Operativsystem

Jerker Björkqvist

Minneshantering

Kapitel 4

Andrew S. Tanenbaum

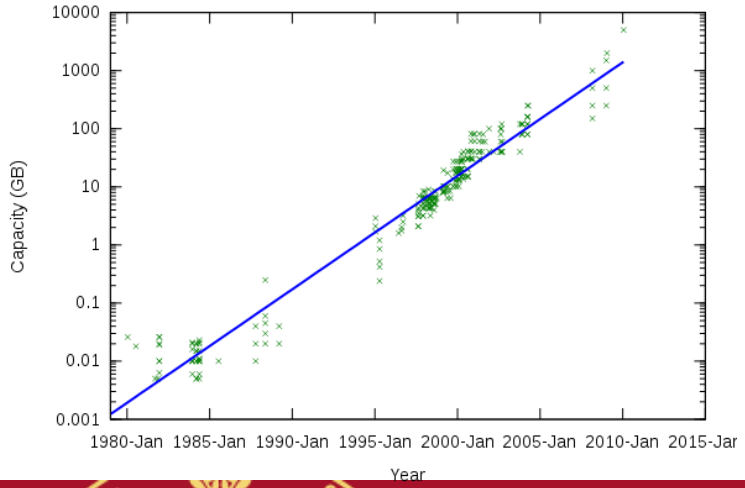
Modern operating systems

Minneshantering

- Minne finns i olika utformningar, med olika kapacitet och snabbhet, det snabbaste vanligen det dyraste
- Operativsystemet bör hålla reda på hur detta minne används
- Primärminne (RAM) används som arbetsminne, minneshantering behandlar främst hur detta minne skall hushållas

Minne generellt

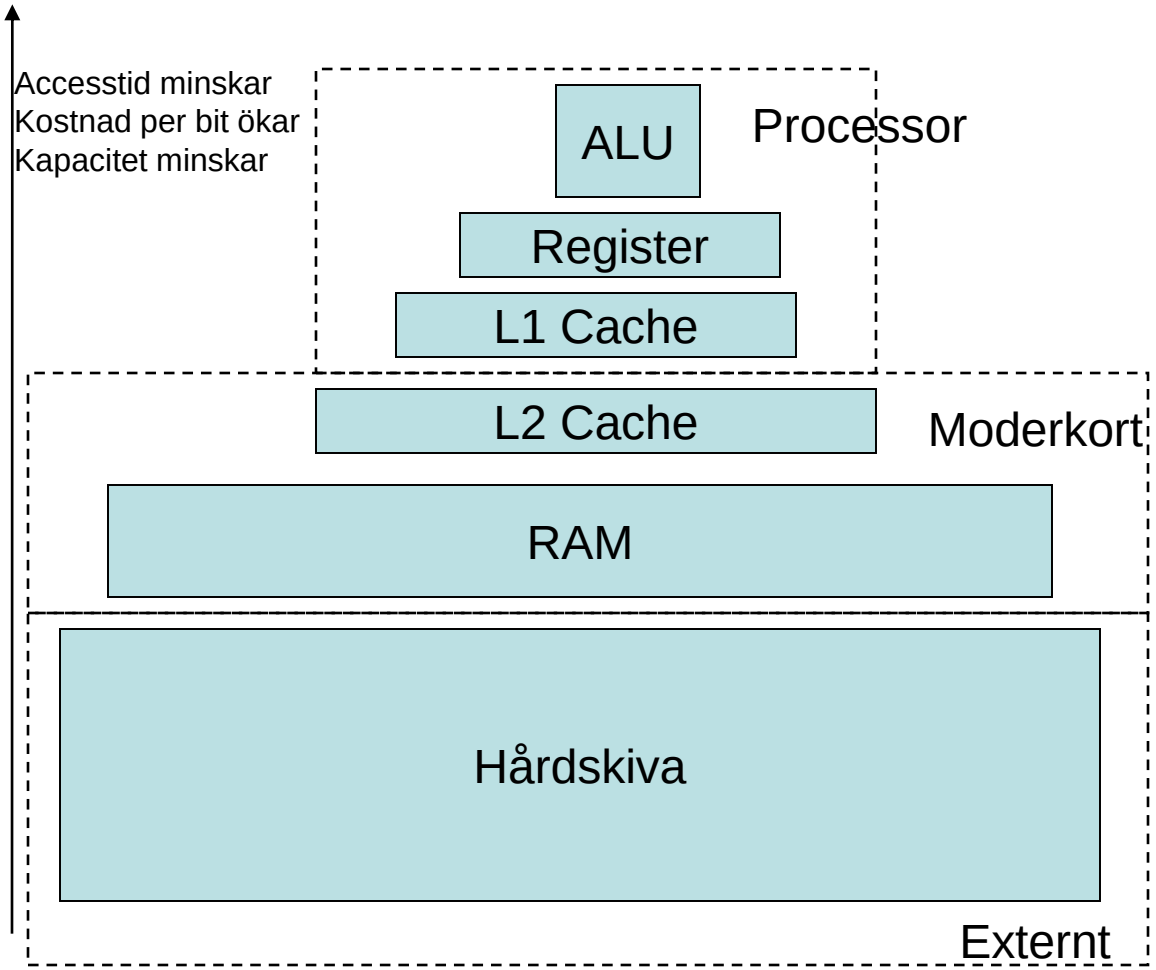
- Datorsystem använder allt mera minne
  - ”640 kB är vad man behöver” (1980-talets PC)
  - VAX system på universitet (80-talet) med 4 MB för ett flertal användare
  - Idag är en normal arbetsstation utrustad med 4-8 GB primärminne, 1000 GB skivminne



Minne

- Mängden primärminne / skivminne brukar anges som potenser av 2:
  - 1 kB = 2^10 Byte = 1,024 Byte
  - 1 MB = 2^20 Byte = 1,048,576 Byte
  - 1 GB = 2^30 Byte = 1,073,741,824 Byte
  - 1 TB = 2^40 Byte = 1,099,511,627,776 Byte
- Men.. T.ex. Nätverk
  - 1Mbit/s = 10^6 bit/s = 1,000,000 bit/s
  - 1 Gbit/s = 10 ^ 9 bit/s = 1,000,000,000 bit/s

Minneshierarki

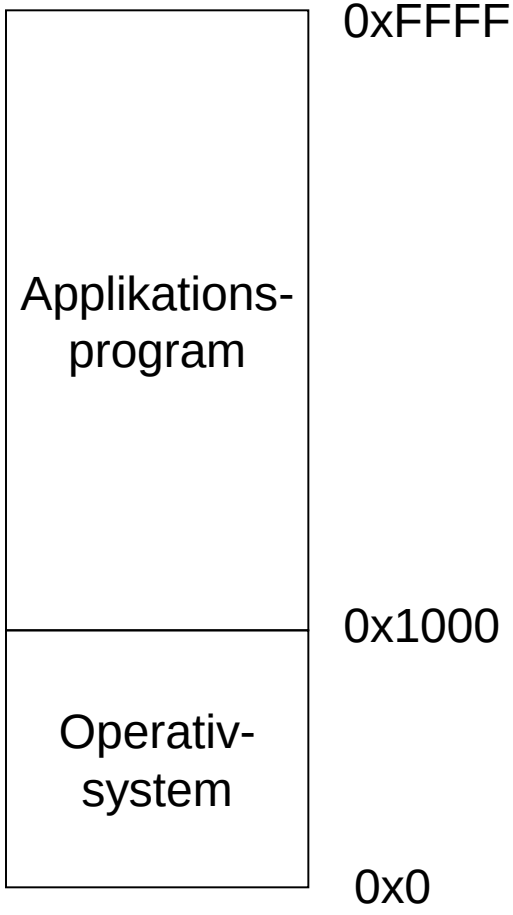


- Vad skall placeras i vilket minne?
- och när?

Minneshanterare sköter!!
(åtminstone delvis)

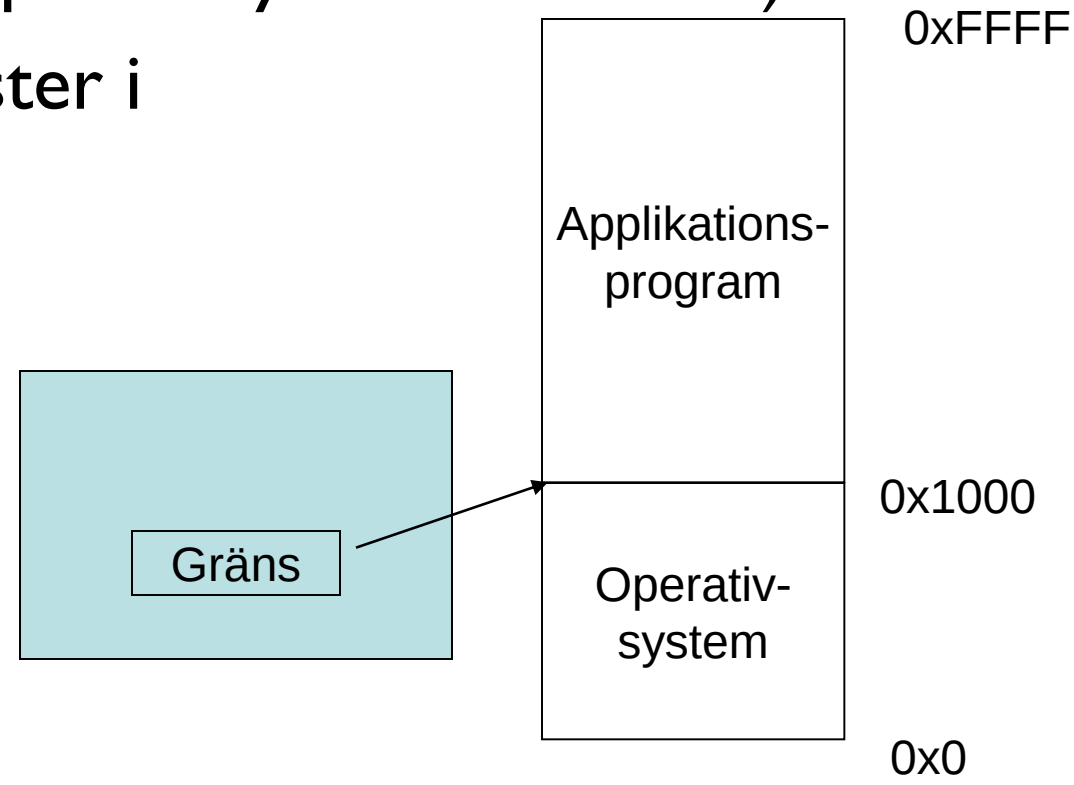
Minneshantering i monoprogrammerade system

- Minnet partitioneras mellan operativsystem och applikationsprogramvara
- Endast ett applikationsprogram åt gången i användning
- Exempel: MS-DOS



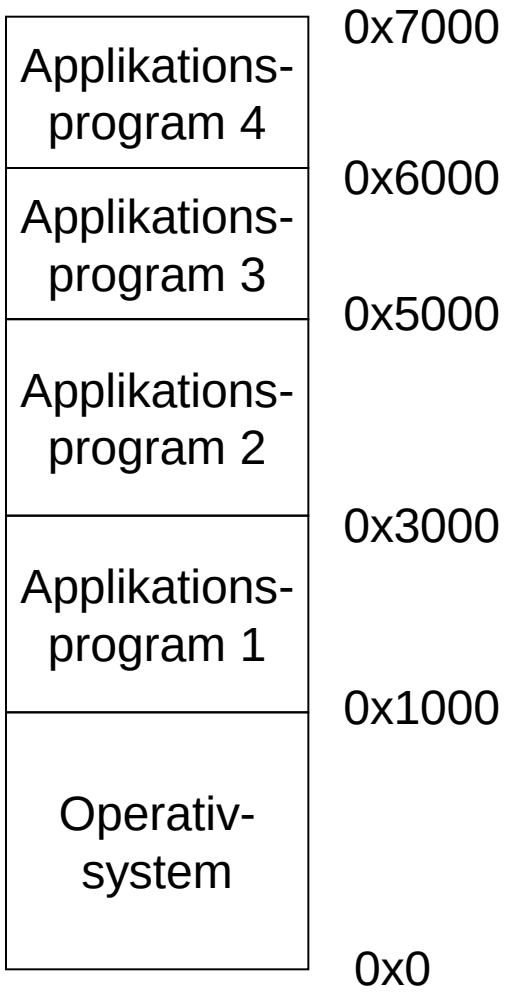
Skydd av minnet

- Hur se till att applikationsprogram inte skriver i otillåtet minne (=operativsystemets minne)
- Inför ett gränsregister i processorn



Multiprogrammering med fixerade partitioner

- Minnet delas upp i partitioner med fixerad storlek
- Vanligen så, att varje minnesområde kan ha ett applikationsprogram aktivt, övriga sätts på kö



Multiprogrammering

- Dagens datorer och operativsystem erbjuder mera. Flere applikationsprogram skall kunna vara aktiva samtidigt
- Eftersom I/O ofta begränsar exekveringen för en process, är det till fördel om CPU:n kan övergår till att exekvera en annan process
  - Detta kräver mer flexibel minneshantering

Probabilistisk analys

- Om vi antar att en process använder en andel p av sin tid för att vänta på I/O
- Med n processer aktiva, innebär detta att alla processer med sannolikheten p^n samtidigt väntar på I/O
- Då är följaktligen nyttjandegraden för processorn $1-p^n$
- Vi kan nu t.ex. beräkna, att om processer använder 80 % av sin tid för att vänta på I/O, måste minst 10 processer vara aktiva för att få en CPU-nyttjandegrad på över 90 %

Svappning (swapping)

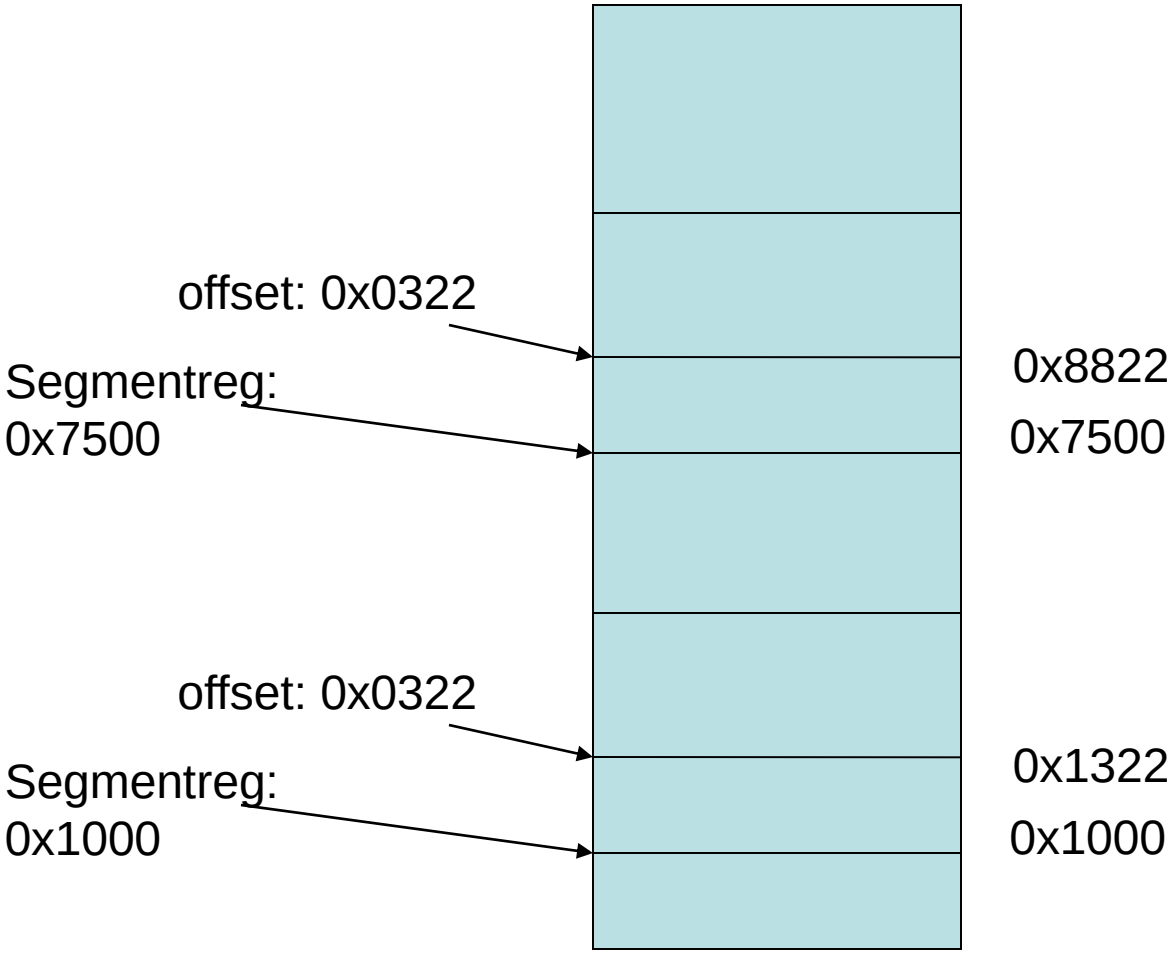
- För att tillåta flere processer att vara körbara, men bibehålla fixerade minnesområden, kan man tillämpa svappning (swapping)
- Då en process väntar på t.ex. I/O, kan en annan ”svappas” in :
- Svappning innebär att minnesrymden från en process skrivs till hårdskiva, och ersätts med minnesrymden för en annan process

Svappning - problem

- Då minnesområden av olika storlek byts in och ut uppstår ”hål” i minnesrymden = fragmentering
- Dessa hål använder med tiden allt större del av minnet -> måsta packa ihop minnet, dvs. omorganisera minnesområdena →
  - behöver mekanism för att pekare och dylikt inne i program också är i kraft efter omflyttning av minnesområden
  - ändra alla pekarvärden??? Nej, för komplext
    - T.ex. Intel 8086/80286/80386/Pentium/... Införde segmentregister, behöver endast ändra segmentregistrets värde

Segment

- pekare: segment:offset
  - ds:0x0322, där ”ds” står för datasegment-registret

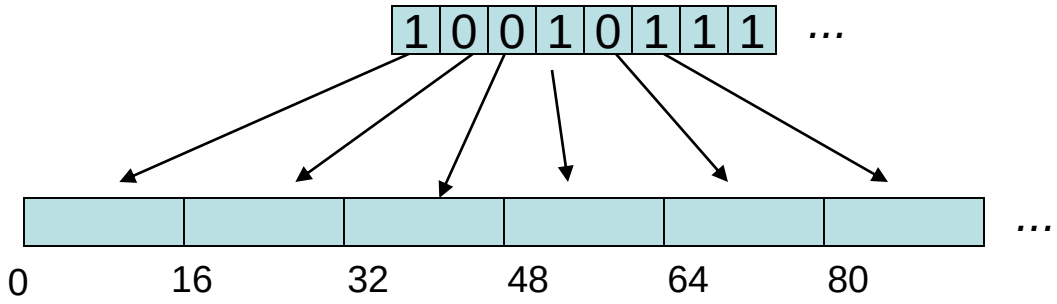


Minnesallokering

- Då OS dynamiskt allokerar minne till processer, måste det hålla reda vilka delar av minnet som är i användning
- Man kan dela upp allokeringsstrategier i två grupper
  - baserade på bitkartor
  - listor med använda/lediga minnesområden

Minneshantering med bitkartor

- Man låter varje bit i en bitkarta representera ett minnesblock
- Storleken på detta minnesblock kan variera mycket
- Andelen minne som används för själva bitmappen är $1/(k+1)$, där k är allokeringsblockets storlek i antal bitar



Problem: T.ex. att hitta ett tillräckligt stort ledigt minnesområde, ty måste hitta en tillräckligt stor mängd konsekutiva 0:or

Minneshantering med länkade listor

- Vi använder en länkad lista innehållande allokerade / lediga minnesområden
- Vi allokering: traversera listan tills tillräckligt stort ledigt minnesområde hittas, dela detta i två delar: Det nya allokerade samt det kvarvarande lediga
- Vid deallokering: Minnet frigörs och kombineras om möjligt med bredvidliggande lediga minnesområden

Algoritmer för att hitta minnesområden

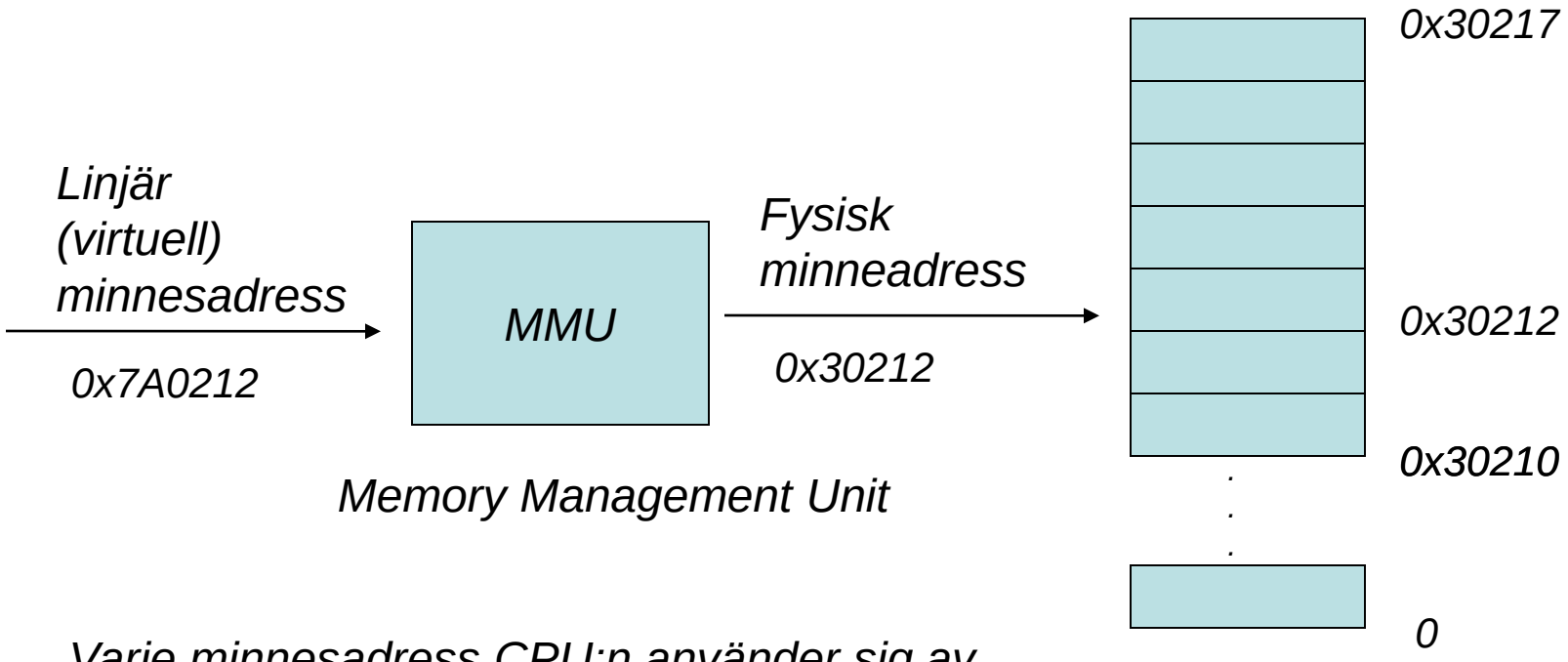
- *First fit* – väljer det första området som är tillräckligt stort
- *Next fit* – som *first fit*, men fortsätter där förra allokeringen slutade
- *Best fit* – väljer det minsta möjliga hålet (dvs. lediga minnesutrymme)
  - kan sortera listorna enligt storlek
- *Worst fit* – väljer det största möjliga hålet
- *Quick fit* – upprätthåller separata listor för olika minnesstorlekar

Problem med direkt adresserat minne

- Fragmentering vid kontinuerlig allokering / deallokering
- Problem med pekare då minnesområden flyttas (trots lösningen med segment)
- Trots att program kan allokera mycket minne, använder de sällan allt minne samtidigt (onödigt att fylla ett fysiskt minne med sådant som inte används)

Lösning: Virtuellt minne

Virtuellt minne - princip

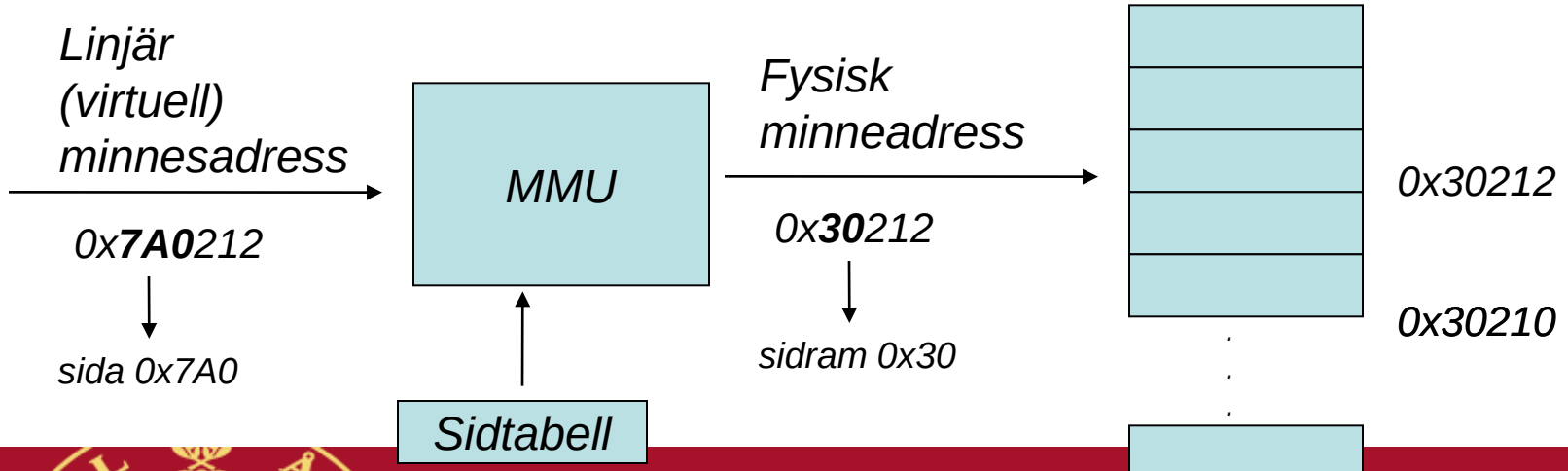


Memory Management Unit

Varje minnesadress CPU:n använder sig av, går via en MMU (hårdvara), där den konverteras till en fysisk minnesadress

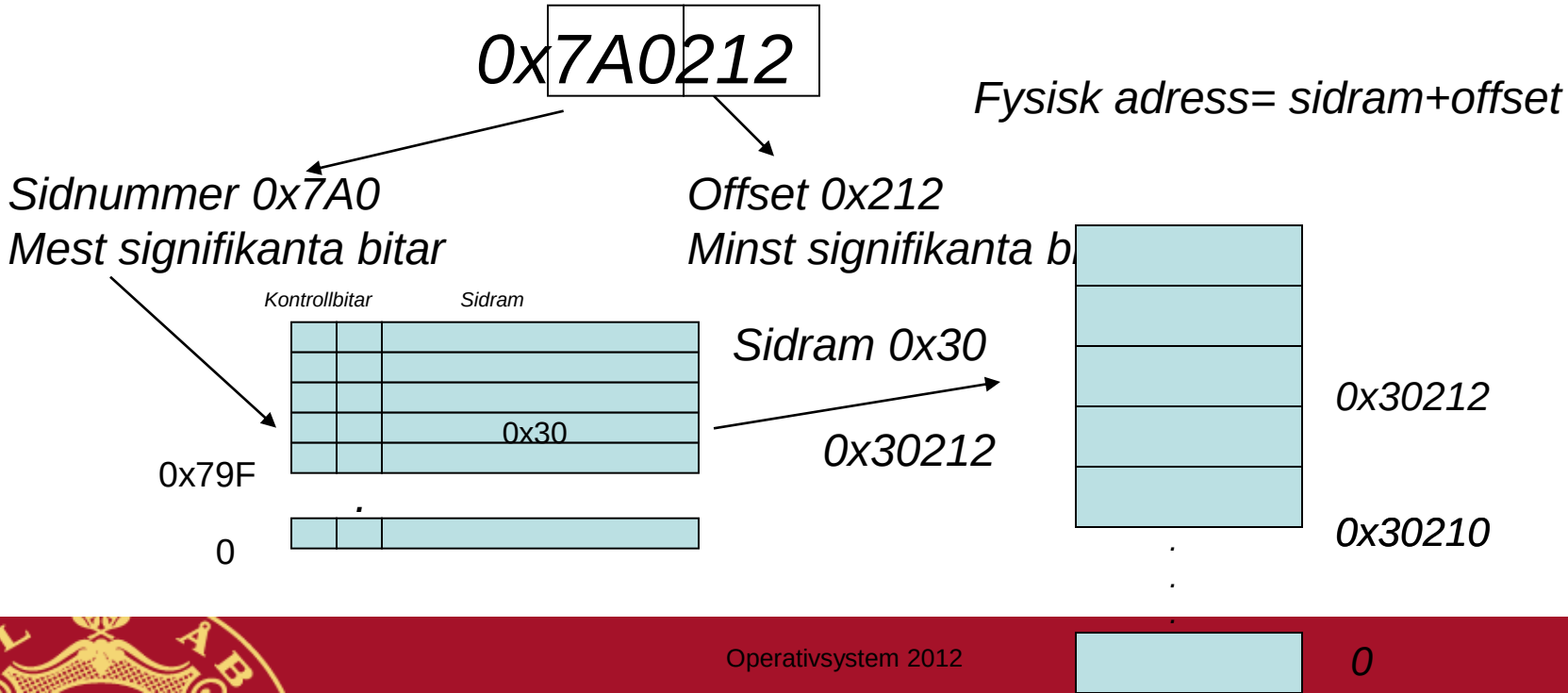
Virtuellt minne och ”paging”

- Det virtuella minnet uppdelas i block med fixerad storlek, **sidor** (**page**)
  - Varje minnesadress finns alltså inom en sida
  - Sidstorlek ofta 4 kB (t.ex. Linux)
- MMU mappar m.h.a. en sidtabell sidan till motsvarande **sidram (page frame)** i det fysiska minnet



Sidtabell

- Den virtuella adressen indelas i *sidnummer* samt *offset*
  - *Om sidstorlek 4 kB -> 12 bitar offset, resten sidnummer, ger index till sidtabellen*



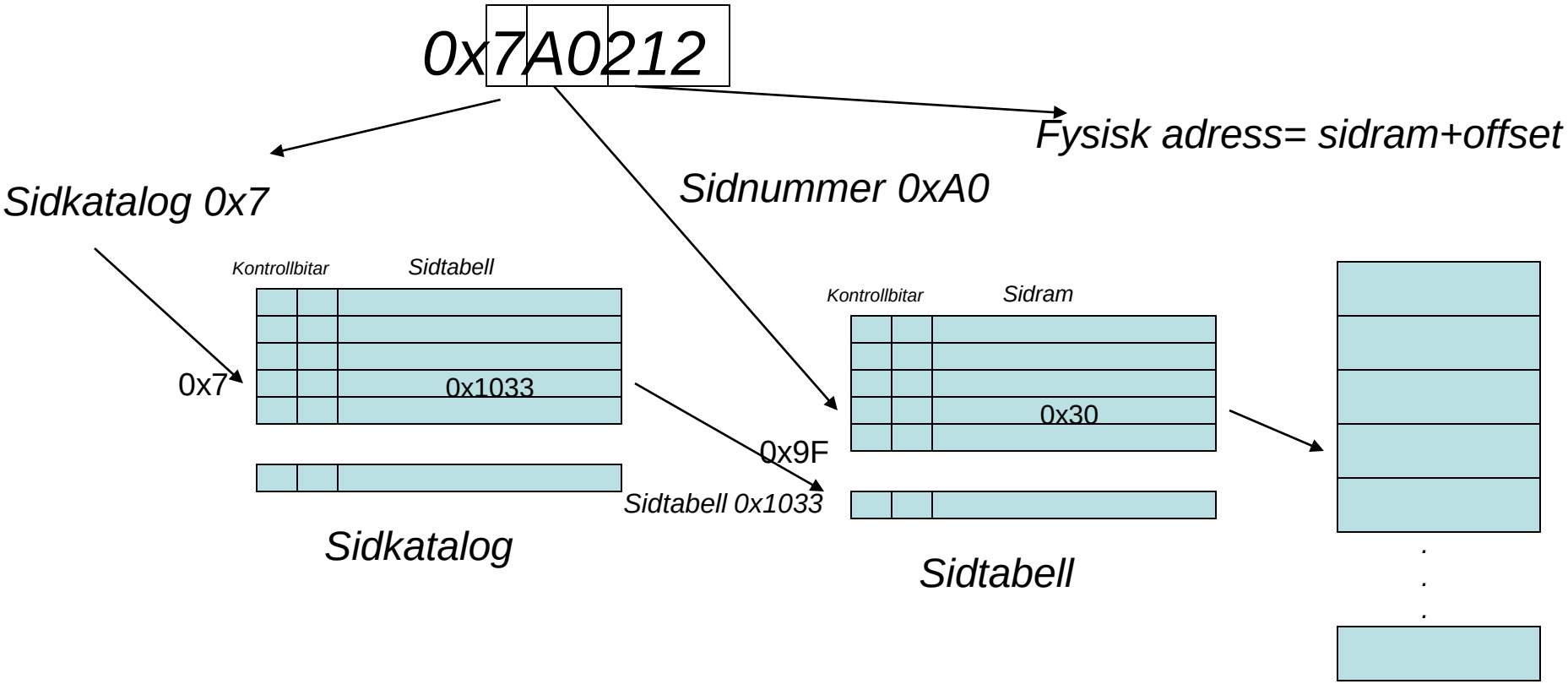
Hierarkiska sidtabeller?

- Sidstorlek 4 kB = 2^12, virtuellt minne totalt 4 GB = 2^32:
  - Antal sidor = 2^20
- Om varje rad i sidtabellen tar 32 bits = 4 bytes
  - Sidtabellen använder då 2^20 * 4 bytes = 4 MB
  - Således, VARJE process använder 4 MB minne enbart för sidtabellen!!

→ Använder sig av 2-nivå-sidtabeller (eller flere), då minskar sidtabellerna till ca 8 kB minne (2^10 * 2^10), dvs normalt 2 st tabeller med 1024 rader i vardera 2x1024’4 byte = 8 kB

Sidtabell – hierarkisk

- Inför sidtabeller i flere nivåer: Sidkataloger + sidtabeller

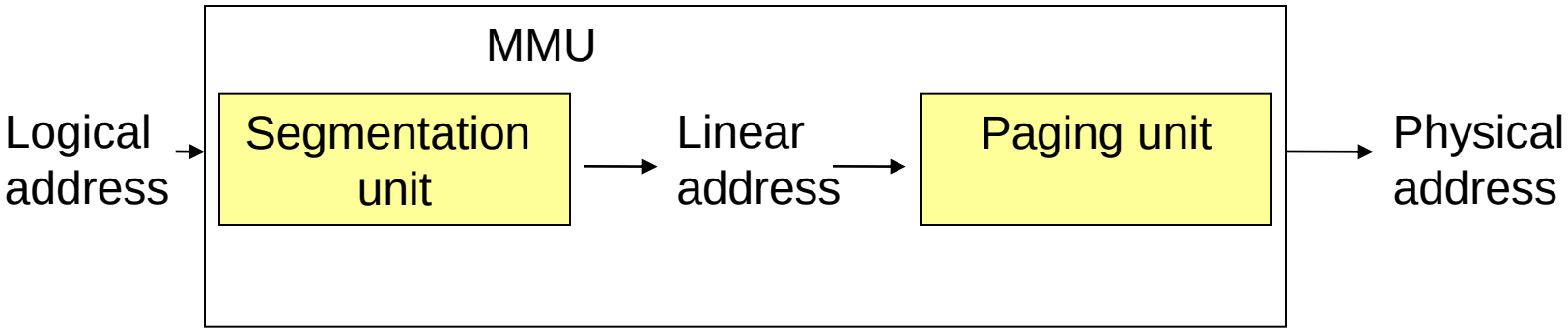


Virtuellt minne - fördelar

- Det blir enkelt att låta flera processer köra samtidigt
- Varje process kan får tillgång till ett minnesområde som är större än det fysiska minnet (t.ex. Linux alltid 3 GB)
- Program kan köras fast det endast delvis finns i fysiska minnet
- Program kan dela samma fysiska representation av t.ex. bibliotek
- Program kan flyttas (relokeras) vart som helst i det fysiska minnet utan några komplikationer
- Det blir enklare att skriva maskinoberoende program (eftersom minnesbilden är lika oberoende av platform)

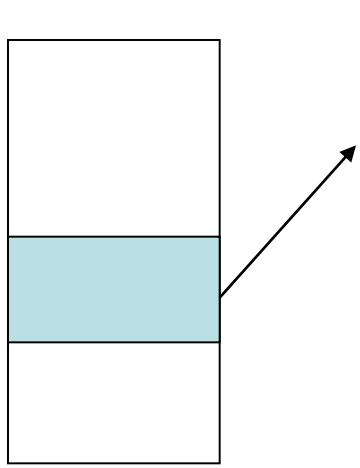
Adressering av minne – I386

Intel 386 – segmentation + paging



Segment selector
in segmentation register:
cs, ss, ds, es, fs, gs

Segment descriptor
- base 32 bit
- limit 20 bit
- flags, type, protection

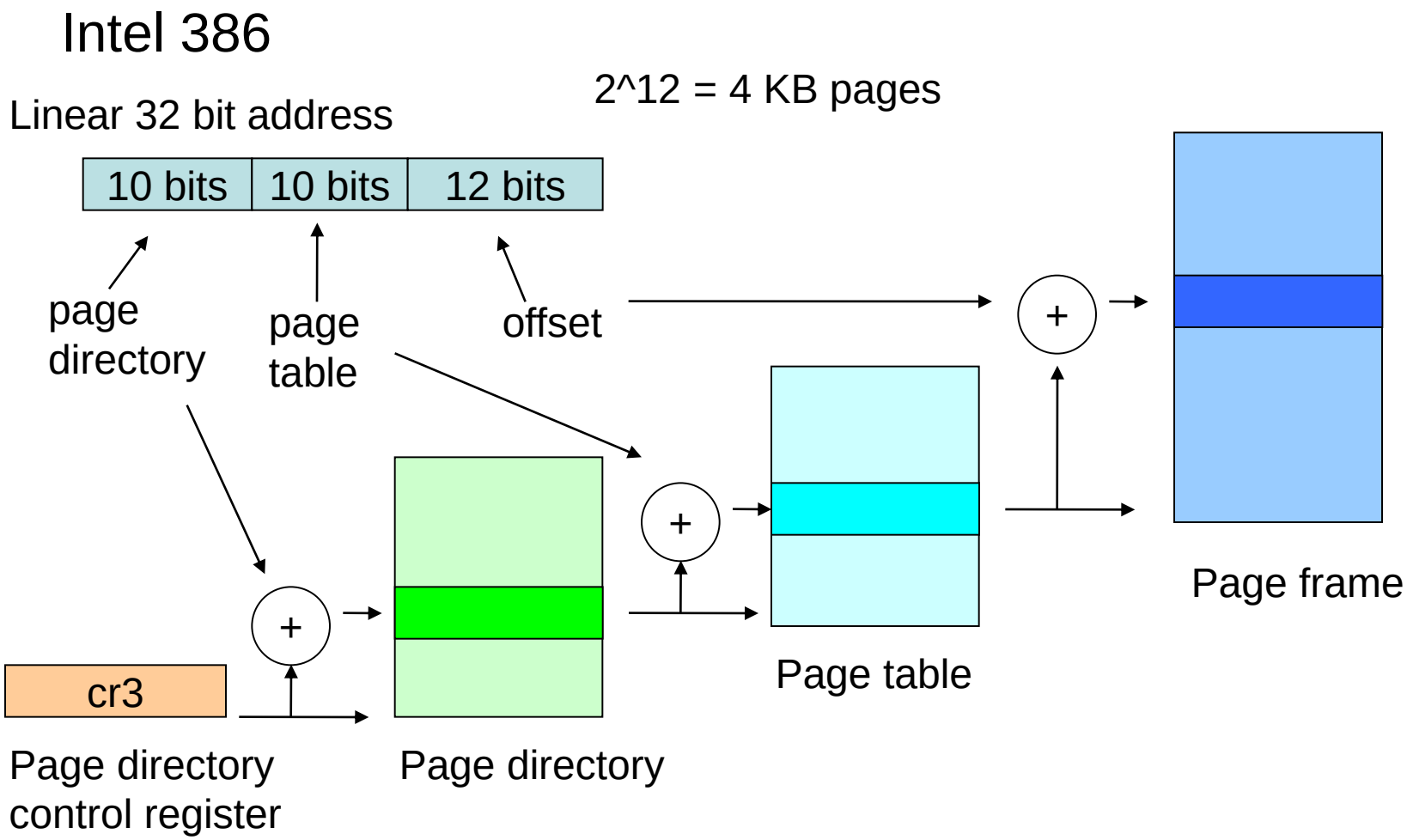


Linear address =
base + offset

E.g. mov ($0x41232), %eax

base from **ds** descriptor
linear = base + $0x41232

Paging i I386



Inverterade sidtabeller

- Problemet med att sidtabellerna blir stora kan även lösas med inverterade sidtabeller
- En inverterad sidtabell innebär att endast sidramar har en rad i sidtabellen
- Leder dock till mycket processering för att hitta sidramar: virtuell-till-fysisk mappning fungererar inte längre via index i sidtabeller – men TLB (se senare) löser delvis detta problem

Rad i sidtabell

- En rad i en sidtabell brukar innehålla ungefär följande fält
  - Flagga för signalering om i fysiskt minne eller ej (present flag)
  - 20 bitar för adress till fysisk sidram (4 kB sidor)
  - Accesserad-flagga (Referenced)
  - Dirty flag (Om modifierad)
  - Read/Write flagga (Skyddsnivå)
  - User/Supervisor flagga (Skyddsnivå)
  - PCD and PWT flags (hardware cache)
  - Page size (gäller för Intel arkitektur)
  - Global flag

Sidfel – *page fault*

- Vad händer när det inte finns en sidram som motsvarar den virtuella adressen (dvs. ”*present flag*” = 0)?
  - Sidfel (*page fault*)
- CPU:n avbryter programexekveringen och skapar ett sidfel (p*age fault*), ett mjukvaruavbrott till OS
- OS måste nu vidta nödvändiga åtgärder för att skapa en sidram för den process som blev avbruten p.g.a. sidfelet

Sidfel - exempel

```
[jbjorkqv@borken tmp]$ ps v
  PID TTY      STAT   TIME  MAJFL   TRS   DRS  RSS %MEM COMMAND
 3491 pts/7    S      0:02      0   269  2942 1892  0.7 -tcsh
 3693 pts/7    T      0:07     48   998  7397 4348  1.7 emacs -nw hello.c
 3779 pts/7    R     24:43      0     1  1350  332  0.1 ./a.out
 3820 pts/7    R      0:01      0    60  3115 1240  0.4 ps v
```

Hantering av sidfel

- Hårdvaran skapa sidfel, mjukvaruavbrott (fault) till OS görs
- OS sparar register,
- OS reder ut vilken sida som efterfrågades
- Checkar om adressen är giltig
- Om det finns lediga sidramar, allokera, annars frigör en sidram
- Om data måste läsas in från skiva, sätt en läsinstruktion från skivan på kö
- Vänta till data tillgängligt (under tiden blockeras nuvarande process)
- Då data finns tillgänligt, fortsätt nuvarande process

Demand paging

- Allokering av sidramar fördröjs tills de verkligen behövs – dvs. tills då processen försöker accessera den sida som inte finns i RAM
- Motivering för ”demand paging”
  - Processer använder inte alla sina minnessidor genast från början
    - Det är tom. möjligt att de aldrig använder en given sida

”Copy On Write”

- I tidiga implementationer av processdeling, gjordes en exakt kopia av processens adressrymd
  - Sidramar för sidtabellerna
  - Sidramar för själva sidorna
  - Initialisering av sidtabeller
  - Kopiering av sidor från föräldern till barnprocessen
- Dagens operativsystem utför ”Copy On Write” (COW)
  - Inställer för att kopiera sidor, delas sidorna ända tills antingen föräldern eller barnet skriver till sidan
    - MMU genererar en exeption, kärnan kan då skapa en kopia av sidan

Reserverade sidramar

- I Linux sparas kärnans kod och data i en grupp reserverade sidramar – dessa swappas aldrig till hårdskivan
- I Linux i386 startar kärnar på fysiska adressen 0x00100000. En typisk konfiguration av kärnar tar mindre än 2 MB RAM.

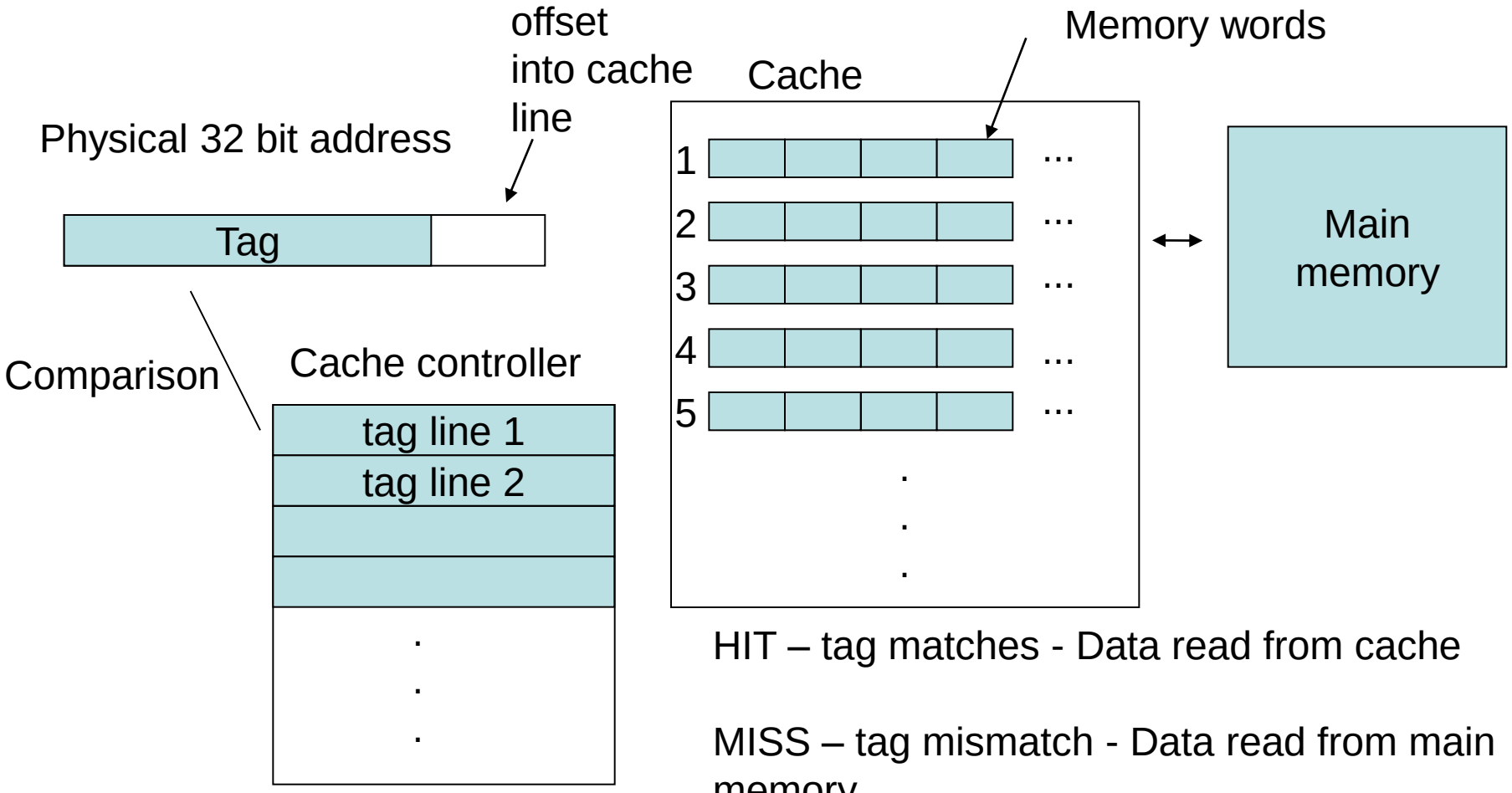
Hårdvaru-cache

- Processorn är mycket snabbare än primärminnet
  - Processorn måste vänta många klock-cykler på data
- Närhetsprincipen på minnessaccess
  - Framtida minnesreferenser är ofta nära nuvarande (t.ex. kod / räckor)
- Mindre och snabbare minnesenheter för det minne som är i användning för tillfället
- Cache-linjer som innehåller några dussin kontinuerliga bytes

Hårdvaru-cache

- Full avbildning
  - En linje i primärminnet sparas alltid på samma ställe i cache-minnet
- Fullt associativ avbildning
  - En linje i pirmärminnet kan sparas varsomhelst i cache-minnet
- N-vägs associativ avbildning (det normala)
  - En linje i primärminnet kan sparas i någon av N linjer i cache-minnet

Hardware cache



HIT – tag matches - Data read from cache

MISS – tag mismatch - Data read from main memory

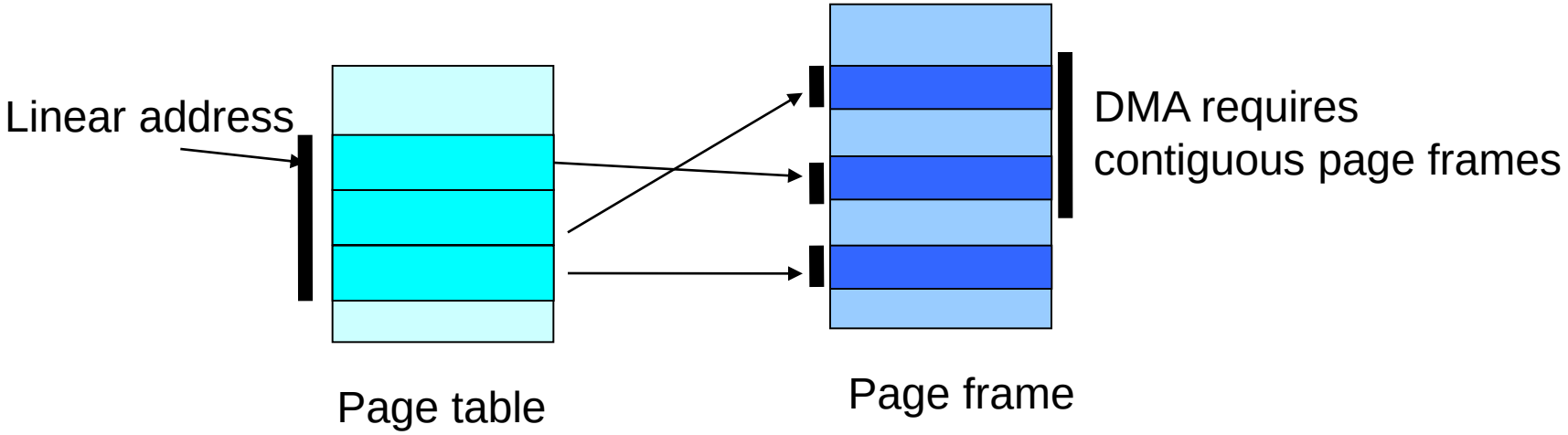
Translation Lookaside Buffers (TLB)

- ”Cache” för sidtabeller (istället för i långsamt RAM-minne)
- Fysiska adresser sparas i TLB:s
  - Referenser till samma linjära adressrymd kan nu snabbt derefereras

NOTERA! Hårdvaru-cache och TLB:s hanteras i huvdsak av hårdvara, helt transparent för användarkod / kod i kärnan. Ofta finns dock en möjlighet att i viss mån påverka hur cache och TLB beter sig genom flaggor.

Physical memory fragmentation - a problem ?

- Normally, the paging system does not require contiguous page frames
- However:
  - DMA may need contiguous page frames, as DMA bypasses the paging system



Fragmentering

- Trots att problemet med fragmentering av det fysiska minnet p.g.a. virtuellt minne i stort har försvunnits, kvarstår dock problem med minnesfragmentering
- Den linjära adressrymd som processer använder sig av kan naturligtvis fragmenteras (upprepade malloc() / free() skapar ”hål” i den virtuella adressrymden)