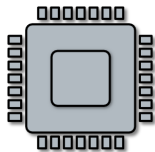
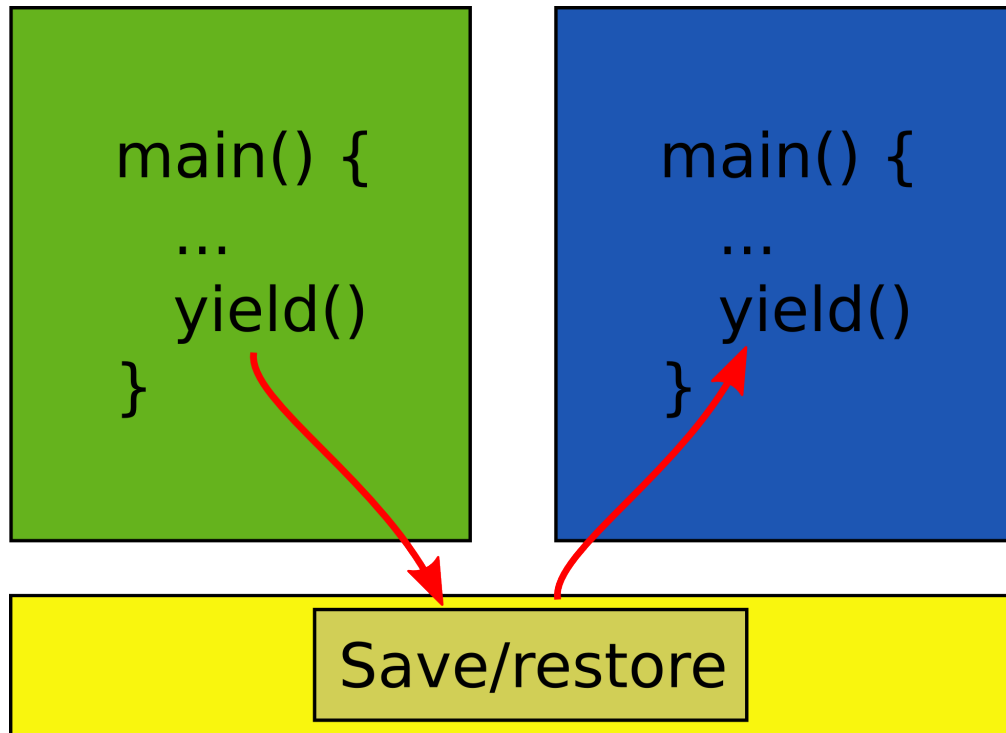


238P: Operating Systems

Lecture 5: Address translation

Anton Burtsev
January, 2018

Two programs one memory



Very much like car sharing



Car rental

What are we aiming for?

- Illusion of a private address space
 - Identical copy of an address space in multiple programs
 - Remember `fork()`?
 - Simplifies software architecture
 - One program is not restricted by the memory layout of the others

Two processes, one memory?

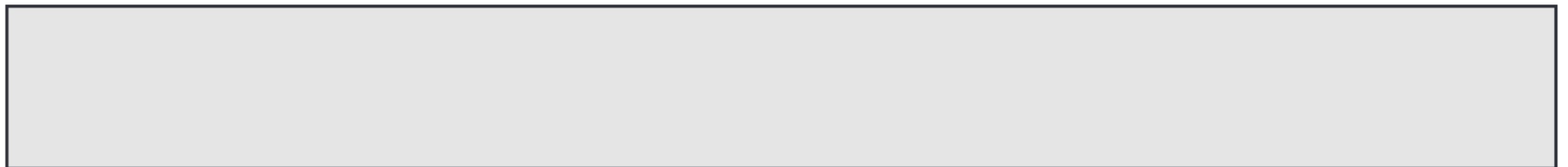
Process 1 (ls)



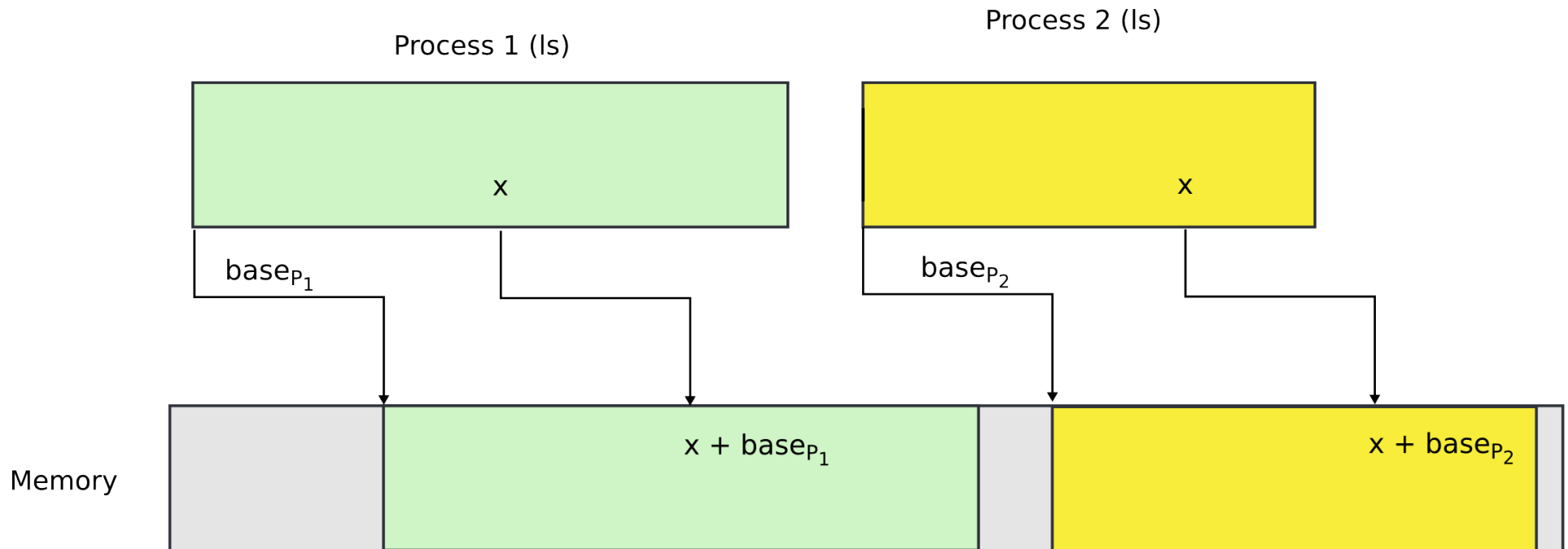
Process 2 (ls)



Memory



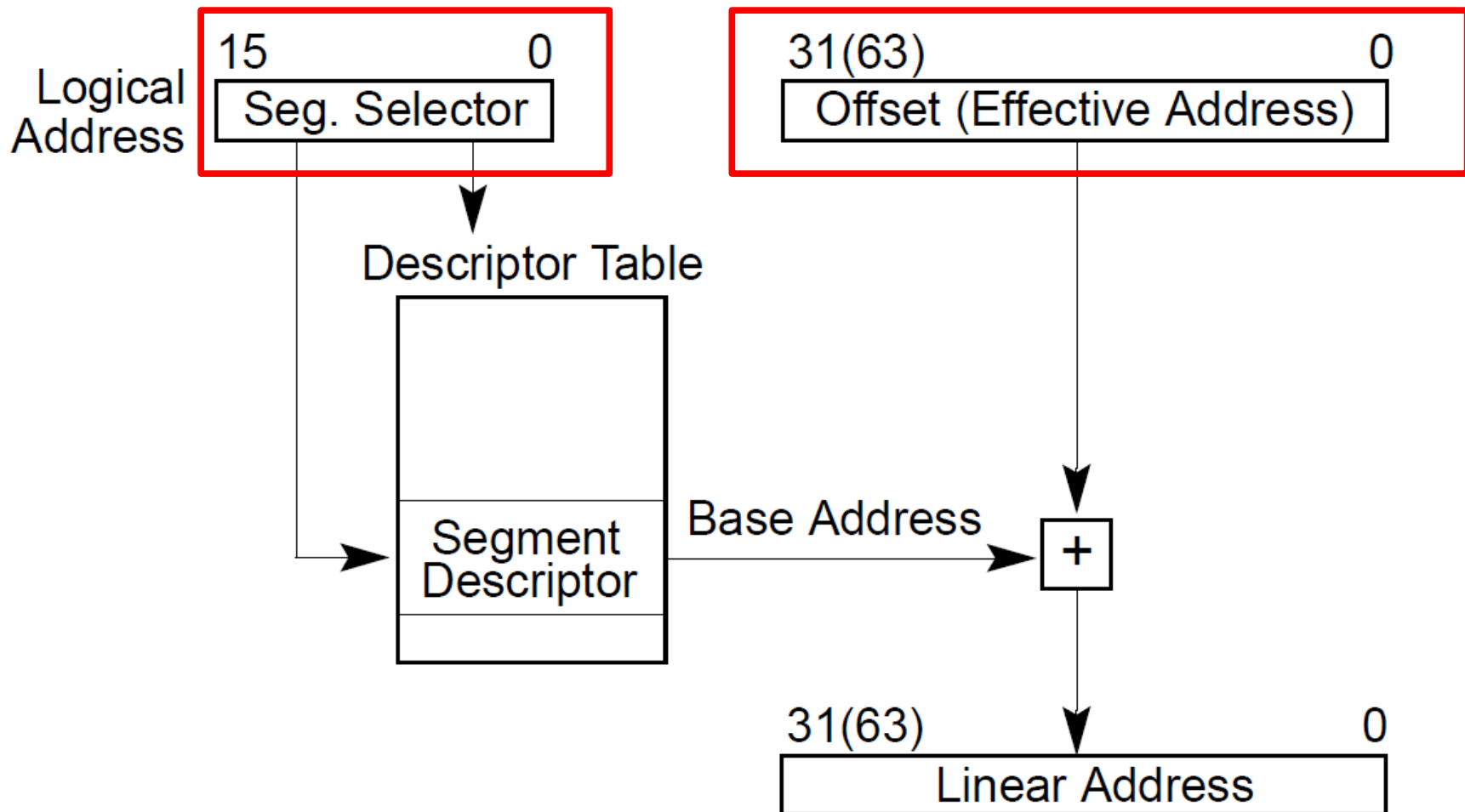
Two processes, one memory?



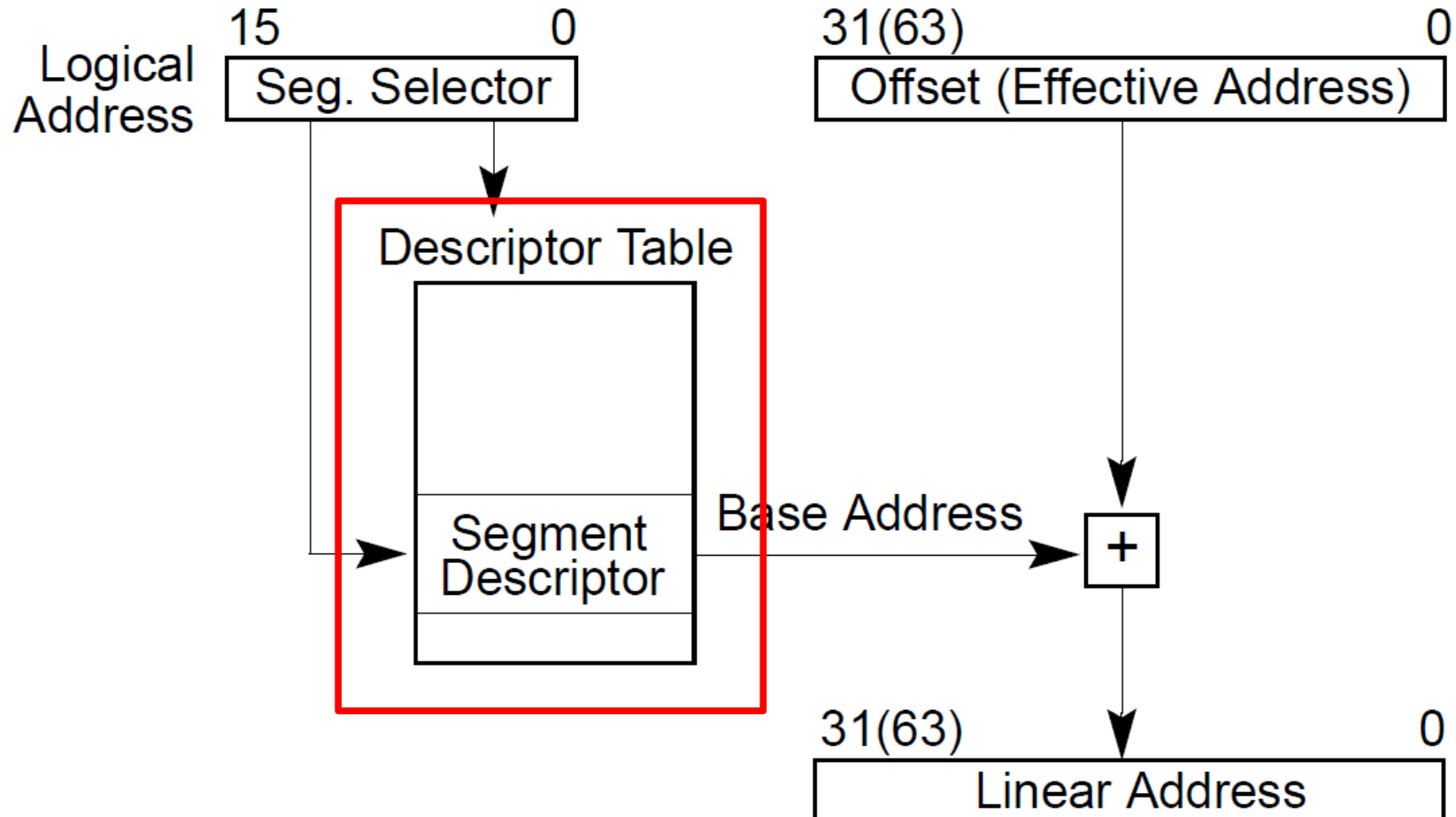
This is called segmentation

All addresses are logical address

- They consist of two parts
 - Segment selector (16 bit) + offset (32 bit)



- Segment selector (16 bit)
 - Is simply an index into an array (Descriptor Table)
 - That holds segment descriptors
 - Base and limit (size) for each segment



Elements of that array are **segment descriptors**

- Base address

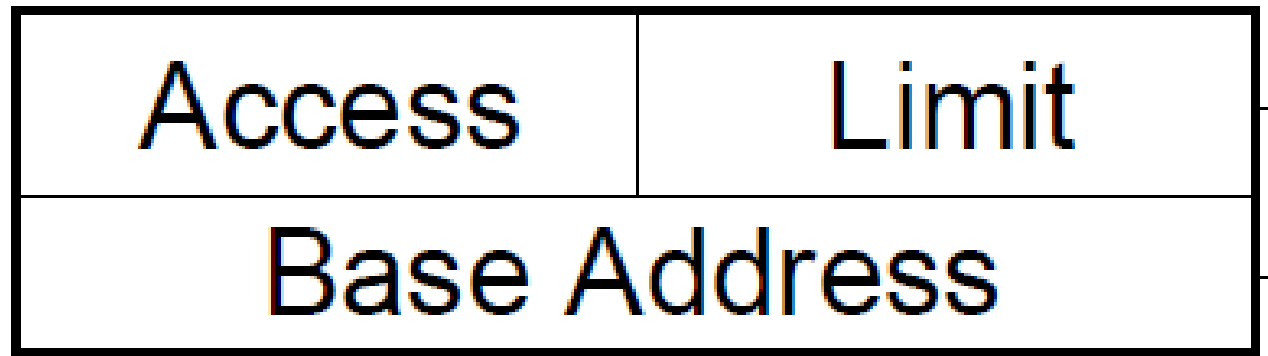
- 0 – 4 GB

- Limit (size)

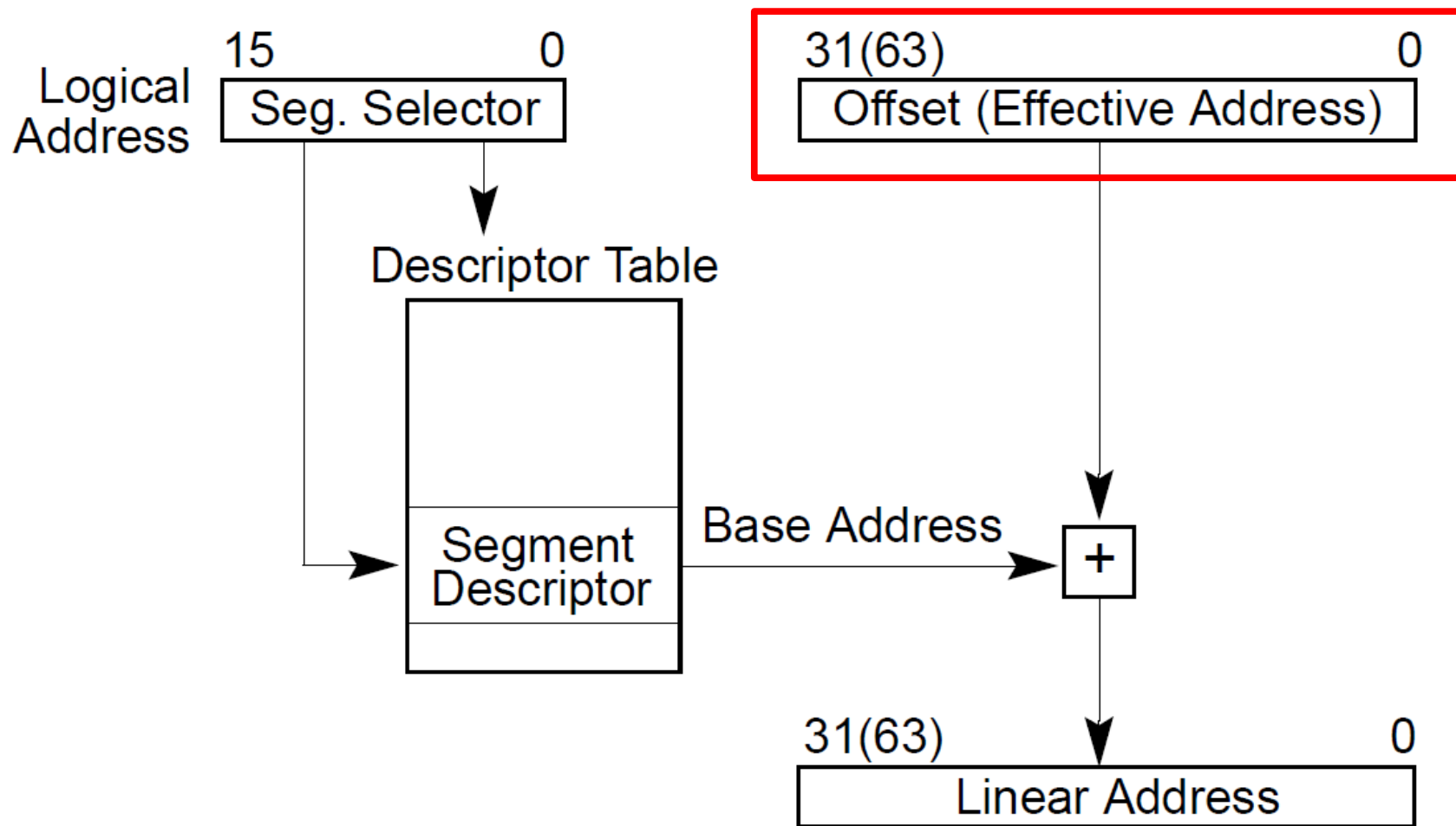
- 0 – 4 GB

- Access rights

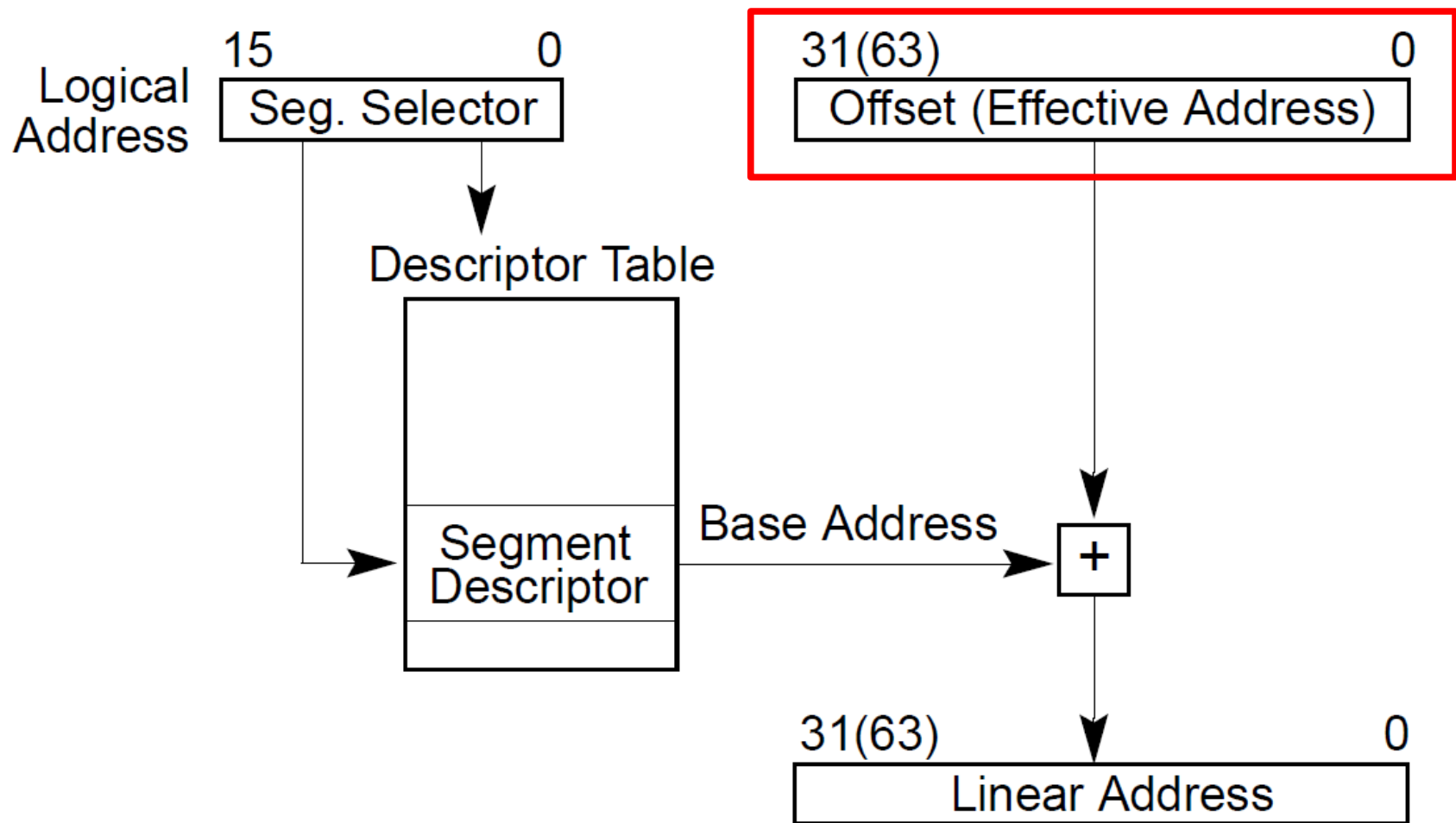
- Executable, readable, writable
- Privilege level (0 - 3)



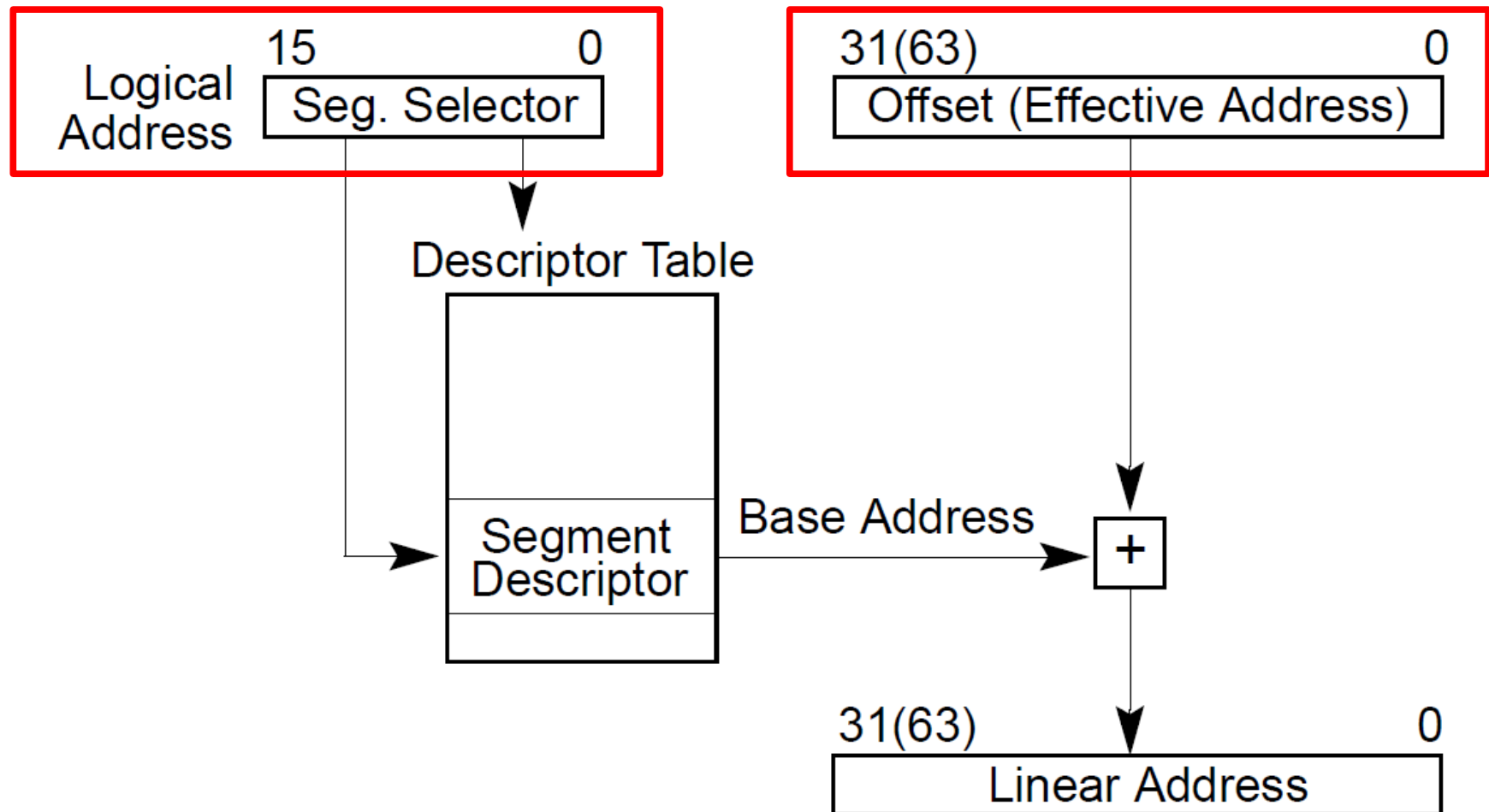
- Offsets into segments (x in our example) or “Effective addresses” are in registers



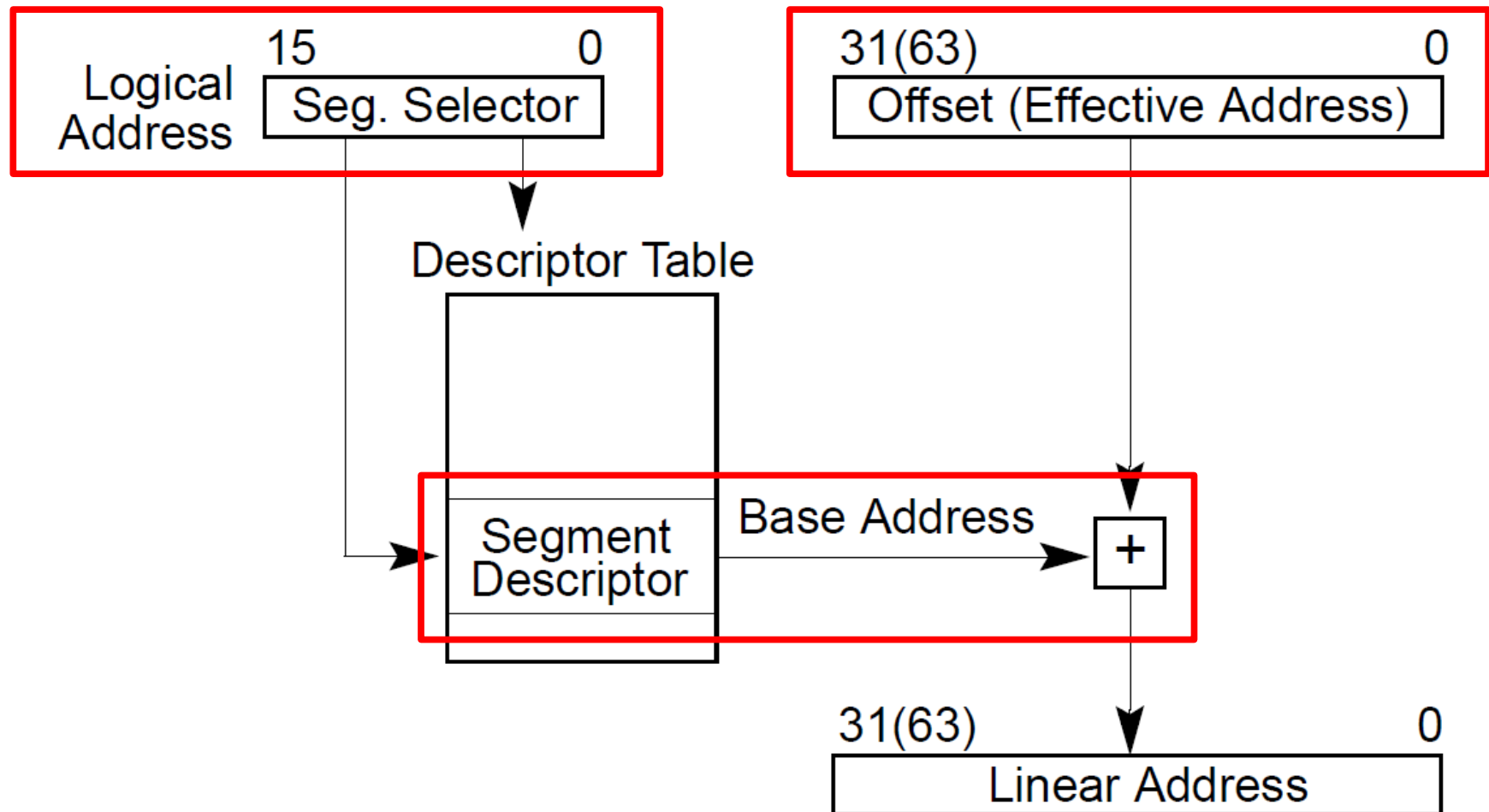
- Logical addresses are translated into physical
 - *Effective address + DescriptorTable[selector].Base*



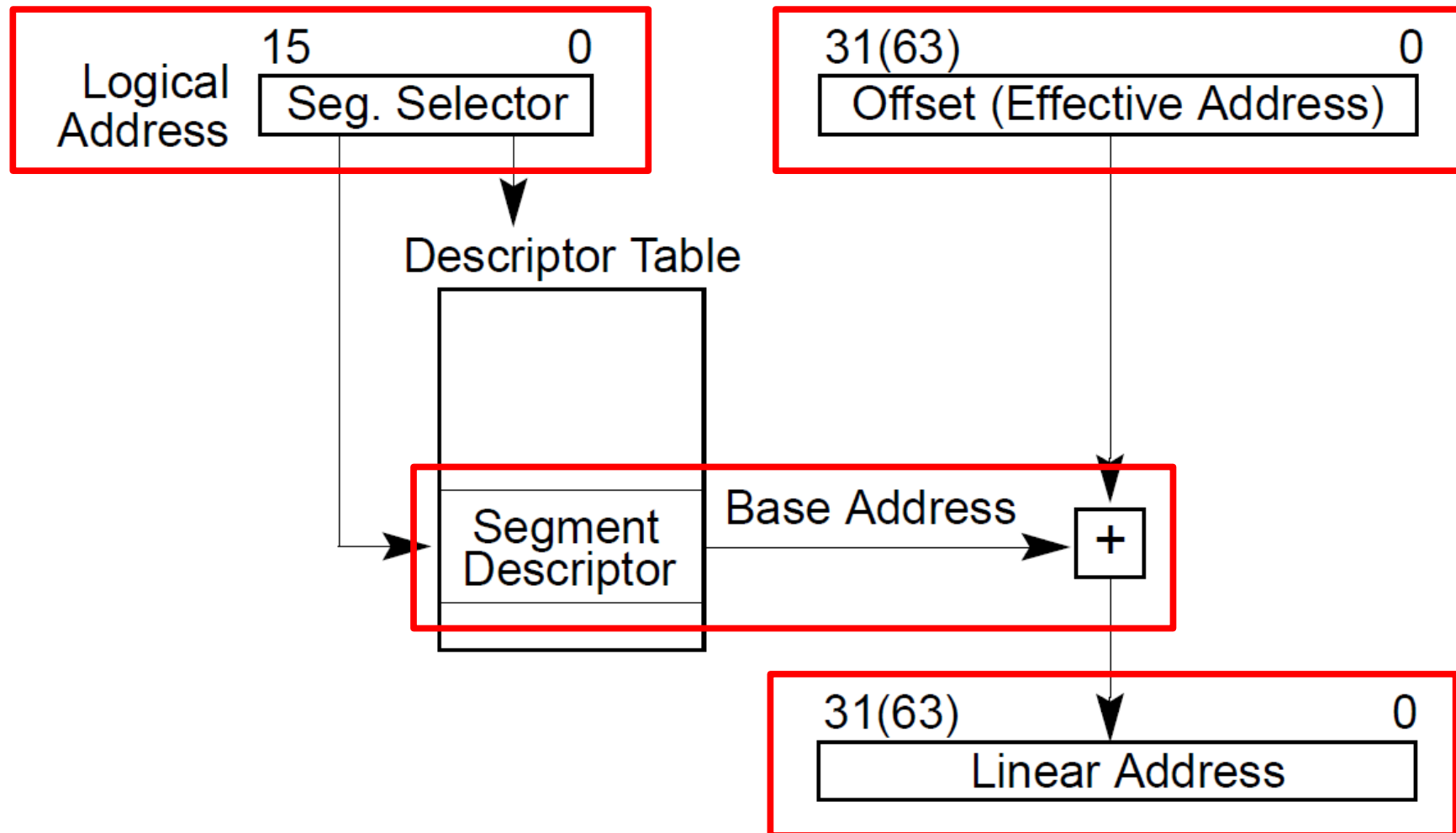
- Logical addresses are translated into physical
 - Effective address + DescriptorTable[selector].Base



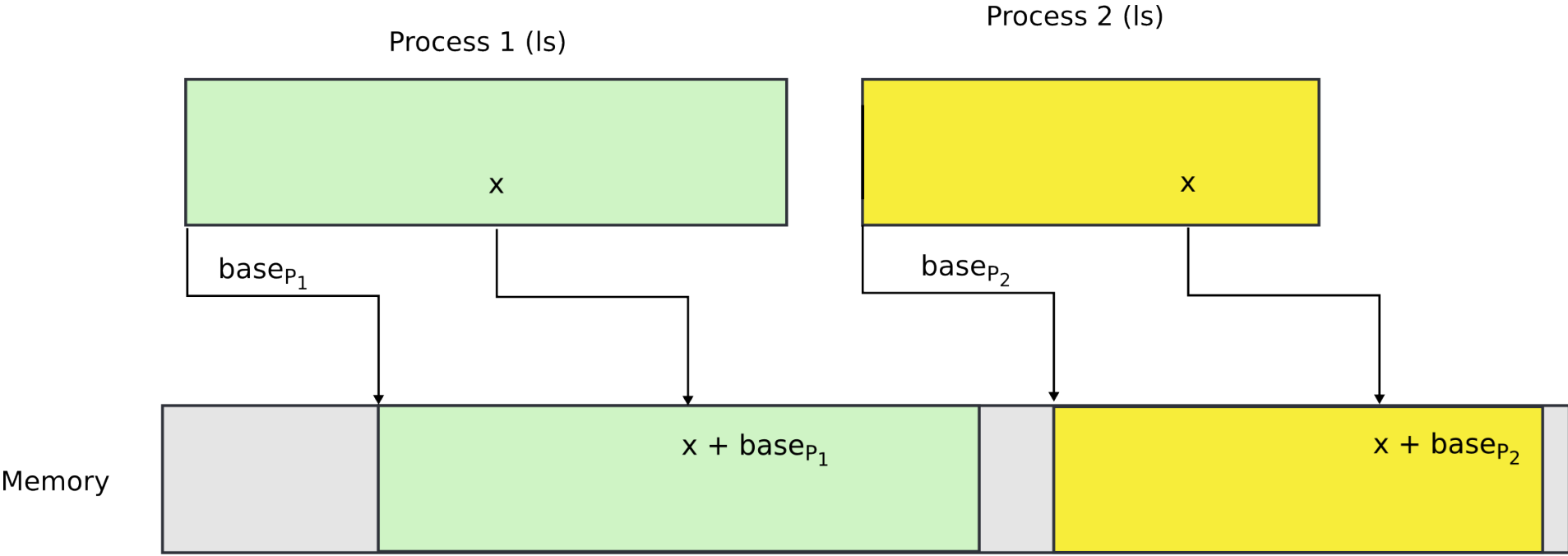
- Logical addresses are translated into physical
 - Effective address + DescriptorTable[selector].Base



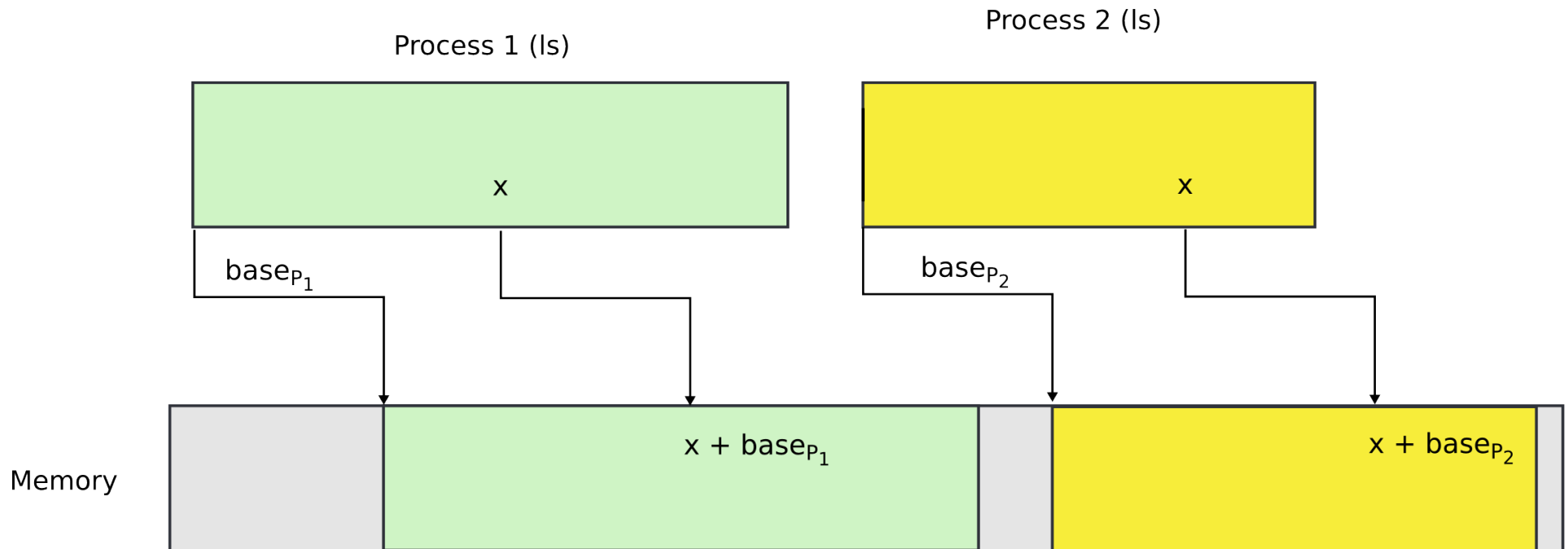
- Logical addresses are translated into physical
 - Effective address + DescriptorTable[selector].Base



Same picture



- Offsets (effective addresses) are in registers
 - *Effective address* + *DescriptorTable[selector].Base*
 - **But where is the selector?**



Right! Segment registers

- Hold 16 bit segment selectors
 - Pointers into a special table
 - Global or local descriptor table
- Segments are associated with one of three types of storage
 - Code
 - Data
 - Stack

Programming model

- Segments for: code, data, stack, “extra”
 - A program can have up to 6 total segments
 - Segments identified by registers: cs, ds, ss, es, fs, gs
- Prefix all memory accesses with desired segment:
 - `mov eax, ds:0x80` (load offset 0x80 from data into eax)
 - `jmp cs:0xab8` (jump execution to code offset 0xab8)
 - `mov ss:0x40, ecx` (move ecx to stack offset 0x40)

Segmented programming (not real)

```
static int x = 1;
int y; // stack
if (x) {
    y = 1;
    printf ("Boo");
} else
    y = 0;
```

```
ds:x = 1; // data
ss:y; // stack
if (ds:x) {
    ss:y = 1;
    cs:printf(ds:"Boo");
} else
    ss:y = 0;
```

Programming model, cont.

- This is cumbersome, so infer code, data and stack segments by instruction type:
 - Control-flow instructions use code segment (jump, call)
 - Stack management (push/pop) uses stack
 - Most loads/stores use data segment
- Extra segments (es, fs, gs) must be used explicitly

Code segment

- Code
 - CS register
 - EIP is an offset inside the segment stored in CS
- Can only be changed with
 - procedure calls,
 - interrupt handling, or
 - task switching

Data segment

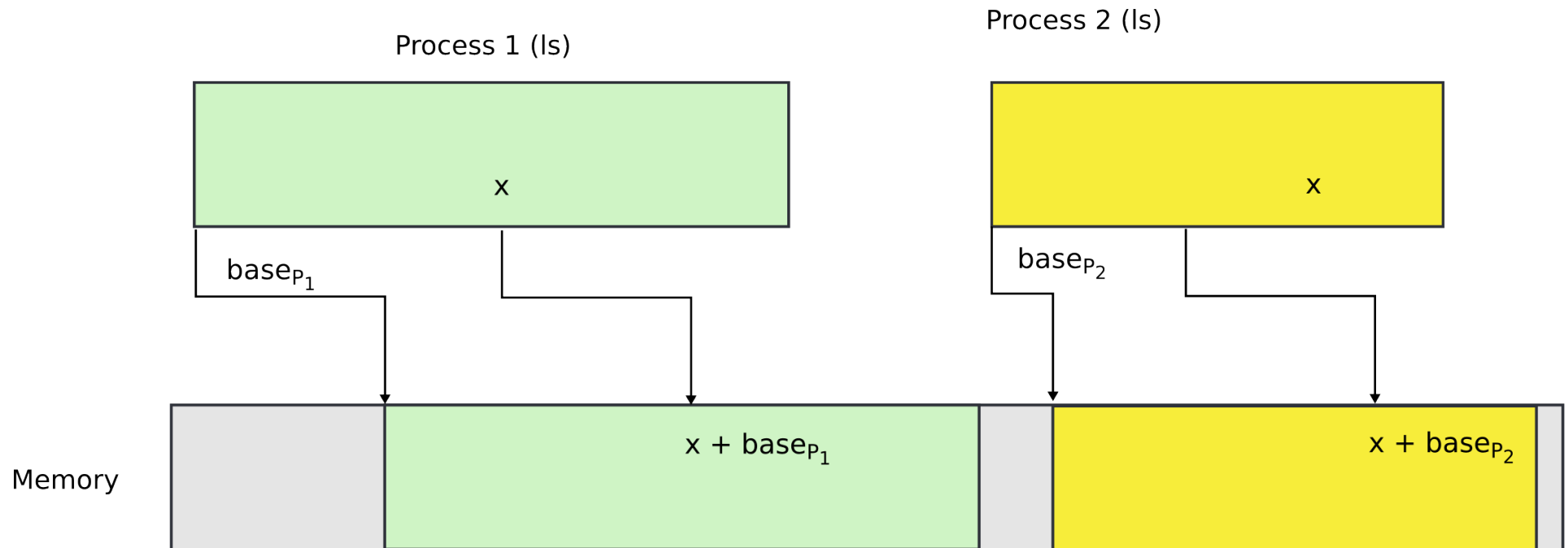
- Data
 - DS, ES, FS, GS
 - 4 possible data segments can be used at the same time

Stack segment

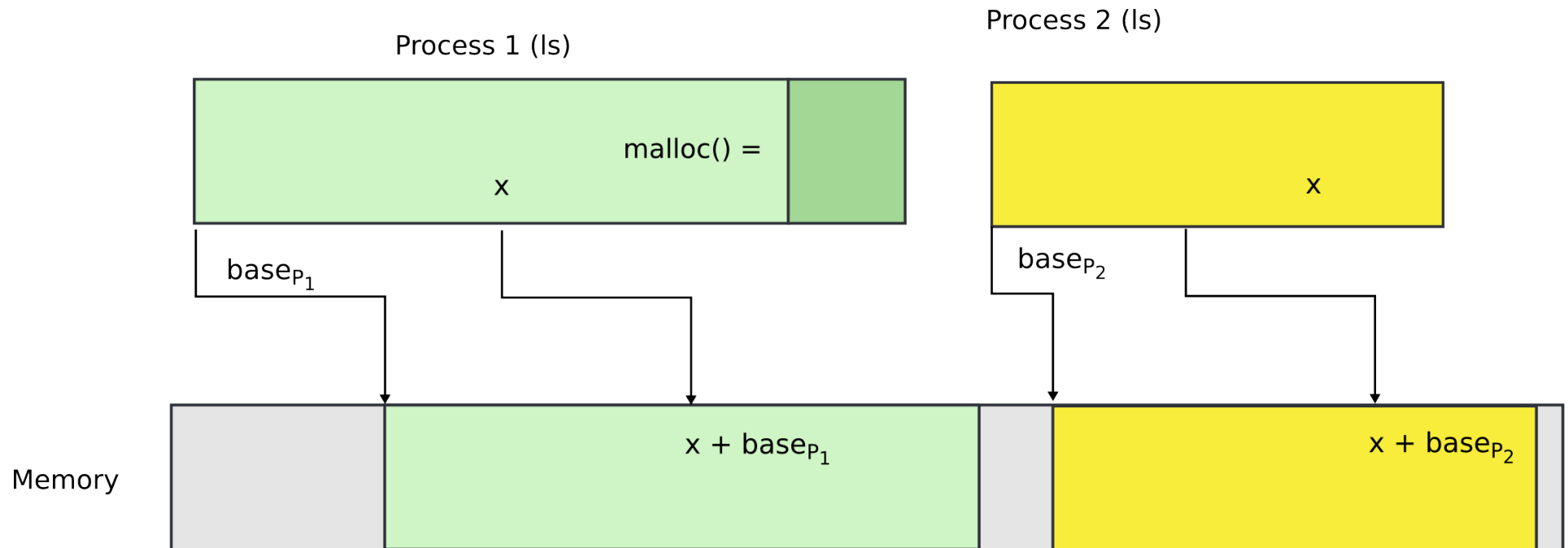
- Stack
 - SS
- Can be loaded explicitly
 - OS can set up multiple stacks
 - Of course, only one is accessible at a time

Segmentation works for isolation, i.e., it does provide programs with illusion of private memory

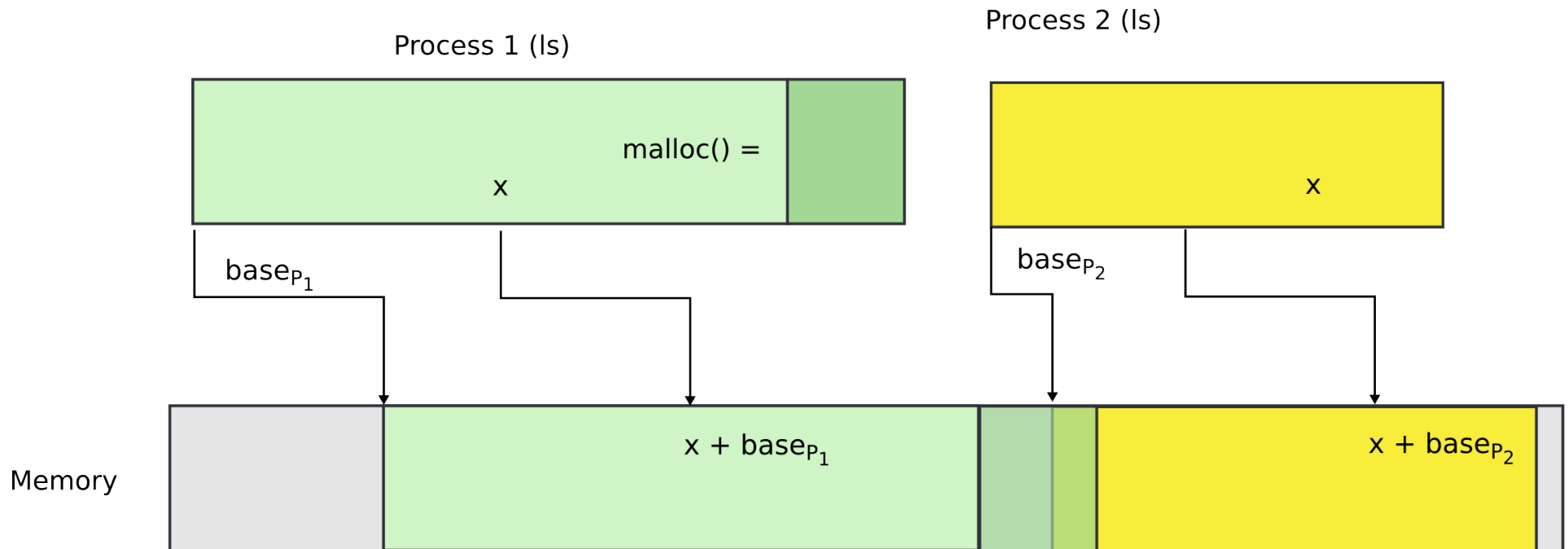
Segmentation is ok... but



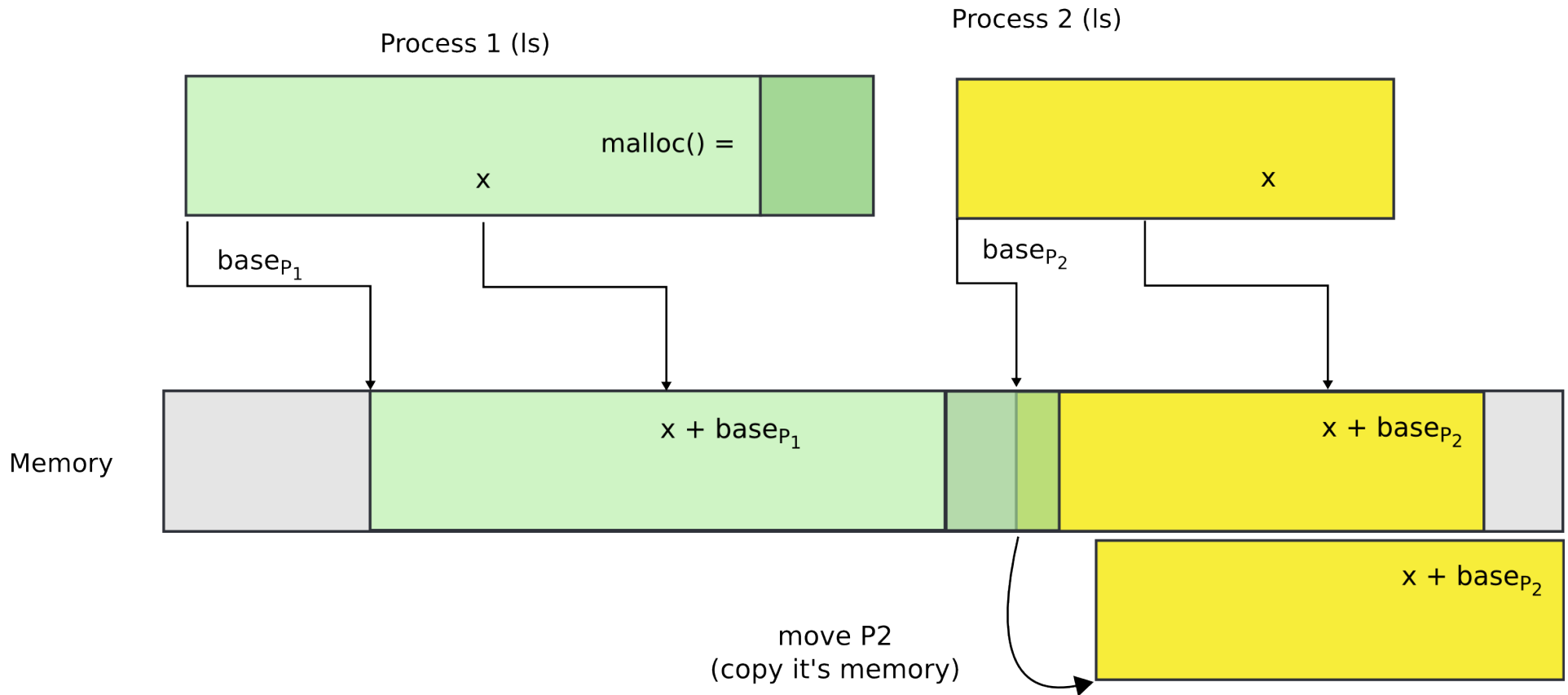
What if process needs more memory?



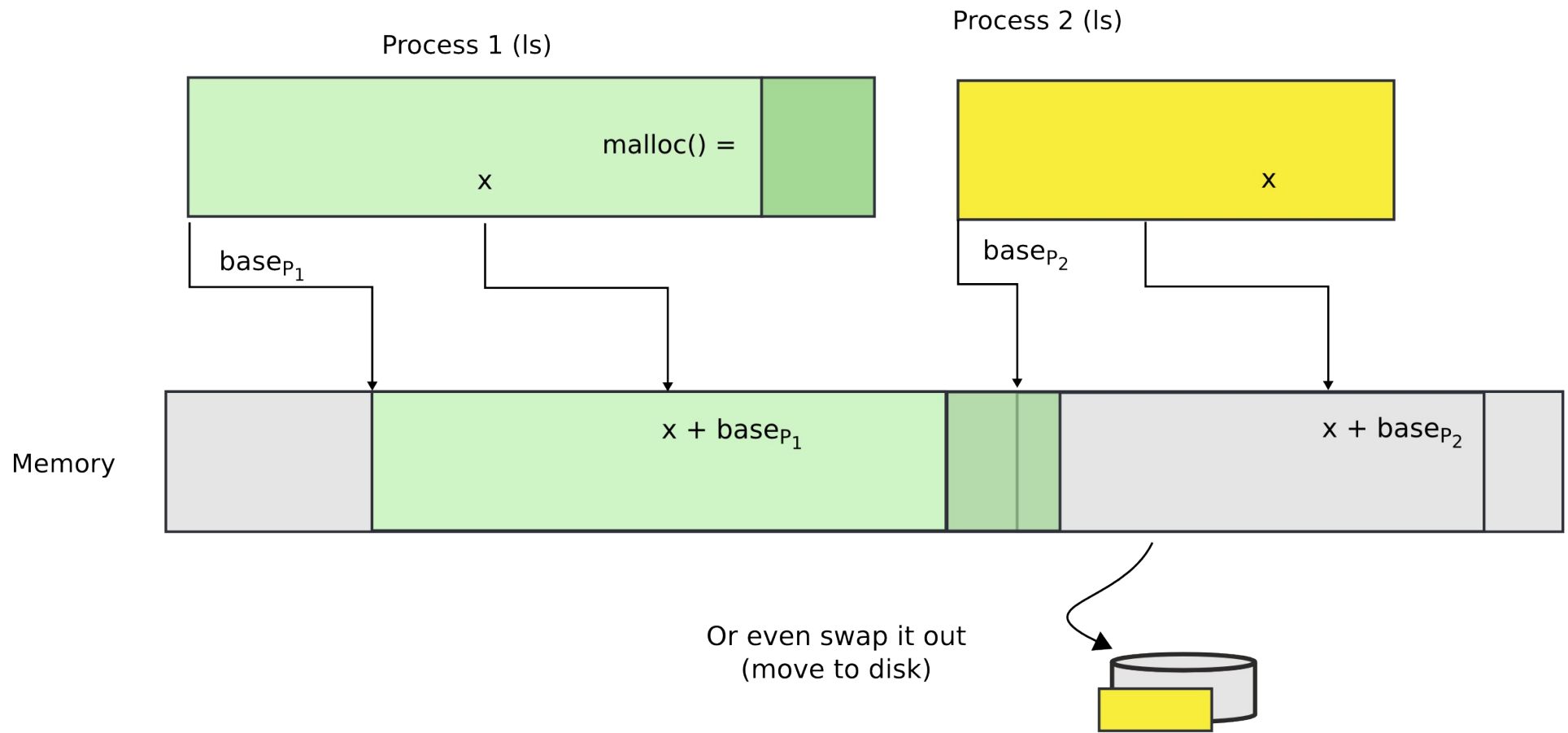
What if process needs more memory?



You can relocate P2



Or even swap it out to disk



Problems with segments

- But it's inefficient
 - Relocating or swapping the entire process takes time
- Memory gets fragmented
 - There might be no space (gap) for the swapped out process to come in
 - Will have to swap out other processes

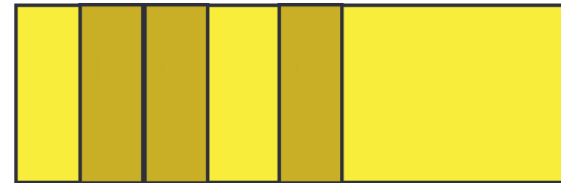
Paging

Pages

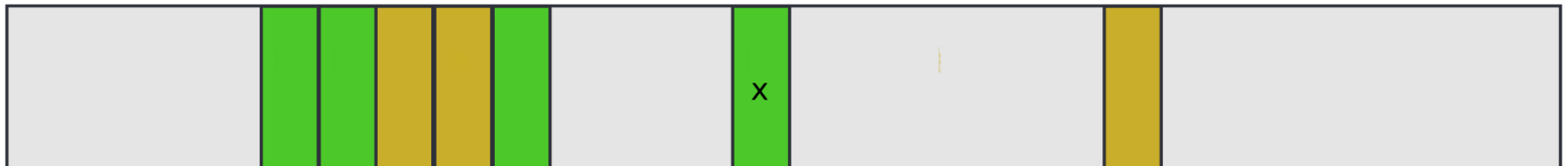
Process 1 (Is)



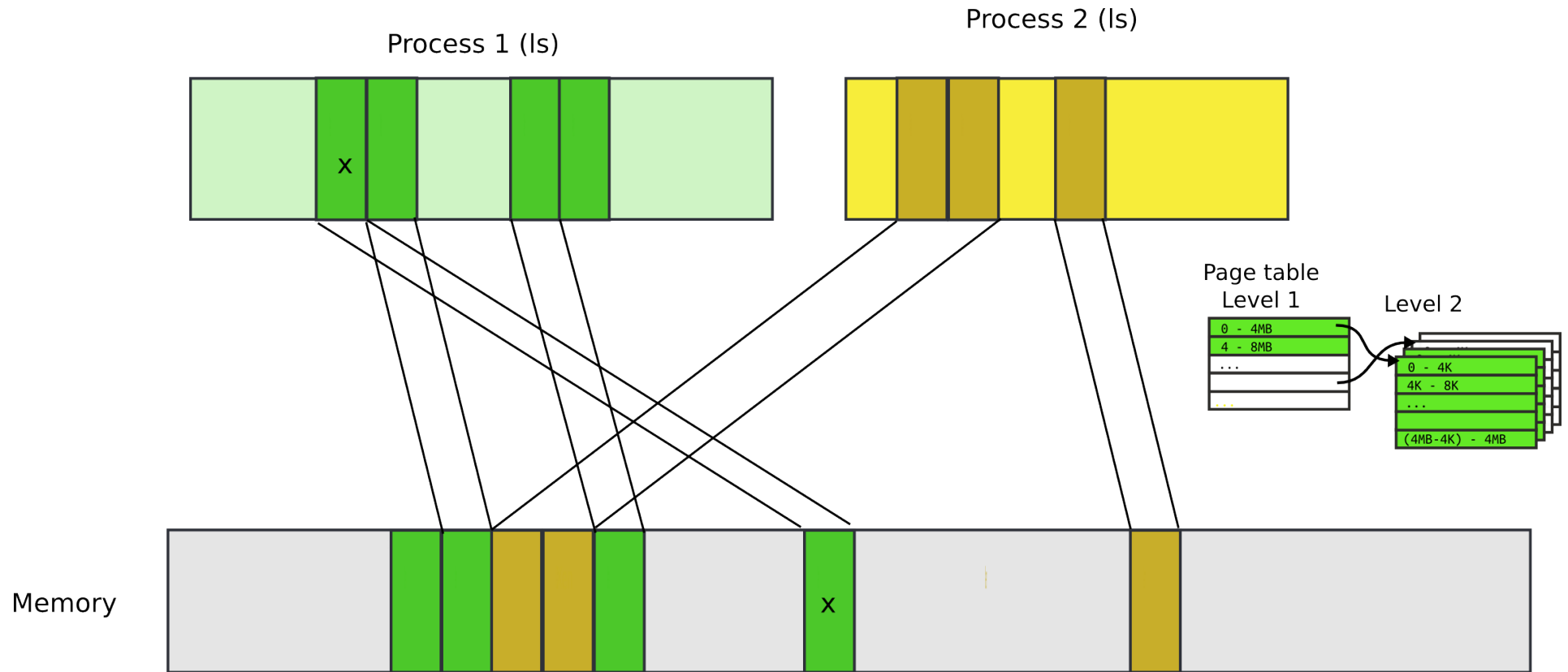
Process 2 (Is)



Memory



Pages



Paging idea

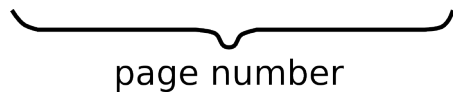
- Break up memory into 4096-byte chunks called pages
 - Modern hardware supports 2MB, 4MB, and 1GB pages
- Independently control mapping for each page of linear address space
- Compare with segmentation (single base + limit)
 - many more degrees of freedom

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

EAX = 0

EBX = 20 983 809

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)

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

Physical
Memory



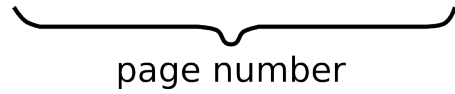
mov (%EBX), EAX # mov value from the location pointed by EBX into EAX

EAX = 0

EBX = 20 983 809

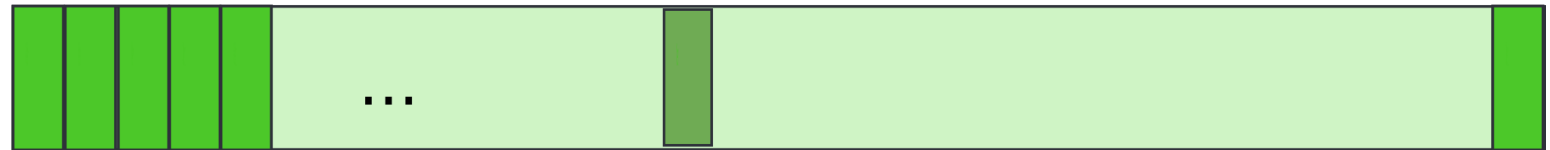
20 983 809 =

00 0000 0101	00 0000 0011	0000 0000 0001
--------------	--------------	----------------


page number

1M (1,048,575)

Virtual Address
Space (or Memory)
of the Process



CR3 = 0

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

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

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



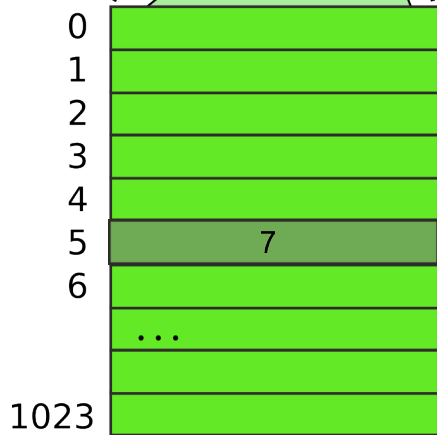
CR3 = 0

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

Physical
Memory



32 bits (4 bytes)



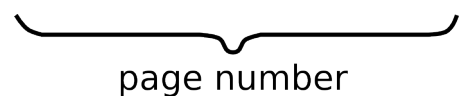
Level 1
(Page Table
Directory)

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

EAX = 0

EBX = 20 983 809

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



1M (1,048,575)



CR3 = 0

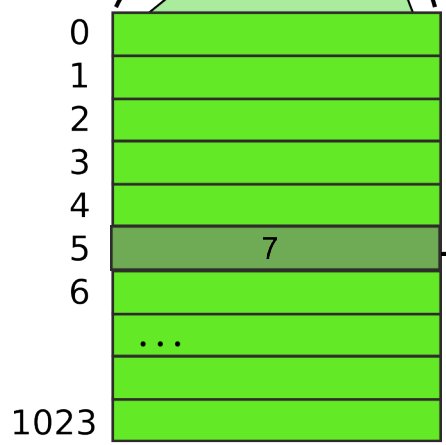
0 1 2

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

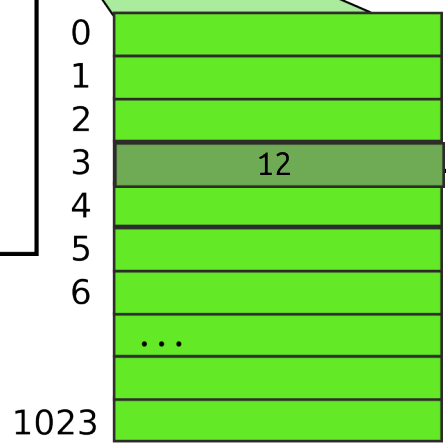
page number = 5123
or (0b1 0100 0000 0011)



32 bits (4 bytes)



Level 1
(Page Table
Directory)



Level 2
(Page Table)

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

EAX = 0

EBX = 20 983 809

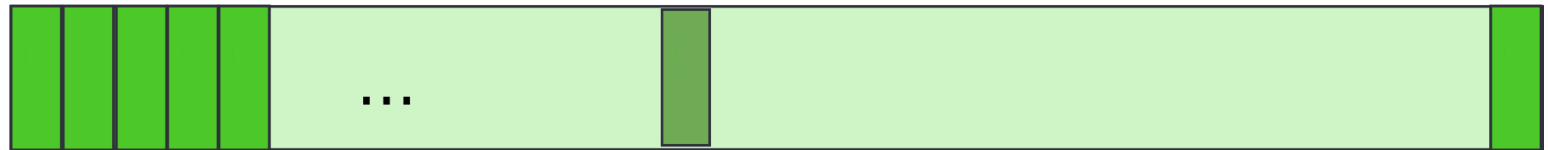
20 983 809 =

00 0000 0101	00 0000 0011	0000 0000 0001
--------------	--------------	----------------

page number

1M (1,048,575)

Virtual Address Space (or Memory) of the Process



CR3 = 0

0 1 2

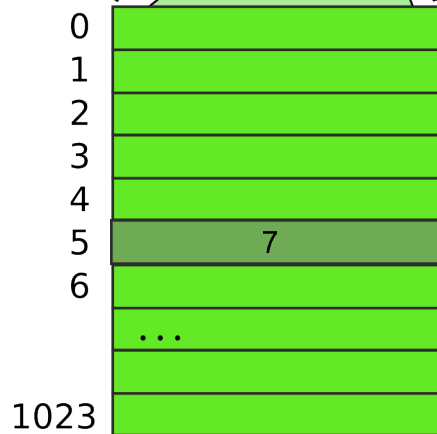
0 1 2 3 4 5 6 7 8 9 10 11 12

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

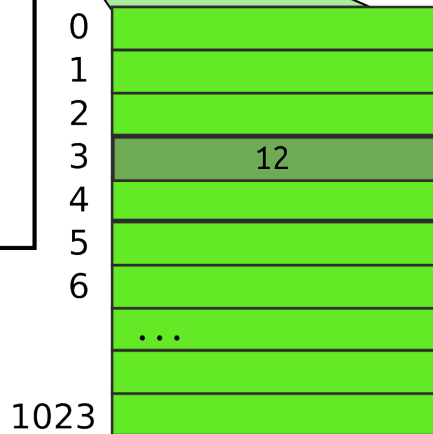
Physical Memory



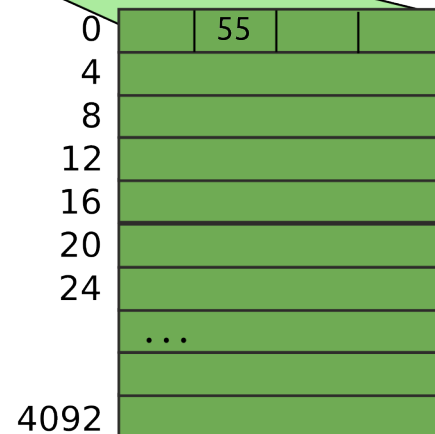
32 bits (4 bytes)



Level 1
(Page Table
Directory)



Level 2
(Page Table)

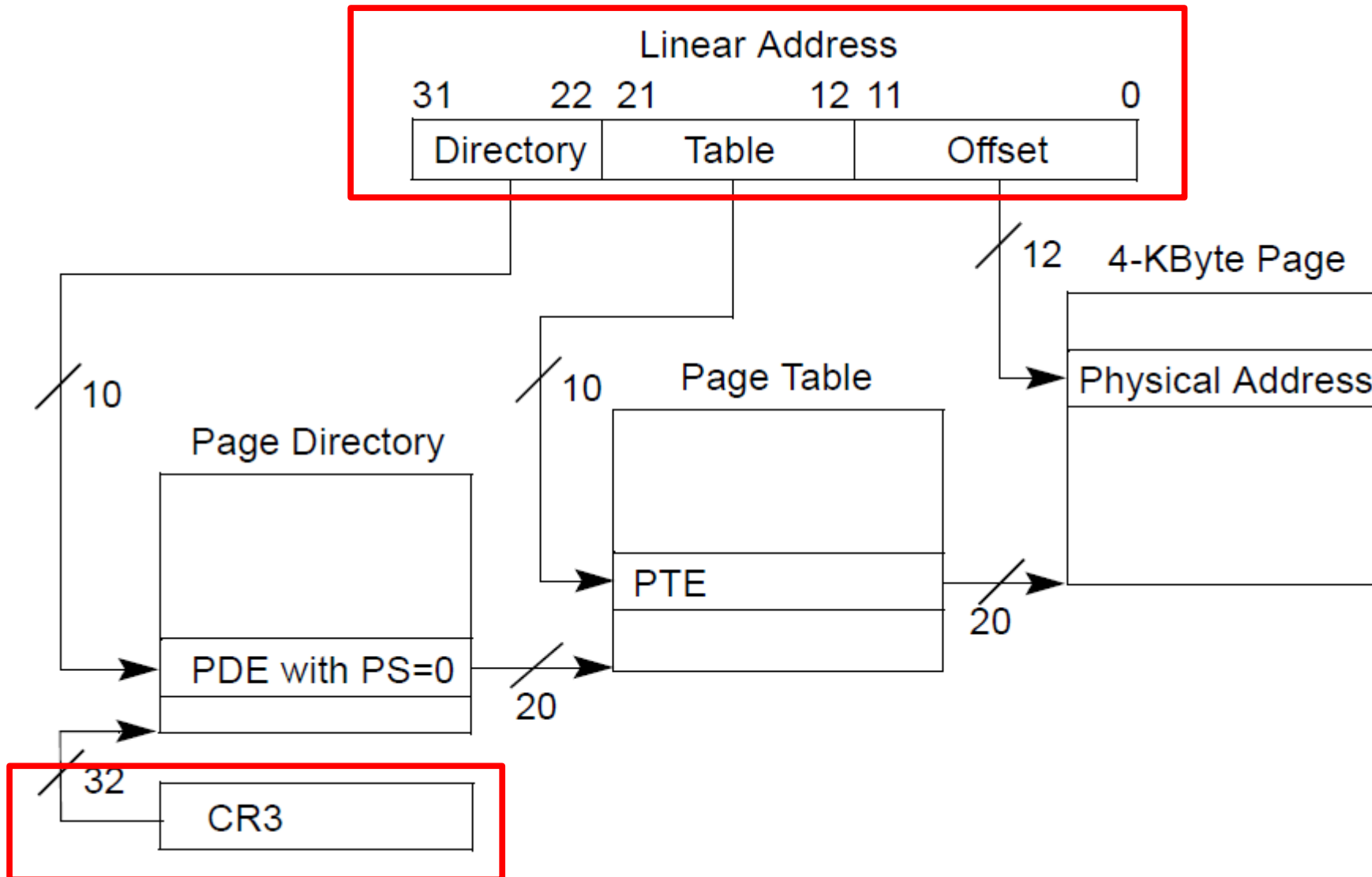


Page

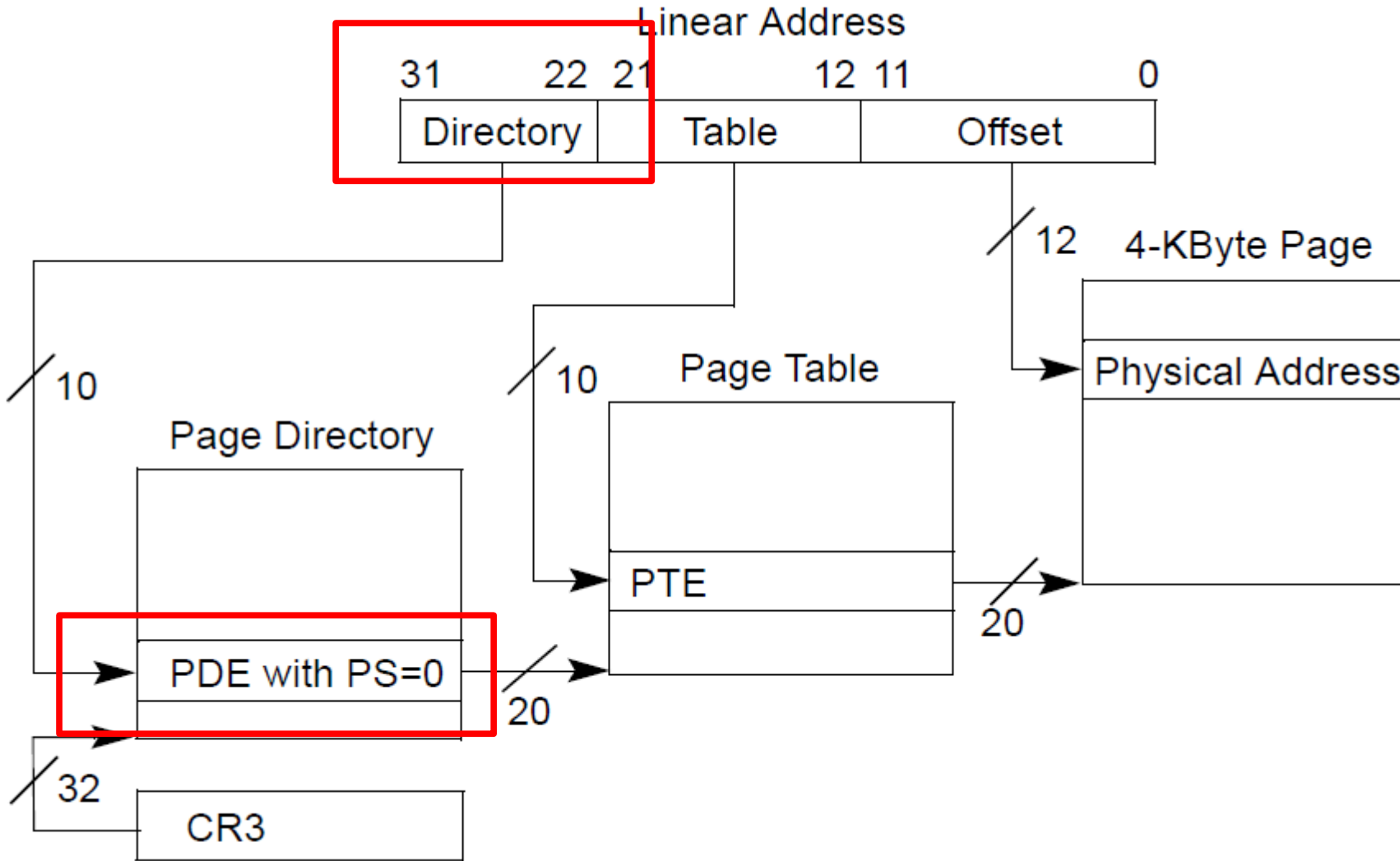
- Result:

- $EAX = 55$

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		<u>0</u>	I g n	A	P C D	P W T	U / S	R / W	<u>1</u>	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>	I g n	A	P C D	P W T	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	I g n	A	P C D	P W T	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
 - Pages start at 4KB (page aligned boundary)

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>	I g n	A	P C D	P W T	U / S	R / W	<u>1</u>	PDE: page table									

- 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>	I g n	A	P C D	P W T	U / S	R / W	<u>1</u>	PDE: page table							

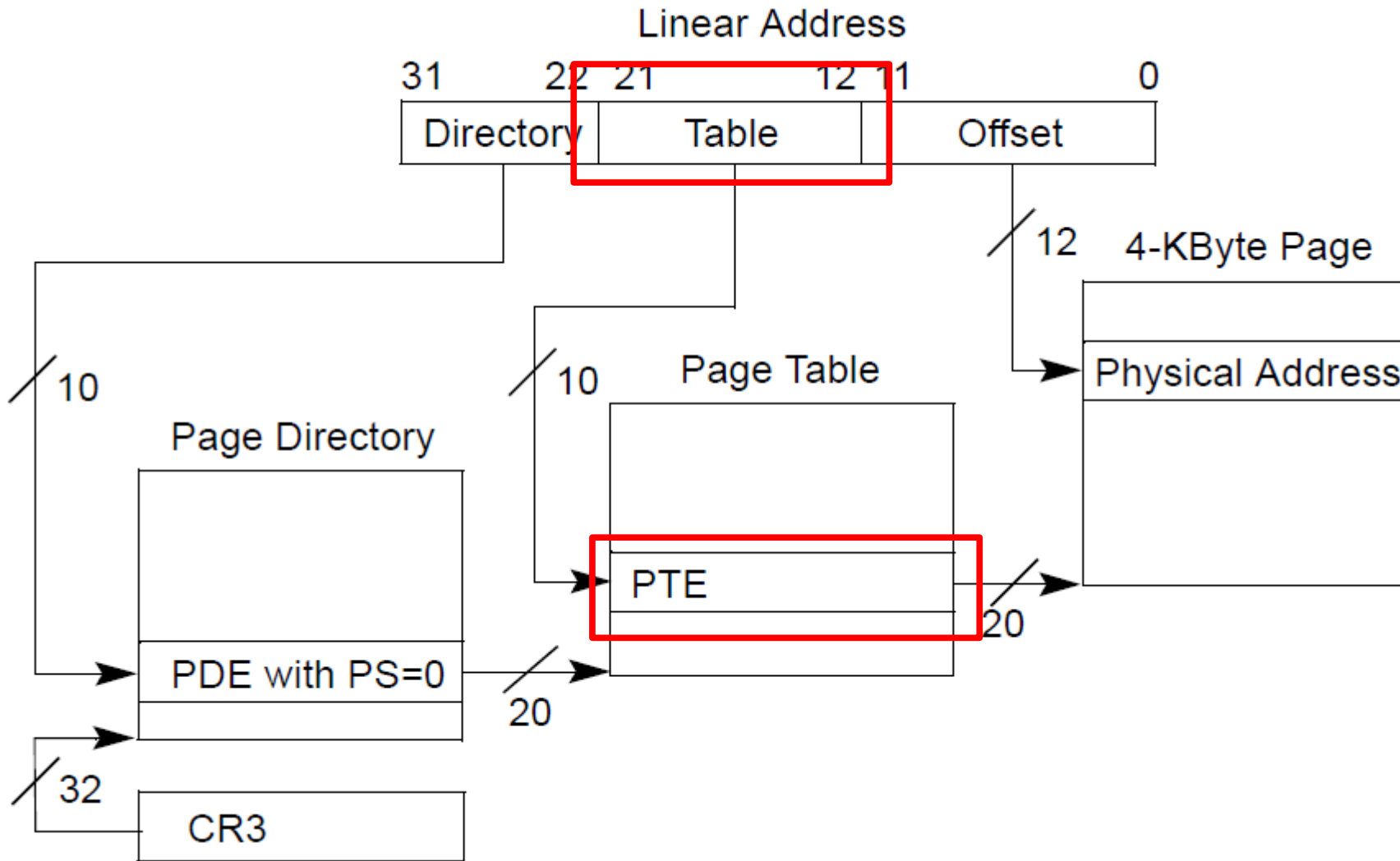
- Bit #1: R/W – writes allowed?
 - But allowed where?
 - One page directory entry controls 1024 Level 2 page tables
 - Each Level 2 maps 4KB page
 - So it's a region of 4KB x 1024 = 4MB

Page directory entry (PDE)

31	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>	I g n	A	P C D	P W T	U / S	R / W	<u>1</u>	PDE: page table										

- Bit #2: U/S – user/supervisor
 - If 0 – user-mode access is not allowed
 - Allows protecting kernel memory from user-level applications

Page translation

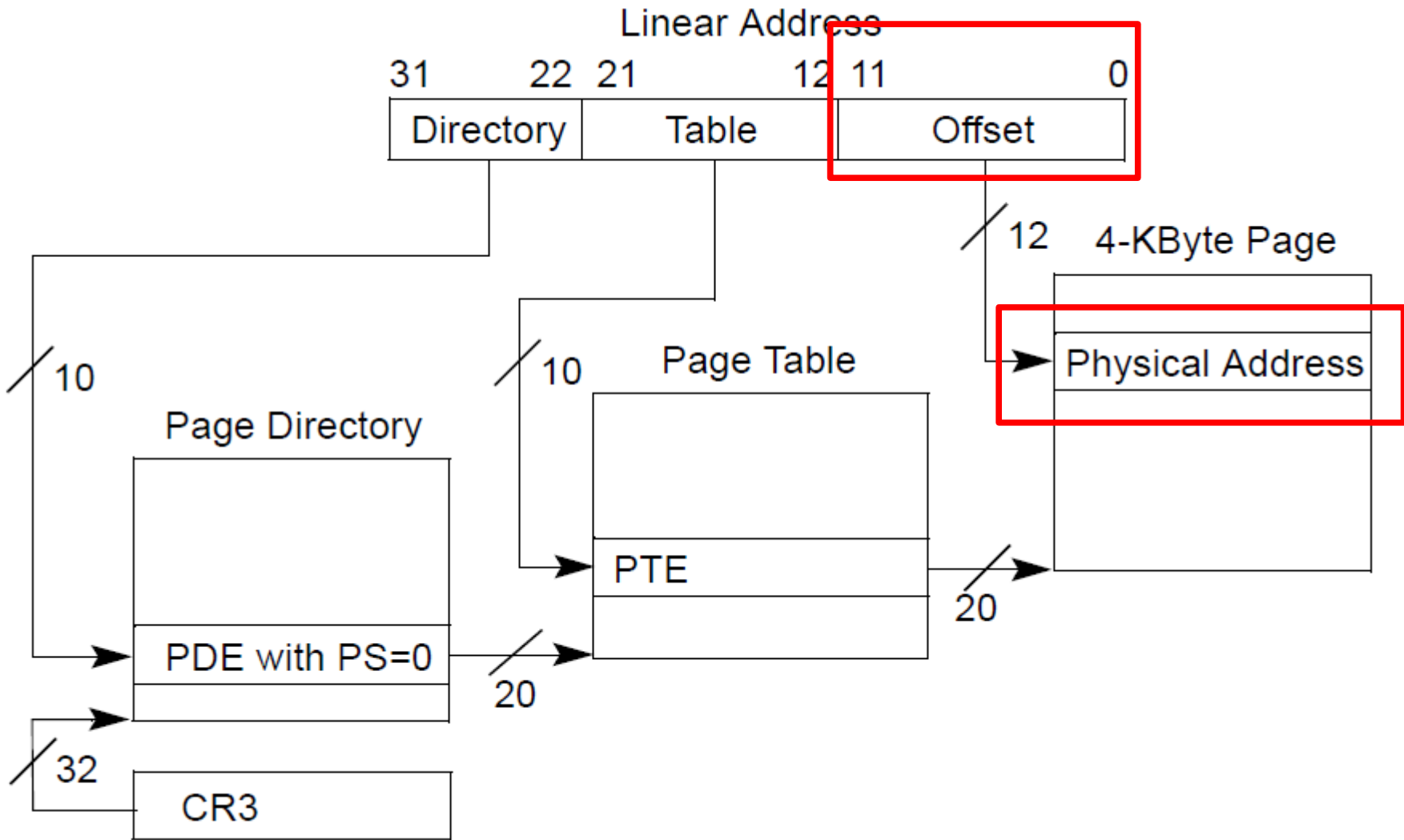


Page table entry (PTE)

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	P A T	D	A	P C D	P W T	U / S	R / W	<u>1</u>	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?

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!

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 4KB 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

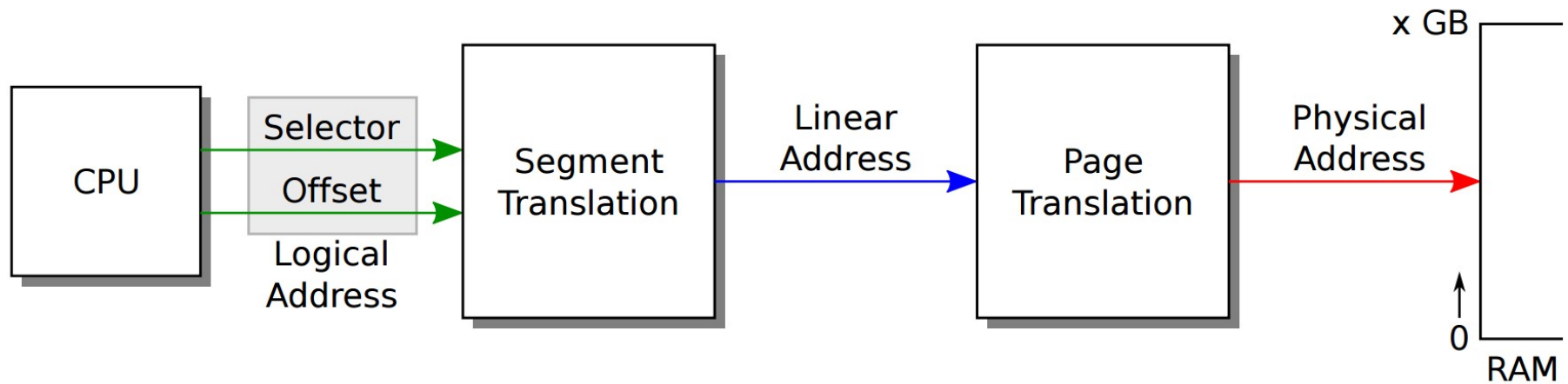
- Instead of arrays?

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!

Recap: complete address translation

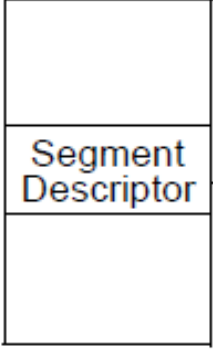


Logical Address
(or Far Pointer)

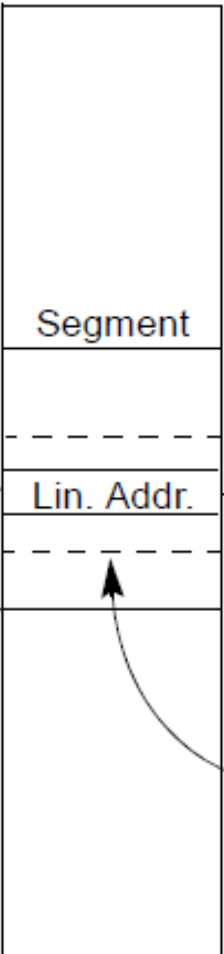
Segment Selector Offset

Linear Address Space

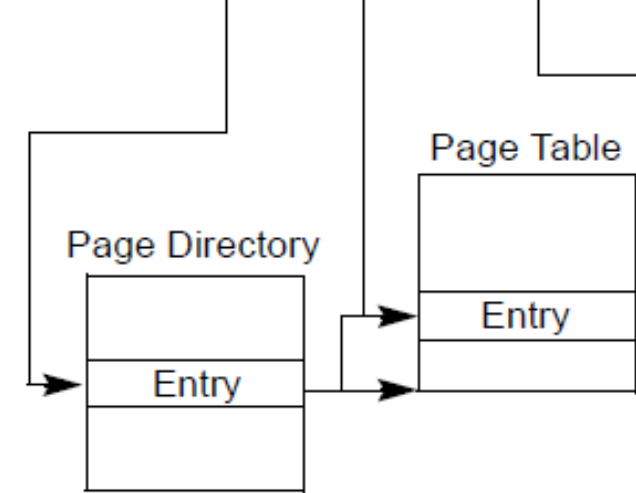
Global Descriptor Table (GDT)



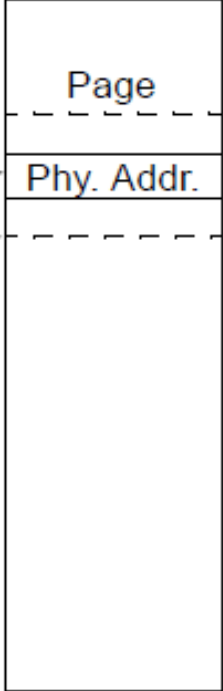
Segment Base Address



Linear Address Dir Table Offset

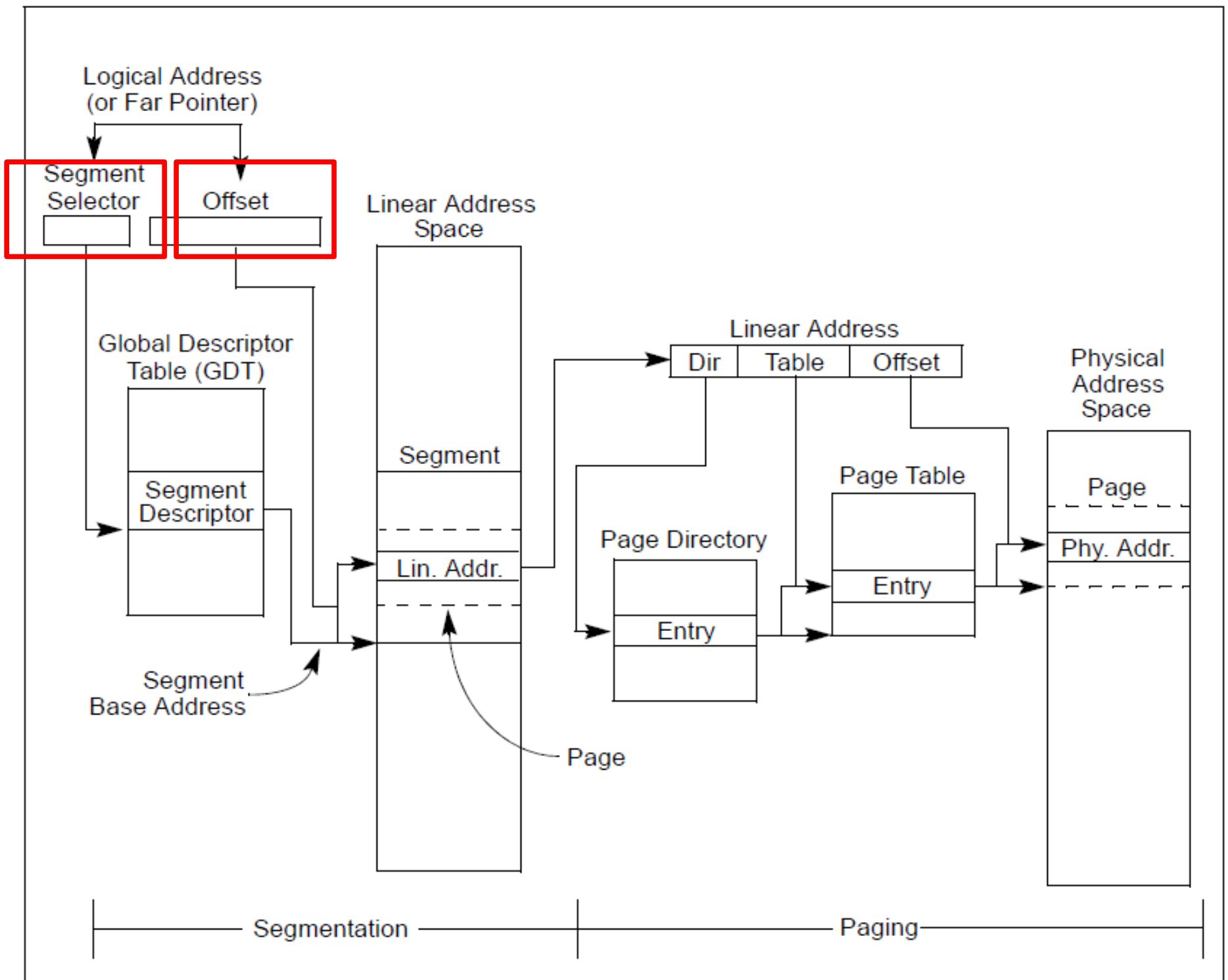


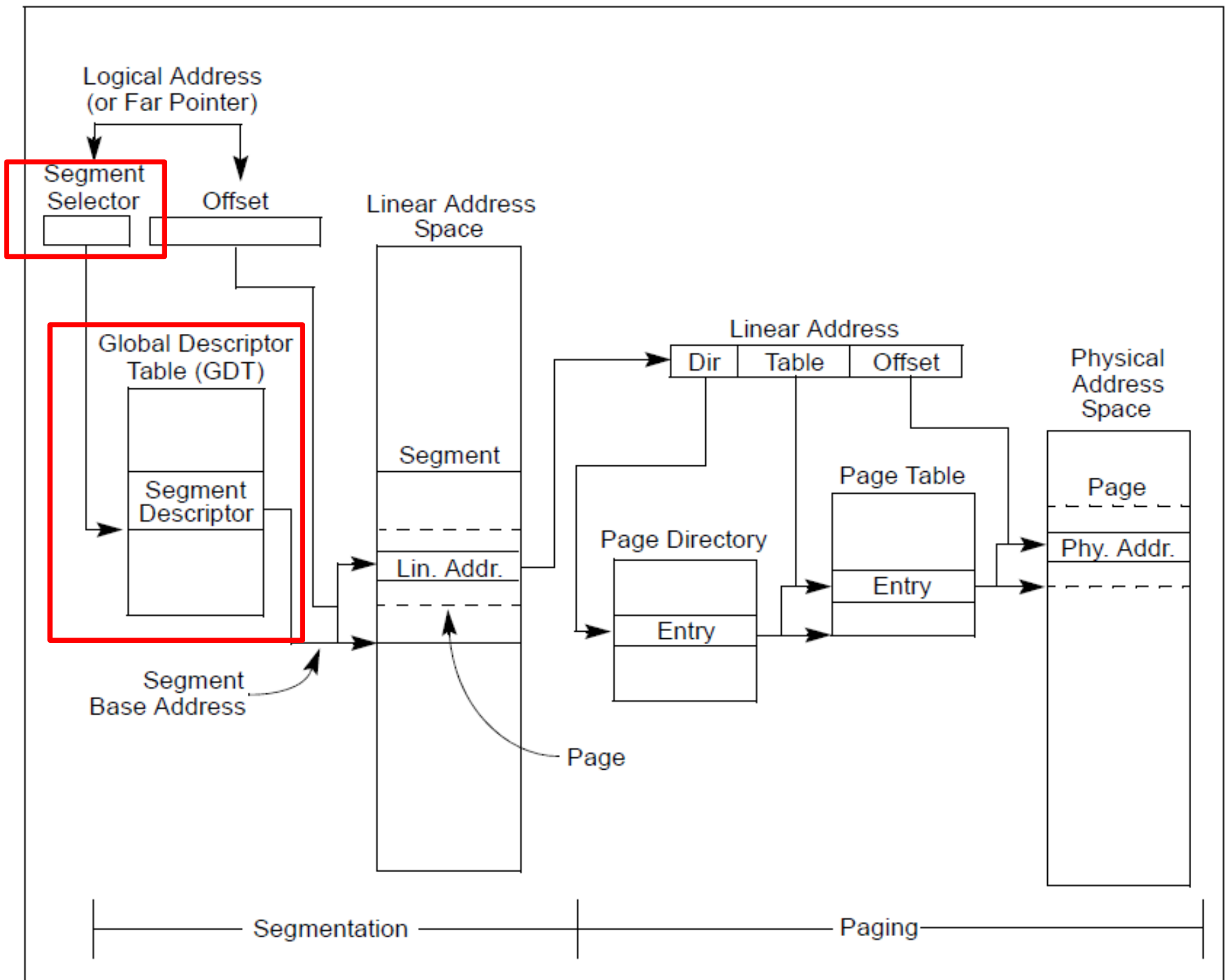
Physical Address Space

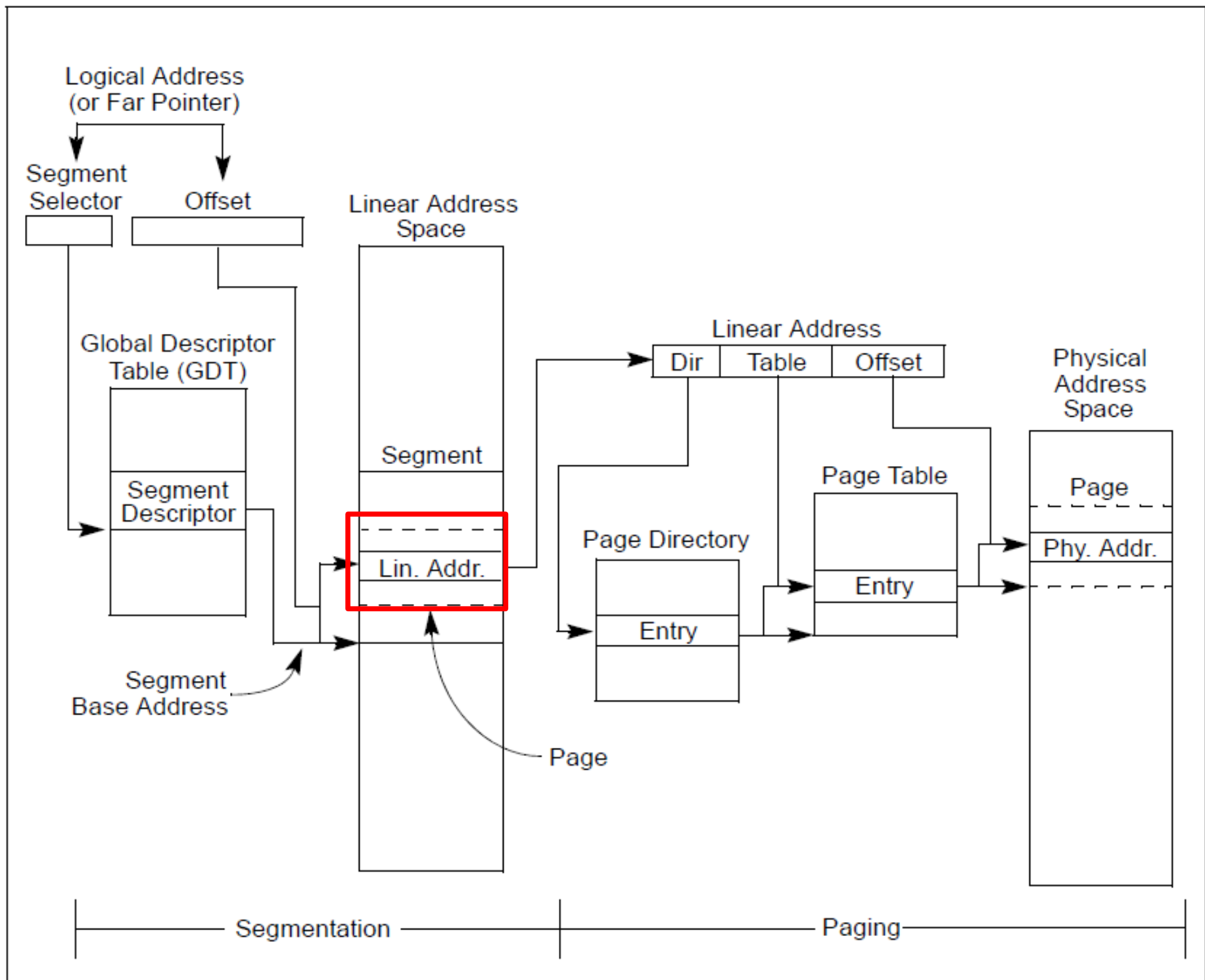


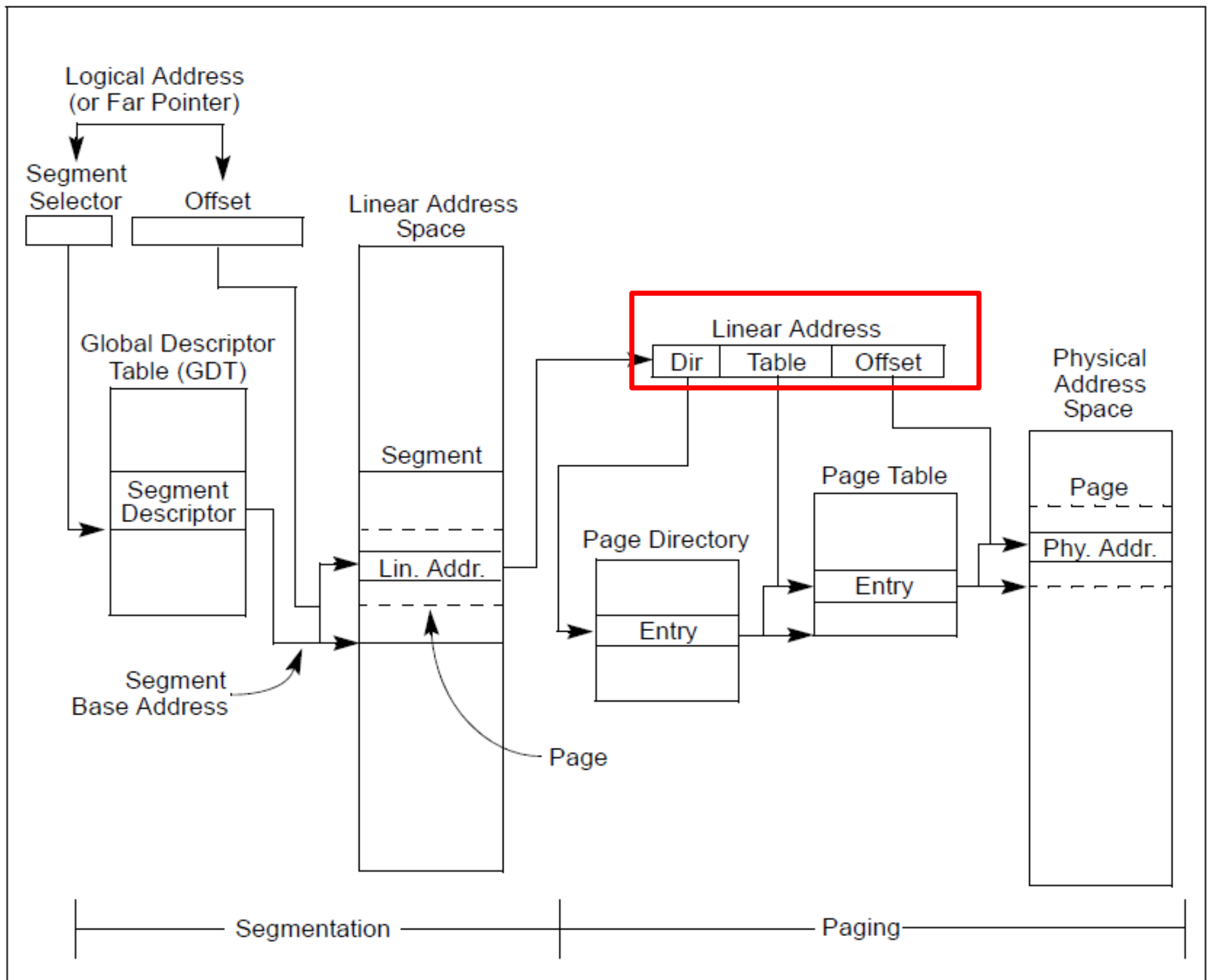
Segmentation

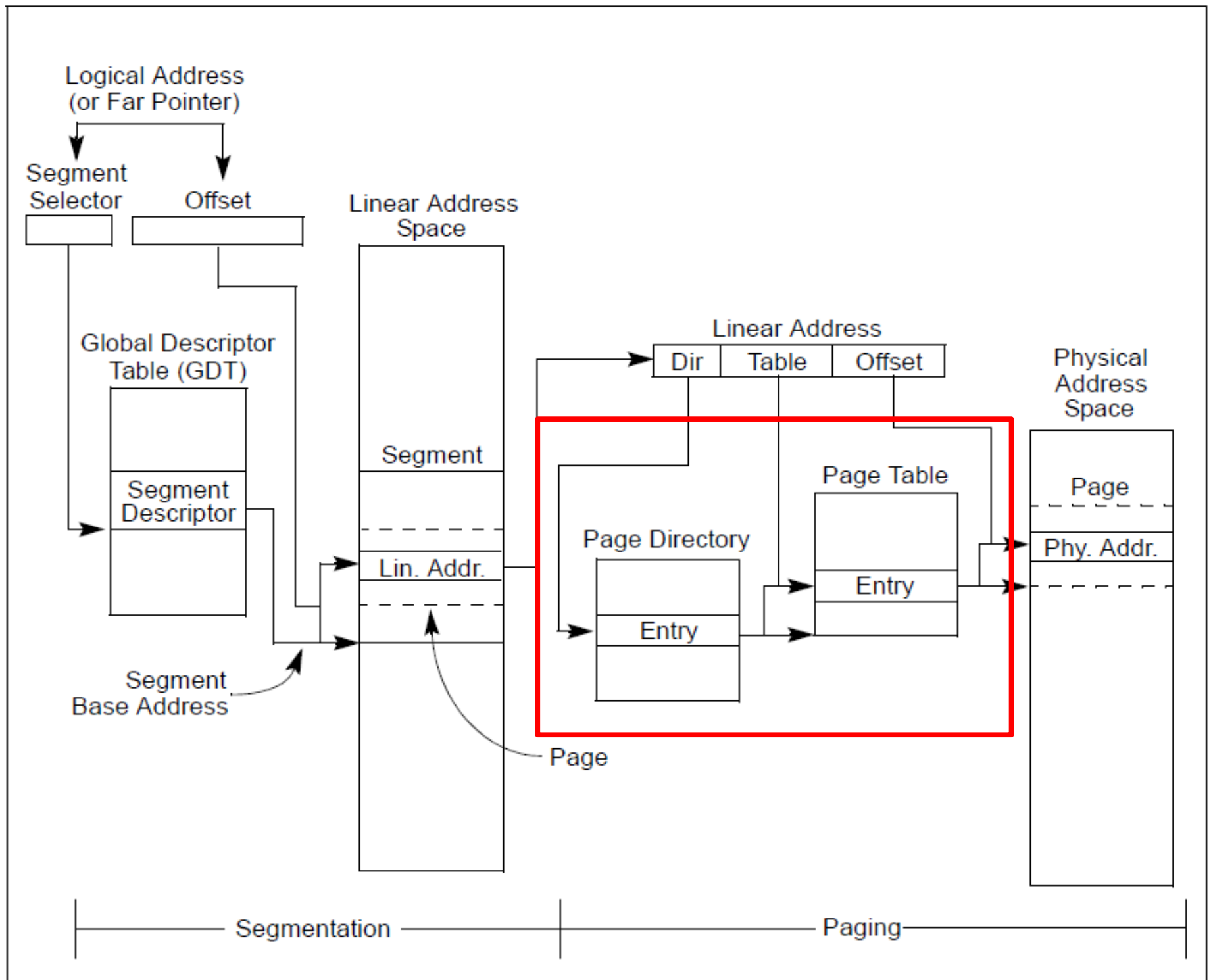
Paging

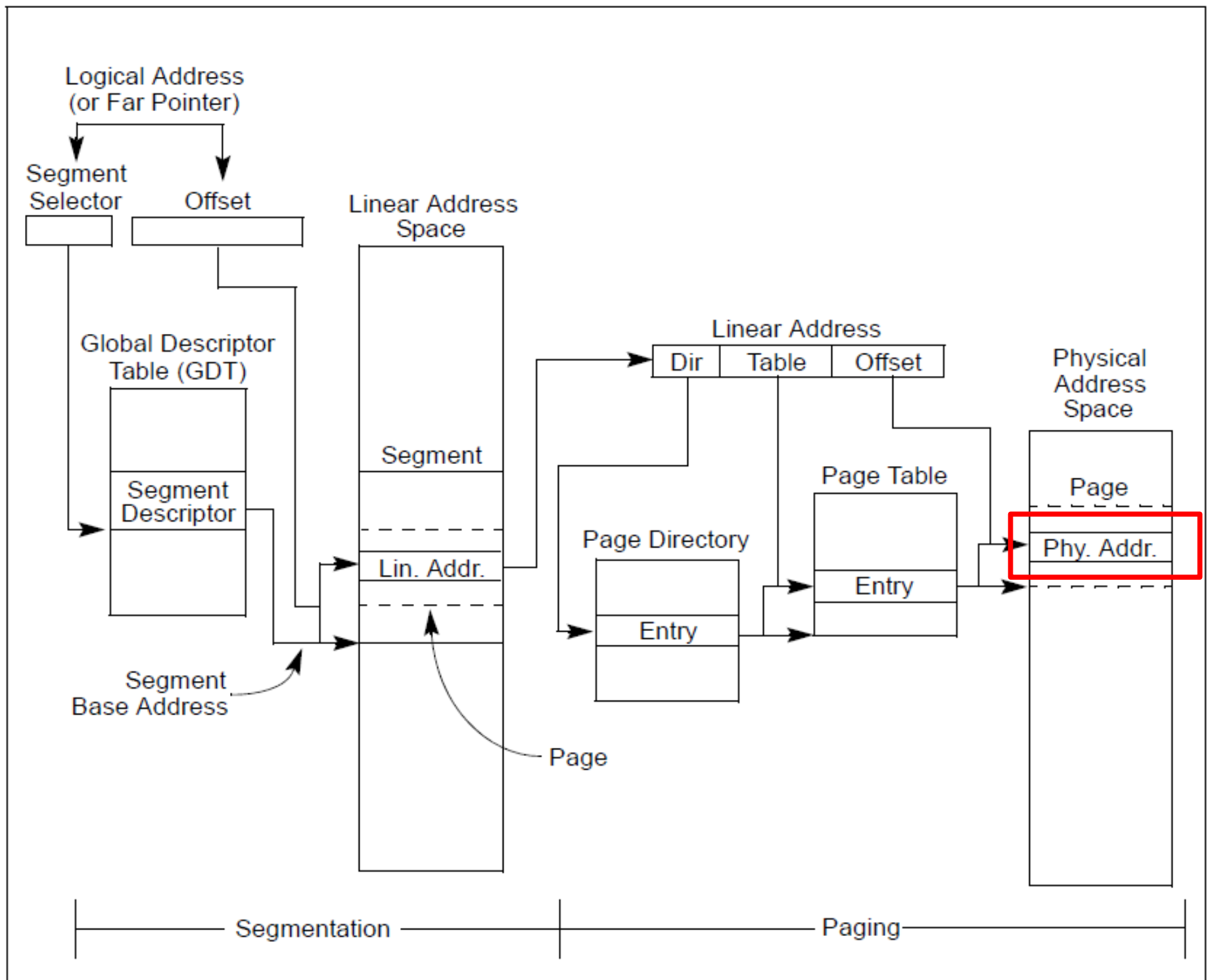


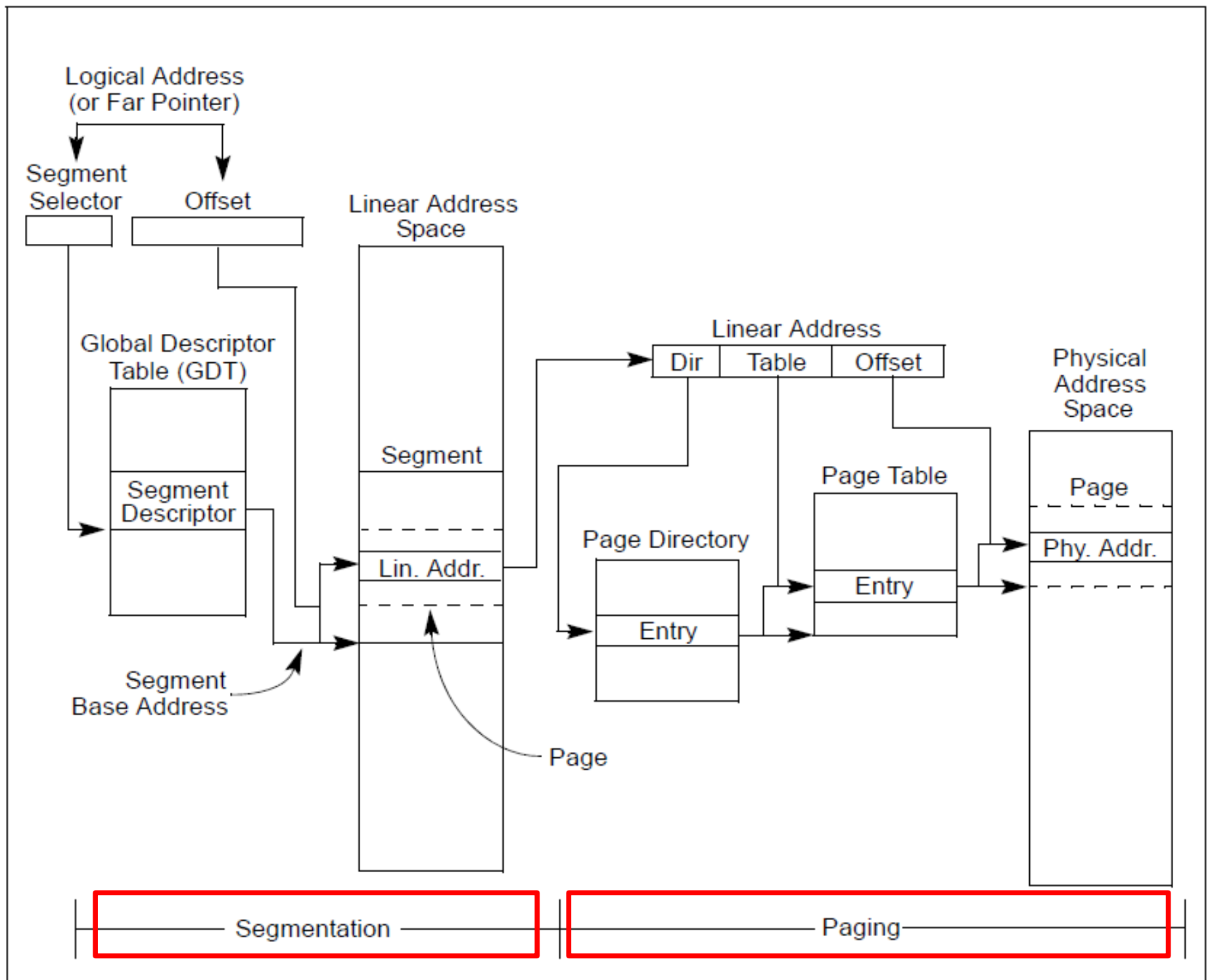






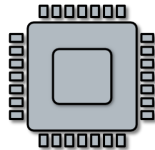
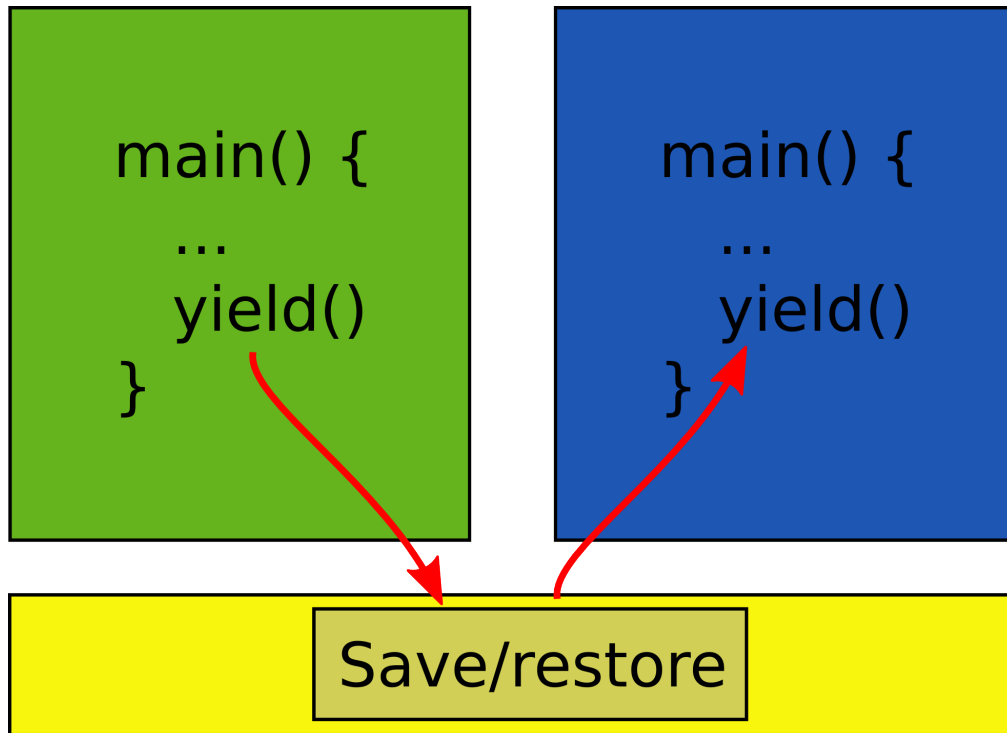




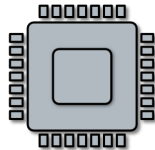
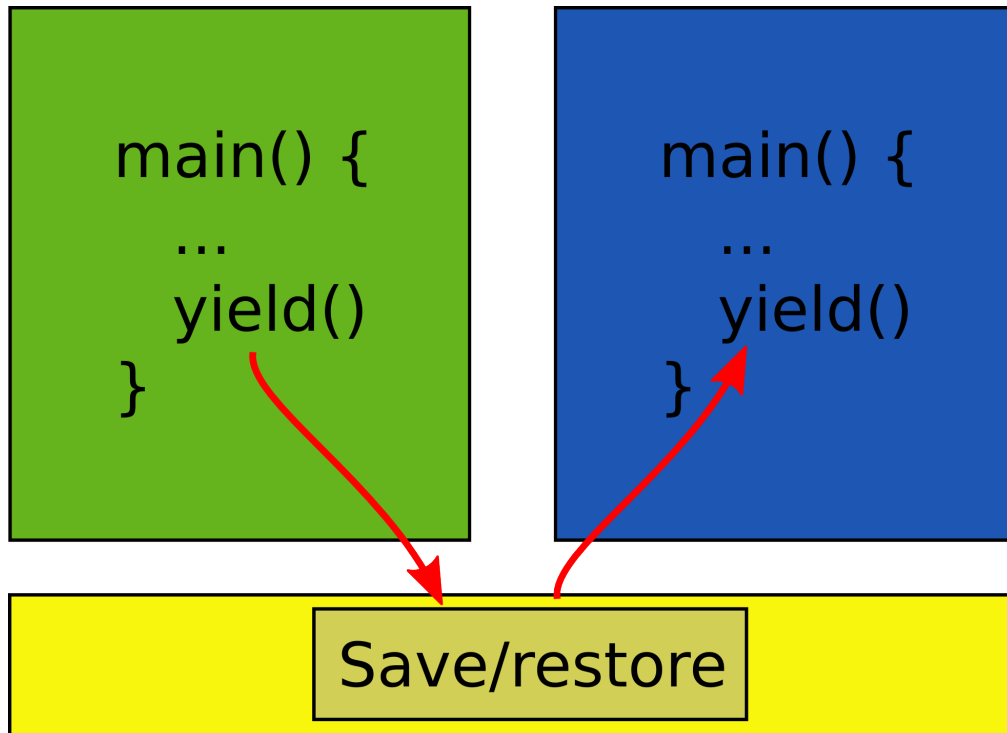


But what about isolation?

- Two programs, one memory?



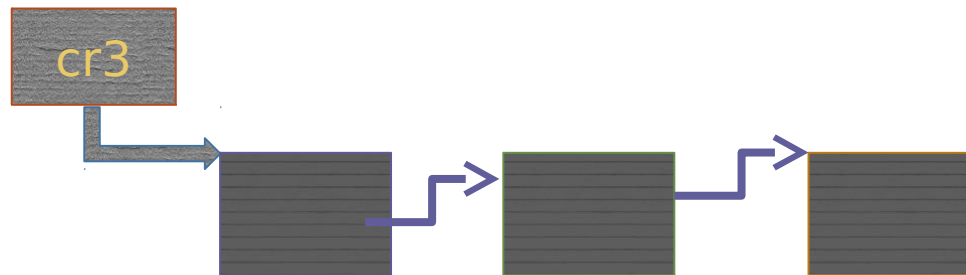
But what about isolation?



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

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



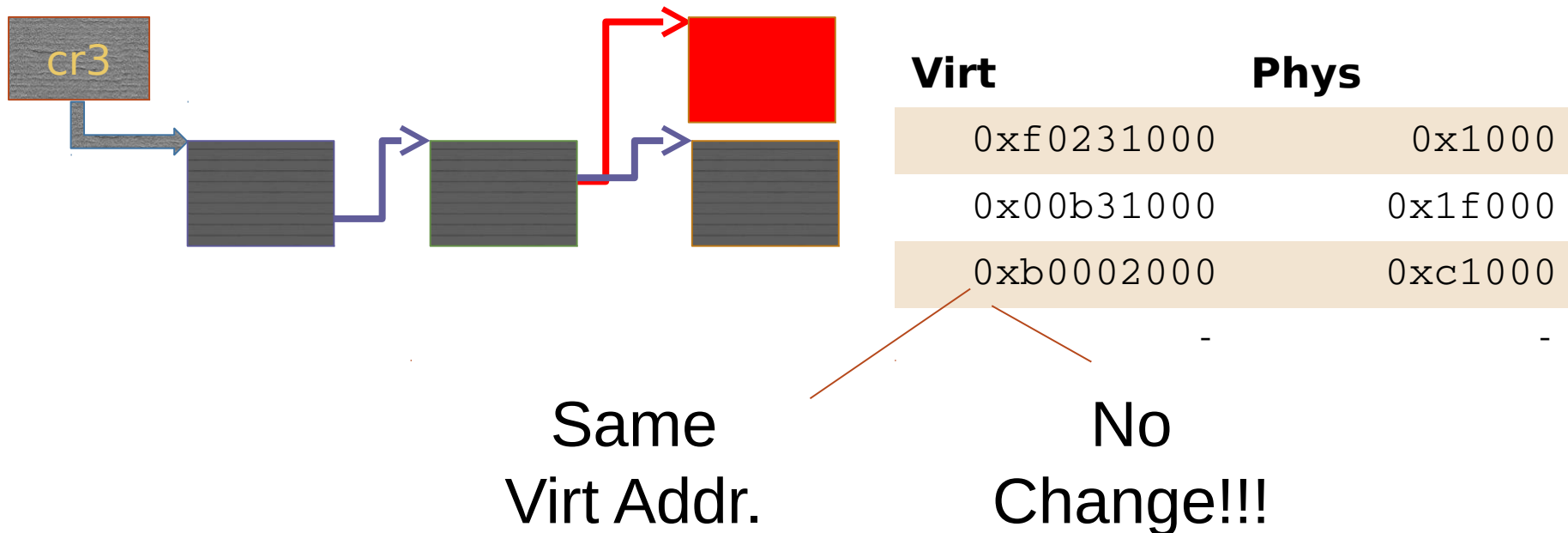
TLB

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

Virt	Phys
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)

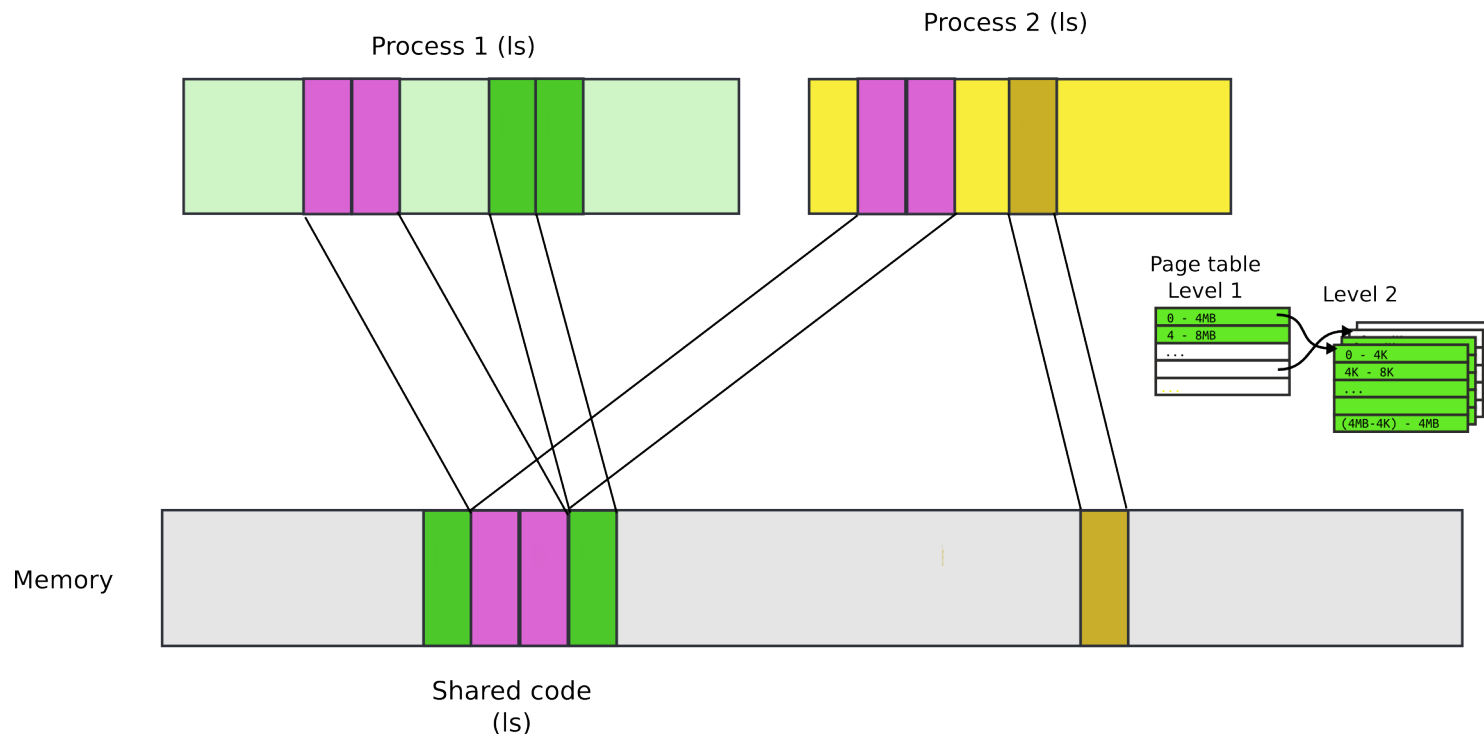
Virt	Phys	Tag
0xf0231000	0x1000	P1
0x00b31000	0x1f000	P2
0xb0002000	0xc1000	P1

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

When would you disable paging?

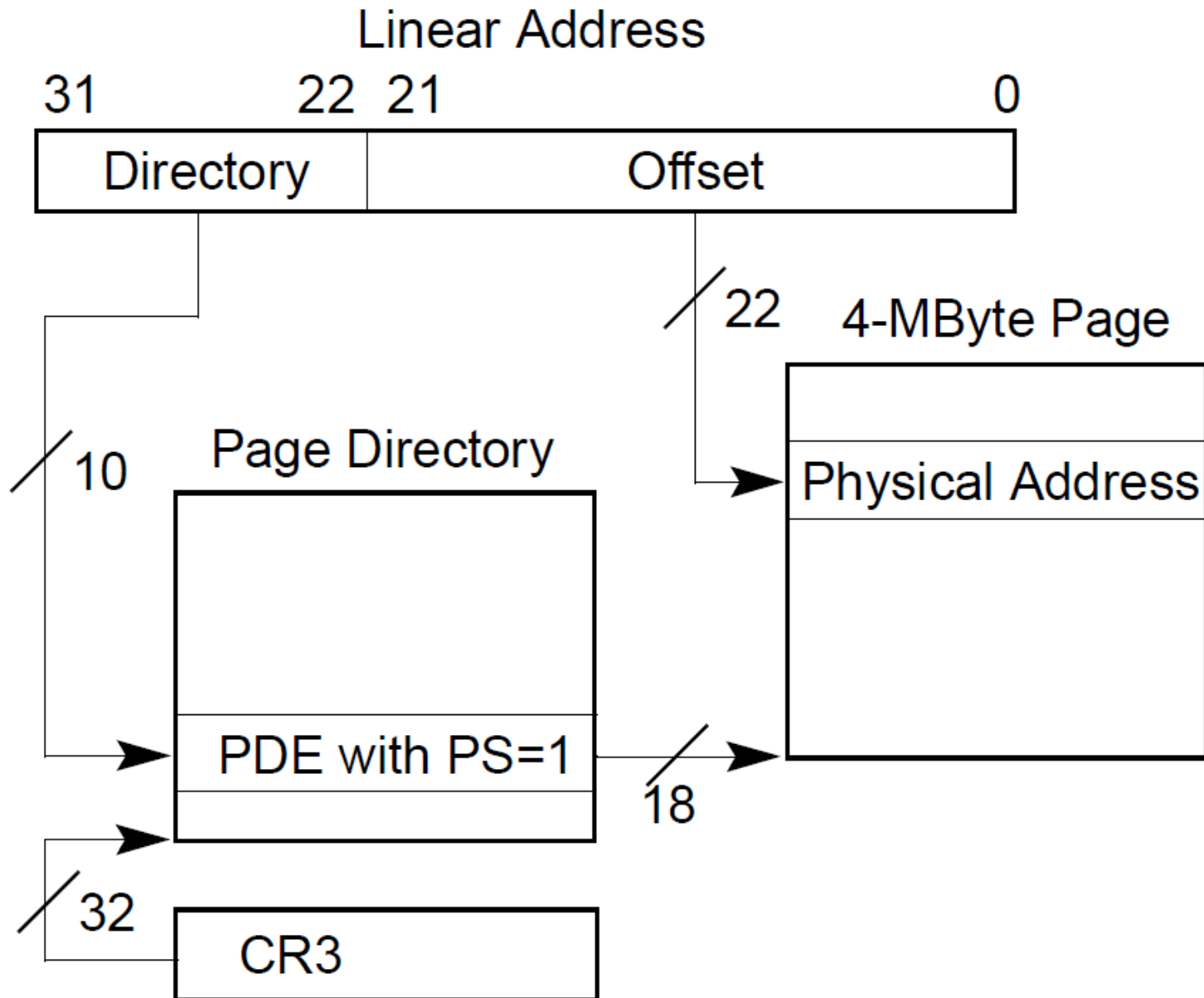
When would you disable paging?

- 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



Questions?

References

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