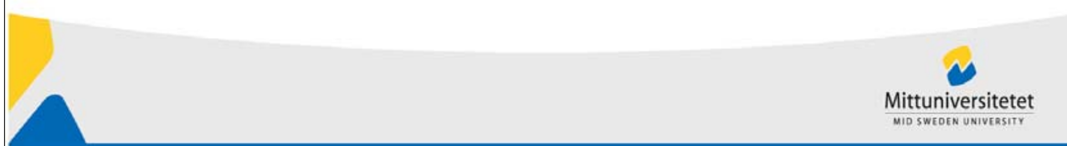
# Virtuellt Minne




**Kapitel 8 (sid. 333-381)**

Nu ska vi komplettera det vi redan lärt oss, om paging och segmentering, med virtuellt minne. Virtuellt ur den synvinkeln att vi tillåter processer att ”använda” mer minne än vad som finns installerat. Ni kanske redan nu förstår hur det går till?

**Hardware and control structures**

*Bild 2:* Paging och segmentation tillsammans med swappning ger oss möjlighet att dölja hur mycket minne vi egentligen har installerat. Den dynamiska översättningen mellan logisk och fysisk adress är en förutsättning för swapping.

En process kan delas upp i mindre bitar och dessa bitar inte behöver ligga kontinuerligt i minnet ger förutsättningen för virtuellt minne. Men viktigast: alla delar behöver inte vara laddade i primärminnet samtidigt.

# Exekvering Av Program

- Operativsystemet hämtar in bara de delar som behövs
- *Resident set* – den del av processen som för ögonblicket finns i primärminnet
- Ett interrupt genereras när en sida som inte finns i minnet behövs
- Operativsystemet placerar processen i *blocking state*

Mittuniversitetet
MID SWEDEN UNIVERSITY

**Exekvering av program vid virtuellt minne**

*Bild 3-4:* OS behöver bara, till primärminnet, hämta in de delar som behövs närmast.

*Resident Set* är den del av processen som för ögonblicket är inläst. När processen refererar till en sida som inte just nu finns inläst i minnet genereras ett interrupt, ett *Page fault*.

Den önskade sidan behöver läsas in från sekundärminnet. Det är ju en I/O-anrop så processens tillstånd blir *blocked* och ett context switch sker (en annan process kan få exekvera). OS begär en *disk I/O read* som lämnar ett interrupt när det är klart. OS kan då göra den blockerade processen i Ready igen.

# Exekvering Av Program

- Den del av processen som innehåller den eftersökta logiska adressen hämtas
  - Operativsystemet skickar en *disk I/O Read request*
  - En annan process kan exekvera under tiden
  - Ett interrupt kommer när läsningen är klar och operativsystemet sätter den blockerade processen i *Ready state*

Mittuniversitetet
MID SWEDEN UNIVERSITY

# Fördelar Med Att Dela Upp En Process

- Fler processer kan hanteras i primärminnet samtidigt
  - Endast en del av processen laddas
  - Med så många samtidiga processer ökar chansen att någon ska vara klar för exekvering
- En enskild process kan vara större än allt tillgängligt totalt minne

*Bild 5:* känner vi ju igen som fördelarna med paging/segmentering, men viktigaste slutsatsen: en enskild process (t o m varje) kan vara större än allt tillgängligt minne.

# Typer Av Minne

- Fysiskt minne
  - Primärminne
- Virtuellt minne
  - Minne på disk
  - Möjliggör effektiv multiprogrammering och befriar användaren från att hantering av primärminnet

**Typer av minne**

*Bild 6:* vi skiljer mellan:

- • Fysiskt minne (*Real Memory*), det primärminne som är installerat. Det är ju bara här som processen exekveras.
- • Virtuellt minne, hela tillgängliga adressområdet.

*Table 8.1* boken sida 336 är viktig. Se speciellt skillnaden mellan *Simple Paging/Segmentation* och *Virtual Memory Paging/Segmentation*.

## *Thrashing*

- Swappar ut en del av en process alldeles innan det behövs igen
- Processorn använder mest av tiden till att swappa fram och tillbaka istället för att exekvera instruktioner

**Locality and Virtual Momory**

*Bild 7:* pekar på ett möjligt problem *Trashing* (egentligen beroende på swapping).

Vid inläsning av en refererad sida, så kastar OS ut en annan sida. Den utkastade sidan kanske refereras i nästa instruktion och måste då läsas in igen. Detta kan upprepas hela tiden och systemet blir upptagen med att bara flytta samma sidor fram och tillbaka. OS måste försöka gissa vilka sidor som kan behövas snart igen.

Denna gissning baseras på *lokalitetsprincipen*, som vi ju redan har tagit upp detta i lektion 1.

Se även på *Figure 8.1* sidan 337. Man ser tydligt hur vissa sidor refereras mycket under en viss tid, innan fokus flyttas till andra sidor.

# Lokalitetsprincipen

- Program och datareferenser tenderar att, i minnet, vara i närheten av varandra
- Under ett kort tidsintervall kommer bara få delar av en process att behövas
- Om man kan göra intelligenta gissningar om vilka delar som kommer att behövas sedan, så kommer det virtuella minnet att bli effektivt

Mittuniversitetet
MID SWEDEN UNIVERSITY

# Krävs För Virtuellt Minne

- Hårdvaran måste stödja *paging* och segmentering
- Operativsystemet måste hantera flyttningen av Pages/segment mellan primär och sekundärminne

*Bild 9:* För att virtuellt minne ska fungera effektivt krävs två saker. Hårdvarustöd för paging/segmentering och mjukvara som hanterar swapping.

# *Paging*

- Varje process har en egen sidtabell
- Varje post i sidtabellen innehåller bindningen mellan ett frame-nummer och motsvarande sida i primärminnet
- En flagga behövs för att visa om sidan finns i primärminnet eller är utswappad

**Paging**

Vi börjar med att titta på ren *Paging* (utan segmentering).

*Bild 10:* För att paging ska fungera krävs att det skapas en sidtabell för varje process. Denna tabell lägger vi i primärminnet eftersom vi ska läsa i den hela tiden.

# *Modify Bit* i Sidtabellen

- En ändrad-flagga visar om innehållet på sidan har ändrats sedan den laddades i primärminnet
- Om sidan inte har ändrats så behöver den inte skrivas till disk när den swappas ut

*Bild 11:* Vi kompletterar tabellens entryn (rader) med två bitar. Den ena visar om sidan finns inläst

(P-resent) och den andra om något har ändrats i sidan, dvs. skrivits (M-odify). Om M inte är satt så kan ju sidan kastas ut utan att behöva uppdateras i primärminnet (vanlig cachehantering).

Det kan även finns fler bitar som hanterar ex vis dataskydd och delning.

# Sidtabellposter

Virtual Address

Page Table Entry

**(a) Paging only**

**Figure 8.2 Typical Memory Management Formats**

*Bild 12:* visar hur den virtuella adressen ser ut och en rad i tabellen. I tabellen går jag in utifrån sidnummer och läser i rätt entry vilken frame sidan ligger i.

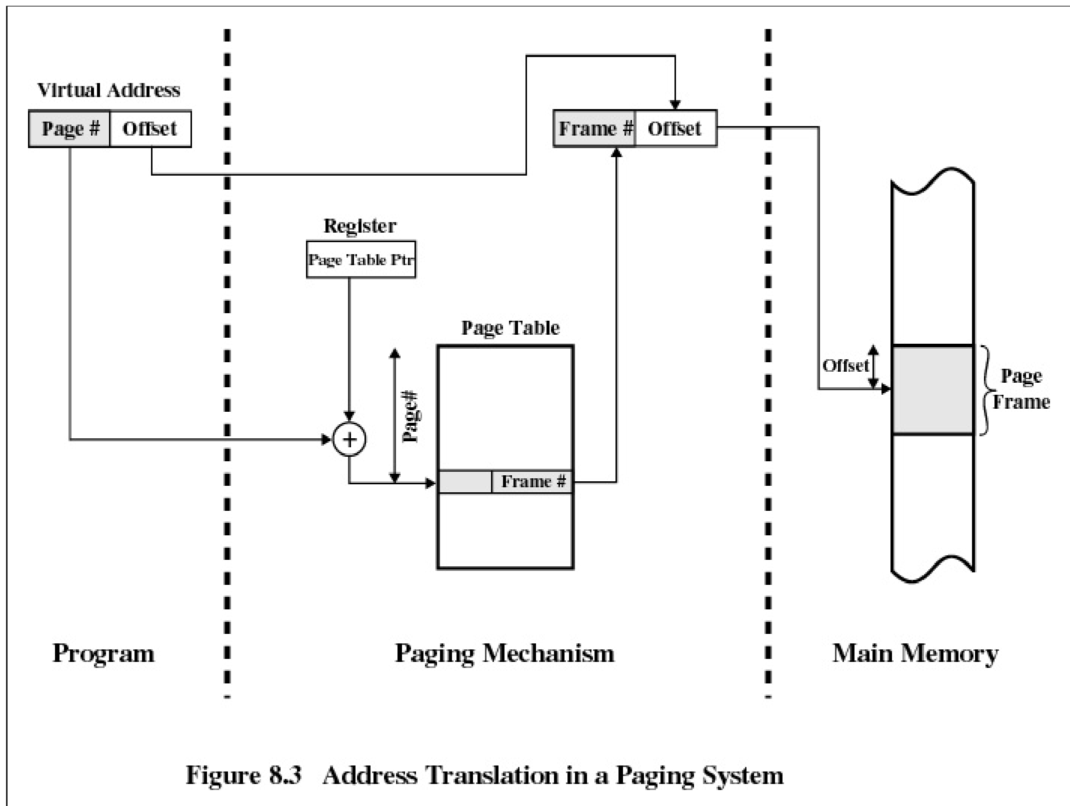


**Figure 8.3 Address Translation in a Paging System**

## Page Table Structure

Varje minnesreferens, som tex. läsning av ett ord från minnet, involverar översättning av logisk adress till fysisk adress m h a sidtabellen.

*Bild 13:* visar detta. Ett speciellt register innehåller en pekare till processens sidtabell. (Detta registers innehåll ska ju ändras vid ett Context switch.) Sidnumret (Page#) från den virtuella adressen adderas till sidtabellens startadress för att ge rätt entry i processens sidtabell. Där får vi rätt ramnummer (Frame#) som adderas till offset och ger rätt fysisk adress.

Se till att ni kan detta i sömnen.

# Schema För 32-bitars Adress, Två Nivåer

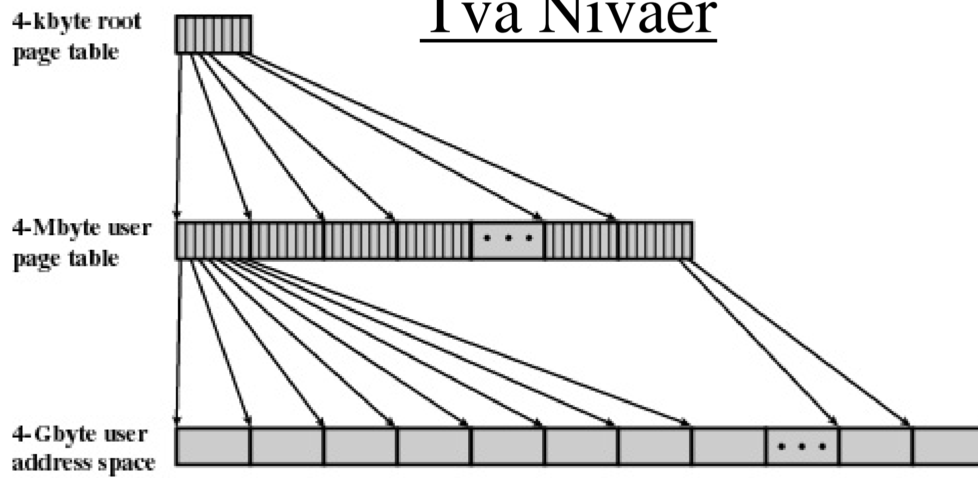



Figure 8.4 A Two-Level Hierarchical Page Table [JACO98a]

# Sidtabeller

- Hela sidtabellen blir så stor att den tar upp för mycket primärminne
- Sidtabeller sparas också virtuellt minne
- När en exekveras så är en del av dess sidtabell i primärminnet

*Bild 15:* (det ska stå När en *process* exekveras…) Sidtabellen kan ju bli väldigt stor. Om en virtuell process tillåts använda 32-bitars adresser och sidstorleken är 512-byte, så skulle det ge 2^22 rader i sidtabellen, för varje process! Det skulle ju inte gå att hålla allt detta i primärminnet, så därför hanteras sidtabellen också m h a virtuellt minne. Dvs. sidtabellen är också utsatt för paging/swapping. Minst en sida av sidtabellen måste i alla fall finnas i minnet när processen exekveras.

Vissa processorer hanterar också sidtabellen hierarkiskt.

Tänk på att det krävs en hårdvarulösning för att hantera adressöversättning, pekare till sidtabell och additioner. Så det är processorberoende.

Pentiumprocessornr använder en modell enligt *Bild 14:* Totalt adressområde med 32 bitar är 4-Gbyte uppdelat i sidor på 4 Kbyte. En massa sidor blir det, närmare bestämt 2^20 stycken. I exemplet med 4 byte/entry skulle det ge en sidtabell på 4 Mbyte. Genom att lösa det med en *Root page table* på 2^10=4 Kbyte (som alltid är inläst) och som pekar till en *User page table* på 2^10 sidor så blir sidtabellen hanterbar.

Översättningen logiska/fysisk blir bara lite mer komplicerad. *Figure 8.5* sidan 341 visar detta. Jag tror/hoppas att ni kan förstå det själva nu. Ni kan ju snart allt.

## *Translation Lookaside Buffer*

- Varje virtuell minnesreferens kan ge upphov till två referenser till fysiskt minne
  - En för att hämta sidtabellen
  - En för att hämta data
- Lösningen är en mycket snabb cache som innehåller sidtabellposter, *TLB - Translation Lookaside Buffer*

Mittuniversitetet
MID SWEDEN UNIVERSITY

**Translation Lookaside Buffer TLB**

*Bild 16-21:* Nu ska vi bara göra det ett snäpp till krångligare. Om vi refererar till en sida, så kanske den delen av sidtabellen först måste hämtas. Först då ser vi också att själva sidan inte heller är inläst. Detta ger upphov till fördubblad accesstid till en sida. Lösningen är en mycket snabb cache, speciellt för pageside-sidor, TLB (självklart va).

Jämför Figure 8.3 sidan 339 med Figure 8.7 sidan 343. Figure 8.8 sidan 343 sammanfattar vad som sker. OBS att det saknas page table register i Figure 8.7, borttaget för att förenkla bilden. Och att det inte används en hierarkisk sidtabell.

# TLB

- Innehåller de senast använda sidtabellposterna
- Fungerar som en vanlig minnescache

Mittuniversitetet
MID SWEDEN UNIVERSITY

# TLB

- Vid en referens till en virtuell adress letar processorn först i TLB
- Om posten finns där (*hit*), så tas sidnumret därifrån och den fysiska adressen beräknas
- Om inte posten fanns i TLB, så används sidnumret som vanligt för att indexera processens sidtabell

Mittuniversitetet
MID SWEDEN UNIVERSITY

# TLB

- Kontrollera först om sidan redan finns I primärminnet
  - Om inte genereras ett *page fault*
- TLB uppdateras så att den innehåller de senaste inlästa posterna

Mittuniversitetet
MID SWEDEN UNIVERSITY

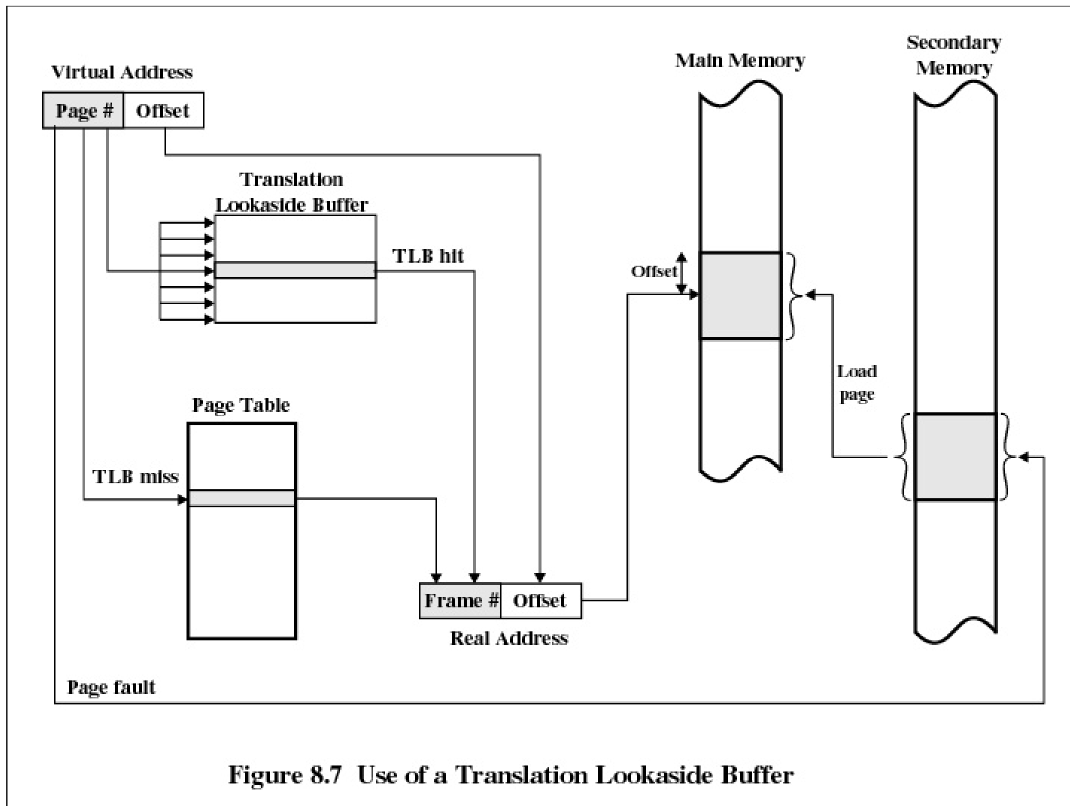



Figure 8.7 Use of a Translation Lookaside Buffer

# Sidstorlek

- Mindre sidstorlek ger mindre intern fragmentering
- Mindre sidor gör att fler behövs per process
- Fler sidor per process kräver större sidtabeller
- Större sidtabeller gör att större del av sidtabellen finns i virtuella minnet
- Sekundärminnet är optimerat för att föra över stora datablock så stora sidor är bättre

Mittuniversitetet
MID SWEDEN UNIVERSITY

**Page size**

Däremot är det intressant att fundera över hur sidstorleken påverkar effektiviteten.

*Bild 22-23:* sammanfattar detta. Kom ihåg att ett *page fault* innebär att en sida måste läsas in vilket kräver I/O-access. Vi vill ha så få page fault som möjligt, men inte till vilket pris som helst.

# Sidstorlek, men..

- Små sidor gör att många sidor kan finnas i primärminnet
- Efter en tids exekvering så kommer de flesta sidorna i minnet att innehålla nyligen refererat data. Vilket ger få page-fault
- Stora sidor kommer att innebära att sidorna innehåller mycket som är långt ifrån nyligen refererat data. Vilket ger många page-fault

Mittuniversitetet
MID SWEDEN UNIVERSITY

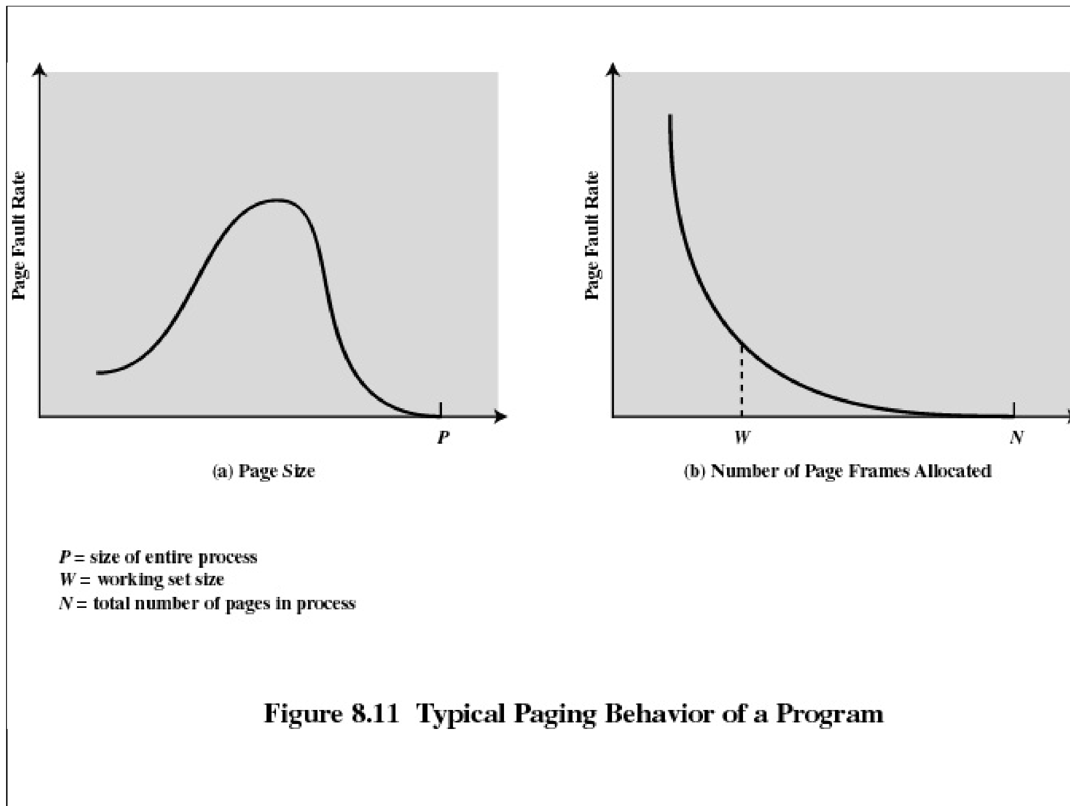


Figure 8.11 Typical Paging Behavior of a Program

*Bild 24:* (a) Visar relationen mellan sidstorlek och page fault rate. Vid P är sidan lika stor som processen. som är inlästa. W = Working set size (= hur många av processens sidor som är inlästa). När W=N så är alla det.

*Bild 24:* (b) Visar page fault rate i förhållande till antal sidor, av processens samtliga sidor N, inlästa.

Tittar vi på (a) så borde vi ha stora sidor, som rymmer hela processen, men då får vi intern fragmentering för mindre processer. (b) pekar på att alla processens sidor ska vara inlästa.

Som vanligt måste det bli en kompromiss. Slutsatsen blir att optimal sidstorlek beror på hur mycket fysiskt minne det finns och programmets storlek.

Modern objektorienterad programmering med många små moduler som samverkar, samt multitrådning leder till att lokalitetsprincipen blir mindre viktig. Detta eftersom programmet använder och refererar till många olika moduler under en kort tid. Detta gör det svårare att välja vilka sidor som ska sparas och vilka som kan kastas.

## Sidstorlek, vad göra

- Olika tillåtna sidstorlekar ger den flexibilitet som krävs för att amvända TLB effektivt
- Stora sidor kan användas för kod
- Små sidor för trådar
- De flesta operativsystem stödjer bara en sidstorlek. Undantag MIPS, UltraSPARC och Pentium

Mittuniversitetet
MID SWEDEN UNIVERSITY

*Bild 25:* En lösning på att välja rätt sidstorlek vore att ha olika storlek på sidorna beroende på programmets storlek. Denna lösning kan utnyttjas av vissa processorer (kom ihåg att paging är hårdvarubaserat). *Bild 26:* listar sidstorleken för vanliga och ovanliga processorer. Hur gör vi då med framestorleken? En stor sida får uppta en multipel av en liten frame.

# Exempel På sidstorlek

**Table 8.2 Example Page Sizes**

| Computer | Page Size |
|---|---|
| Atlas | 512 48-bit words |
| Honeywell-Multics | 1024 36-bit word |
| IBM 370/XA and 370/ESA | 4 Kbytes |
| VAX family | 512 bytes |
| IBM AS/400 | 512 bytes |
| DEC Alpha | 8 Kbytes |
| MIPS | 4 kbyes to 16 Mbytes |
| UltraSPARC | 8 Kbytes to 4 Mbytes |
| Pentium | 4 Kbytes or 4 Mbytes |
| PowerPc | 4 Kbytes |

# Segmentering

- Dynamisk storlek
- Förenklar hanteringen av datastrukturer som ökar i storlek
- Tillåter att program ändras och kompileras oberoende
- Möjliggör att processer delar data
- Möjliggör dataskydd

**Segmentation**, och virtuellt minne

Kommer ni ihåg vad segmentation var? Förra kapitlet. Programmet skrevs i moduler som kunde ha olika och dynamisk storlek (varierande över tiden).

*Bild 27:* och 1-4 sidan 348 i boken, sammanfattar fördelarna.

Precis som med paging så behöver varje process en egen segmentabell

# Segmenttabell

- Varje post i segmenttabellen innehåller referens till motsvarande segments startadress i minnet och längden på segmentet
- En flagga indikerar om segmentet är laddat i primärminnet
- En *modified*-flagga visar om segmentet har ändrats sedan den laddades

Mittuniversitetet
MID SWEDEN UNIVERSITY

# Segmenttabellposter

Virtual Address

Segment Table Entry

**(b) Segmentation only**

**Figure 8.2 Typical Memory Management Formats**

Mittuniversitetet
MID SWEDEN UNIVERSITY

Med virtuellt minne är den uppbyggd enligt *Bild 29*: den virtuella adressens segmentnummer pekar till ett *segment table entry* som ger segmentets startadress och längd. Precis som med paging så behövs P och M flaggor, men även en del andra flaggor för skydd och delningsfunktioner.

Sidan 349 i boken visar hur översättningen mellan virtuell och fysisk adress sker.

# Kombinerad *Paging* och Segmentering

- Paging är transparent för programmeraren
- Paging eliminerar extern fragmentering
- *Paging Segmentation* hanteras av programmeraren
- Segmentering möjliggör växande datastrukturer, modularitet och stödjer delning av data och dataskydd
- Varje segment delas upp i sidor med fast storlek

Mittuniversitetet
MID SWEDEN UNIVERSITY

**Combined Paging and Segmentation**

Ska vi sätta ihop allt nu då?

*Bild 30:* Paging som eliminerar extern fragmentering och ger effektivt utnyttjande av minnet och segmentation som kan klarar av dynamiska moduler och kan hantera delning och dataskydd kan kombineras. Paging hanteras av hårdvaran och segmenteringen av OS.

Adressområdet är, av programmeraren, uppdelat i mindre segment. Dessa är i sin tur uppdelade i sidor av OS.

# Kombinerad *Paging* och Segmentering

Virtual Address

Segment Table Entry

Page Table Entry

P= present bit
M = Modified bit

**(c) Combined segmentation and paging**

**Figure 8.2 Typical Memory Management Formats**

*Bild 31*: visar att vi nu måste ha både en segmenttabell och en sidtabell per segment. Översättningen från virtuell till fysisk adress sker nu i två steg.

Figure 8.13 sidan 350 visar detta.

- • *Segment table pointer* innehåller adressen till aktuell process segmenttabell.
- • Denna adderas till segmentnummer och ger en pekare (adressen) till segmentets sidtabell
- • Denna tillsammans med sidnummret ger den frame som sidan just nu är inläst i (eller ev. ett page fault, om den var utswappad)
- • Tillsammans med offset fås den fysiska adressen som offset inom en frame. Var framen börjar vet OS, som ju har delat upp minnet i frames.

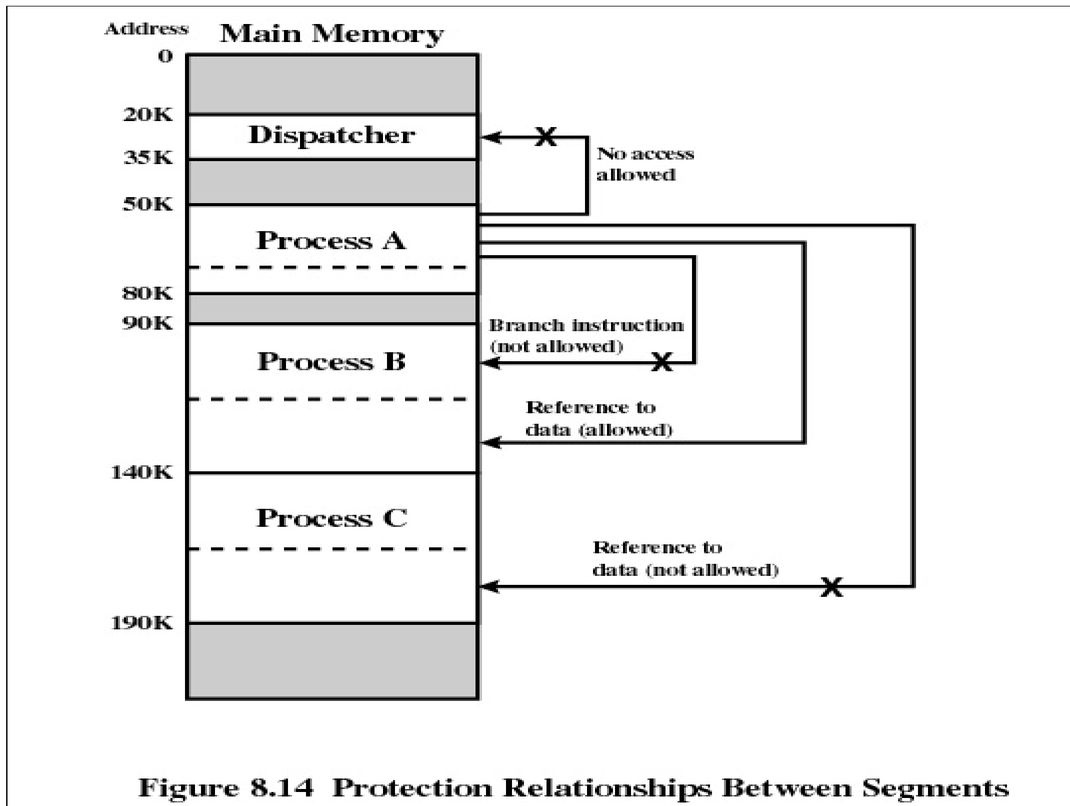


**Figure 8.14 Protection Relationships Between Segments**

## Protection and Sharing

*Bild 32:* visar hur segmenteringen stöder både dataskydd och delning. Segmenttabellen innehåller ju startadress och längd på varje segment så det är lätt att förbjuda eller tillåta access till ett segment.

När det här kapitlet är solklart för er kan ni behöva ta en fika.

# Fetch Policy

- Vilka sidor ska hämtas
  - Bestämmer när en sida ska hämtas in
  - *Demand paging* - endast refererade sidor hämtas
    - Många page-fault första gången processen körs
  - *Prepaging* - hämtar flera sidor än det behövs
    - Mera effektivt att hämta flera följande sidor från disken

Mittuniversitetet
MID SWEDEN UNIVERSITY

**8.2 Operating System Software**

Hoppas att kaffet var starkt. Andra uppiggande droger är fusk.

Om ni tänker skriva ett eget OS så finns det många saker ni måste fundera över ex vis:

- Ska jag använda virtuellt minne?
- Ska jag tillåta paging, segmentering eller båda?
- Hur ska jag skriva algoritmerna som ska hantera swapping, Fetch, Placement, Replacement osv? Se Table 8.3 sidan 353.

Virtuellt minne verkar bra att använda, de flesta gör det i alla fall. De flesta använder också ren paging så vi väljer väl det. Då är det bara algoritmerna kvar. Vi tar dom i tur och ordning.

- **Fetch policy** *Bild 33:* dvs. när ska vi läsa in sidor till minnet. Två varianter:

  - o *Demand paging*, läs in vid referens. Om bara en sida läses in så kommer det att bli massor med page fault vid start av en process.
  - o *Prepaging*, läser in mer än vad som absolut behövs. Passar disk som ju har sidorna liggande i en följd och i alla fall läser större bitar än en sida. Nackdelen är att dessa extra sidor kanske inte alls ska användas.

- **Placement Policy**: dvs. var, fysiskt i minnet, vi ska placera de inlästa sidorna.

Är inte så intressant eftersom vi valde att arbeta med paging. Sidorna kan ju flytta var som helst och denna relocation hanteras av pagingsystemet.

## *Replacement Policy*

- Vilka sidor ska kastas ut
  - Någon sida måste bytas ut
  - Den sida som kastas bör ha minst sannolikhet att refereras snart igen
  - Försöker förutsäga framtiden genom att titta på historien

• **Replacement Policy** *Bild 34:* dvs. vilken sida ska vi byta ut. (Är däremot viktig)

Vi måste börja med att bestämma:

o o Hur många frames varje process kan få ha samtidigt

o o Om sidan som ska kastas måste vara en av processens ”egna” eller om alla sidor kan sitta löst

o o Hur vi väljer vilken sida det blir

## *Replacement Policy*

- Sidlåsning
  - Om en sida är låst så kan den inte bytas ut exempel
    - Operativsystemets Kernel
    - Kontrollstrukturer
    - I/O-buffertar
  - En *lock*-flagga per frame

Det finns en mängd olika sätt att lösa detta. Men en sak måste vi börja med att hantera, sidlåsning. Se *Bild 35:*

## *Basic Replacement Algorithms*

- Optimal policy
  - Byt ut den sida som det kommer att dröja längst tid tills den refereras igen
  - Omöjligt att veta exakt vad som händer i framtiden

Det finns sedan fyra olika *Basic Replacement Policy* dvs. huvudprinciper för hur det ska göras. *Bild 36-39:* beskriver dessa.

- o *Optimal*
- o *LRU*
- o *FIFO*
- o *Clock*

*Optimal Policy* beskriver vad vi önskar att vi kunde göra, men den är inte möjlig att implementera. Vi kan ju inte titta fram i framtiden och se vilka sidor som vi kommer att behöva då. Fungerar som en perfekt referensmodell. De tre andra försöker komma så nära Optimal som möjligt.

Nu börjar det bli riktigt krångligt, igen. Gå inte djupare än att ni förstår OH-bilderna och Figure 8.15 sida 356 i boken.

<u>Lite kursivt fram till Page buffering.</u>

# *Basic Replacement Algorithms*

- Least Recently Used (LRU)
  - Byt ut den sida som det var längst sedan den refererades
  - Enligt lokalitetsprincipen kan det vara den sida som som minst troligt behövs snart igen
  - Varje sida skulle kunna tidsstämplas när senast refererad, men det skulle ge mycket overhead

Mittuniversitetet
MID SWEDEN UNIVERSITY

## *Basic Replacement Algorithms*

- *First-in, first-out (FIFO)*
  - Behandla en process allokerade sidor som om de vore i en cirkulär buffert
  - Sidorna byts ut som i round-robin
  - Enklast att implementera
  - Den sida som varit längst i minnet byts ut, men kan behövas snart igen

Mittuniversitetet
MID SWEDEN UNIVERSITY

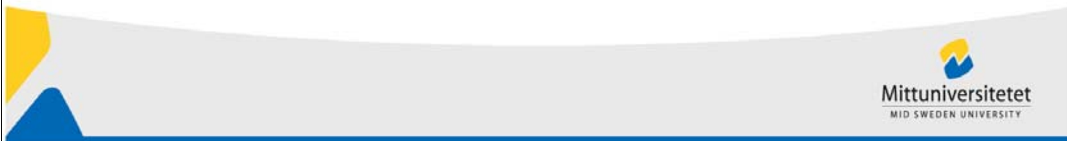
# *Basic Replacement Algorithms*

- *Clock Policy*
  - En flagga/sida *use bit*
  - När en sida laddas nollställs flaggan
  - När sidan refereras sätts flaggan
  - När en sida ska bytas ut tas den första som har sin flagga nollställd
  - Under sökningen nollställs alla sidornas flagga, dvs. om inte refererad till nästa gång så risk at bytas ut

Mittuniversitetet
MID SWEDEN UNIVERSITY

# *Basic Replacement Algorithms*

- Sidbuffring, större kostnad att byta ut en sida som har ändrats, måste sparas
- Utbytt sida läggs till en av två listor
  - *free page list* innehåller dom som inte ändrats
  - Annars i *modified page list*




**Page Buffering**

*Bild 42:* LRU och Clock fungerar ju bättre än FIFO, men är mer komplexa och ger overhead. Dessutom är det viktigt att tänka på att en sida som är förändrad ju måste skrivas tillbaka, inte bara kastas. Det är därför ”dyrare” (tar tid) att byta ut en ändrad sida än en som bara lästs.

Genom att inte omedelbart skriva tillbaka en ändrad sida utan lägga den i en cache-lista (modified page list) kan återskrivningen ske klustervis vilket passar disken. Detta blir då snabbare.

Djupare går vi inte och det som står i boken är helvirrigt (nästan som om jag hade sagt det).

## *Resident Set Size*

- Hur stor del av en process sidor ska hämtas
- Fast allokering
  - Ger en process ett fast antal sidor att exekvera i
  - Vid page-fault måste en av processens sidor bytas
- Dynamisk allokering
  - Antalet allokerade sidor varierar under en process livstid

**Resident Set Management**

*Bild 43: Resident Set Size* dvs. hur många sidor en process har inläst, är det också viktigt att fundera över innan vi skriver vårt OS. Hur många sidor dvs. hur mycket minne ska en process få ha? Färre sidor för varje process gör att flera processer får plats, men ökar risken för page fault.

Två varianter:

• • *Fixed-allocation*, vid processens skapande bestäms minnesalokeringen utifrån typ av process. Vid ett page fault måste en av processens egna sidor bytas ut.

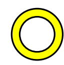
• • *Variable-allocation* (dynamisk allokering) OS tilldelar processen ett antal från början. Beroende på antal page fault så kan antalet sidor ökas, eller minskas. Detta ger dock overhead (extra arbete både för OS och hårdvara).

# Dynamisk allokering, Global Scope

- Enklast att implementera och vanligast
- Operativsystemet har en lista med fria sidor
- Fria sidor adderas till processens *reident set* vid page-fault
- Om det inte finns fria sidor så tas en från en annan process

Sedan var det ju frågan om vi skulle byta ut en av processens egna sidor eller sno av någon annan. Då kommer detta med *scope* in. Vi kan ha *Global scope* (alla sidor) eller *Local scope* (bara processens egna). *Bild 44-45:* och Table 8.4 sidan 364 som visar de olika varianterna.

Sedan tar vi lite kursivt igen fram till sidan 368



# Dynamisk allokering, Local Scope

- När en ny process laddas allokeras ett antal sidor baserat på typ av applikation, vad programmet begär och andra kriterier
- Vid page-fault byt ut en av processens egna sidor
- Utvärdera och ändra allokeringen under tiden

Mittuniversitetet
MID SWEDEN UNIVERSITY

# *Cleaning Policy*

- *Demand cleaning*
  - En sida kastas bara när platsen behövs för annan sida
- *Precleaning*
  - Sidor som ändrats och alltså ska skrivas tillbaka till disk buntas ihop och kastas flera i taget

Mittuniversitetet
MID SWEDEN UNIVERSITY

## *Cleaning Policy*

- Effektivast är sidbuffring
  - Utbytta sidor placeras I en av två listor
    - Modified/unmodified
  - Listan med sidorna som har ändrats skrivs med jämna mellanrum tillbaka
  - Sidorna som inte ändrats kommer antingen att refereras senare eller kastas när annan sida behöver använda platsen i minnet

**Cleaning Policy**

En viktig sak är kvar *Cleaning Policy* (som jag inte har någon bra översättning på). Det handlar inte om den upprensning av allokerat minne som sker i ett program, utan om hur och när vi ska skriva tillbaka ändrade sidor till disken.

*Bild: 46:* Två varianter även här:

• • Demand cleaning, *Bild 47:* sidan skrivs tillbaka när en ny sida ska in i framen.

• • Precleaning, *Bild 48:* ändrade sidor skrivs tillbaka klustervis för att passa disken, även om framen inte behövs. Sidan ligger kvar i minnet och kan bli ändrad igen.

Som vanligt så har båda för och nackdelar.

Precleaning var ju effektivt för disken, men alla sidorna ligger ju kvar och kan ändras igen. Alltså var det bortkastat arbete, speciellt som diskaccess tar tid.

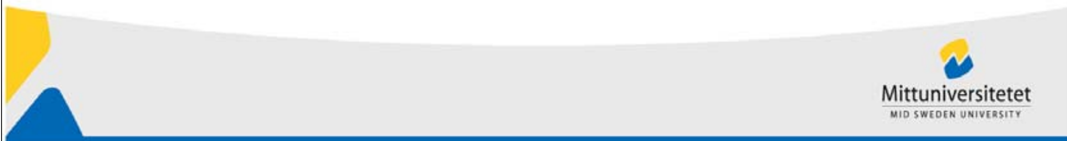
Vid ren Demand cleaning så måste ju en ändrad sida skrivas tillbaka innan en ny kan läsas in. En process med hög PFF (*page fault frequency)* måste vänta på två sidflyttningar (en ut + en in) för varje page fault, vilket inte är effektivt.

Lösningen är igen buffring *Bild 47:*

Det sista innan ni går och börjar hacka ert OS:

## Load Control

- Bestämmer hur många processer som får finnas i primärminnet
- Om det är för få är risken att alla är blockerade av I/O-anrop stor och mycket processortid går åt till swapping
- För många processer leder till thrashing




**Load Control**

*Bild 48:* load control är det som bestämmer antalet samtidiga processer. Vi ska ju hålla processorn sysselsatt, så tillräckligt många ska det vara, men för många leder ju till trashing. Varje process får ju då ett så litet resident Set att risken för page fault är stor. Processorn kommer bara att hinna med att swappa processer med för små resident set.

Hoppa fram till:

## *Process Suspension*

- Vilken process ska avbrytas
- Den med lägst prioritet
- Ej körklar process
  - Kommer inte att ha sina sidor i primärminnet så den kommer att bli blockerad i alla fall
- Senast aktiva process
  - Minst troligt att den har hunnit bli körklar

Mittuniversitetet
MID SWEDEN UNIVERSITY

**Process Suspension**

För swappningen måste vi också bestämma vilken process som ska swappas ut. Boken anger sex olika alternativ. Se *Bild 49-50:* och boken sidan 372.

Så nu kan ni börja skriva ert eget OS. För att ni ska få lite idéer om hur ni ska välja bland alla dessa möjligheter, så avslutar vi med att titta kort på Windows 2000.

Kapitel 8.3 - 8.4 sidorna 371-378 om Unix/Linux är helt kursiva, men intressanta

# *Process Suspension*

- Processen med minst *resident set*
  - “billigast” att ladda igen
- Största processen
  - Ger flest fria sidor
- Processen som har längst återstående exekveringstid

Mittuniversitetet
MID SWEDEN UNIVERSITY

# UNIX och Solaris Minneshantering

- Två olika strategier för kernel och processer
- För processer
  - Sidor, sidtabeller
  - Allokerar diskblock till sidorna
  - *Page frame data table* - beskriver sidans innehåll
  - *Swap-use table* - beskriver sidans användning

Mittuniversitetet
MID SWEDEN UNIVERSITY

# Datastrukturer

| Page frame number | Age | Copy on write | Modify | Reference | Valid | Protect |
|---|---|---|---|---|---|---|

(a) Page table entry

| Swap device number | Device block number | Type of storage |
|---|---|---|

(b) Disk block descriptor

**Figure 8.22 UNIX SVR4 Memory Management Formats**

Mittuniversitetet
MID SWEDEN UNIVERSITY

# Datastrukturer

| Page state | Reference count | Logical device | Block number | Pfdata pointer |
|---|---|---|---|---|

(c) Page frame data table entry

| Reference count | Page/storage unit number |
|---|---|

(d) Swap-use table entry

Figure 8.22 UNIX SVR4 Memory Management Formats

Mittuniversitetet
MID SWEDEN UNIVERSITY

# UNIX och Solaris Minneshantering

- Sidbyte
  - Variant av *clock policy*
- *Kernel Memory Allocator*
  - De flesta block är mindre än en typisk sida

# Linux Minneshantering

- Struktur i flera nivåer
- *Page directory*
- *Page middle directory*
- *Page table*

Mittuniversitetet
MID SWEDEN UNIVERSITY

# Windows 2000 Minneshantering

- Varje process kan använda hela minnesområdet 4 GByte
- W2K Sidor kan vara
  - *Available*
  - *Reserved*
  - *Committed*

**Windows 2000 Memory Management**

*Virtual memory manager* (se sidan 87 igen) sköter allokering av minnet och paging. Sidstorleken, som ju är hårdvaruberoende, kan vara från 4 Kbyte till 64 Kbyte. På Intel-plattform 4 Kbyte (=4096 byte).

*Bild 56-57:* (**OBS det ska stå frames och inte sidor på Bild 56**).

Varje process kan använda hela 32-bitars adressområdet dvs. 4 Gbyte, men 2 Gbyte av dessa är normalt avsatt för systemet.

Minnesområdet är delat i fasta lika stora ramar (normal paging). Ramarna kan ha olika status:

• • *Available*, (tillgänglig) används inte för ögonblicket av processen, ledig för inläsning.

• • *Reserved*, ett antal frames i följd reserverade för processen. Används som en extra cache.

• • *Commited*, (upptagen) upptagna frames som har en koppling till diskens *pagefile* (dvs där de sidor som swappas ut hamnar) där de har en reserverad plats.

Resident set manager använder variabel allokering med lokalt scope. Processen kan alltså bara swappa ut ”sina egna” sidor. Den variabla minnesallokeringen gör att, om det finns gott om minne så läses nya sidor in utan att någon gammal kastas, resident set blir större.

Om det däremot finns lite ledigt minne så återvinns minne genom att OS kastar ut sidor som inte refererats nyligen.

Nu kan ni börja med minneshanteringen i ert nya OS. **Go for IT!**

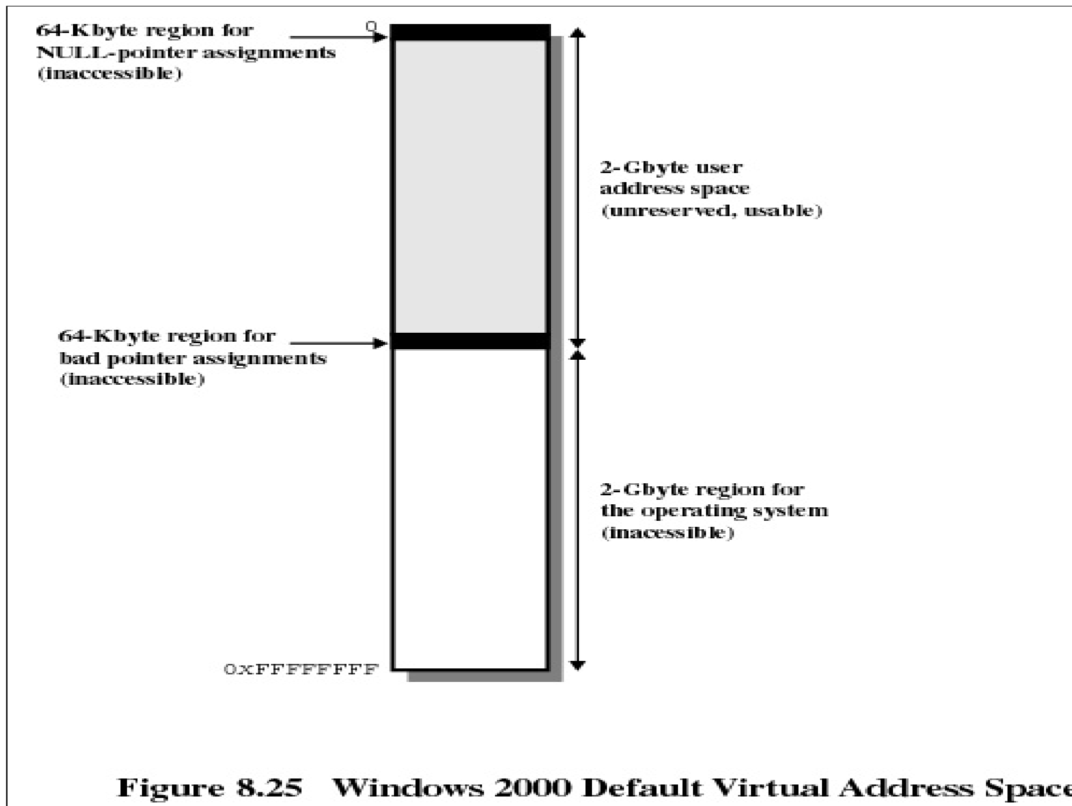



**Figure 8.25 Windows 2000 Default Virtual Address Space**