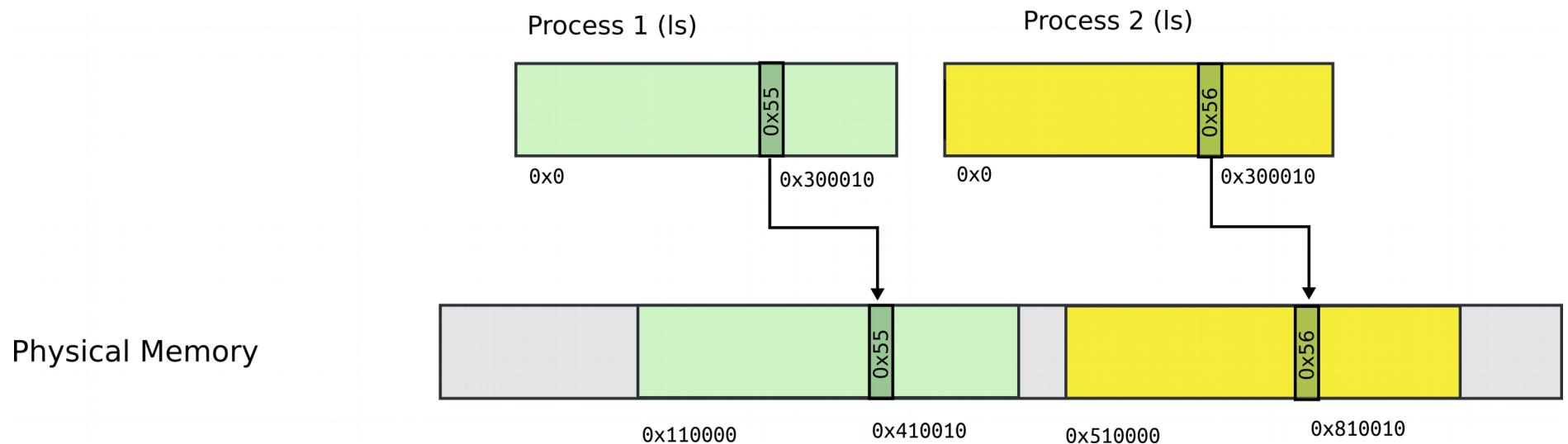


# 143A: Principles of Operating Systems

## Lecture 5: Address translation (Part 2)

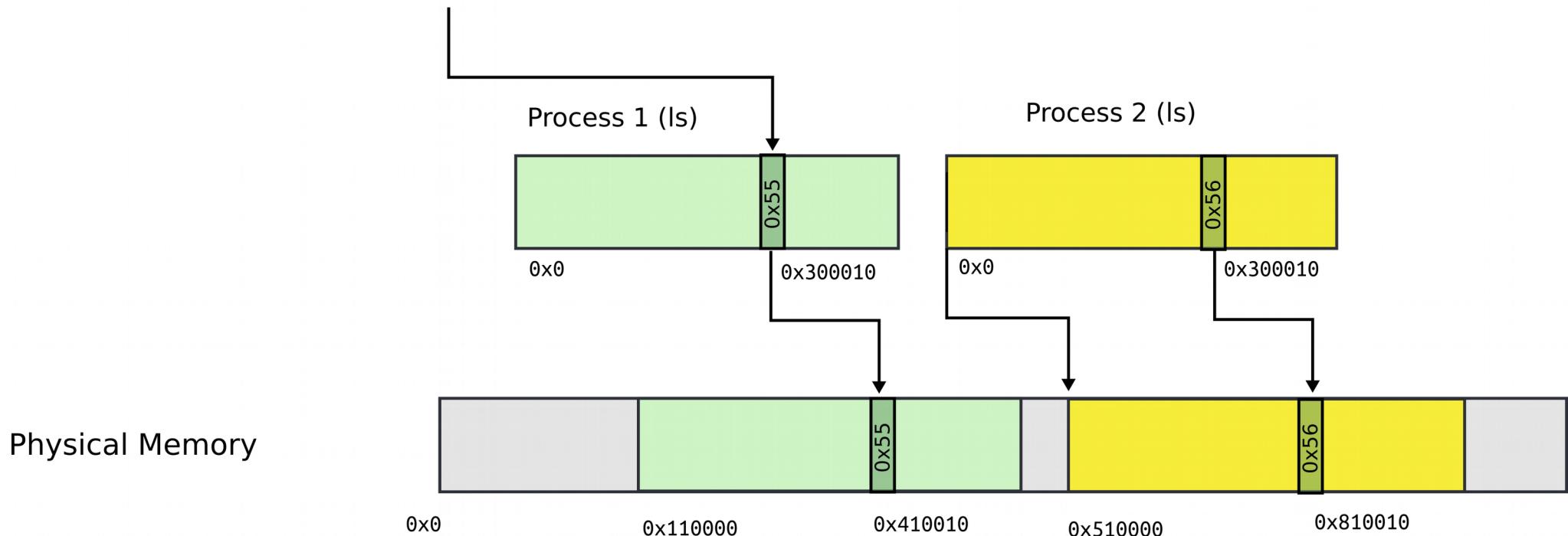
Anton Burtsev  
October, 2018

# Segmentation



# Segmentation: example

```
mov (%EBX), EAX # mov value from the location pointed by EBX into EAX  
EAX = 0x0  
EBX = 0x300010
```

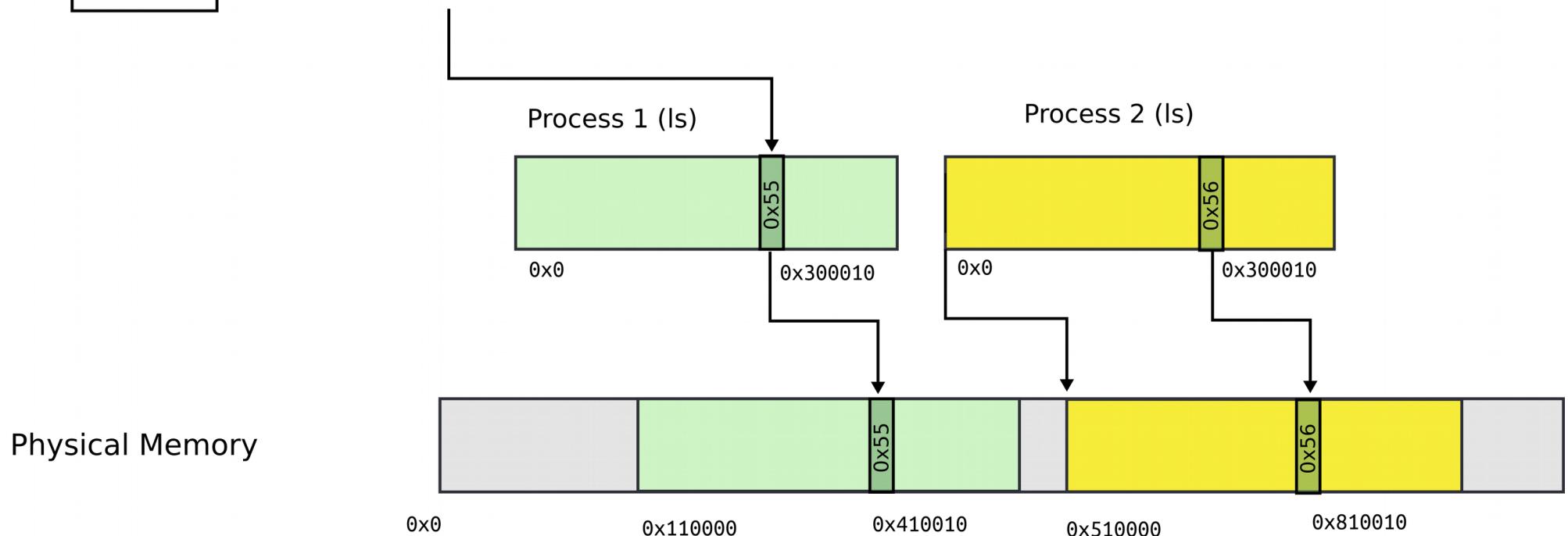


# Segmentation: address consists of two parts

**Segment register  
(CS, SS, DS, ES, FS, GS)**

0x1

mov (%EBX), EAX # mov value from the location pointed by EBX into EAX  
EAX = 0x0  
**EBX = 0x300010, DS = 0x1**



- Segment register contains segment selector
- General registers contain offsets
- Intel calls this address: “logical address”

# Segmentation: Global Descriptor Table

Segment register  
(CS, SS, DS, ES, FS, GS)

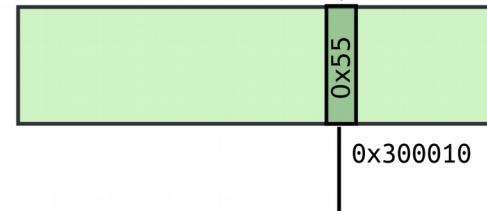
0x1

mov (%EBX), EAX # mov value from the location pointed by EBX into EAX  
EAX = 0x0  
EBX = 0x300010, DS = 0x1

Global Descriptor Table  
(table of segment sizes and bases)

0x0
0x110000
0x510000
...

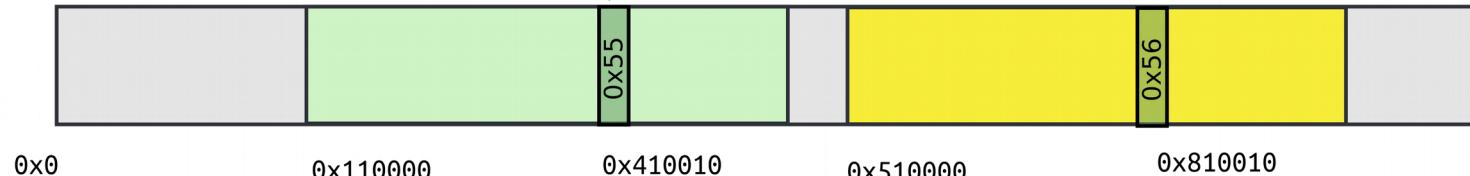
Process 1 (ls)



Process 2 (ls)

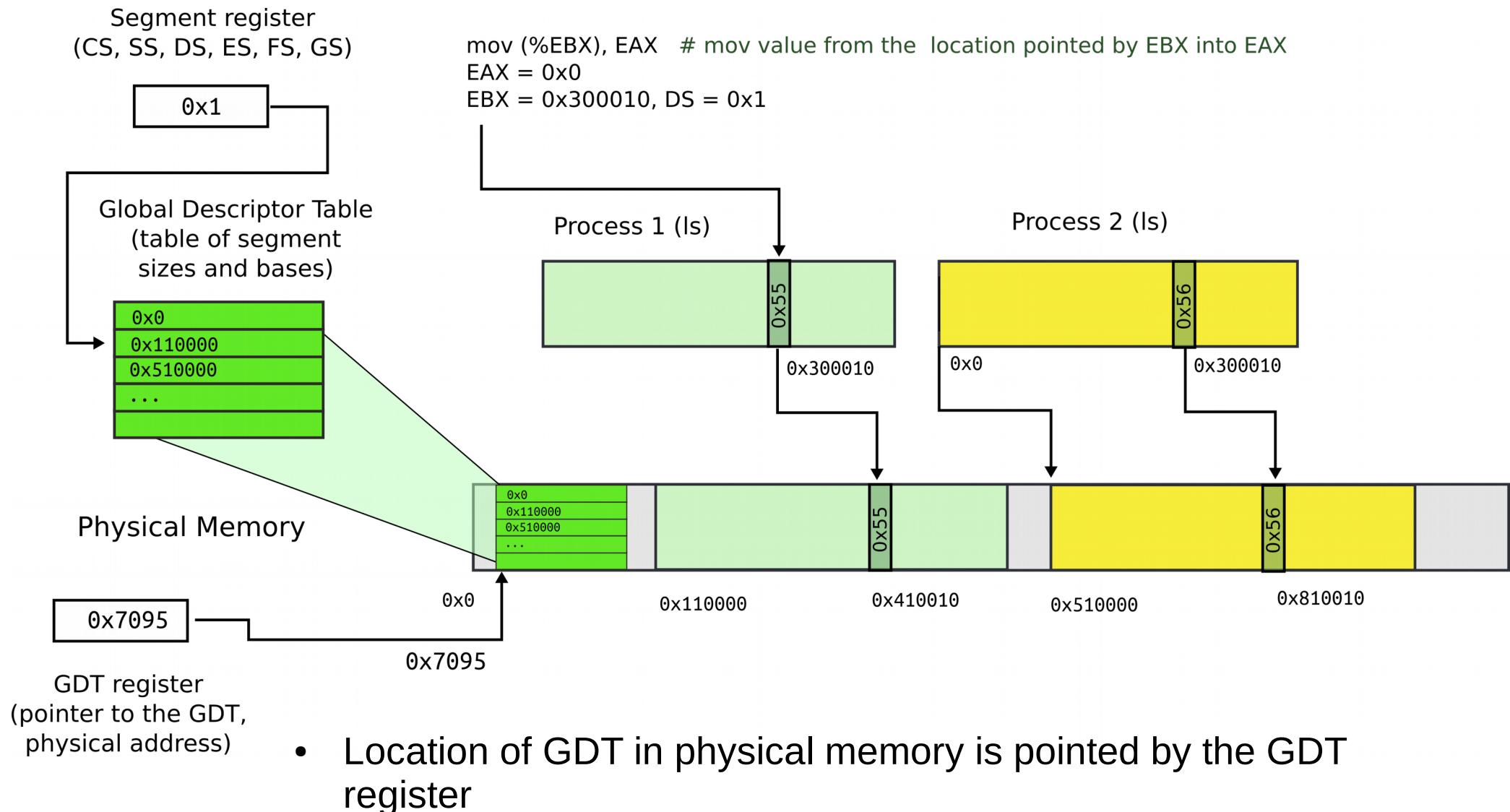


Physical Memory



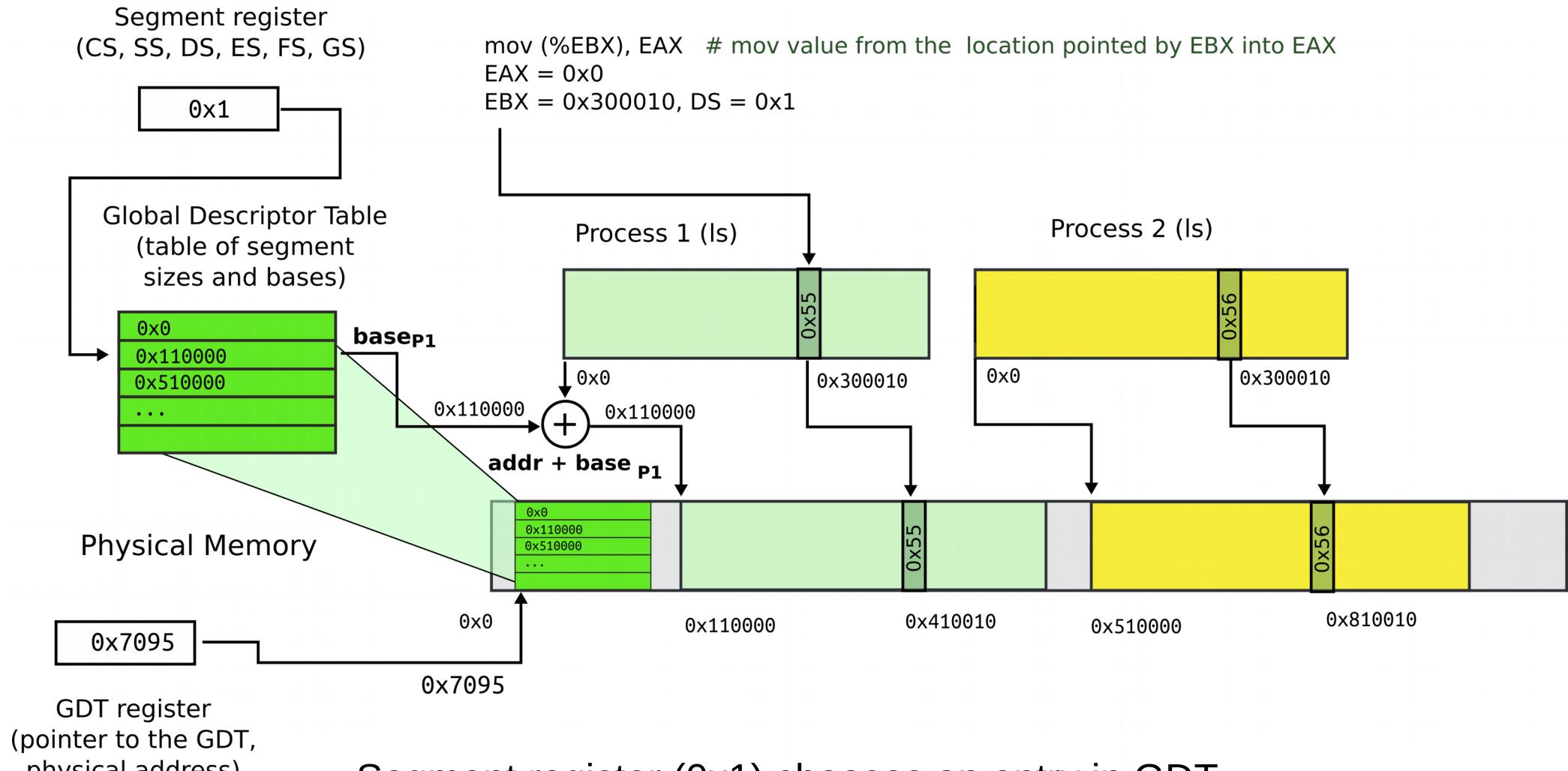
- GDT is an array of segment descriptors
  - Each descriptor contains base and limit for the segment
  - Plus access control flags

# Segmentation: Global Descriptor Table



- Location of GDT in physical memory is pointed by the GDT register

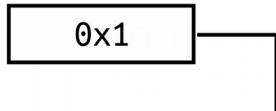
# Segmentation: base + offset



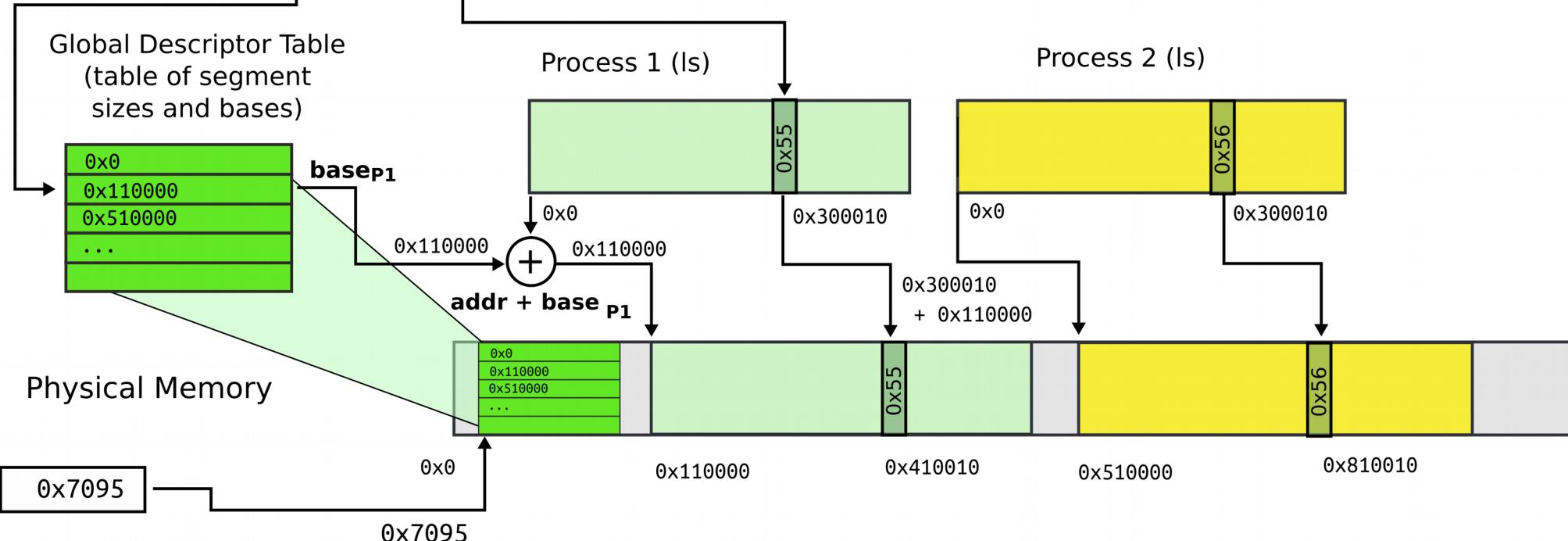
- Segment register (0x1) chooses an entry in GDT
- This entry contains base of the segment (0x110000) and limit (size) of the segment (not shown)

# Segmentation: base + offset

Segment register  
(CS, SS, DS, ES, FS, GS)



mov (%EBX), EAX # mov value from the location pointed by EBX into EAX  
EAX = 0x0  
EBX = 0x300010, DS = 0x1



- **Physical address:**

- $0x410010 = 0x300010 \text{ (offset)} + 0x110000 \text{ (base)}$
- Intel calls this address “linear”

# Segmentation: process 2

Segment register  
(CS, SS, DS, ES, FS, GS)

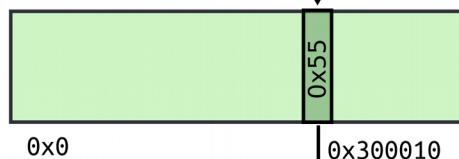
0x1

mov (%EBX), EAX # mov value from the location pointed by EBX into EAX  
EAX = 0x0  
EBX = 0x300010, DS = 0x1

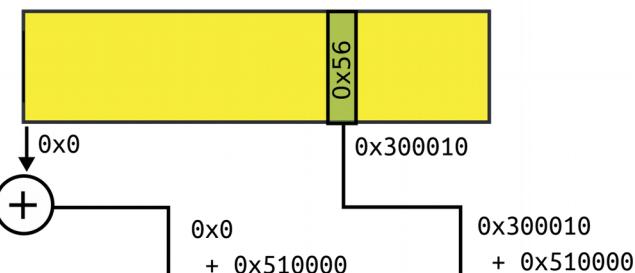
Global Descriptor Table  
(table of segment sizes and bases)

0x0	base <sub>P1</sub>
0x510000	
...	
...	

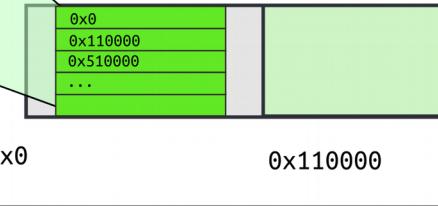
Process 1 (ls)



Process 2 (ls)



addr + base<sub>P1</sub>



Physical Memory

0x890000

0x0  
0x890000  
0x110000  
0x410010  
0x510000  
0x810010

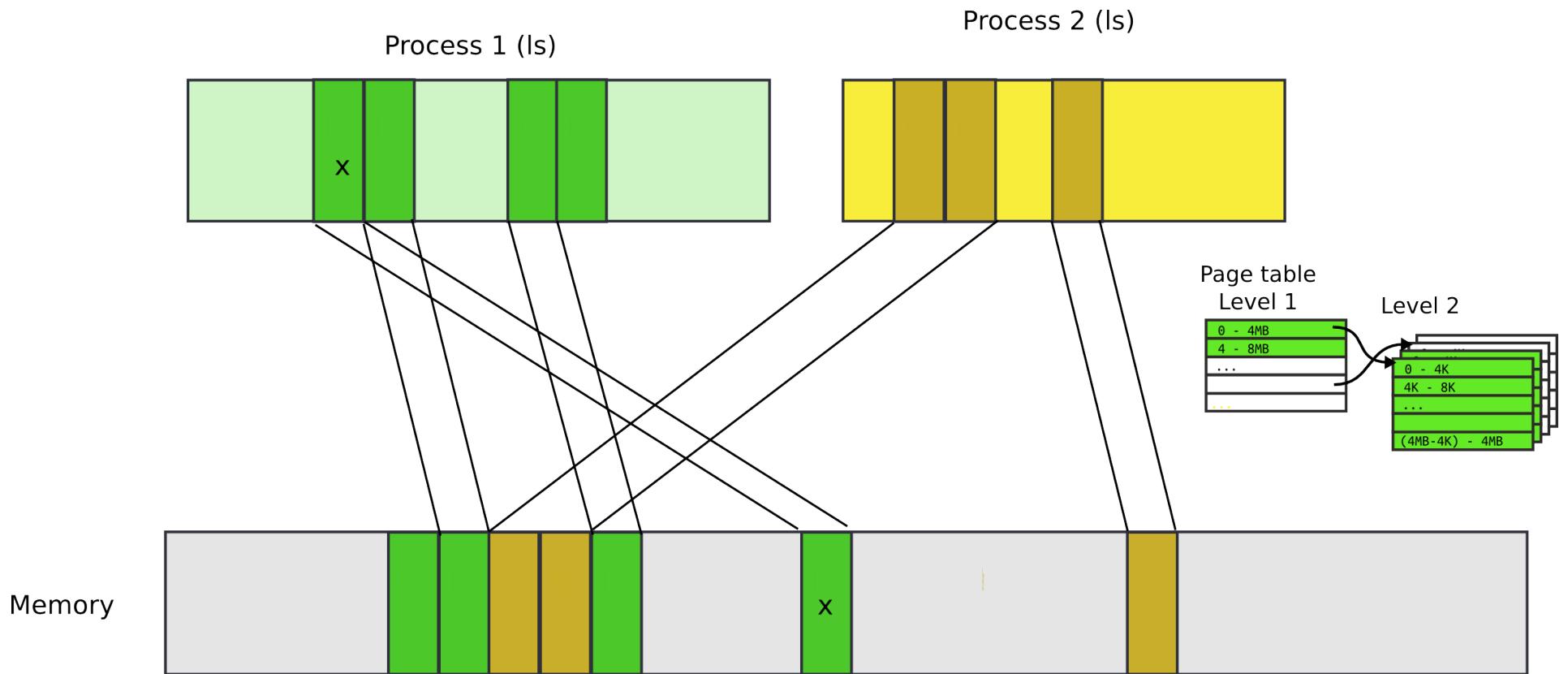
GDT register  
(pointer to the GDT,  
physical address)

- Each process has a private GDT
  - OS will switch between GDTs

# Segmentation: what did we achieve

- Illusion of a private address space
  - Identical copy of an address space in multiple programs
    - We can implement `fork()`
- Isolation
  - Processes cannot access memory outside of their segments

# Paging

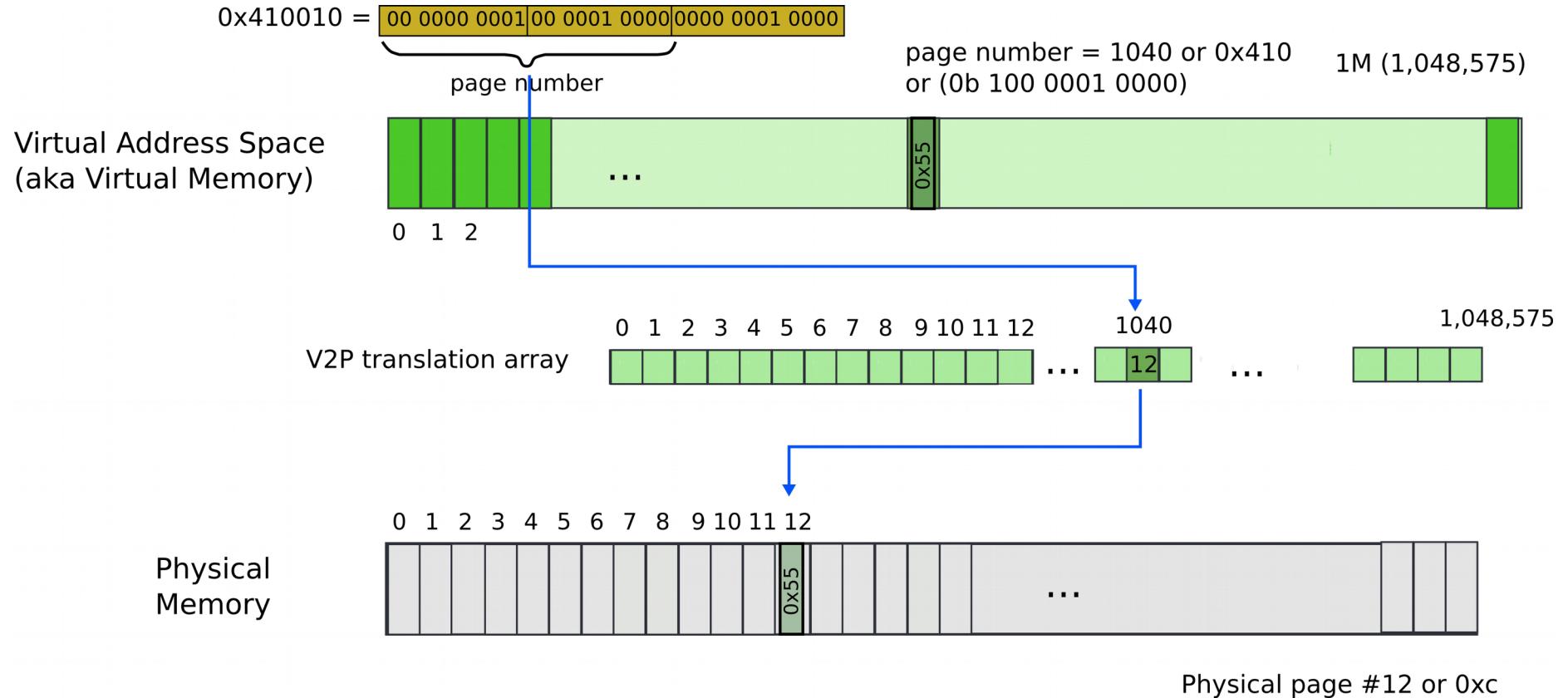


# Paging idea

- Break up memory into 4096-byte chunks called pages
  - Modern hardware supports 2MB, 4MB, and 1GB pages
- Independently control mapping for each page of linear address space
- Compare with segmentation (single base + limit)
  - many more degrees of freedom

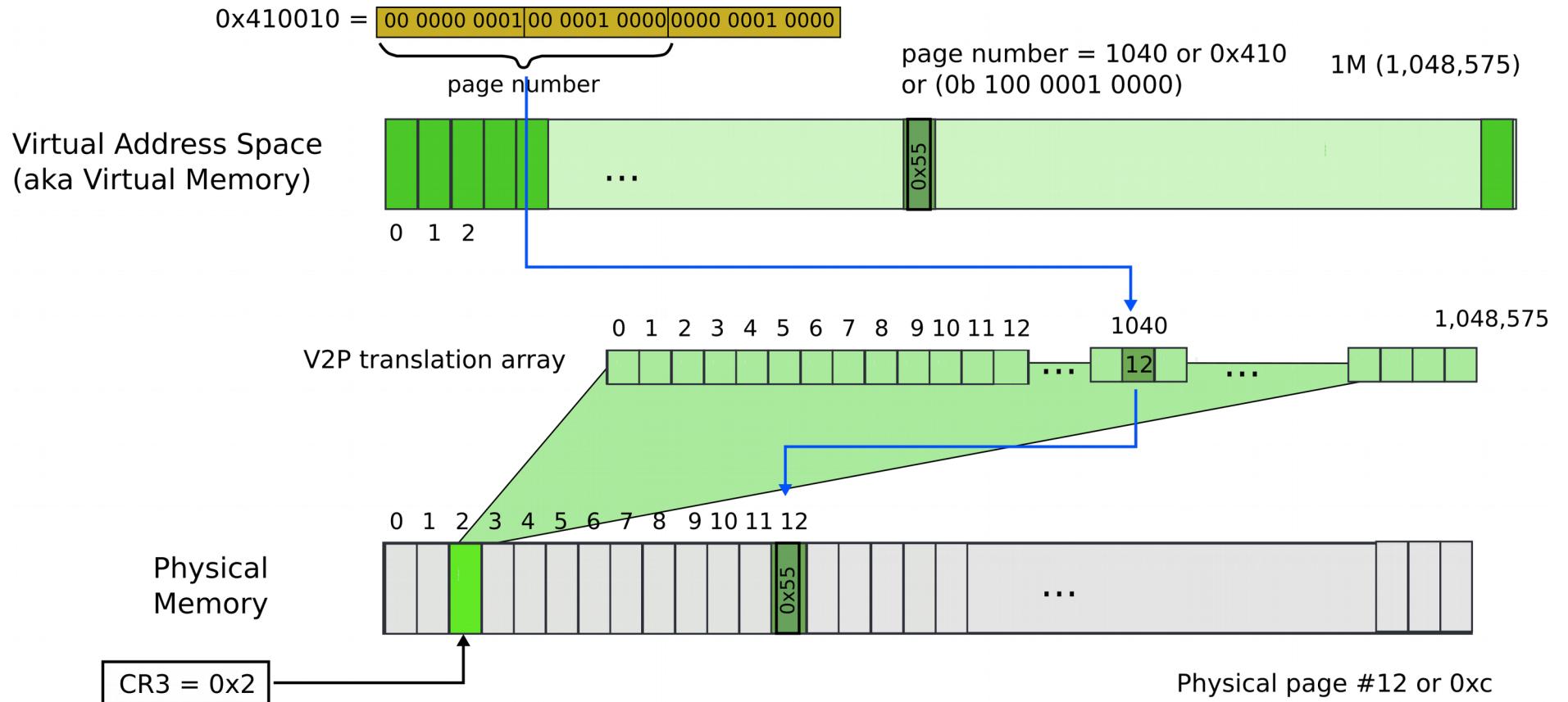
# How can we build this translation mechanism?

# Paging: naive approach: translation array



- Linear address  $0x410010$ 
  - Remember it's result of logical to linear translation (aka segmentation)
    - $0x410010 = 0x300010$  (offset) +  $0x110000$  (base)

# Paging: naive approach: translation array



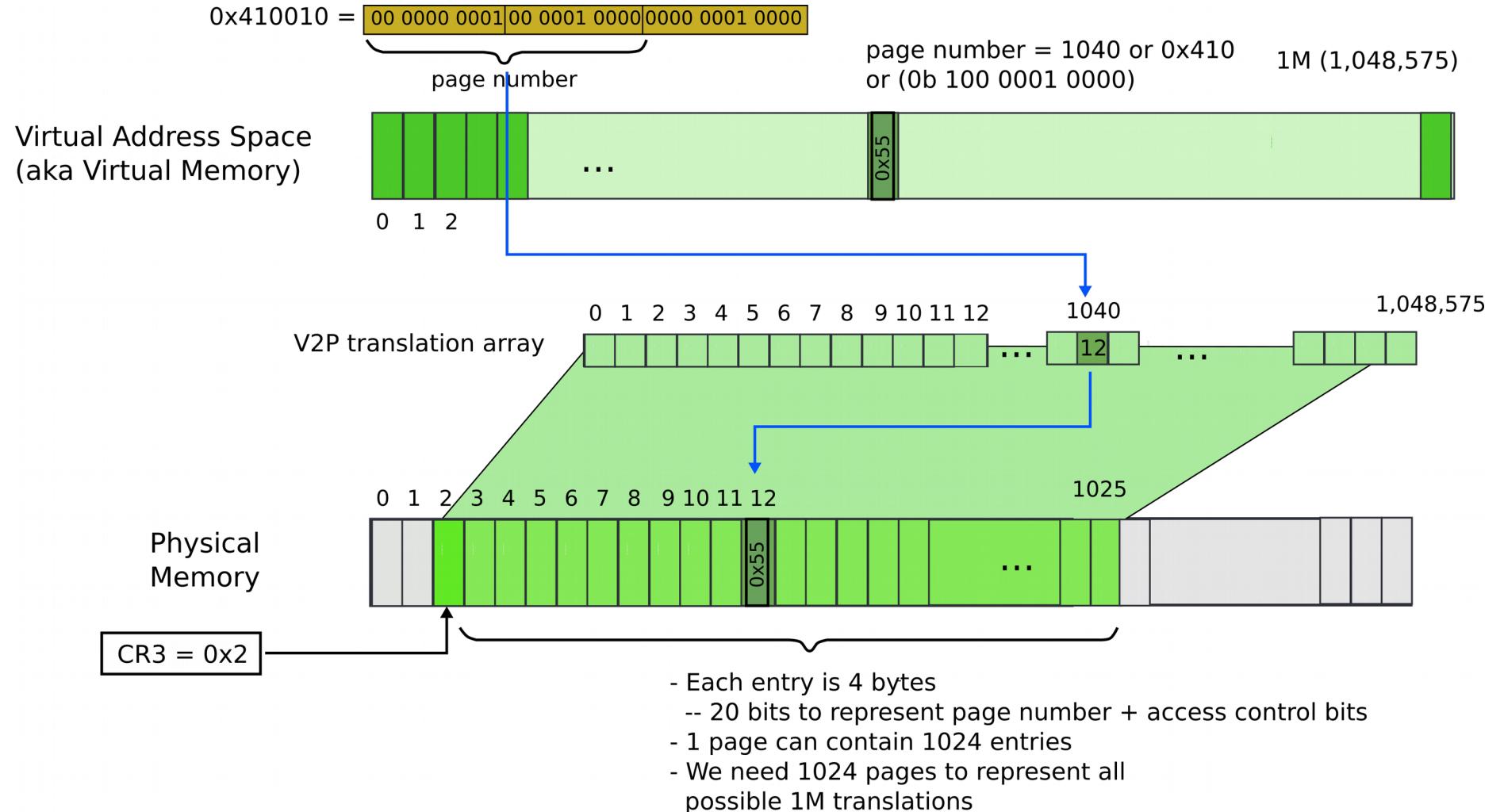
- Linear address 0x410010
  - Remember it's result of logical to linear translation (aka segmentation)
    - $0x410010 = 0x300010 \text{ (offset)} + 0x110000 \text{ (base)}$

# What is wrong?

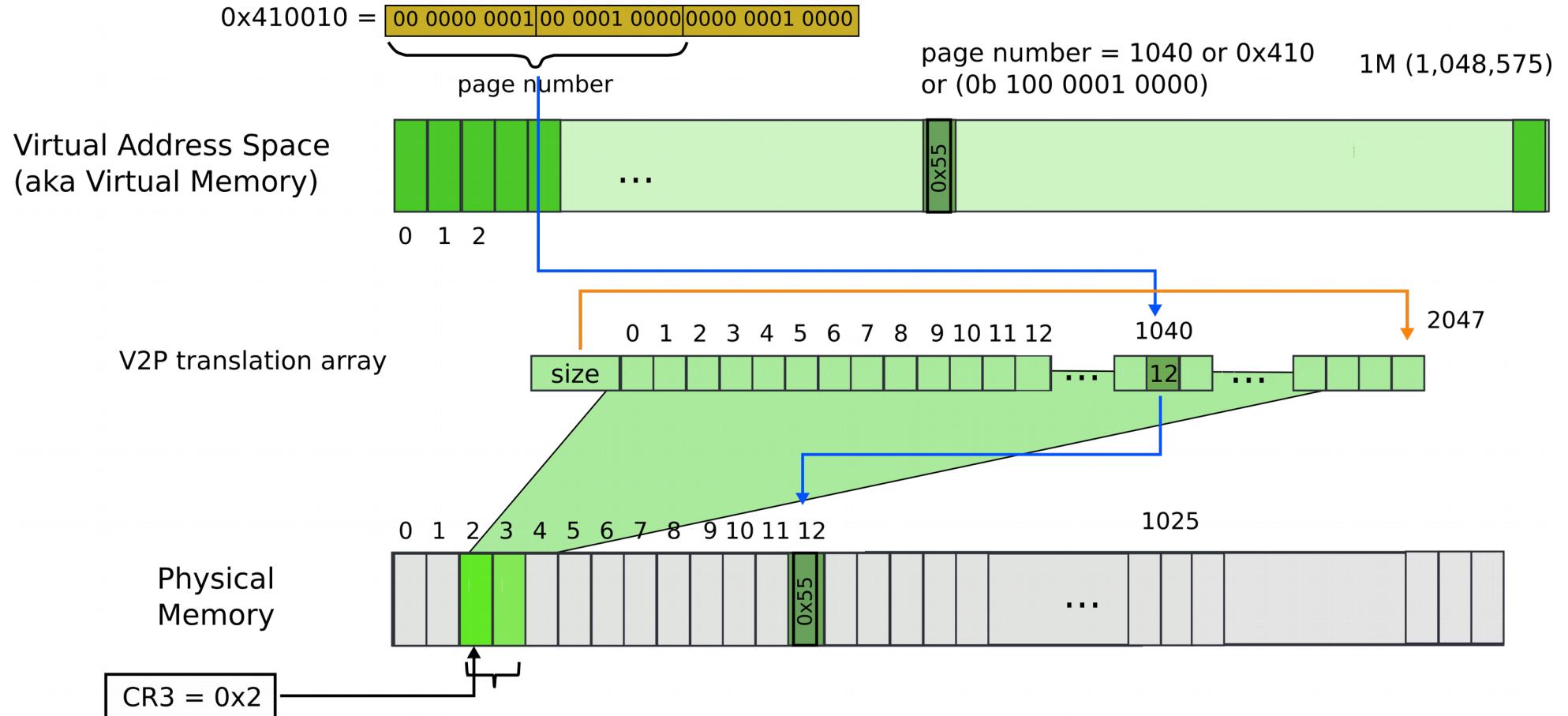
# What is wrong?

- We need 4 bytes to relocate each page
  - 20 bits for physical page number
  - 12 bits of access flags
- Therefore, we need array of 4 bytes x 1M entries
  - 4MBs

# Paging: naive approach: translation array



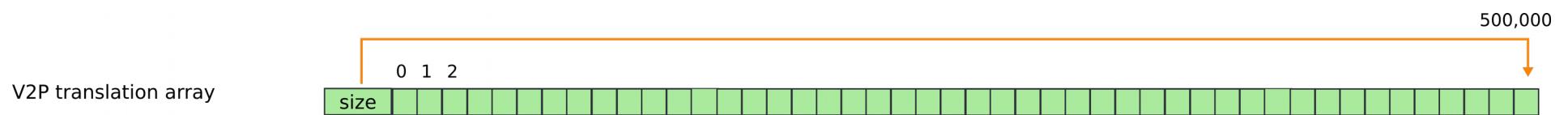
# Paging: array with size



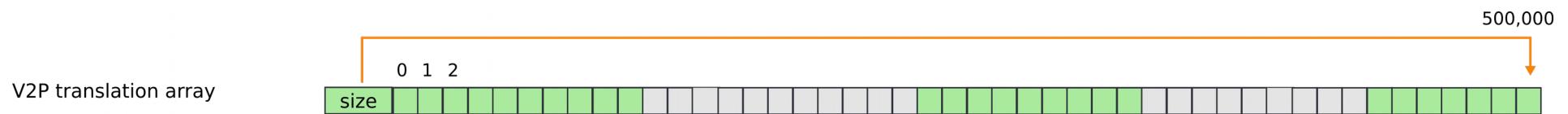
- The size controls how many entries are required

But still what may go wrong?

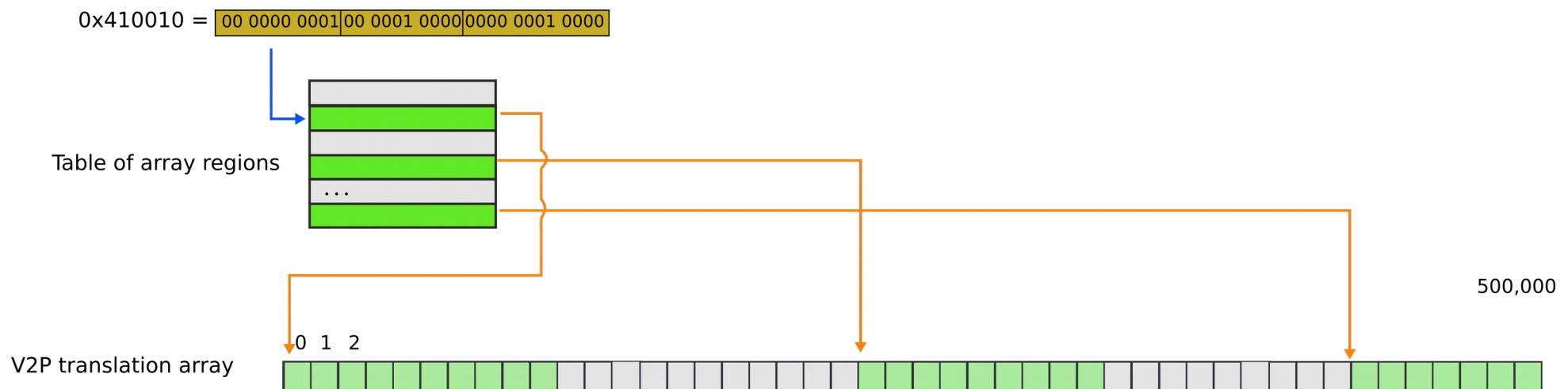
# Paging: array with size



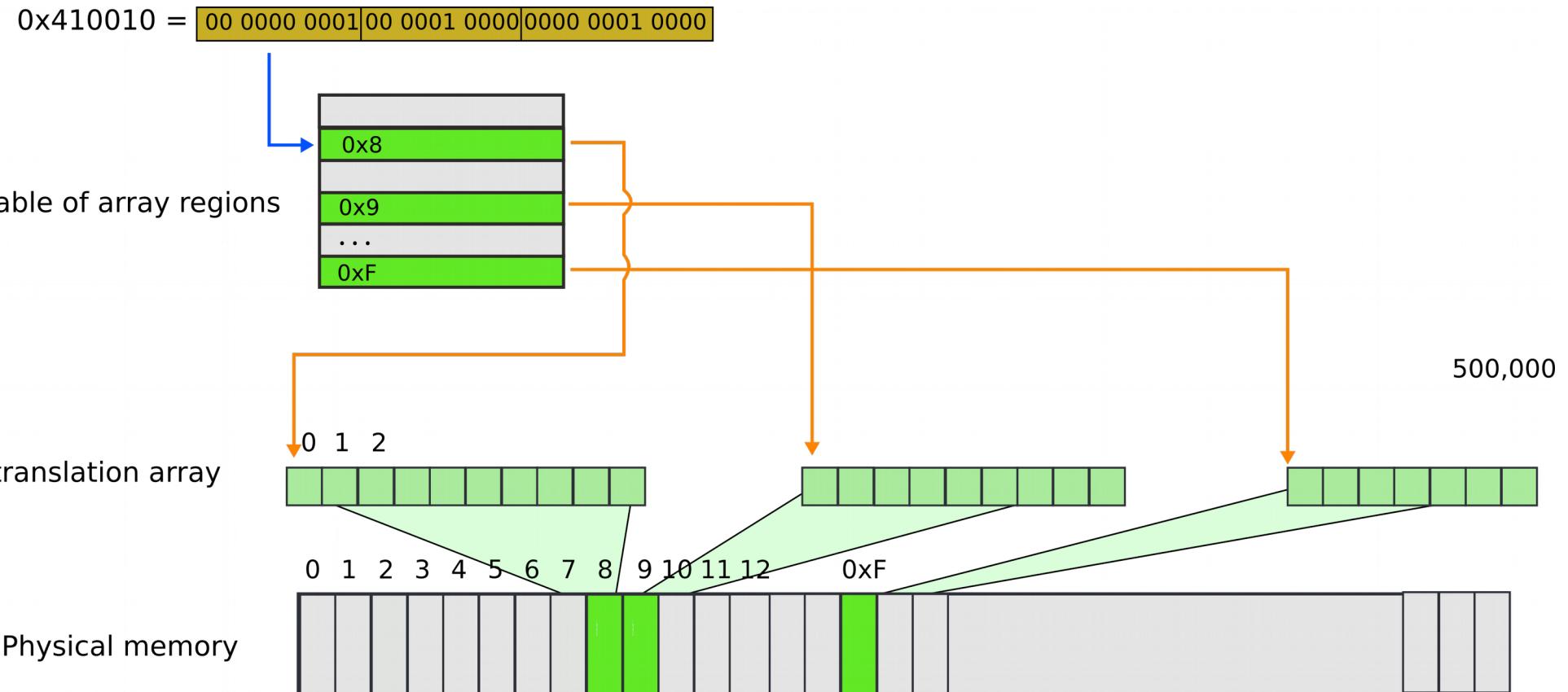
# Paging: array with size



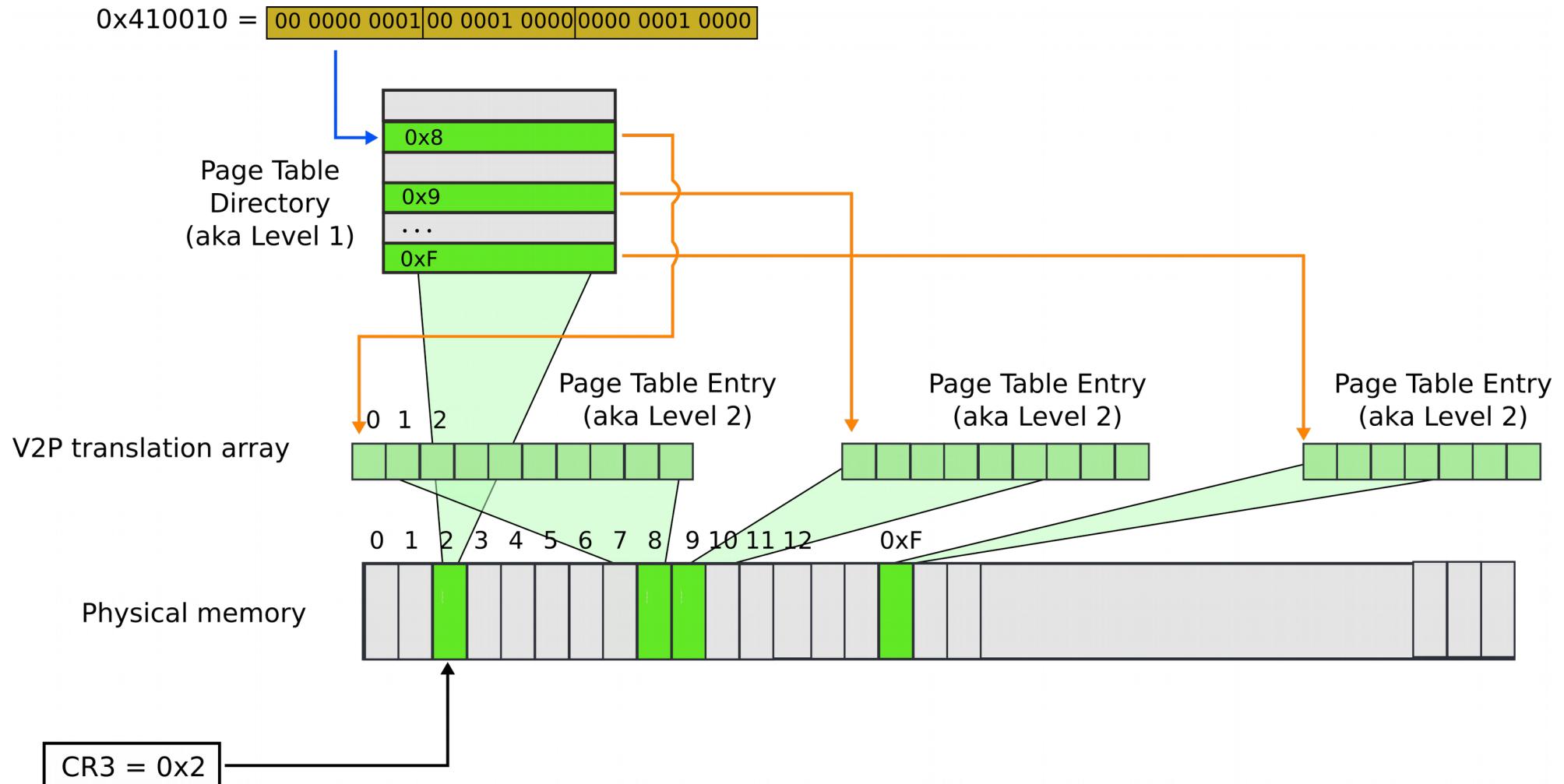
# Paging: array with chunks



# Paging: array with chunks

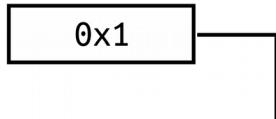


# Paging: page table

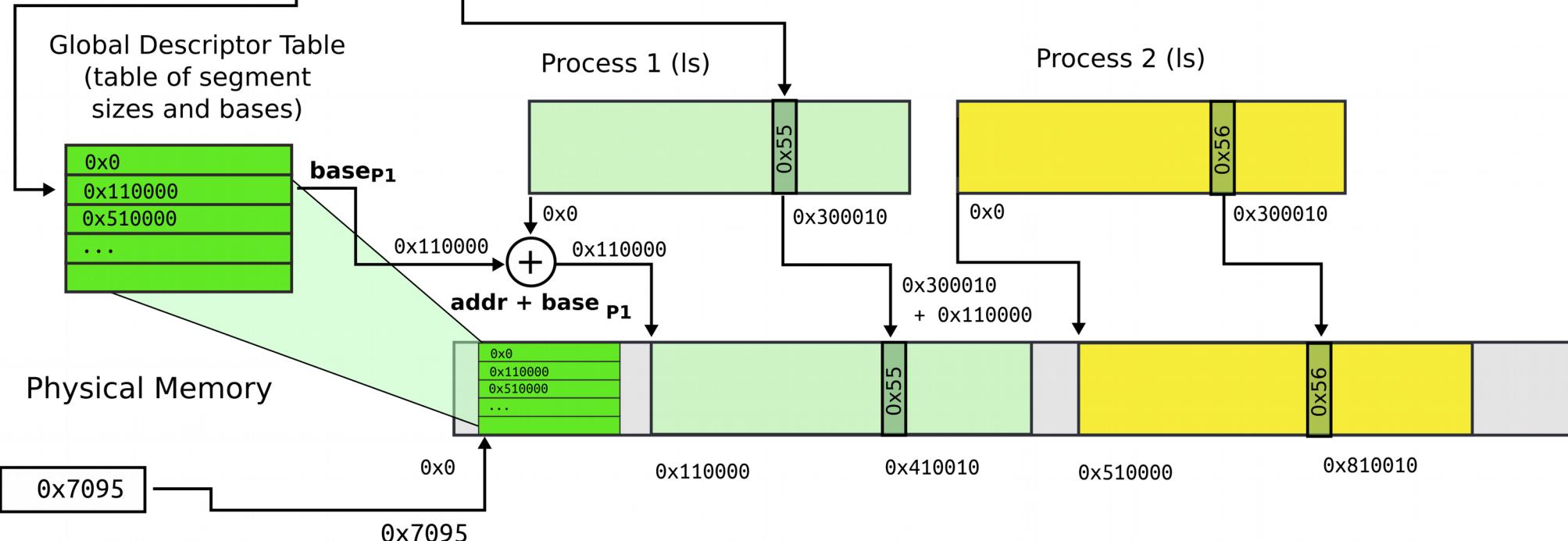


# Back to real page tables

Segment register  
(CS, SS, DS, ES, FS, GS)



mov (%EBX), EAX # mov value from the location pointed by EBX into EAX  
EAX = 0x0  
EBX = 0x300010, DS = 0x1



GDT register  
(pointer to the GDT,  
physical address)

- Physical address:

- $0x410010 = 0x300010 \text{ (offset)} + 0x110000 \text{ (base)}$
- Intel calls this address “linear”

# Paging

Segment register  
(CS, SS, DS, ES, FS, GS)

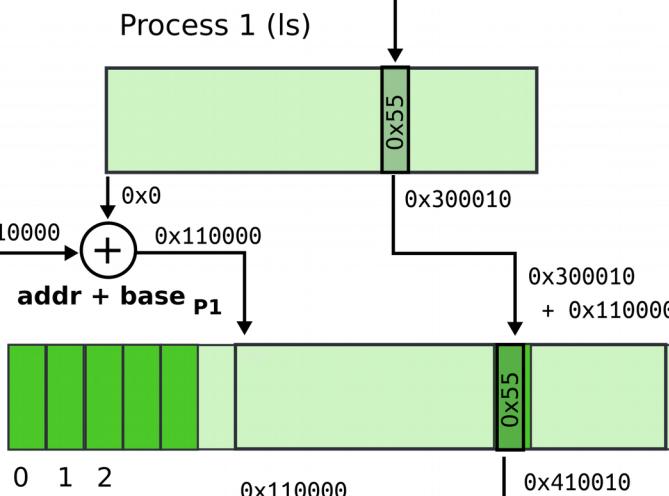
0x1

mov (%EBX), EAX # mov value from the location pointed by EBX into EAX  
EAX = 0x0  
EBX = 0x300010  
DS = 0x1

Global Descriptor Table  
(table of segment sizes and bases)

0x0
0x110000
0x510000
...
0x55

base<sub>P1</sub>



Virtual Address Space  
(aka Virtual Memory,  
aka Linear Address in Intel's terms)

Physical Memory

0x7095

0x0  
0x7095

0xc010

- Each process has a private GDT
- OS will switch between GDTs

GDT register  
(pointer to the GDT,  
physical address)

# Paging

```
mov (%EBX), EAX # mov value from the location pointed by EBX into EAX
```

EAX = 0

EBX = 0x300010

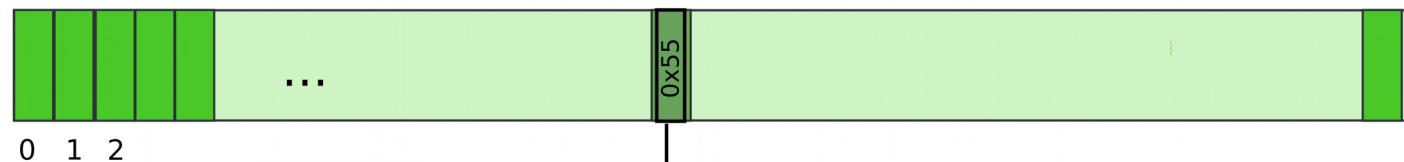
DS = 0x1

Linear address for 0x300010 is 0x410010

0x410010 =   
page number

1M (1,048,575)

Virtual Address Space  
(aka Virtual Memory)



0 1 2

...

0x55

page number = 1040 or 0x410  
or (0b 100 0001 0000)

CR3 = 0x2

0 1 2 3 4 5 6 7 8 9 10 11 12

Physical  
Memory

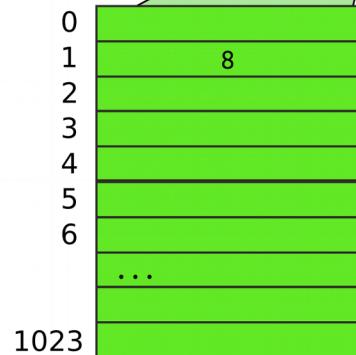


...

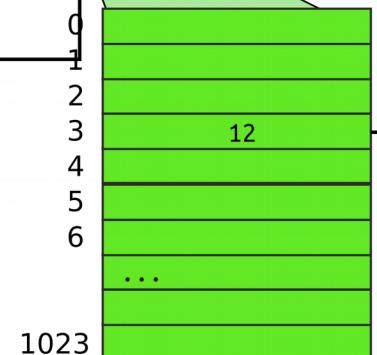
...

Physical page #12 or 0xc

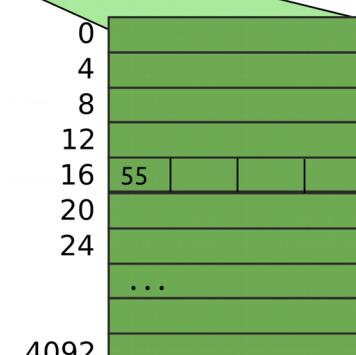
32 bits (4 bytes)



Level 1  
(Page Table  
Directory)

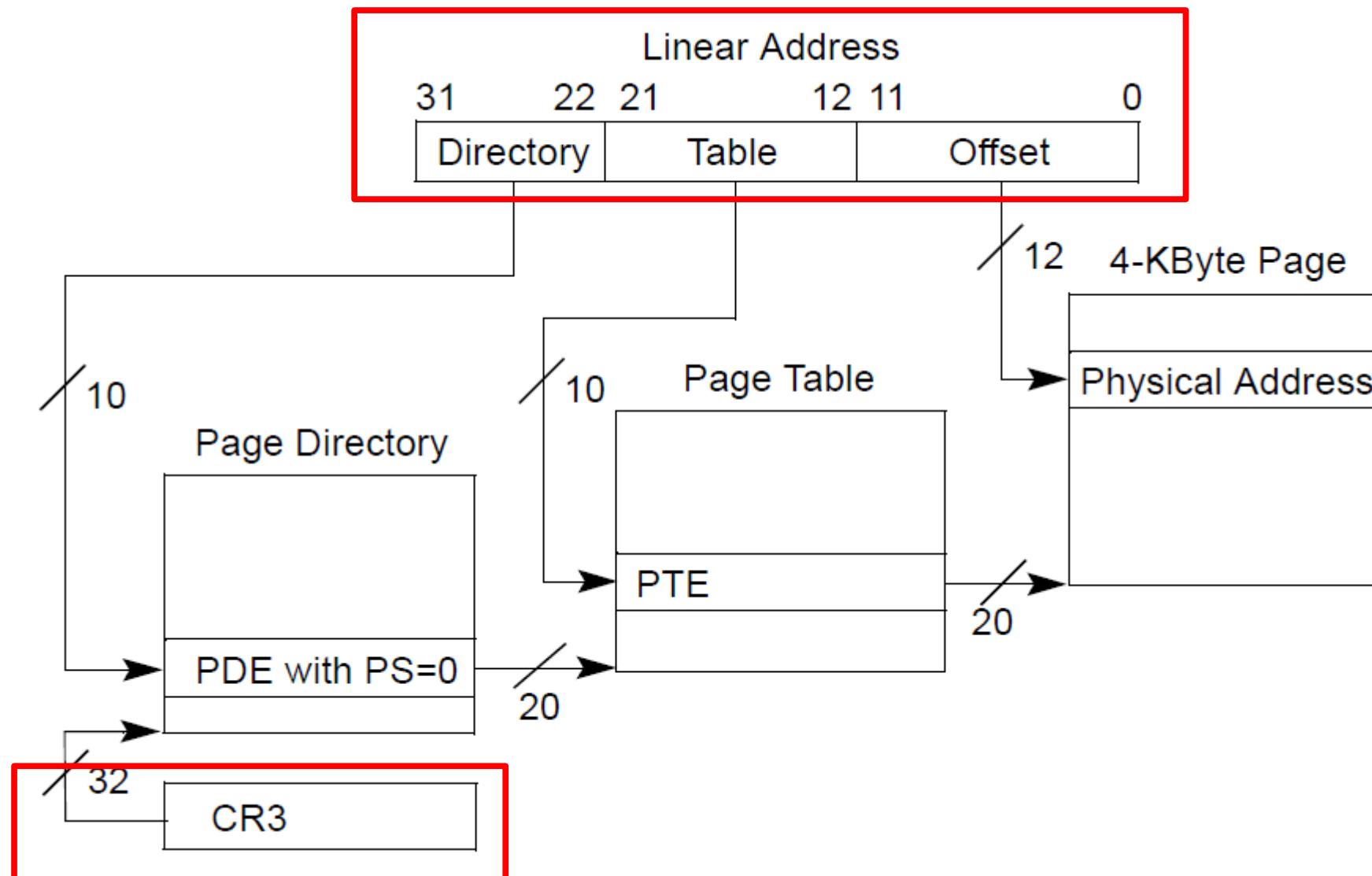


Level 2  
(Page Table)

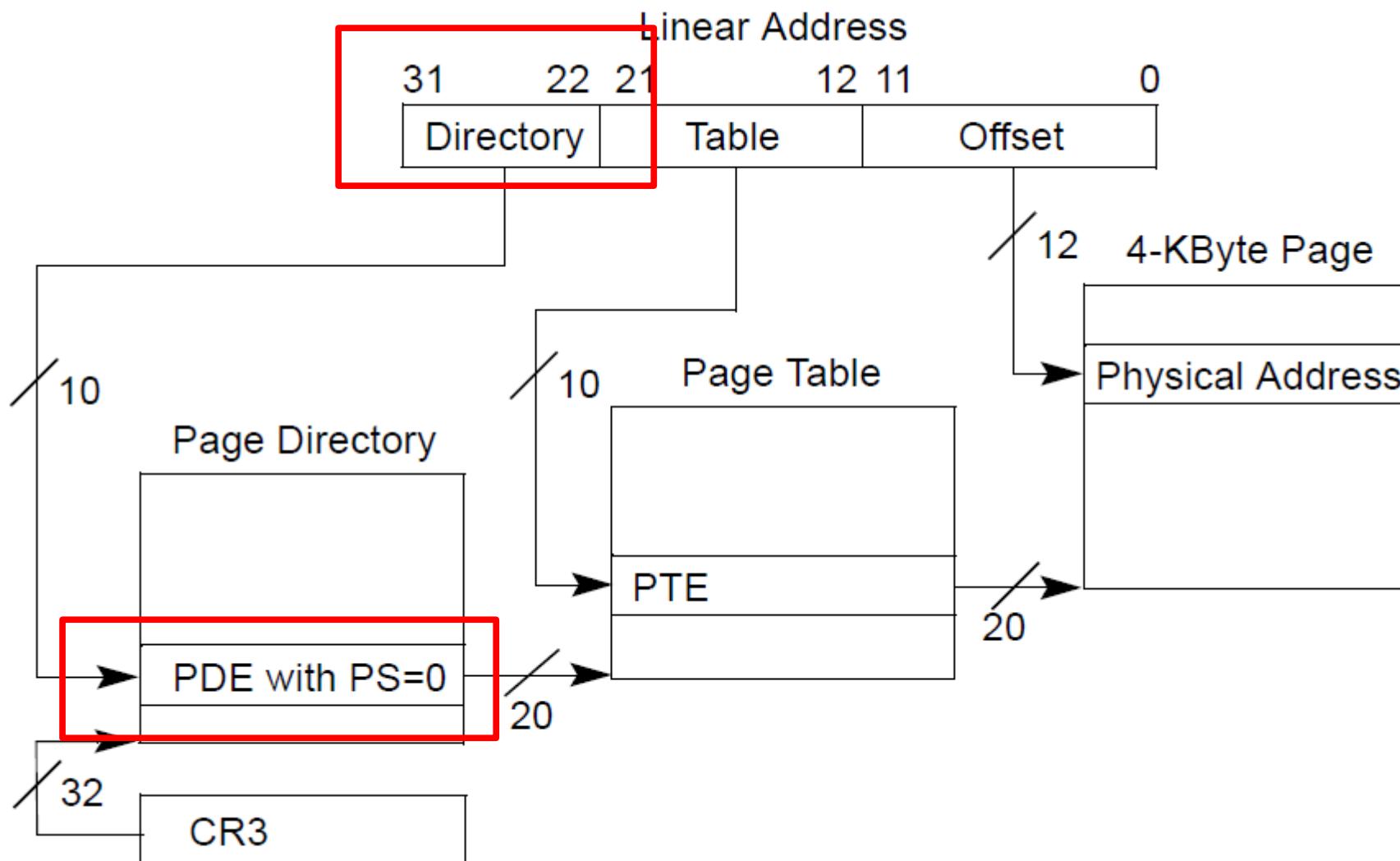


Page

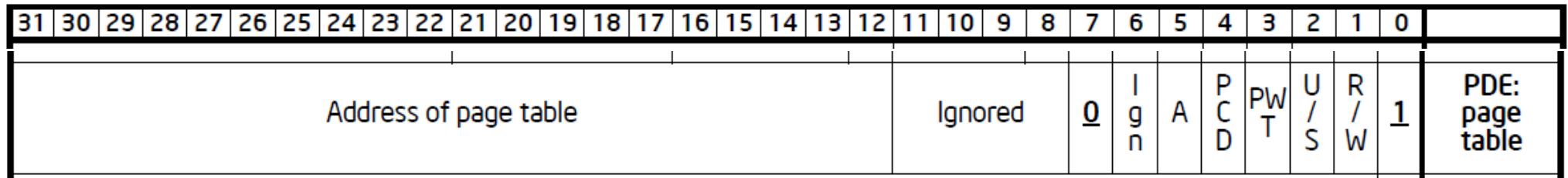
# Page translation



# Page translation

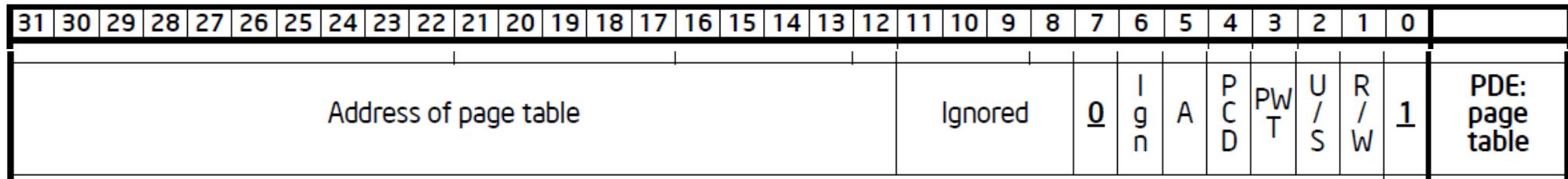


# Page directory entry (PDE)



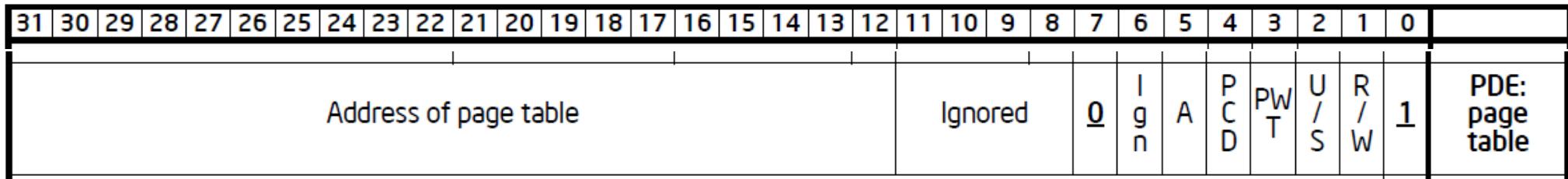
- 20 bit address of the page table

# Page directory entry (PDE)



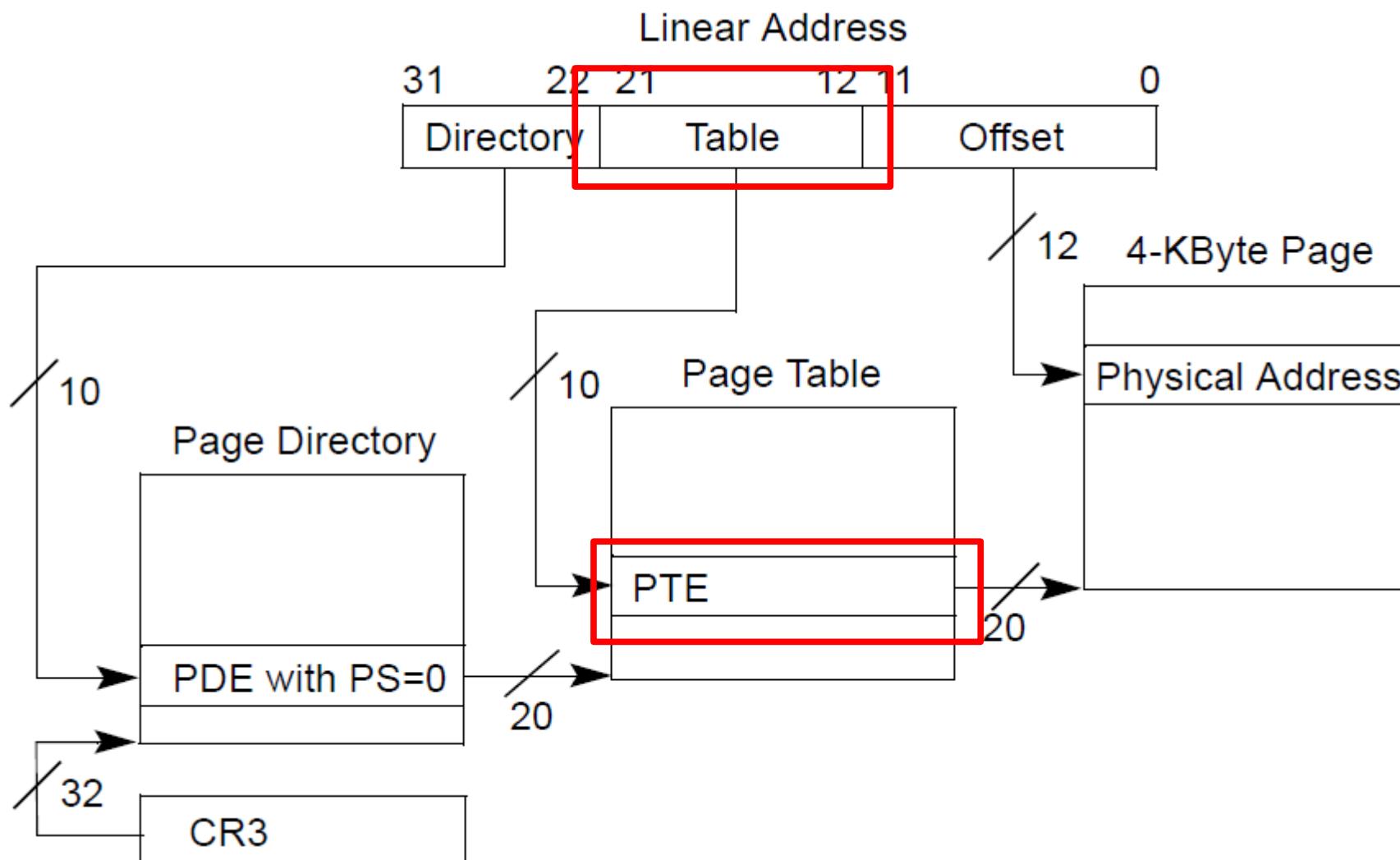
- 20 bit address of the page table
- Wait... 20 bit address, but we need 32 bits

# Page directory entry (PDE)

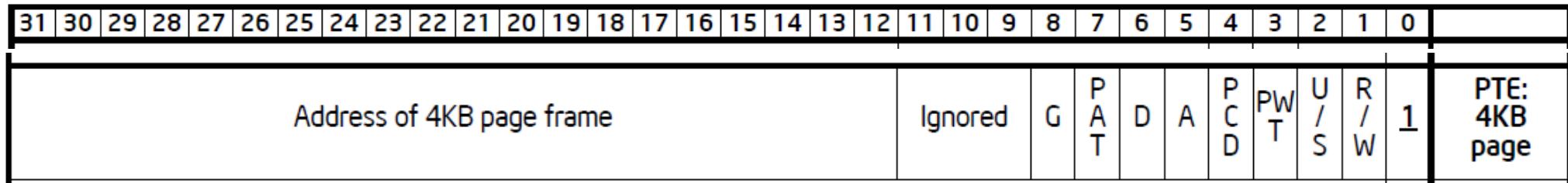


- 20 bit address of the page table
- Wait... 20 bit address, but we need 32 bits
  - Pages 4KB each, we need 1M to cover 4GB
  - Pages start at 4KB (page aligned boundary)

# Page translation

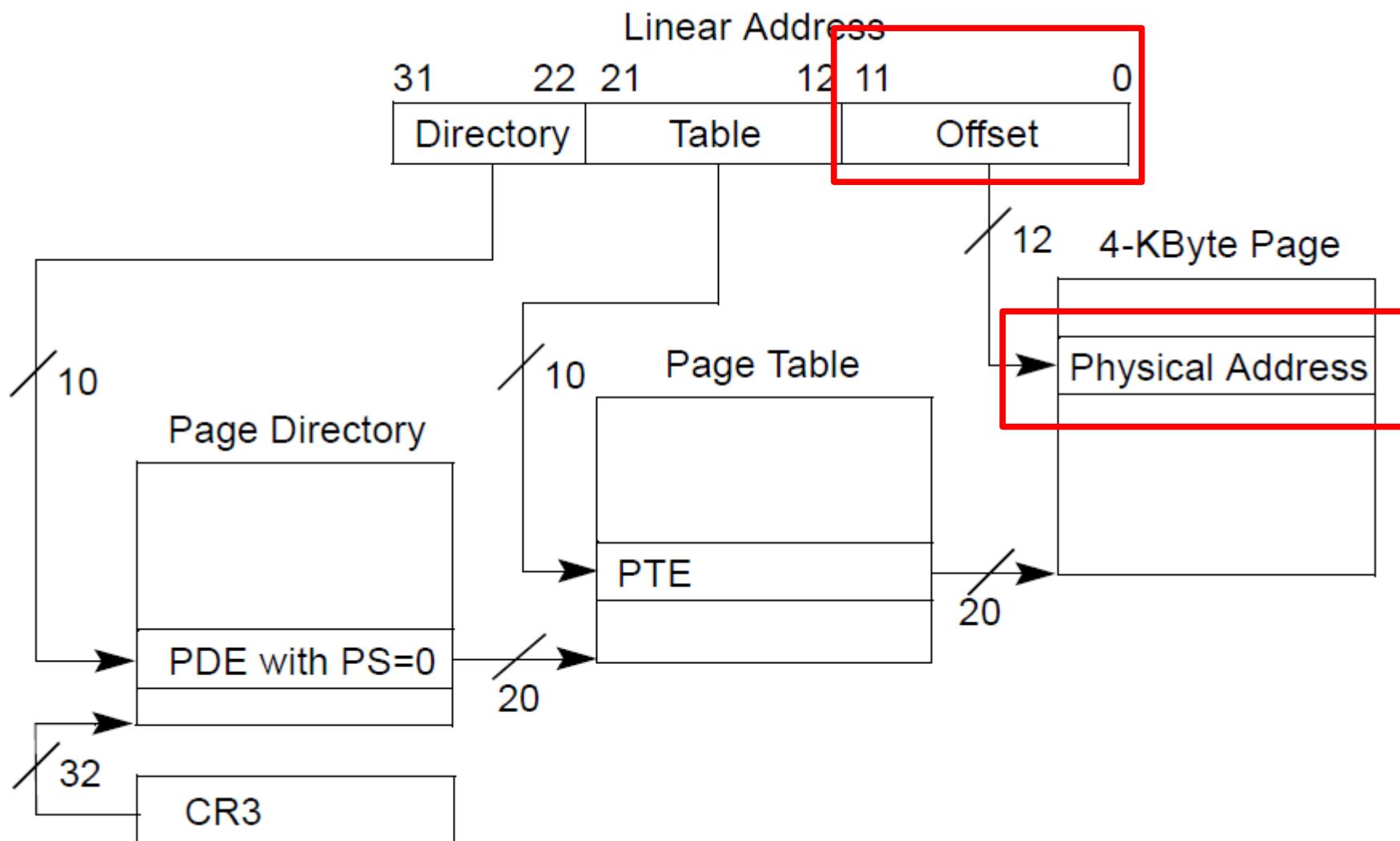


# Page table entry (PTE)



- 20 bit address of the 4KB page
  - Pages 4KB each, we need 1M to cover 4GB

# Page translation



```
mov (%EBX), EAX # mov value from the location pointed by EBX into EAX  
EAX = 0  
EBX = 20 983 809
```

20 983 809 = 

  
page number

1M (1,048,575)

Virtual Address  
Space (or Memory)  
of the Process

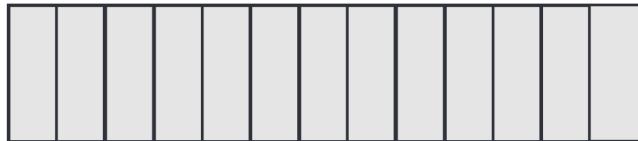


0 1 2

page number = 5123  
or (0b1 0100 0000 0011)

0 1 2 3 4 5 6 7 8 9 10 11 12

Physical  
Memory



```
mov (%EBX), EAX # mov value from the location pointed by EBX into EAX  
EAX = 0  
EBX = 20 983 809
```

20 983 809 = **00 0000 0101|00 0000 0011|0000 0000 0001**

page number

1M (1,048,575)

Virtual Address  
Space (or Memory)  
of the Process



0 1 2

page number = 5123  
or (0b1 0100 0000 0011)

CR3 = 0

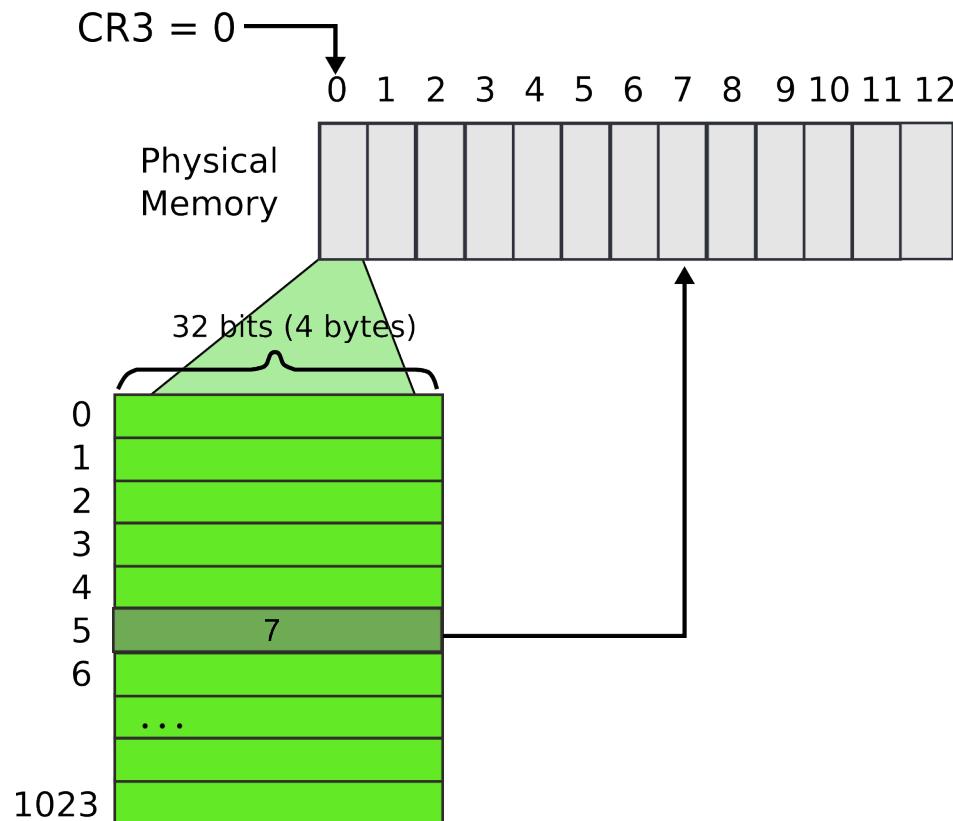
0 1 2 3 4 5 6 7 8 9 10 11 12

Physical  
Memory



```
mov (%EBX), EAX # mov value from the location pointed by EBX into EAX  
EAX = 0  
EBX = 20 983 809
```

20 983 809 =   
 page number 1M (1,048,575)



Level 1  
(Page Table  
Directory)

```

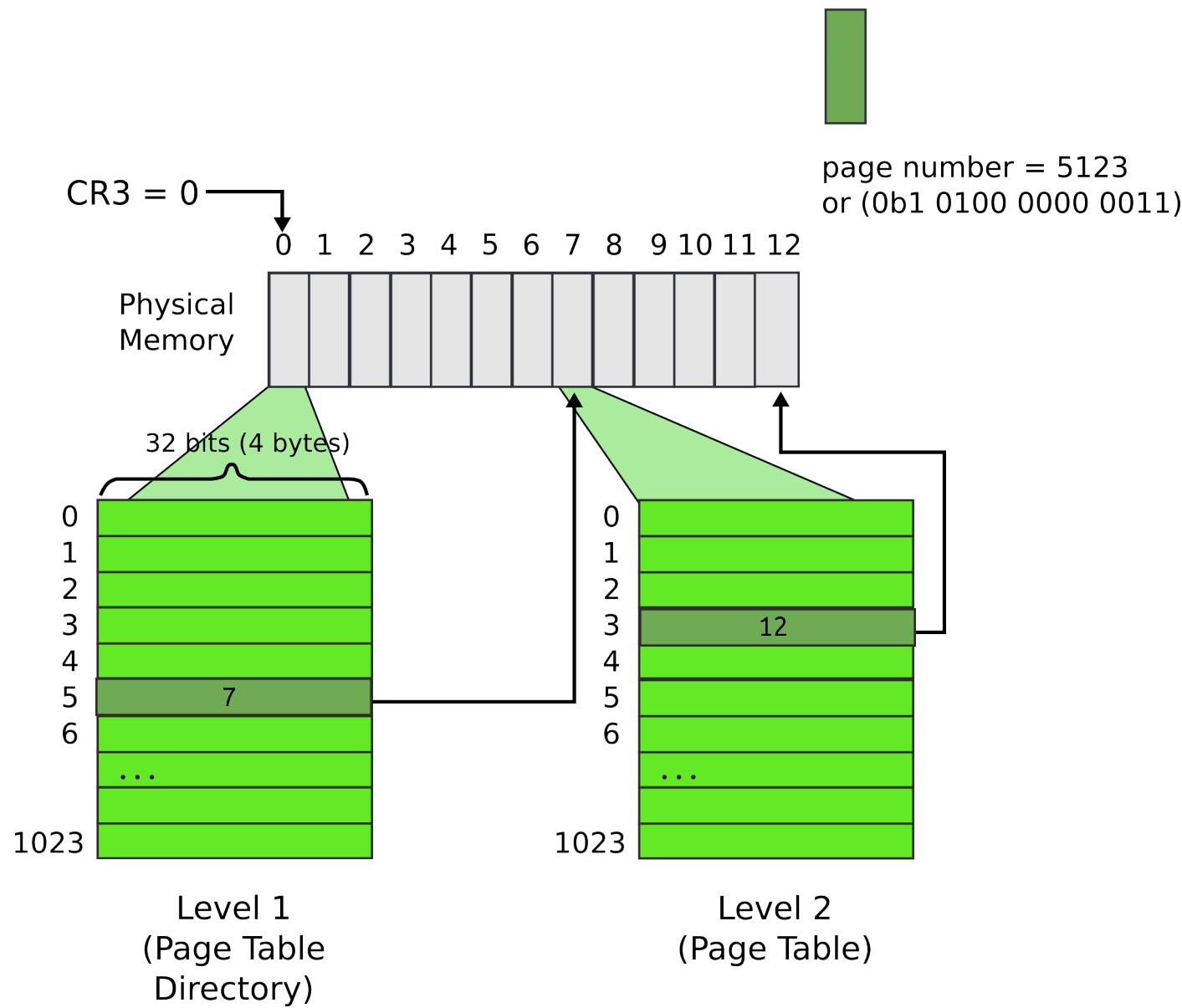
mov (%EBX), EAX # mov value from the location pointed by EBX into EAX
EAX = 0
EBX = 20 983 809

```

$20\ 983\ 809 = \boxed{00\ 0000\ 010} \boxed{00\ 0000\ 0011} \boxed{0000\ 0000\ 0001}$

page number

1M (1,048,575)



```

mov (%EBX), EAX # mov value from the location pointed by EBX into EAX
EAX = 0
EBX = 20 983 809

```

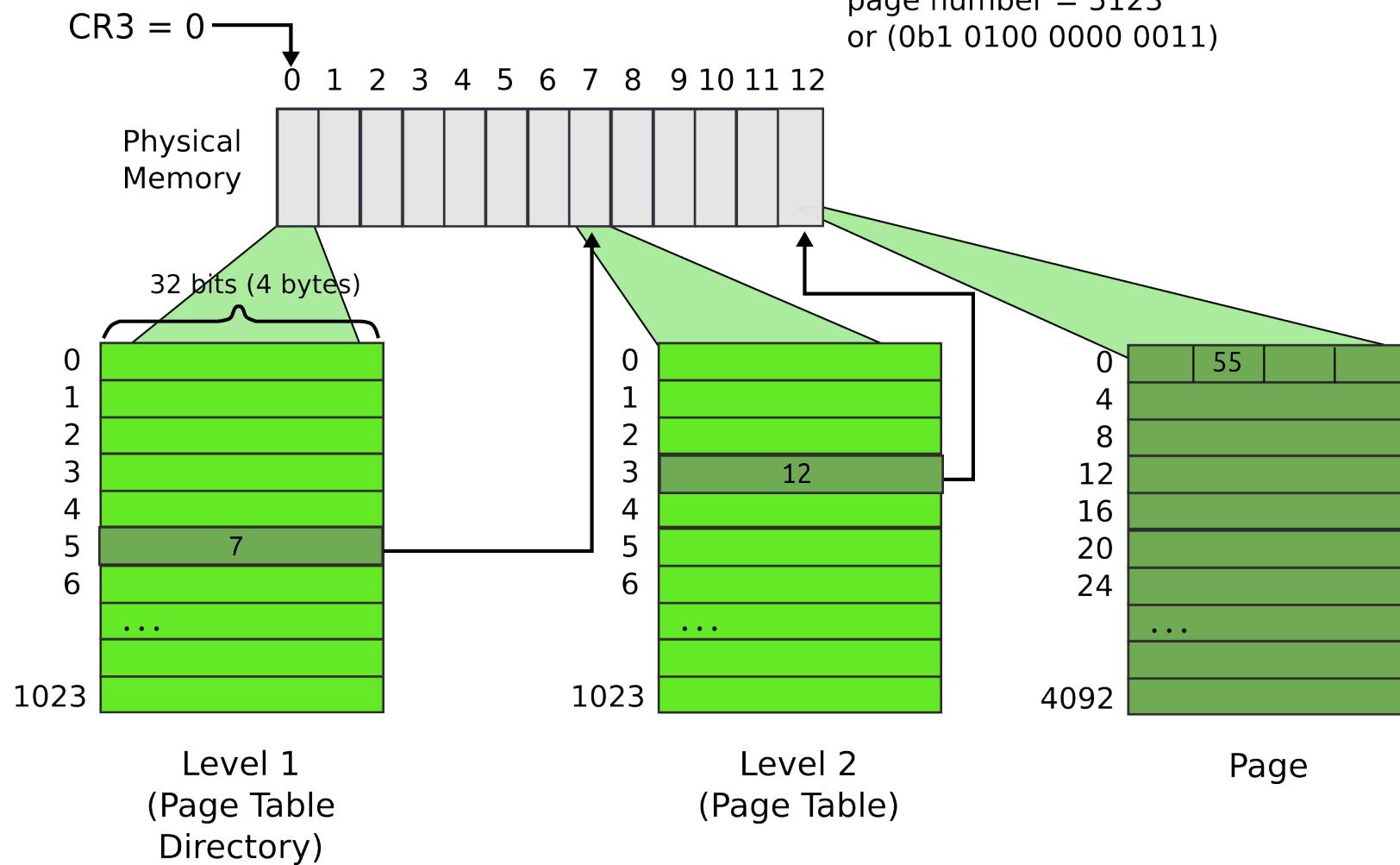
$20\ 983\ 809 = \boxed{00\ 0000\ 0101} \boxed{00\ 0000\ 0011} \boxed{0000\ 0000\ 0001}$

page number

1M (1,048,575)



page number = 5123  
or (0b1 0100 0000 0011)



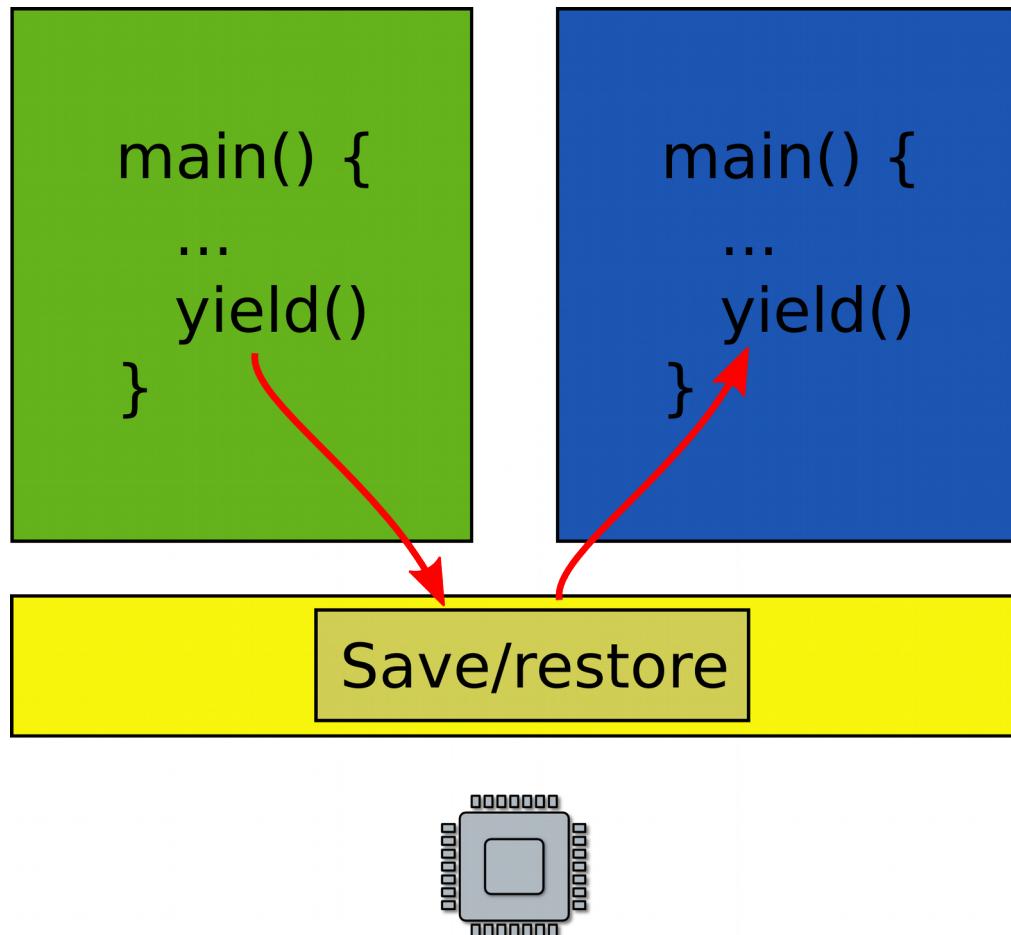
- Result:
  - $EAX = 55$

# Benefit of page tables

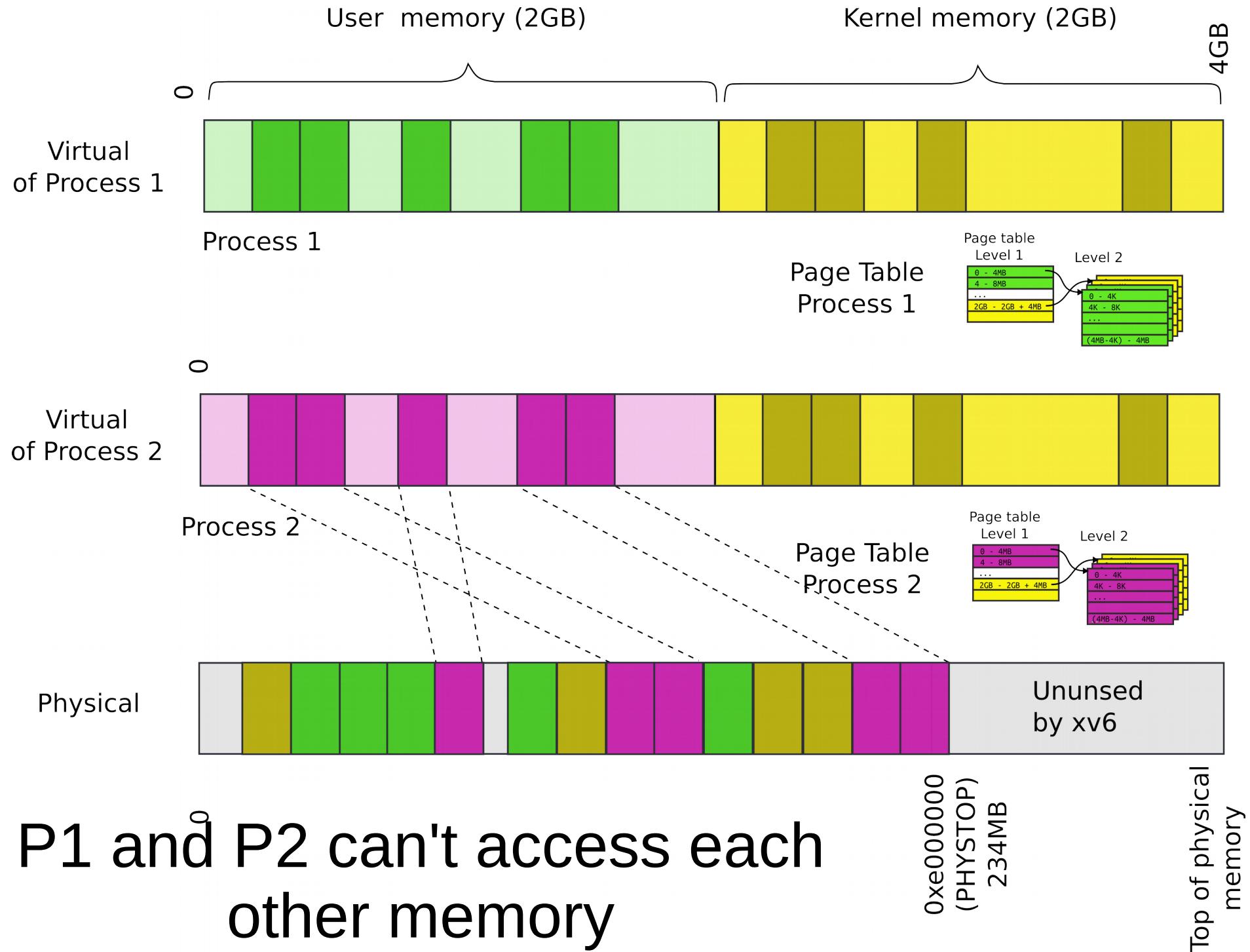
... Compared to arrays?

- Page tables represent sparse address space more efficiently
  - An entire array has to be allocated upfront
  - But if the address space uses a handful of pages
  - Only page tables (Level 1 and 2 need to be allocated to describe translation)
- On a dense address space this benefit goes away
  - I'll assign a homework!

# What about isolation?



- Two programs, one memory?
- Each process has its own page table
  - OS switches between them

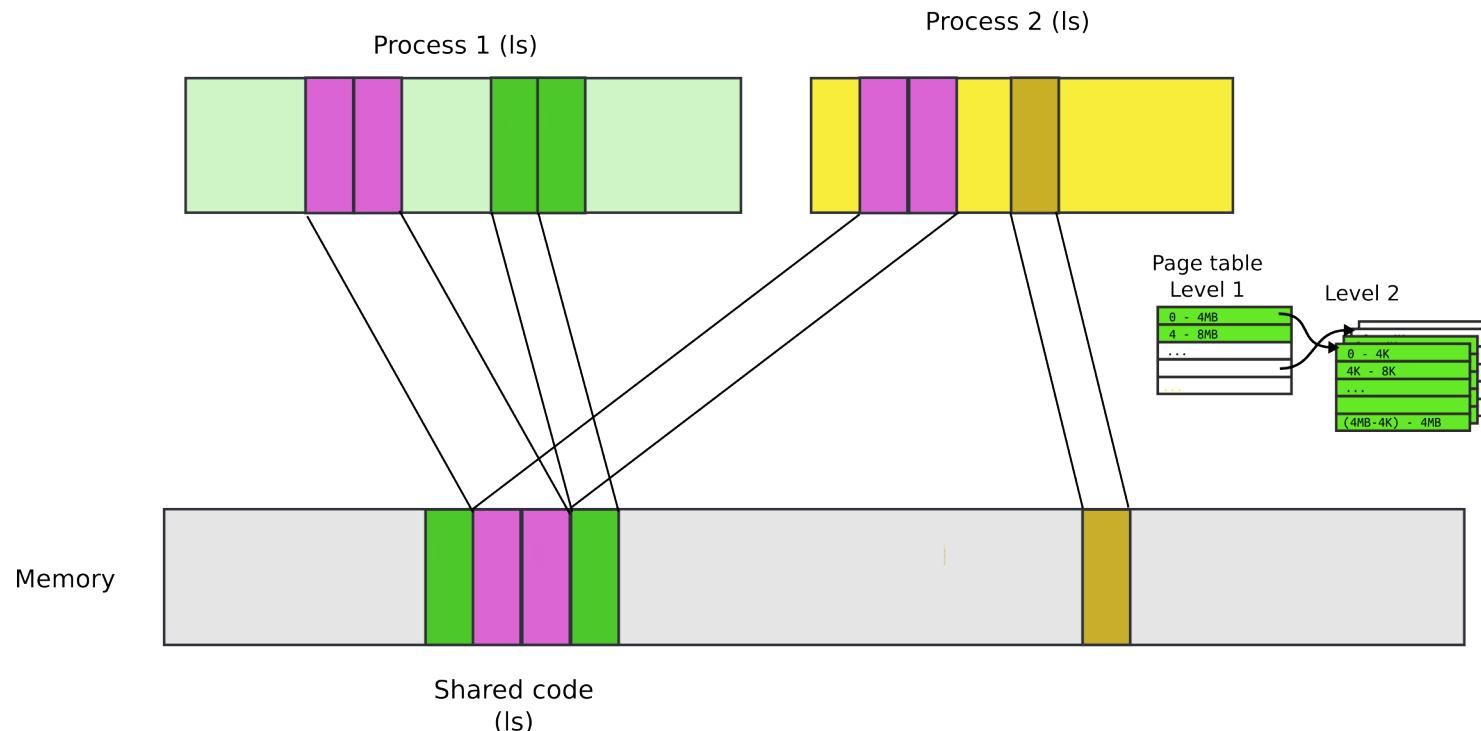


# Compared to segments pages allow ...

- Emulate large virtual address space on a smaller physical memory
  - In our example we had only 12 physical pages
  - But the program can access all 1M pages in its 4GB address space
  - The OS will move other pages to disk

# Compared to segments pages allow ...

- Share a region of memory across multiple programs
  - Communication (shared buffer of messages)
  - Shared libraries



# More paging tricks

- Protect parts of the program
  - E.g., map code as read-only
    - Disable code modification attacks
    - Remember R/W bit in PTD/PTE entries!
  - E.g., map stack as non-executable
    - Protects from stack smashing attacks
    - Non-executable bit

# More paging tricks

- Determine a working set of a program?

# More paging tricks

- Determine a working set of a program?
  - Use “accessed” bit

# More paging tricks

- Determine a working set of a program?
  - Use “accessed” bit
- Iterative copy of a working set?
  - Used for virtual machine migration

# More paging tricks

- Determine a working set of a program?
  - Use “accessed” bit
- Iterative copy of a working set?
  - Used for virtual machine migration
  - Use “dirty” bit

# More paging tricks

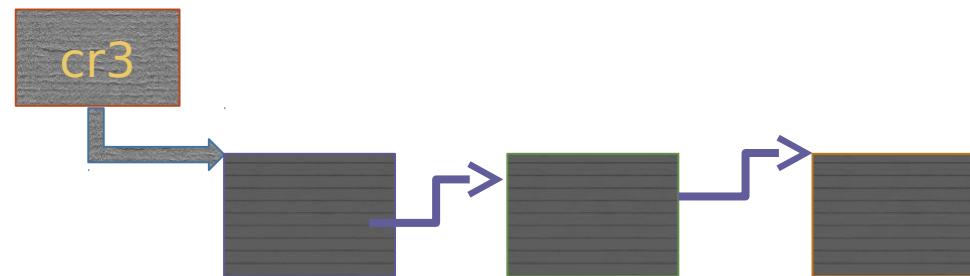
- Determine a working set of a program?
  - Use “accessed” bit
- Iterative copy of a working set?
  - Used for virtual machine migration
  - Use “dirty” bit
- Copy-on-write memory, e.g. lightweight fork()?

# More paging tricks

- Determine a working set of a program?
  - Use “accessed” bit
- Iterative copy of a working set?
  - Used for virtual machine migration
  - Use “dirty” bit
- Copy-on-write memory, e.g. lightweight fork()?
  - Map page as read/only

# TLB

- Walking page table is slow
  - Each memory access is 240 (local) - 360 (one QPI hop away) cycles on modern hardware
  - L3 cache access is 50 cycles



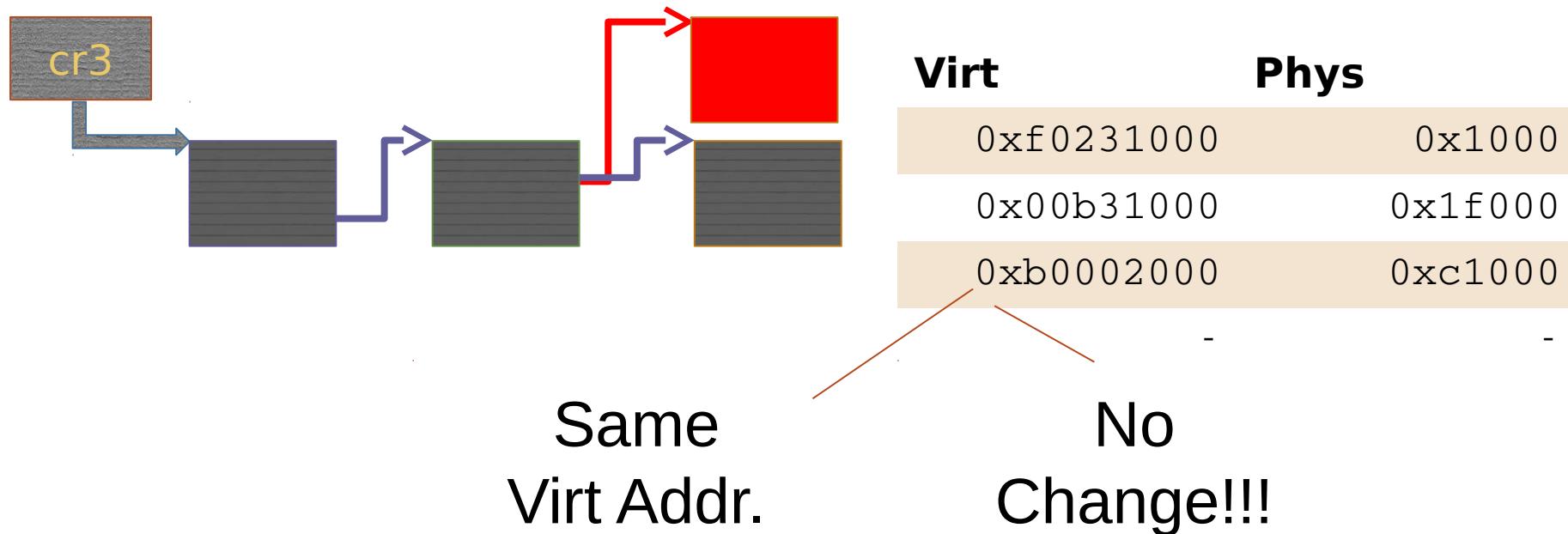
# TLB

- CPU caches results of page table walks
  - In translation lookaside buffer (TLB)

<b>Virt</b>	<b>Phys</b>
0xf0231000	0x1000
0x00b31000	0x1f000
0xb0002000	0xc1000

# TLB invalidation

- TLB is a cache (in CPU)
  - It is not coherent with memory
  - If page table entry is changes, TLB remains the same and is out of sync



# TLB invalidation

- After every page table update, OS needs to manually invalidate cached values
  - Flush TLB
    - Either one specific entry
    - Or entire TLB, e.g., when CR3 register is loaded
    - This happens when OS switches from one process to another
  - This is expensive
    - Refilling the TLB with new values takes time

# Tagged TLBs

- Modern CPUs have “tagged TLBs”,
  - Each TLB entry has a “tag” – identifier of a process
  - No need to flush TLBs on context switch
- On Intel this mechanism is called
  - Process-Context Identifiers (PCIDs)

Virt	Phys	Tag
0xf0231 000	0x1000	P1
0x00b31 000	0x1f000	P2
0xb0002 000	0xc1000	P1

# When would you disable paging?

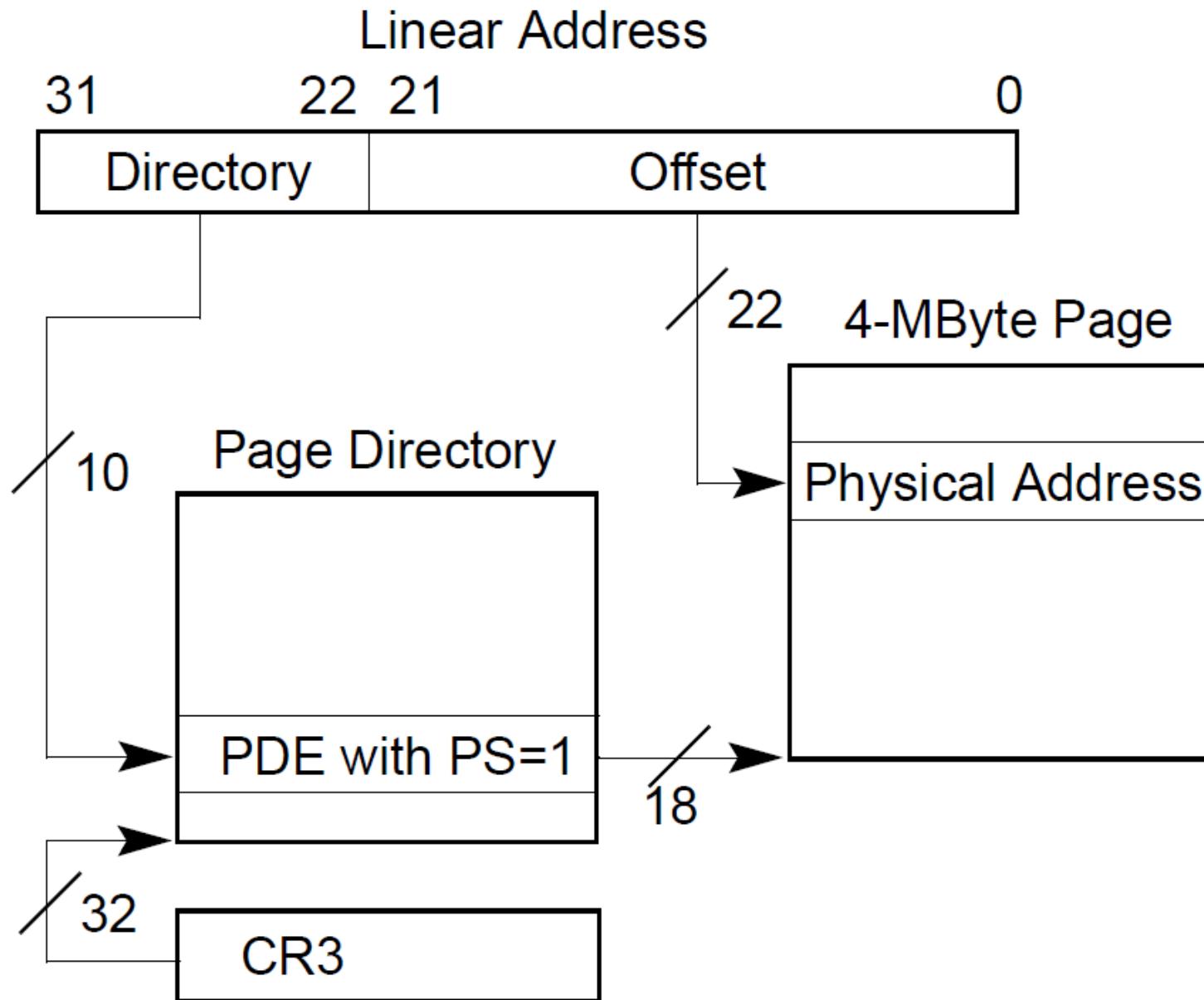
# When would you disable paging?

- Imagine you're running a memcached
  - Key/value cache
- You serve 1024 byte values (typical) on 10Gbps connection
  - 1024 byte packets can leave every 835ns, or 1670 cycles (2GHz machine)
  - This is your target budget per packet
-

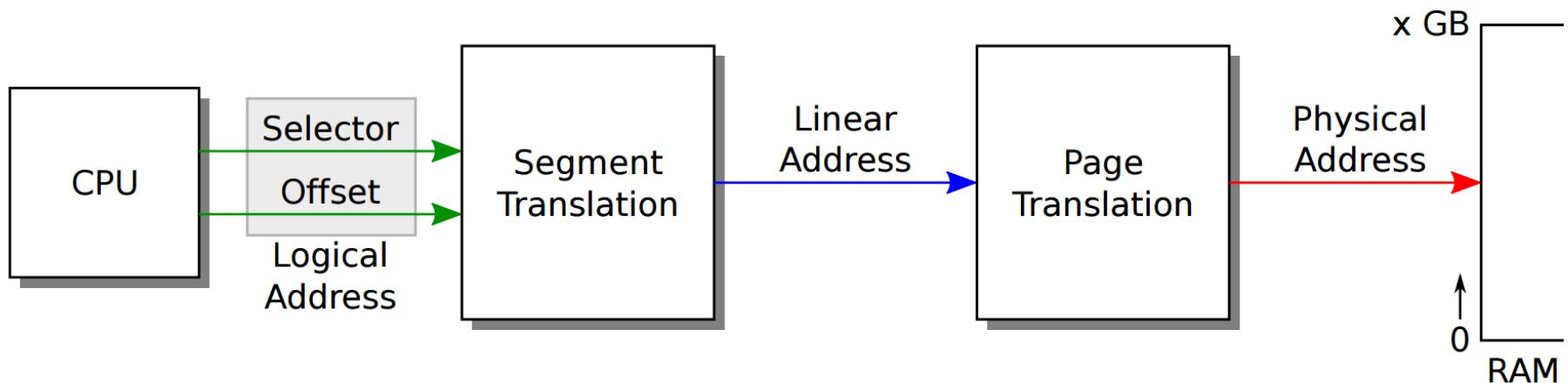
# When would you disable paging?

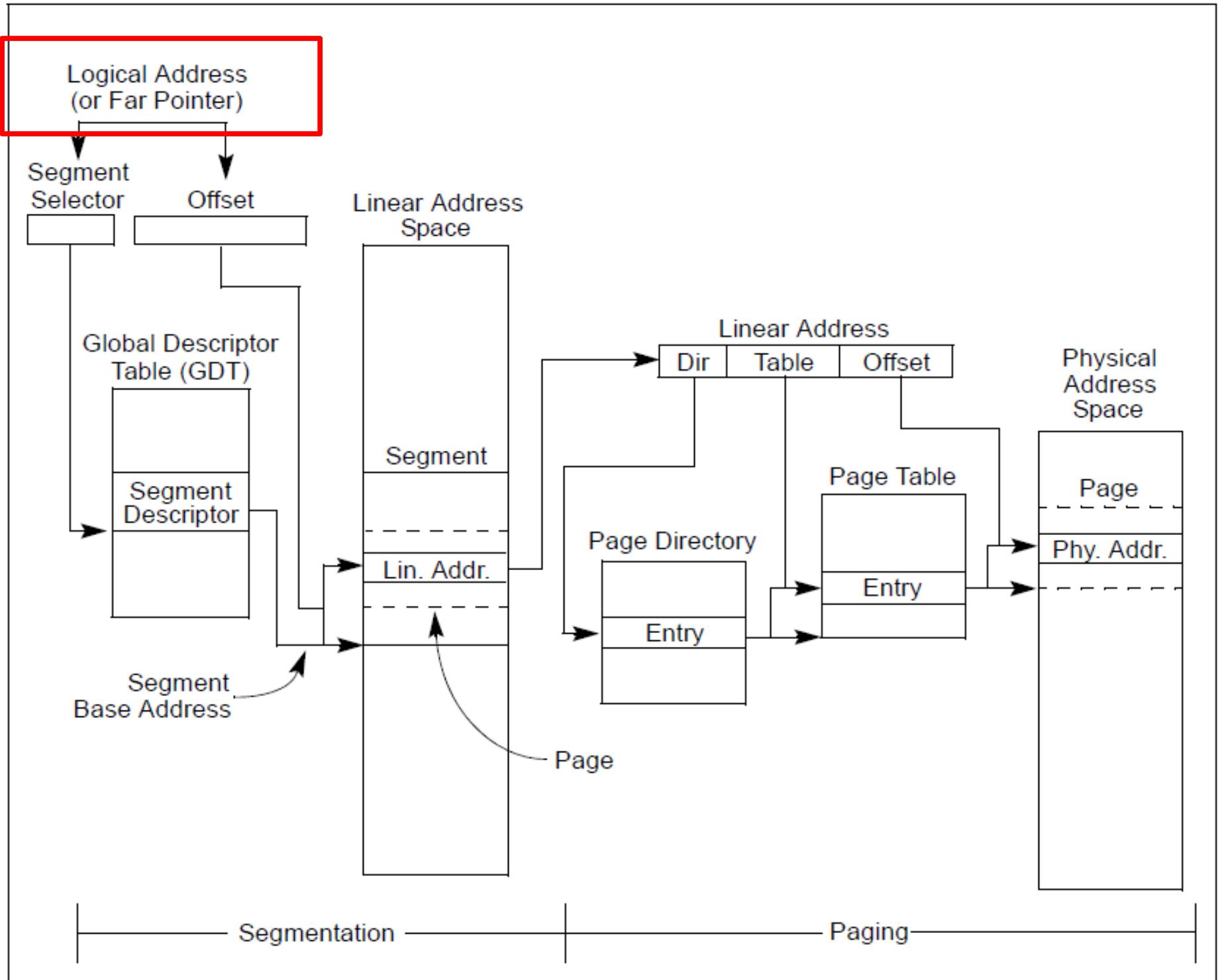
- Now, to cover 32GB RAM with 4K pages
  - You need 64MB space
  - 64bit architecture, 3-level page tables
- Page tables do not fit in L3 cache
  - Modern servers come with 32MB cache
- Every cache miss results in up to 3 cache misses due to page walk (remember 3-level page tables)
  - Each cache miss is 250 cycles
- Solution: 1GB pages

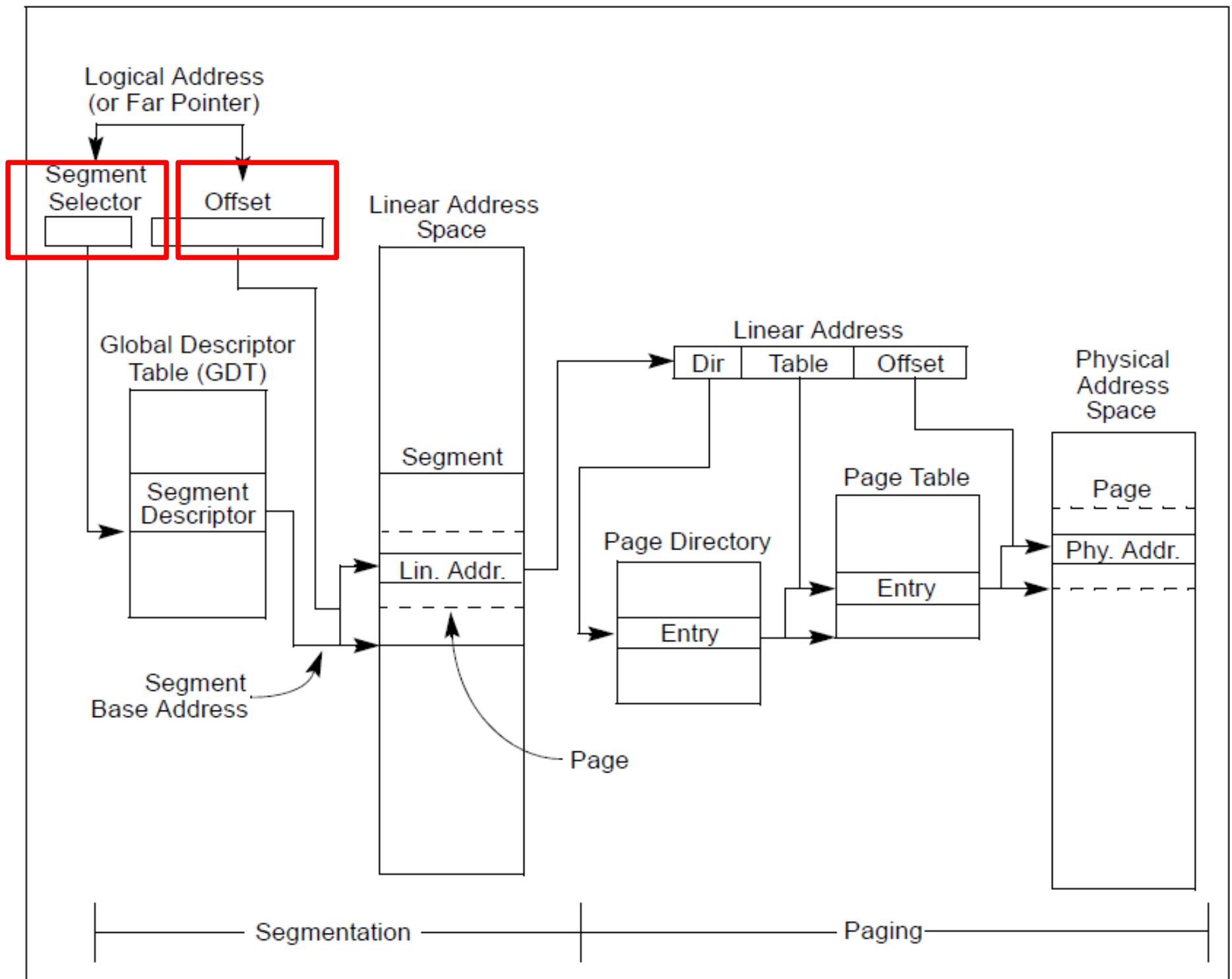
# Page translation for 4MB pages

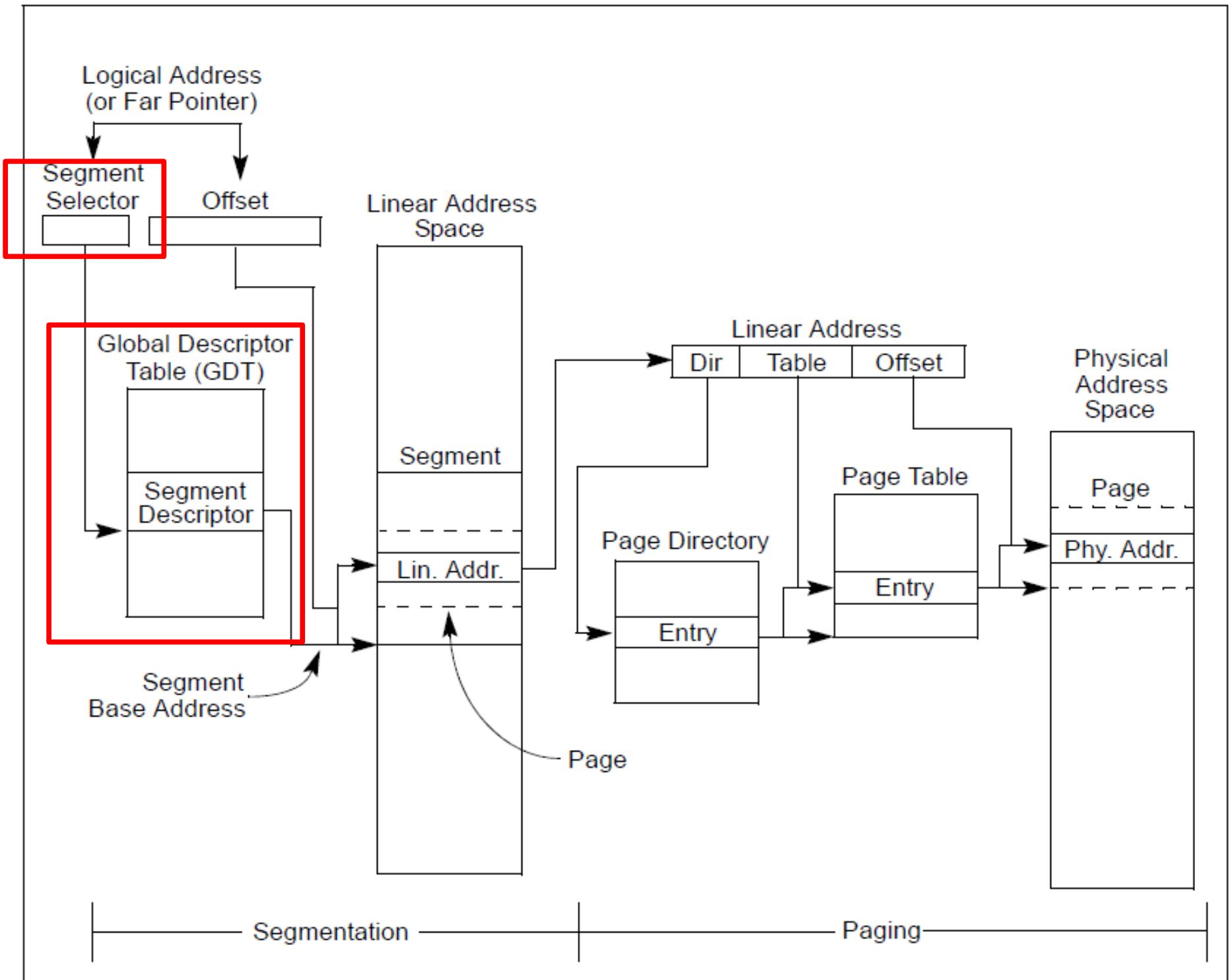


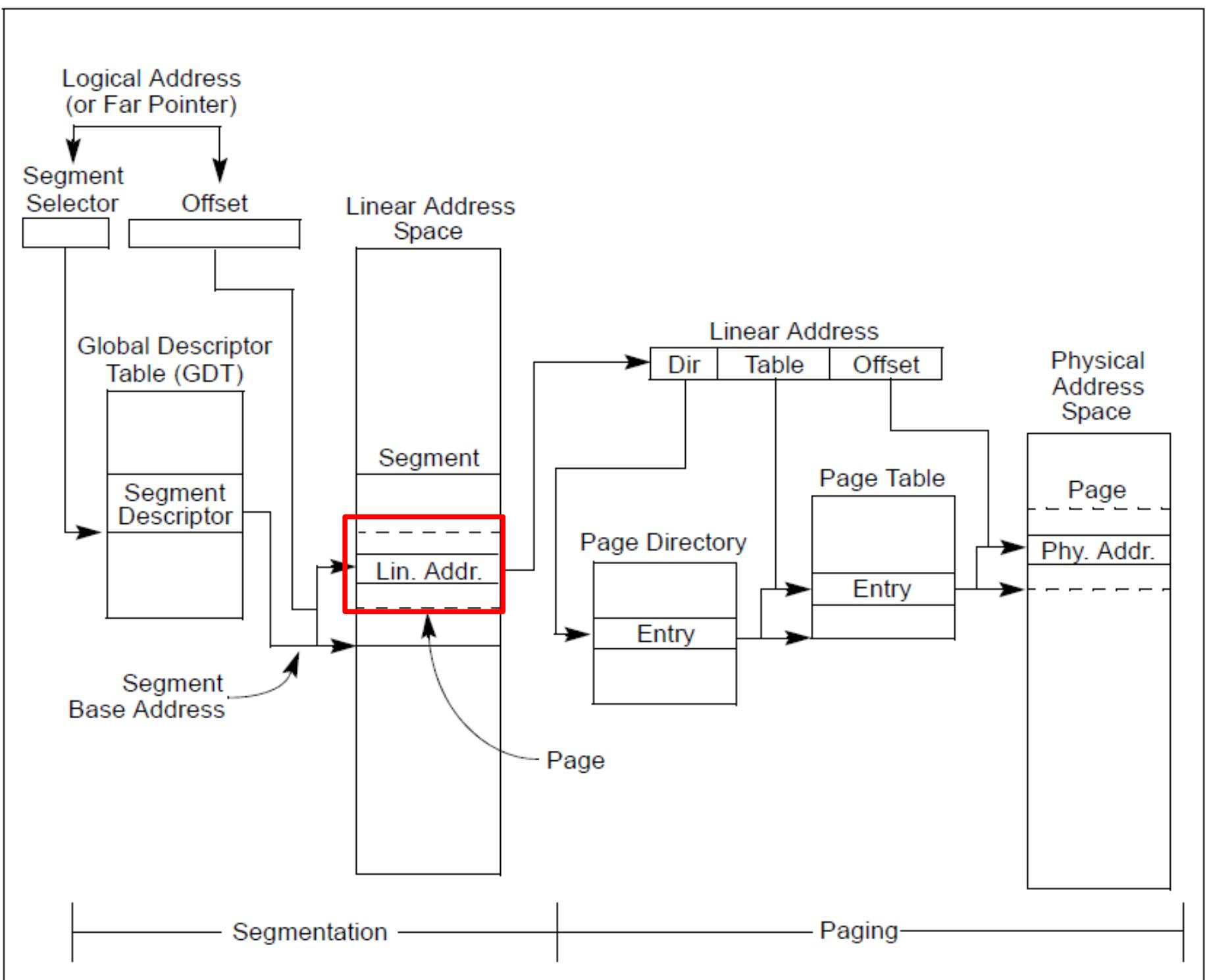
# Recap: complete address translation

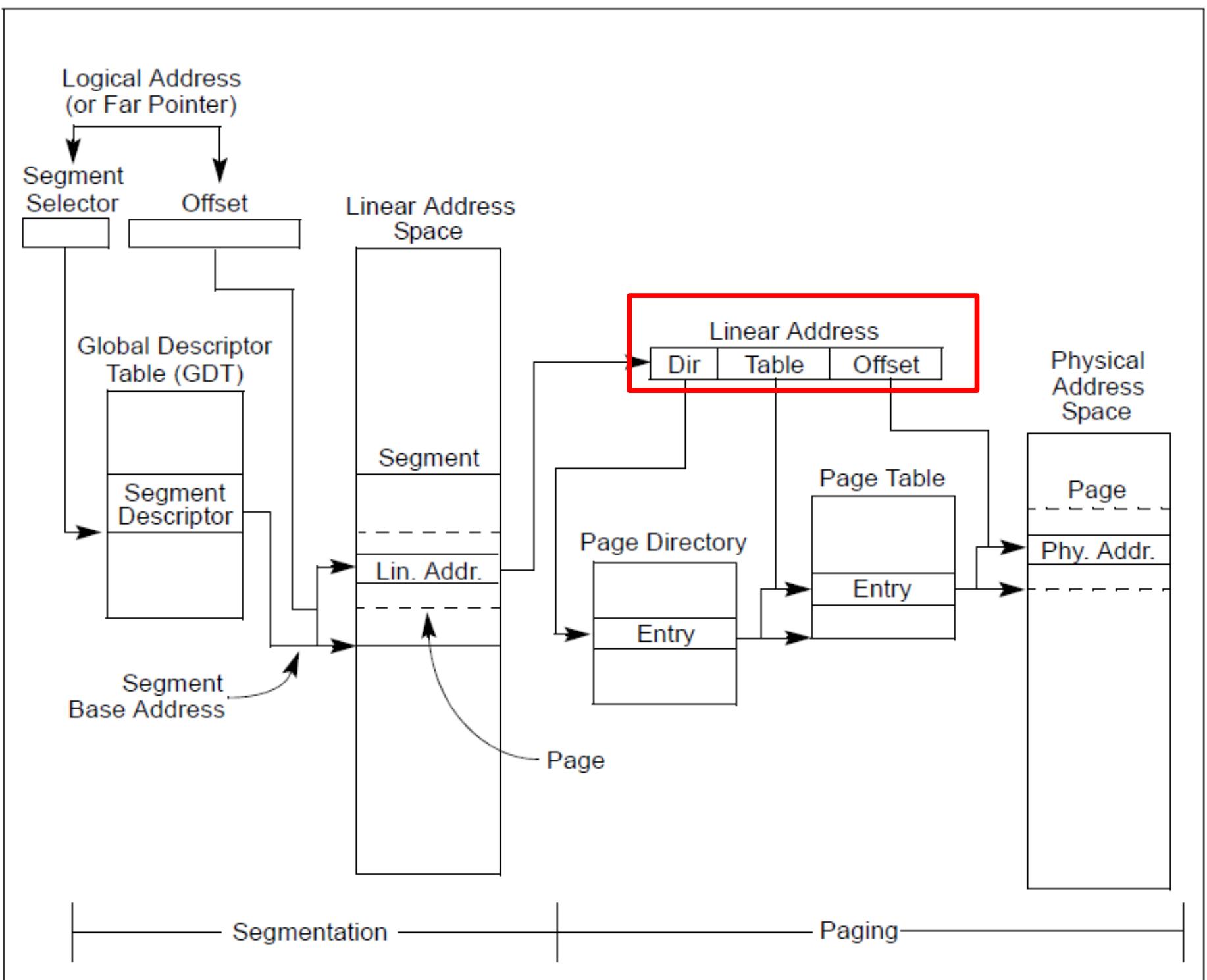


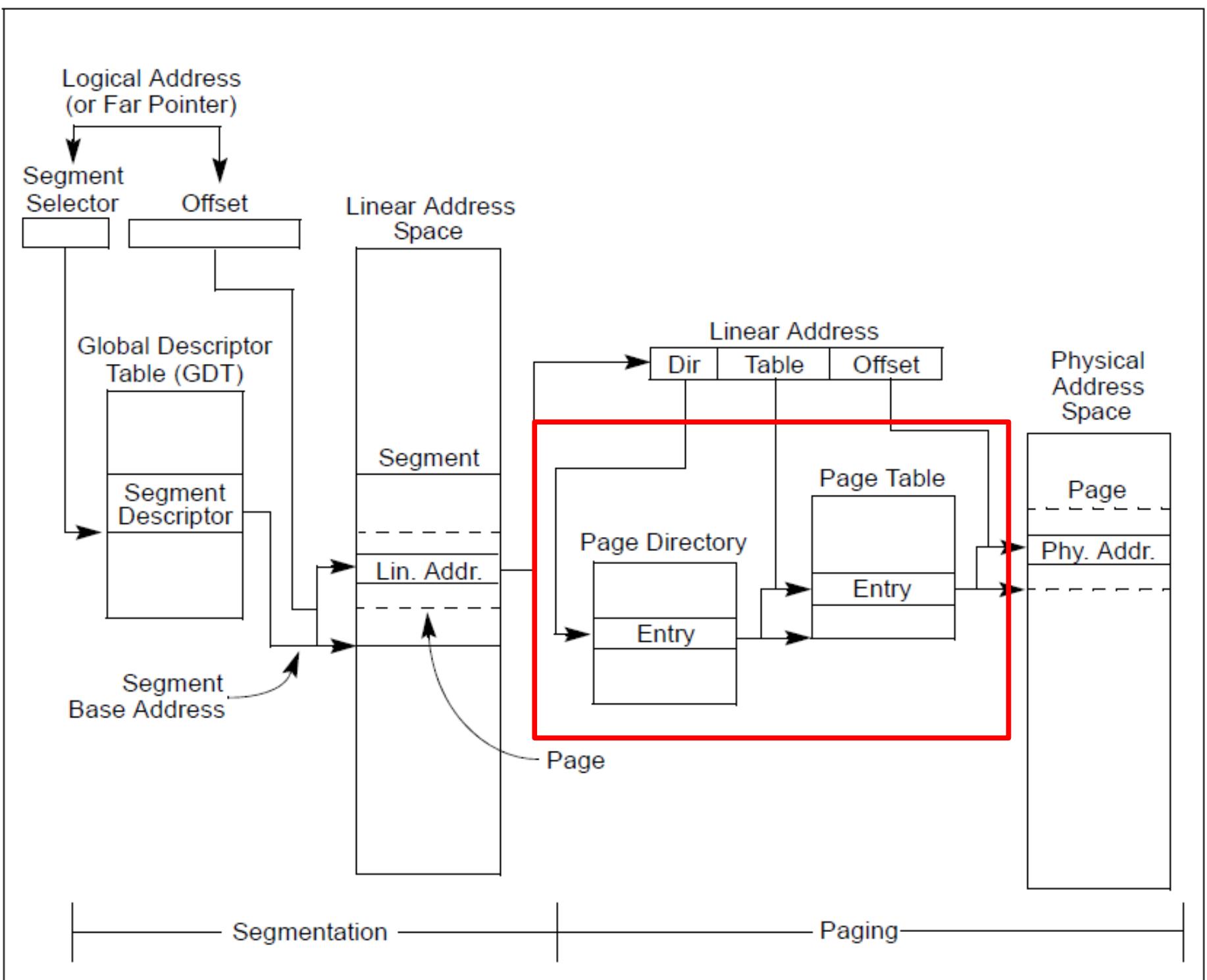


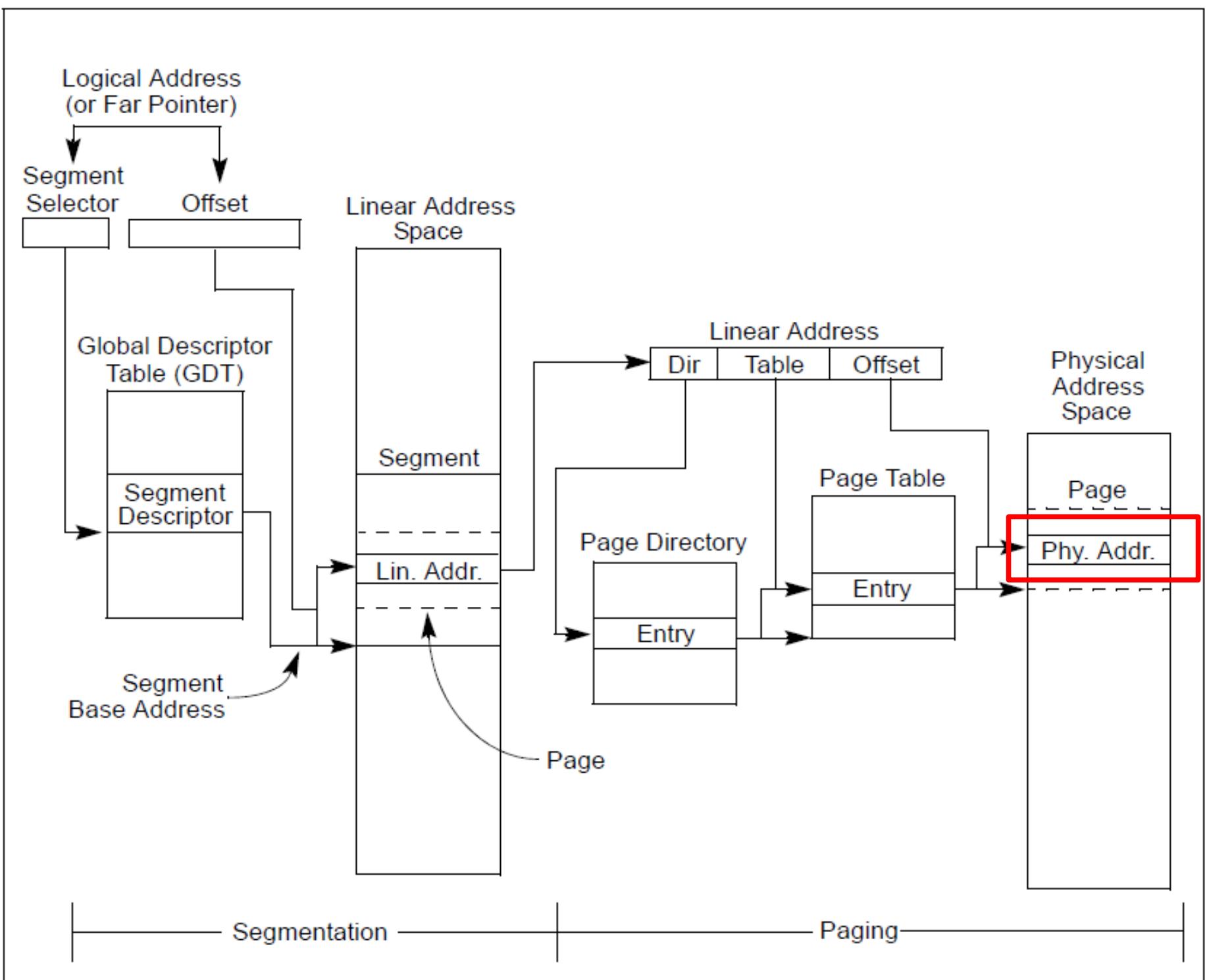


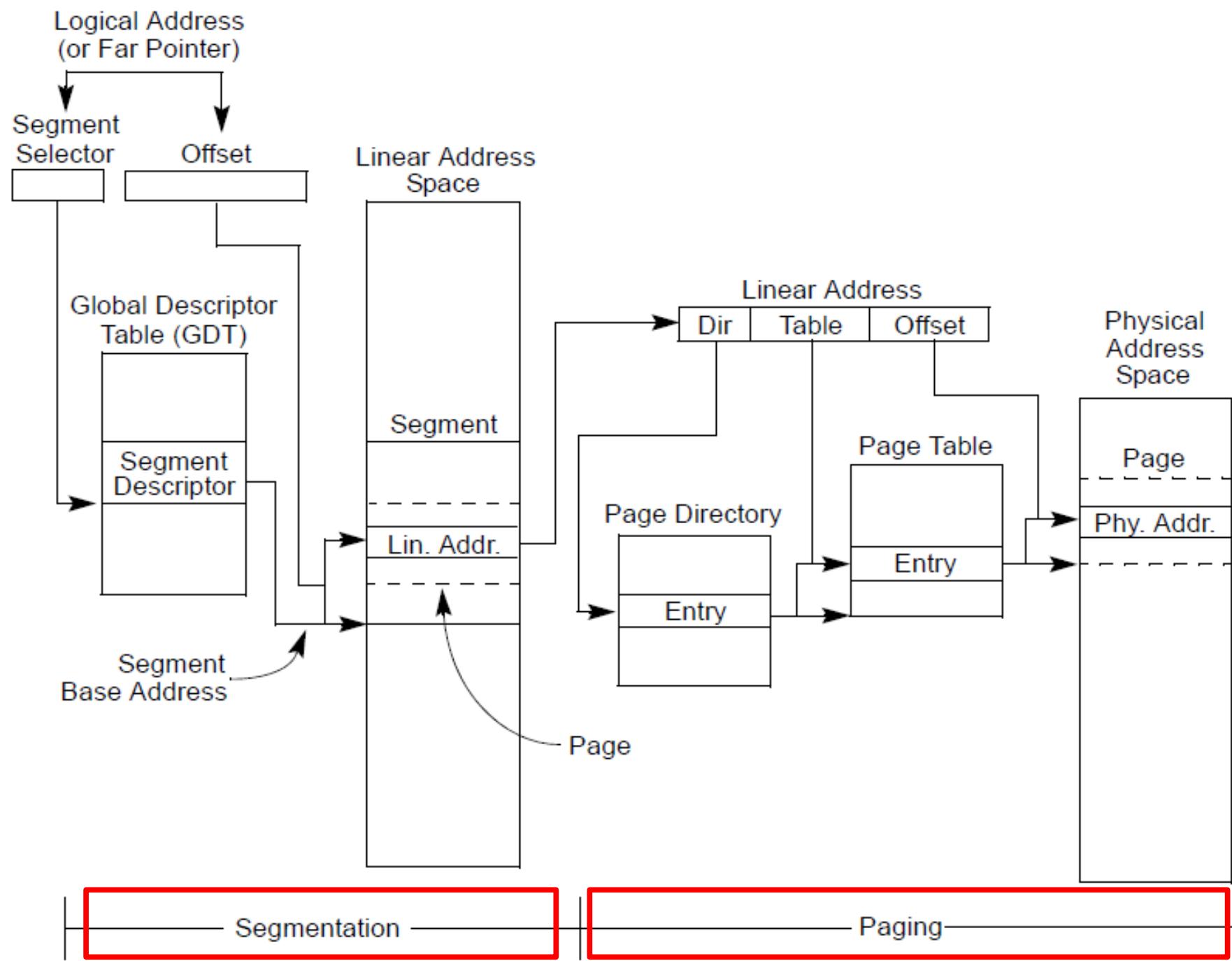












Questions?

# Back of the envelope

- If a page is 4K and an entry is 4 bytes, how many entries per page?

# Back of the envelope

- If a page is 4K and an entry is 4 bytes, how many entries per page?
  - 1k

# Back of the envelope

- If a page is 4K and an entry is 4 bytes, how many entries per page?
  - 1k
- How large of an address space can 1 page represent?

# Back of the envelope

- If a page is 4K and an entry is 4 bytes, how many entries per page?
  - 1k
- How large of an address space can 1 page represent?
  - $1\text{k entries} * 1\text{page/entry} * 4\text{K/page} = 4\text{MB}$

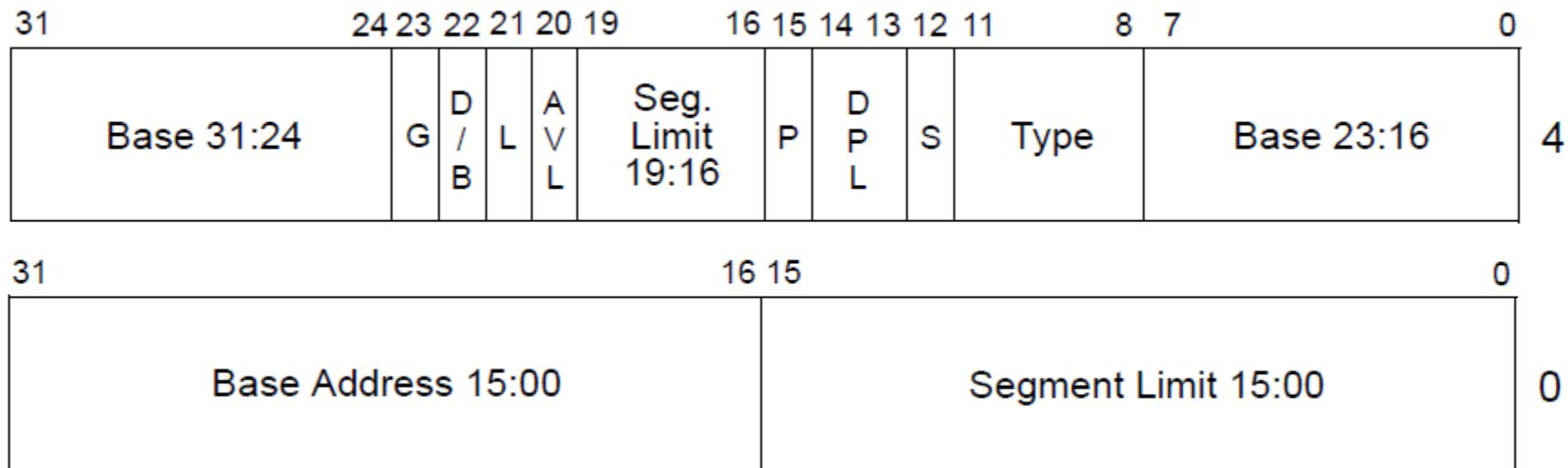
# Back of the envelope

- If a page is 4K and an entry is 4 bytes, how many entries per page?
  - 1k
- How large of an address space can 1 page represent?
  - $1\text{k entries} * 1\text{page/entry} * 4\text{K/page} = 4\text{MB}$
- How large can we get with a second level of translation?

# Back of the envelope

- If a page is 4K and an entry is 4 bytes, how many entries per page?
  - 1k
- How large of an address space can 1 page represent?
  - $1\text{k entries} * 1\text{page/entry} * 4\text{K/page} = 4\text{MB}$
- How large can we get with a second level of translation?
  - $1\text{k tables/dir} * 1\text{k entries/table} * 4\text{k/page} = 4 \text{ GB}$
  - Nice that it works out that way!

# Segment descriptors



L — 64-bit code segment (IA-32e mode only)

AVL — Available for use by system software

BASE — Segment base address

D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)

DPL — Descriptor privilege level

G — Granularity

LIMIT — Segment Limit

P — Segment present

S — Descriptor type (0 = system; 1 = code or data)

TYPE — Segment type