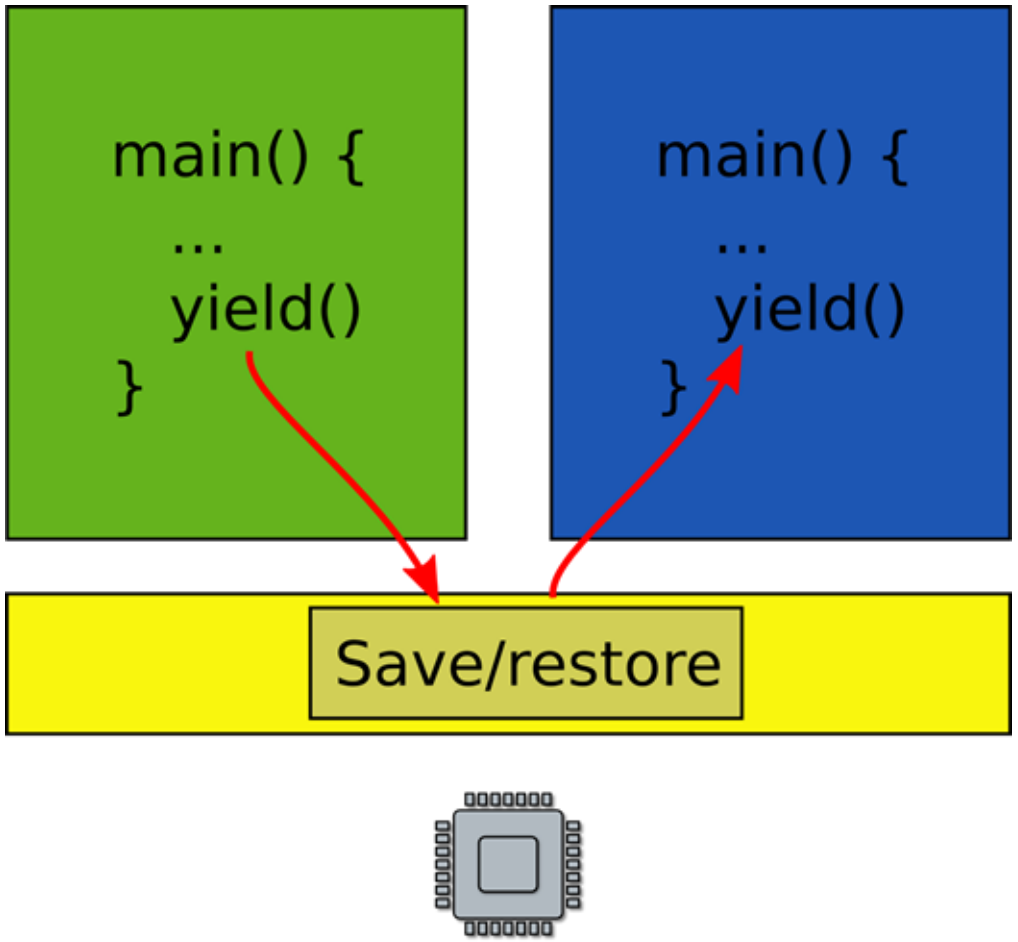
cs5460/6460: Operating Systems

Address translation (Segmentation and Paging)

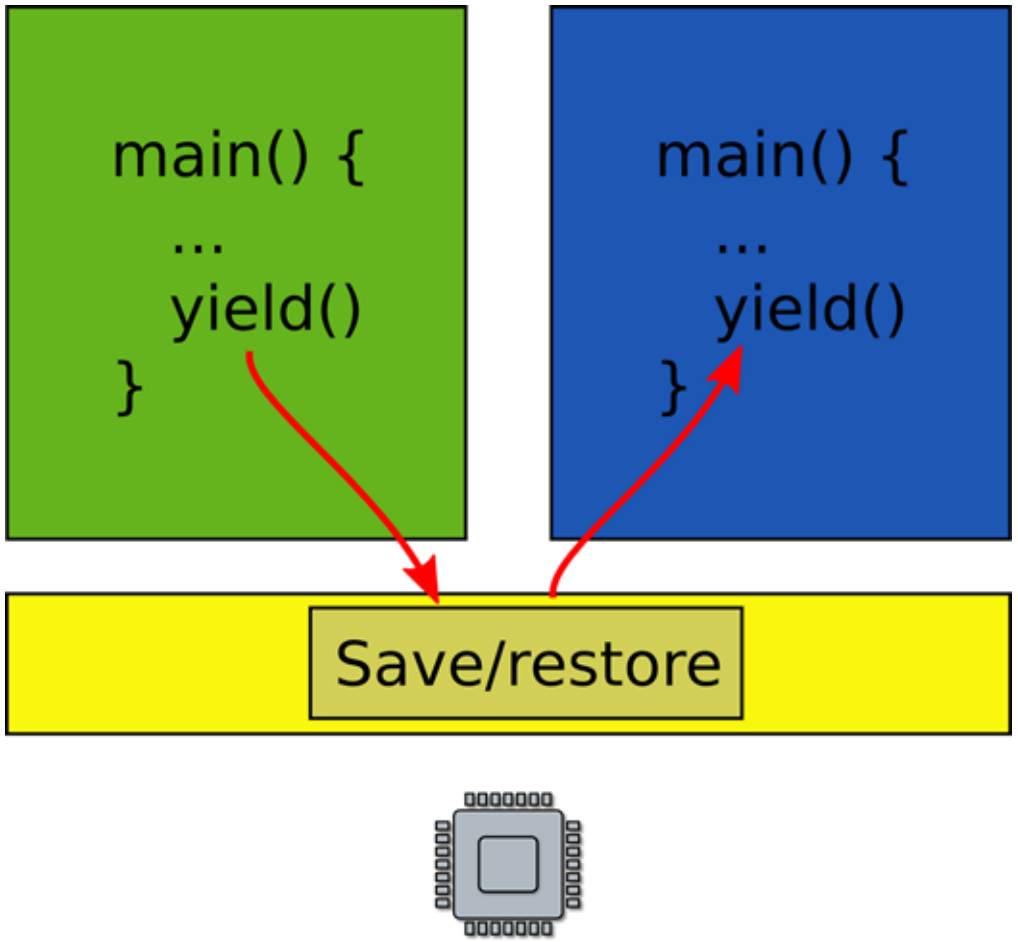
Anton Burtsev

February, 2026

Two programs one memory



Two programs one memory

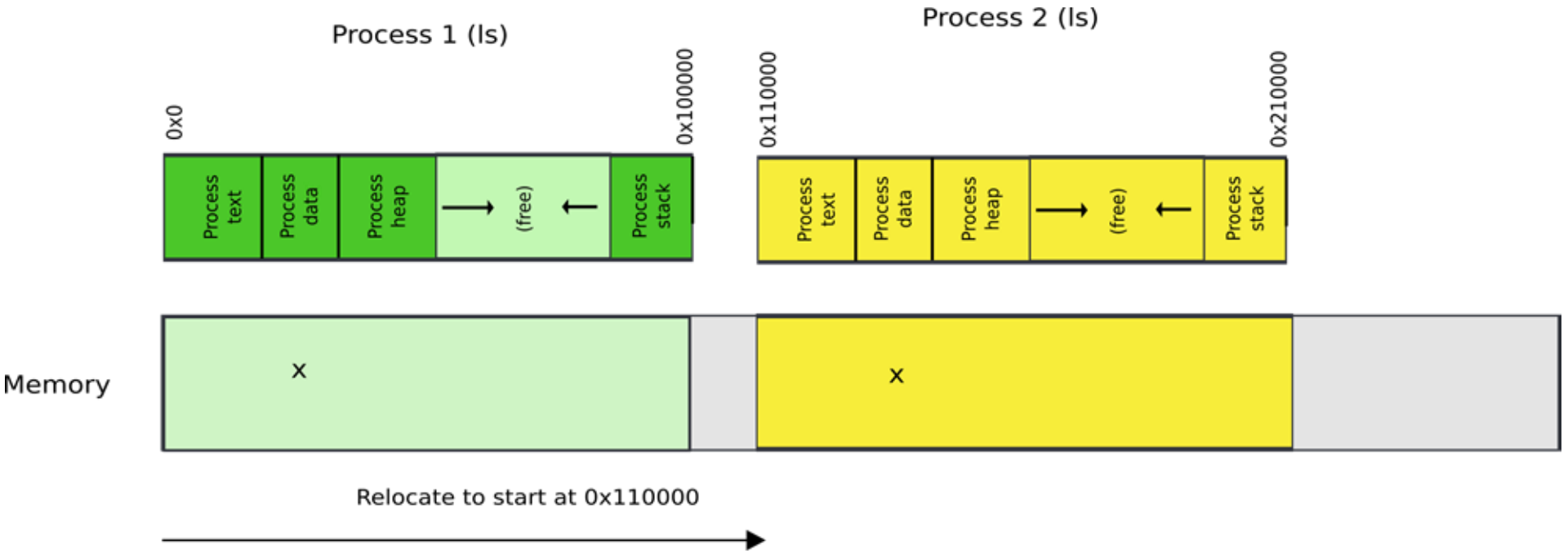


- How can we do this?

Relocation

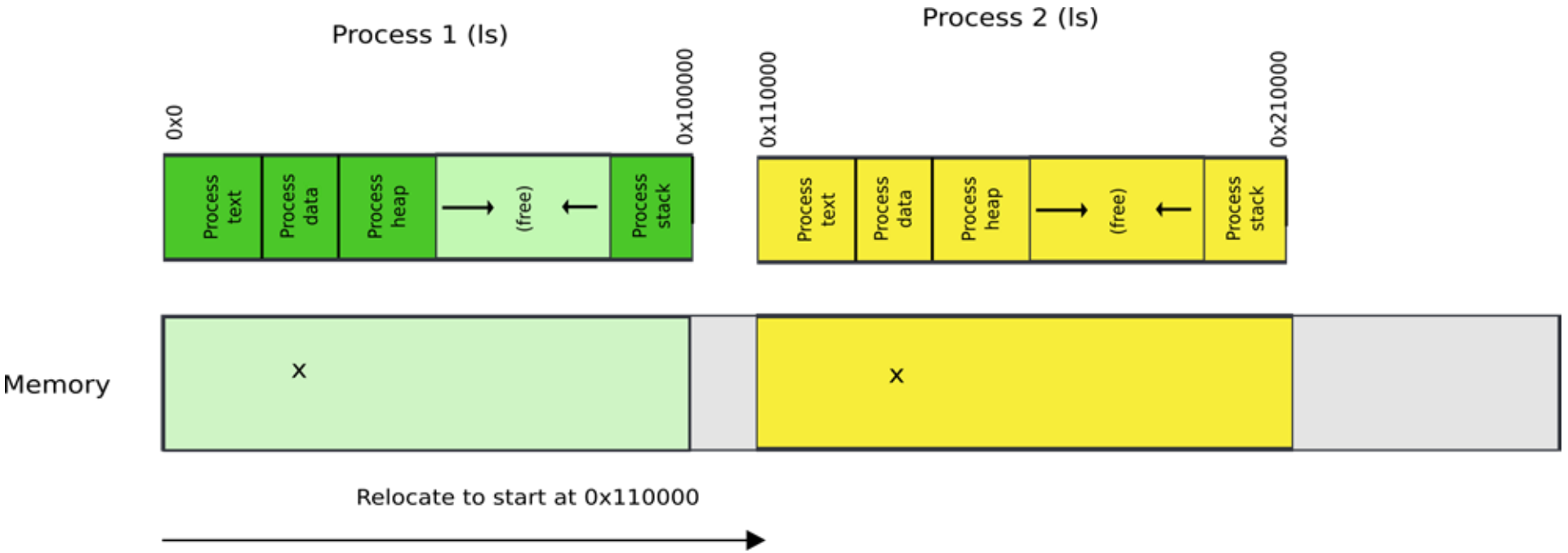
- One way to achieve this is to relocate program at different addresses
- Remember relocation?

Relocate binaries to work at different addresses



- One way to achieve this is to relocate program at different addresses
  - Remember relocation?
- This works! But not ideal
- What is the problem?

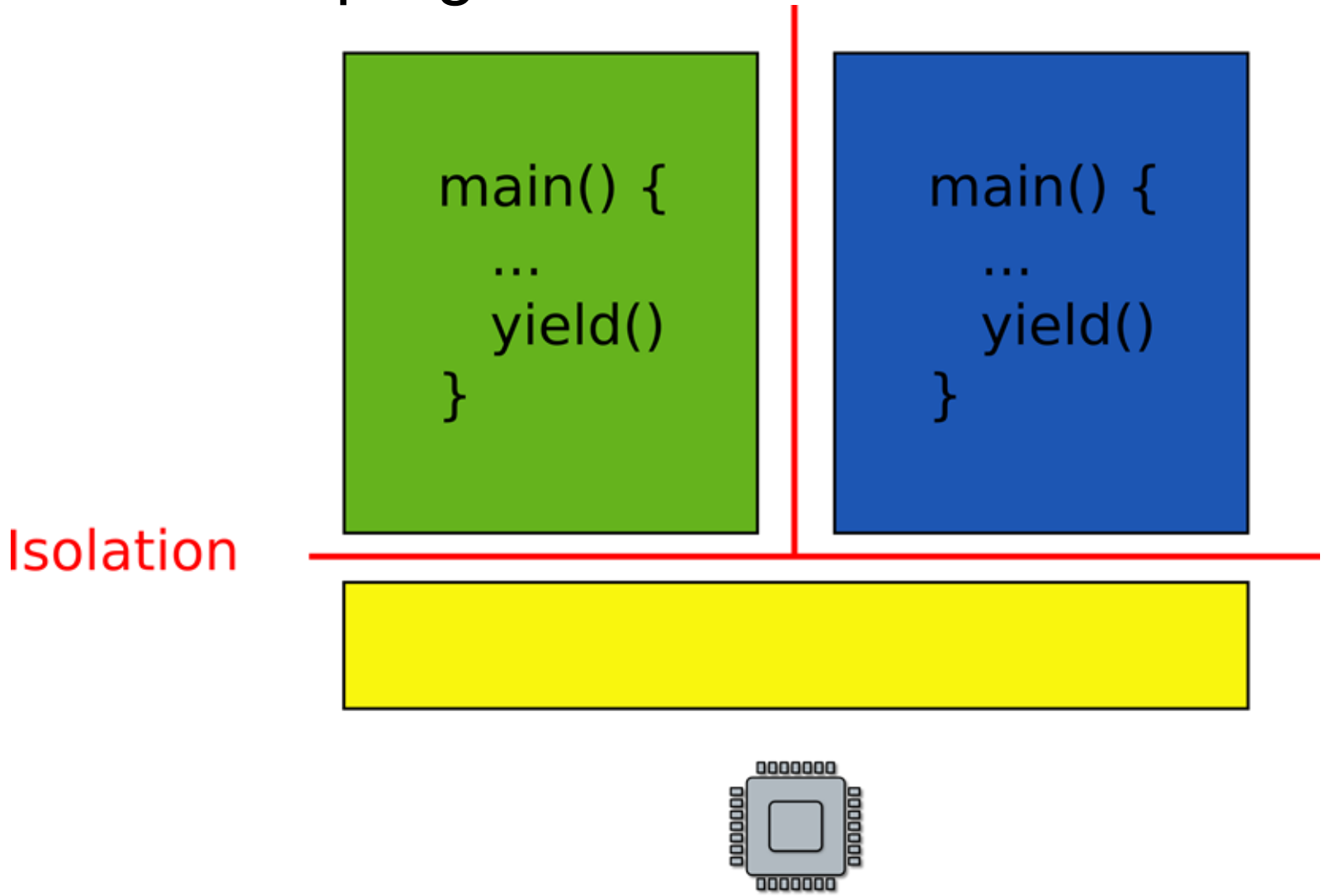
Relocate binaries to work at different addresses



- What is the problem?

Isolation

- What if one faulty program corrupts the kernel?
- Or other programs?



Problem: isolation

- How can we enforce isolation?

Problem: isolation

- How can we enforce isolation?
- Isolation can be enforced in software
- Software Fault Isolation (SFI)
  - Google NaCl (Chrome Sandbox)
  - WASM (Web Assembly, another sandbox standard)

Actually, how?

```
#include <stdio.h>

int main(int ac, char **av)
{
   int a = 5, b = 6;
   return a + b;
}

00000000 <main>:
  0:   55                    push   ebp
  1:   89 e5                 mov    ebp,esp
  3:   83 ec 10              sub    esp,0x10
  6:   c7 45 f8 05 00 00 00  mov    DWORD PTR [ebp-0x8],0x5
  d:   c7 45 fc 06 00 00 00  mov    DWORD PTR [ebp-0x4],0x6
 14:   8b 45 fc              mov    eax,DWORD PTR [ebp-0x4]
 17:   8b 55 f8              mov    edx,DWORD PTR [ebp-0x8]
 1a:   01 d0                 add    eax,edx
 1c:   c9                    leave
 1d:   c3                    ret
```

PollEv.com/antonburtsev

```
#include <stdio.h>

int main(int ac, char **av)
{
   int a = 5, b = 6;
   return a + b;
}

00000000 <main>:
  0:   55                    push   ebp
  1:   89 e5                 mov    ebp,esp
  3:   83 ec 10              sub    esp,0x10
  6:   c7 45 f8 05 00 00 00  mov    DWORD PTR [ebp-0x8],0x5
  d:   c7 45 fc 06 00 00 00  mov    DWORD PTR [ebp-0x4],0x6
 14:   8b 45 fc              mov    eax,DWORD PTR [ebp-0x4]
 17:   8b 55 f8              mov    edx,DWORD PTR [ebp-0x8]
 1a:   01 d0                 add    eax,edx
 1c:   c9                    leave
 1d:   c3                    ret
```

Detour: Software Fault Isolation

Will appear in the 2009 IEEE Symposium on Security and Privacy

Native Client: A Sandbox for Portable, Untrusted x86 Native Code

Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth,
Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar
Google Inc.

Abstract

This paper describes the design, implementation and evaluation of Native Client, a sandbox for untrusted x86 native code. Native Client aims to give browser-based applications the computational performance of native applications without compromising safety. Native Client uses software fault isolation and a secure runtime to direct system interaction and side effects through interfaces managed by Native Client. Native Client provides operating system portability for binary code while supporting performance-oriented features generally absent from web application programming environments, such as thread support, instruction set extensions such as SSE, and use of compiler intrinsics and hand-coded assembler. We combine these properties in an open architecture that encourages community review and 3rd-party tools.

1. Introduction

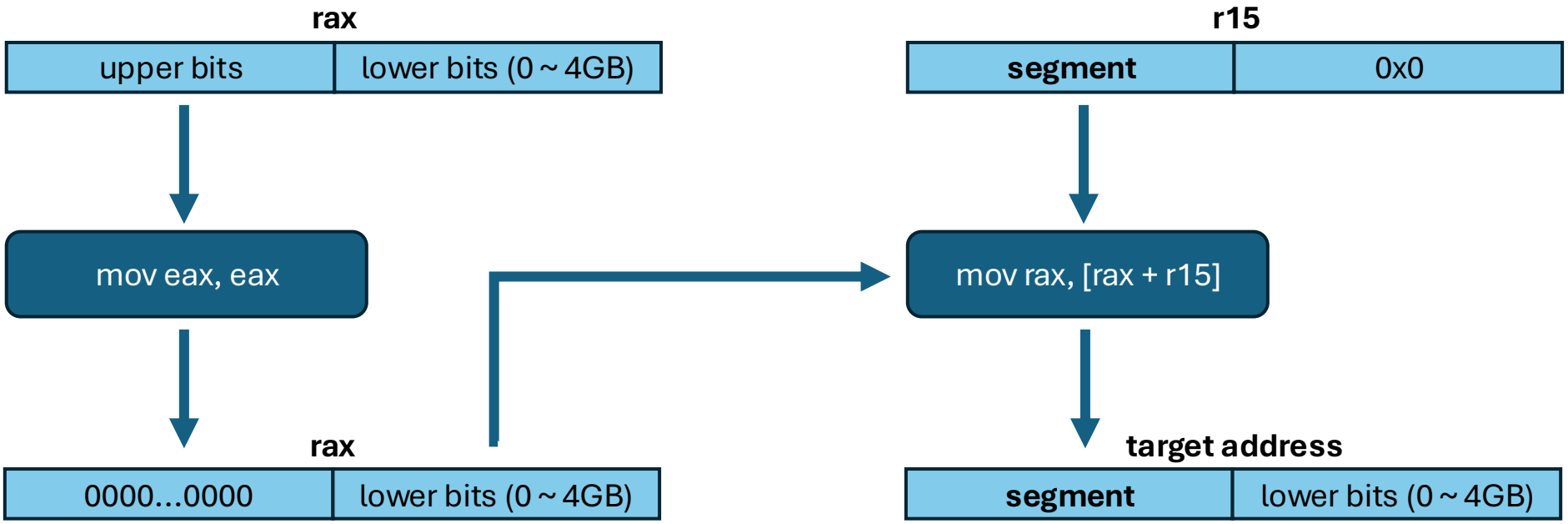
As an application platform, the modern web browser brings together a remarkable combination of resources, including seamless access to Internet resources, high

as a secondary consideration. Given this organization, and the absence of effective technical measures to constrain these plugins, browser applications that wish to use native-code must rely on non-technical measures for security; for example, manual establishment of trust relationships through pop-up dialog boxes, or manual installation of a console application. Historically, these non-technical measures have been inadequate to prevent execution of malicious native code, leading to inconvenience and economic harm [10], [54]. As a consequence we believe there is a prejudice against native code extensions for browser-based applications among experts and distrust among the larger population of computer users.

While acknowledging the insecurity of the current systems for incorporating native-code into web applications, we also observe that there is no fundamental reason why native code should be unsafe. In Native Client, we separate the problem of safe native execution from that of extending trust, allowing each to be managed independently. Conceptually, Native Client is organized in two parts: a constrained execution environment for native code to prevent unintended side effects, and a runtime for hosting these native code extensions through which allowable side effects may occur

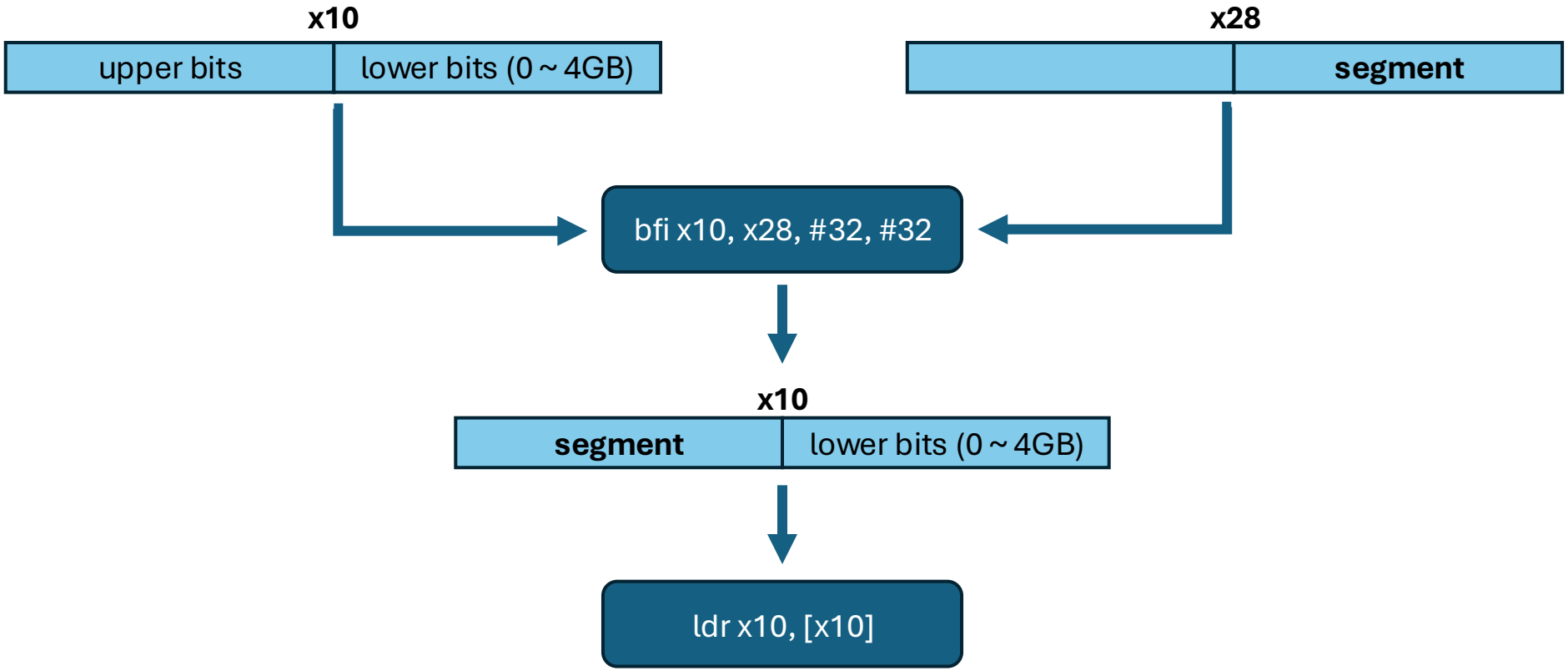
https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/34913.pdf

Short primer on SFI (x86 NaCl, Wasm)

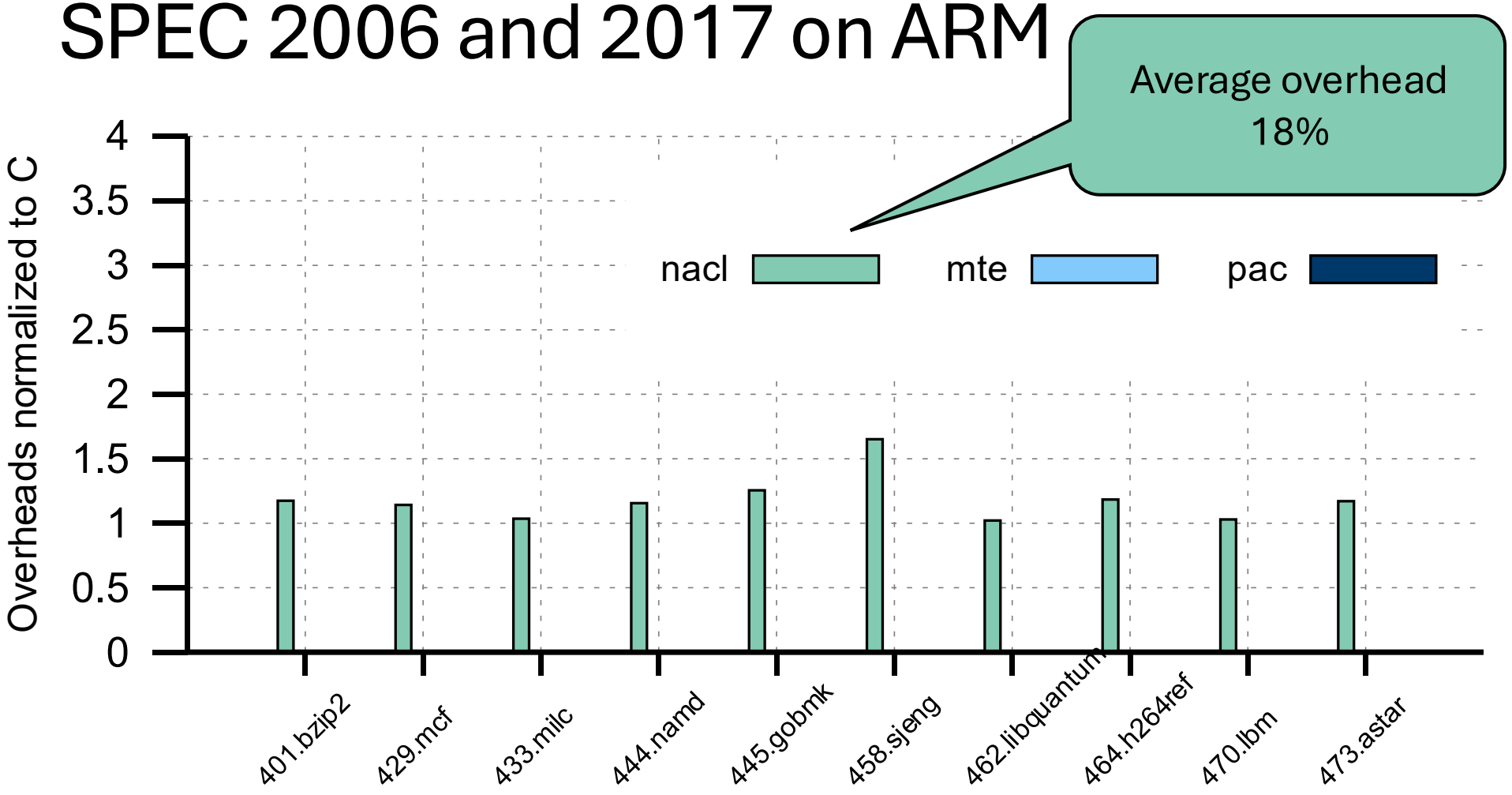



```
1  ; load value at [rax] to rcx
2  mov eax, eax ; eax contains 0-4GB
3  mov rcx, [r15, rax, 1] ;memory access within [r15 + 0-4GB]
```

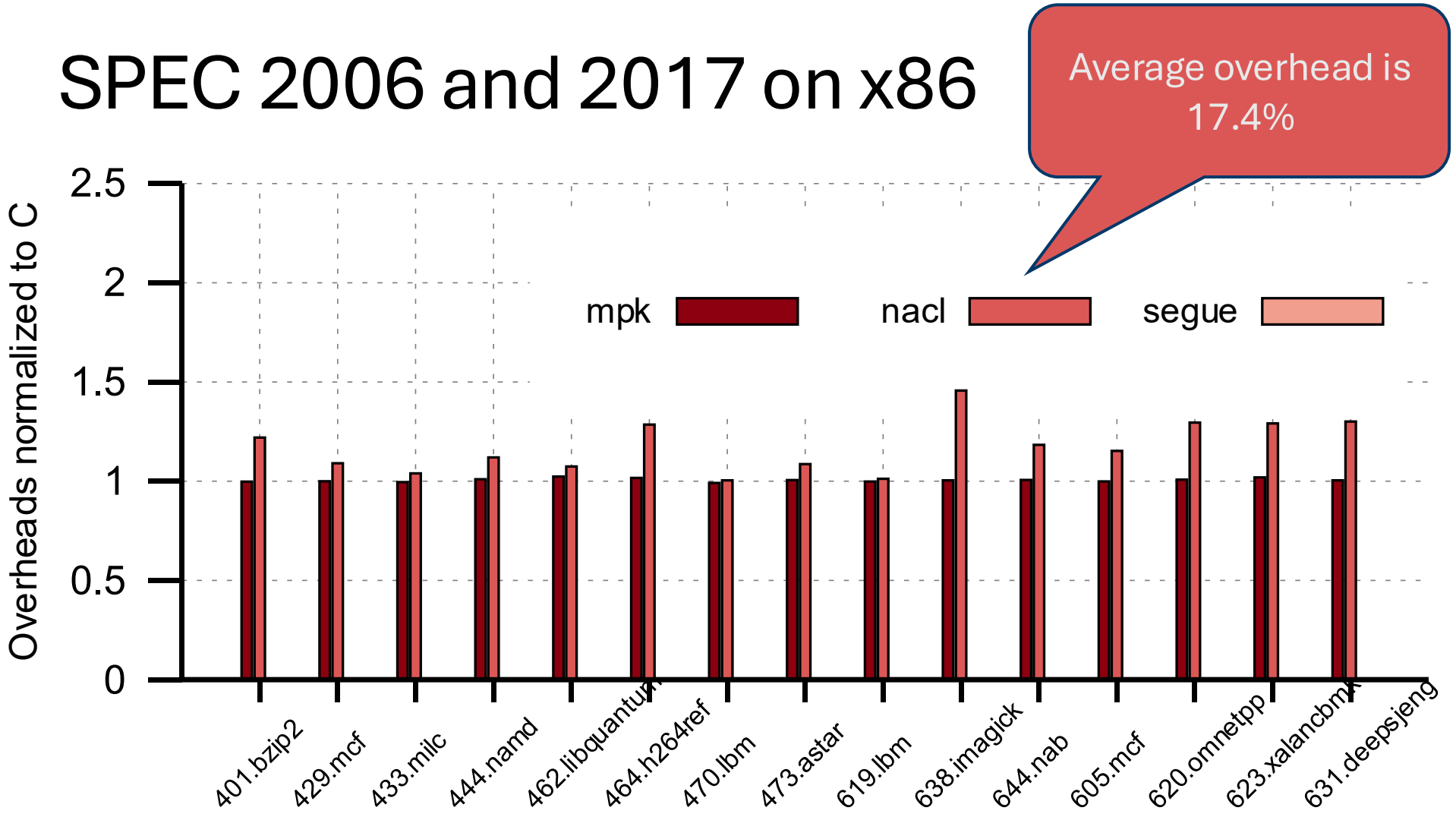
SFI on ARM


x10
upper bits
lower bits (0 ~ 4GB)
x28
segment
bfi x10, x28, #32, #32
x10
segment
lower bits (0 ~ 4GB)
ldr x10, [x10]

SPEC 2006 and 2017 on ARM



SPEC 2006 and 2017 on x86



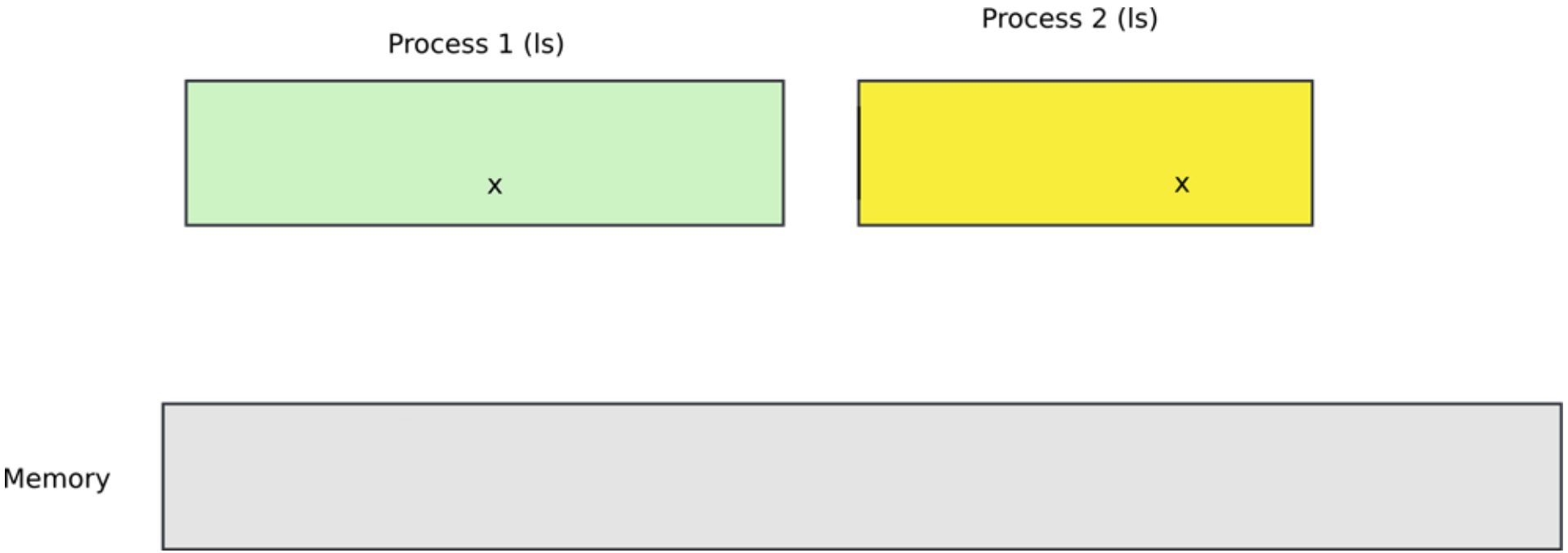
Another way is to ask for hardware support

Segmentation

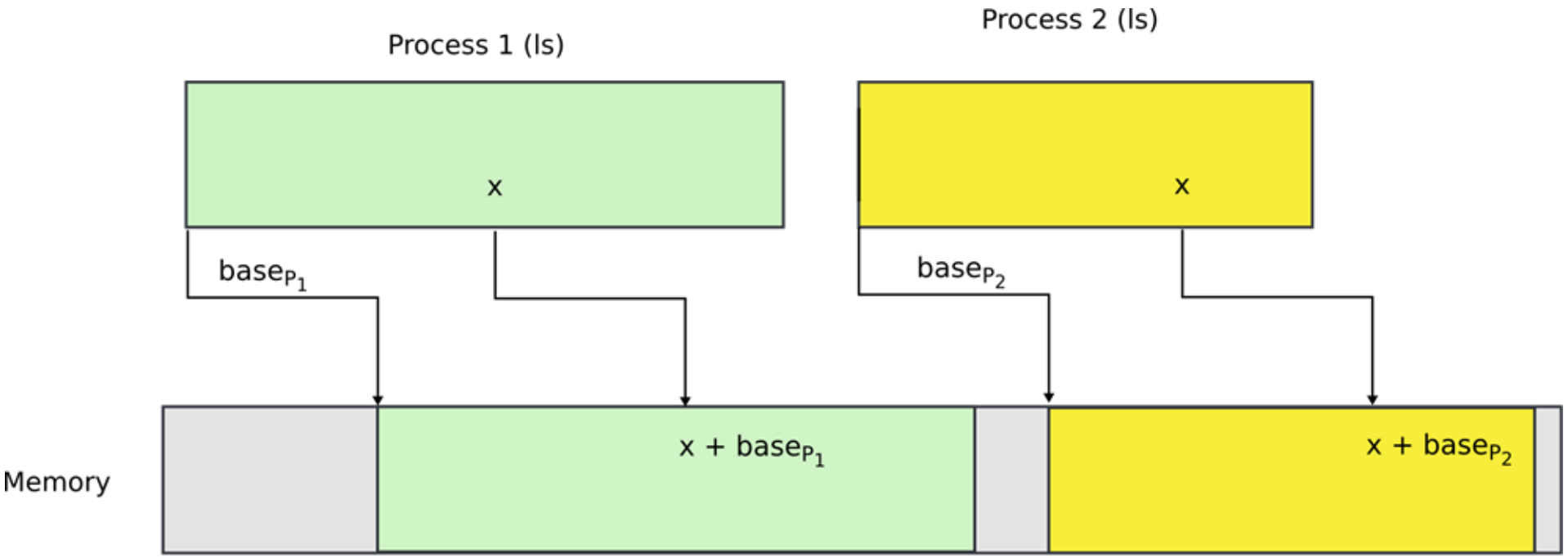
What are we aiming for?

- Illusion of a private address space
- Identical copy of an address space in multiple programs
- 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

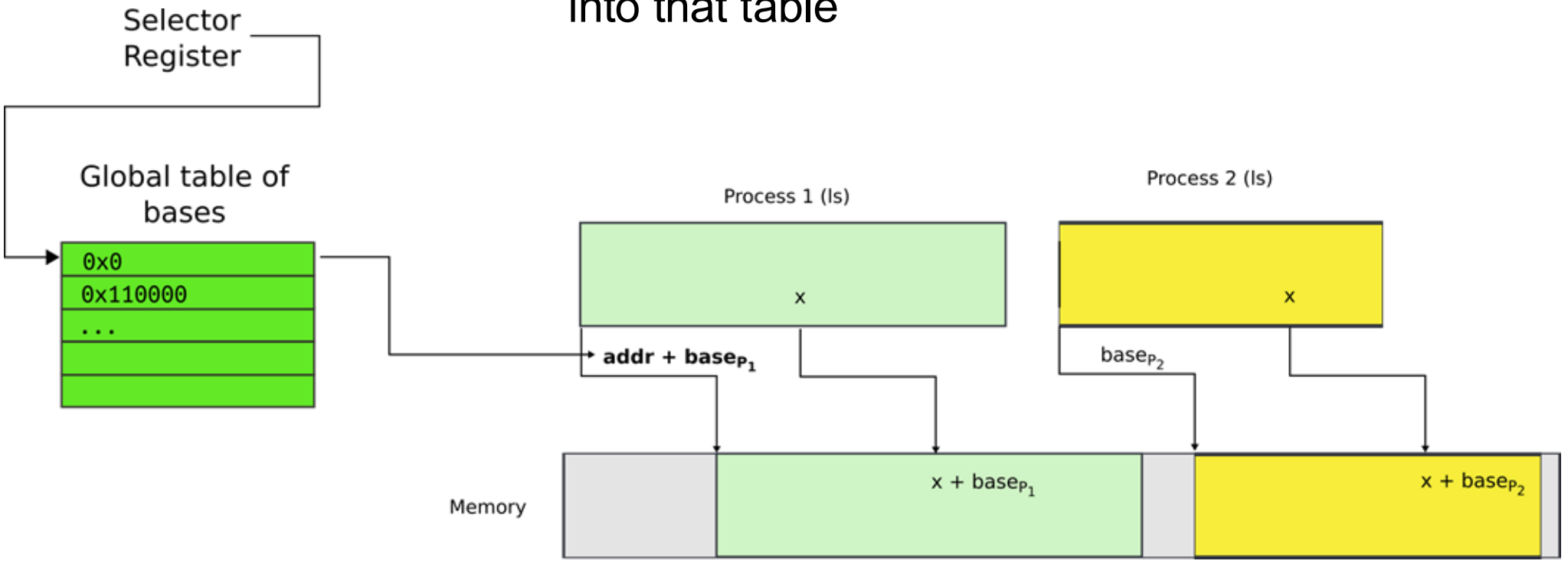
Seems easy

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

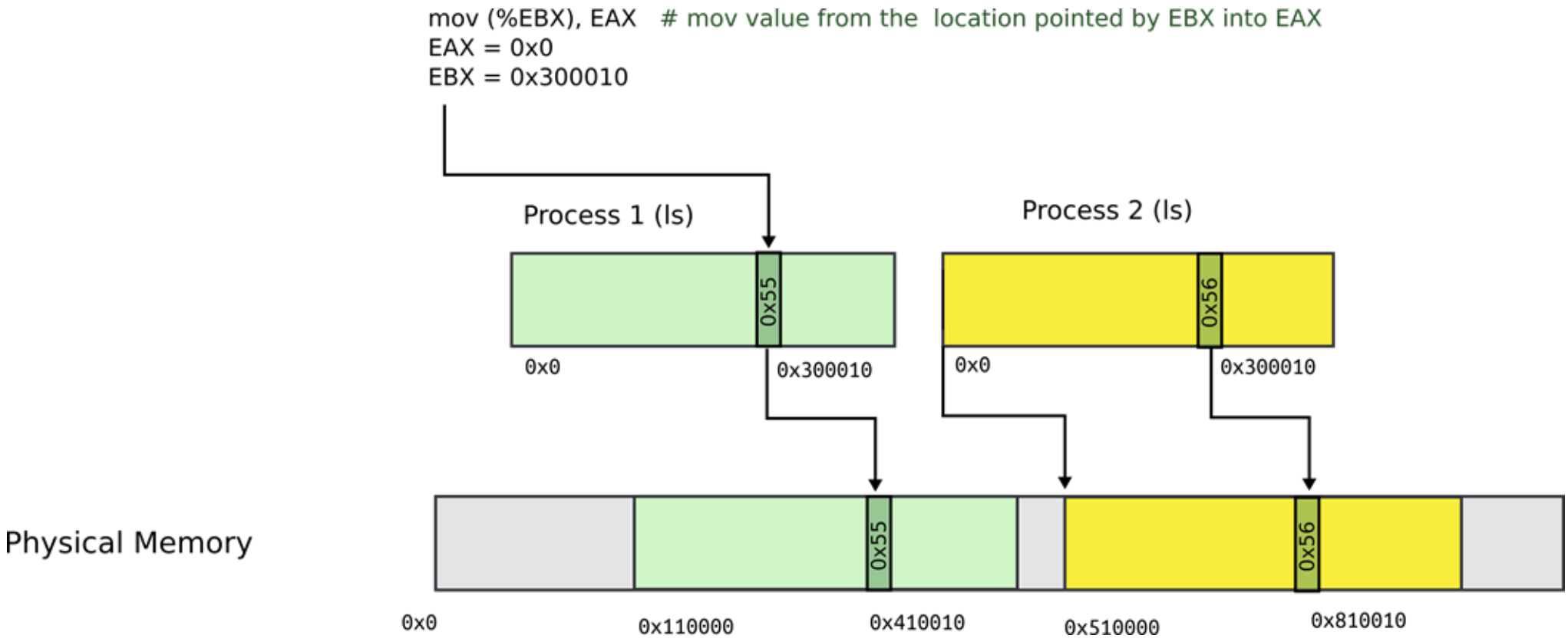
Seems easy

- 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

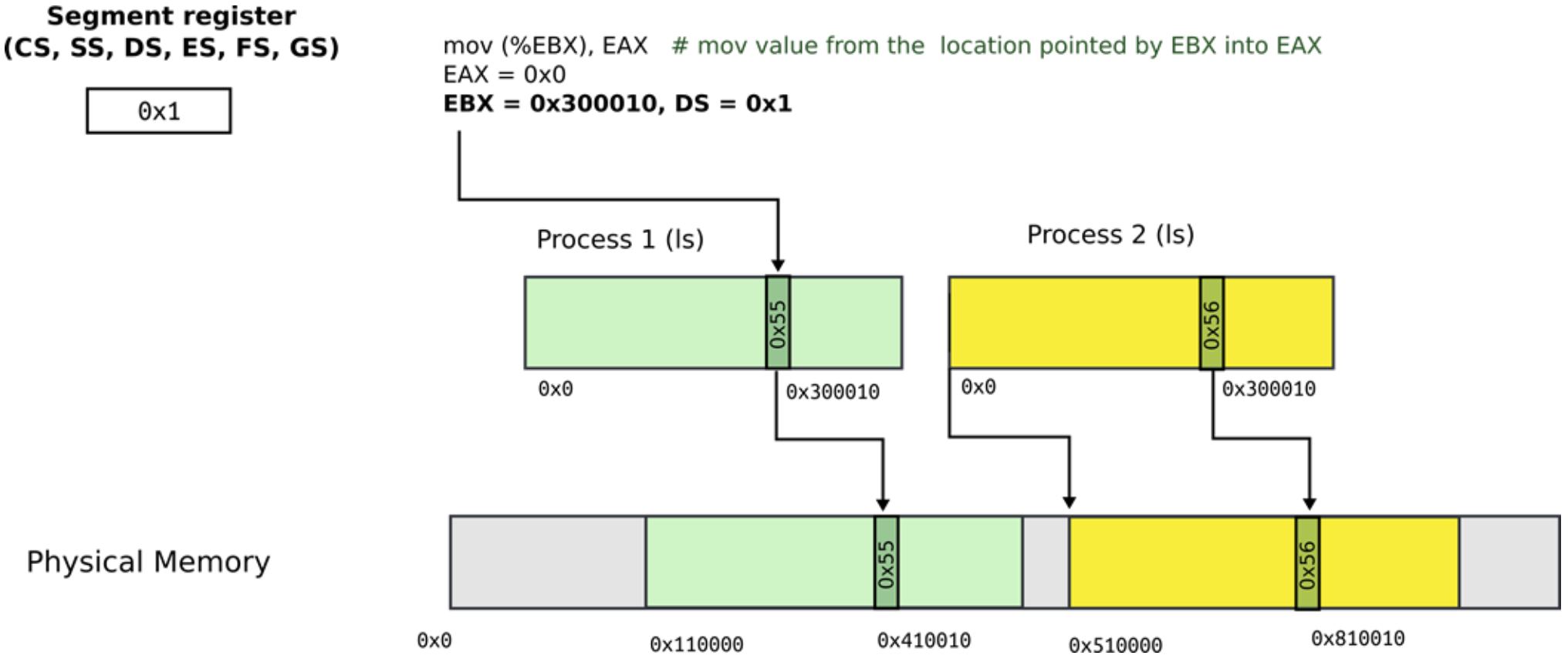
- One problem
- Where does this base address come from?
- Hardware can maintain a table of base addresses
  - One base for each process
- Dedicate a special register to keep an index into that table



Segmentation: example

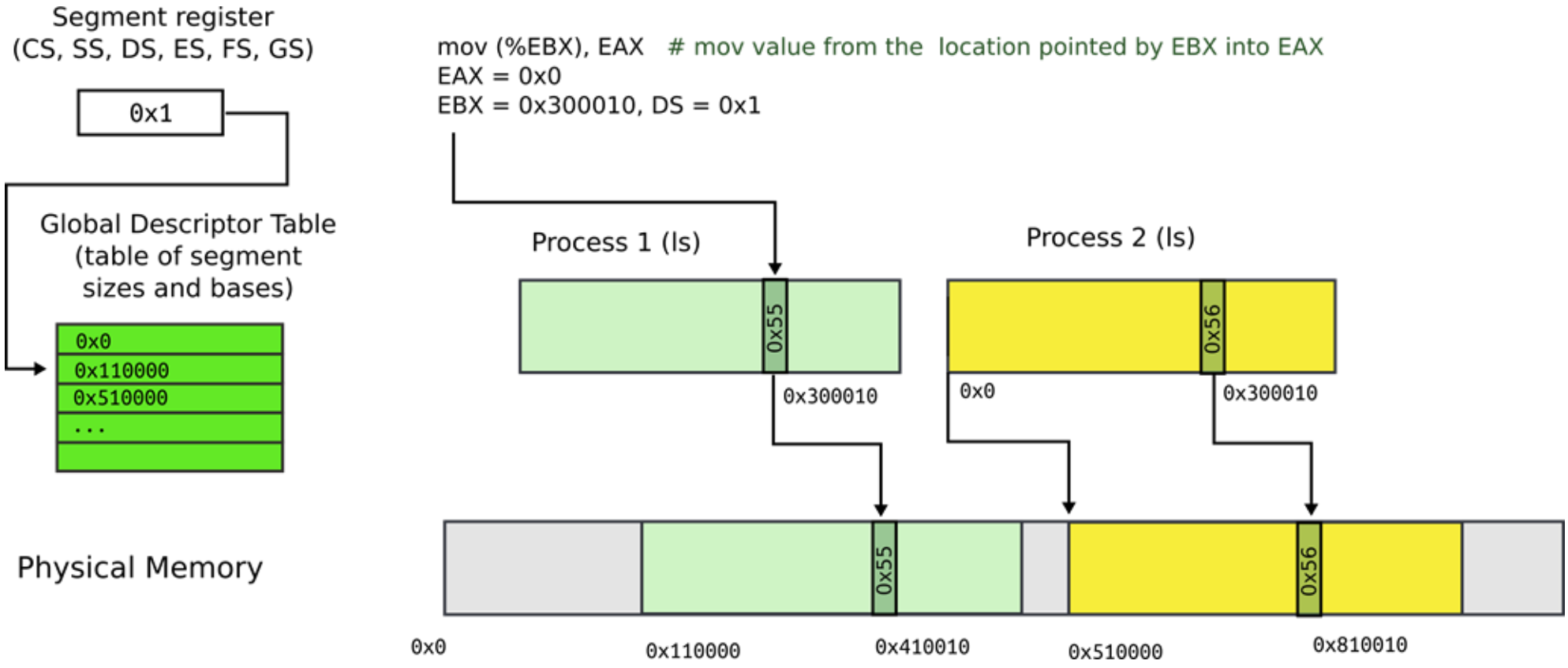


Segmentation: address consists of two parts



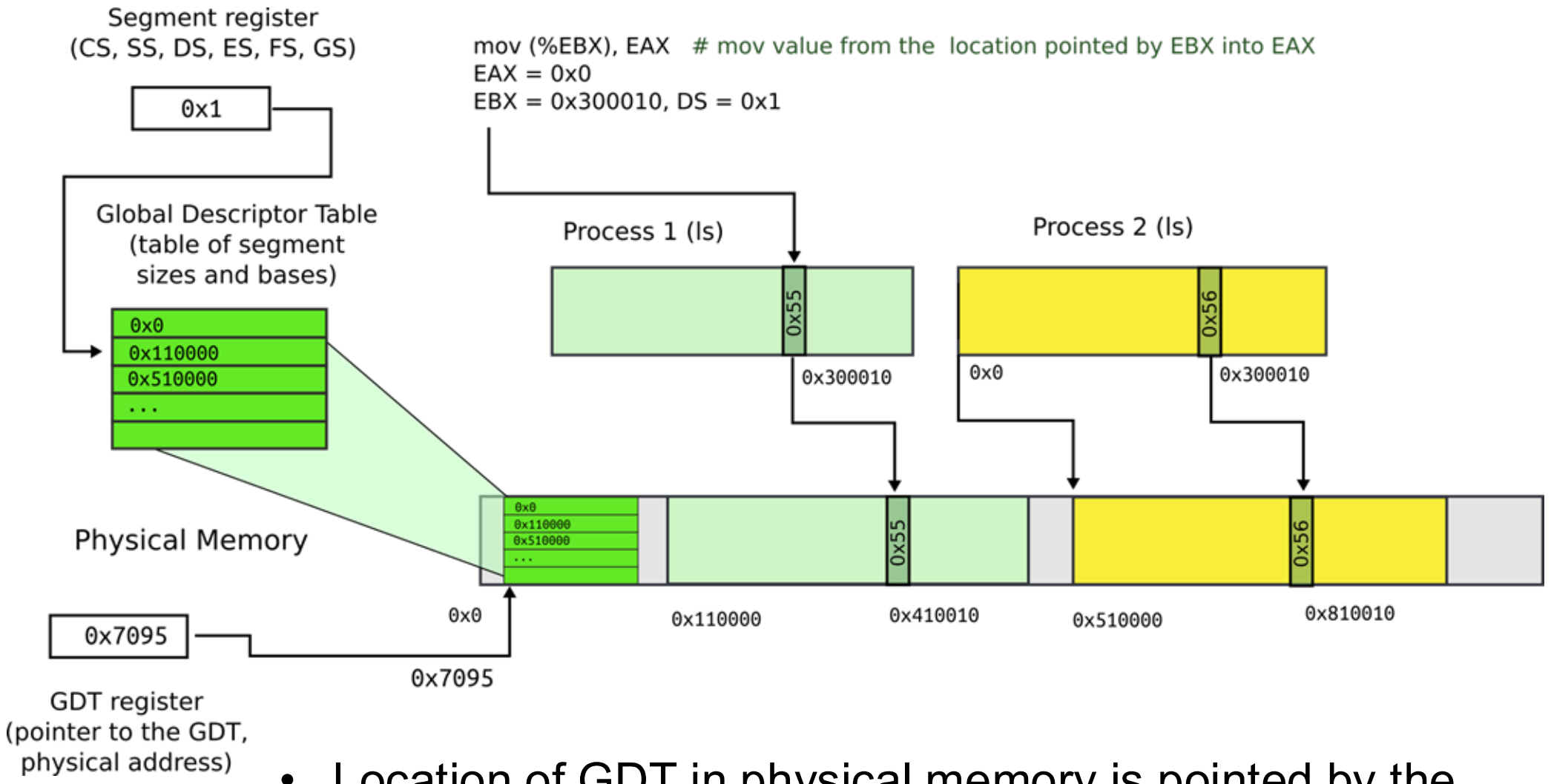
- Segment register contains segment selector
- General registers contain offsets
- Intel calls this address: “logical address”

Segmentation: Global Descriptor Table



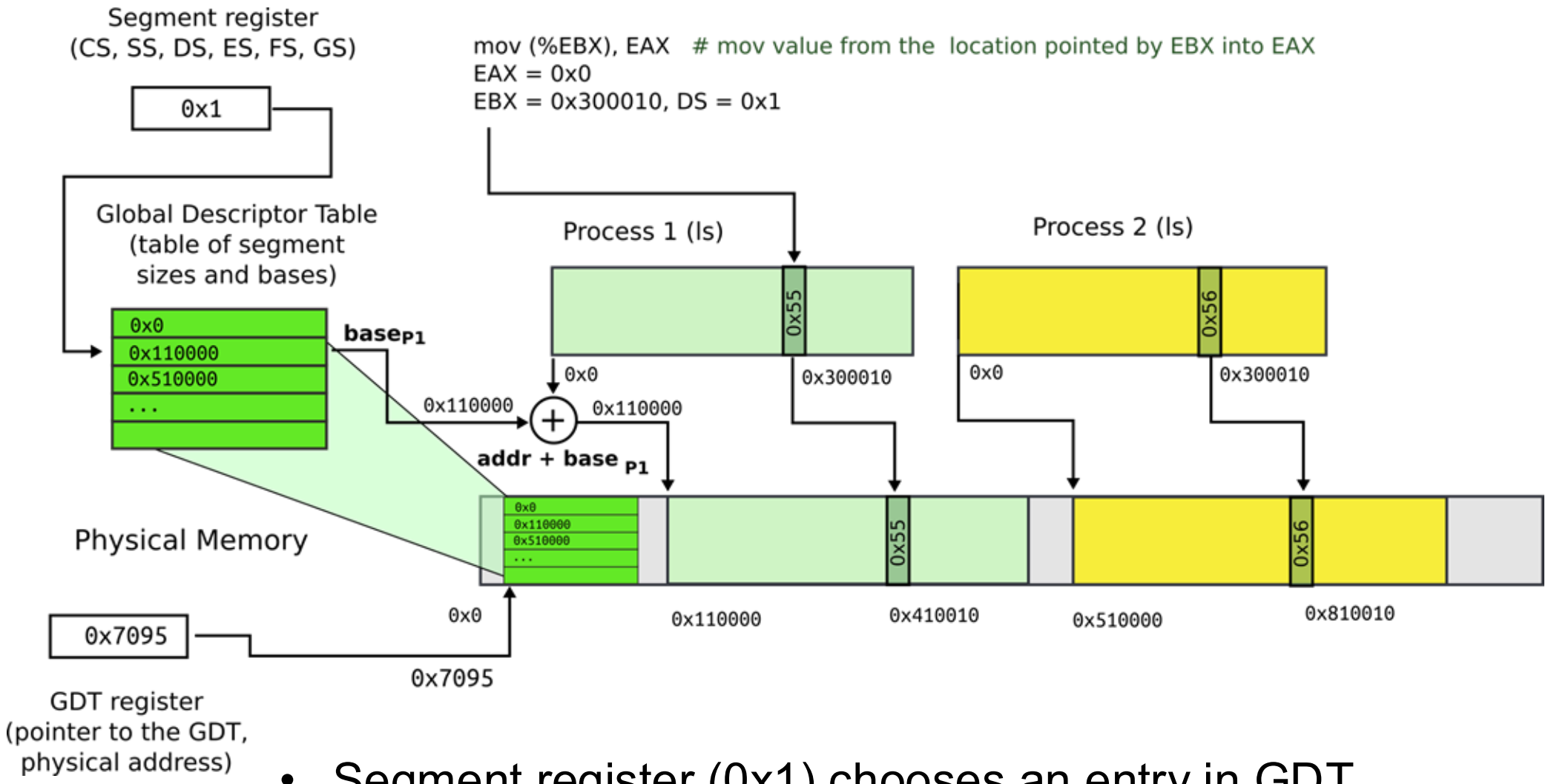
- GDT is an array of segment descriptors
- Each descriptor contains base and limit for the segment
- Plus access control flags

Segmentation: Global Descriptor Table



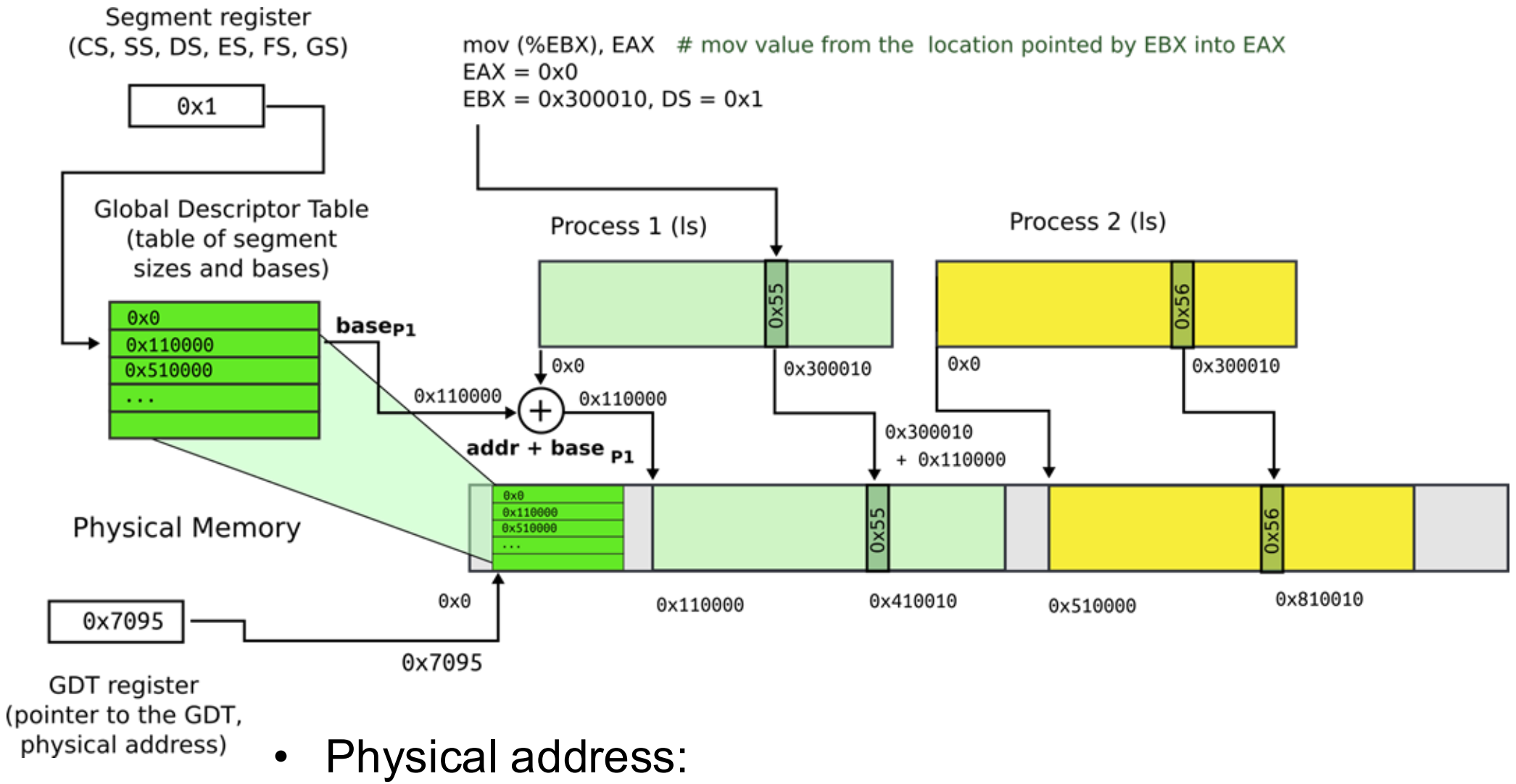
- Location of GDT in physical memory is pointed by the GDT register

Segmentation: base + offset



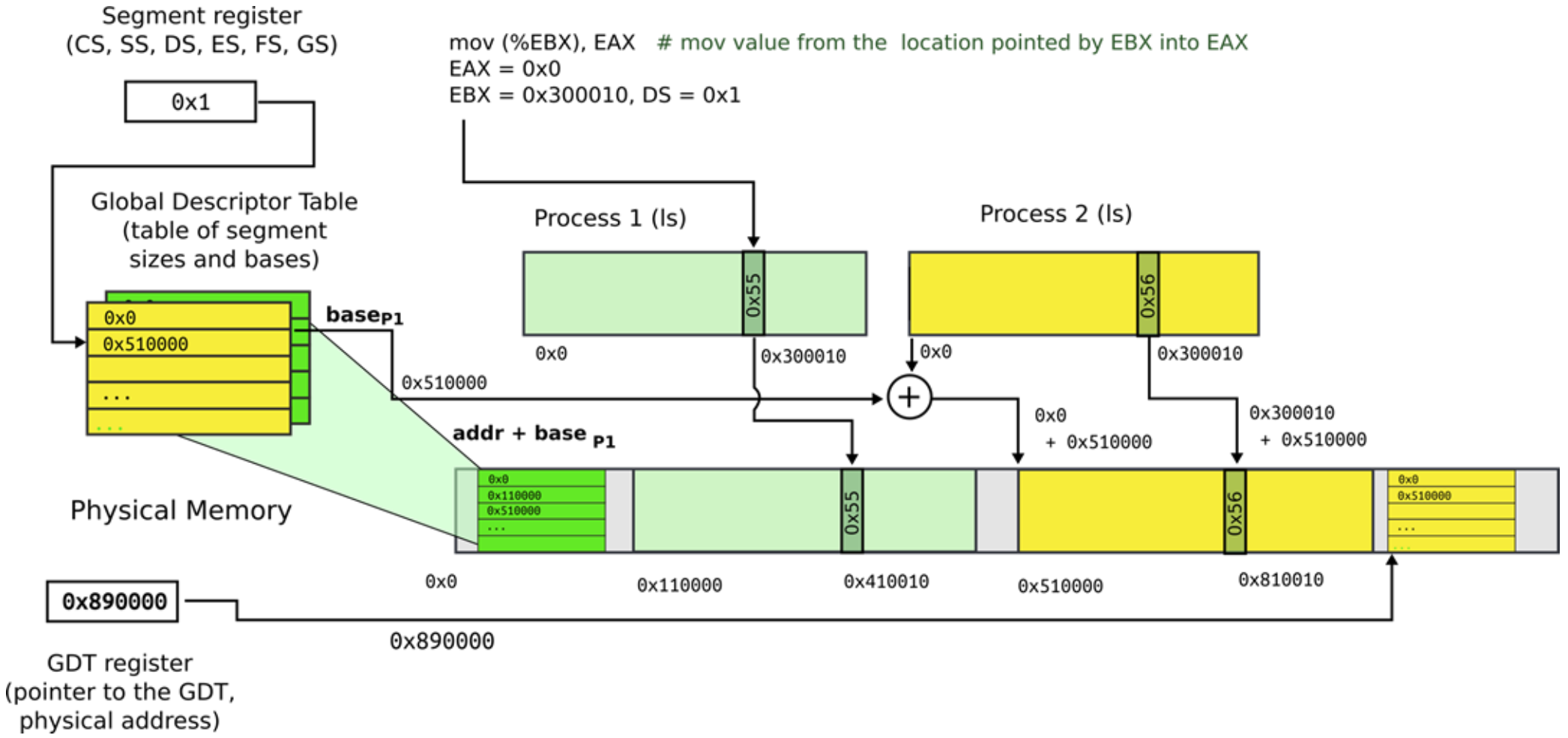
- Segment register (0x1) chooses an entry in GDT
- This entry contains base of the segment (0x110000) and limit (size) of the segment (not shown)

Segmentation: base + offset



- Physical address:
- 0x410010 = 0x300010 (offset) + 0x110000 (base)
- Intel calls this address “linear”

Segmentation: process 2



- Each process has a private GDT
  - Alternatively you reload the content of GDT on process switch
- OS switches between GDTs

Poll: Segmentation

Assume that GDT has the following base addresses for each entry:

GDT

0x1 -> 0x100

0x2 -> 0x300

0x3 -> 0x400

0x4 -> 0x800

https://pollev.com/cs5460

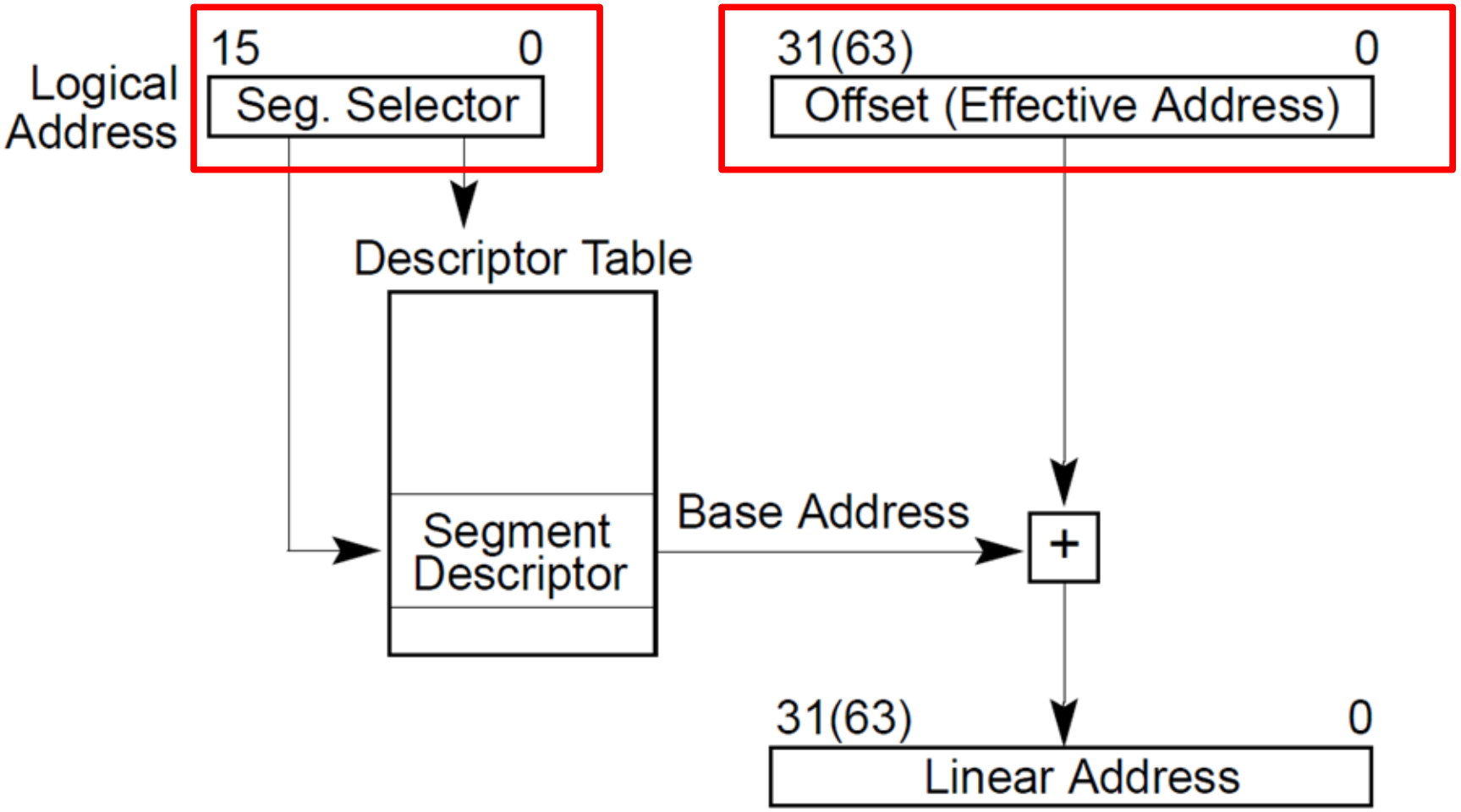
What would be the address of `ds:eax` if DS register selects entry #3 in the GDT, and EAX contains value 0x3245

New addressing mode:

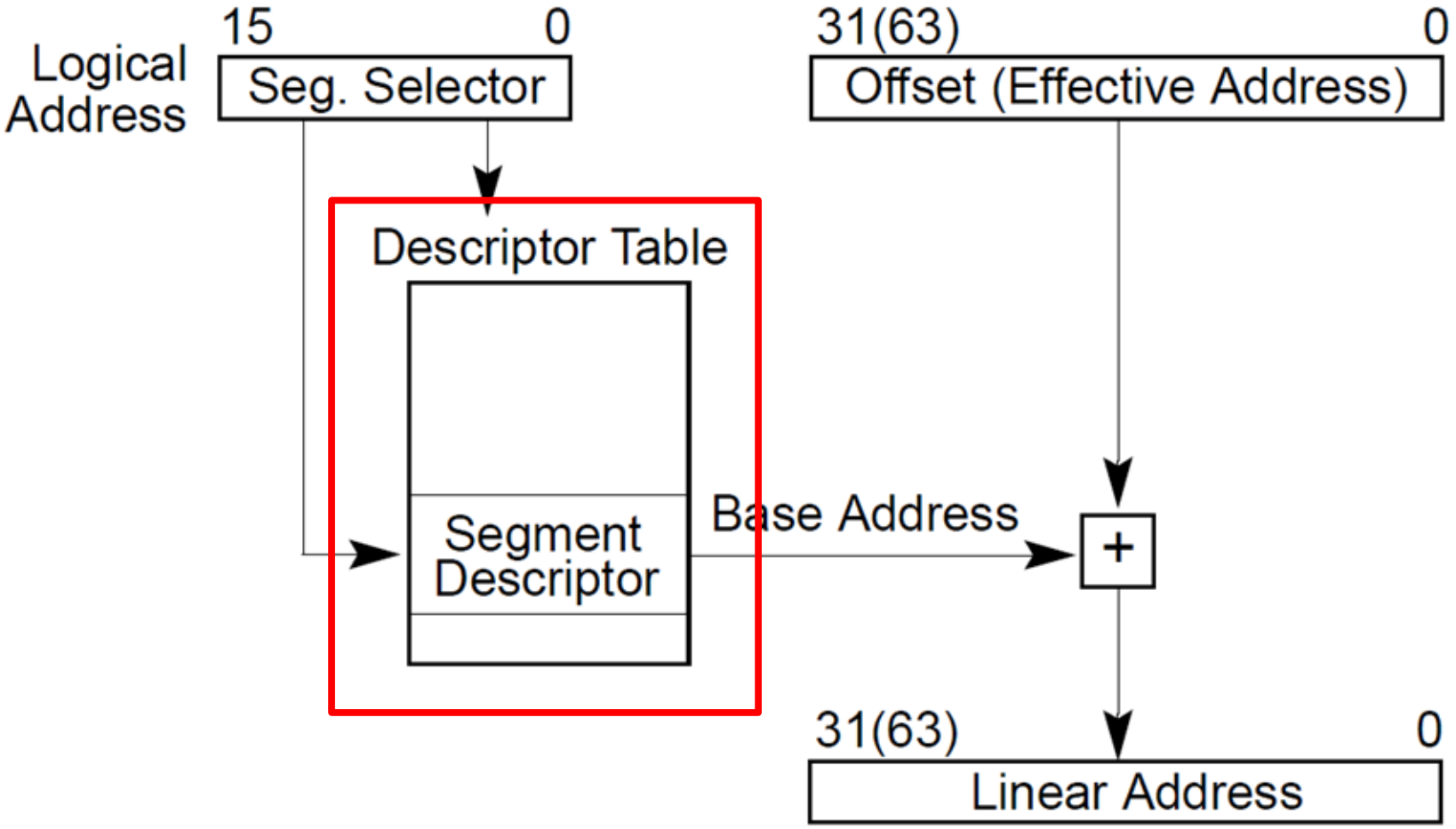
“logical addresses”

All addresses are logical address

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

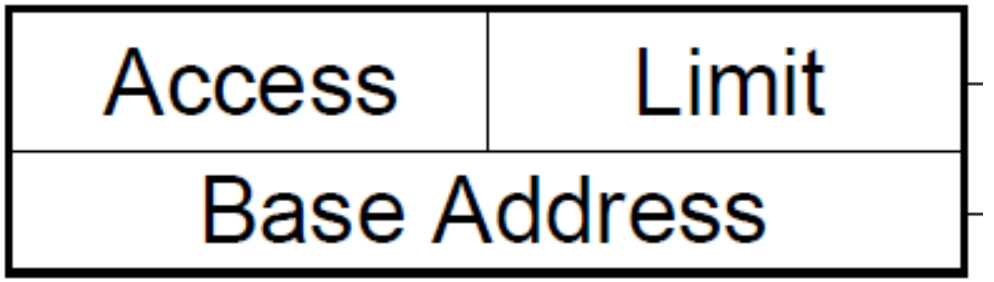


- Segment selector (16 bit)
- Is simply an index into an array (Descriptor Table)
- That holds segment descriptors
  - Base and limit (size) for each segment

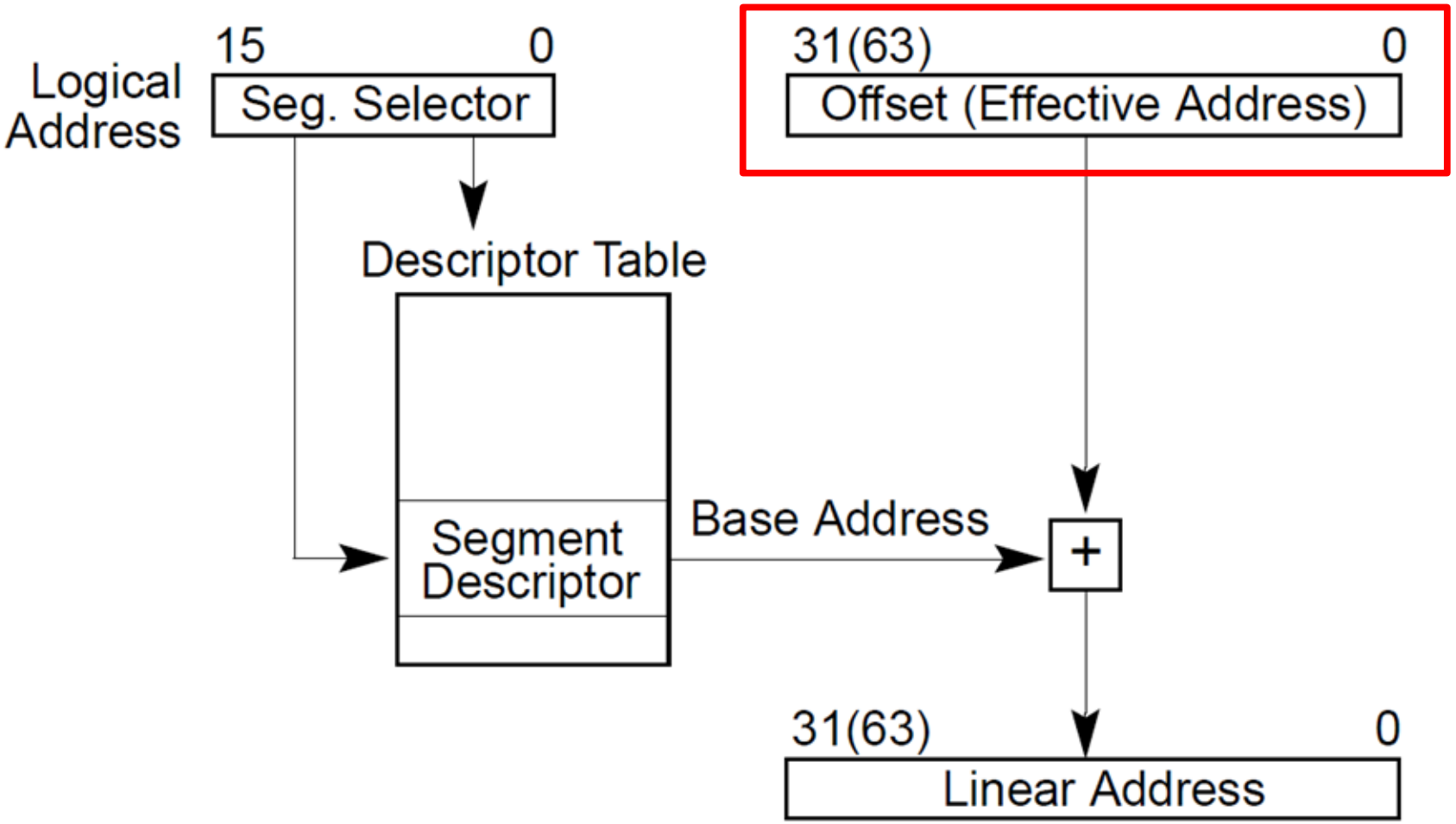


Elements of the descriptor table are segment descriptors

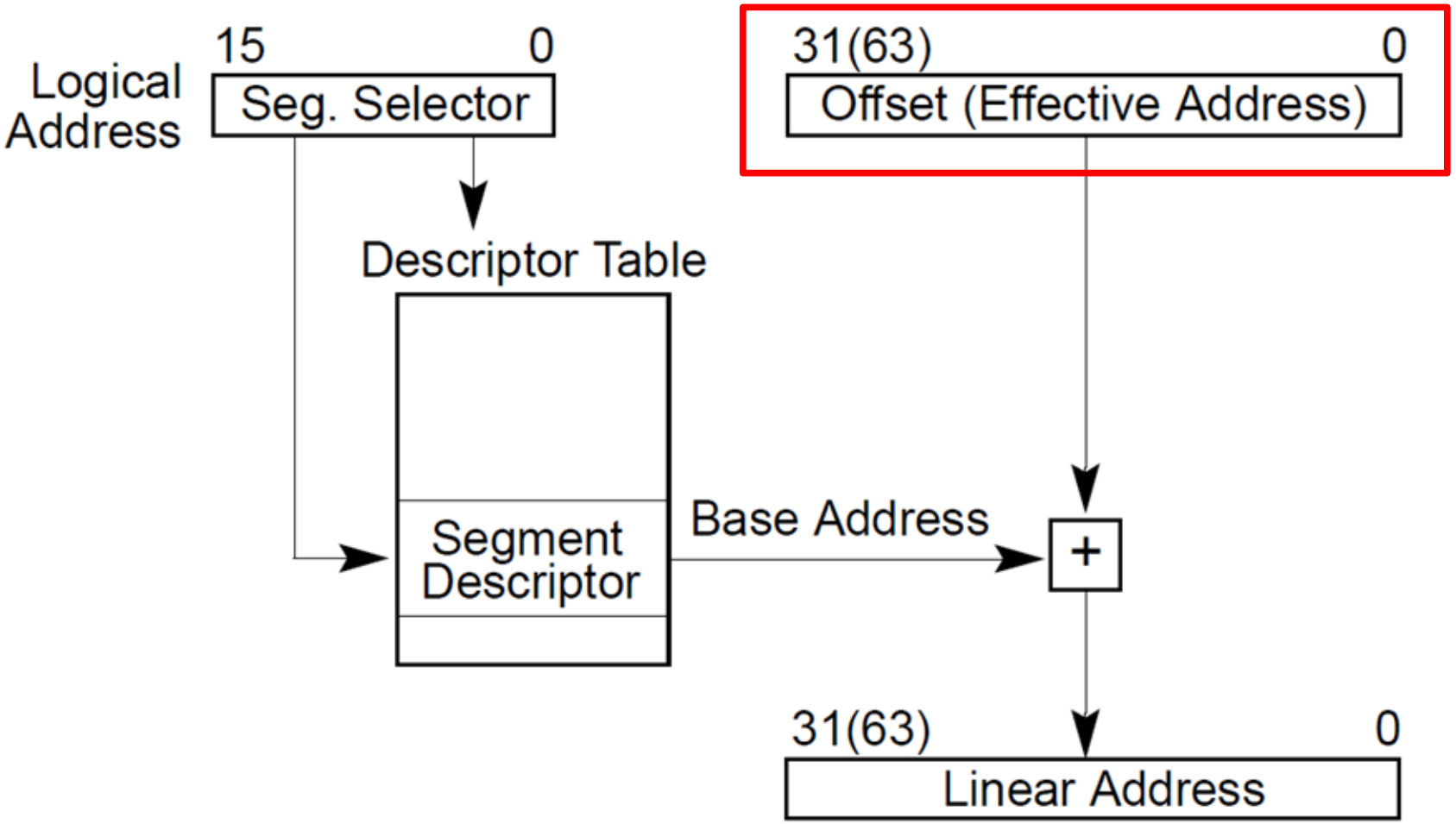
- Base address
  - 0 – 4 GB
- Limit (size)
  - 0 – 4 GB
- Access rights
  - Executable, readable, writable
- Privilege level (0 - 3)



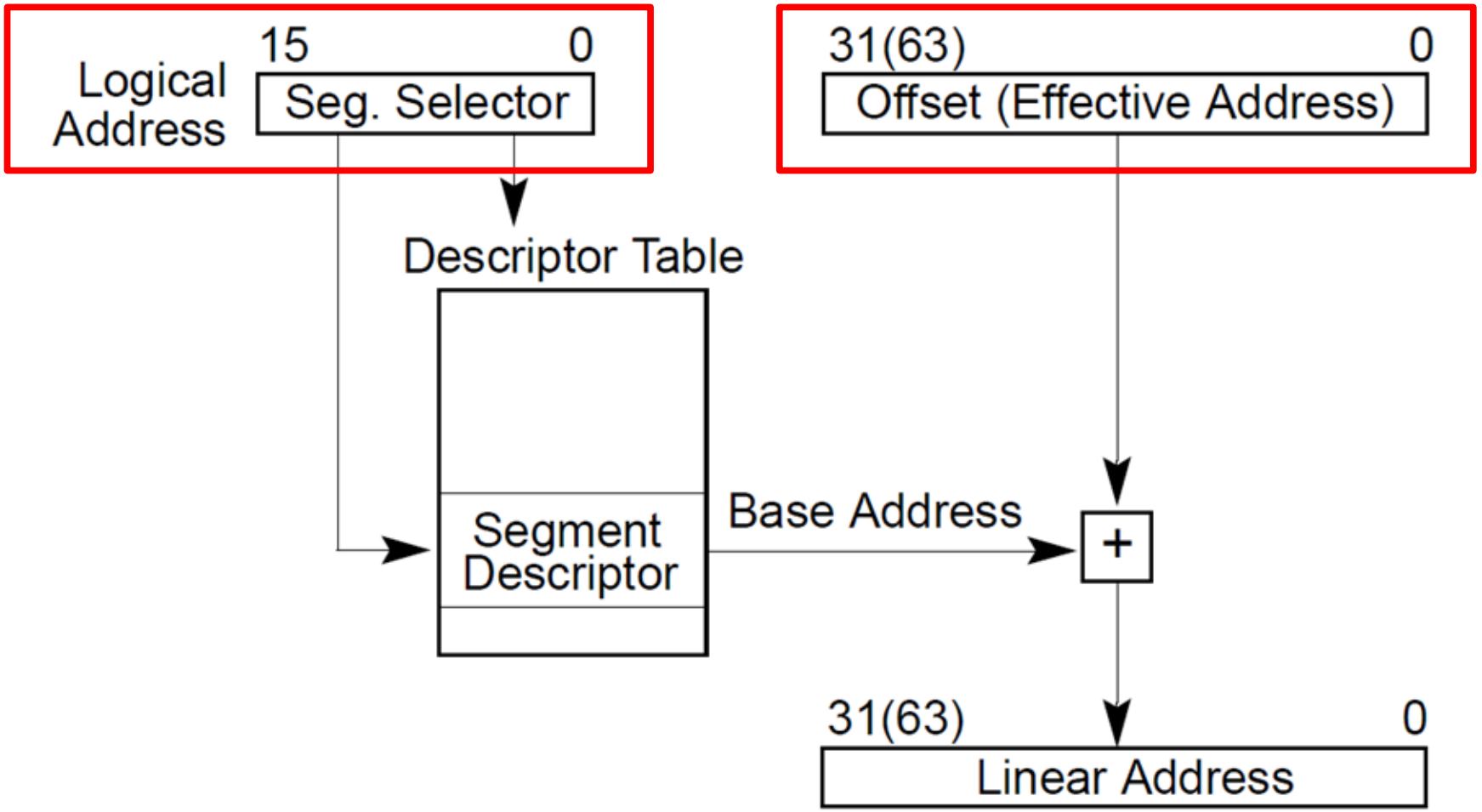
- Offsets into segments (x in our example) or "effective addresses" are in registers



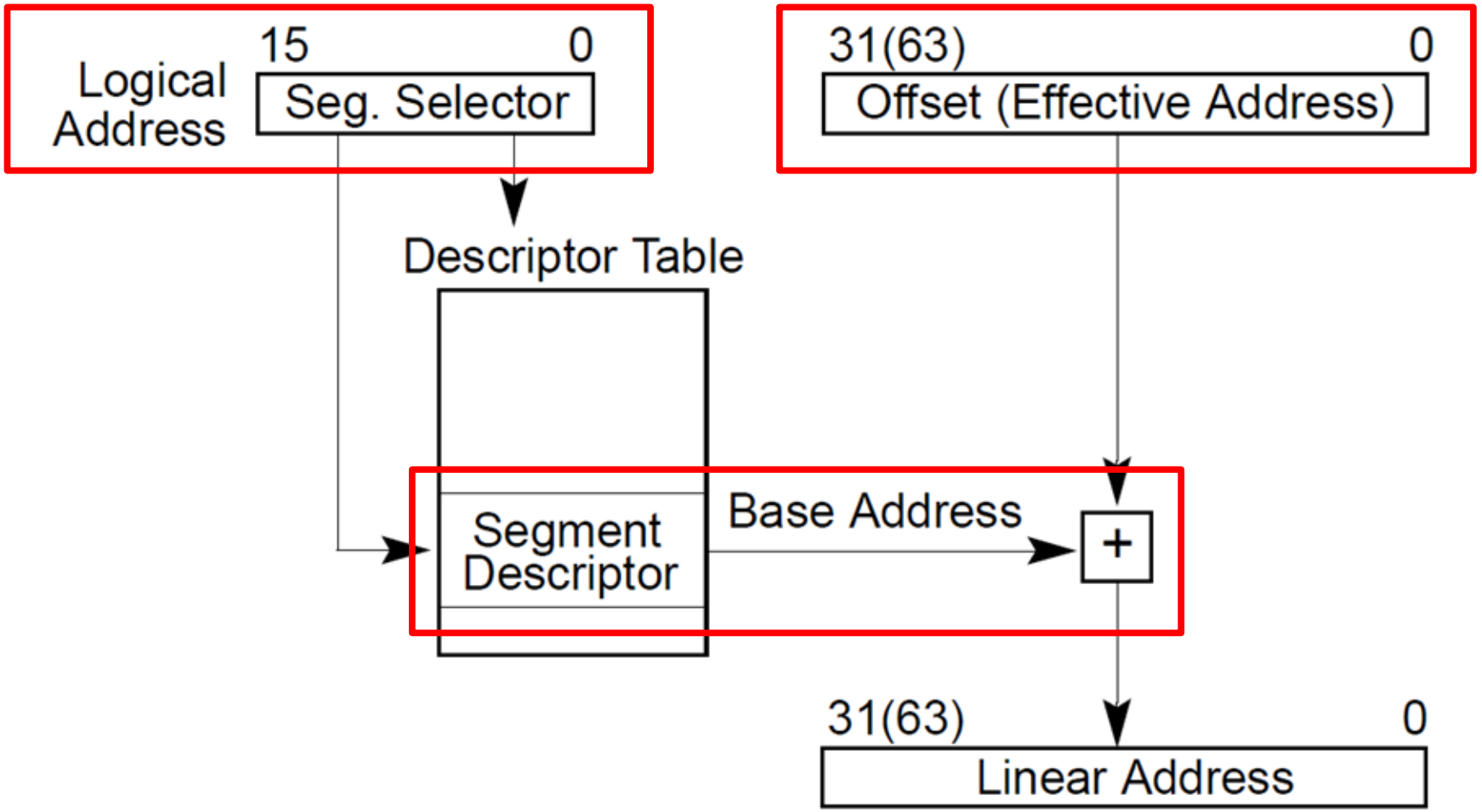
- Logical addresses are translated into physical (linear)
  - *Effective address + DescriptorTable[selector].Base*



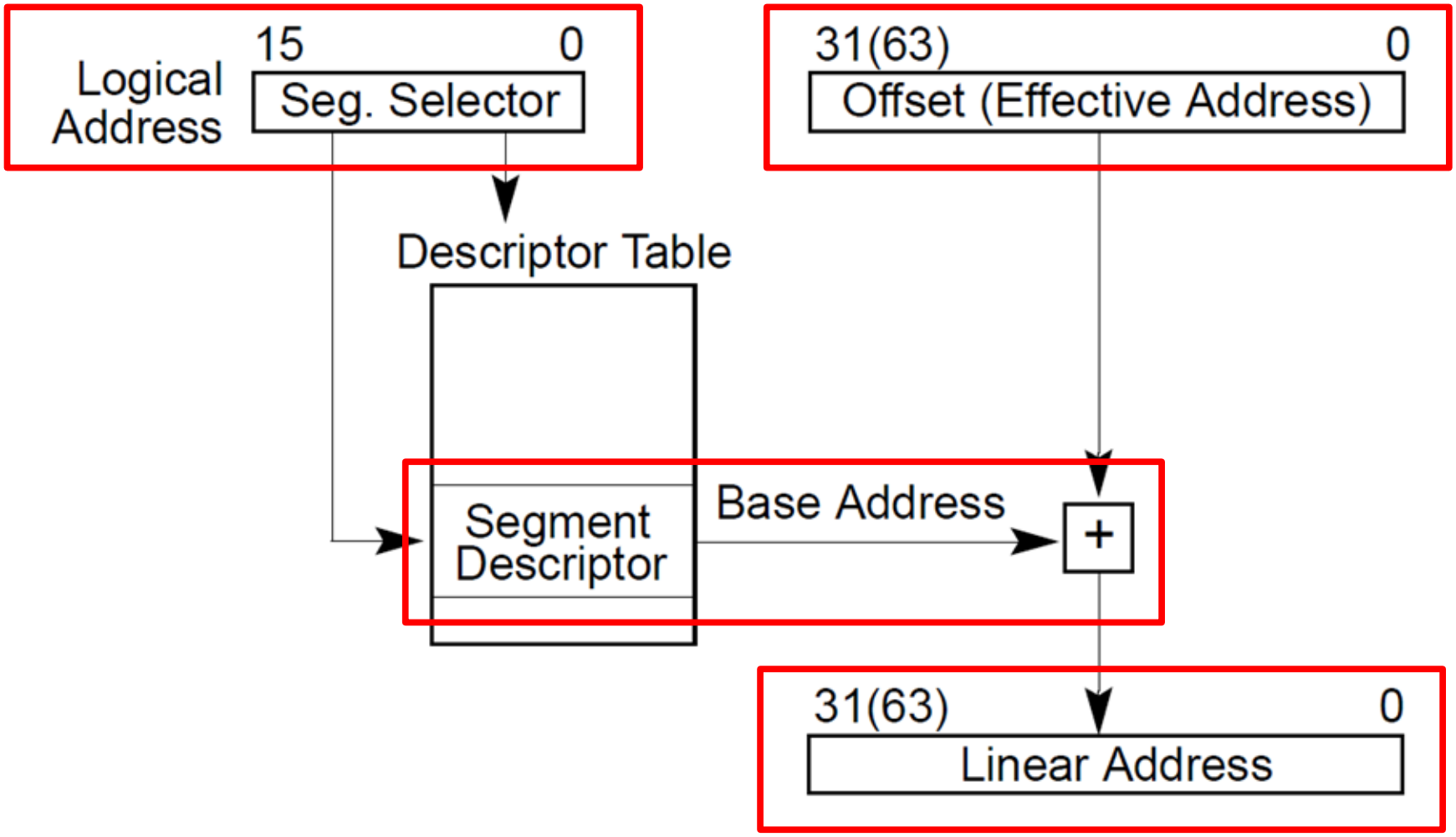
- Logical addresses are translated into physical
  - *Effective address* + *DescriptorTable[selector].Base*



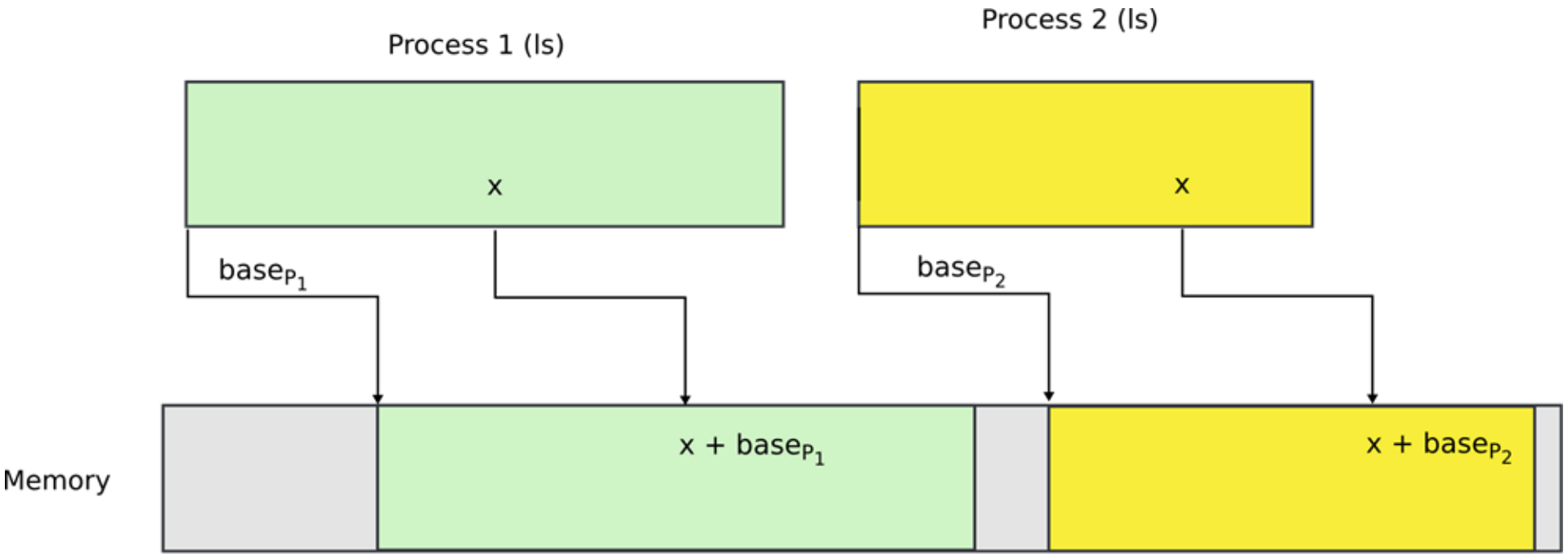
- Logical addresses are translated into physical
  - *Effective address* + *DescriptorTable[selector].Base*



- 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



Segment registers

- Hold 16 bit segment selectors
  - Indexes into GDT

- Segments are associated with one of three types of storage
  - Code
  - Data
  - Stack

Programing with segements (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 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.

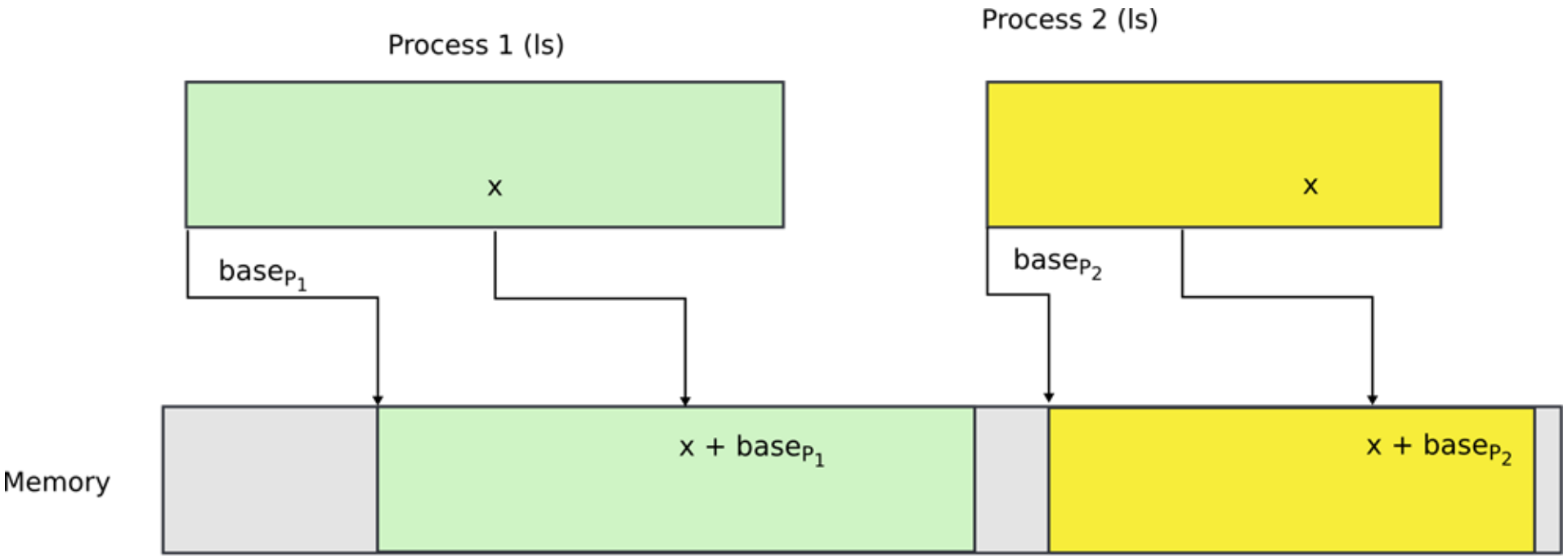
- This is cumbersome
- Instead the idea is: infer code, data and stack segments from the instruction type
  - Control-flow instructions use code segment (jump, call)
  - Stack management (push/pop) uses stack
  - Most loads/stores use data segment
- Extra segments (es, fs, gs) must be used explicitly

Segmentation: what did we achieve

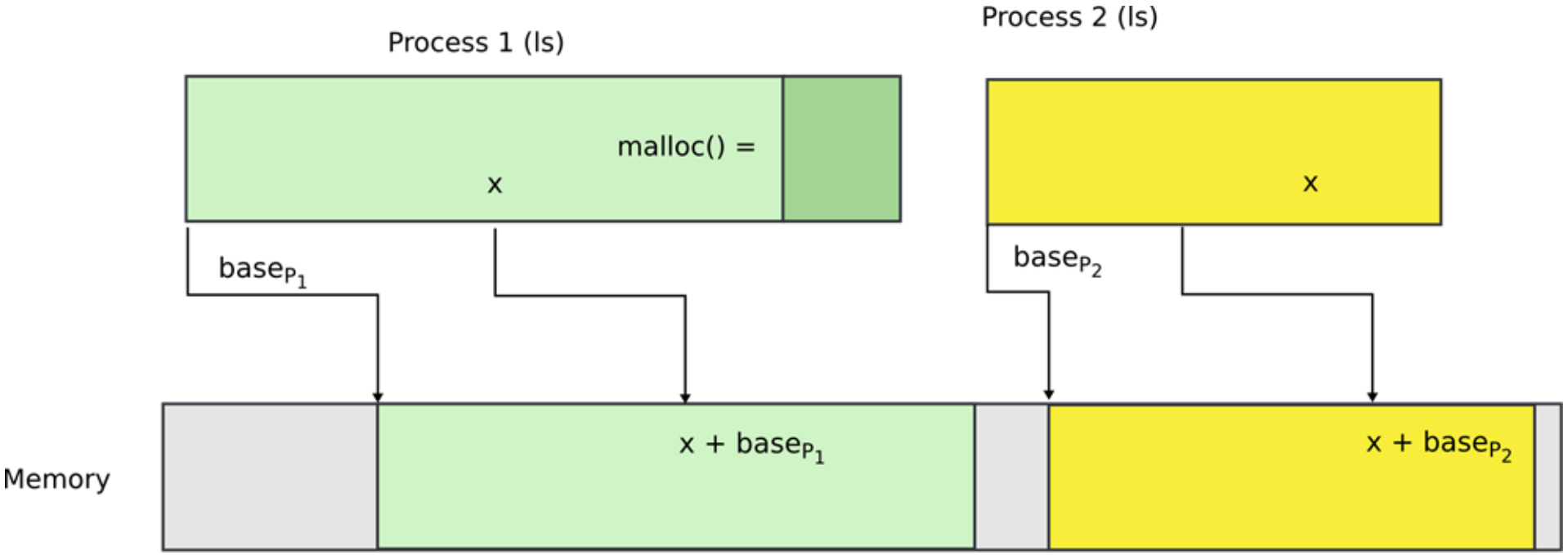
- Illusion of a private address space
- Identical copy of an address space in multiple programs
  - We can implement fork()
- Isolation
  - Processes cannot access memory outside of their segments

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

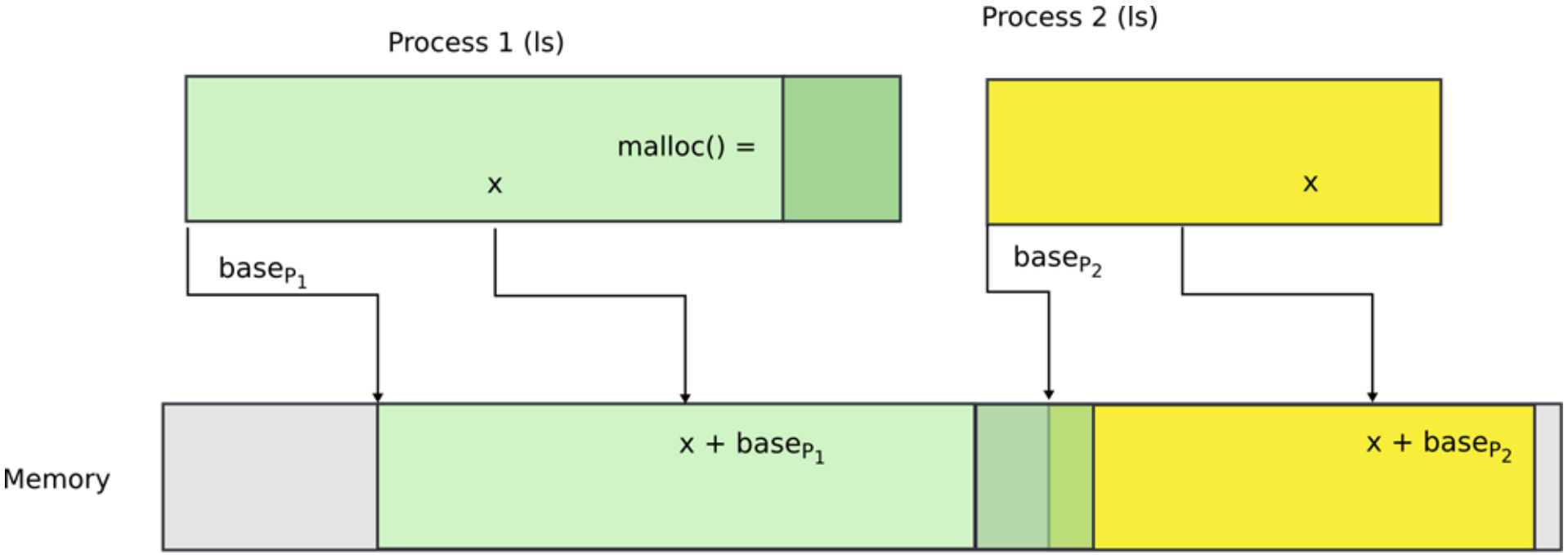
Segmentation is ok... but



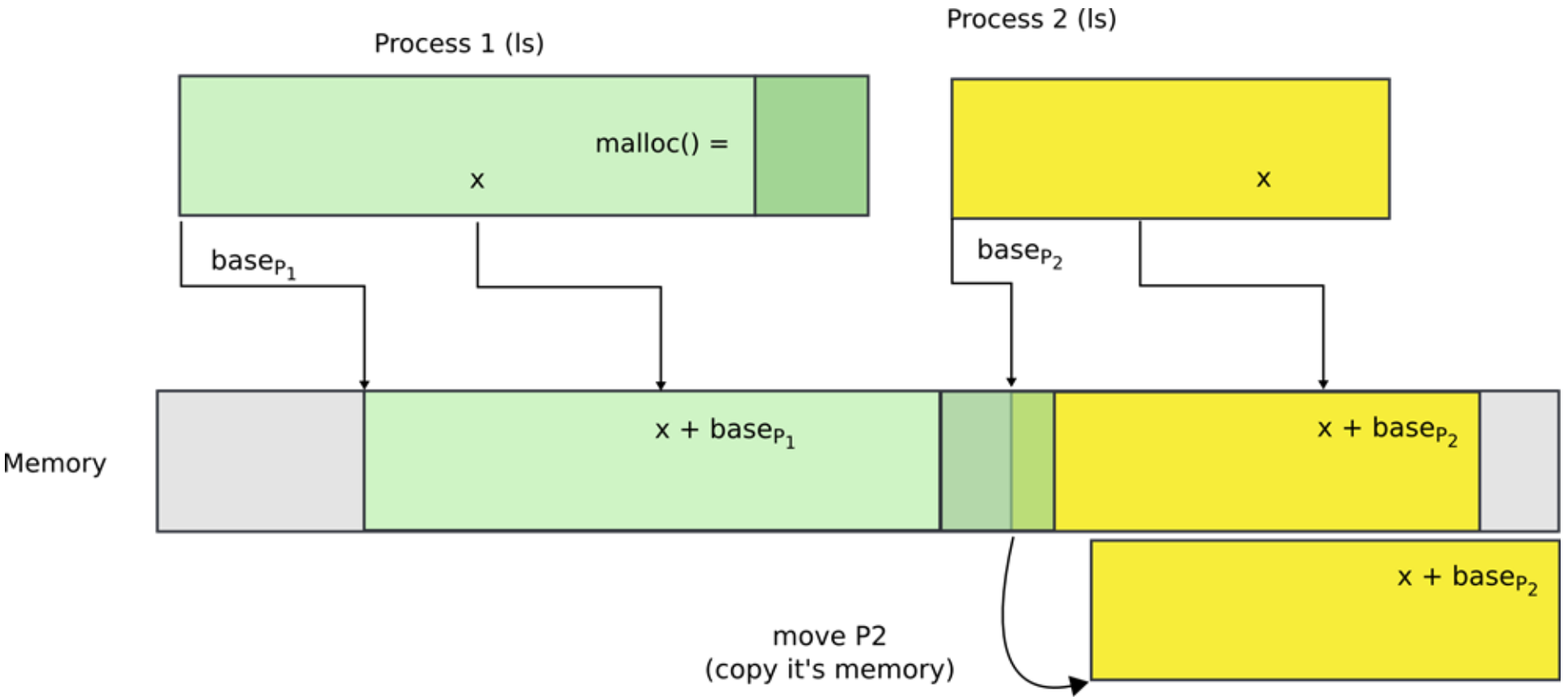
What if process needs more memory?



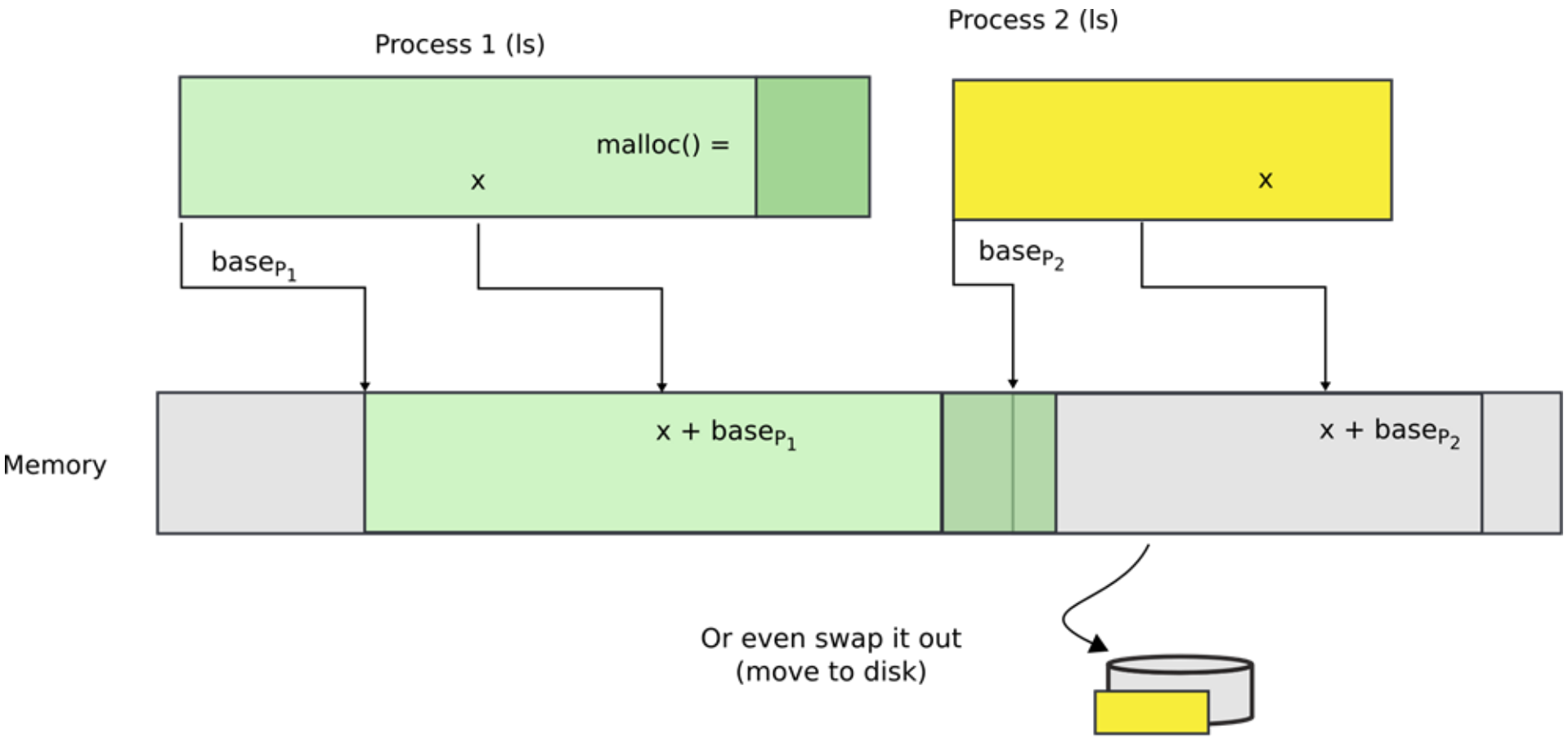
What if process needs more memory?



You can move P2 in memory



Or even swap it out to disk



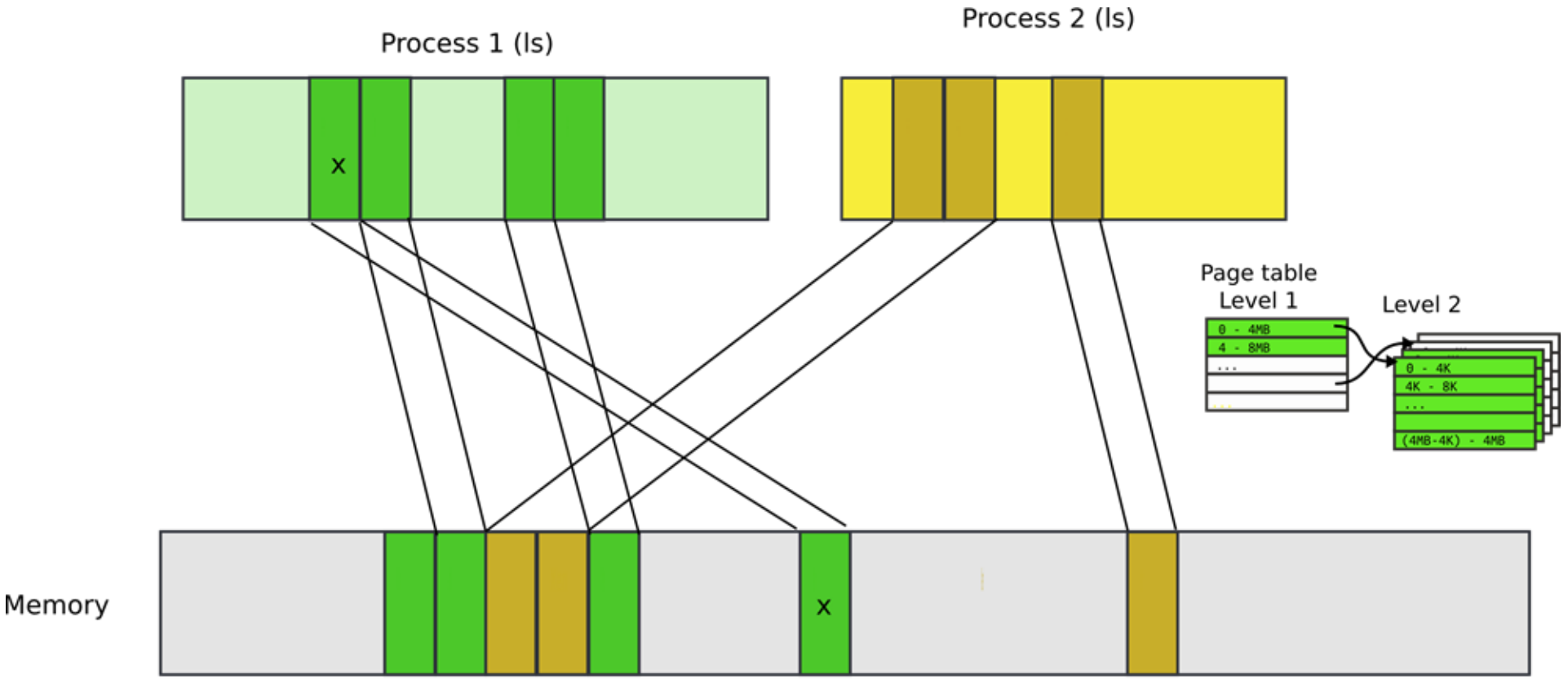
Problems with segments

- Segments are somewhat inconvenient
  - 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)
Process 2 (ls)
X
Page table
Level 1
Level 2
0 - 4MB
4 - 8MB
...
0 - 4K
4K - 8K
...
(4MB-4K) - 4MB
Memory
X

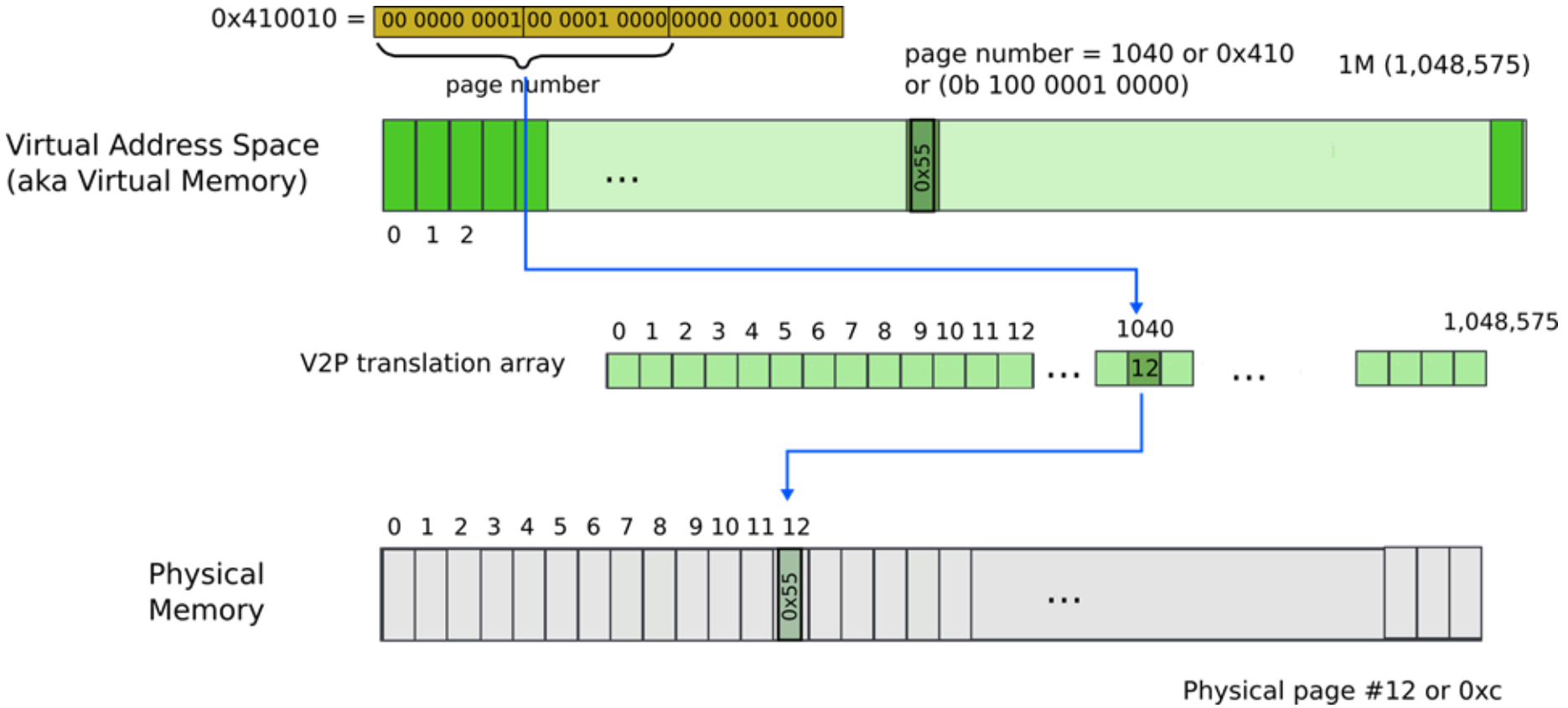
Paging idea

- 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

- Compared with segmentation (single base + limit)
  - Much more flexibility

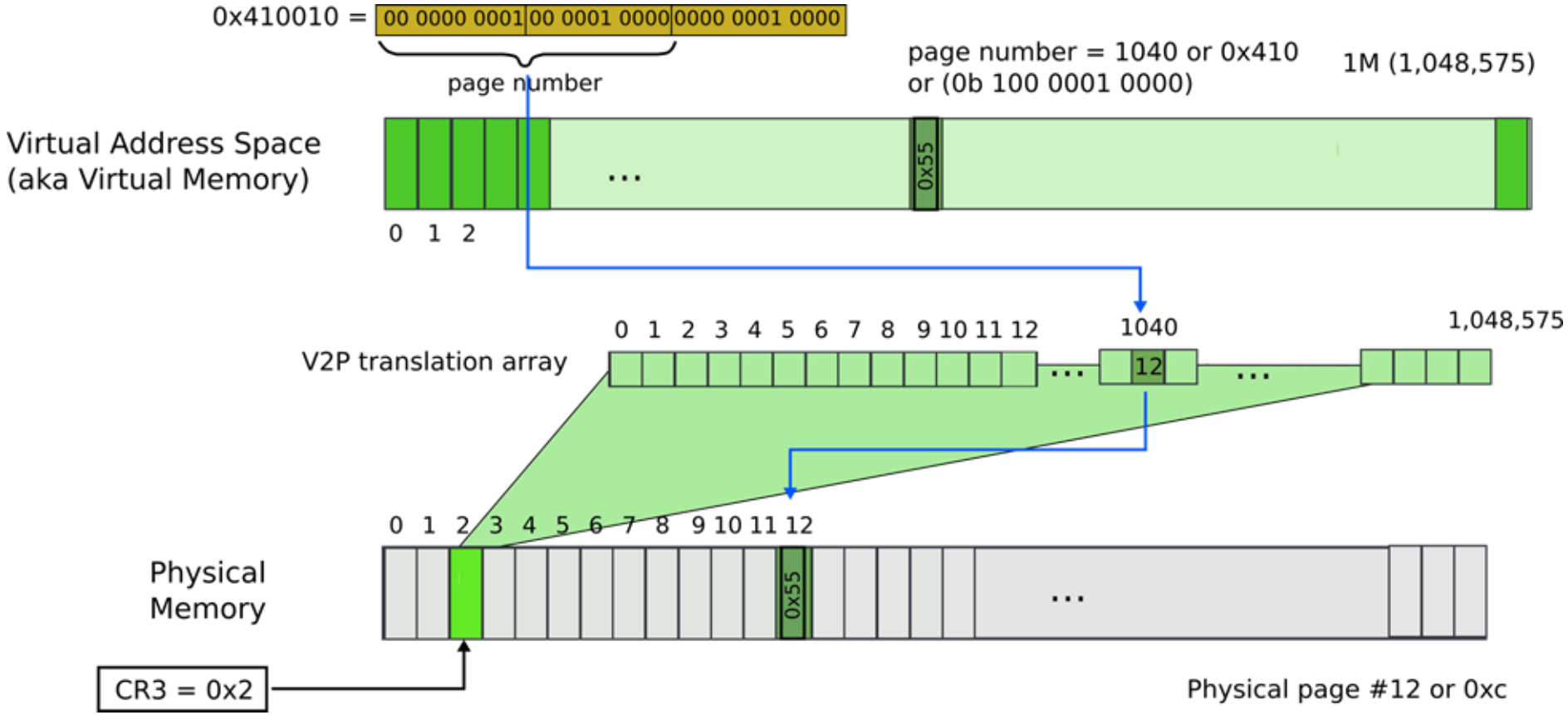
How can we build this translation mechanism?

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 (offset) + 0x110000 (base)

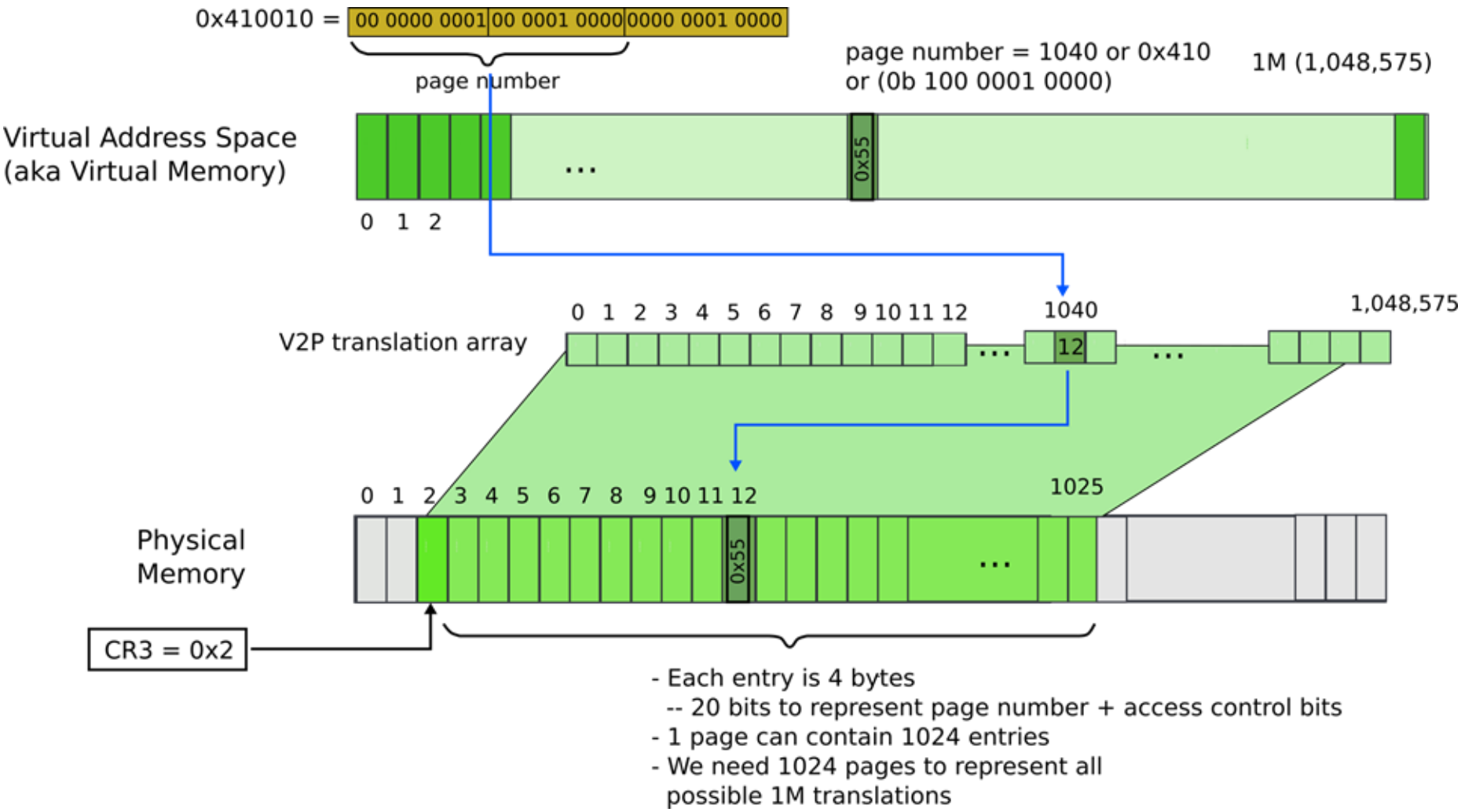
What is wrong?

What is wrong?

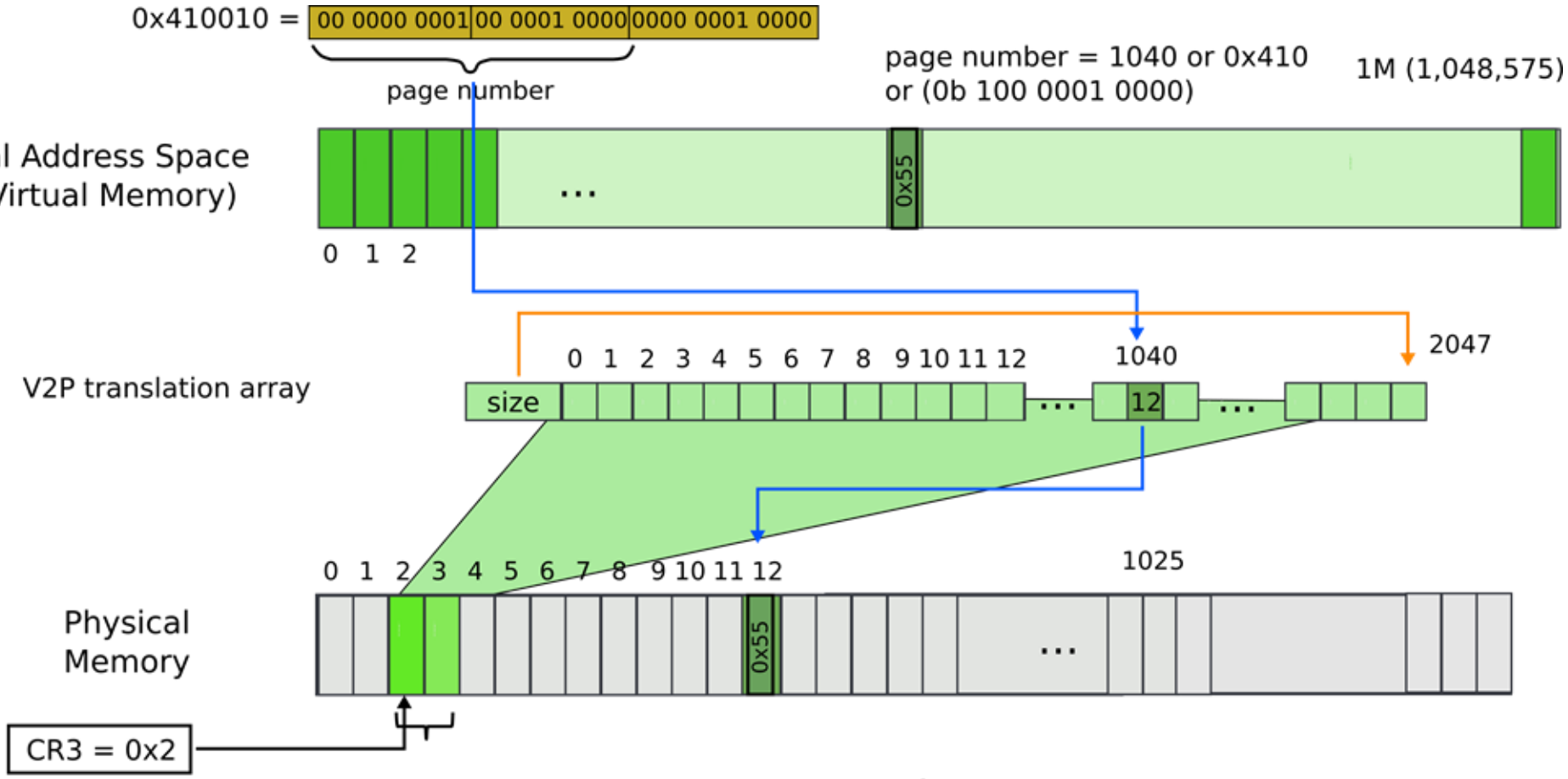
- We need 4 bytes to relocate each page
- 20 bits for physical page number
- 12 bits of access flags

- Therefore, we need array of 4 bytes x 1M entries
  - 4MBs

Paging: naive approach: translation array



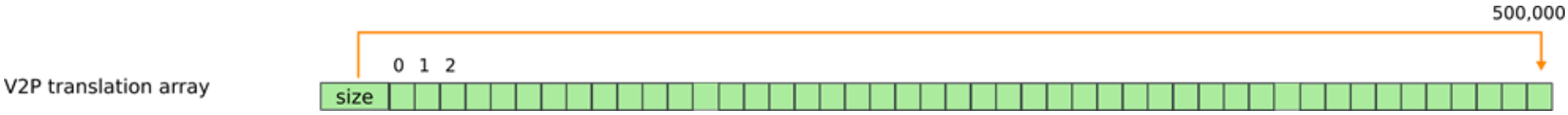
Paging: array with size



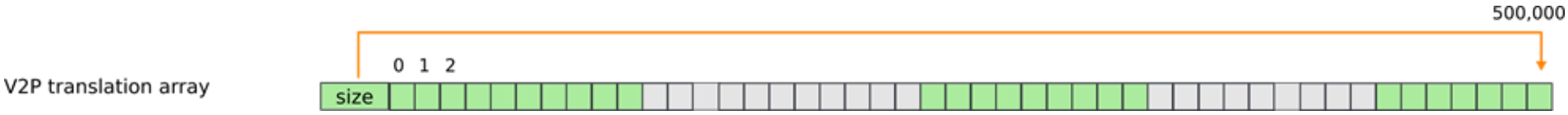
- The size controls how many entries are required

But still what may go wrong?

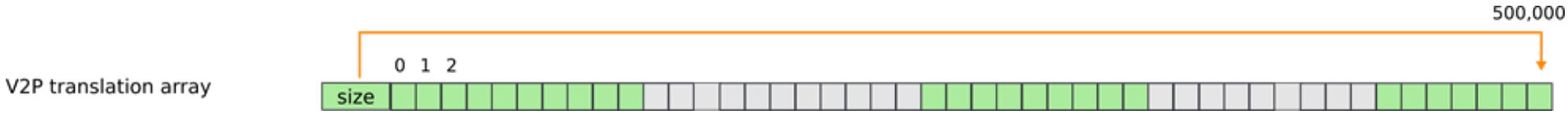
Paging: array with size



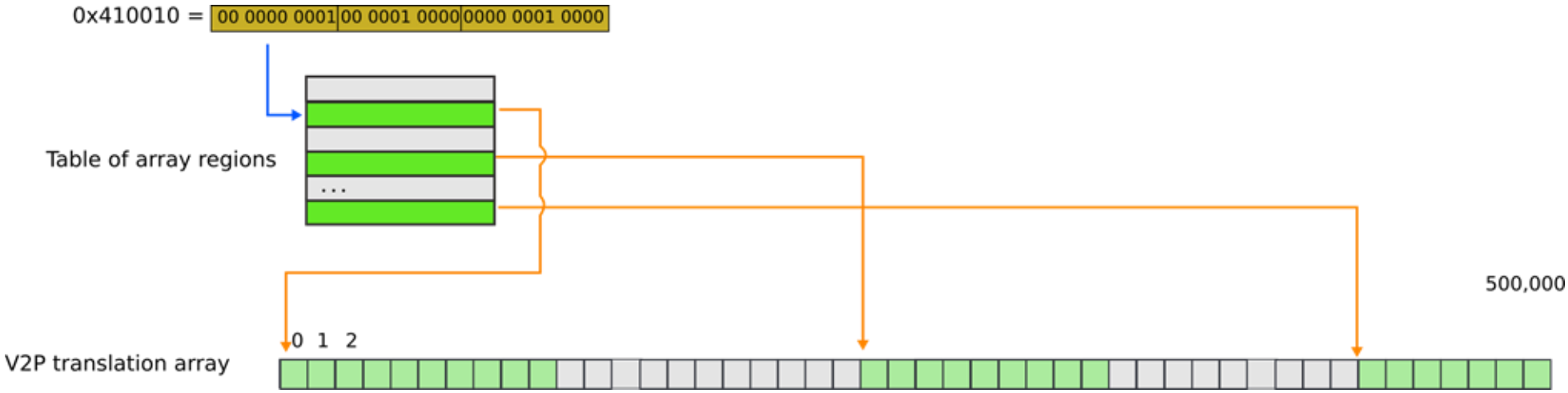
Paging: array with size



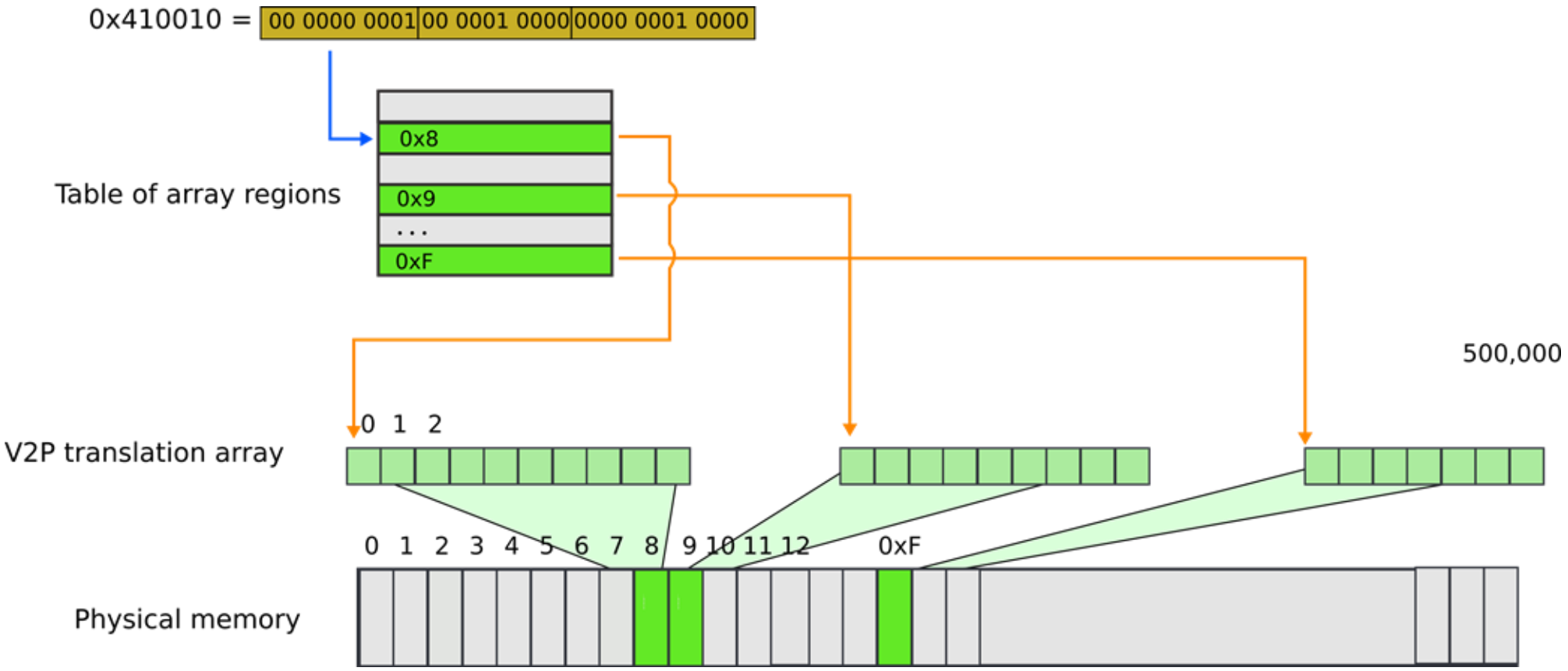
Can we improve?



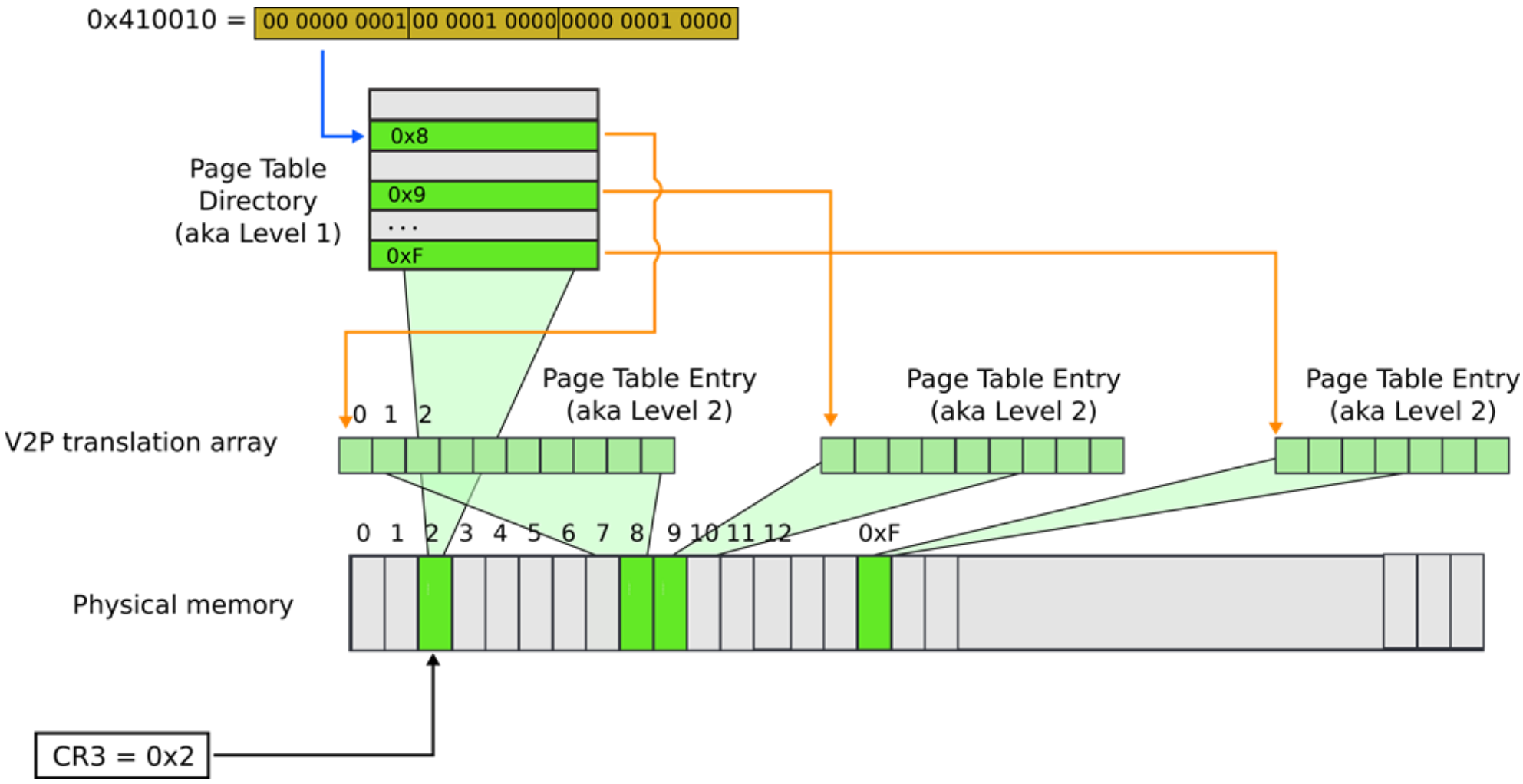
Paging: array of arrays



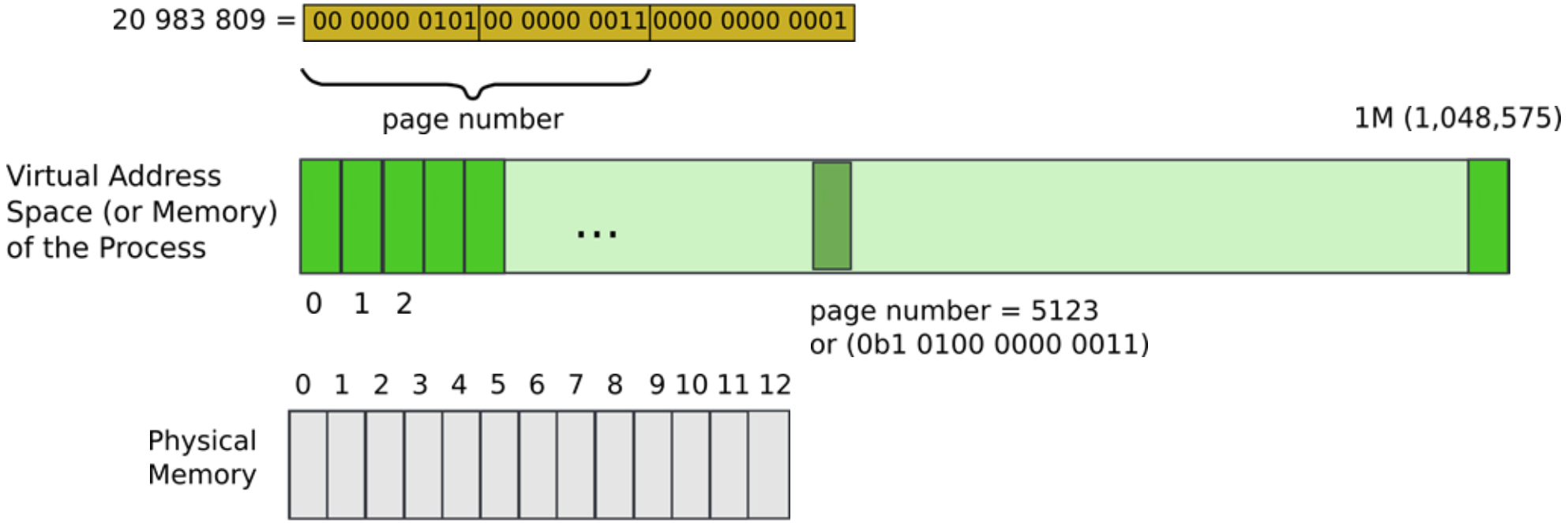
Paging: array of arrays

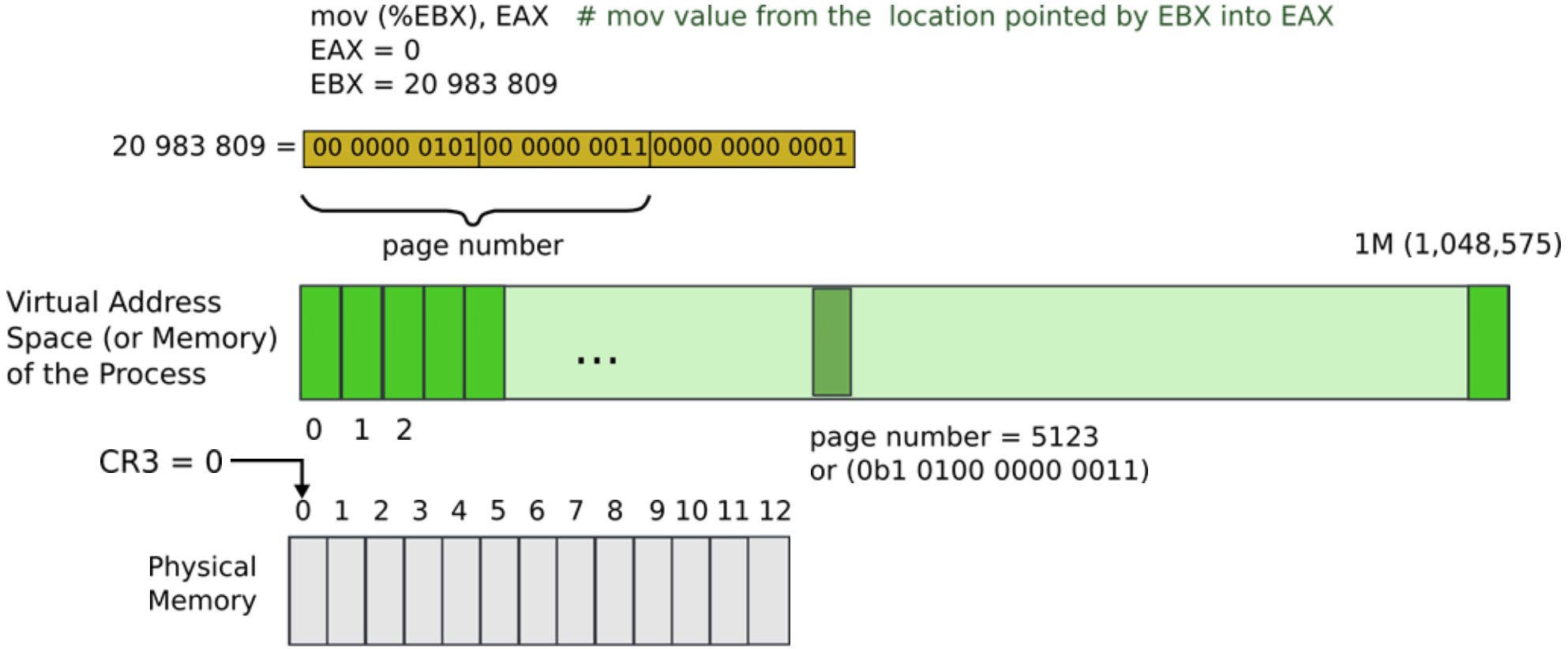


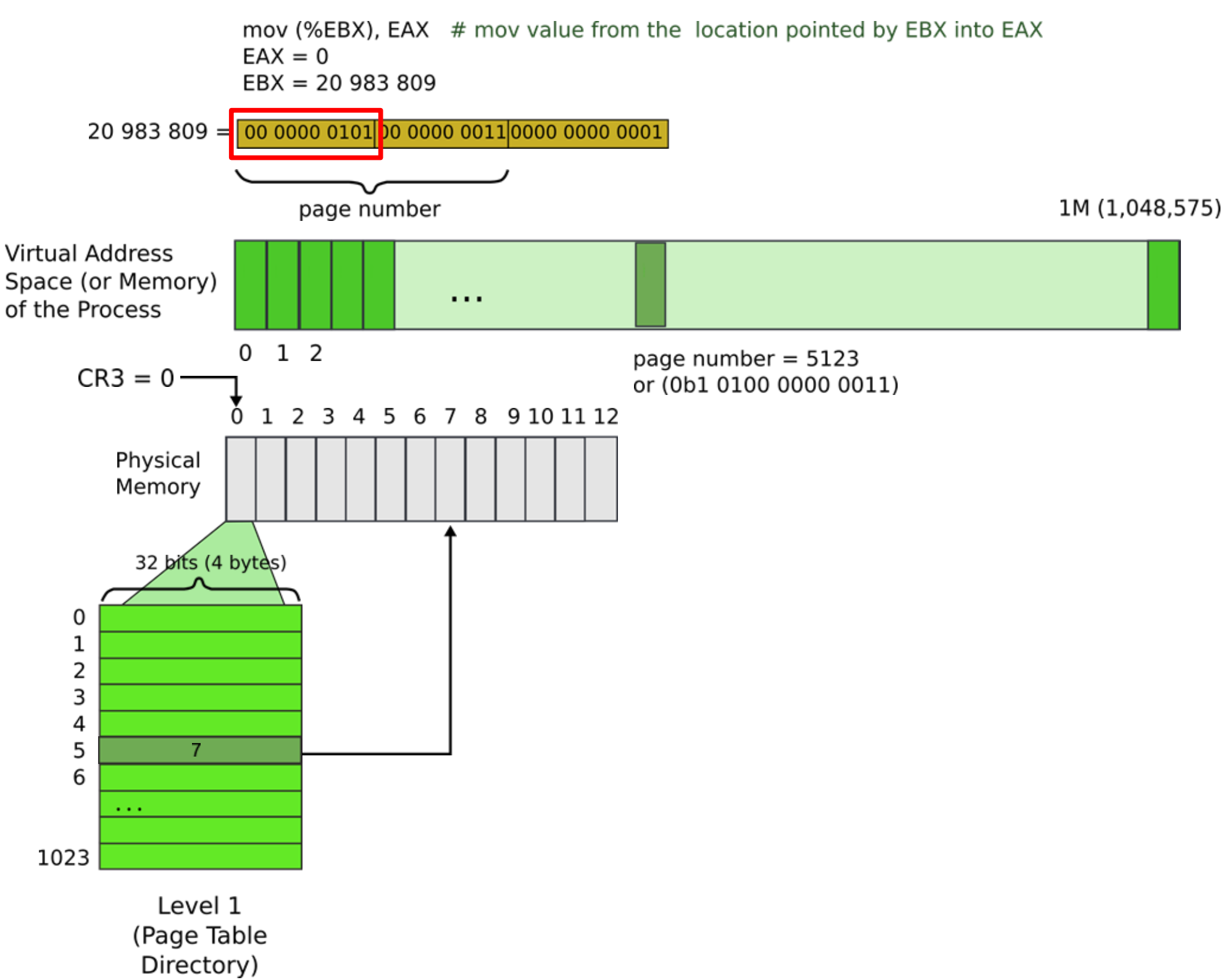
Paging: page table

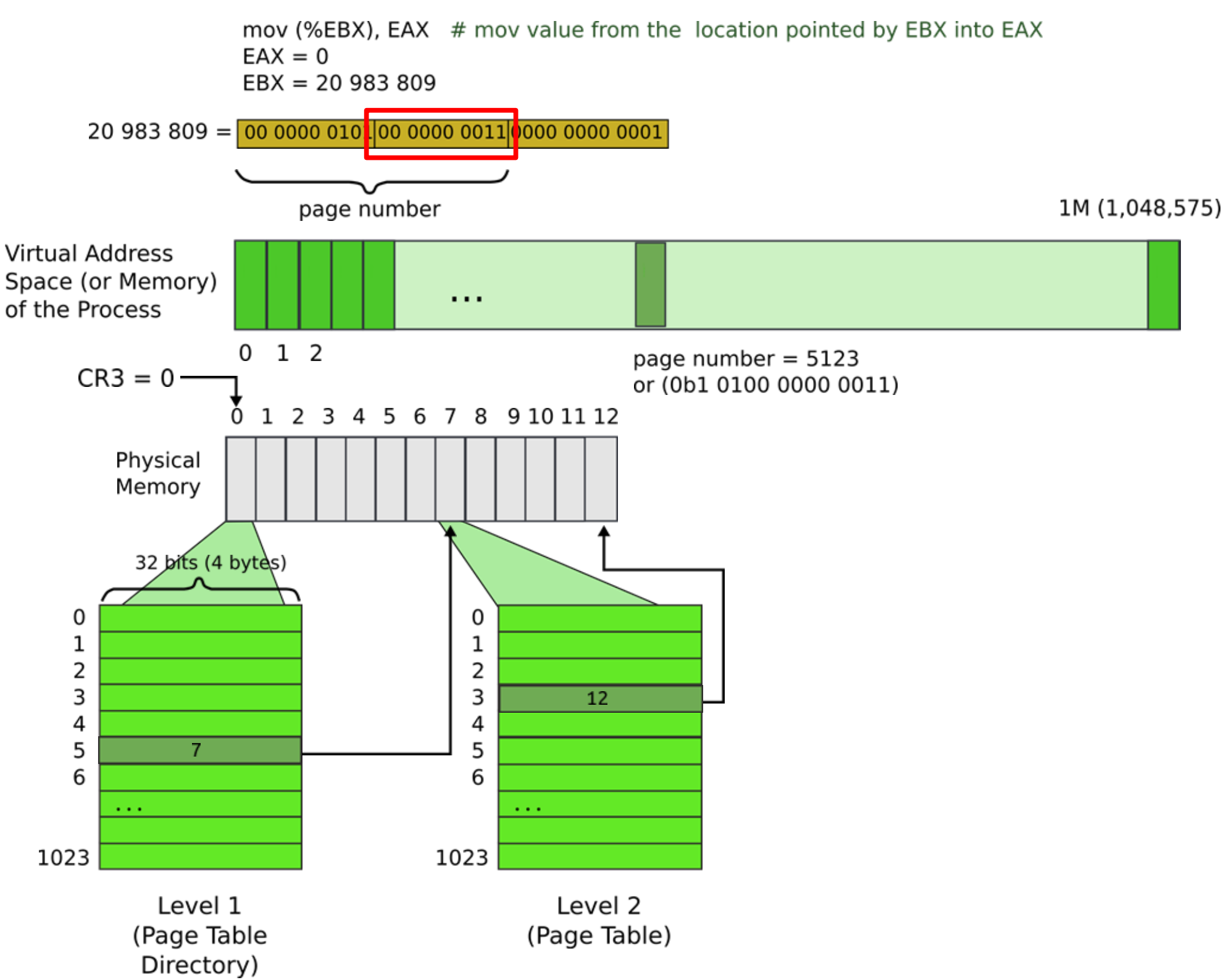


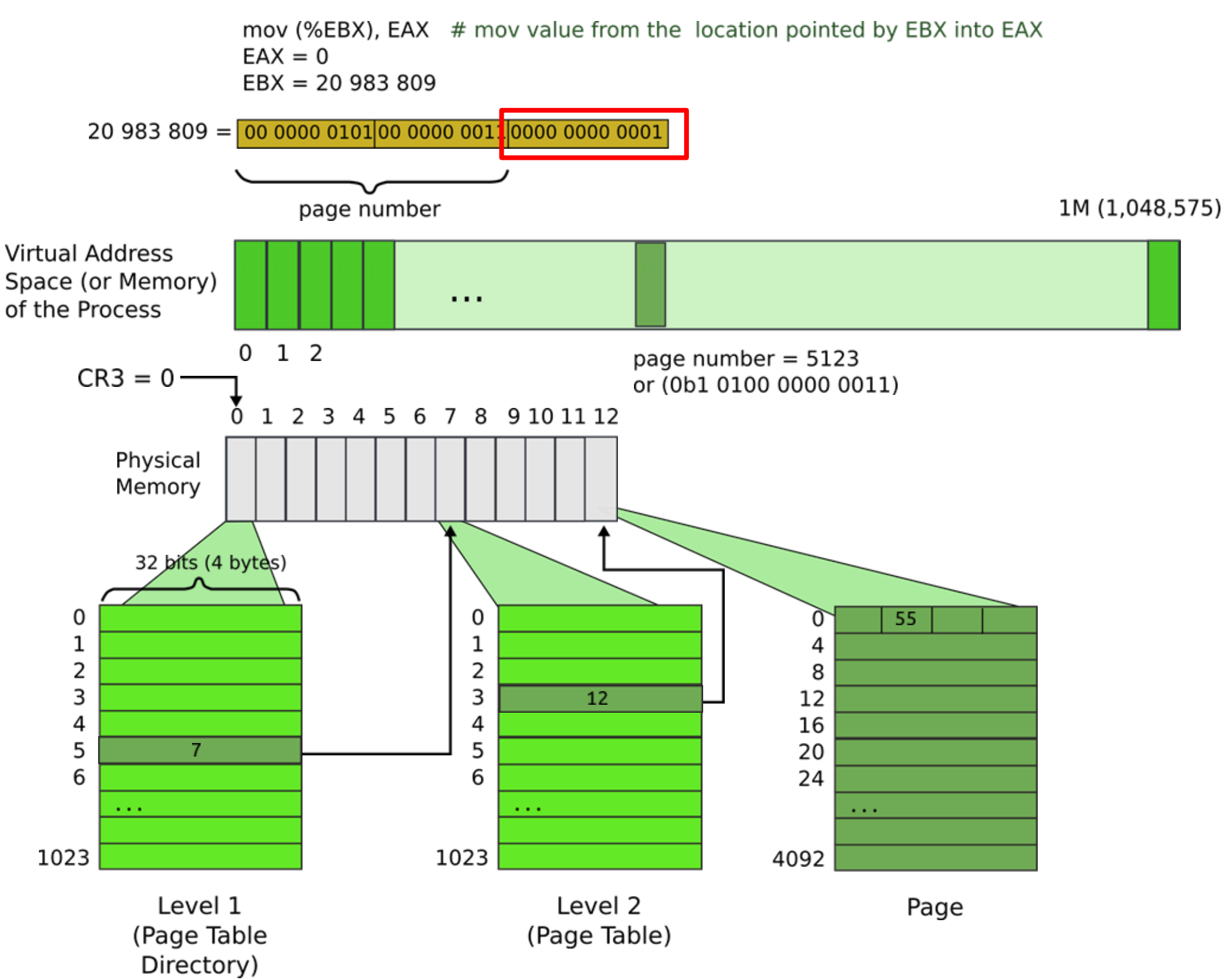
```
mov (%EBX), EAX   # mov value from the  location pointed by EBX into EAX
EAX = 0
EBX = 20 983 809
```

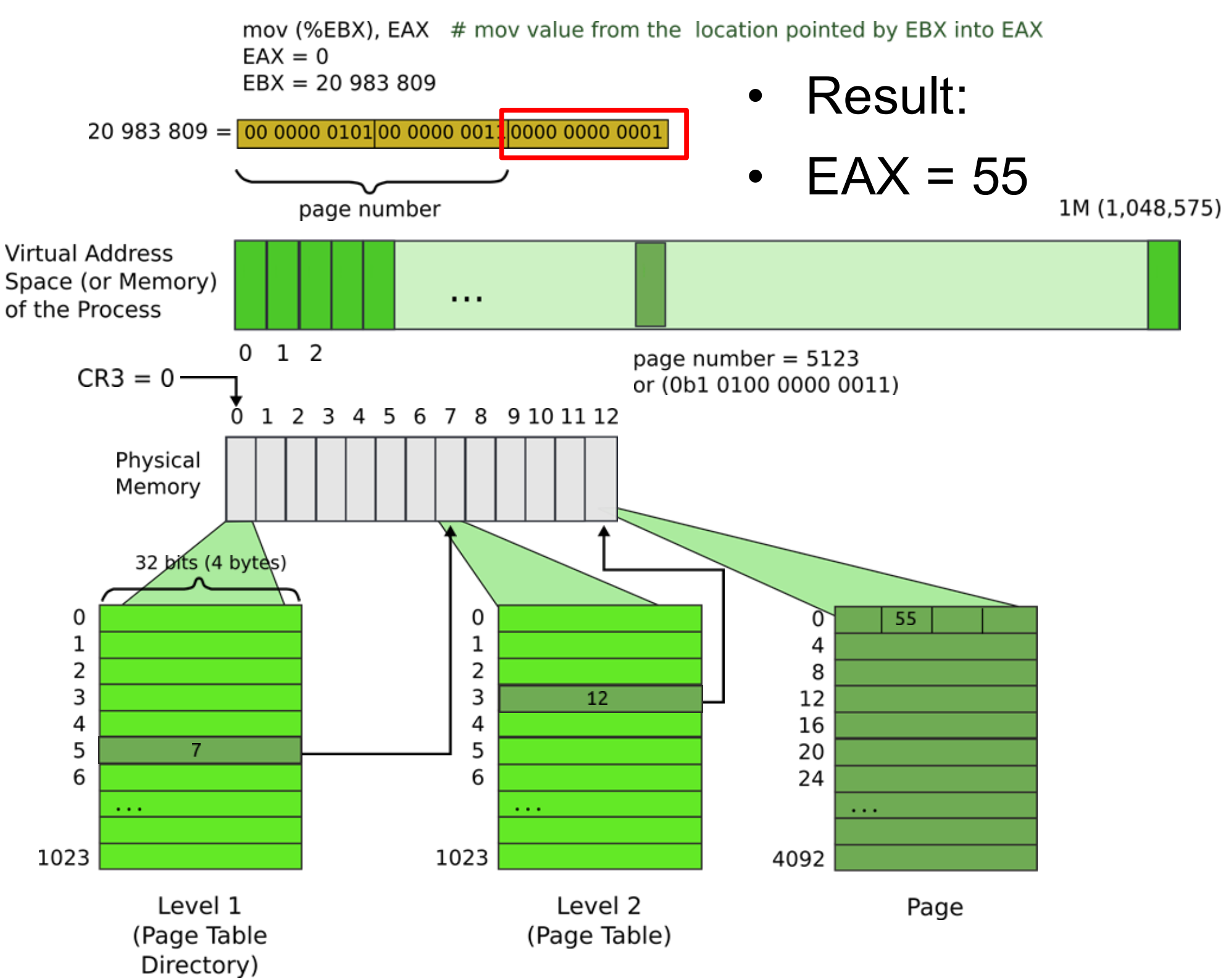


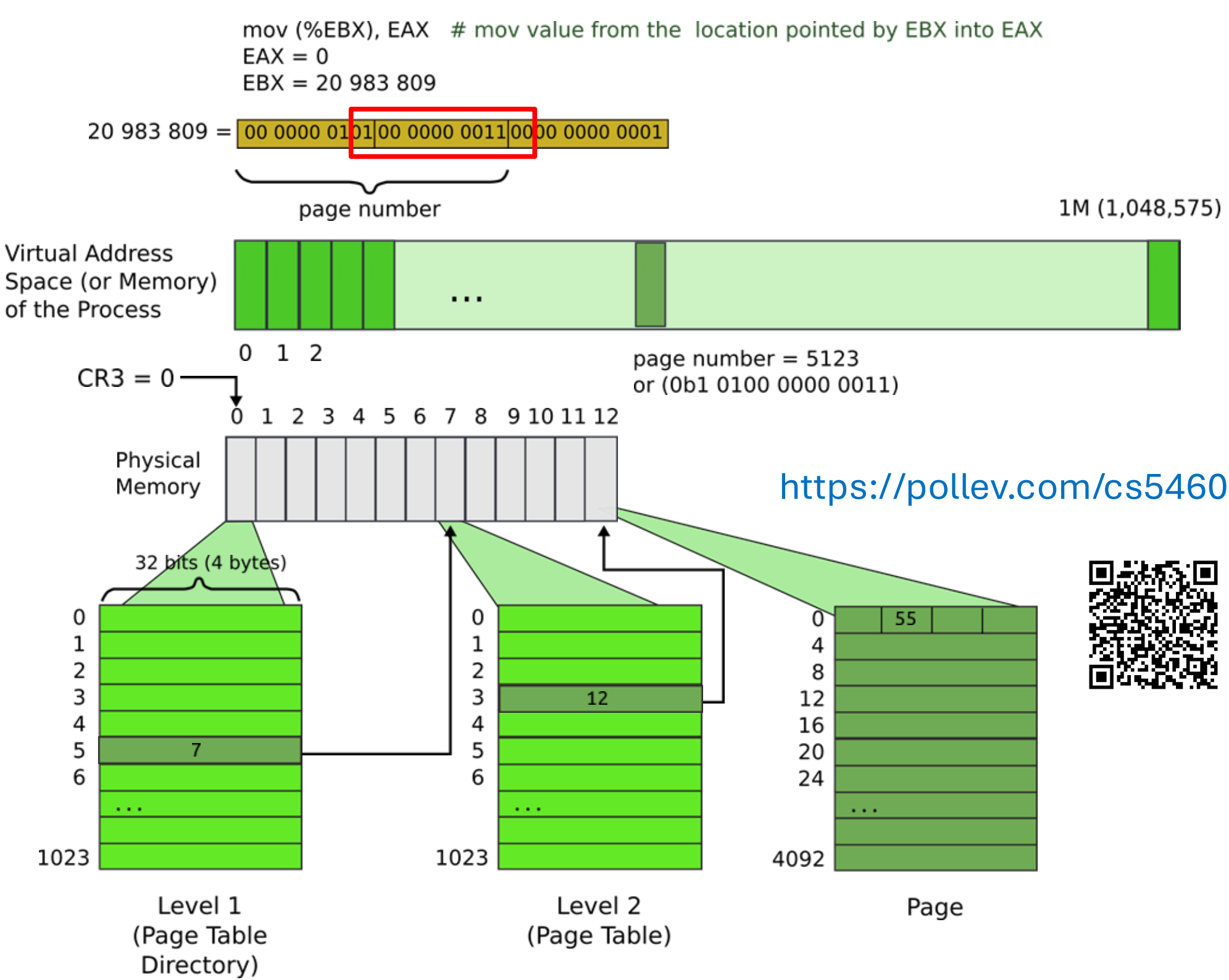

mov (%EBX), EAX # mov value from the location pointed by EBX into EAX
EAX = 0
EBX = 20 983 809
20 983 809 =
00 0000 0101
00 0000 0011
0000 0000 0001
page number
1M (1,048,575)
Virtual Address Space (or Memory) of the Process
...
0 1 2
page number = 5123
or (0b1 0100 0000 0011)
CR3 = 0
0 1 2 3 4 5 6 7 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)
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
32 bits (4 bytes)
0
1
2
3
4
5
6
7
1023
Level 1 (Page Table Directory)

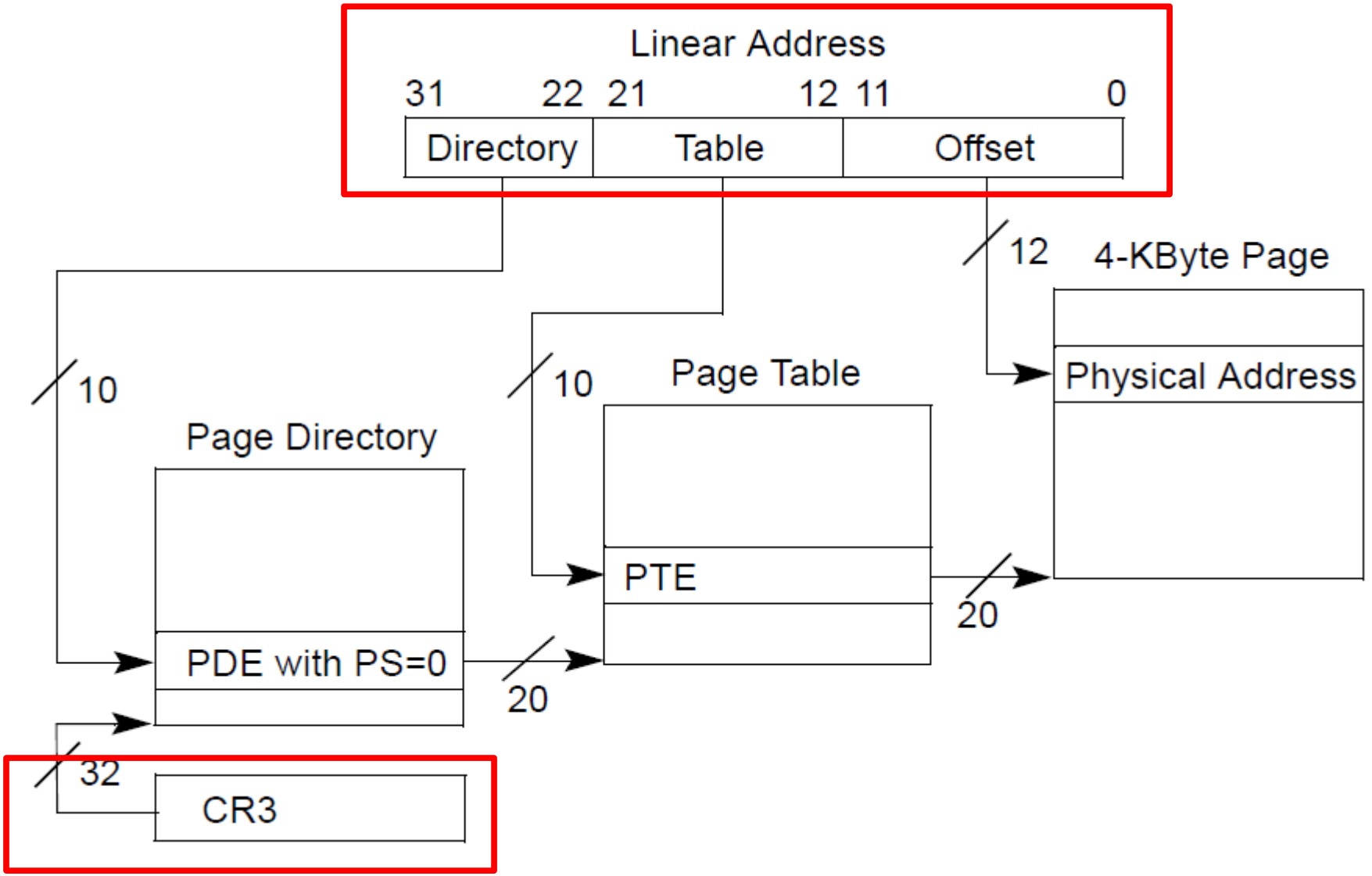

mov (%EBX), EAX # mov value from the location pointed by EBX into EAX
EAX = 0
EBX = 20 983 809
20 983 809 =
00 0000 0101
00 0000 0011
0000 0000 0001
page number
1M (1,048,575)
Virtual Address Space (or Memory) of the Process
0 1 2
page number = 5123
or (0b1 0100 0000 0011)
CR3 = 0
0 1 2 3 4 5 6 7 8 9 10 11 12
Physical Memory
32 bits (4 bytes)
0
1
2
3
4
5
6
7
1023
0
1
2
3
4
5
6
12
1023
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
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
32 bits (4 bytes)
0 1 2 3 4 5 6 7 ... 1023
0 1 2 3 4 5 6 12 ... 1023
0 4 8 12 16 20 24 55 ... 4092
Level 1 (Page Table Directory)
Level 2 (Page Table)
Page

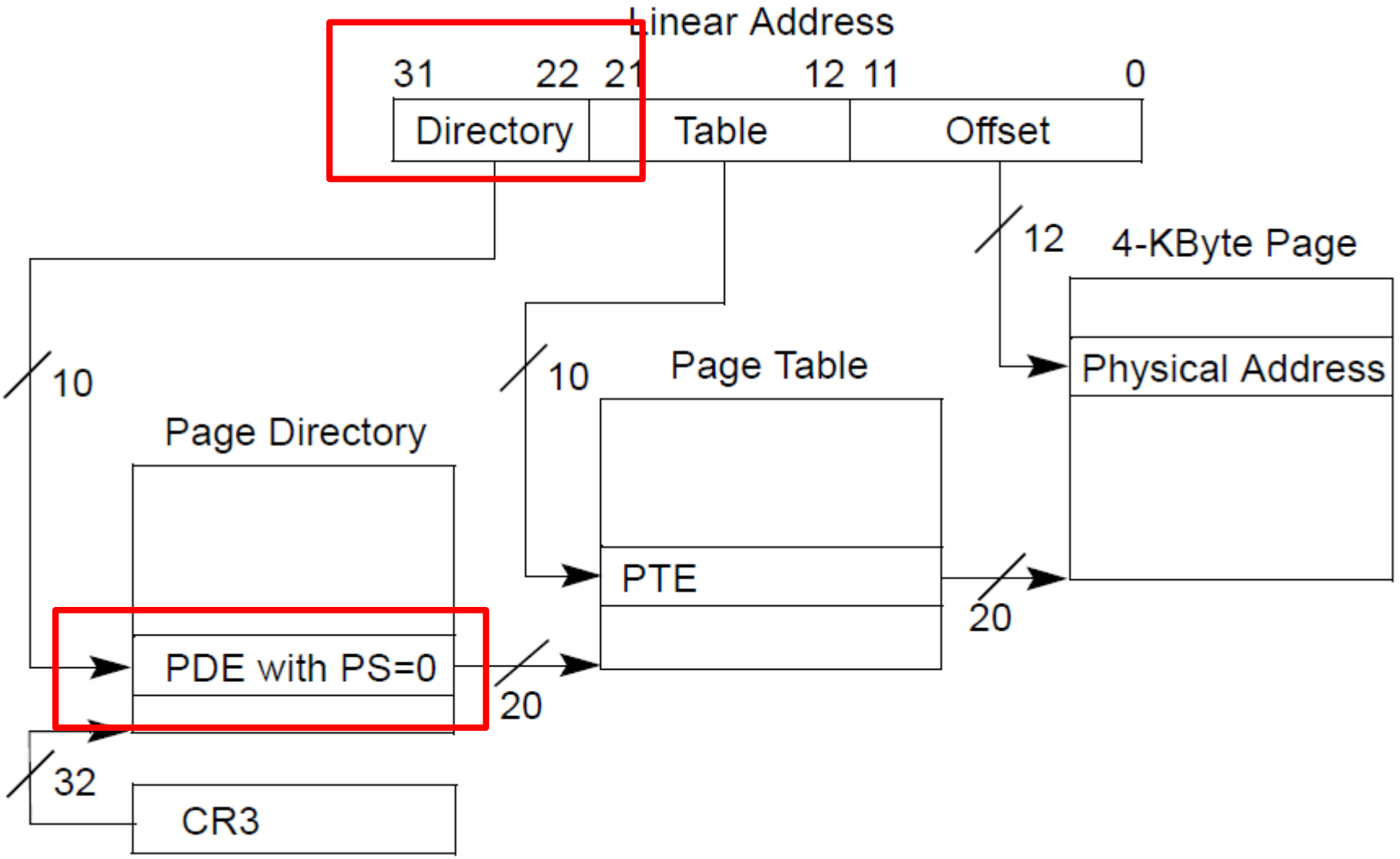

mov (%EBX), EAX # mov value from the location pointed by EBX into EAX
EAX = 0
EBX = 20 983 809
20 983 809 = 00 0000 0101 00 0000 0011 0000 0000 0001
page number
Result:
EAX = 55
1M (1,048,575)
Virtual Address Space (or Memory) of the Process
0 1 2
page number = 5123
or (0b1 0100 0000 0011)
CR3 = 0
0 1 2 3 4 5 6 7 8 9 10 11 12
Physical Memory
32 bits (4 bytes)
0 1 2 3 4 5 6 7 ... 1023
0 1 2 3 4 5 6 12 ... 1023
0 4 8 12 16 20 24 55 ... 4092
Level 1 (Page Table Directory)
Level 2 (Page Table)
Page


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
https://pollev.com/cs5460
32 bits (4 bytes)
0 1 2 3 4 5 6 7 ... 1023
0 1 2 3 4 5 6 12 ... 1023
0 4 8 12 16 20 24 55 ... 4092
Level 1 (Page Table Directory)
Level 2 (Page Table)
Page

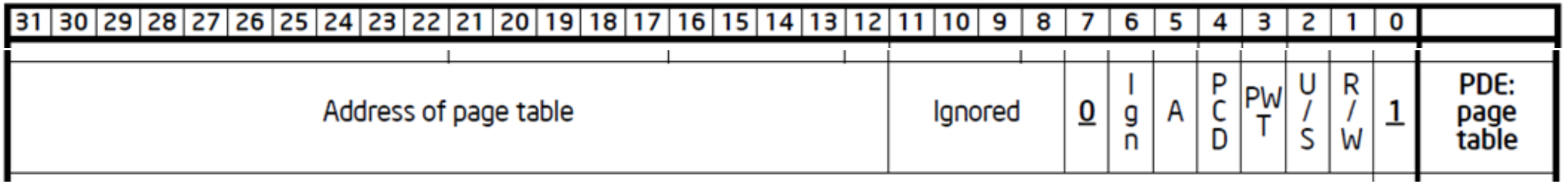
Page translation



Page translation

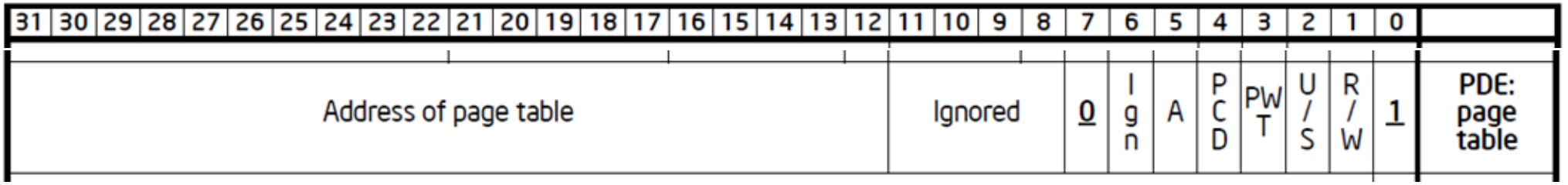


Page directory entry (PDE)



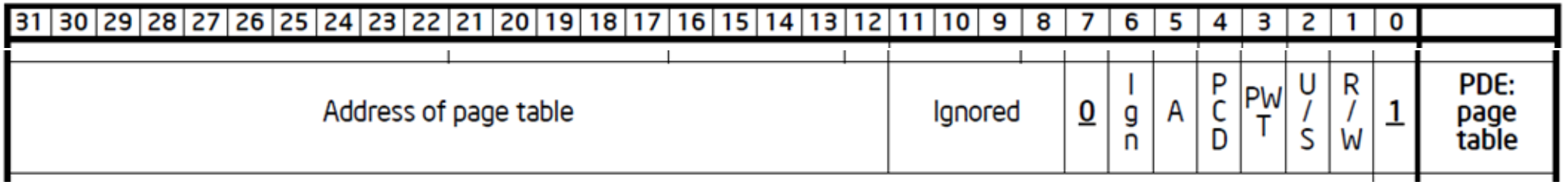
- 20 bit address of the page table

Page directory entry (PDE)



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

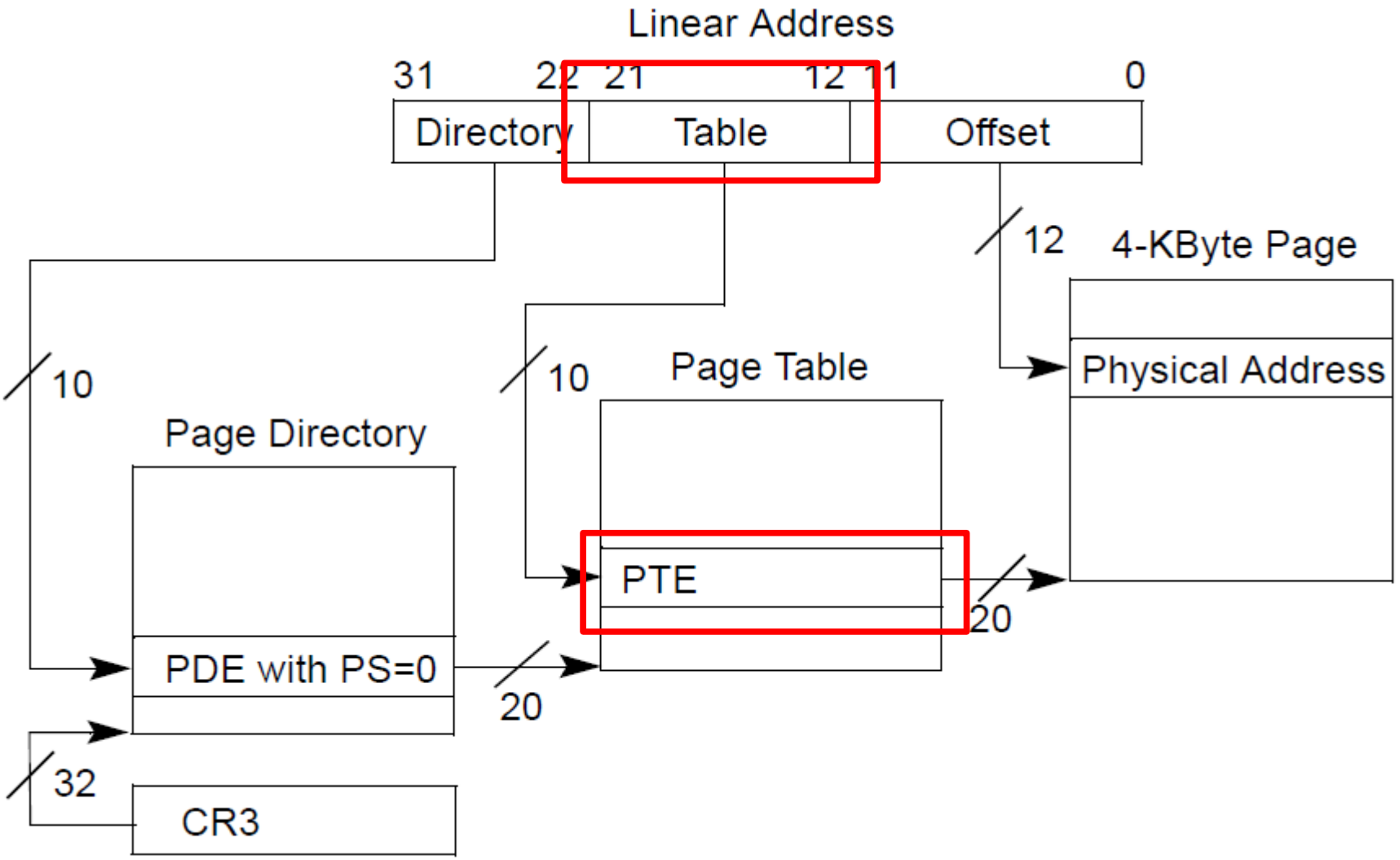
Page directory entry (PDE)



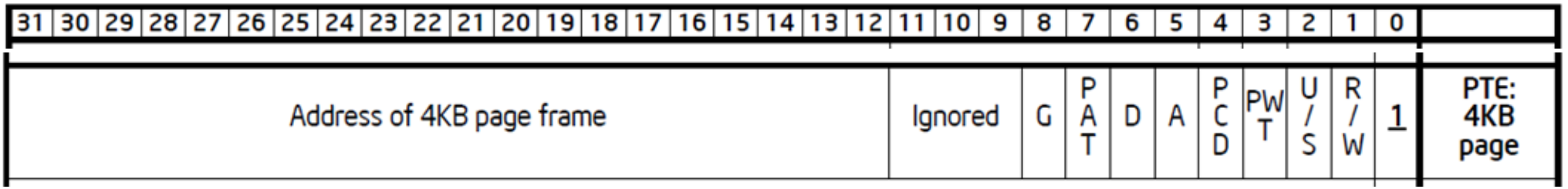
- 20 bit address of the page table
- Wait... 20 bit address, but don't we need 32 bits?

- Pages 4KB each, we need 1M to cover 4GB
- Pages start at 4KB (page aligned boundary)

Page translation

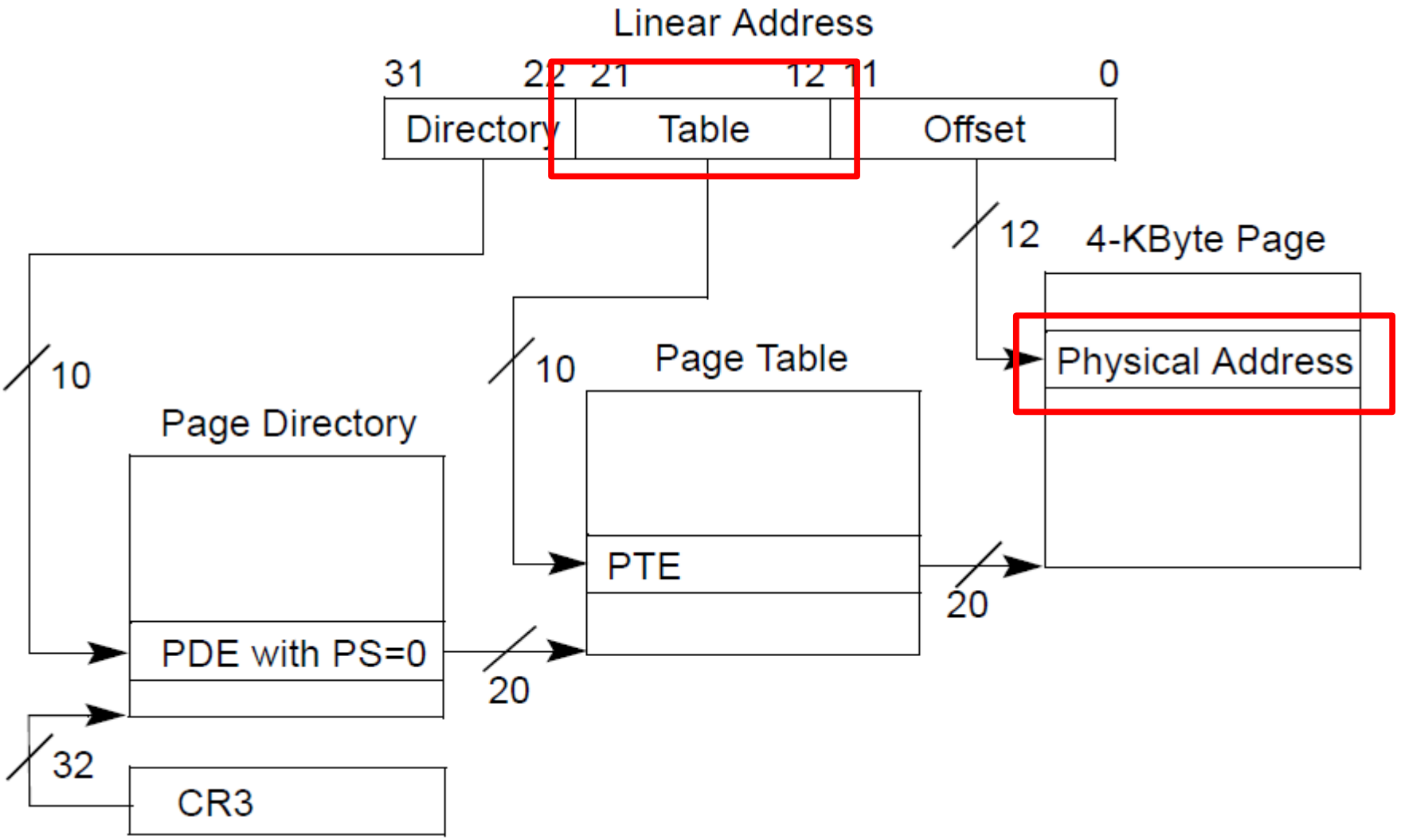


Page table entry (PTE)



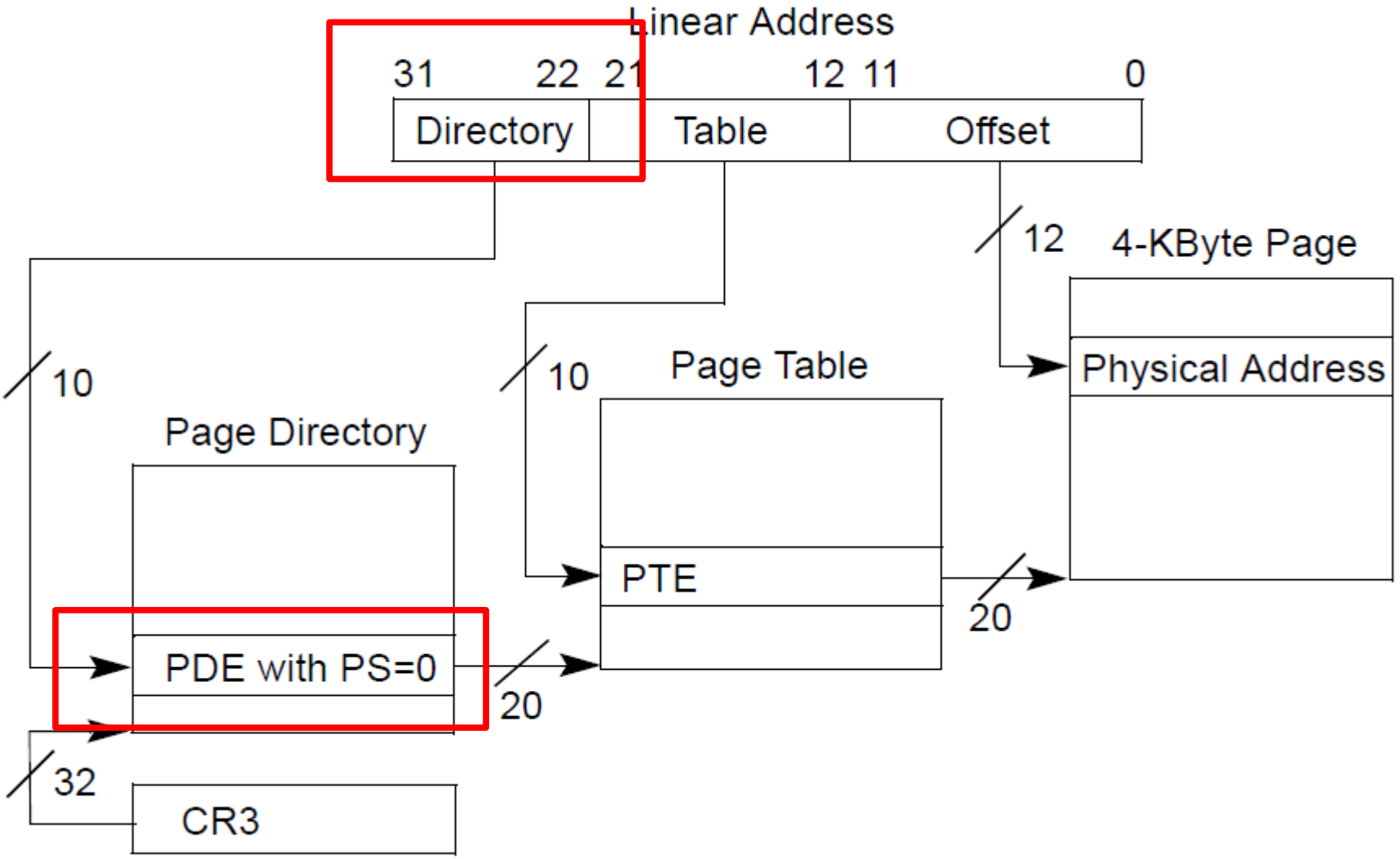
- 20 bit address of the 4KB page
- Pages 4KB each, we need 1M to cover 4GB

Page translation

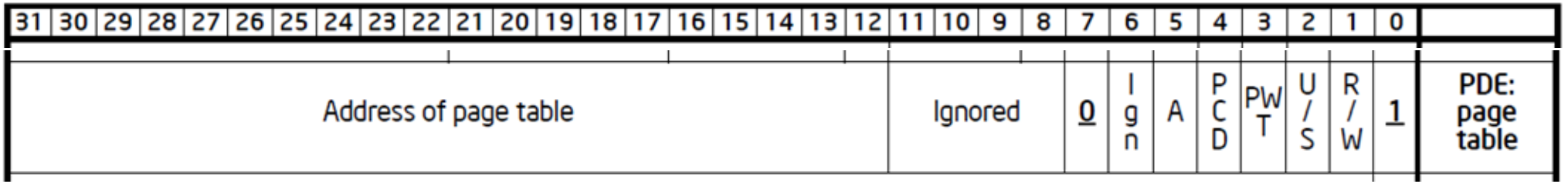


Flags

Page translation

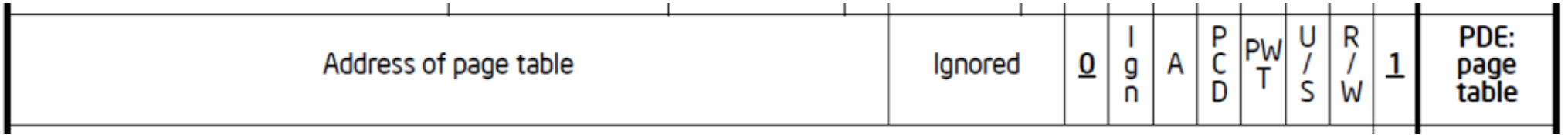


Page directory entry (PDE)



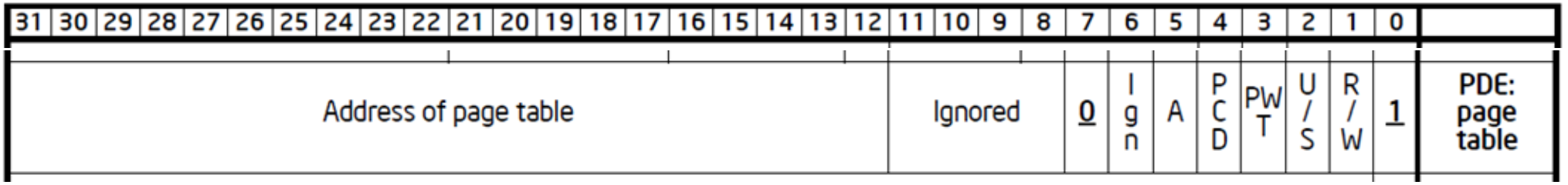
- Bit #1: R/W – writes allowed?
  - But allowed where?

Page directory entry (PDE)



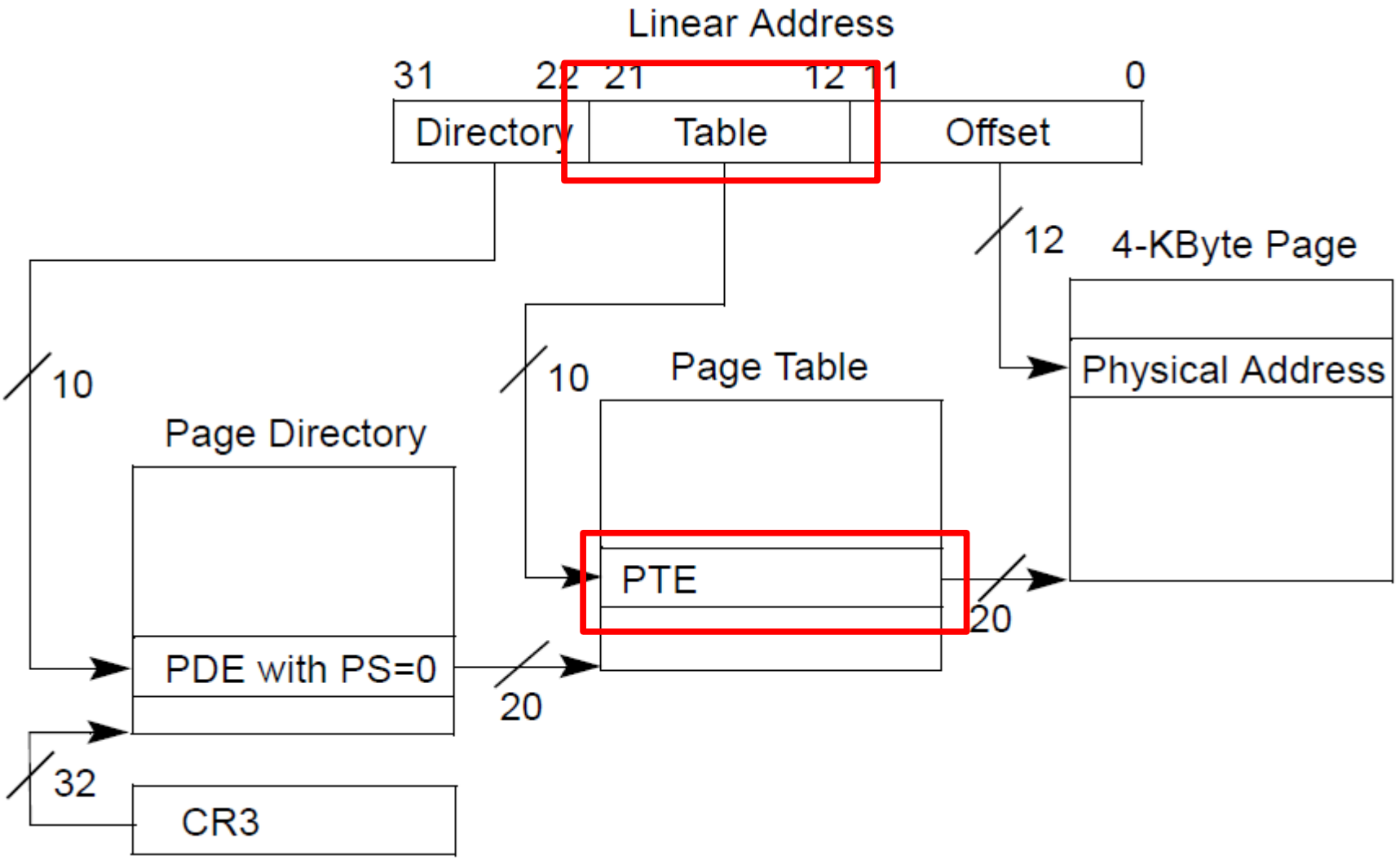
- Bit #1: R/W – writes allowed?
  - But allowed where?
- One page directory entry controls 1024 Level 2 page tables
  - Each Level 2 maps 4KB page
- So it's a region of 4KB x 1024 = 4MB

Page directory entry (PDE)

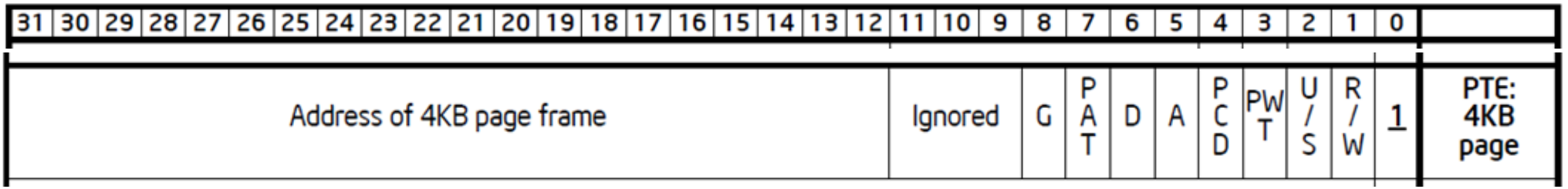


- Bit #2: U/S – user/supervisor
- If 0 – user-mode access is not allowed
- Allows protecting kernel memory from user-level applications

Page translation



Page table entry (PTE)



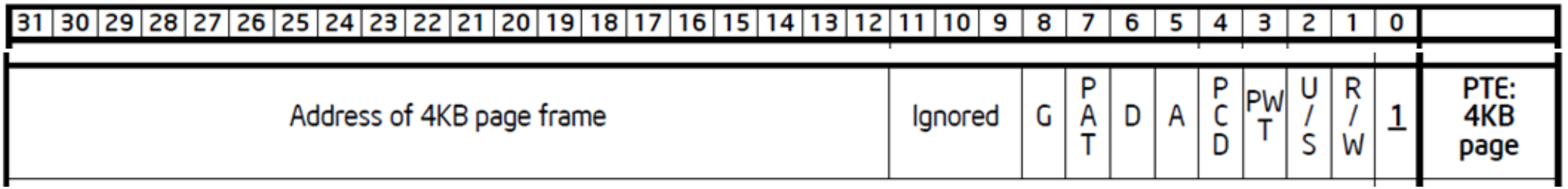
- 20 bit address of the 4KB page
  - Pages 4KB each, we need 1M to cover 4GB
- Bit #1: R/W – writes allowed?
  - To a 4KB page
- Bit #2: U/S – user/supervisor
  - If 0 user-mode access is not allowed
- Bit #5: A – accessed

Paging tricks

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

- Determine a set of pages that were updated
- This is useful for virtual machine migration
  - Use “dirty” bit

Page table entry (PTE)



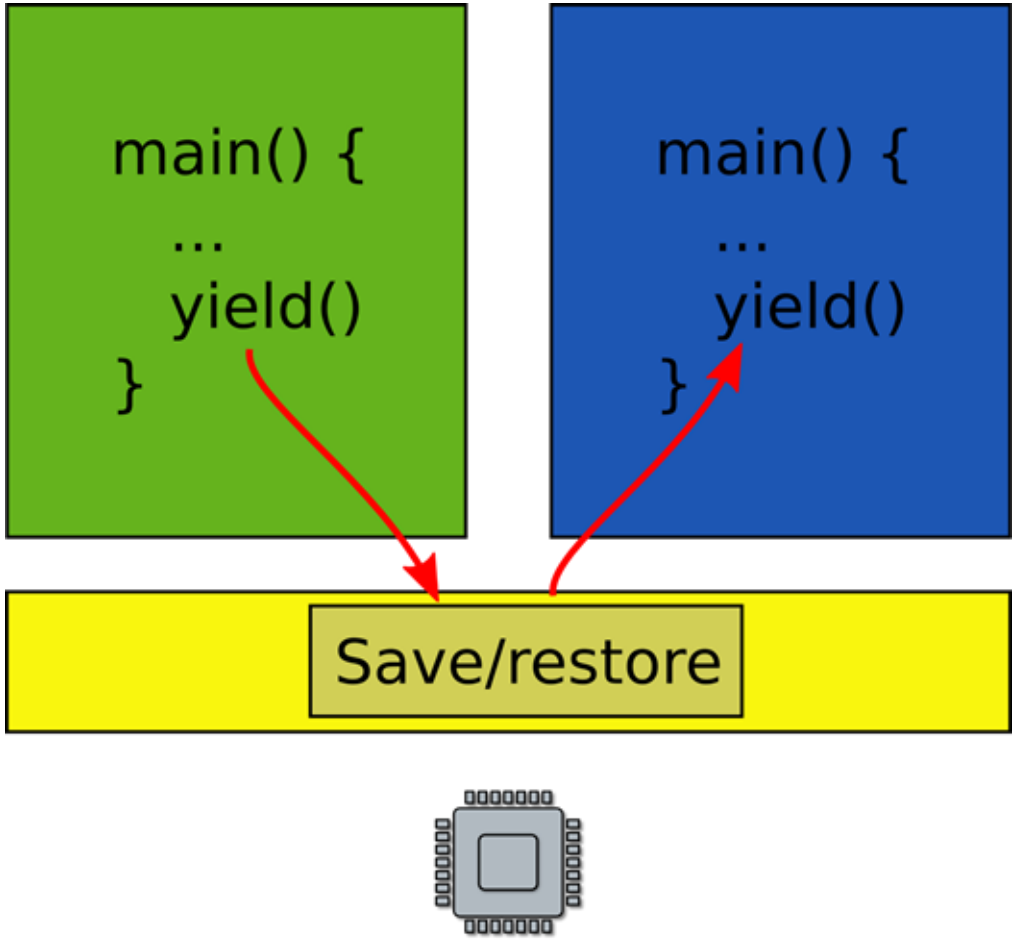
- 20 bit address of the 4KB page
  - Pages 4KB each, we need 1M to cover 4GB
- Bit #1: R/W – writes allowed?
  - To a 4KB page
- Bit #2: U/S – user/supervisor
  - If 0 user-mode access is not allowed
- Bit #5: A – accessed
- Bit #6: D – dirty – software has written to this page

Benefits of page tables

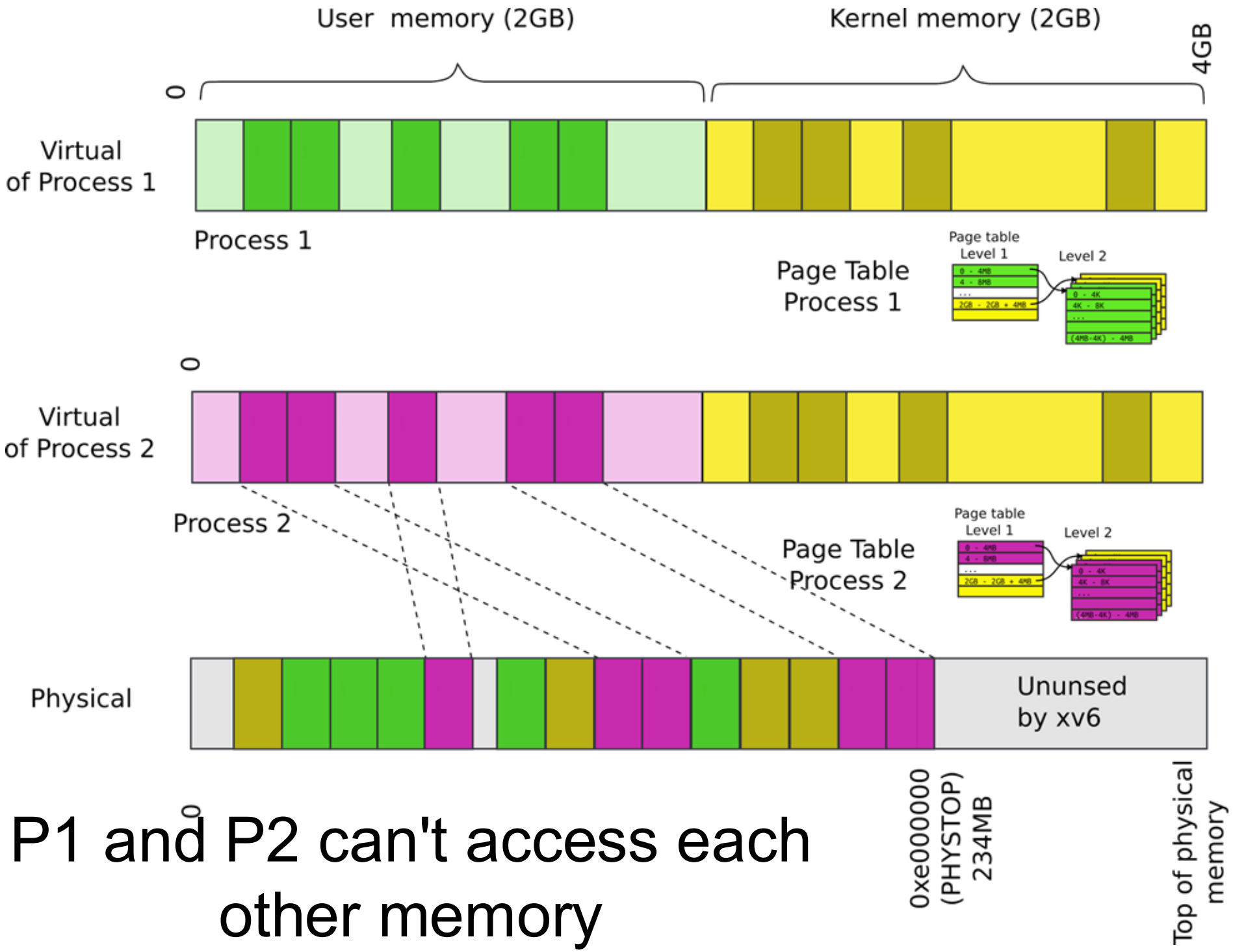
… Compared to arrays?

- 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



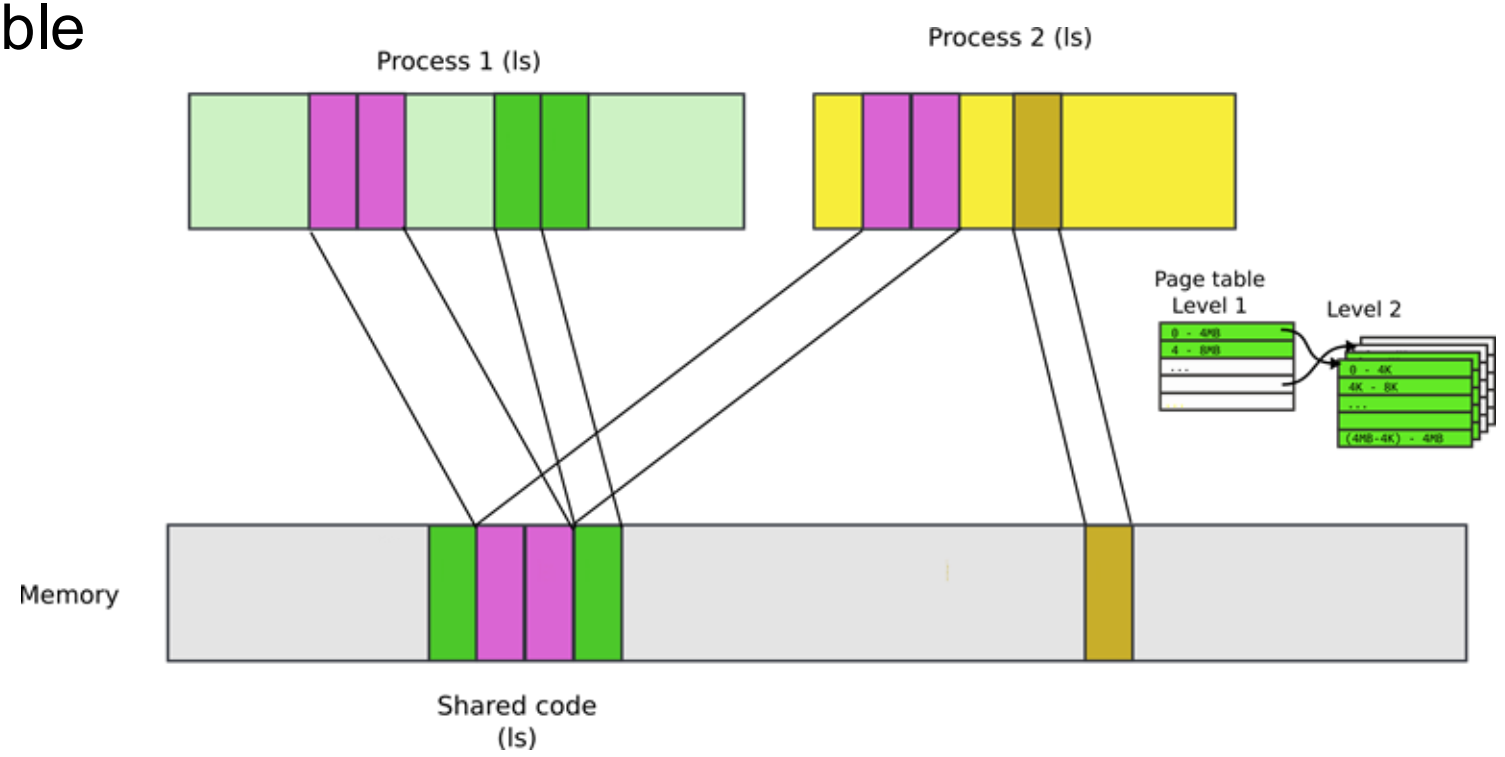
P1 and P2 can't access each other memory

Compared to segments pages allow ...

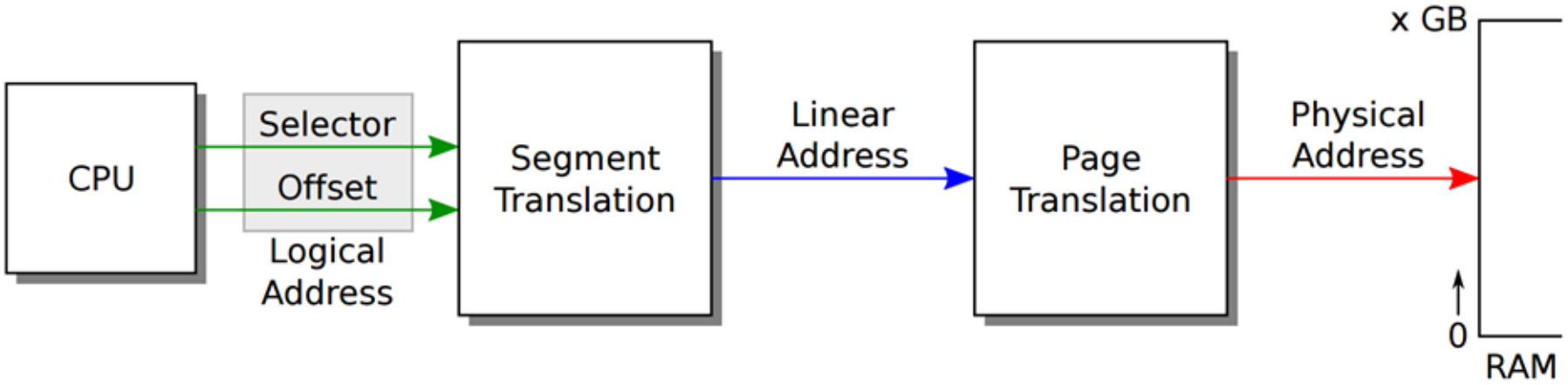
- Emulate large virtual address space on a smaller physical memory
- In our example we had only 12 physical pages
- But the program can access all 1M pages in its 4GB address space
- The OS will move other pages to disk

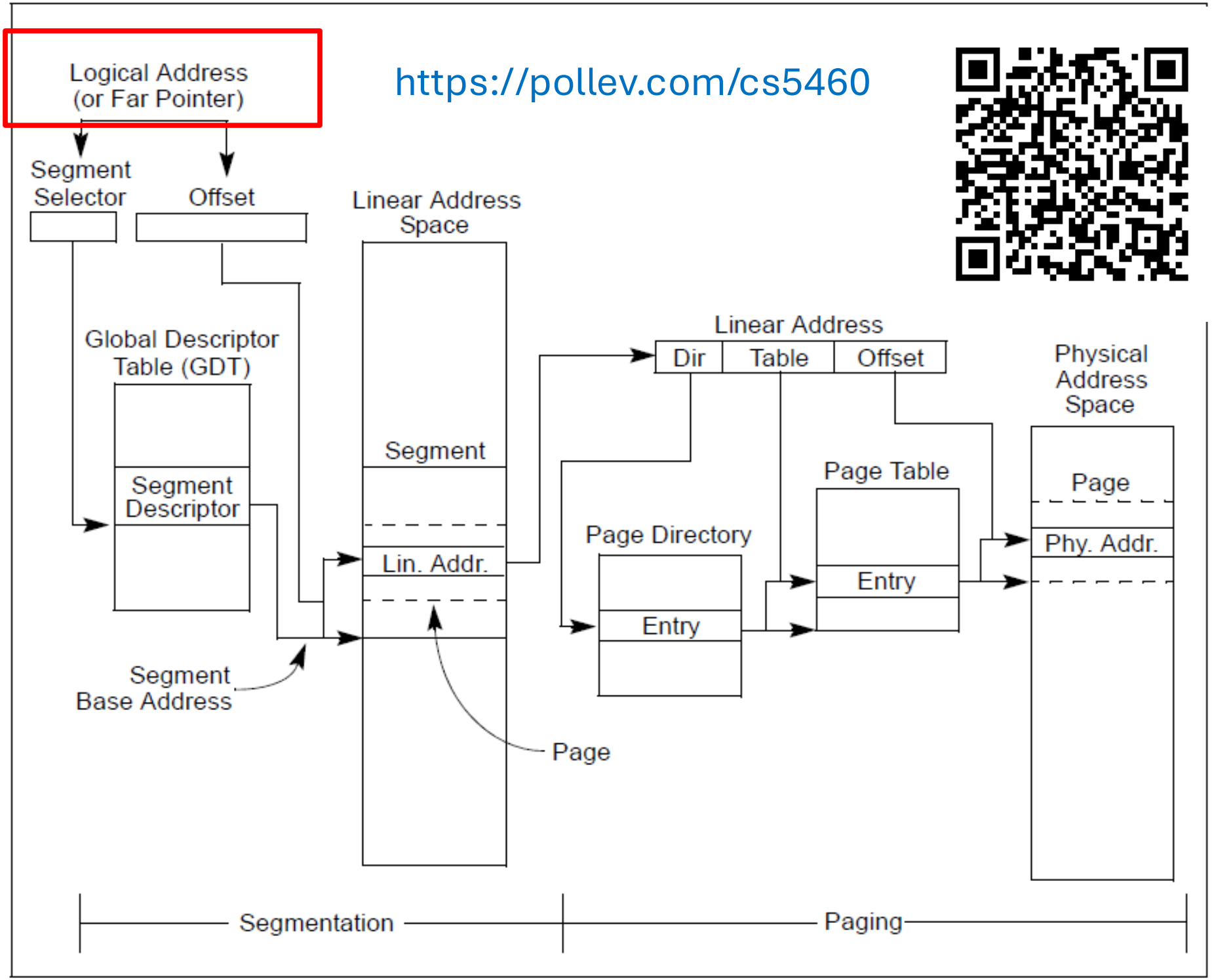
Compared to segments, pages …

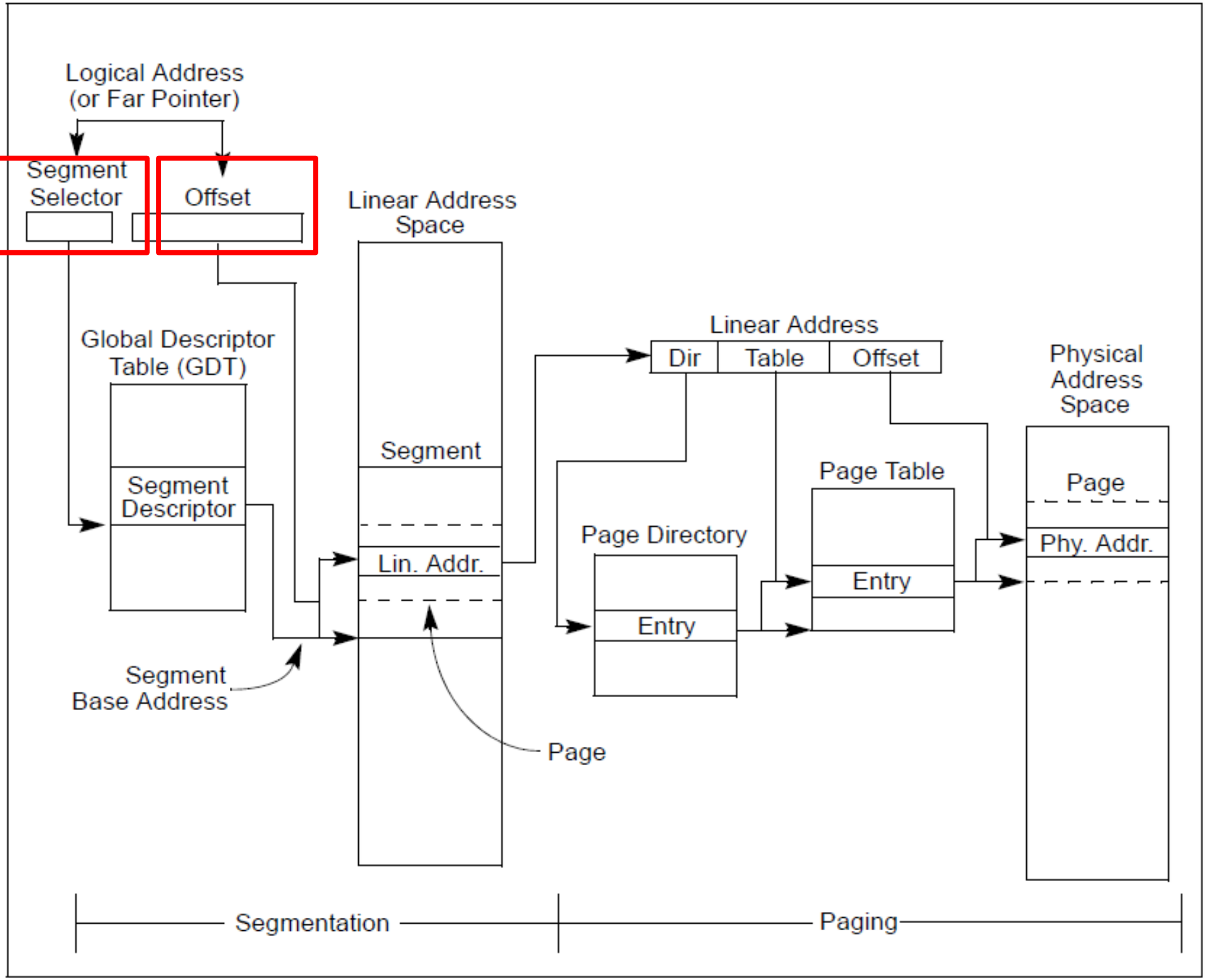
- Share a region of memory across multiple programs
  - Communication (shared memory to exchange messages, fast frame buffer)
  - Shared libraries
- Well… segmentation can do this too, but paging is a bit more flexible

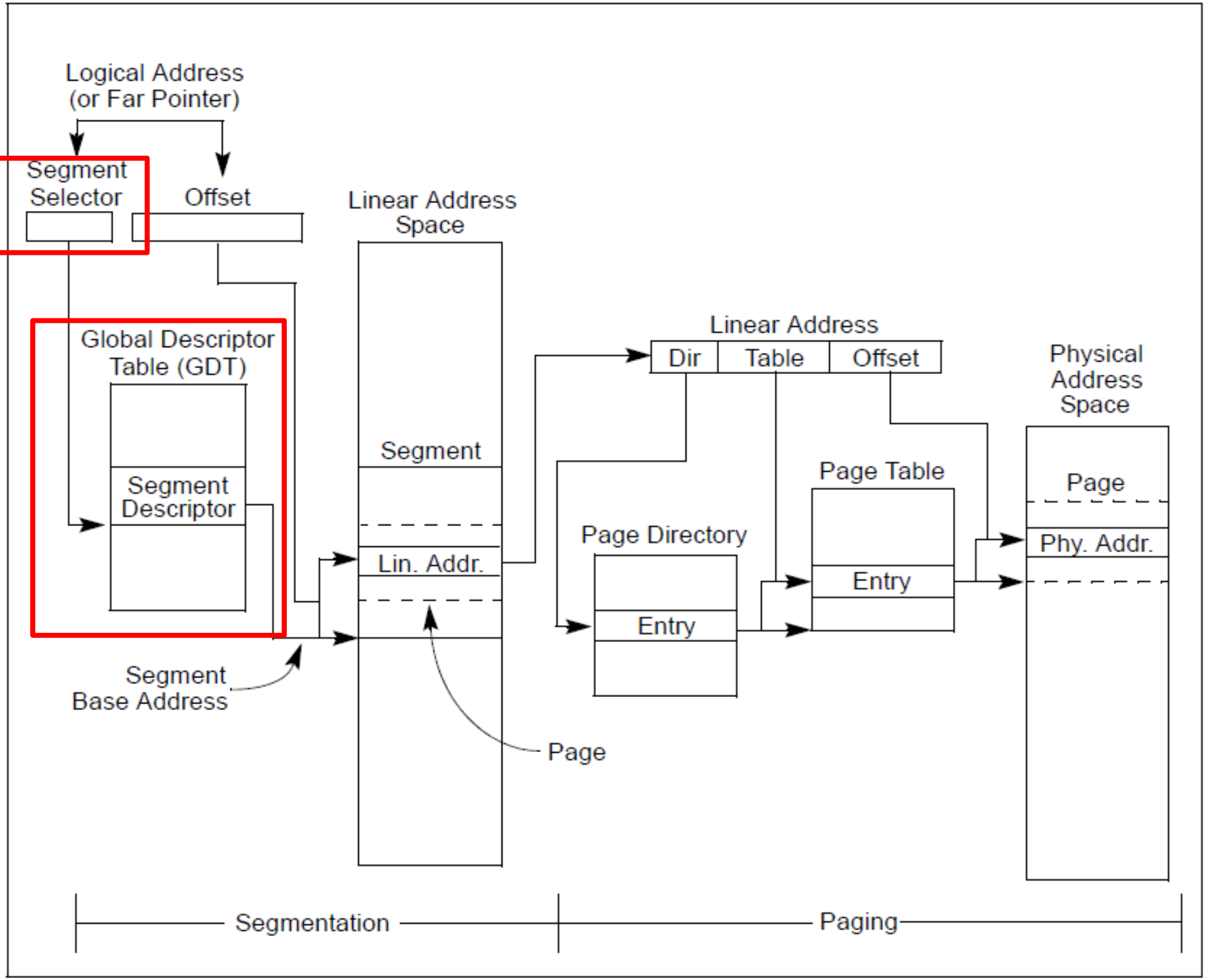


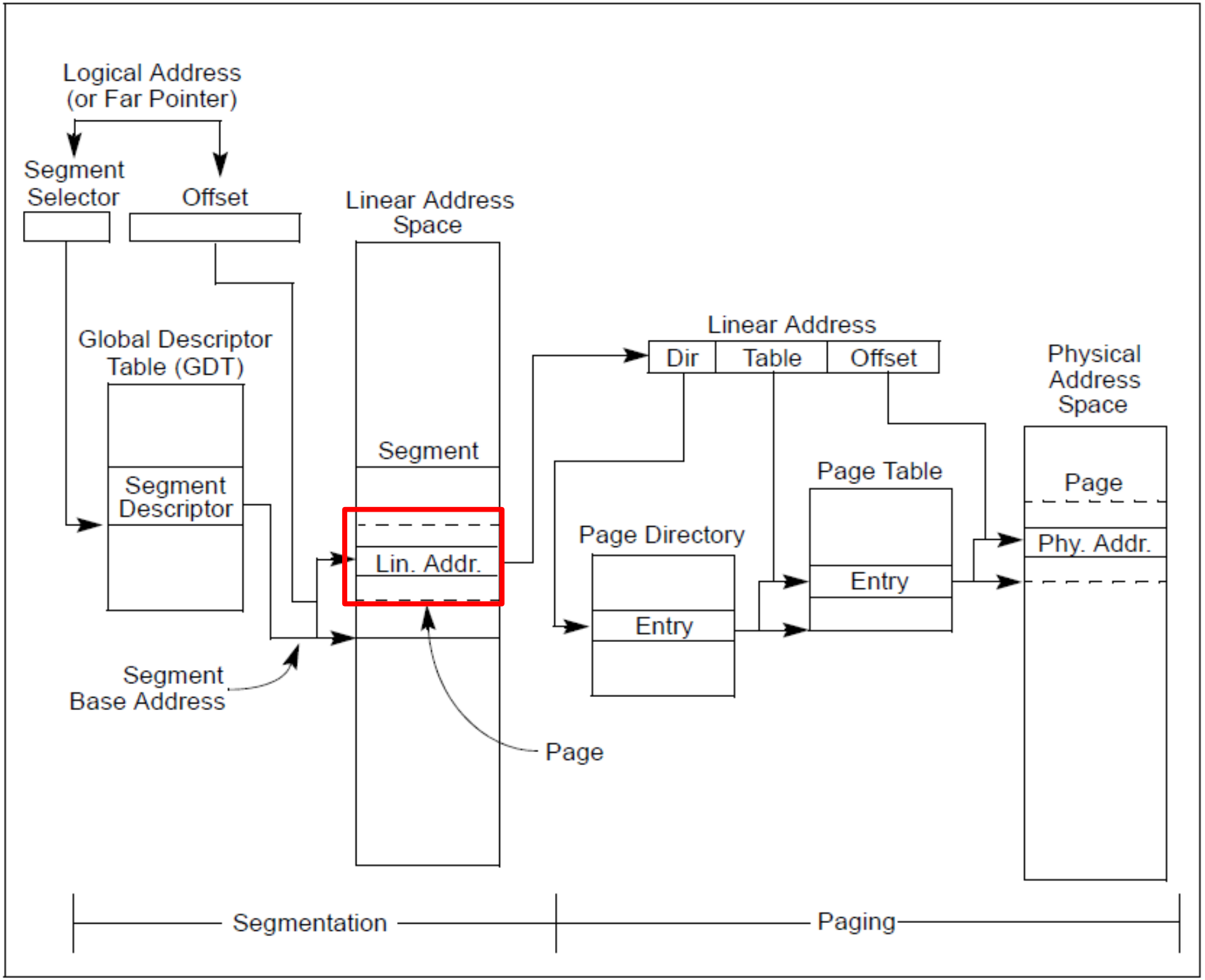
Recap: complete address translation

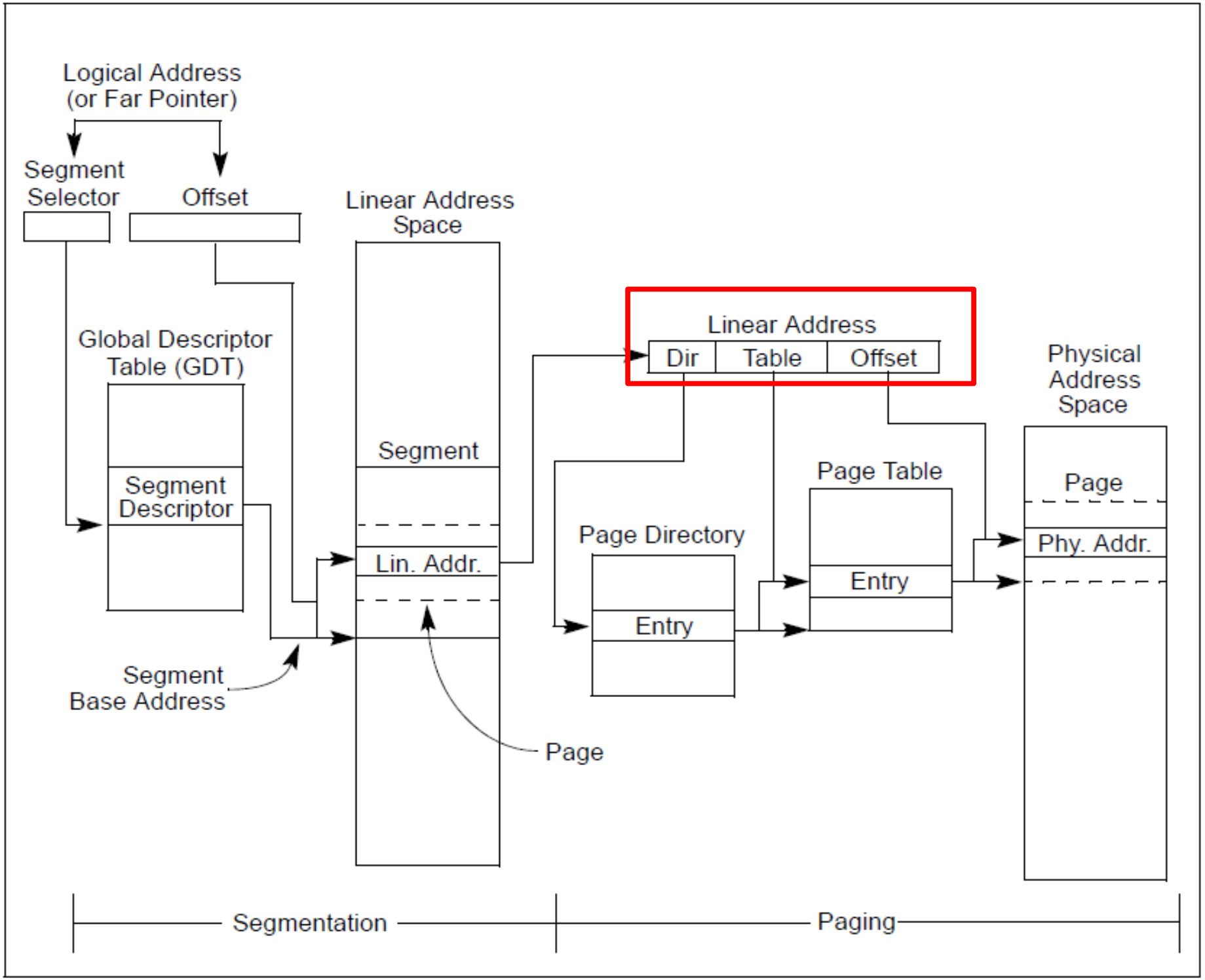


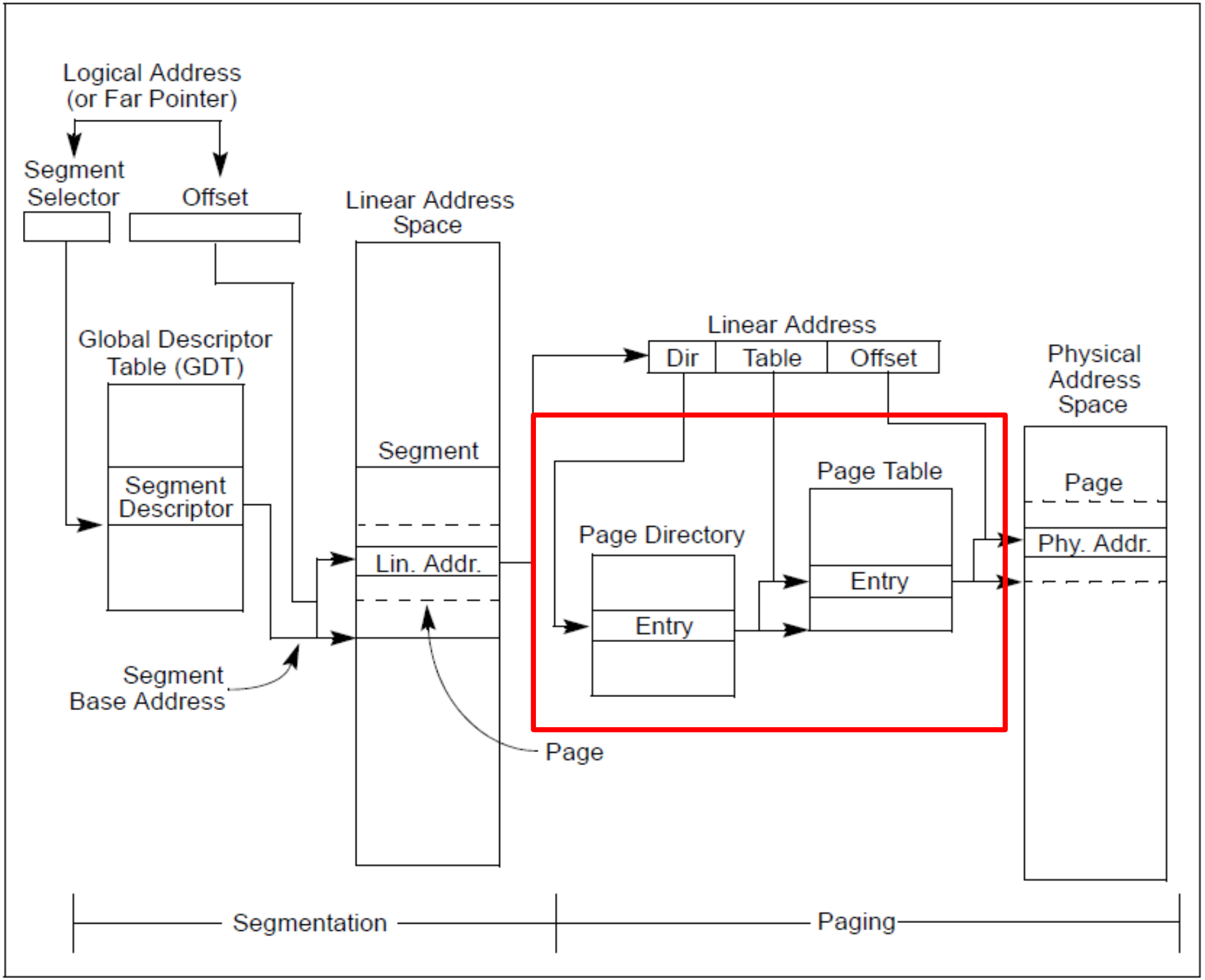

Logical Address
(or Far Pointer)
https://pollev.com/cs5460
Segment
Selector
Offset
Linear Address
Space
Global Descriptor
Table (GDT)
Segment
Descriptor
Segment
Lin. Addr.
Linear Address
Dir
Table
Offset
Physical
Address
Space
Page Table
Page
Page Directory
Phy. Addr.
Entry
Entry
Segment
Base Address
Page
Segmentation
Paging

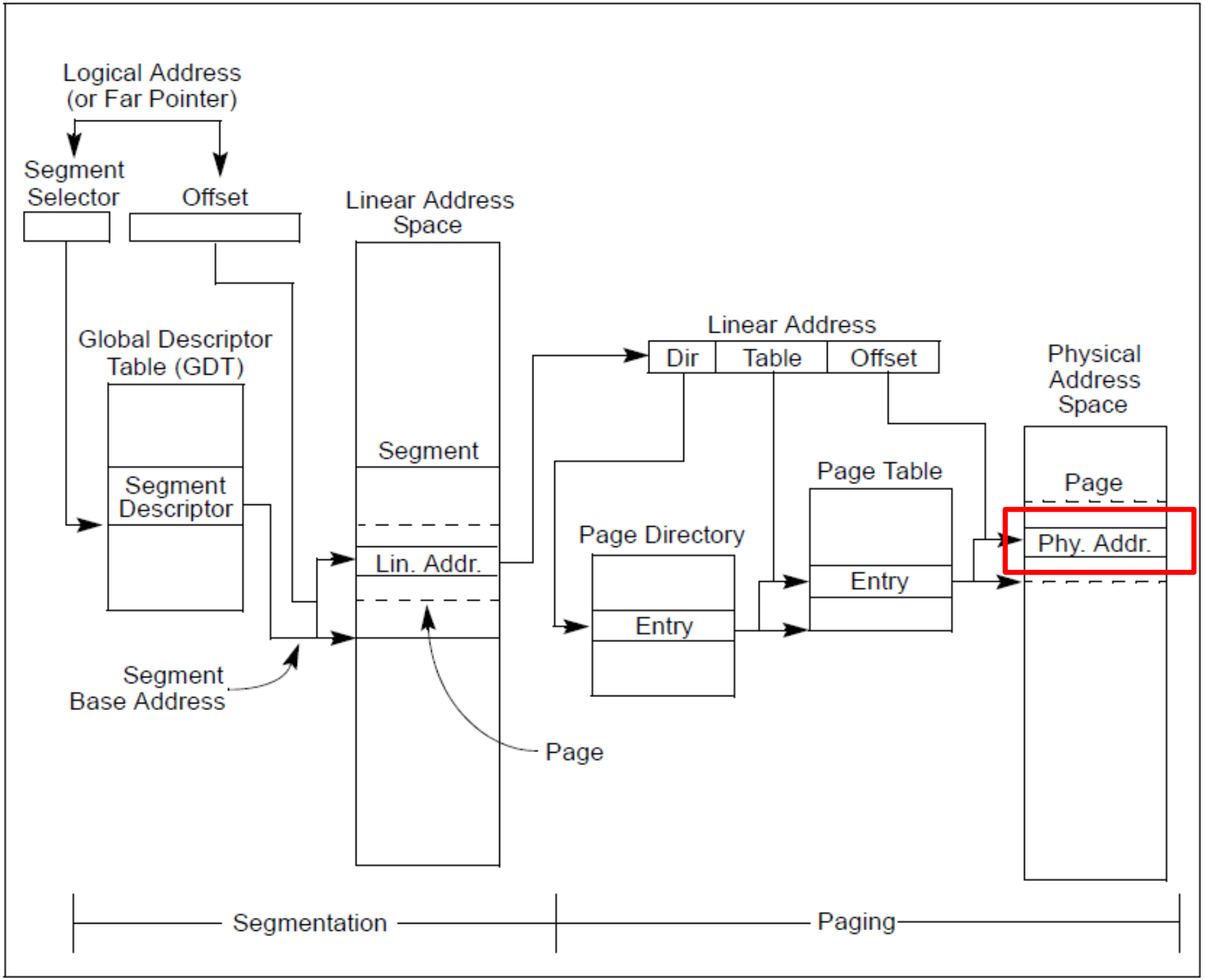

Logical Address
(or Far Pointer)
Segment
Selector
Offset
Linear Address
Space
Global Descriptor
Table (GDT)
Segment
Descriptor
Segment
Lin. Addr.
Linear Address
Dir
Table
Offset
Physical
Address
Space
Page Table
Page
Phy. Addr.
Page Directory
Entry
Entry
Segment
Base Address
Page
Segmentation
Paging

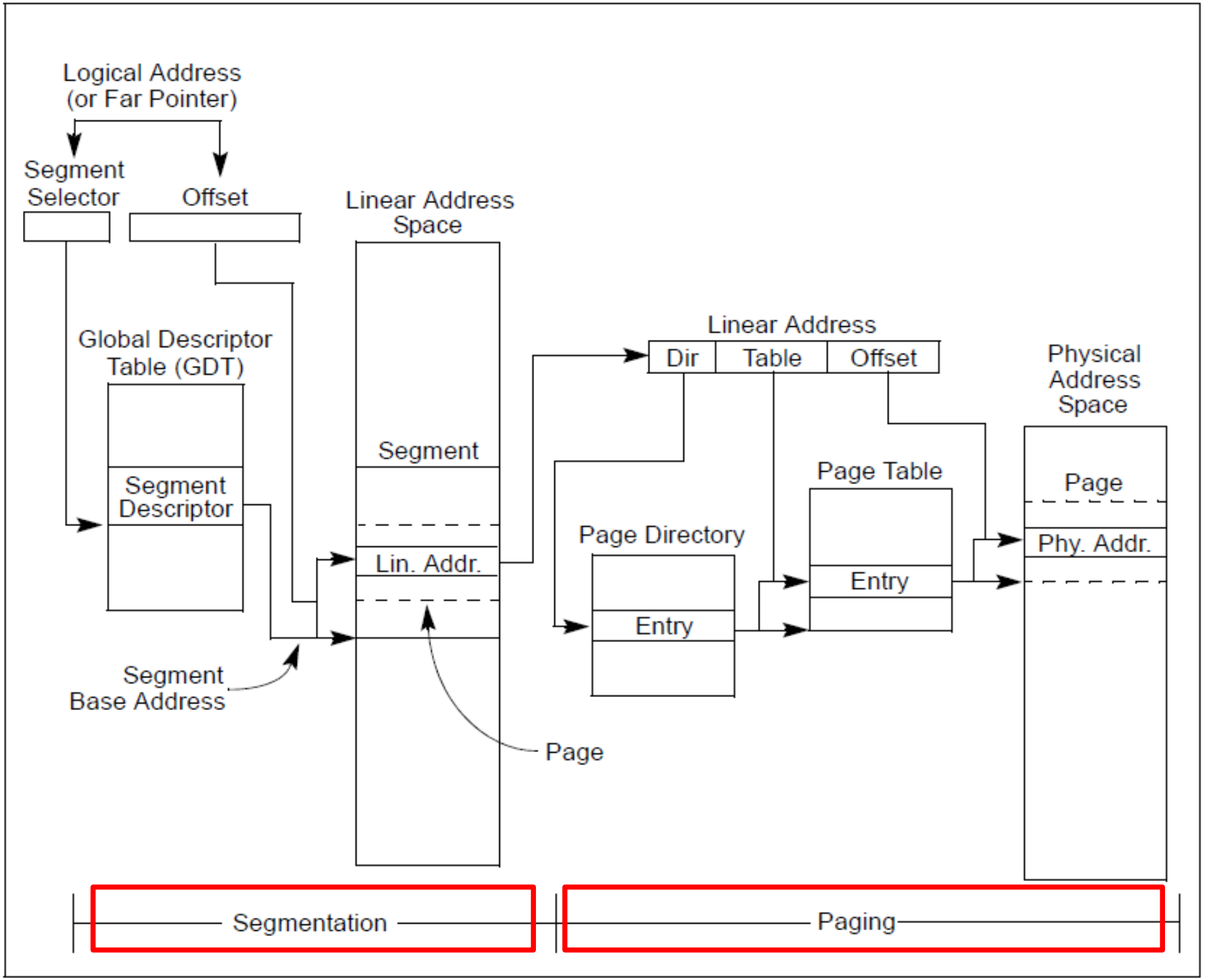

Logical Address
(or Far Pointer)
Segment
Selector
Offset
Linear Address
Space
Global Descriptor
Table (GDT)
Segment
Descriptor
Segment
Lin. Addr.
Segment
Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical
Address
Space
Page
Phy. Addr.
Segmentation
Paging


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

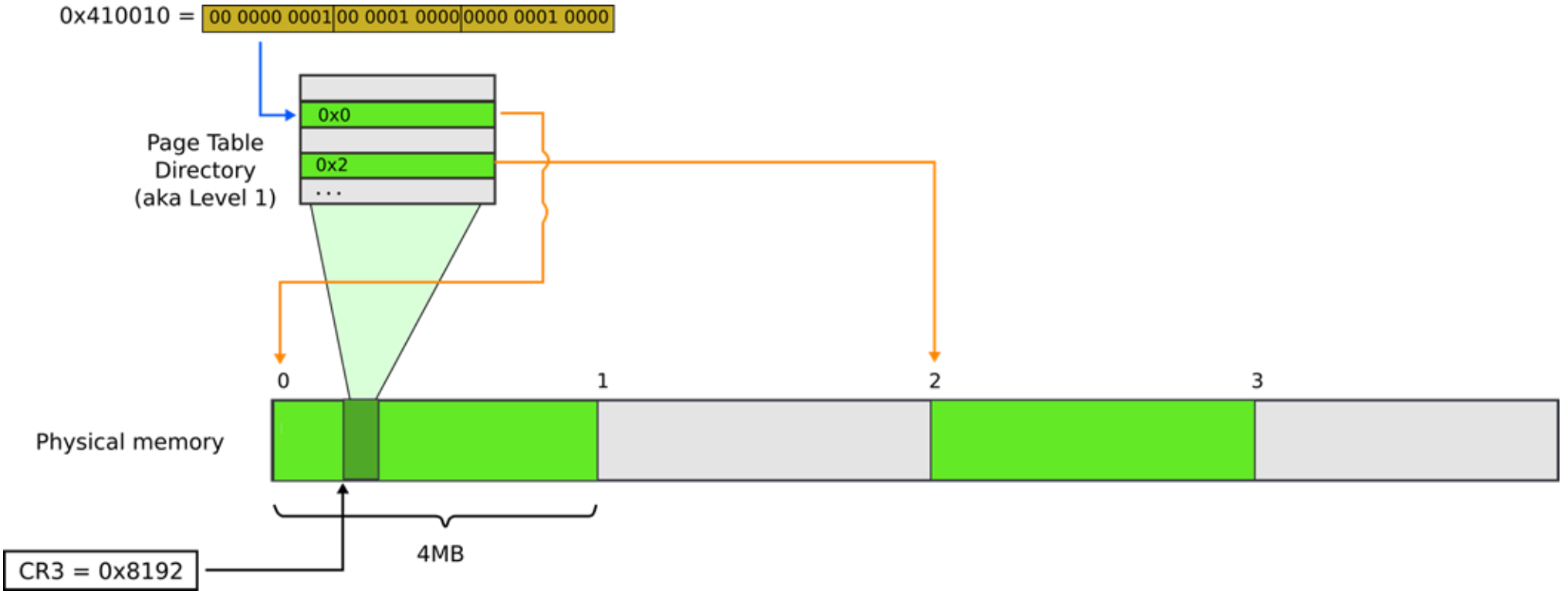

Logical Address
(or Far Pointer)
Segment
Selector
Offset
Linear Address
Space
Global Descriptor
Table (GDT)
Segment
Descriptor
Segment
Lin. Addr.
Linear Address
Dir
Table
Offset
Physical
Address
Space
Page Table
Page
Phy. Addr.
Page Directory
Entry
Entry
Segment
Base Address
Page
Segmentation
Paging


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

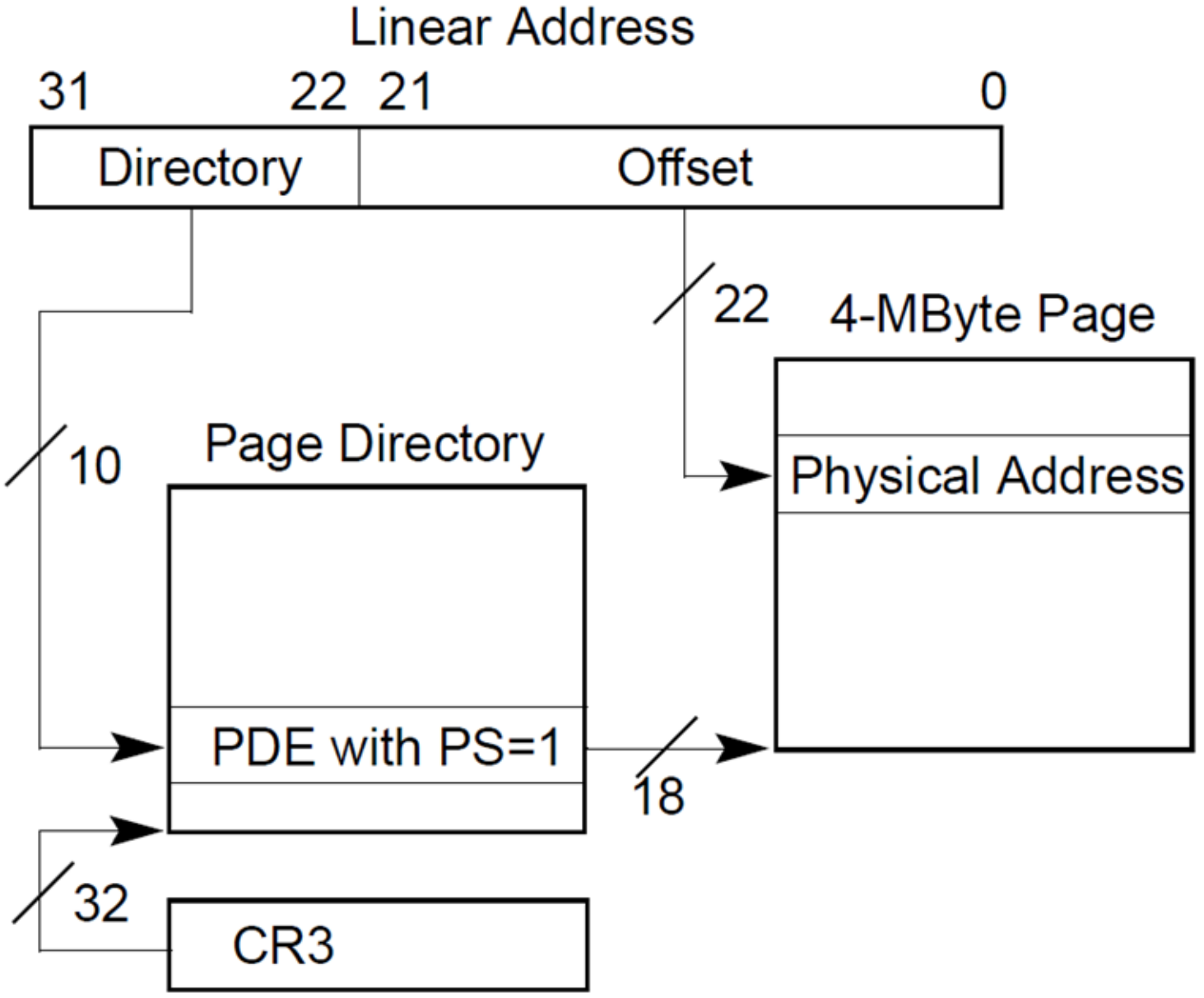
32bit x86 supports two page sizes

- 4KB pages
- 4MB pages

Page translation for 4MB pages



Page translation for 4MB pages



Paging in 64bit mode

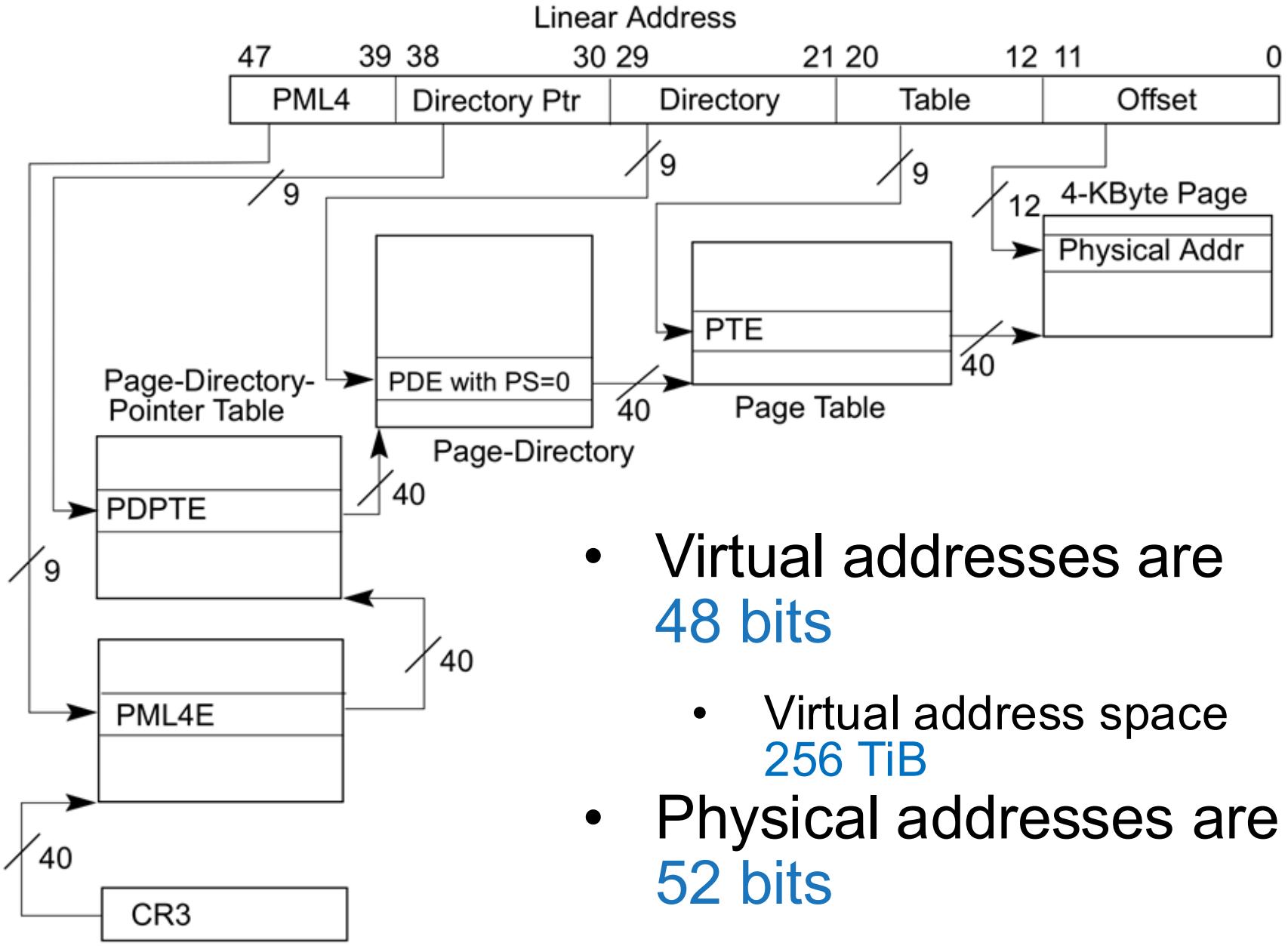
Page translation in 64bit mode

X86 64bit mode supports two page table organizations

- 4-level
- 5-level

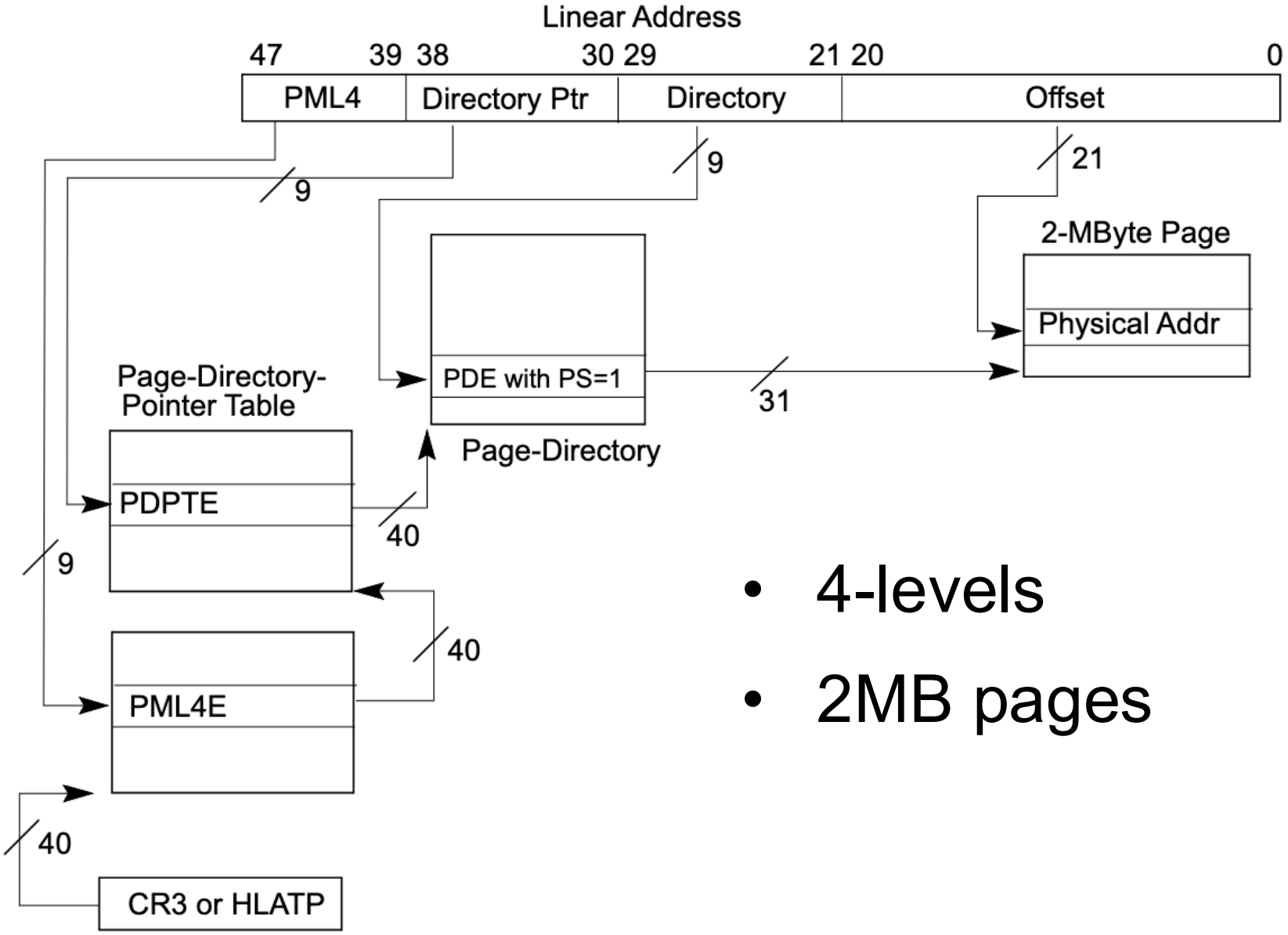
- And 3 page sizes
  - 4KB
  - 2MB
  - 1GB

Page translation in 64bit mode



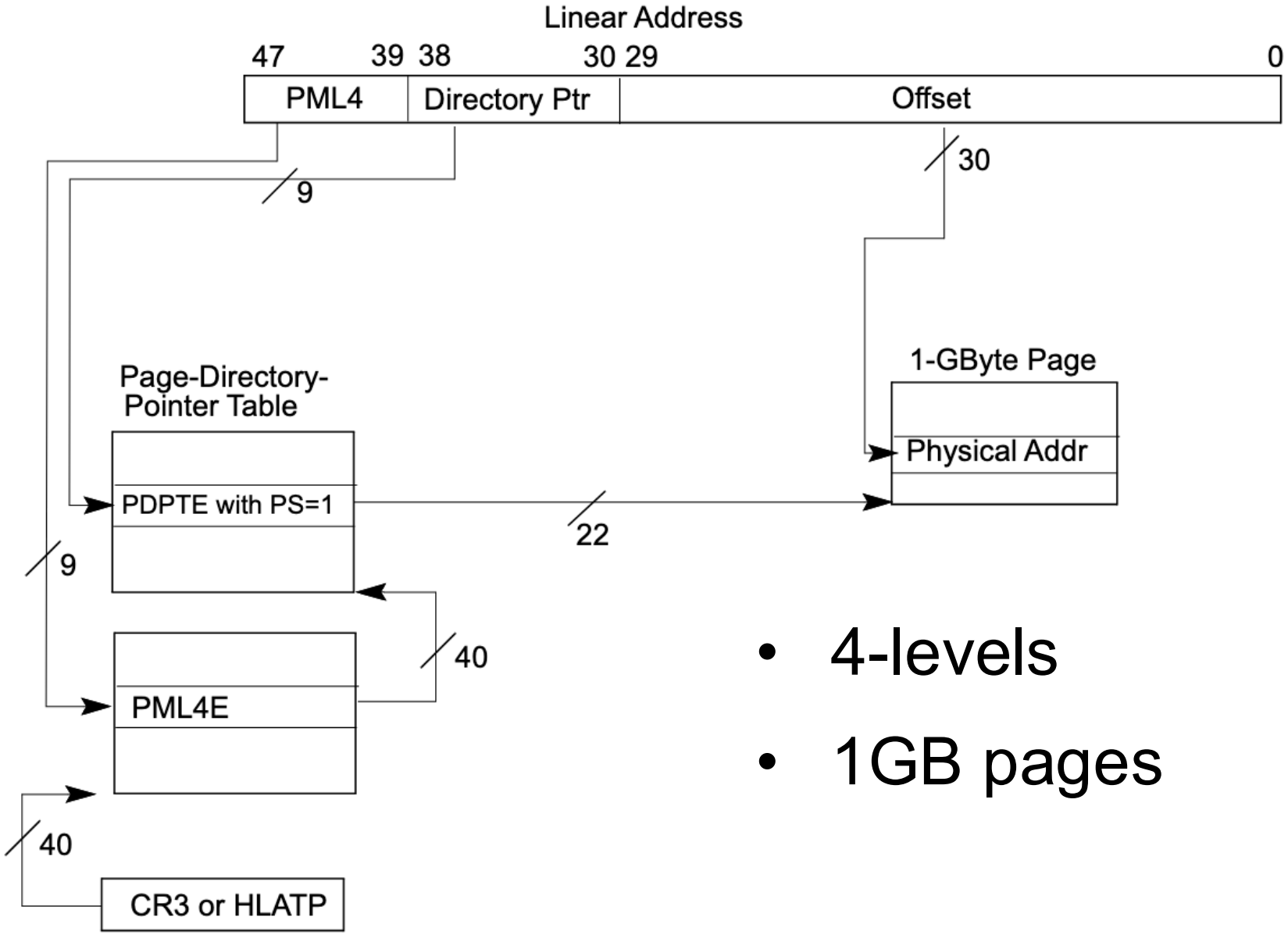
- Virtual addresses are 48 bits
  - Virtual address space 256 TiB
- Physical addresses are 52 bits

2MB pages



- 4-levels
- 2MB pages

1GB pages

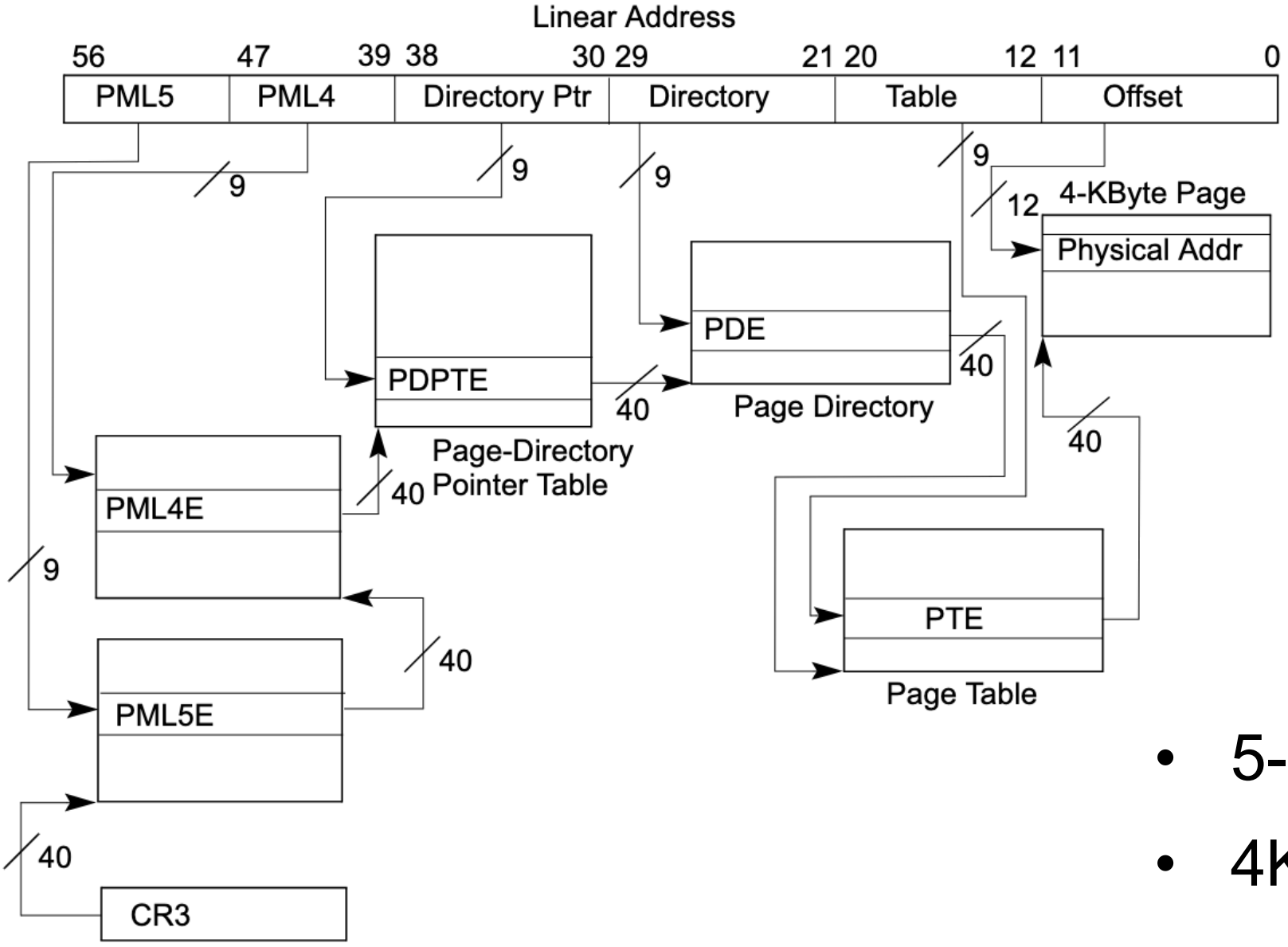


- 4-levels
- 1GB pages

5-level page table

- 5-level paging adds another 9 bit page table descriptor, making it possible to use bits 0 through 56
- This multiplies the address space by 512 and increases the limit to 128 PiB.

5-level page table

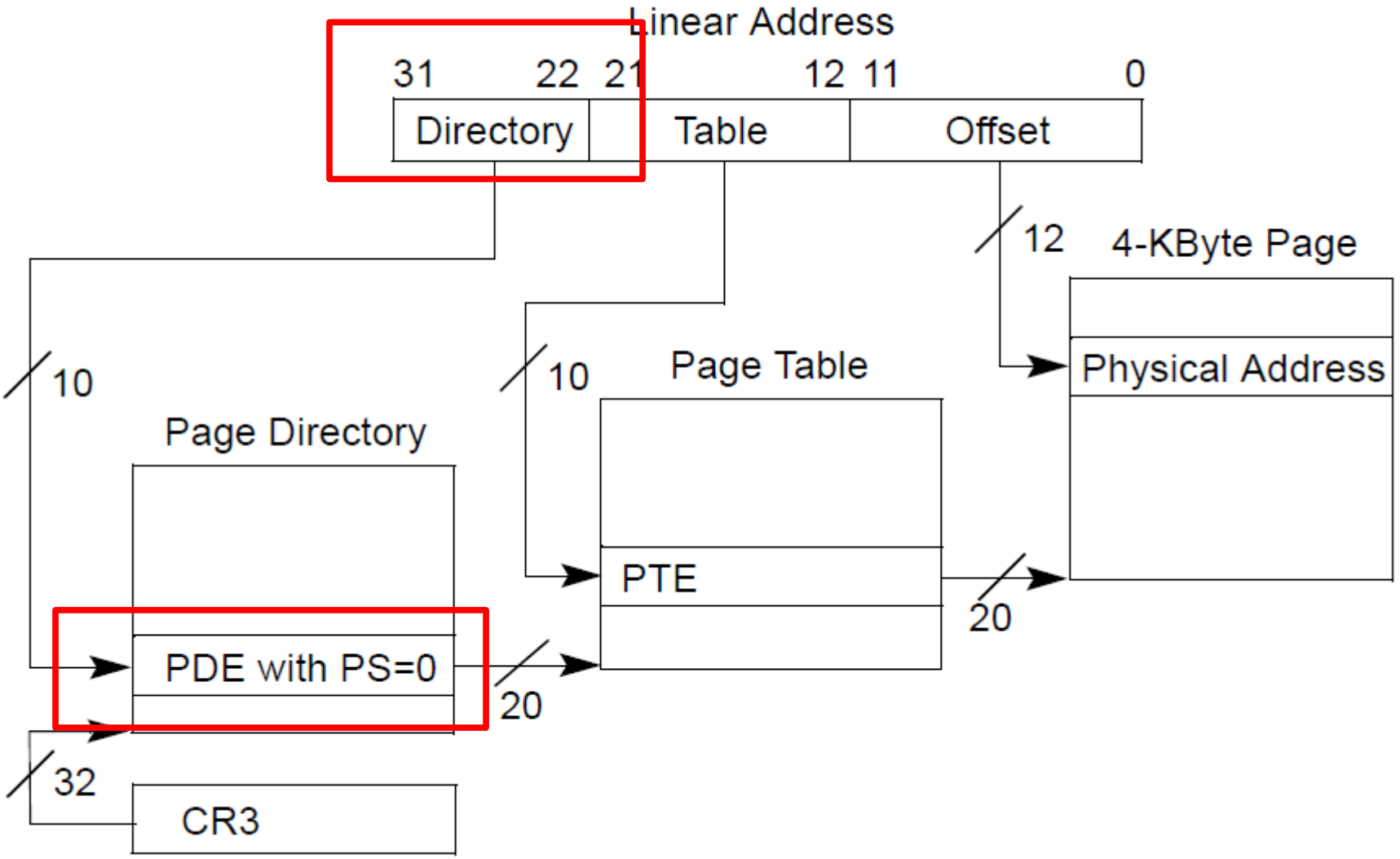


- 5-levels
- 4KB pages

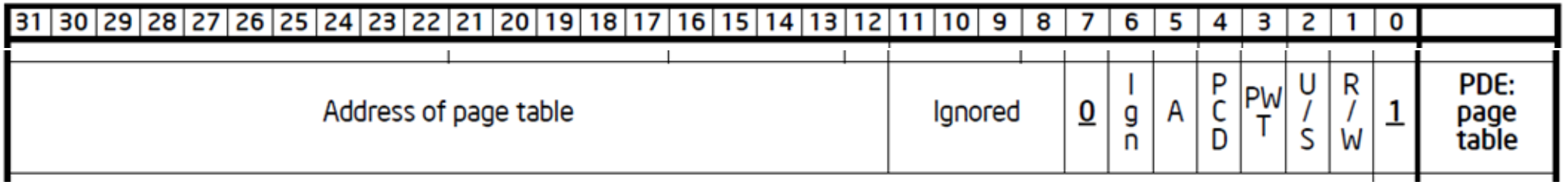
What pages are used for

- 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

Page translation



Page directory entry (PDE)



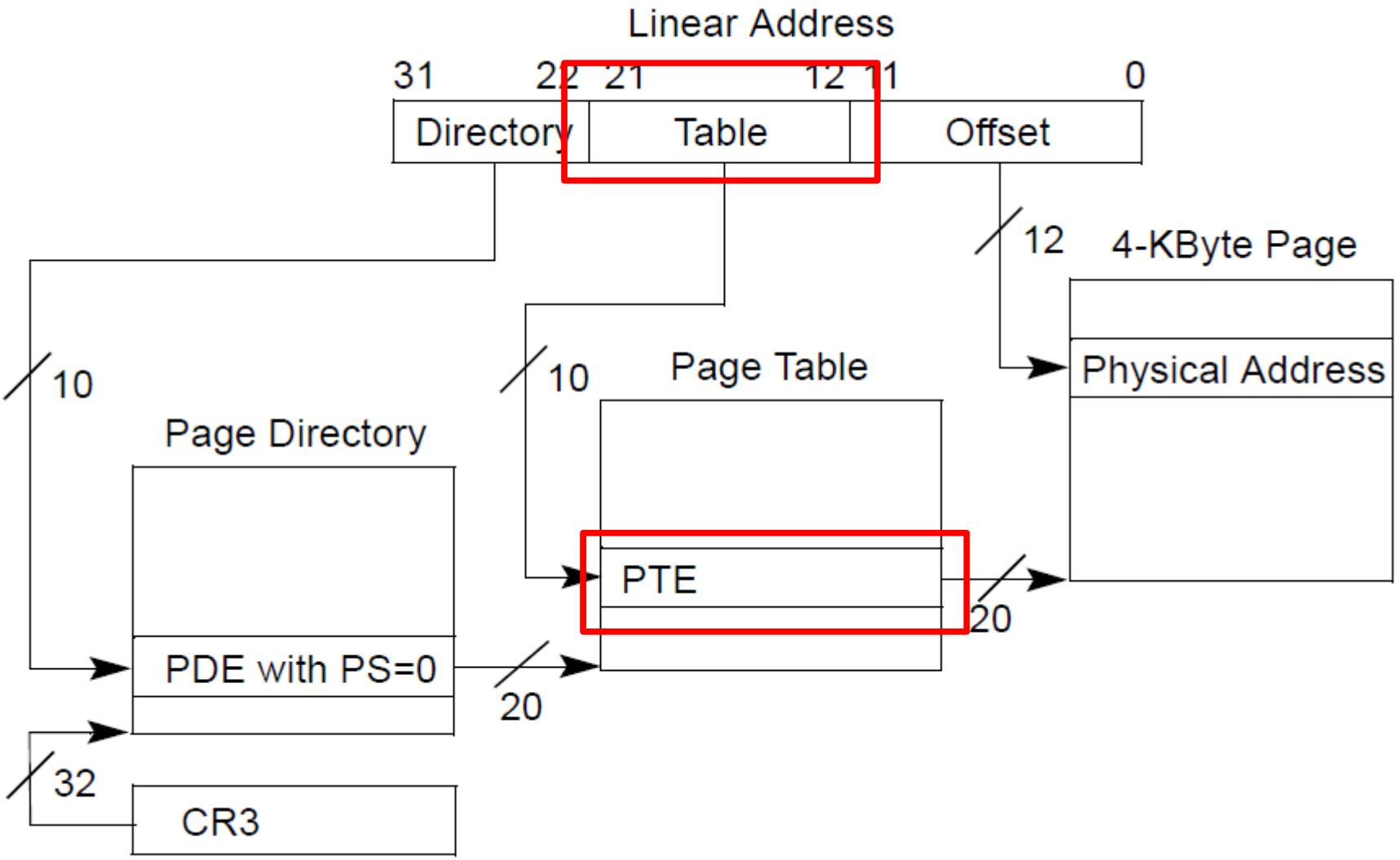
- 20 bit address of the page table

PollEv.com/antonburtsev

More paging tricks

- Determine a working set of a program?

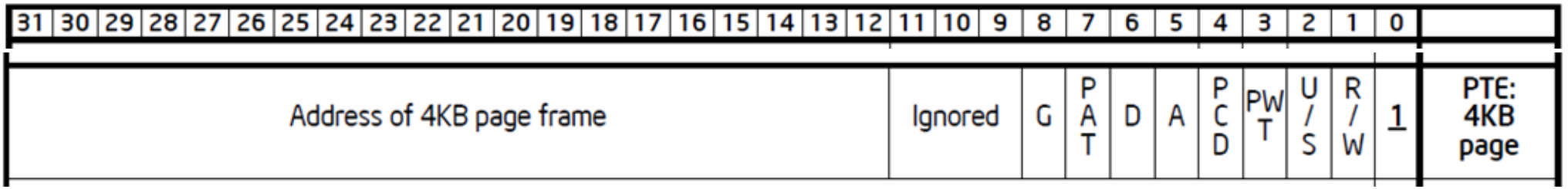
Page translation



More paging tricks

- Copy-on-write memory, e.g. lightweight fork()?

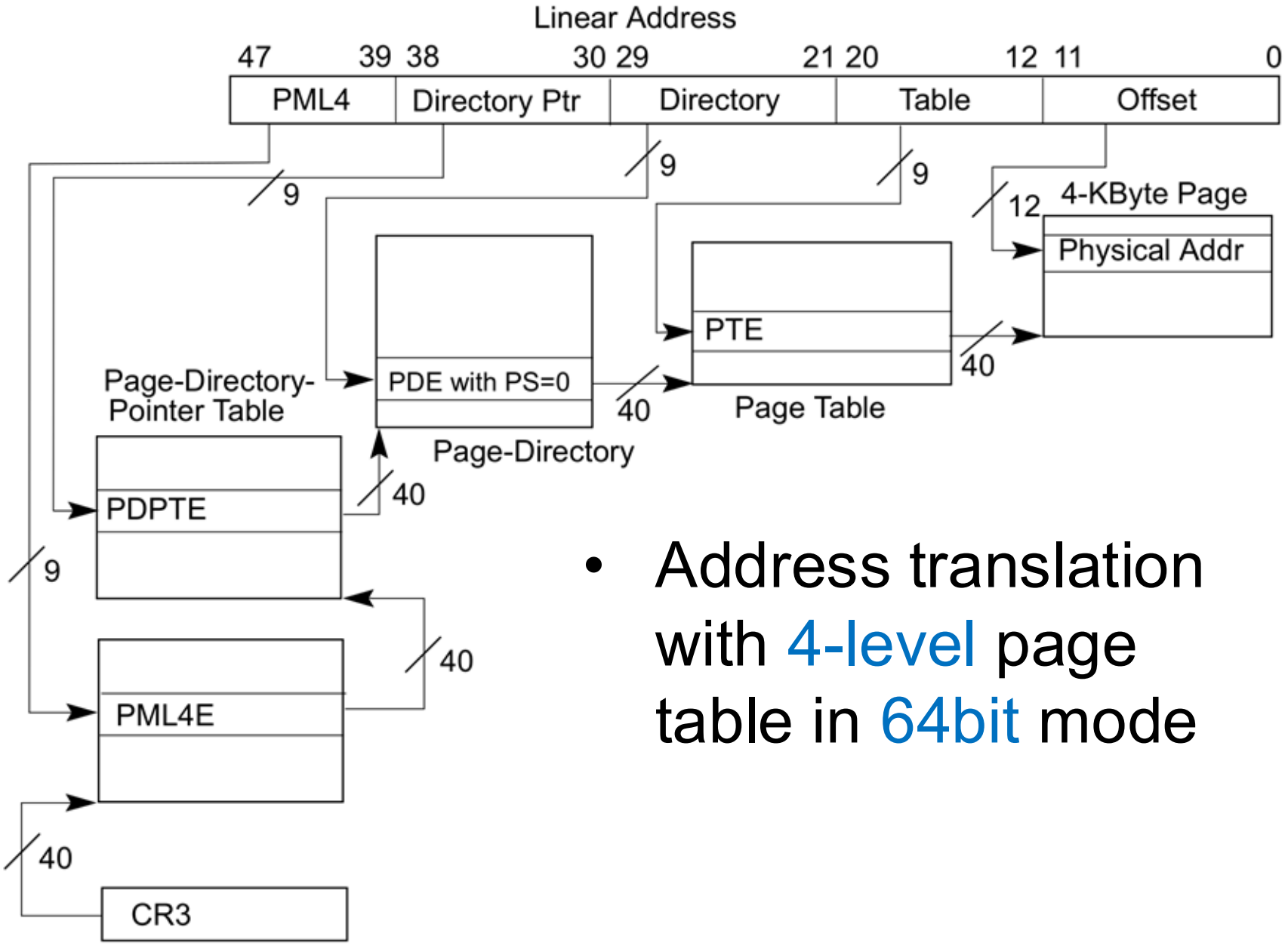
Page table entry (PTE)



- Map page as read only (bit #1)
- Keep track of that memory has to be copied on write there

Performance

Walking page table can be costly



- Address translation with 4-level page table in 64bit mode

Translation lookaside buffer (TLB)

- CPU caches results of page table walks

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

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

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 arrive 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, 4-level page tables (or 5-levels now)
- Page tables do not fit in L3 cache
- Modern servers come with 32MB cache
- Every cache miss results in up to 4 cache misses due to page walk (remember 4-level page tables)
- Each cache miss is 250 cycles

- Solution: 1GB pages

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?
- 1k entries * 1page/entry * 4K/page = 4MB

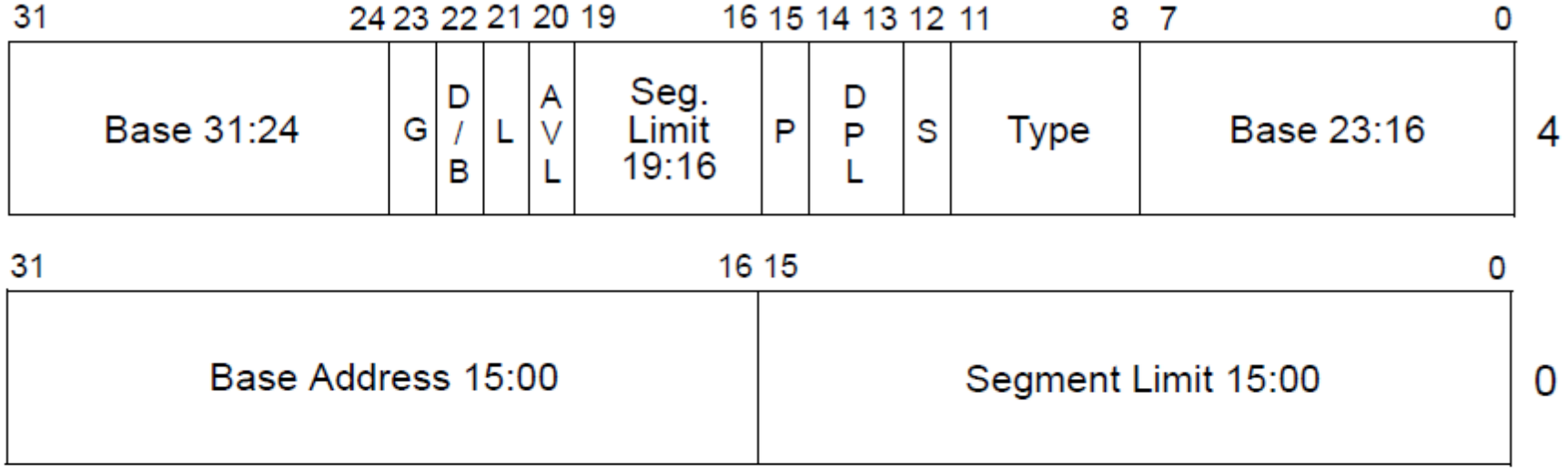
Back of the envelope

- 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?

Back of the envelope

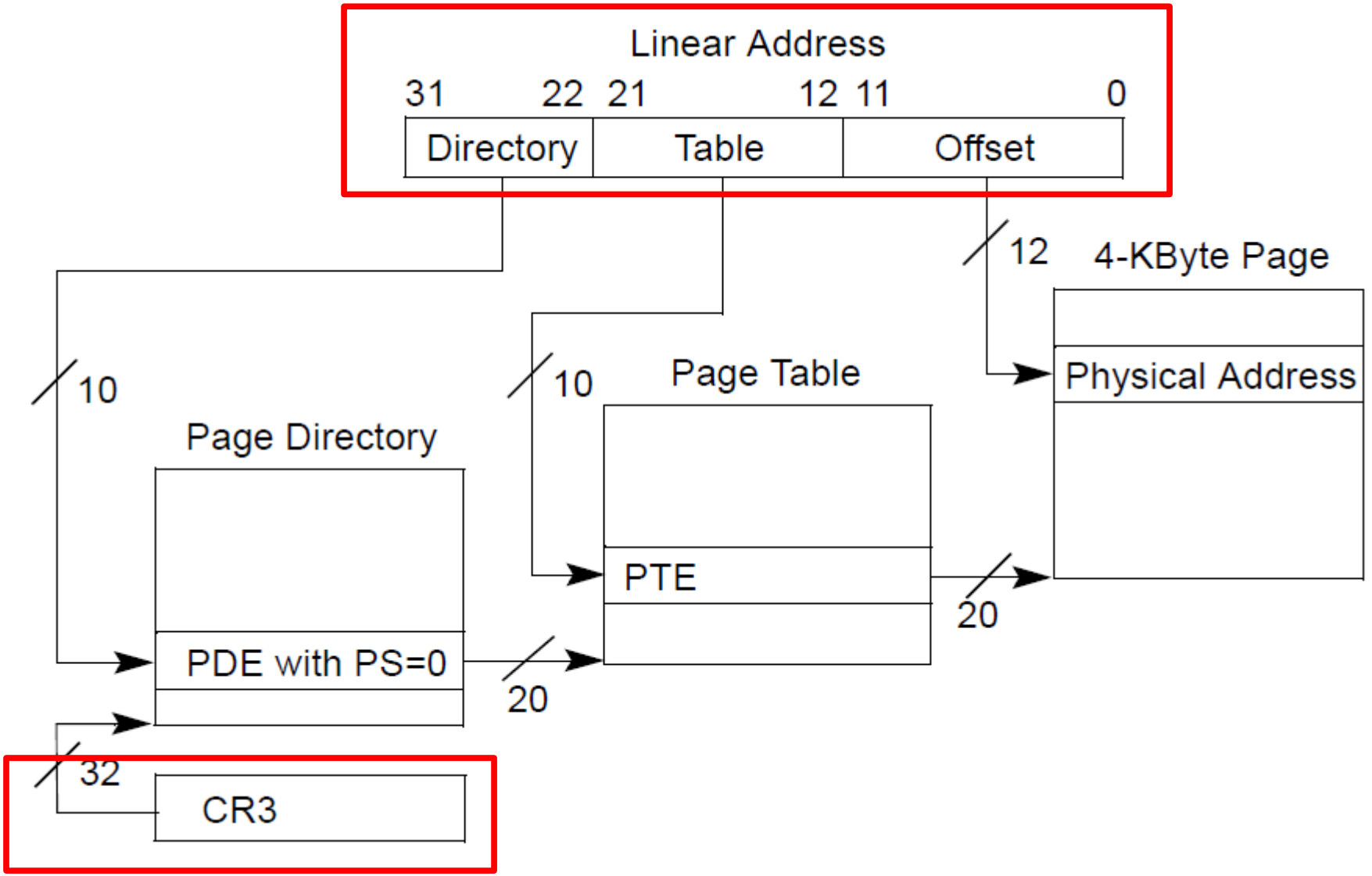
- If a page is 4K and an entry is 4 bytes, how many entries per page?
- 1k
- How large of an address space can 1 page represent?
- 1k entries * 1page/entry * 4K/page = 4MB
- How large can we get with a second level of translation?
- 1k tables/dir * 1k entries/table * 4k/page = 4 GB
- Nice that it works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

Segment descriptors

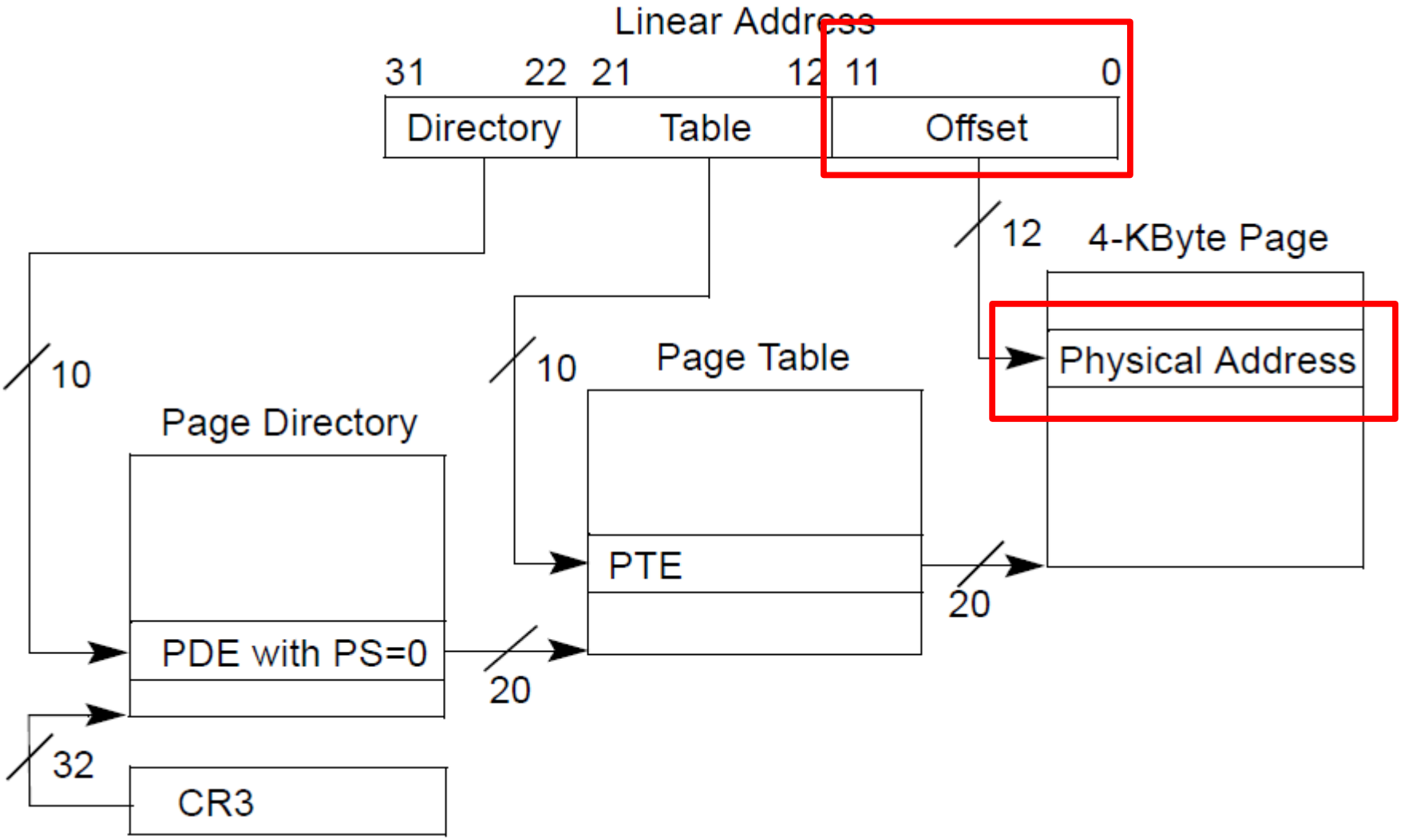


L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

Page translation



Page translation



Questions?