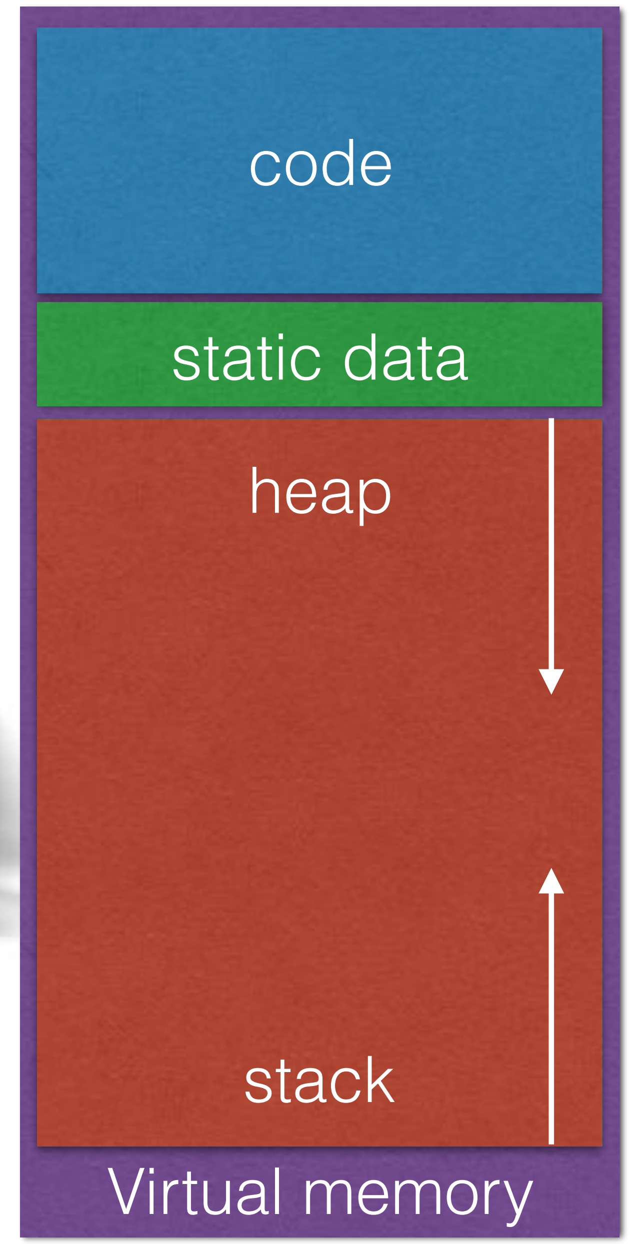
When memory hierarchy meets virtual memory

Hung-Wei Tseng

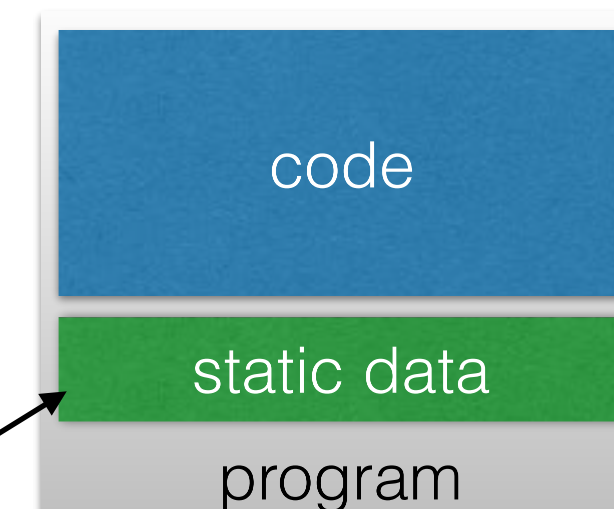
Virtual memory

What happens when creating a process



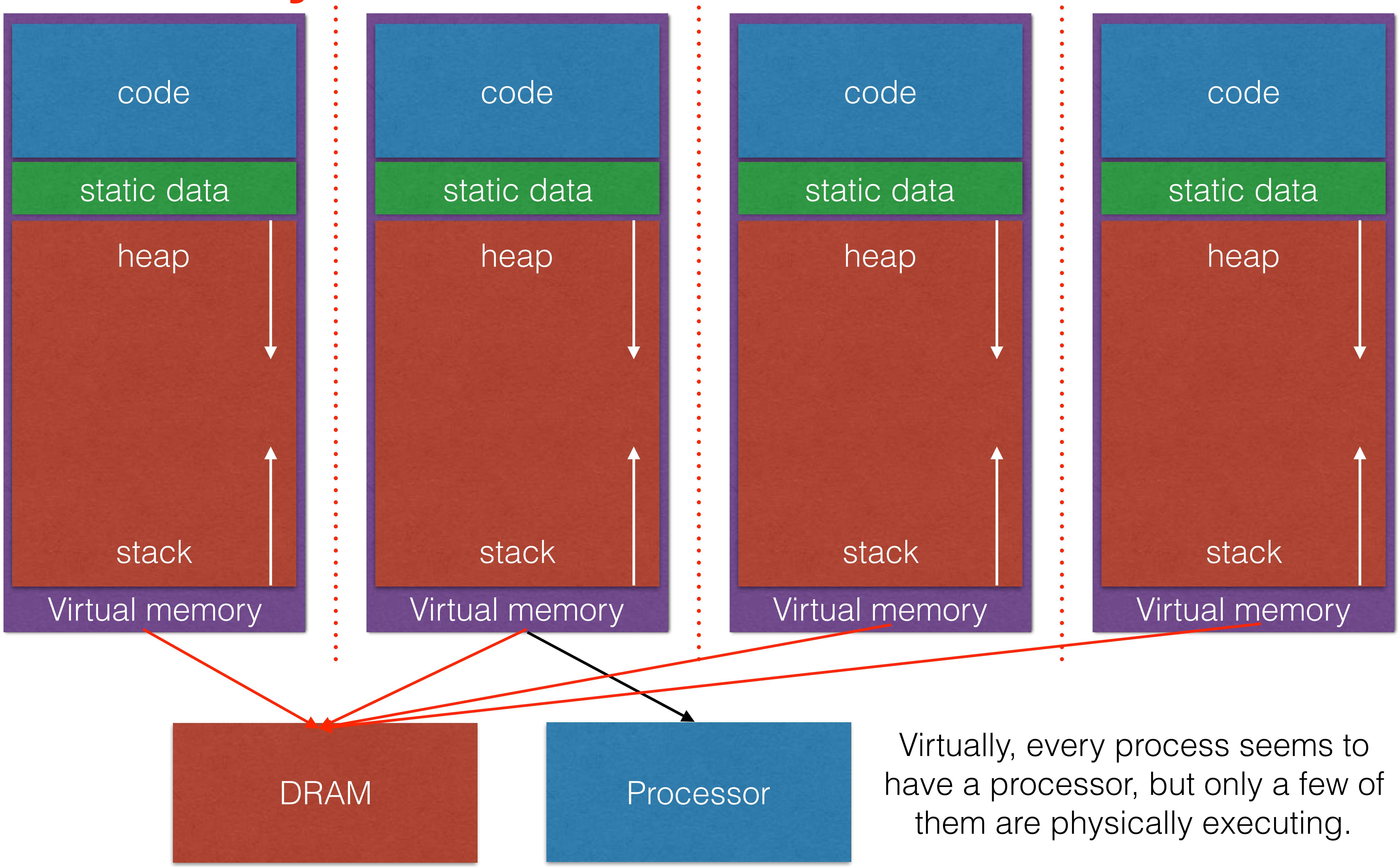
Dynamic allocated data: `malloc()`

Local variables, arguments



Linux contains a .bss section for uninitialized global variables

Previously, we talked about virtualization

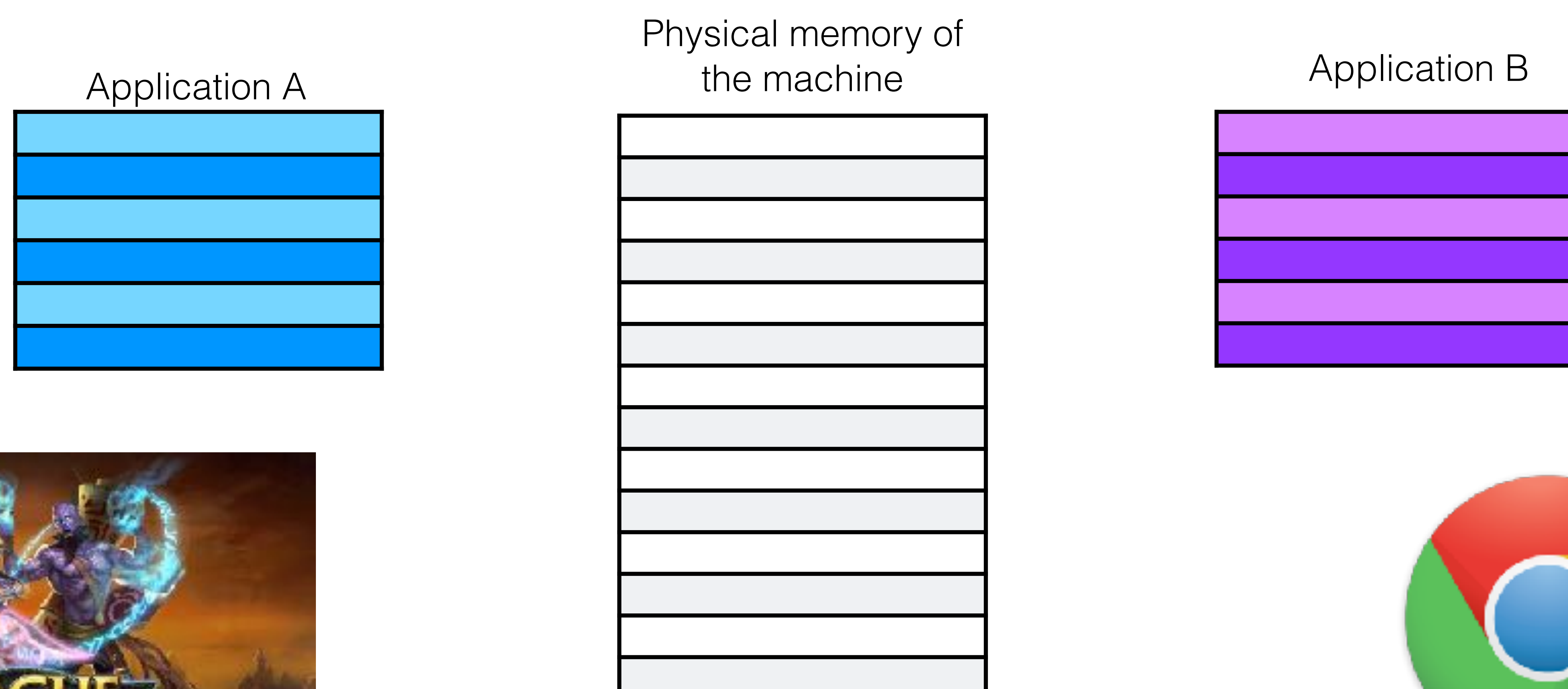


Virtually, every process seems to have a processor, but only a few of them are physically executing.

How to share DRAM?

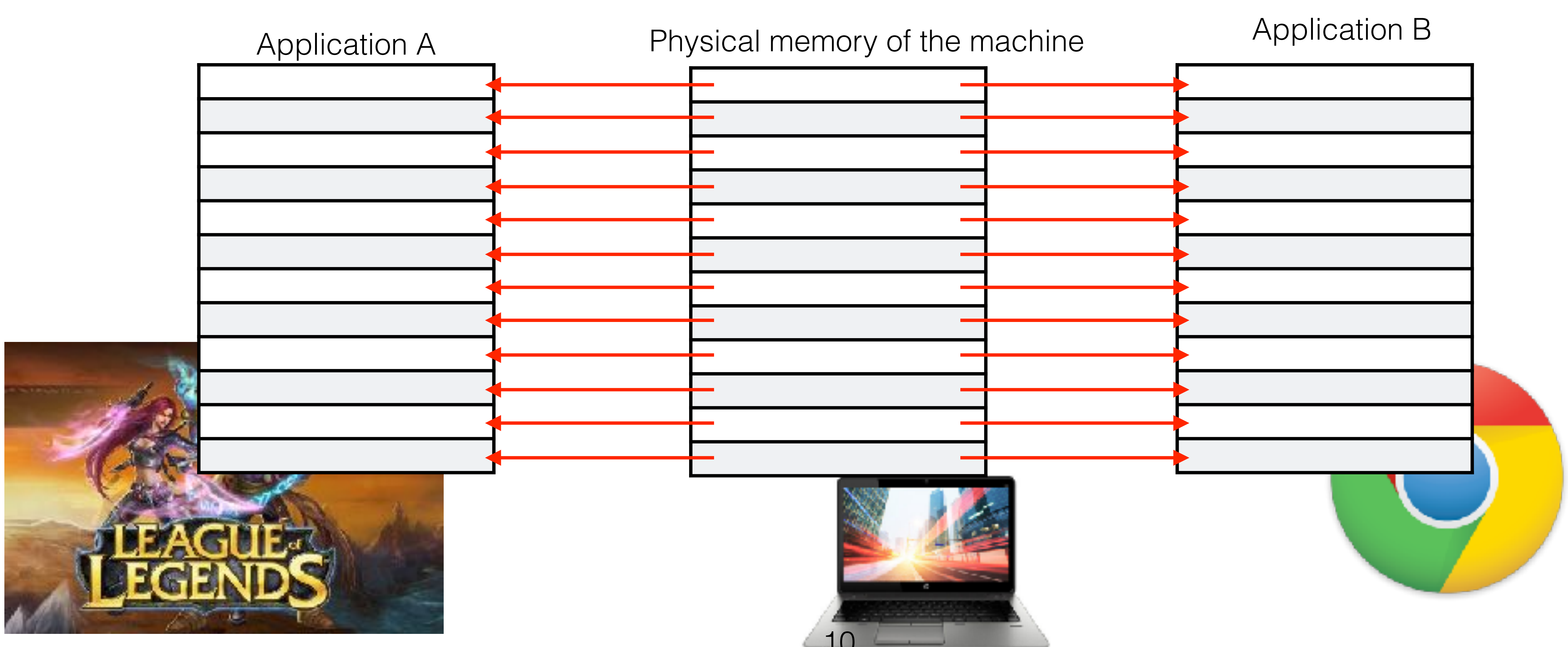
Many applications, one memory

- Both application A and application B would like to use the same machine, the same physical memory
- Each application wants to own memory
- Each application should not touch data of the other



Or — we just cannot have big enough memory

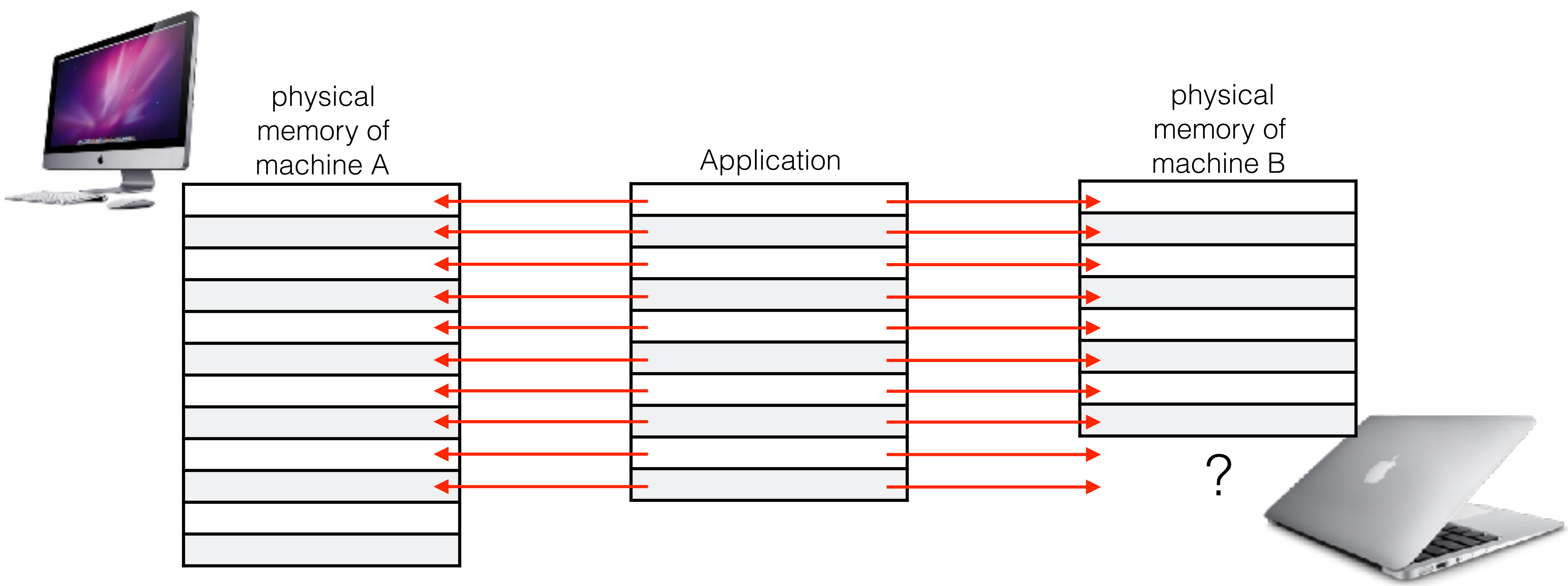
- Both application A and application B would like to use the same machine and the sum of their memory demands exceeds the available physical memory?



Or what if — mine is larger than yours?

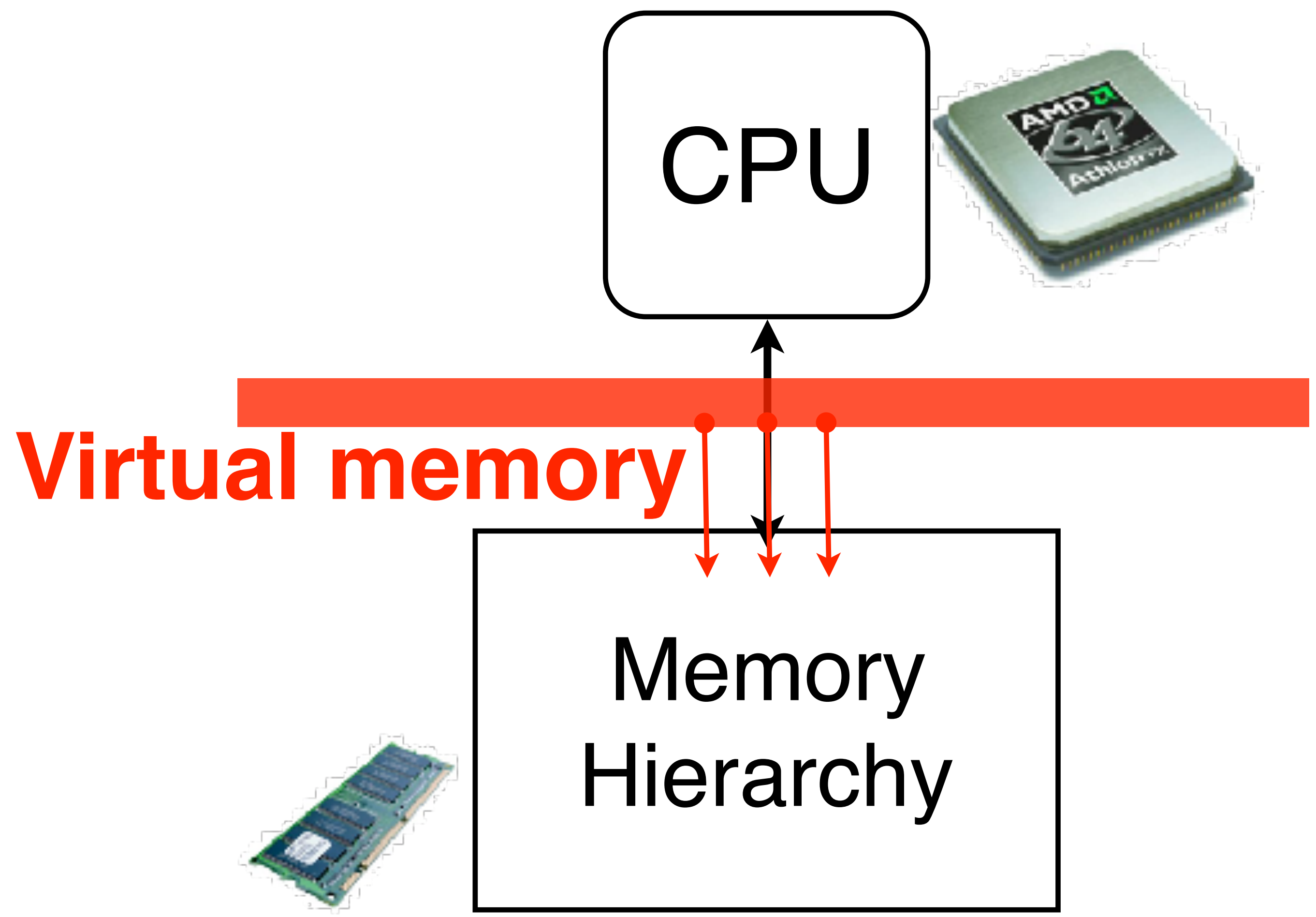
- My program fit in machine A's memory, but not in machine B?

Portability

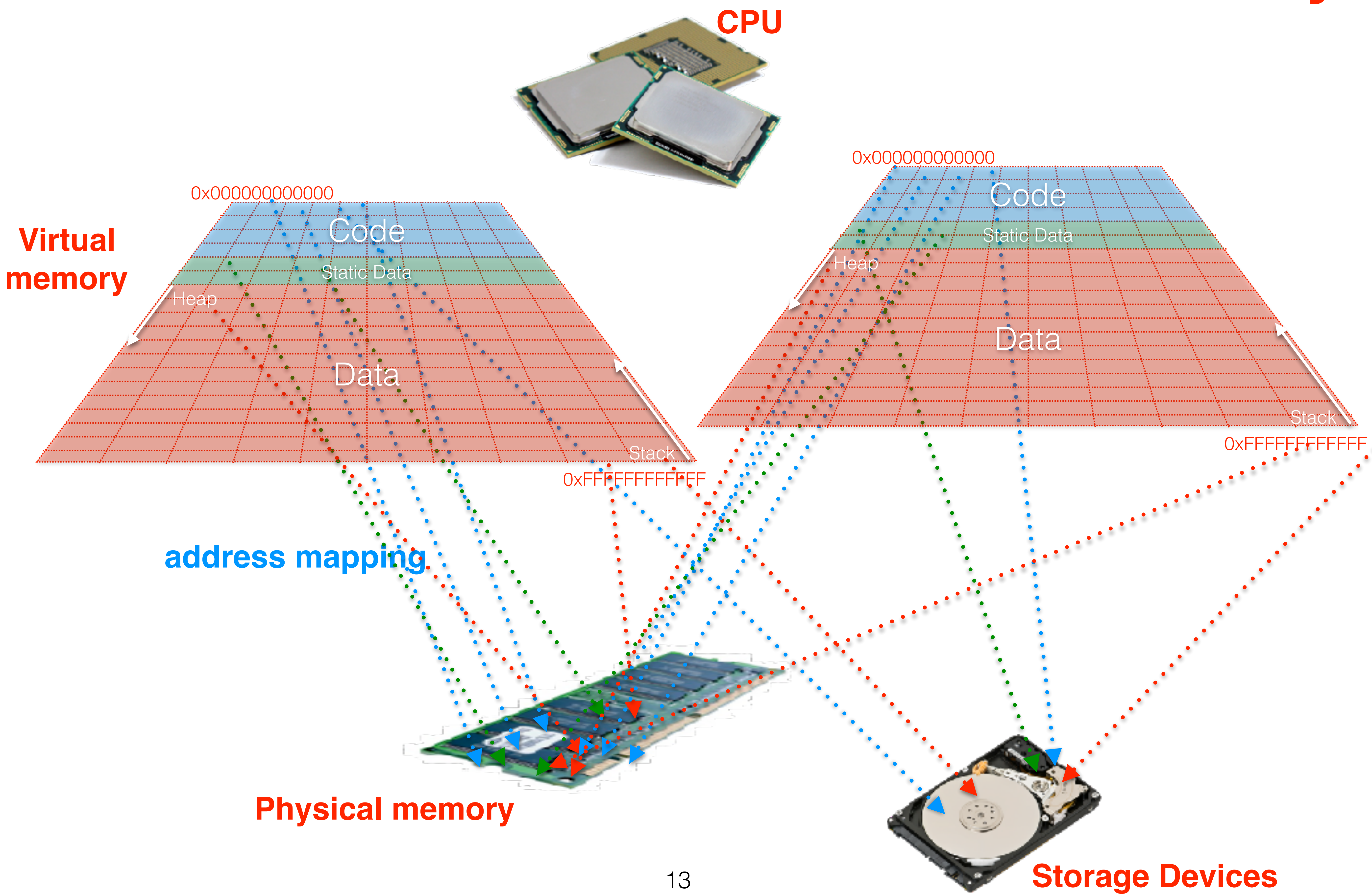


Virtual memory

- Every program lives in the virtual memory address space
- The machine instructions use virtual memory addresses
- The data are allocated in the virtual memory address space
- The CPU works with OS to figure out how virtual memory address map to physical locations



When cache meets virtual memory

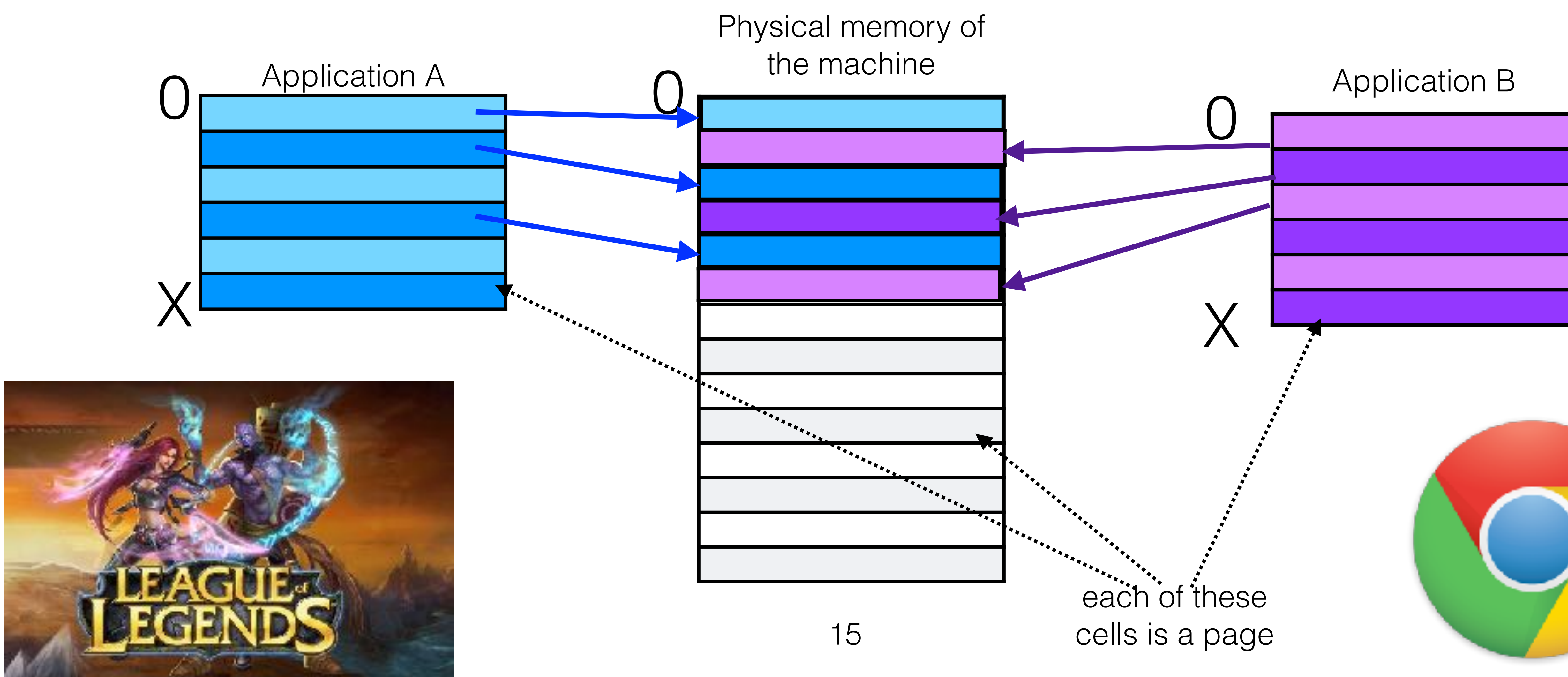


Demand paging

- **Paging**: partition virtual/physical memory spaces into fix-sized pages
- **Demand paging**: Allocate a physical memory page for a virtual memory page when the virtual page is needed
  - There is also **shadow paging** used by embedded systems, mobile phones — they load the whole program/data into the physical memory when you launch it

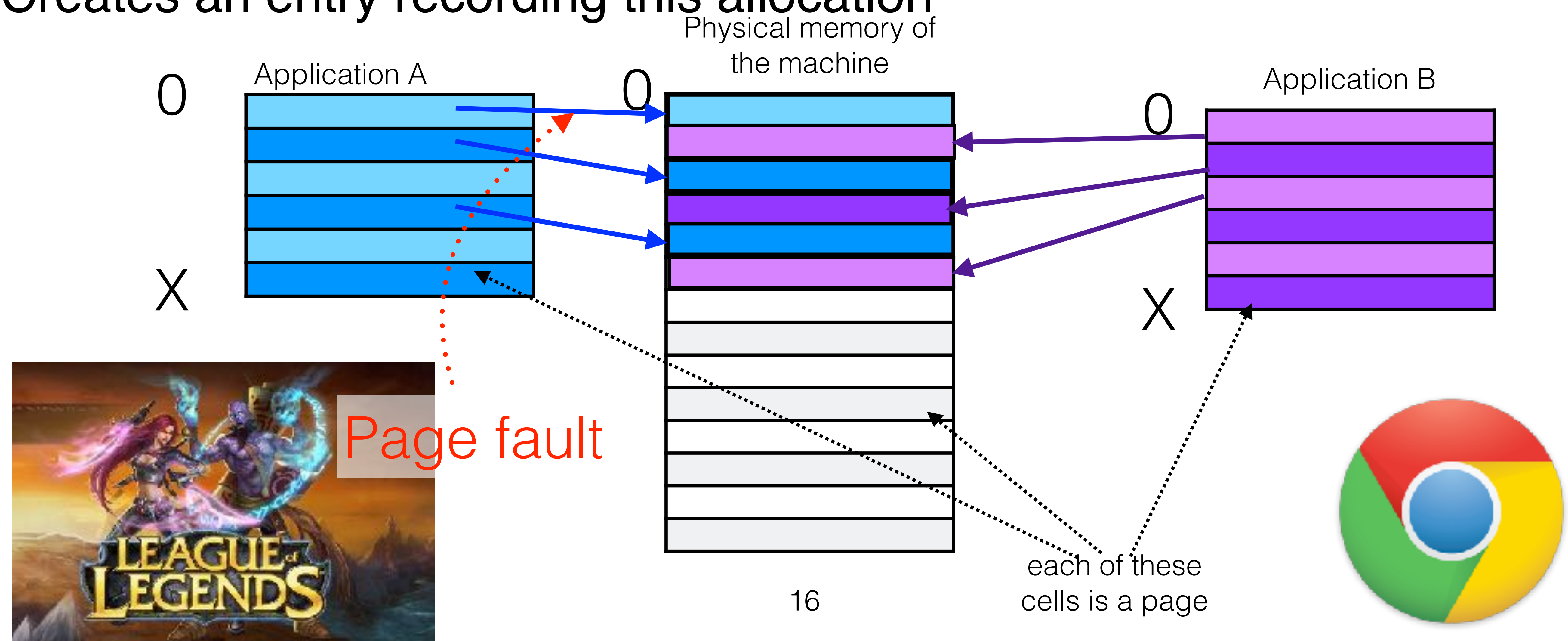
Demand paging

- **Paging**: partition virtual/physical memory spaces into fix-sized pages
- **Demand paging**: Allocate a physical memory page for a virtual memory page when the virtual page is needed



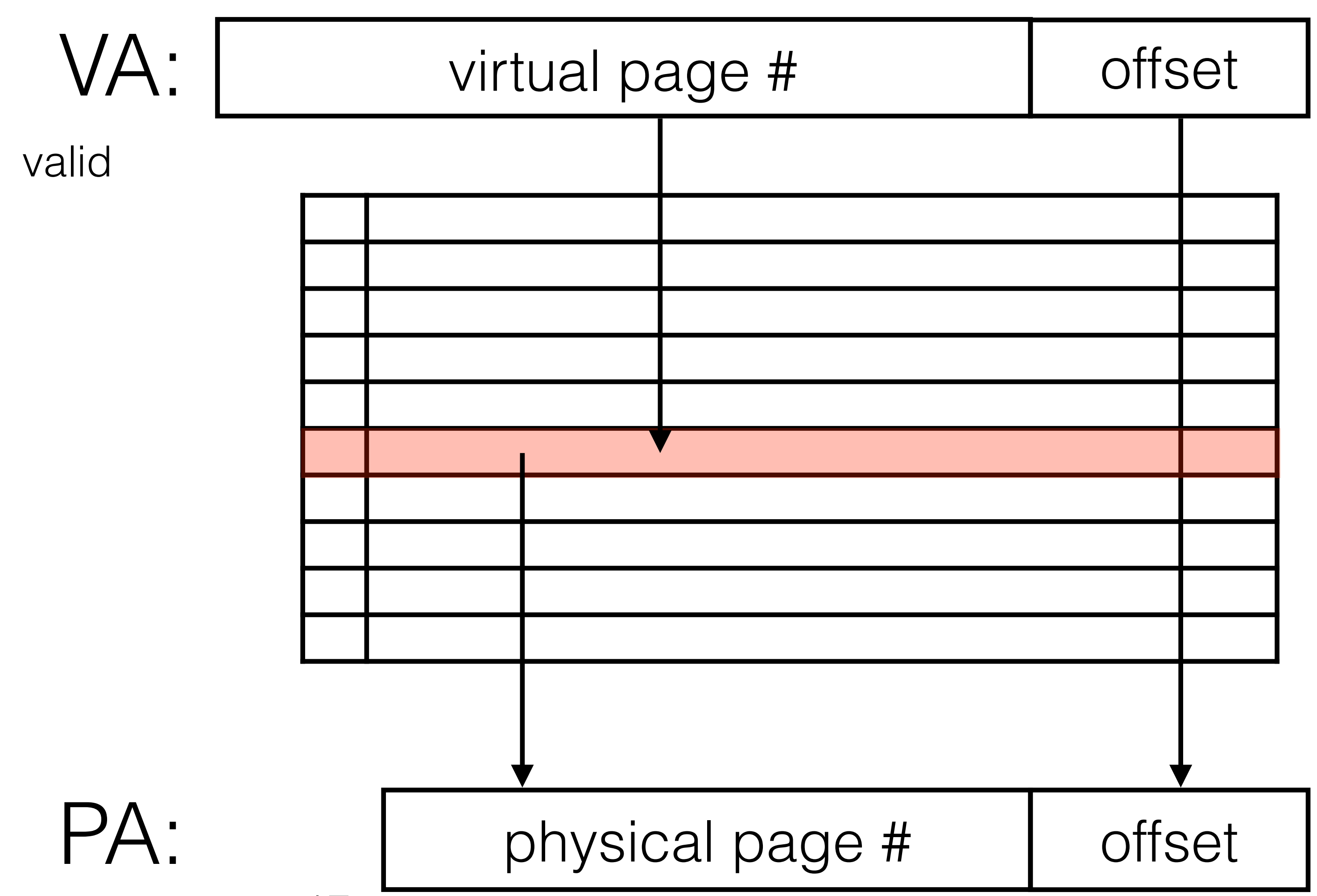
Page fault

- Page fault: if the demanding page is not in the physical memory
- How to handle page fault: the processor raises an **exception** and transfers the control (change the PC) to the page fault handler in OS code
    - Allocates a physical memory location for the page
    - Creates an entry recording this allocation **— where?**



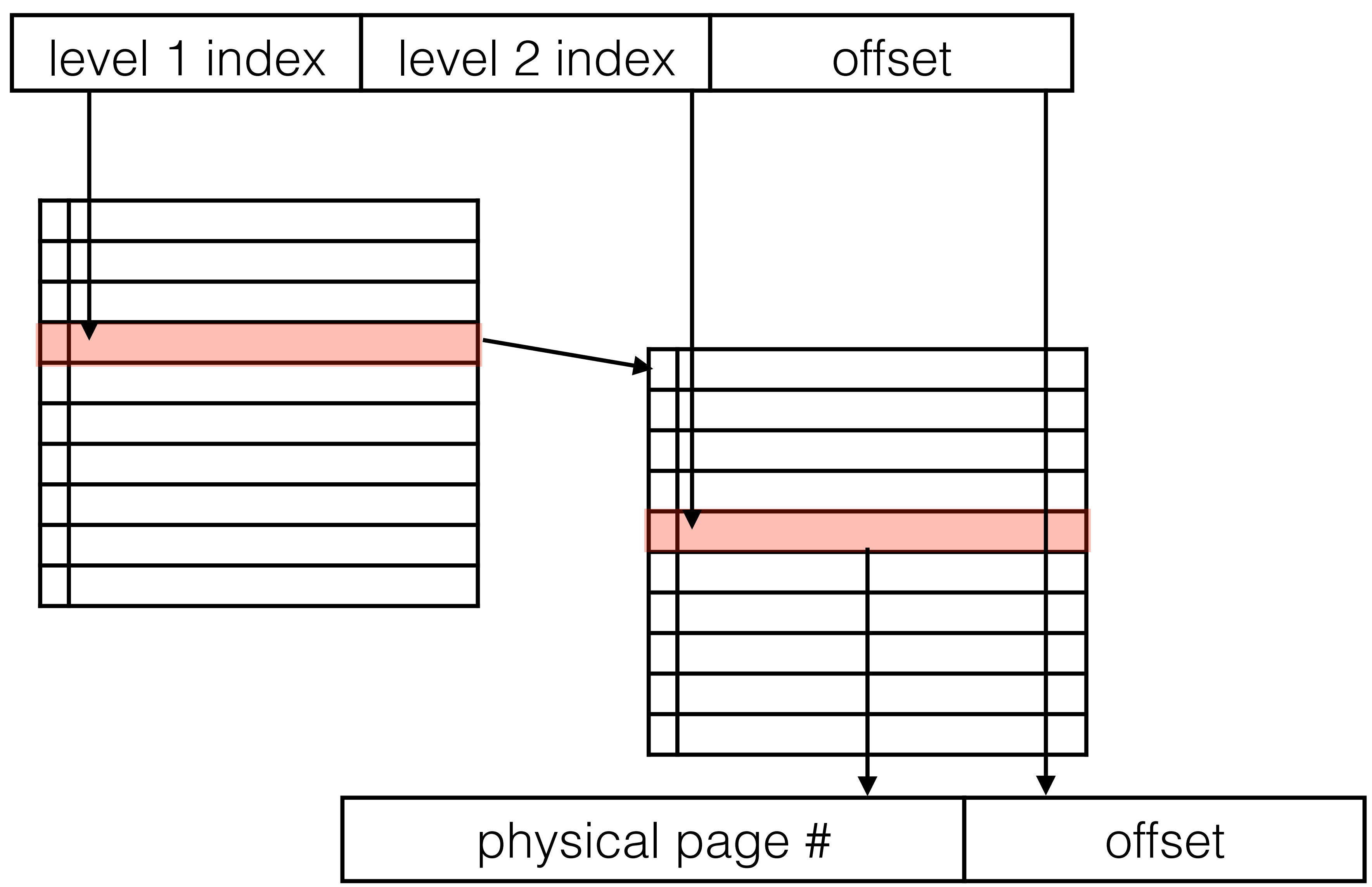
Address translation

- Processor receives virtual addresses from the running code, main memory uses physical memory addresses
- Virtual address space is organized into "pages"
- The system references the **page table** to translate addresses
  - Each process has its own page table
  - The page table content is maintained by OS



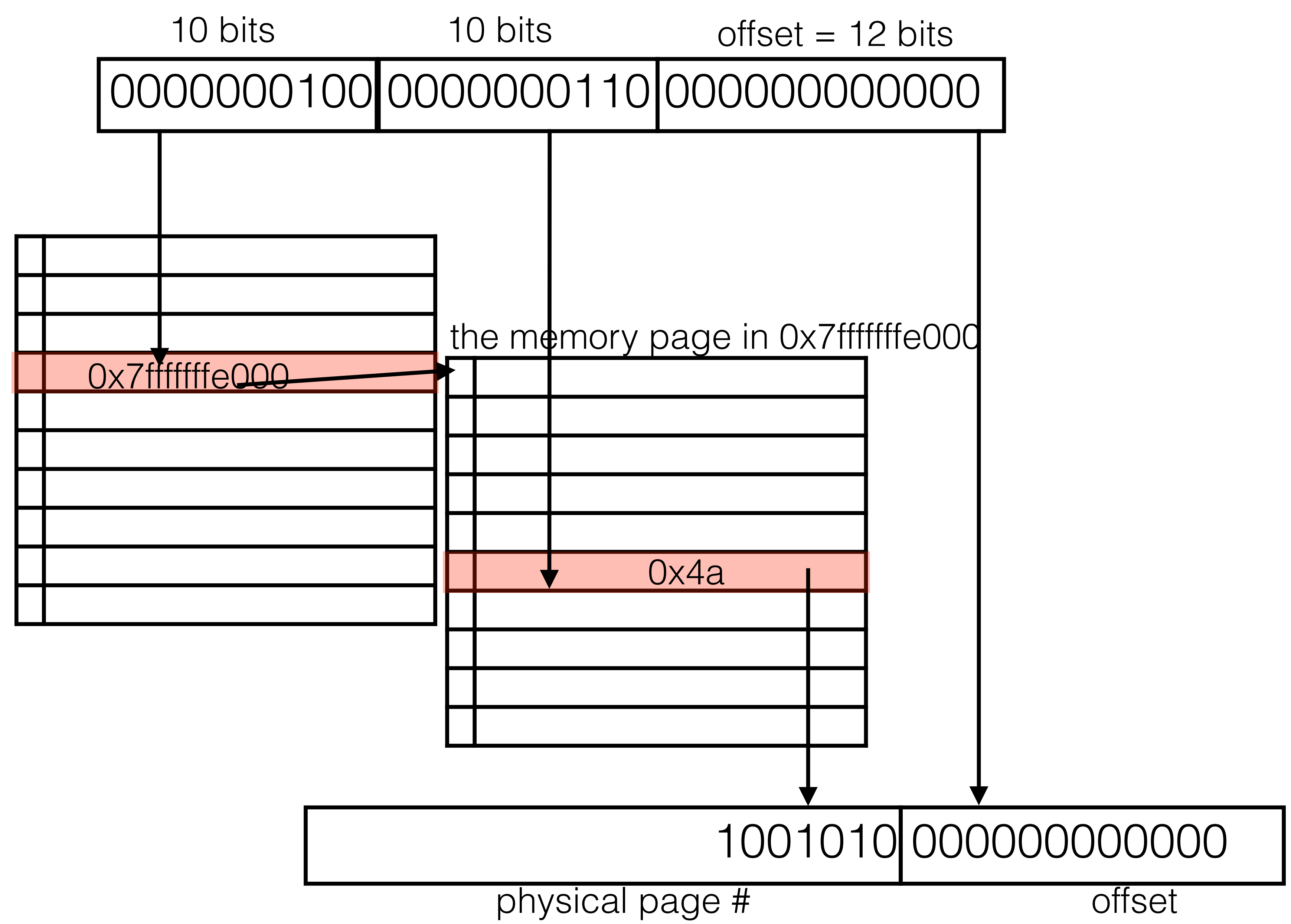
Hierarchical page table

- Break the virtual page number into several pieces
- If one piece has N bits, build an 2^N-ary tree
- Only store the part of the tree that contain valid pages
- Walk down the tree to translate the virtual address



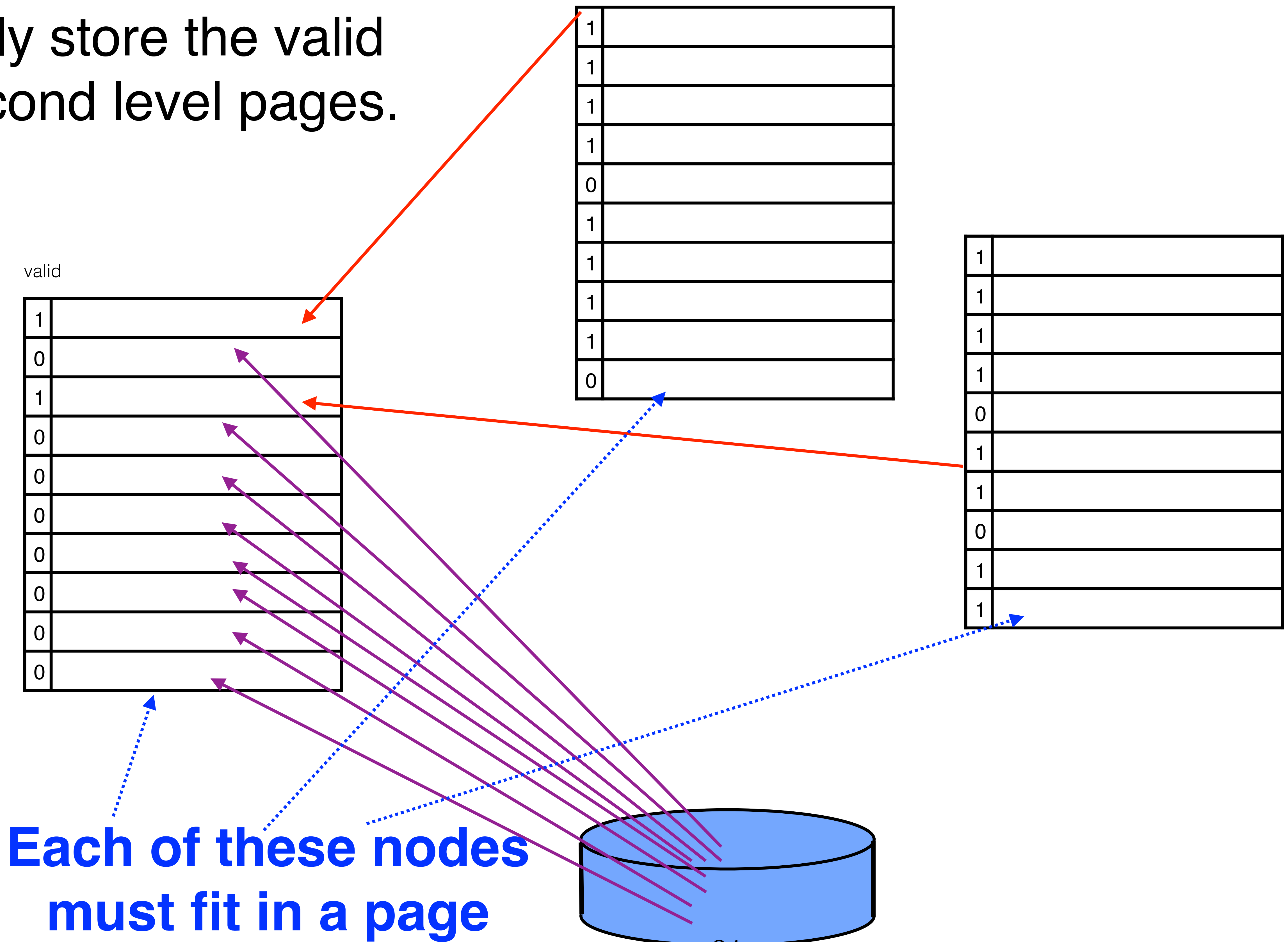
Page table walking example

- Two-level, 4KB, 10 bits index in each level
- If we are accessing 0x1006000 now…



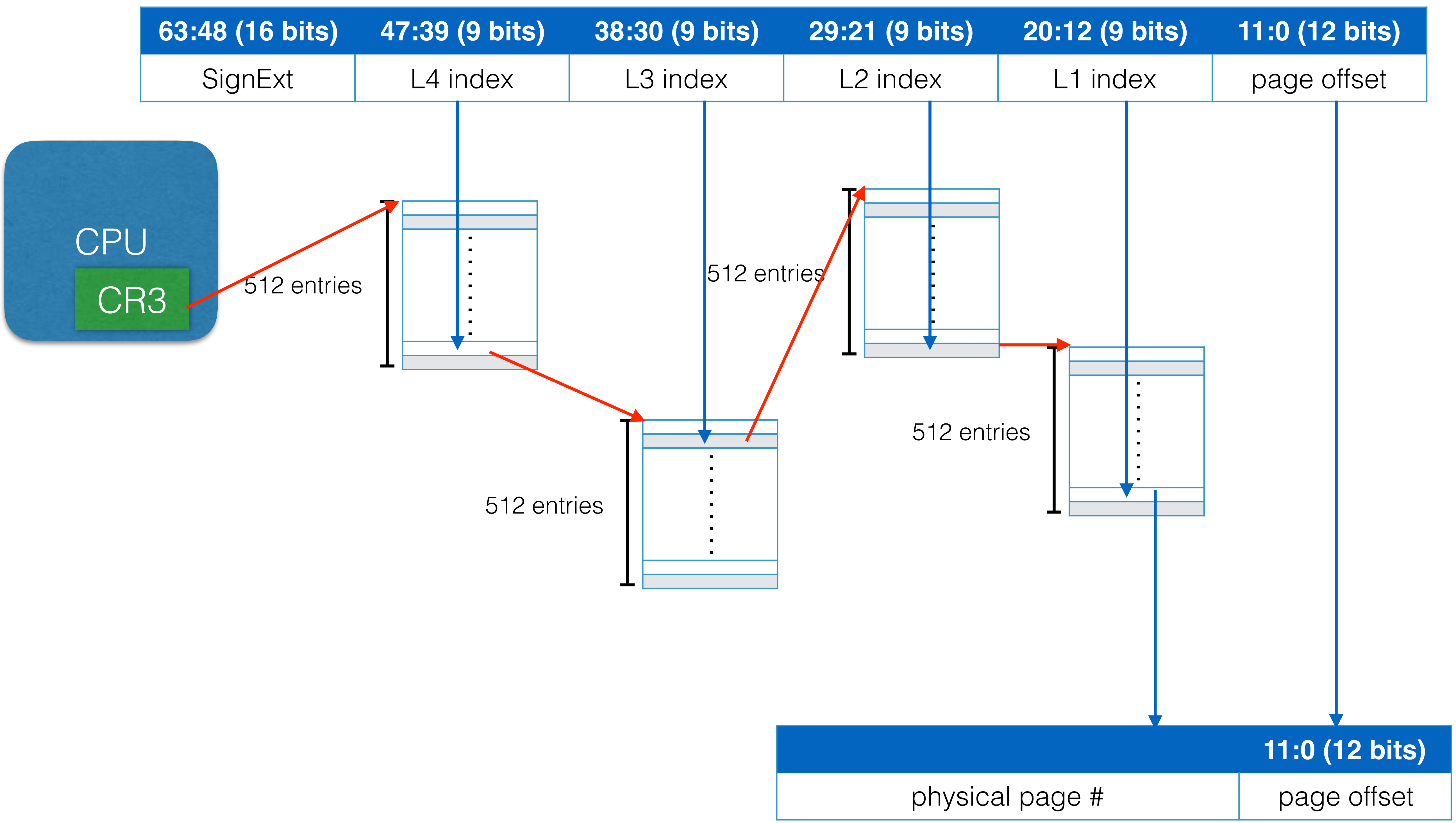
Hierarchical page table

- Only store the valid second level pages.

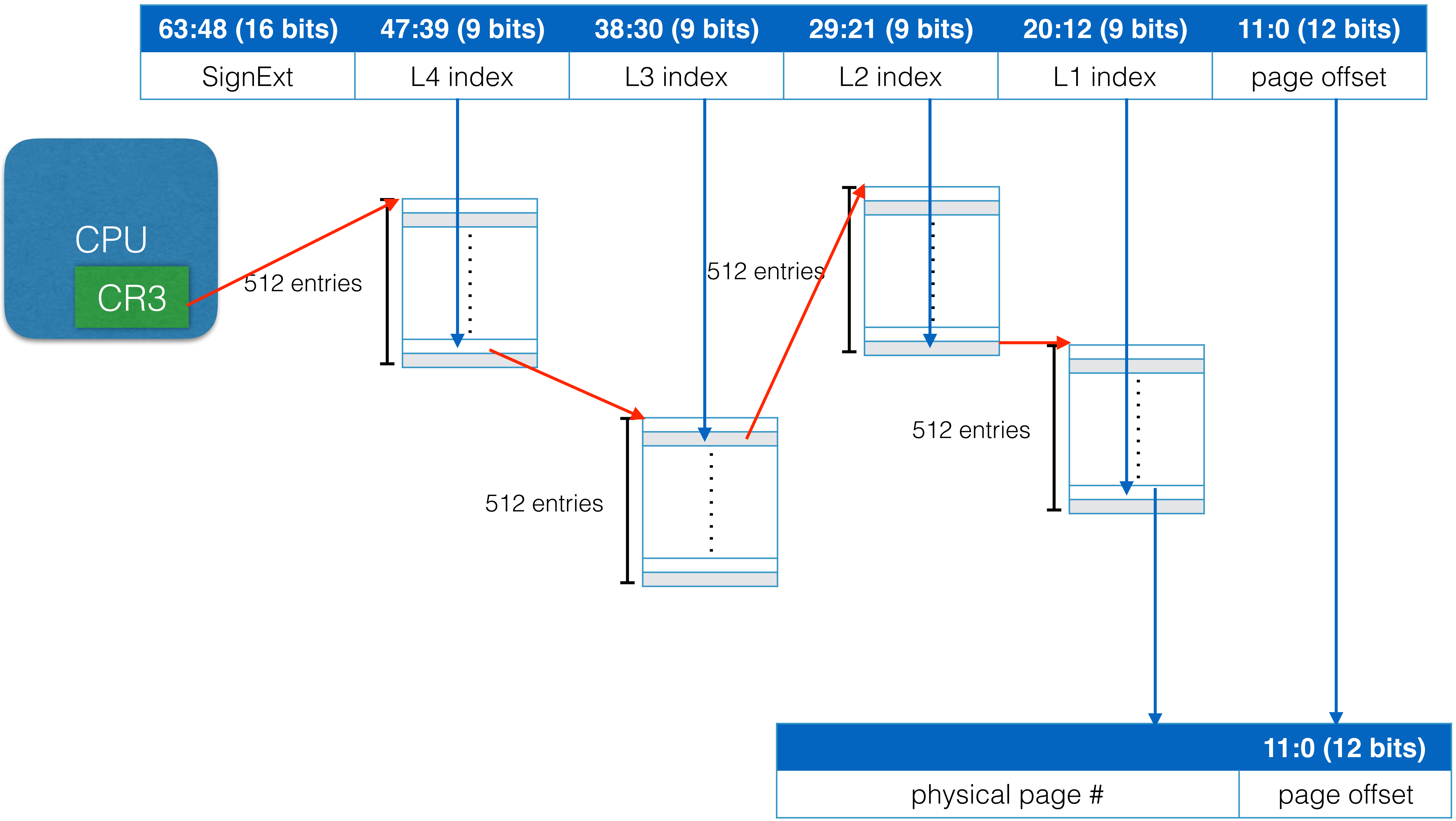


Virtual memory in practice

Address translation in x86-64

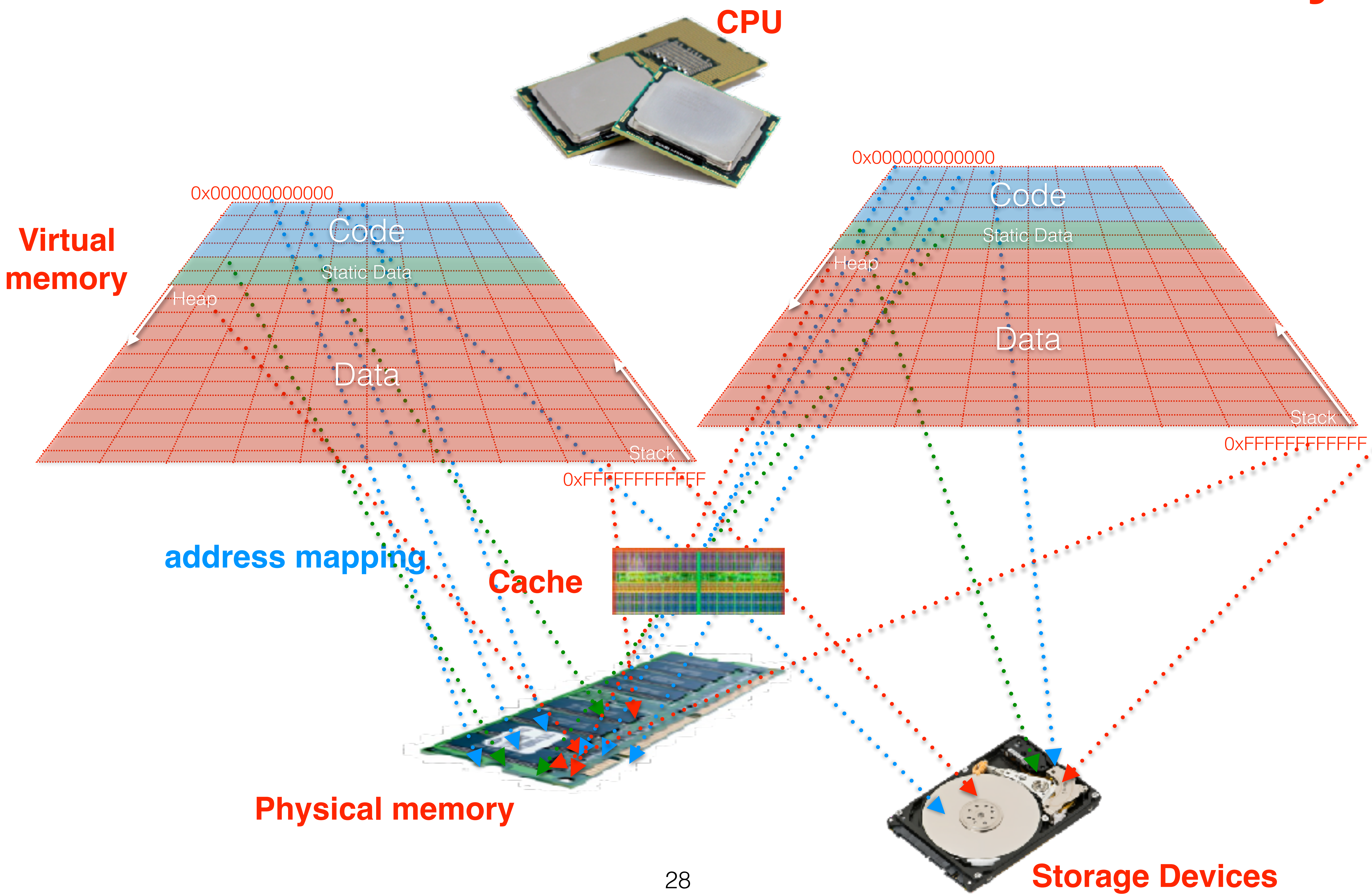


Address translation in x86-64

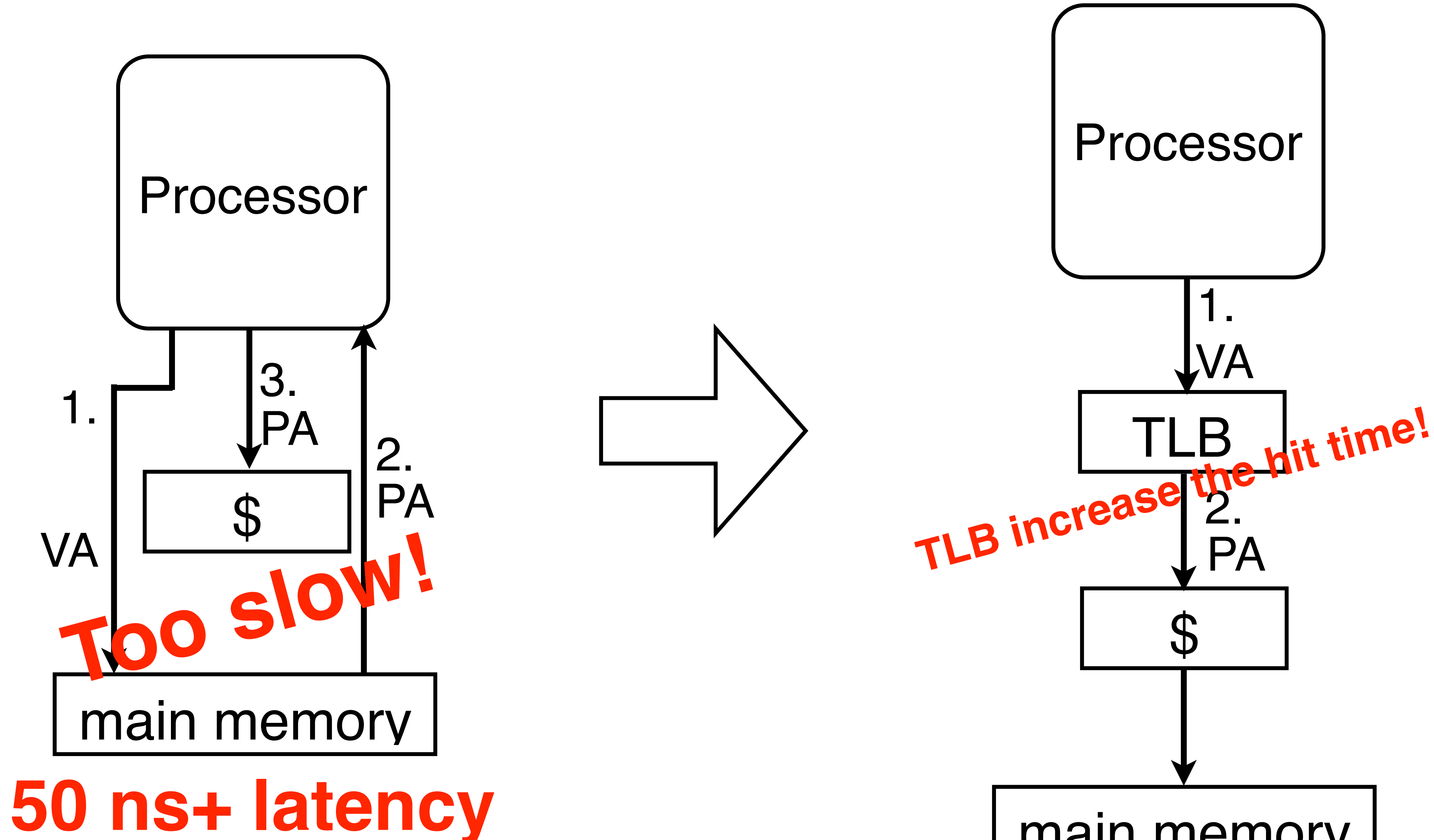


May have 10 memory accesses for a "MOV" instruction!

When cache meets virtual memory

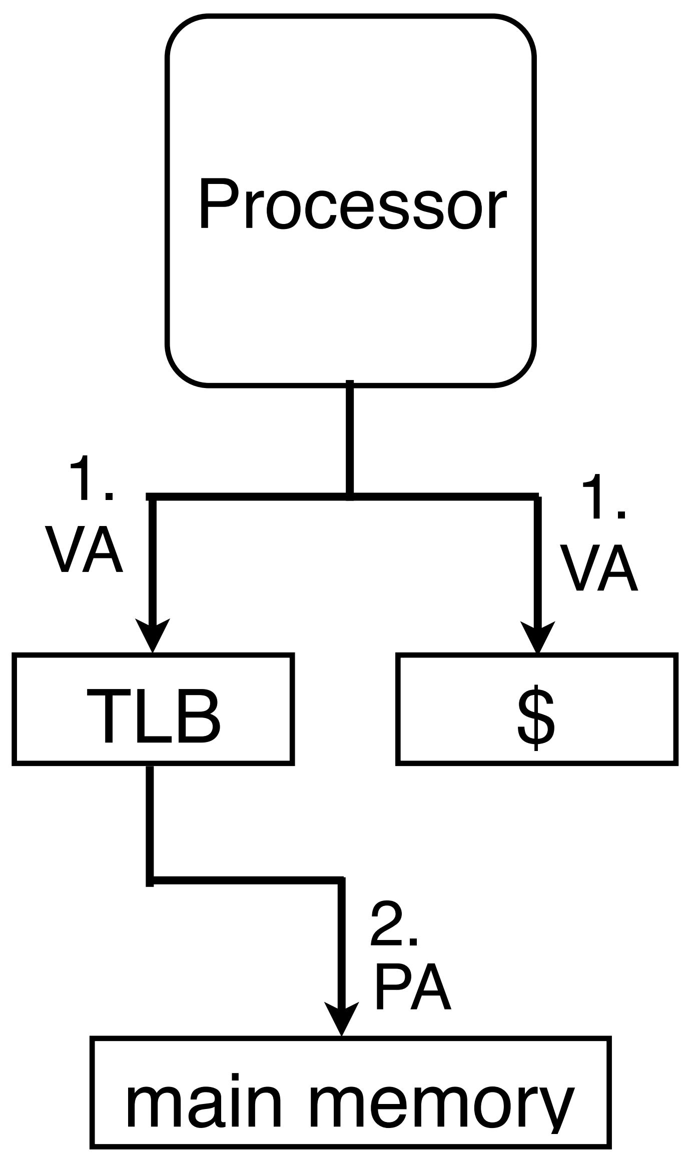


Cache + Virtual Memory



- TLB: Translation Look-aside Buffer
  - a cache of page table
    - small, high-associativity
    - miss penalty: access to page table in main memory

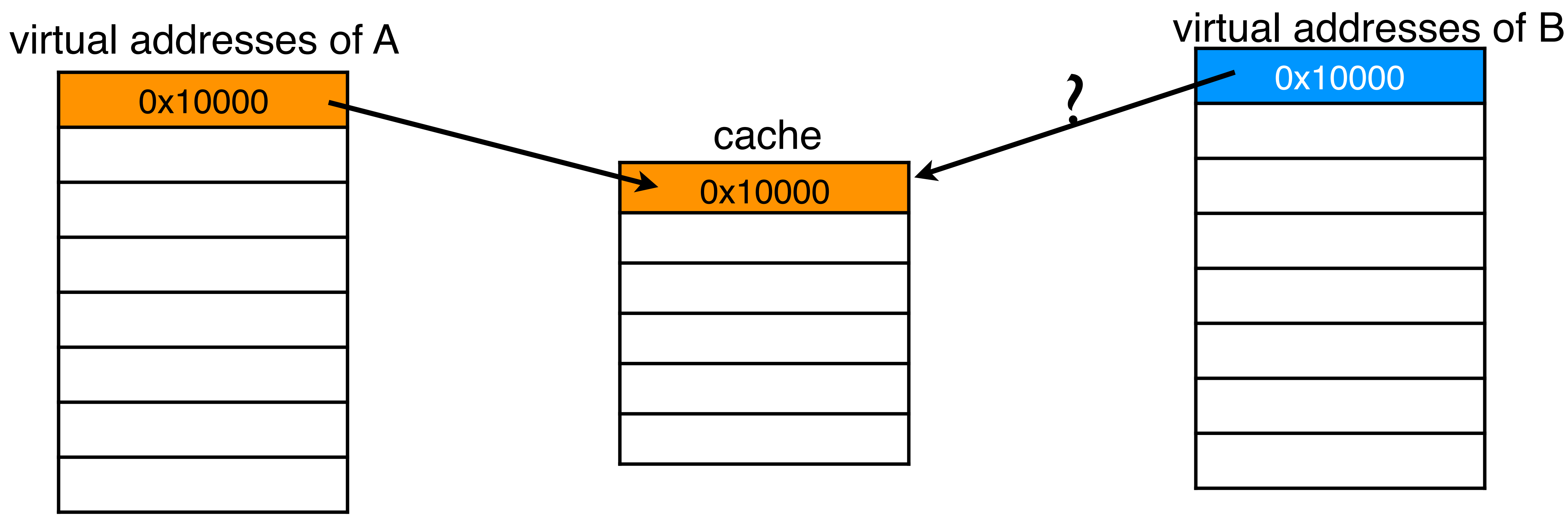
Cache+Virtual Memory



- Virtual Cache
  - The cache also uses virtual addresses
  - Address translation is required only when miss.

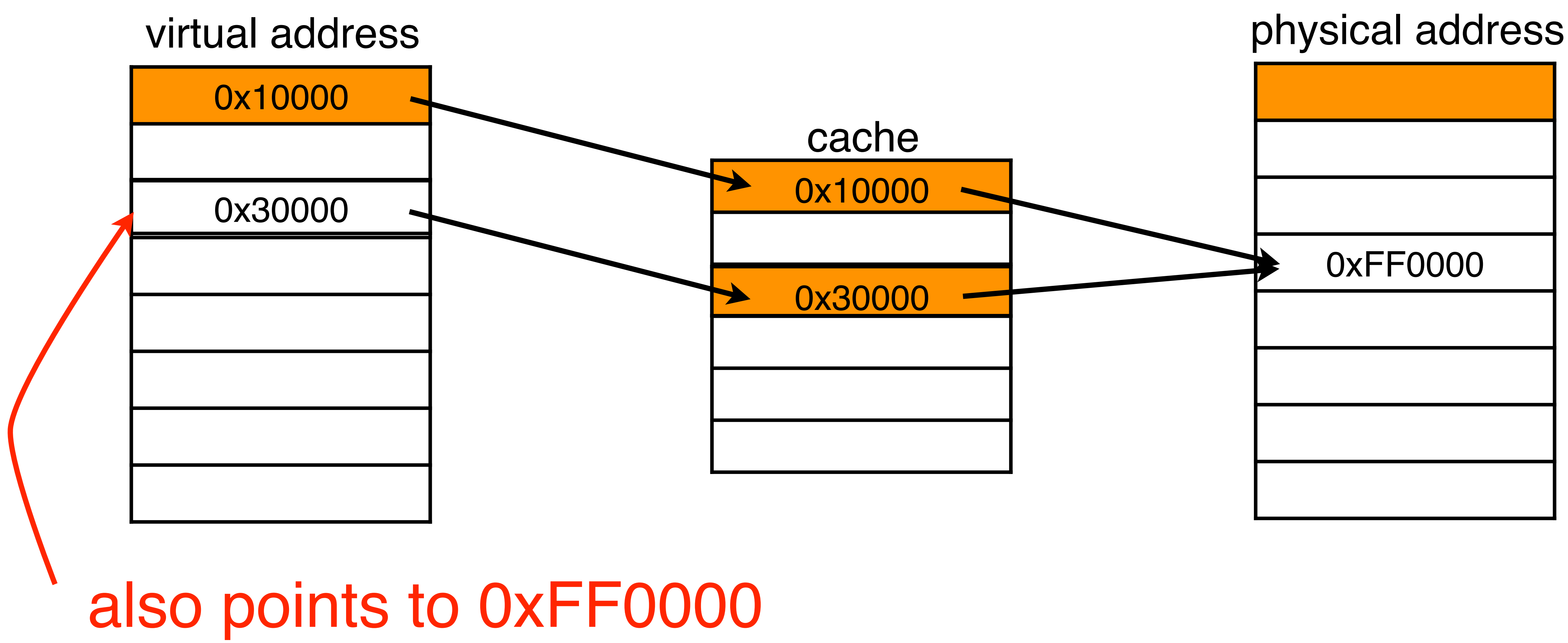
Problems of Virtual Cache

- Multiple processes accessing to the same virtual address — shm_open & mmap function
  - Process A accessed 0x10000. Process B also want to access 0x10000
    - Flush the cache when context switch **slowdown multiprogrammed systems**
    - Attach PID to cache **increase hardware costs**



Problems of Virtual Cache

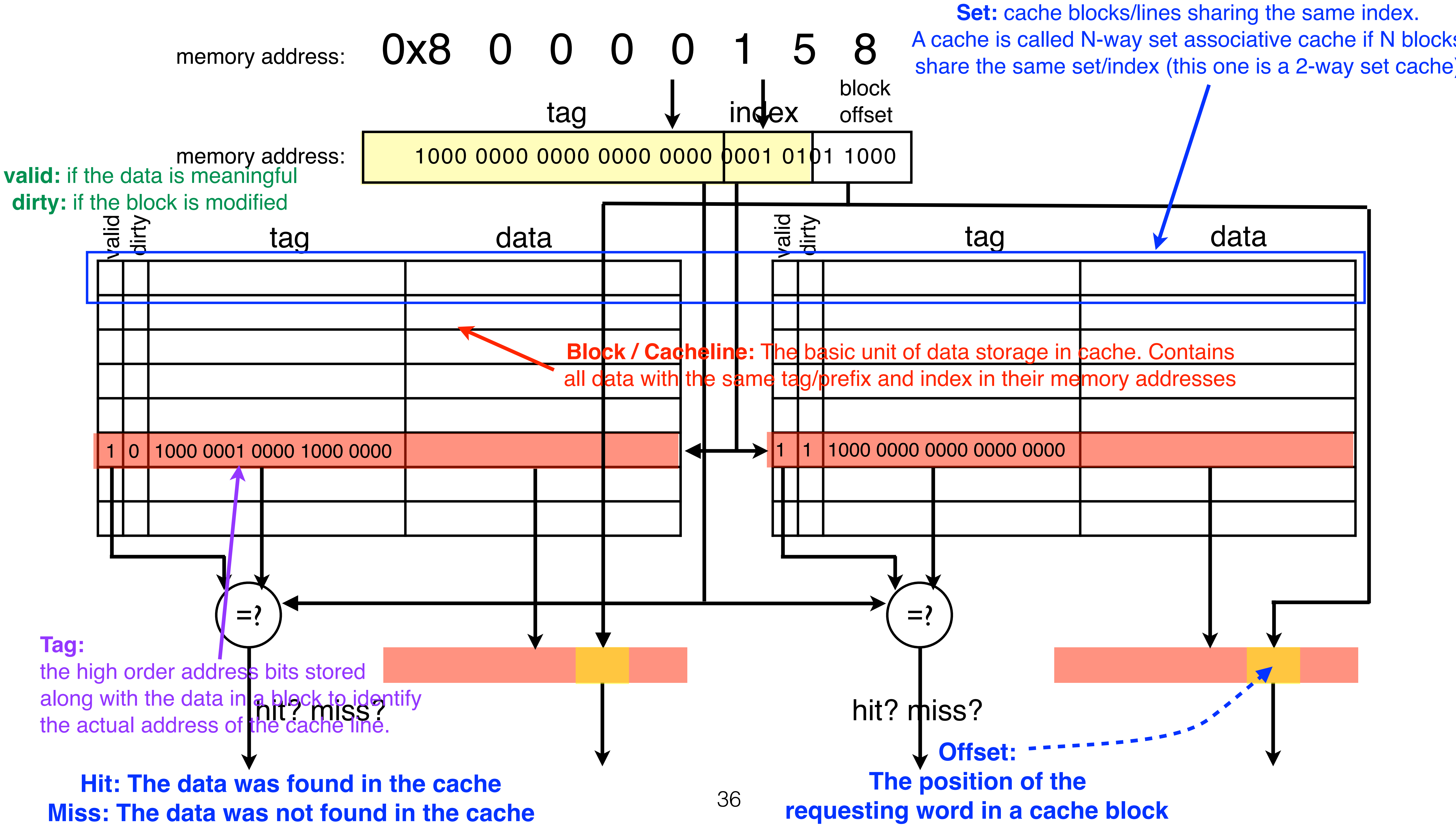
- Alias: A physical address maps to different virtual addresses
  - Two copies of data in cache due to copy on write. One may get the wrong data if the other is modified.



What do we need?

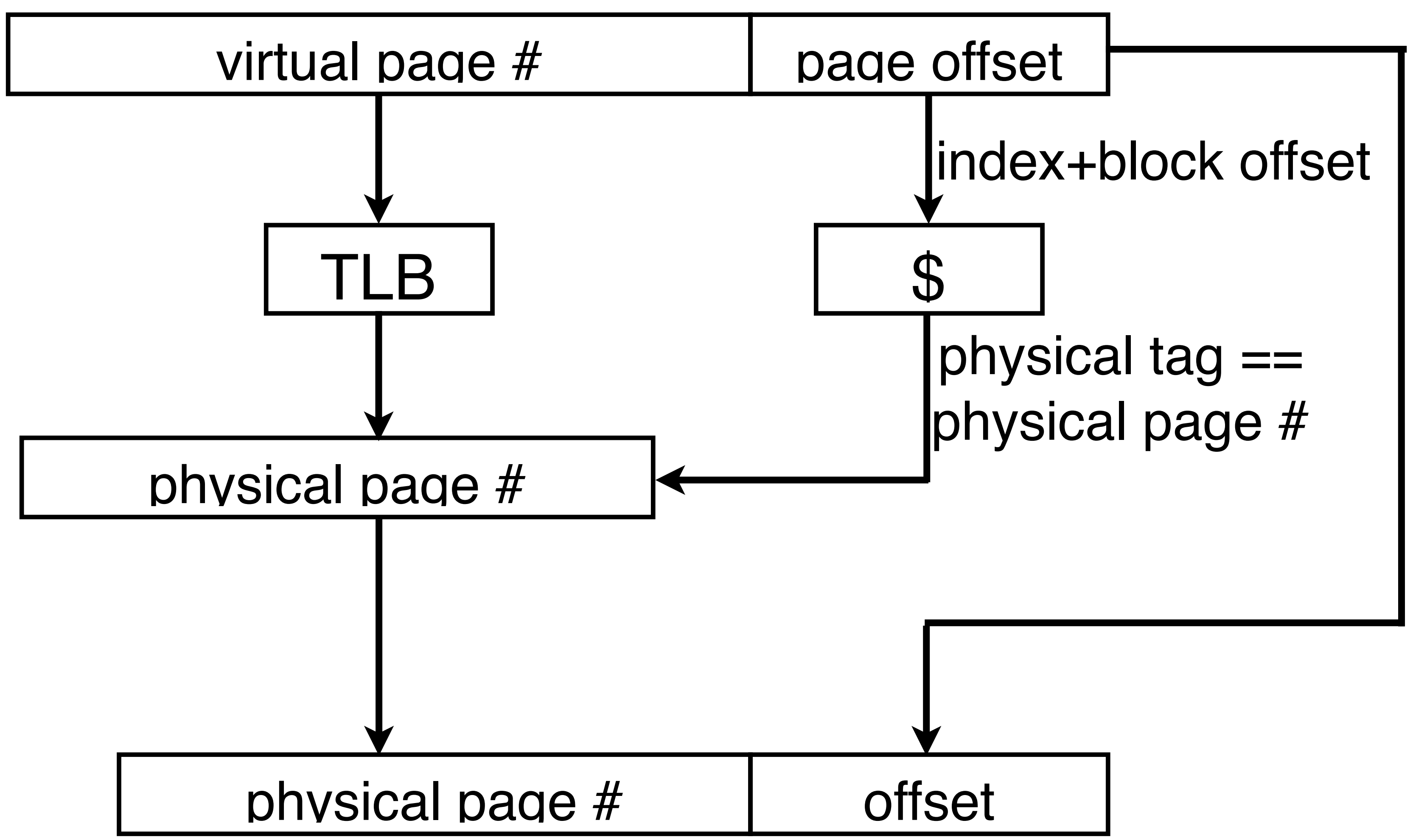
- TLB and cache can be accesses at the same time — the cache must accept virtual address
- No matter what virtual addresses are used, as long as they map to the same physical locations, the mapped cache block need to be at the same location in cache — cache can use your virtual memory address to reach exactly the same block in the cache using physical address
- The cache needs to store physical address as the tag to identify if they are the same data?

Accessing the cache



What's the solution?

- Force aliasing virtual addresses mapped to the same cache location.
- Cache stores tag fields of "physical addresses"
  - the physical tag is also the physical page number!



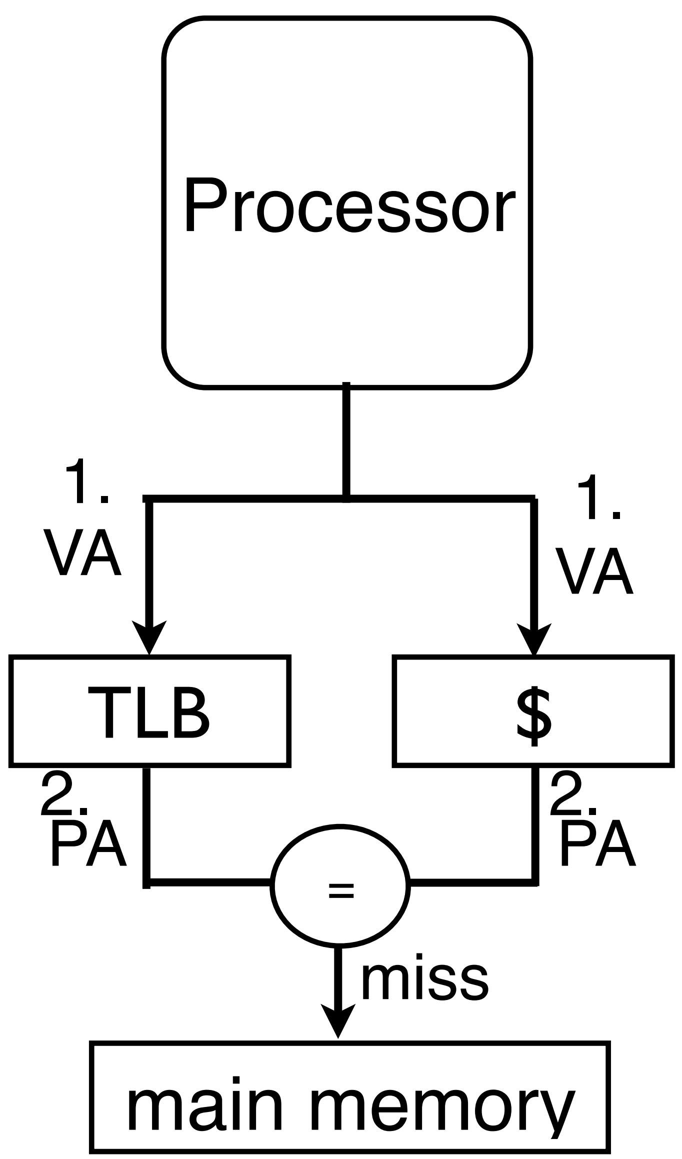
$C = ABS$

$lg(S)+lg(B) = 12$

if $A = 1$ (DM cache)

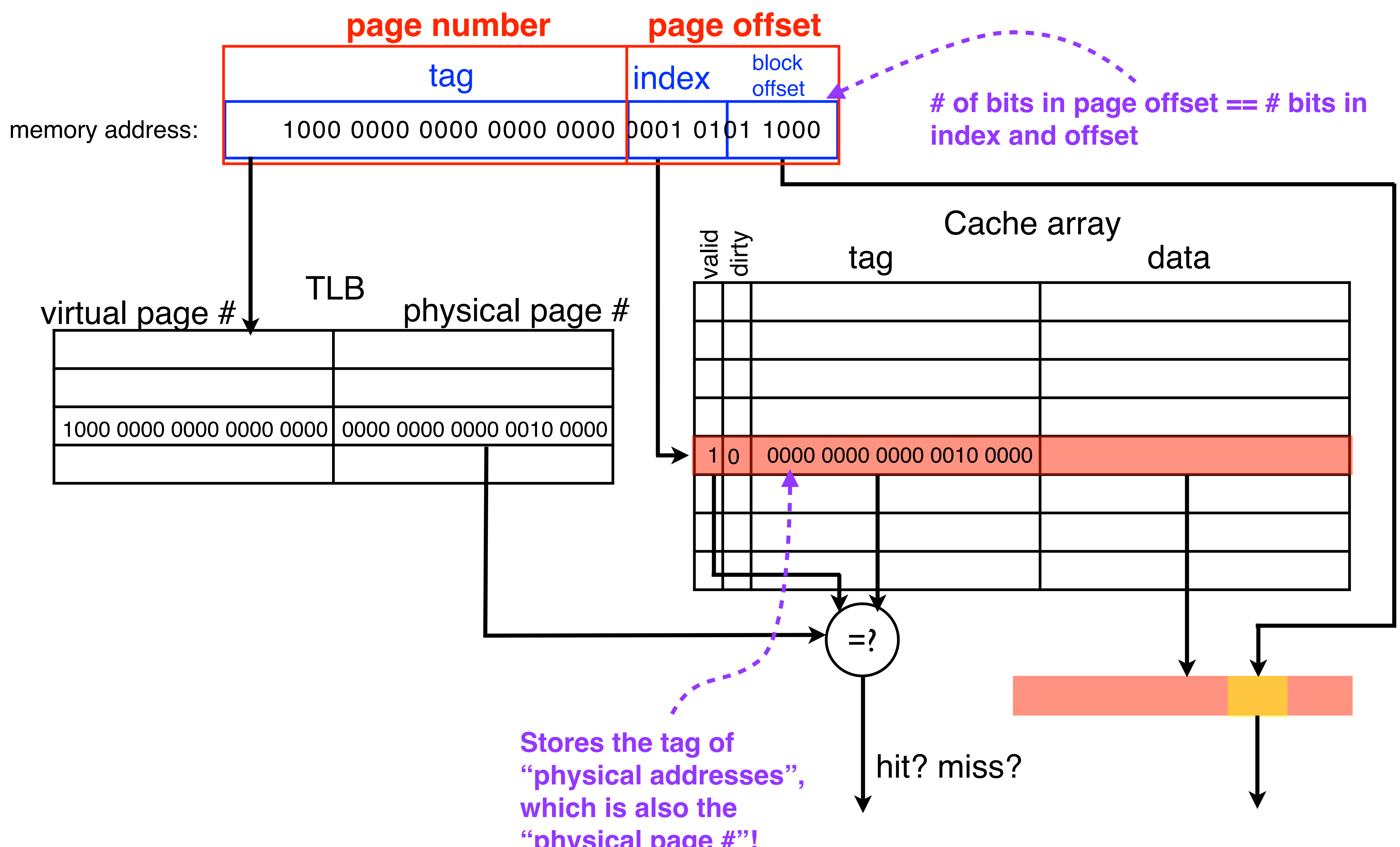
$C = 1*(2^{12}) = 4KB$

Virtually indexed, physically tagged cache



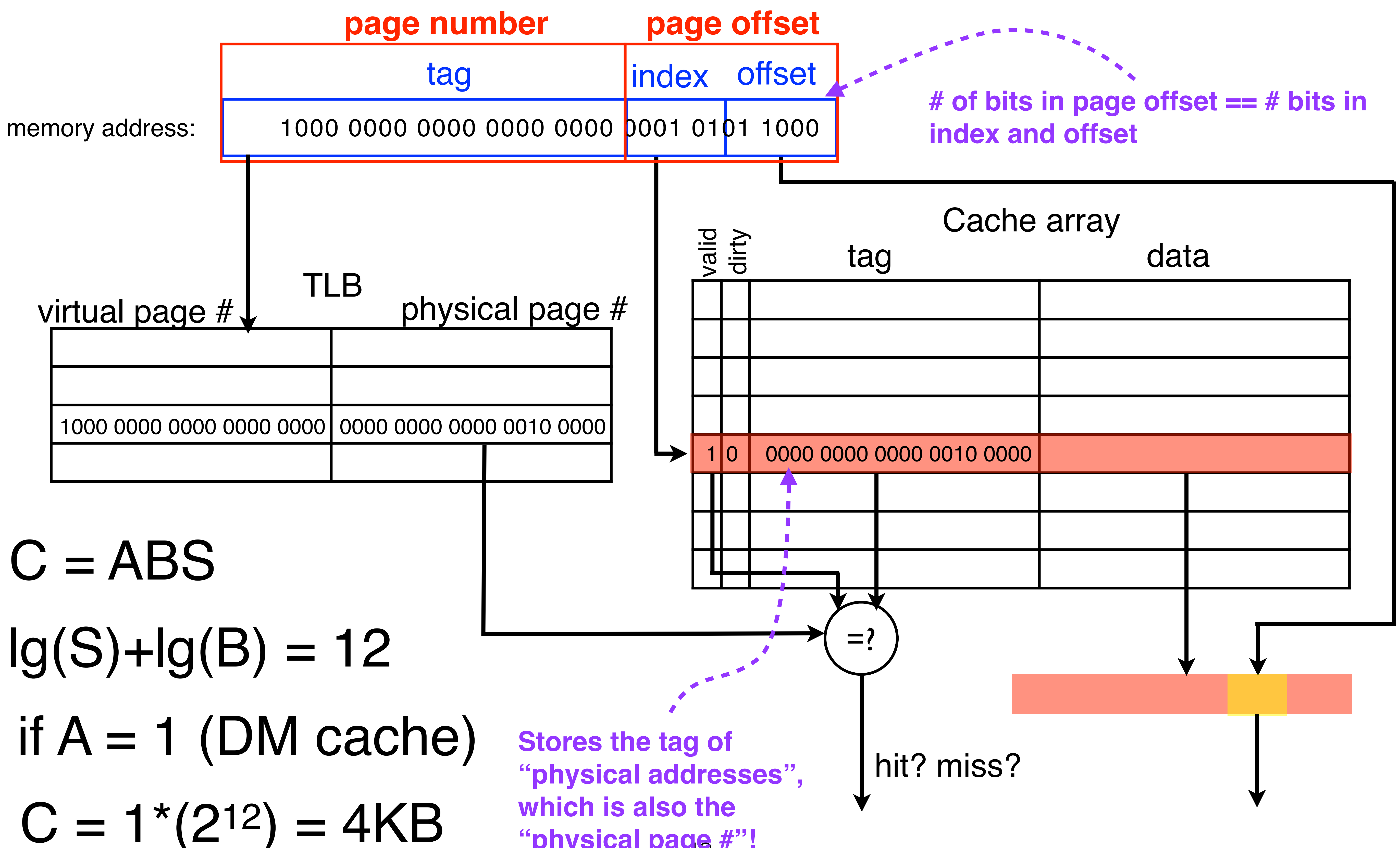
- Force aliasing virtual addresses mapped to the same cache location.
    - The cache uses the “index” field to place data blocks
    - Page offset remains the same in virtual and physical addresses
    - index field must be inside the page offset to guarantee that aliasing are mapped to the same place
- Cache stores tag fields of “physical addresses”

TLB + cache



page number
page offset
tag
index
block offset
memory address:
1000 0000 0000 0000 0000 0001 0101 1000
# of bits in page offset == # bits in index and offset
Cache array
valid
dirty
tag
data
TLB
virtual page #
physical page #
1000 0000 0000 0000 0000
0000 0000 0000 0010 0000
1
0
0000 0000 0000 0010 0000
=?
hit? miss?
Stores the tag of "physical addresses", which is also the "physical page #"!

Virtually indexed, physically tagged cache



$C = ABS$

$lg(S)+lg(B) = 12$

if $A = 1$ (DM cache)

$C = 1*(2^{12}) = 4KB$

Cache & Performance

- The processor runs@2GHz. 20% are L/S
  - L1 I-cache miss rate: 5%, hit time: 1 cycle
  - L1 D-cache miss rate: 10%, hit time: 1 cycle, 10% evicted blocks are dirty
  - L2 U-Cache miss rate: 20%, hit time: 10 cycles, 20% evicted blocks are dirty
  - L1 TLB miss rate: 1%, hit time < 1 cycle
    - 200 cycles penalty
  - Main memory hit time: 100 cycles
  - All caches are write-back, write-allocate

$CPI_{average}$=1+

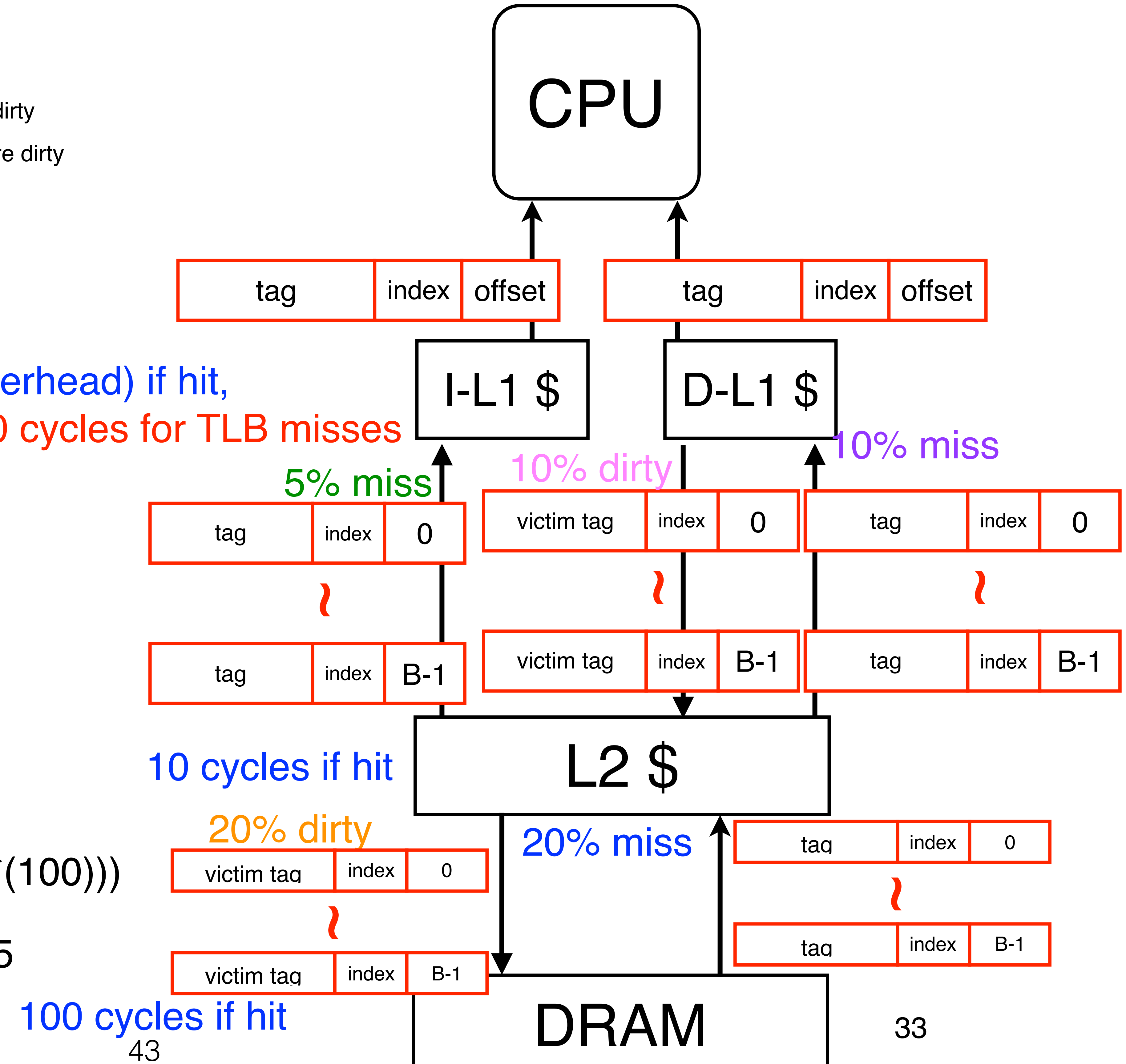
20%*(1%*200+10%*(1+10%)*(10+20%*(1+20%)*(100)))

+1%*200+1*(5%*(10+20%*(1+20%)*(100))) =5.85

Put it all together

- The processor runs@2GHz. 20% are L/S
  - L1 I-cache miss rate: 5%, hit time: 1 cycle
  - L1 D-cache miss rate: 10%, hit time: 1 cycle, 10% evicted blocks are dirty
  - L2 U-Cache miss rate: 20%, hit time: 10 cycles, 20% evicted blocks are dirty
  - L1 TLB miss rate: 1%, hit time < 1 cycle
    - 200 cycles penalty
  - Main memory hit time: 100 cycles
  - All caches are write-back, write-allocate

1 cycle (no overhead) if hit,
1% needs 200 cycles for TLB misses



$CPI_{average} = 1+$

$20\%*(1\%*200+10\%*(1+10\%)*(10+20\%*(1+20\%)*(100)))$

$+1\%*200+1*(5\%*(10+20\%*(1+20\%)*(100))) = 5.85$