Virtual memory

Acknowledgements:

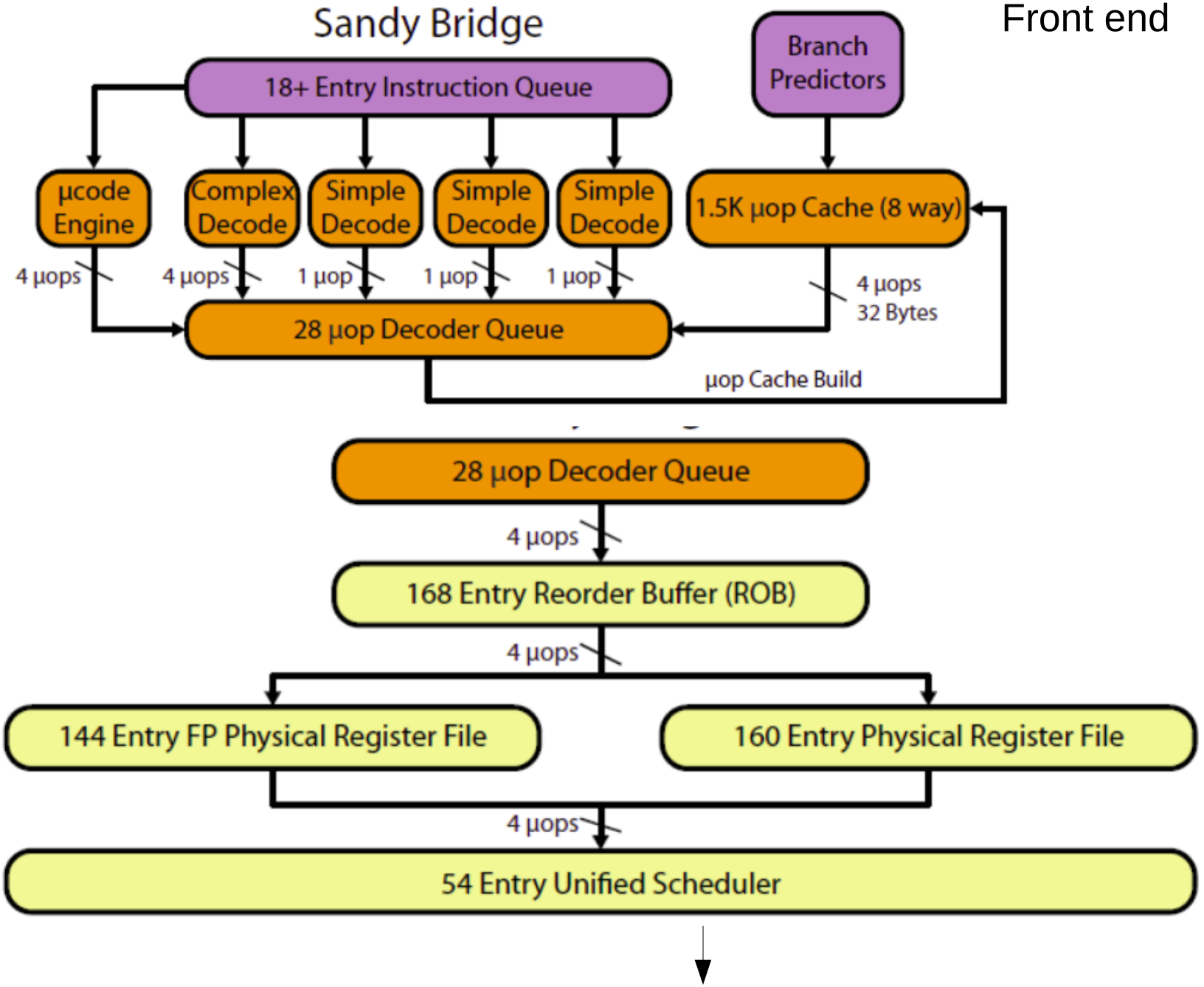
CO and design- Chapter 5 - Henessey and Patterson

CA- Quant approach- H & P

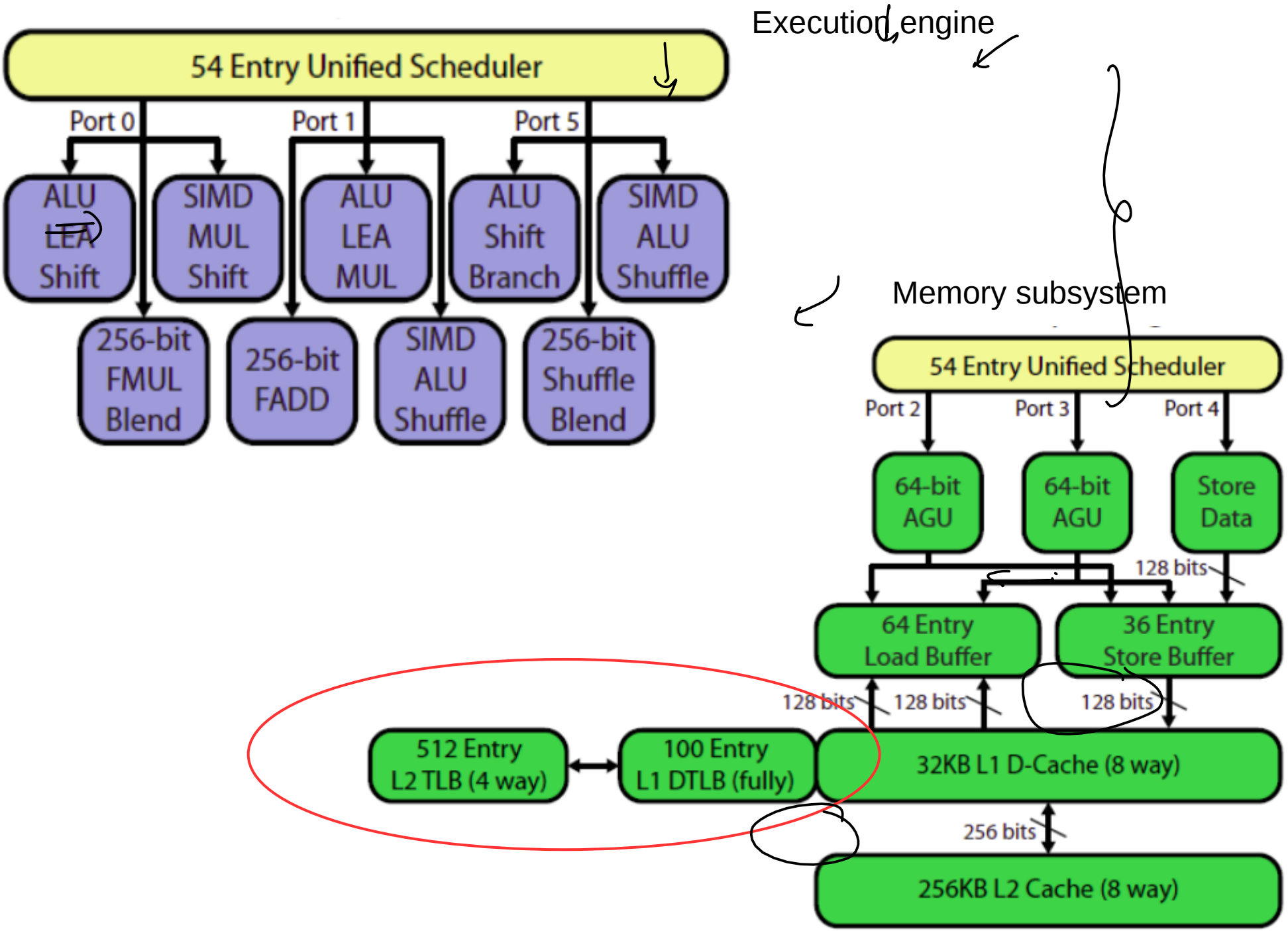
HPCA- Georgia Tech- Virtual memory
https://www.youtube.com/watch?v=Dz9Hgq65iJw&list=PLAwxTw4SYaPn79fsplIuZG34KwbkYSedj

Sandybridge architecture

Front end



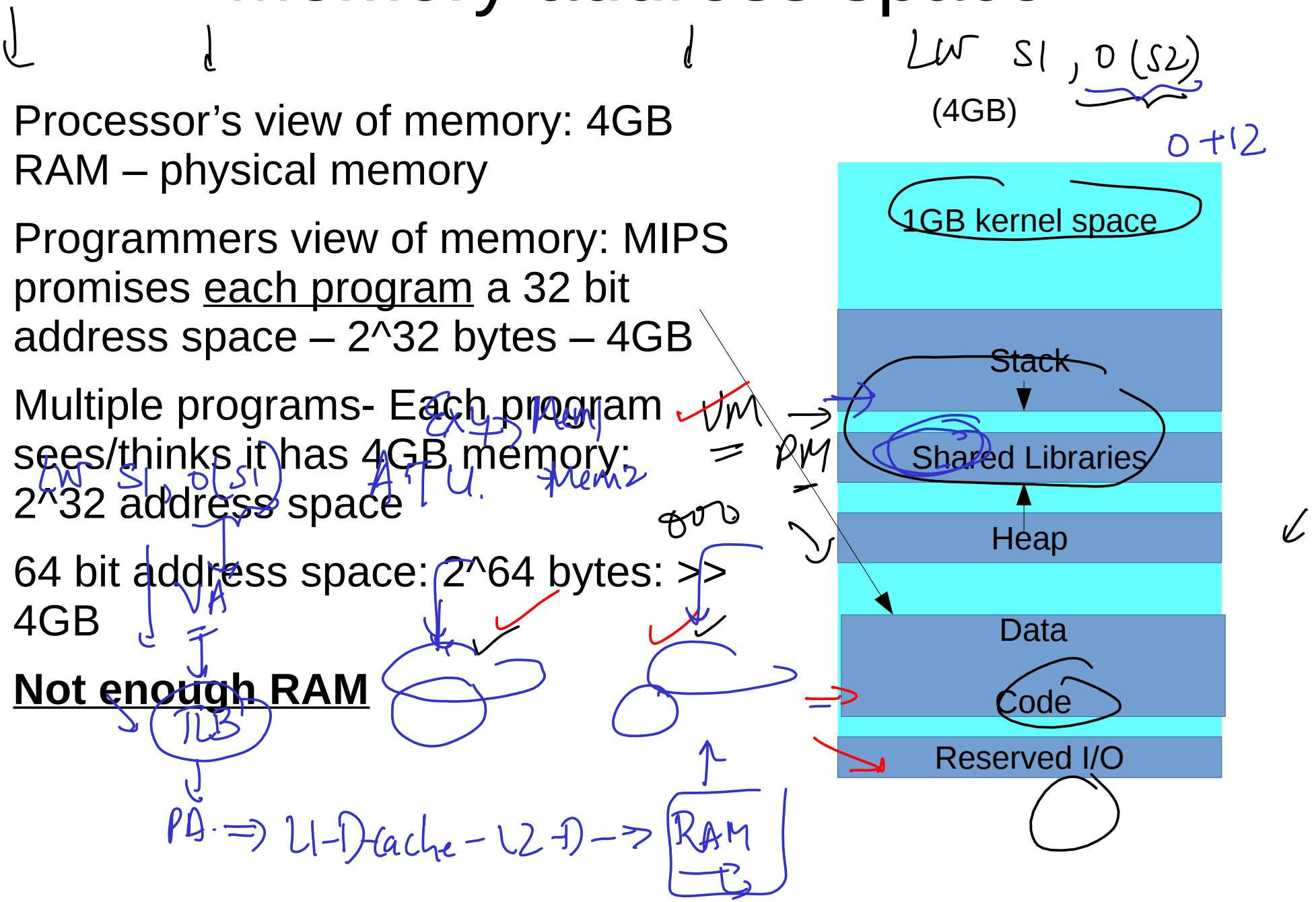
https://www.realworldtech.com/sandy-bridge/10/



https://www.realworldtech.com/sandy-bridge/10/

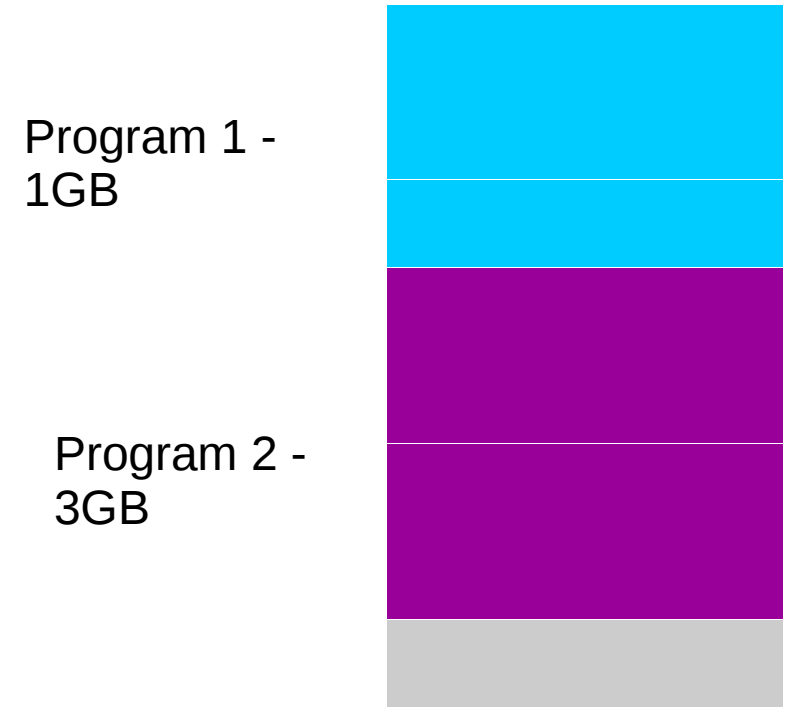
Memory address space

- Processor's view of memory: 4GB RAM – physical memory
- Programmers view of memory: MIPS promises <u>each program</u> a 32 bit address space – 2^32 bytes – 4GB
- Multiple programs- Each program sees/thinks it has 4GB memory: 2^32 address space
- 64 bit address space: 2^64 bytes: >> 4GB
- **<u>Not enough RAM</u>**



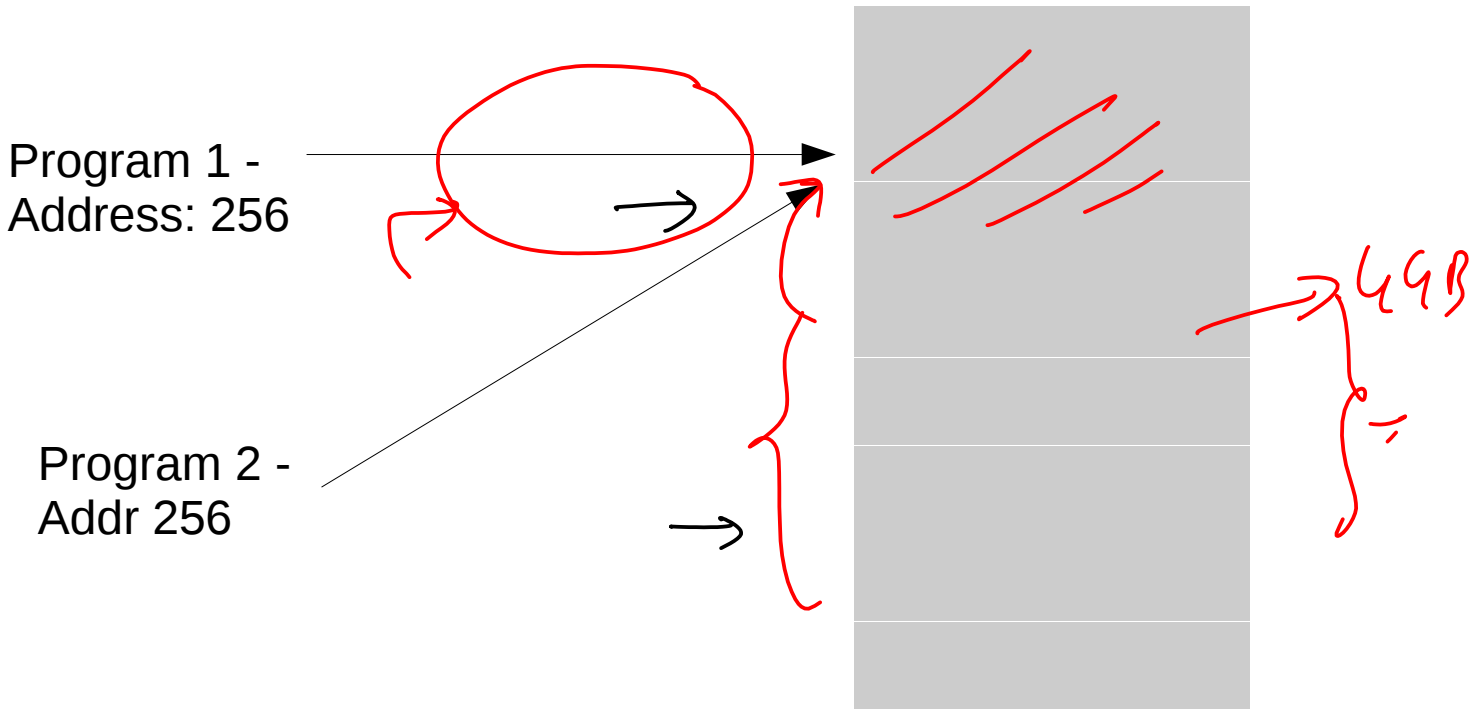
Holes in physical memory

- Quit Program 1 --> Free 1GB space
- Assume program 3 needs 2GB space- cannot run
- Memory fragmentation



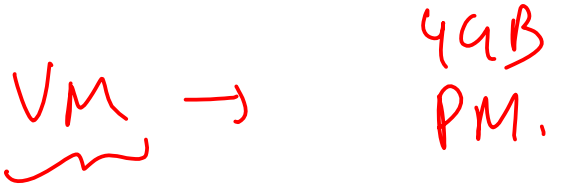
Security

- 2 programs access same address location unless sharing data
- Corrupt data/Crash
- Problem: Same memory space
- Solution: Give each program its own “virtual memory space”

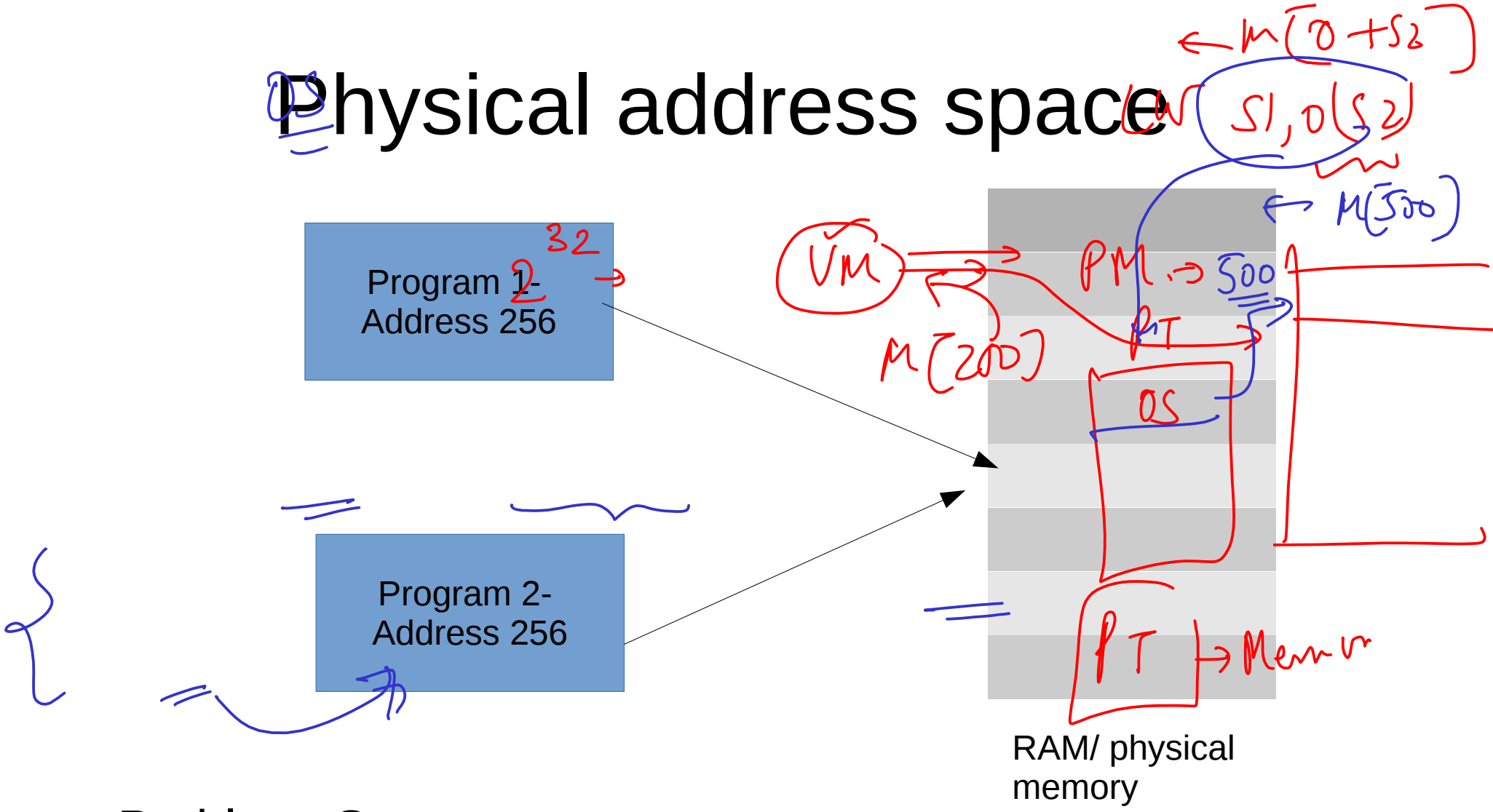


Virtual memory

- What is VM?
- Page tables and translation
- Page faults, Multi-level page tables
- Translation look aside buffer (TLB)
- TLB and Caches



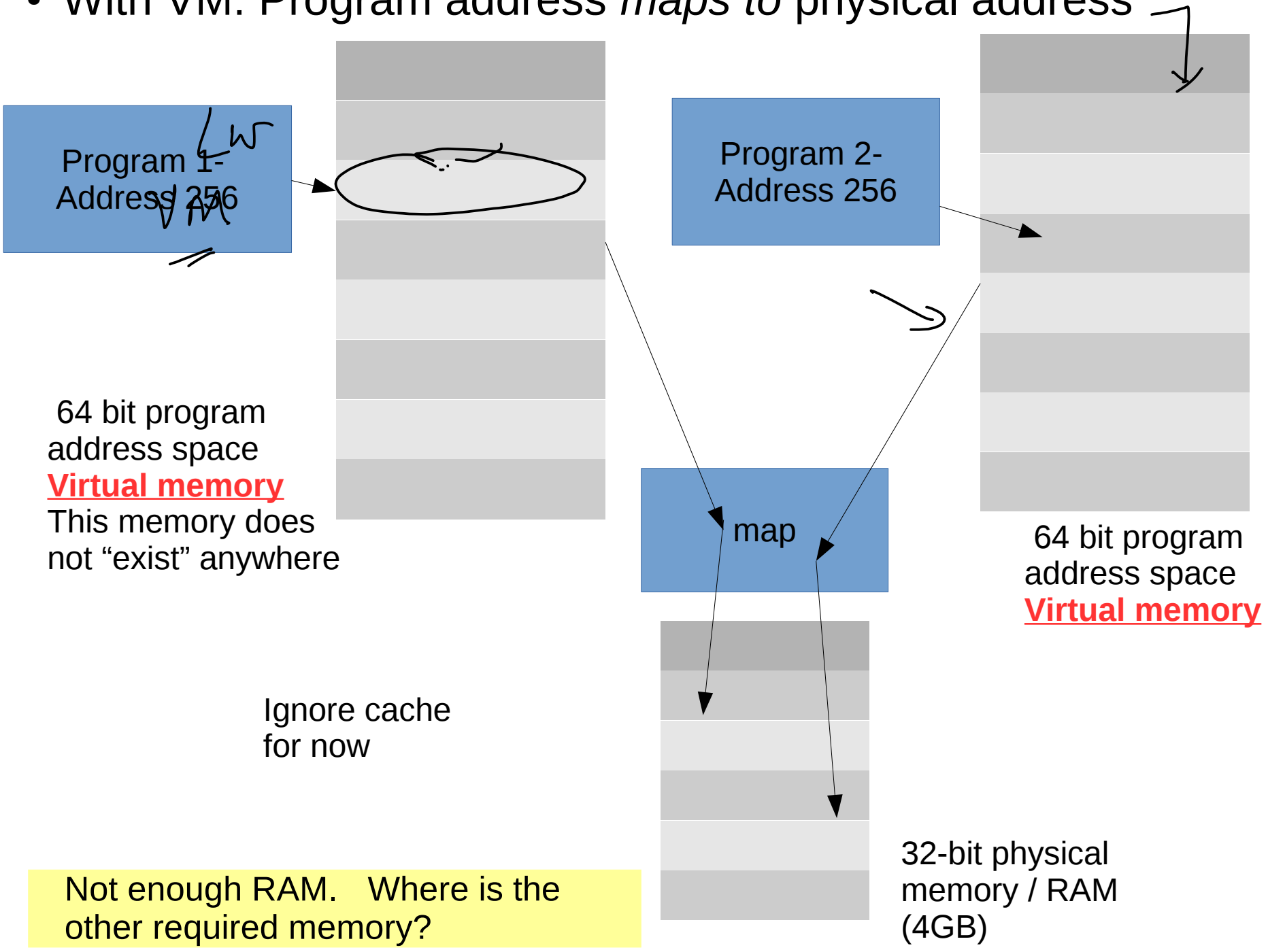
Physical address space

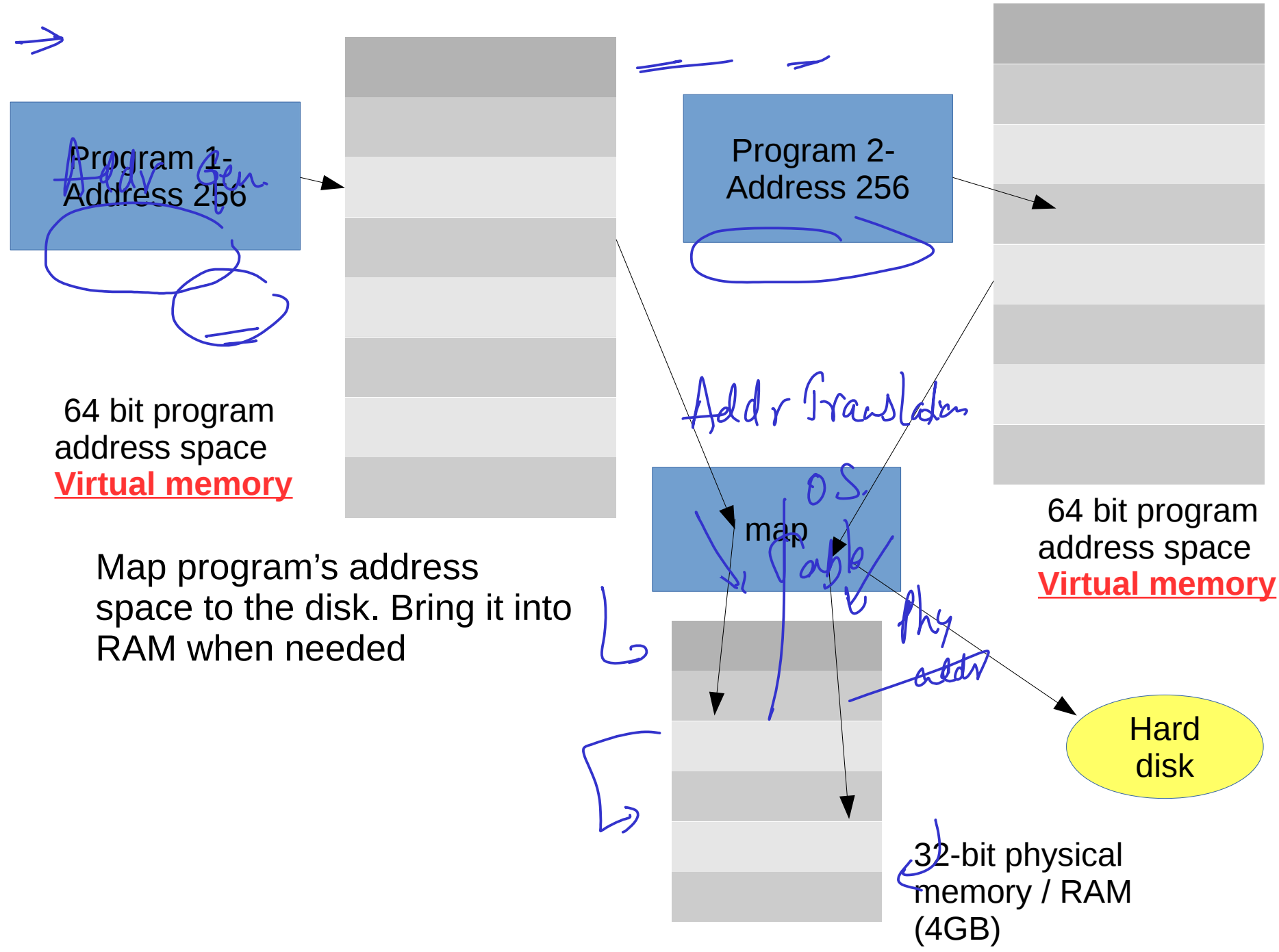


- Problem: Same memory space
- Solution: Give each program its own “virtual memory space”

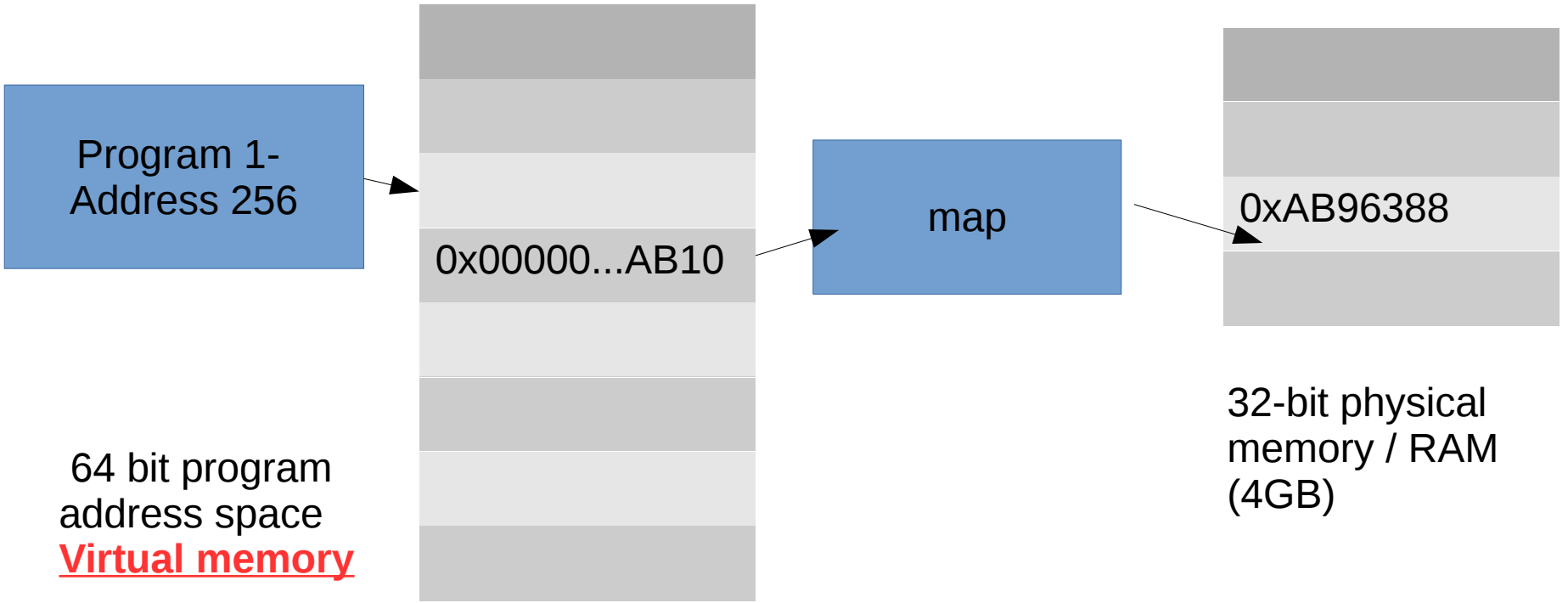
  --> Map to the physical RAM memory space

- Without VM: Program address = RAM/physical address
- With VM: Program address *maps to* physical address



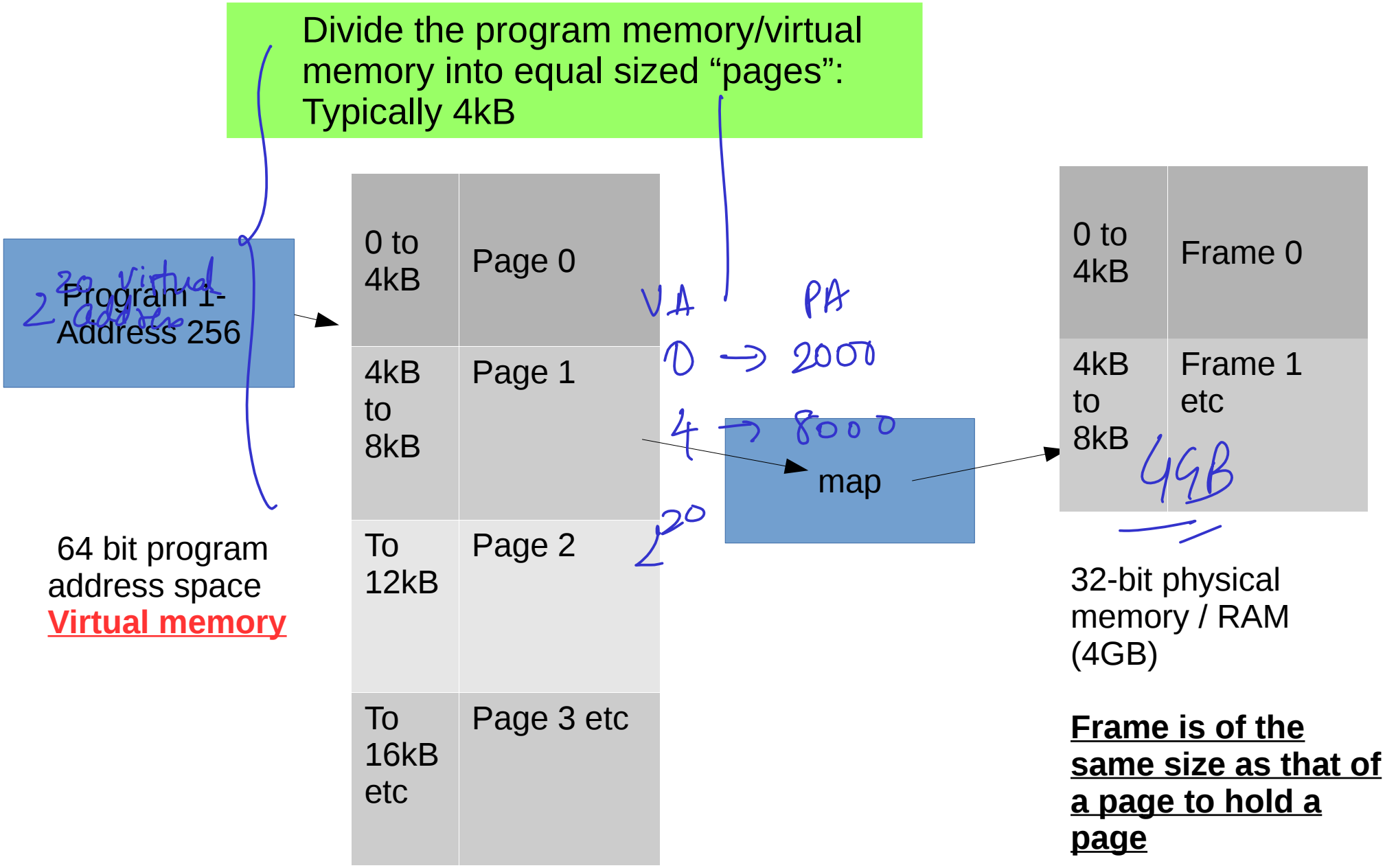


Mapping gives the illusion of an unlimited memory as long as Disk has 4GB space --> Every program can get 4GB of space. Data will be brought from disk to RAM



Problem of mapping:

- If we have a word aligned memory, how many mappings do you need?
  - 2^30 --> 1 billion --> x 32 bits per entry (PA=32 bits) --> 1GB for mappings entries
  - Mapping each virtual address to a physical address would result in a large number of mapping entries


Divide the program memory/virtual memory into equal sized “pages”: Typically 4kB
Program 1- Address 256
0 to 4kB
Page 0
4kB to 8kB
Page 1
To 12kB
Page 2
To 16kB etc
Page 3 etc
map
0 to 4kB
Frame 0
4kB to 8kB
Frame 1 etc
64 bit program address space
Virtual memory
32-bit physical memory / RAM (4GB)
Frame is of the same size as that of a page to hold a page

Program (LW) generates a Virtual address. Processor needs a physical address to access

Program 1 VM

0 to 4kB	Page 0
4kB to 8kB	Page 1
To 12kB	Page 2 . . .
To 32kB	Page 7 etc

Program 2 VM

0 to 4kB	Page 0
4kB to 8kB	Page 1
To 12kB	Page 2 . . .
To 32kB	Page 7 etc

The OS does the mapping

Page table- translates virtual to physical address

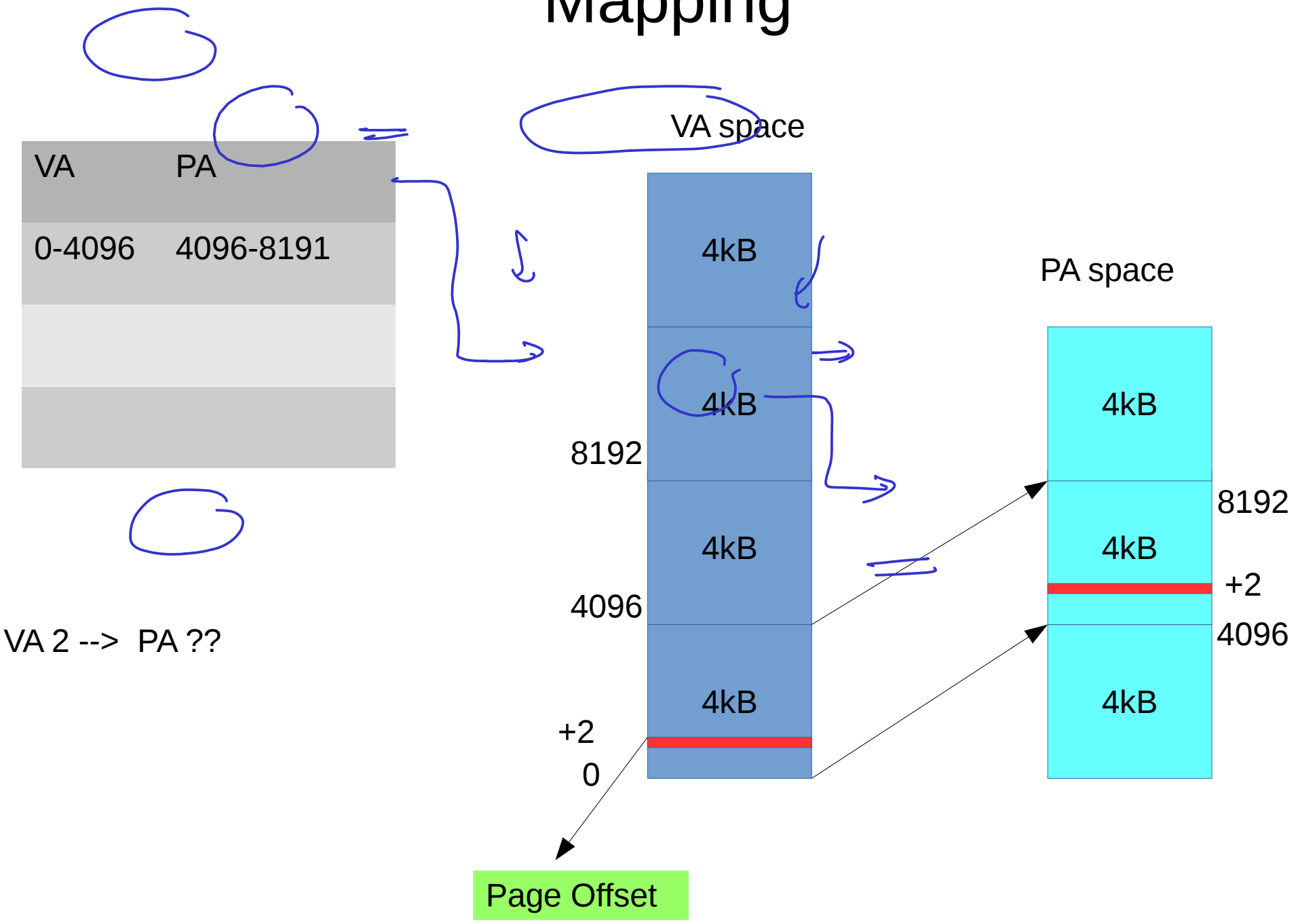
page	frame
0	2
1	0
2	
3	1

Page table: Stored in physical memory/ RAM

0 to 4kB	Frame/ page 0
4kB to 8kB	Frame/ page 1 etc
	Frame 2
	Frame 3

32-bit physical memory / RAM (4GB)

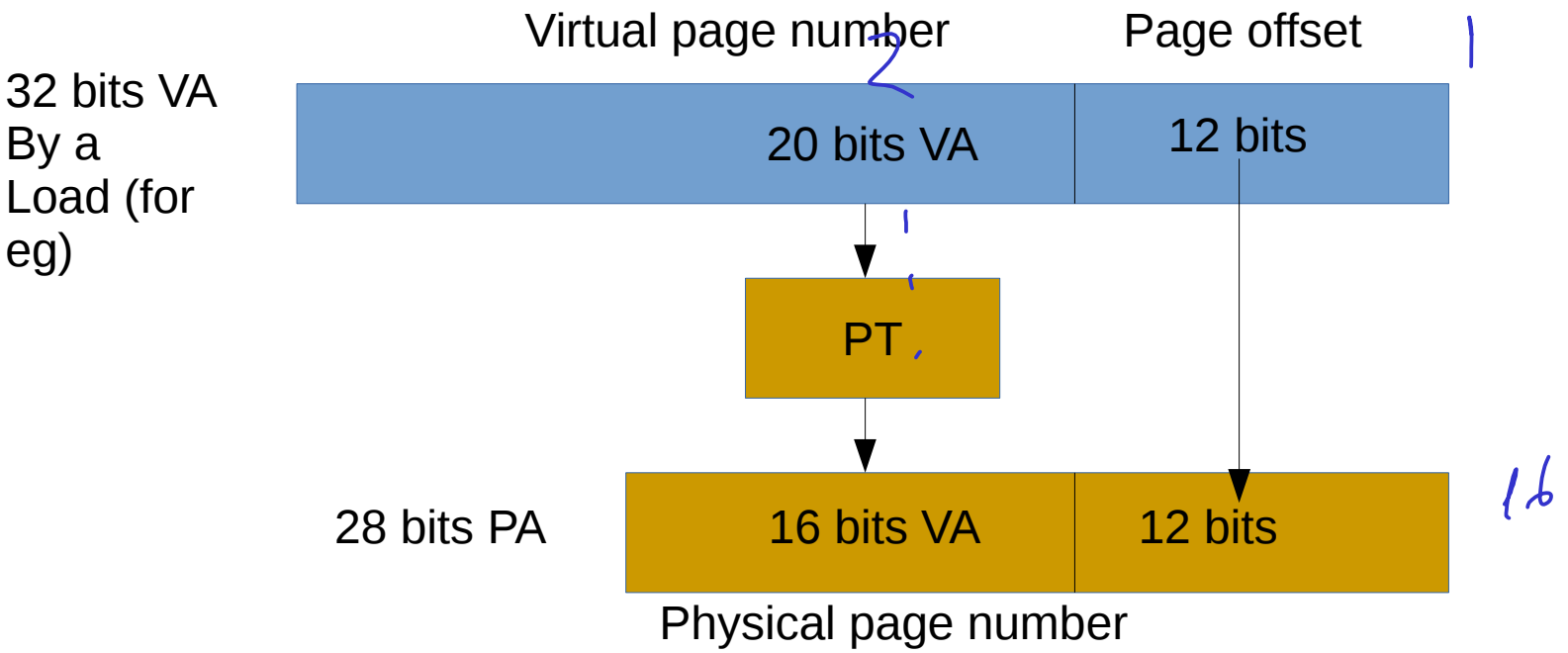
Mapping



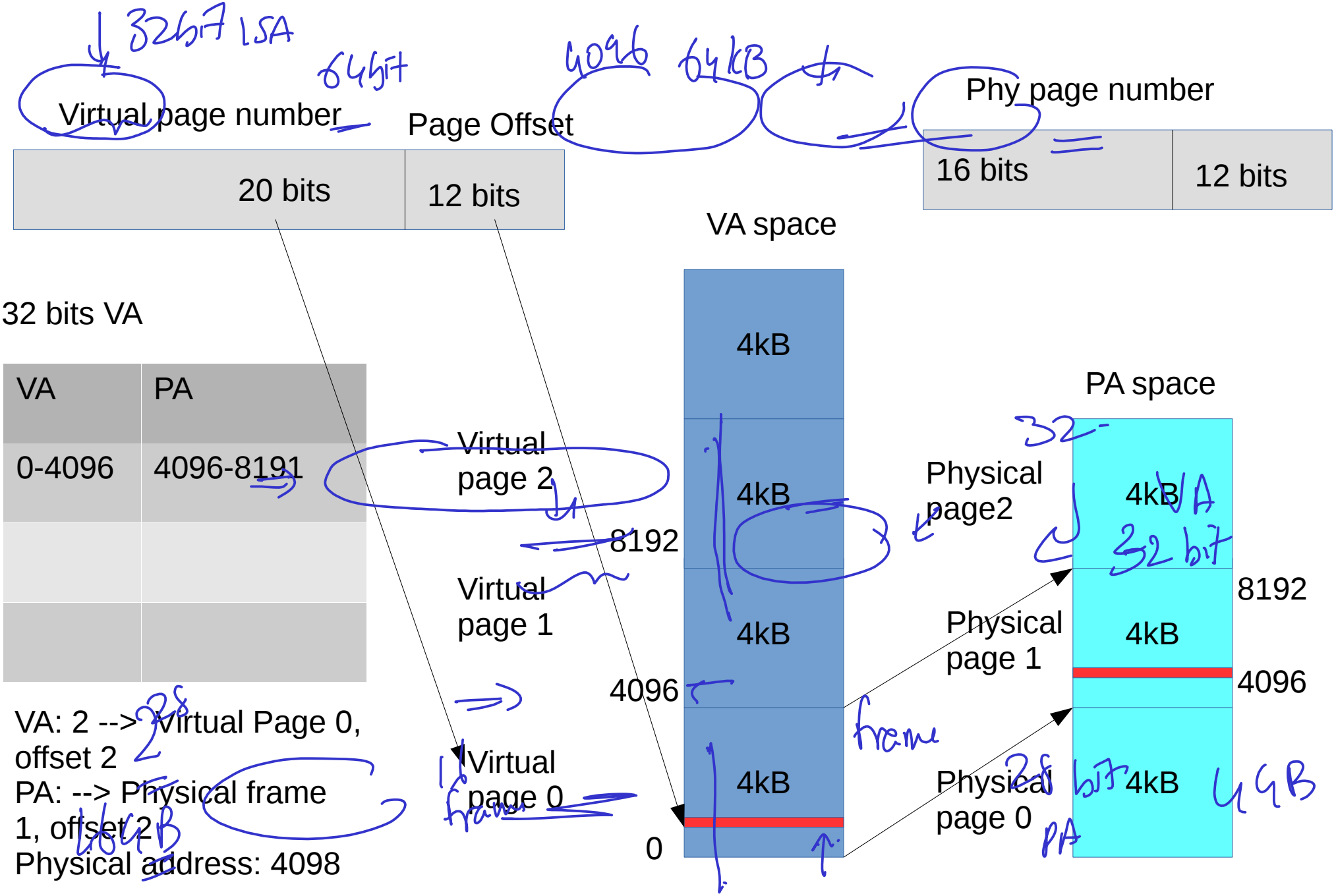
VA 2 --> PA ??

Address translation

- 32 bit virtual address space, 4kB page, 256MB (2^28) RAM
  - Each page has 4096 addresses – Each address do not need translation. It is part of a page offset- Form the lower 12 bits of VA
  - Each page needs a translation
  - Or in other words: 2^32 / 4096 = 2^ 20 pages --> These page numbers need translation to physical address. These form 20 bits of VA



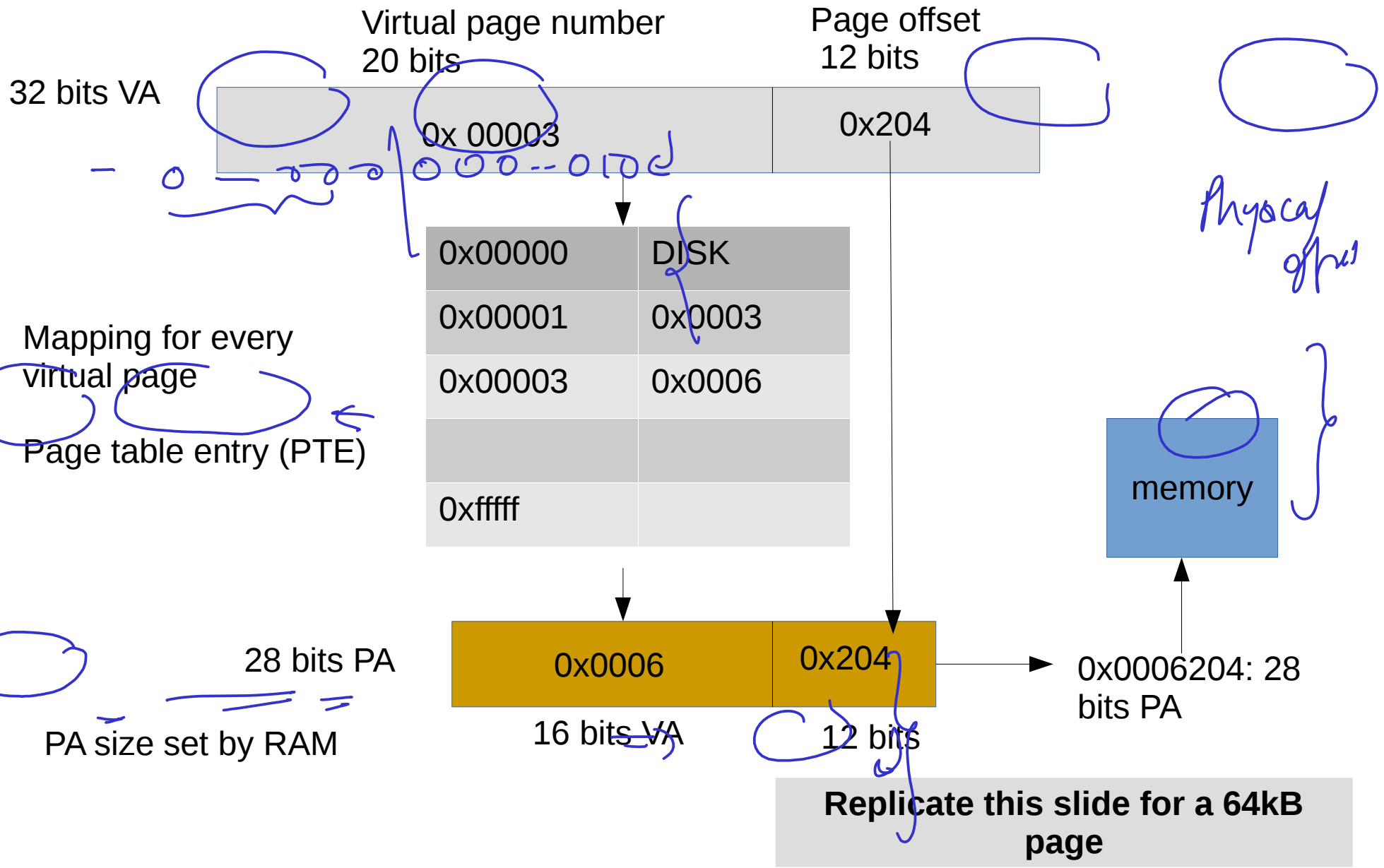
Mapping



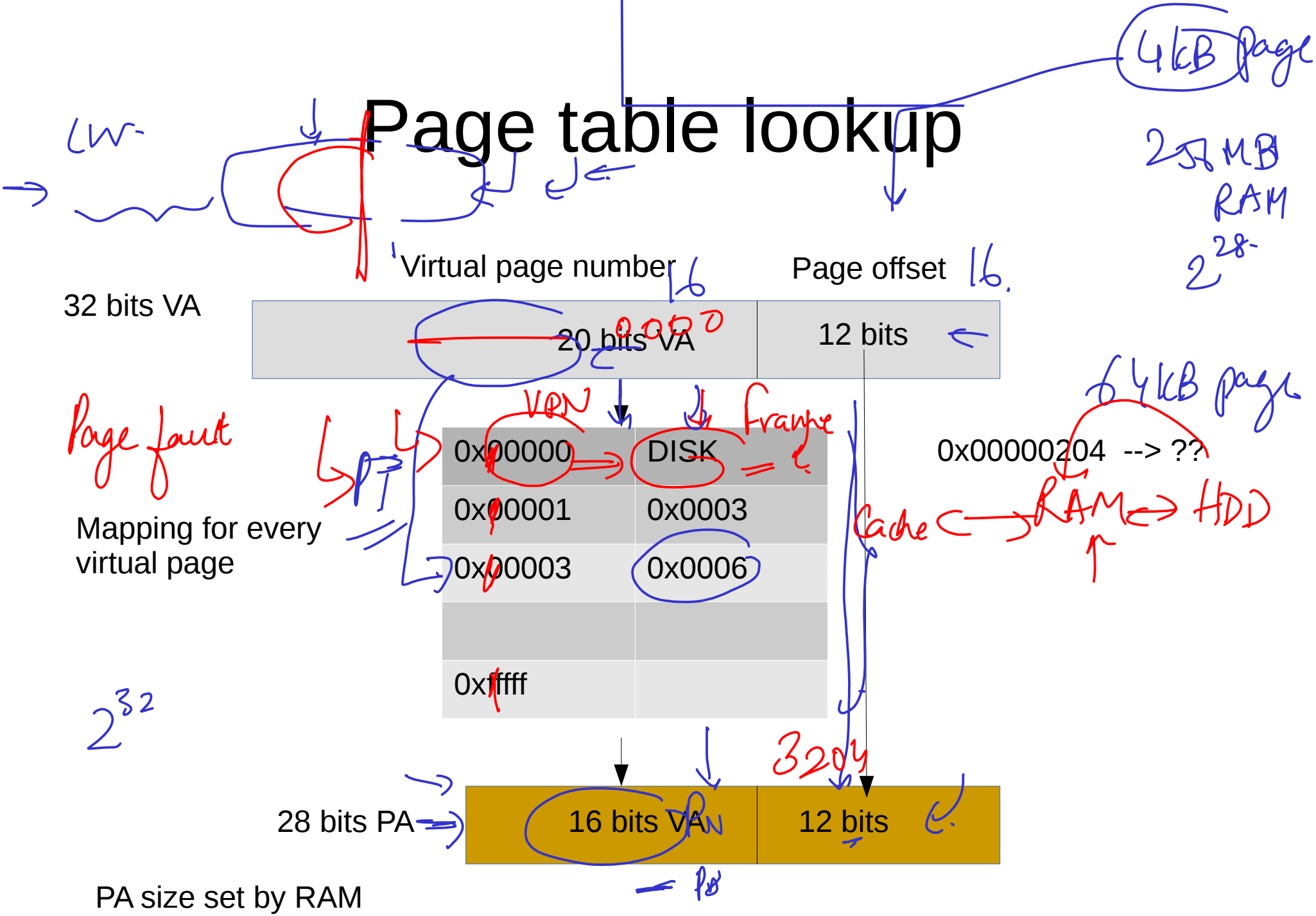
VA	PA
0-4096	4096-8191

Page table lookup

32 bit VA: 0x 00003204

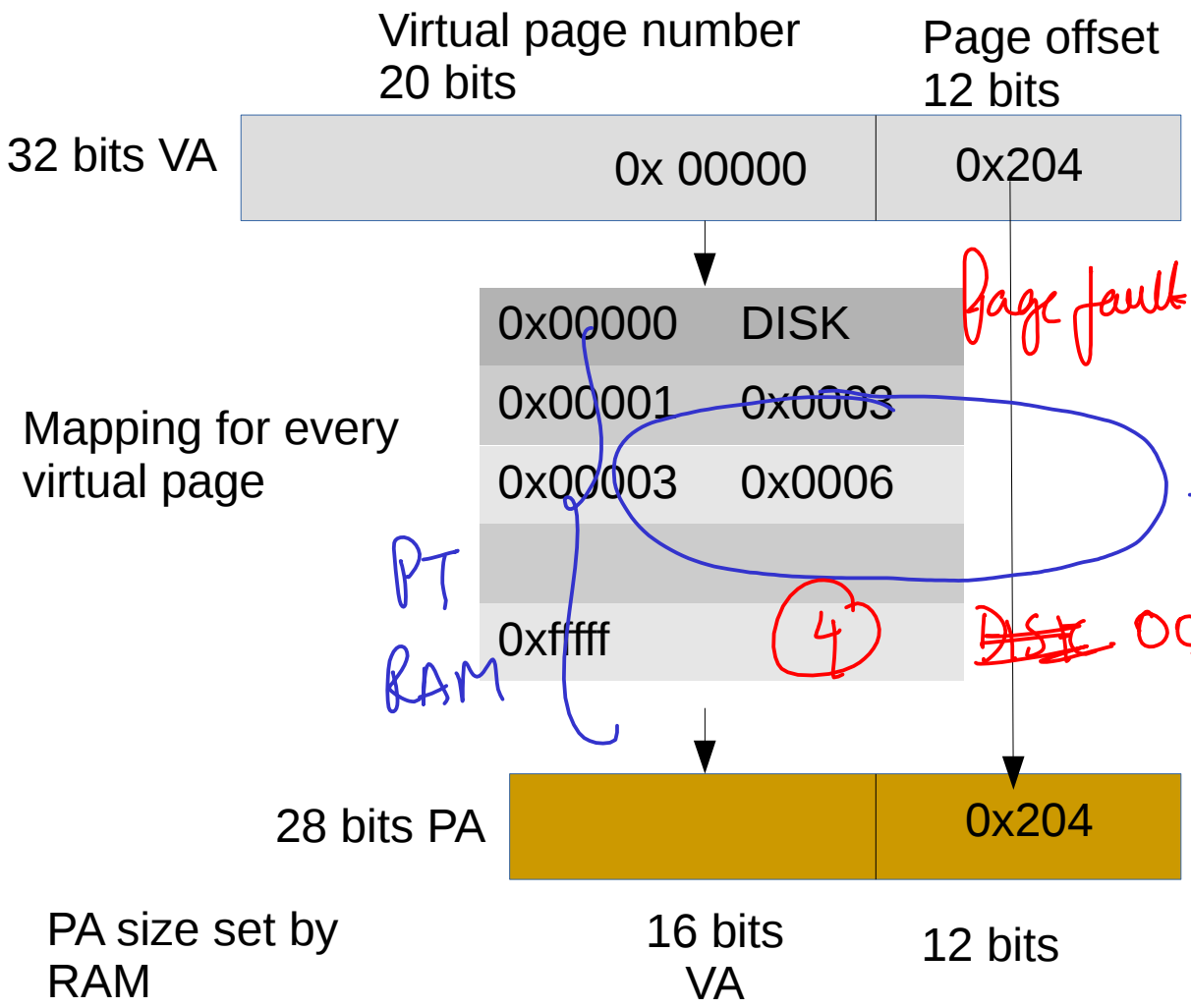


Page table lookup



Page fault

VA: 0x00000204



1)LW R3, 40(R2) --> Address is the virtual address

2)PT Lookup, Compute physical address

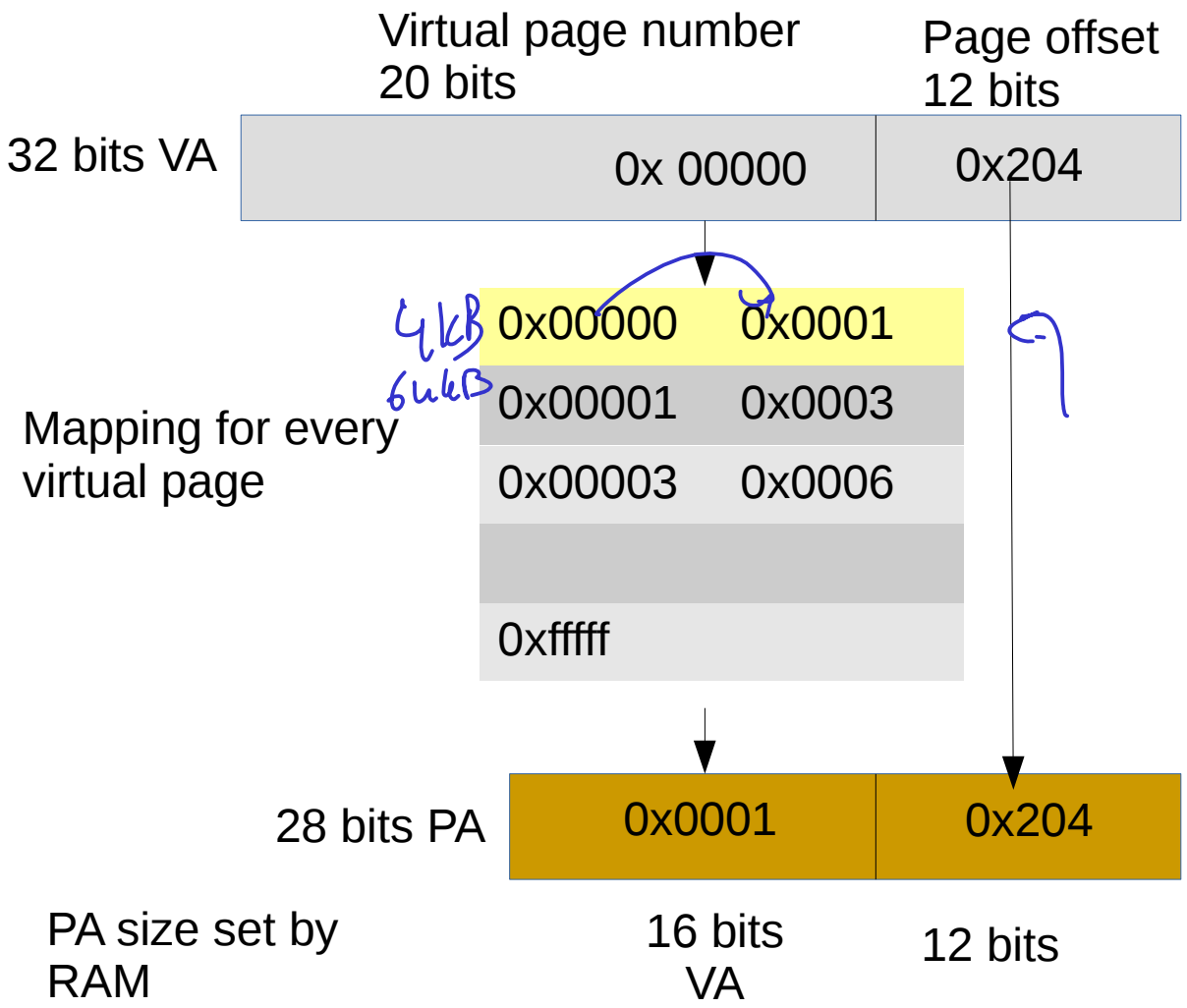
3)0x00000204 --> Data not in RAM --> CPU generates **page fault** -->exception --> OS page fault handler

4)OS chooses a victim to evict from RAM (based on some replacement policy), writes it to disk if the page is dirty

5)OS invokes a disk interrupt routine --> reads page to RAM and updates PTE

Page fault

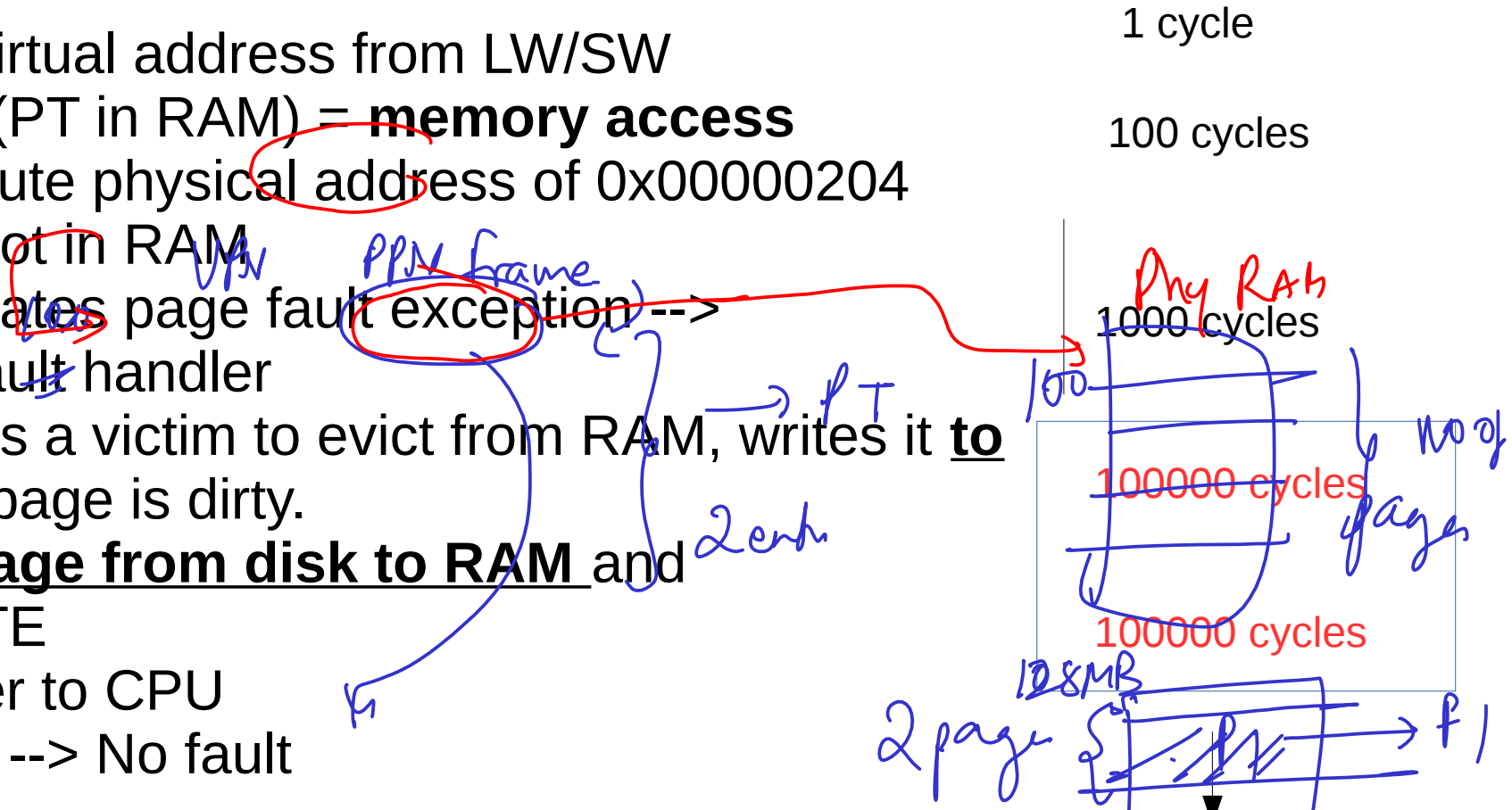
VA: 0x00000204



1)Read page from the disk to RAM and OS updates PTE
2)Hand it over to CPU
3)No page fault now

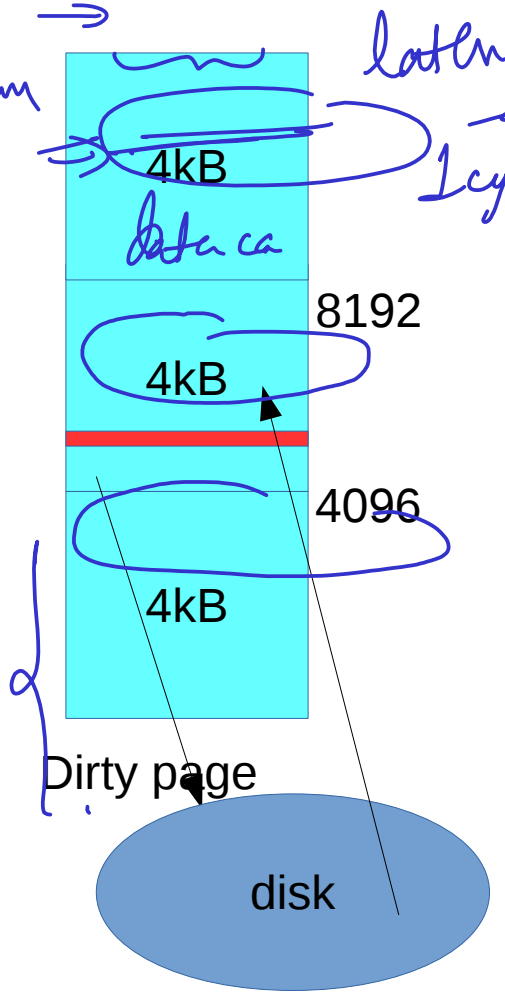
Page fault – time taken

- Compute virtual address from LW/SW
- PT lookup (PT in RAM) = **memory access** and Compute physical address of 0x00000204
- Say Data not in RAM
- CPU generates page fault exception -->
- OS page fault handler
- OS chooses a victim to evict from RAM, writes it **to disk** if the page is dirty.
- OS **read page from disk to RAM** and updates PTE
- Hand it over to CPU
- PT Lookup --> No fault



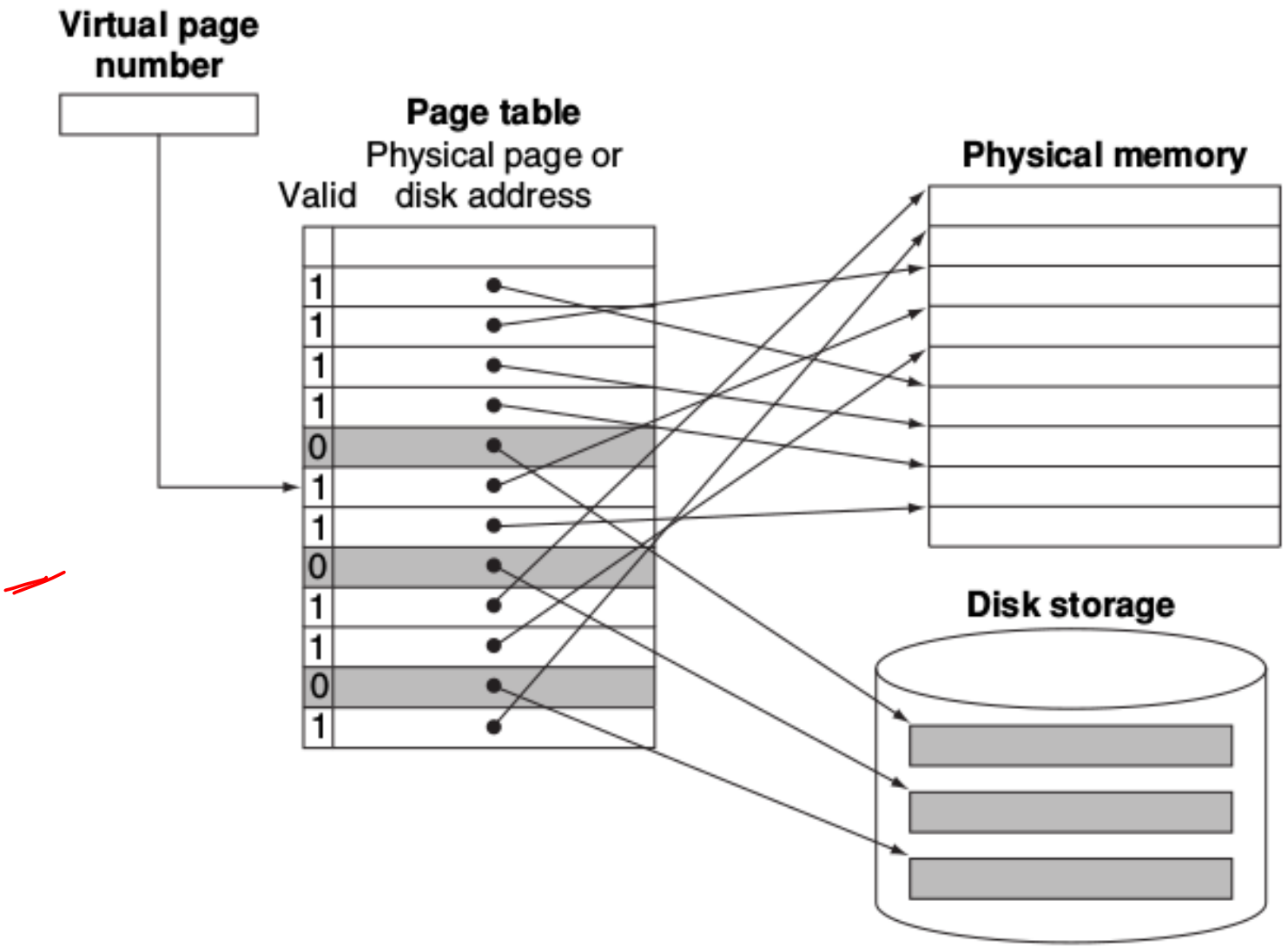
Page fault: Summary

- Compute virtual address from LW/SW
- PT lookup (PT in RAM) = memory access and Compute physical address of 0x00000204
- Say Data not in RAM
- CPU generates page fault exception --> OS page fault handler
- OS chooses a victim to evict from RAM, based on its replacement policy, writes it **<u>to disk</u>** if the page is dirty.
- Save the context (registers, PC etc) of the current process, switch to another process
- Disk interrupt routine --> **<u>reads page from disk to RAM</u>** and updates PTE
- Faulting instruction is restored, PT Lookup --> No fault



Role of OS

- Page fault:
  - How does the OS know the location of the page on disk, given the virtual address?
- For each page of a process, in the virtual address space, OS keeps track of the location on disk in a “swap space” on disk. This happens when it creates a process
  - This information can also be part of PT with the address on disk


Virtual page number
Page table
Physical page or disk address
Valid
1
1
1
1
0
1
1
0
1
1
0
1
Physical memory
Disk storage

Writes to pages in memory

- Note that physical memory or main memory acts as a cache for the hard disk
- Virtual memory systems must use write-back, performing writes into the page in memory, and copying the page back to disk when it is replaced in the memory.
  - “Dirty bit” concept

Page table entries

- Consider a physical memory of size 4GB

  Virtual memory 32GB. Page size: 4kB

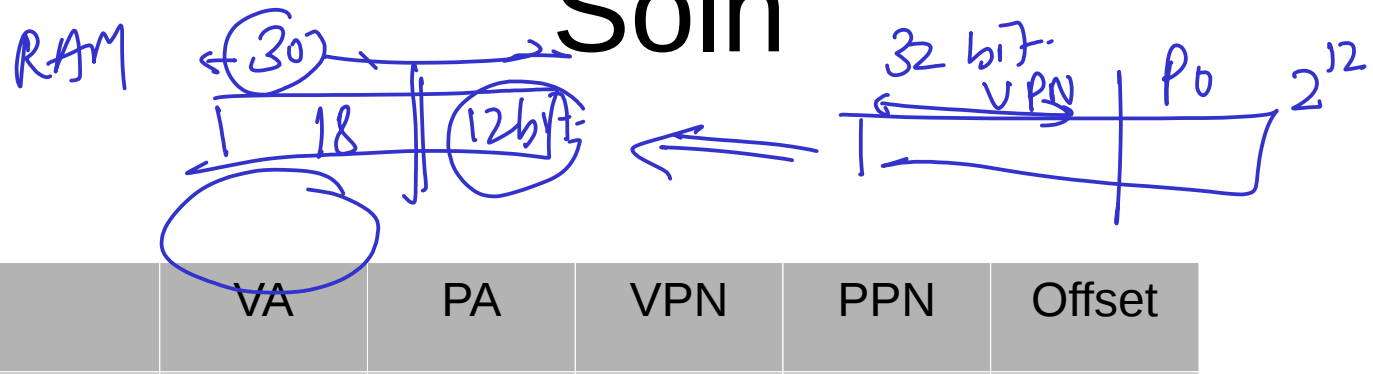
- How many page table entries does the page table has?

Eg

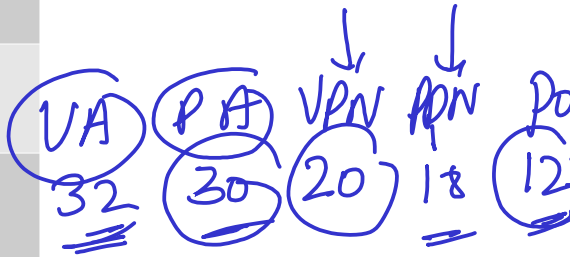
How many bits are needed for each of the following: Virtual address, Physical address, Virtual page number, Physical page number, Page Offset

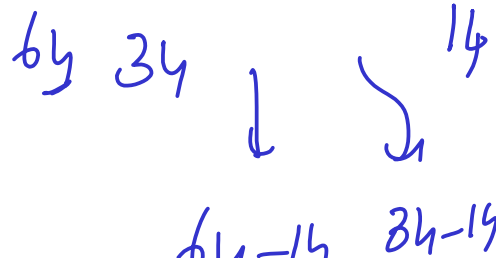
- 32-bit ISA, 4-KB pages, 1 GB of RAM
- 32-bit ISA, 16-KB pages, 2 GB of RAM
- 64-bit ISA, 16-KB pages, 16 GB of RAM

Soln



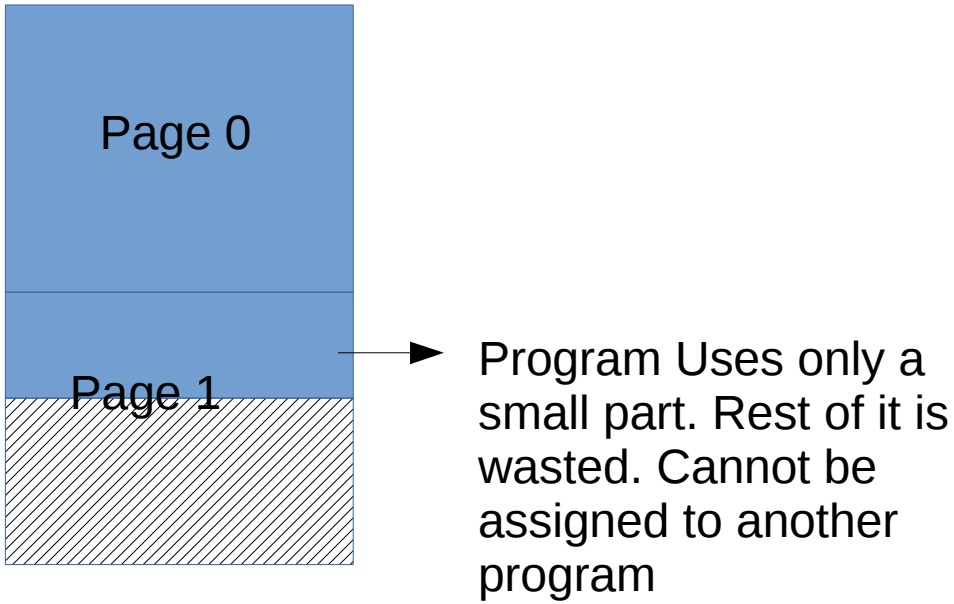
	VA	PA	VPN	PPN	Offset
1	32	30	20	18	12
2	32	31	18	17	14
3	64	34	50	20	14





Page size

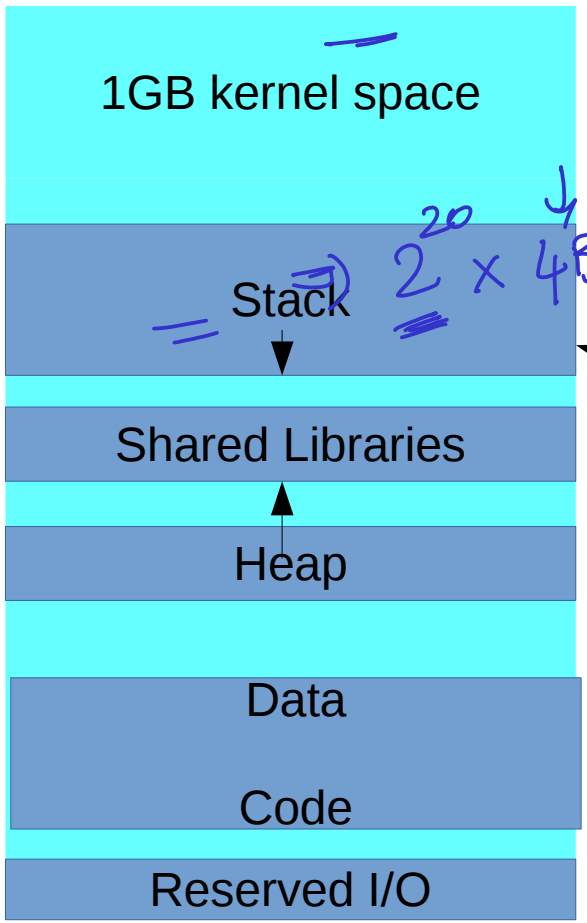
- Small pages: Large Page table
- Large pages: Small PT, but will have to replace pages in large chunks.
  - Also suffer from internal fragmentation
- Typical sizes: Few kB to a few MB (4kB, 32MB)



Problem 1: Size of a flat page table

- Page tables are stored in RAM/physical memory
- 1 entry per page in Virtual address space
  - Number of page table entries (PTE) = Virtual memory size/ Page size
    - 4GB/4kB
  - PT Size = No of PTE * Size of each entry
    - (4GB/4kB)*4B = 4MB per program or per process
    - If Virtual address space is 64 bits, then PT size will not even fit in RAM! (4*10^15)
  - Solution: Multi-level page tables

Problem 2: Virtual address space



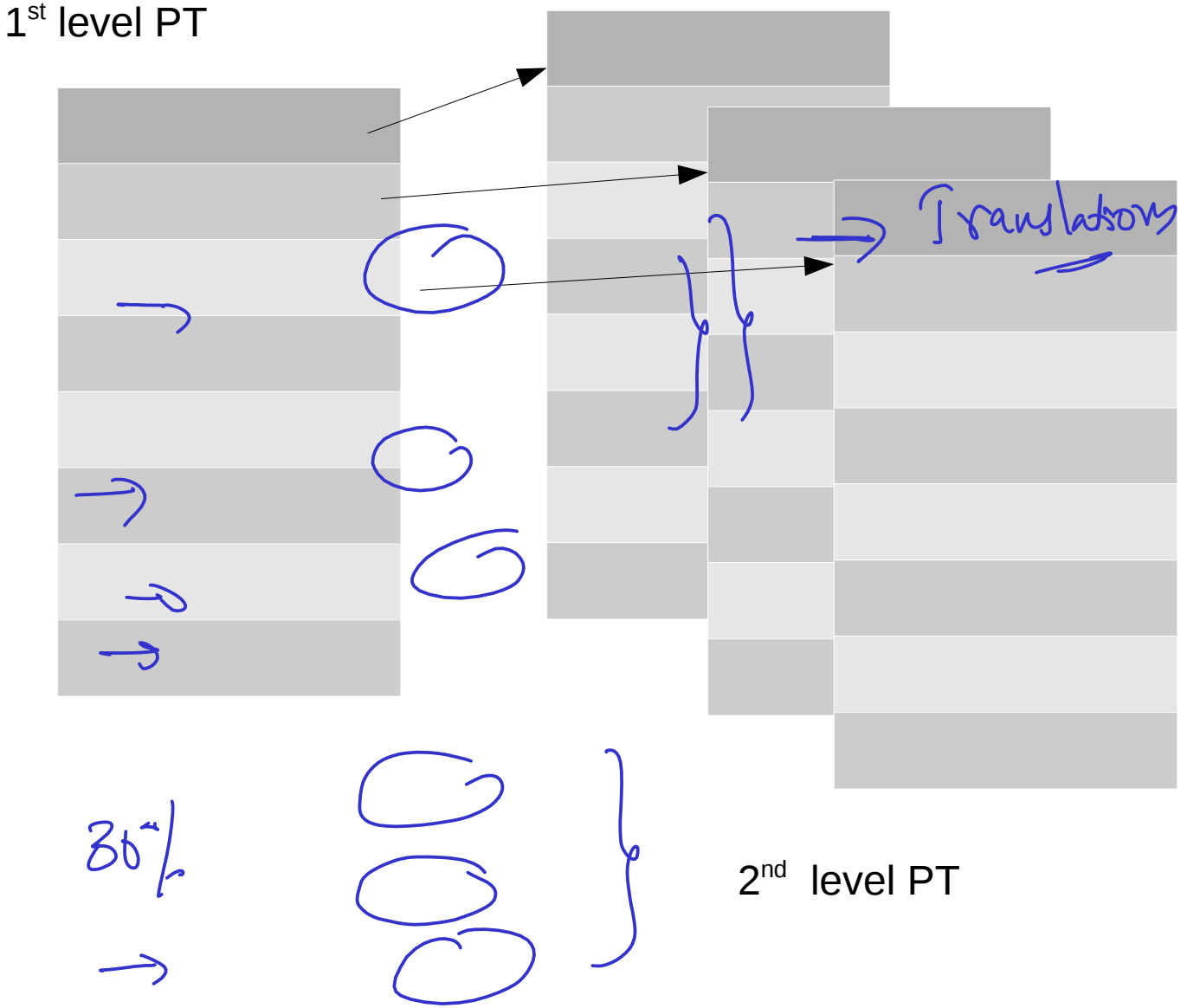
The entire 4GB Virtual address space will not be used a program

A program uses only a limited part of this address space: Code, Data, Stack, heap

But, the Page table entries will still be available for the entire space (like the unused part, kernel space etc) and **these occupy space**

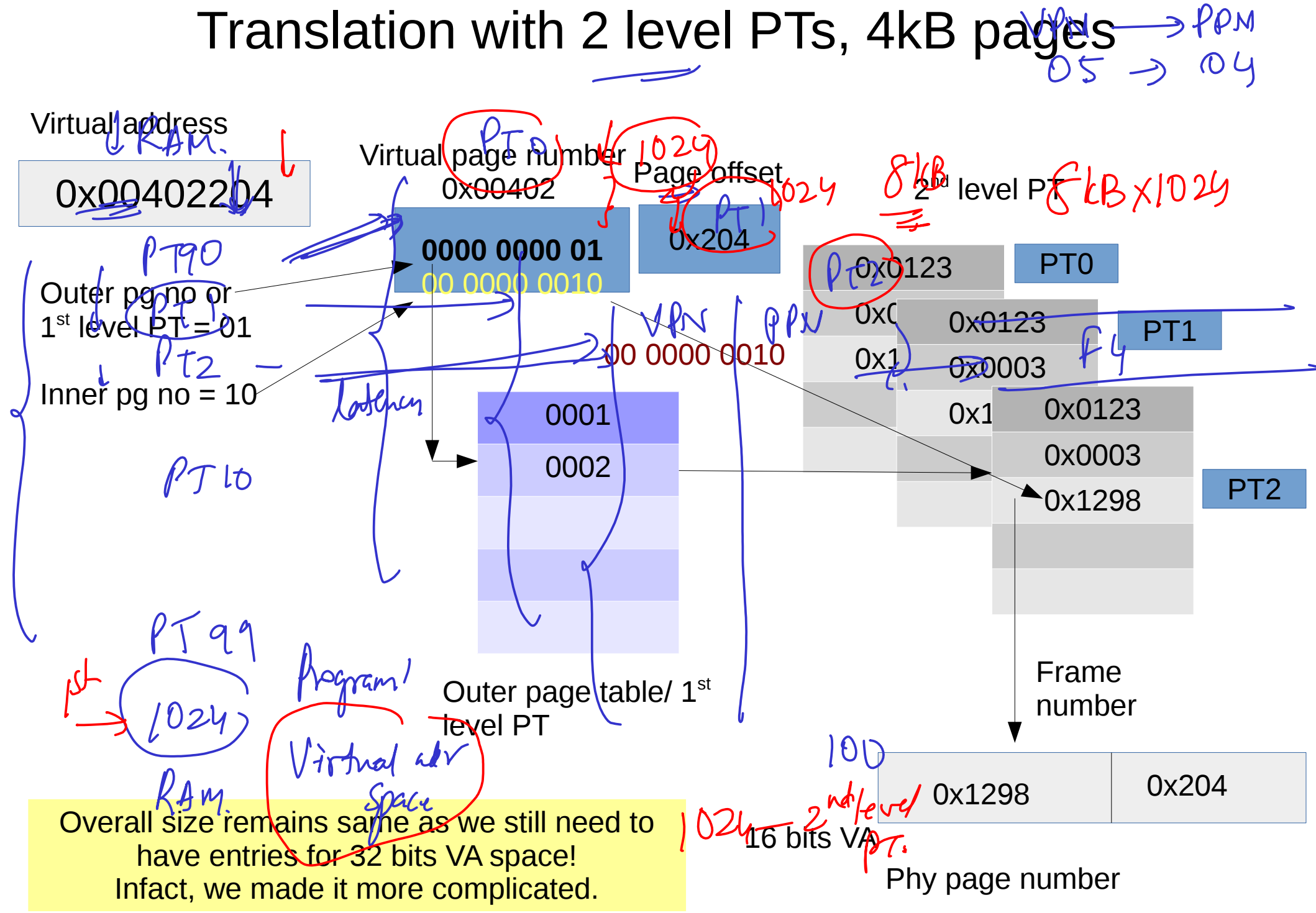
Can we omit these page table entries?

Multi-level PT
Paging the page tables

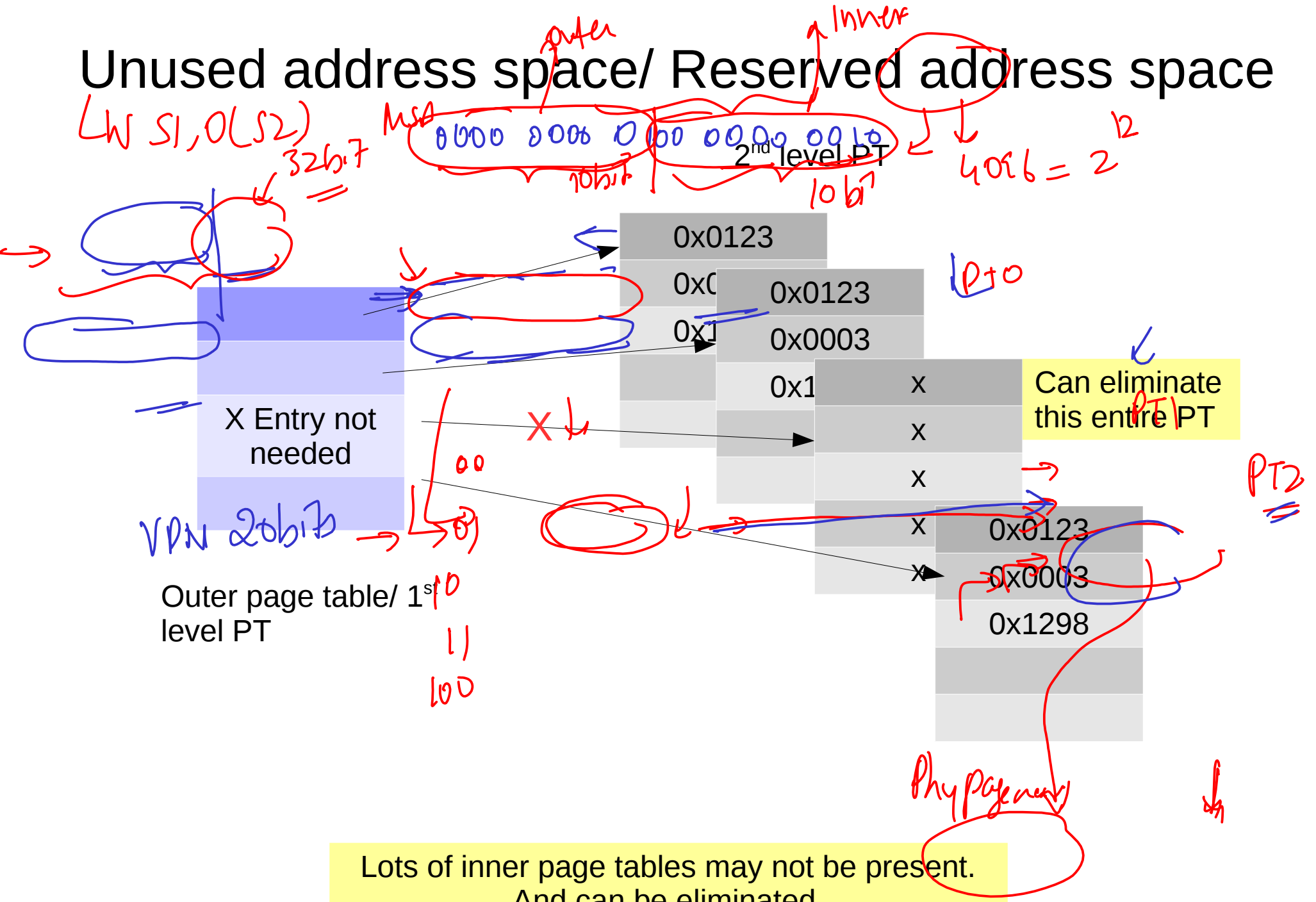


Memory (RAM)

Translation with 2 level PTs, 4kB pages

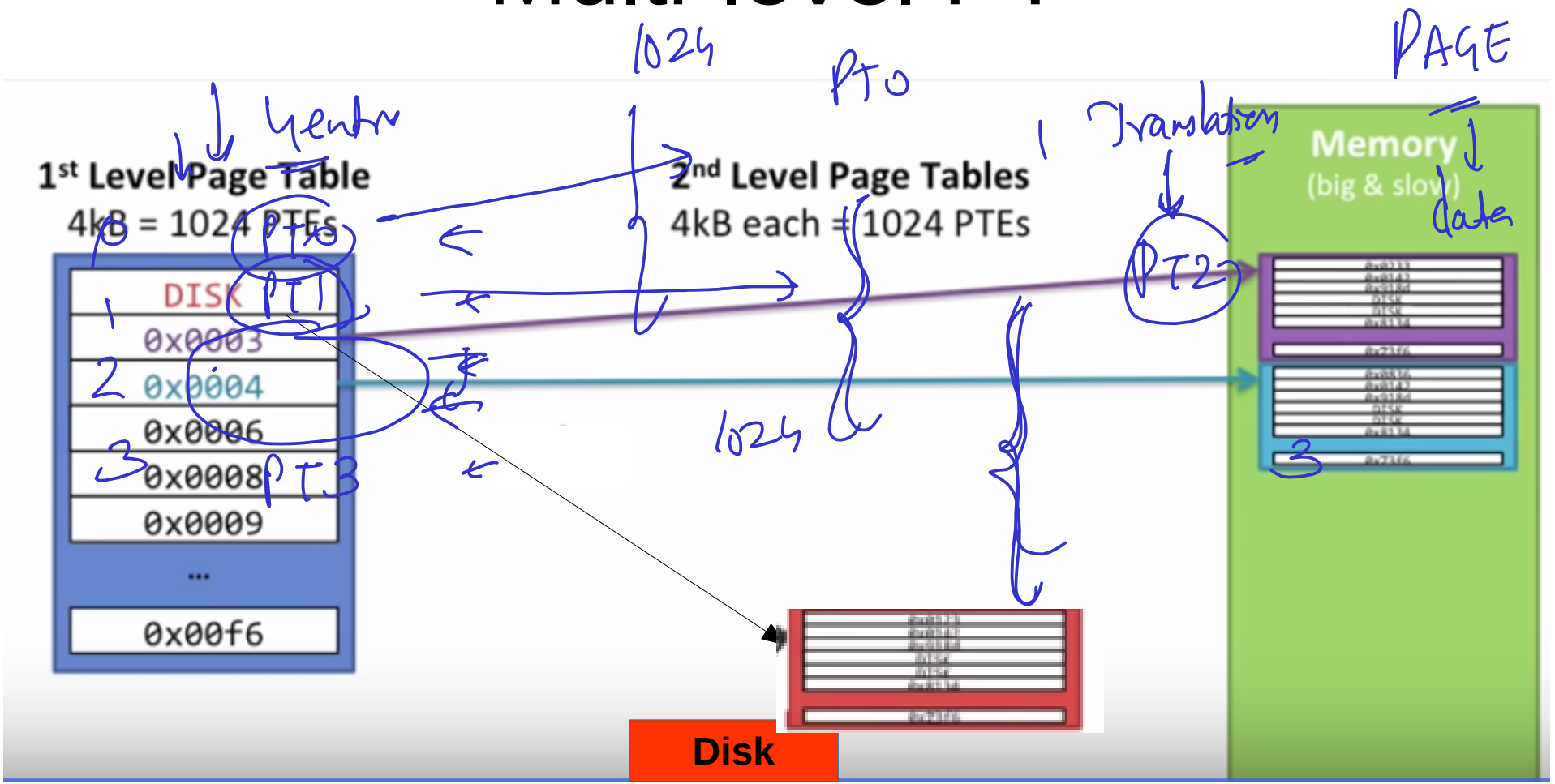


Unused address space/ Reserved address space



Lots of inner page tables may not be present.
And can be eliminated.
This will not be possible in a flat page table

Multi-level PT



1st level PT should be in RAM, 2nd level can be on disk, since we can easily find the PA through 1st level PT

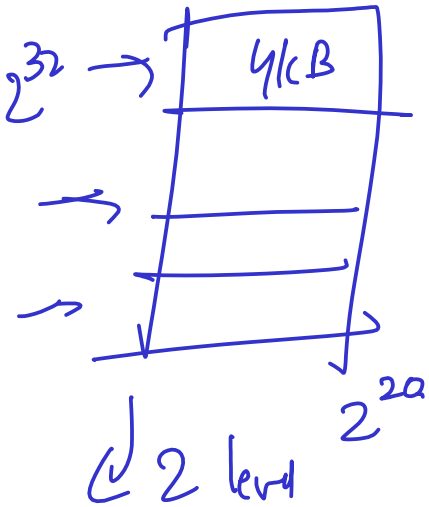
Single vs 2 level PT

- 32 bit VA, 4kB page, 8 bytes per entry
- 1024 entry outer and inner page tables
- Program uses VM from address

  0 to 0x00010000

Single or flat Page Table size:
Virtual address size/Page size = No of pages
2^32 VA/2^12 = 2^20 pages or PT entries
2^20 * 8 bytes per entry = 8MB

Single vs 2 level PT

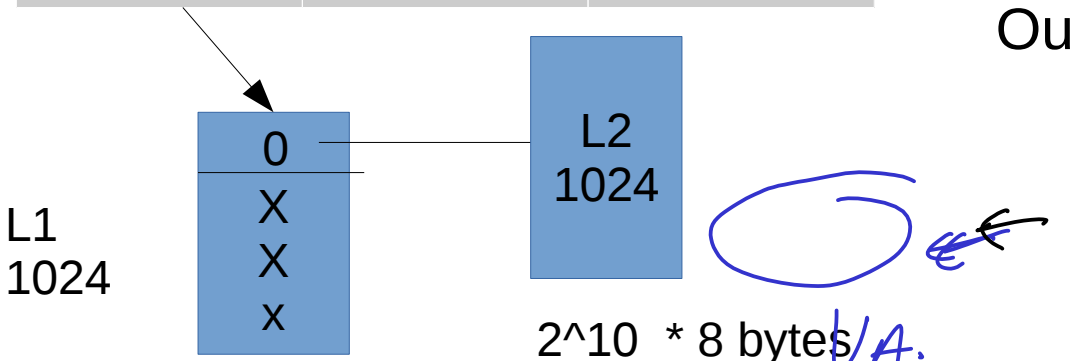


- 32 bit VA, 4kB page, 8 bytes per entry
- 1024 entry outer and inner page tables
- Program uses VM from address

  0 to 0x00010000

Virtual address: 32 bits

Outer pg size	Inner pg size	offset
10	10	12



32 bit virtual address

0000 0000 00 00 0001 000...000

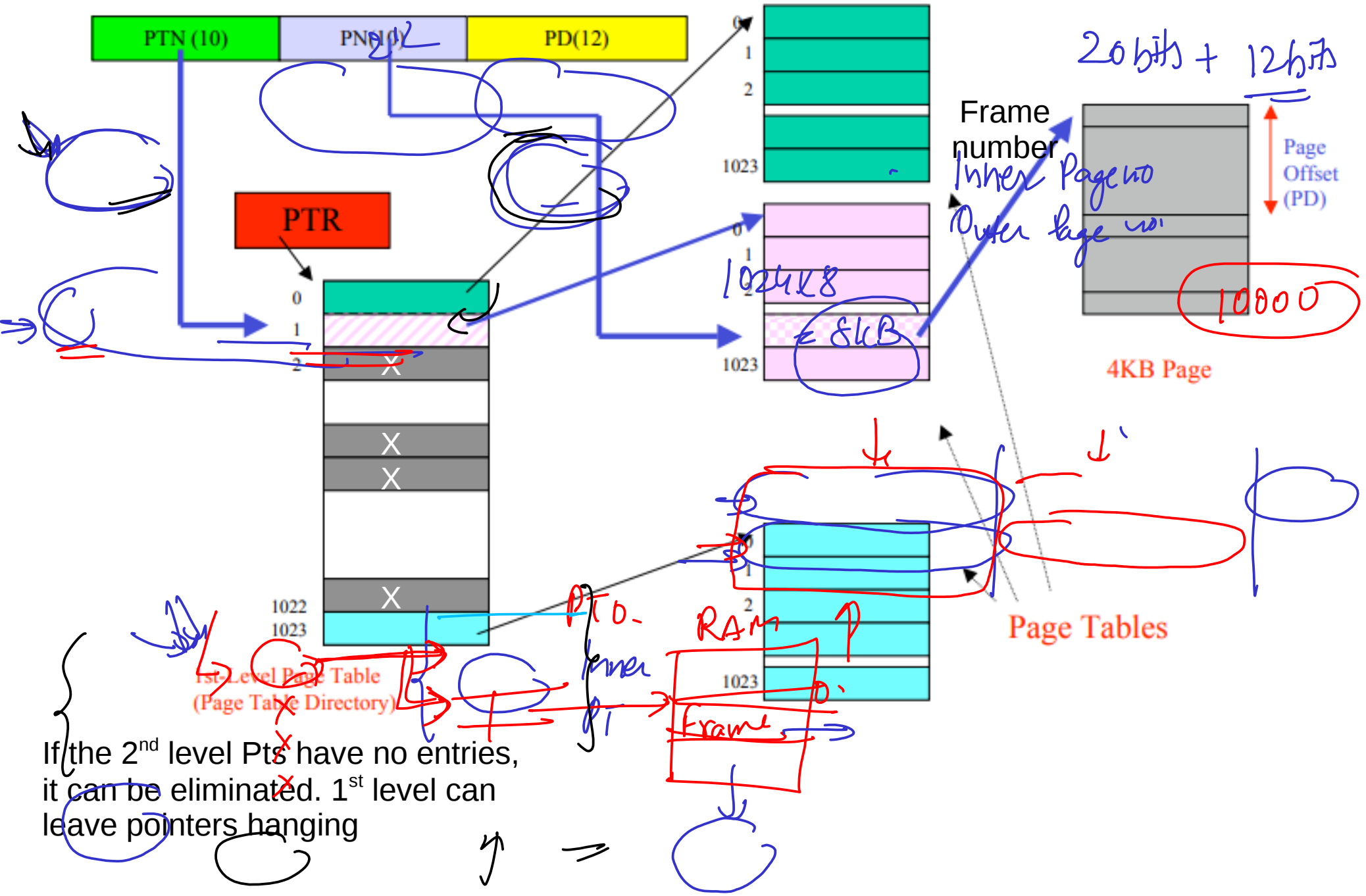
Outer pg no | Inner pg no | Offset

Only Outer Page no – 0 being used for this entire address range

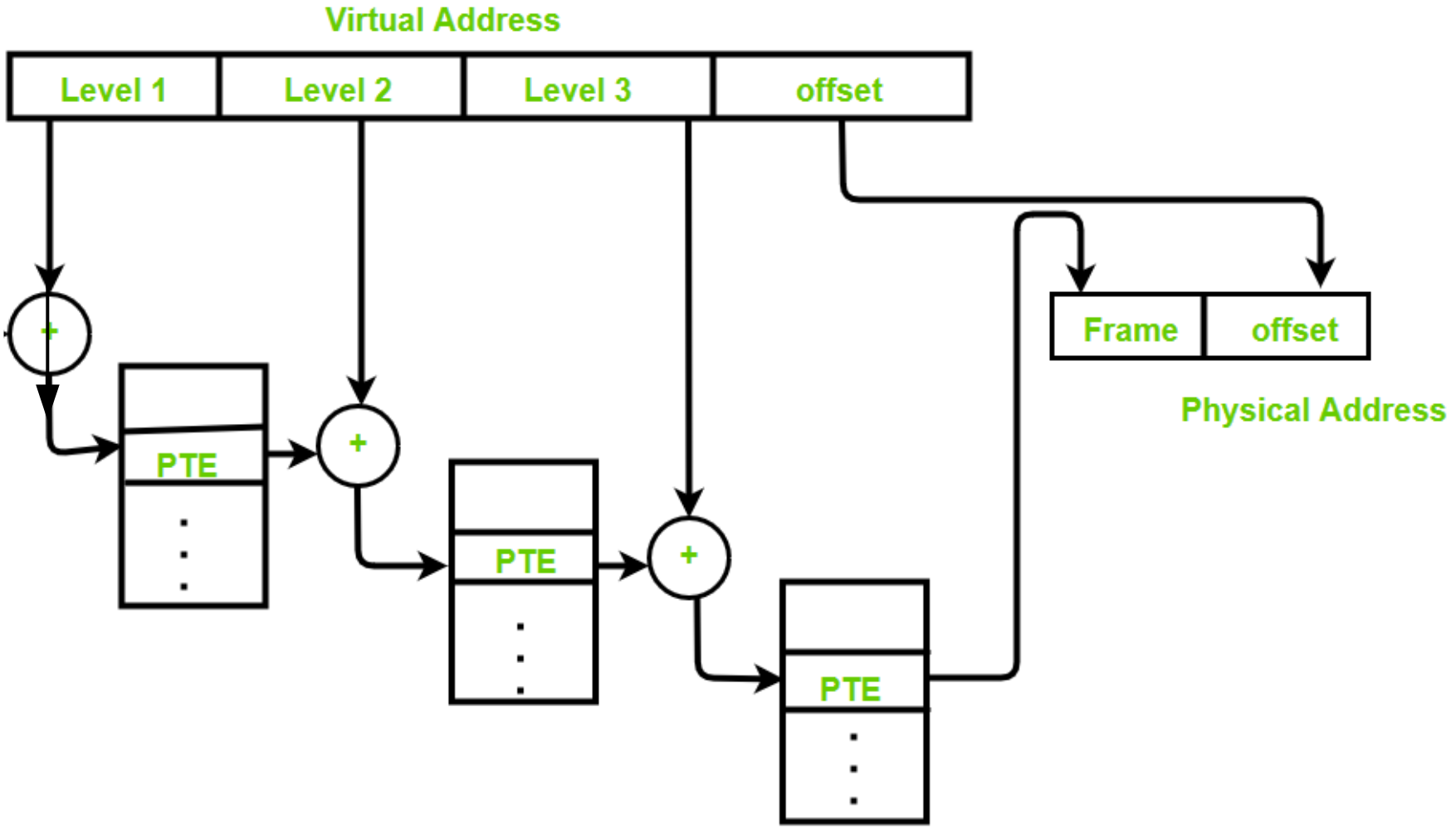
Others can be left: x

16kB total size- huge memory saving compared to flat page table

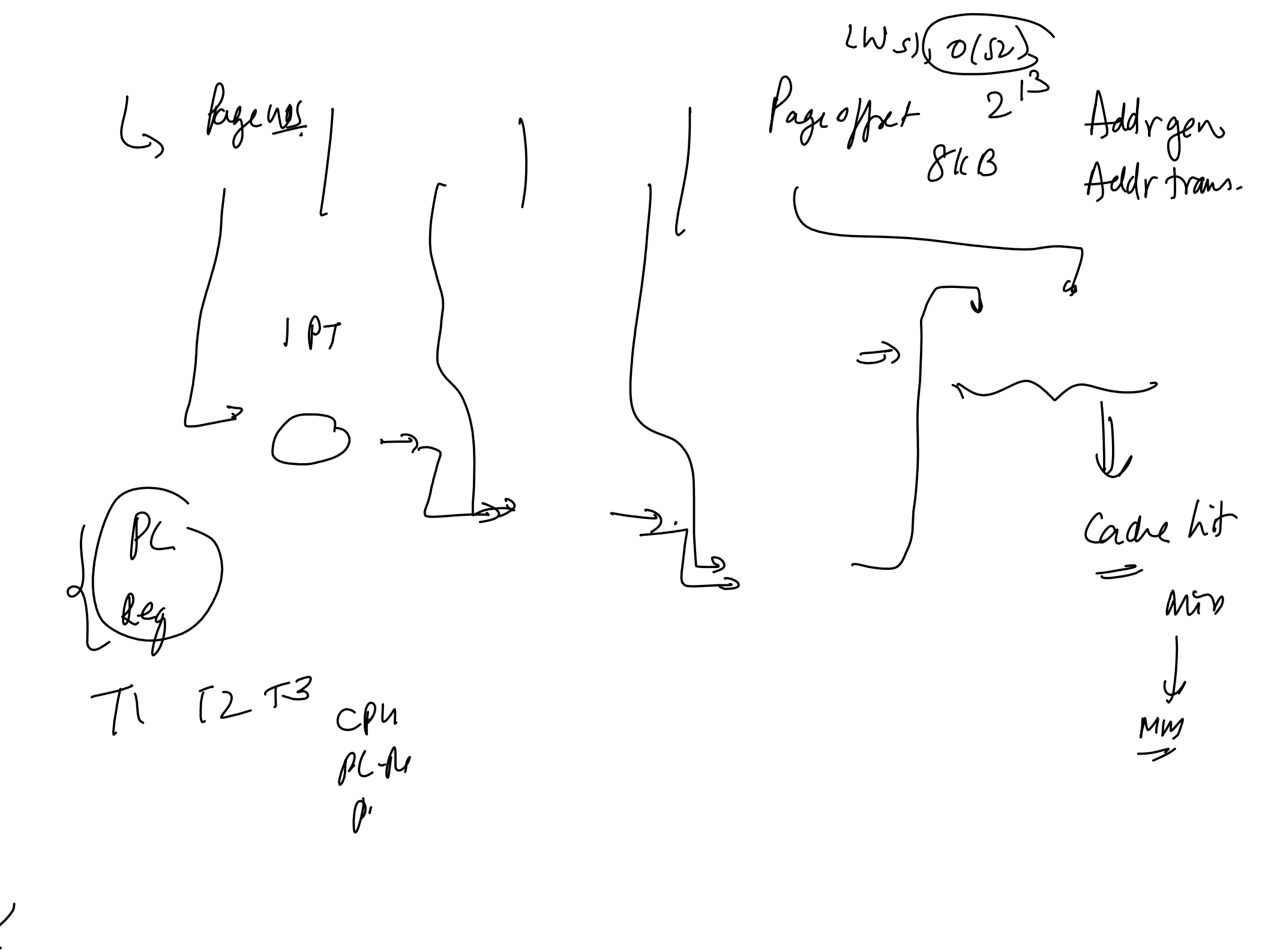
https://www.clear.rice.edu/comp425/slides/L31.pdf



If the 2nd level Pts have no entries, it can be eliminated. 1st level can leave pointers hanging



3 Level paging system


LW s1, 0(s2)
Page offset
2^{13}
8kB
Addr gen
Addr trans.
Page no.
IPT
PC
Reg
T1 T2 T3
CPU
Cache hit
MM