Mäluhaldus

- Mälu ja aadressid
- Dünaamiline laadimine
- Swapp'imine
- Mälu struktuurid
- Leheküljed, segmendid
- Virtuaalmälu

Mäluhaldus – Mälu ja aadressid

- Mälu koosneb suurest hulgast baitidest ning on tavaliselt grupeeritud 1, 2, 4 või 8 kaupa
- Protsessor loeb mälust programmi käske vastavalt programmi loenduri poolt näidatud asukohale
- Osad nendest käskudest võivad põhjustada uusi mälust lugemisi või sinna kirjutamisi
- Käsutsükkel koosneb järgmistest sammudest:
  - Käsu mälust lugemine ning dekodeerimine
  - Operandide mälust laadimine (vajadusel)
  - Käsu käivitamine
  - Tulemuste salvestamine mällu (vajadusel)

Mäluhaldus – Mälu ja aadressid

- Programmi võib vaadelda binaarse koodjadana, mis tavaliselt asub kettal
- Käivitamiseks tuleb programm mällu laadida
- Mällulaadimist ootavate programmide jada nimetatakse sisendjärjekorraks (*input queue*)
- Kui programm lõpetab, vabastatakse tema käes olnud mälupiirkonnad

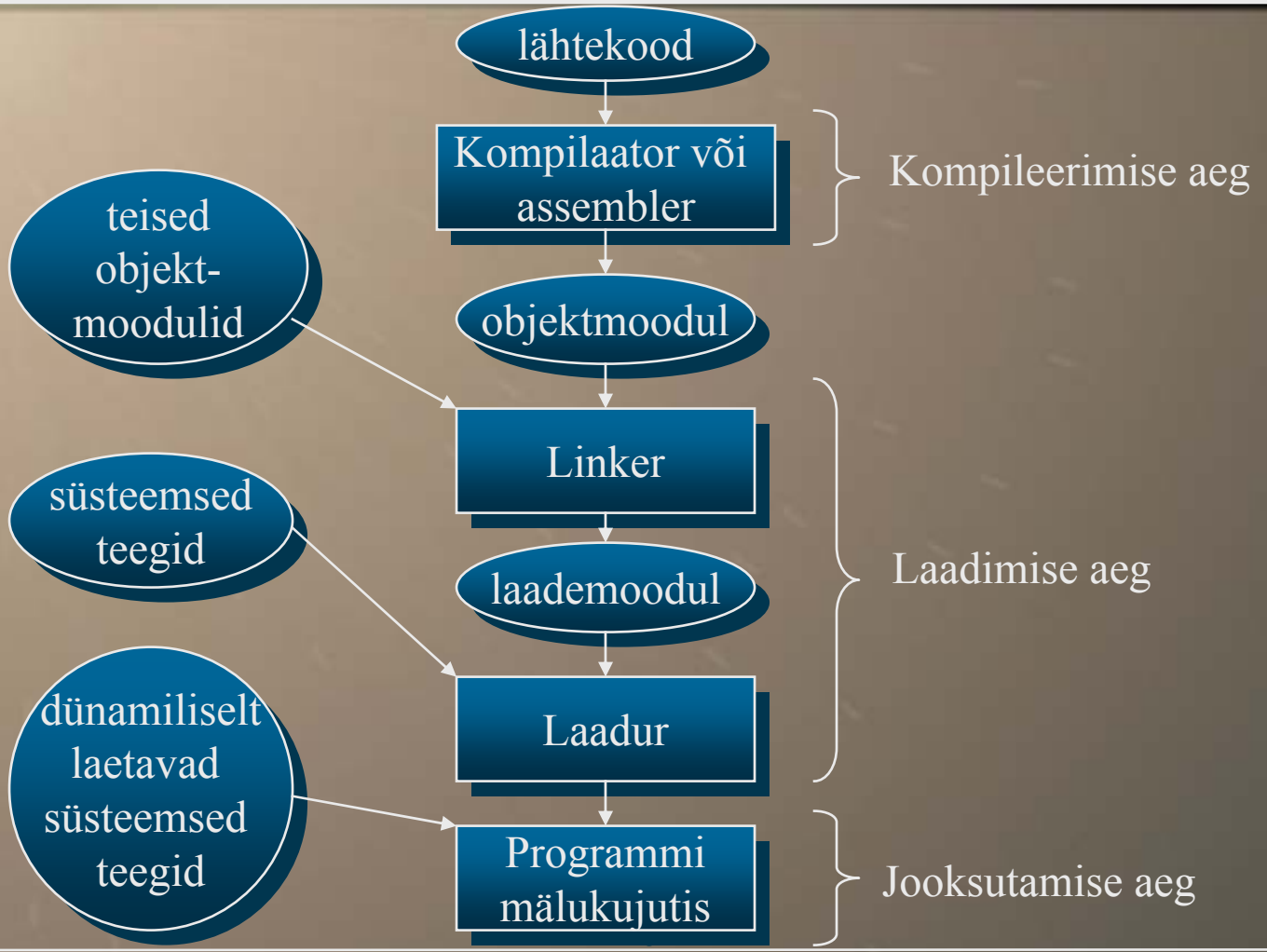
Mäluhaldus – Mälu ja aadressid

- Suurem osa süsteeme lubavad programme laadida enam-vähem suvalisse mälupiirkonda
- Osa mälupiirkondadest on OS käes (näiteks alumised aadressid alates 000000-st) ja sinna kasutaja-programme laadida ei tohi
- Programmi loomisel tuleb moodustada side tema eri osade vahel (funktsioonid, moodulid jne)
- Kuna programmile antav mälupiirkond ei pruugi kattuda varem valmis genereeritud aadressidega, siis tuleb neid kuidagi modifitseerida ja siduda (*binding*)

Mäluhaldus – Mälu ja aadressid

- Kui koodi kompileerimise ajal on teada tema tulevane asukoht mälus, võib genereerida **absoluutse** koodi (seda kasutasid MS-DOS `.com` programmid)
- Laadimise ajal saab siduda programmi ja reaalse mälu omavahel spetsiaalse tabeli kaudu (*relocation table*) ja saame **nihutatava** koodi
- Sidumise võib jätta ka koodi täitmise etapile juhul, kui programmi võib töö ajal mälus ümber paigutada (seda moodust kasutavad enamus üldkasutatavaid OS-e)

Mäluhaldus – Mälu ja aadressid

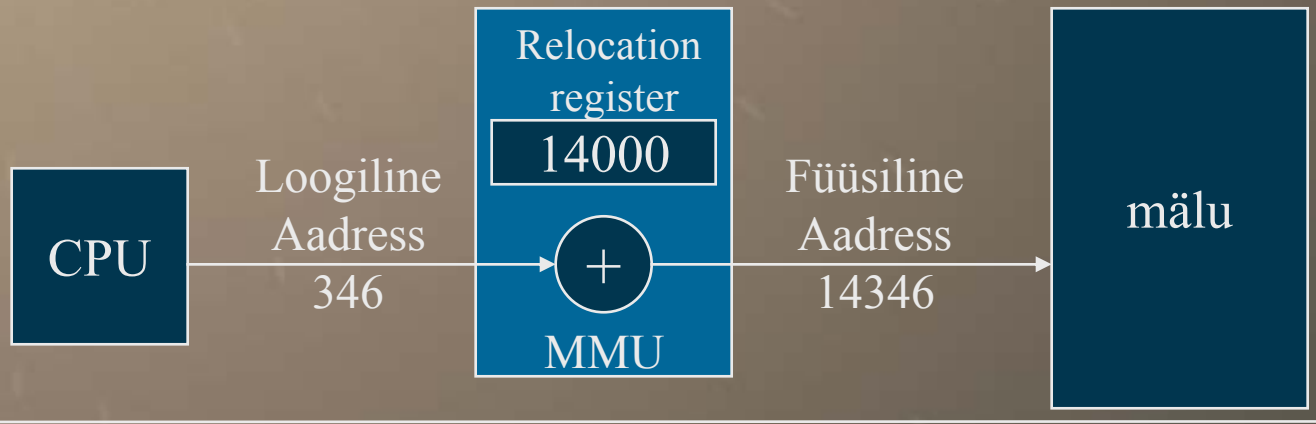


Mäluhaldus – Mälu ja aadressid

- Loogiline ja füüsiline aadress
  - Tavaliselt viidatakse (protsessi seisukohast) protsessori poolt genereeritud aadressile kui loogilisele aadressile (*logical address*)
  - Mälu poolt nähtav e. mälu aadressiregistrisse (*memory address register*) loetud väärtus on aga füüsiline (*physical address*)
  - Esimese kahe sidumise - kompileerimise ja laadimise ajal teostatava - puhul on loogiline ja füüsiline aadress samad; töötamise ajal aga võivad need erineda
  - Viimasel juhul nimetatakse loogilist aadressi tihti virtuaalseks (*virtual address*)

Mäluhaldus – Mälu ja aadressid

- Loogiline ja füüsiline aadress
  - Kõiki genereeritud loogilisi aadresse kokku nimetatakse loogilise aadressi ruumiks (*locical address space*)
  - Füüsiliste aadresside kogumikku vastavalt füüsilise aadressi ruumiks (*physical address space*)
  - Jooksva mäluaadresside sidumise eest vastutab mäluhaldur (memory-management unit = MMU)



Mäluhaldus – Mälu ja aadressid

- Loogiline ja füüsiline aadress
  - Programm ei näe kunagi tegelikku füüsilist aadressi
  - Põhimõtteliselt võib sama programmi mitu eksemplari joosta erinevate loogiliste aadresside peal, samas kui nende füüsilised aadressid võivad kattuda

Mäluhaldus – Dünaamiline laadimine

- Seni eeldasime et kogu programm loetakse mällu ühe korraga – piiravaks teguriks on sel juhul vaid mälu maht
- Mäluruumi parema kasutuse eesmärgil võime rakendada dünaamilist laadimist (*dynamic loading*) – vajaminev programmiosa laetakse mällu vaid siis, kui seda tõesti vaja läheb
- Mõlemal juhul loetakse peaprogramm mällu käivitamisel; kui dünaamilise laadimise puhul kutsutakse välja käsk, mida seni veel mälus pole, siis laetakse see sinna automaatselt

Mäluhaldus – Dünaamiline laadimine

- Eelis – harva kasutatav programmi osa ei ole meil kogu aeg jalus ja isegi kui programmi enda maht on suur, ei pruugi korraga mälus olev osa seda olla
- Mõned OS-id lubavad vaid staatilist linkimist (*static linking*), teised ka dünaamilist (*dynamic linking*)
  - Staatiline – linker ühendab kõik kasutatud teegid ühte käivitatavasse moodulisse
  - Dünaamiline – eri teegid luuakse/jäetakse eri laademoodulitesse

Mäluhaldus – Dünaamiline linkimine

- Dünaamiline linkimine
  - toimub linkimise, mitte laadimise ajal
  - kasutatakse peamiselt süsteemsete teekide puhul (programmeerimiskeele standardfunktsioonid jne)
  - programmile lisatakse pisike koodijupats (*stub*), mis määrab ära kuidas vajaminevaid funktsioone mälust leida või neid sinna laadida

Mäluhaldus – Dünaamiline linkimine

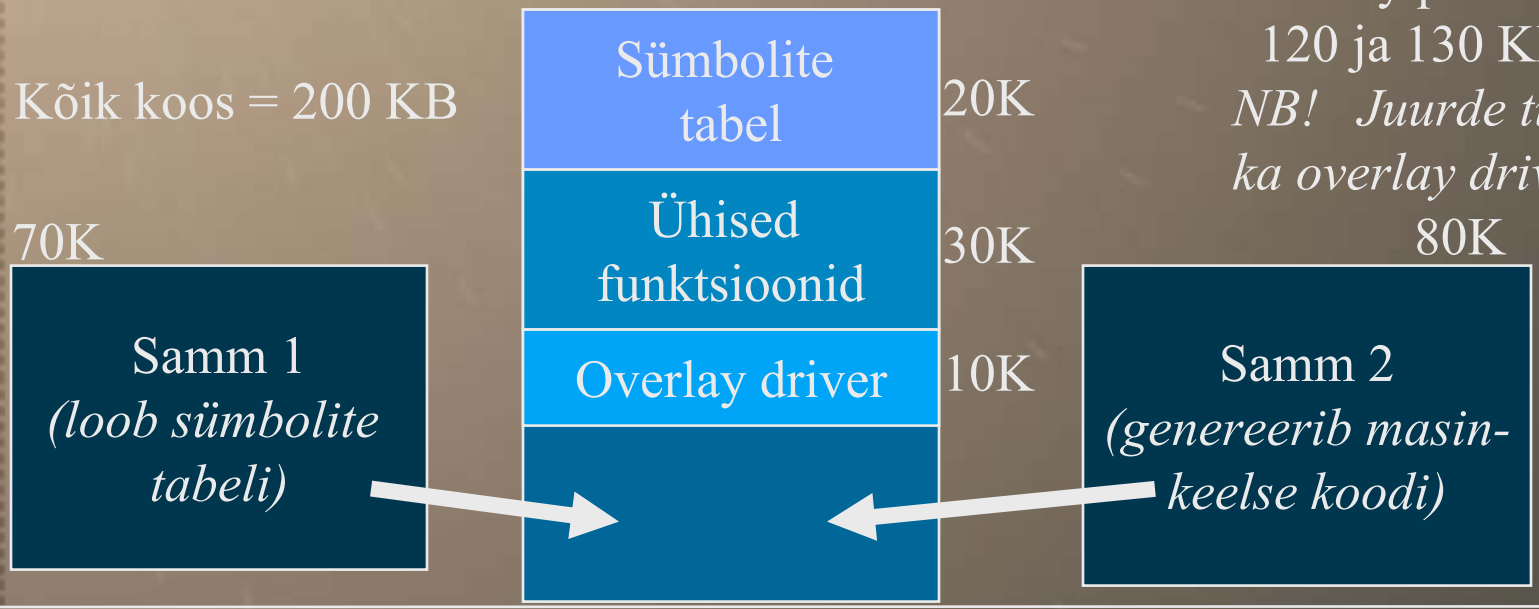
- Dünaamiline linkimine
  - "koosneb" nn. teekidest (library)
  - teegi võib iga kell asendada uuega (uus versioon või parandatud vead) ilma, et ühtegi teist seda teeki kasutavat programmi muuta tuleks *(Ilma dünaamilise linkimiseta peaks sel juhul kõik programmid uuesti kokku linkima)*
  - Jagatud teekidel (*shared library*) on tihti kaasas verisooniinfo mille põhjal igale programmile sobiv valitakse

    NB! Eri versioonide puhul tuleb olla võimalike funktsionaaluse ja/või liidese erinevuste suhtes väga hoolikas!!!
  - Erinevalt dünaamilisest laadimisest, vajab linkimine OS abi – peab teadma, kas ühele protsessi võib anda ligipääsu teise protsessi poolt kasutatavale teegile

Mäluhaldus – Overlay

- *Overlay* (ülekatte) idee seisneb selles, et mälus hoitakse vaid seda koodi osa, mida tõesti hetkel vaja läheb - kui vajatakse mingit uut teeki mida veel mälus polnud, siis visatakse vana välja ning selle **asemele** loetakse uus (assembleri näide:)

- Overlay'd hoitakse kettal absoluutsete mälupiltidena



Mäluhaldus – Overlay

- *Overlay* puhul peab **programmeerija**
  - looma korrektse *overlay* struktuuri (otsustama, millised komplektid on vahetatavad ja millised on alati vajalikud)
- See tähendab, et
  - meil peab olema programmi struktuurist ja andmetest täielik ülevaade
  - *overlay*'de loomine on väga töömahukas ja keeruline
- Praegusel ajal kasutusel mikroarvutites ja teistes süsteemides, kus on piiratud mälu või OS võimed

Mäluhaldus - Swapping

- Vastavalt protsesside planeerimise vajadustele võib osa teekidest mälust välja tõsta (*swapping*) ja hiljem jälle tagasi tuua

- Sellist mudelit võib vaadelda näiteks eri prioriteetidega protsesside juures: kui tuleb kõrgema prioriteediga protsess siis tõstetakse madalama prioriteediga välja, kuni tähtsam lõpetab ning siis jälle tagasi (*roll out, roll in*)

- Kui tegu on kompileerimise või laadimise faasis sidumisega, siis tuleb *swap*'ida kettale (vms välismälu seadmele);
kui tegu töötamise faasis sidumisega, siis võib swappida ka teisele mäluaadressile

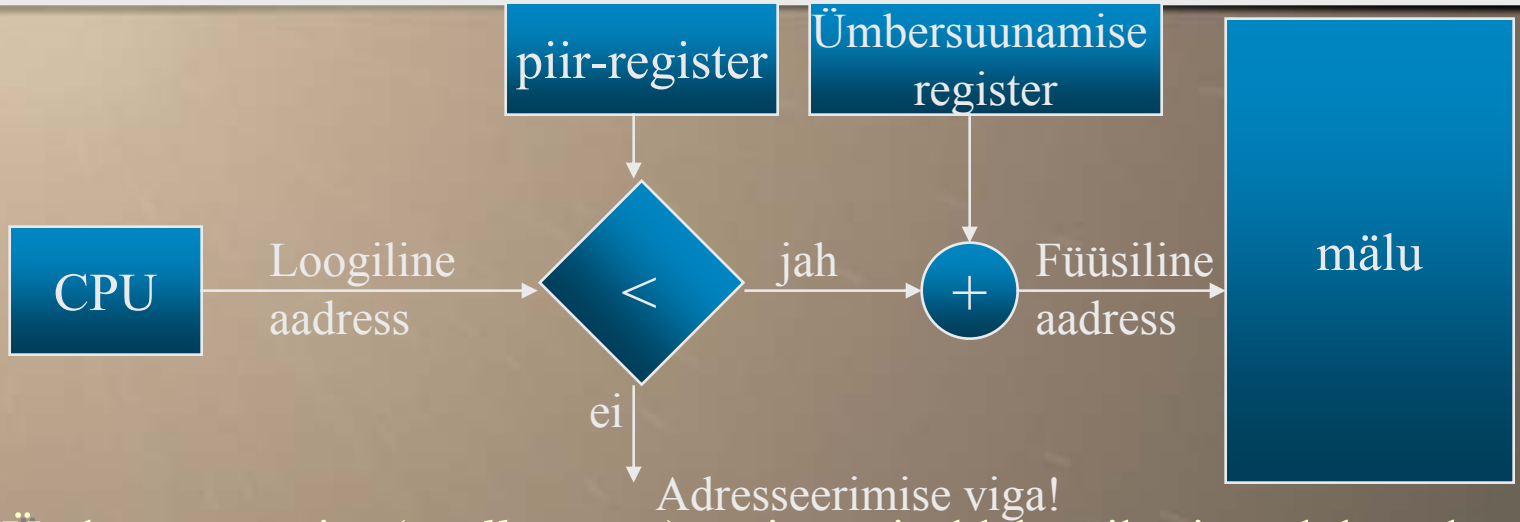
Mäluhaldus - Swapping

- Kettale salvestamine
  - aeglane
  - protsessi ajakvant peab olema suurem kui potensiaalsele swappimisele kuluv aeg
  - põhiline swappimise aeg kulub traspordile
- Selleks, et süsteem teaks kui palju mälu ühele või teisele protsessile vaja on, tuleb OS-i igast muutusest informeerida (`request memory, release memory`)
- Kui me tahame tervet protsessi swappima hakata, peame olema kindlad, et ta on jõude (*idle*) olekus (ei oota sisendi/väljundi vms taga)

Mäluhaldus – Pidev mälu allokeerimine

- Tavaliselt on mälu jagatud kahte ossa
  - OS piirkond
  - kasutaja-programmide piirkond
- OS-i võib panna mälu ette- või tahaotsa ja peamiseks otsustajaks on katkestusvektorite asukoht (tavaliselt esimestel mäluaadressidel)
- Me peame garanteerima OS-i ja muude programmide vahelise ”viisaka käitumise”

Mäluhaldus – Pidev mälu allokeerimine



- Ümbersuunamise (*reallocation*) register sisaldab väikseimat lubatud füüsilise aadressi väärtust (näiteks 100’040)
- Piiri-register (*limit register*) sisaldab lubatud loogiliste aadresside vahemikku (näiteks 74’600)
- Iga programmi poolt kasutatud aadress peab olema pisem kui piir-registri väärtus
- Füüsiline aadress jääb 100’040 ja 174’640 vahele

Mäluhaldus – Pidev mälu allokeerimine

- Kõige lihtsamaks mälujaotuse meetodiks on kogu mälu jagamine fikseeritud suurusega osadeks (*partitions*) – igas osas on 1 protsess
  Osade arv määrab multiprogrammilisuse taseme. Selline lähenemine oli kasutuse IBM OS/360 masinates (kutsutakse ka MFT)

- Teine edasiarendus (tuntud nime MVT all) sisaldab vabade mäluosade tabelit
  Kui protsess mällu loetakse, otsitakse talle sobiva suurusega auk (*hole*) mälus

Mäluhaldus – Pidev mälu allokeerimine

- Igal ajahetkel on teada vabade aukude arv
- Augud ei pruugi olla järjest
- Protsessidele mälu jagamisel võib mõni suuremate vajadustega ootama jääma ning planeerija valib järgmise
- Mälu vabanemisel liidetakse kaks kõrvuti olevat auku üheks
- Siin tuleb mängu üldine dünaamilise mälujaotuse (*dynamic storage-allocation*) probleem – kuidas rahuldada n-suurusega mälu vajadust paljude vabade lõikude nimekirja alusel

Mäluhaldus – Pidev mälu allokeerimine

- First-fit
  Valime esimese piisava suurusega augu. Niipea kui leitud, lõpetame otsimise

- Best-fit
  Otsime väikseima piisava suurusega augu. Peame läbi otsima kogu nimekirja (va. Juhul kui ta on sorteeritud)

- Worst-fit
  Otsime suurima sobiva augu. Ka sel puhul peame kogu nimekirja läbi otsima aga erinevalt esimesest võib jätta suuremaid jääkauke

Mäluhaldus – Pidev mälu allokeerimine

- Simulatsioonide põhjal on kindlaks tehtud, et esimesed kaks on efekiivsemad nii kiiruse kui ruumi kasutuse suhtes
- Neist kahest on esimene üldiselt kiirem
- Kõik kolm kannatavad välise fragmenteerumise (*external fragmentation*) all – vaba mälu jaguneb ajapikku paljudeks pisikesteks juppideks

  Kõige hullemal juhul võib iga teine blokk olla ülejääk. First-fit puhul on välja tulnud, et ~ ⅓ mälust on fragmenteerunud (iga n bloki puhul lisangub 0,5n kasutamiskõlbmatut – 50% reegel)

Mäluhaldus – Pidev mälu allokeerimine

- Kui meil on auk suurusega 18’464 ja protsess soovib saada 18’462, siis jääb meil järele 2 ühikut
- Tihti ületab selliste jääkide üle arve pidamine nendest jääkidest tekkiva probleemsuse ning seetõttu jagatakse mälu fikseeritud suurusega blokkideks
- See tekitab sisemise fragmenteerumise (*internal fragmentation*) – bloki sees olevaid jääke ei kasutata

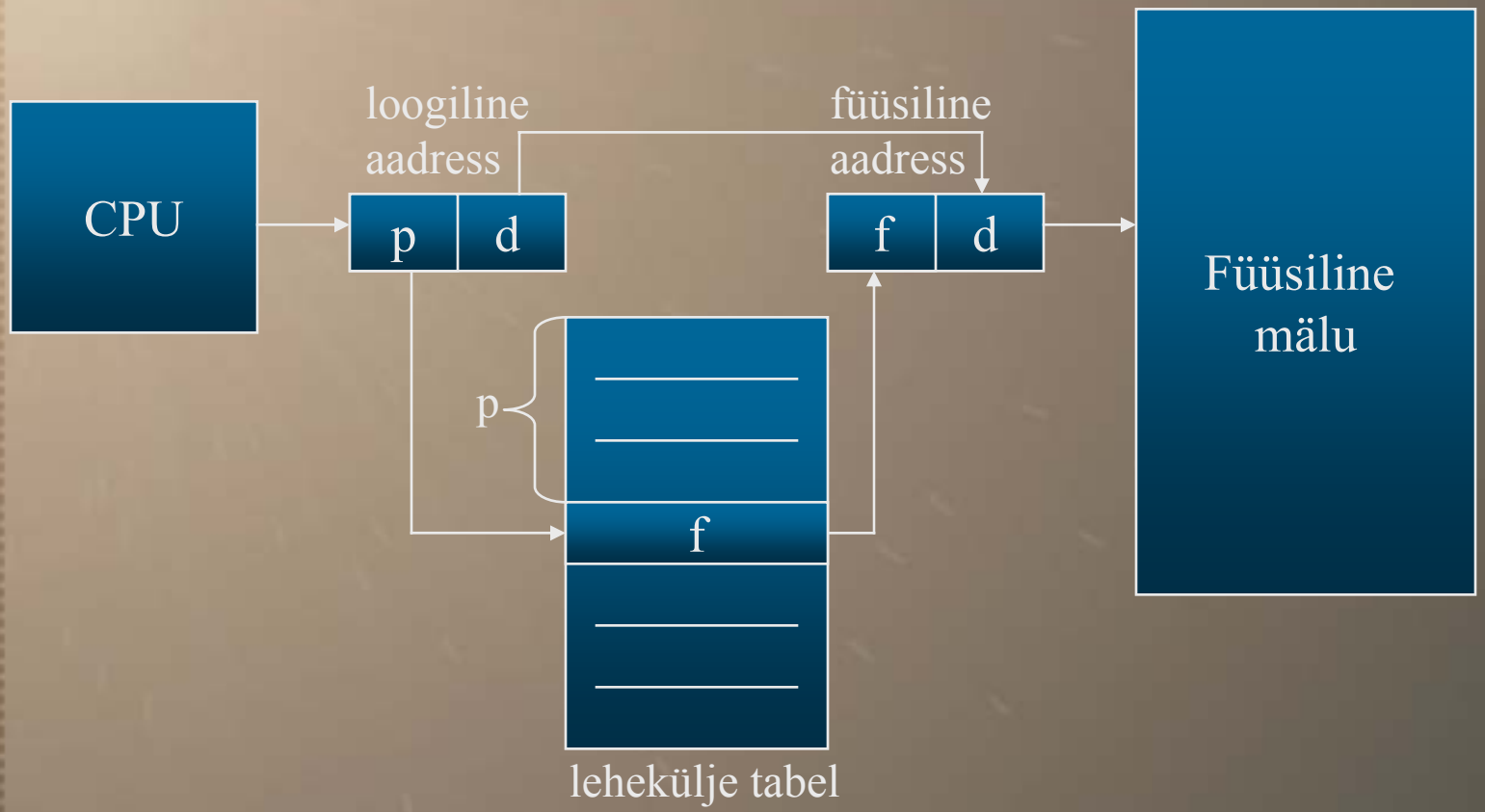
Mäluhaldus – Pidev mälu allokeerimine

- Üks välisest fragmenteerumisest üle saamise meetod on kokkusurumine (*compaction*) – nihutame kõik mälublokid selliselt, et jäägid moodustaksid ühe terviku
- Kahjuks pole see alati võimalik – näiteks kui kasutame kaht esimest dünaamilise laadimise mudelit
- Teiseks lahenduseks on võimalus jagada allokeeritud mälu tükkideks (ei pea olema üks tervik) – selle saavutamiseks on kaks meetodit:
  - Leheküljed
  - Segmendid

Mäluhaldus – Leheküljed

- Põhiline meetod
  - Füüsiline mälu jagatakse fikseeritud suurusega blokkideks – raamideks (*frames*)
  - Loogiline mälu jagatakse ka osadeks (sama suurusega) – lehekülgedeks (*pages*)
  - Protsessi käivitamisel loetakse temale antud leheküljed vabadesse raamidesse
  - Iga aadress on protsessoris jagatud kaheks
    - Lehekülje number (p) – indeks lehekülje tabelis
    - Nihe lehekülje sees (*page offset*) (d)

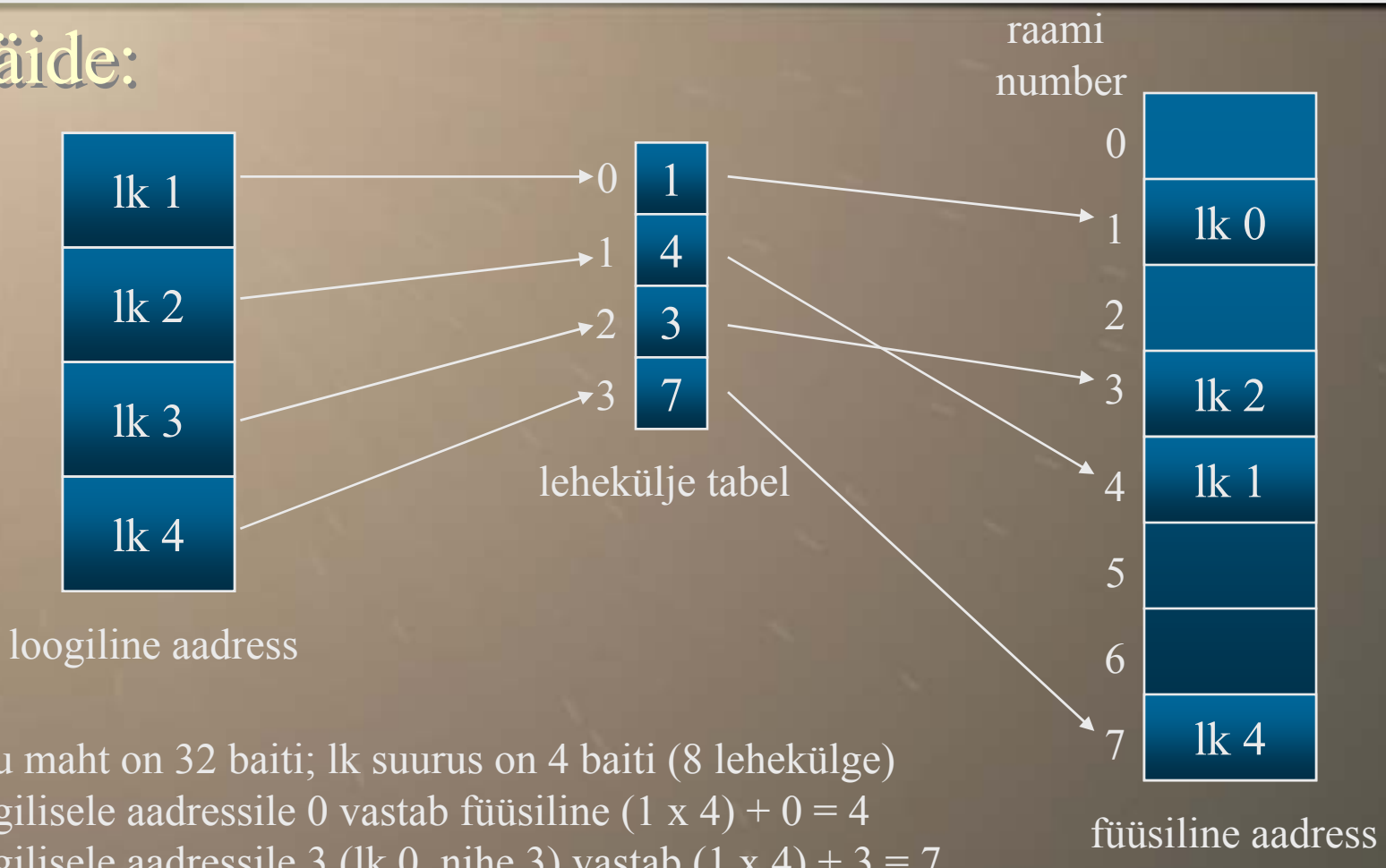
Mäluhaldus – Leheküljed



- Tavaliselt on lehekülje suuruseks 2 astmed (512 KB – 16 MB) ning see on määratud riistvara poolt

Mäluhaldus – Leheküljed

Näide:



Mälu maht on 32 baiti; lk suurus on 4 baiti (8 lehekülge)
Loogilisele aadressile 0 vastab füüsiline (1 x 4) + 0 = 4
Loogilisele aadressile 3 (lk 0, nihe 3) vastab (1 x 4) + 3 = 7

Mäluhaldus – Leheküljed

- Lehekülgede puhul väline fragmenteerumine puudub; sisemine võib aga ikkagi esineda
- Kuna allokatsioonid toimuvad fikseeritud suuruste kaupa, siis võib juhtuda, et meil on vaja protsessi mahutamiseks *n* lehekülge + veel 1 bait : seega peaaegu terve lehekülg läheb raisku
- Siit võiks järeldada, et pisemad leheküljed on paremad. Kahjuks kaasneb sellega suurem tabelite maht ning vastavalt ka töö nendega
- Osad protsessorid ja süsteemid lubavad eri suurusega lehekülgi (näiteks Solaris toetab 4KB ja 8KB lehekülgi)

Mäluhaldus – Leheküljed

- Kasutaja näeb mälu ühe tervikuna mis sisaldades vaid üht programmi; füüsiliselt võib see olla aga tükkidena laiali ja jagatud mitmete programmide vahel

- Füüsilise mälu aadressi leidmine on organiseeritud riistvaraning juhitav OS poolt. Tavaprogramm vahele ei pääse

- Selleks, et OS suudaks paremini otsustada, on olemas ka raamide tabel (*frame table*) – üks sissekanne iga raami kohta

Mäluhaldus – Leheküljed

- Tihti on lehekülgede tabelid realiseeritud riistvaras spetsialiseeritud registrite abil (mis peavad ühtlasi olema ka väga kiired)

- Kuna modernsed süsteemid lubavad väga suuri tabeleid (kuni miljoneid sissekandeid), siis selle riistvaras hoidmine võimatu. Selle asemel on 1 baasregister (*page-table base register = PTBR*), kus asetseb viit mäluaadressile, kus tegelik tabel asub – selline moodus on aga aeglane

- Teine lahendus on kasutada pisikest kiiret riistvaralist buhvrit (*associative registers; translation look-aside buffers (TLBs)*). Iga register koosneb võti-väärtus paarist (õpiku joonis 9.10).
- Tavaliselt on TLB-de arv 8 - 2048

Mäluhaldus – Leheküljed

- TLB Põhimõte
  - Tabelis on vaid mõni lehekülje ja raami number
  - Kui otsitav on tabelis, saab vastuse kiiresti
  - Kui otsitavat veel pole, siis lisatakse ta sinna (kui ruumi on)
  - Kui tabel oli täis, siis tuleb uue lisamiseks leida ”ohver”, milline uuega asendatakse
  - Kahjuks peab iga protsessi vahetusega tühjendama (*flush*) ka kogu tabeli, et mitte valele protsessile valesid aadresse anda
  - Tabeli efektiivsust mõõdetakse pihtasaamiste sagedusega (*hit ratio*)

Mäluhaldus – Leheküljed

- TLB Põhimõte
  - Oletame, et registrist sissekannet ei leitud (20 ns). Me peame pöörduma mälu poole tabeli info järele (100 ns) ning siis pöörduma mälu poole tegeliku info järele (100 ns) = kokku 220 ns
  - Efektiivne mälupöördus aeg (*effective memory-access time*) näitab kui kaua läheb meil aega andmete kättesaamiseks – arvestame tõenäosusi ühel ja teisel puhul:
    - Kui hit-ratio = 80%, saame 0.8*120 + 0.2*220 = 140 ns
    - Kui hit-ratio = 98%, saame 0.98*120 + 0.02*220 = 122

Mäluhaldus – Leheküljed

- TLB kaitse
  - Iga raamiga on seostatud kaitse-bitt
  - Tavaliselt hoitakse infot lehekülje tabelis ning see määrab, kas lehekülg on loetav/kirjutatav või üksnes loetav
  - Asja võib teha veel täpsemaks – määrates ka käivitamise õiguse
  - Lisaks nimetatule on iga lk-ga seotud kehtivuse (*valid-invalid*) bitt. Kui invalid, siis ei kuulu lk antud protsessi alla
  - Tihti ei kasuta protsessid kogu nende kätte antud mälu ja lk registreid kasutamata mälu alla panna oleks mõttetu. Selle vältimiseks on mitmel pool olemas lk-tabeli pikkuse register (*page-table length register = PTLR*)

Mäluhaldus – Leheküljed

- Mitmetasemelised (*multilevel*) leheküljed
  - Suurte mälumahtude juures oleks mõttekas mälu jagada pisemateks osadeks (joonised 9.12 ja 9.13)
  - Üheks võimaluseks on 2-tasemeline lehekülgedeks jagamine – lehekülje tabel on ise ka lehekülgedeks jagatud
  - 64-bitiste süsteemide puhul on vaja enamat – vajame 3- või enamatasemelist
  - Efektiivne mälupöörduse aeg on siin
    0.98 * 120 + 0.02 * 520 = 128 ns
    ehk meil tuleb vaid 28%-line aeglustumine

Mäluhaldus – Leheküljed

- Pööratud (*inverted*) leheküljed
  - Üldjuhul on iga lehekülgede tabel seotud protsessiga (joonis 9.14)
  - Pööratud tabel omab ühte sissekannet iga reaalse raami kohta, mis sisaldab infot virtuaalse aadress ja teda omava protsessi kohta

    <process-id, page-number, offset>
  - Kuigi tabeli hoidmiseks vajaliku mälu maht väheneb, suureneb aeg (otsing toimub virtuaalse, tabel on aga füüsilise aadressi järgi)

Mäluhaldus – Leheküljed

- Jagatud (*shared*) leheküljed
  - Üheks lehekülgede eeliseks on koodi jagamise võimalus

    Oletame et meil töötab masinaga 40 inimest, igaühel on ees tekstiredaktor (150 KB koodi ja 50 KB andmete all => 8000 KB). Kui koodi osa toetab mitut samaaegset kasutajad, võime seda jagada. Andmed on aga igaühe jaoks omad – siit saame 2150 KB

Mäluhaldus – Segmendid

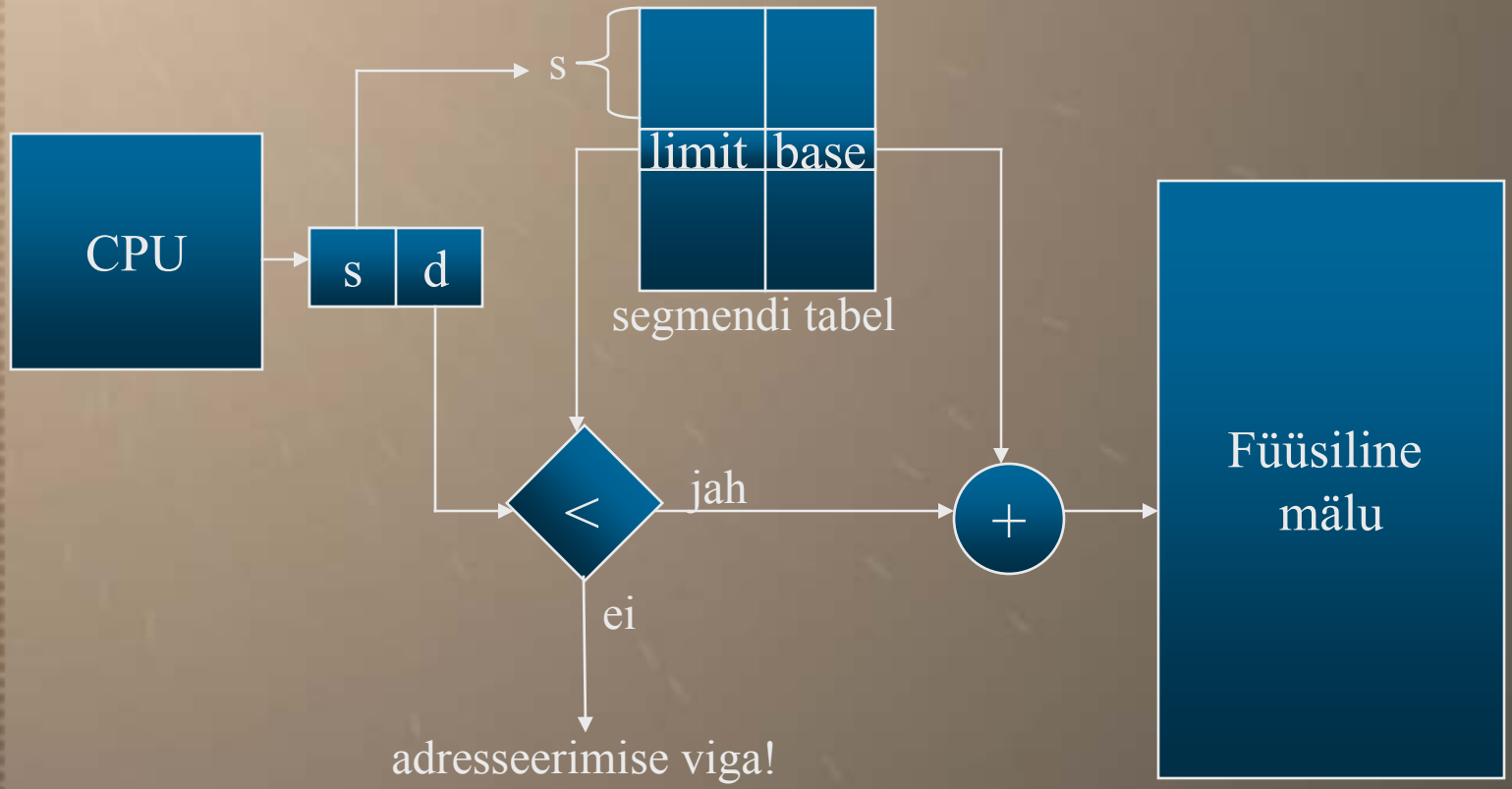
- Kuidas kasutaja näeb mälu? Kas ta vaatleb seda kui lineaarset aadresside jada?
- Ei, kasutaja vaatab mälu parema meelega kui ala kus hoitakse mingeid programmi osi – funktsioone, andmeid, teeke jne – teiste sõnadega segmente
- Loogilise aadressruumi võib jagada segmentideks, igaühel neist on nimi ja pikkus
- Aadress sisaldab segmendi nime või numbrit ja nihet selles <segmendi-number, nihe>

Mäluhaldus – Segmendid

- Segmente võib luua iga programmi jaoks nii palju kui vaja
  - näiteks Pascali puhul võib luua segmendid globaalsetele muutujatele, protseduuride väljakutseks vajalikule pinule (*stack*), programmi koodile ja lokaalsetele muutujatele iga funktsiooni jaoks
  - FORTRANi puhul võib luua näiteks eraldi segmendid iga programmibloki jaoks

Mäluhaldus – Segmendid

- Riistvaraline segmentide toetus



Mäluhaldus – Segmendid

- Kaitse ja jagamine
  - Iga segmendiga saab siduda semantiliselt tihedalt seotud kaitse – osa segmentides on kood ja teistes andmed
  - Kui näiteks paneme massiivi temale määratud segmenti, kontrollitakse automaatselt mälupiire
  - Samuti on võimalik osasid segmente jagada mitmete protsesside vahel – olgu see siis kas andmed või kood

Mäluhaldus – Segmendid

- Fragmenteerumine
  - Protsesside planeerija peab suutma allokeerida mälu kõikidele protsesside poolt tarbitavatele segmentidele
  - Segmendid on muutuva pikkusega; leheküljed fikseeritud pikkusega
  - Segmentide puhul võib kasutada first-fit või best-fit algoritme ning esineda võib välist fragmenteerumist

Mäluhaldus – Segmendid ja leheküljed

- Neid kahte meetodit võib ka koos kasutada
  - Motorola 68000 protsessorid kasutavad lamedat mälu mudelit
  - Inteli x86 ja Pentium põhinevad segmentidel
  - Mõlemad kasutavad lehekülgede ja segmentide segamudelit

Mäluhaldus – Segmendid ja leheküljed

- i386 protsessori seletus:
  - Üldised omadused
    - Maksimaalselt 16K segmenti protsessi kohta
    - Iga segment maksimaalselt 4 GB suurune
    - Lehekülje suurus on 4 KB
  - Põhimõte
    - Loogiline aadress on jagatud kaheks – esimeses kuni 8K privaatset segmenti; teises kuni 8K jagatud segmenti
    - Info esimese kohta hoitakse kohalikus deskriptorite tabelis (*Local Descriptor Table = LDT*); teine globaalses (*Global Descripton Table = GDT*)
    - Iga tabeli sissekanne on 8 baiti (info segmendi baasregistri ja pikkuse kohta)

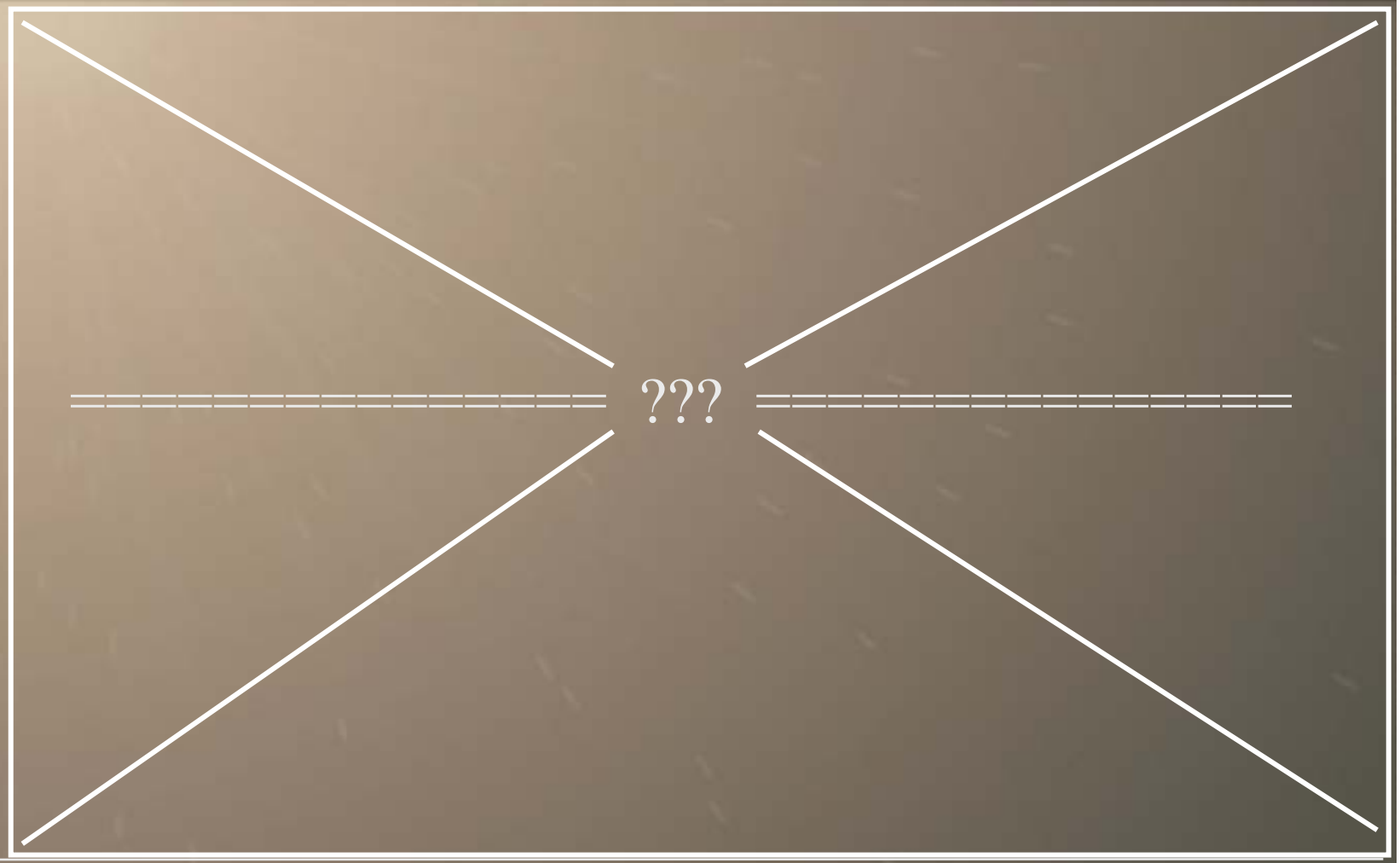
Mäluhaldus – Segmendid ja leheküljed

- i386 protsessori seletus:
  - Põhimõte
    - 6 segmendi registrit – st korraga saab adresseerida kuni 6 segmenti
    - Lisaks on 6 8-baidilist deskriptorite registrit
    - Füüsiline aadress on 32-bitine
      - Kõigepealt moodustatakse lineaarne aadress loogilises aadressis sisalduva viite abil deskriptorite tabelile ja nihkega
      - Lineaarne aadress koosneb 3 osast – kataloogist ja leheküljest (kokku annavad 2-tasemelise lehekülje) ning nihkest

Kordamisküsimused

- Mis vahe on loogilisel ja füüsilisel aadressil?
- Mis vahe on sisemisel ja välisel fragmenteerumisel?
- Kirjelda first-fit, best-fit ja worst-fit meetodeid
- Kui on mälu jaotused 100K, 500K, 200K, 300K ja 600K, siis kuidas need kolm meetodit paikutaks protsessid mälunõuetega 212K, 417K, 112K ja 426K?
- Miks on lehekülgede suurus alati 2 aste?
- Miks lehekülgede puhul protsess ei saa näppida teisele protsessile kuuluvat mälu?
- Mis vahe on lehekülgedel ja segmentidel? Mille poolest sarnased?

Küsimused?



Virtuaalmälu

- Virtuaalmälu lubab töötavatel protsessidel olla mälus vaid osaliselt – mingi osa on kuhugi välja tõstetud
- Miks võib osa olla välja tõstetud?
  - Programmid võivad sisaldada erandlike situatsioone töötlevat koodi, mida on vaja väga harva
  - Massiivid, nimekirjad, tabelid jms sarnased luuakse teatava varuga ning nad pole kuigi tihti ääreni täis
  - Osa programmi funkstionaalusest pole kogu aeg kasutusel
- Tänu virtuaalmälule saame kasutada programme mis nõuavad rohkem mälu kui reaalselt arvutis olemas on

Virtuaalmälu

- Virtuaalmälu eelised (kuna mälus vaid tähtsamad)
  - Programmi kasutust ei piira enam füüsiliselt olemasoleva mälu maht – programmil on oma virtuaalne aadressväli
  - Kuna iga programm võtab vähem ruumi, on meil võimalik rohkem programme samaaegselt tööle lasta
  - Vaja vähem S/V selleks et programme laadida / swappida – programmid jooksevad kiiremini
- Virtuaalmäluga süsteemidest on ülekatte (*overlay*) võimalus praktiliselt kadunud – ei oma enam mõtet

Virtuaalmälu - Demand paging

- Virtuaalmälu realiseeritakse tihti nõudmisel baseeruvate lehekülgede abil (*demand paging*). Sel juhul ennustatakse milliseid lehekülgi protsess vajama hakkab
- Lehekülgede asemel võivad olla ka segmendid
- Kuna nüüd on tegemist lehekülgede, mitte tervete protsesside vahetusega, siis kasutatakse *swapper*'i asemel terminit *pager*
- Virtuaalmälu kasutamine eeldab teatavat riistvaralist tuge. Võime ära kasutada *valid-invalid* bitte (valid = mälus; invalid = invalid või mälust väljas)

Virtuaalmälu - Demand paging

- Kui pöörduti lehekülje poole mis oli mälus, siis on kõik OK
- Lehekülgi, mis on kogu protsessi eluea jooksul mälus, nimetatakse residentseteks (*memory resident*)
- Kui lehekülge polnud mälus, tekib lehekülje viga (*page-fault trap*), mis püütakse OS-i poolt kinni ning lehekülge minnakse otsima virtuaalmälu hoidlast (näiteks failist kettal)
- Äärmusliku vormina on puhas nõudmisel põhinev (*pure demand paging*) meetod – ühtegi lehekülge ei laeta mällu kuni seda vaja pole

Virtuaalmälu – Demand paging

- Programmidel kipub olema viidete suhteline lokaalsus (*locality of reference*), mille tõttu osutub *demand paging* efektiivseks

- Riistvaralised nõudmised on samad:
  - Lehekülgede tabel
  - Sekundaarne mälu (*swap space; backing store*)

- Jõudlus – efektiivne mälupöördusaeg
  = (1 – *p*) * *ma* + *p* * page fault time,

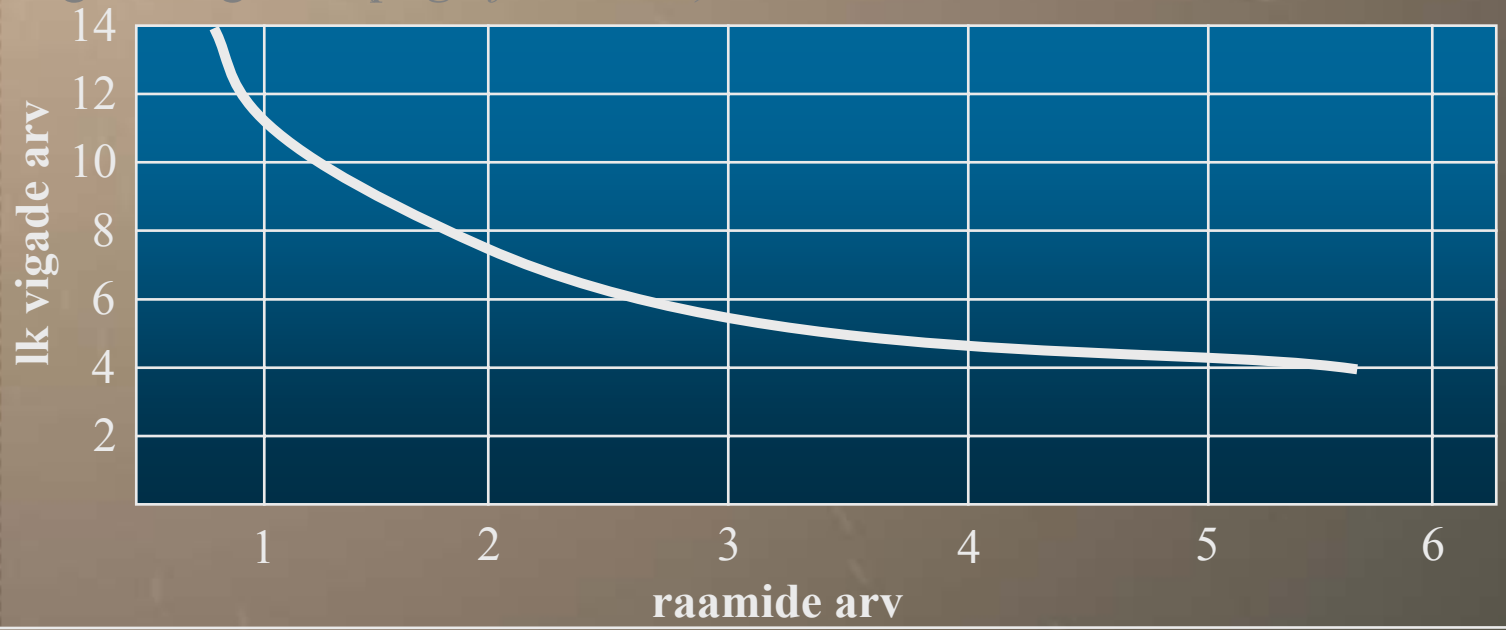
  kus *p* = vea tõenäosus; *ma* = memory access time
  Tihti tulemuseks *ma* = 100 ns; kettalt info saamine 25 ms. Viga põhjustab aeglustumise ~250 korda

Virtuaalmälu – Lehekülgede vahetus

- Nii või teisiti saabub aeg, kus tuleb mälus olevaid lehekülgi uute mahutamiseks välja tõstma hakata
- Lehekülgede vahetuse korral tuleb meil pöörduda sekundaarse mälu poole 2 korda
- Kiirendamiseks võime lisada igale leheküljele uue biti, mis näitab lk sisu muutumist mällu lugemise ajast (*modify bit*) – kui muutust pole, ei pea me seda lk-d kettale kirjutama – ta juba on seal

Virtuaalmälu – Lehekülgede vahetus

- Me peame *demand paging*'u puhul lahendama kaks ülesannet:
  - Looma raami-allokatsiooni algoritmi
  - Looma lehekülgede vahetuse algoritmi
- Millist vahetuse algoritmi kasutada? Üldiselt seda millel on madalam lk-vigade sagedus (*page-fault rate*)

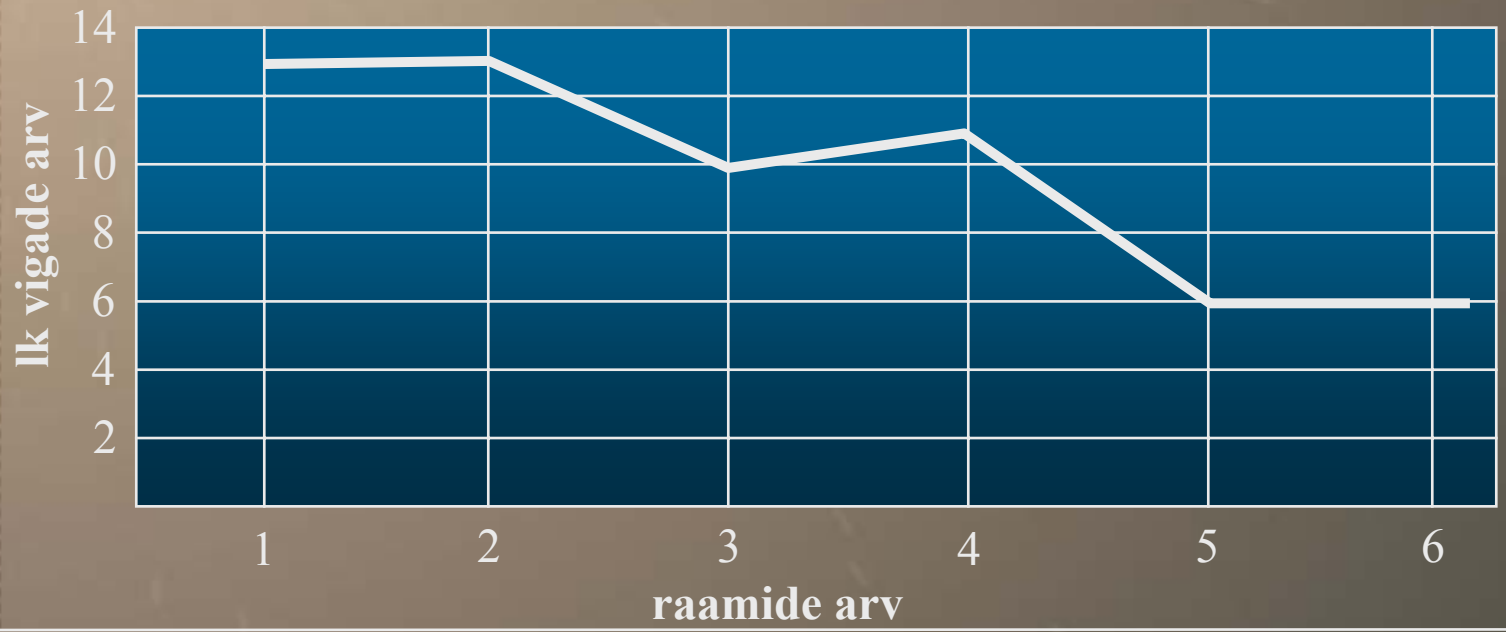


Virtuaalmälu – Lehekülgede vahetus

- FIFO (first-in first-out) lehekülgede vahetus
  - Iga leheküljega seotakse aeg millal ta esimest korda mällu loeti.
  - Täpne aeg pole oluline, oluline on järjekord
  - Välja visatakse kõige vanem
  - See meetod pole kõige efektiivsem – võib-olla oli tegemist initsialiseerimise funktsioonidega; võib-olla aga tihedalt kasutatavate muutujatega. Mõlemad võisid olla loodud üsna ammu
  - Kui me viskame välja tihedalt kasutatava variandi, tekkib üsna peatselt uus lehekülje-viga ning vana tuuakse tagasi mällu. Kõik muutub aeglasemaks

Virtuaalmälu – Lehekülgede vahetus

- FIFO (first-in first-out) lehekülgede vahetus
  - Bedlady' anomaalia – mõndade vahetusalgoritmide puhul võib vigade arv suureneda kui suureneb allokeeritud raamide arv



Virtuaalmälu – Lehekülgede vahetus

- Optimaalne vahetus
  - Välja tuleb visata lehekülg, mida kõige pikema perioodi vältel **ei kasutata**
  - Selline variant nõuab tuleviku teadmist
  - Kuna üldjuhul me ennustada ei suuda, siis kasutatakse seda meetodit vaid võrdluste läbi viimiseks – näiteks võib olla kasulik teada, et meie uus algoritm pole küll optimaalne, kuid halvimal juhul 12.3 % ja parimal 4.7 % sellest kehvem

Virtuaalmälu – Lehekülgede vahetus

- LRU (Least Recently Used) meetod
  - Kui me ei suuda tulevikku ennustada, võime vaadelda lähiminevikku kui selle approksimatsiooni – ehk
  - millist lehekülge kasutati kõige kauem aega tagasi
  - See on kõige levinum meetod
  - Teostus on keeruline:
    - Vajame loedureid – loendurit suurendatakse iga mälu poole möördumisel
    - Pinu – hoiame pinu viimati kasutatud lehekülje numbritega
  - Mõlemad meetodid vajavad riistvaralist tuge

Virtuaalmälu – Lehekülgede vahetus

- LRU approksimeerimise meetod
  - Vähesed süsteemid omavad sellist riistvaralist tuge nagu puhas LRU meetod vajaks
  - Võime lisada referentsi (*reference*) biti, mis ei näita ära järjekorda, kuid näitab millised on kasutatud ja millised mitte
  - Võime kasutada näiteks lisabitte (*additional-reference-bits*)
    - Näiteks 1 bait järjekorra hoidmiseks
    - Mingi aja tagant OS nihutab neid bitte 1 võrra paremale visates ära kõige pisema
    - Siin pole unikaalsus garanteeritud – mitmel lk-l võib olla sama väärtus

Virtuaalmälu – Lehekülgede vahetus

- LRU approksimeerimise meetod
  - Teise võimaluse (*Second-Chance*) meetod
    - Kui referentsi bitt = 0, siis visatakse leht välja
    - Kui bitt = 1, siis antakse talle teine võimalus – ta reference bitt nullitakse ja saabumise ajaks pannakse käesoleva hetke aeg
  - Laiendatud *Second-Chance* meetod
    - Võime vaadelda referentsi ja modifitseerimise bitte kui järjestatud paare
      - (0,0) – pole hiljuti kasutatud ega muudetud – parim valik
      - (0,1) – muudetud – peame lehe kettale kirjutama
      - (1,0) – kasutati hiljuti – tõenäoliselt varsti jälle
      - (1,1) – kasutati ja on muudetud

Virtuaalmälu – Lehekülgede vahetus

- Loendamisel põhinev (*counting-based*) meetod
  - Võime luua kaks uut loendamisel põhinevat moodust:
    - LFU (Least Frequently Used)
      Väikseima väärtusega leht asendatakse. Häda siis kui alguses kasutatakse tihedalt, hiljem aga vähe
    - MFU (Most Frequently Used)
      Siin tähendab väikseim väärtus, et lehte on hiljuti kasutatud või toodi just mällu

Virtuaalmälu – Lehekülgede vahetus

- Lehekülgede buhverdamise meetod
  - Kasutusel on pisike lehekülgede ”mahuti”
  - Uus lehekülg loetakse kõigepealt sinna ning vahetatakse siis mõnega välja
  - Väljavisatu pannakse sellesse vaheruumi ootama kuni ta kas uuesti tagasi loetakse või kettale kirjutatakse

Virtuaalmälu – Raamide allokeerimine

- Kuidas jagada fikseeritud mahuga füüsiliset mälu protsesside vahel?
- Kui meil on tegemist ühe-kasutaja süsteemiga, siis osa mälust läheb OS-i alla, ülejäänu jääb kasutaja programmide käsutusse
- Me võime kasutada meetodit kus algselt pannakse kõik kasutajale määratud raamid vabade nimekirja ning hakatakse siis täitma vastavalt saadavatele vigadele

Virtuaalmälu – Raamide allokeerimine

- Minimaalne raamide arv
  - Kui allokeeritud raamide arv väheneb, suureneb lk vea saamise tõenäosus
  - Kui lk-viga tekib enne käsu töö lõppemist, tuleb kogu käsk uuesti käivitada
  - Käsus viidatavad operandid võivad olla kaudsed – viidata teistele mäluaadressidele

Virtuaalmälu – Raamide allokeerimine

- Minimaalne raamide arv
  - Osad käsud (sõltuvalt CPU arhitektuurist) võivad olla pikemad kui 1 bait – seega potensiaalselt ületades raami piire
  - Kõige hullemad on kaudsed viited andmetele – äärmuslikul juhul viitavad nad läbi kõigi raamide
  - Kaudsete viidete hädadest aitab üle maksimaalne kaudsuse määr (näiteks 16 viitamist)
  - Minimaalne raamide arv on määratud arvuti arhitektuuriga; maksimaalne vaba mäluga

Virtuaalmälu – Raamide allokeerimine

- Allokeerimise algoritmid
  - Võrdne jaotus (*equal allocation*)
    - Kõige lihtsam *n* raami jagamine *m* protsessi vahel on anda igale võrdne osa *m/n*.
    - Ülejäägi võib panna vabade raamide buhvrisse
  - Proportsionaalne (*proportional*) jaotus
    - Tihti pole mõttekas mälu võrdselt jagada (näiteks 127 KB andmebaas ja 10 KB tekstiredaktor)
    - Jaotatakse protsessi suuruse järgi
  - Algoritmi valik sõltub multi-programmeerimise tasemest
    - Me võime näiteks anda rohkem mälu kõrgema prioriteediga protsessile, et tema tegevust kiirendada

Virtuaalmälu – Raamide allokeerimine

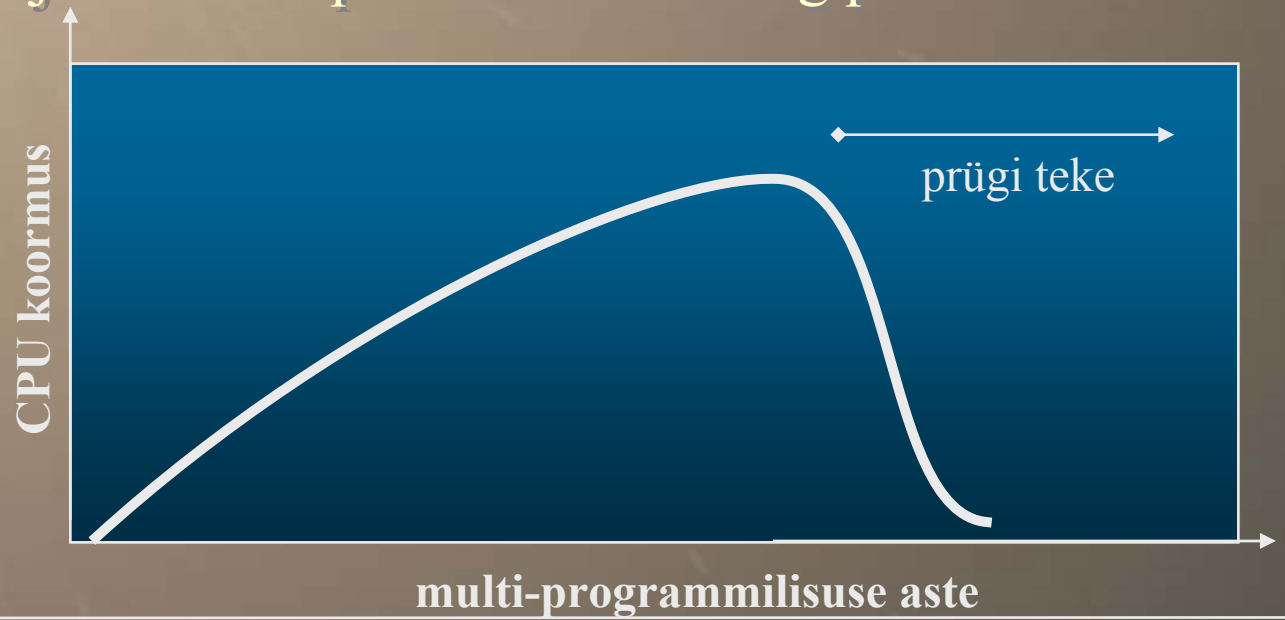
- Globaalne vs. Lokaalne allokeerimine
  - Globaalse allokeerimise puhul on lubatud valida uus raam kõigi vabade hulgast
    Sel puhul allokeeritud raamide arv võib muutuda
  - Lokaalse puhul aga ainult enda alla kuuluvate raamide hulgast
    Siin on raamide arv alati sama
- Võime lasta kõrgema prioriteediga protsessidel valida nii enda kui alamate hulgast (loomulikult alamate arvelt)
- Globaalse puhul ei sõltu lk-vigade arv protsessist endast vaid OS-ist ja teistest protsessidest

Virtuaalmälu – Prügi

- Kui madala prioriteediga protsessi raamide arv langeb alla tööks vajaliku, siis tuleb protsess seisata
- Seiskamisega koos tuleb meil välja visata ka kõik teised tema poolt kasutatavad raamid
- Kui tekib situatsioon, kus protsessi suhtes aktiivseid raame on rohkem kui mällu mahub, peab hakkama neid pidevalt vahetama – seda nimetatakse prügi tekkimiseks (*trashing*)
- Teiste sõnadega tegeleb protsess rohkem lehekülgede vahetamisega kui millegi kasulikuga

Virtuaalmälu – Prügi

- OS tõstab protsesside arvu kui CPU vähe koormatud on
- Ühest hetkest alates hakkab tekkima prügi
- Prügi vähendamiseks võib kasutada lokaalset allokeerimist
- Ka sel juhul teiste protsesside ooteaeg pikeneb



Virtuaalmälu – Prügi

- Vältimiseks tuleb hakata analüüsima protsessi lokaalsuse mudelit (*locality model*) – programm liigub ühest lokaalsusest teise (näiteks peaprogrammist alamprogrammi ja tagasi)

- Kui allokeerida vähem raame kui on lokaalsuse aste, hakkab tekkima “prügi”

Virtuaalmälu – Prügi

- Tööhulga mudel (*working-set model*)
  - Defineerib Δ kui tööhulga akna
  - Idee seisneb viimaste Δ lehtede poole pöördumiste vaatlusel
  - Viimase pöördumisega seotud Δ lehtede hulk on nn. tööhulk (*working set*)
  - Kui mingi aja jooksul aktiivse lehe poole ei pöörduta, eemaldatakse see tööhulgast
  - Kõige olulisem on selle hulga suurus
  - Kogu nõutud raamide arv D saadakse (WSS = Working Set Size)

$$D = \sum_i WSS_i$$

Virtuaalmälu – Prügi

- Leheküljevigade sagedus (page-fault frequency = PFF)
  - Tööhulga meetod on edukas kuid keeruline realiseerida
  - Lihtsam lähenemine on leheküljevigade sageduse arvestamine
  - Idee on lihtne – kui PFF sagedus suureks läheb, vajab protsess rohkem ruumi
  - Võib defineerida kaks piiri
    - Ülemine piir, millest suurem PFF viitab rohkem vajatavatele raamidele
    - Alumine piir, millest väiksema PFF puhul võib raame teistele protsessidele anda

Virtuaalmälu – OpSüsteemid

- Windows NT
  - Kasutab leheküljenõuete klasterdamist – mällu ei tooda mitte ainult vajaminev vaid ka selle lähemad naabrid
  - Igale protsessile määratakse ka tööhulga min ja max – kui on ruumi võib protsessile anda kuni max raami
  - Kasutusel on automaatne tööhulga suuruse muutmine – vajadusel vähendades kuni protsessi min väärtuseni

Virtuaalmälu – OpSüsteemid

- Solaris 2
  - OS tuum kontrollib 4 x sekundis vaba mälu hulka
  - Kui väärtus on alla *minfree*, siis hakatakse lehti välja viskama (sarnane *second-chance* meetodile)
  - Väljaviskamine jätkub kuni vaba mälu hulk ületab *lotsfree* väärtuse

Virtuaalmälu – Muud tähelepanekud

- Eelnev lehtede valik (*prepaging*)
  - Juhul kui protsess tööd alustab, siis tekib palju lk-vigu
  - Strateegiline käik oleks tuua mällu kõik vajaminevad leheküljed
  - Vajadusel võime ära kasutada pikemaid I/O operatsioone lisalehtede/raamide mällu toomiseks (seda eriti tööhulga mudeli puhul)

Virtuaalmälu – Muud tähelepanekud

- Lehekülgede suurus
  - Lehekülje suurus on alati 2 aste ning on enamjaolt vahemikus 2^{12} (4’096) kuni 2^{22} (4’194’304)
  - Üheks määrajaks on lehekülje tabeli suurus (4MB mälu puhul oleks 1024 B juures 4096 lehekülge, 8192 juures aga ainult 512
  - Samas on pisemate suuruste puhul mälutäituvus efektiivsem
  - Määravaks on ka lehe lugemisele/kirjutamisele kuluv aeg

Virtuaalmälu – Muud tähelepanekud

- Pööratud lehekülgede tabel
  - Sellest oli juttu eespool ning ideeks oli virtuaalse ja füüsilise aadressi vahelise sideme loomine
  - Seda sorti tabel ei sisalda enam täielikku infot kõikide virtuaalsete aadresside kohta
  - Kui seda meetodit kasutada, peame kusagil ikkagi hoidma lehekülgede tabelit (üks protsessi kohta)
  - Selliseid väliseid lisatabeleid kasutatakse vaid lk-vea saamise korral

Virtuaalmälu – Muud tähelepanekud

- Programmi struktuur
  - Oletame et meil on Java kood 128x128 maatriksi nullimiseks
  - Esimese juhul käiakse järjest läbi kõik leheküljed ja seda 128 korda – kokku 128x128 = **16'384 lk-viga !!!**

```
int A[][] = new int[128][128];

for (int j = 0; j < 128; j++)
  for (int i = 0; i < 128; i++)
    A[i][j] = 0;
```

  - Peale parandust saame vaid 128 lk-viga

```
int A[][] = new int[128][128];

for (int i = 0; i < 128; i++)
  for (int j = 0; j < 128; j++)
    A[i][j] = 0;
```

Virtuaalmälu – Muud tähelepanekud

- Sisendi/väljundi vastastikune lukustumine
  - Meil on vahest vaja lukustada mingit mäluosa kuhu kirjutab/loeb eraldi protsessor
  - Tuleb tagada, et I/O operatsiooni palunud protsessi mäluosa, kust/kuhu andmeid siirdama hakatakse, vahepeal välja ei tõsteta
  - Appi võib võtta *locked* biti
  - Luku bitt võib ka ohtlik olla – tema võib peale keerata aga maha enam kunagi ei võeta

Virtuaalmälu – Muud tähelepanekud

- Tööd reaalajas
  - Siiani oleme püüdnud maksimeerida kogu süsteemi efektiivsust; üksikud protsessid võivad samas aga palju kannatada
  - Tavaliselt reaalajaga tegelevates süsteemides virtuaalmälu üldse ei kasutata

Kordamisküsimused

- Millistel tingimustel leheküljevead tekivad?
- Mis on lk-vigade alumine ja ülemine piir?
- Millised programmi struktuuridest on virtuaalmälu suhtes head ja millised halvad= Miks?
  - Pinu
  - Järjestikuline otsimine
  - Kahendotsimine
  - Puhas programmikood
  - Vektorite operatsioonid
  - Kaudsed viited andmetele

Küsimused?