Computer Arkitektur
Virtuel Hukommelse

Forelæsning 10
Brian Nielsen

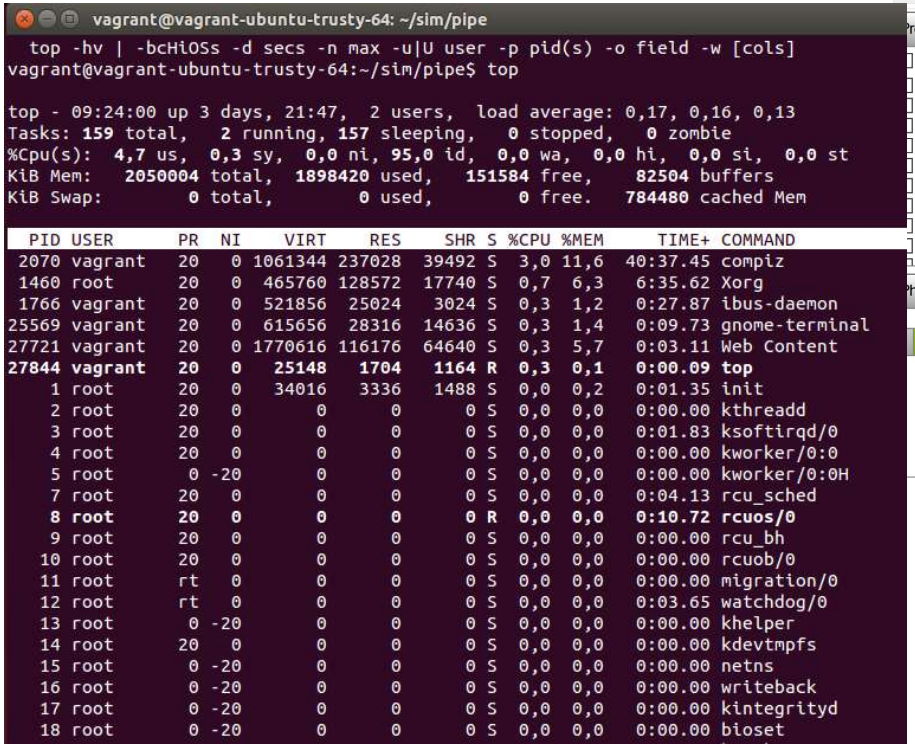
Credits to
Randy Bryant & Dave O'Hallaron (CMU)
Youjip Won (KAIST)

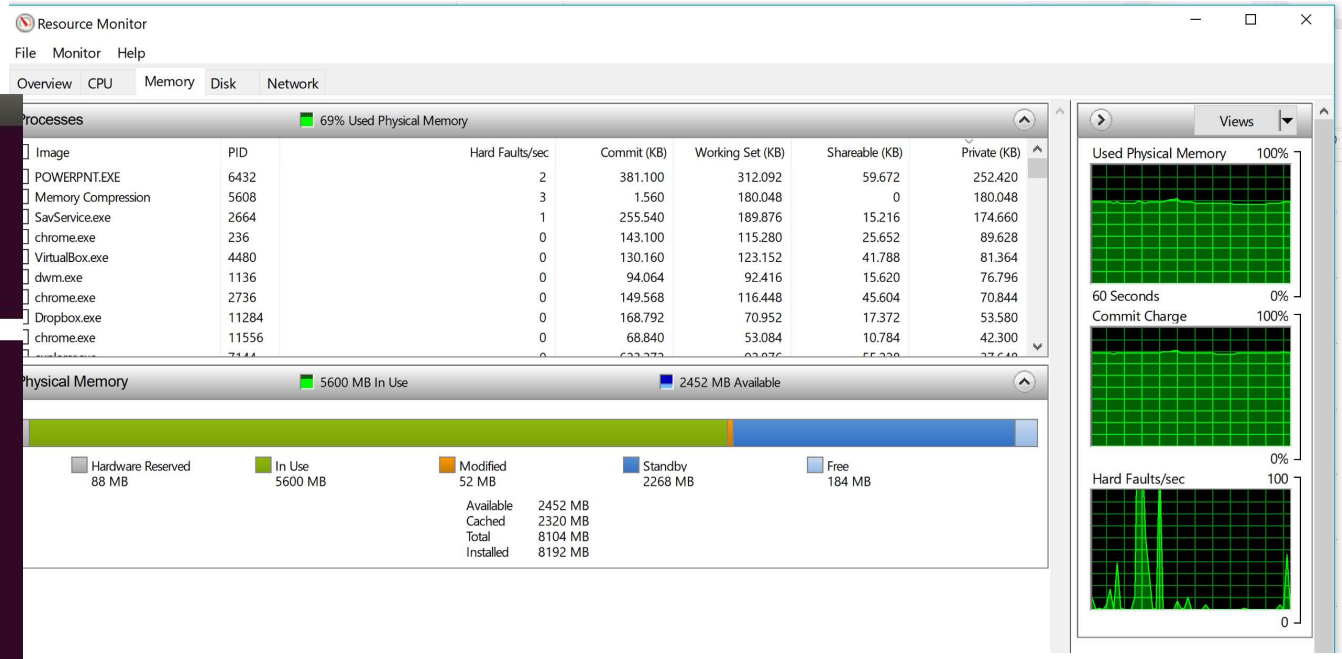
Virtuel og Fysisk hukommelse

Processer:

Manger kørende programmer = mange processer

- Hver med: Eksekverings-tråd og Adresserum





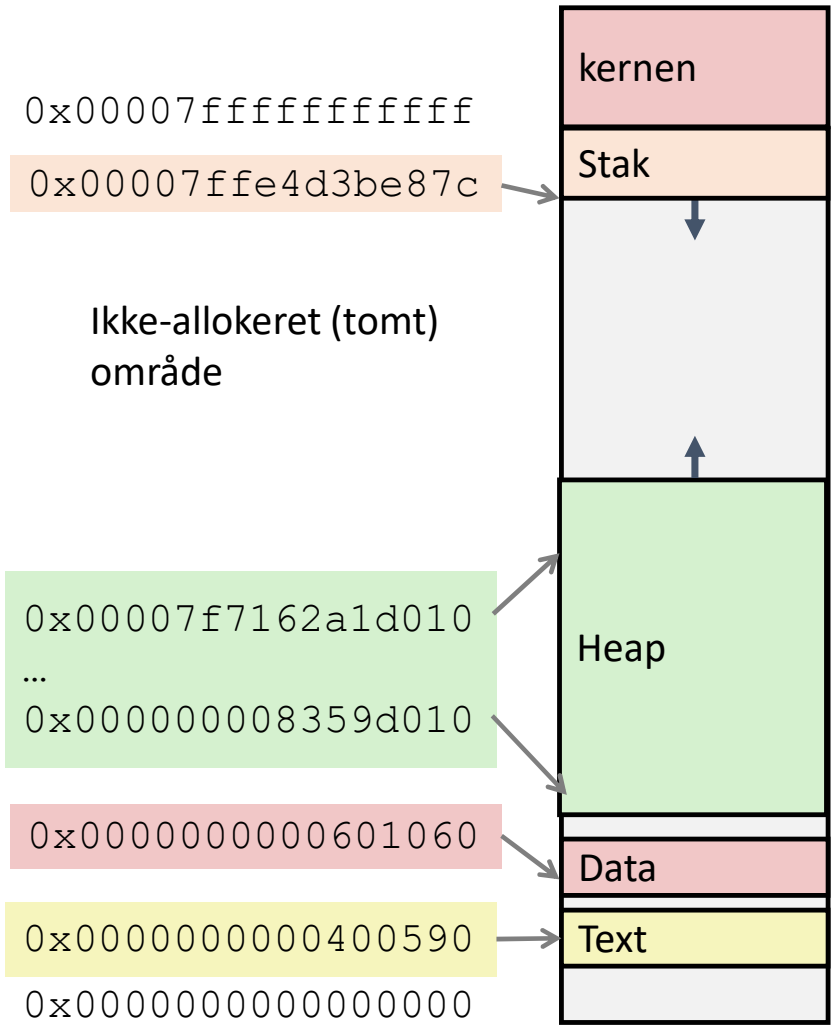
Hvorfor Virtuel Hukommelse (VM)?

- Mere effektiv udnyttelse af primær hukommelsen
  - Større adresserum end fysisk ram tillader!
  - Afvikling af flere processer end fysisk ram tillader!
  - Bruge DRAM som cache for dele af de virtuelle adresserum
- Simplificerer administration af hukommelsen
  - Mindsker spild
  - Hver proces får ensartet lineært adresserum og layout
  - Hvad er ledigt/brugt?
- Isolerer adresserum
  - En proces kan ikke læse eller overskrive en andens hukommelse
  - Et bruger-niveau program kan ikke tilgå data og kode i operativsystemskernen (privilegeret information)

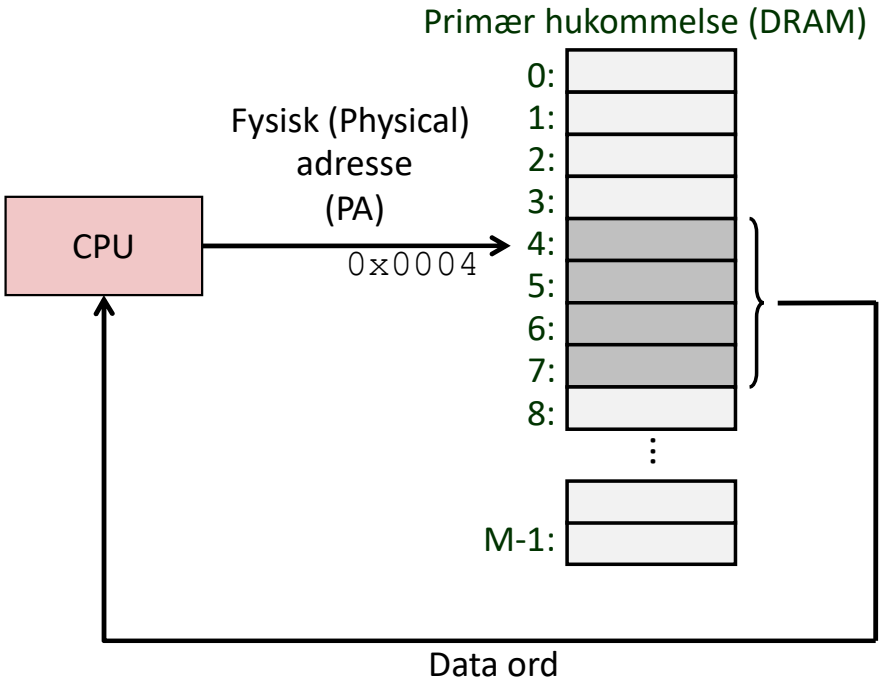
Adresserum

- Lineært adresserum (address space): Ordnet mængde af sammenhængende positive heltal:
  {0, 1, 2, 3 … }
- Virtuelt adresserum: Mængde af N = 2^n virtuelle adresser
  {0, 1, 2, 3, …, N-1}
- Fysisk (Physical) adresserum: Mængde af M = 2^m fysiske adresser
  {0, 1, 2, 3, …, M-1}
- Proces = kørende program:
  - **sit eget private virtuelle adresserum**
  - **Samme ensartede layout**
- Ikke alle adresser er nødvendigvis i brug: kan indeholde (store) uallokerede områder

Virtuelt adresserum for en proces

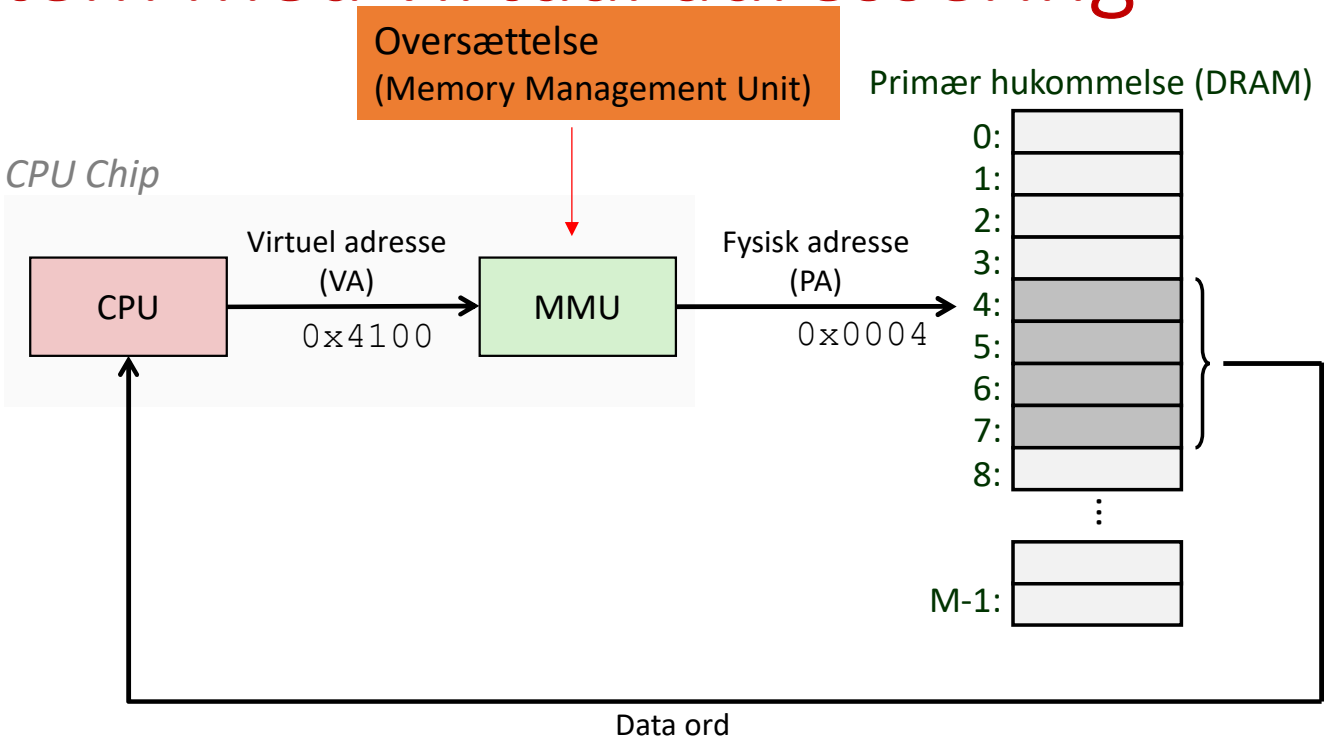


Et system der bruger fysisk adressering



- Bruges i simple systemer, som micro-controller i ”indlejrede systemer” og apparater (biler, elevatorer, sensorer, …)

Et system med virtual adressering



- Bruges i alle moderne PC'ere, bærbare, smart-phones, Linux baserede IoT gateways
  - Anvendes ikke i micro-controllere, DSPer, særligt tidskritiske systemer, visse super-computere
- En af de "geniale" abstraktioner indenfor software/datalogi

Eksempel på adresse oversættelse

- C - Kode

```
void func()
        int x=3000;
        ...
        x = x + 3; // this is the line of code we are interested in
```

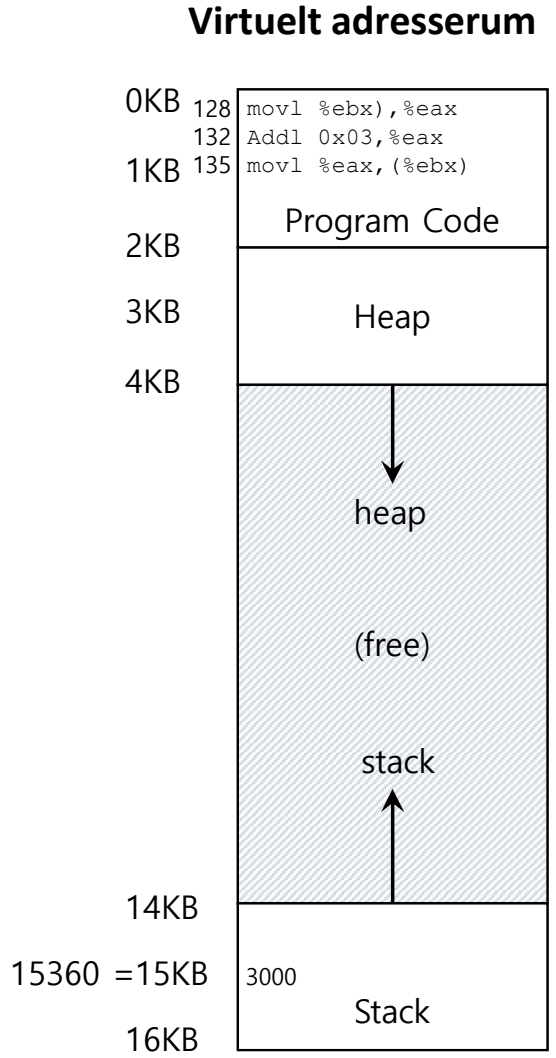
- Assembler

#Antag at *adressen (15360) på 'x' er gemt i ebx register.*

```
128 : movl (%ebx), %eax    ; load Mem[%ebx] into eax
132 : addl $0x03, %eax     ; add 3 to eax register
135 : movl %eax, (%ebx)    ; store eax back to mem[%ebx]
```

- Indlæs instruktion fra adresse 128
- Udfør: læs (load) fra adresse 15KB = 15*1024= 15360
- Indlæs instruktion fra adresse 132
- Udfør ingen hukommelses adgang
- Indlæs instruktion fra adresse 135
- Udfør: gem (store) til adresse 15KB = 15*1024= 15360

*) 1KB=1024 bytes

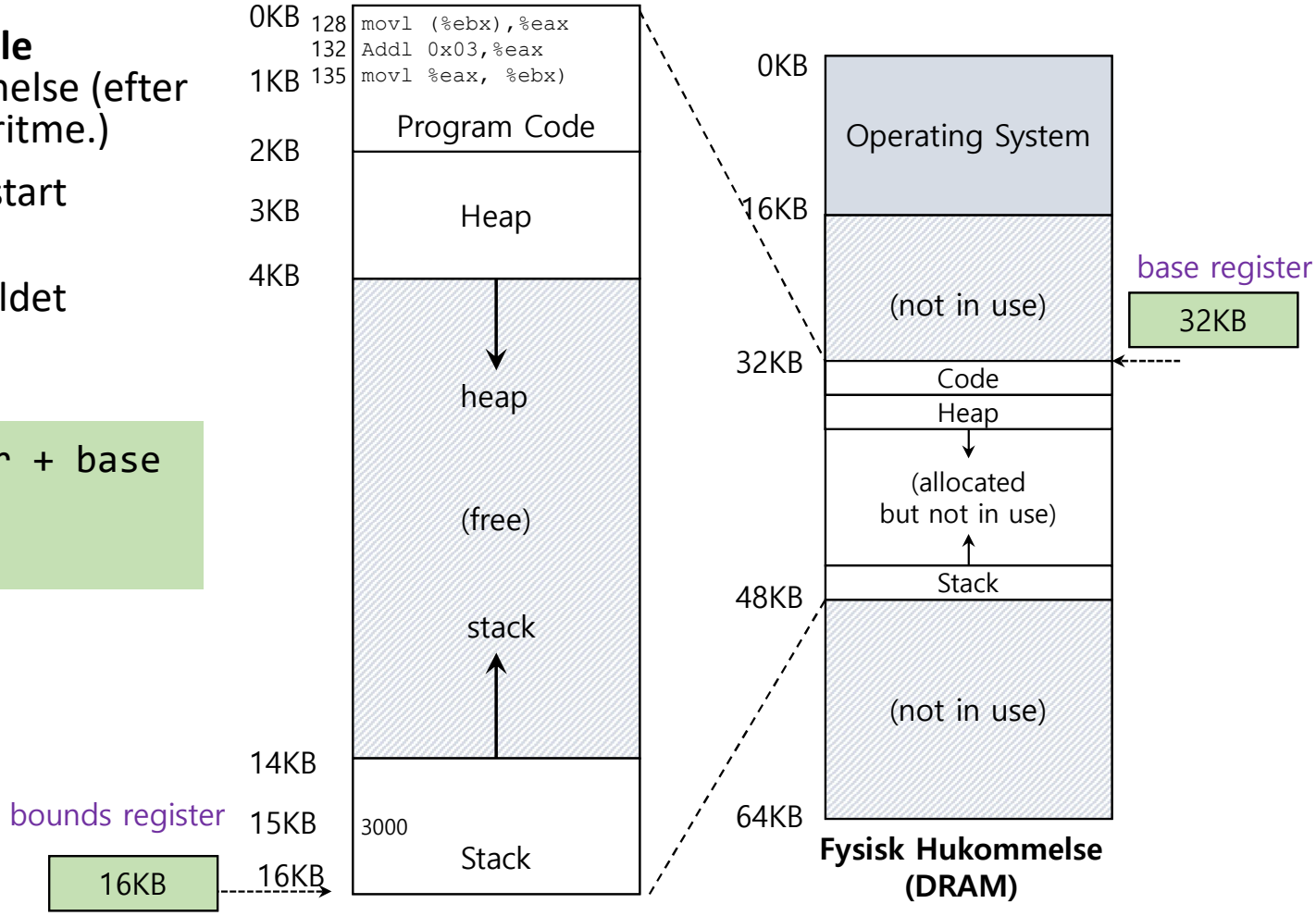


Oversættelse vha. base og grænse registre

- OS indplacerer processens **virtuelle adresserum** i den fysiske hukommelse (efter en eller anden indplacerings algoritme.)
- **Base register** angive dets fysiske start adresse
- **Grænse (Bounds) register** indeholdet længde
- MMU laver adresseoversættelse:

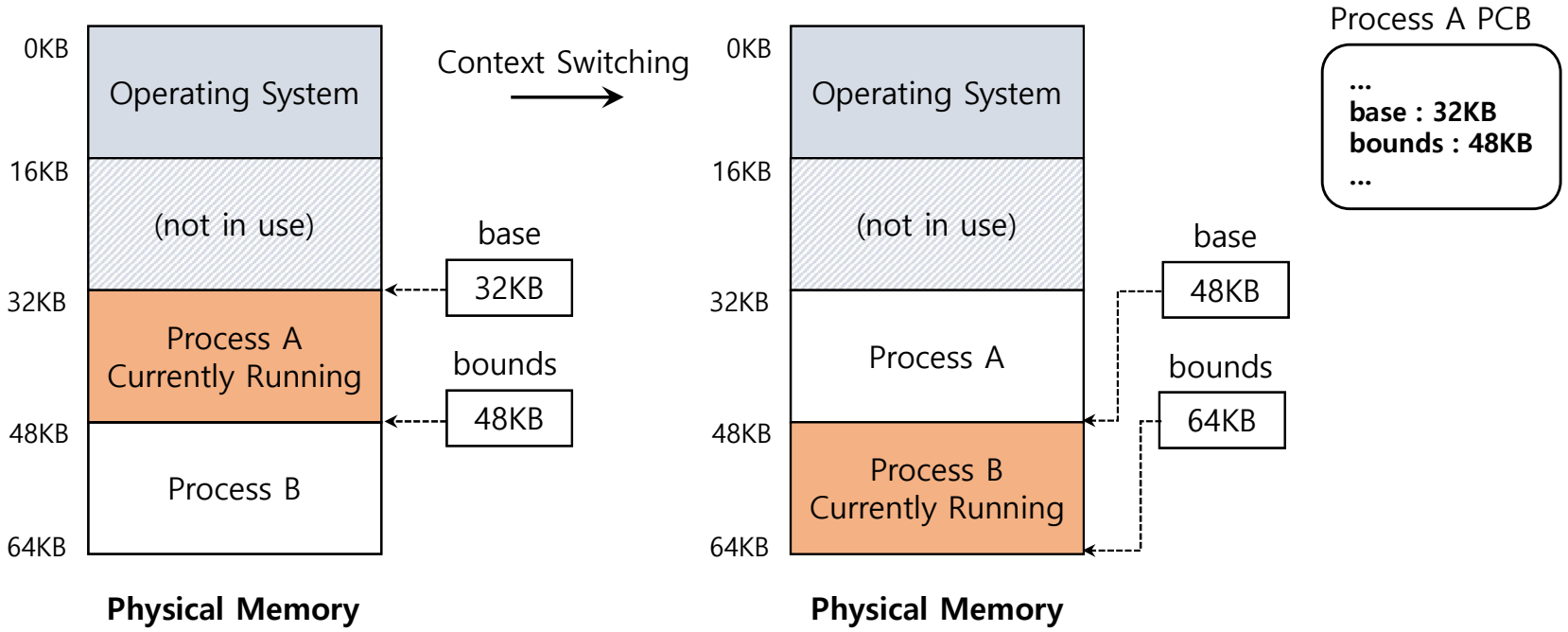
```
1. Fysisk addr = virtuel addr + base
2. Virtuel addr < bounds
   (ellers exception)
```

- Ex: load instr. på adresse 128
  - 32896=128+32KB($base$)
- Ex: store på adresse 15KB
  - 47KB=15KB+32KB($base$)



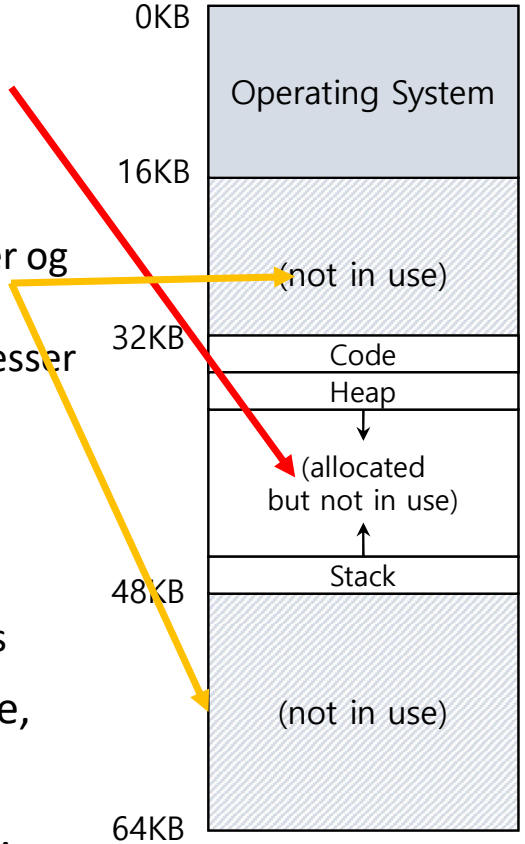
Kontext-skifte fra A til B

- Base og grænse registre for kørende process A gemmes i PCB A
- Base og grænse for ny process indlæses fra PCB B

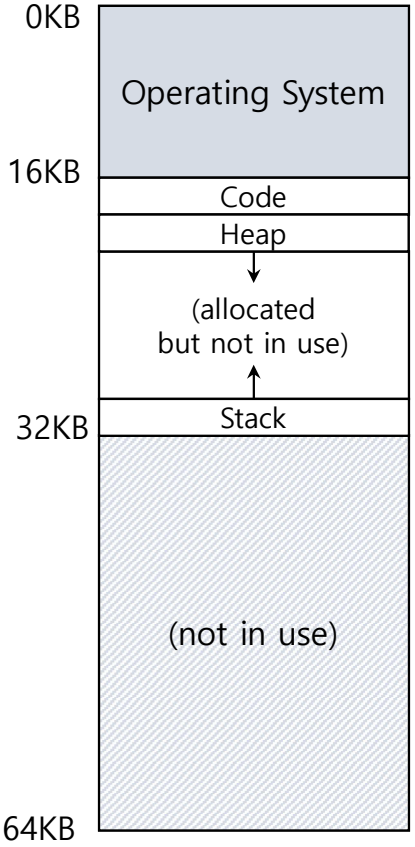


Begrænsninger ved base+grænse metoden

- Intern fragmentering
  - Allokeret plads internt i proces ikke udnyttet
- Ekstern fragmentering
  - Efter nogen tid hvor processer starter og terminerer bliver der "huller"
  - For små ledige "huller" mellem processer
  - Fx ny proces X fylder 20 KB kan ikke indplaceres
- Sammenpresning (Compaction)
  - men er dyrt da store hukommelsesområder skal kopieres
- Segmentering afhjælper delvist: kode, data, hop, stak segmenter
- Men stadig begrænset af fysisk hukk.



Fragmenteret Fysisk Hukommelse



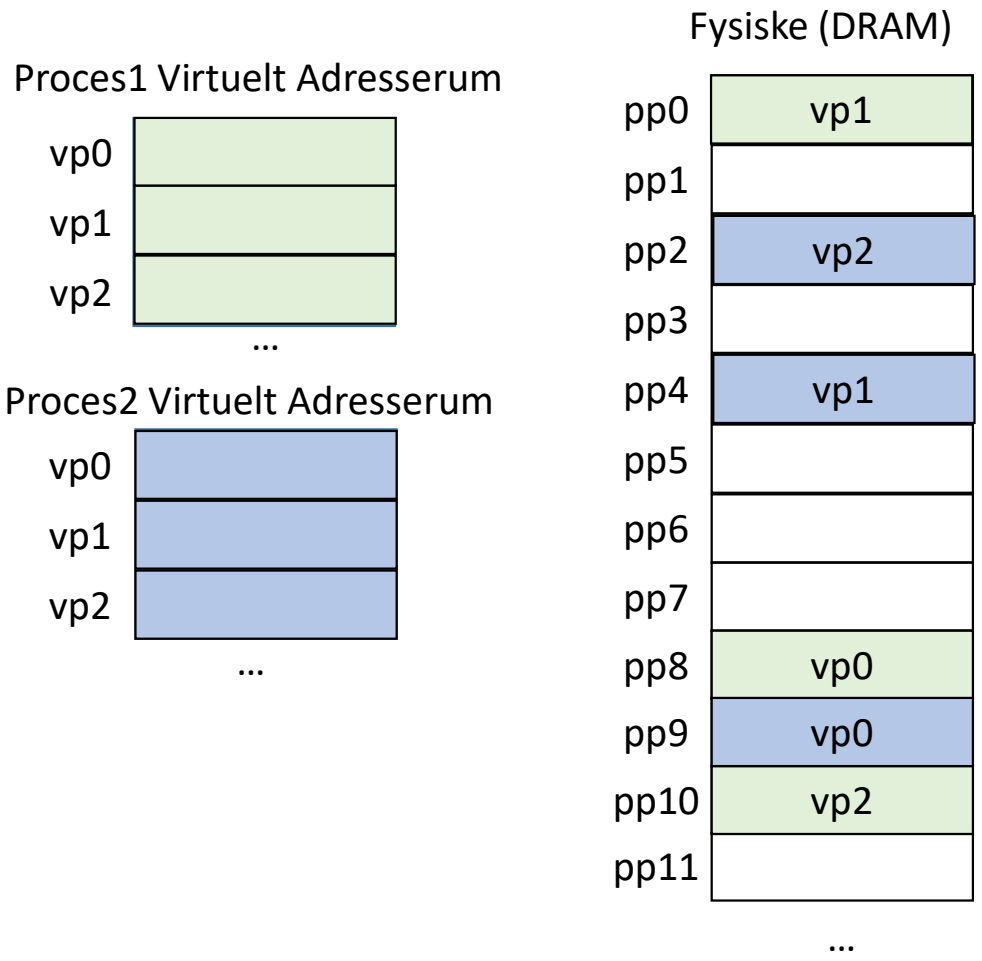
"Compacted" Fysisk Hukommelse

Side-baseret hukommelsesorganisering (Paging)

Side-baseret hukommelses-organisering (Paging)

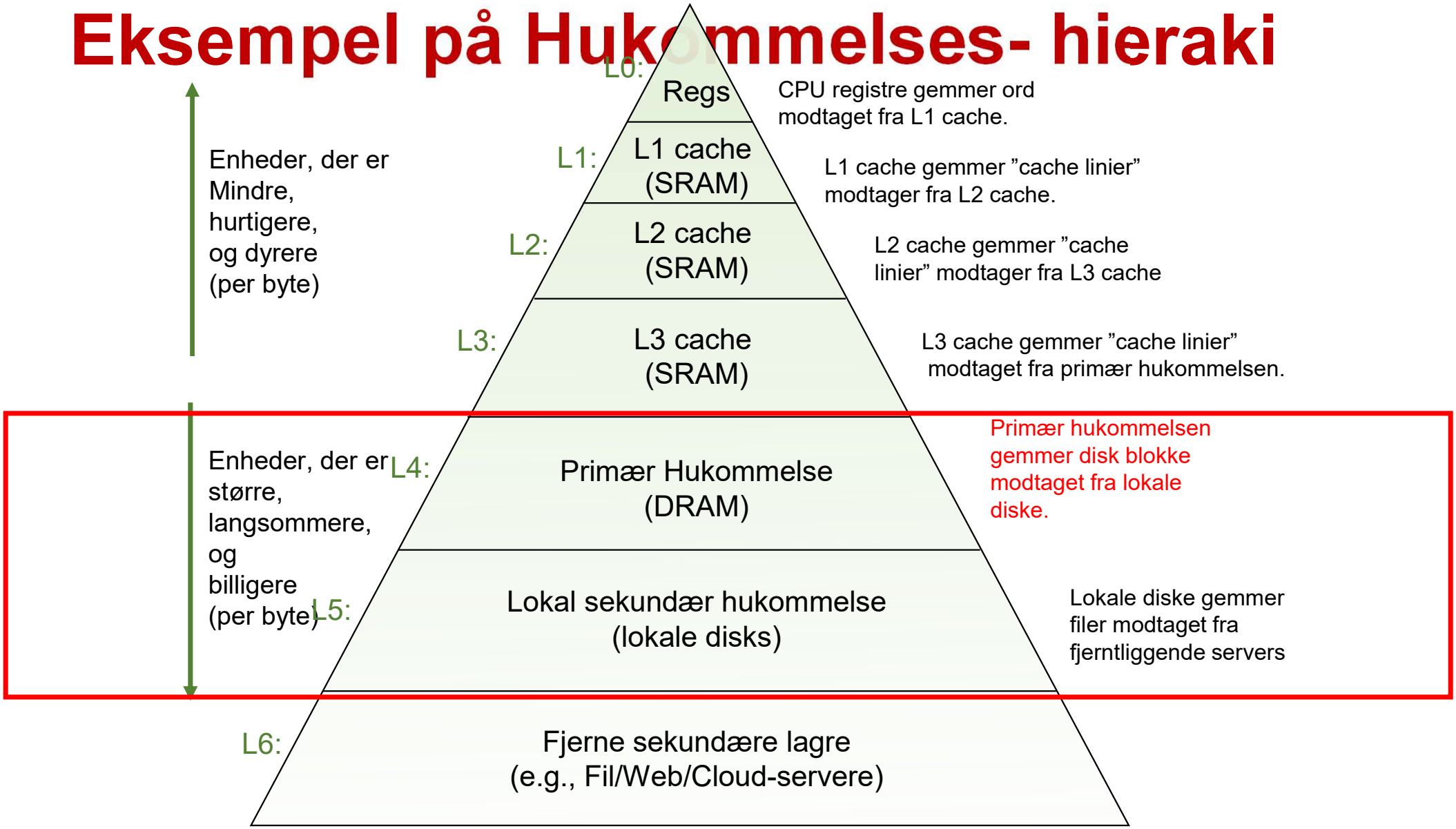
- Logisk opdeling af (både virtuel og fysisk) hukommelsen i mindre blokke af ens størrelse
  - **Virtuelle Sider** (Virtual pages)
  - **Fysiske Sider** (Physical pages) også kaldet frames
- Virtuelle sider kan frit indplaceres i en ledig fysisk side
- Eksempel størrelse = 4kB.

- Mekanismen giver 3 vigtige bidrag
  - Virtuel Stor Hukommelse (VM)
  - Nemmere og smart hukommelses administration
  - Beskyttelse



VM til caching

Eksempel på Hukommelses- hieraki

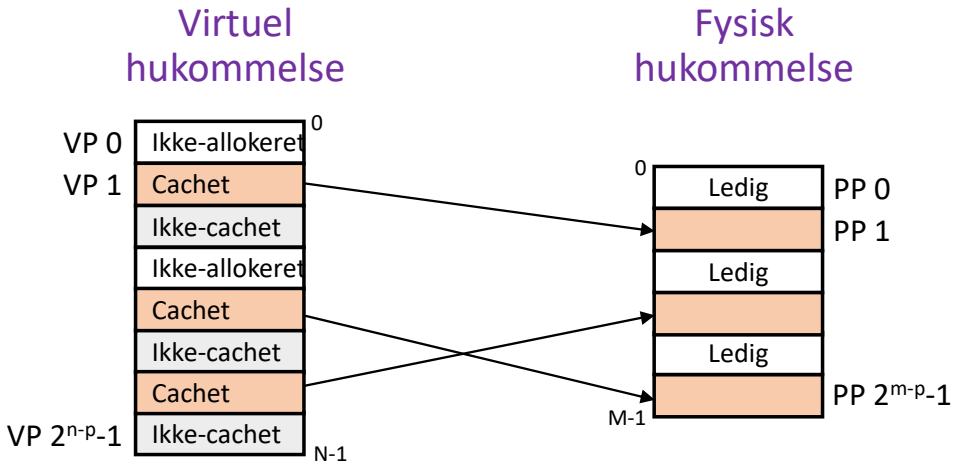


VM som caching mekanisme

- *Forståelsesmæssigt er Virtual hukommelse* er et array of N sammenhængende bytes gemt på disk.
- Indholdet caches i *fysisk hukommelse* (*DRAM cache*)
  - I VM, kaldes cachede blokke kaldes for sider (*pages)* (af størrelsen P = 2^p bytes)

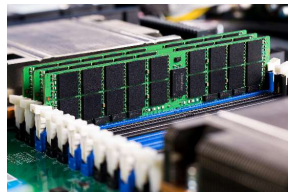
Fx side med p=12: P =2^{12} bytes=4 kiB

- En side kan være
  - Allokeret, i cache, eller
  - Allokeret, pt. ikke i cache, eller
  - Ikke allokeret



Virtuelle sider (VPs) gemt på sekundær hukommelse (disk)

Fysiske sider (PPs) cached i primær hukommelsen (DRAM)



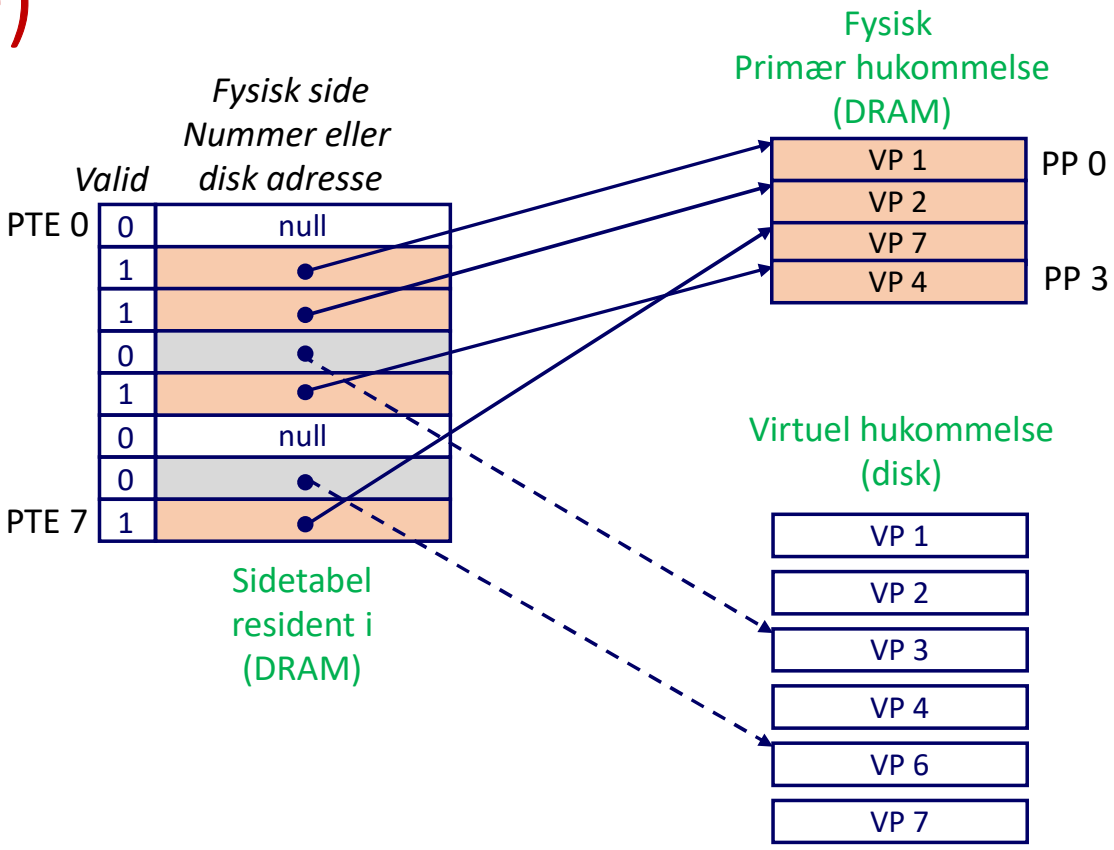
Opbygning af DRAM Cache

- Organisering og styring af DRAM cache bestemmes ud fra dens enorme pris for miss (penalty)
  - DRAM er ca. ***10x*** langsommere end SRAM
  - Disk er ca. ***10,000x*** langsommere end DRAM

- Konsekvenser
  - Relativt store sider (blokke) størrelse: typisk 4 kB, sommetider op til 4 MB
  - Fuldt associativ
    - Enhver VP kan indplaceres i enhver PP
    - Kræver "stor" funktion til oversættelsen – forskelligt fra andre caches
  - Ofring skal ske omhyggeligt: avancerede side-erstatningsalgoritmer
    - Styres i software (af OS): For kompliceret og for mange muligheder/politikker for at det kan laves i hardware
  - Bruger tilbageskrivningsprincip (Write-back), ikke gennemskrivning (write-through)

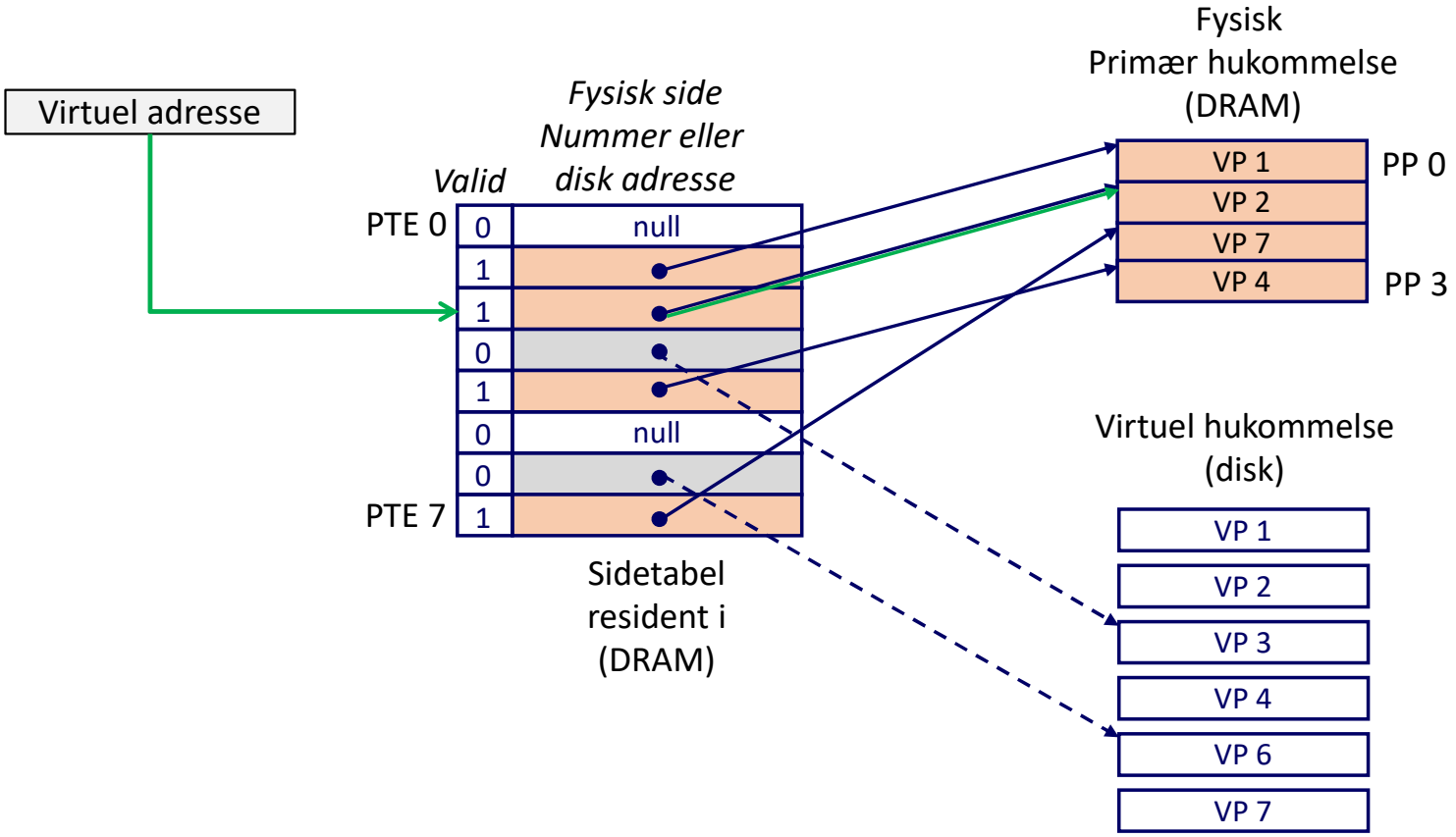
Data Struktur til VM: Sidetabel (Page Table)

- En sidetabel (*page table)* er et array af side-tabel indgange PTEs (page table entries) som mapper virtuelle sider til fysiske sider
  - Hver proces har sin egen sidetabel, gemt som Operativ System kerne data i DRAM
  - Udpeget i PCB
- Valid bit angiver om siden pt er cachet i DRAM eller ej



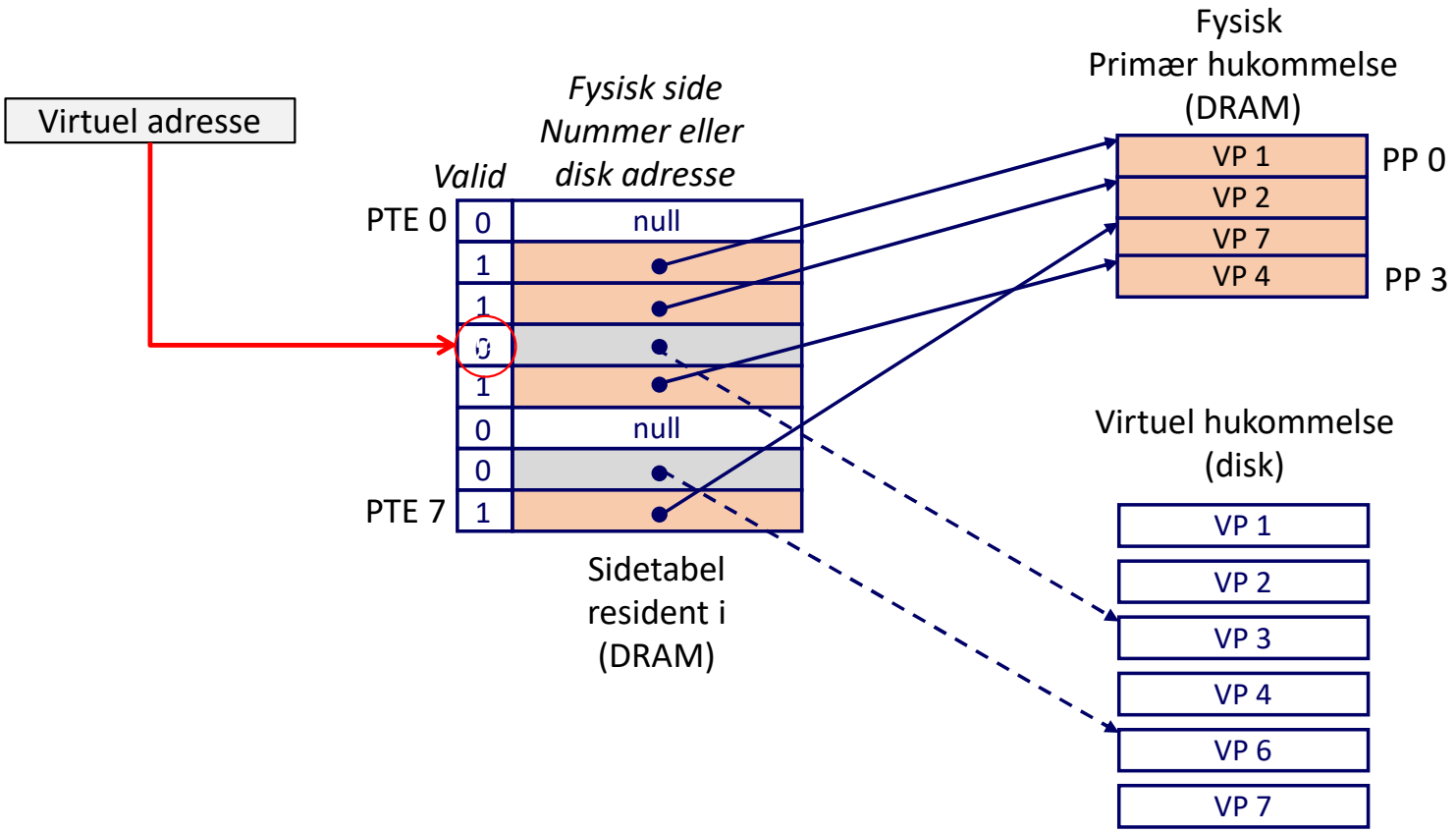
Side Hit (Page Hit)

- *Side Hit:* reference til ord i VM som er findes i fysisk hukommelse (DRAM cache hit)



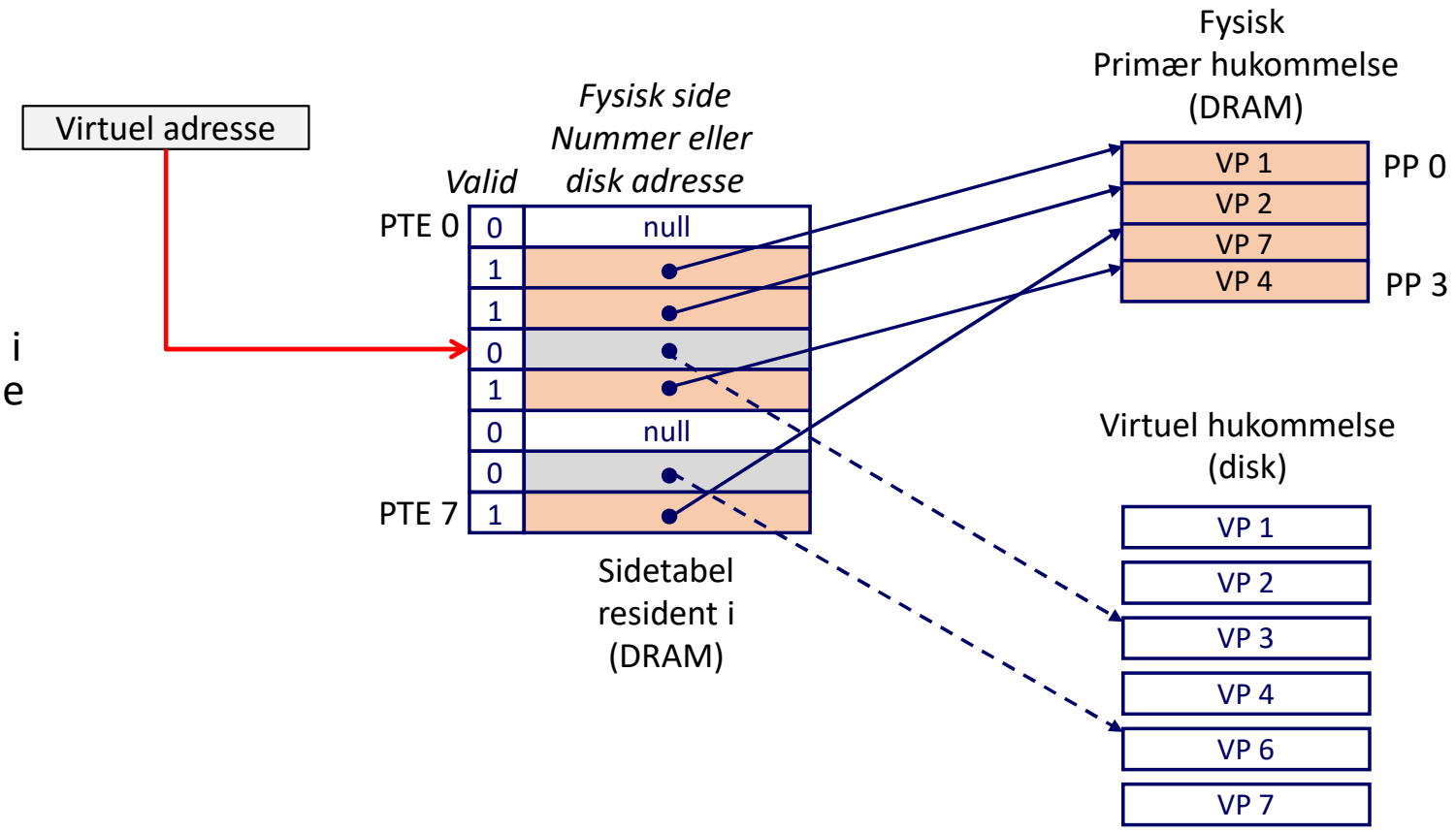
Side Fejl (Page Fault)

- *Side Fejl:* *Processor udsteder* reference til ord i VM som ikke findes i fysisk hukommelse (DRAM cache miss)



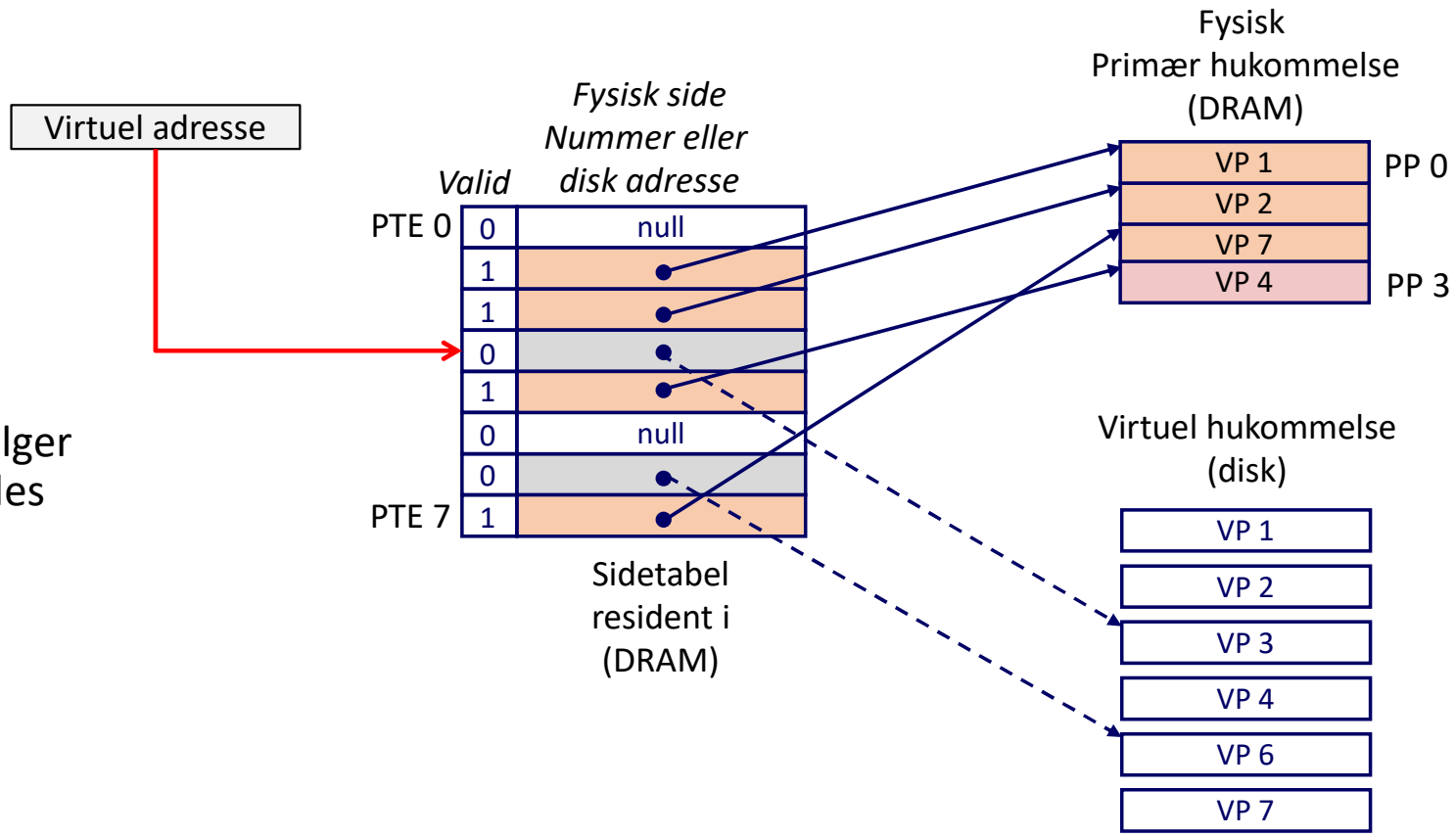
Håndtering af sidefejl

- Side-fejl forårsager en undtagelsessituation i processoren (exception)
- Vækker Operativ System (OS) i stedet for den aktuelt kørende proces



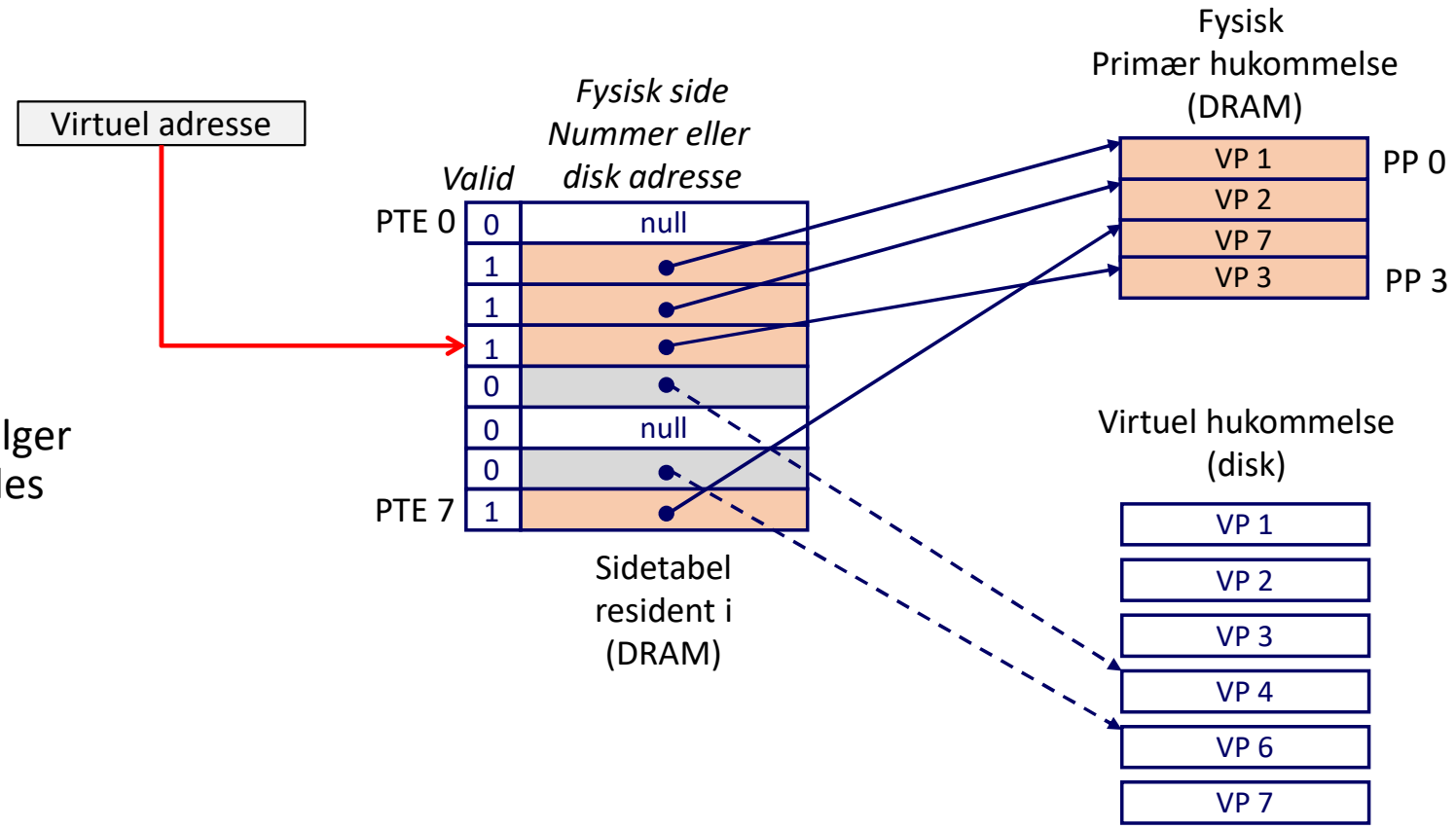
Håndtering af sidefejl

- Side-fejl forårsager en undtagelsessituation i processoren (exception)
- Vækker OS
- OS (page-fault-handler) udvælger et sig et ”offer” som skal smides ud (her VP 4)



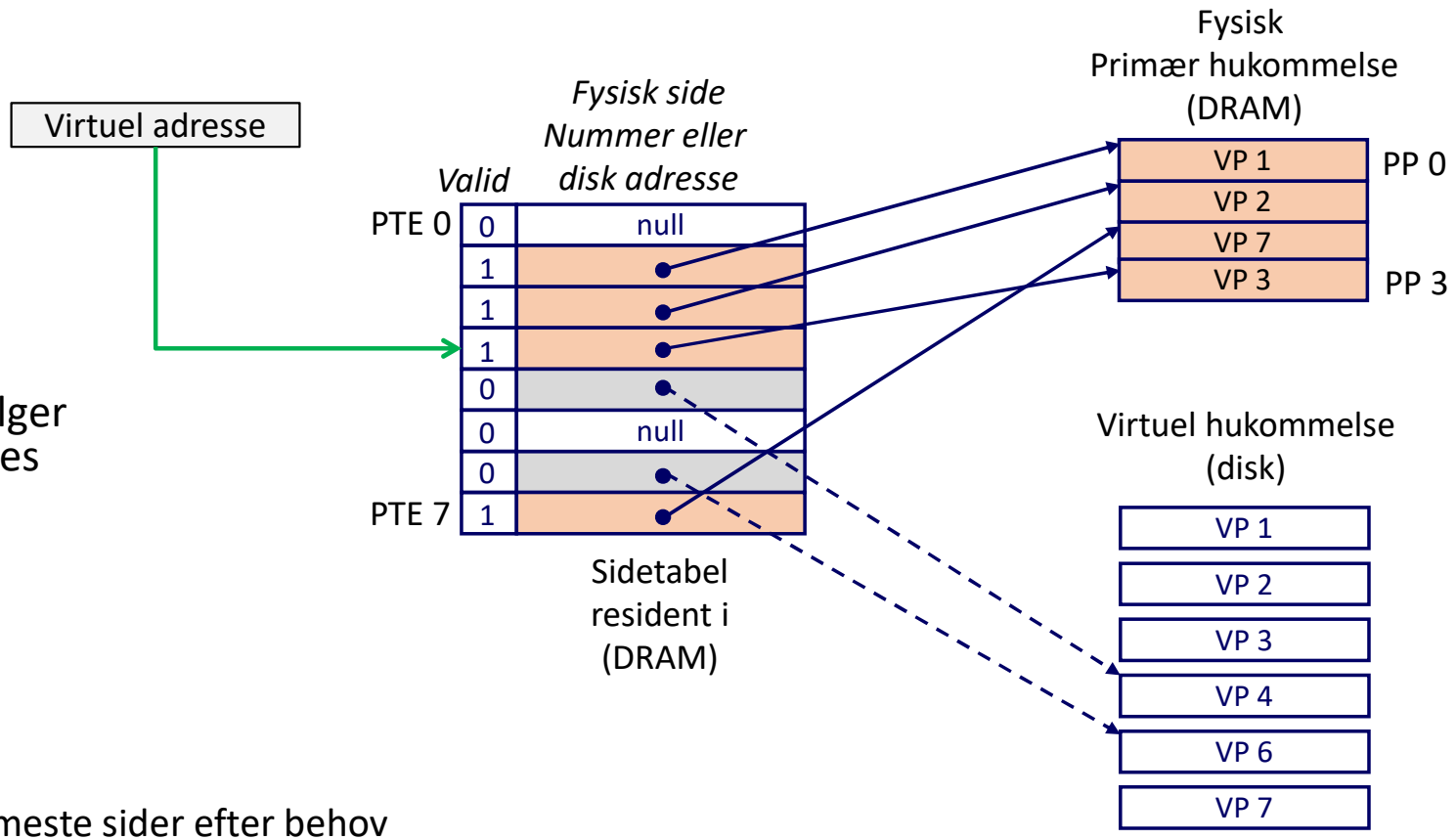
Håndtering af sidefejl

- Side-fejl forårsager en undtagelsessituation i processoren (exception)
- Vækker OS
- OS (page-fault-handler) udvælger et sig et ”offer” som skal smides ud (her VP 4)
  - Gemmer VP4, hvis modificeret
  - Indlæser VP 3



Håndtering af sidefejl

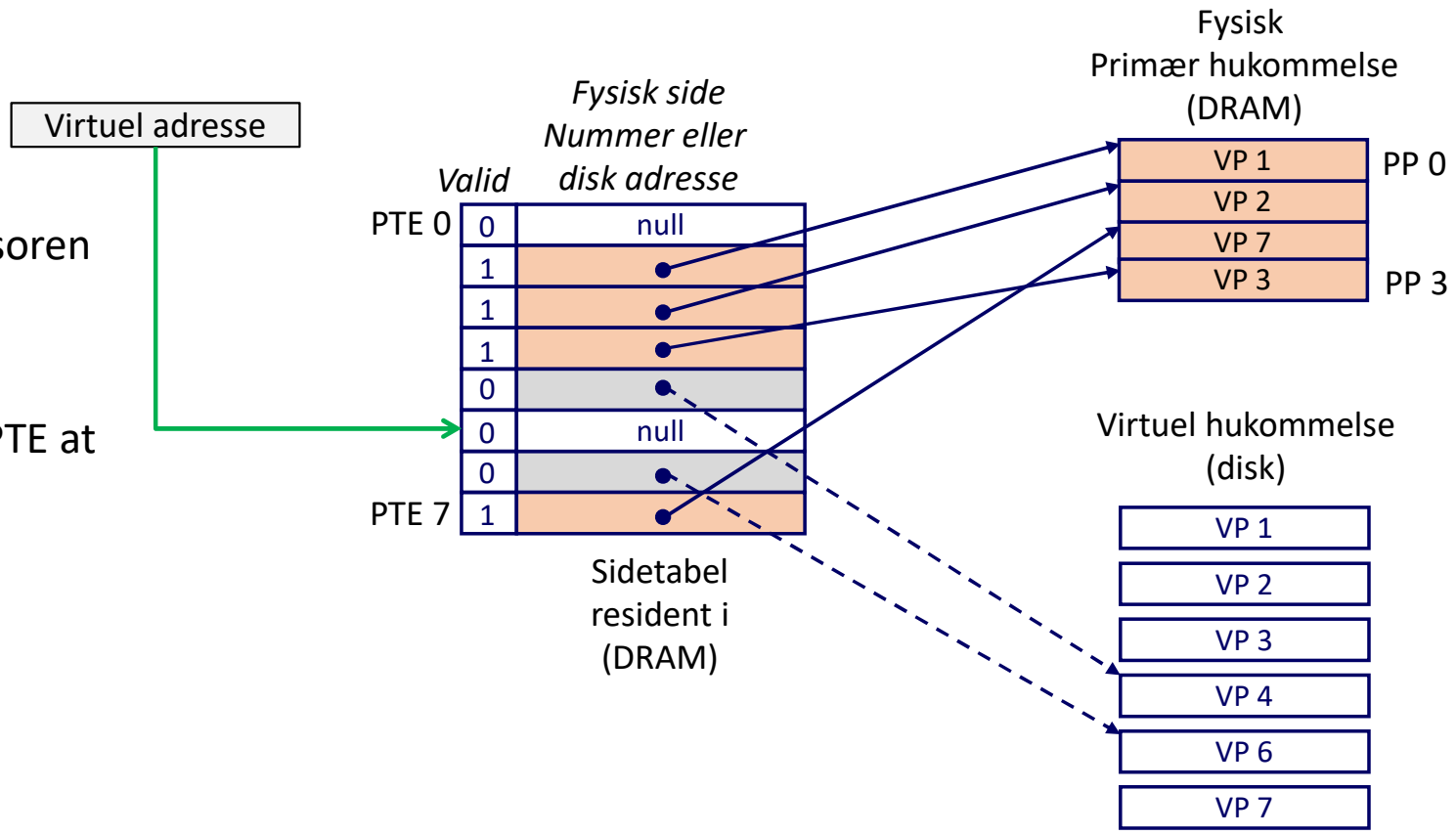
- Side-fejl forårsager en undtagelsessituation i processoren (exception)
- Vækker OS
- OS (page-fault-handler) udvælger et sig et ”offer” som skal smides ud (her VP 4)
  - Gemmer VP4, hvis modificeret
  - Indlæser VP 3
- Instruktionen genstartes



OBS: Systemet indlæser for det meste sider efter behov (ved side fejl) *demand paging*

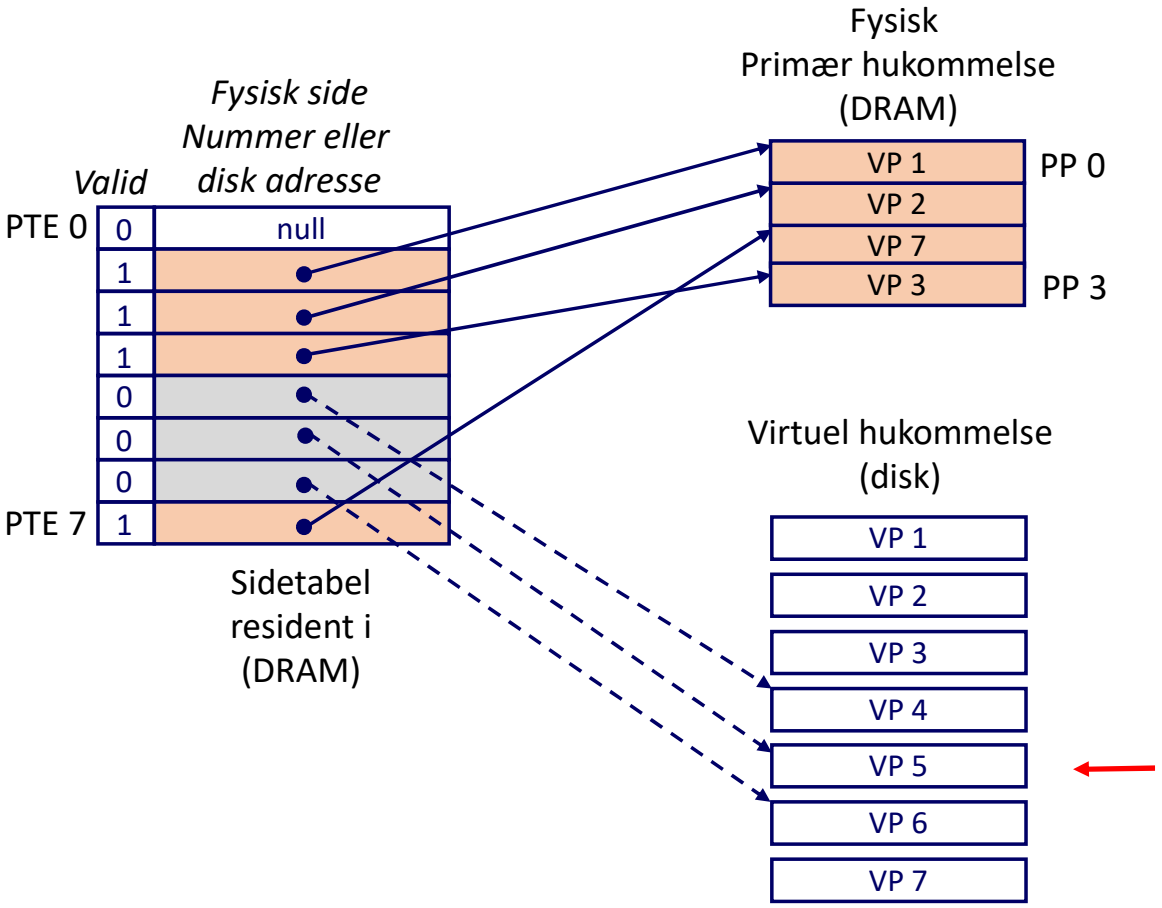
Håndtering af sidefejl: Uallokeret side

- Side-fejl forårsager en undtagelsessituation i processoren (exception)
- Vækker OS
- OS (page-fault-handler) ser i PTE at siden ikke er allokeret
- =>OS terminerer proces m. SEGMENTATION FAULT



Allokering af nye sider

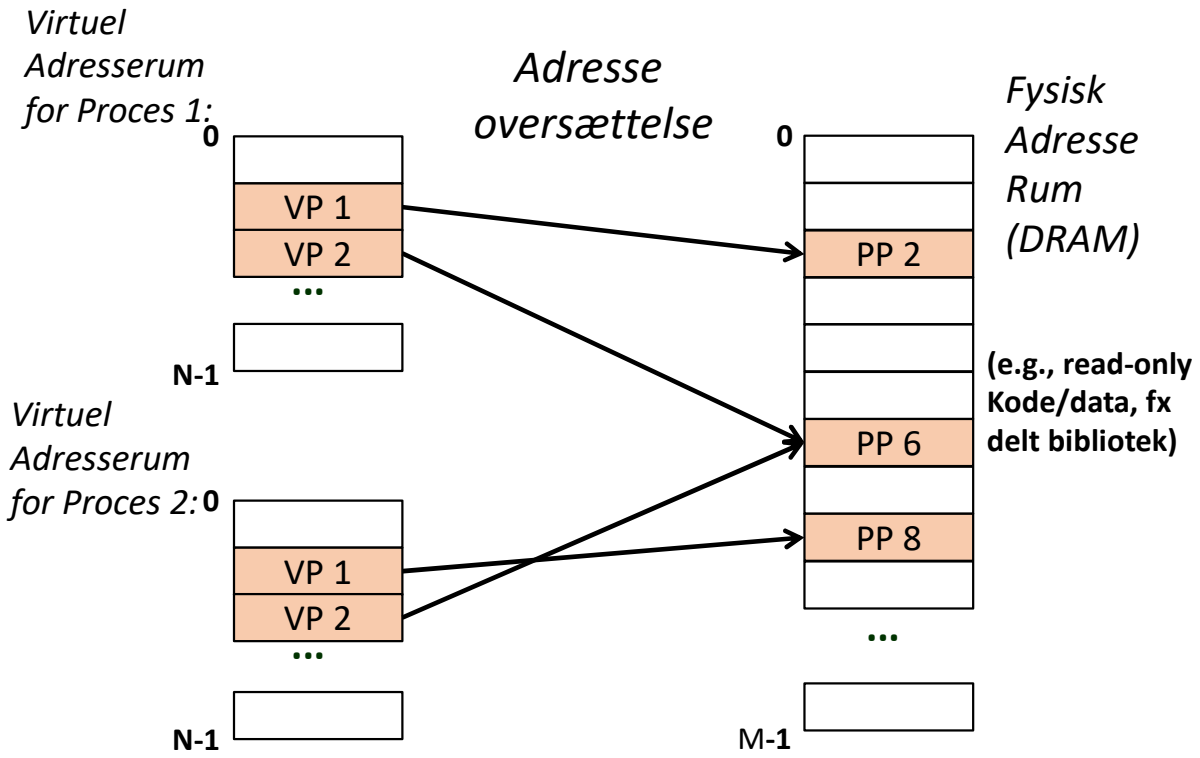
- Allokering af ny VM side (VP 5) .
  - Plads afsættes på disk
  - Findes stadig ikke i DRAM valid=0
  - Overføres først til DRAM når siden refereres



VM til hukommelses-administration

VM som redskab til hukommelses administration

- Simplificerer hukommelses allokering
  - Hver proces har sit eget virtuelle adresserum
    - Den kan betragte hukommelsen som et simpelt lineært array
    - Ensartet layout: code, data,stak,hob
  - Fuld fleksibel fysisk indplacering:
    - Hver virtuel side kan indplaceres i enhver fysisk side
    - En virtual side kan gemmes i forskellige fysiske sider fra gang til gang den kommer i cache
    - Ingen spild ”huller” i DRAM
- Deling af kode og data blandt processer
  - Mappe virtuelle sider til samme fysiske side (her: PP 6)
  - ”de-duplication”

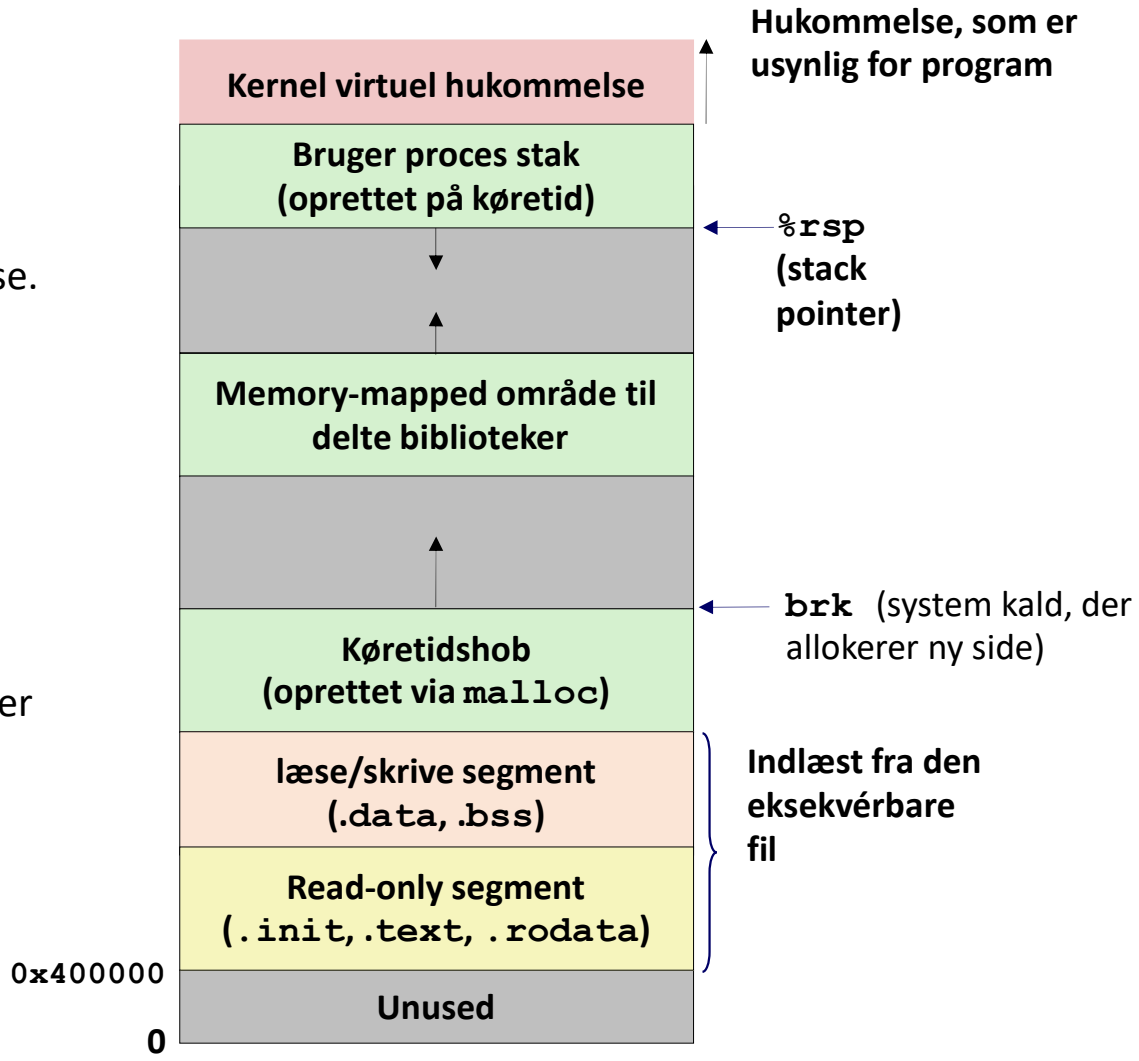


Simplificerer Linking og Loading

- Linking
  - Hvert program har ensartet virtuelt adresserum
  - Kode, Data, og Hob starter altid på samme adresse.

- Loading
  - **`execve`** (Linux funktion til opstart af nyt program) allokerer virtuelle sider til .text og .data områder og opretter PTEs markeret som invalid
    - PTEs for .text peger på filen
  - **`.text`** og **`.data områder`** kopieres side efter side, on-demand, af VM systemet

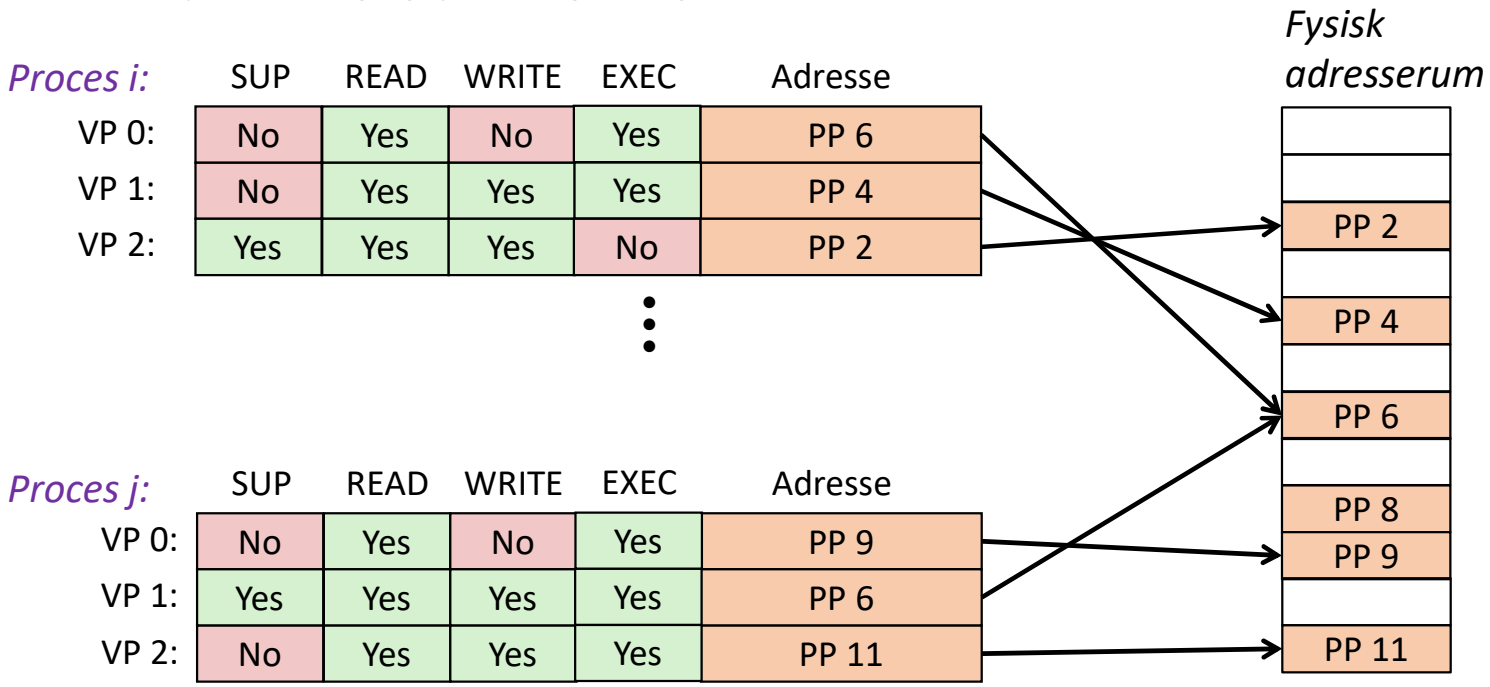
Uden VM skal compiler generere ”relokérbar” kode, og linker/loader indplacere i DRAM (typisk i simple indlejrede systemer)



VM til hukommelses-beskyttelse

VM redskab til beskyttelse af hukommelsen

- Udvid PTEs med bits til permissioner
  - Read, Write, Execute
  - SUP angiver SUPERVISOR mode: (kan kun tilgåes af OS kernen)
- MMU checker disse bits ved hver adgang
  - Genererer exception hvis adgang ej OK (kan give ”Segmentation Fault”)



Adresse-oversættelse

Adresse Oversættelse

- Virtuelt Adresserum
  - *V = {0, 1, …, N–1}*
- Fysisk Adresserum
  - *P = {0, 1, …, M–1}*
- Adresse Oversættelse
  - ***MAP: V → P U {∅}***
  - For virtual adresse ***a***:
    - ***MAP(a) = a'*** hvis data på virtuel adresse ***a*** findes på fysisk adresse ***a'*** i ***P***
    - ***MAP(a) =*** ∅ hvis data på virtuel adresse ***a*** ikke findes i fysisk hukommelse
      - Enten ugyldig eller gemt på disk

Fortolkning af adresser

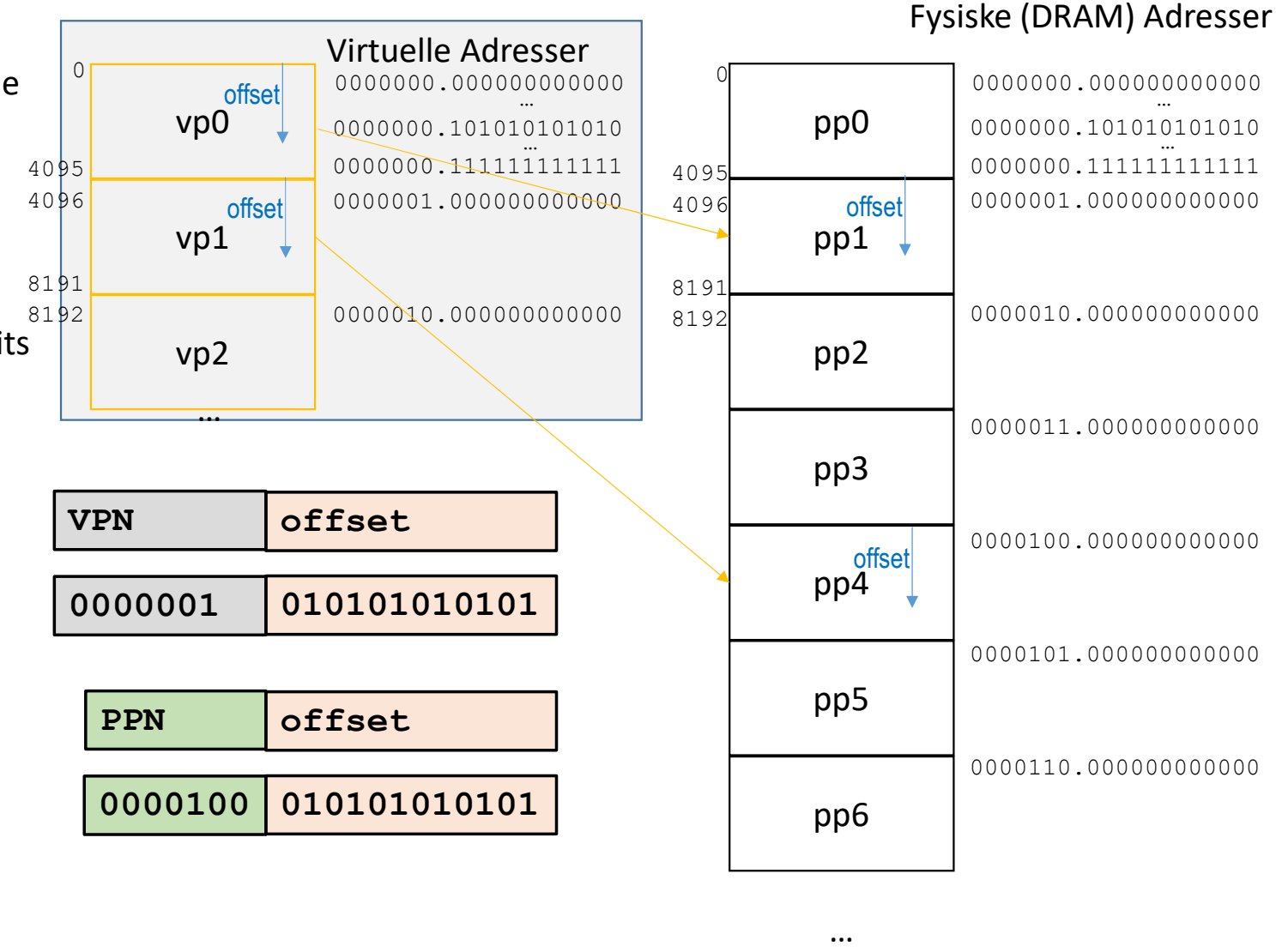
En adresse fortolkes som bestående af 2 dele:

- Side-nummer: mest betydende bits i adresse
- Offset: mindst betydende bits i adressen

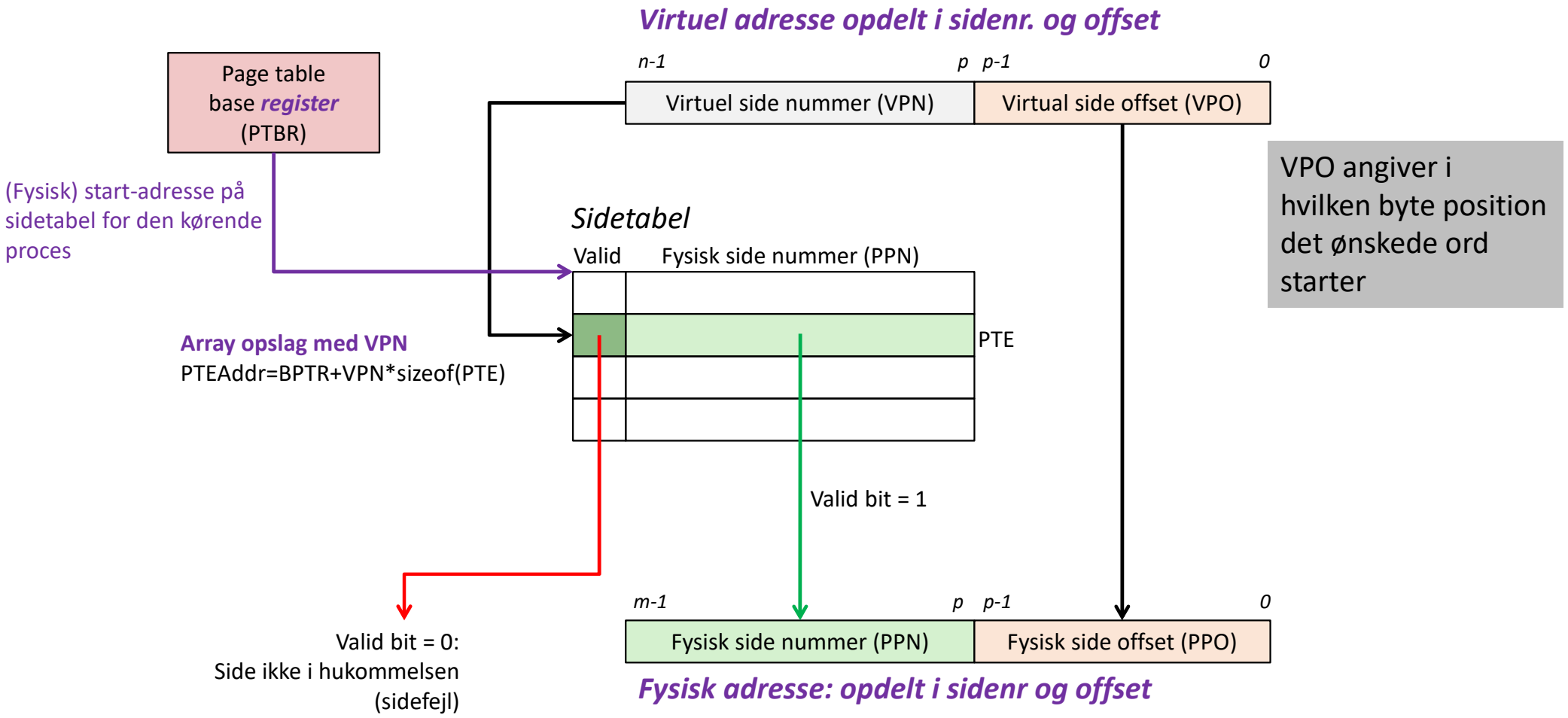
FX blok størrelse på (fx. 4kB = 12 bits til offset)

Virtuel adresse:
5461 =
Virtuel Side 1
Virtuel Side Offset: 1365

Fysisk adresse:
17749 =
Fysisk side 4,
Fysisk Side Offset: 1365

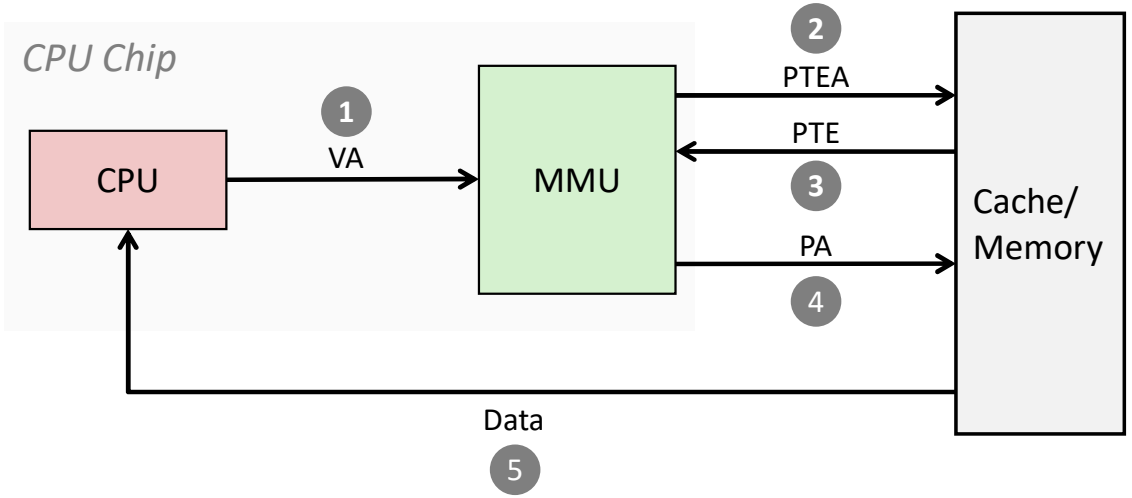


Adresse-oversættelse vha. lineær sidetabel



VPO angiver i hvilken byte position det ønskede ord starter

Adresseoversættelse: Hit



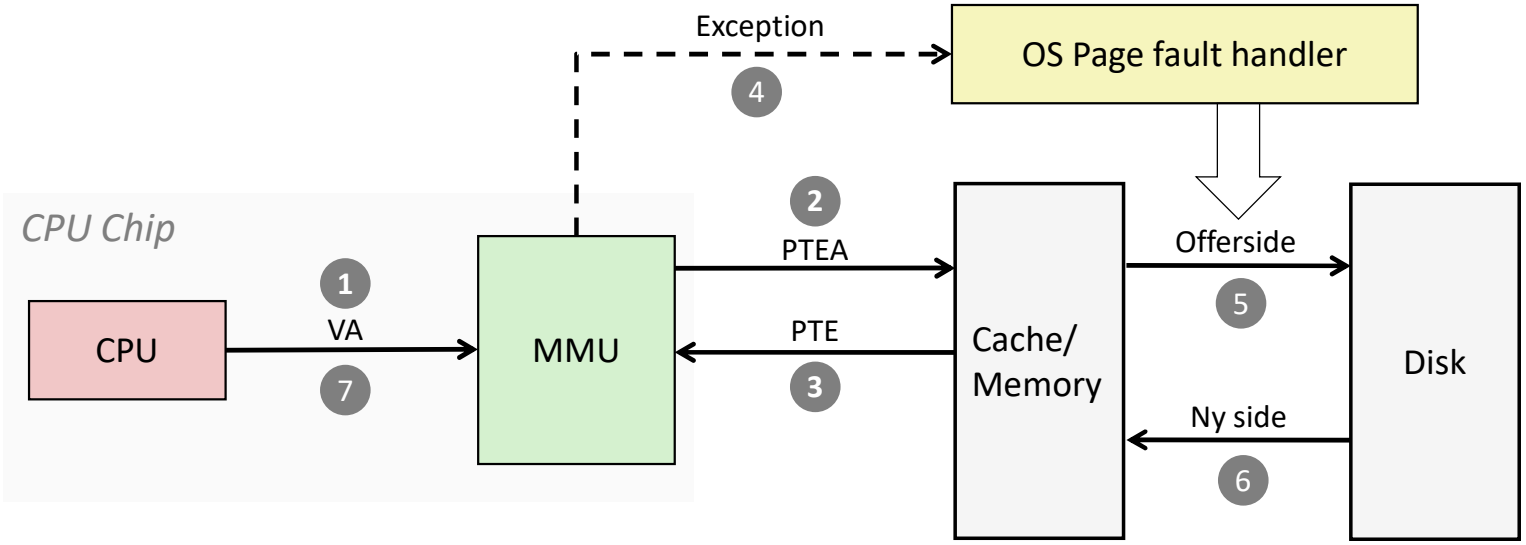
1) Processor udsteder virtuel adresse til MMU

2-3) MMU læste PTE fra sidetabel fra hukommelsen

4) MMU sender fysisk adresse til cache/hukommelsen

5) Cache/hukommelsen svarer med data-ord til processor

Adresse-oversættelse: Miss (Page Fault)



1) Processor udsteder virtuel adresse til MMU

2-3) MMU læste PTE fra sidetabel fra hukommelsen

4) Valid bit=0, så MMU trigger en sidefejlsundtagelse (page fault exception)

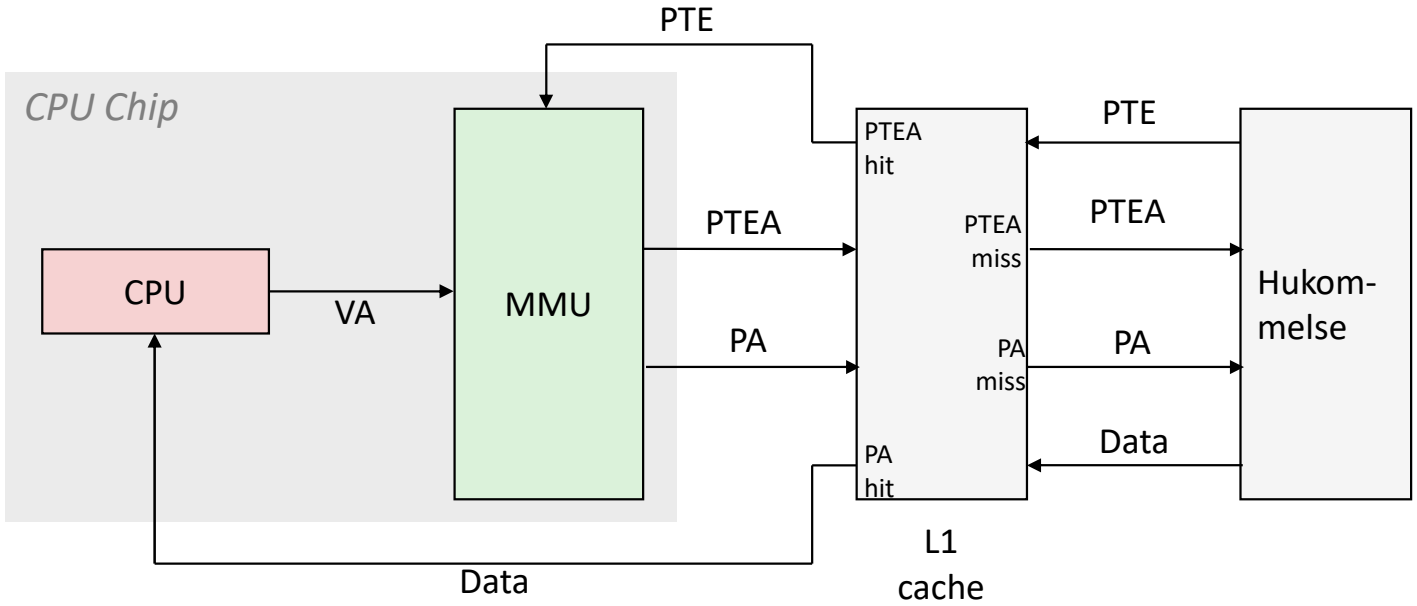
5) OS-handler identificerer offer (og, hvis modificeret (dirty), skriver den til disk)

6) OS-handler indlæser den anmodede side og opdaterer PTE

7) OS-handler returnerer til kørende proces, og processor genstarter den fejlgivende instruktion

Integration af VM og Cache

- Hvilke adresser bruges i L1-L3 cache? Virtuelle adresser eller Fysiske adresser??
  - Typisk fysiske adresser, så delt data mellem processer caches uden ekstra mekanismer
  - (Adgangskontrol sker som den del af adresseoversættelsen)



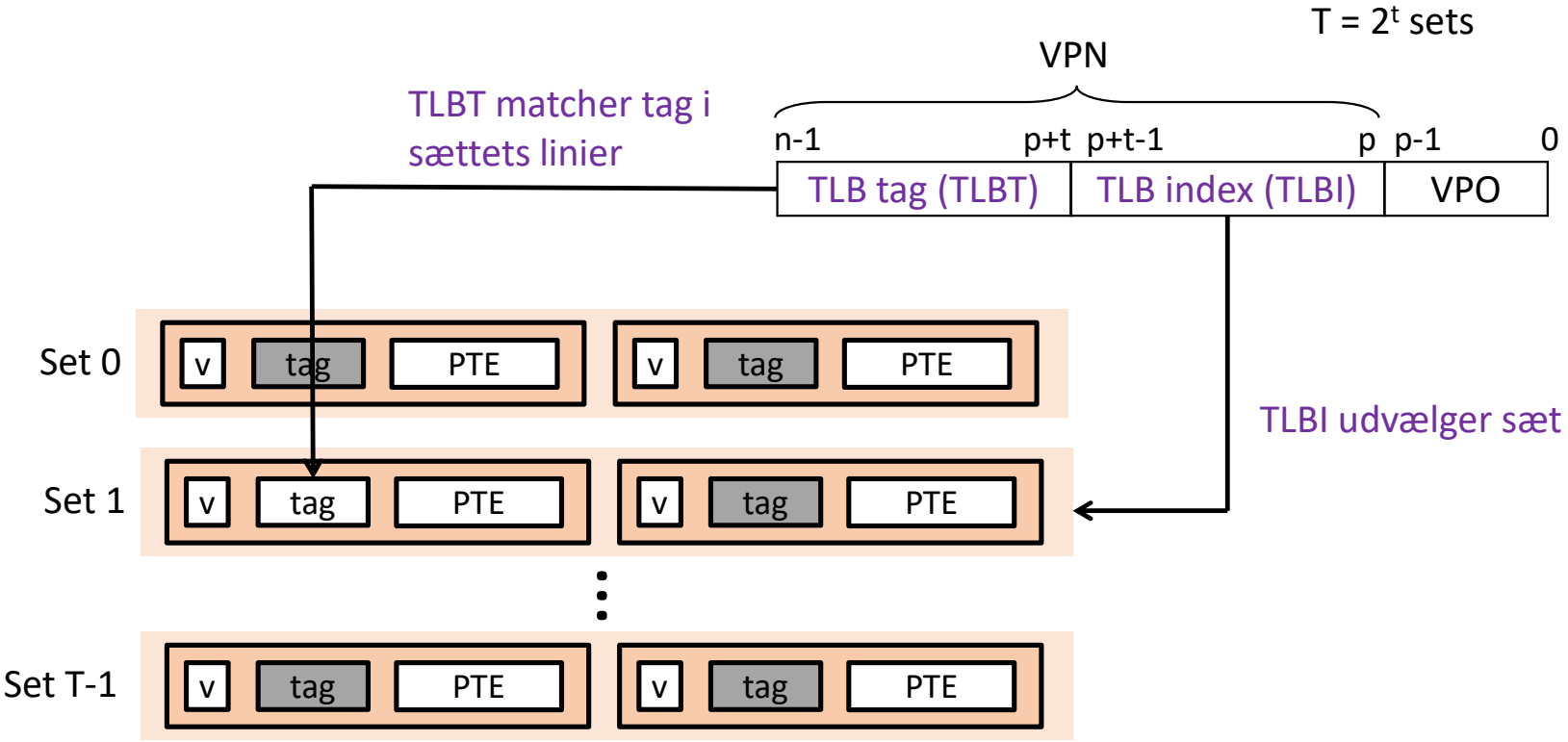
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Hurtigere oversættelse vha. ”TLB”

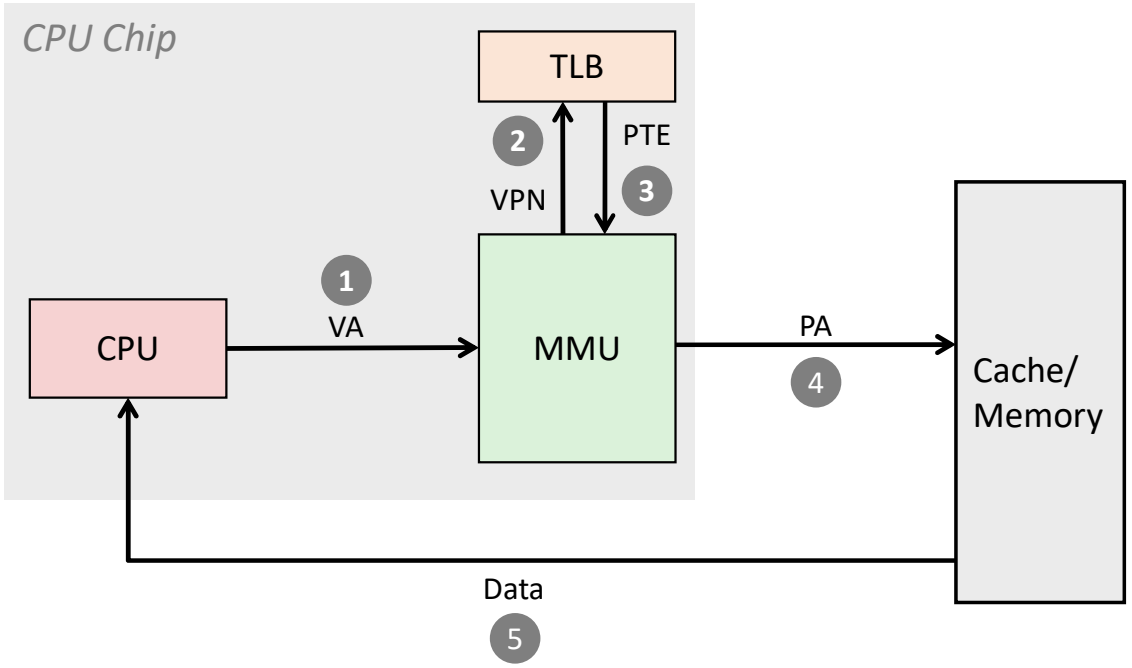
- Skrivning/læsning til hukommelse i VM kræver 2 adgange: utåleligt dyrt
  1. Opslag i sidetabel
  2. Læsning/skrivning af ordet.
- Sidetabel-indgange (PTEs) caches i L1 som enhver andet ord fra hukommelsen
  - PTEs kan blive ofret af andre data-referencer: dyrt L1 miss
  - PTE hit kræver stadig forsinkelse forårsaget af L1 cache
- Løsning: *Translation Lookaside Buffer* (TLB)
  - Lille dedikeret hardware cache internt i MMU:
  - Gemmer nylige oversættelser
  - Indeholder sidetabel indgange (PTEs) for et lille antal sider

Oversættelse vha. TLB

- MMU bruger VPN fra den virtuelle adresse til at slå op (associativt) i TLB cache:
- VPN opdeles i cache tag og indeks

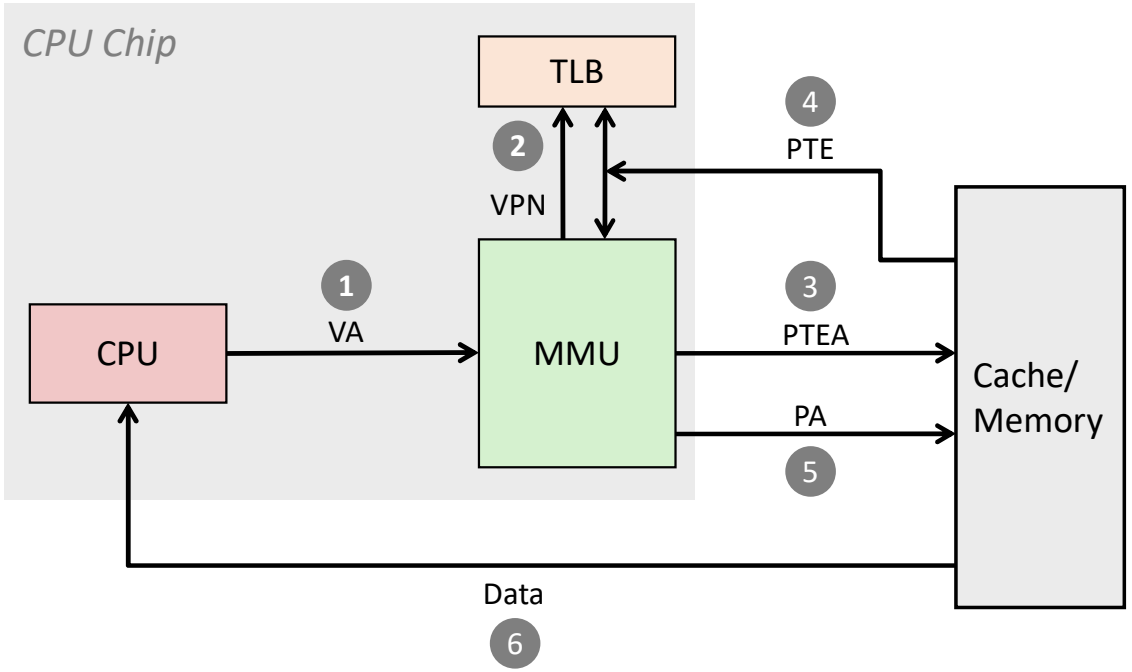


TLB Hit



A TLB hit undgår en hukommelsesadgang til sidetabel

TLB Miss



Et TLB miss forårsager en ekstra adgang til hukommelsen (opslag i PTE)

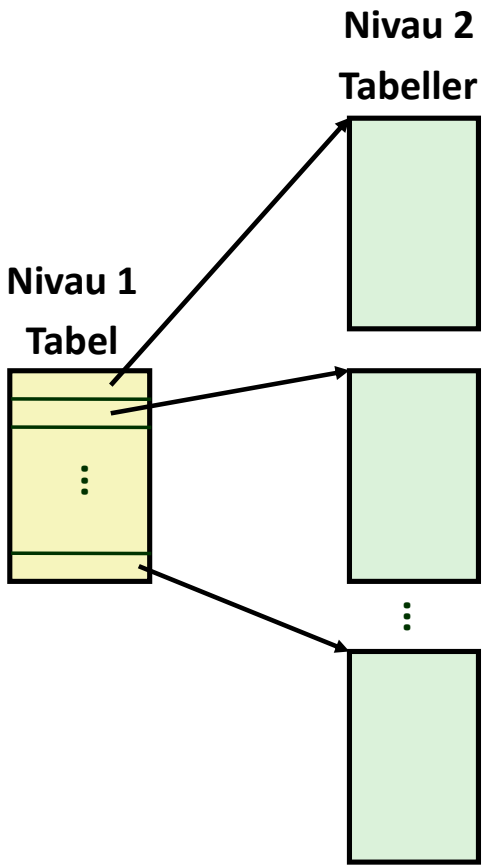
Heldigvis er TLB miss sjældent! Hvorfor???

Proces-skifte

- TLB indeholder information fra *virtuelle* adresser
  - Kun gyldige indenfor hver proces
  - Hvordan holdes virtuelle adresser fra forskellige processer adskilt?
- Ved proces-skifte
  - OS tømmer MMU TLB cache, eller
  - Nogle MMUer understøtter tagging med ekstra information fra “Address Space Identifiers”
    - OS tilknytter forskellig ASID til forskellige processer så de holdes adskilt af MMU
- OS indskriver også ”page table base register (PTBR)” ud fra ny proces’ PCB

Pladsforbrug af side-tabeller

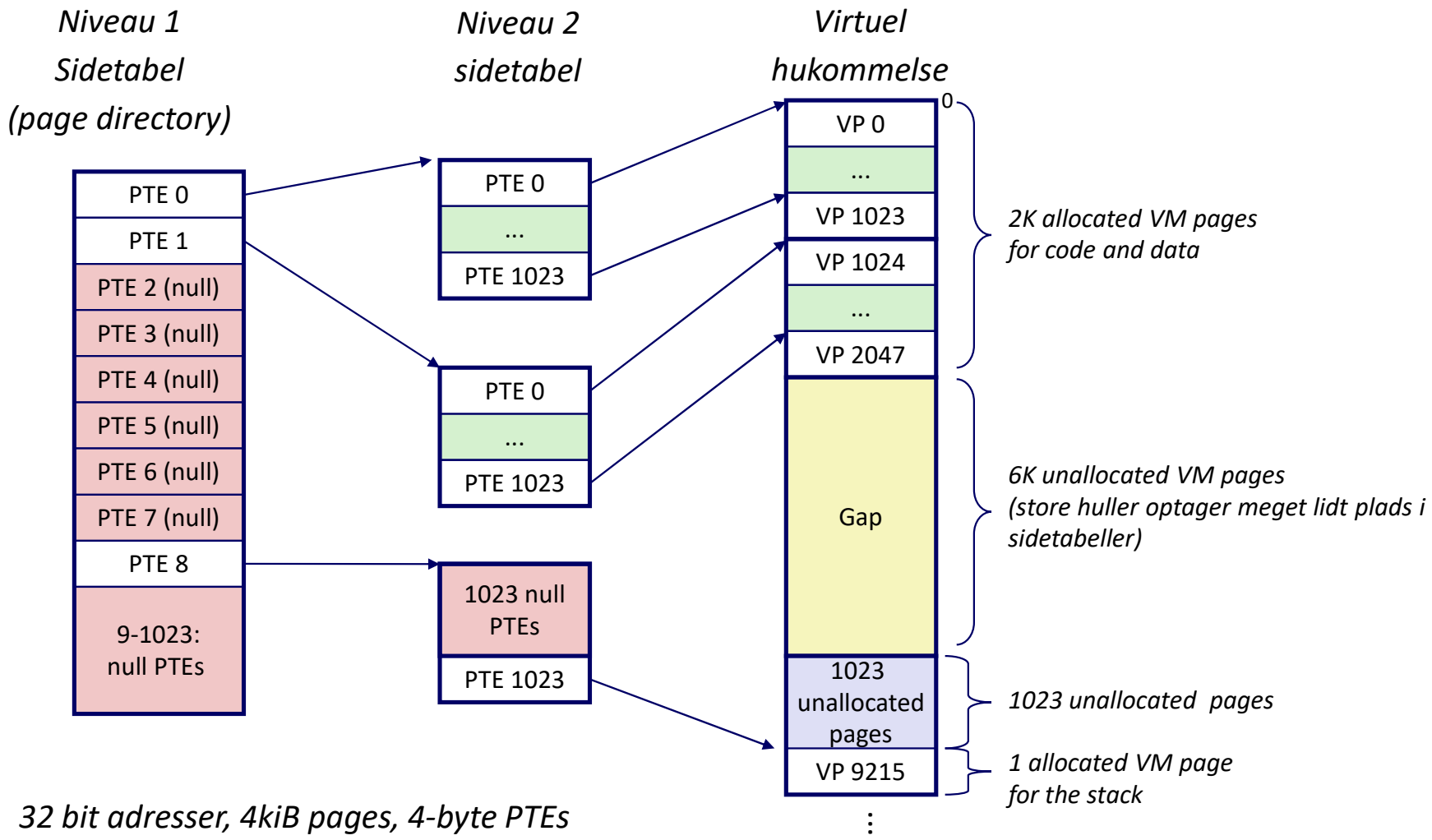
- Overslag over pladsforbrug
  - Antag:
    - 4KB (2^{12}) sidestørrelse, 48-bit adresserum, 8-byte PTE
  - Fx, Hvis vi vil kunne adressere alt vil det kræve 512 GiB DRAM til éen sidetabel!
    - $2^{48-12} * 2^3 = 2^{39}$ bytes
  - Fx Hvis vi vil have 100 processer a 4 GiB kørende vil det kræve ca. 1GB til sidetabeller
    - (4GiB/4KB)*8B * 100 processer = 800 MiB
- Løsning?: Større sider => Større intern fragmentering
- Almindelig løsning: Lav side-tabeller i flere niveauer
- Fx: 2-niveau sidetabel
  - Niveau 1 tabel: hver PTE peger på sidetabel (altid ***resident*** i hukommelsen)
  - Niveau 2 tabel: Hver PTE peger på en side (gemmes i ***virtuel*** hukommelse på samme på som alt anden data)



*) 1KB=1024 bytes

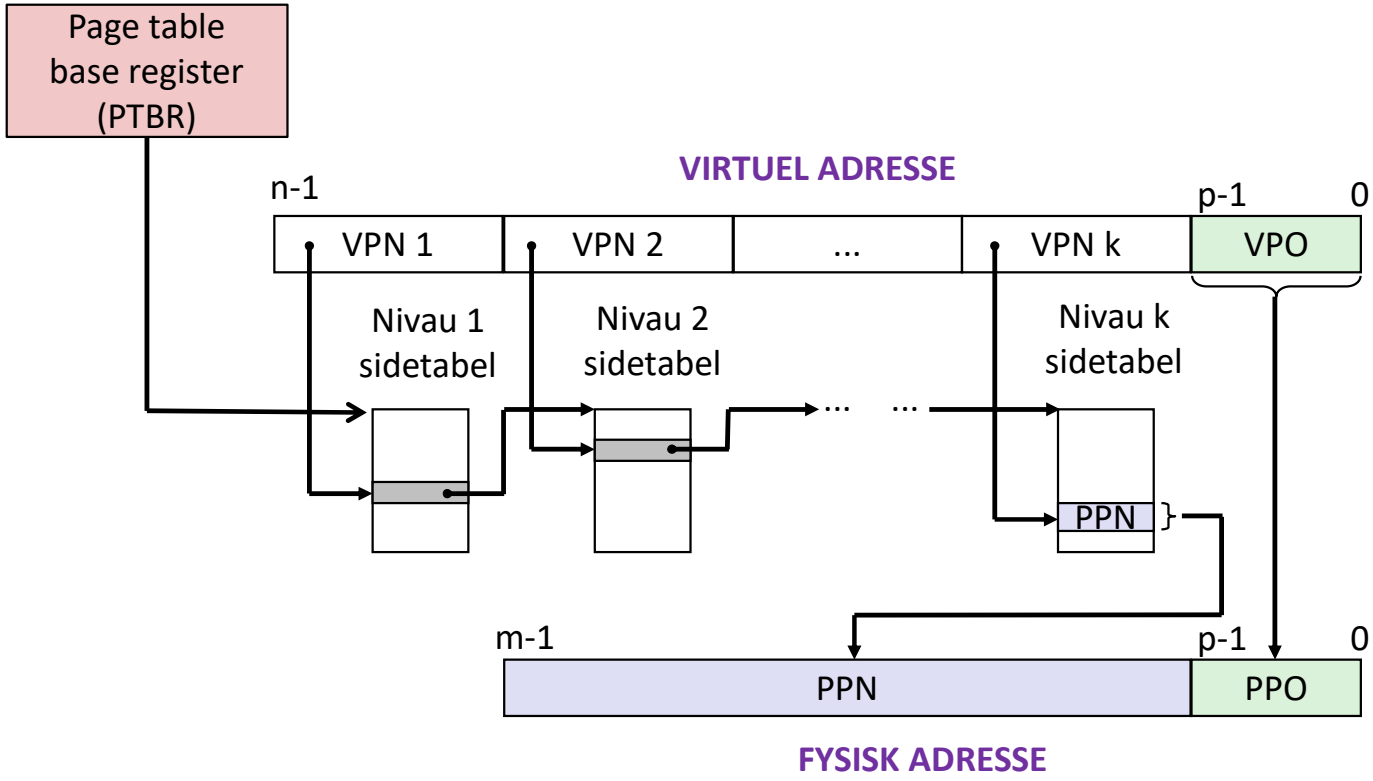
Sidetabeller i 2 niveauer

1024*1024*4096 bytes = 4GiB



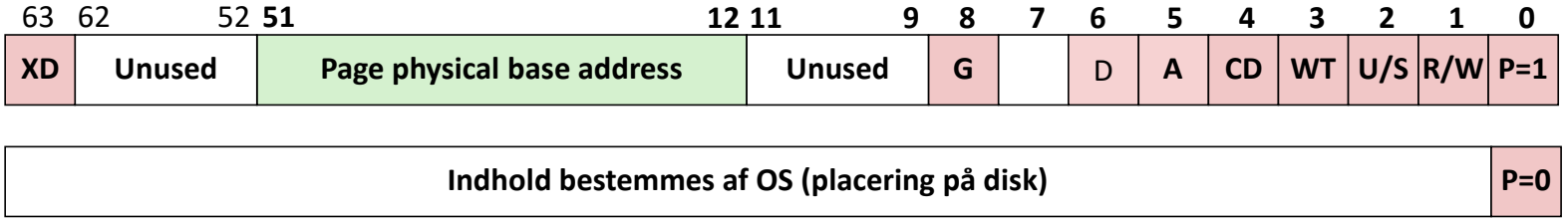
32 bit adresser, 4kiB pages, 4-byte PTEs

Adresse-oversættelse med k-niveau sidetabel



- Koster ind i mellem ekstra tid (flere hukommelsesadgange) til adresse-oversættelse (ved TLB miss), men sparer DRAM plads
- "Klassisk" tids/plads kompromis som ofte ses i algoritmer og datastrukturer

Information i sidetabel-indgang (PTE)



Core i7 Sidetabel-indgange for Niveau 4:

Page physical base address: **PFN**: 40 mest signifikante bits af (48-bit) fysisk side adresse (4k alignment af sider)

P: Present: Side findes i fysisk hukommelse (1) eller ej (0). (Valid bit)

A: Refereret bit / Used bit (sat a MMU ved læsning eller skrivning, nulstilles af OS)

D: Dirty/Modified bit (sat af MMU ved skrivning, nulstilles af OS når siden er gemt)

Beskyttelse

- R/W: Read-only eller read-write adgangs-permission
- U/S: User eller supervisor mode adgangs-permission
- XD: Tillad/forbyd indlæsning af instruktioner for sider som kan nås fra denne.

WT: Write-through eller write-back cache politik for denne side)

CD: Caching disabled

G Global page (behold i TLB ved process.skifte).

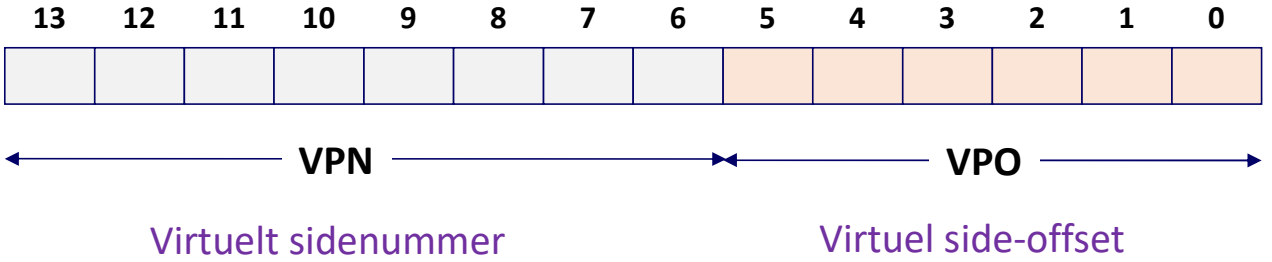
Lille VM eksempel

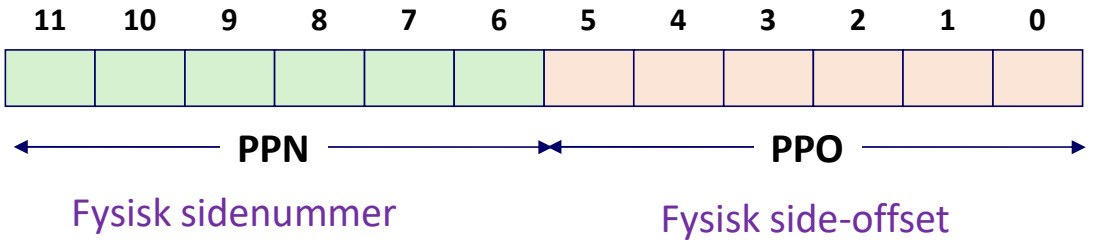
Symboler ved Adresseoversættelse

- Grundlæggende Parametre
  - **N = 2^n** : Antal adresser i det virtuelle adresserum
  - **M = 2^m** : Antal adresser i det fysiske adresserum
  - **P = 2^p** : Sidestørrelse (bytes)
- Opbygning af den virtuelle adresse (VA)
  - **VPO**: Virtuel side (page) offset
  - **VPN**: Virtual side (page) nummer
  - **TLBI**: TLB index
  - **TLBT**: TLB tag
- Opbygning af den fysiske adresse (PA)
  - **PPO**: Offset i den fysiske side (physical page offset): samme som VPO)
  - **PPN:** Fysisk side nummer (physical page no).

Eksempel på simpelt VM system

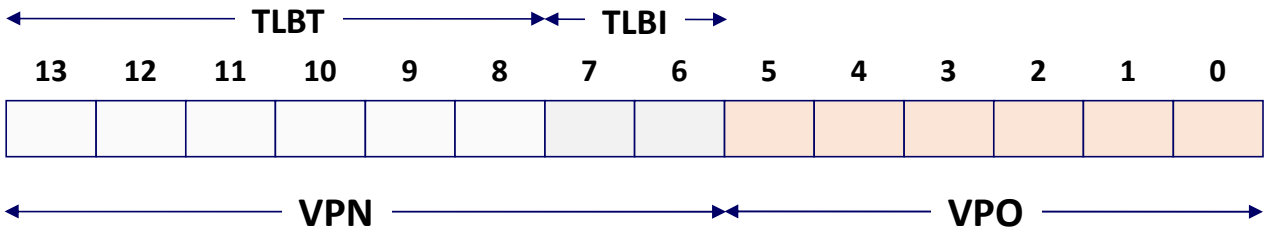
- Adressering
  - 14-bit virtuelle adresser
  - 12-bit fysiske adresser
  - Sidestørrelse = 64 bytes (6 bits til VPO), 256 sider (8 bits til VPN)





1. Simpel tilhørende TLB

- 16 indgange, 4-vejs associativ
- 4-sæt: sæt-indeks kræver 2 bits



Nb!
Hex Tal

Set	*Tag*	*PPN*	*Valid*	*Tag*	*PPN*	*Valid*	*Tag*	*PPN*	*Valid*	*Tag*	*PPN*	*Valid*
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

2. Simple tilhørende sidetabel

Eksemplet viser kun de første 16 indgange (ud af 256)

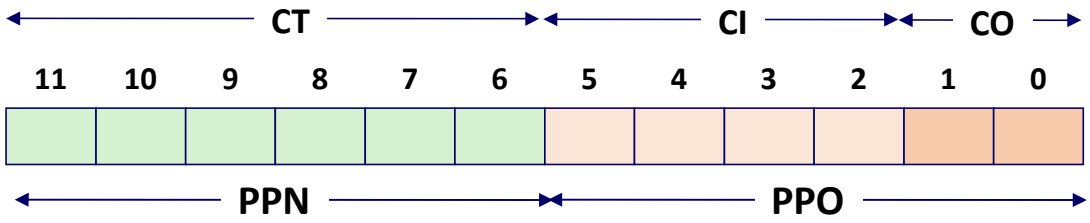
VPN	*PPN*	*Valid*
00	28	1
01	–	0
02	33	1
03	02	1
04	–	0
05	16	1
06	–	0
07	–	0

VPN	*PPN*	*Valid*
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	–	0
0D	2D	1
0E	11	1
0F	0D	1

Nb!
Hex Tal

3. Simpel tilhørende Cache

- Indekseres med fysiske adresser.
- Direkte koblet
- 16 sæt, 4-byte blokke: CI=4 bits, CO=2 bits



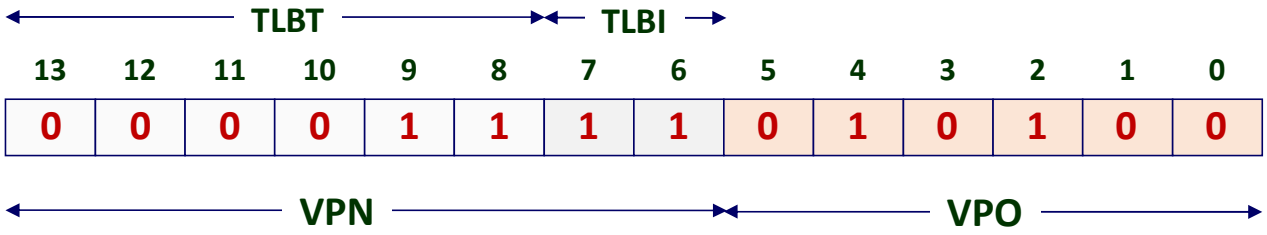
Nb!
Hex Tal

Idx	*Tag*	*Valid*	*B0*	*B1*	*B2*	*B3*
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

Idx	*Tag*	*Valid*	*B0*	*B1*	*B2*	*B3*
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

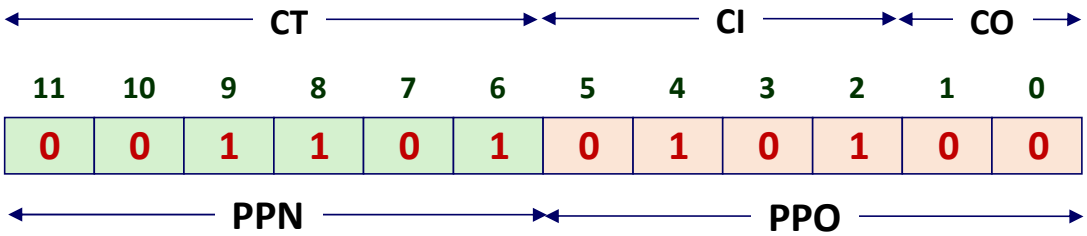
Eksempel #1 på adresse-oversættelse

Virtuel Adresse: `0x03D4`



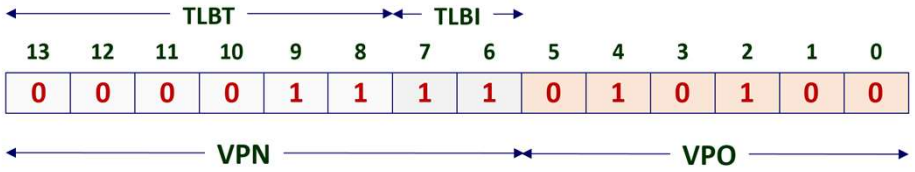
VPN **0x0F** TLBI **0x3** TLBT **0x03** TLB Hit? **Y** Page Fault? **N** PPN: **0x0D**

Fysisk Adresse:



CO **0** CI **0x5** CT **0x0D** Hit? **Y** Byte: **0x36**

Virt. Adresse



VPN 0x0F TLBI 0x3 TLBT 0x03 TLB Hit? Y Page Fault? N PPN: 0x0D

TLB

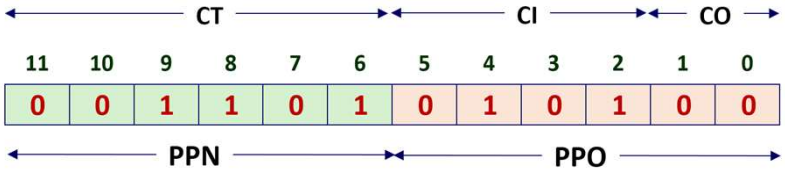
Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

Sidetabel

VPN	PPN	Valid
00	28	1
01	–	0
02	33	1
03	02	1
04	–	0
05	16	1
06	–	0
07	–	0

VPN	PPN	Valid
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	–	0
0D	2D	1
0E	11	1
0F	0D	1

Fys. Adresse



CO 0 CI 0x5 CT 0x0D Hit? Y Byte: 0x36

L1-cache

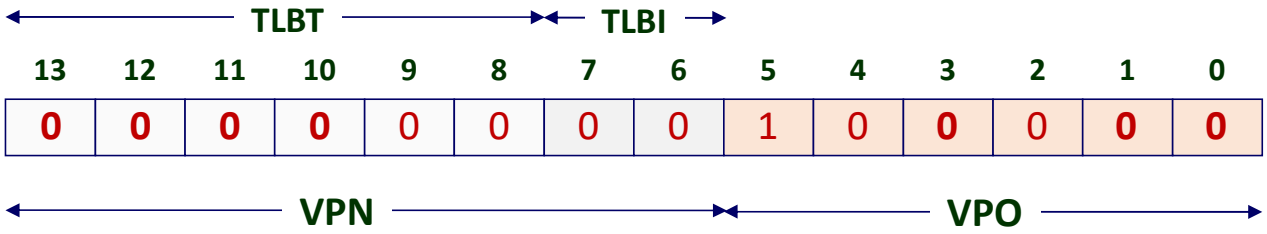
Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

Idx	Tag	Valid	B0	B1	B2	B3
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

TLB HIT,
L1 HIT

Eksempel #2 på adresse-oversættelse

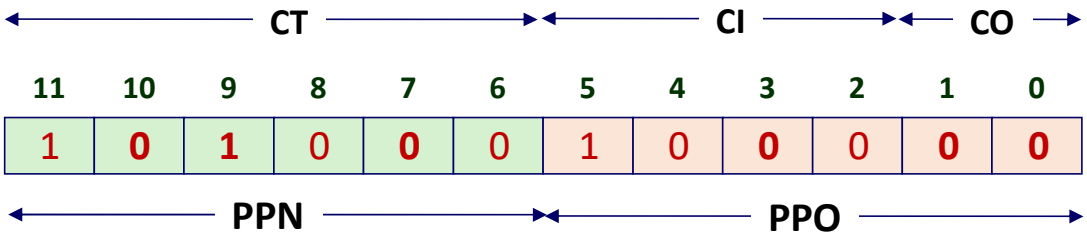
Virtuel Adresse: `0x0020`



VPN **0x00** TLBI **0** TLBT **0x00** TLB Hit? **N** Page Fault? **N** PPN: **0x28**

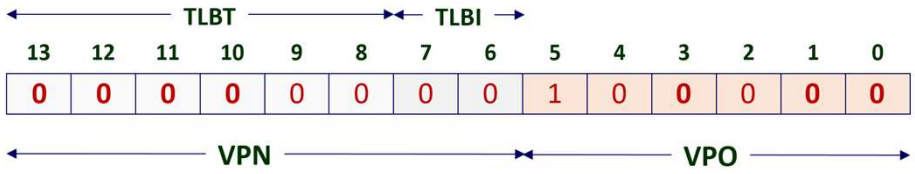
→ Brug sidetabel

Fysisk Adresse



CO **0** CI **0x8** CT **0x28** Hit? **N** Byte: **Mem**

Virt. Adresse

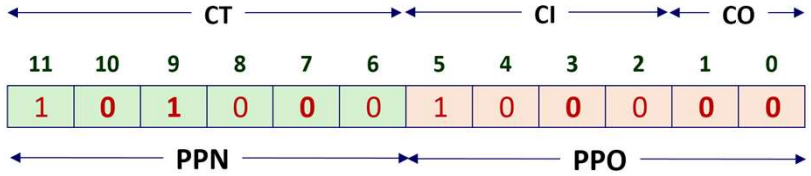


VPN 0x00 TLBI 0 TLBT 0x00 TLB Hit? N Page Fault? N PPN: 0x28

TLB

Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

Fys. Adresse



CO 0 CI 0x8 CT 0x28 Hit? N Byte: Mem

Sidetabel

VPN	PPN	Valid
00	28	1
01	–	0
02	33	1
03	02	1
04	–	0
05	16	1
06	–	0
07	–	0

VPN	PPN	Valid
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	–	0
0D	2D	1
0E	11	1
0F	0D	1

L1-cache

Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

Idx	Tag	Valid	B0	B1	B2	B3
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

TLB MISS,
PT valid
L1 MISS

Side-erstatningspolitikker

Problemstilling

- VM cache miss er ekstremt dyrt: involverer disk adgange (> *1000 langsommere)
  - Så få cache-misses som muligt
- Når systemet løber tør for fysiske rammer må det ofre nogle cachede sider
  - cache eviction
  - Foretages af OS
- Hvilken ofrings-strategi skal vi vælge?
  - Tilfældig?
  - Ældst (FIFO)?
  - Lær fra historik:
    - Nyhedsværdi: Least-Recently-used
    - Hyppighed: Least-Frequently-Used
  - Findes en optimal strategi?

- Da det er så dyrt kan det (måske) betale sig at være mere snedig i strategi end HW L1_l3 caches har

Belady's OPTimal

- Offer den side som skal bruges længst ude i fremtiden
- Eksempel med
  - Reference streng (work-load)
  - Cache med 3 fysiske sider
- Belady's algoritme giver beviseligt færrest mulige antal misses, men OS kender ikke fremtiden?
- Bruges som udgangspunkt (baseline) ved sammenligning med realisérbare politikker

Sekvens af side referencer

0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		0,1,2
1	Hit		0,1,2
3	Miss	2	0,1,3
0	Hit		0,1,3
3	Hit		0,1,3
1	Hit		0,1,3
2	Miss	3	0,1,2
1	Hit		0,1,2

6 hits, 2 misses (excl. koldstart)

FIFO

- Offer den ældste side
- Simpel implementation: en liste af sider, som opdateres når ny side indlæses.
- Tager ikke hensyn til fx at side 0 bliver brugt gentagende gange

Sekvens af side referencer

0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		0,1,2
1	Hit		0,1,2
3	Miss	0	1,2,3
0	Miss	1	2,3,0
3	Hit		2,3,0
1	Miss		3,0,1
2	Miss	3	0,1,2
1	Hit		0,1,2

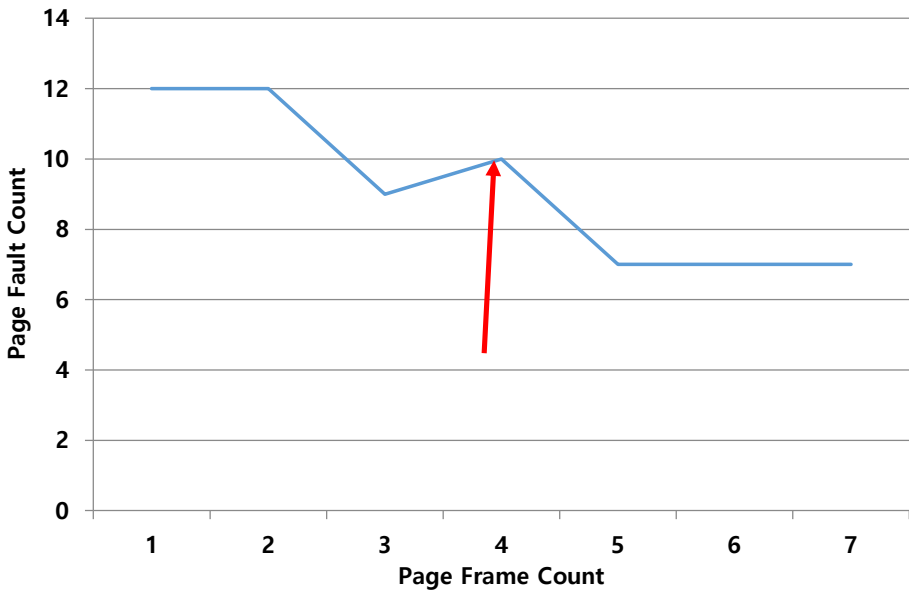
5 hits, 3 misses (excl. koldstart)

FIFO – Belady's anomalitet

- Vi vil normalt forvente at større caches mindsker misraten!
- Det gør FIFO ikke nødvendigvis

Sekvens af side referencer

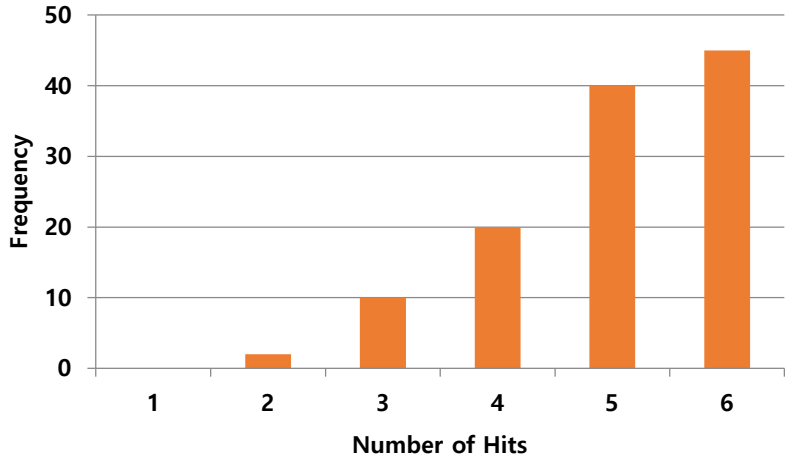
1 2 3 4 1 2 5 1 2 3 4



Cache størrelse (antal fysiske sider)

Randomiseret udvælgelse

- Tager en tilfældig (altså vha. tilfældigheds- generator)
- Performance afhænger af held
  - Nogle gange lige så god som OPT!



Sekvens af side referencer

0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		0,1,2
1	Hit		0,1,2
3	Miss	0	1,2,3
0	Miss	1	2,3,0
3	Hit		2,3,0
1	Miss	3	2,0,1
2	Hit		2,0,1
1	Hit		2,0,1

5 hits, 3 misses (excl. koldstart)

LRU – Least recently used

- Den ”mindst-fornyligt-brugte” side ofres
- Ide: Temporal lokalitet!
- Sider som vi har brugt for nyligt skal formentligt bruges igen i nærmeste fremtid!

Sekvens af side referencer

0 1 2 0 1 3 0 3 1 2 1

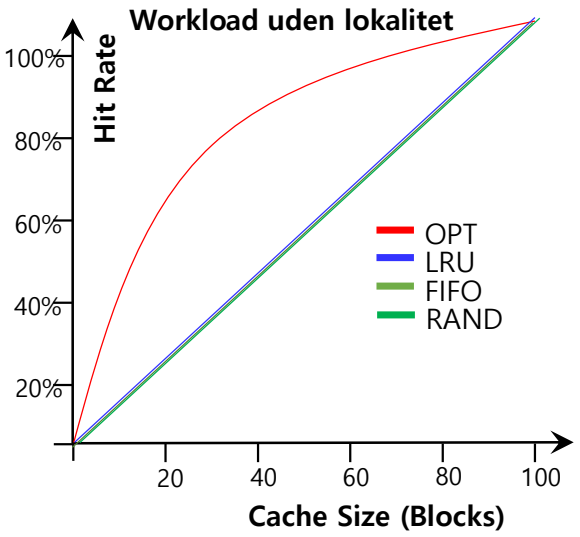
Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		1,2,0
1	Hit		2,0,1
3	Miss	2	0,1,3
0	Hit		1,3,0
3	Hit		1,0,3
1	Hit		0,3,1
2	Miss	0	3,1,2
1	Hit		3,2,1

6 hits, 2 misses (excl. koldstart)

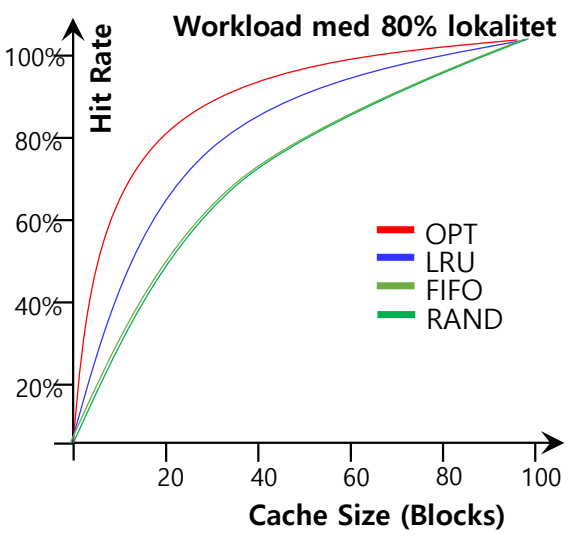
Arbejdsbelastning (workload) - simulering

100 forskellige sider refereres efter forskelligt mønster (workload), 10000 gange ialt.

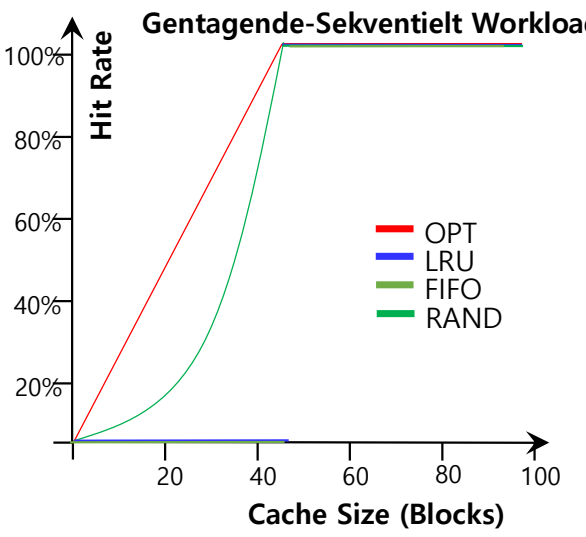
1. **Uden-lokalitet:** næste side som skal tilgås udvælges tilfældigt
2. **80-20 lokalitet:** 80% af referencerne laves til 20% af siderne: de resterende 20% henvisninger går til de resterende 80% af siderne
3. **Gentagende sekventielt** gennemløb: 0,1,3,4,…49,0,1,2,3,4, …



- Når cache stor: politik ligegyldig
- Ingen signifikant forskel på heuristiske politikker



- LRU nærmer sig bedre OPT end øvrige
- Bedre til at beholde de ”varme” sider



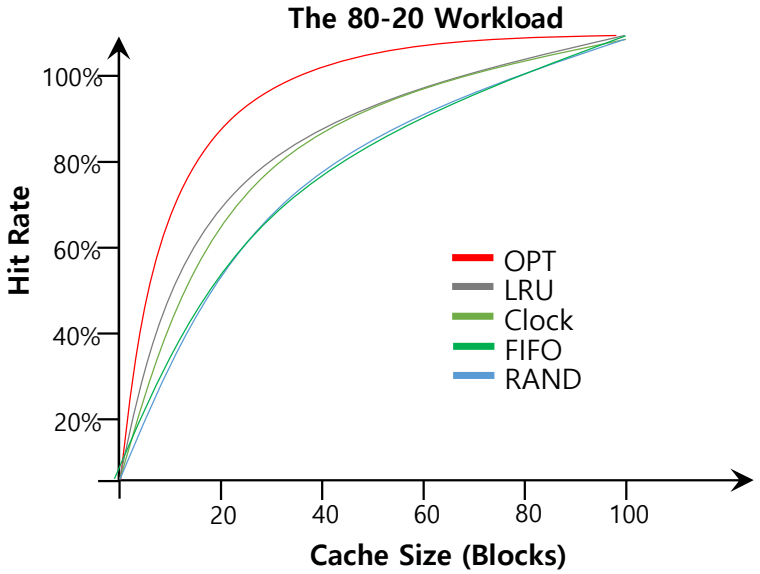
- LRU alene dårlig med lille cache
  - Spatial lokalitet?
- RAND fornuftig

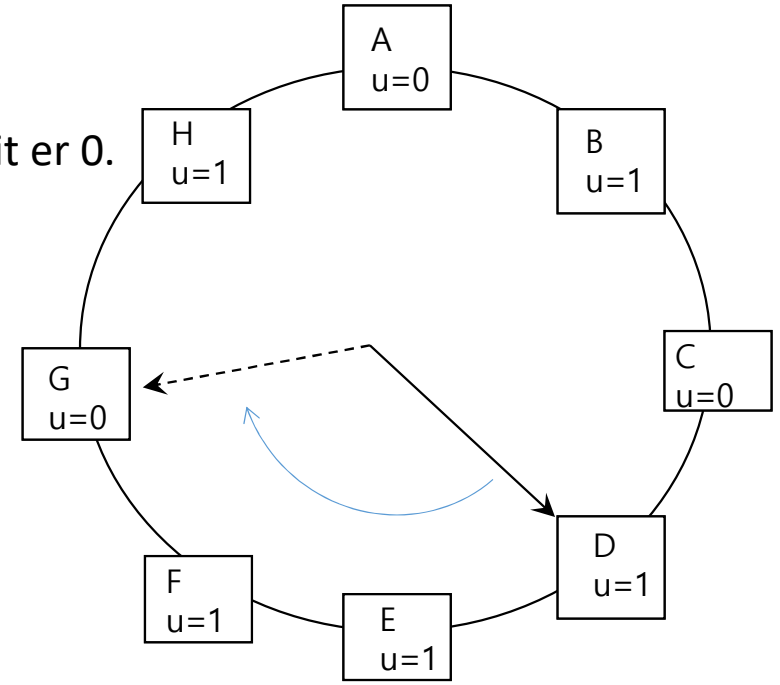
Realisering af LRU

- LRU er problematisk at realisere eksakt; vil kræve
  - En liste skal opdateres så en nyligt brugt side kommer forrest (for hver hukommelsesadgang!!), eller
  - MMU sætter tidsstempel i sidetabellen ved hver hukommelsesadgang: mange bits+gennemløb af hele-sidetabel ved ofring
- Aproximation
  - Side-tabel udvides med en ”used/referenced”-bit
  - MMU sætter den ved hver hukommelses adgang
  - OS ofrer sider hvor r-bit=0.
  - CLOCK algoritme

Realisering af LRU: Clock Algoritme

- Alle sider arrangeres i en a cirkulær liste.
- MMU: Sætter Used-bit = 1 i PTE ved læsning/skrivning til siden.
- OS: Når et offer skal findes,
  - Alg. Søger fra viserens position efter en side, hvis used-bit er 0.
  - Sætter passerede u-bits til 0
  - Gemmer viserens position til næste gang



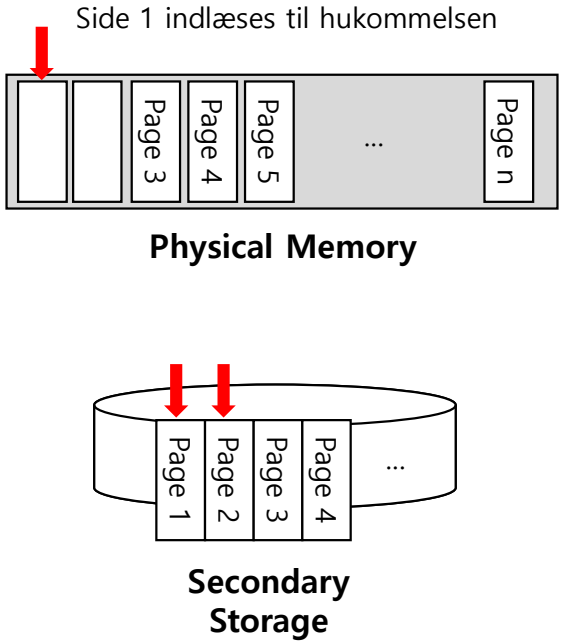


Swap Dæmon / page dæmon

- Hvornår starter side erstatning:
  - Når OS er løbet helt tør for ledige fysiske sider?
- Foretages af OS i baggrunden af Page Daemon proces baseret på "vandstand"
  - Når der er færre end LW (Low Watermark) ledige sider
  - ofres sider indtil HW (High Watermark) sider er ledige

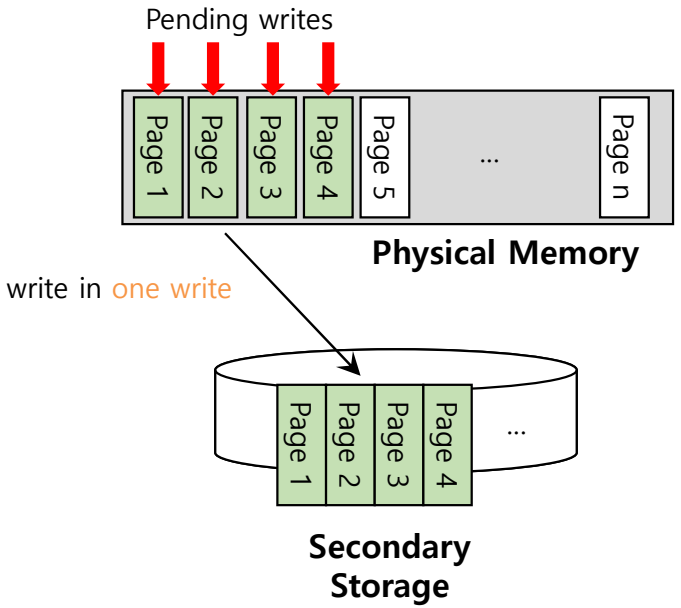
Optimering: forud-indlæsning af sider

- Forudindlæsning (pre-fetching): næste side
- Fx: VM cache miss af side 1: sandsynligt at side 2 også skal bruges i nær fremtid (spatial lokalitet), så indlæsning påbegyndes kort inden behov opstår



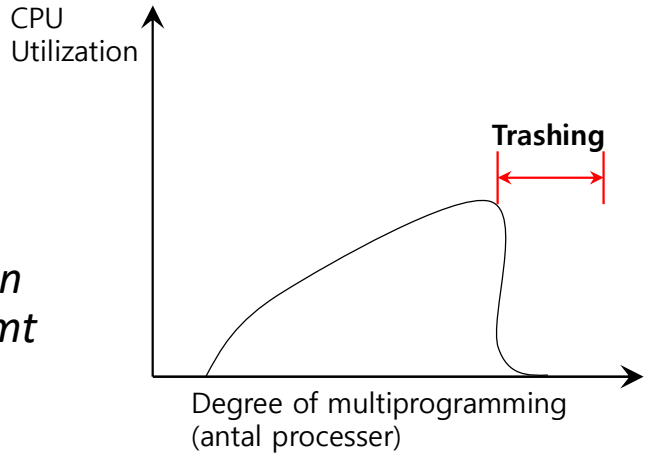
Optimering: gruppering af skrivninger

- OS samler et antal sider med udestående skrivning, og skriver dem samlet
  - Evt sorteret efter disk adresse for at minimere seek/rotations forsinkelse
- Mere effektivt end mange enkelt skrivninger



Begrænsninger ved VM cache: Trashing

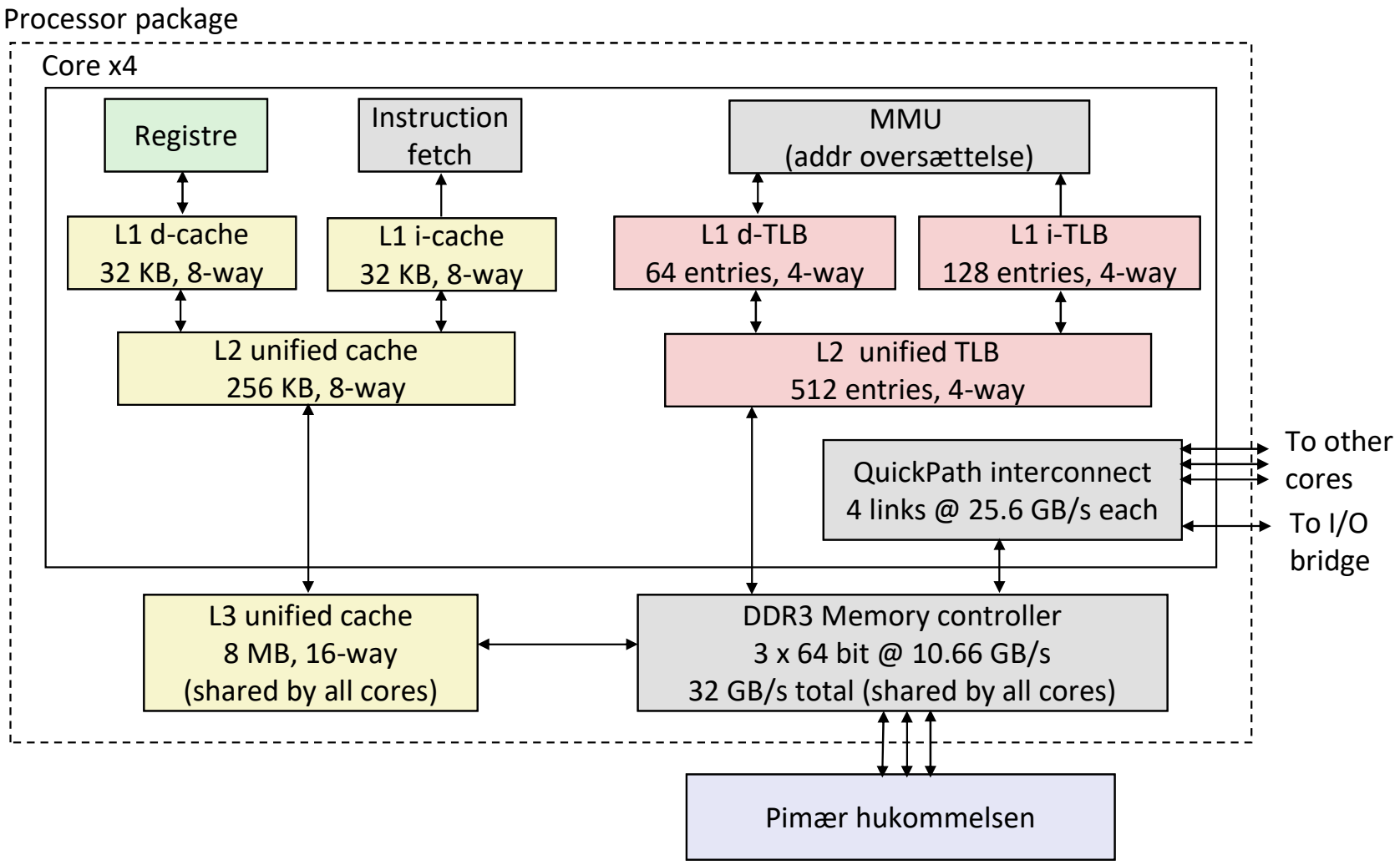
- Virtuel hukommelse virker håbløst ineffektivt, men virker pga.
  - lokalitet.
  - MMU understøttelse, mange optimeringer
  - Gode OS alg og datastrukturer.
- Til ethvert tidspunkt, har processer en tendens til kun at tilgå en del-mængde af dets virtuelle sider. Aktive sider = *working set*
  - Programmer med god lokalitet har normalt mindre working sets
- IF (SUM(working sets) < størrelse af DRAM)
  - God VM cache performance
- ELSE
  - *Nedsmeltning (Thrashing): Yderst ringe yde-evne hvor sider hele tiden udskiftes – adgangstid domineres af disk = laaaaaaaaaaaaaaangsomt*



Supplerende

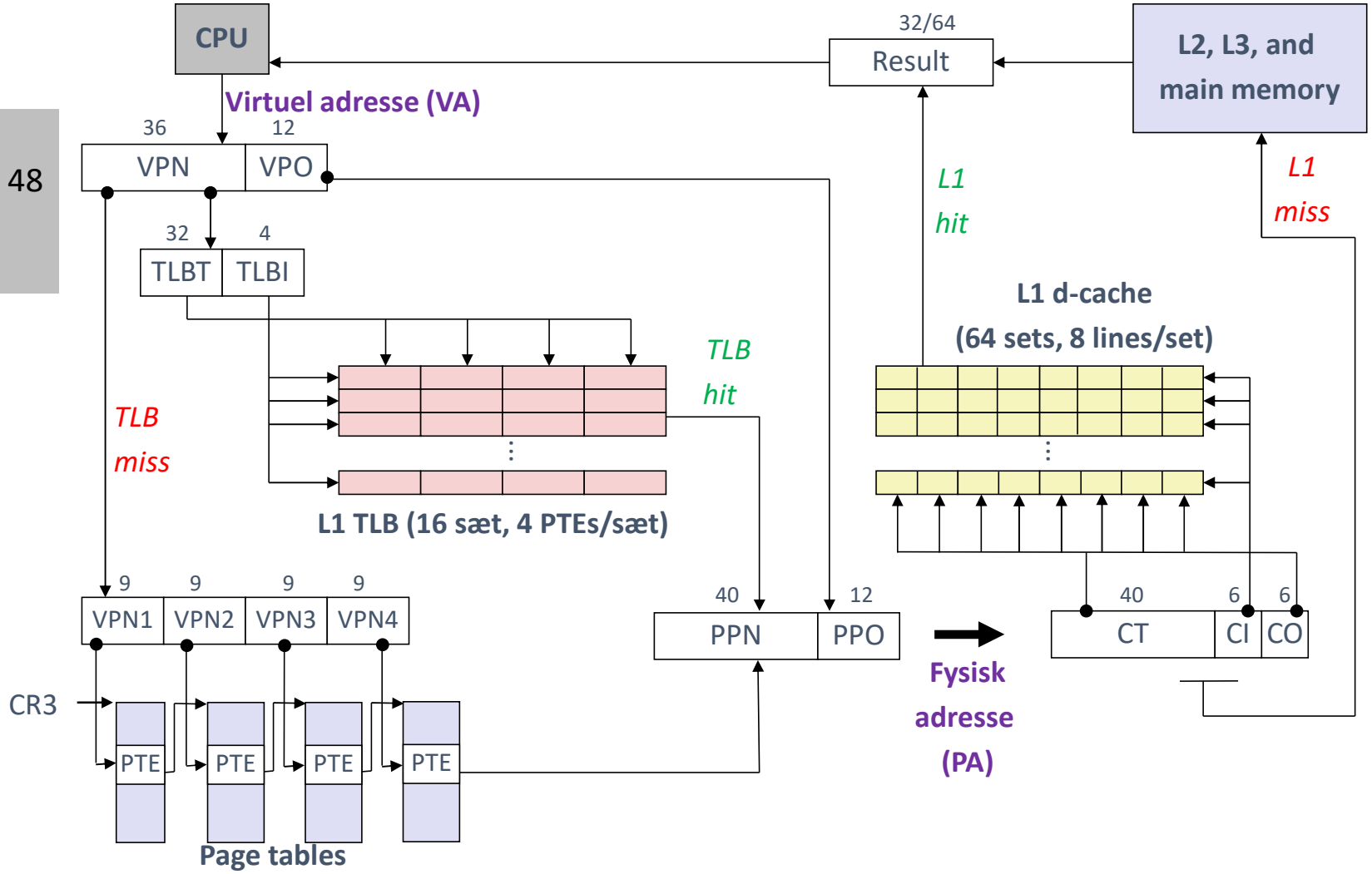
Hukommelsessystemet i core-i7 caset.

Hukommelses-system for Intel Core i7



Adresseoversættelse i Core i7

Nuværende i7 ”begrænset” til 48 bits VA=256 TB og 52 bits PA



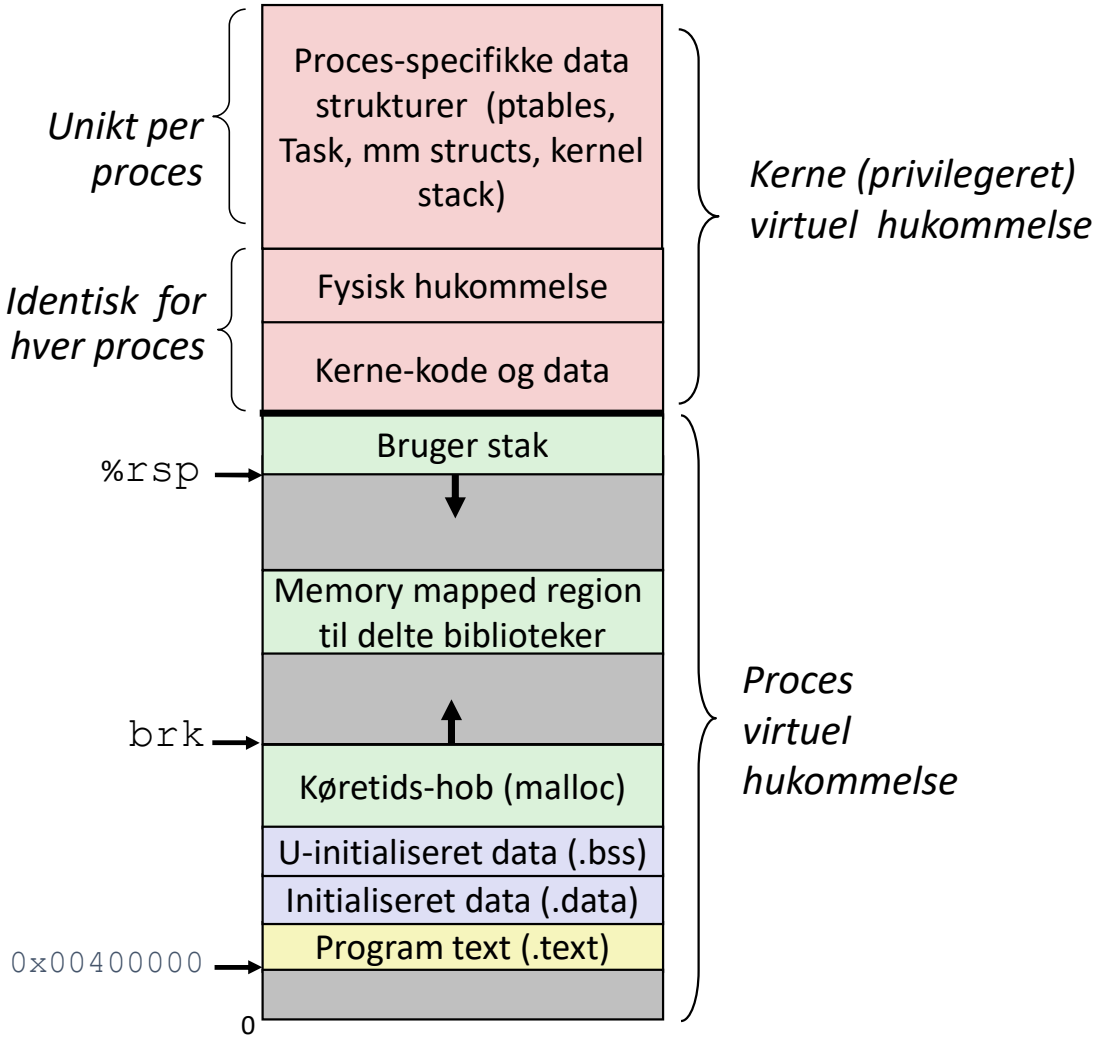
Tricks med adresserum

Virtuelt Adresserum for Linux Proces

Hurtigere ”system-kald”

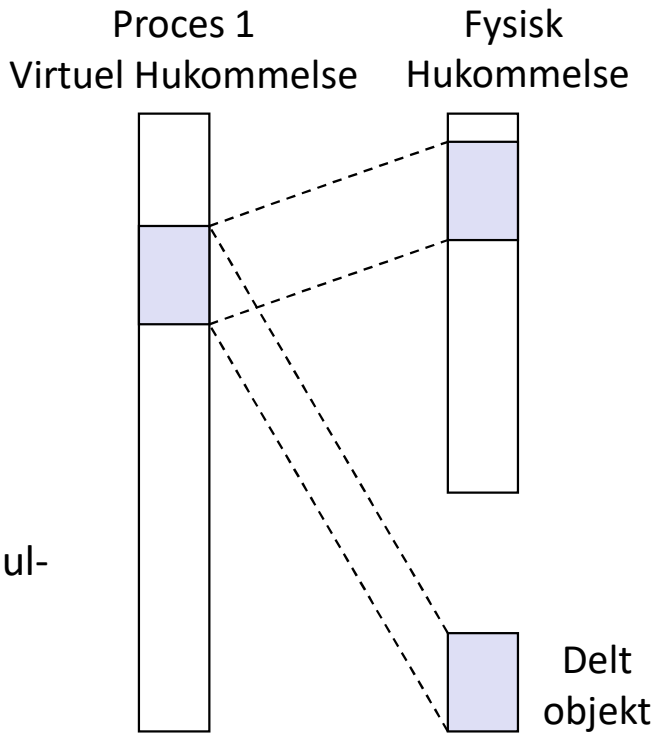
Ved system-kald er både kernen kode/data og processens data tilrådighed for kernen i samme adresse-rum

=> Slipper for at udskifte sidetabellen ved system kald



Memory mapping

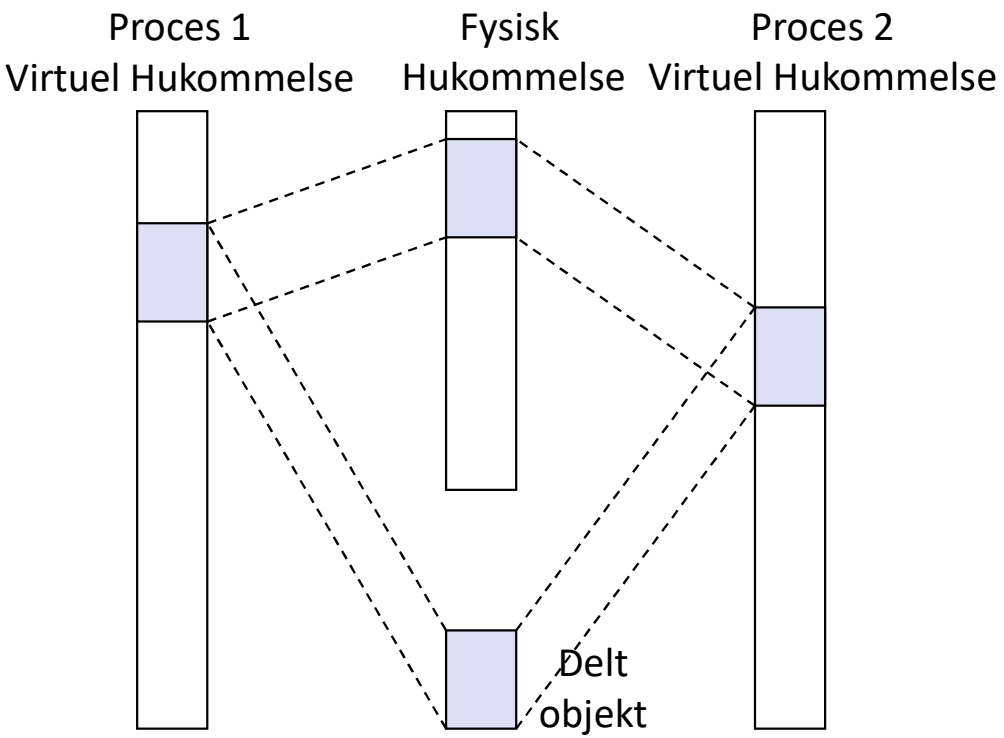
- ***Memory mapping.*** VM områder initialiseres ved at knytte dem til disk filer.
- Område understøttes af:
  - ***Alm. Fil*** på disk (f.x, eksekverbar fil)
    - Indlæses sidevise i fysisk hukommelse ved sidefejl
  - ***Speciel Anonym fil***
    - Første sidefejl allokerer ny fysisk side, nul-stillet (***demand-zero page***)
    - Hvis skrevet, håndteres den som andre side
- Skrevne sider (dirty pages) kopieres fra og tilbage mellem hukommelsen og speciel *swap fil.*



- Proces 1 mapper det delte objekt.

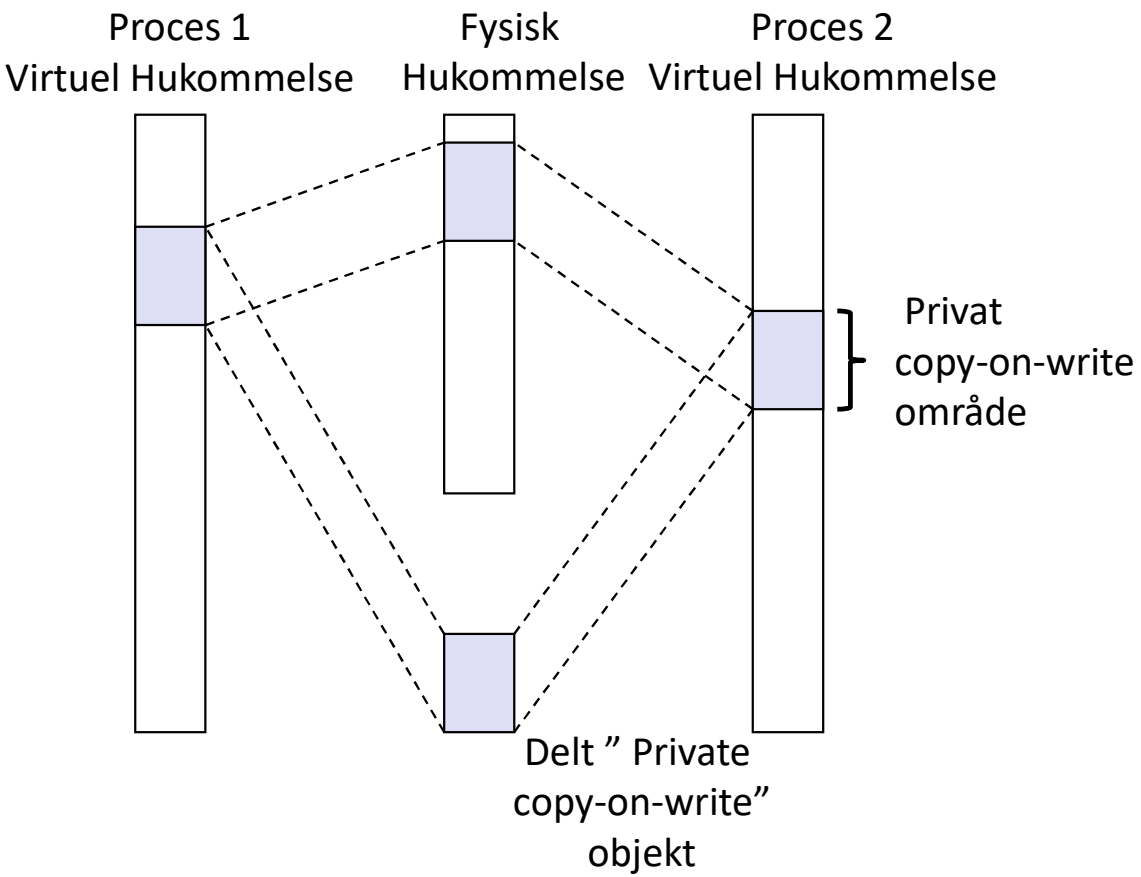
Delte Objekter

- **Proces 2 mapper det delte objekt.**
- Til eget virtuelle adresserum
- Potentiel på forskellige virtuelle adresser.
- **Kun én kopi i fysisk-hukommelse**
- Kan også anvendes til interproces kommunikation



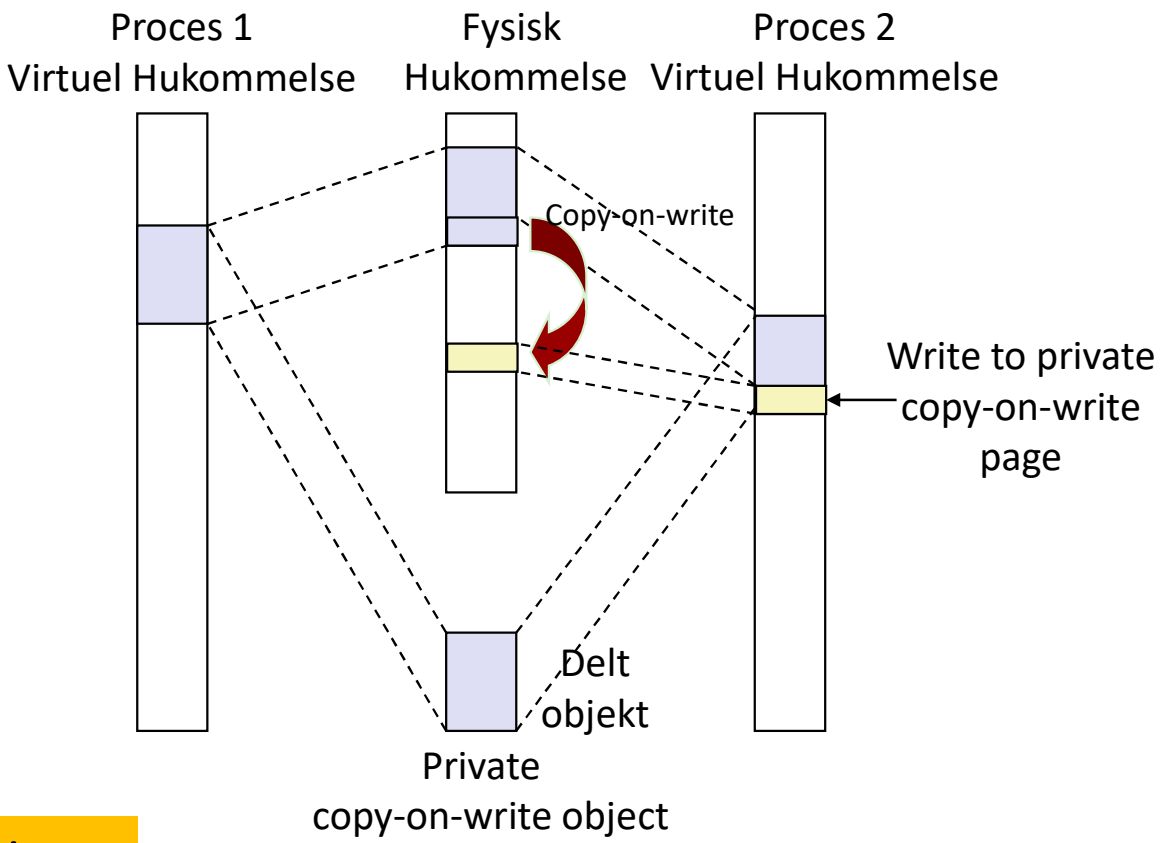
Deling: Copy-on-write

- To processer mapper et delt objekt markeret som ” ”*private copy-on-write (COW)”* .
- PTEs in privat områder har read-only-flag sat.



Deling: Copy-on-write

- Den skrivende instruktion udløser sidefejl.
- OS-Handler opretter ny R/W side.
- Instruktionen genstarter.
- Kopiering udsat til det er nødvendigt!

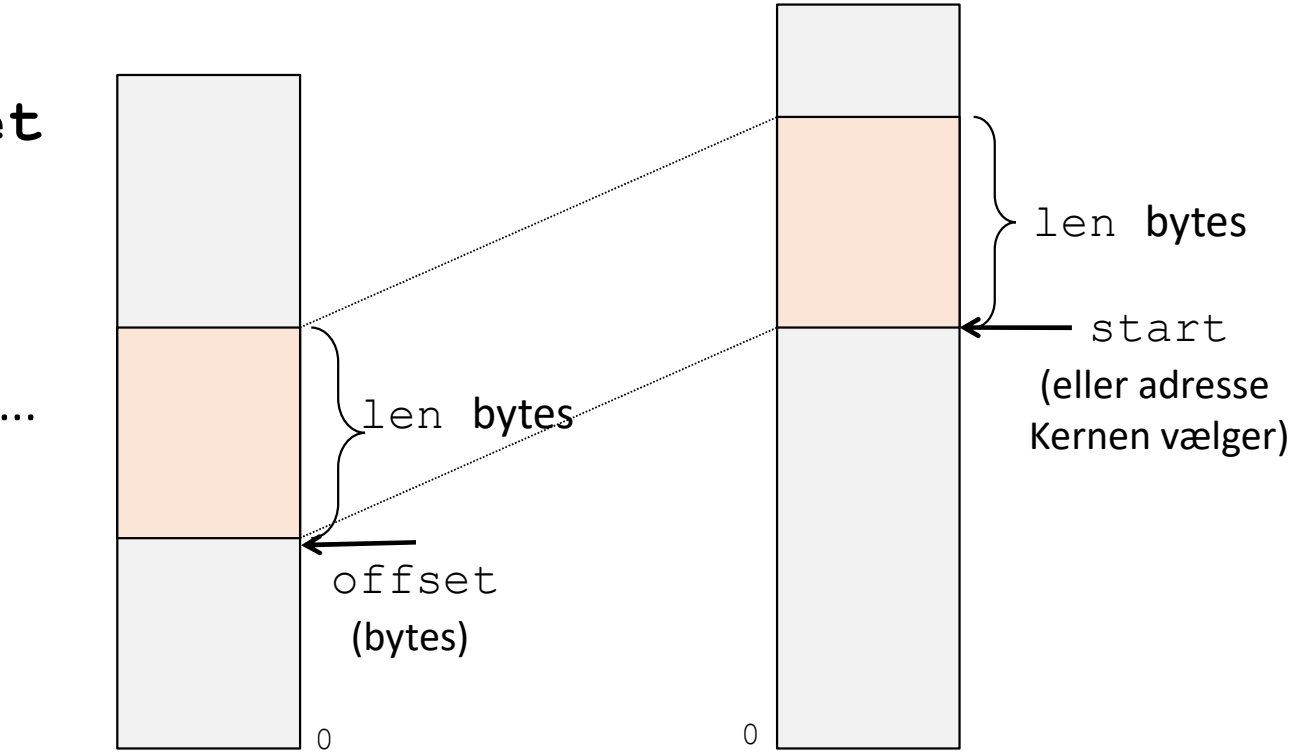


Bruges fx. i Linux til effektiv implementation af `EXECVE()` og `FORK()`

Bruger-niveau Mapning

```
void *mmap(void *start, int len,int prot, int flags, int fd, int offset)
```

- Map **`len`** bytes fra offset **`offset`** i filen angivet med fil-description **`fd`**, (om mulig) i adresse **`start`**
  - **`start`**: Hvis 0, lader OS vælge
  - **`prot`**: PROT_READ, PROT_WRITE, ...
  - **`flags`**: MAP_ANON, MAP_PRIVATE, MAP_SHARED, ...
- Returnerer pointer til start af det mappede område



Demo: Brug `mmap` til filadgang

- Læs og skriv til fil som data-array.

```
#include <stdio.h>
#include <sys/mman.h>
//…
int main(int argc, char *argv[])   {
      int fd=0, offset=0;
      char *data;

      if ((fd = open("mmapdemo.c", O_RDWR)) == -1) {
         perror("open");
         exit(1);
      }

      if (stat("mmapdemo.c", &sbuf) == -1) { //get filesz
         perror("stat");
         exit(1);
      }
```

mmapdemo.c

```
      if ((data = mmap((caddr_t)0, sbuf.st_size,
PROT_WRITE, MAP_SHARED,   fd, 0)) == (caddr_t)(-1)) {
         perror("mmap");
         exit(1);
      }

      for (offset=0; offset<50;offset++)
            printf("byte at offset %d is '%c'\n",
            offset, data[offset]);

      printf("Now writing HELLO to file at position 50\n");
      data[50]='H';data[51]='E'; data[52]='L';
      data[53]='L'; data[54]='O';

      return 0;
}
```

mmapdemo.c

Resumé

- Programmøren's syn på VM
  - Hver proces har sit eget private (store) lineære adresserum
  - Kan ikke læses/skrives fra andre processor (undtagen, hvad som er explicit tilladt som delt)

- Systemets syn på VM
  - Bruger hukommelsen effektive ved at cache virtuelle sider
    - Virker kun pga. lokalitetsprincippet
  - Simplificerer hukommelses-administration og programmering
  - Simplificerer beskyttelse ved at give en bekvem lejlighed til at checke permissions