143A: Principles of Operating Systems

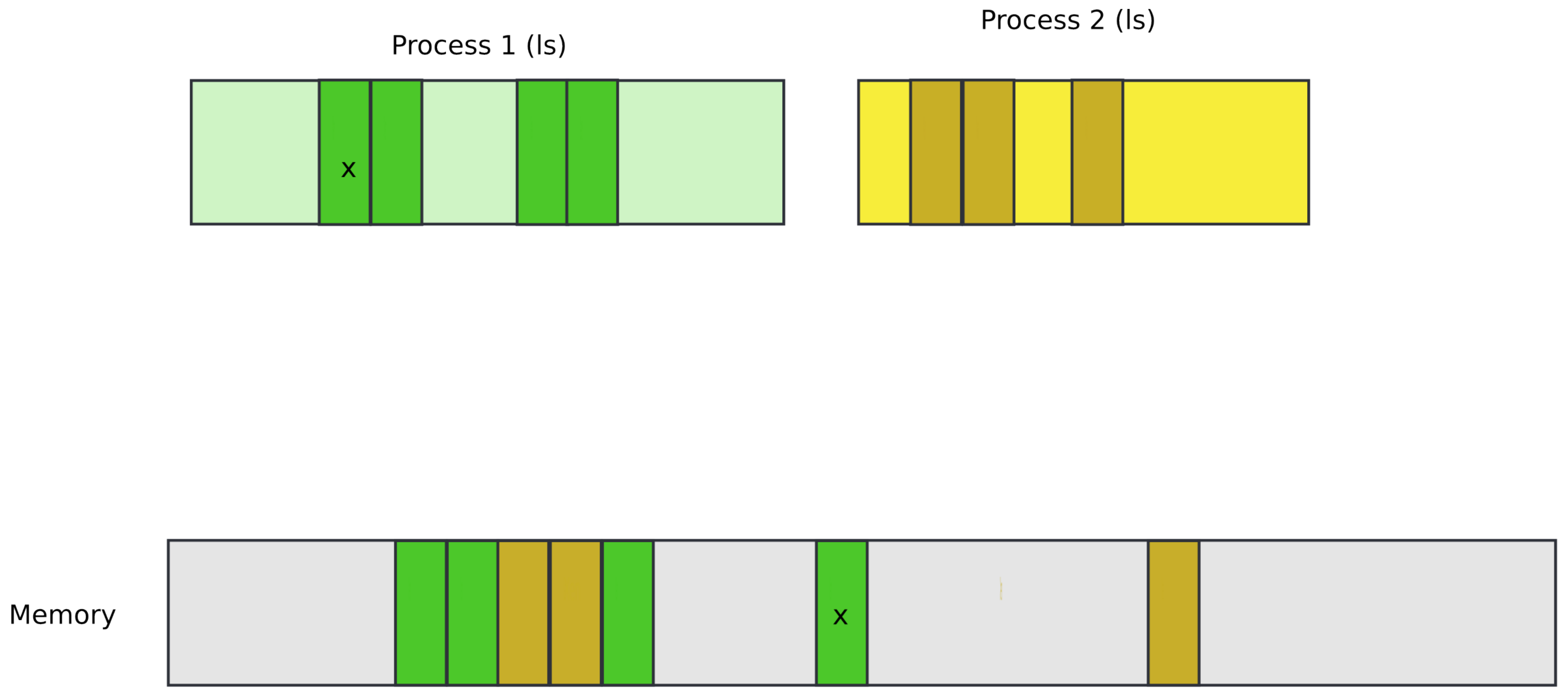
Lecture 6: Address translation (Paging)

Anton Burtsev

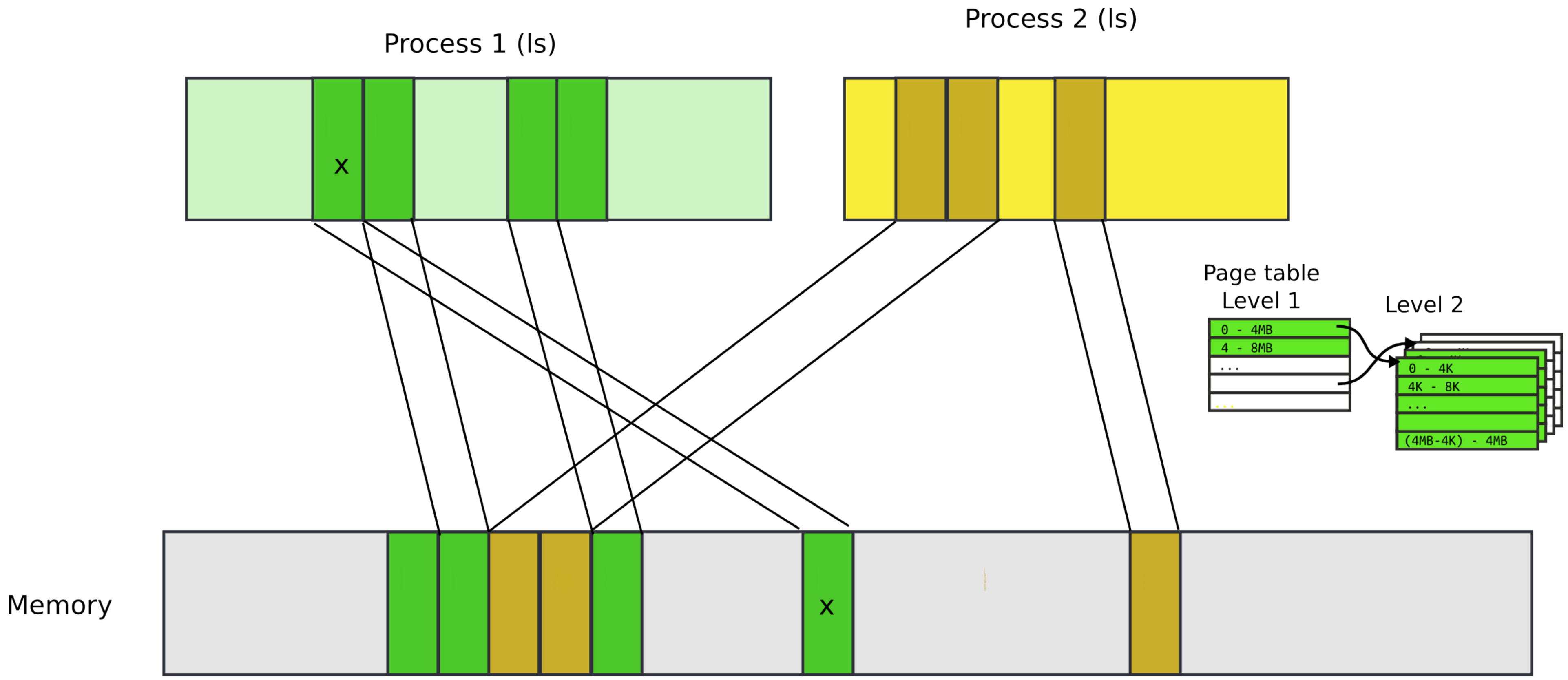
October, 2017

Paging

Pages



Pages

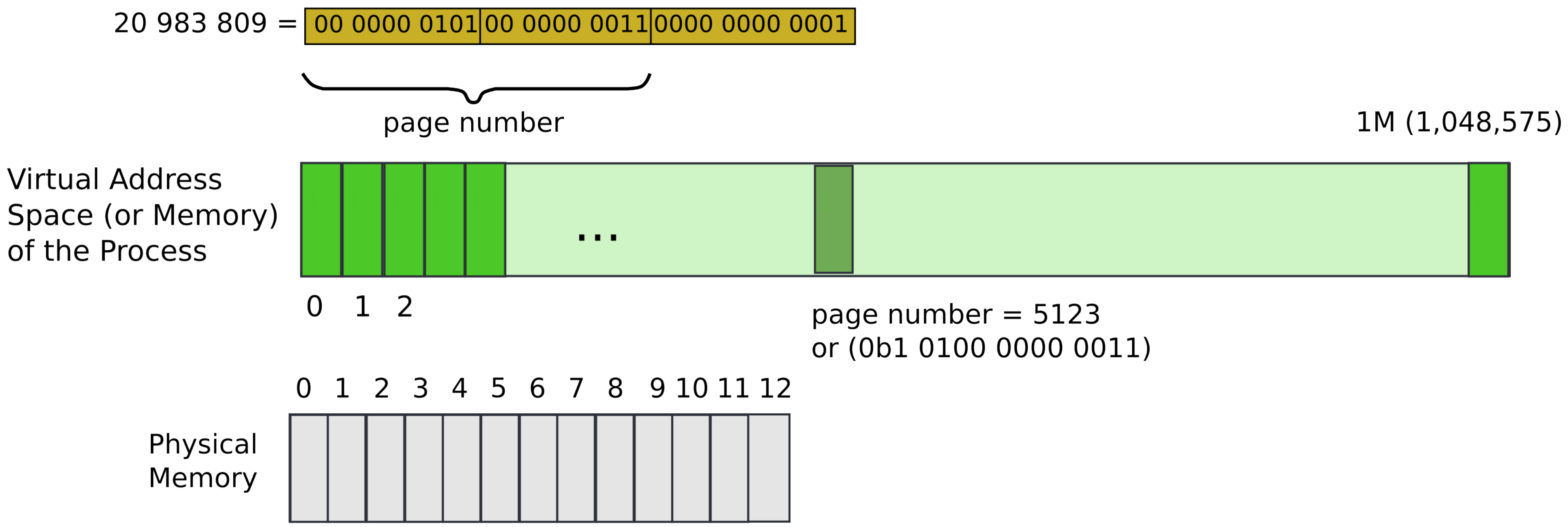


Process 1 (ls)
Process 2 (ls)
x
Page table
Level 1
Level 2
0 - 4MB
4 - 8MB
...
0 - 4K
4K - 8K
...
(4MB-4K) - 4MB
Memory
x

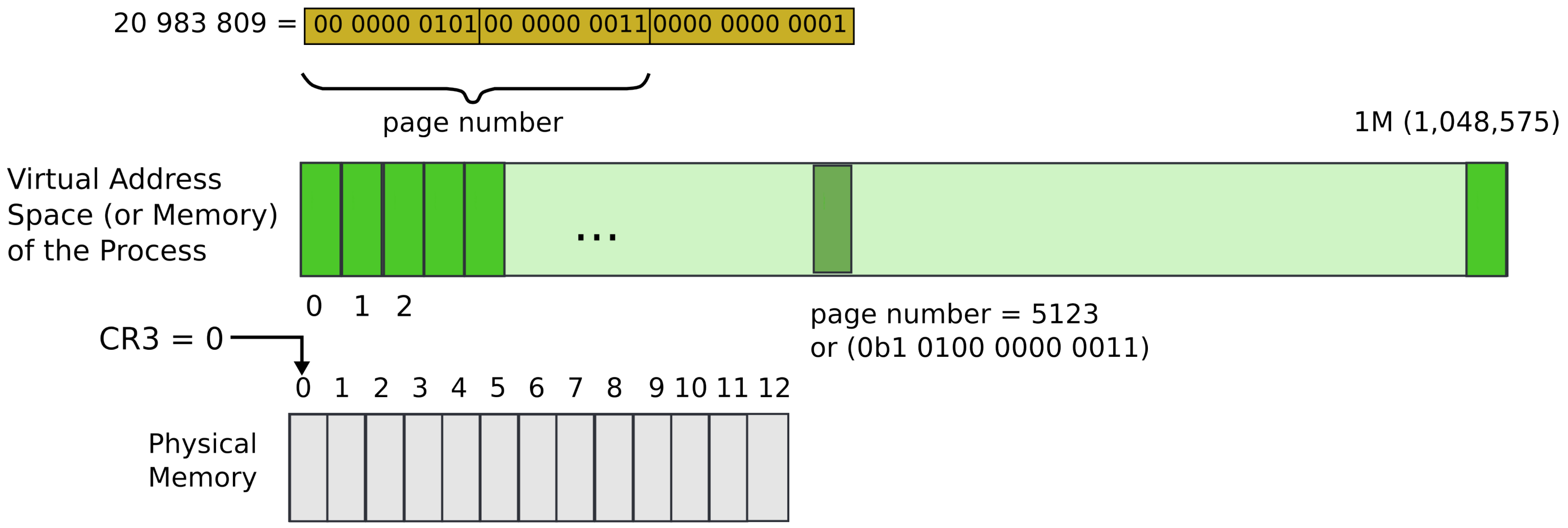
Paging idea

- 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

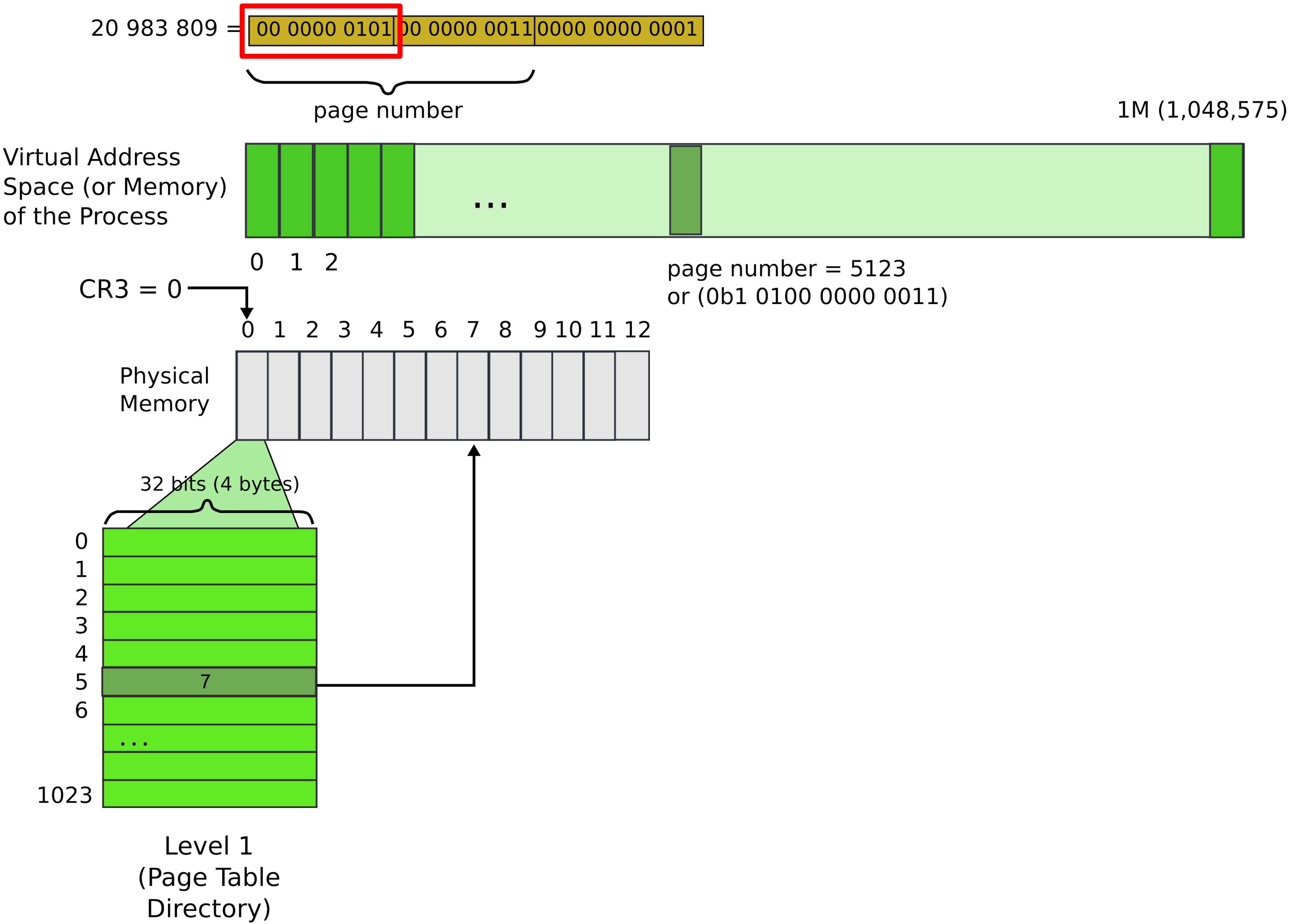
```
mov (%EBX), EAX   # mov value from the  location pointed by EBX into EAX
EAX = 0
EBX = 20 983 809
```



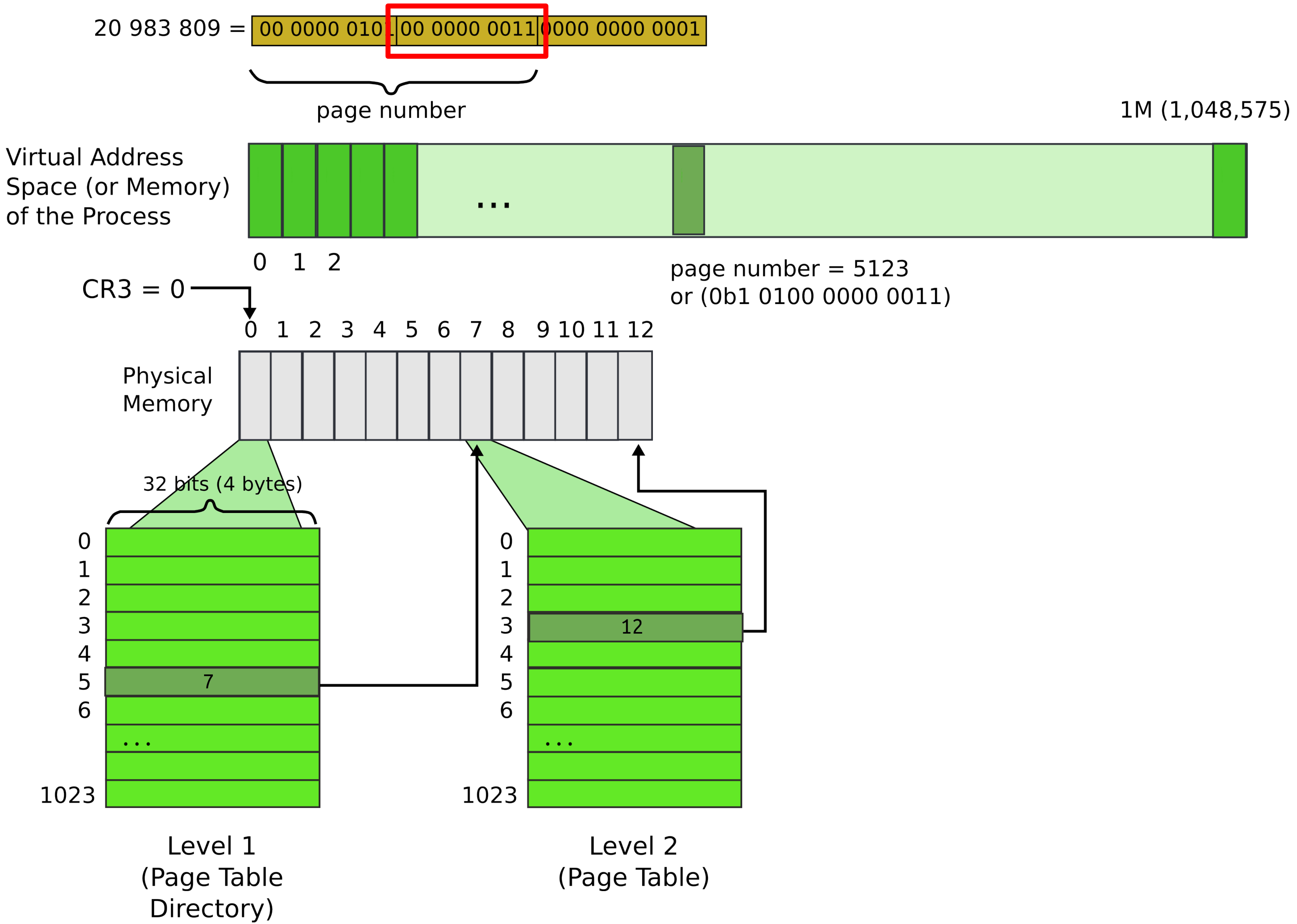
```
mov (%EBX), EAX   # mov value from the  location pointed by EBX into EAX
EAX = 0
EBX = 20 983 809
```



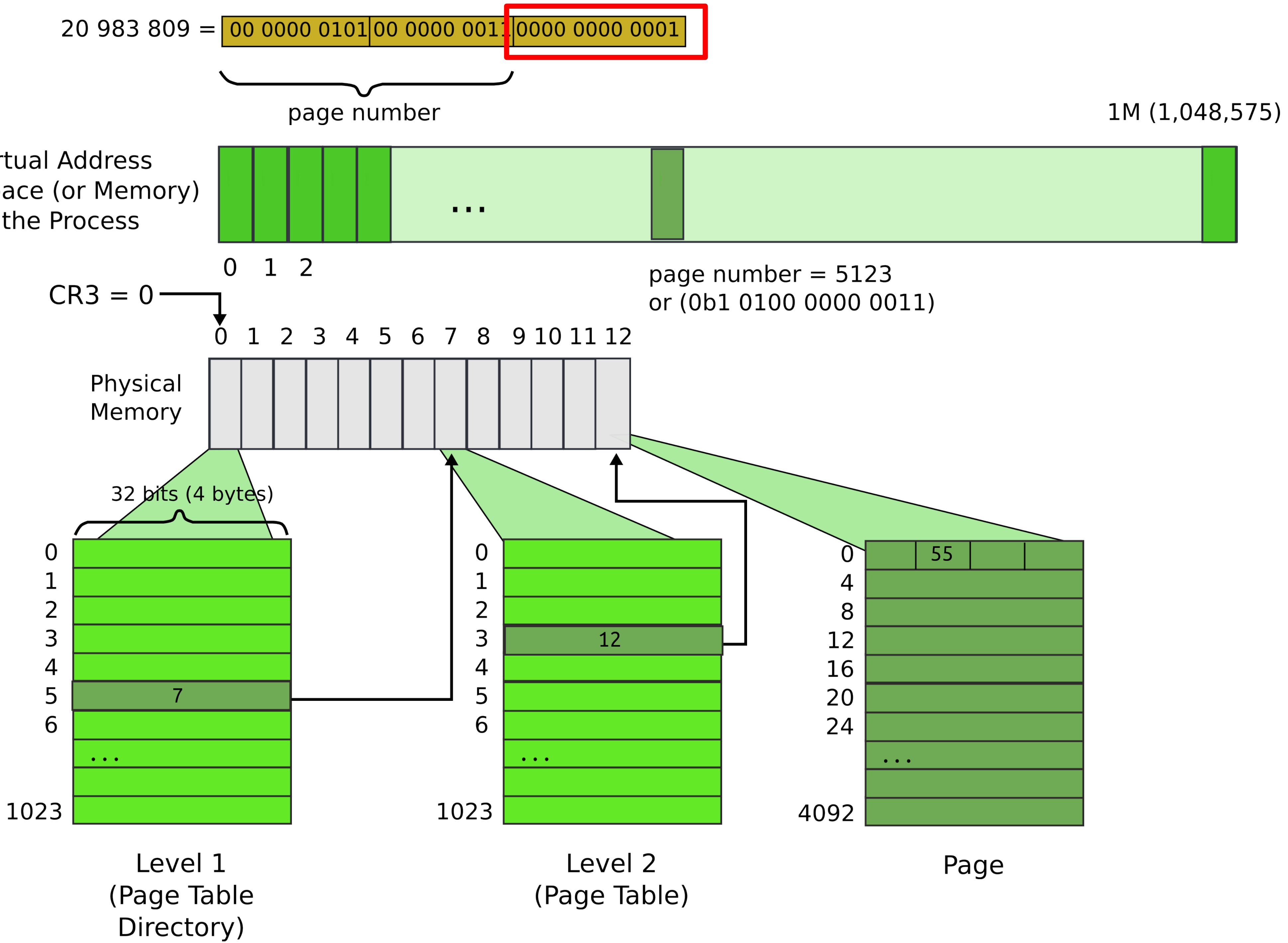
```
mov (%EBX), EAX   # mov value from the  location pointed by EBX into EAX
EAX = 0
EBX = 20 983 809
```



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

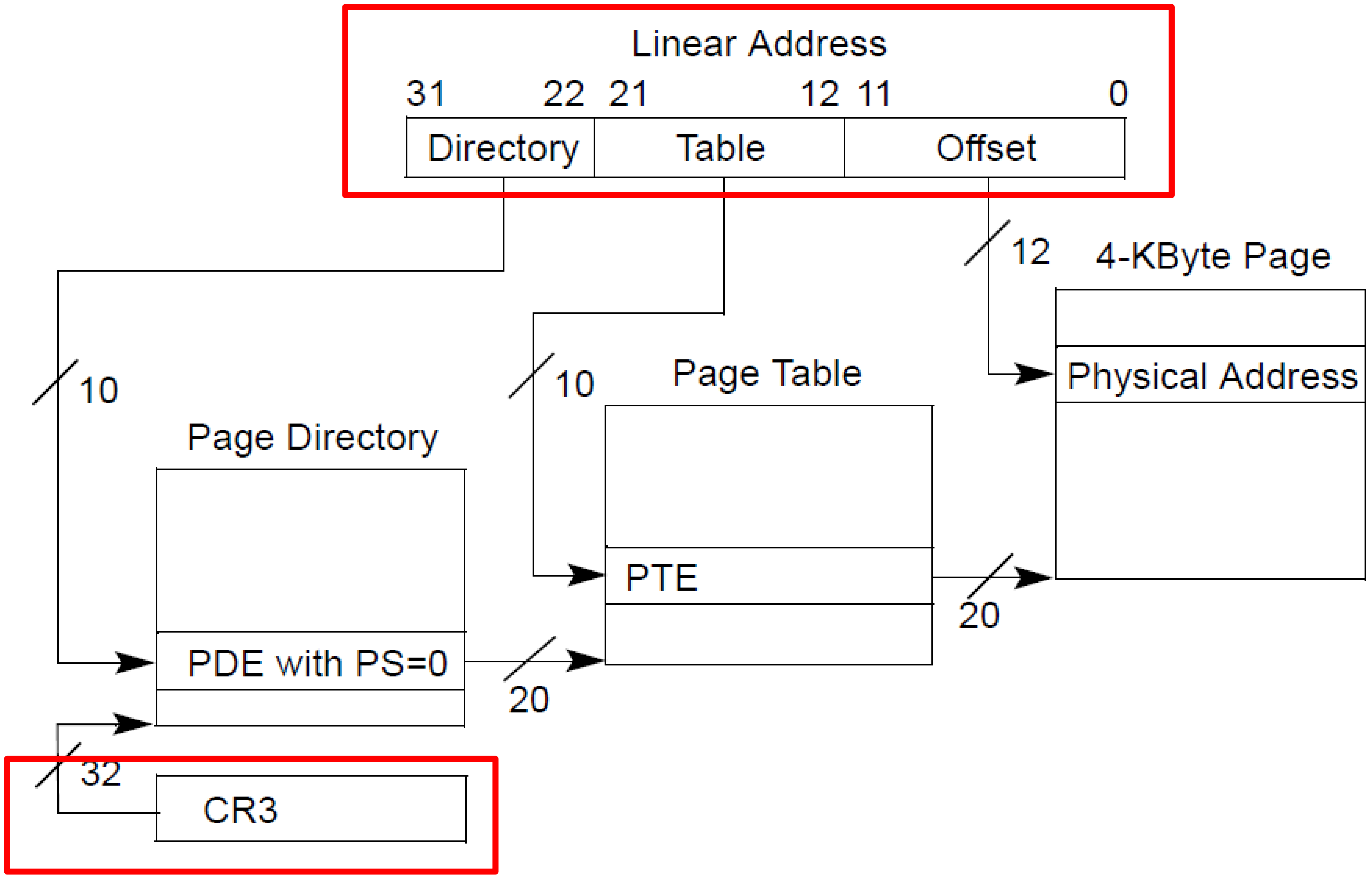


```
mov (%EBX), EAX   # mov value from the  location pointed by EBX into EAX
EAX = 0
EBX = 20 983 809
```

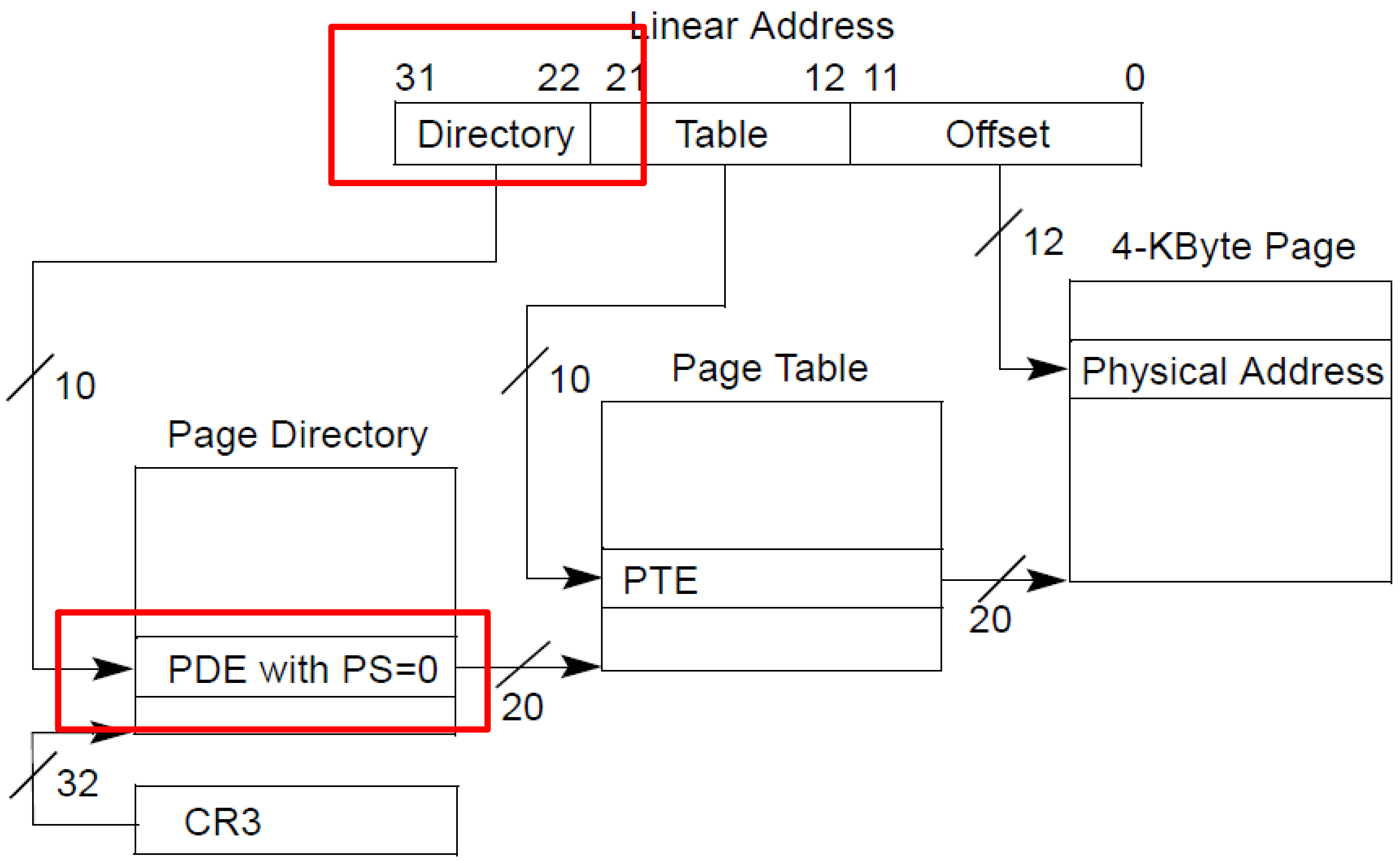


- Result:
  - EAX = 55

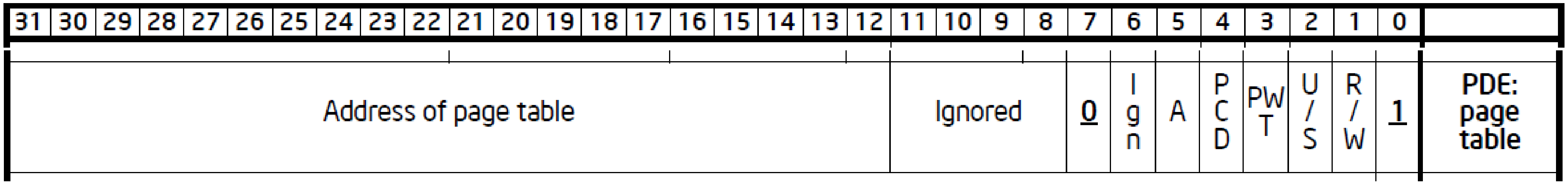
Page translation



Page translation



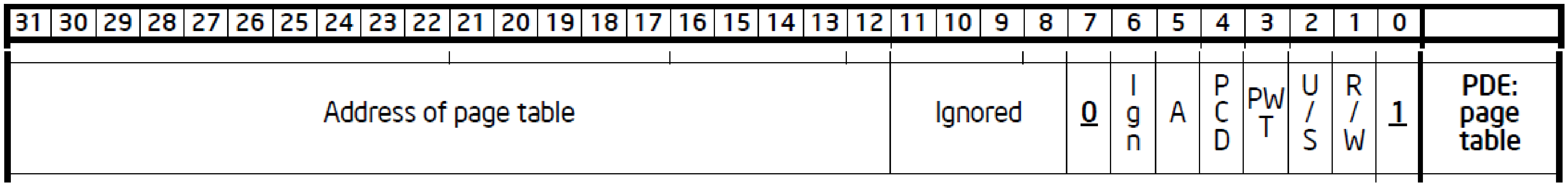
Page directory entry (PDE)



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page table																				Ignored				**0**	Ign	A	PCD	PWT	U/S	R/W	**1**	**PDE: page table**

- 20 bit address of the page table

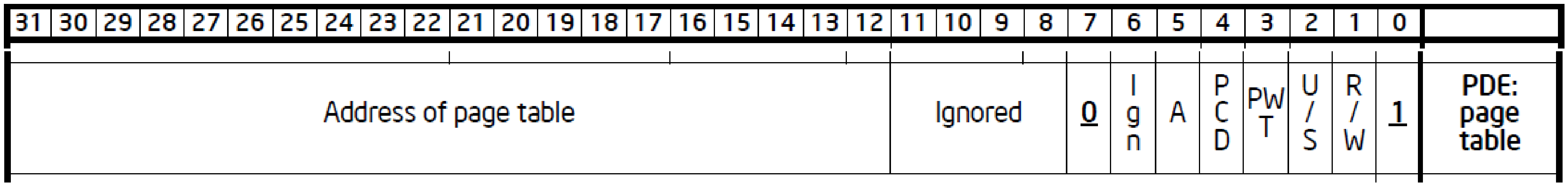
Page directory entry (PDE)



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page table																				Ignored				<u>0</u>	Ign	A	PCD	PWT	U/S	R/W	<u>1</u>	**PDE: page table**

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

Page directory entry (PDE)

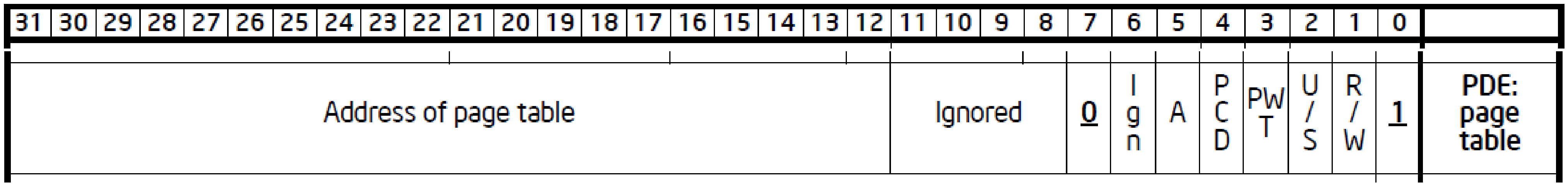


31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page table																				Ignored				0	Ign	A	PCD	PWT	U/S	R/W	1	**PDE: page table**

- 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

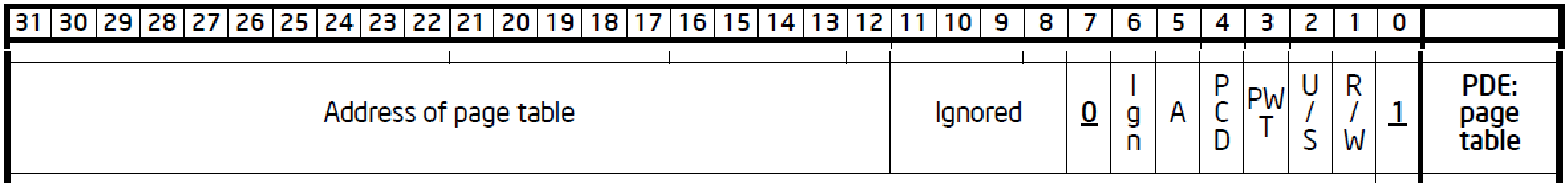
Page directory entry (PDE)



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page table																				Ignored				0	Ign	A	PCD	PWT	U/S	R/W	1	**PDE: page table**

- 20 bit address of the page table
  - Pages 4KB each, we need 1M to cover 4GB
- Bit #1: R/W – writes allowed?
  - But allowed where?

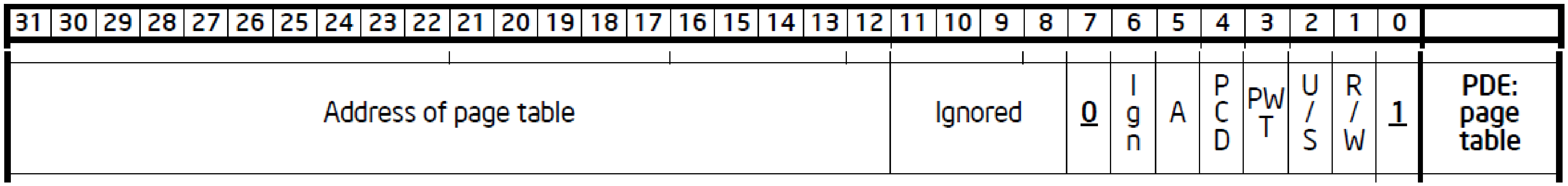
Page directory entry (PDE)



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page table																				Ignored				<u>0</u>	Ign	A	PCD	PWT	U/S	R/W	<u>1</u>	**PDE: page table**

- 20 bit address of the page table
  - Pages 4KB each, we need 1M to cover 4GB
- Bit #1: R/W – writes allowed?
  - But allowed where?
  - One page directory entry controls 1024 Level 2 page tables
    - Each Level 2 maps 4KB page
  - So it's a region of 4KB x 1024 = 4MB

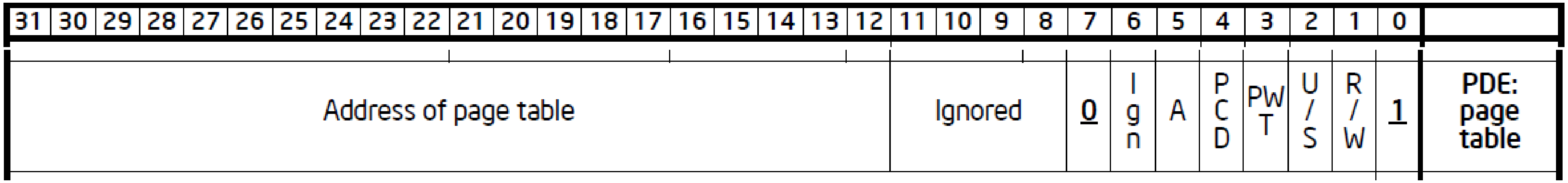
Page directory entry (PDE)



31–12	11–8	7	6	5	4	3	2	1	0	
Address of page table	Ignored	**0**	Ign	A	PCD	PWT	U/S	R/W	**1**	**PDE: page table**

- 20 bit address of the page table
  - Pages 4KB each, we need 1M to cover 4GB
- Bit #1: R/W – writes allowed?
  - To a 4MB region controlled by this entry
- Bit #2: U/S – user/supervisor
  - If 0 – user-mode access is not allowed
- A – accessed

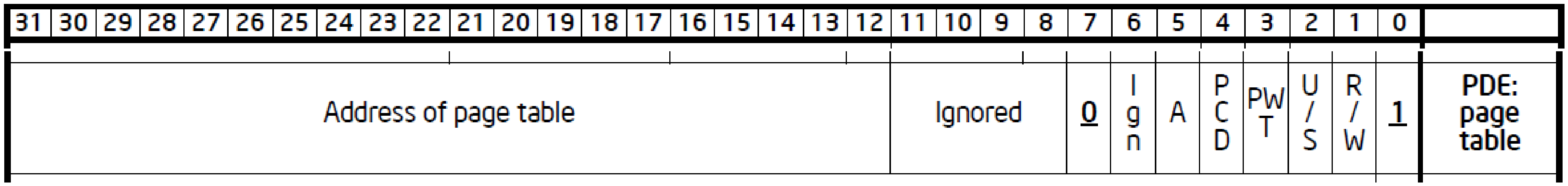
Page directory entry (PDE)



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page table																				Ignored				**0**	Ign	A	PCD	PWT	U/S	R/W	**1**	**PDE: page table**

- 20 bit address of the page table
  - Pages 4KB each, we need 1M to cover 4GB
- Bit #1: R/W – writes allowed?
  - To a 4MB region controlled by this entry
- Bit #2: U/S – user/supervisor
  - If 0 – user-mode access is not allowed
  - Allows protecting kernel memory from user-level applications

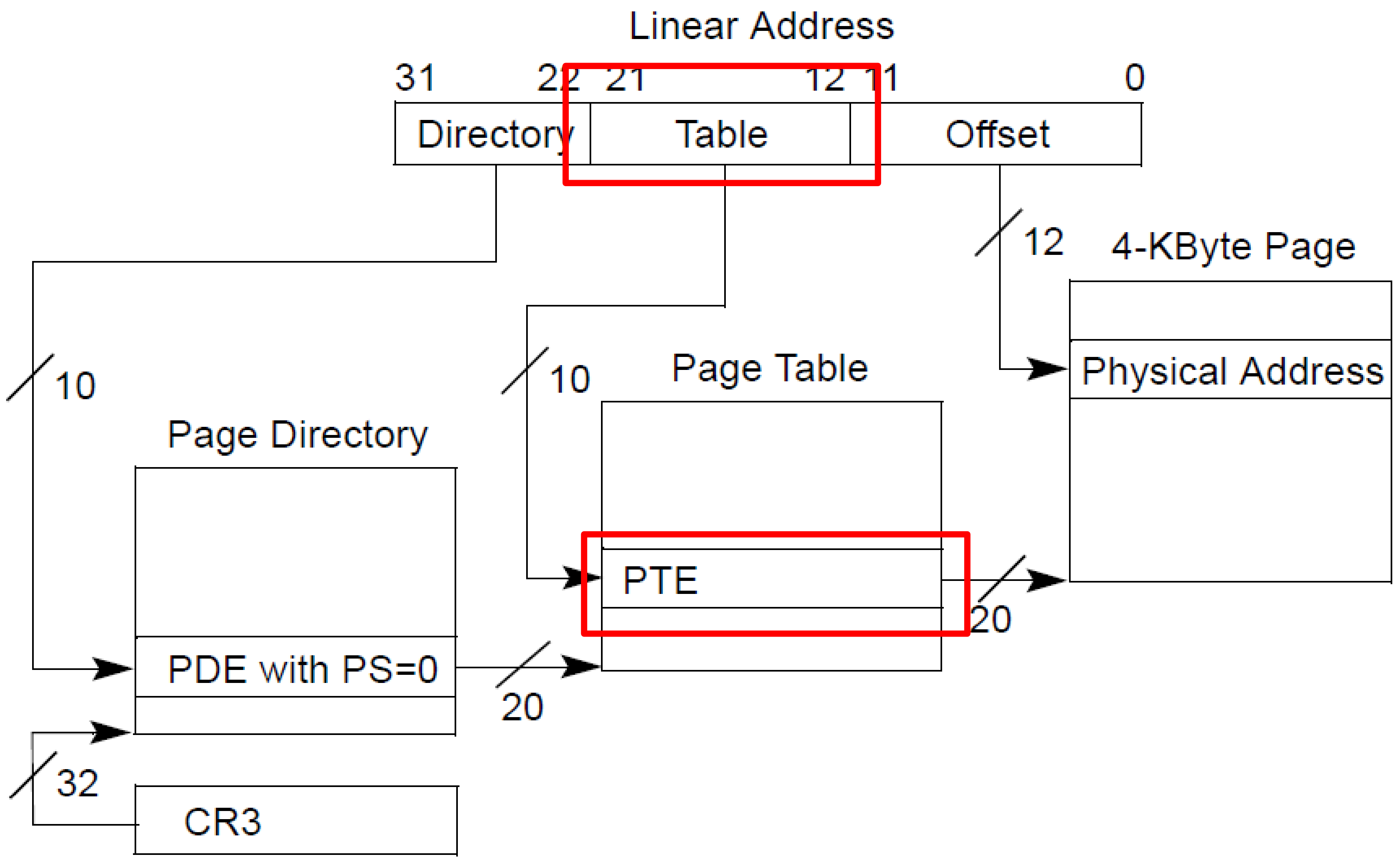
Page directory entry (PDE)



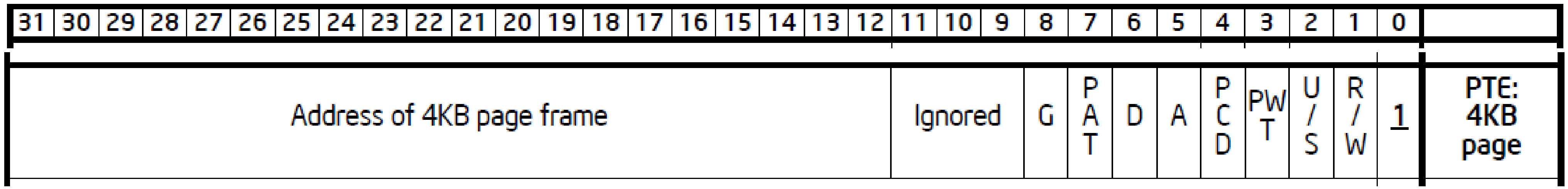
31 … 12	11 … 8	7	6	5	4	3	2	1	0	
Address of page table	Ignored	<u>0</u>	Ign	A	PCD	PWT	U/S	R/W	<u>1</u>	**PDE: page table**

- 20 bit address of the page table
  - Pages 4KB each, we need 1M to cover 4GB
- Bit #1: R/W – writes allowed?
  - To a 4MB region controlled by this entry
- Bit #2: U/S – user/supervisor
  - If 0 – user-mode access is not allowed
  - Allows protecting kernel memory from user-level applications
- Bit #5: A – accessed

Page translation



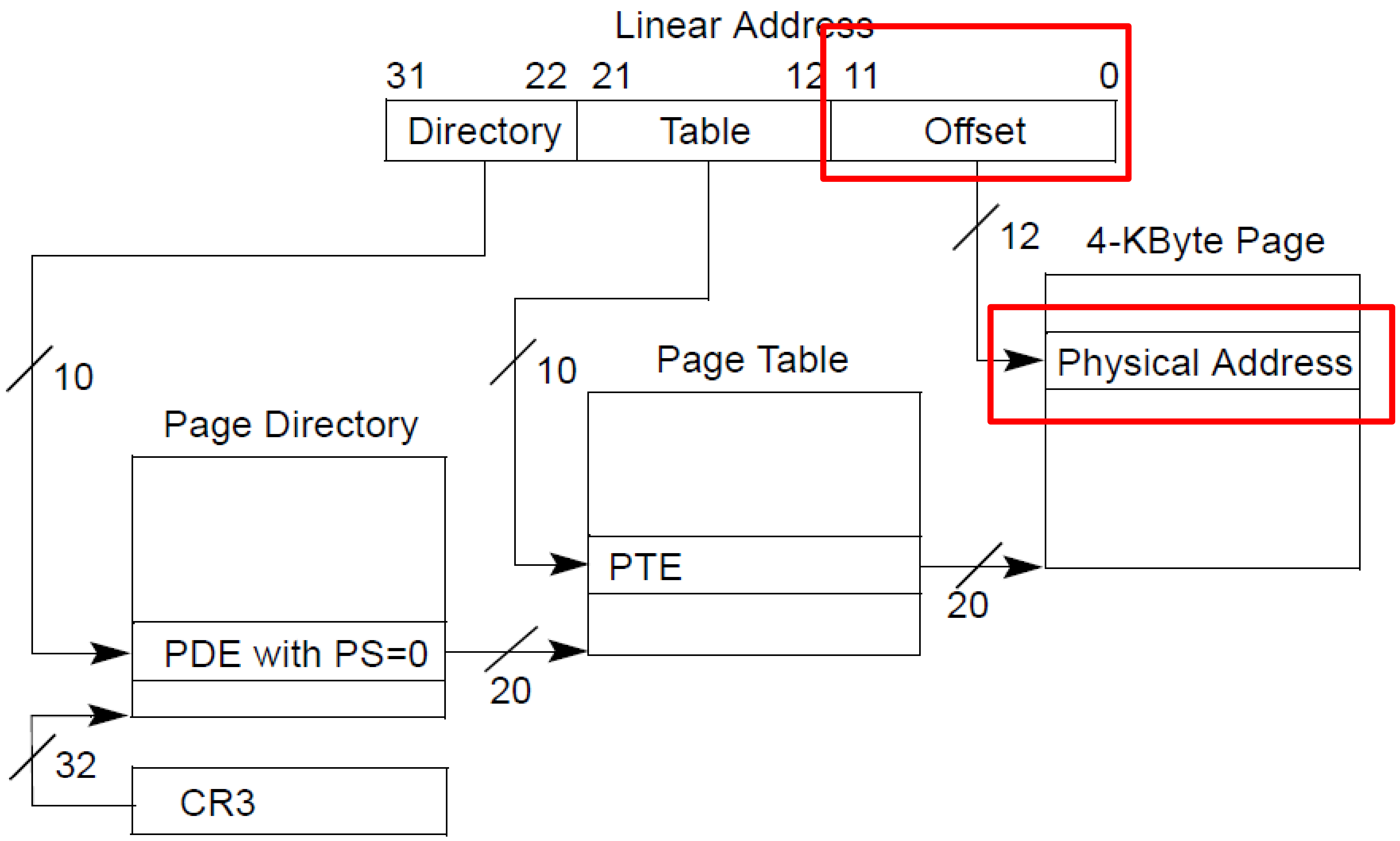
Page table entry (PTE)



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of 4KB page frame																				Ignored			G	PAT	D	A	PCD	PWT	U/S	R/W	1	**PTE: 4KB page**

- 20 bit address of the 4KB page
  - Pages 4KB each, we need 1M to cover 4GB
- Bit #1: R/W – writes allowed?
  - To a 4KB page
- Bit #2: U/S – user/supervisor
  - If 0 user-mode access is not allowed
- Bit #5: A – accessed
- Bit #6: D – dirty – software has written to this page

Page 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?
  - 1k entries * 1page/entry * 4K/page = 4MB
- How large can we get with a second level of translation?
  - 1k tables/dir * 1k entries/table * 4k/page = 4 GB
  - Nice that it works out that way!

Why do we need paging?

- Compared to segments pages provide fine-grained control over memory layout
  - No need to relocate/swap the entire segment
    - One page is enough
    -
- You're trading flexibility (granularity) for overhead of data structures required for translation

Example 1: Ultimate flexibility

- Each byte can be relocated anywhere in physical memory
- What's the overhead of page tables?
  - Imagine we use array instead of page tables (for simplicity)

Example 1: Ultimate flexibility

- Each byte can be relocated anywhere in physical memory
- What's the overhead of page tables?
  - Imagine we use array instead of page tables (for simplicity)
  - We need 4 bytes to relocate each other byte
    - 4 bytes describe 32bit address
  - Therefore, we need array of 4 bytes x 4B entries
    - 16GBs

Example 2: Reasonable flexibility

- Each 4K bytes (a page) can be relocated anywhere in physical memory
- What's the overhead of page tables?
  - Again, imagine we use array instead of page tables (for simplicity)

Example 2: Reasonable flexibility

- Each 4K bytes (a page) can be relocated anywhere in physical memory
- What's the overhead of page tables?
  - Again, imagine we use array instead of page tables (for simplicity)
  - We need 4 bytes to relocate each 4KB page
    - 4 bytes describe 32bit address
  - Therefore, we need array of 4 bytes x 1M entries
    - If we split 4GB address space, into 4GB pages, we need 1M pages
  - We need 4MB array

Example 3: Less flexibility

- Each 1M bytes (a 1MB page) can be relocated anywhere in physical memory
- What's the overhead of page tables?
  - Again, imagine we use array instead of page tables (for simplicity)
  - We need 4 bytes to relocate each 1MB page
    - 4 bytes describe 32bit address
  - Therefore, we need array of 4 bytes x 4K entries
    - If we split 4GB address space, into 1MB pages, we need 4K pages
  - We need 16KB array
    - Wow! That's much less than 4MB required for 4KB pages

But why do we need page tables

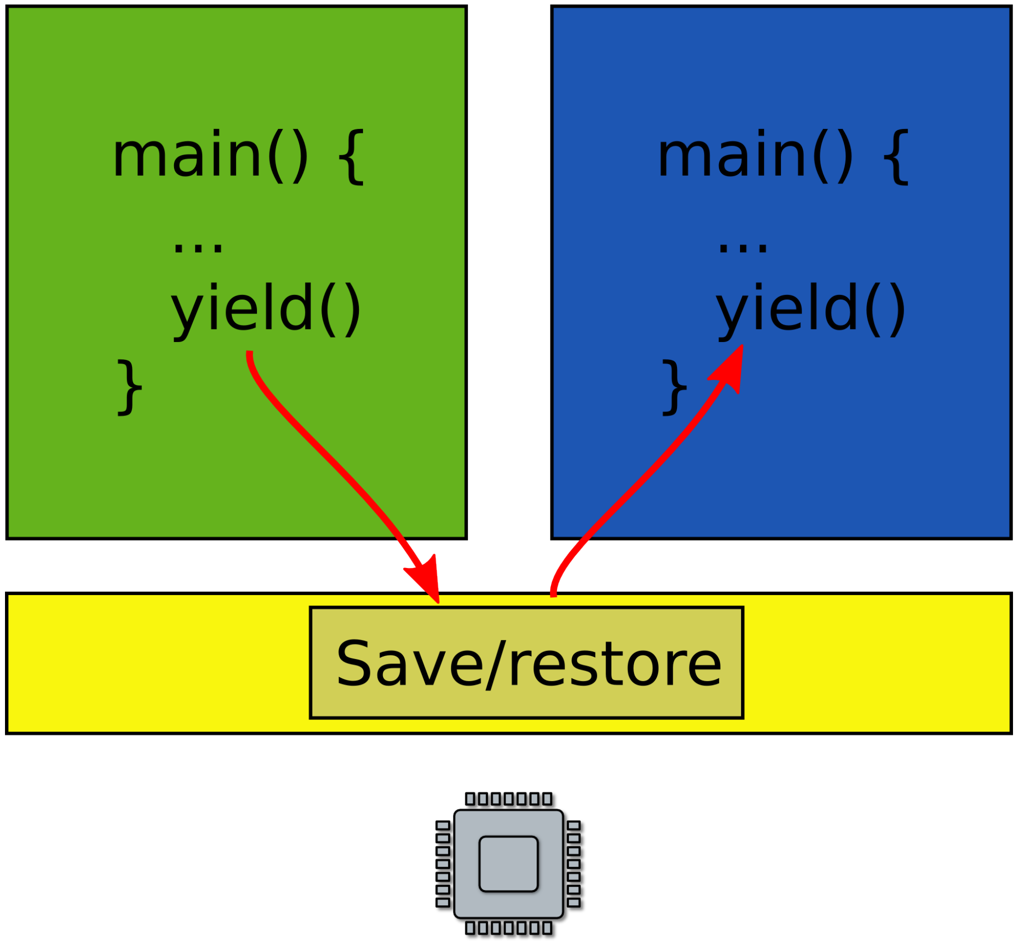
- Instead of arrays?

But why do we need page tables

… Instead of arrays?

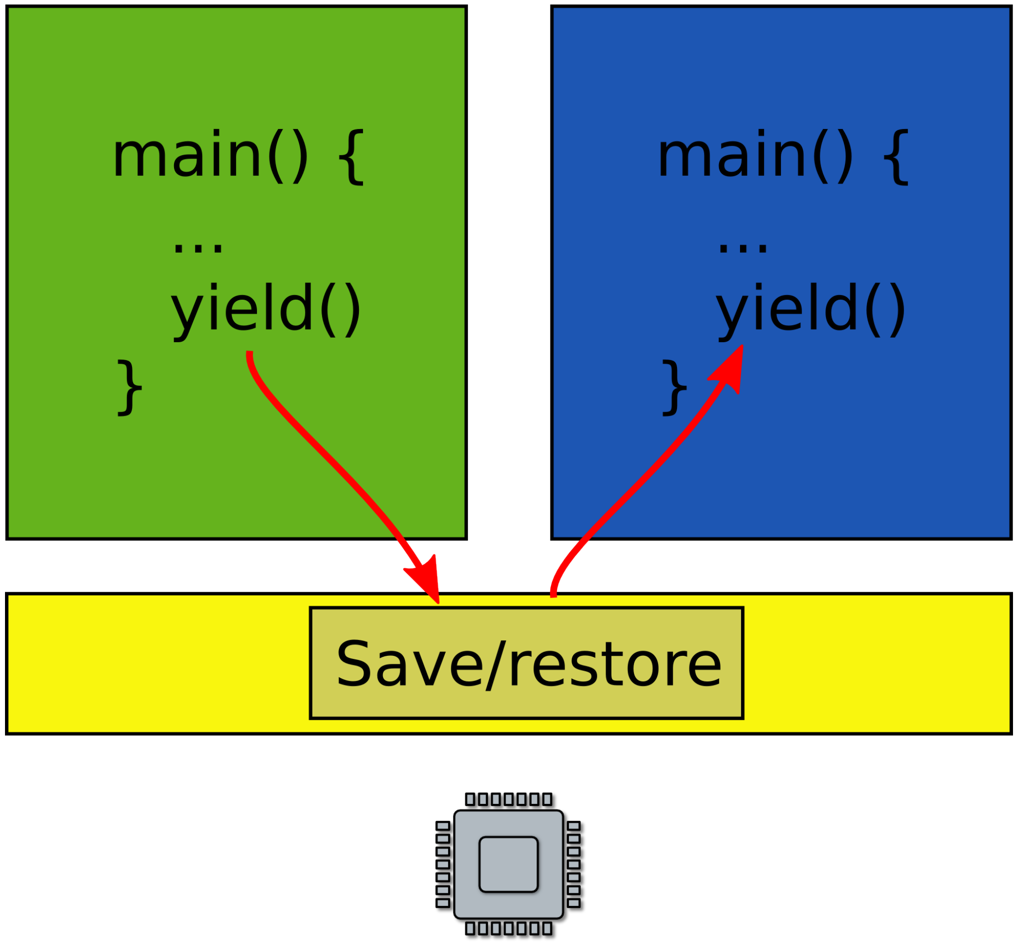
- Page tables represent sparse address space more efficiently
  - An entire array has to be allocated upfront
  - But if the address space uses a handful of pages
  - Only page tables (Level 1 and 2 need to be allocated to describe translation)
- On a dense address space this benefit goes away
  - I'll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

But what about isolation?



- Two programs, one memory?

But what about isolation?



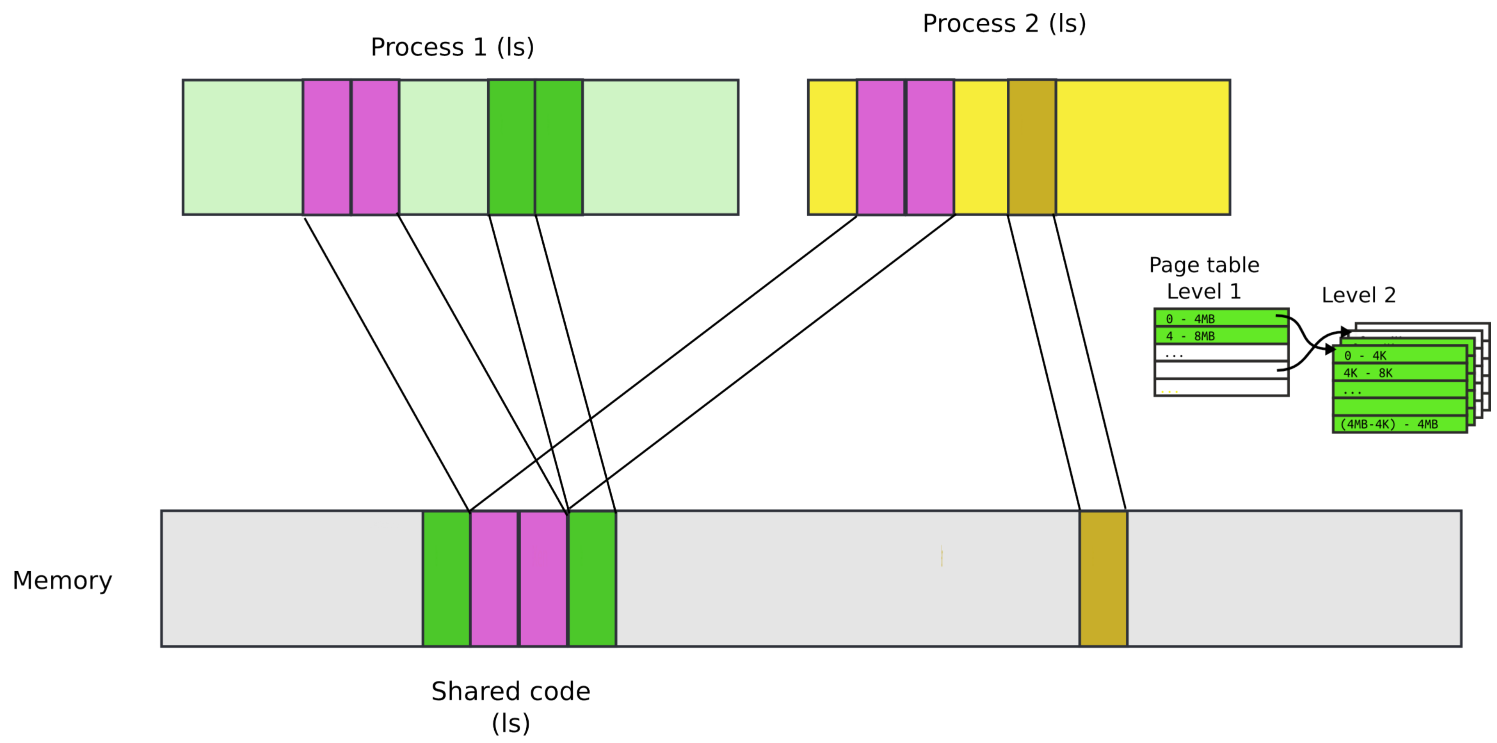
- Two programs, one memory?
- Each process has its own page table
  - OS switches between them

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 ...

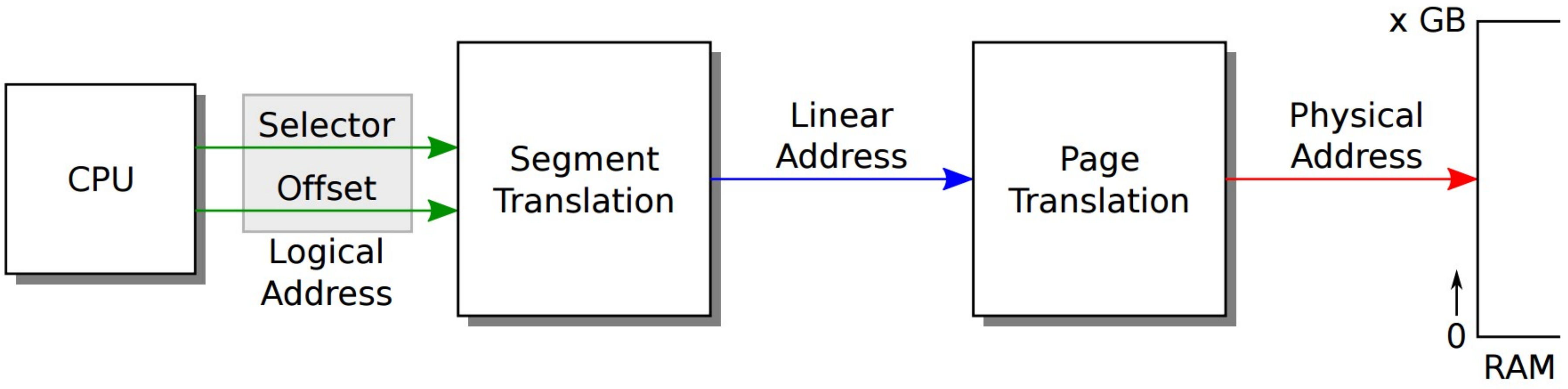
- Share a region of memory across multiple programs
  - Communication (shared buffer of messages)
  - Shared libraries

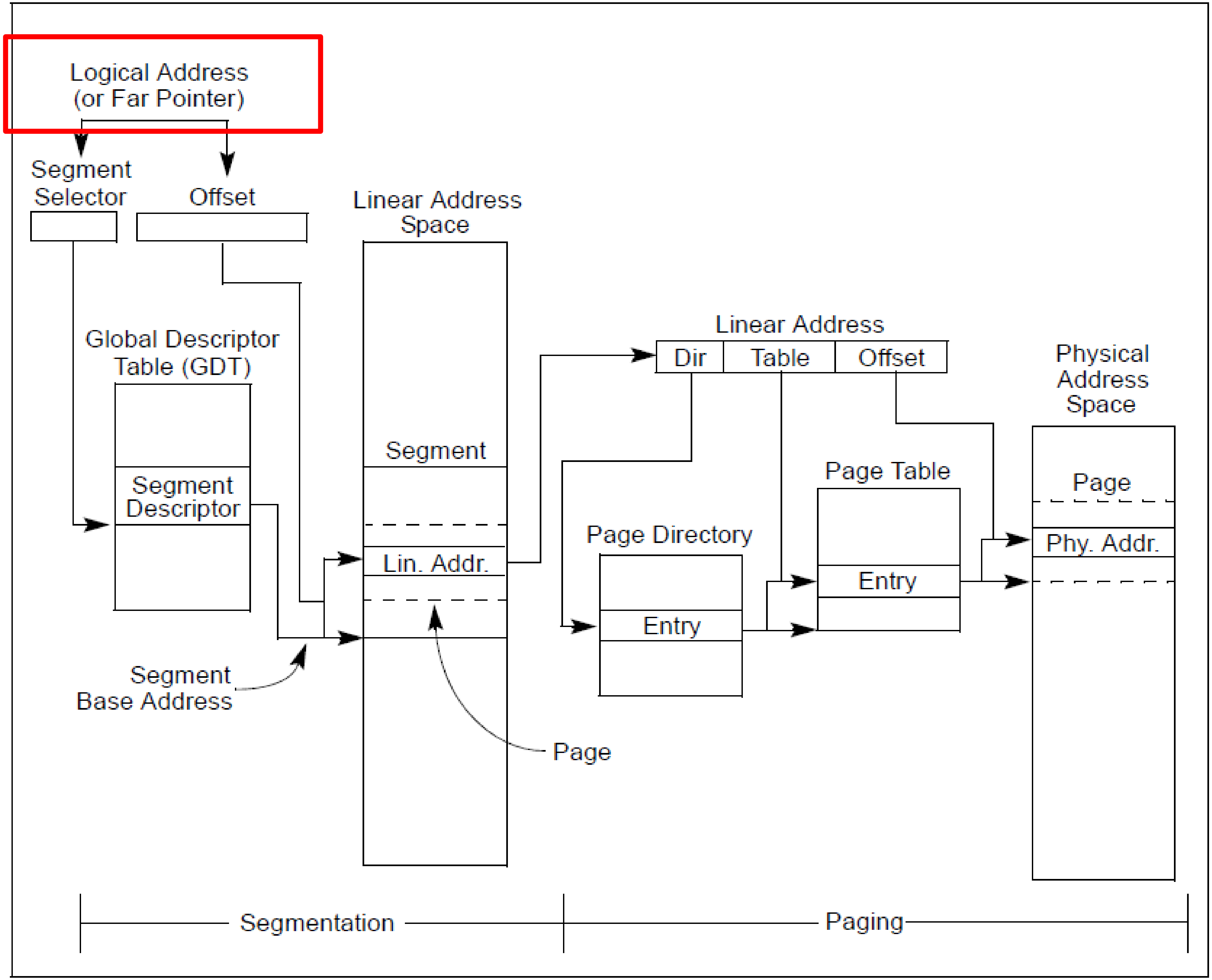


More paging tricks

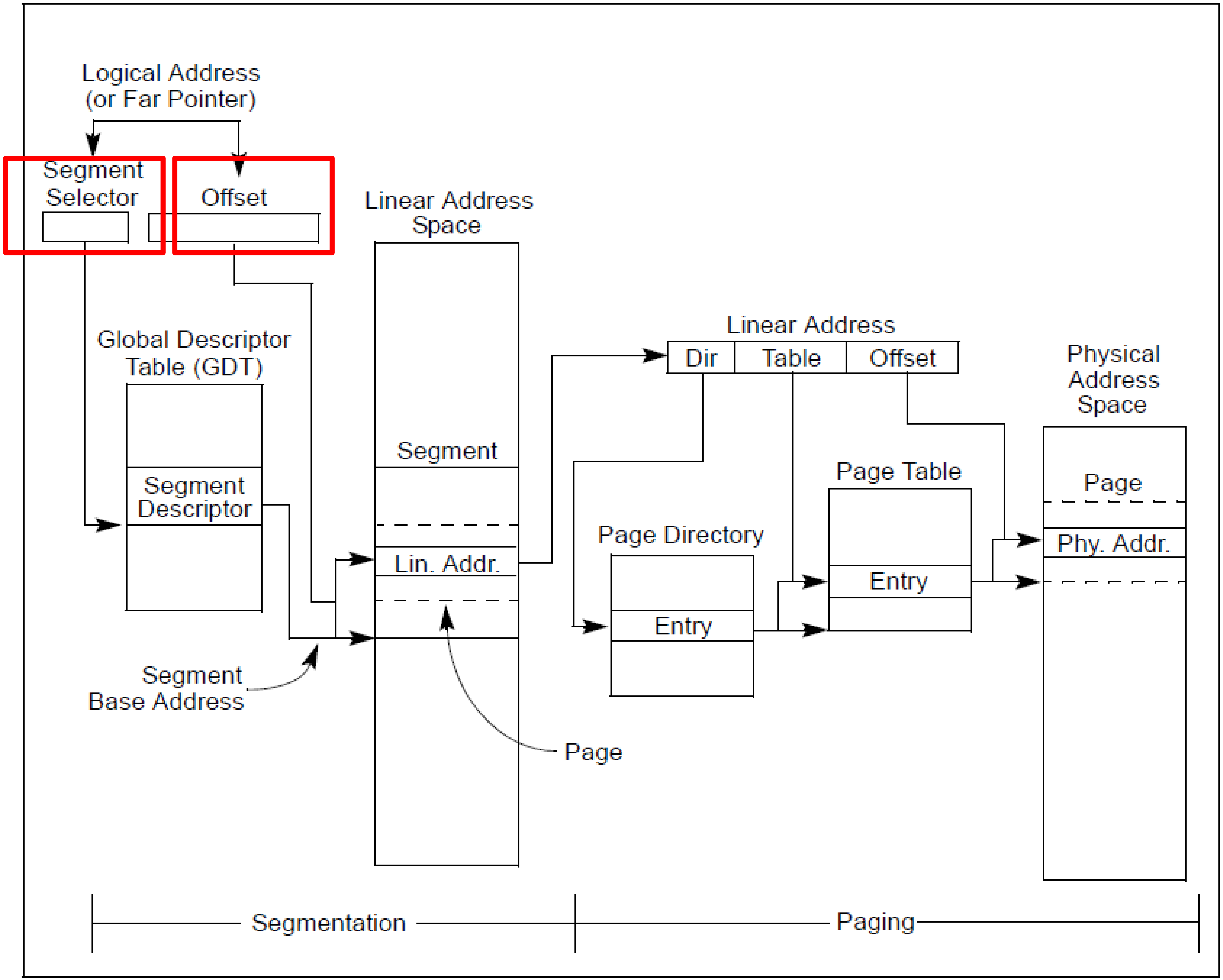
- Protect parts of the program
  - E.g., map code as read-only
    - Disable code modification attacks
    - Remember R/W bit in PTD/PTE entries!
  - E.g., map stack as non-executable
    - Protects from stack smashing attacks
    - Non-executable bit

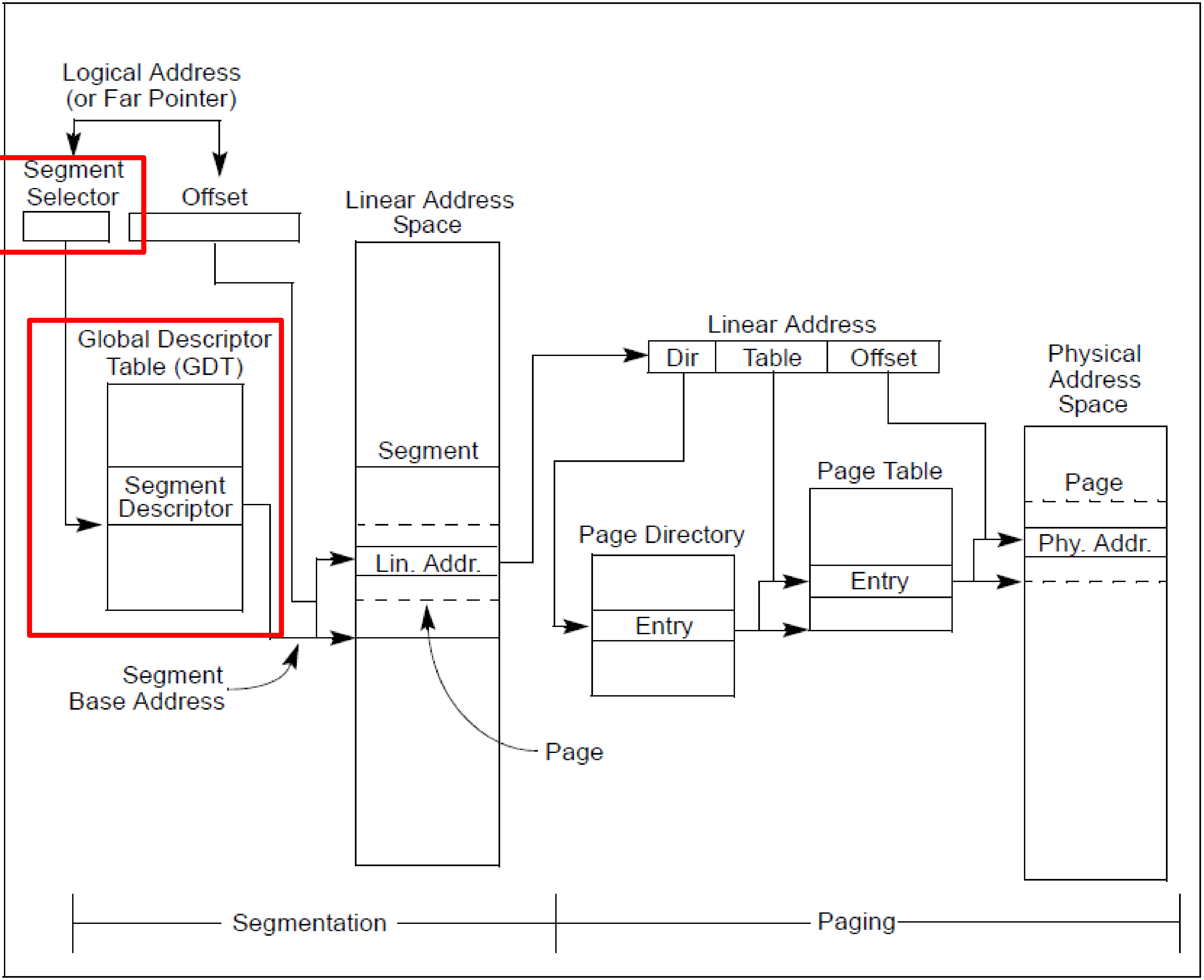
Address translation



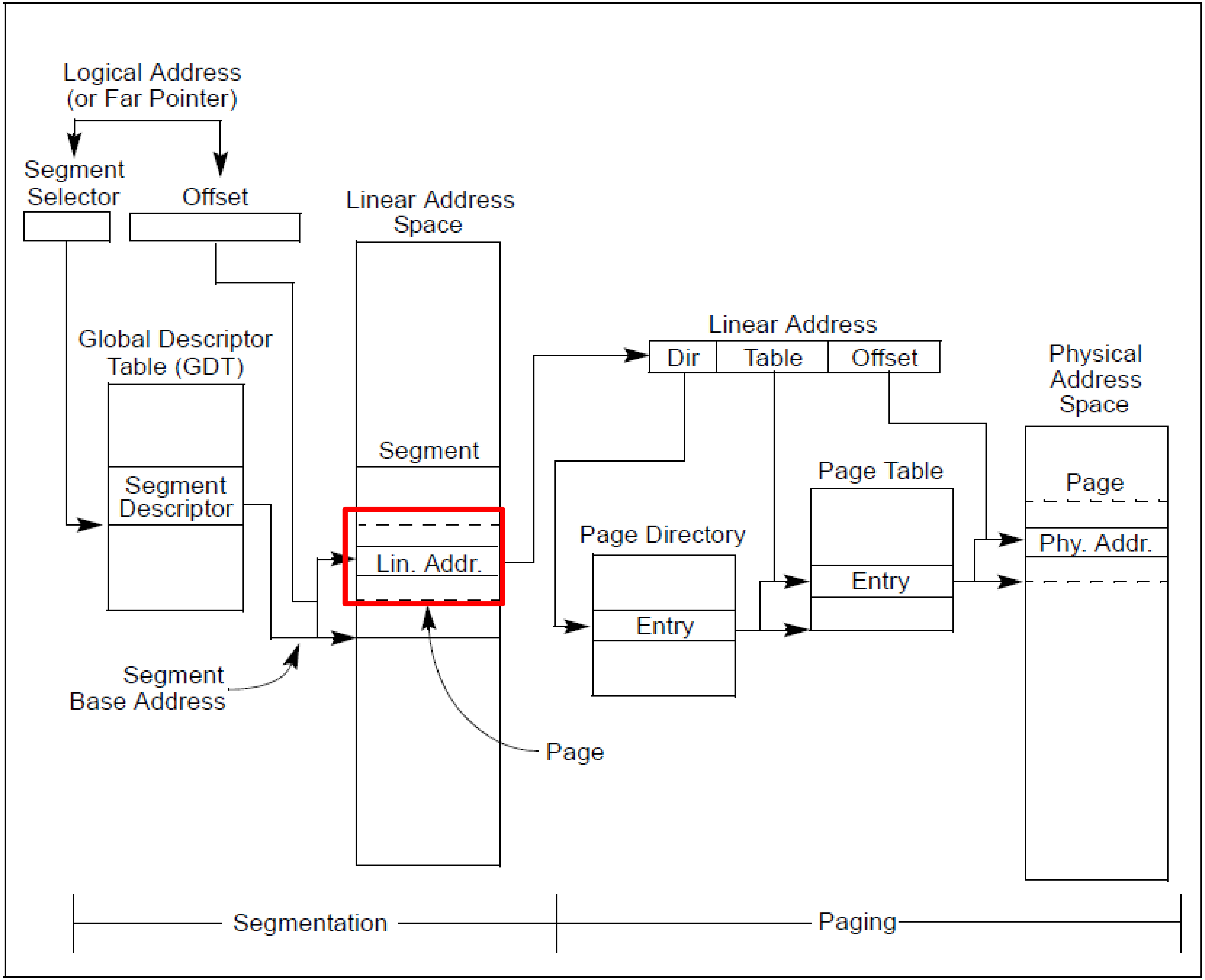


Logical Address
(or Far Pointer)
Segment Selector
Offset
Linear Address Space
Global Descriptor Table (GDT)
Segment Descriptor
Segment
Lin. Addr.
Segment Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical Address Space
Page
Phy. Addr.
Segmentation
Paging

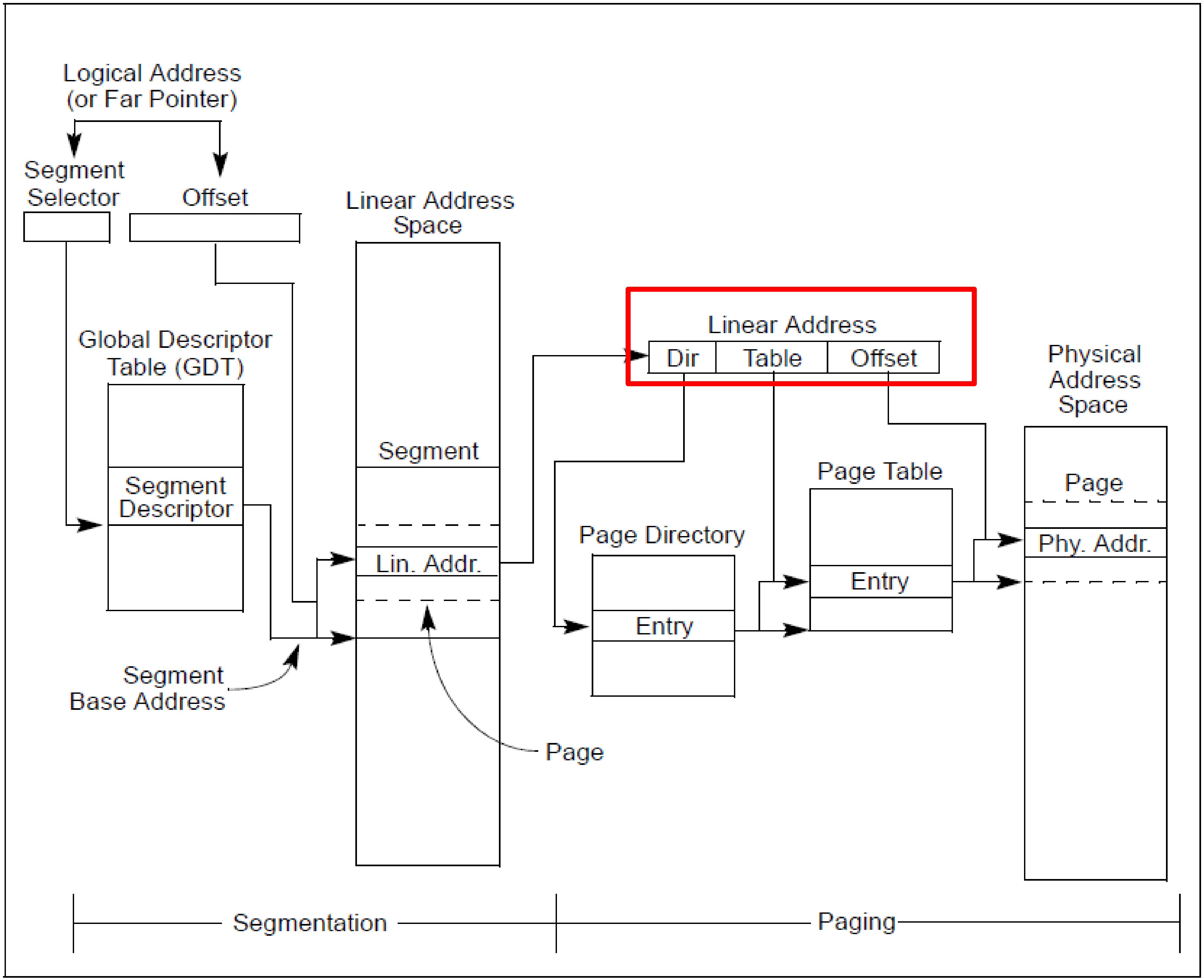

Logical Address (or Far Pointer)
Segment Selector
Offset
Linear Address Space
Global Descriptor Table (GDT)
Segment Descriptor
Segment
Lin. Addr.
Segment Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical Address Space
Page
Phy. Addr.
Segmentation
Paging

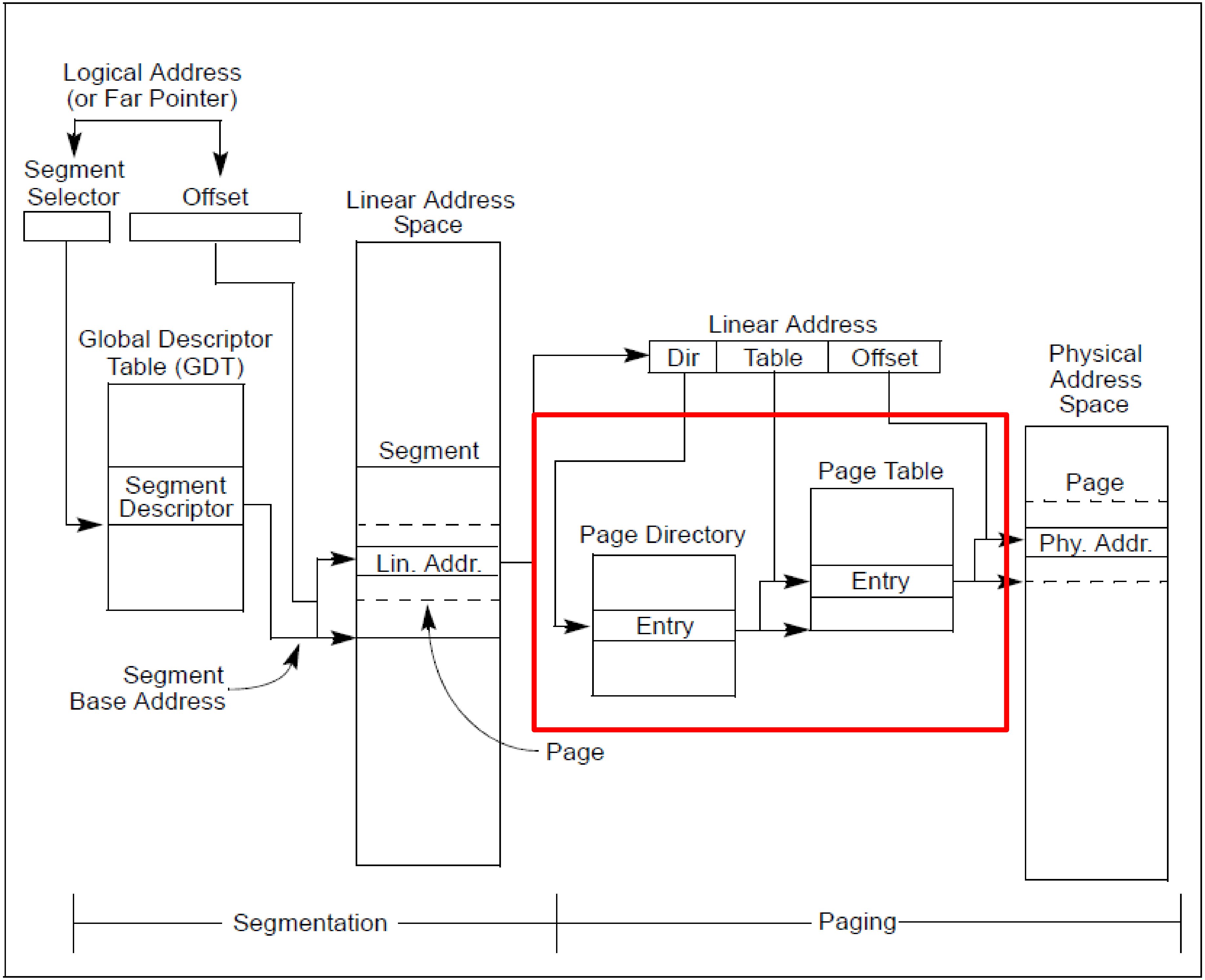


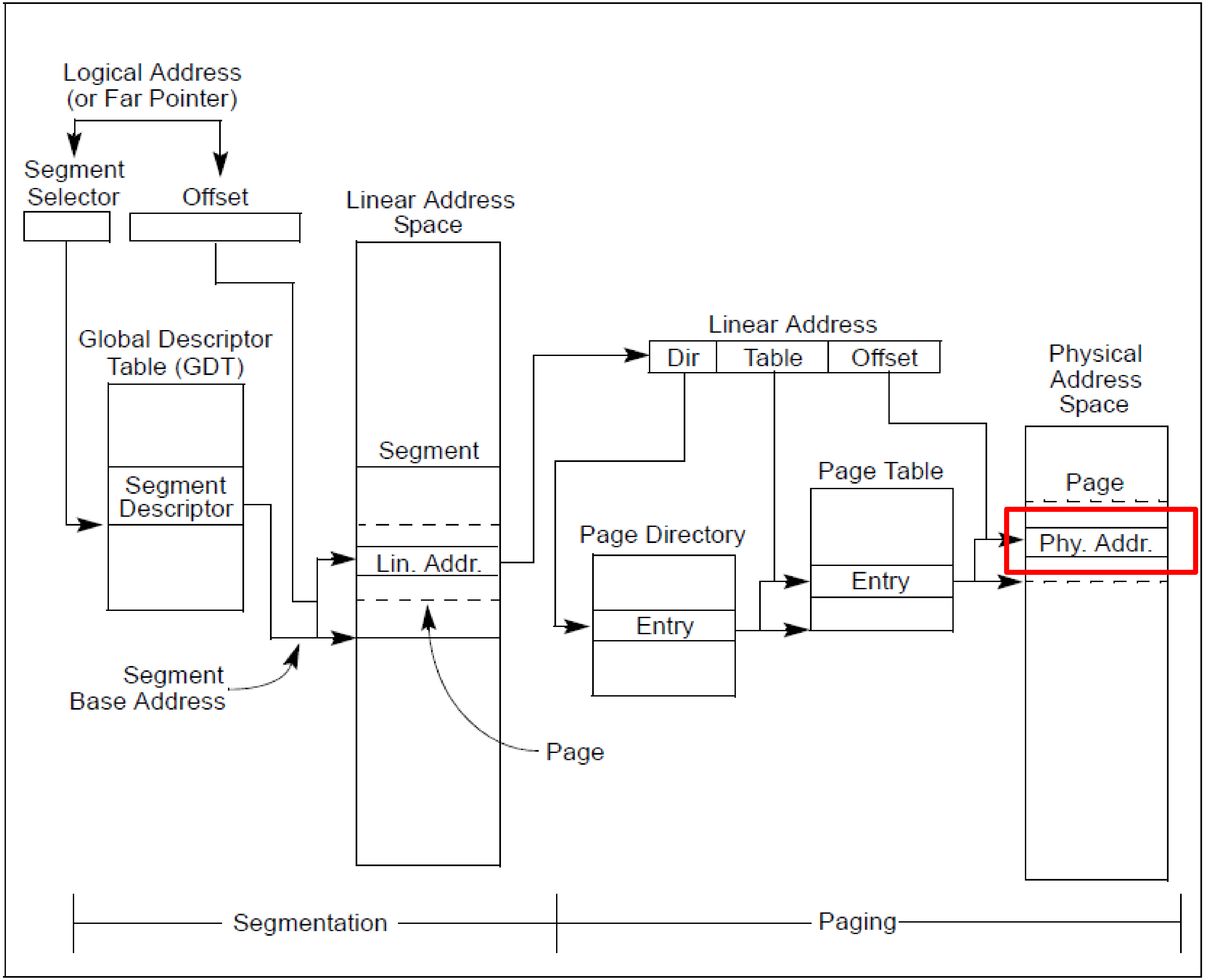
Logical Address (or Far Pointer)
Segment Selector
Offset
Linear Address Space
Global Descriptor Table (GDT)
Segment Descriptor
Segment
Lin. Addr.
Segment Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical Address Space
Page
Phy. Addr.
Segmentation
Paging

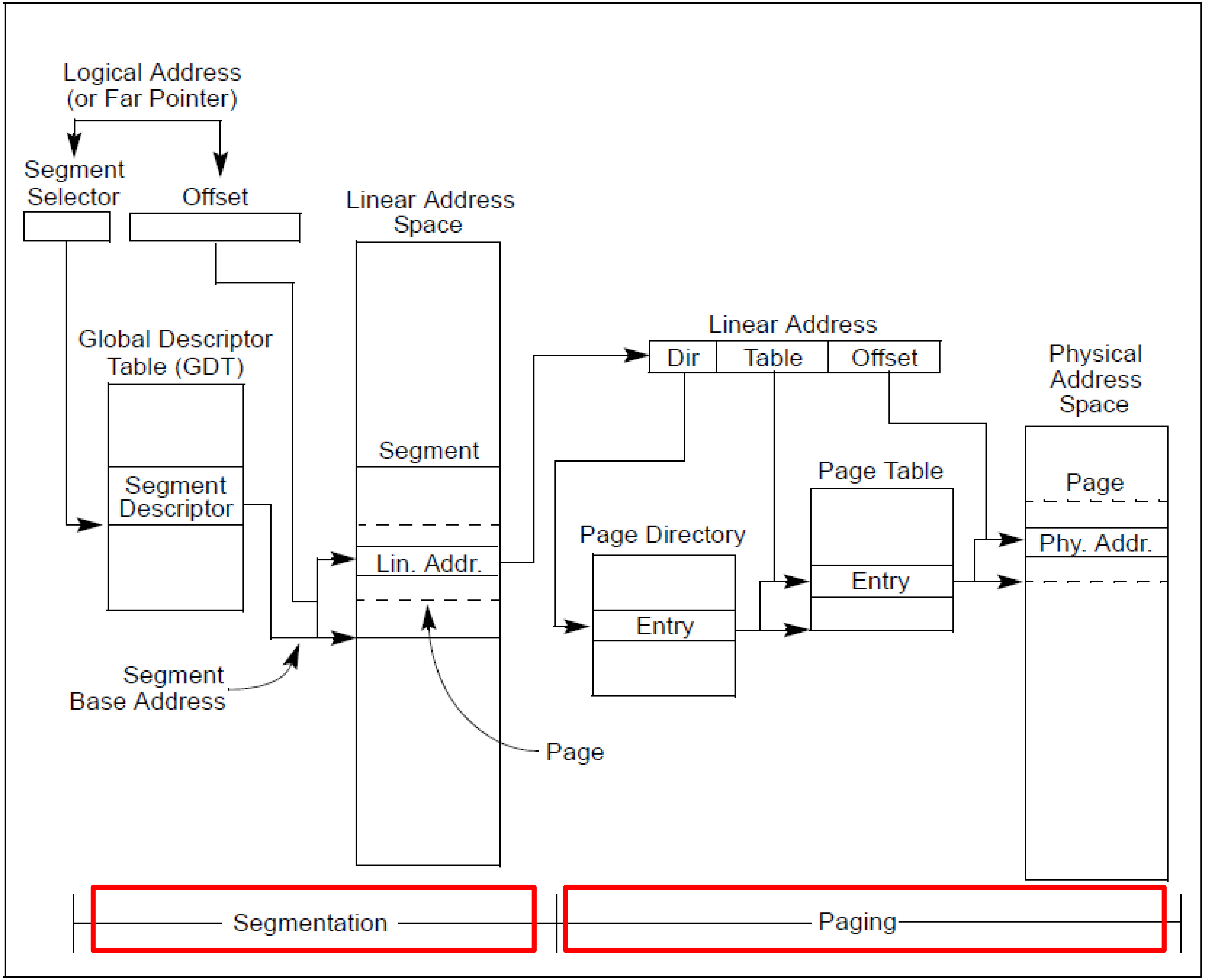


Logical Address
(or Far Pointer)
Segment
Selector
Offset
Linear Address
Space
Global Descriptor
Table (GDT)
Segment
Descriptor
Segment
Lin. Addr.
Segment
Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical
Address
Space
Page
Phy. Addr.
Segmentation
Paging


Logical Address
(or Far Pointer)
Segment Selector
Offset
Linear Address Space
Global Descriptor Table (GDT)
Segment Descriptor
Segment
Lin. Addr.
Segment Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical Address Space
Page
Phy. Addr.
Segmentation
Paging


Logical Address
(or Far Pointer)
Segment Selector
Offset
Linear Address Space
Global Descriptor Table (GDT)
Segment Descriptor
Segment
Lin. Addr.
Segment Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical Address Space
Page
Phy. Addr.
Segmentation
Paging


Logical Address
(or Far Pointer)
Segment Selector
Offset
Linear Address Space
Global Descriptor Table (GDT)
Segment Descriptor
Segment
Lin. Addr.
Segment Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical Address Space
Page
Phy. Addr.
Segmentation
Paging

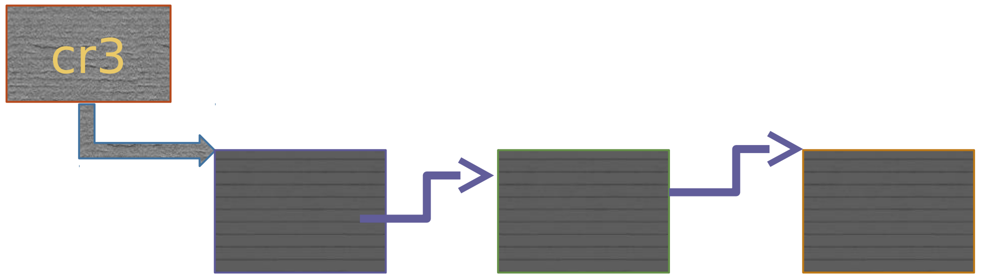

Logical Address
(or Far Pointer)
Segment Selector
Offset
Linear Address Space
Global Descriptor Table (GDT)
Segment Descriptor
Segment
Lin. Addr.
Segment Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical Address Space
Page
Phy. Addr.
Segmentation
Paging

Questions?

References

TLB

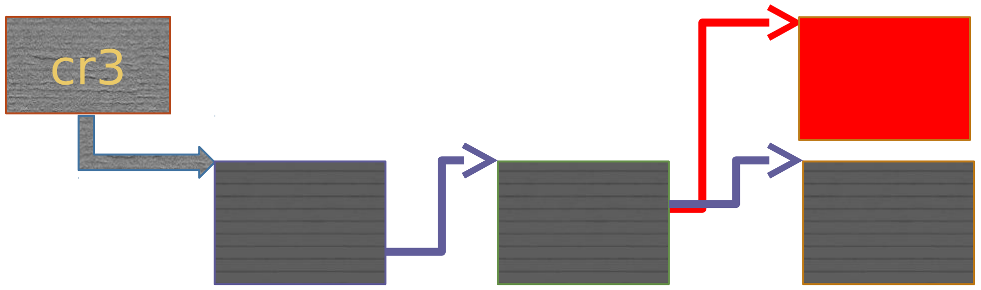
- CPU caches results of page table walks
  - In translation lookaside buffer (TLB)
- Walking page table is slow
  - Each memory access is 200-300 cycles on modern hardware
  - L3 cache access is 70 cycles



Virt	Phys
0xf0231000	0x1000
0x00b31000	0x1f000
0xb0002000	0xc1000
-	-

TLB

- 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



Virt	Phys
0xf0231000	0x1000
0x00b31000	0x1f000
0xb0002000	0xc1000
-	-

Same Virt Addr.

No Change!!!

Invalidating TLB

- After every page table update, OS needs to manually invalidate cached values
- 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)

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. lightweigh `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

When would you disable paging?

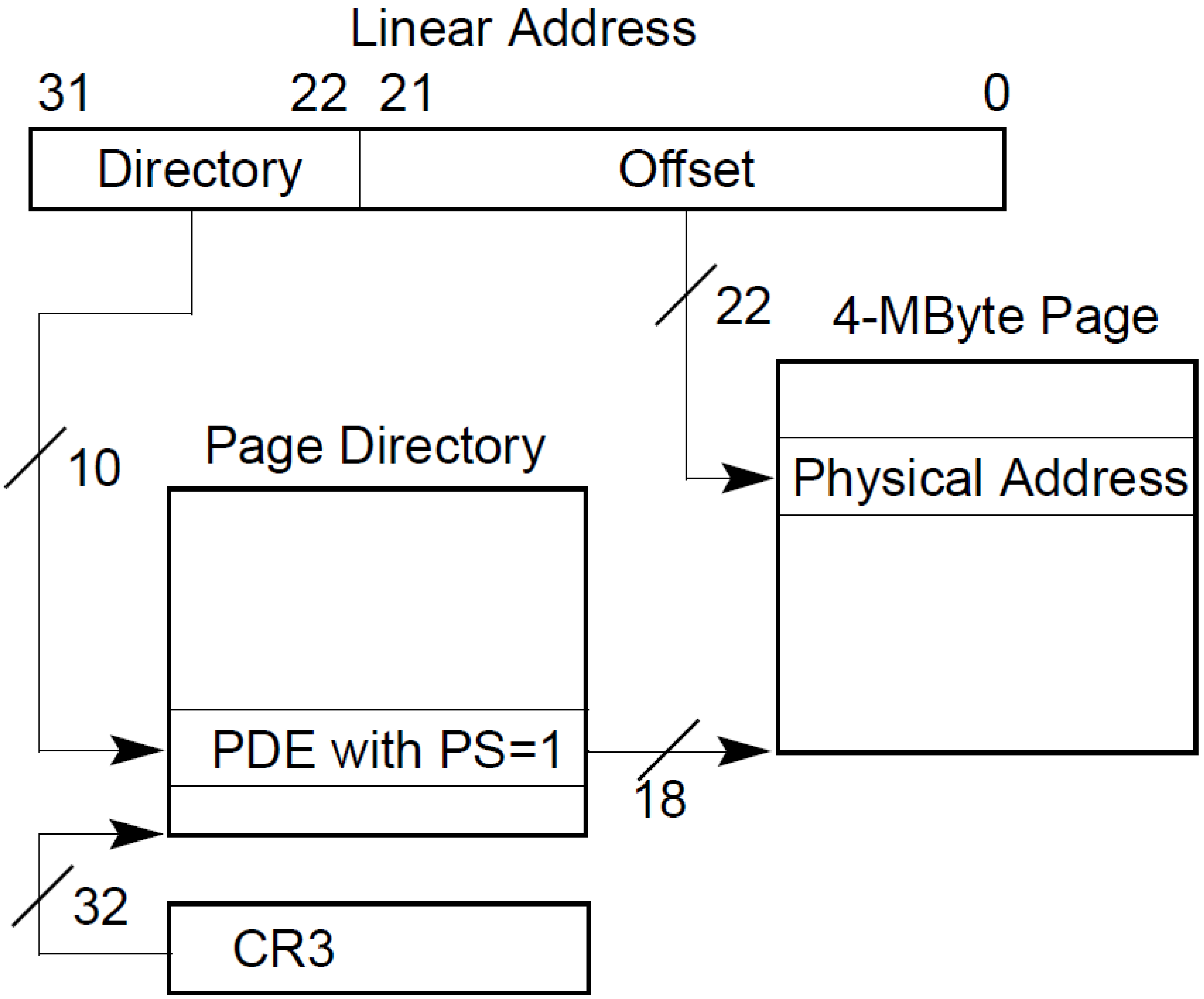
When would you disable paging?

- 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
-

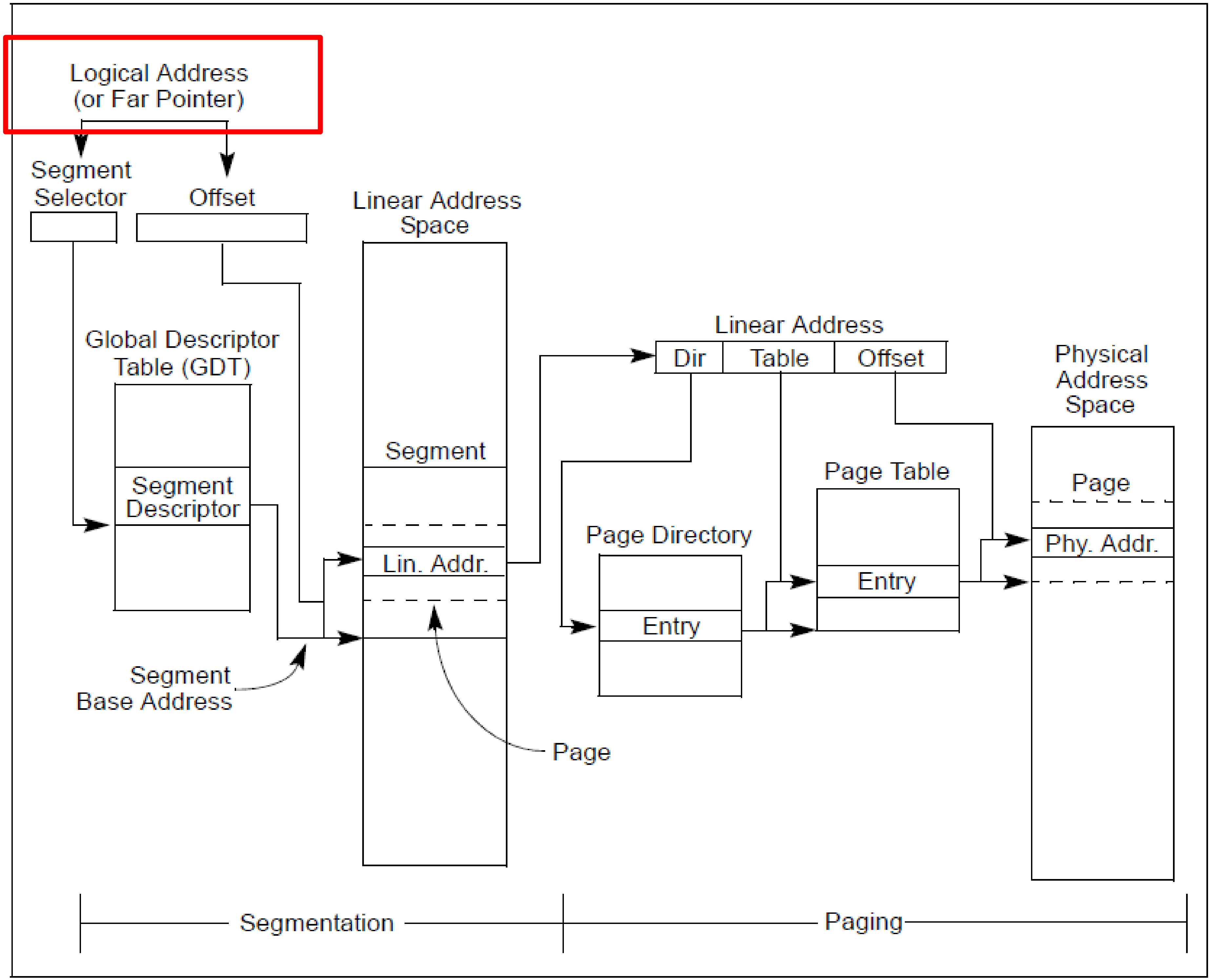
When would you disable paging?

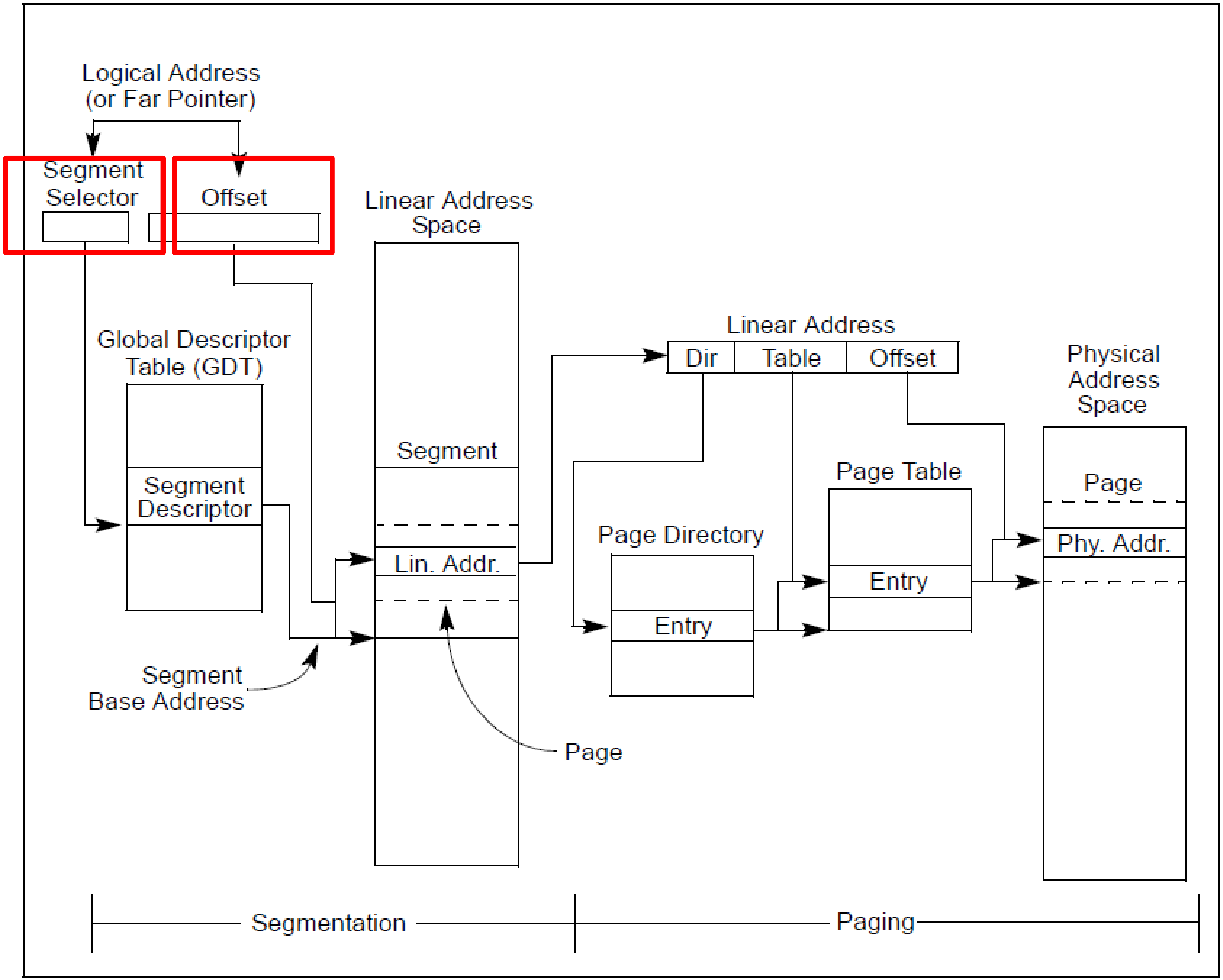
- Now, to cover 32GB RAM with 4K pages
  - You need 64MB space
  - 64bit architecture, 3-level page tables
- Page tables do not fit in L3 cache
  - Modern servers come with 32MB cache
- Every cache miss results in up to 3 cache misses due to page walk (remember 3-level page tables)
  - Each cache miss is 200 cycles
- Solution: 1GB pages

Page translation for 4MB pages

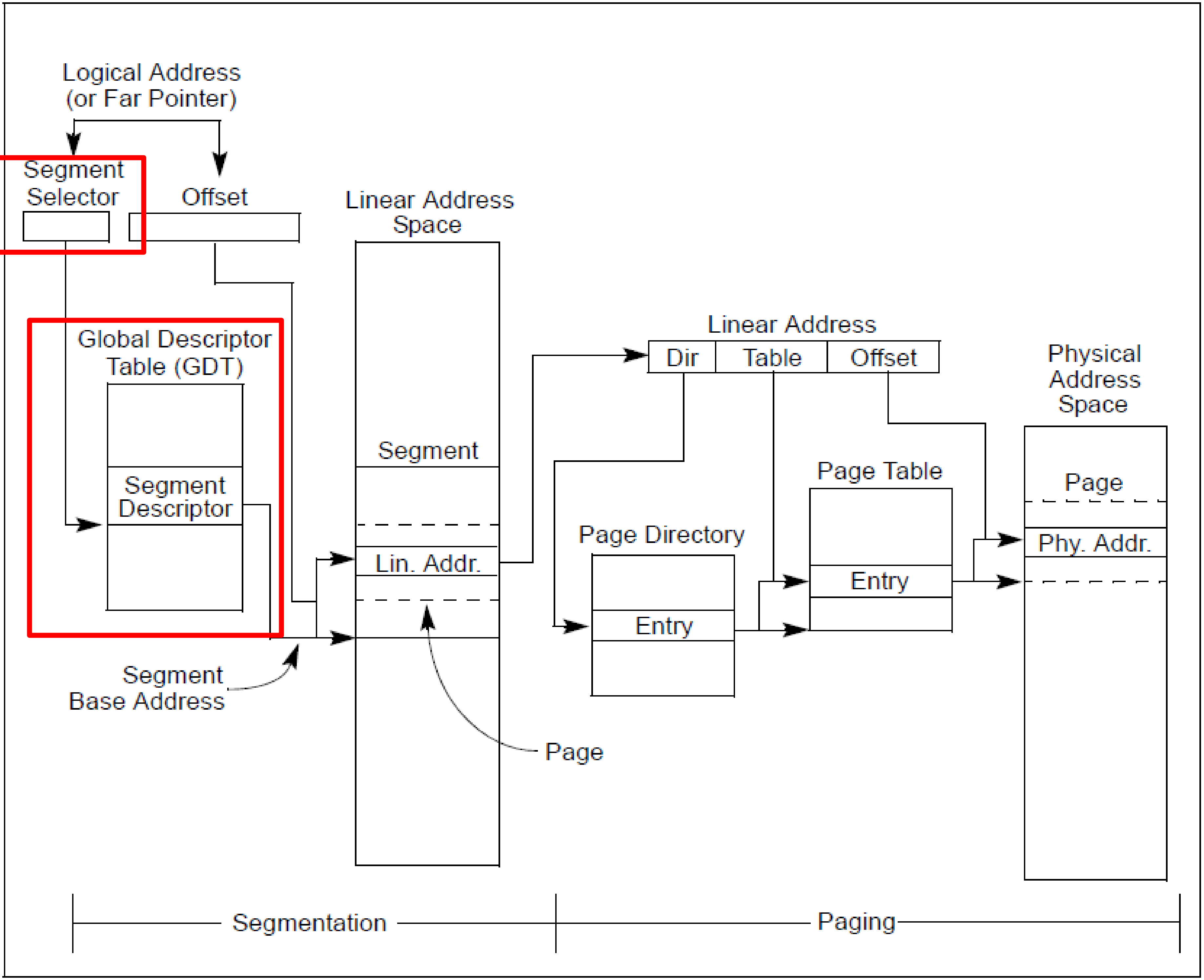


Segmentation

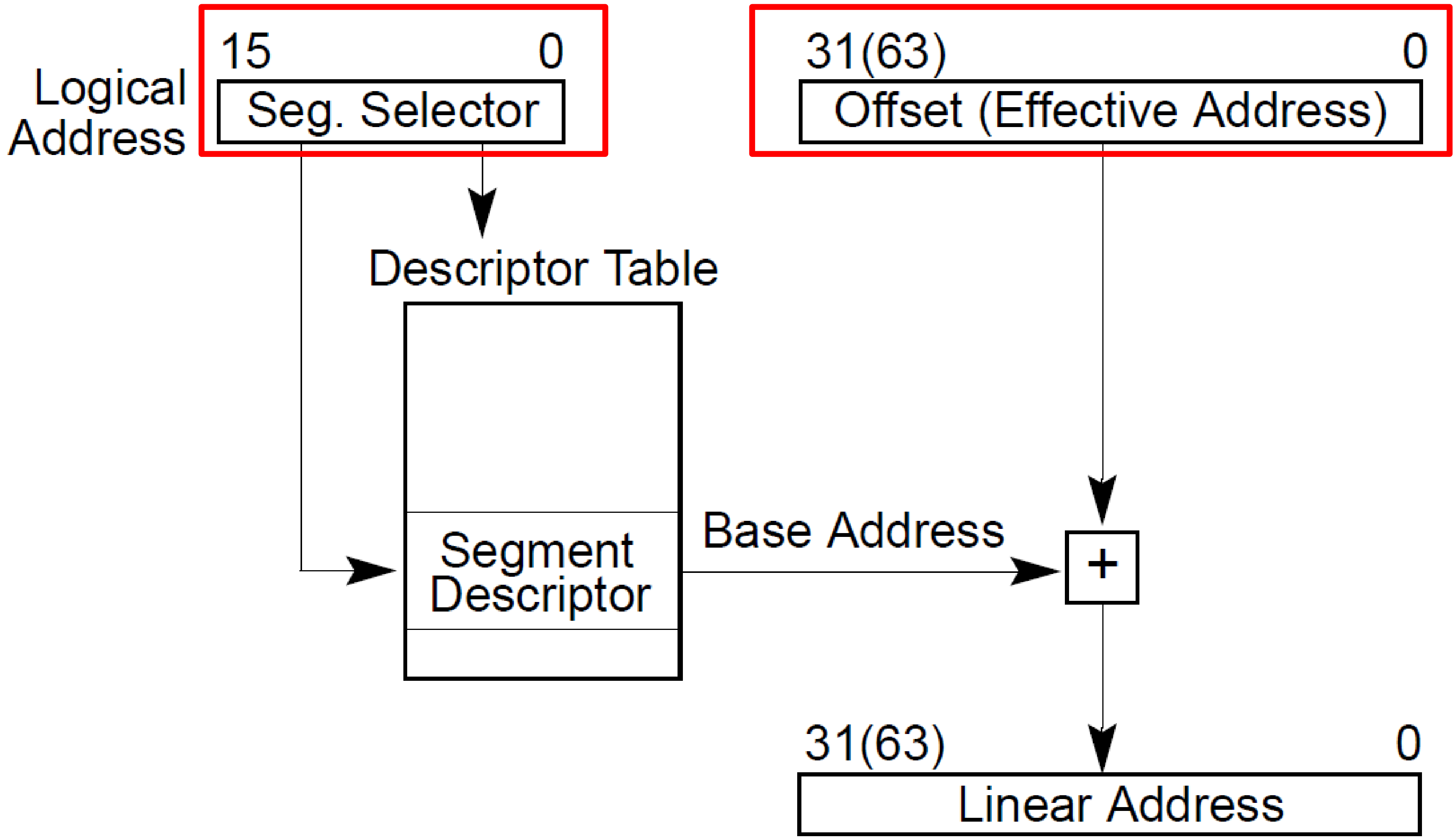

Logical Address
(or Far Pointer)
Segment Selector
Offset
Linear Address Space
Global Descriptor Table (GDT)
Segment Descriptor
Segment
Lin. Addr.
Segment Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical Address Space
Page
Phy. Addr.
Segmentation
Paging



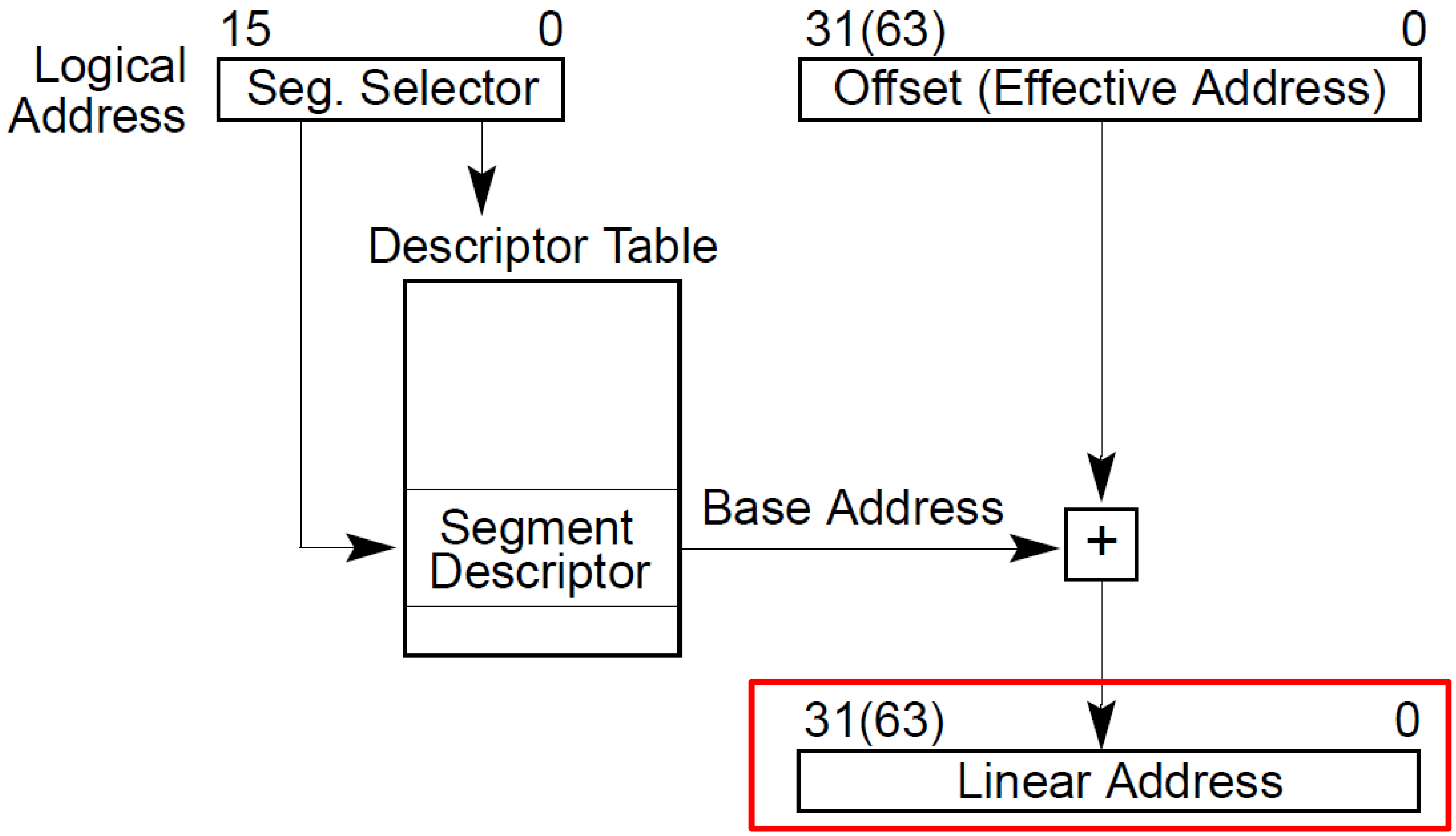
Logical Address
(or Far Pointer)
Segment Selector
Offset
Linear Address Space
Global Descriptor Table (GDT)
Segment Descriptor
Segment
Lin. Addr.
Segment Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical Address Space
Page
Phy. Addr.
Segmentation
Paging

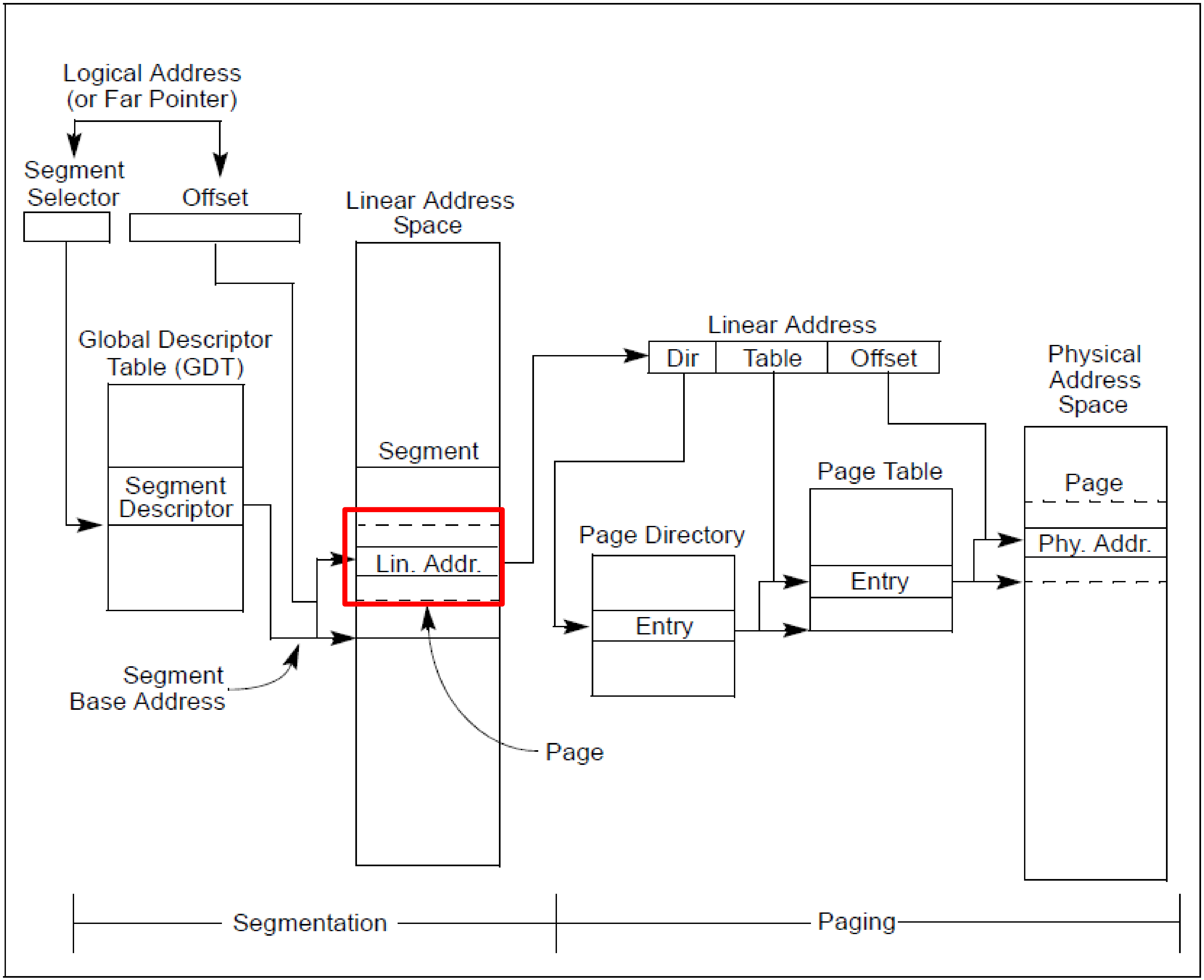

Logical Address
(or Far Pointer)
Segment Selector
Offset
Linear Address Space
Global Descriptor Table (GDT)
Segment Descriptor
Segment
Lin. Addr.
Segment Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical Address Space
Page
Phy. Addr.
Segmentation
Paging

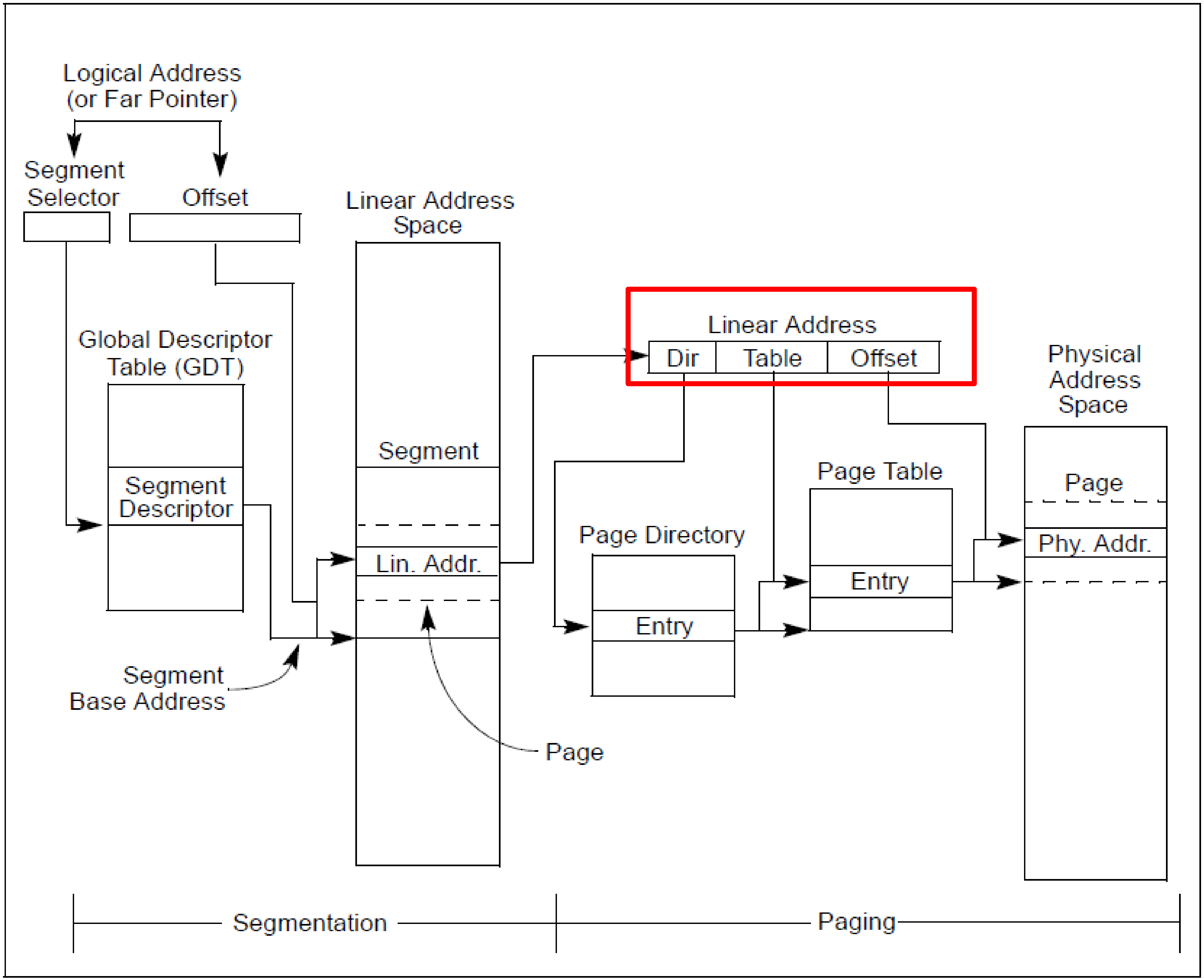
Descriptor table

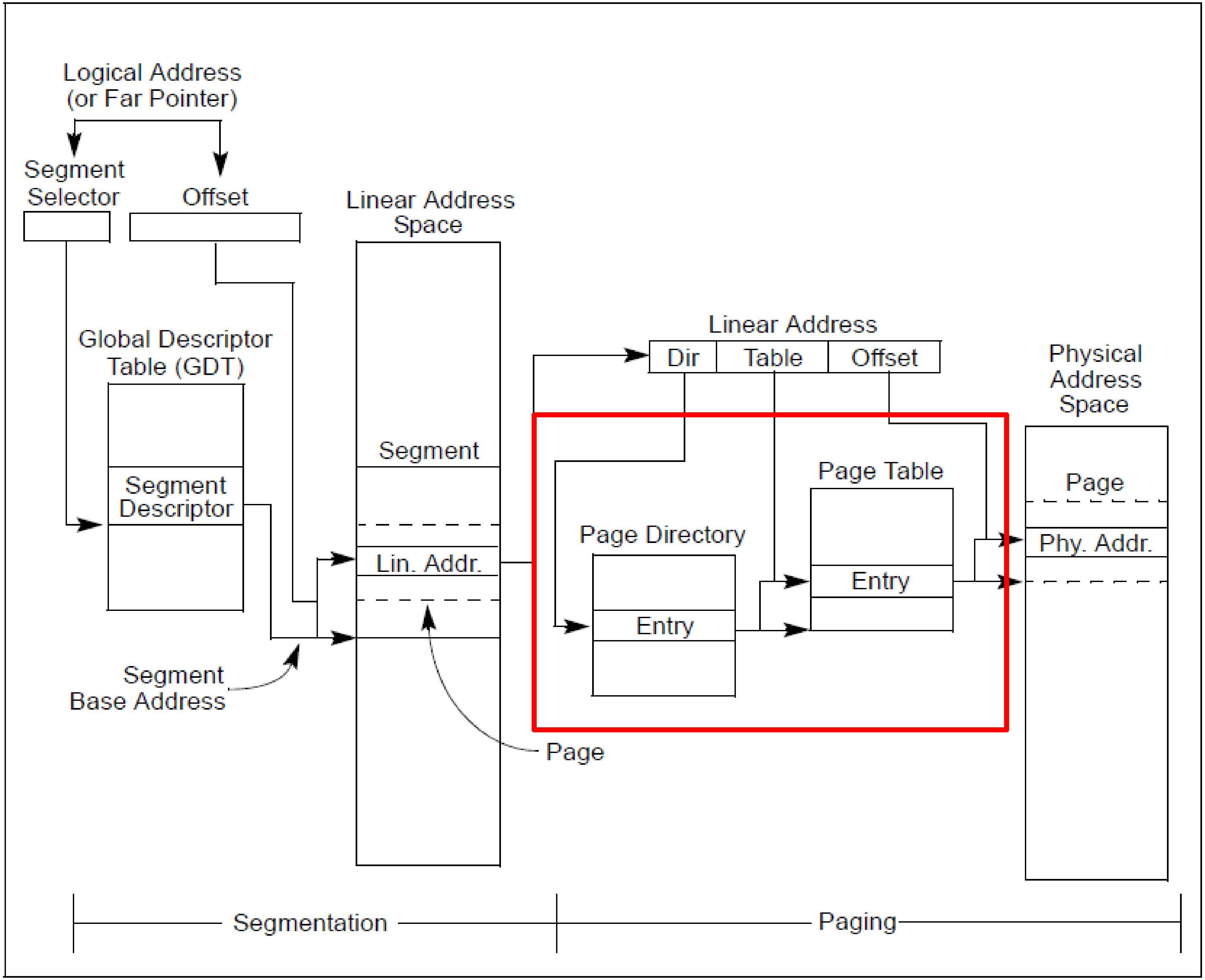


Descriptor table

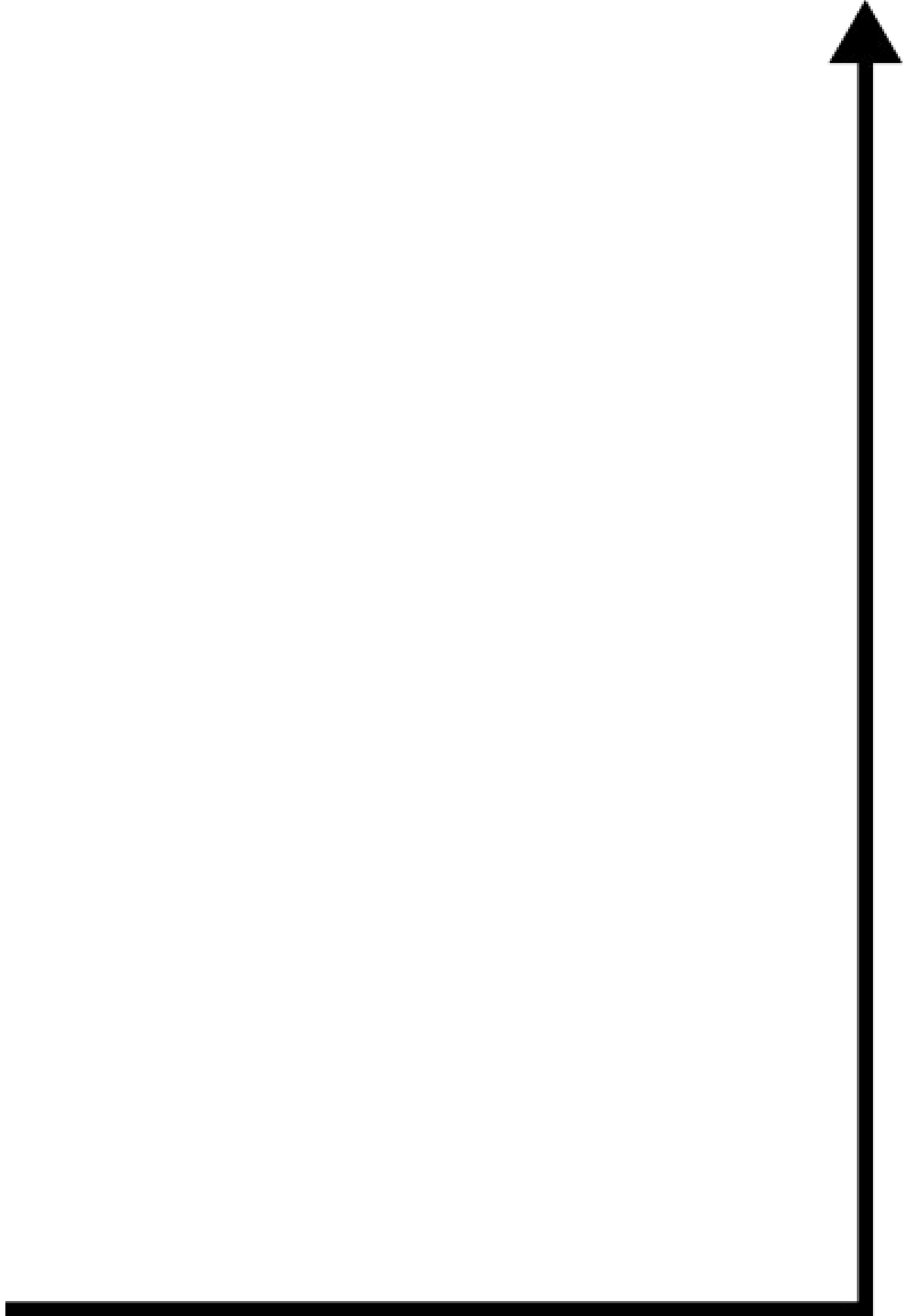


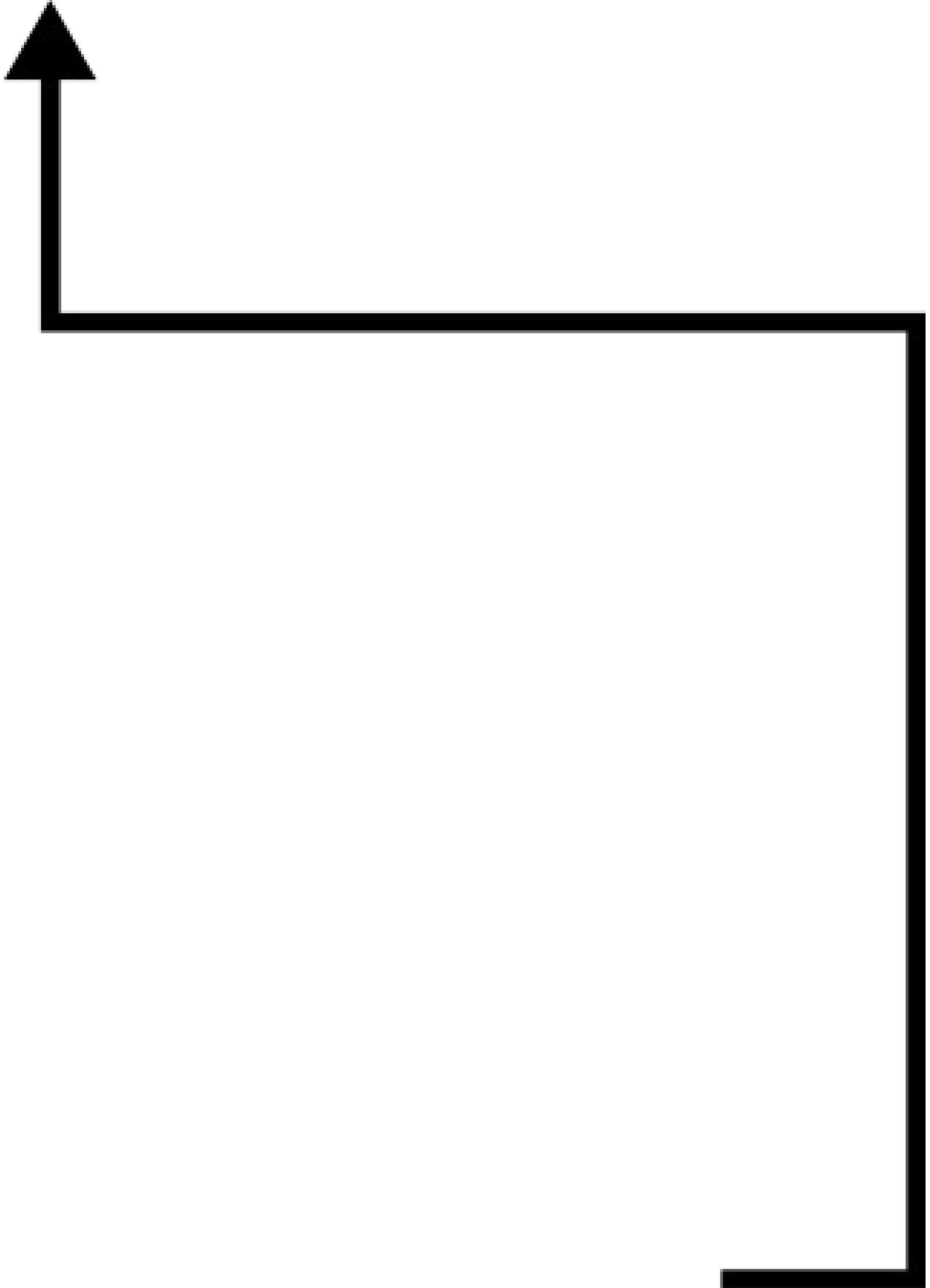

Logical Address
(or Far Pointer)
Segment Selector
Offset
Linear Address Space
Global Descriptor Table (GDT)
Segment Descriptor
Segment
Lin. Addr.
Segment Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical Address Space
Page
Phy. Addr.
Segmentation
Paging

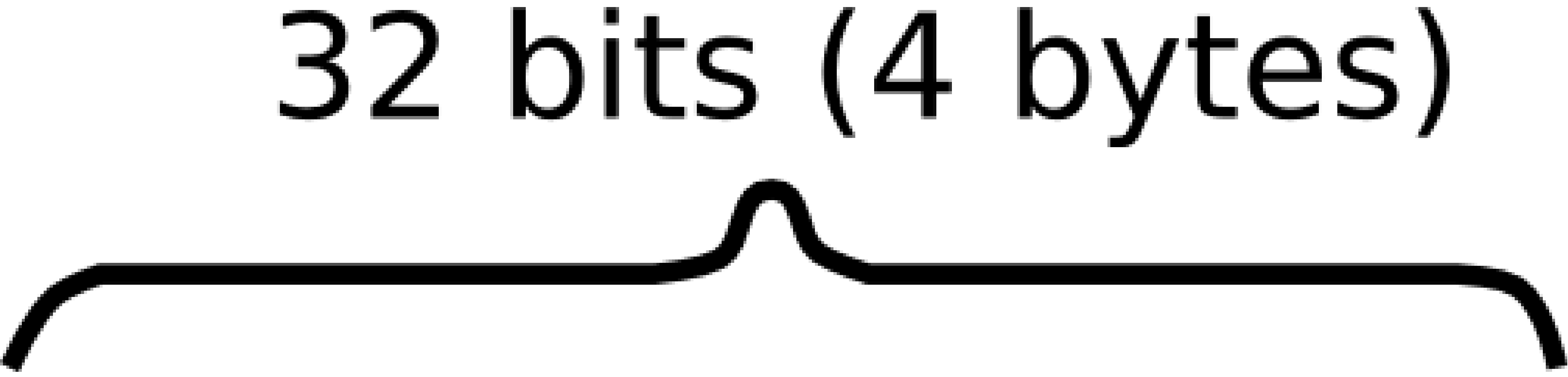

Logical Address (or Far Pointer)
Segment Selector
Offset
Linear Address Space
Global Descriptor Table (GDT)
Segment Descriptor
Segment
Lin. Addr.
Segment Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical Address Space
Page
Phy. Addr.
Segmentation
Paging


Logical Address
(or Far Pointer)
Segment Selector
Offset
Linear Address Space
Global Descriptor Table (GDT)
Segment Descriptor
Segment
Lin. Addr.
Segment Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical Address Space
Page
Phy. Addr.
Segmentation
Paging

1M (1,048,575)



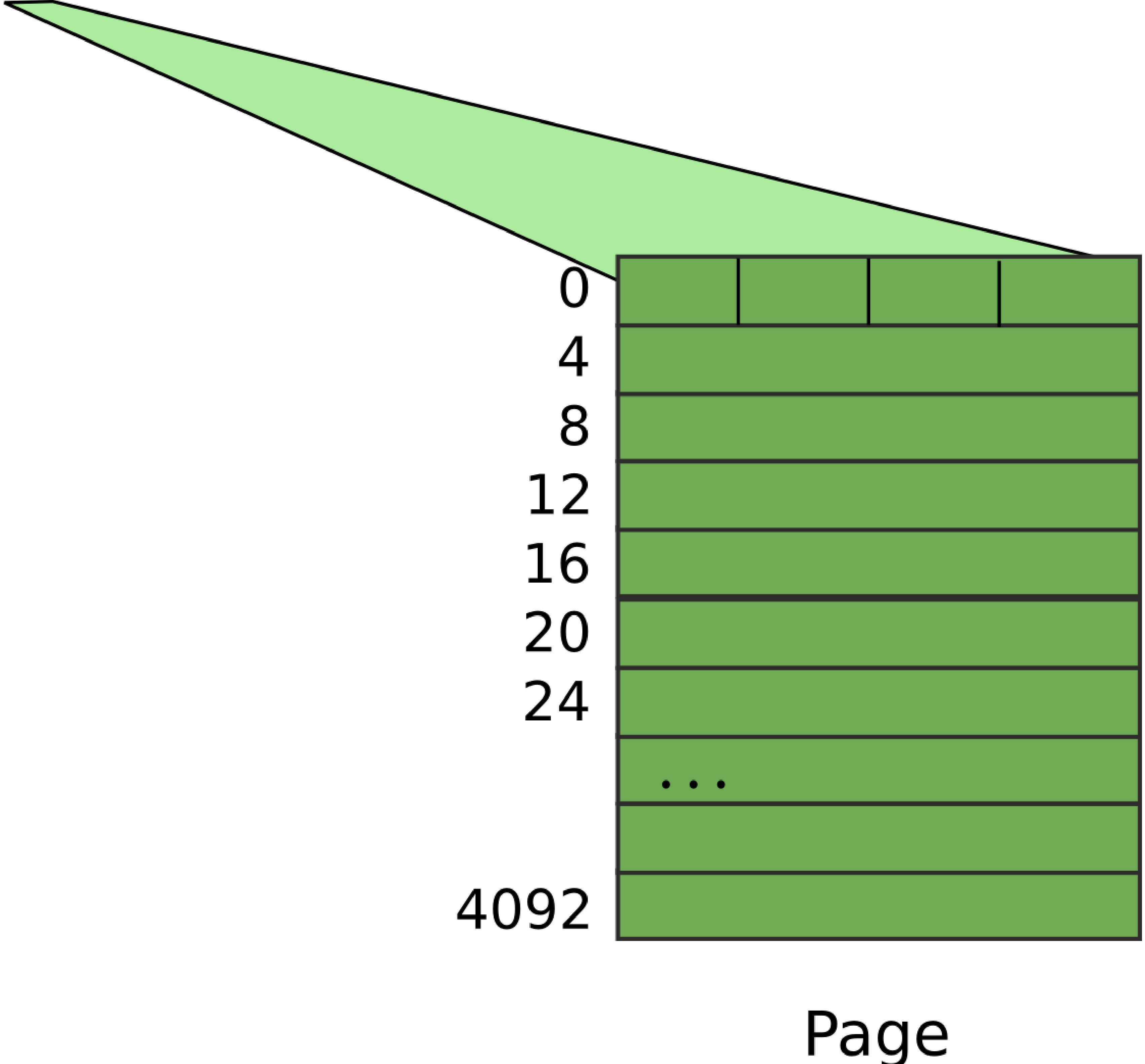


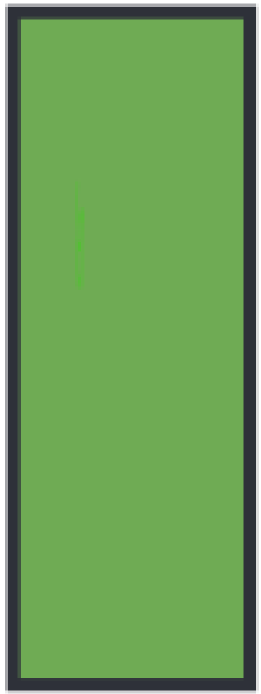

32 bits (4 bytes)

CR3 = 0

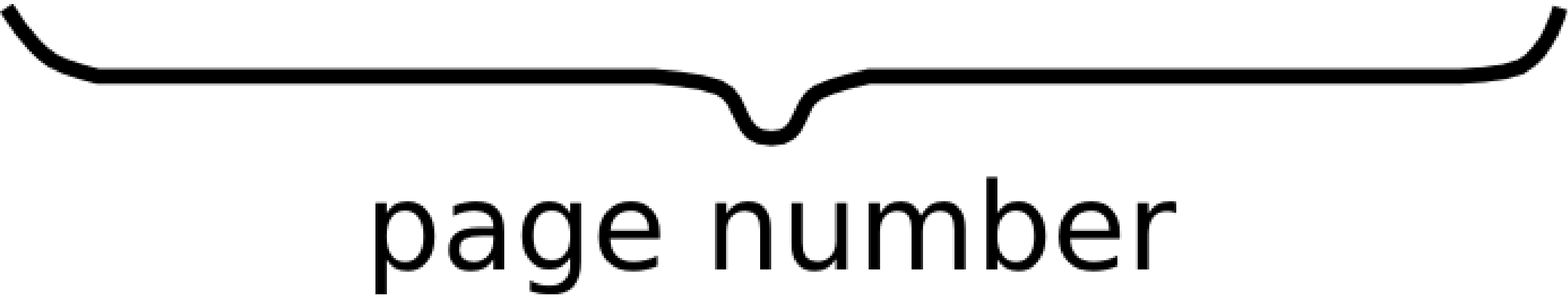
```
mov (%EBX), EAX   # mov value from the  location pointed by EBX into EAX
EAX = 0
EBX = 20 983 809
```

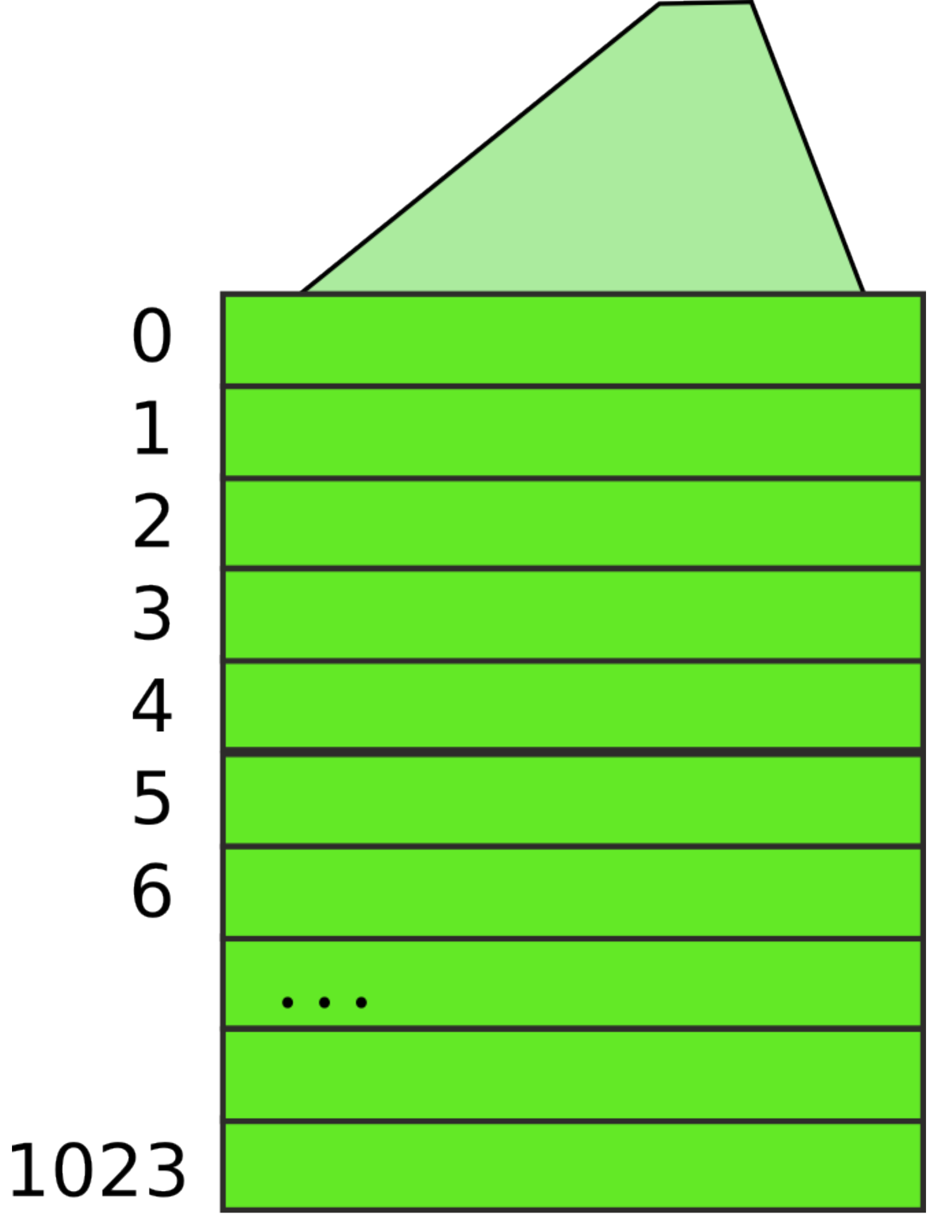
20 983 809 =	00 0000 0101	00 0000 0011	0000 0000 0001


0
4
8
12
16
20
24
. . .
4092
Page

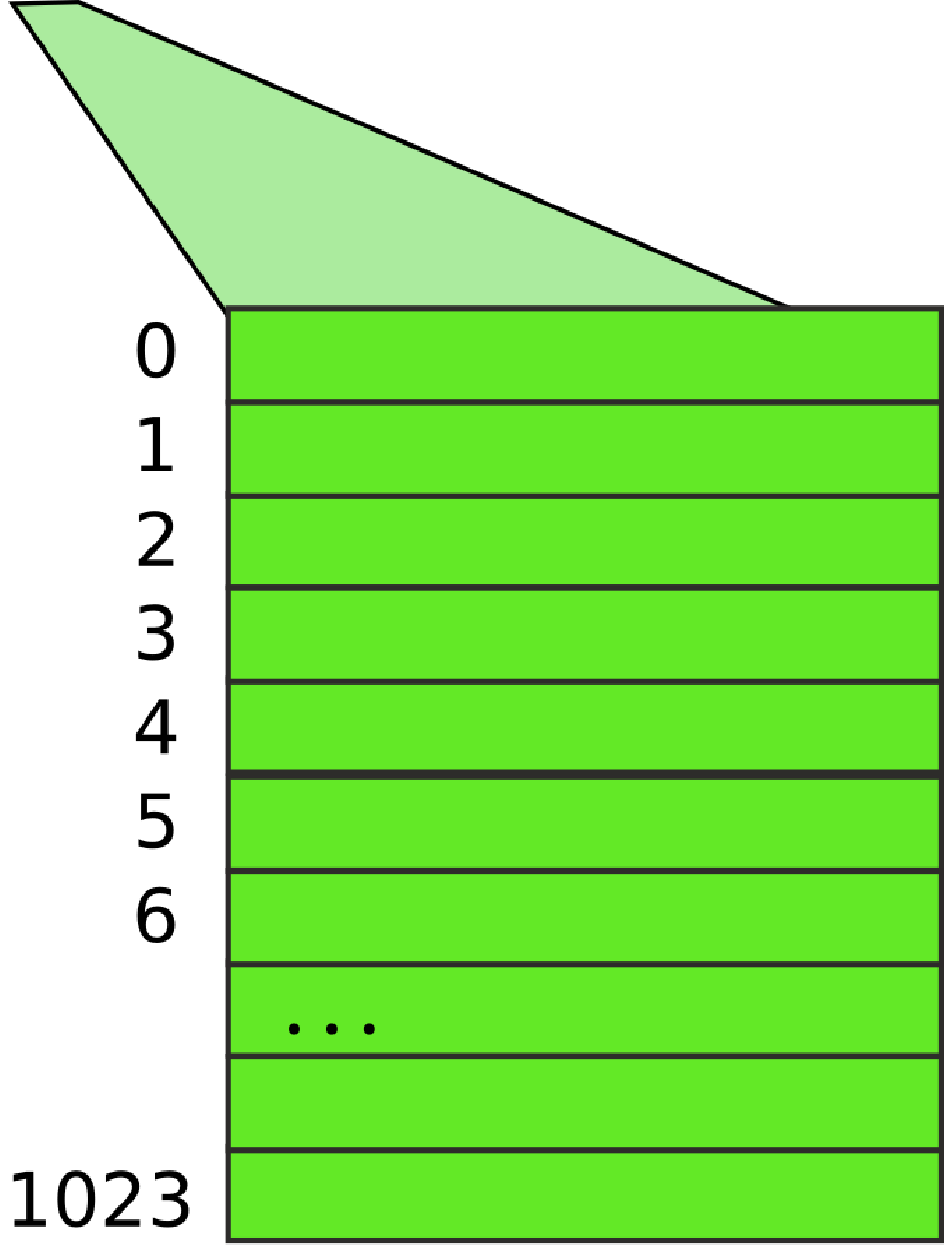


page number = 5123
or (0b1 0100 0000 0011)

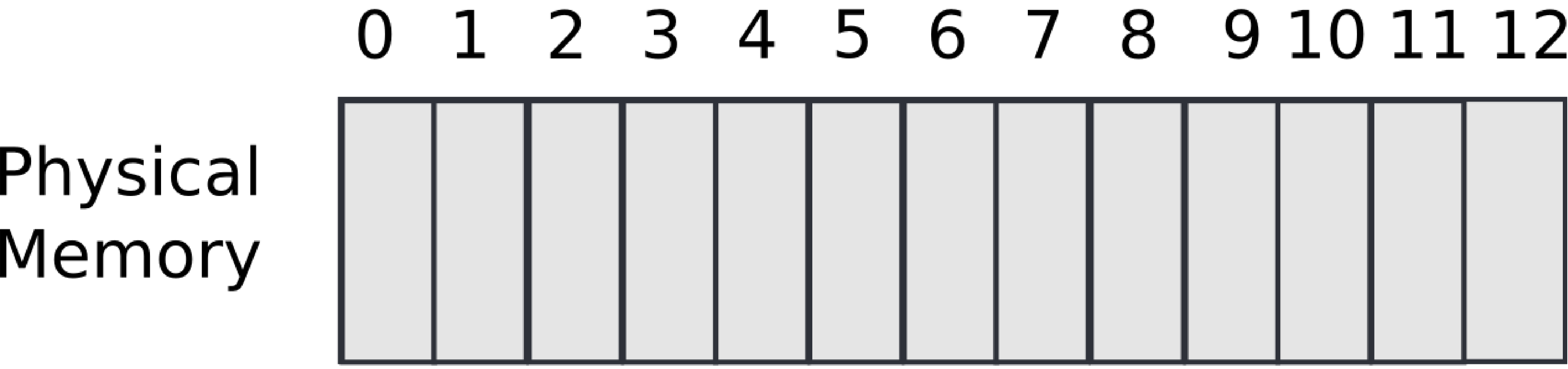

page number



Level 1
(Page Table
Directory)



Level 2
(Page Table)


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