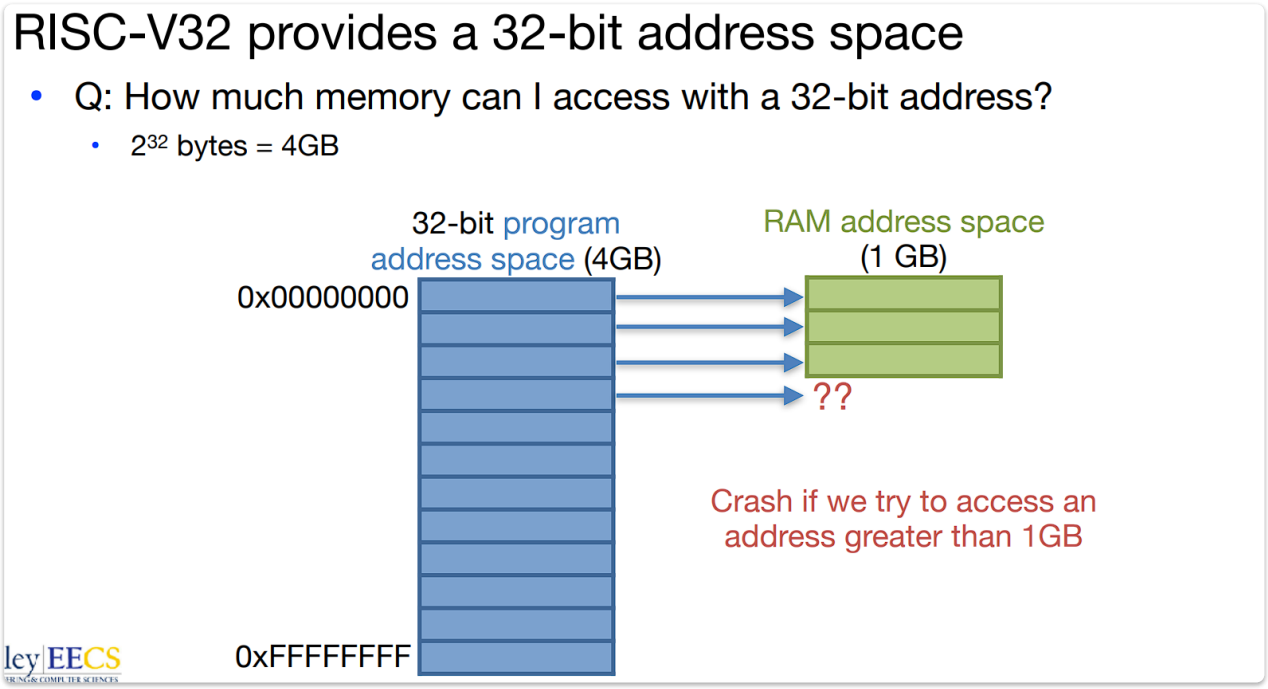
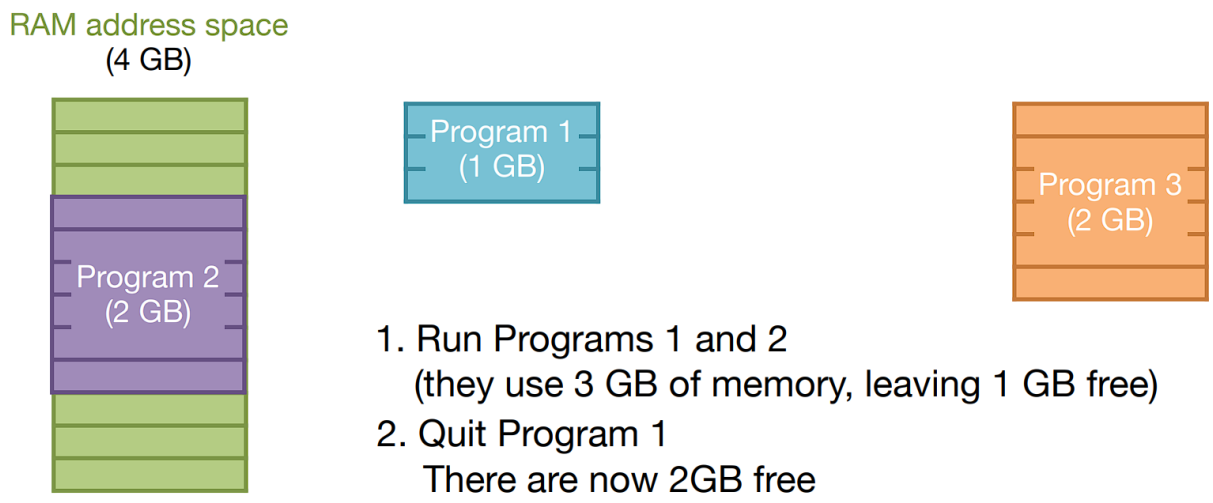
Three Problems with Memory without Translation

VM can be considered as a special type of cache which uses RAM to be the cache of the virtual memory

1. Not enough space.

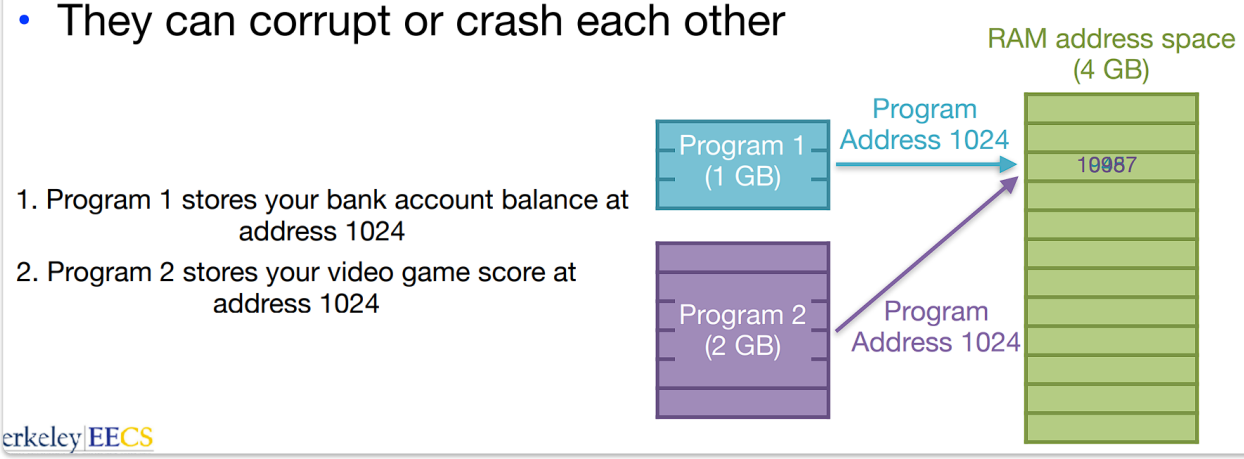


2. Holes in Address Space



3. Ensuring protection from other programs.

- Each program can access any 32-bit memory address
- What if multiple programs access the same address?
- They can corrupt or crash each other



- If all programs have access to the same 32-bit address space
  - Can crash if there is less than 4GB of RAM
  - Can run out of space if we run multiple programs
  - Can corrupt other programs' data

VM

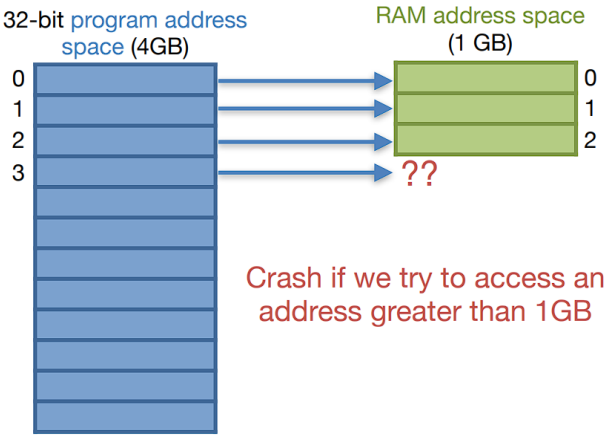
Indirection

Virtual Memory: Indirection

Virtual memory takes program addresses and maps them to RAM addresses

No Virtual Memory

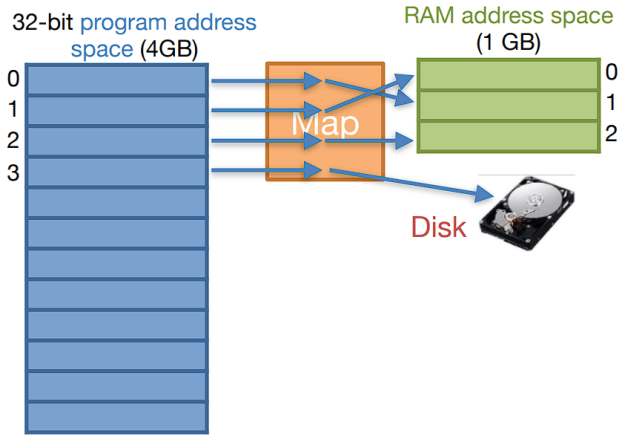
program address = RAM address



Crash if we try to access an address greater than 1GB

Virtual Memory

program address maps to RAM address

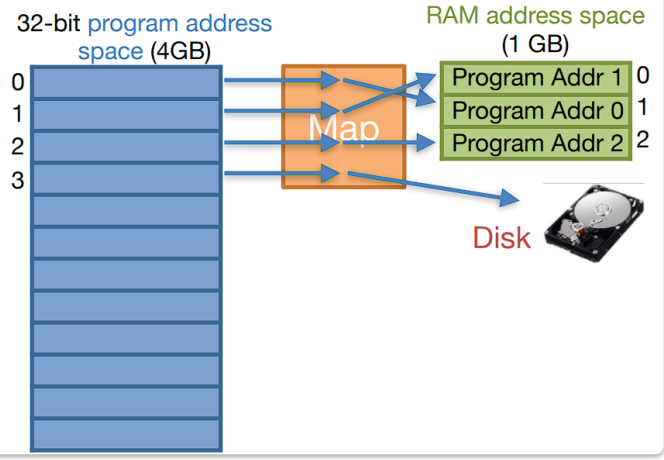


Solving Problem #1: Not Enough Memory

- Map some of the program's address space to disk
- When we need it, bring it into memory

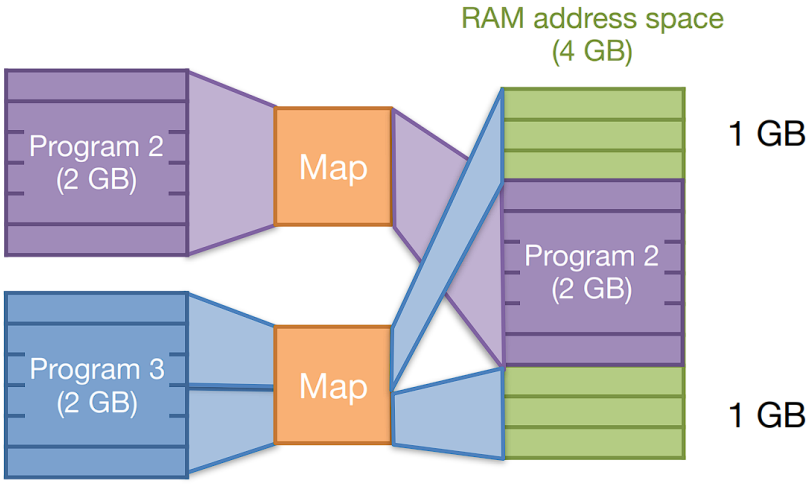
1. Program accesses address 0
2. Program accesses address 1
3. Program accesses address 2
4. Program accesses address 3

Move out least recently accessed data
Bring in address 3 from disk



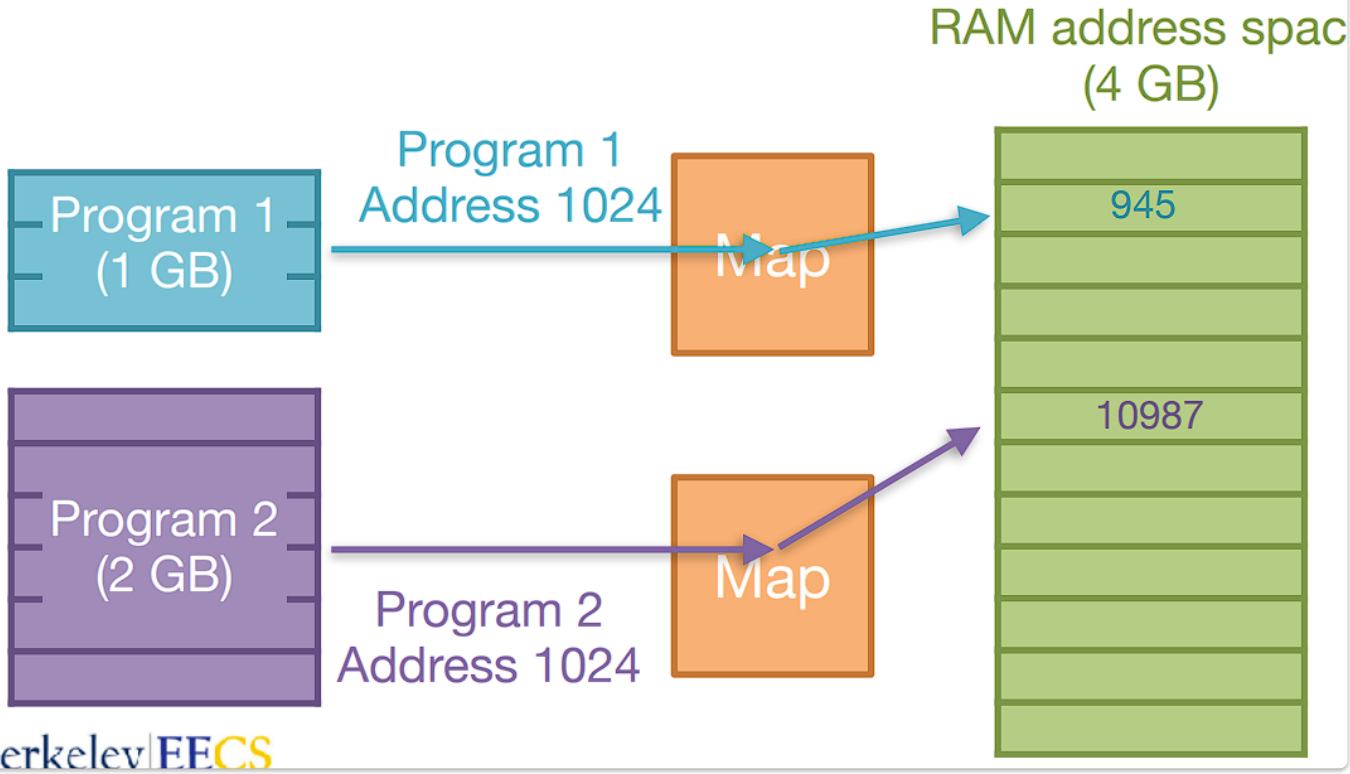
Solving Problem #2: Holes in Address Space

- How can we fit our program in the available space?

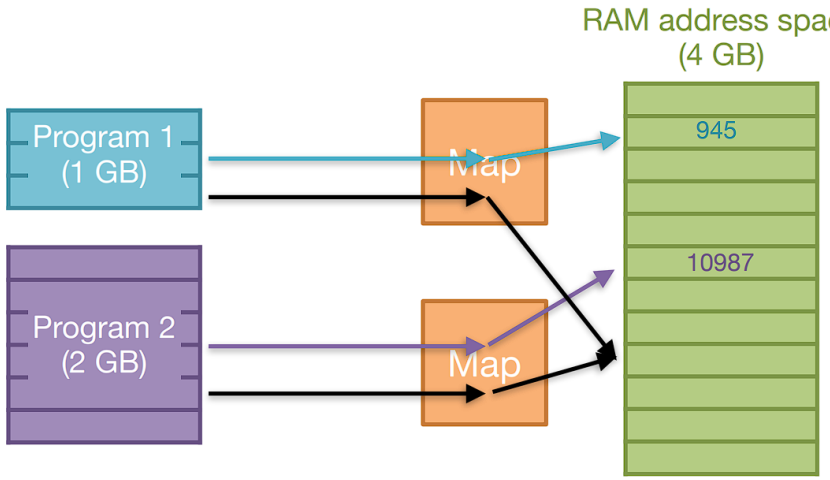


Each program has its own mapping

Mappings let us put our program data wherever we want in RAM



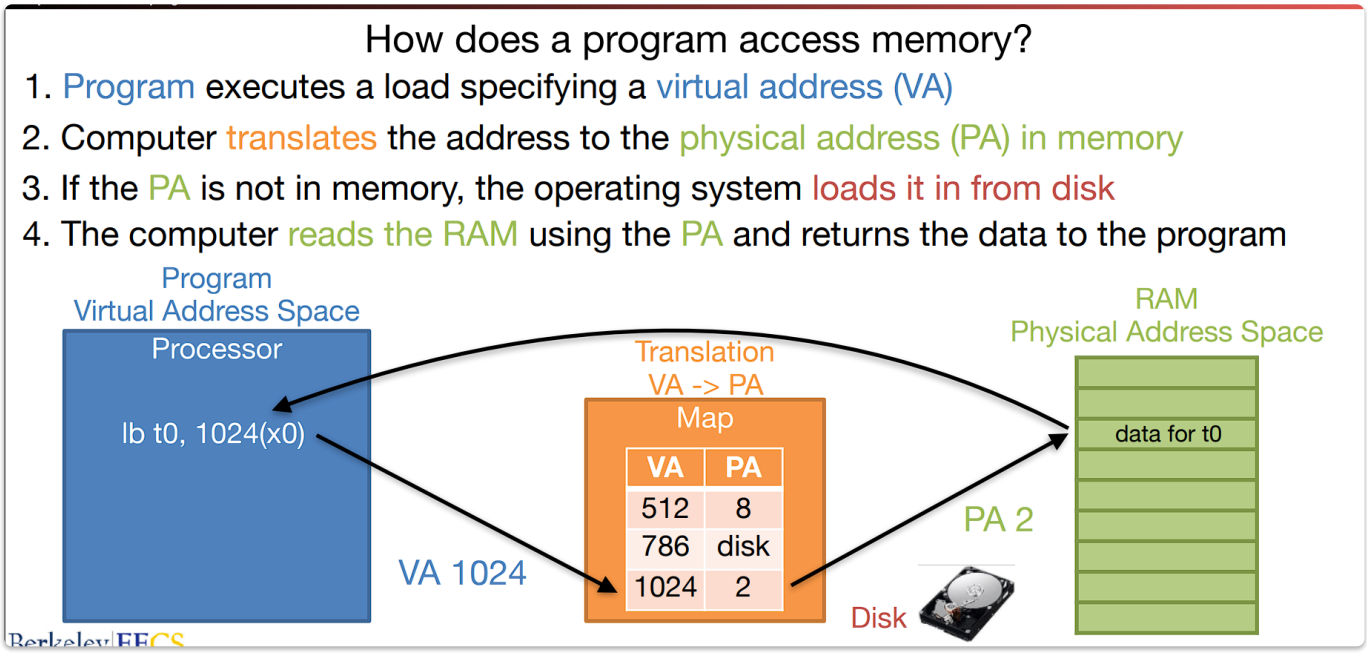
- Programs share a lot of data (eg libraries)
- VM enables us to easily share data while protecting private data!



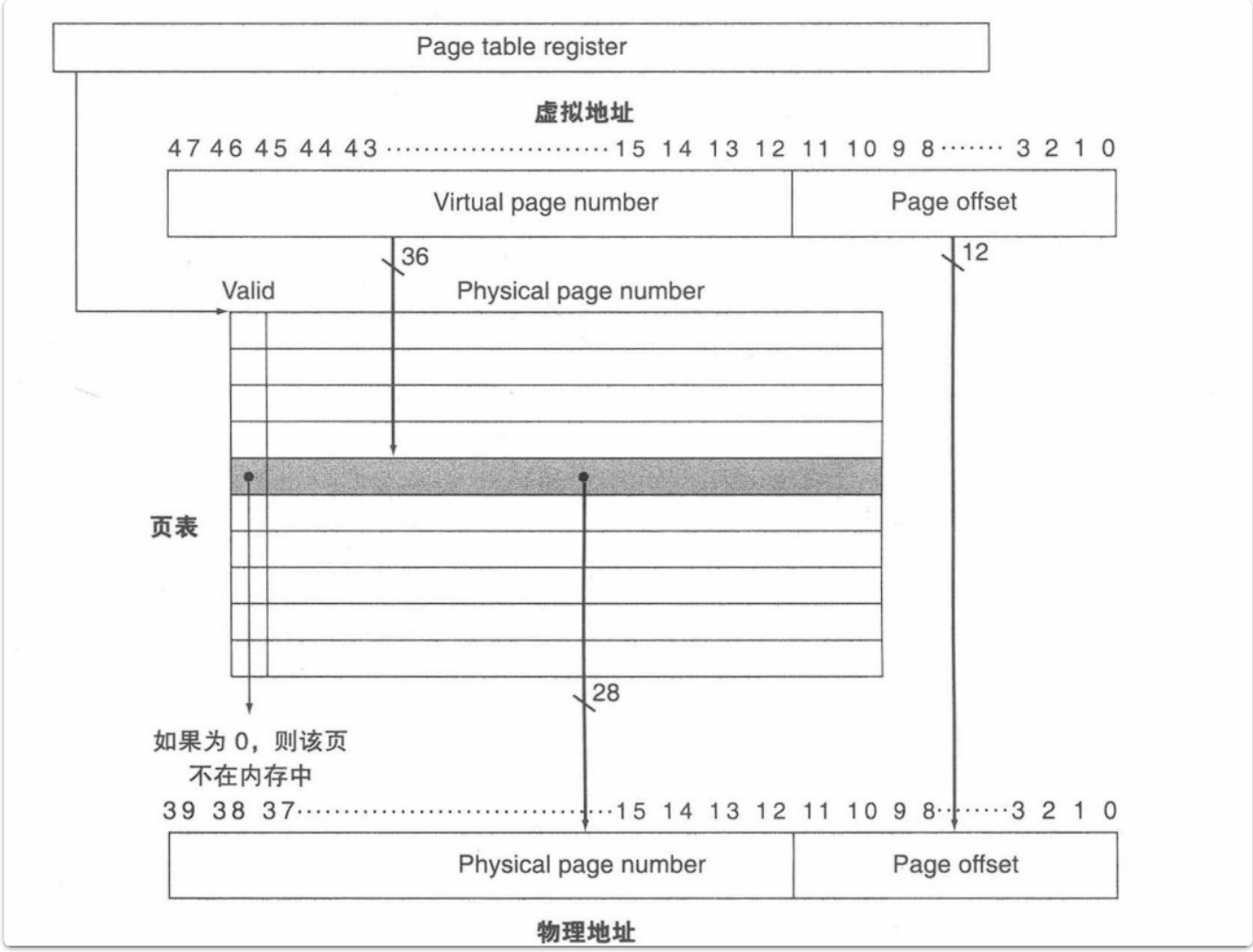
How does VM work ?

- Virtual Address (VA)
  - What the program sees
  - e.g. lw t0, 1024(x0) # read virtual address 1024
  - In RV32I, we can access bytes 0 to 2^{32} - 1
- Physical Address (PA)
  - The physical RAM in the computer
  - Address space is determined by how much RAM you have
  - If you have 2GB of RAM, you can access bytes 0 to 2^{31} - 1

Translation


How does a program access memory?
1. Program executes a load specifying a virtual address (VA)
2. Computer translates the address to the physical address (PA) in memory
3. If the PA is not in memory, the operating system loads it in from disk
4. The computer reads the RAM using the PA and returns the data to the program
Program
Virtual Address Space
Processor
lb t0, 1024(x0)
VA 1024
Translation
VA -> PA
Map
VA
PA
512
8
786
disk
1024
2
PA 2
Disk
RAM
Physical Address Space
data for t0
BerkeleyEECS

Page Table



- The map from Virtual Addresses (VA) to Physical Addresses (PA) is the Page Table
- So far we have had one Page Table Entry (PTE) for every Virtual Address
- If we have one entry for every word in our address space, how many entries would we have?
  - 2^{30} = 1 billion!

Fine Grain

Map each word address

2^{30} entries

Page Table
VA -> PA

Map

VA	PA
500	32
1088	64
544	0

Coarse Grain

Map chunks of addresses

Fewer entries

Page Table
VA -> PA

Map

VA	PA
0-4095	4096-8191
…	…
…	…

Each entry now covers 4KB of data

- Today, page tables are usually 4KB (1024 words)
- Q: How many pages do we need in our page table with 4KB pages on a 32-bit machine?
  - 2^{32} bytes / 2^{12} bytes = 2^{20} = 1 million

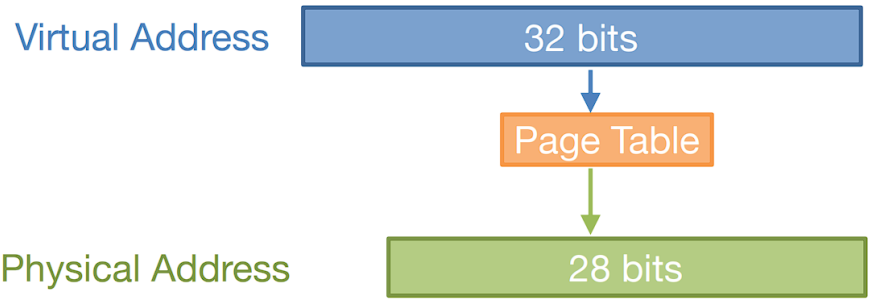
Number of bytes in memory

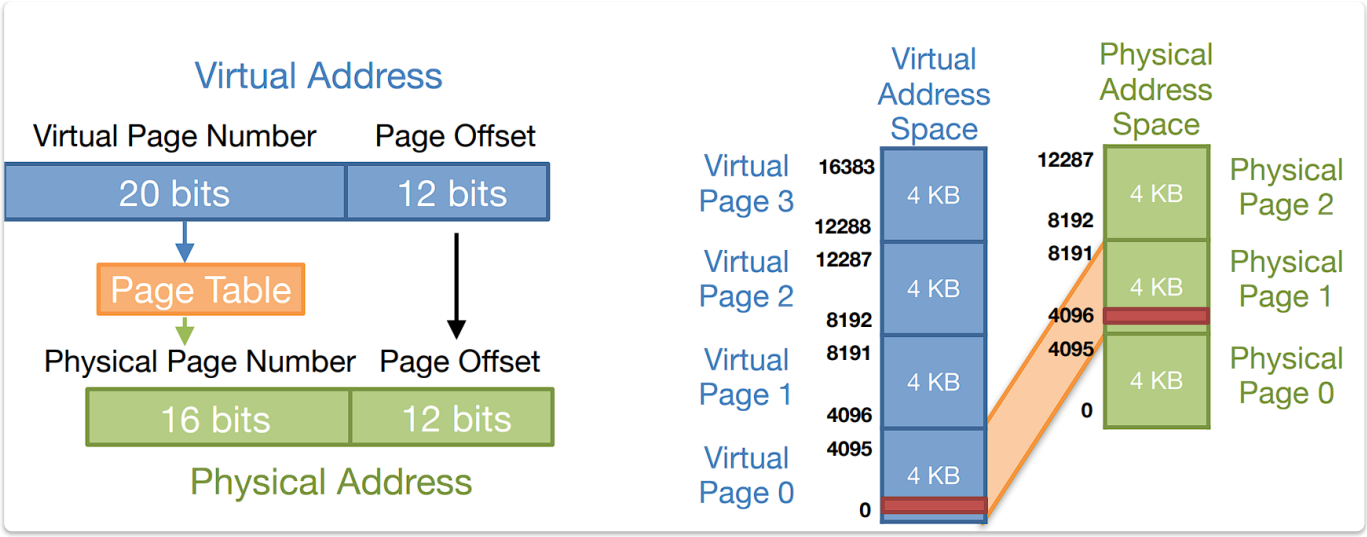
4 KB

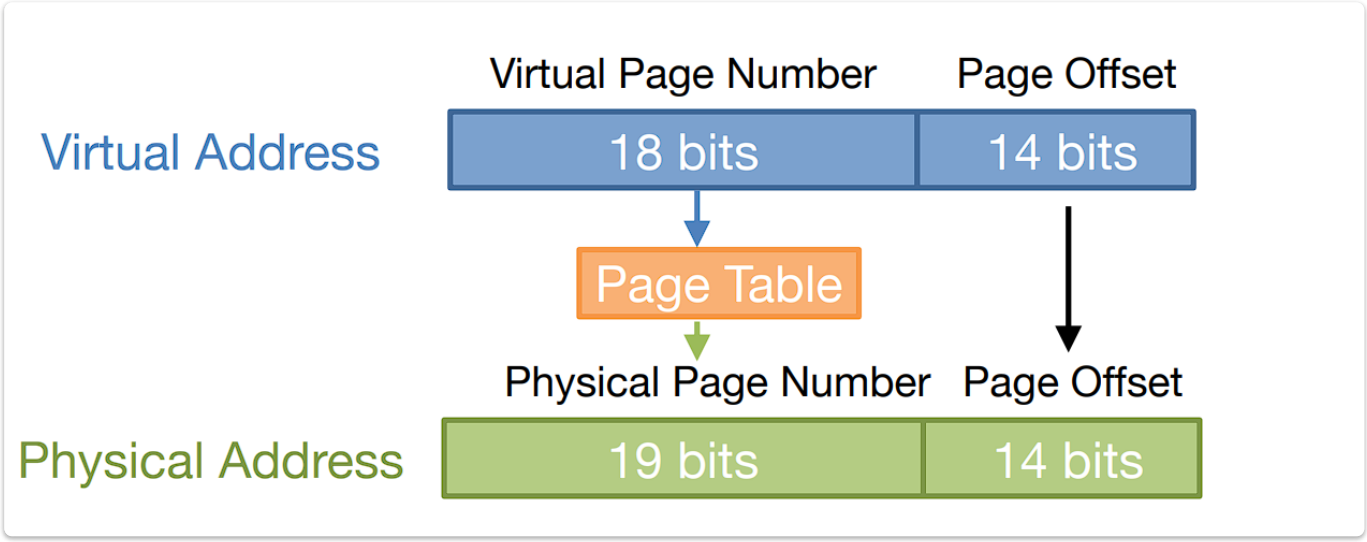
- Q: How many entries do we have in our page table?
  - 1 million

What is the size of our virtual address and physical address on a 32 bit machine with 256 MB of RAM and 4KB pages?

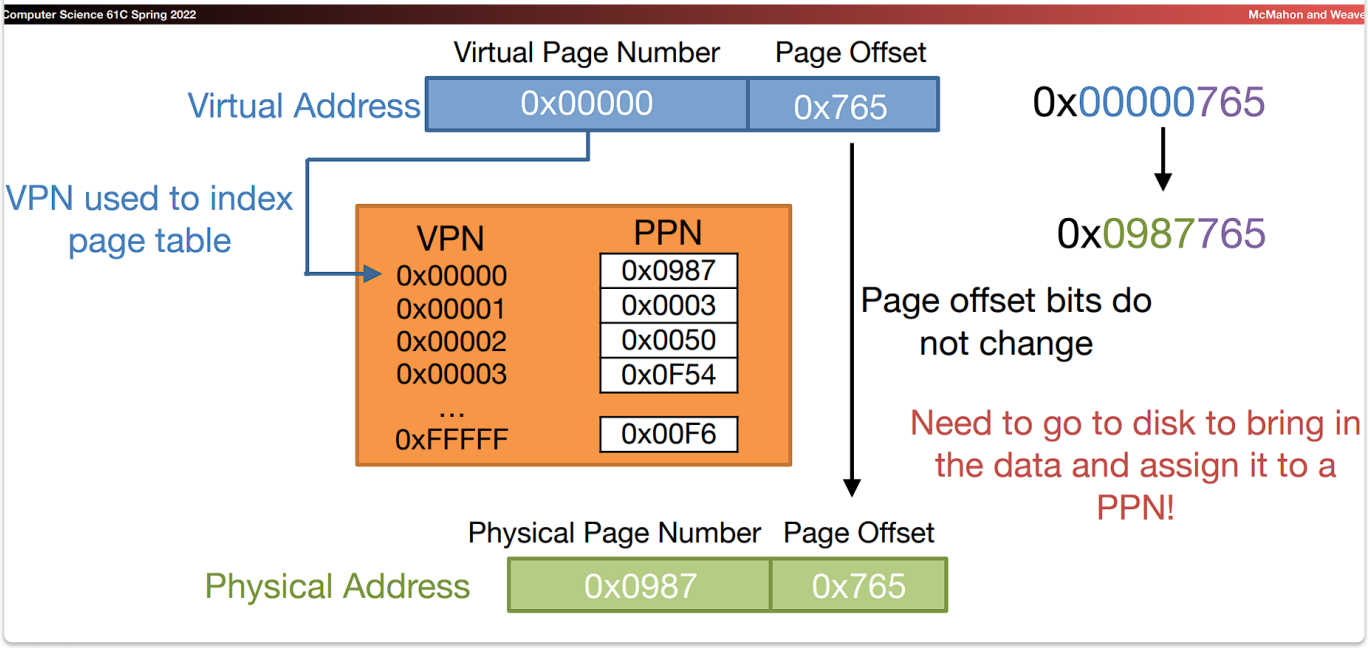
- VA size = 32 bits
- PA size = $\log_2(256\ MB) = \log_2(2^8 * 2^{20})$ = 28 bits







Example Translation #2



Page Faults

How do I know if the page is in disk?

- The PTE points to disk.

What happens when a page is not in RAM?

1. Hardware (CPU) generates a **page fault exception**
2. The hardware jumps to the OS page fault handler to fix it
3. The OS chooses a page to evict from RAM (if no more free space)
4. If the page is **dirty**, it needs to be written back to disk first
5. The OS updates the corresponding PTE to point to disk
6. The OS brings in the page we wanted disk and puts it in RAM
7. The OS updates the PTE of the new page
8. The OS jumps back to the instruction that caused the page fault (This time it won't cause a page fault since the page has been loaded.)

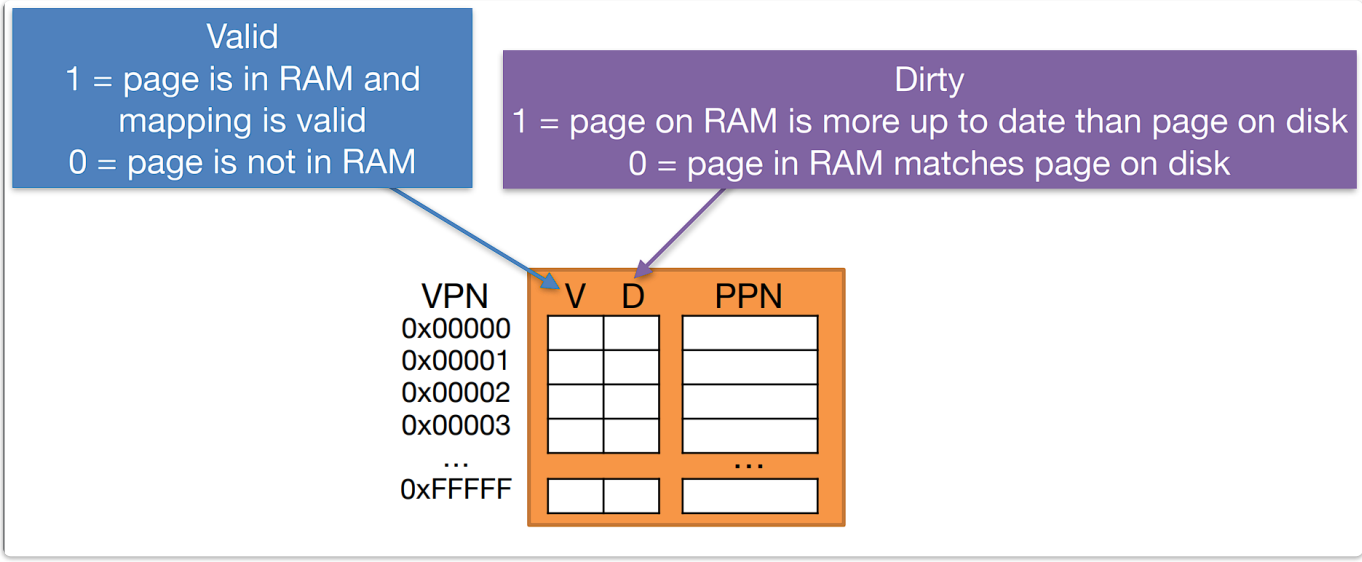
Page Replacement Policies

- FIFO
  - Easy to implement.
- LRU
  - Most efficient but expensive to implement.

- RANDOM
  - LRU tends to be better, but there are exceptions
  - Also, LRU is expensive
    - Need to update on each access
    - Lots of metadata
    - We usually use an approximation
  - Random would be better for the following access pattern if we can only store 4 pages in memory
    - 0, 1, 2, 3, 4, 0, 1, 2, 3

Memory Protection

Additional Page Table Metadata



Just like cache.

Page Protection Bits

- Can specify how a process can use each page

  Read

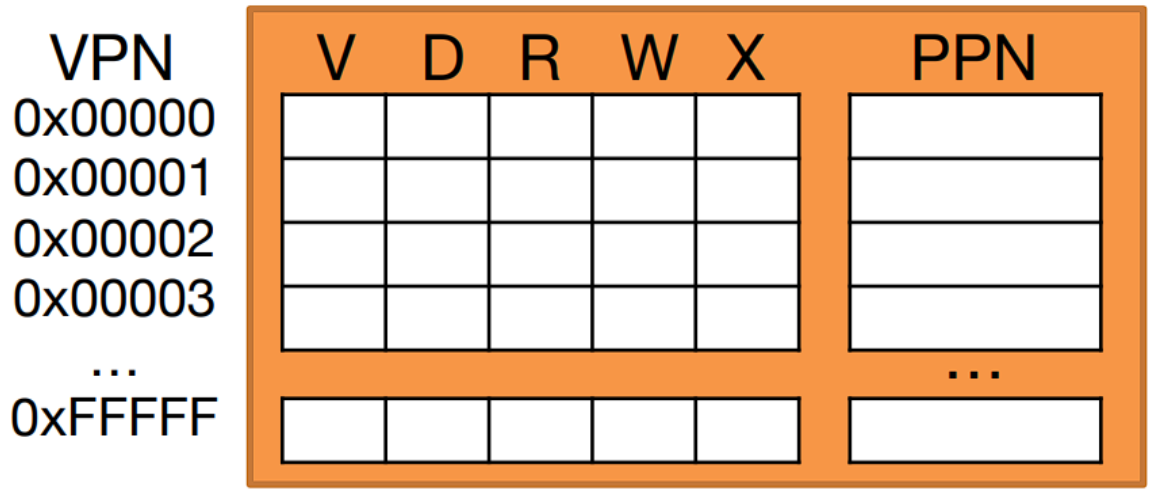
Write

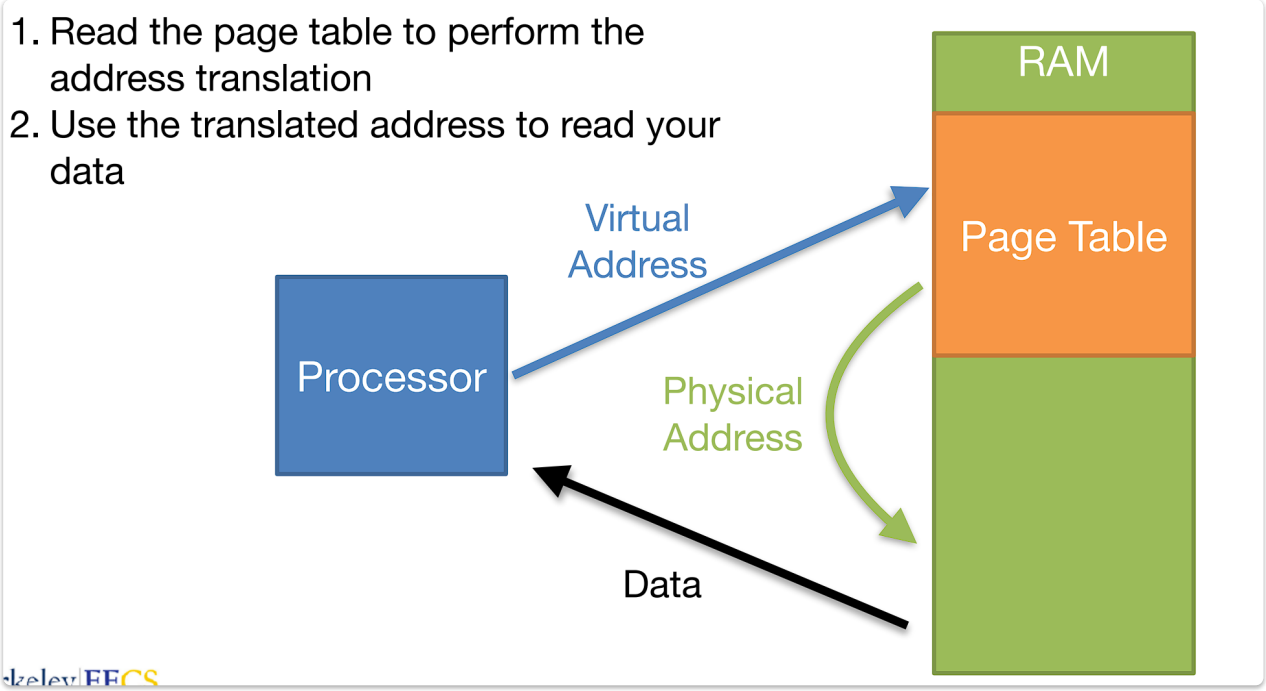
* execute

If don't have permission to do what you are trying, you'll get segfault or memory protection fault.

1 人赞同了该回答

页表基址寄存器和页表项里记录的都是页表/页面的物理地址或者说页框号



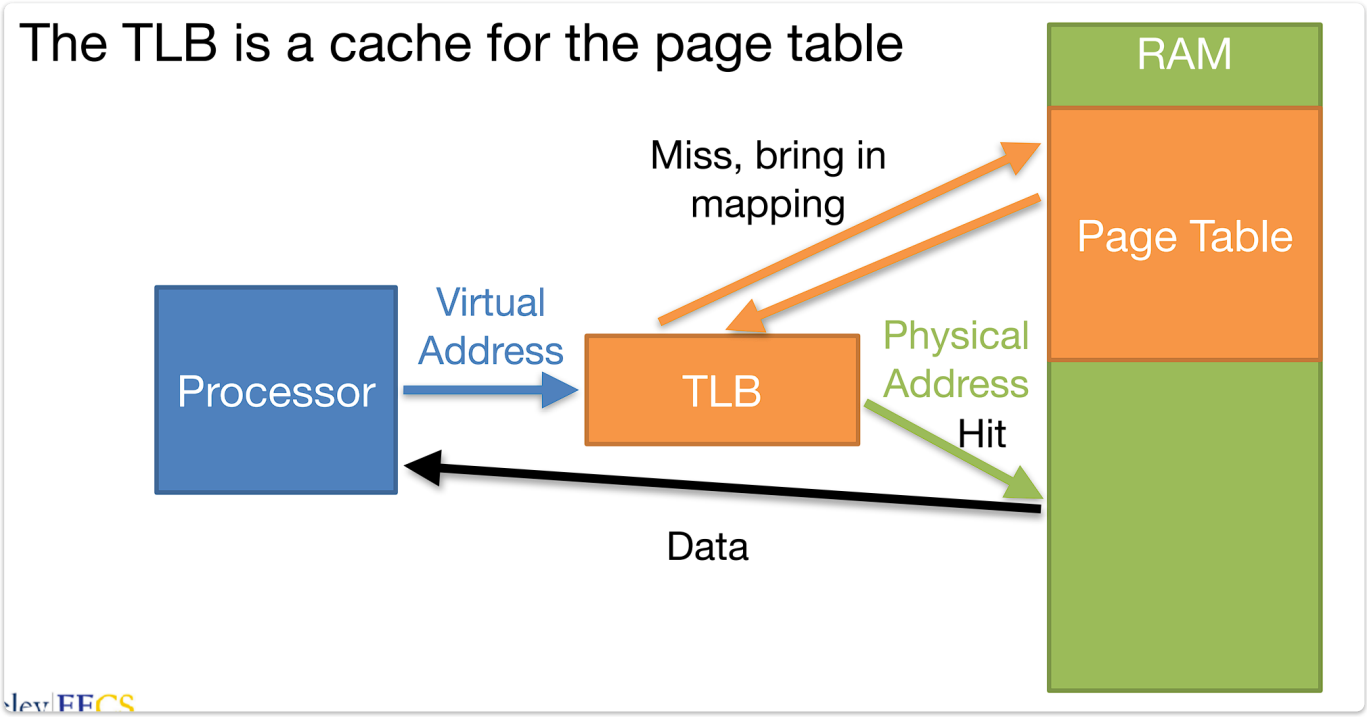


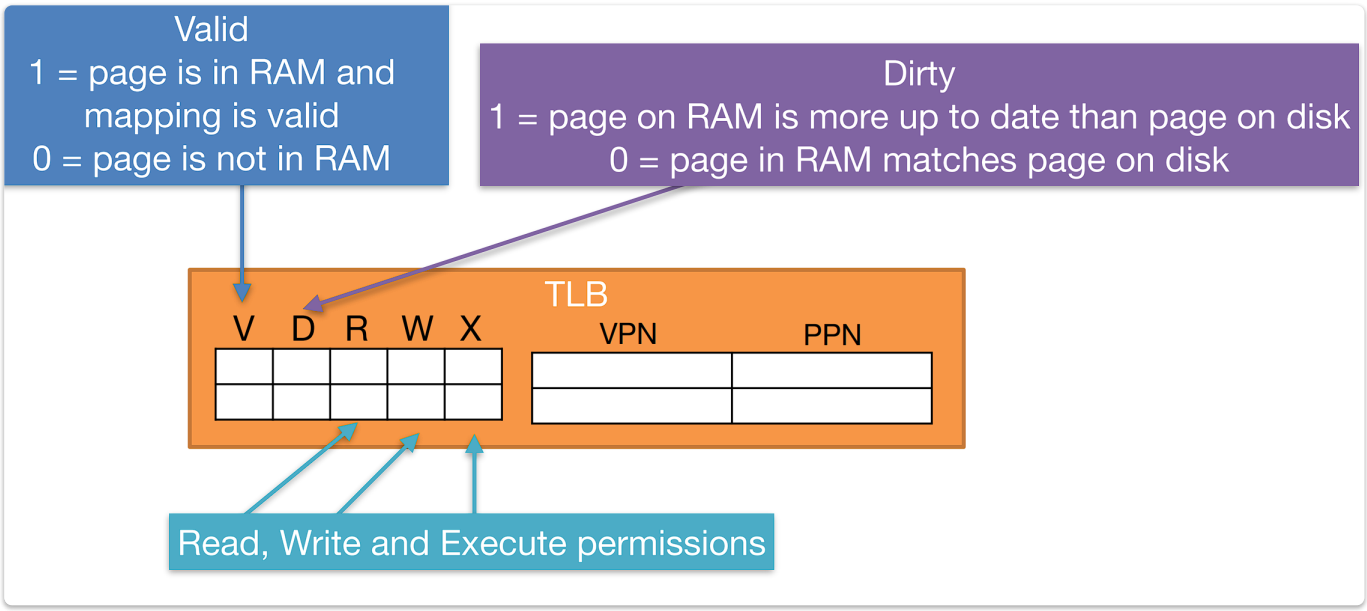
With virtual memory, it takes one more time to access to memory because you need to look up in

page table first.

To make this faster, we can implement a **Page Table Cache**

Translation Lookaside Buffer(TLB)





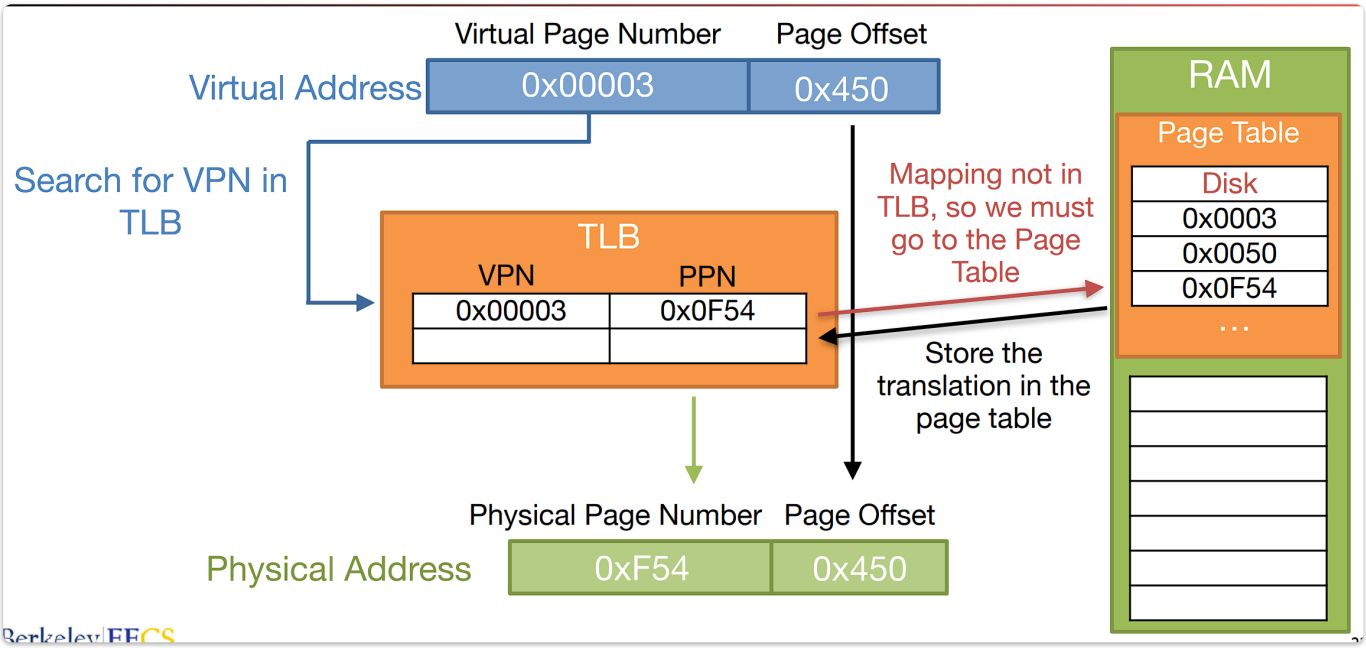
- To be fast, TLBs must be small
- Usually Fully Associative
- Typically 32-128 entries
- Each entry maps to a large page
  - Takes advantage of spatial and temporal locality
- Random or FIFO replacement policy

TLB Flush

Context switch - changing which thread is executing

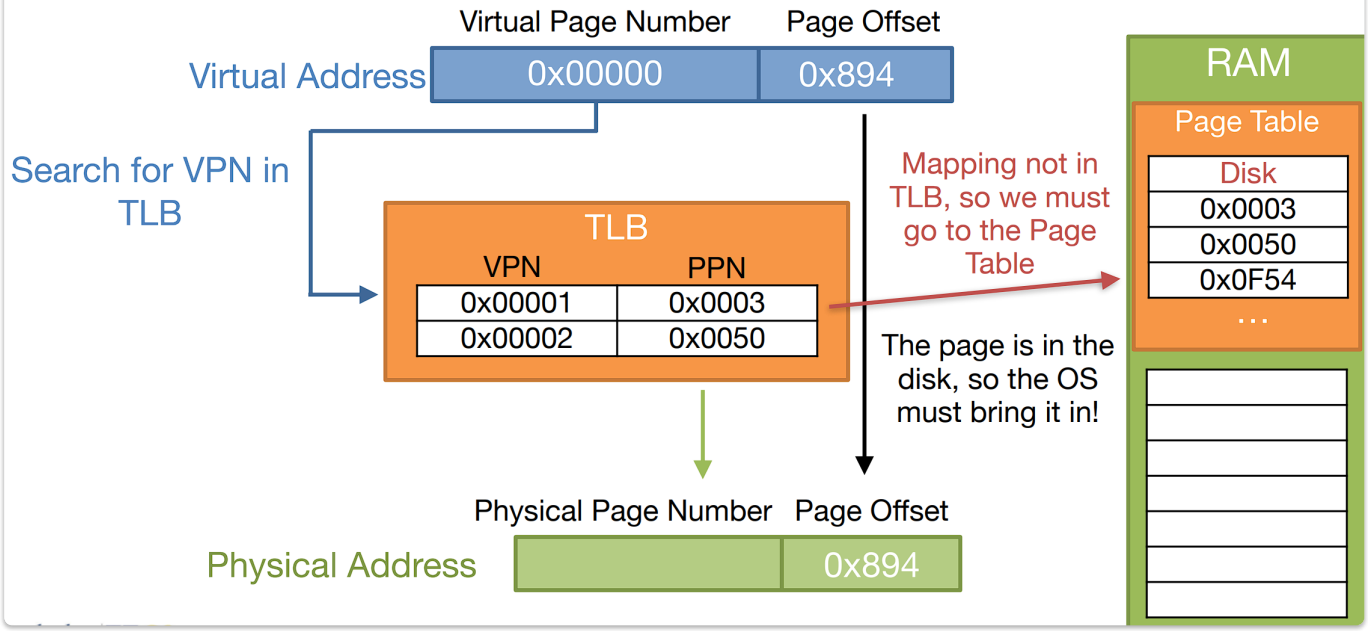
The entries in the TLB correspond to the currently active process

On a context switch, the TLB is flushed (all entries are invalidated)

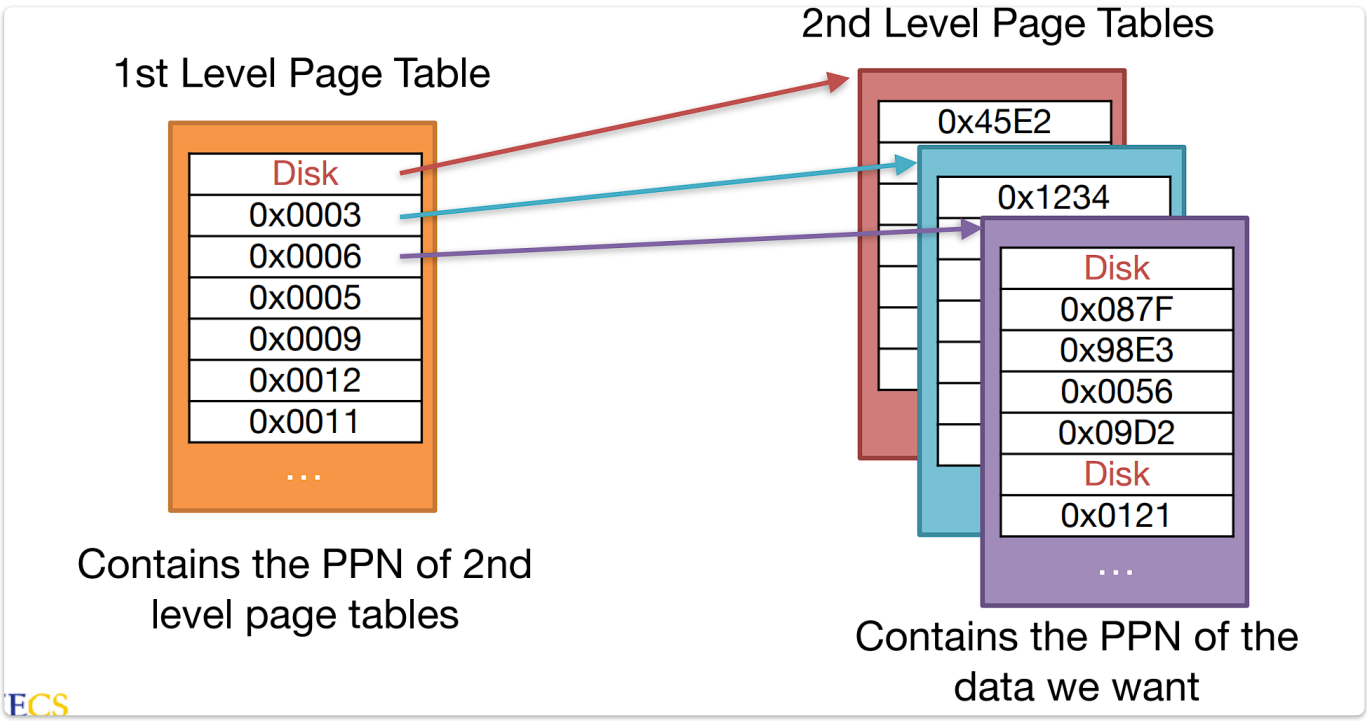


TLB Example #5

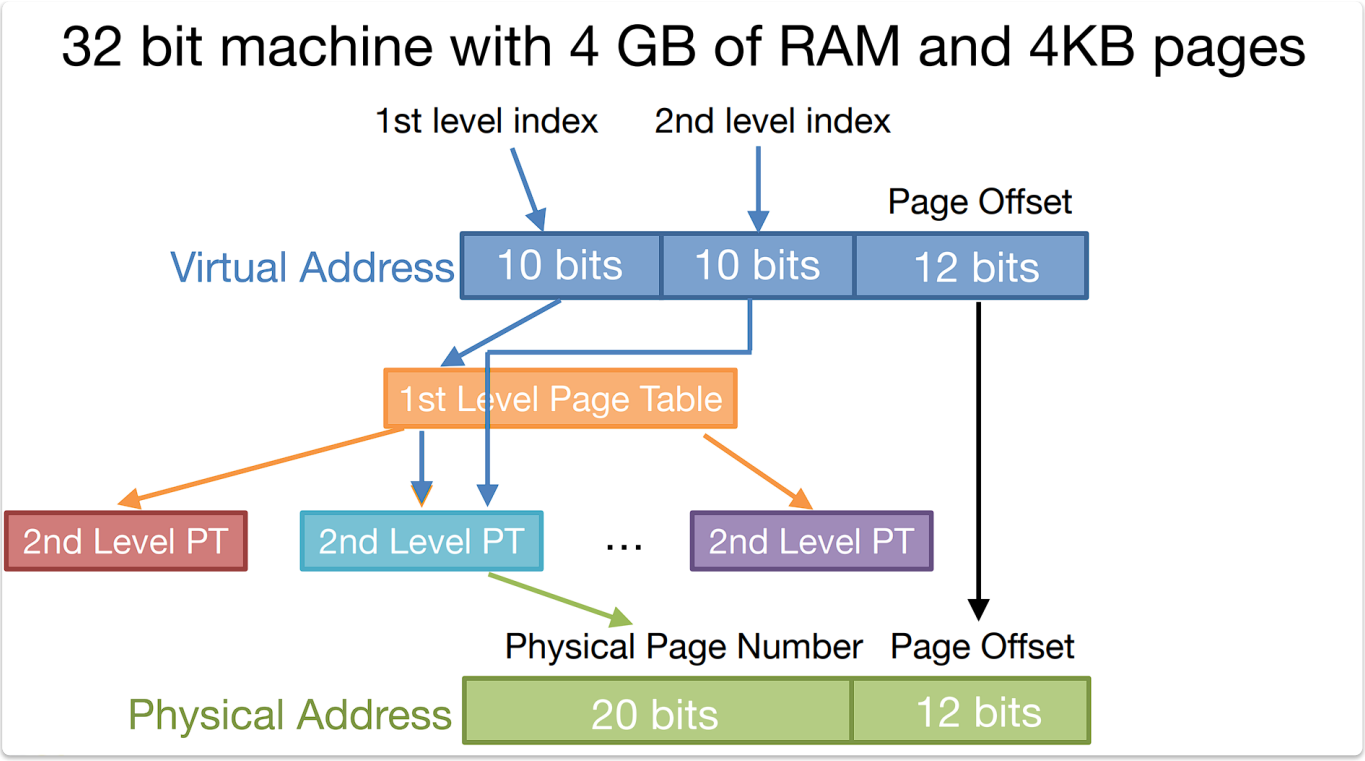
0x00000894



Multi-level Page Tables

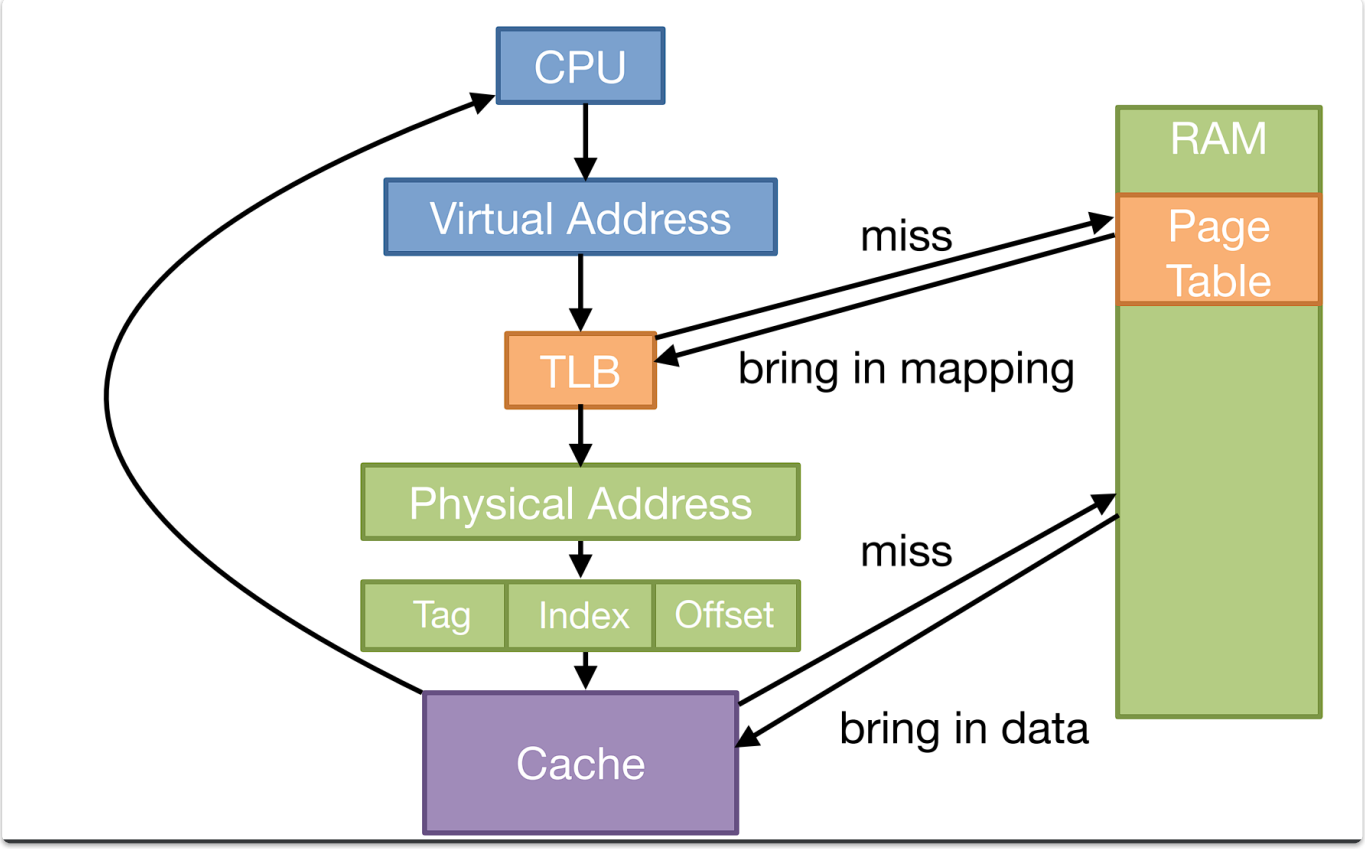


- The 1st level page table must ALWAYS be in RAM
- The 2nd level page tables can be paged out to disk because we can find them through the 1st level page table



Caches and Virtual Memory

Physically Indexed, Physically Tagged Cache

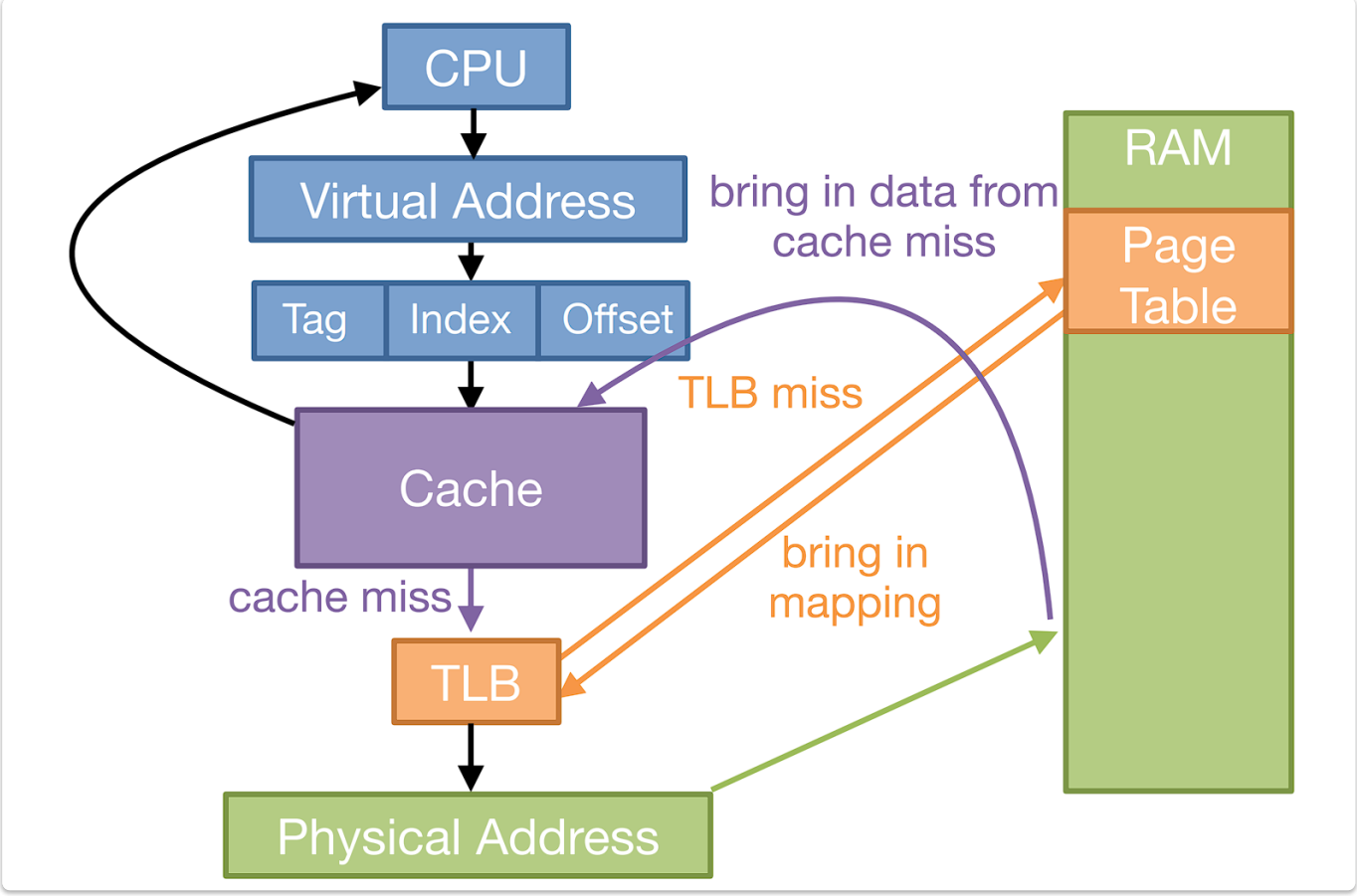


- Can we use the same cache for multiple processes?
  - Yes! If the cache is physically indexed, physically tagged, we can use the same cache
- Downside
  - Must translate address before accessing the cache

Synonyms : Different virtual address can map to same physical address.

Homonyms : Same virtual address can map to different physical address.

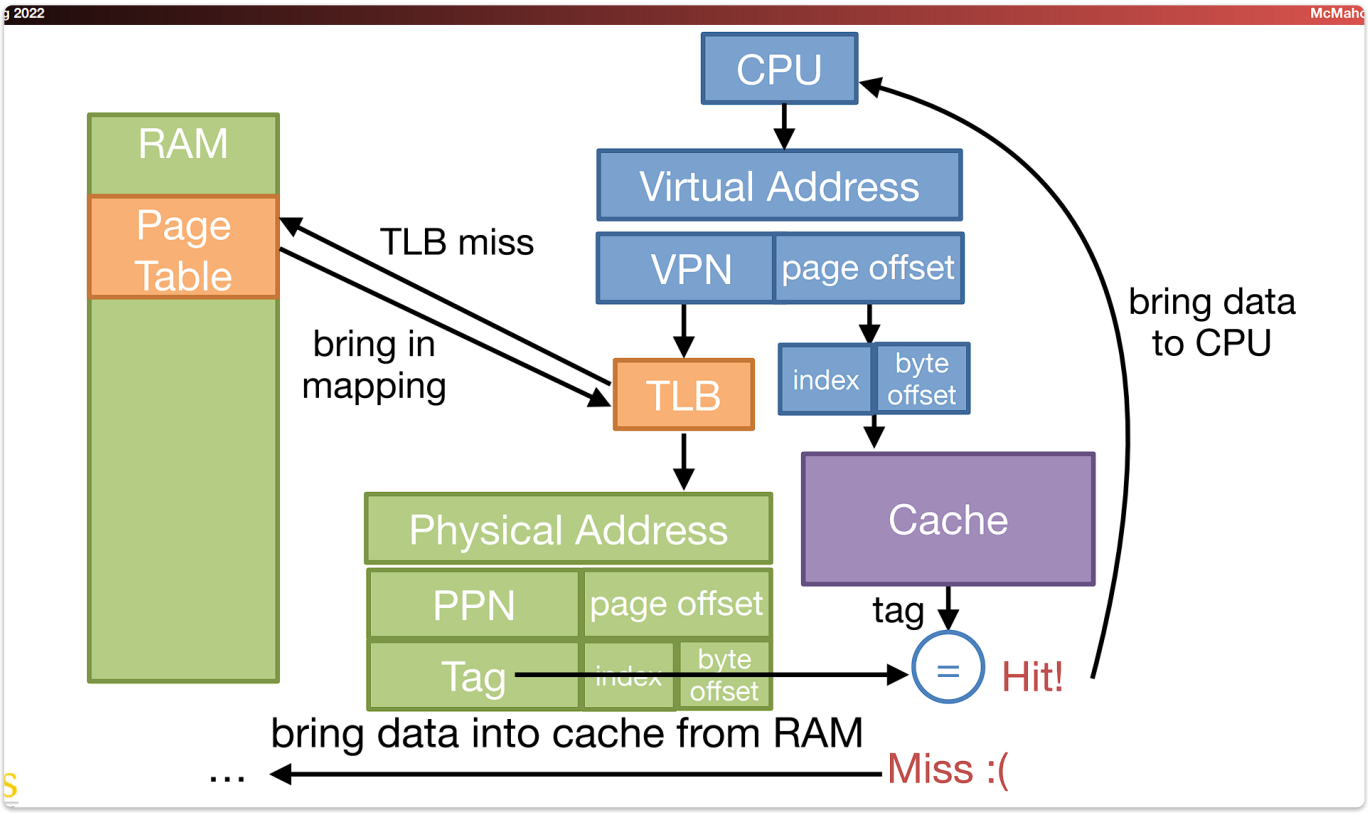
Virtually Indexed, Virtually Tagged Cache



- Can we use the same virtual cache for different processes?
  - No because of synonyms and homonyms
- How do we solve this issue?
  - Flushing the cache on a context switch

Virtually Indexed, Physically Tagged Caches

- We want to use the physical address to access the cache
  - To avoid synonym and homonym problem
- Can we perform the TLB lookup and cache index lookup in parallel?
- How can we get part of the physical address from the virtual address without going through the TLB?
  - Some of the bits remain the same after translation



Start form the index to find the location

- The size of our cache is limited by the number of page offset bits
- Q: With 4kB pages, how many bytes can a direct-mapped (1-way) VIPT cache store?
  - 4kB
  - We can only use the page offset bits (12 bits for 4kB pages) to index into the cache. So the index can only address 12 bits of address, or 4kB of data