meg
• A P és Q blokkokat célszerű itt is az adatblokkok közé keverni

RAID 0+1, RAID 1+0

• A RAID 0 teljesítményét ötvözhetjük a RAID 1 megbízhatóságával, ha kombináljuk a kettőt
• Szerencsés esetben a tömb akár egyszerre több diszk kiesését is elviseli
• A RAID 1+0 némileg
megbízhatóbb
• A RAID 1-hez hasonlóan drága megoldás

RAID összefoglalás

• A gyakorlatban csak a 0., 1., 5. és 6. szinteket alkalmazzák, illetve az 1+0, 0+1 kombinált megoldásokat
• A komponens diszkek méretének egyformának kell lennie
• Új diszkek menet közbeni hozzáadásával nem lehet növelni a tárkapacitást, újra létre kell hozni a tömböt
• A 6. illetve szerencsés esetben az 1+0, 0+1 kivételével valamennyi szint csak egyetlen diszk kiesését viseli el
• Választási szempontok:
    - Magas megbízhatóság: 1, 5, 6, 1+0, 0+1
    - Nagy teljesítmény: 0, 1, 1+0, 0+1
    - Alacsony költség: 0, 5, 6
    - Ezek közül bármelyik kettőt teljesíthetjük

• Minden rendszernél külön kell mérlegelni, melyik a legmegfelelőbb megoldás; gyakran több különböző RAID szintű és méretű tömböt definiálunk

• Általában lehetőség van készenléti diszkek definiálására, melyeket a rendszer automatikusan üzembe állít, ha egy diszk kiesik

- Az új diszk szinkronizációja időbe telik, ezalatt a tömb teljesítménye csökken
- A szinkronizáció közbeni új diszkhiba végzetes (a RAID-6 ez ellen védelmet nyújt)

• A redundáns tömbök nem nyújtanak védelmet minden hibalehetőség ellen

- Emberi tévedések
- Programhibák
- Az egész rendszert érintő hibák

• Váratlan leállások
• Túlfeszültség
• Természeti katasztrófák
• A fejlett operációs rendszerek kötetkezelő rendszerekkel (volume manager) könnyítik meg a hibatűrő logikai diszkek létrehozását és üzemeltetését

- Új indirekciós szint a logikai blokkok és a RAID tömbök fizikai blokkjai között
- A kötetkezelő rendszertől a partíciókhoz hasonló, de rugalmasan átméretezhető, hibatűrő logikai tárterületek (kötetek) igényelhetők
- A rábízott fizikai partíciók tárterületével a kötetkezelő automatikusan gazdálkodik