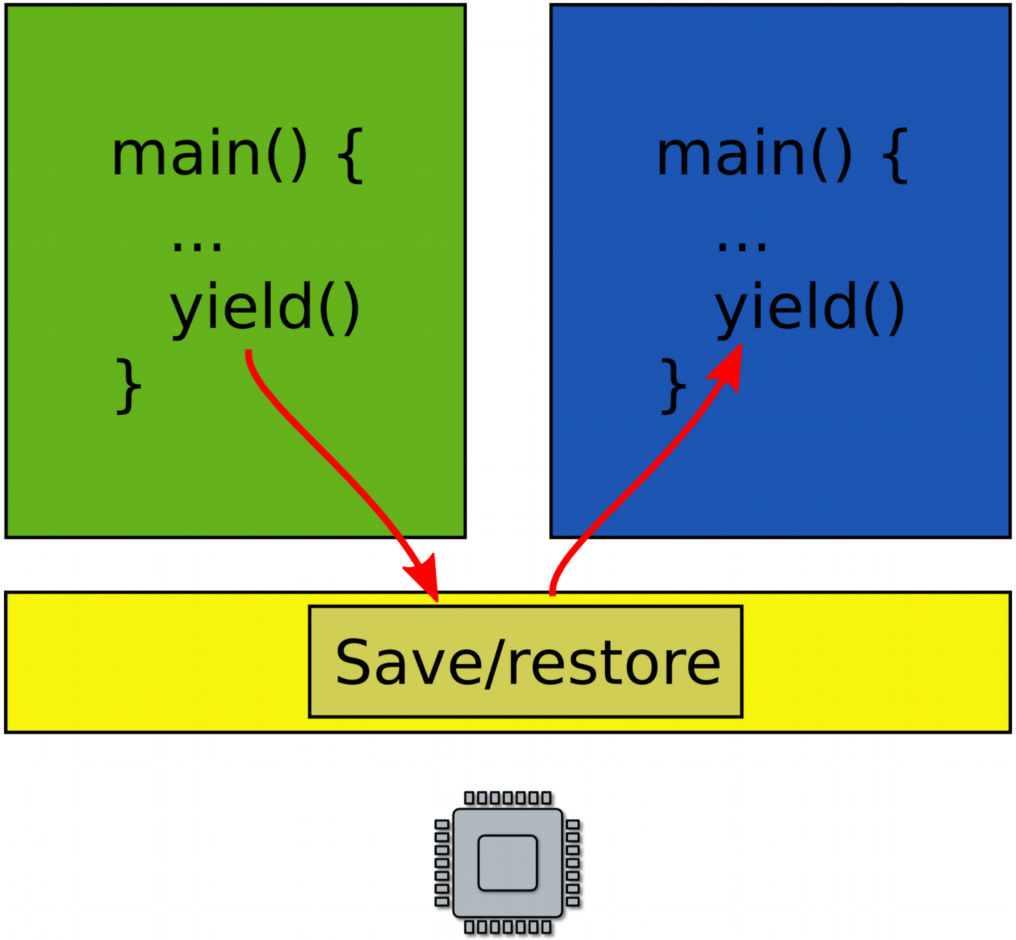
238P: Operating Systems

Lecture 5: Address translation

Anton Burtsev
October, 2018

Two programs one memory

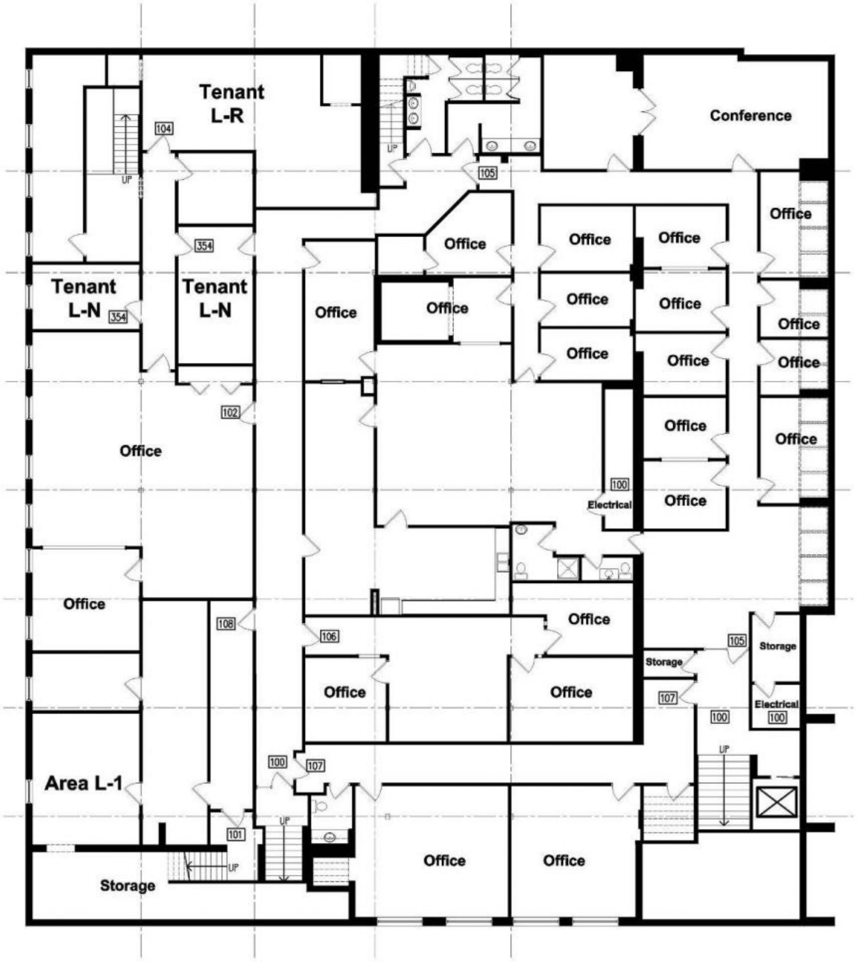


```
main() {
   ...
   yield()
}
```

```
main() {
   ...
   yield()
}
```

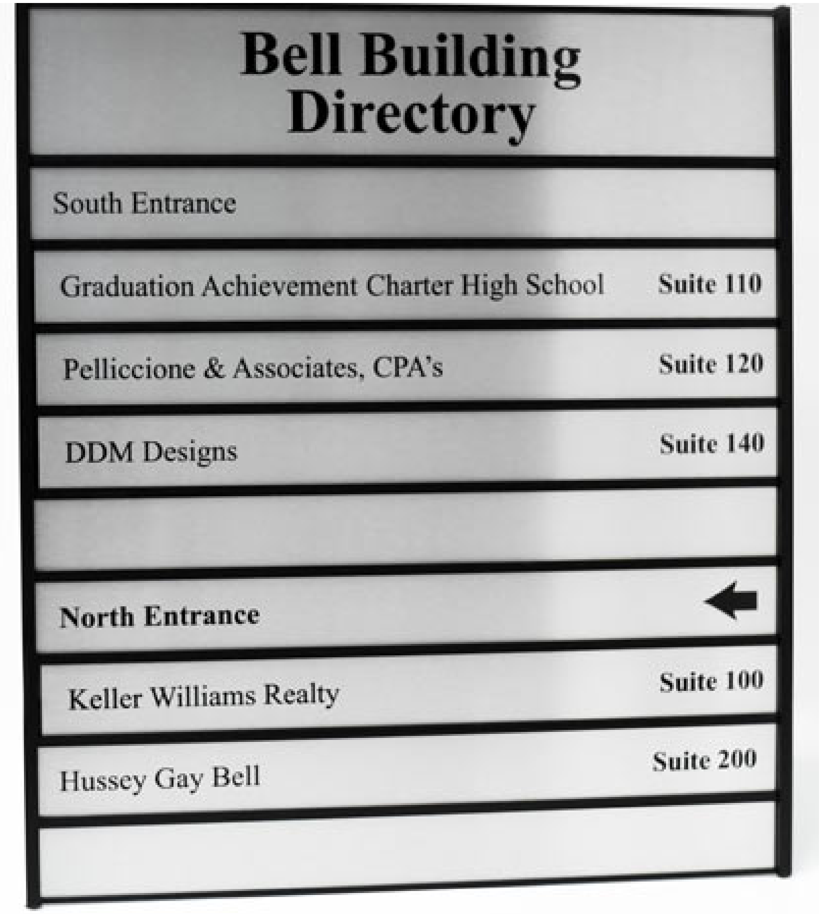
Save/restore

Or more like renting a set of rooms in an office building



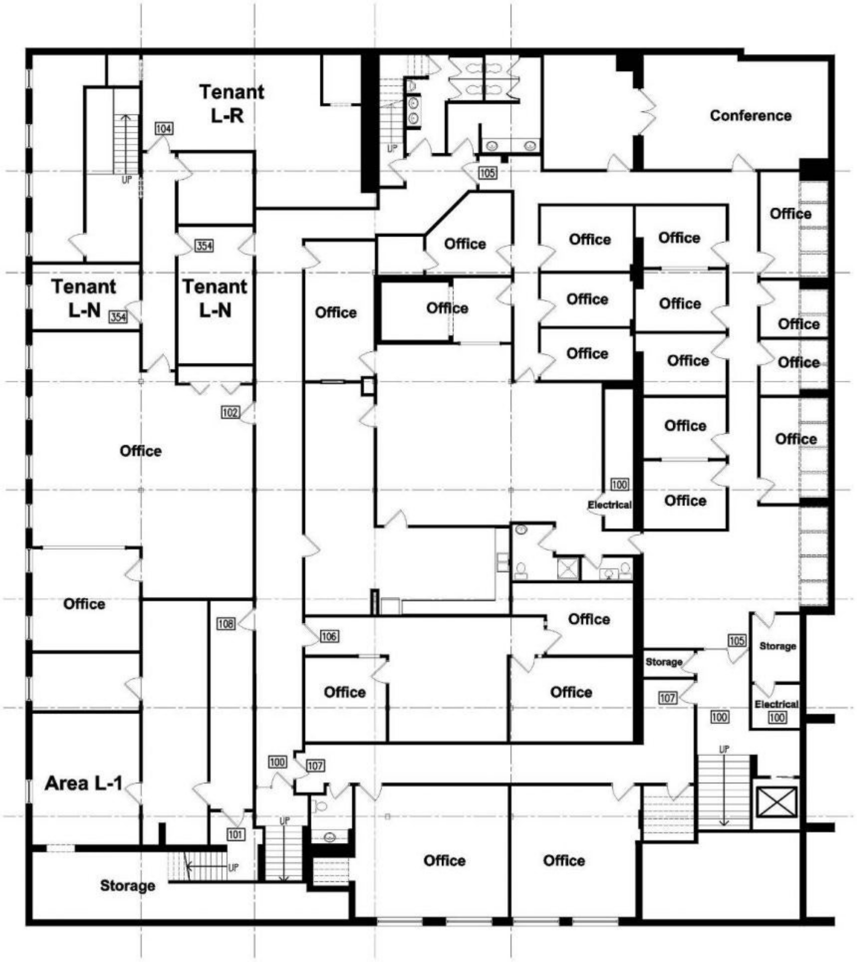
13852 Sq.Ft.
Lower Level
1/16" = 1'-0"

Or more like renting a set of rooms in an office building



Bell Building Directory

South Entrance	
Graduation Achievement Charter High School	Suite 110
Pelliccione & Associates, CPA's	Suite 120
DDM Designs	Suite 140
North Entrance	←
Keller Williams Realty	Suite 100
Hussey Gay Bell	Suite 200

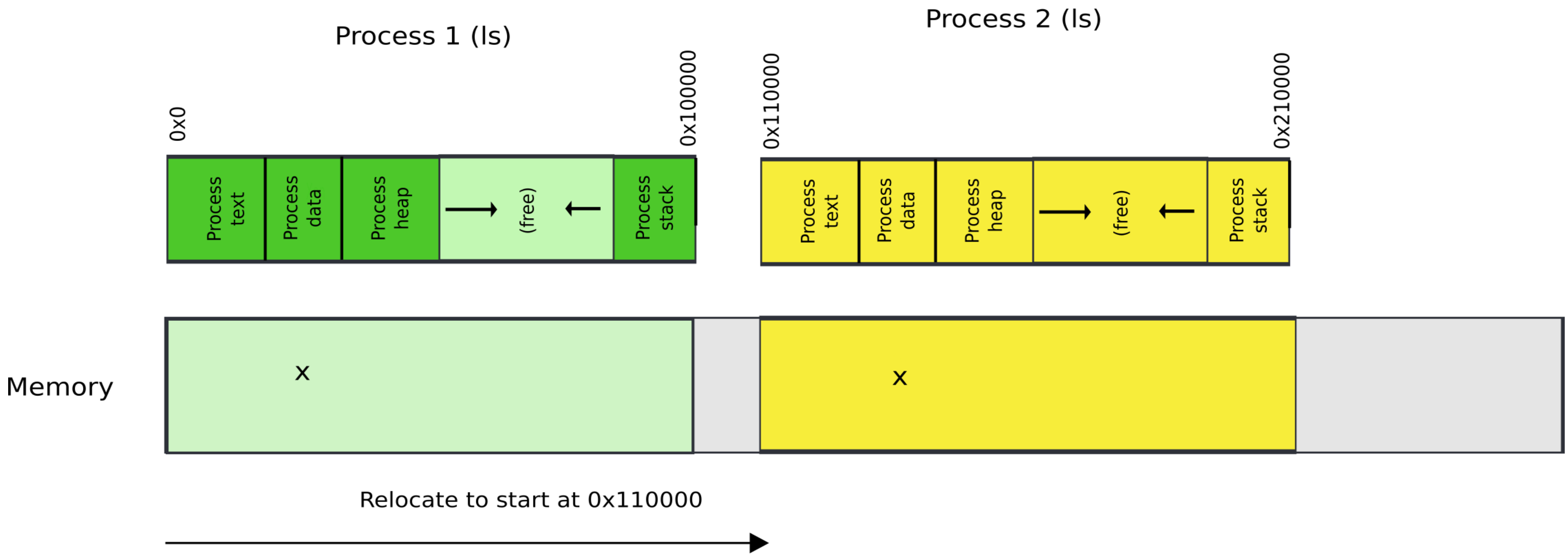


13852 Sq.Ft.
Lower Level
1/16" = 1'-0"

Relocation

- One way to achieve this is to relocate program at different addresses
  - Remember relocation (from linking and loading)

Relocate binaries to work at different addresses



- One way to achieve this is to relocate program at different addresses
- What is the problem?

- One way to achieve this is to relocate program at different addresses
- What is the problem?
  - No isolation

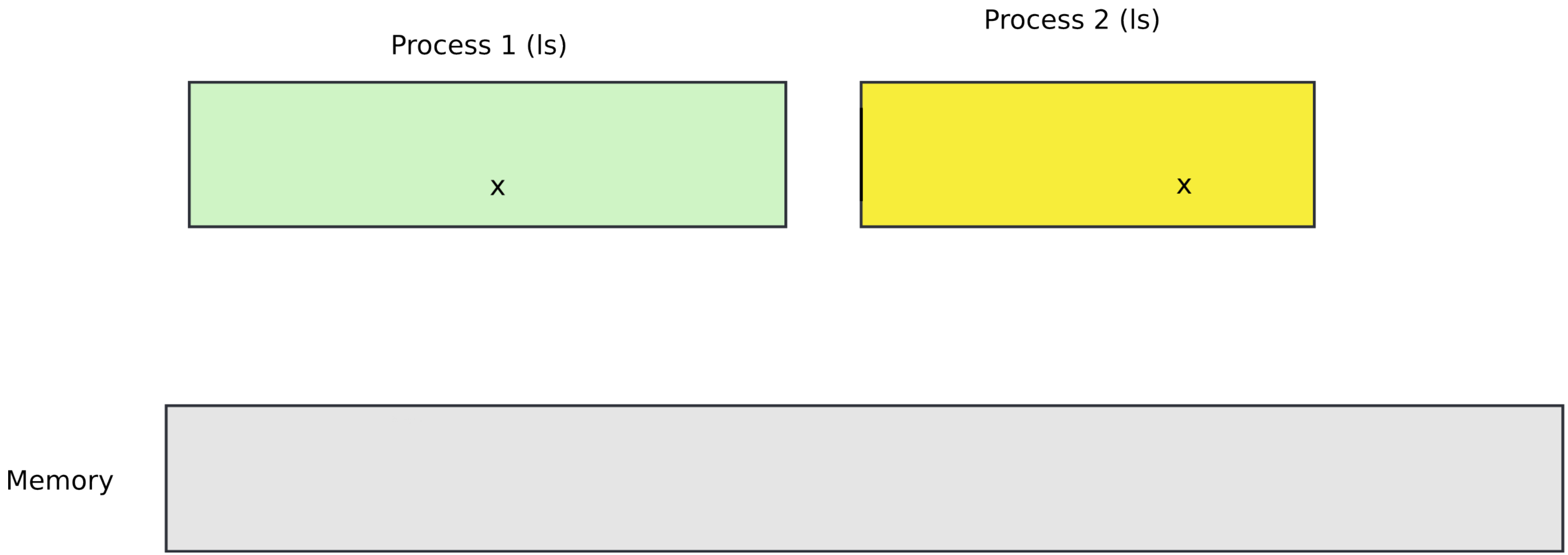
- Another way is to ask for hardware support

This is called segmentation

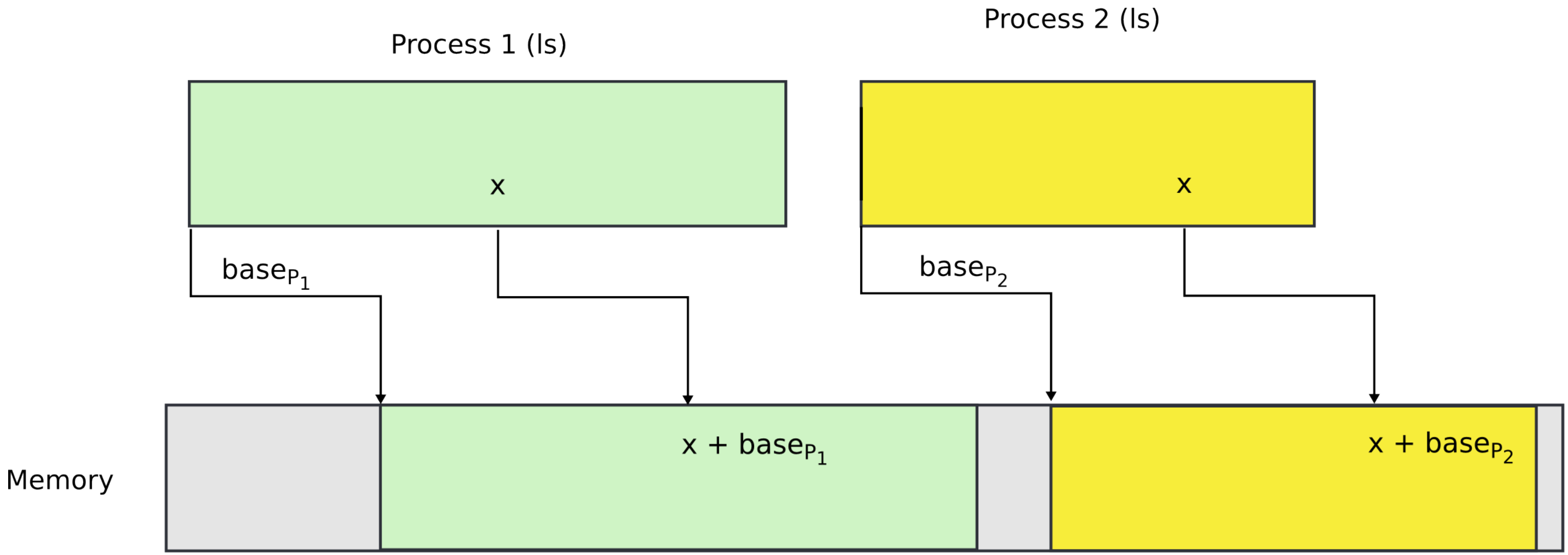
What are we aiming for?

- 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?



Two processes, one memory?



- We want hardware to add base value to every address used in the program

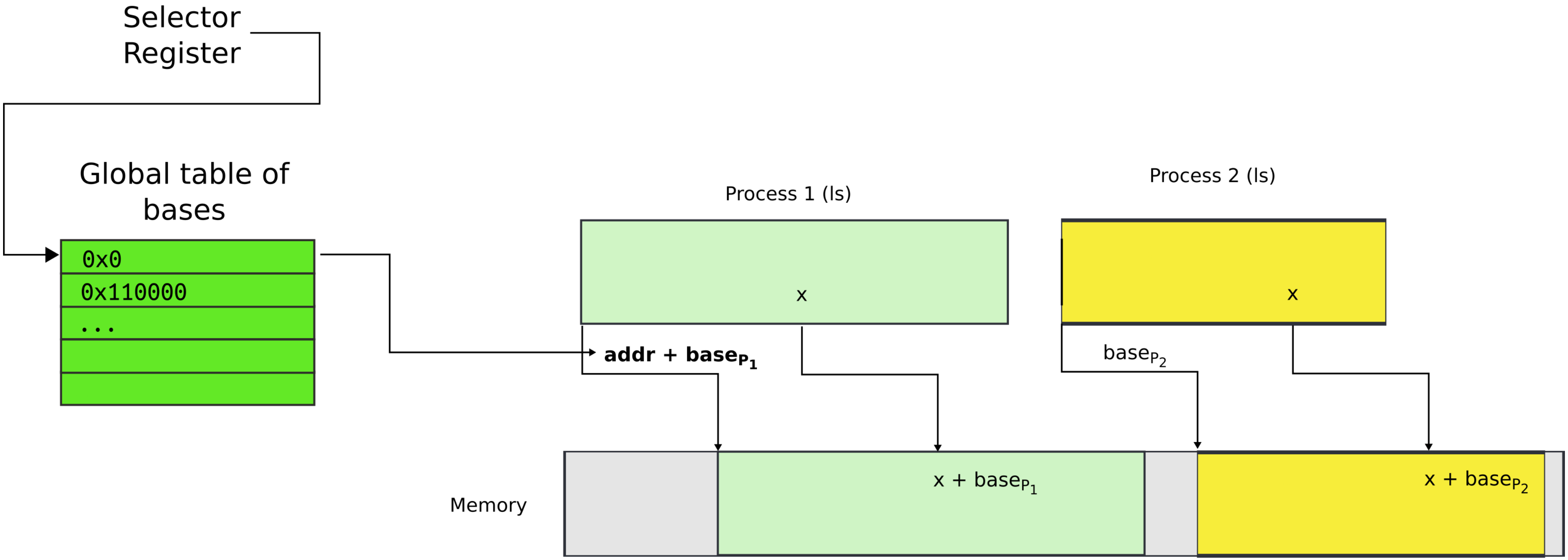
Seems easy

- One problem
  - Where does this base address come from?

Seems easy

- One problem
  - Where does this base address come from?
  - Hardware can maintain a table of base addresses
    - One base for each process
  - Dedicate a special register to keep an index into that table

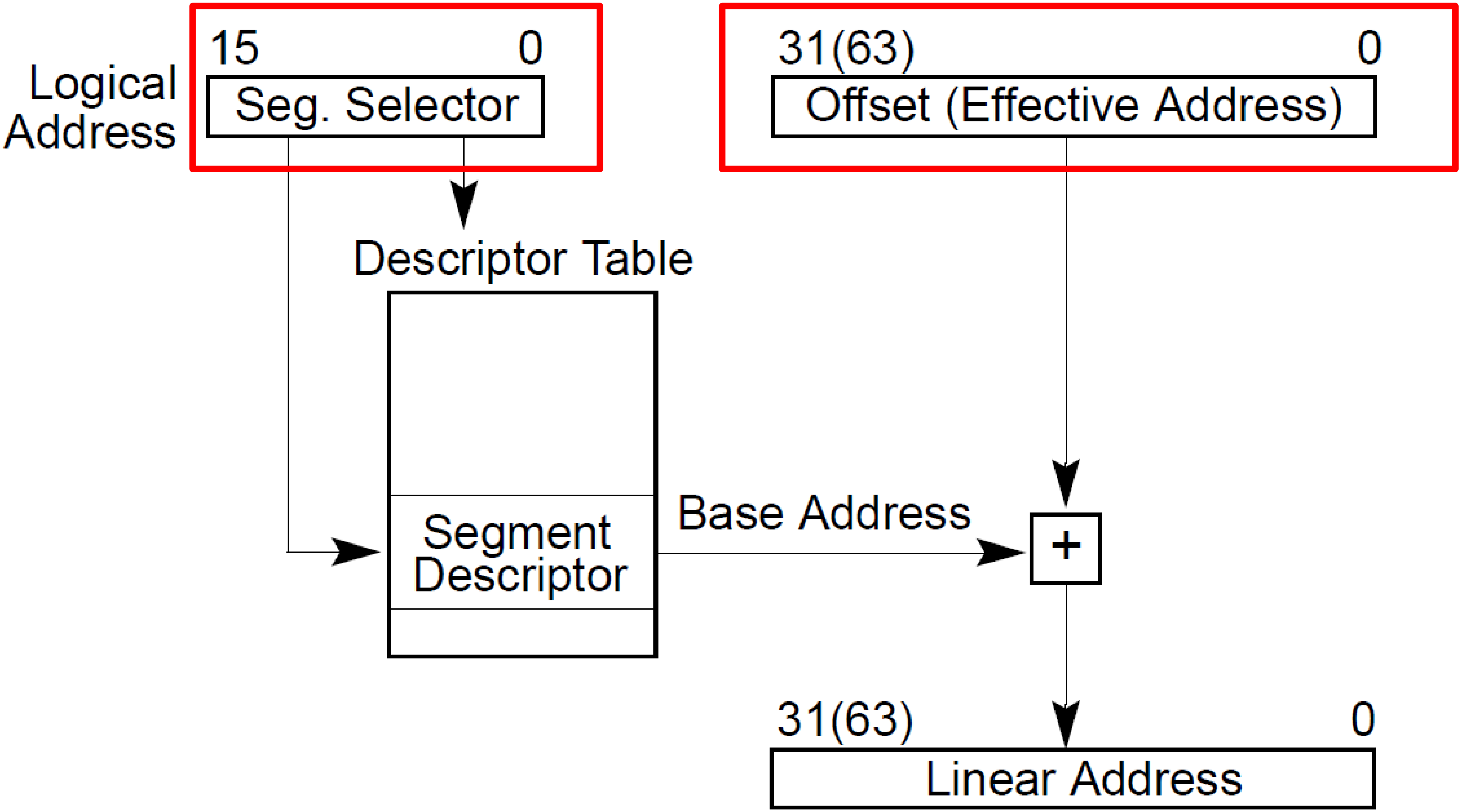
- One problem
  - Where does this base address come from?
  - Hardware can maintain a table of base addresses
    - One base for each process
  - Dedicate a special register to keep an index into that table



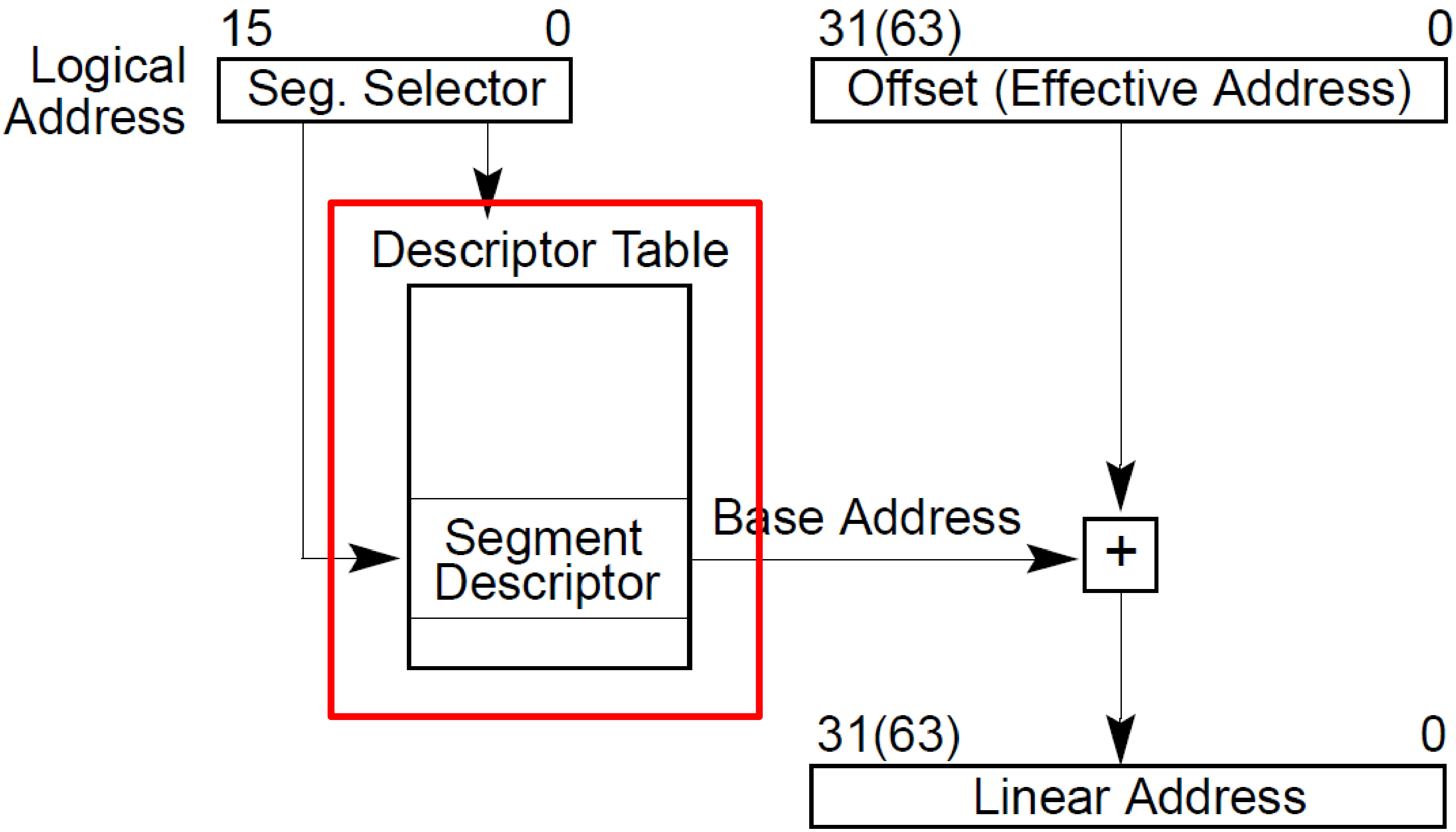
New addressing mode

All addresses are logical address

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

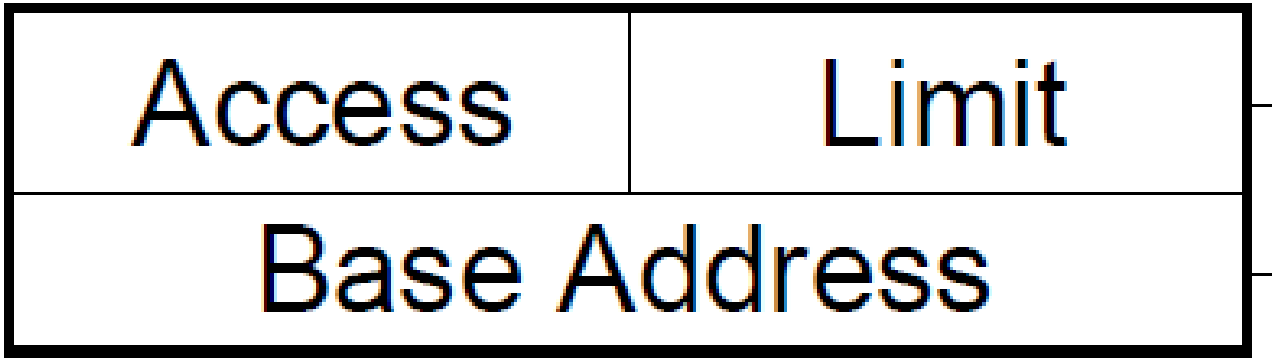


- Segment selector (16 bit)
  - Is simply an index into an array (Descriptor Table)
  - That holds segment descriptors
    - Base and limit (size) for each segment



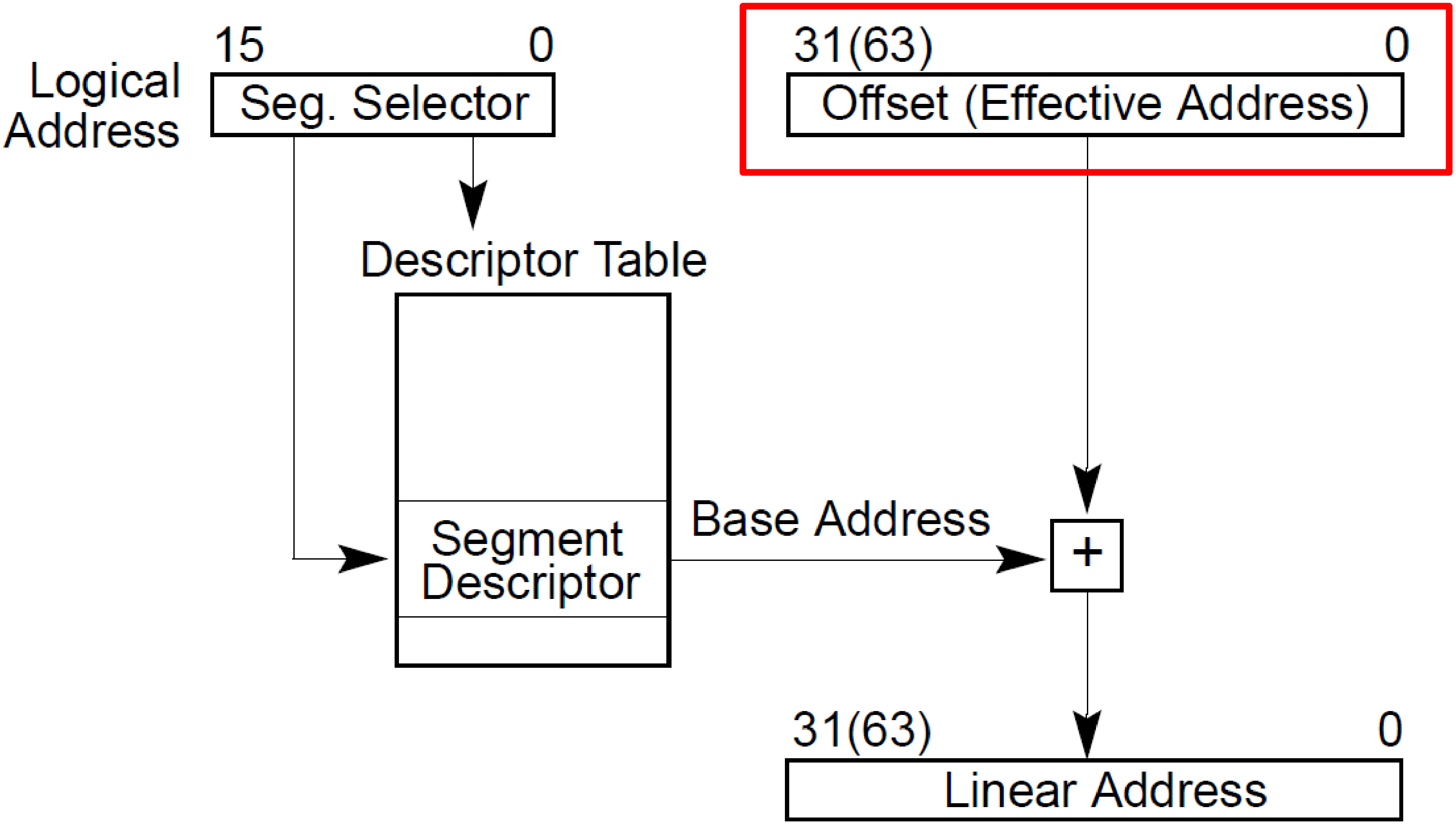
Elements of the descriptor table are **segment descriptors**

- Base address
  - 0 – 4 GB
- Limit (size)
  - 0 – 4 GB
- Access rights
  - Executable, readable, writable
  - Privilege level (0 - 3)

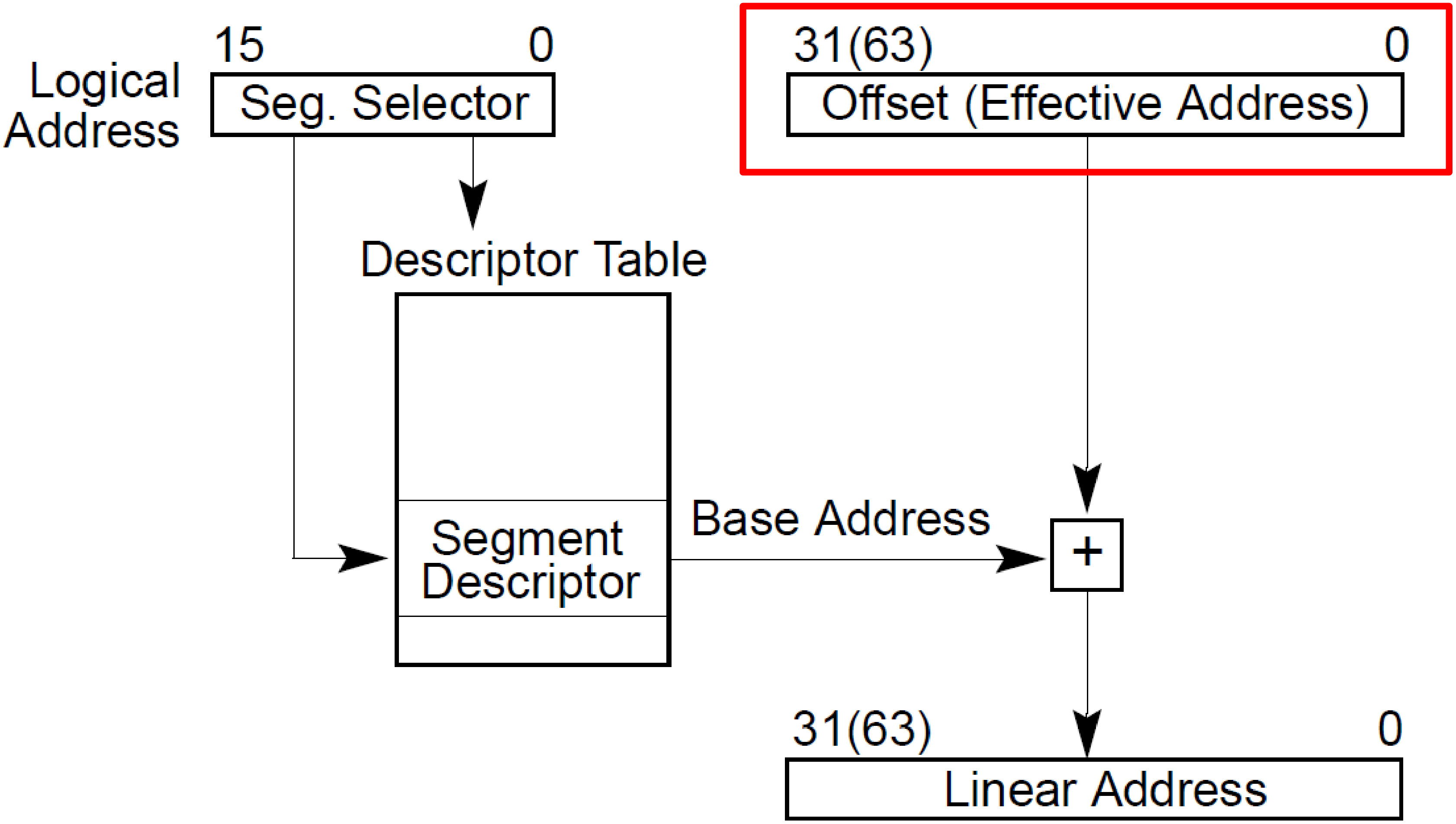


Access	Limit
Base Address	

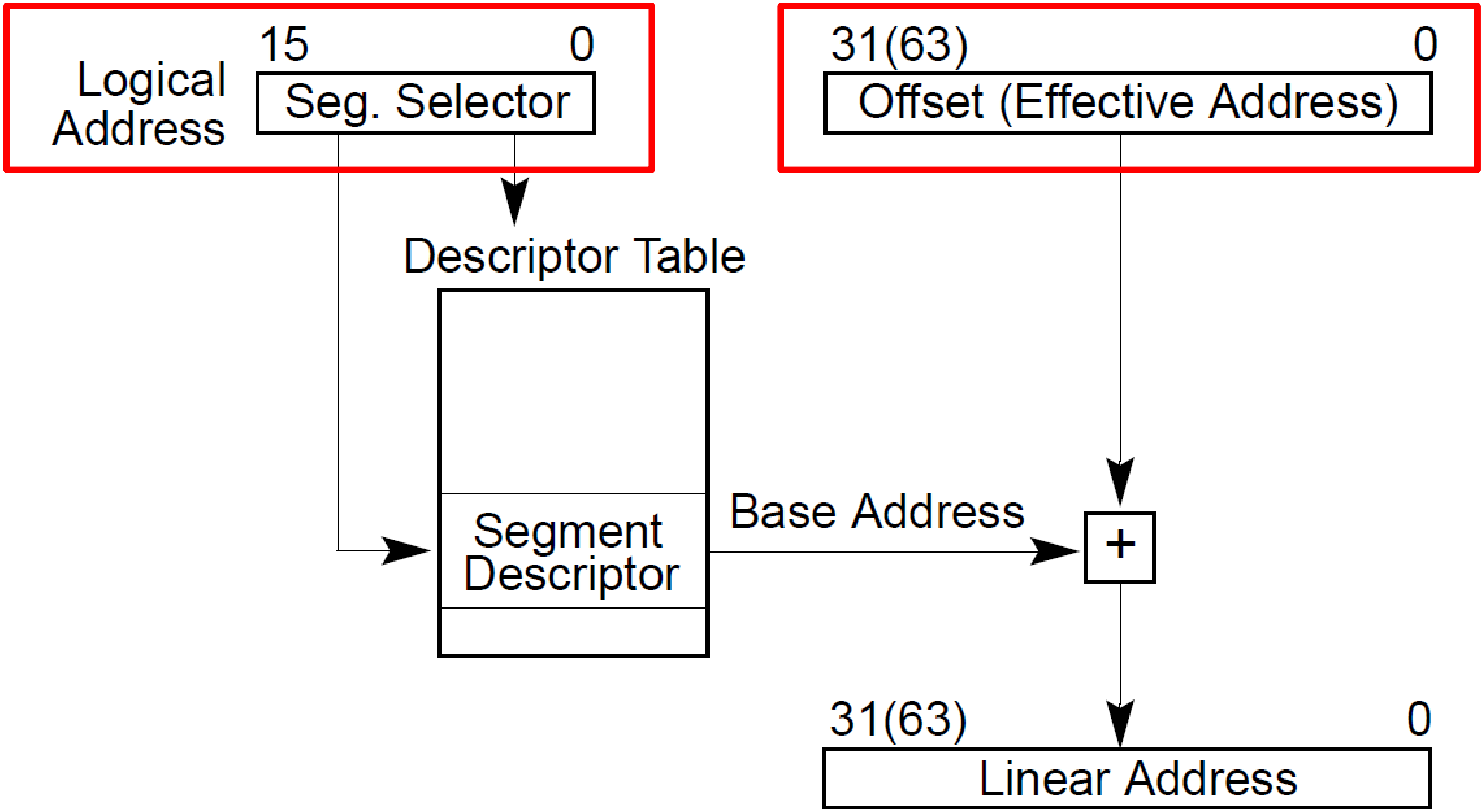
- Offsets into segments (x in our example) or “Effective addresses” are in registers



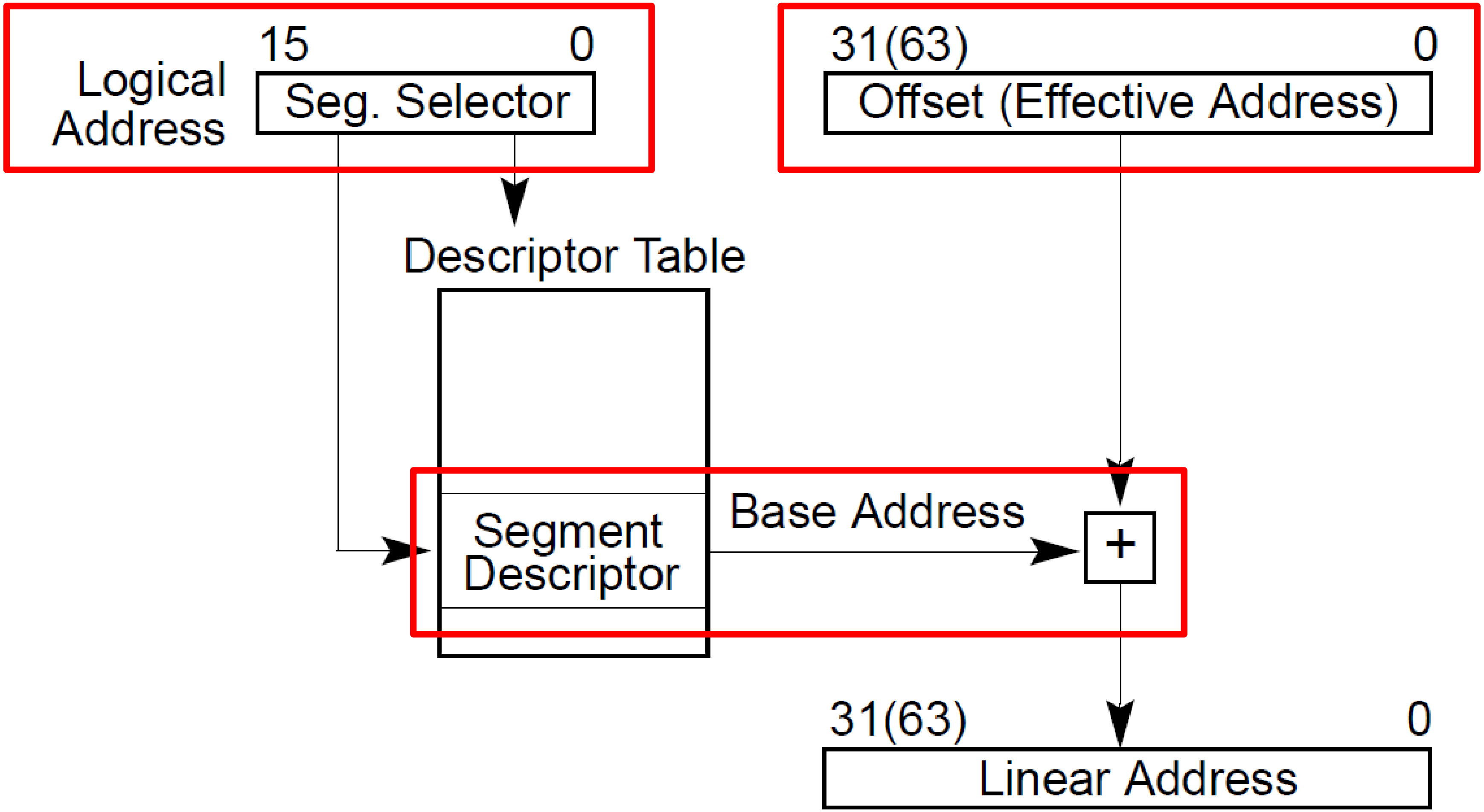
- Logical addresses are translated into physical
  - *Effective address + DescriptorTable[selector].Base*



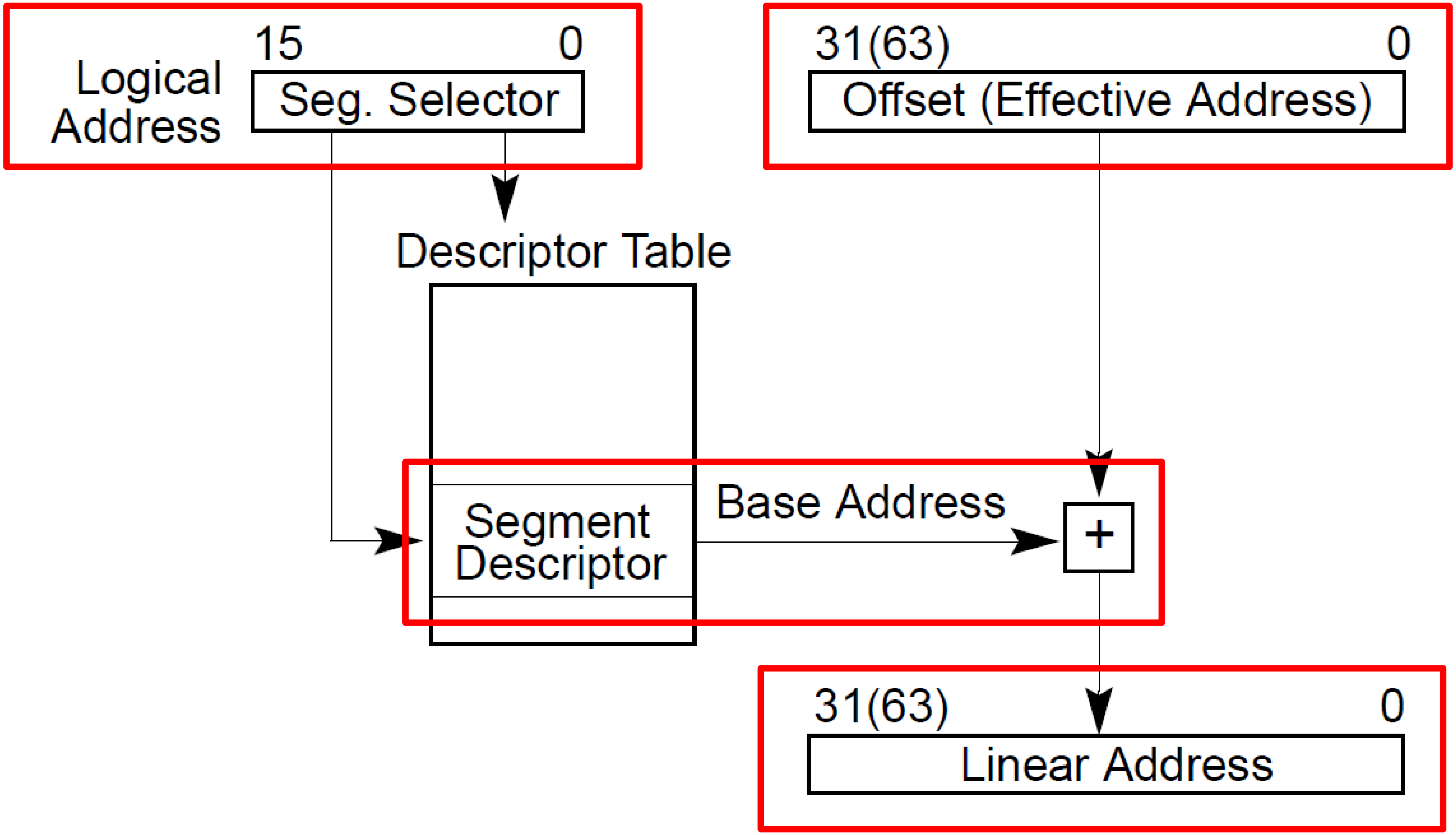
- Logical addresses are translated into physical
  - Effective address + DescriptorTable[selector].Base



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



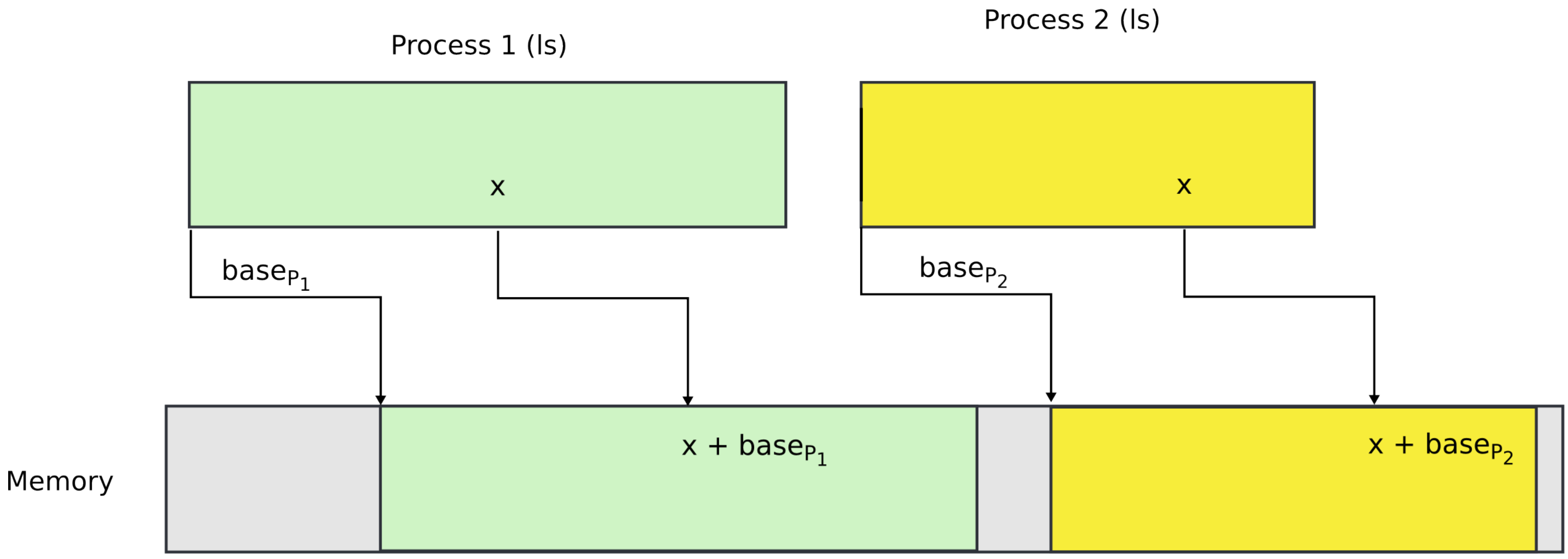
- Logical addresses are translated into physical
  - Effective address + DescriptorTable[selector].Base



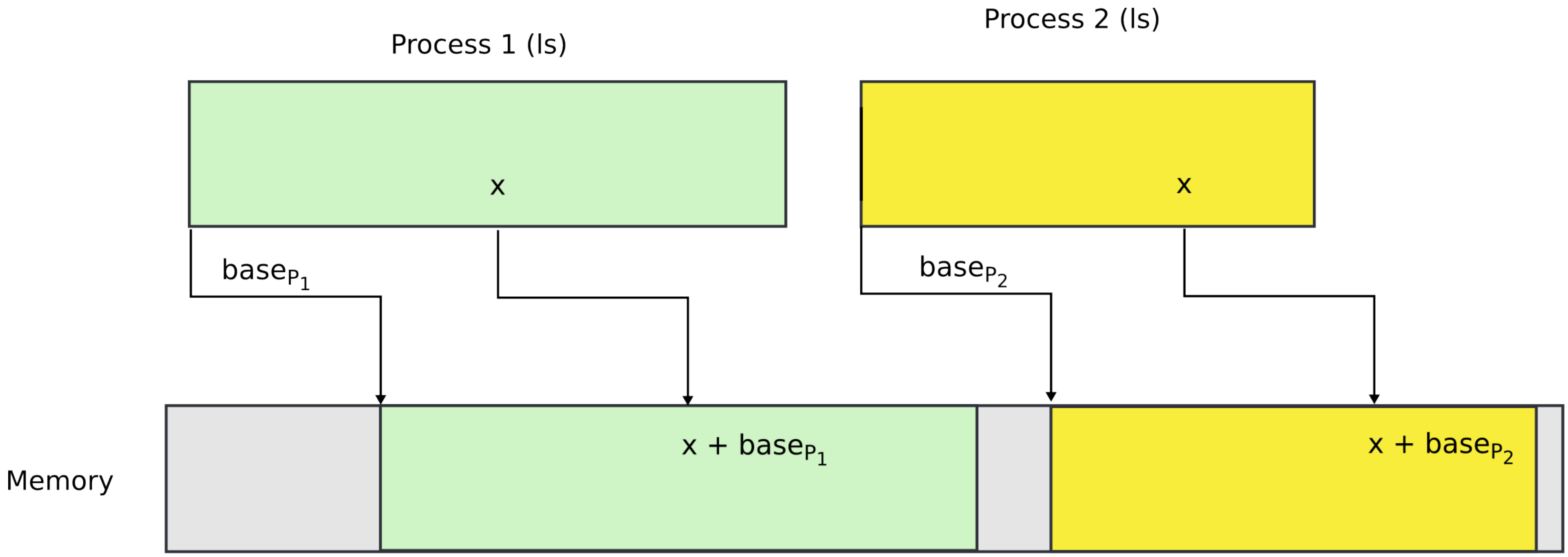
- *Physical address =*

  Effective address + DescriptorTable[selector].Base

  - Effective addresses (or offsets) are in registers
  - **Selector is in a special register**



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



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

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

- 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)

Programming model, cont.

- This is cumbersome,
- Instead the idea is: infer code, data and stack segments from the 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

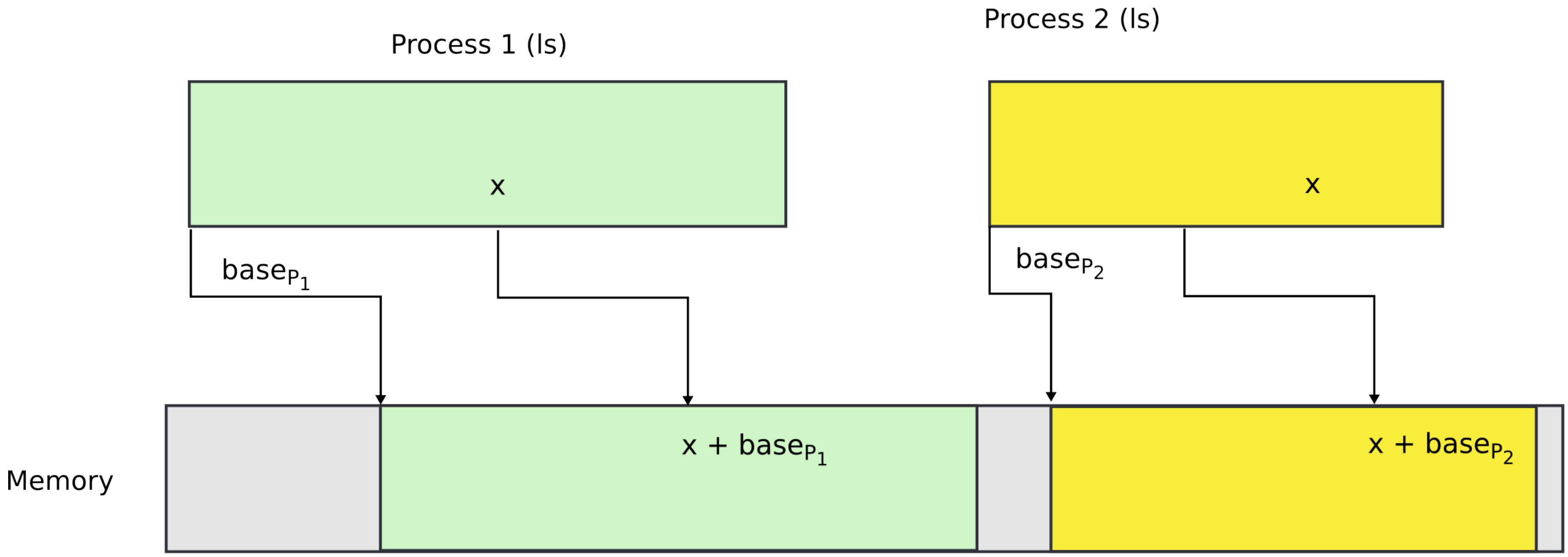
- Data
  - DS, ES, FS, GS
  - 4 possible data segments can be used at the same time

Stack segment

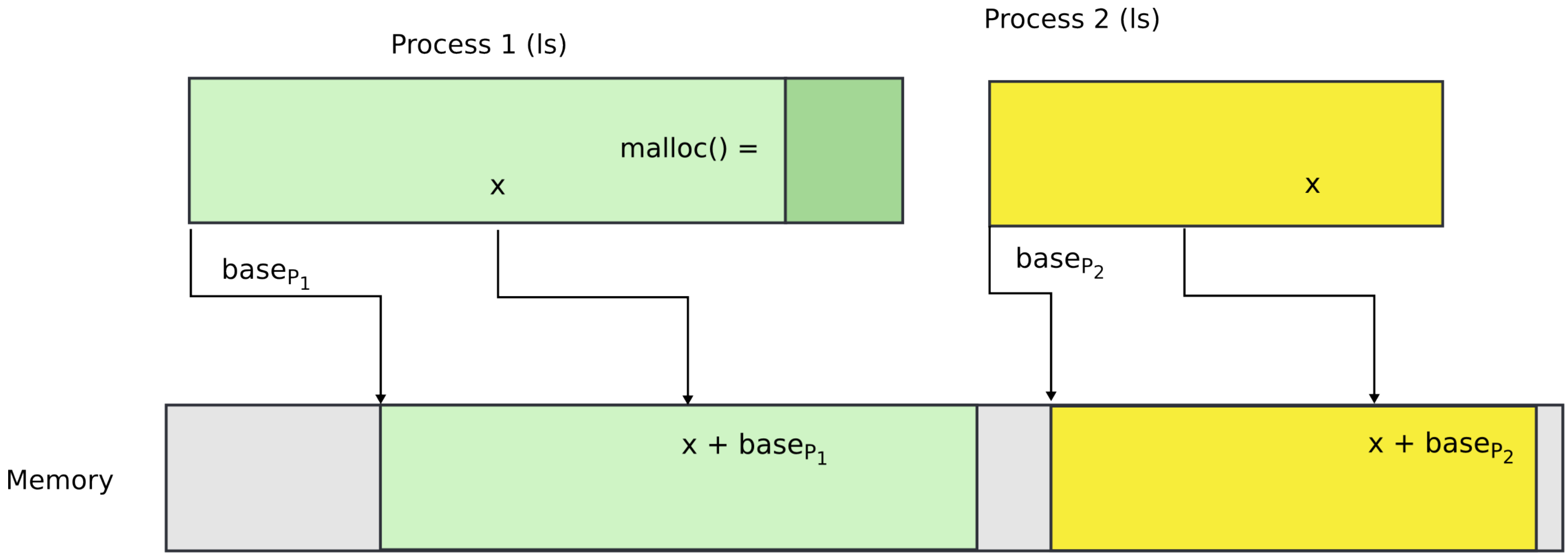
- Stack
  - SS
- Can be loaded explicitly
  - OS can set up multiple stacks
  - Of course, only one is accessible at a time

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

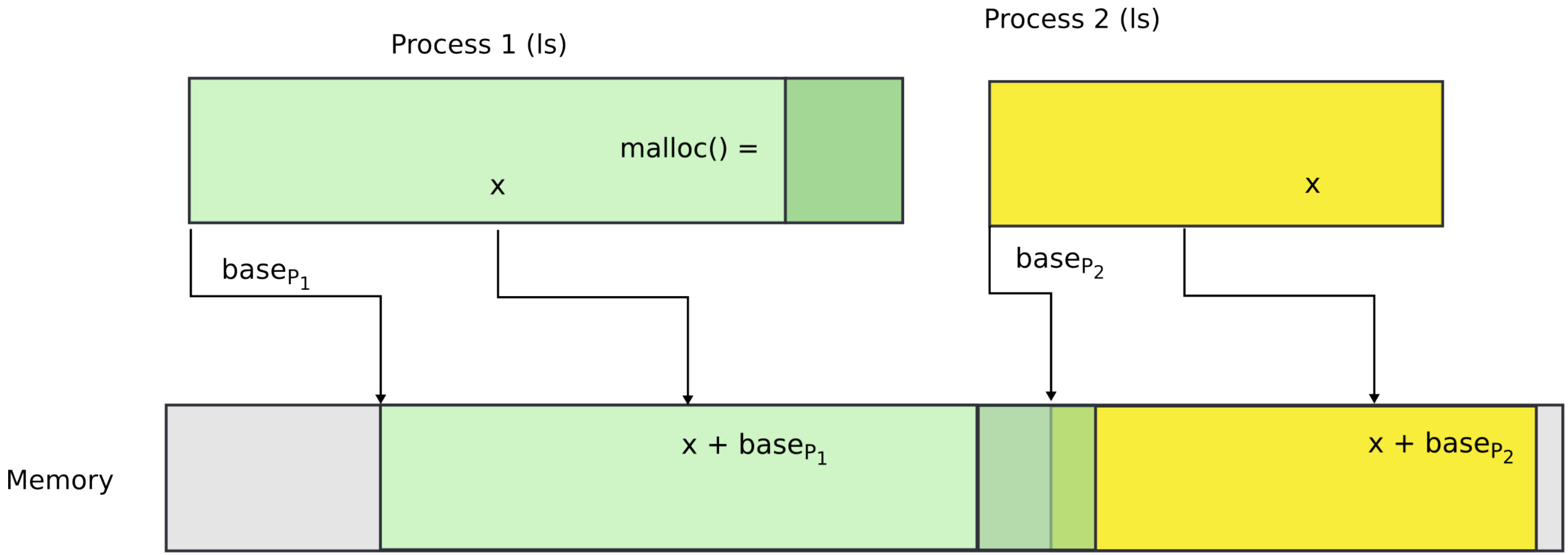
Segmentation is ok... but



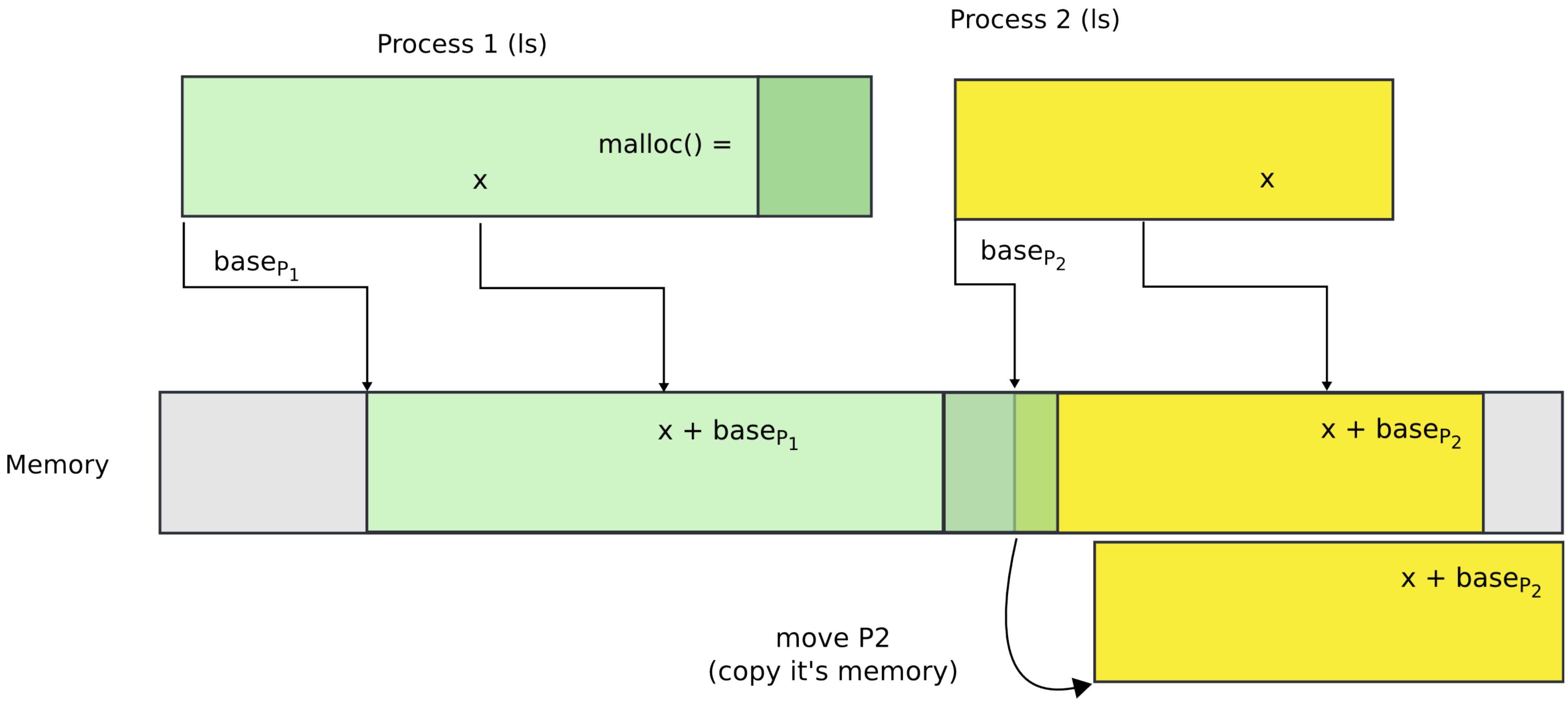
What if process needs more memory?



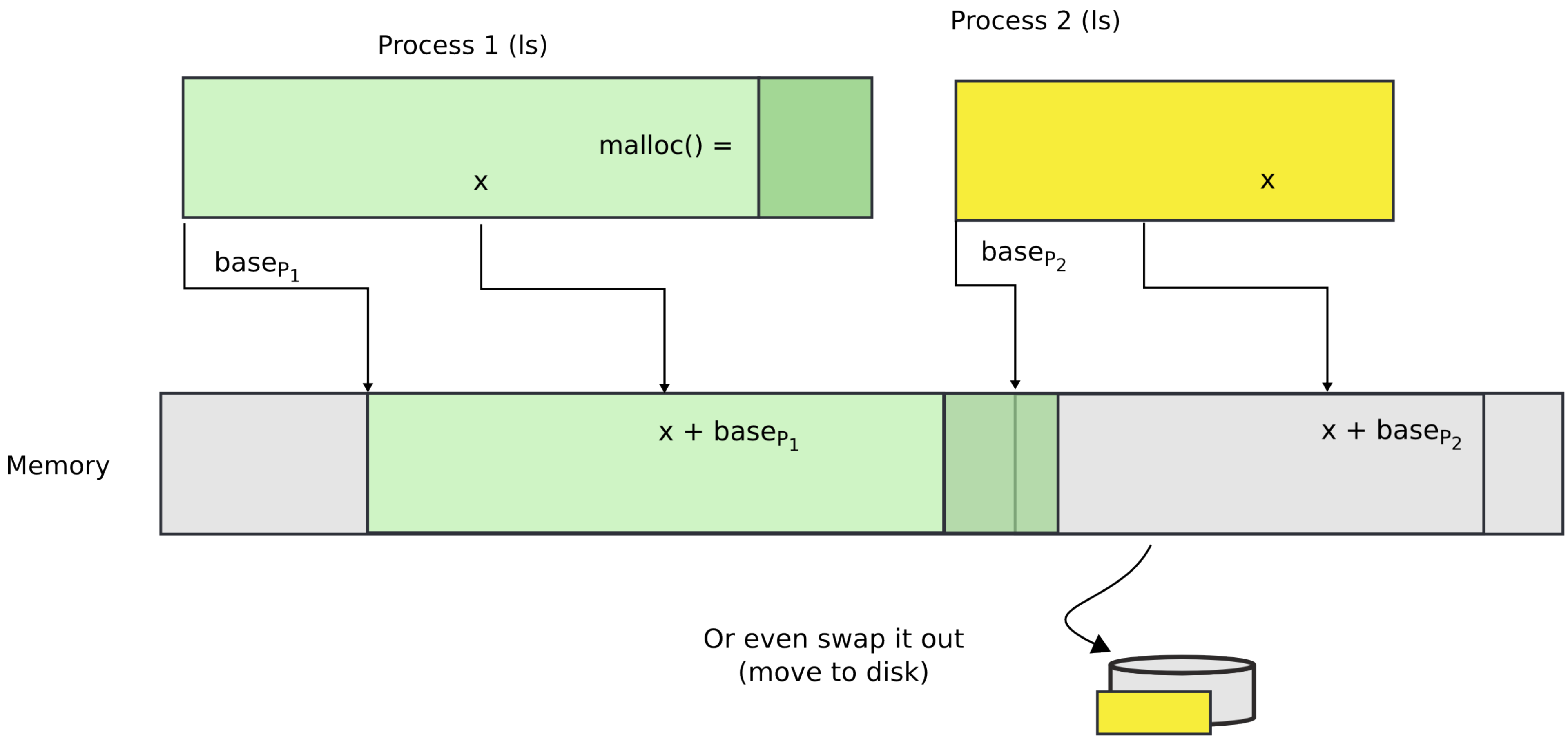
What if process needs more memory?



You can move P2 in memory



Or even swap it out to disk

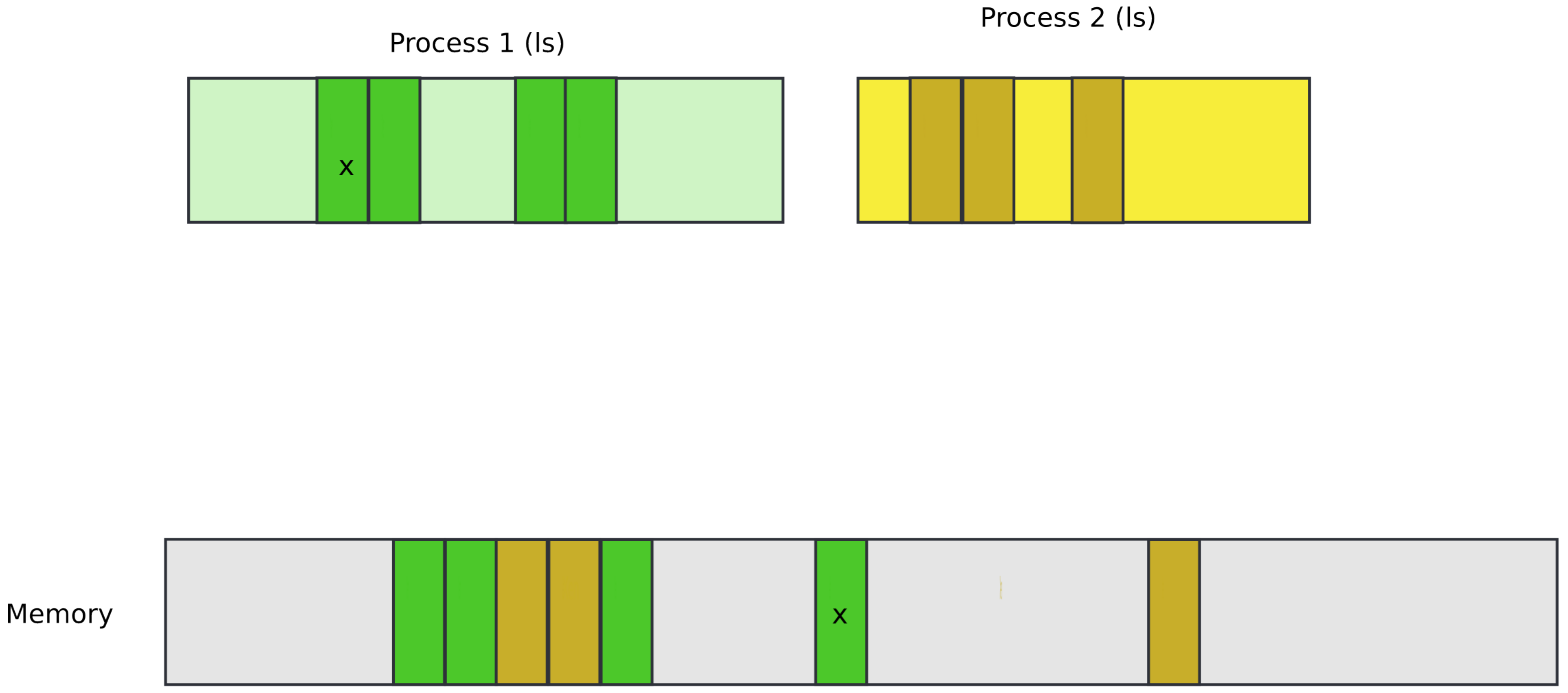


Problems with segments

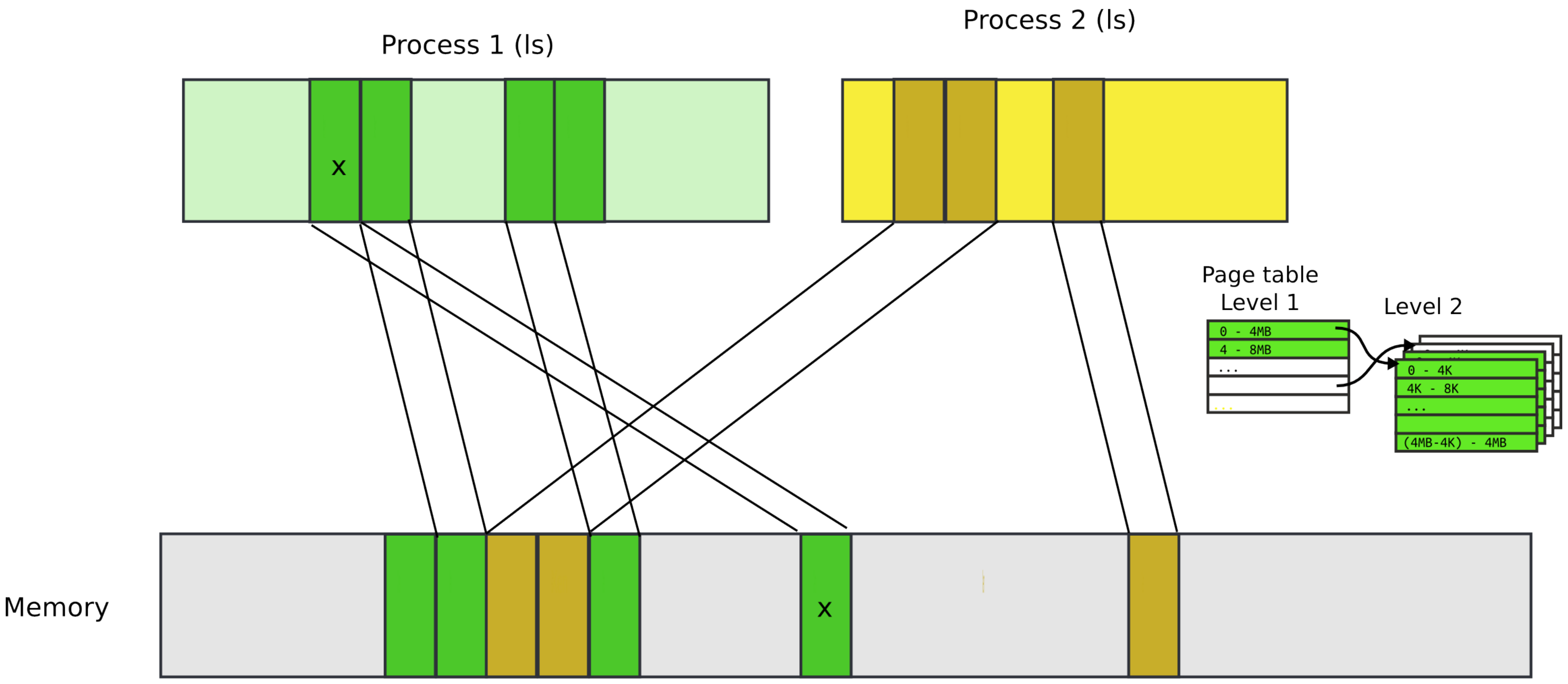
- But it's inefficient
  - Relocating or swapping the entire process takes time
- Memory gets fragmented
  - There might be no space (gap) for the swapped out process to come in
  - Will have to swap out other processes

Paging

Pages



Pages

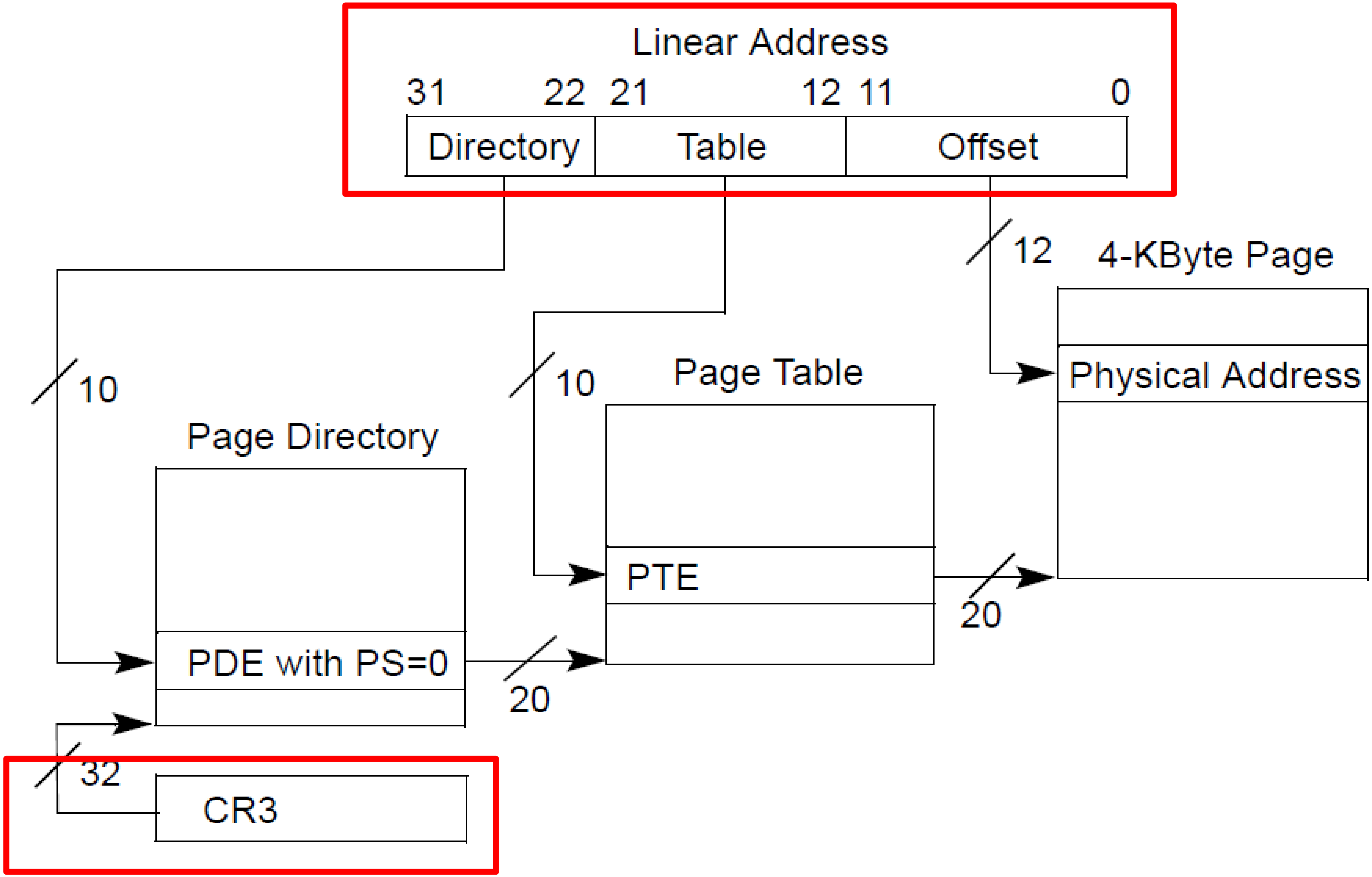


Process 1 (ls)
x
Process 2 (ls)
Page table
Level 1
0 - 4MB
4 - 8MB
...
...
Level 2
0 - 4K
4K - 8K
...
(4MB-4K) - 4MB
Memory
x

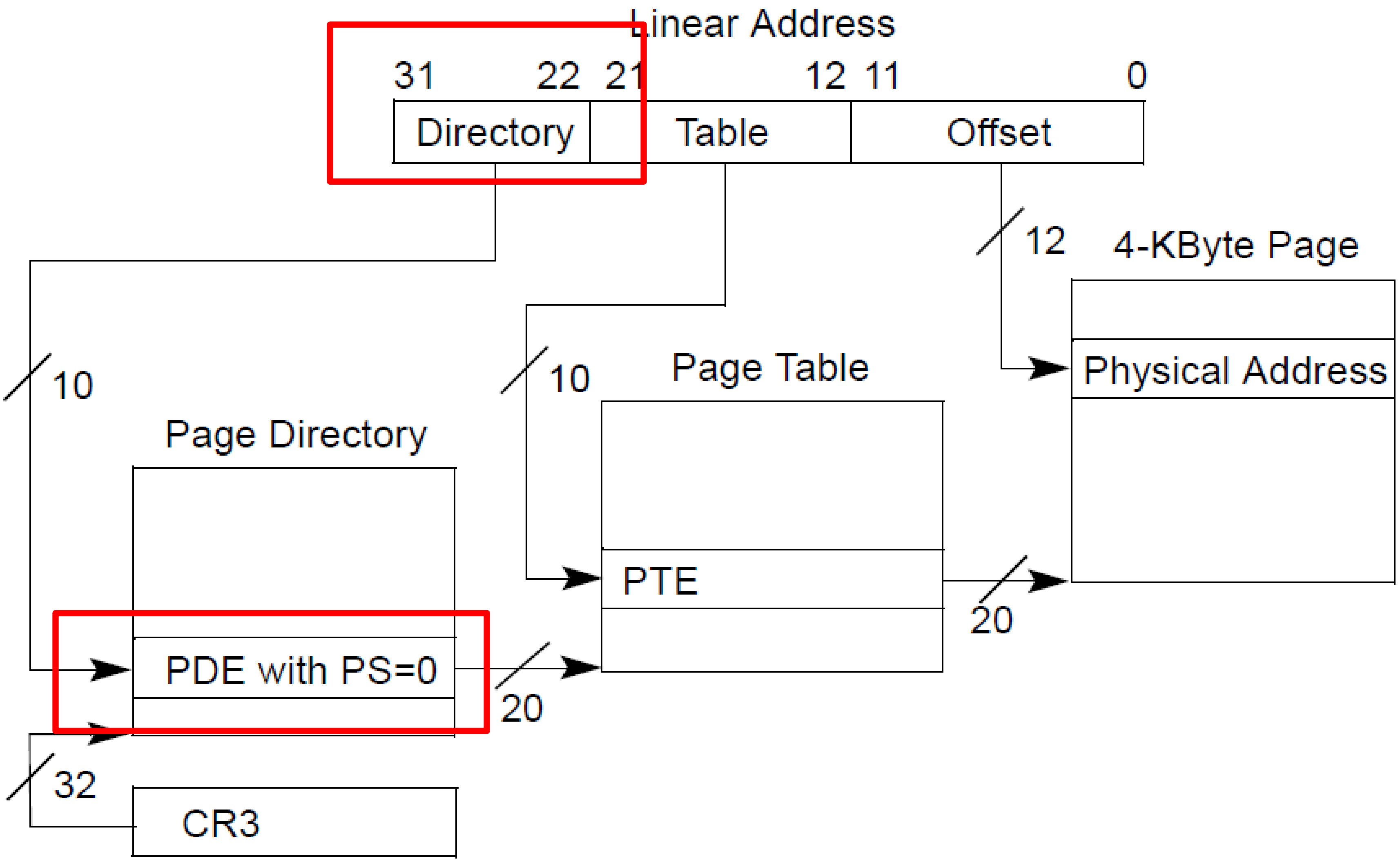
Paging idea

- 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

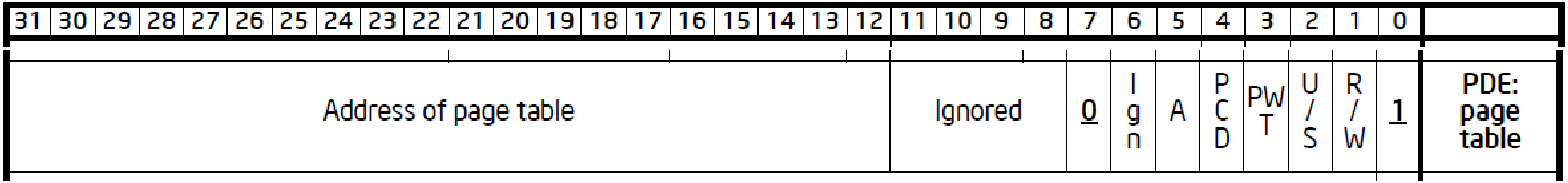
Page translation



Page translation



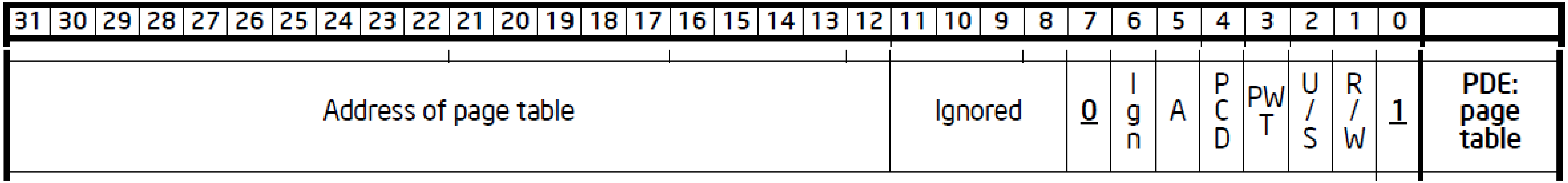
Page directory entry (PDE)



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

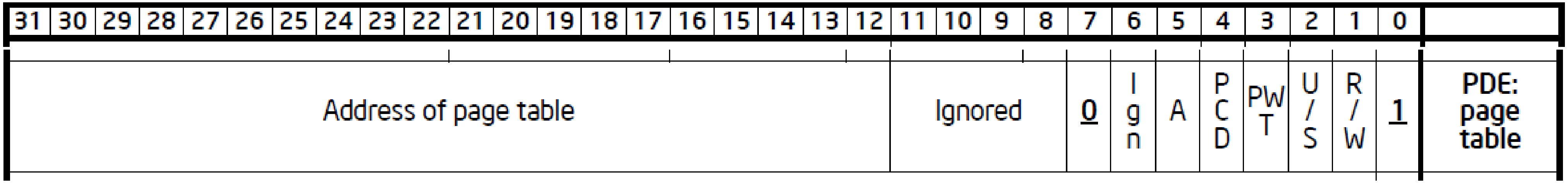
Page directory entry (PDE)



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				**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

Page directory entry (PDE)

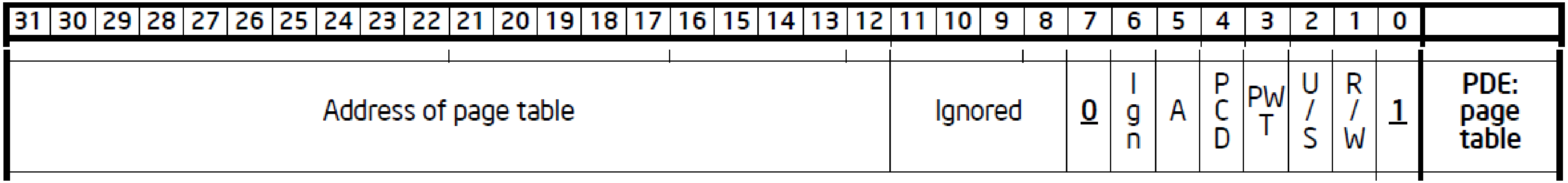


31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page table																				Ignored				**0**	Ign	A	PCD	PWT	U/S	R/W	**1**	**PDE: page table**

- 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
  - Pages start at 4KB (page aligned boundary)

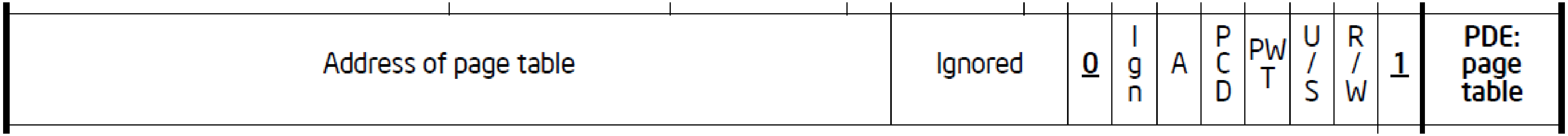
Page directory entry (PDE)



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				**0**	Ign	A	PCD	PWT	U/S	R/W	**1**	**PDE: page table**

- Bit #1: R/W – writes allowed?
  - But allowed where?

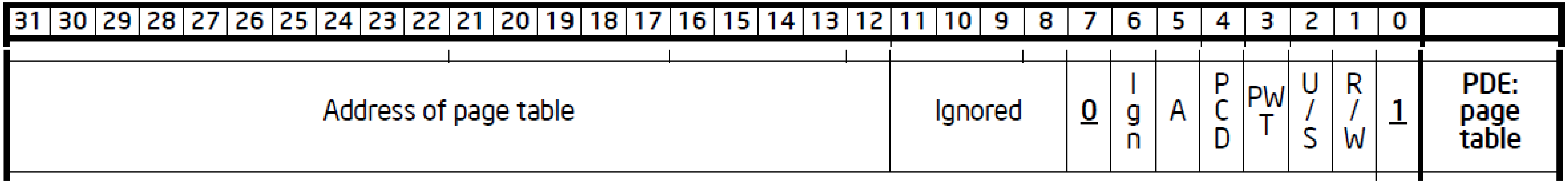
Page directory entry (PDE)



Address of page table	Ignored	**0**	Ign	A	PCD	PWT	U/S	R/W	**1**	**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

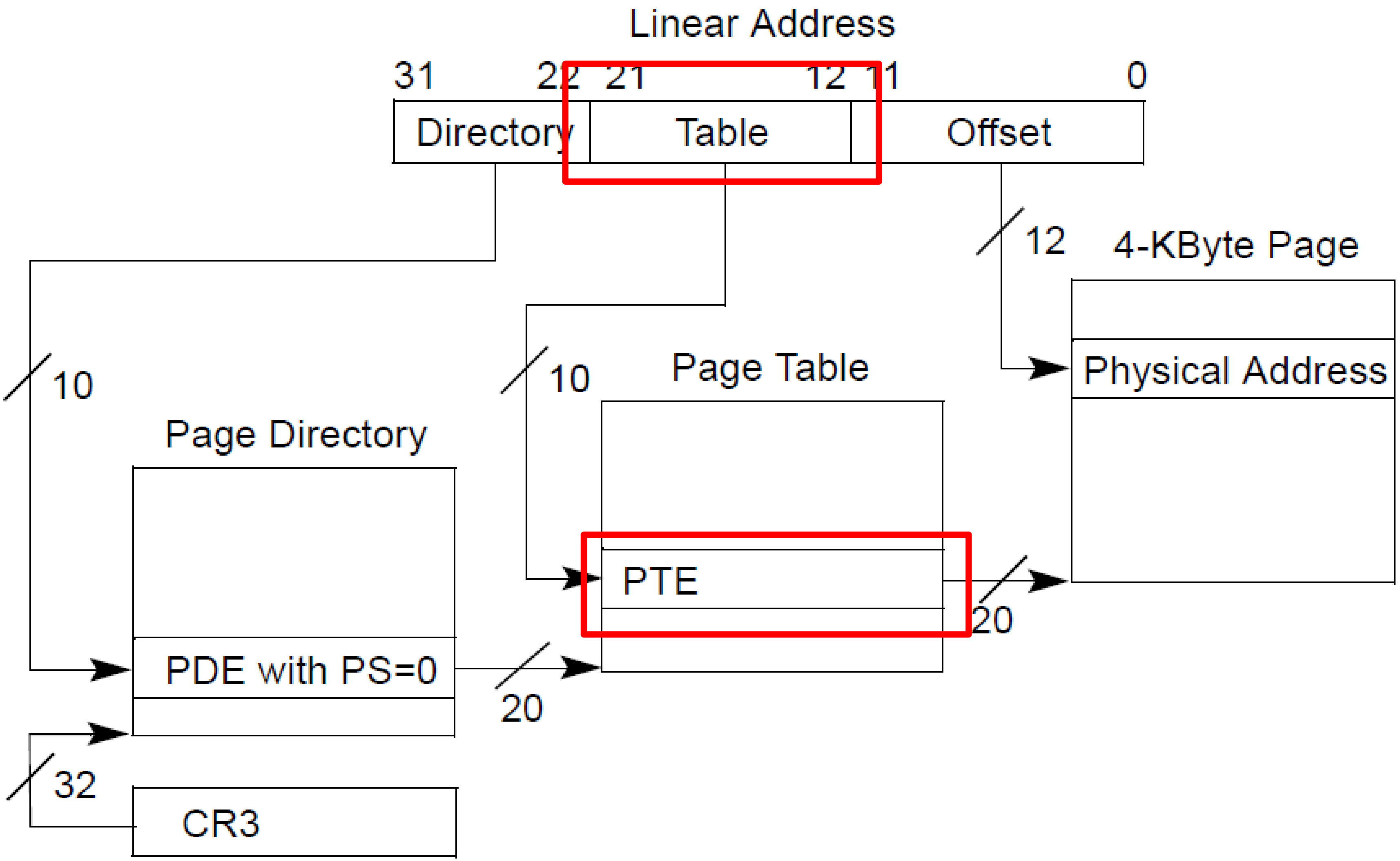
Page directory entry (PDE)



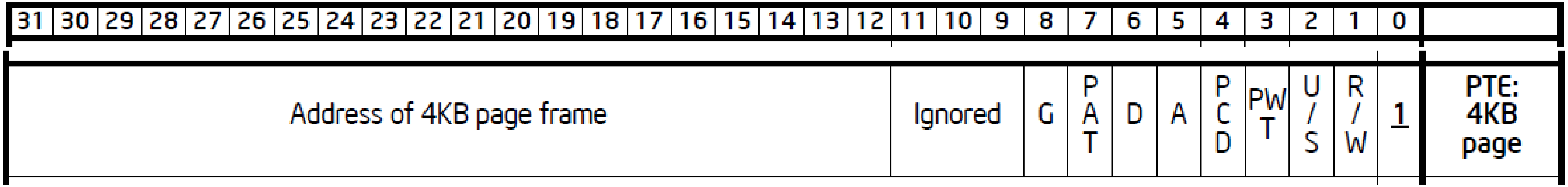
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page table																				Ignored				0	Ign	A	PCD	PWT	U/S	R/W	1	**PDE: page table**

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

Page translation



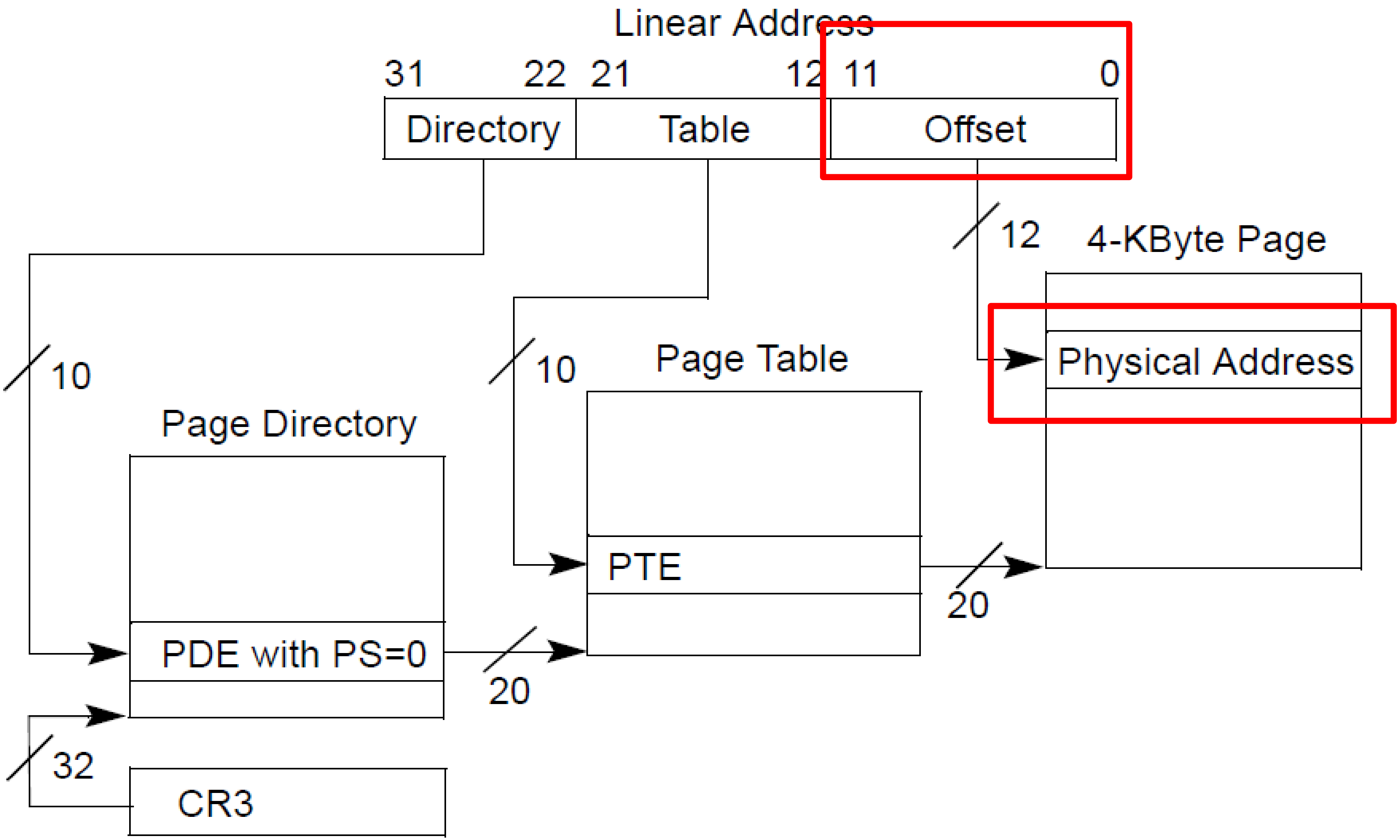
Page table entry (PTE)



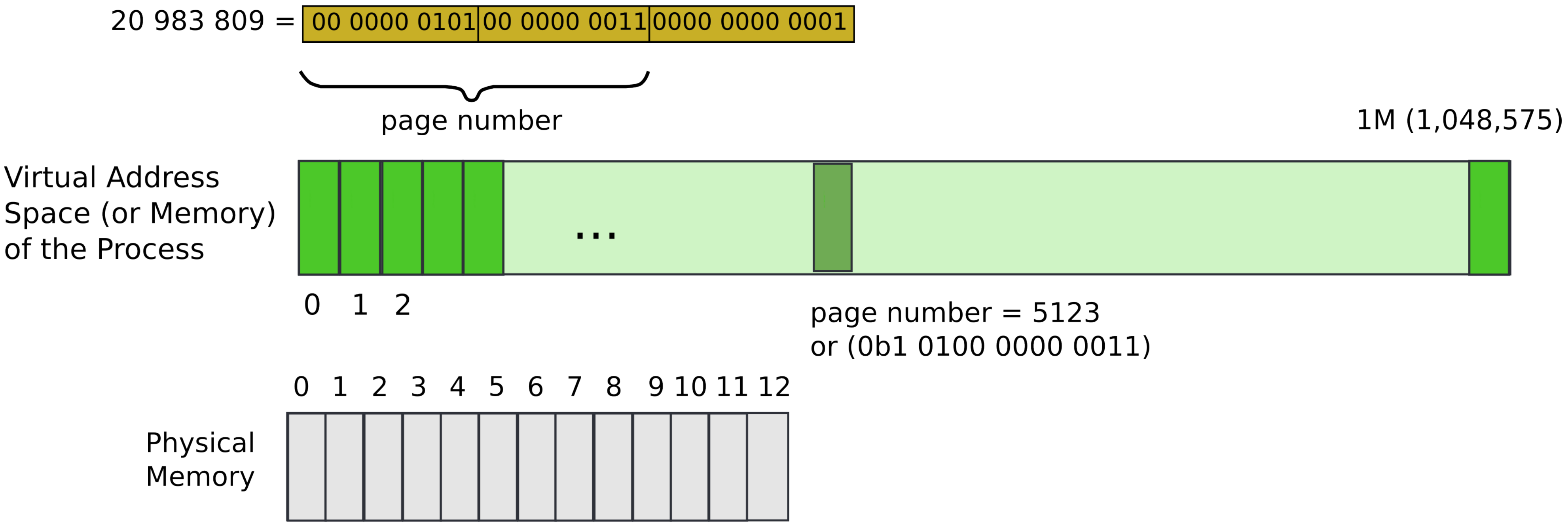
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of 4KB page frame																				Ignored			G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page

- 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



```
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

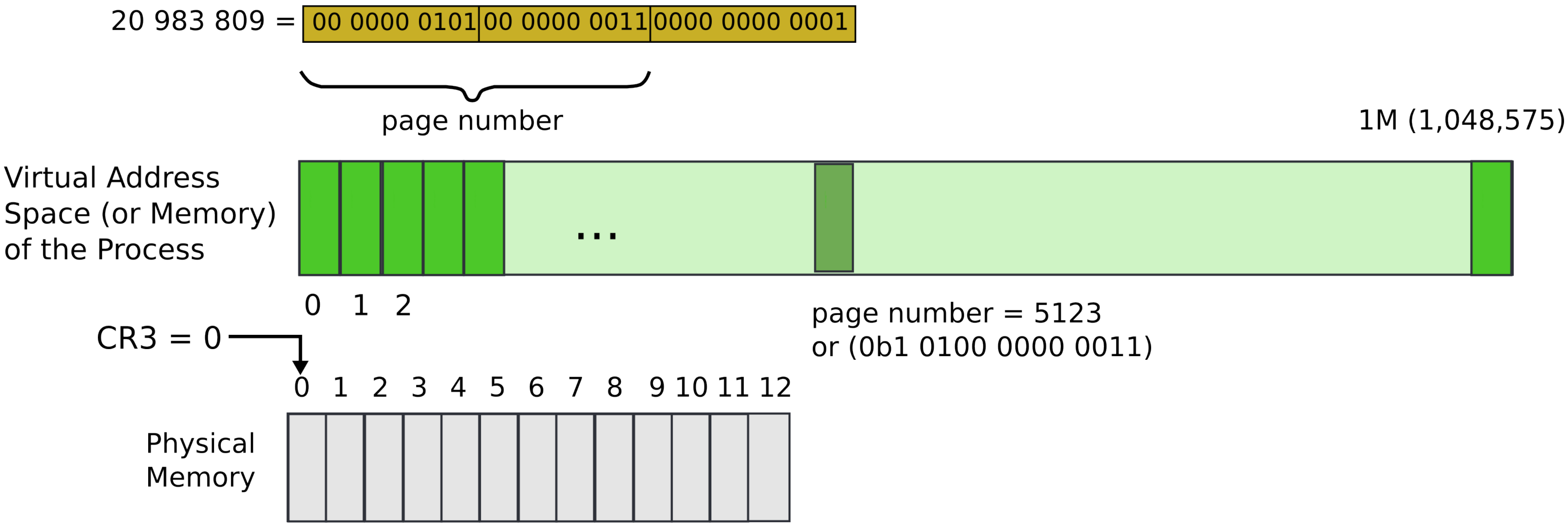
1M (1,048,575)

Virtual Address Space (or Memory) of the Process

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

...

0 1 2

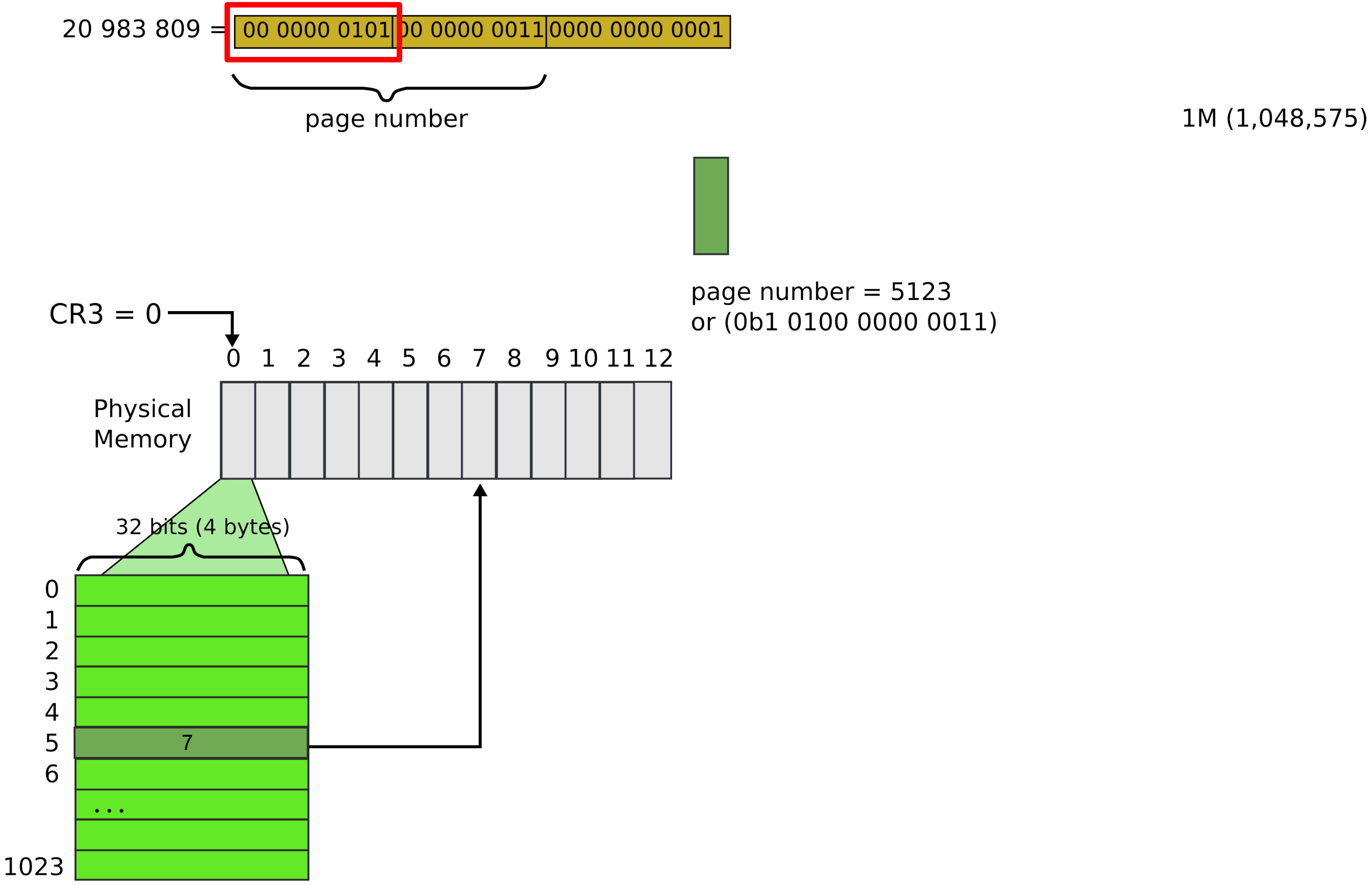
page number = 5123
or (0b1 0100 0000 0011)

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 = 00 0000 0101 00 0000 0011 0000 0000 0001

page number

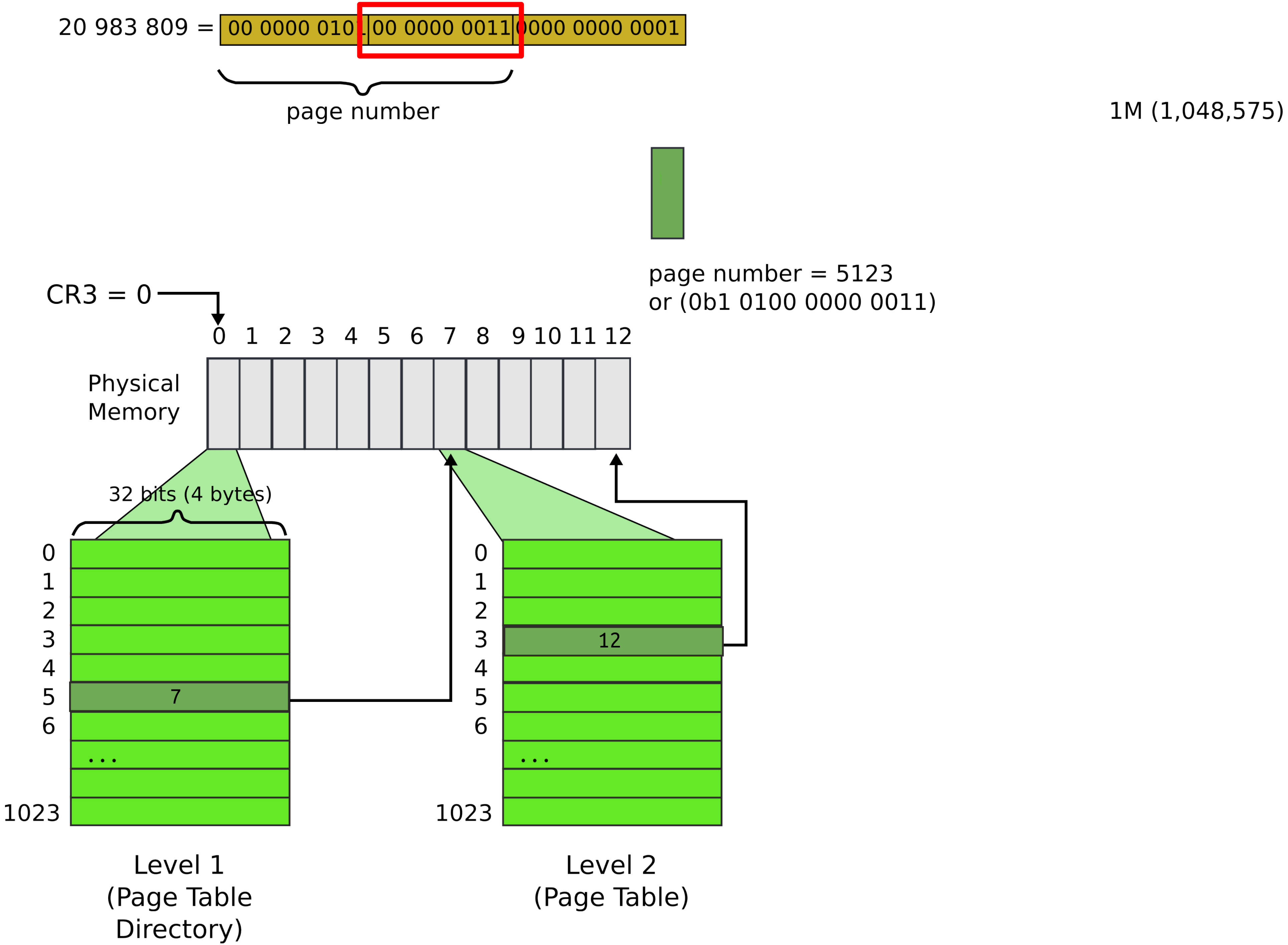
1M (1,048,575)

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

CR3 = 0

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

page number

1M (1,048,575)

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

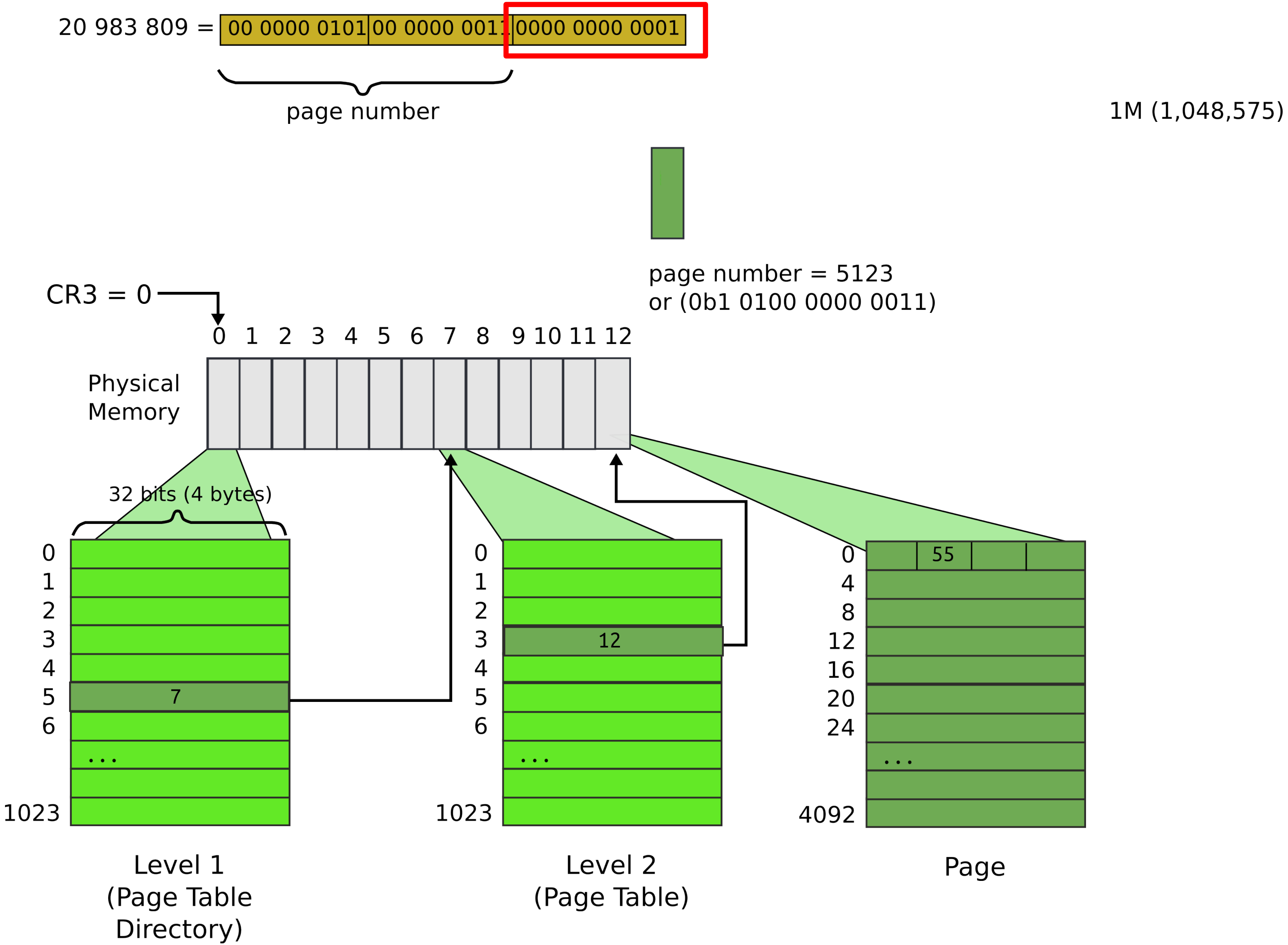
CR3 = 0

Physical Memory

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
```



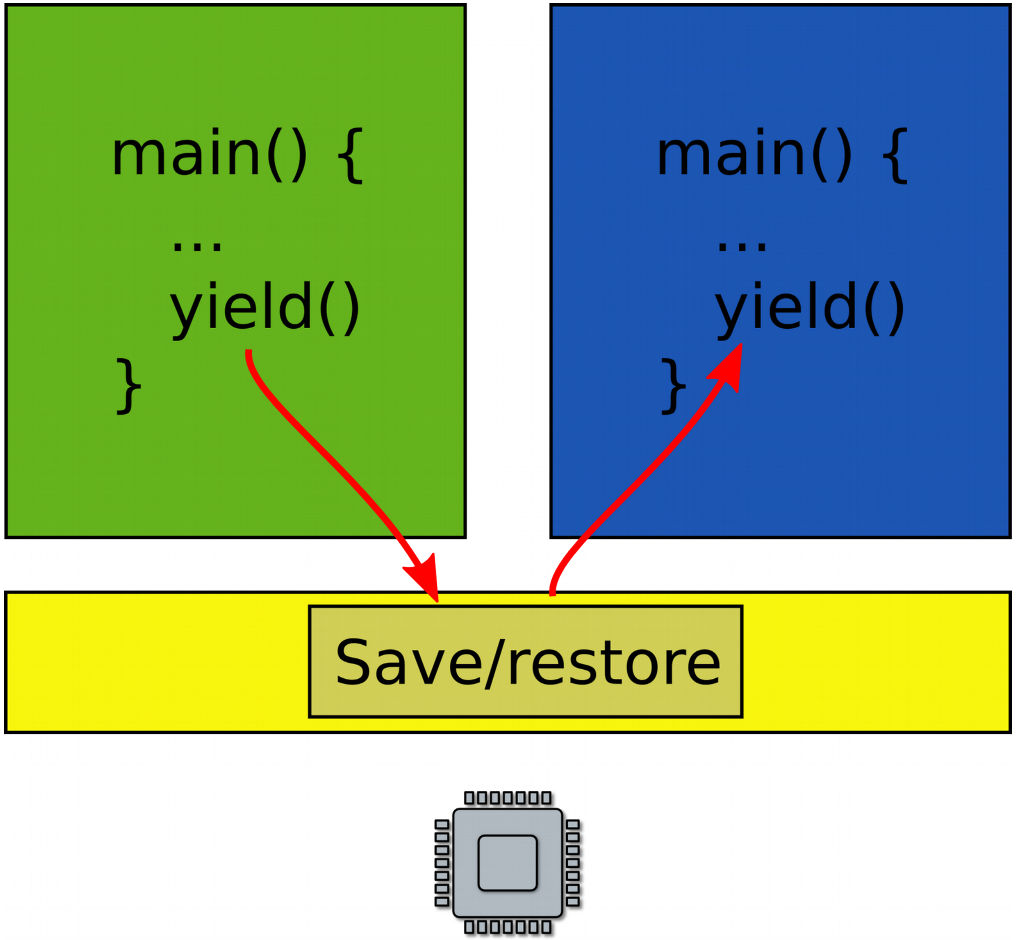
- Result:
    - EAX = 55

But why do we need page tables

… Instead of arrays?

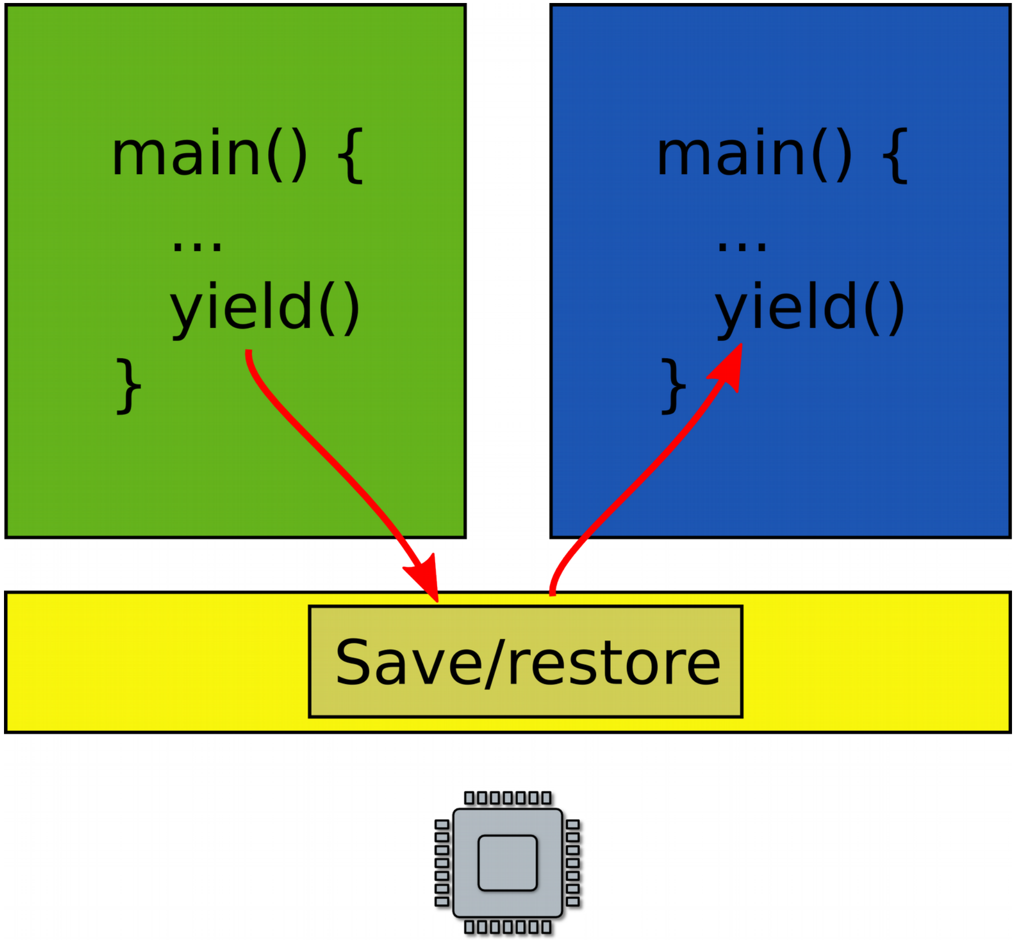
- Page tables represent sparse address space more efficiently
  - An entire array has to be allocated upfront
  - But if the address space uses a handful of pages
  - Only page tables (Level 1 and 2 need to be allocated to describe translation)
- On a dense address space this benefit goes away
  - I'll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

What about isolation?



- Two programs, one memory?

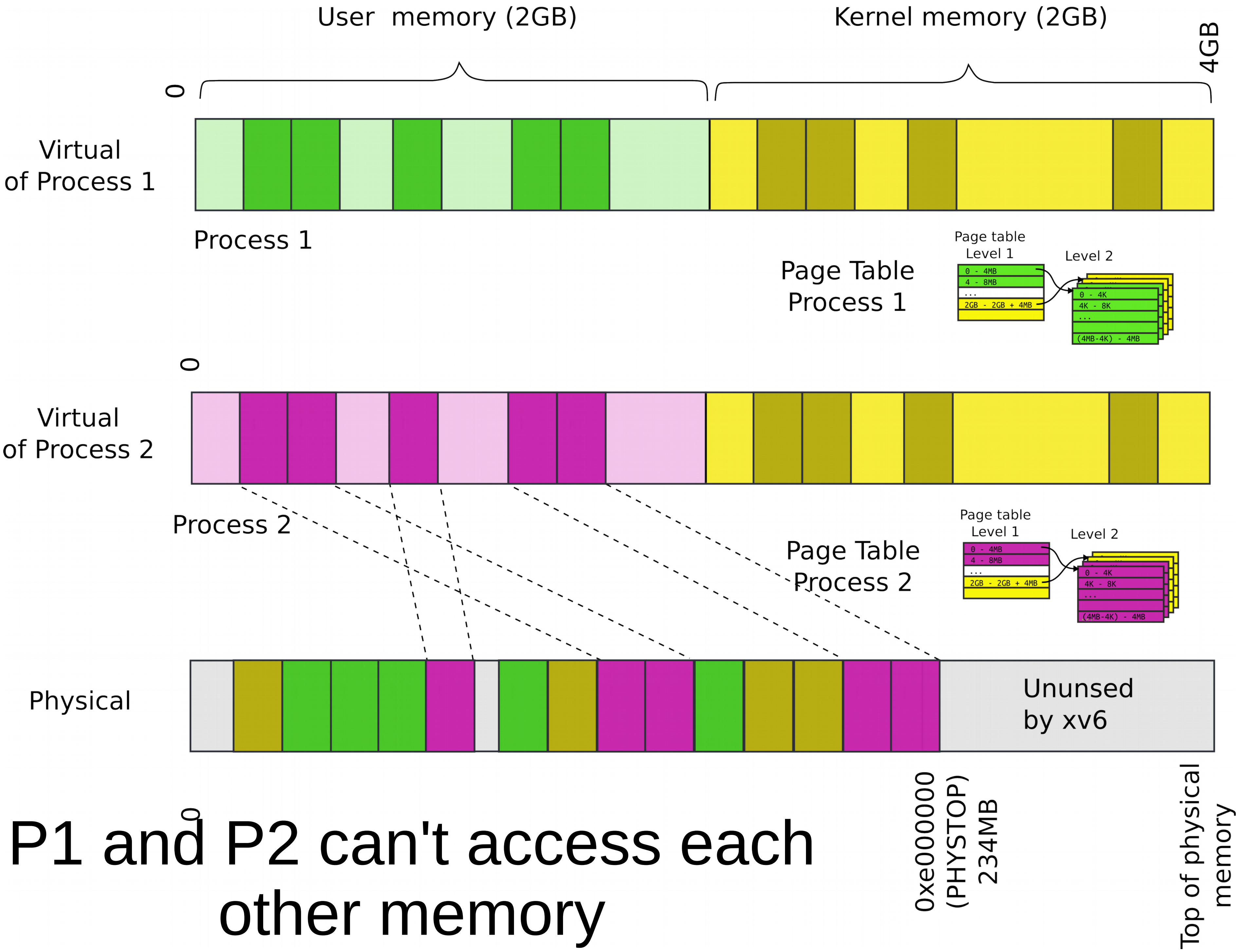
What about isolation?



```
main() {
  ...
  yield()
}
```

```
main() {
  ...
  yield()
}
```

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



User memory (2GB)

Kernel memory (2GB)

0

4GB

Virtual
of Process 1

Process 1

Page Table
Process 1

Page table
Level 1

0 - 4MB
4 - 8MB
...
2GB - 2GB + 4MB

Level 2

0 - 4K
4K - 8K
...
(4MB-4K) - 4MB

0

Virtual
of Process 2

Process 2

Page Table
Process 2

Page table
Level 1

0 - 4MB
4 - 8MB
...
2GB - 2GB + 4MB

Level 2

0 - 4K
4K - 8K
...
(4MB-4K) - 4MB

Physical

Unused
by xv6

0

0xe000000
(PHYSTOP)
234MB

Top of physical
memory

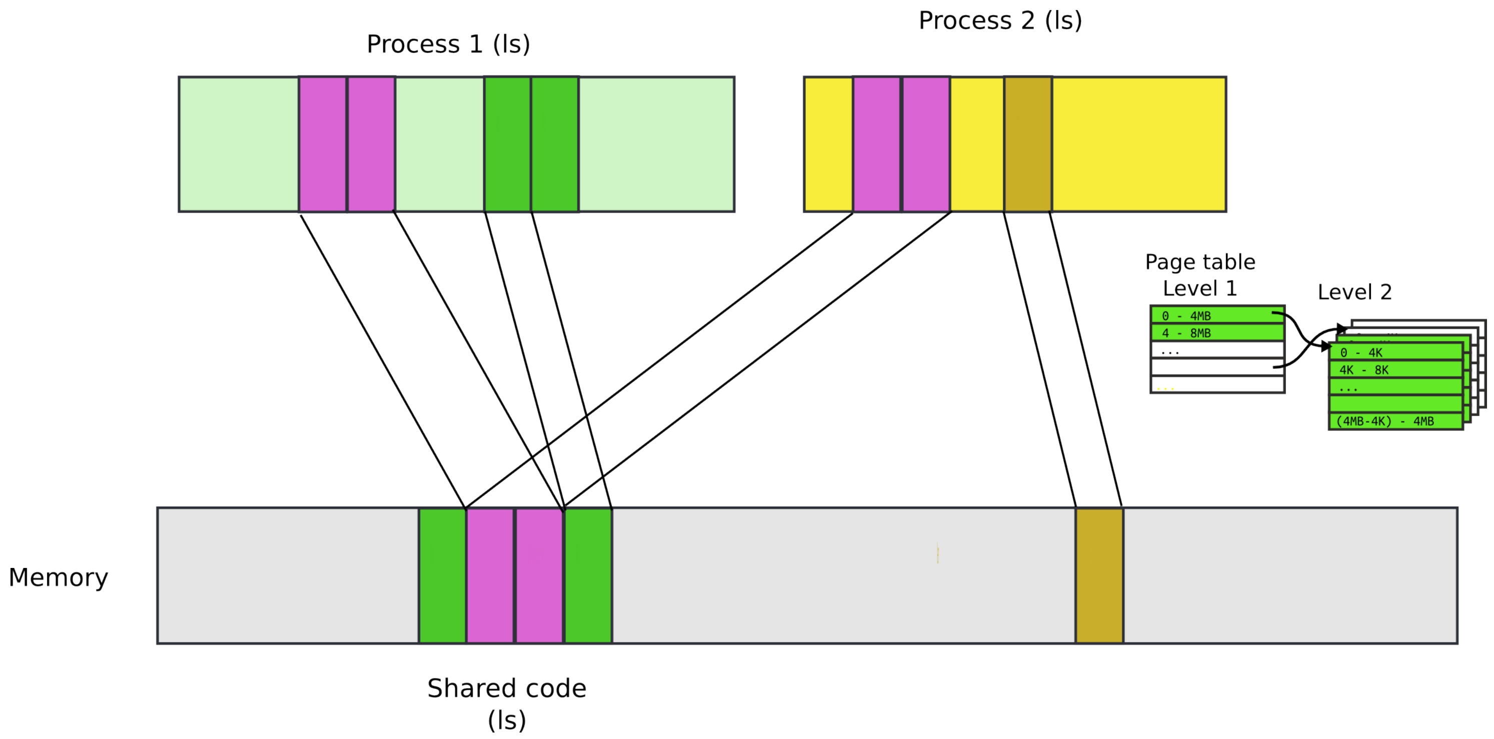
P1 and P2 can't access each other memory

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 ...

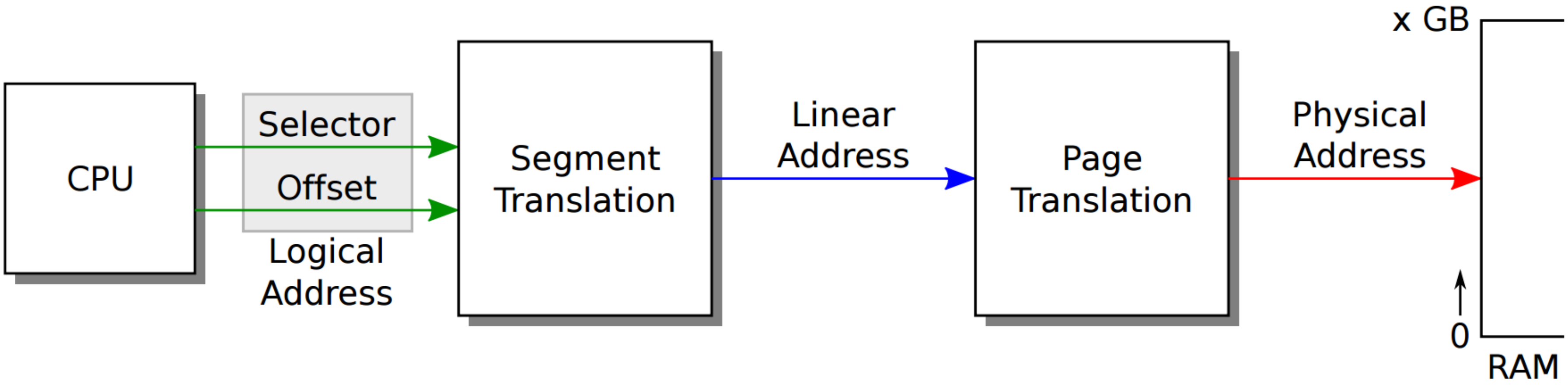
- Share a region of memory across multiple programs
  - Communication (shared buffer of messages)
  - Shared libraries

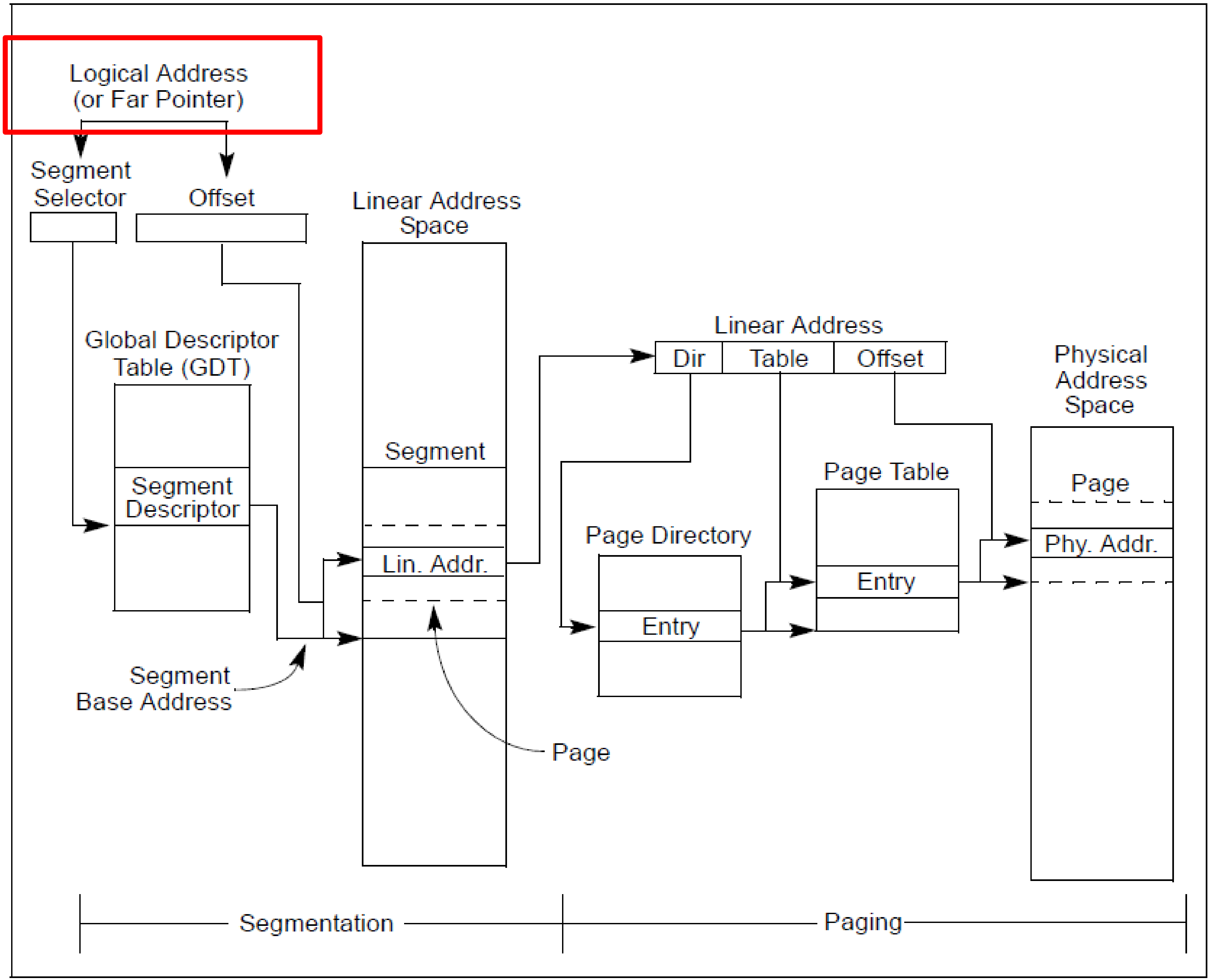


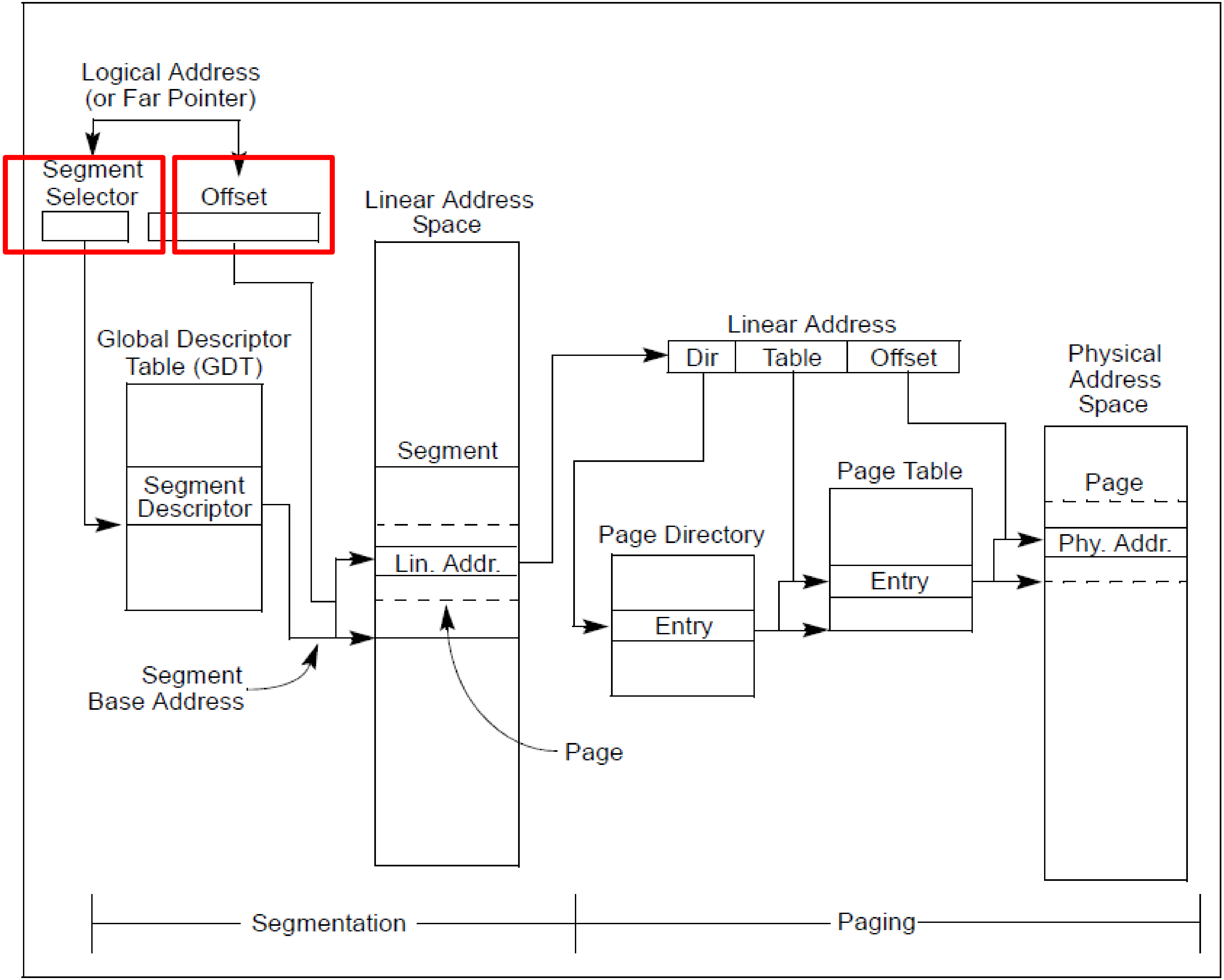
More paging tricks

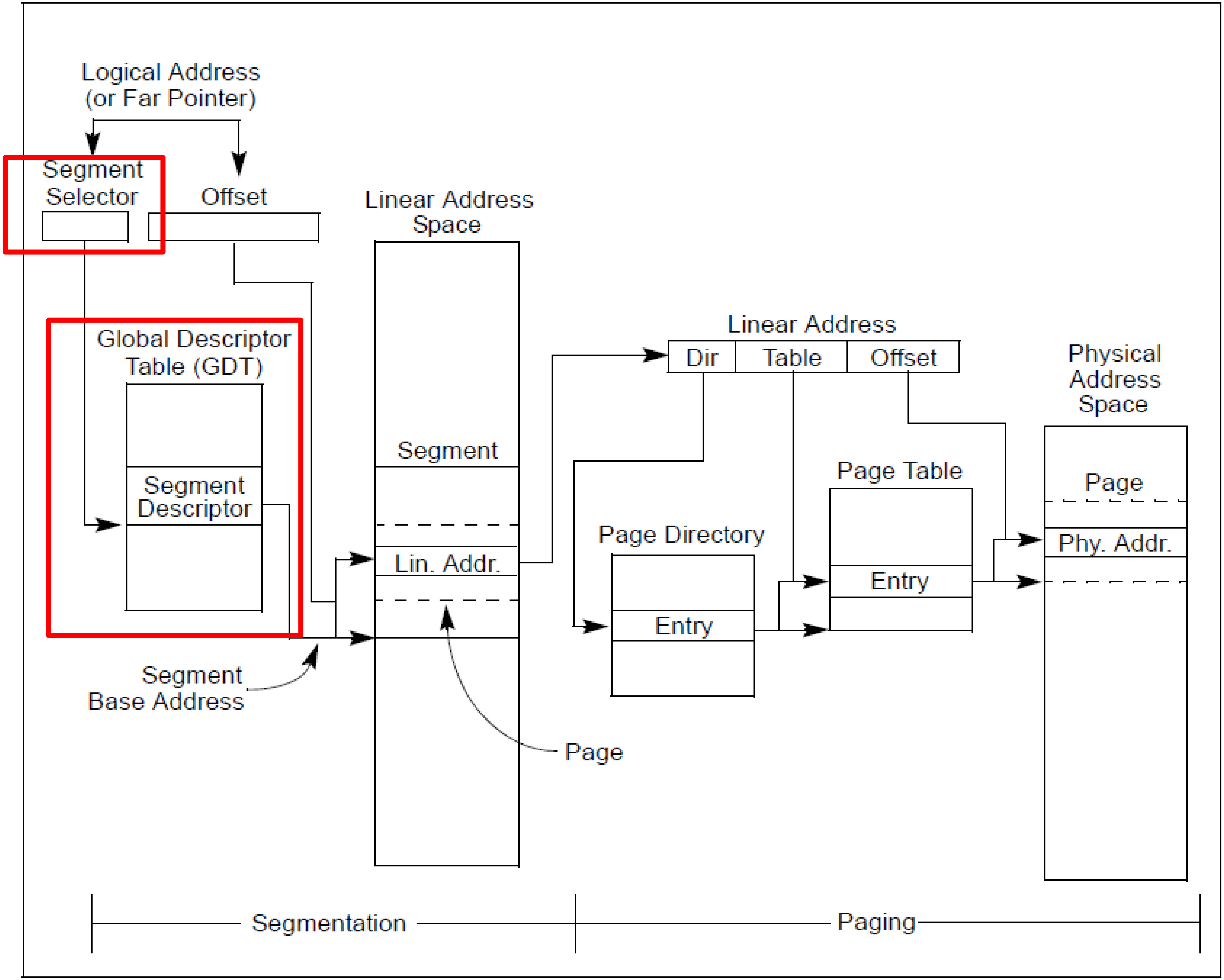
- Protect parts of the program
  - E.g., map code as read-only
    - Disable code modification attacks
    - Remember R/W bit in PTD/PTE entries!
  - E.g., map stack as non-executable
    - Protects from stack smashing attacks
    - Non-executable bit

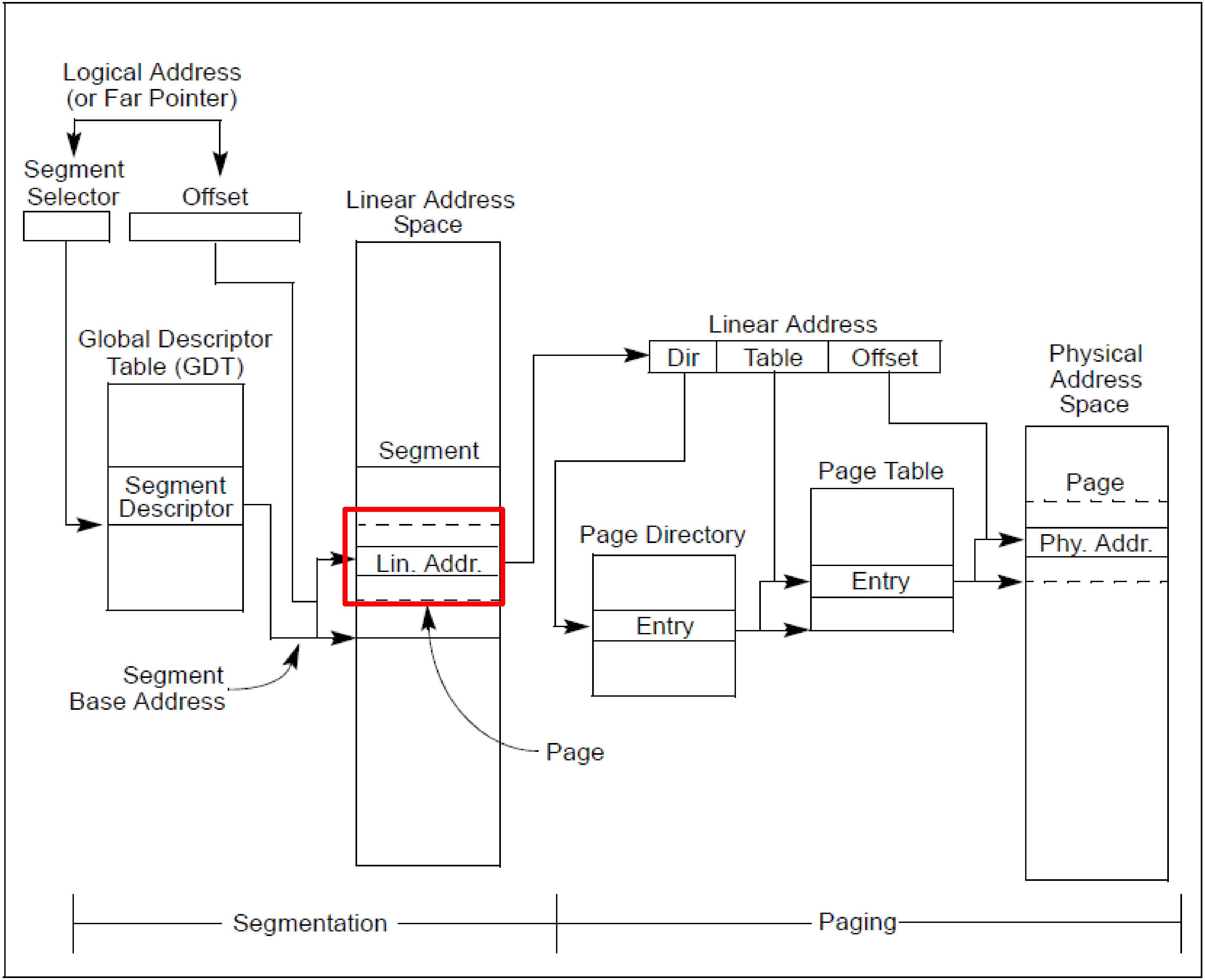
Recap: complete address translation

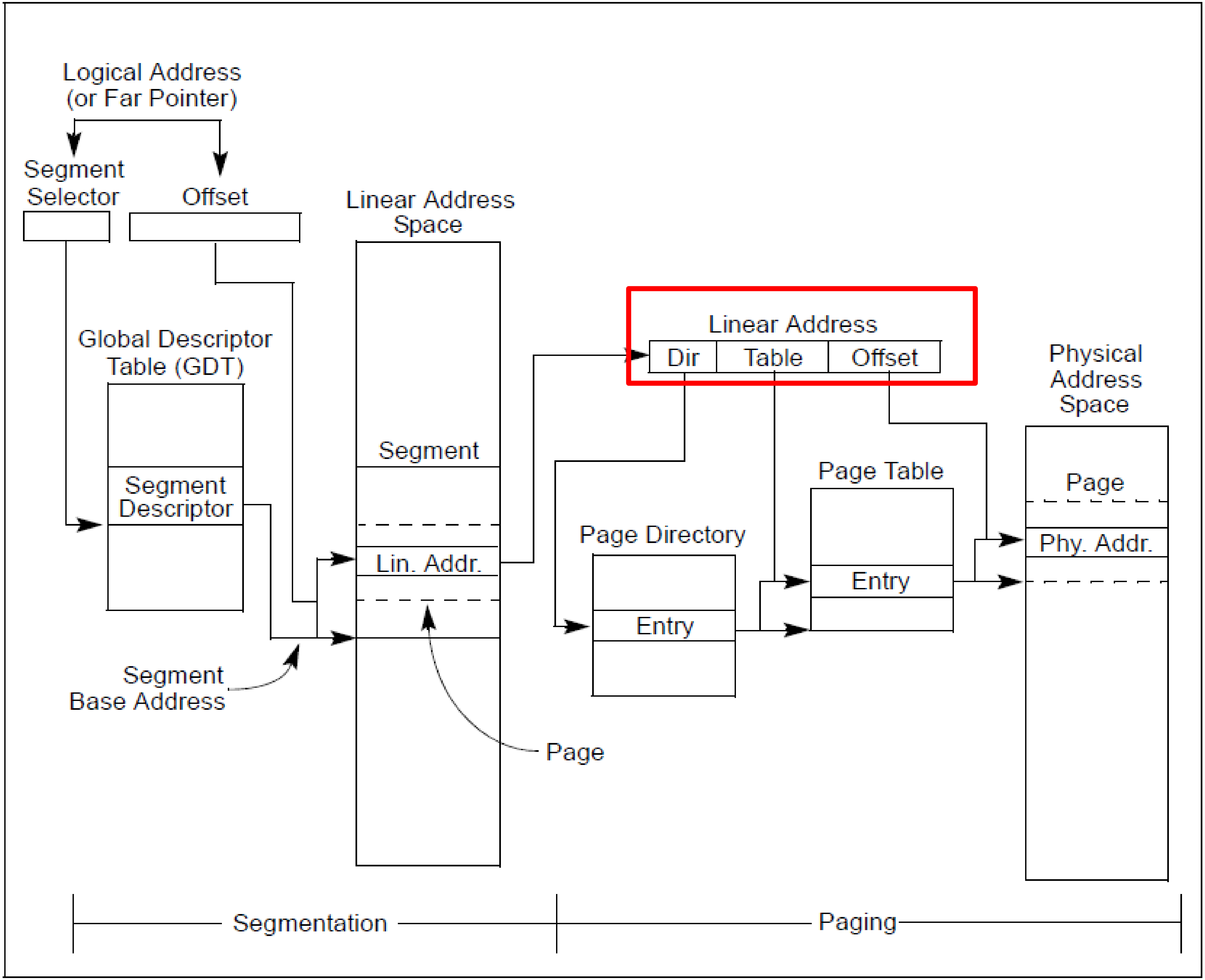


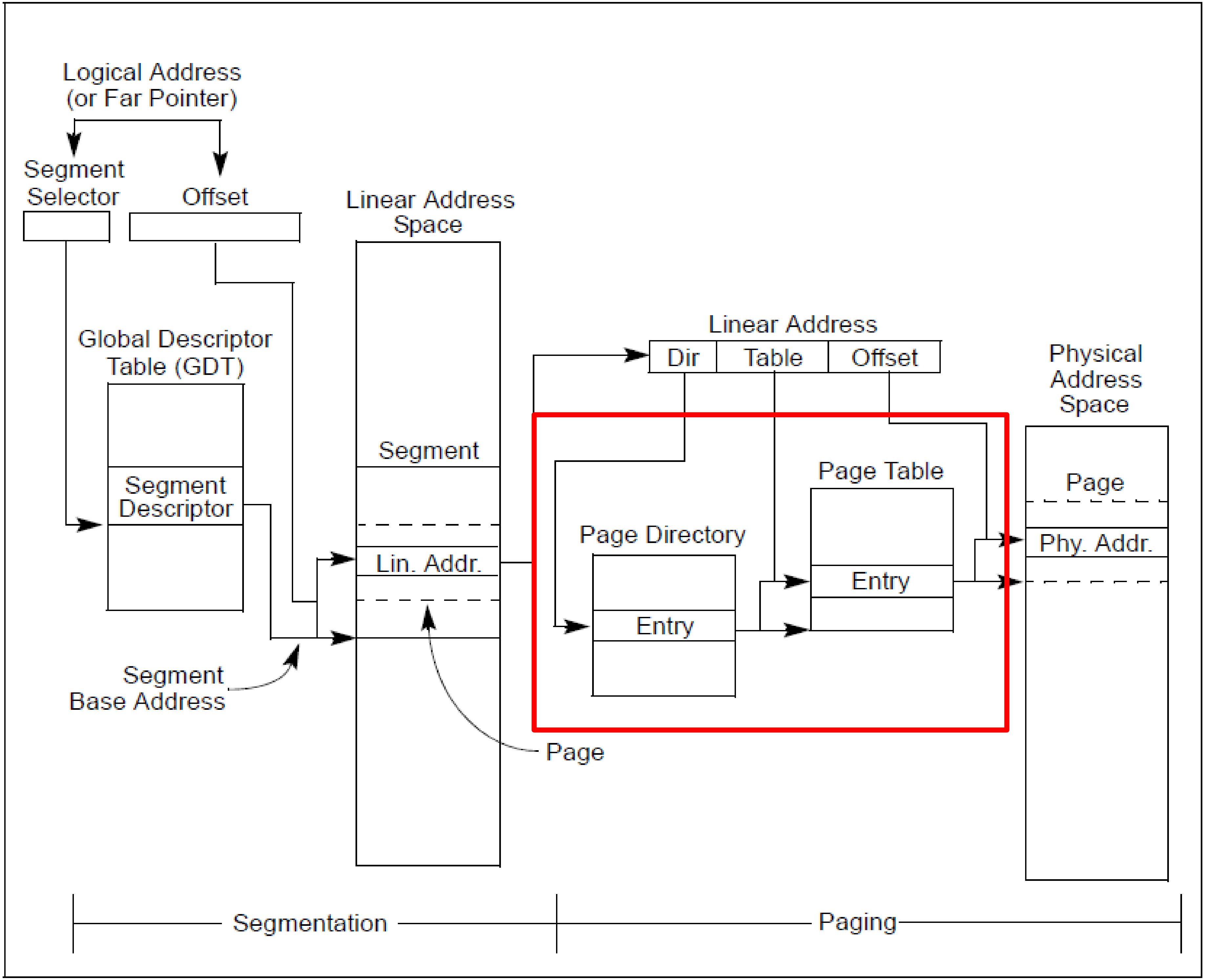

Logical Address
(or Far Pointer)
Segment
Selector
Offset
Linear Address
Space
Global Descriptor
Table (GDT)
Segment
Descriptor
Segment
Lin. Addr.
Segment
Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical
Address
Space
Page
Phy. Addr.
Segmentation
Paging

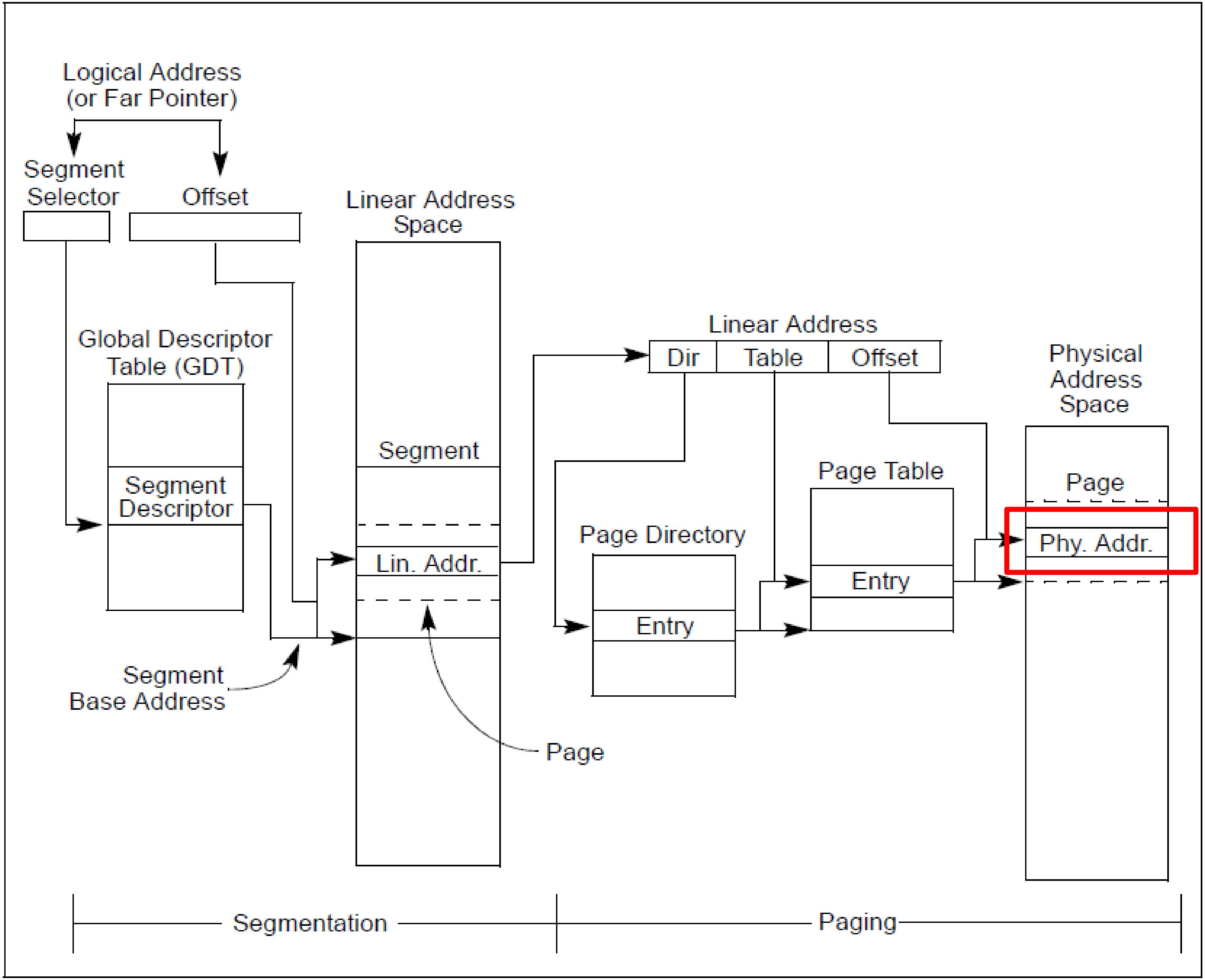

Logical Address
(or Far Pointer)
Segment
Selector
Offset
Linear Address
Space
Global Descriptor
Table (GDT)
Segment
Descriptor
Segment
Lin. Addr.
Segment
Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical
Address
Space
Page
Phy. Addr.
Segmentation
Paging

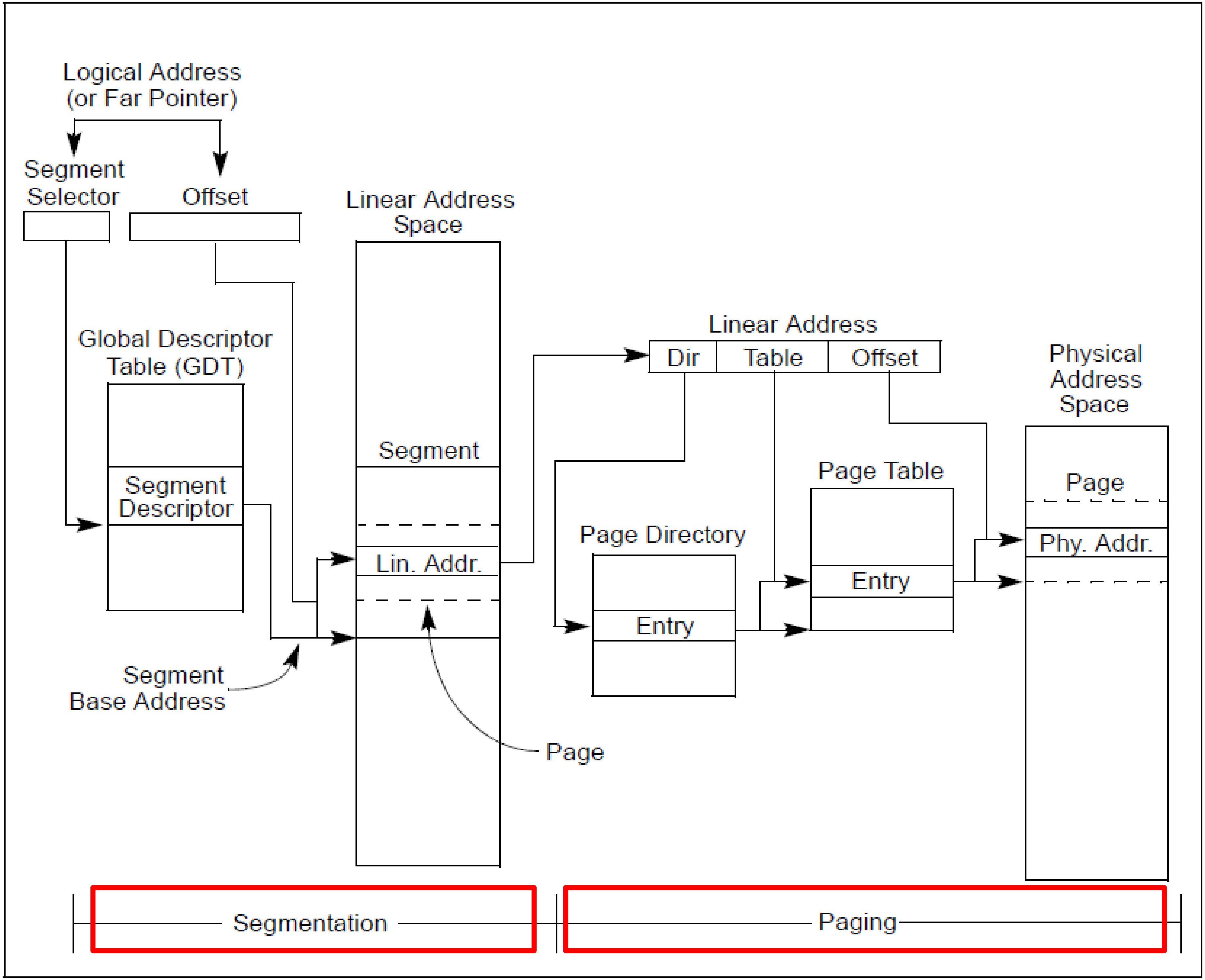

Logical Address
(or Far Pointer)
Segment Selector
Offset
Linear Address Space
Global Descriptor Table (GDT)
Segment Descriptor
Segment
Lin. Addr.
Segment Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical Address Space
Page
Phy. Addr.
Segmentation
Paging


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
Physical
Address
Space
Page Table
Page Directory
Entry
Entry
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

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

Questions?