Virtual Memory

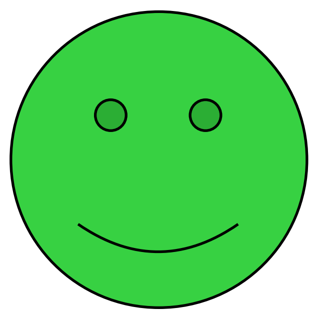
Today

- Quiz 8
- Virtual memory
- Micro project 3

Key Points

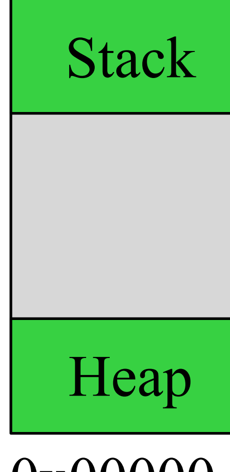
- What is virtual memory?
- Why is it useful?
- What is address translation?
- What is a page?
- What are virtual and physical addresses?
- What is page table and how can we store and access it efficiently?
- What is a TLB?
- What are possible organizations of the TLB and L1 cache? What are their advantages and disadvantages?
- What is an cache alias?

Learning to Play Well With Others



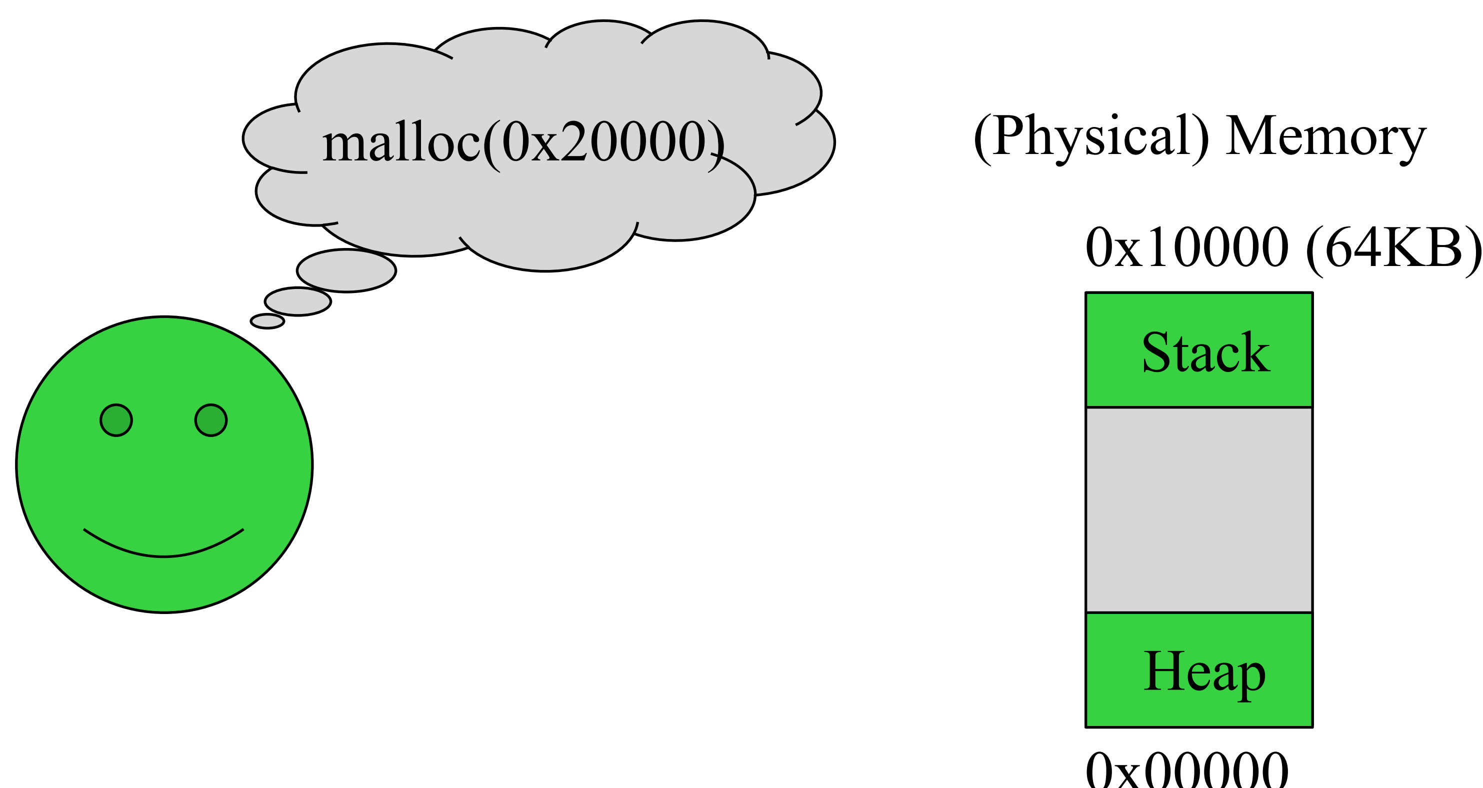
(Physical) Memory

0x10000 (64KB)

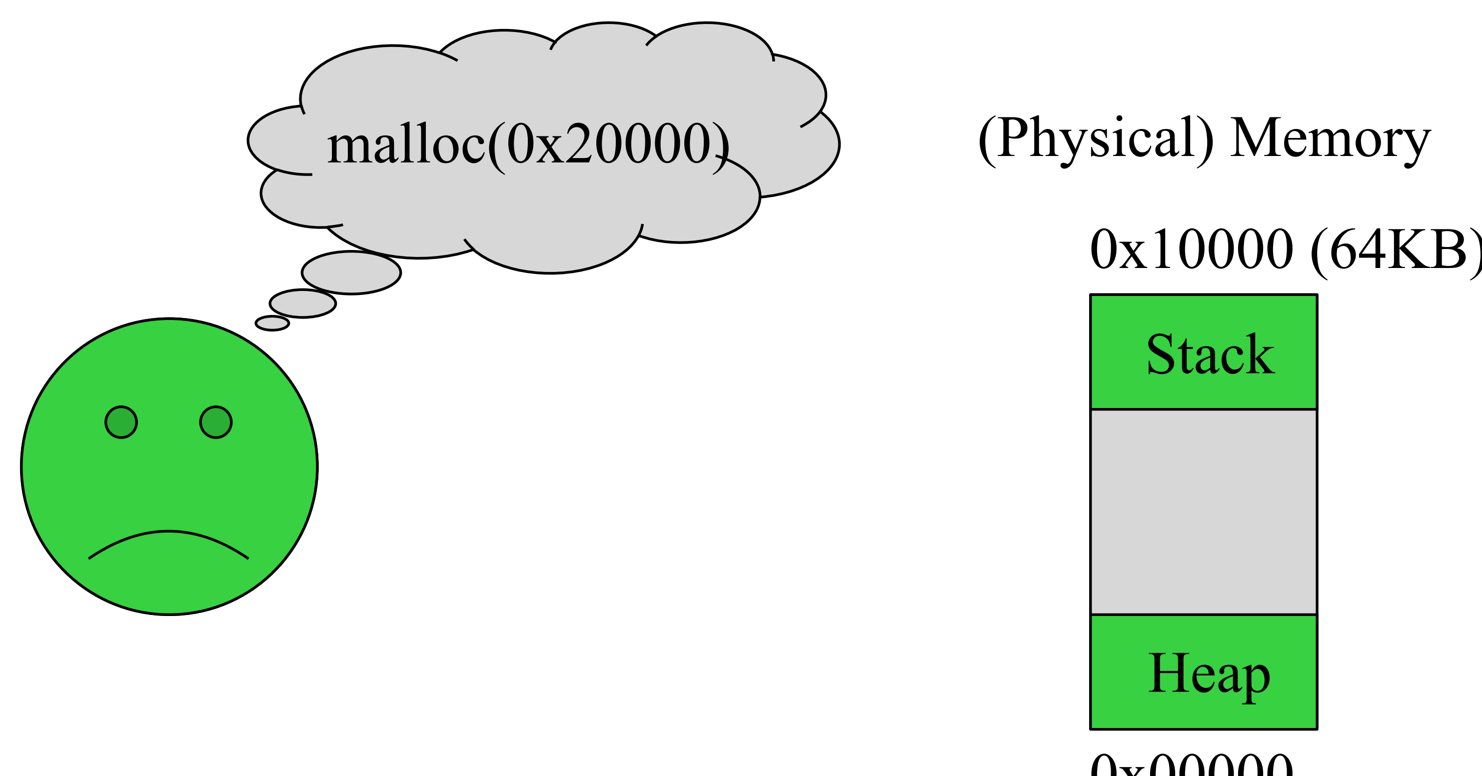


0x00000

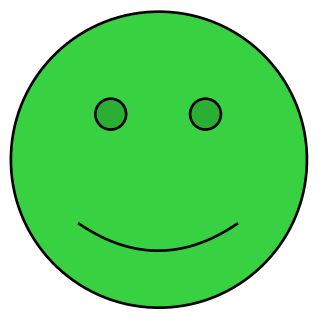
Learning to Play Well With Others



Learning to Play Well With Others

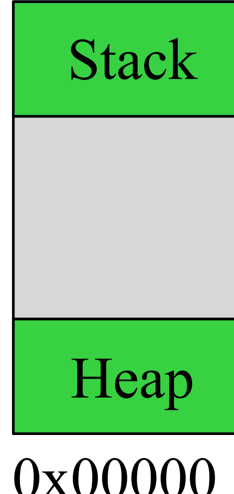


Learning to Play Well With Others



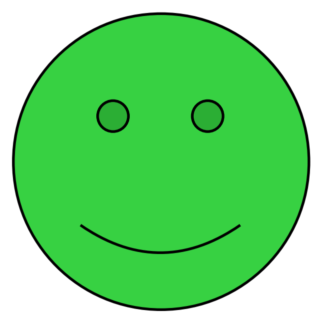
(Physical) Memory

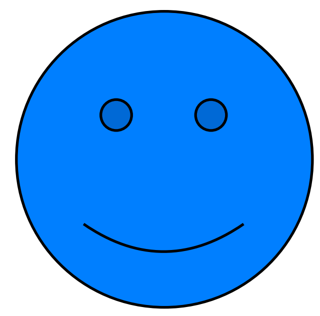
0x10000 (64KB)



0x00000

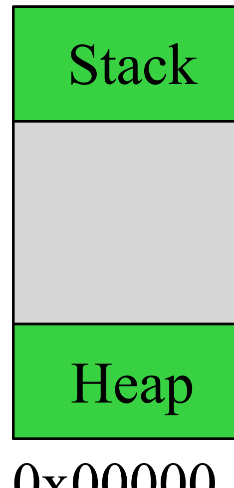
Learning to Play Well With Others





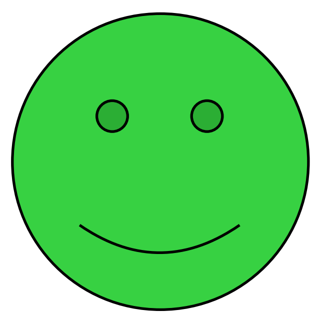
(Physical) Memory

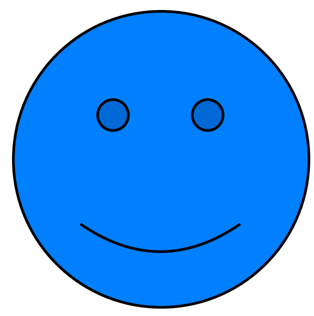
0x10000 (64KB)



0x00000

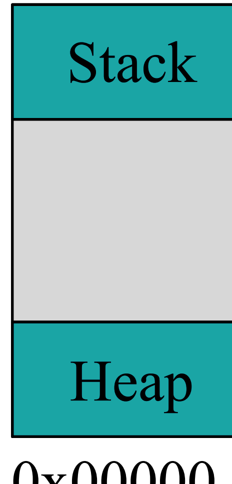
Learning to Play Well With Others





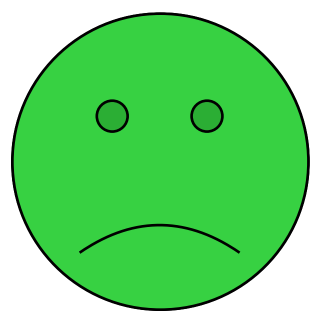
(Physical) Memory

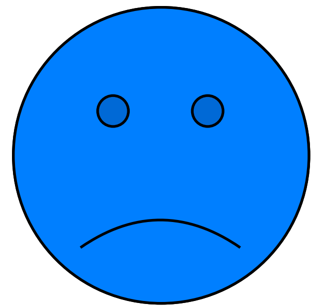
0x10000 (64KB)



0x00000

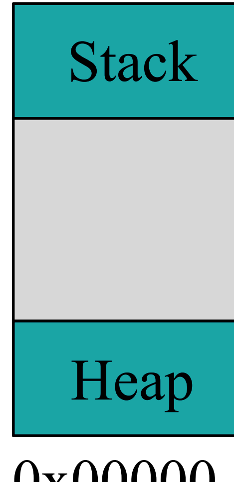
Learning to Play Well With Others





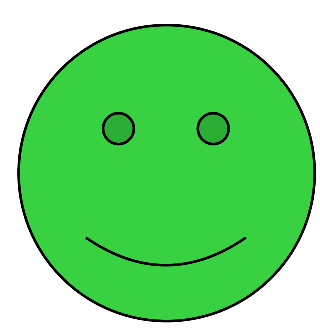
(Physical) Memory

0x10000 (64KB)



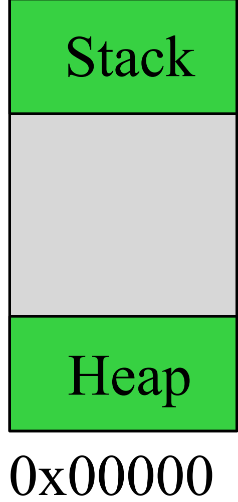
0x00000

Learning to Play Well With Others



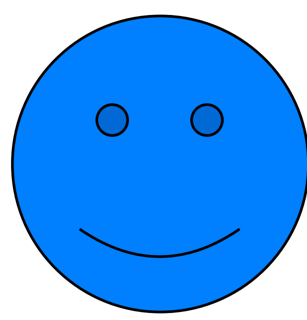
Virtual Memory

0x10000 (64KB)



Virtual Memory

0x10000 (64KB)

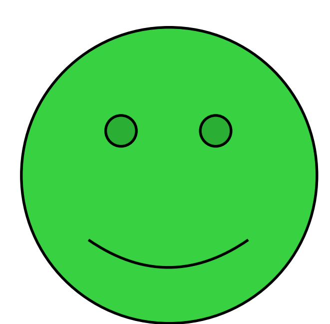


Stack

Heap

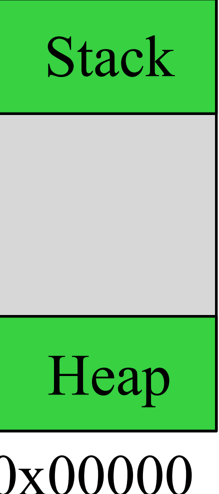
0x00000

Learning to Play Well With Others



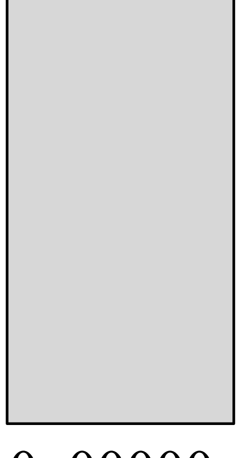
Virtual Memory

0x10000 (64KB)



Physical Memory

0x10000 (64KB)



0x00000

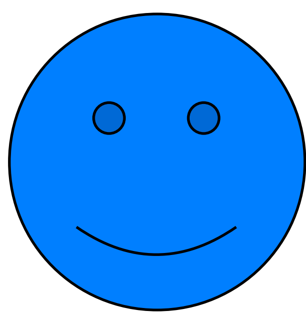
Virtual Memory

0x10000 (64KB)

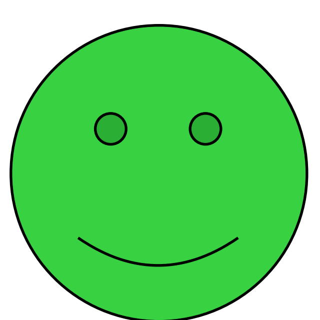
Stack

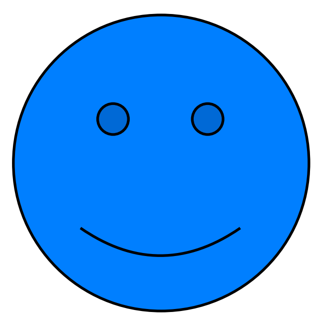
Heap

0x00000



Learning to Play Well With Others





Virtual Memory

0x10000 (64KB)

Stack

Heap

0x00000

Virtual Memory

0x10000 (64KB)

Stack

Heap

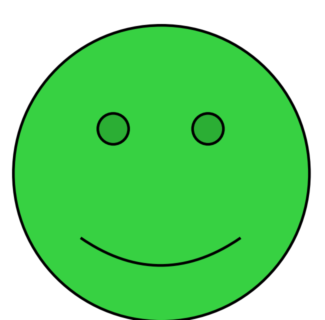
0x00000

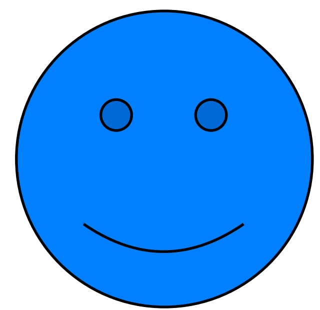
Physical Memory

0x10000 (64KB)

0x00000

Learning to Play Well With Others





Virtual Memory

0x400000 (4MB)

Stack

Heap

0x00000

Virtual Memory

0xF000000 (240MB)

Stack

Heap

0x00000

Physical Memory

0x10000 (64KB)

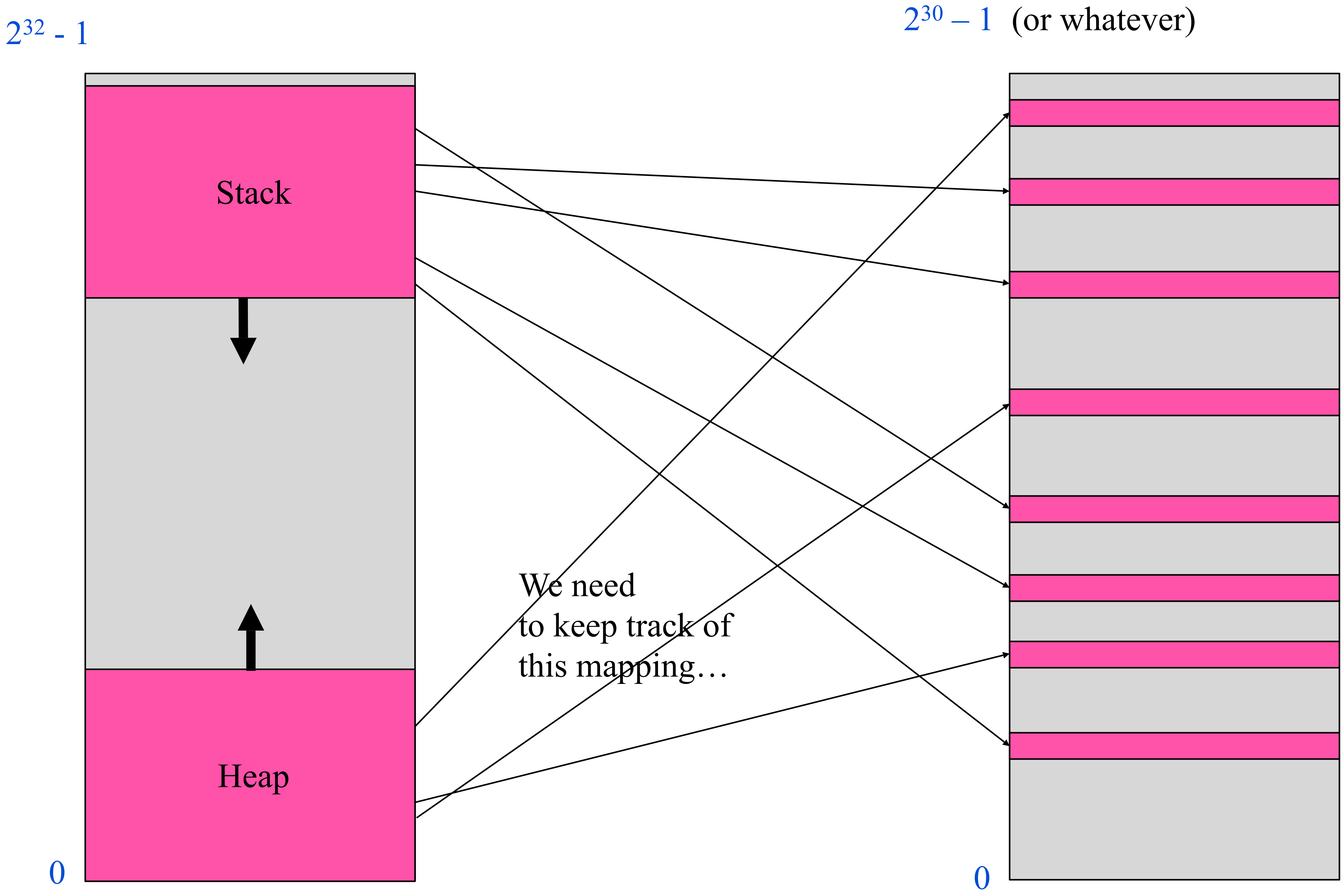
0x00000

Disk
(GBs)

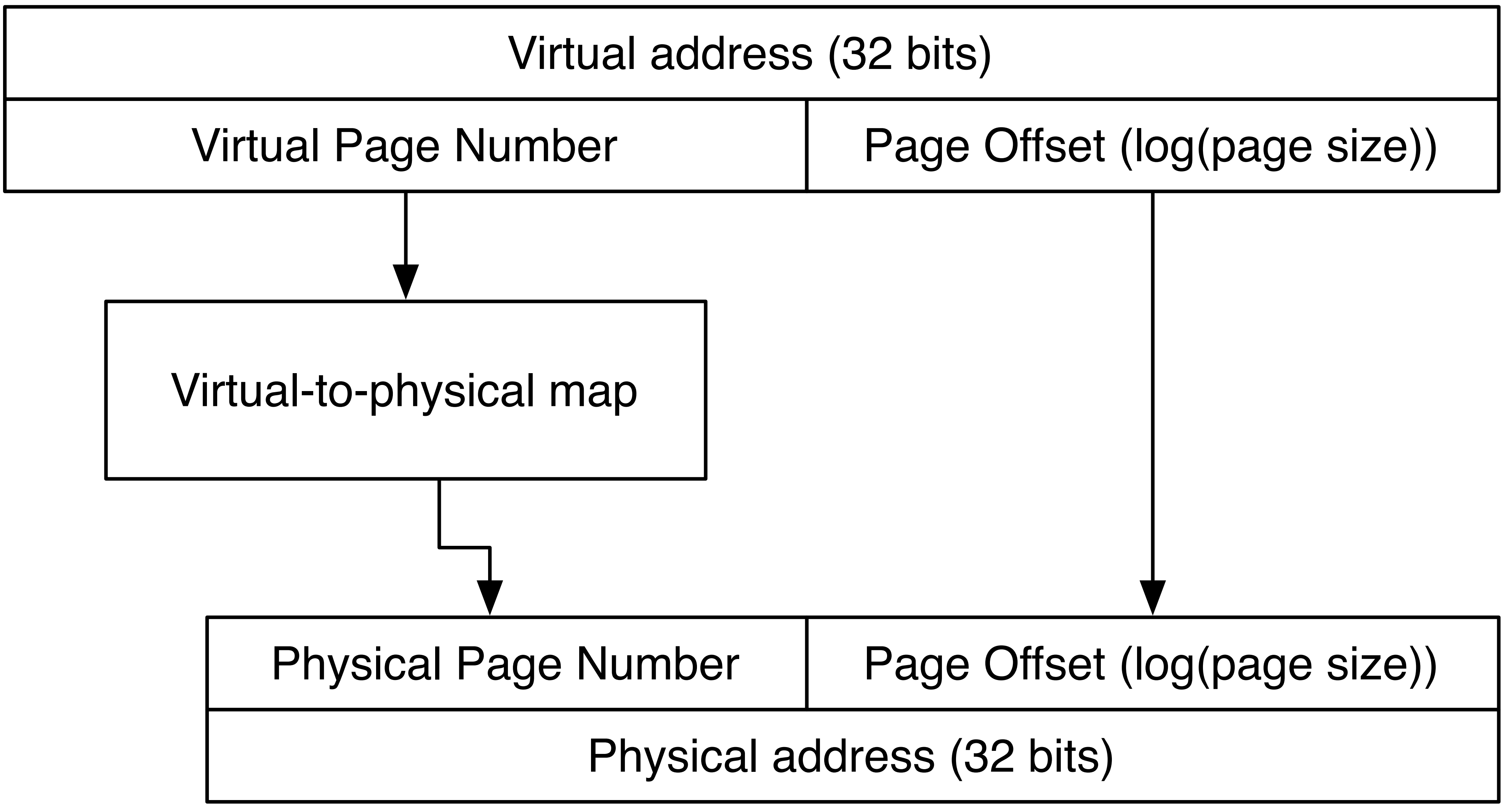
Mapping

- Virtual-to-physical mapping
  - Virtual --> "virtual address space"
  - physical --> "physical address space"
- We will break both address spaces up into "pages"
  - Typically 4KB in size, although sometimes larger
- Use a "page table" to map between virtual pages and physical pages.
- The processor generates "virtual" addresses
  - They are translated via "address translation" into physical addresses.

Implementing Virtual Memory



The Mapping Process



Two Problems With VM

- How do we store the map compactly?
- How do we translation quickly?

How Big is the map?

- 32 bit address space:
  - 4GB of virtual addresses
  - 1million Pages
  - Each entry is 4 bytes (a 32 bit physical address)
  - 4MB of map
- 64 bit address space
  - 16 exabytes of virtual address space
  - 4PetaPages
  - Entry is 8 bytes
  - 64PB of map

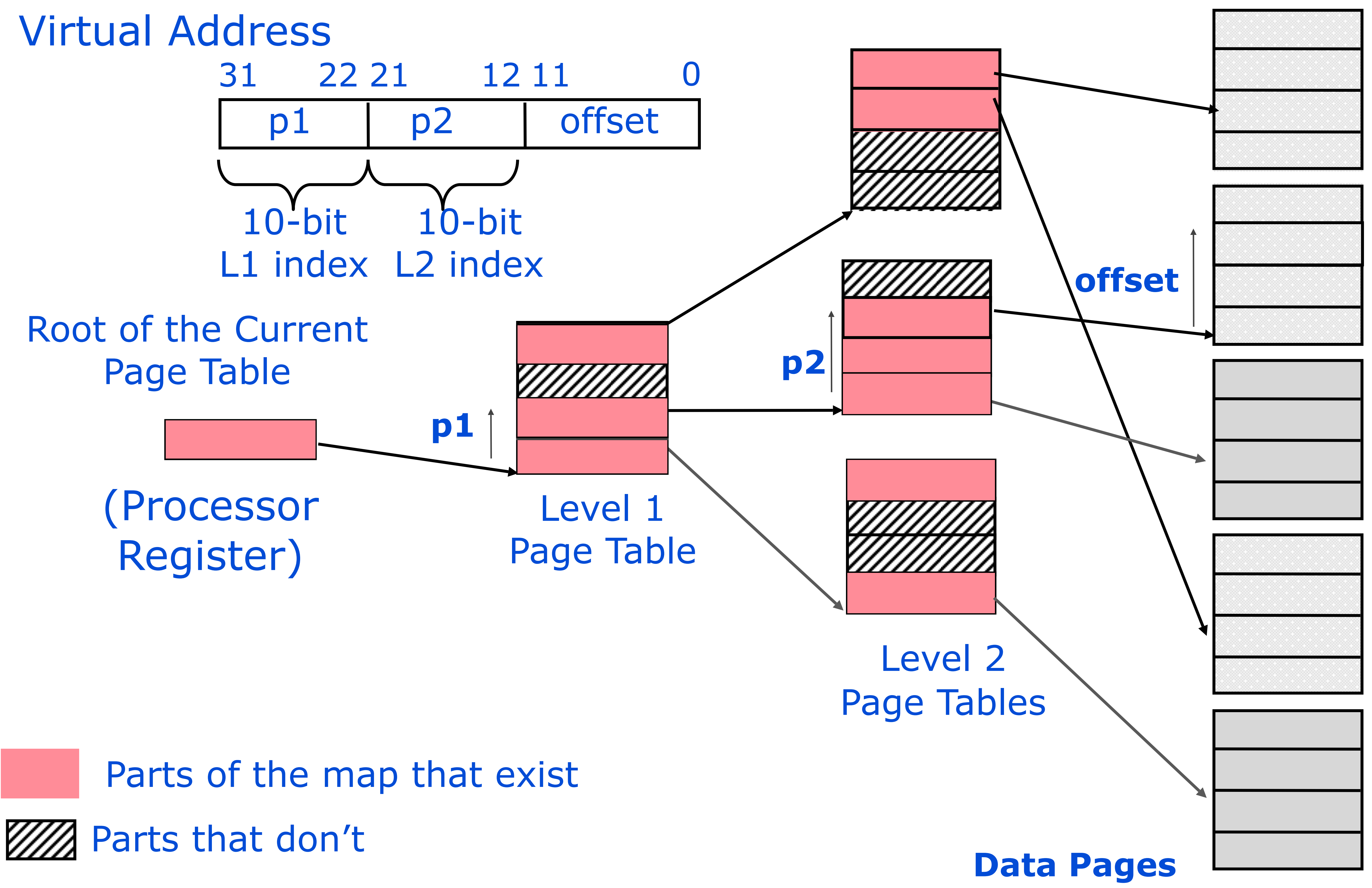
Shrinking the map

- Only store the entries that matter (i.e., enough for your physical address space)
- 64GB on a 64bit machine
  - 16M pages, 128MB of map
- This is still pretty big.
- Representing the map is now hard
  - The OS allocates stuff all over the place in the virtual address space.
  - For security, convenience, or caching optimizations
- How do you represent this “sparse” map?

Hierarchical Page Tables

- Break the virtual page number into several pieces
- If each piece has N bits, build an 2^N-ary tree
- Only store the part of the tree that contain valid pages
- To do translation, walk down the tree using the pieces to select with child to visit.

Hierarchical Page Table



Making Translation Fast

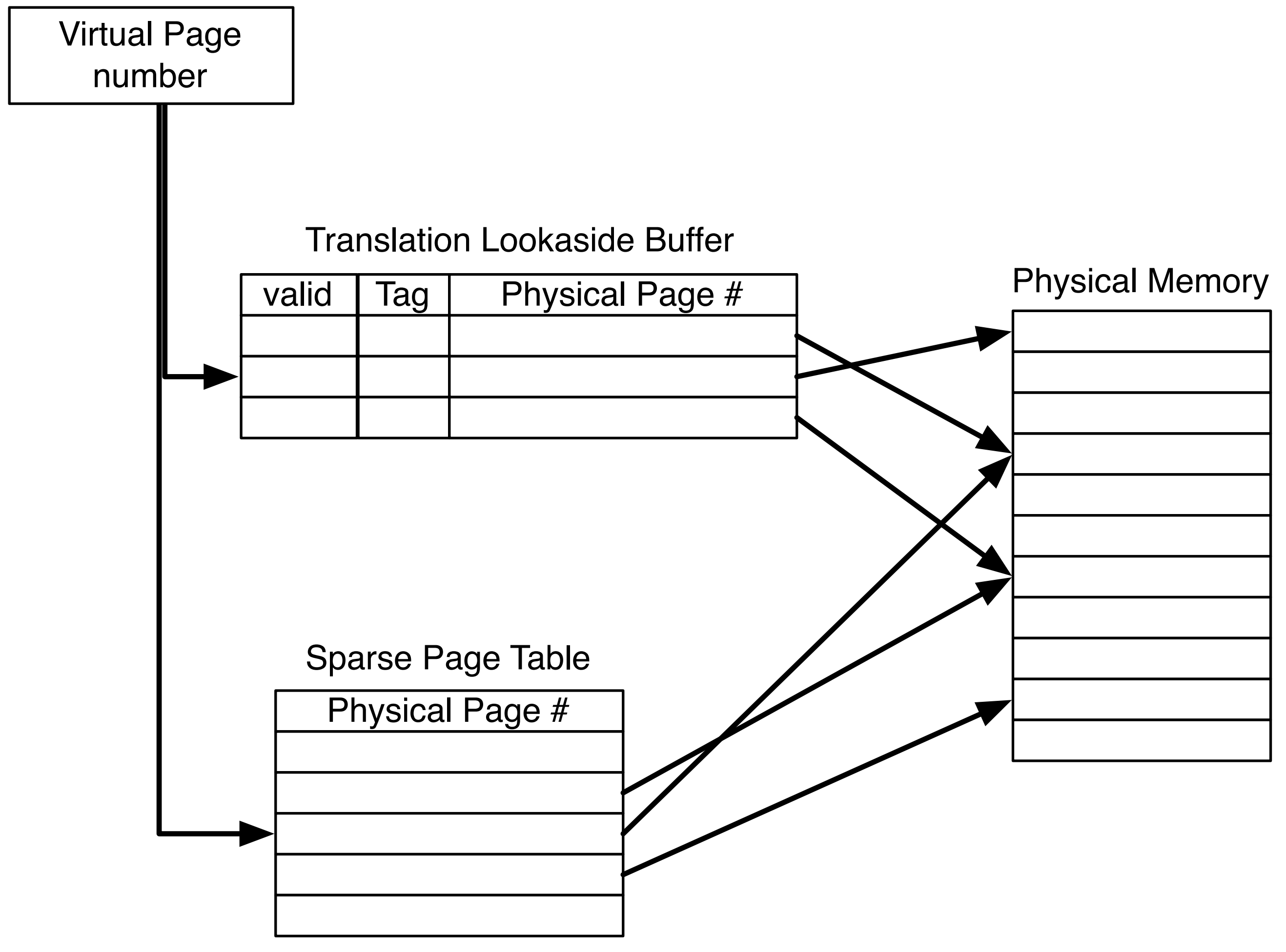
- Address translation has to happen for *every* memory access
- This potentially puts it squarely on the critical path for memory operation (which are already slow)

“Solution 1”: Use the Page Table

- We could walk the page table on every memory access
- Result: every load or store requires an additional 3-4 loads to walk the page table.
  - Does the page table have good locality?
  - What kind?
- Unacceptable performance hit.

Solution 2: TLBs

- We have a large pile of data (i.e., the page table) and we want to access it very quickly (i.e., in one clock cycle)
- So, build a cache for the page mapping, but call it a "translation lookaside buffer" or TLB

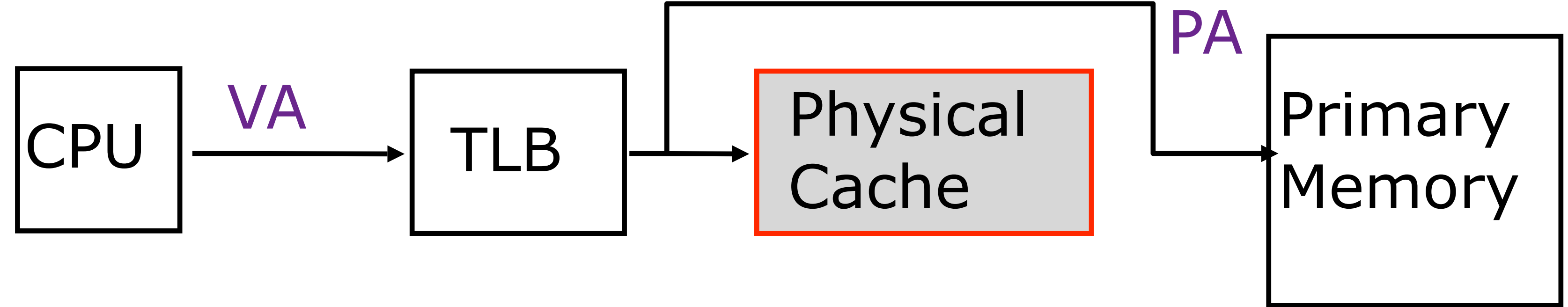


TLBs

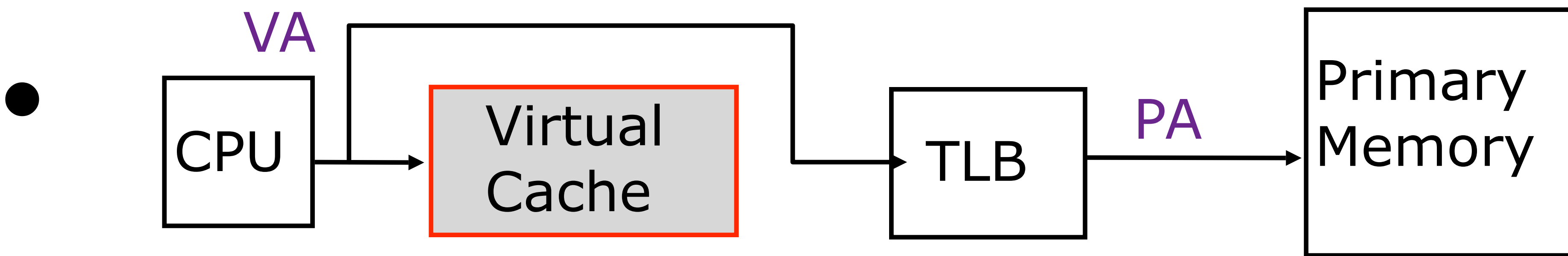
- TLBs are small (usually 128 entries or smaller), highly-associative (often fully-associative) caches for page table entries.
- This raises the possibility of a TLB miss
  - On a TLB Miss, the CPU must fetch the page table entry and update the TLB before proceeding with the memory access.
- TLB misses are expensive
  - To make them cheaper, there are “hardware page table walkers” -- specialized state machines that can load page table entries into the TLB without OS intervention
  - This means that the page table format is now part of the big-A architecture.
  - Typically, the OS can disable the walker and implement its own format in TLB miss exception handler.

Solution 3: Defer translating Accesses

- If we translate before we go to the cache, we have a "physical cache", since cache works on physical addresses.
  - Critical path = TLB access time + Cache access time



- Alternately, we could translate after the cache
  - Translation is only required on a miss.
  - This is a "virtual cache"
  -



The Danger Of Virtual Caches (1)

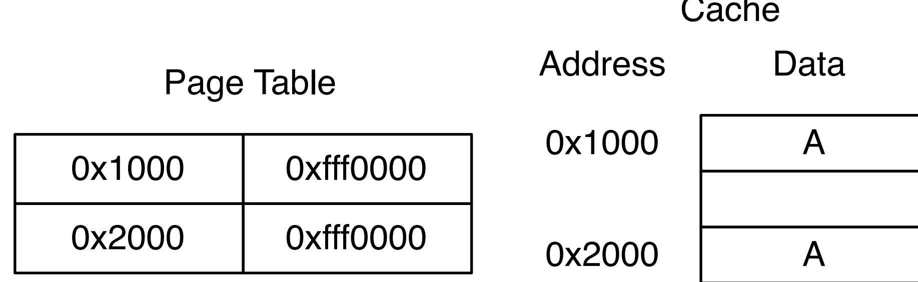
- Process A is running. It issues a memory request to address 0x10000
  - It is a miss, and is brought into the virtual cache
- A context switch occurs
- Process B starts running. It issues a request to 0x10000
- Will B get the right data?
-

The Danger Of Virtual Caches (1)

- Process A is running. It issues a memory request to address 0x10000
  - It is a miss, and is brought into the virtual cache
- A context switch occurs
- Process B starts running. It issues a request to 0x10000
- Will B get the right data?
- No! We must flush virtual caches on a context switch.

The Danger Of Virtual Caches (2)

- There is no rule that says that each virtual address maps to a different physical address.
- When this occurs, it is called "aliasing"
- Example: An alias exists in the cache



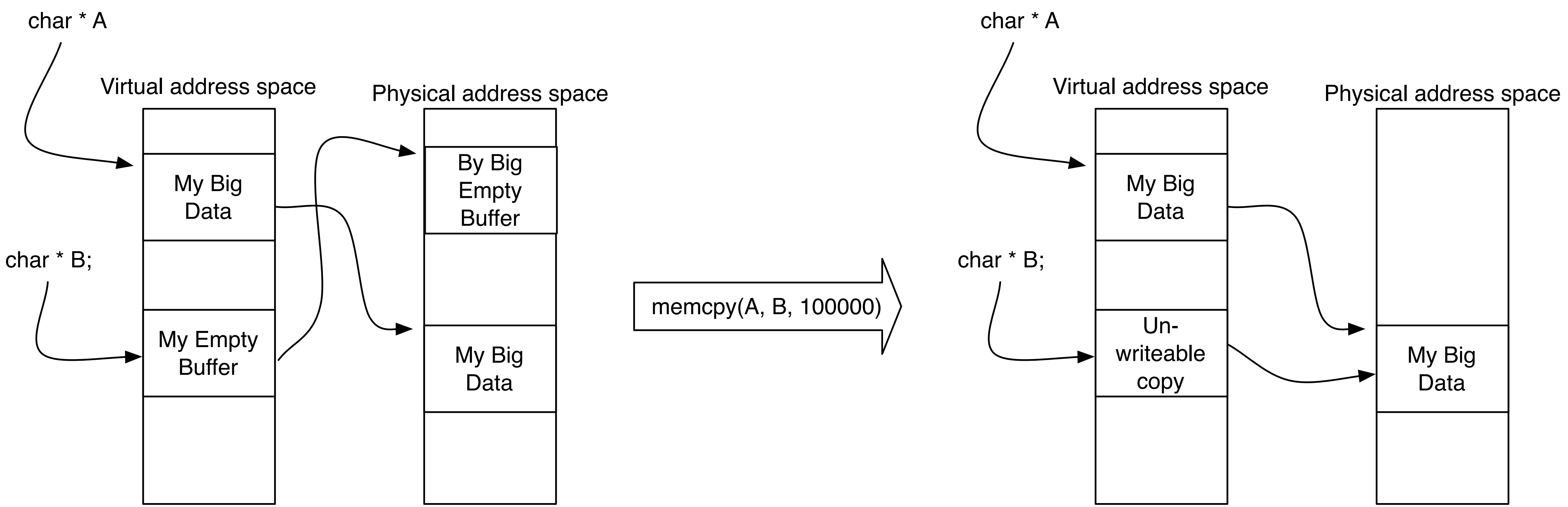
- Store B to 0x1000

Cache
Page Table
Address
Data
0x1000
0xfff0000
0x2000
0xfff0000
0x1000
B
0x2000
A

- Now, a load from 0x2000 will return the wrong value

Tricks With Virtual Memory

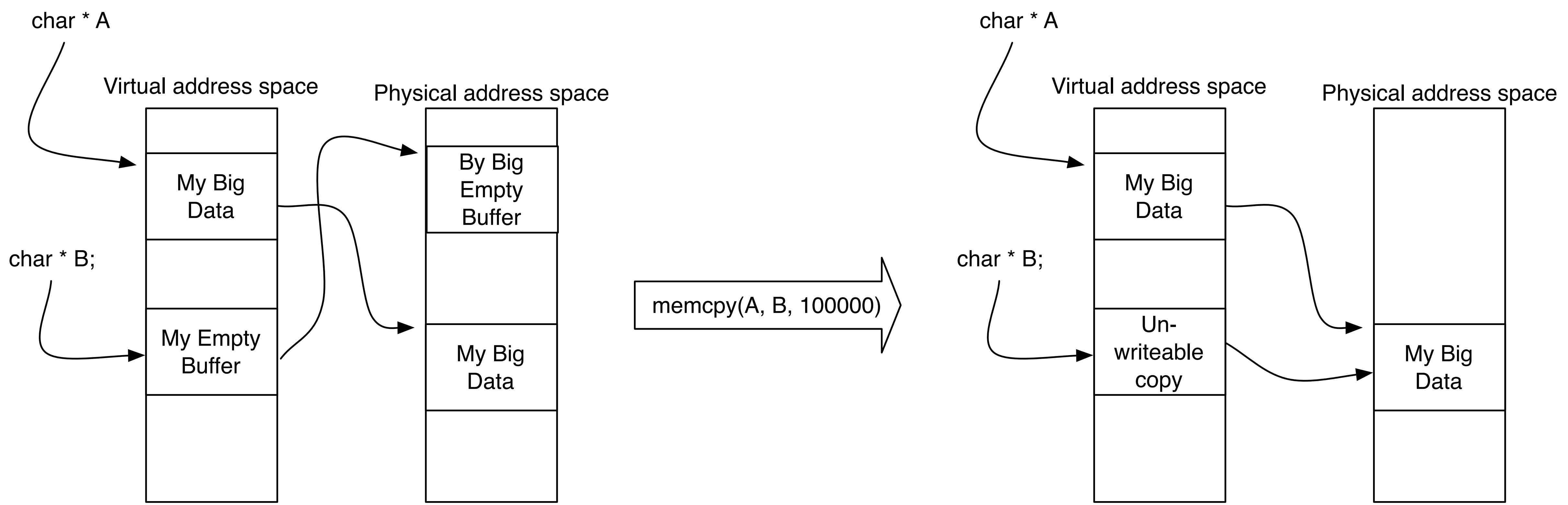
- There is no rule that says that each virtual address maps to a different physical address.
- Copy on write:



- The initial copy is free, and the OS will catch attempts to write to the copy, and do the actual copy lazily.
- There are also system calls that let you do this arbitrarily.

Tricks With Virtual Memory

- There is no rule that says that each virtual address maps to a different physical address.
- Copy on write:



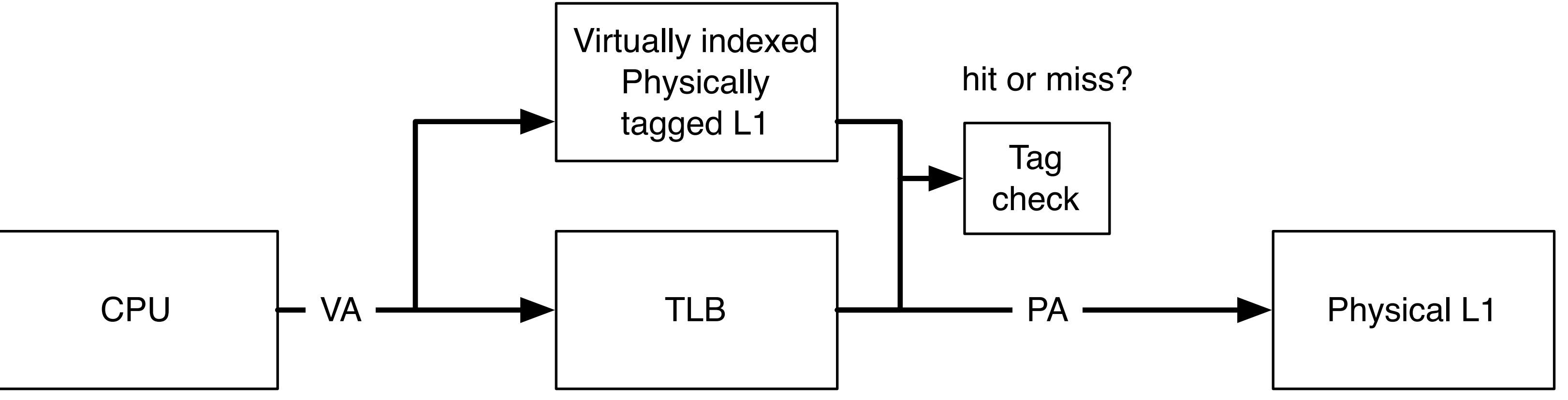
- The initial copy is free, and the OS will catch attempts to write to the copy, and do the actual copy lazily.
- There are also system calls that let you do this arbitrarily.

Avoiding Aliases

- If the system has virtual caches, the operating system must prevent alias from occurring.
- This means that any addresses that may alias must map to the same cache index.
  - If VA1 and VA2 are aliases,
  - VA1 mod (cache size) == VA2 mod (cache size)

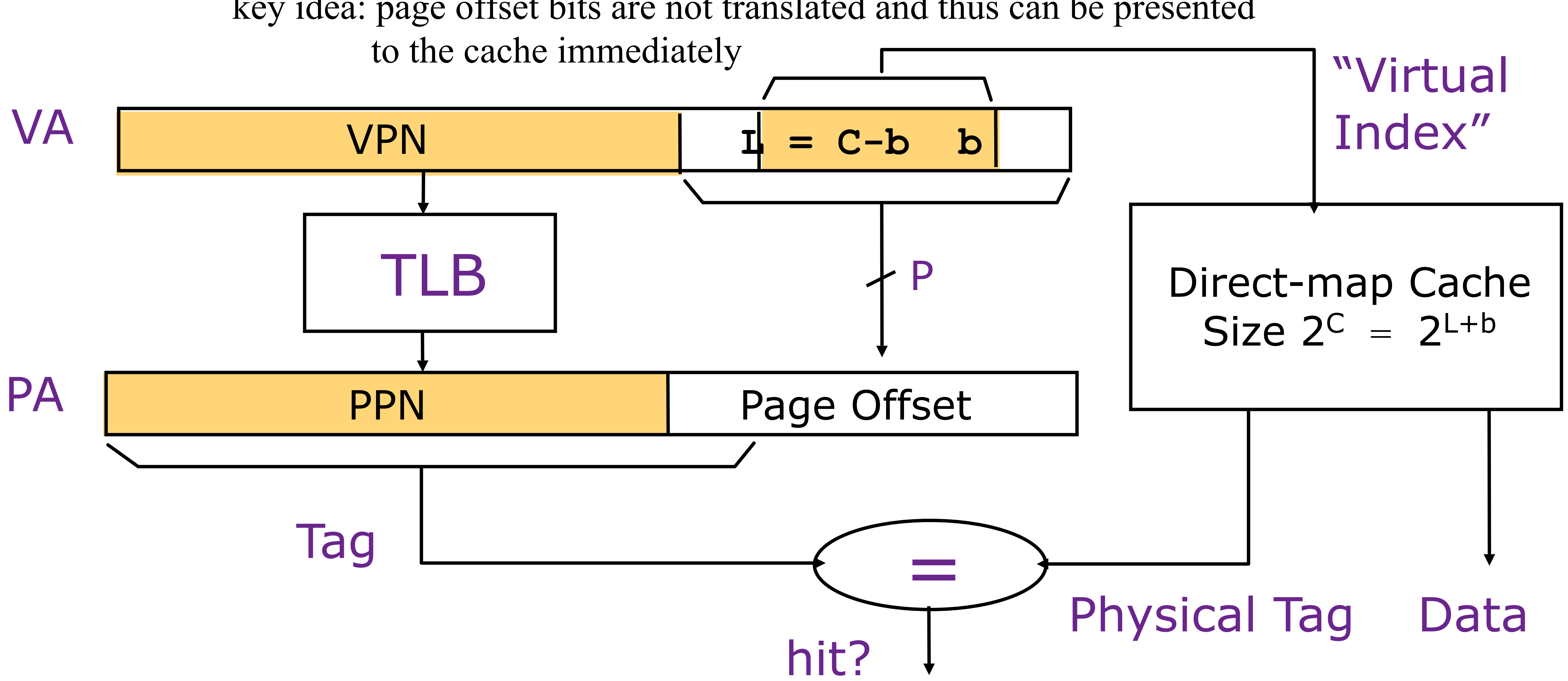
Virtually Indexed/Physically Tagged

- Get the TLB off the critical path by doing the translation and the L1 access in parallel.



-

Solution (4): Virtually indexed physically tagged



Index L is available without consulting the TLB

⇒ *cache and TLB accesses can begin simultaneously*

Critical path = max(cache time, TLB time)!!!

Tag comparison is made after both accesses are completed

Works if Cache Way Size ≤ Page Size (→ C ≤ P)
because then all the cache inputs do not need to be translated

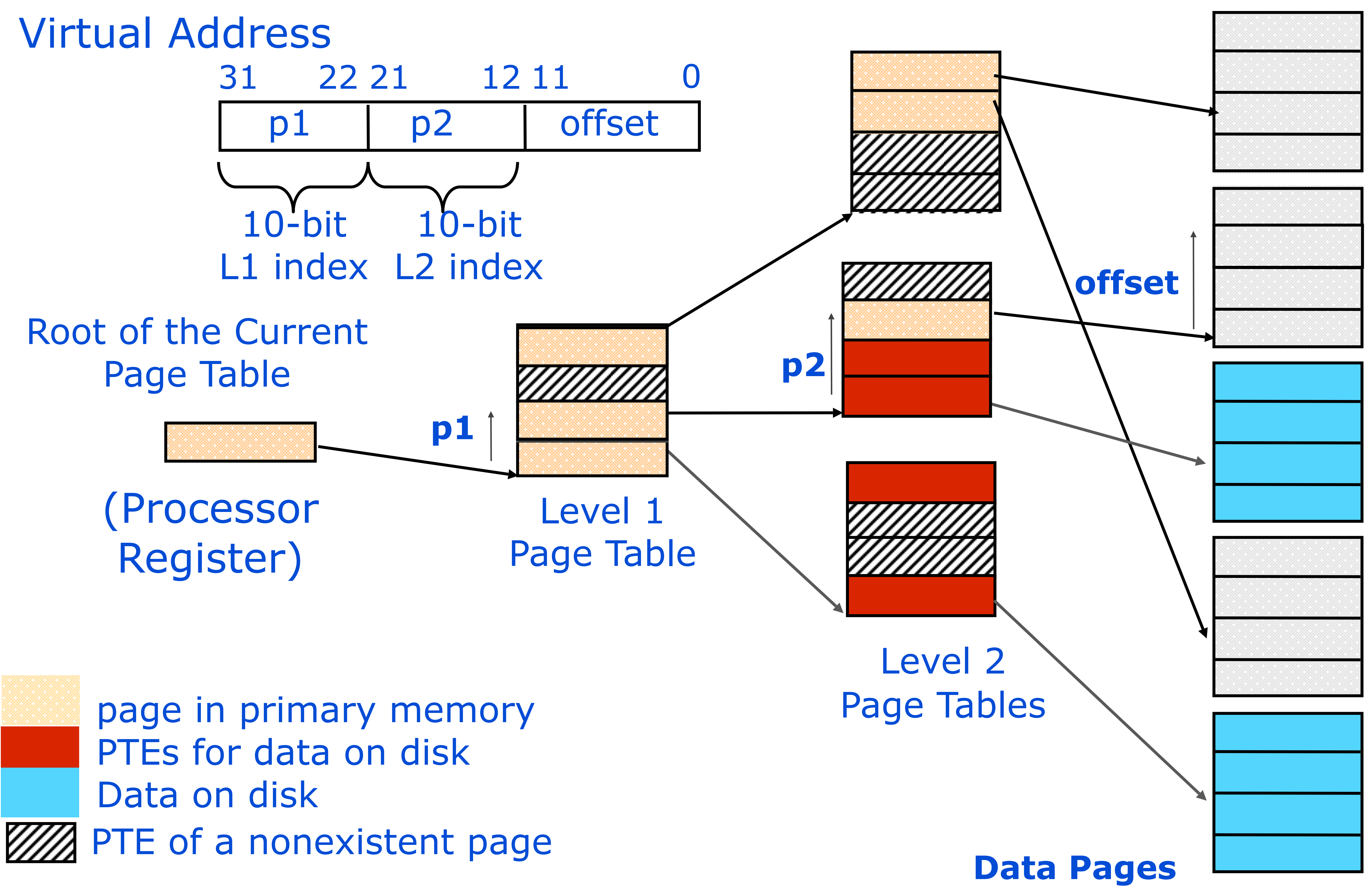
In the Real World

- L1 caches are virtually indexed, physically tagged.
- Lower levels are pure physical
  - Once you go physical, it is not possible (or desirable) to go back.

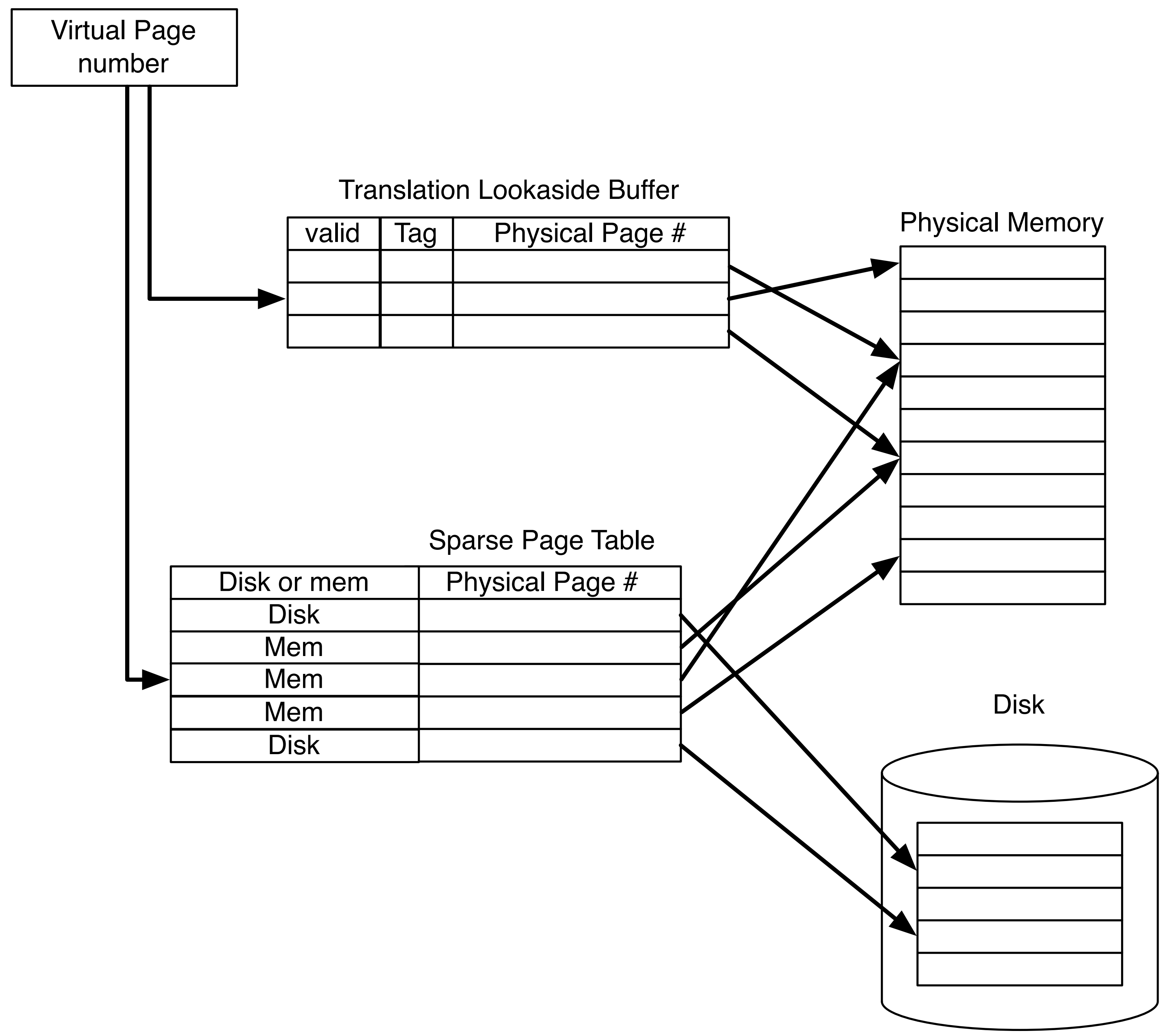
Other uses for VM

- VM provides us a mechanism for adding "meta data" to different regions of memory.
  - The primary piece of meta data is the location of the data in physical ram.
  - But we can support other bits of information as well
- Backing memory to disk
  - next slide
- Protection
  - Pages can be readable, writable, or executable
  - Pages can be cachable or un-cachable
  - Pages can be write-through or write back.
- Other tricks
  - Arrays bounds checking
  - Copy on write, etc.

Page table with pages on disk



The TLB With Disk



- TLB entries always point to memory, not disks

Accessing Memory

- Generate address
- Translate address
    - TLB hit or miss?
        - hit -- yay!
        - miss -- Check page table
            - In memory?
                - yes -- update TLB
                - no -- fetch from Disk, update page table, update TLB
- Access L1
    - L1 hit or miss?
        - hit -- yay! return data
        - miss -- select line for eviction
            - If it's dirty, write it back (potential L2 miss here)
            - Check L2
                - L2 hit or miss?
                    - hit -- yay! return data, update L2
                    - miss -- select line for eviction
                        - if it's dirty, write it back.
                        - Check L3...

Computing AMat

TLB hit time	0
TLB miss rate	9%
Page table access time	44 cycles
Page fault rate	11%
Disk access time	2222222 cycles
L1 hit time	1
L1 miss rate	6%
L2 hit time	33 cycles
L2 miss rate	11%
Memory access time	555 cycles

- Total Amat = Translation time + Memory Time
- This is just the data access side. The math instruction side is the same.

Average TLB access time

- 0 +   // free, since it's in parallel with the L1 access
- TLB miss rate [0.09] * TLB miss penalty
- TLB miss penalty =
  - page table access time [44 cycles] + page fault rate [11%] * page fault time [2222222 cycles]
- 0 + 0.09 * (44 + 0.11* 2222222)

Calculating data access time

- Data access time = L1 hit time [1 cycle] + L1 miss rate [6%] * L1 miss penalty (i.e., L2 access time)
- L2 access time = L2 hit time [33 cycles] + L2 Miss rate [11%] * L2 miss penalty (i.e., memory access time)
  - (33 + 0.11 * 555)
- Memory access time = 555 cycles
- So data access time is
  - 1 + (0.06 * (33 + 0.11 * 555))
- Translation + data access time =
- (0 + 0.09 * (44 + 0.11* 2222222)) + (1 + (0.06 * (33 + 0.11 * 555)))