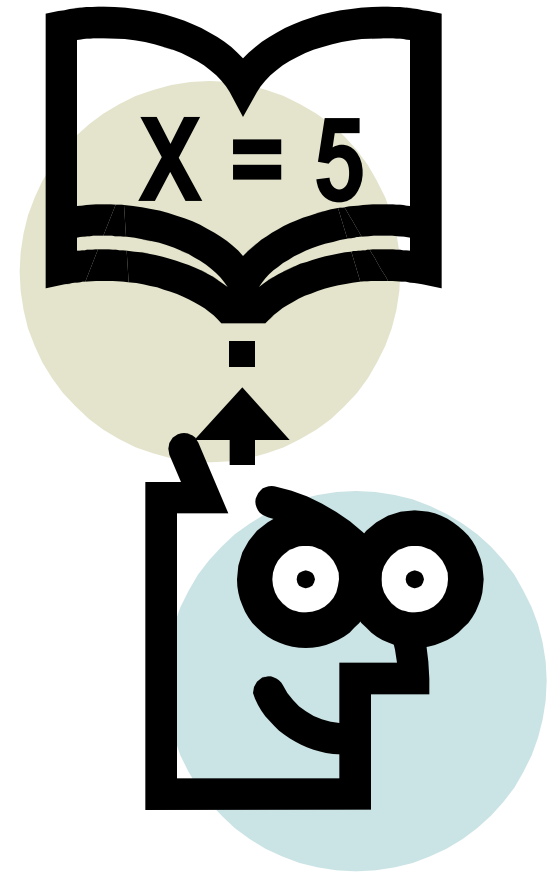

Lecture 17

Virtual Memory



Adapted from slides originally developed by Profs. Hardavellas, Hill, Falsafi, Marculescu, Patterson, Rutenbar and Vijaykumar of Northwestern, Carnegie Mellon, Purdue, UC-Berkley, UWisconsin

Today's Menu:

▶ **Virtual Memory**

- ▷ Virtual vs. Physical Address Spaces
- ▷ Page Table
- ▷ Address Translation
- ▷ Page Fault
- ▷ Replacement Policy
- ▷ Protection

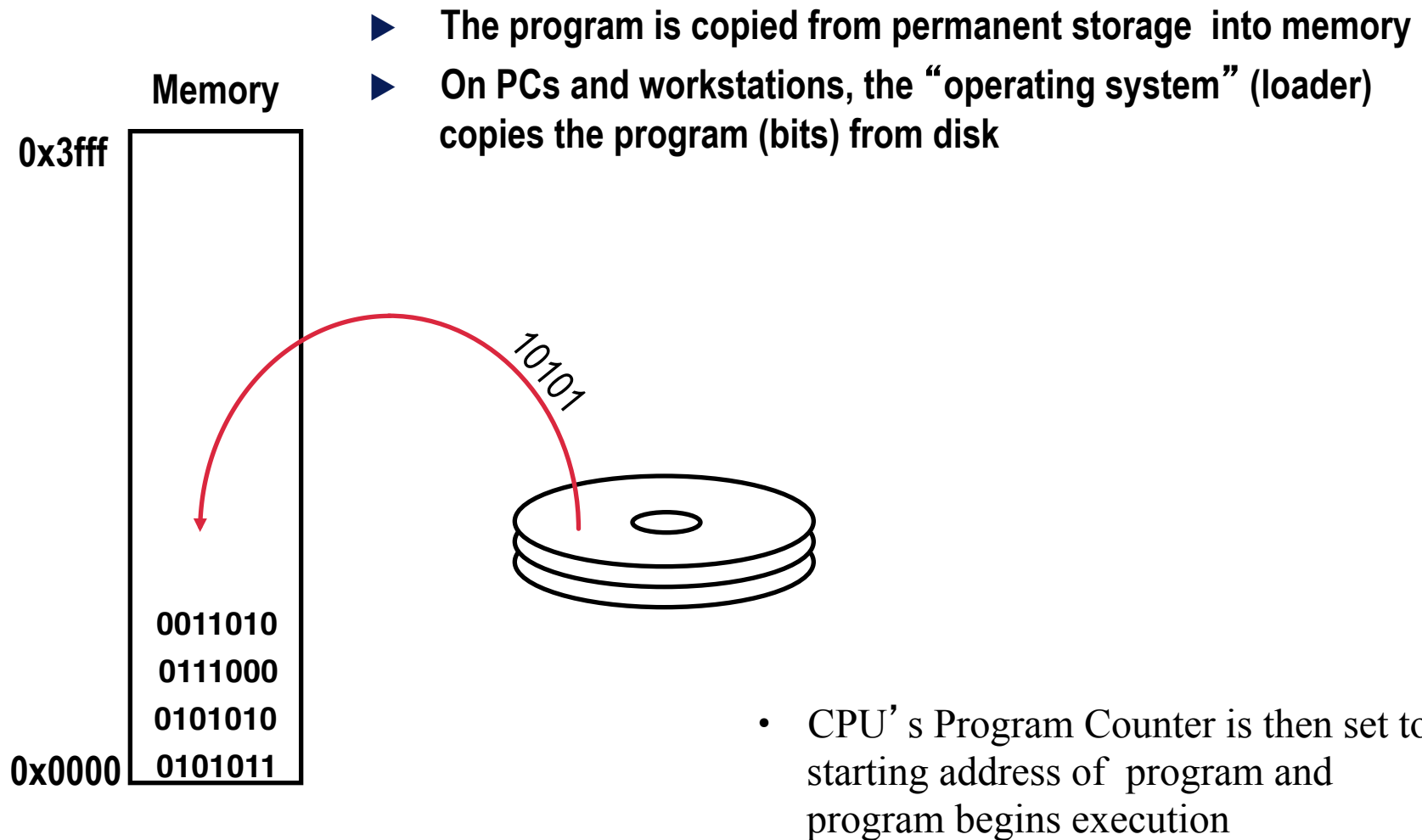
▷ **Accelerating address translation**

- ▷ Translation Lookaside Buffer (TLB)

▷ **Cache and TLB interactions**

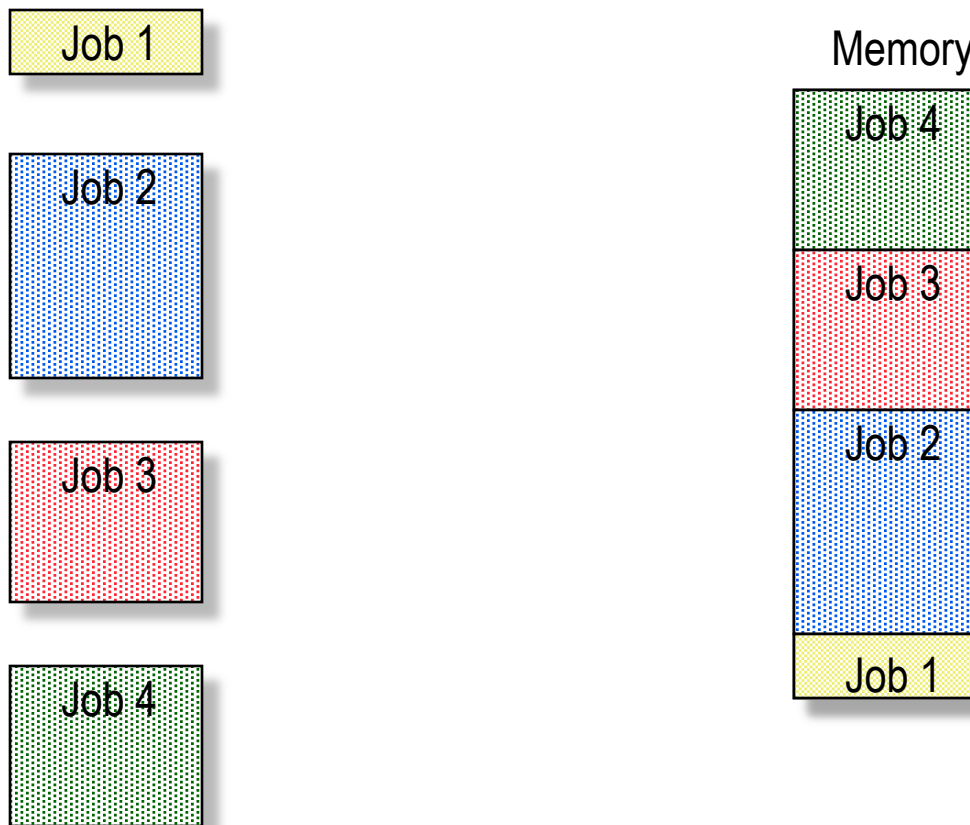
- ▷ Physical Caches vs Virtual Caches

How Does a Program Start Running?



Memory Allocation for Multiprogramming

- Multiple jobs (multiprogramming) each require memory



4 jobs and their memory requirements

Virtual Memory

► What is **virtual memory**?

- ▷ Technique that allows execution of a program that
 - ▷ can reside in non-contiguous memory locations
 - ▷ does not have to completely reside in memory
- ▷ Allows the computer to give the illusion to the program that
 - ▷ memory is contiguous
 - ▷ memory space is larger than physical memory

► Why is VM important?

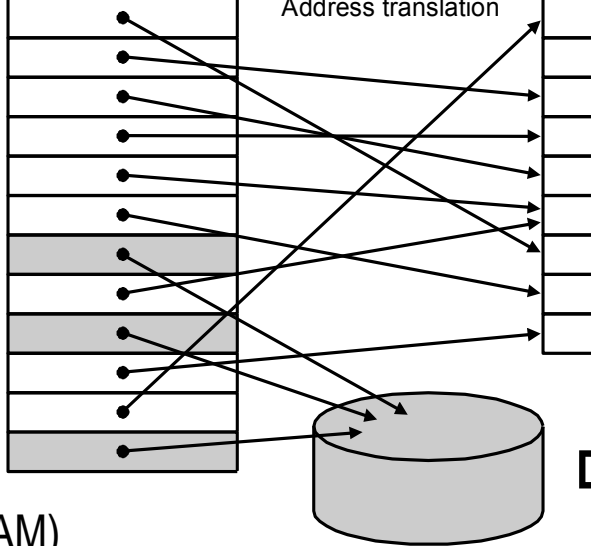
- ▷ Cheap—no longer have to buy lots of DRAM
- ▷ Removes burden of memory resource management from the programmer
- ▷ Enables multiprogramming, time-sharing, protection

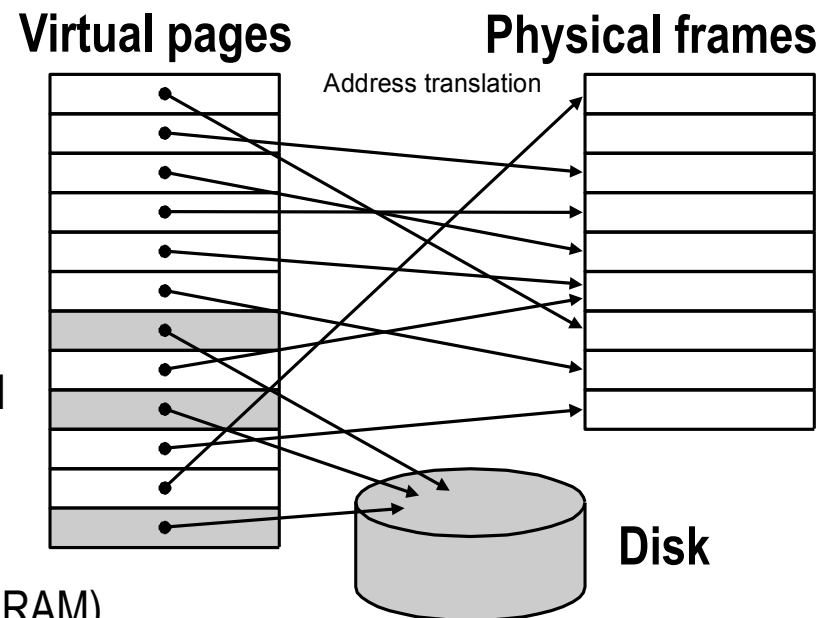
How Does VM Work

► Two memory “spaces”

- ▷ **Virtual memory space** - what the program “sees”
- ▷ **Physical memory space** - what the program runs in (e.g., size of DRAM)

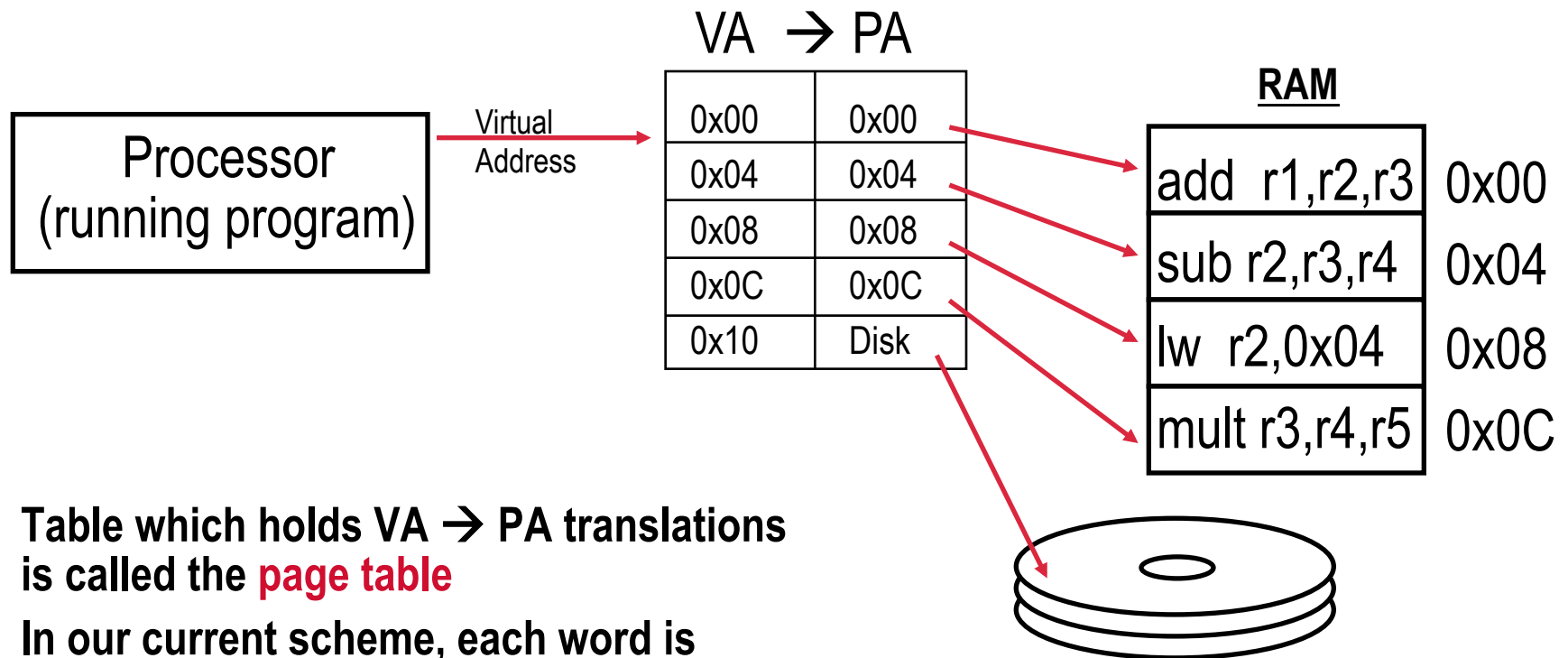
► On program startup

- ▷ OS copies program into DRAM
 - ▷ program relocates in pages
 - ▷ If there is not enough DRAM
 - ▷ OS stops copying program, starts running it
 - ▷ only portion of the program loaded in DRAM
 - ▷ When the program accesses memory
 - ▷ Translate virtual addr into physical addr
 - ▷ If accessed page not in physical memory (DRAM)
 - ▷ OS copies that part of the program from disk into DRAM (just like a cache)
 - ▷ In order to copy pages from disk to DRAM, OS must evict pages already in DRAM
 - ▷ OS writes back to disk the dirty evicted pages (just like a cache with writeback)
- 

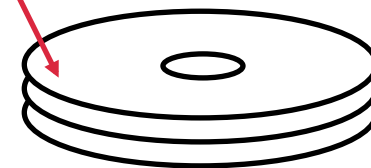


Basic VM Algorithm

- ▶ Program presents virtual address (load, store, instruction fetch)
- ▶ Computer translates **virtual addr. (VA)** to **physical addr. (PA)**
- ▶ Computer reads RAM using PA, returning the data to program

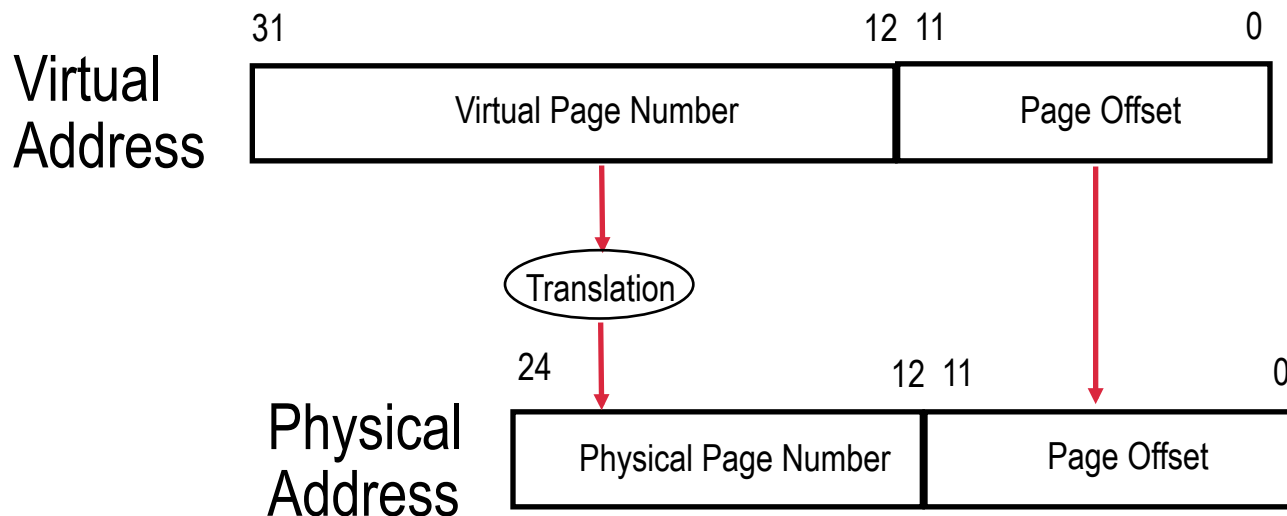


- ▶ Table which holds VA → PA translations is called the **page table**
- ▶ In our current scheme, each word is translated from a virtual address to a physical address
 - ▷ How big is the page table?

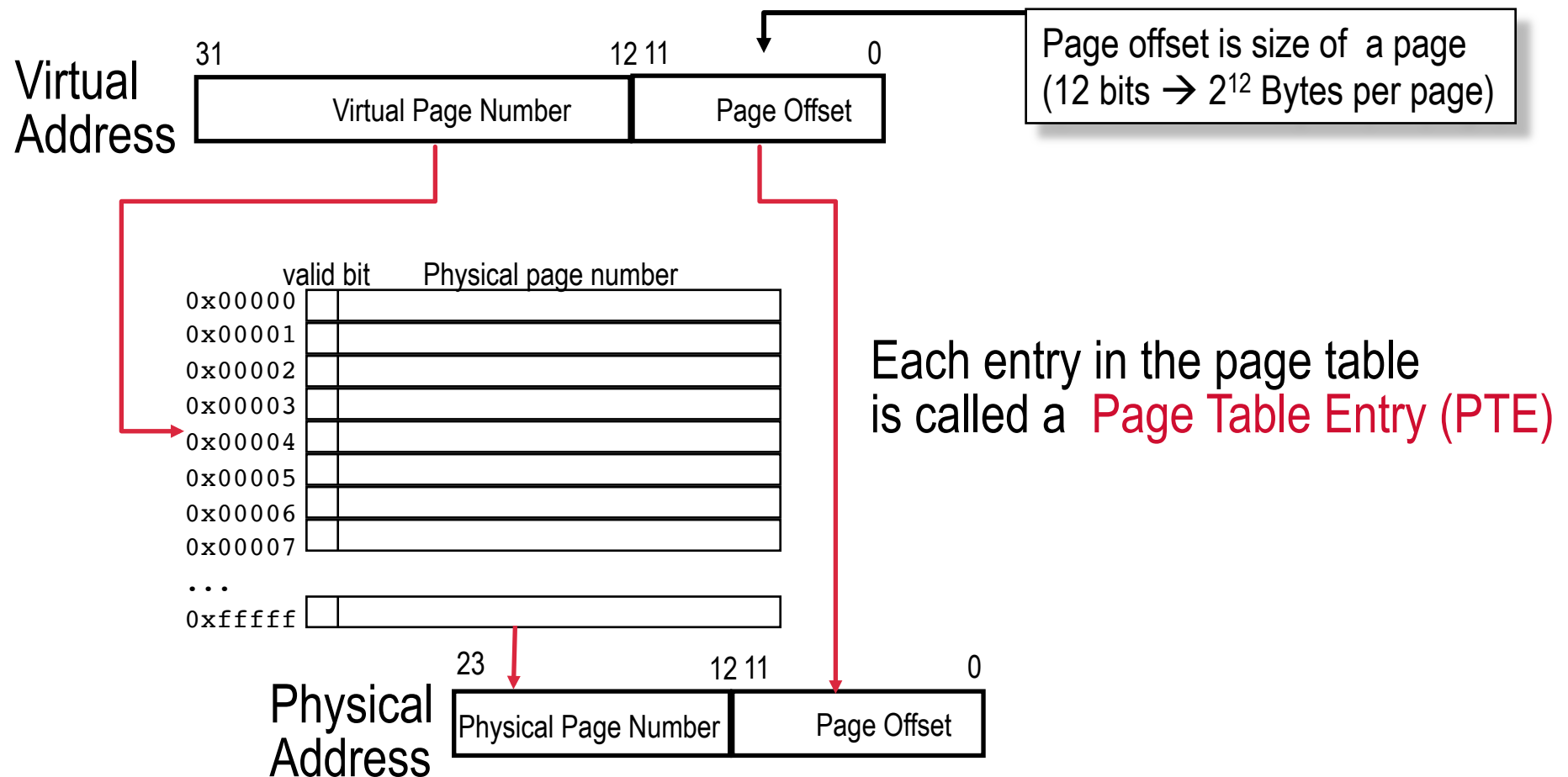


Page Tables (cont.)

- ▶ Instead of a “fine-grain” VM where any word in VM can map to any RAM word location, we partition memory into bigger chunks called **pages**
 - ▷ Typical page size today is 4 or 8 Kbytes
- ▶ Reduces the number of VA → PA translation entries
 - ▷ Only one translation per page
 - ▷ For 4 KByte page, that's one VA → PA translation for every 1,024 words
- ▶ Every address within a virtual page maps to the same location within a physical page frame
 - ▷ In other words, bottom $\log_2(\text{page size in bytes})$ addr bits **not** translated



Page Table



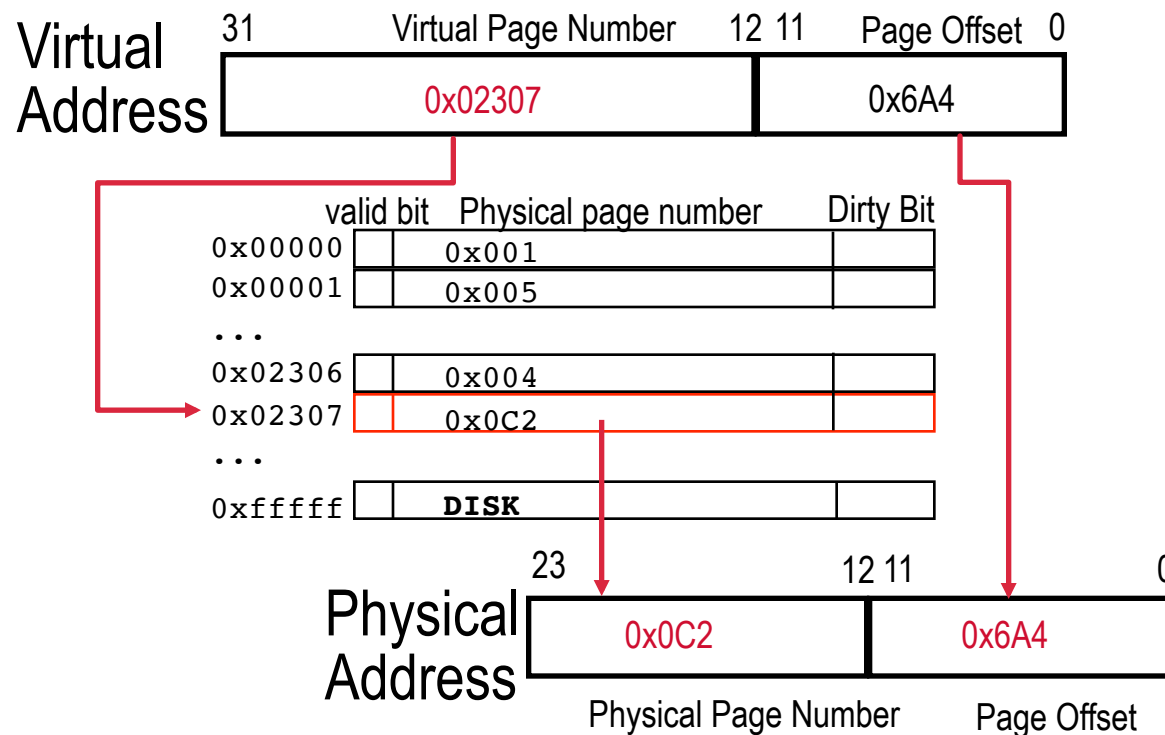
of bits in page offset = $\text{Log}_2(\text{Page Size})$
of bits in physical address = $\text{Log}_2(\text{\# RAM Bytes})$

Page Table Example

- 1) Break VA into virtual page number and page offset
- 2) Copy page offset to physical address
- 3) Use virtual page number as index into page table
 - 3a) check the valid bit
- 4) Copy physical page number from page table to the physical address

VA = 0x023076A4

PA = _____



What Happens if Page is not in RAM?

- ▶ **How do we know it's not in RAM?**
 - ▷ PTE's valid bit is set to **INVALID** (DISK)
- ▶ **What do we do?**
 - ▷ Hardware asks OS to fetch the page from disk - we call this a **page fault**
- ▶ **Before page is read from disk, OS must evict a page from RAM (if RAM is full)**
 - ▷ The page to be evicted is called the **victim page**
 - ▷ If the page to be evicted is dirty, write the page back to disk
 - ▷ Only data pages can be dirty
- ▶ **OS then reads the requested page from disk**
- ▶ **OS changes the page table to reflect the new mapping**
- ▶ **Hardware restarts at the faulting virtual address**

Which Page Should We Evict?

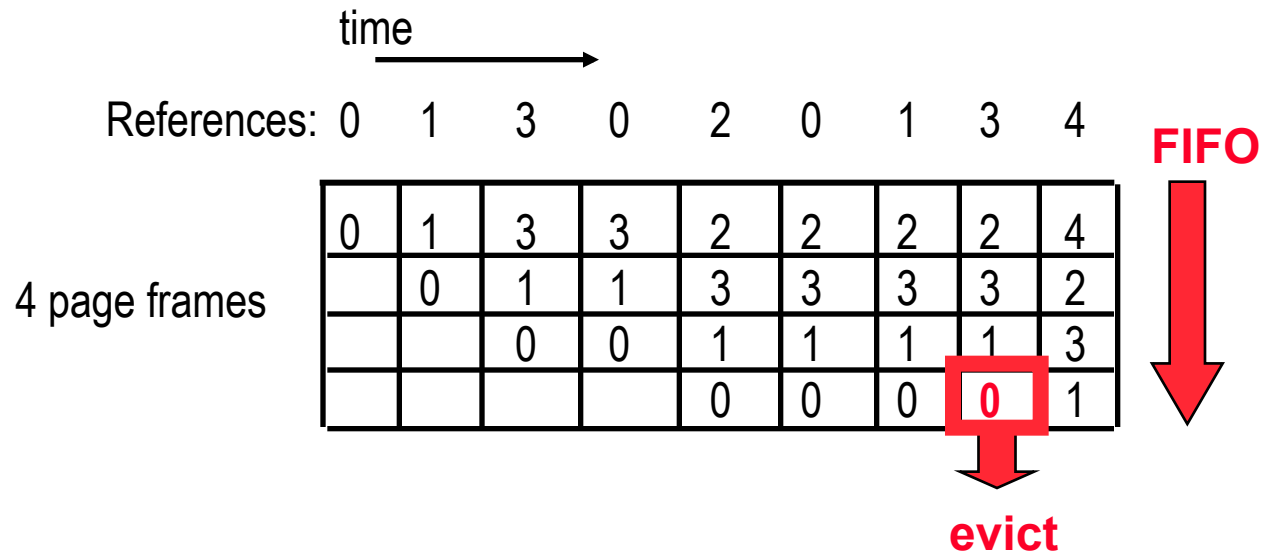
- ▶ **Optimal solution: evict a page that won't be referenced (used) again**
- ▶ **If all pages will be used again, then evict the page that will not be used for the longest period of time**
 - ▷ Guarantees the lowest possible page fault rate (# of faults per second)
 - ▷ Can't be done unless we can tell the future
- ▶ **Other page replacement algorithms**
 - ▷ First-in, First-out (FIFO)
 - ▷ Least Recently Used (LRU)

First-in, First-out (FIFO)

- ▶ Oldest page is evicted
- ▶ How do we know which page is oldest?
 - ▷ Keep a list of 1st-time references
 - ▷ When a page must be evicted, pick the page at the end (bottom) of the list
- ▶ Example w/ 4 pages

Page Reference Pattern

0, 1, 3, 0, 2, 0, 1, 3, 4

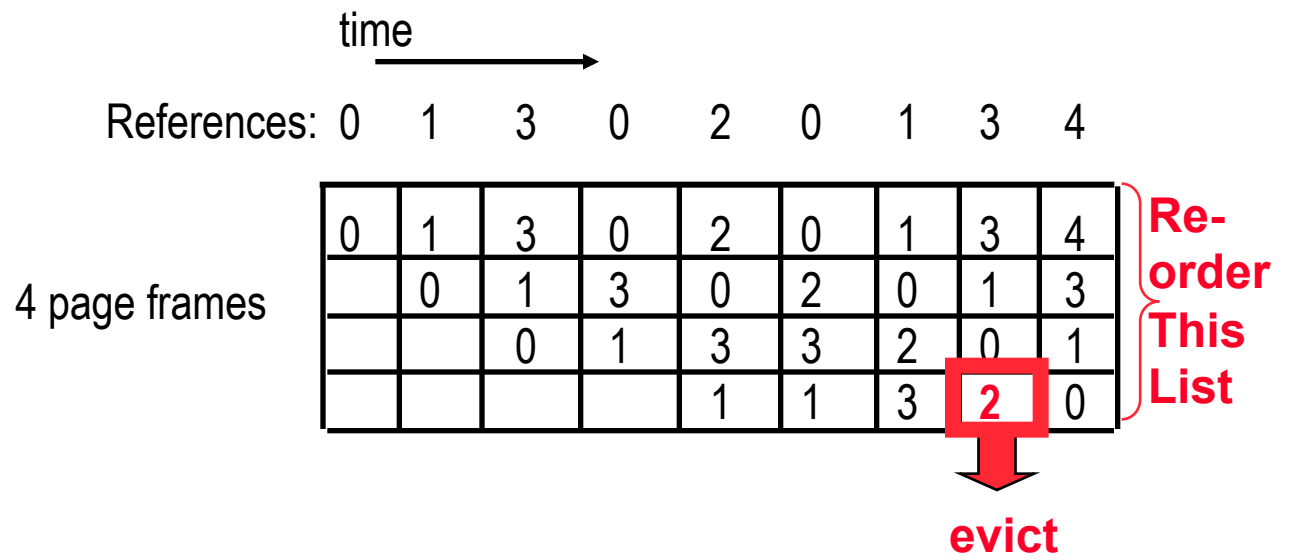


Least Recently Used (LRU)

- ▶ Evict the page that has not been used for the longest period of time
- ▶ How do we know which page is oldest?
 - ▷ Keep a list of pages ordered by reference
 - ▷ When a page must be evicted, pick the page at the end (bottom) of the list
- ▶ Example w/ 4 pages

Page Reference Pattern

0, 1, 3, 0, 2, 0, 1, 3, 4



Performance of Virtual Memory

- ▶ If every program in a multiprogramming environment fits into RAM, then virtual memory *never* “pages” (goes to disk)
- ▶ If any program doesn’t fit into RAM, then the VM system *must* page between RAM and disk
 - ▷ Paging is very costly
 - ▷ A disk access (4KBytes) can take ~10 ms ... in 10 ms, a processor can execute ~20 Million instructions
 - ▷ Basically, you really don’t want to page very often, if you don’t have to

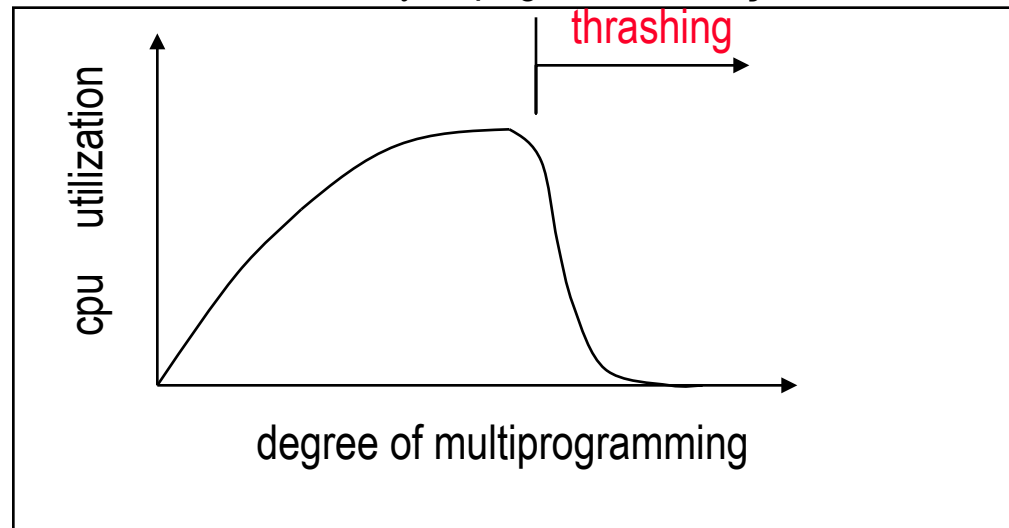
When Bad Things Happen to Good Memories...

► So, can anything go wrong here?

- ▷ ...i.e., this sounds like a great idea. Have reasonable size RAMs, big disks.
- ▷ What bad stuff can happen?

► Answer: **thrashing**

- ▷ Your program is so large, or, your machine has so many separate jobs (or users), that very little of the program(s) fit in RAM.
- ▷ Your physical RAM is not providing you even minimal spatial locality for your address refs, so very very often, an address faults, and you page **constantly**

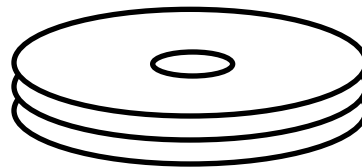


Pathological Thrashing

► **Consider the following reference pattern (in pages)**

- ▷ 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5
- ▷ What happens if there are only 4 page frames in RAM?
- ▷ Not all of the program will fit into RAM
- ▷ Assume LRU

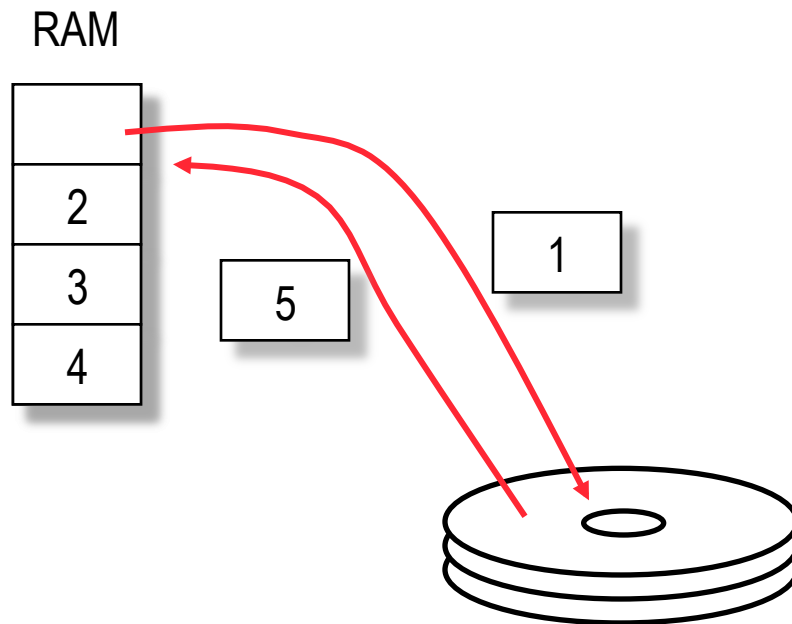
RAM



Pathological Thrashing (cont.)

► Consider the following reference pattern (in pages)

- ▷ 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5
- ▷ What happens if there are only 4 page frames in RAM?
- ▷ Not all of the program will fit into RAM
- ▷ Assume LRU

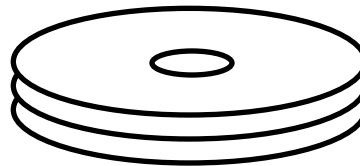
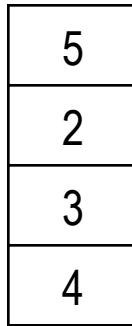


Pathological Thrashing (cont.)

► **Consider the following reference pattern (in pages)**

- ▷ 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5
- ▷ What happens if there are only 4 page frames in RAM?
- ▷ Not all of the program will fit into RAM
- ▷ Assume LRU

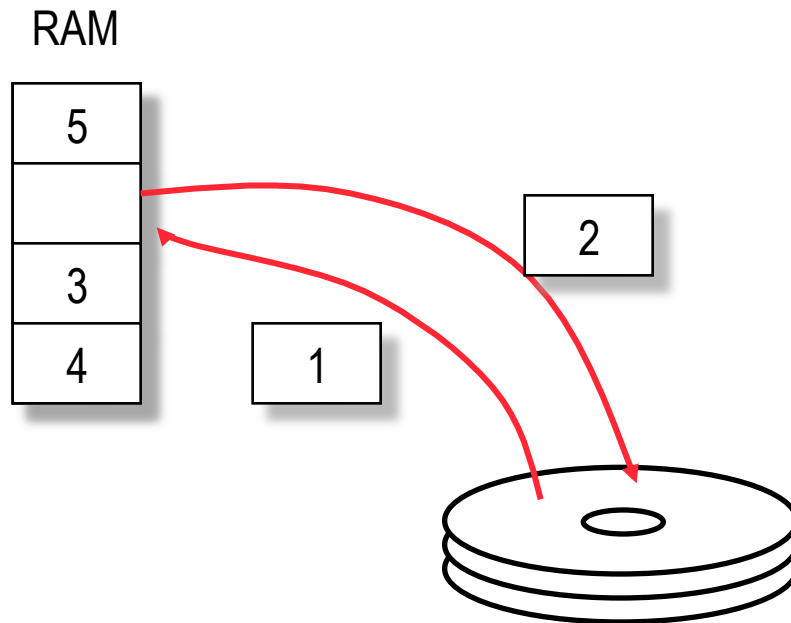
RAM



Pathological Thrashing (cont.)

► Consider the following reference pattern (in pages)

- ▷ 1, 2, 3, 4, 5, **1**, 2, 3, 4, 5, 1, 2, 3, 4, 5
- ▷ What happens if there are only 4 page frames in RAM?
- ▷ Not all of the program will fit into RAM
- ▷ Assume LRU



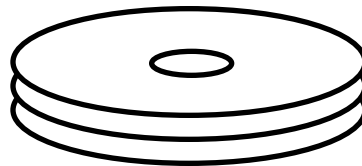
Pathological Thrashing (cont.)

► **Consider the following reference pattern (in pages)**

- ▷ 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5
- ▷ What happens if there are only 4 page frames in RAM?
- ▷ Not all of the program will fit into RAM
- ▷ LRU can result in thrashing ... Random replacement or some other policy could avoid this problem

RAM

5
1
3
4



Next Issue: Page Table Size

► Page table size: Linux on Intel Core i7

- ▷ 4K page
- ▷ 36-bit physical address space $\rightarrow 36 - 12 = 24$ bits physical page number
- ▷ 48-bit virtual address space (architecture defines 64 bits, current implementations use 48)
- ▷ 36 bits for the virtual page number $\rightarrow 2^{36}$ virtual pages
 - ▷ $2^{36} = 68,719,476,736$ PTEs
- ▷ Each PTE is at least 4 Bytes (24 bits PPN + 1 bit valid + 1 bit dirty = 26 bits)
 - ▷ 2^{36} PTEs * 4 Bytes/PTE = **256 GBytes**

► Hey: you said each process has its own page table!

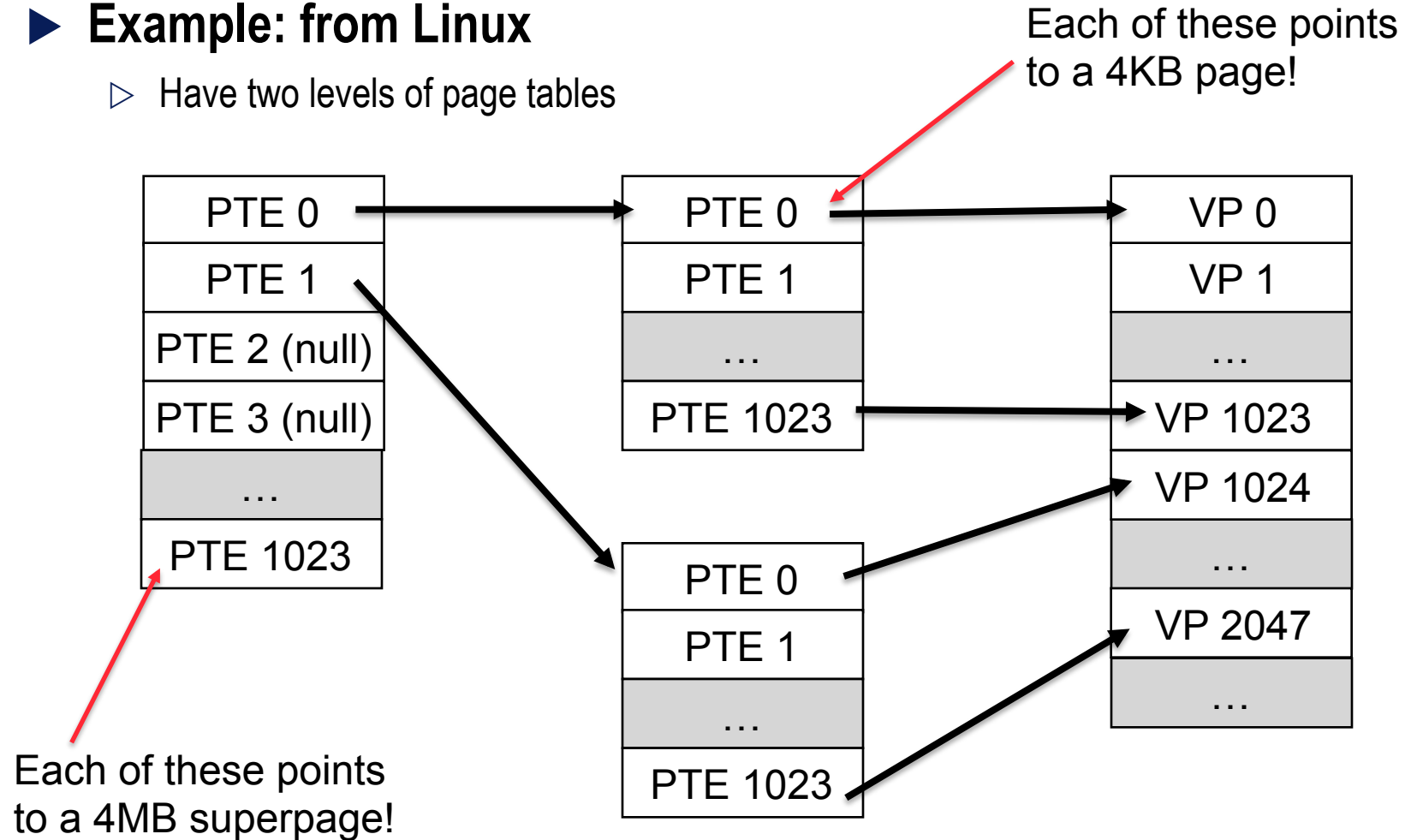
- ▷ 100 processes = **25 TBytes** of memory for page tables!!!!
- ▷ If the page table is not in memory, you can't find it
 - ▷ The page table cannot be swapped out!

► What's the deal?

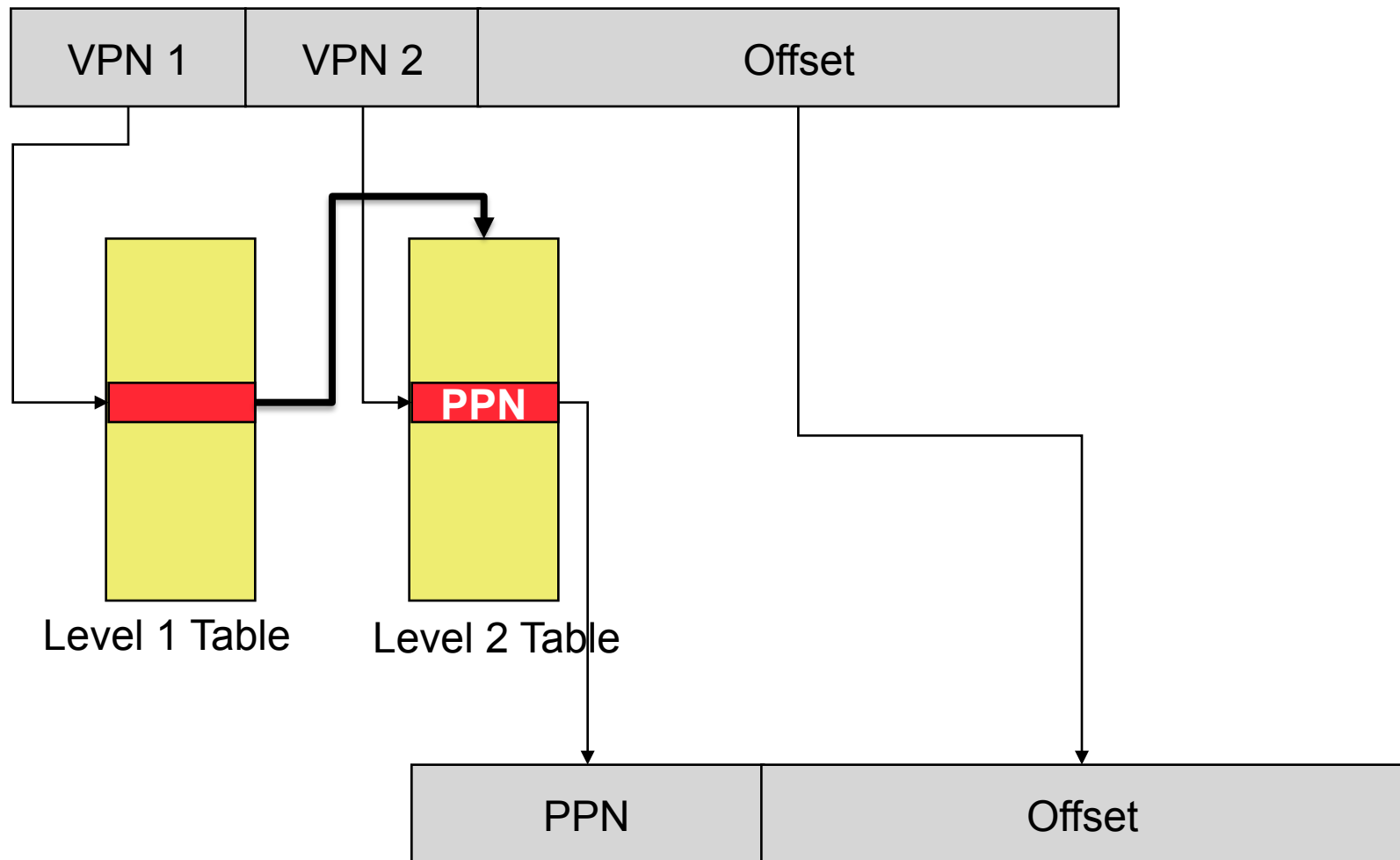
Solution: Multi-level page tables

► Example: from Linux

- ▷ Have two levels of page tables



Multi-level Page Tables



Multi-level Advantages

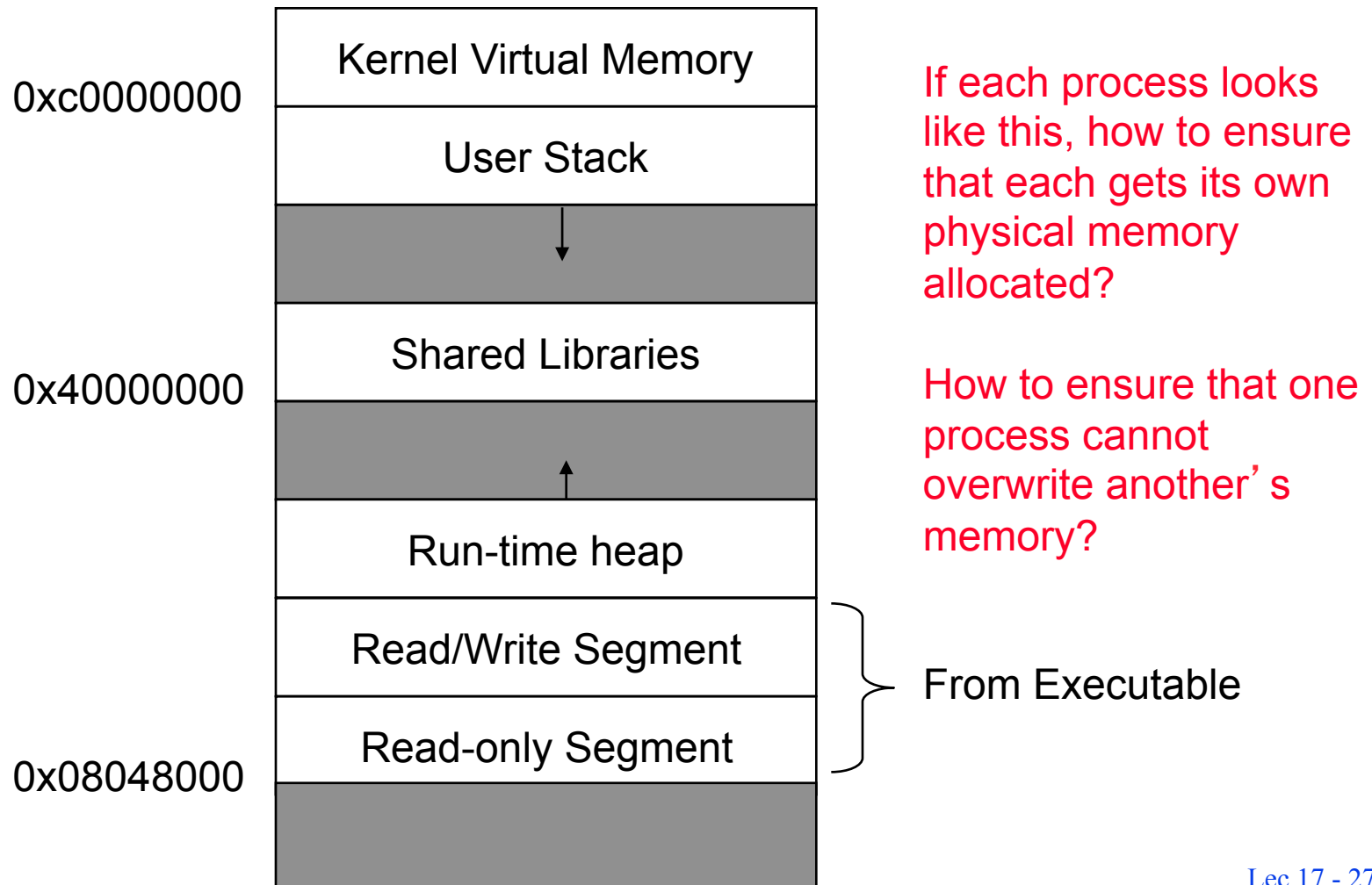
- ▶ **Page table size scales with memory usage**
- ▶ **Only level 1 MUST be in real memory**
 - ▷ Level 2 tables can be swapped just like any other page
 - ▷ Decrease page table's memory footprint
- ▶ **Multi-level and TLB integration**
 - ▷ TLB only stores Level 2 PTEs
 - ▷ Don't have to do two TLB accesses to do VA → PA translation

Multiprogramming and Protection

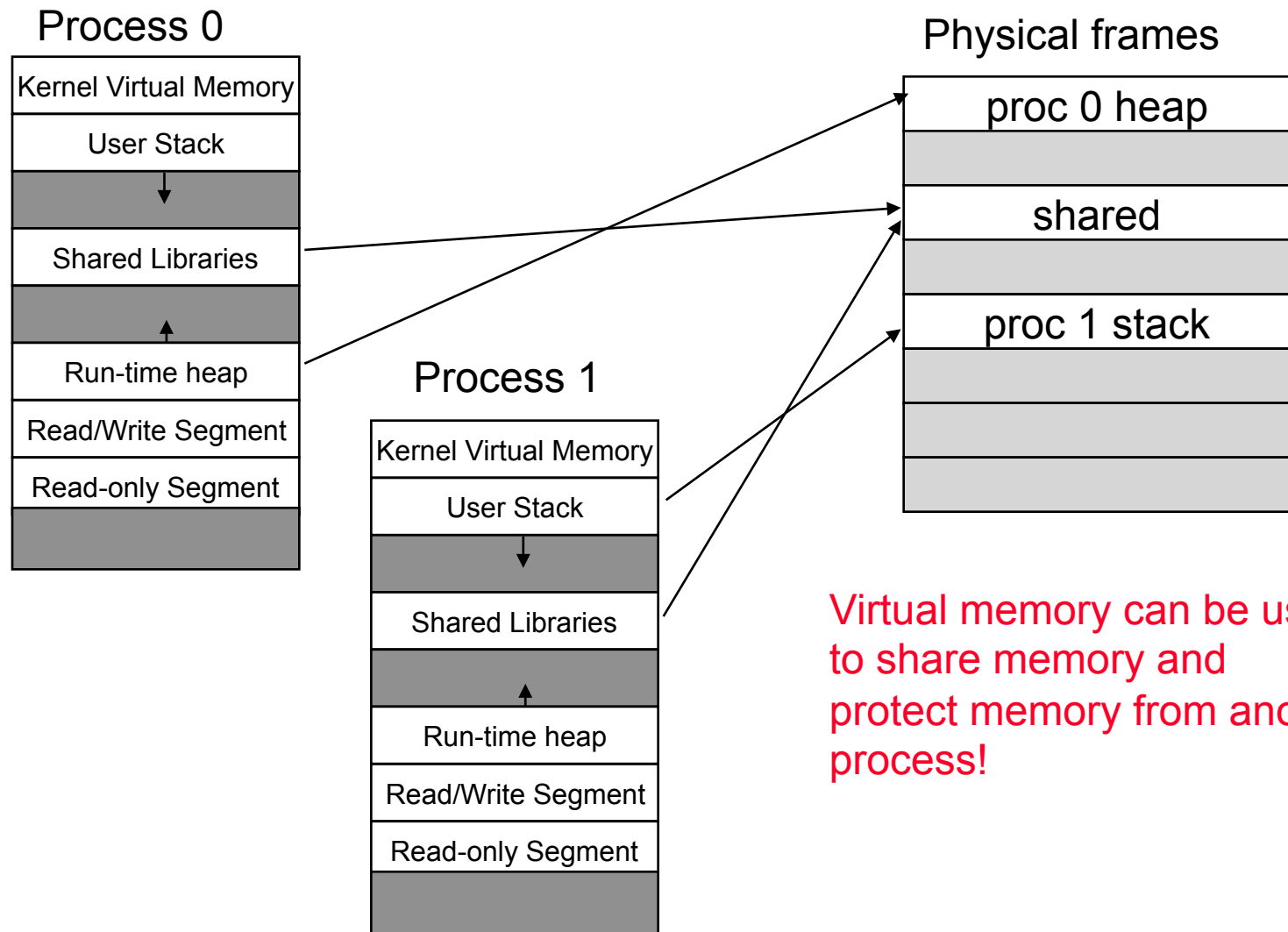
- ▶ **Most machines run multiple *processes* (also called *jobs*, *tasks*)**
 - ▷ The processes share RAM
 - ▷ Example: simultaneously running emacs, verilog and X-windows system
 - ▷ Example: simultaneously running IExplorer, WORD, Quake
- ▶ **Need mechanism to provide the memory to each process**
- ▶ **Need mechanism to protect the memory of each process**
 - ▷ Avoid one process reading or writing the memory of another
 - ▷ Avoid one process trashing another, especially the OS
 - ▷ Usually referred to as “**protection**”
- ▶ **If access type not compatible with access rights**
 - ▷ Protection-violation fault: trap to hardware, microcode, or software fault handler
 - ▷ Also protect your own pages: read-only, read-write, execute only permissions

VM and Memory Management

- This is how EACH process in Linux sees memory:



Mapping to Physical Memory



Protection with Private Page Table

- ▶ Give each user process its own page table
- ▶ Because every address necessarily goes thru VA → PA translation, the process can *only* access physical pages listed in its own page table
- ▶ OS guarantees that no user-level process can alter any process's page table
 - ▷ Page tables are mapped into *kernel memory* where only the OS can read or write

	valid bit	Physical page number
0x00000		0x001
0x00001		0x005
0x00002		0x00A
0x00003		0x004
0x00004		0x008

Process 1's Page Table

	valid bit	Physical page number
0x00000		0x002
0x00001		0x006
0x00002		0x00B
0x00003		0x003
0x00004		0x009

Process 2's Page Table

0x00	
0x01	P1
0x02	P2
0x03	P2
0x04	P1
0x05	P1
0x06	P2
0x07	
0x08	P1
0x09	P2
0x0A	P1
0x0B	P2

VM Support

- ▶ **Each process has its own page table**
- ▶ **Extra bits to allow/disallow**
 - ▷ Read access
 - ▷ Write access
 - ▷ Access when in supervisor mode only (kernel code)
- ▶ **Support shared pages (libraries or memory)**
- ▶ **If a process accesses a memory location that it does not have proper access to:**

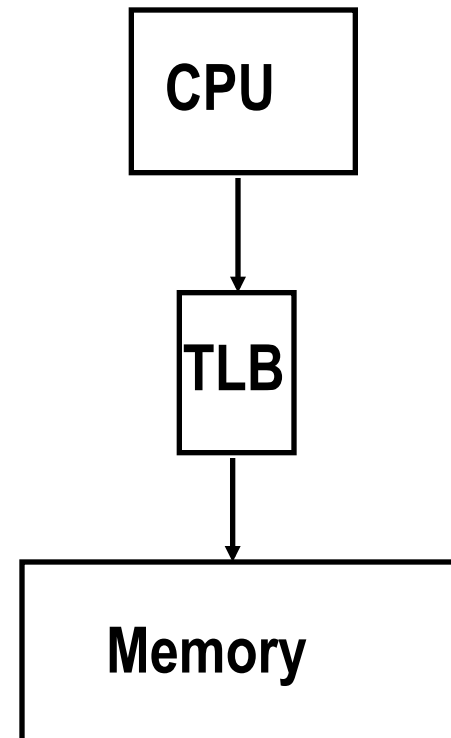
Segmentation Fault

Who Does the VA → PA Translation?

- ▶ **Software would be too slow for every memory reference**
 - ▷ If not in software, then must be done in hardware
 - ▷ **Hardware needs to hold the translations (PTEs)**
- ▶ **Can hardware hold **all** of the PTEs (i.e., the entire page table)?**
 - ▷ Too large to hold entire page table *inside* CPU chip itself
 - ▷ Most processors “**cache**” the most recently used PTEs.... **Where?**

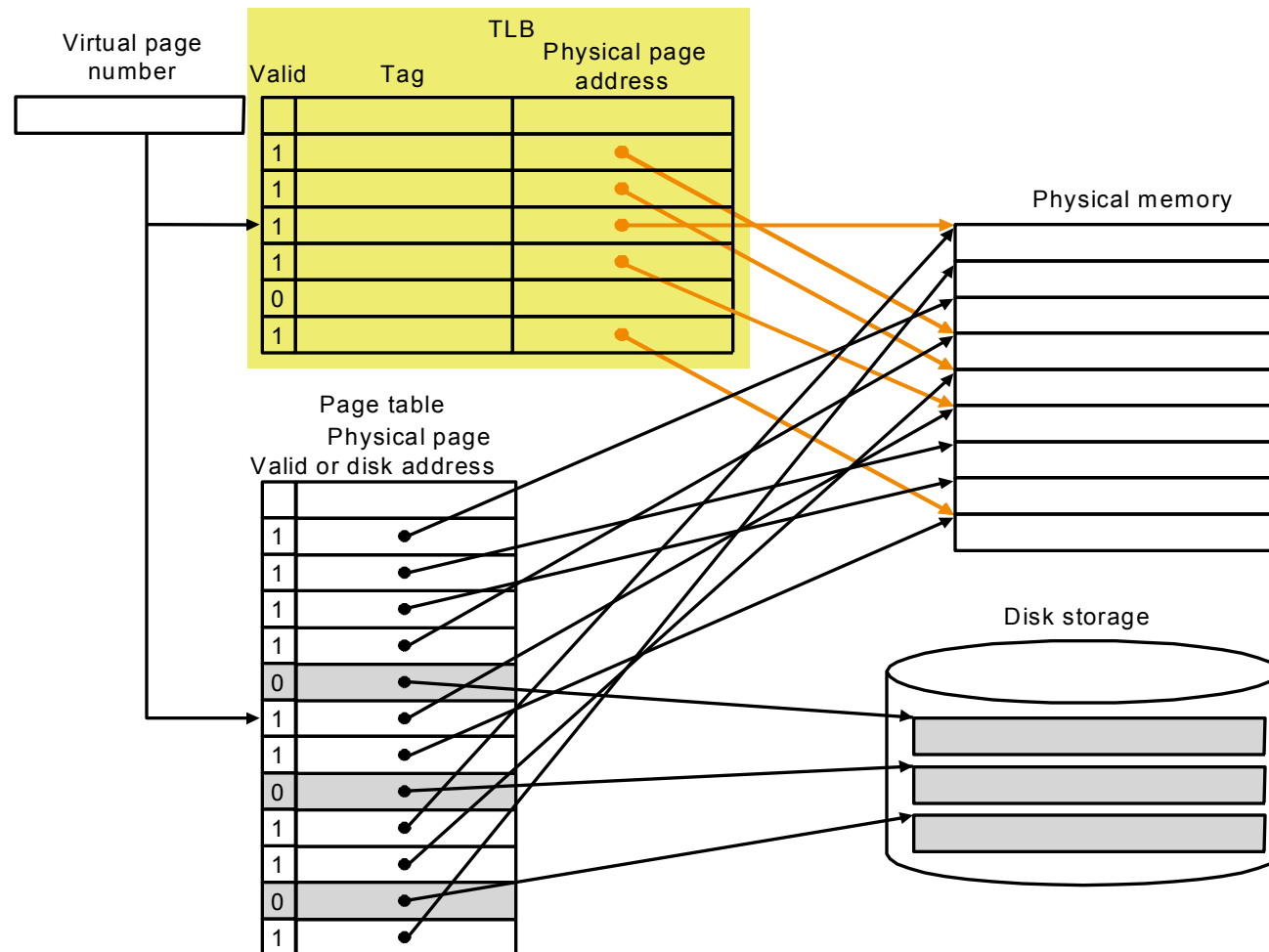
Where? Translation Lookaside Buffer (TLB)

- ▶ **TLB Caches most recently referenced PTEs**
 - ▷ Very similar to instruction and data caches
- ▶ **Can be accessed in \leq single cycle**



Making Address Translation Fast

- A cache for address translations: **translation lookaside buffer**



Where is My Page? My Page Table Entry?

▶ Page in physical mem, PTE in TLB

- ▷ Life is good. Best answer. TLB lookup hits, VA → PA fastest, address resolved

▶ Page in physical mem, PTE **not** in TLB, PTE only in phys mem

- ▷ Life is OK. TLB lookup misses. TLB reloads from page table out in memory. (20-500 cycles)
VA → PA takes longer, but at least your address is already in physical memory

▶ Page not in physical mem, PTE in TLB

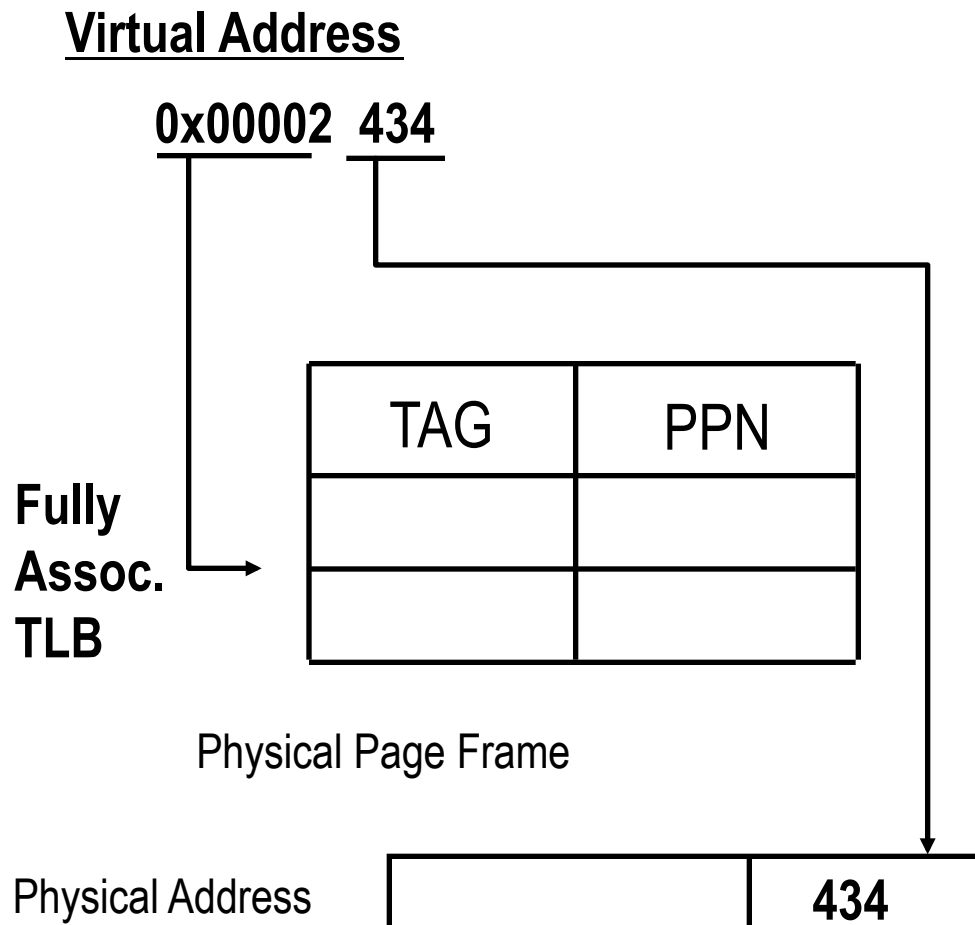
- ▷ Life stinks. TLB hits, quickly tells you your address is not in physical memory. Page fault, you wait to go load the page.

▶ Page **not** in phys mem, PTE **not** in TLB, PTE only in phys mem

- ▷ Life sucks. TLB lookup misses. TLB reloads from page table out in memory. During that reload, you discover your page itself is not in physical memory. Page fault, you wait to go load the page.

TLB Example: Address Ref 1

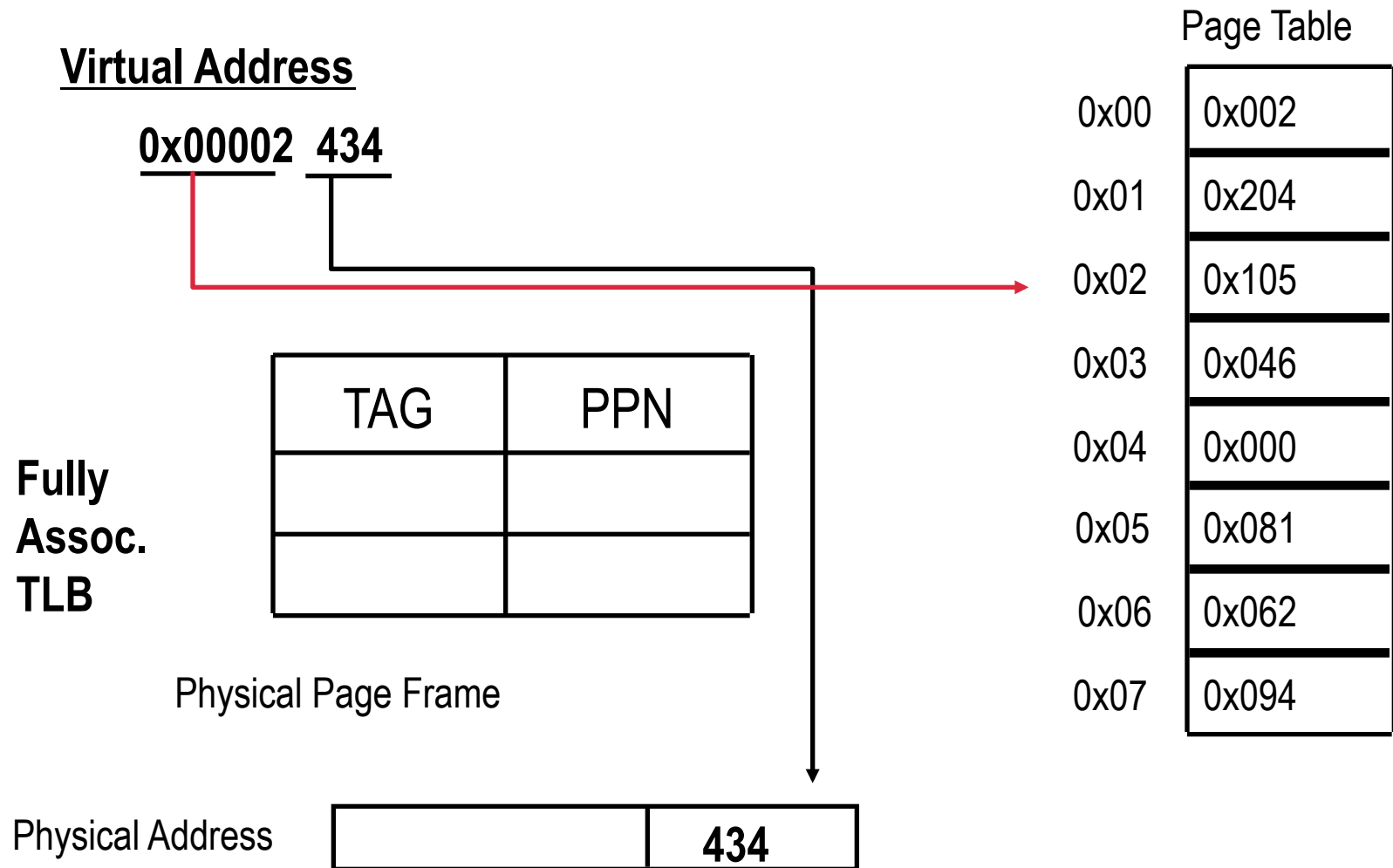
Assume a 4K-Byte pages and a 32 bit address space



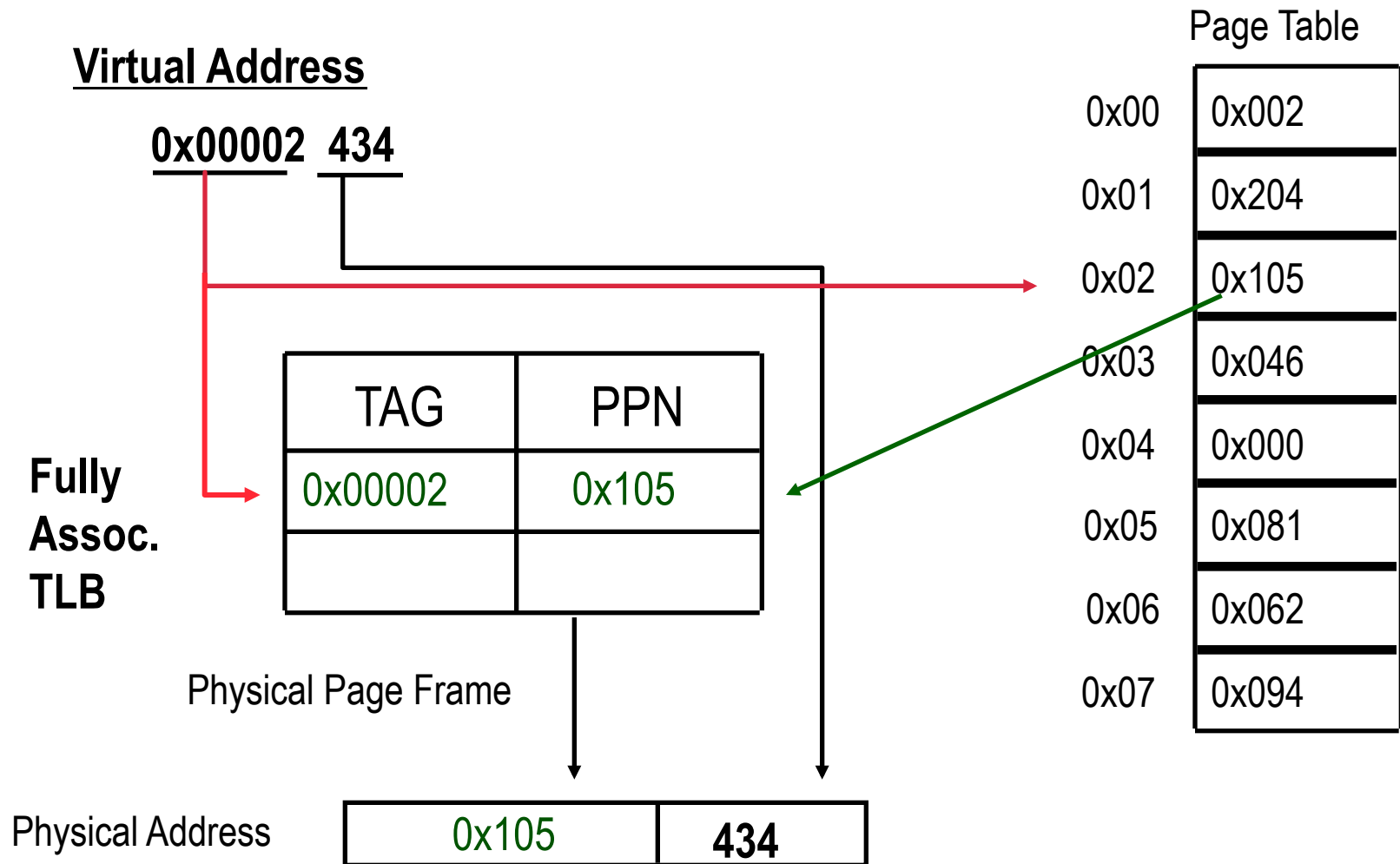
Page Table

0x00	0x002
0x01	0x204
0x02	0x105
0x03	0x046
0x04	0x000
0x05	0x081
0x06	0x062
0x07	0x094

TLB Example: Address Ref 1 Misses in TLB



TLB Example: Address Ref 1 Misses in TLB



TLB Structure & Performance

▶ TLB structure

- ▷ 32 to 4K entries (slots)
- ▷ Can be direct mapped, set associate, fully associative
- ▷ Easier to be fully-associative here, as TLBs are often pretty small

▶ TLB miss cost

- ▷ 20 to 500 cycles
- ▷ Hardware and software based

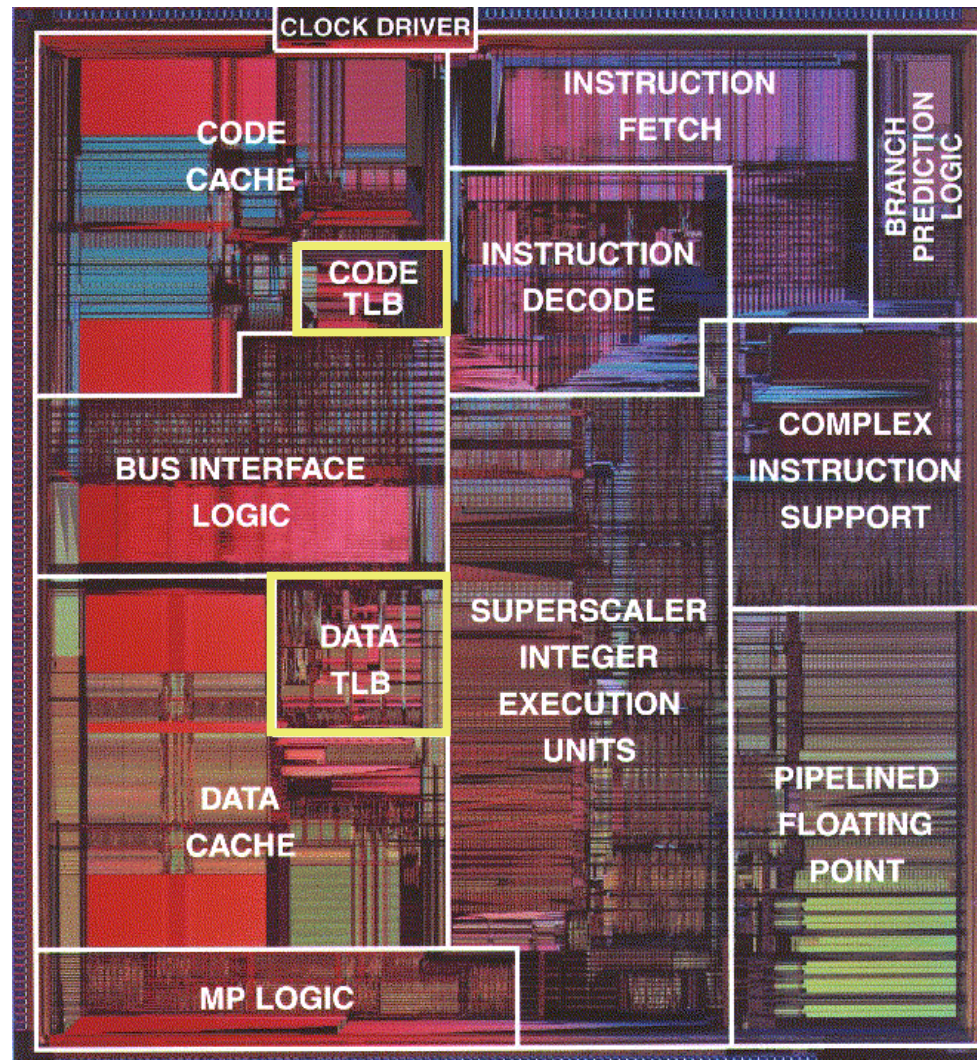
▶ TLB miss rates

- ▷ 4% to 8% for typical Unix workloads
- ▷ Can be much higher for large applications
- ▷ TLB can't be too big (need to access in $\ll 1$ cycle). Thus, the TLB reach is low (4K entries of 4KB pages \rightarrow TLB can reach 16 MB). But today's memories > 2 GB.
- ▷ Operating systems can significantly influence the TLB miss rate

▶ Page size

- ▷ 4K or 8K Bytes (often support multiple page sizes—up to 1 Gbyte today)
- ▷ Why? To increase the TLB reach

Real Life Examples: Pentium Pro



Next Issue: Cache & TLBs, How They Interact

- ▶ **We do memory hierarchies to hide unpleasant facts**

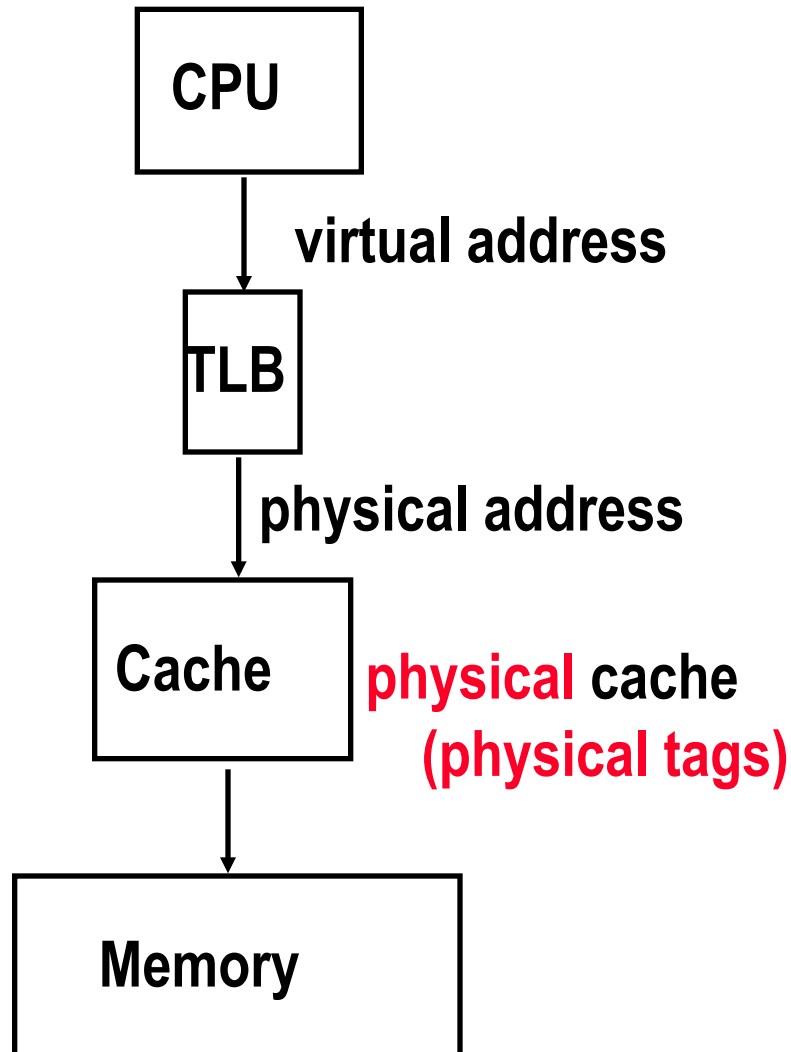
- ▷ We don't have as much fast memory as we would ideally like. So, cache hierarchy gives illusion of speed--most of the time. Occasionally it's slow.
- ▷ We don't have as much randomly accessible memory as we would like. So, VM hierarchy gives the illusion of size--most of the time. Occasionally it's slow.

- ▶ **Roughly put: *We do cache for speed. We do VM for size.***

- ▶ **So, we have “regular” cache for fast access, and we have a TLB for fast translation $VA \rightarrow PA$.**

- ▷ How do they interact? They *must* interact somehow...
- ▷ Do we wait for $VA \rightarrow PA$ translation before looking in the cache?
- ▷ Is the cache full of virtual or physical addresses?

Simplest Scheme is Sequential: TLB then Cache

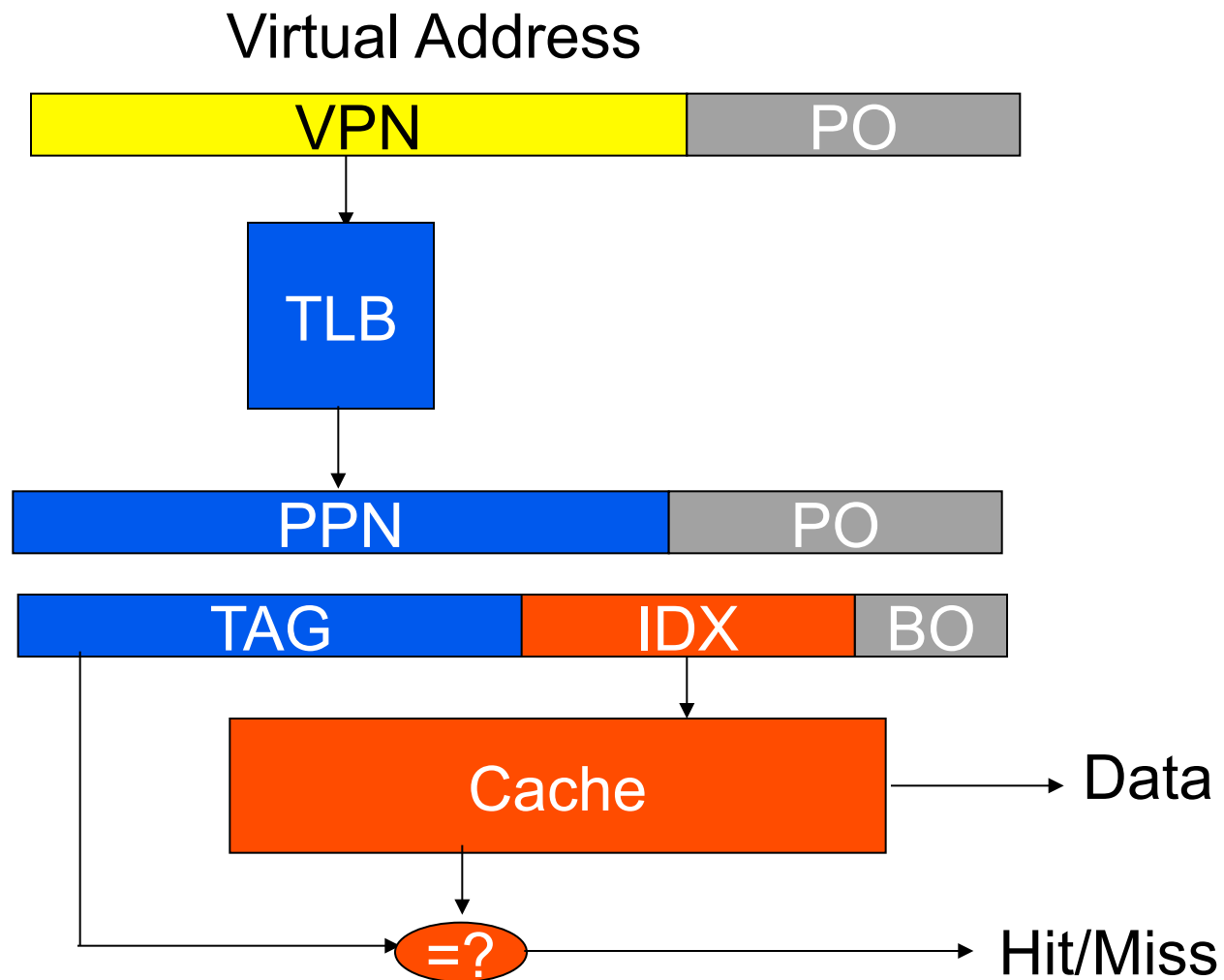


► Slowest, but simplest

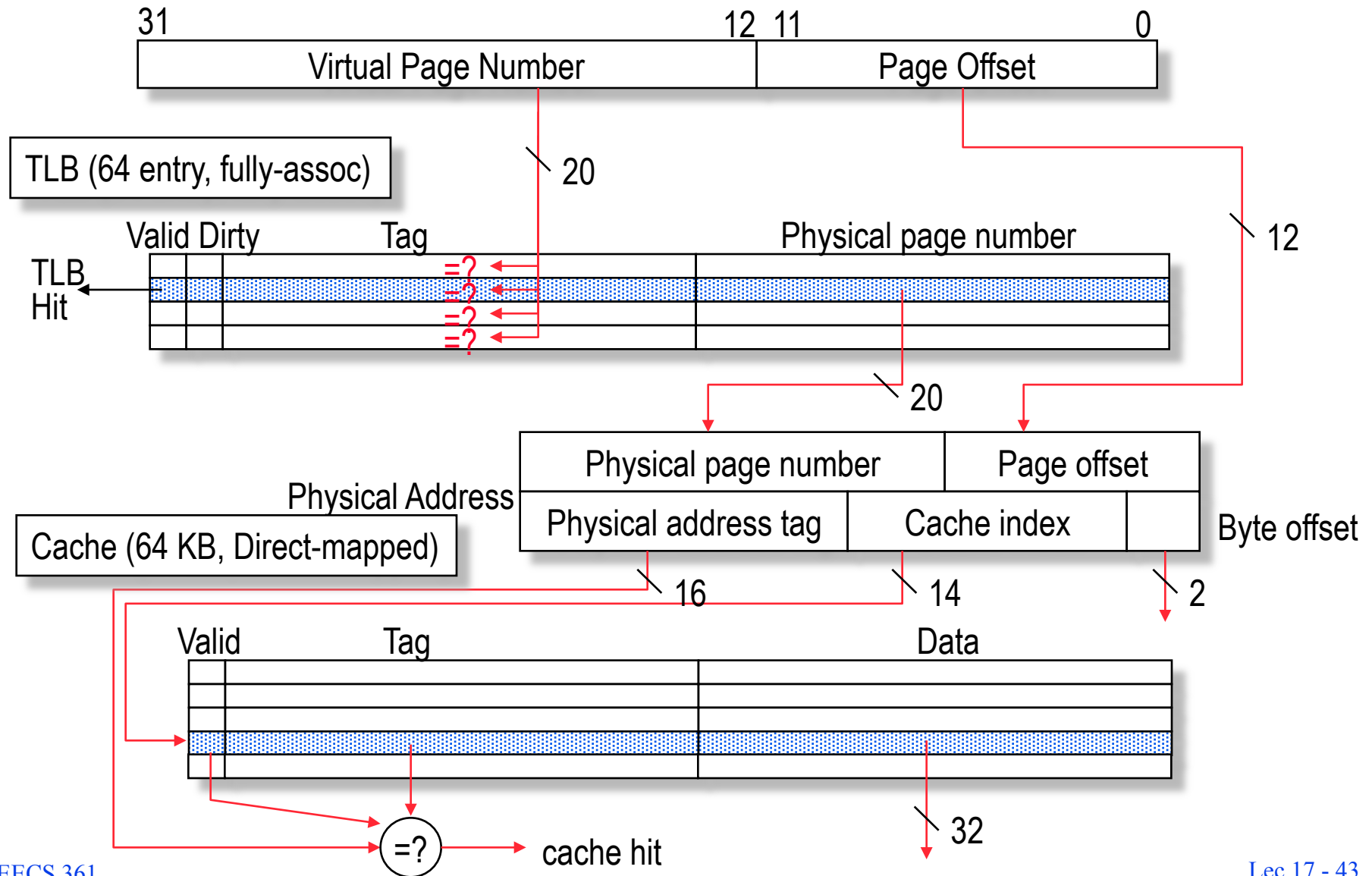
- ▷ CPU sends out virtual address
- ▷ TLB translates to physical, or page faults and we wait for page to load
- ▷ On TLB hit, translated physical address sent to cache
- ▷ Cache lookup gives data access fast, or...
- ▷ Cache miss goes to main mem

Address Translation/Cache Lookup

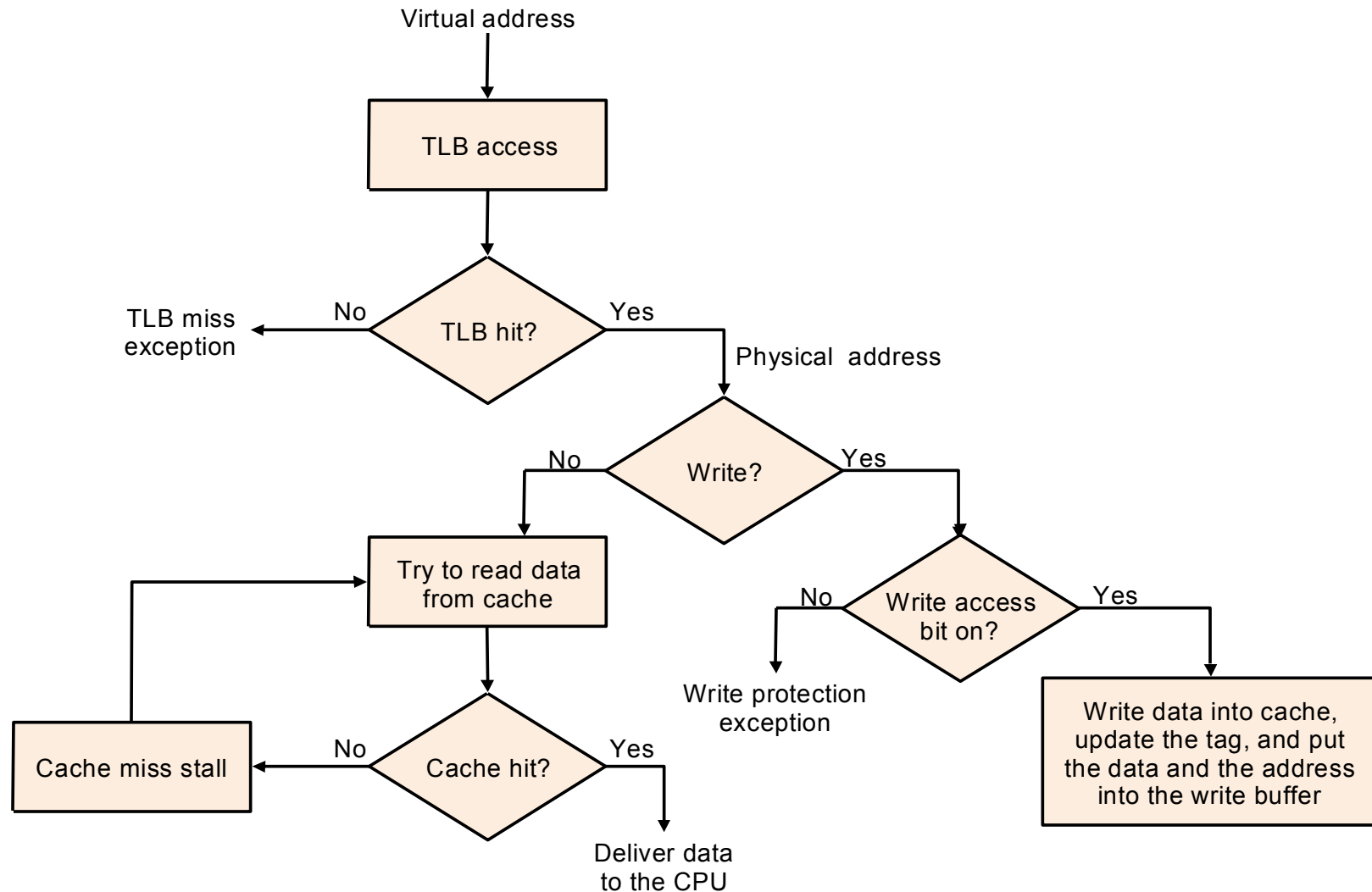
► Simple!



Real Example: DECstation 3100

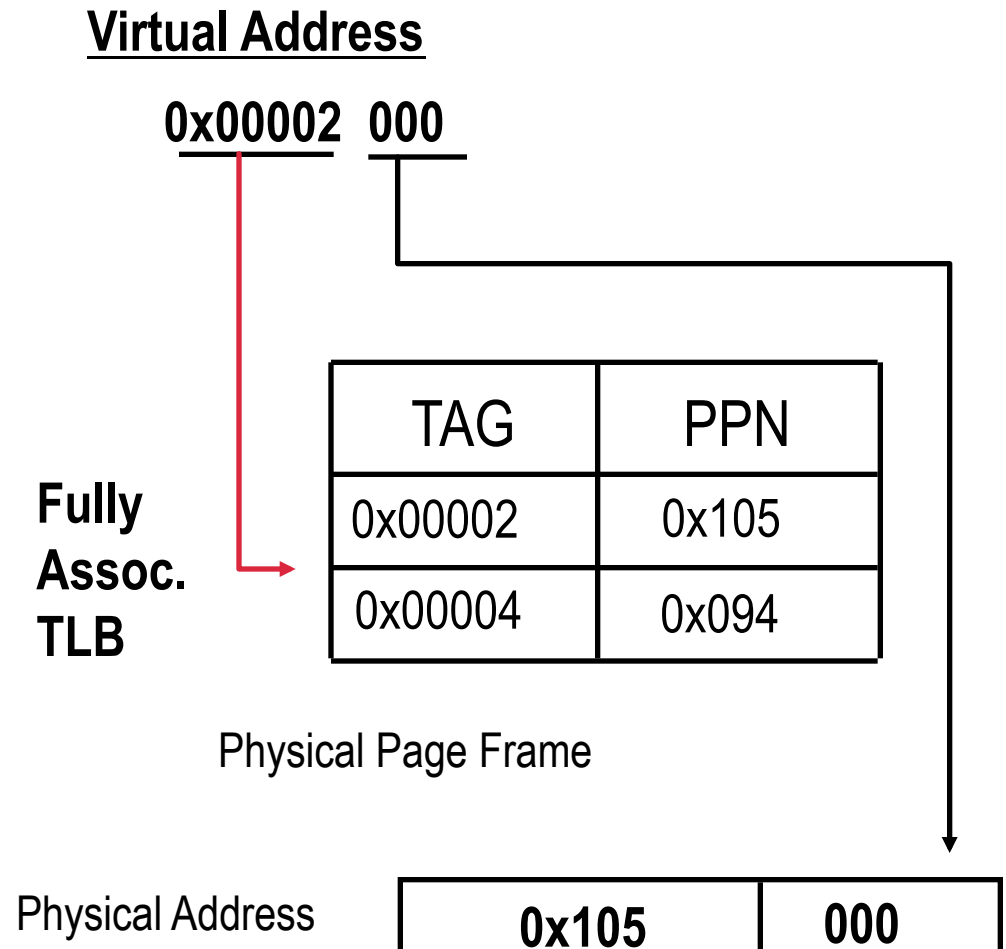


TLBs and Caches: Basic Flow for Access



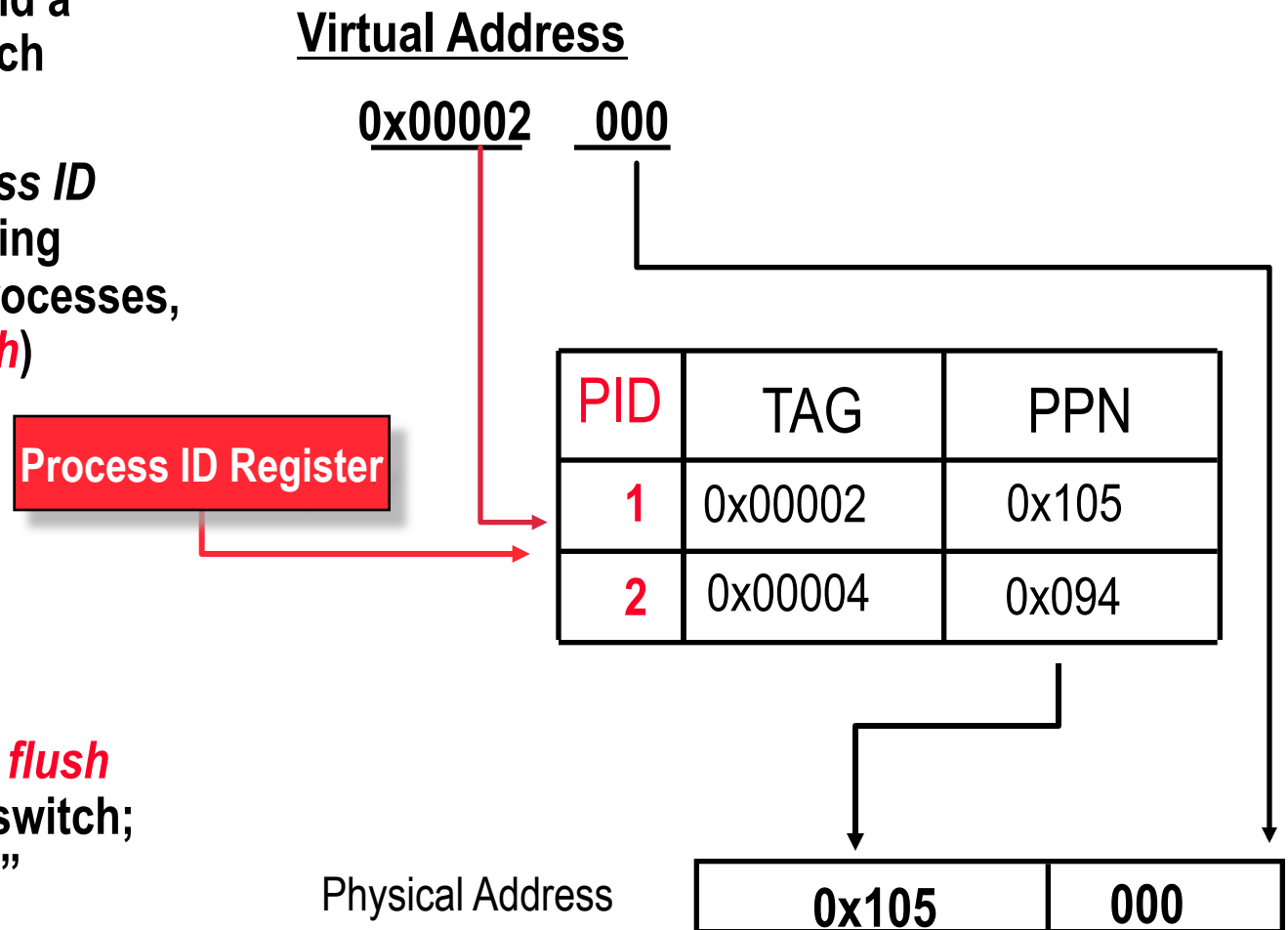
Protection and the TLB

- ▶ A process presents a Virtual Address to the TLB for translation
- ▶ Ex: either process could present virtual address 0x2000
- ▶ *How does the TLB know which process's VA → PA mapping is held in the TLB?*
 - ▷ *There are 2 separate VM address spaces here, one per process*
 - ▷ *There is only 1 TLB, NOT one per process*



Protection and the TLB (cont.)

- ▶ Many machines append a PID (process ID) to each TLB entry
- ▶ OS maintains a *Process ID Register* (updated during the switch between processes, called a *context switch*)
- ▶ Another solution is to *flush* the TLB on a context switch; flush means “empty it”



Speed & Timing Impacts

► If we do these accesses sequentially, big impact on speed

- ▷ You have to do lookup in the TLB...
- ▷ ...then you have to do lookup in the cache
- ▷ Involves a lot of memory access time

► Solution: pipelining

- ▷ Spread the accesses across stages of the pipeline
- ▷ TLB and cache are just like any other resource in the pipeline
 - ▷ You gotta be careful to know how long they take (impacts pipe cycle time)
 - ▷ You gotta know who is trying use the resource in what pipe stage
 - ▷ You can have hazards, need stalls, need forwarding paths, etc

Speeding it Up

▶ TLB and then Cache... Why?

Two options

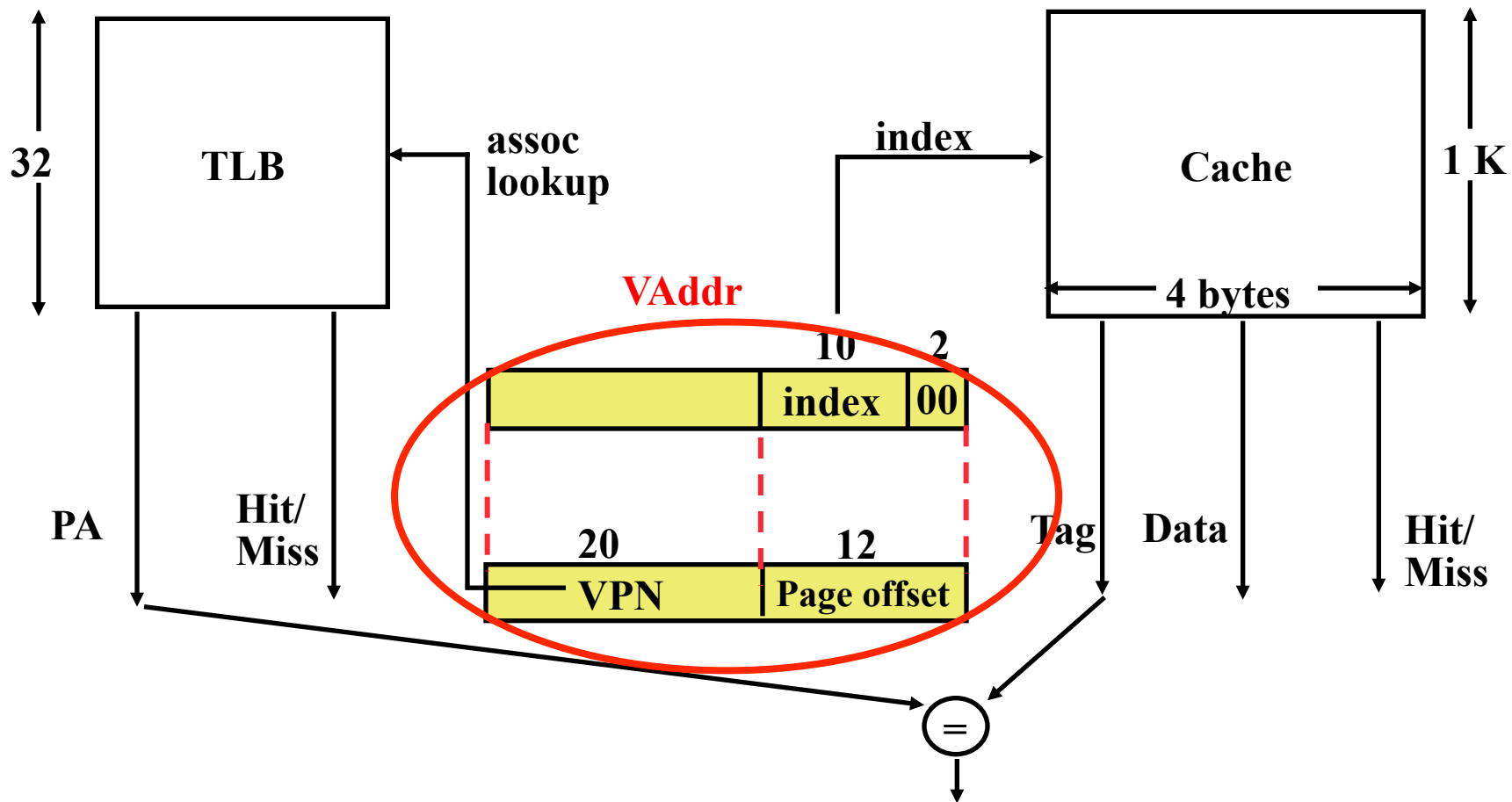
1. Overlap cache & TLB access (done in parallel)

- ▶ What are the limitations?

2. Why does our cache use physical addresses?

- ▷ Could it store virtual addresses?
- ▷ What are the problems/considerations?

Overlapped Cache & TLB Access



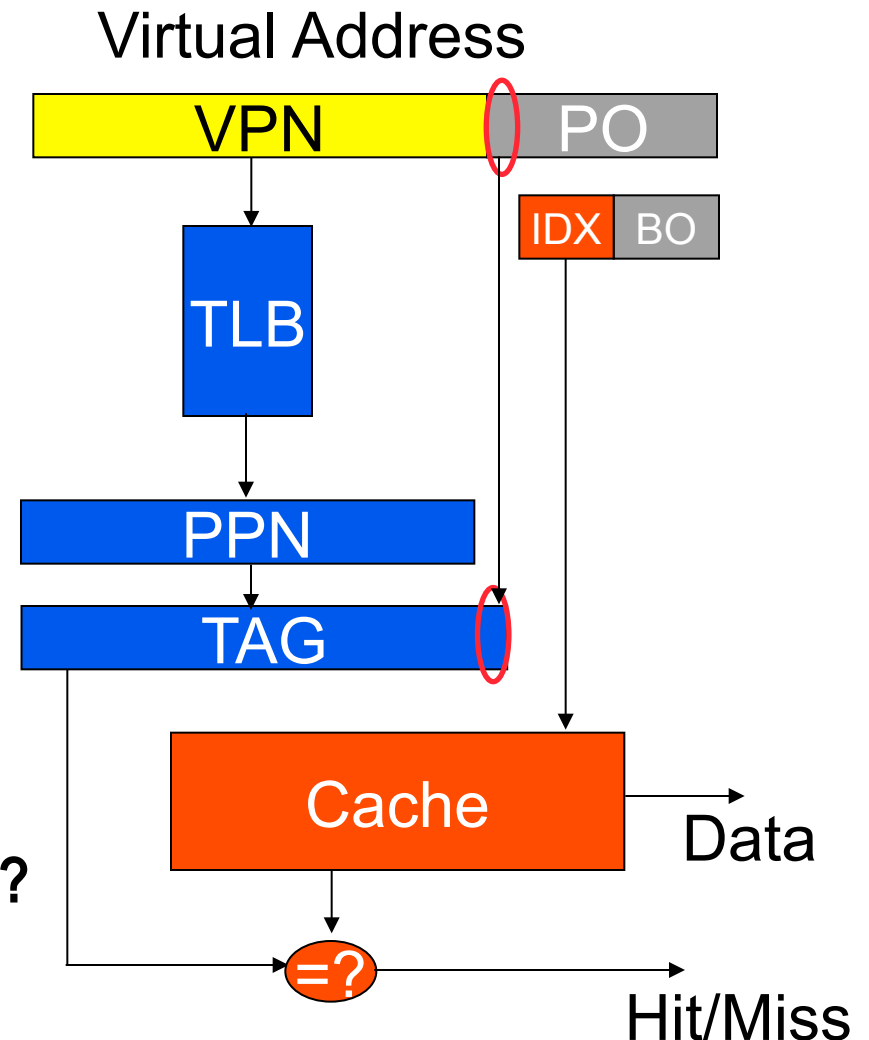
IF (cache hit) AND (cache tag = PA) THEN deliver data to CPU
ELSE IF [cache miss OR (cache tag != PA)] AND (TLB hit) THEN
 access memory with the PA from the TLB
ELSE do standard VA translation

How Overlapping Reduces Translation Time

- ▶ **Basic plumbing**
- ▶ **High order-bits of the VA are used to look in the TLB...**
 - ▷ Remember: low order bits are the page offset—denote which byte within the page
 - ▷ High order bits are what really changes, from virtual to physical address
- ▶ **...while low-order bits are used as index into cache**
 - ▷ Remember: lowest bits are the cache line offset—denote which byte within the cache line
 - ▷ The “intermediate” bits are the cache index: which line in the cache should we check to see if the data at the address we want are actually loaded into the cache
 - ▷ The highest order bits are the cache tag: when we look in the cache at a line of cache, we must compare these bits with the cache tag, to see if what’s stored in the cache is really the address we want, or just another line of memory that happens to map to this same place in the cache
- ▶ **The “action” here is on the cache tag high order bits...**

Overlapped Cache & TLB Access

- ▶ **Simple for small caches**
 - ▷ $IDX + \text{BlockOffset} \leq \text{PageOffset}$
- ▶ **Must satisfy**
Cache size/Assoc \leq Page size
- ▶ **Assume 4K pages &**
2-way set-associative caches
What is the max size allowed for
parallel address translation to work?



Two Cache Architectures

1. **Virtually-indexed Virtually-tagged Caches**

- ▶ Also known as Virtually-Addressed or Virtual-Address or just “Virtual” Caches
- ▶ The VPN bits are used for both tags and index bits

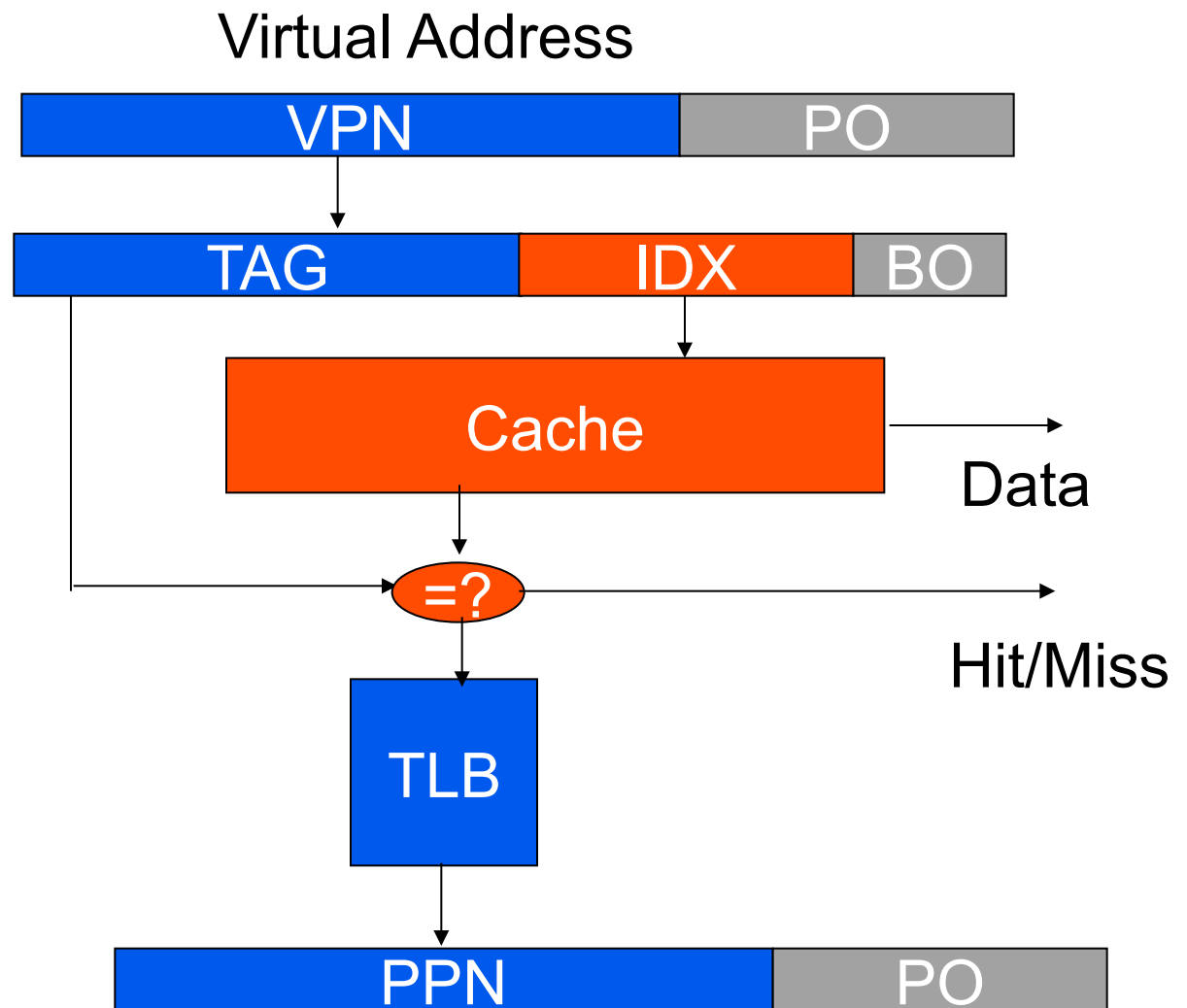
2. **Virtually-indexed Physically-tagged Caches**

- ▶ The VPN bits only contribute to the index
- ▶ The tag is physical and requires a TLB lookup in parallel

- ▶ **The third one you already know:**
Physically-indexed Physically-tagged (or just “physical” caches)

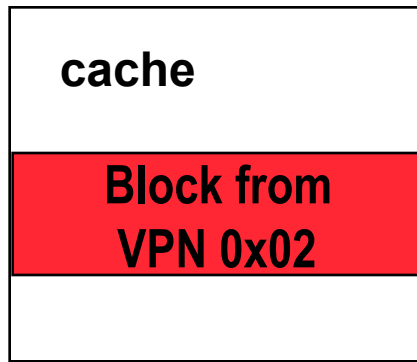
Virtual-Address Cache

- ▶ Lookup using VA
- ▶ TLB access on miss
- ▶ Use PA to access lower level (L2)

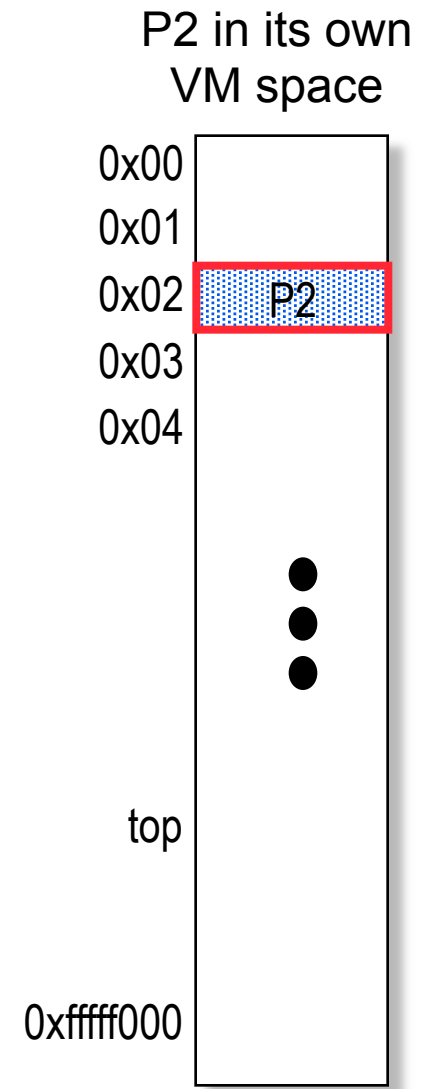
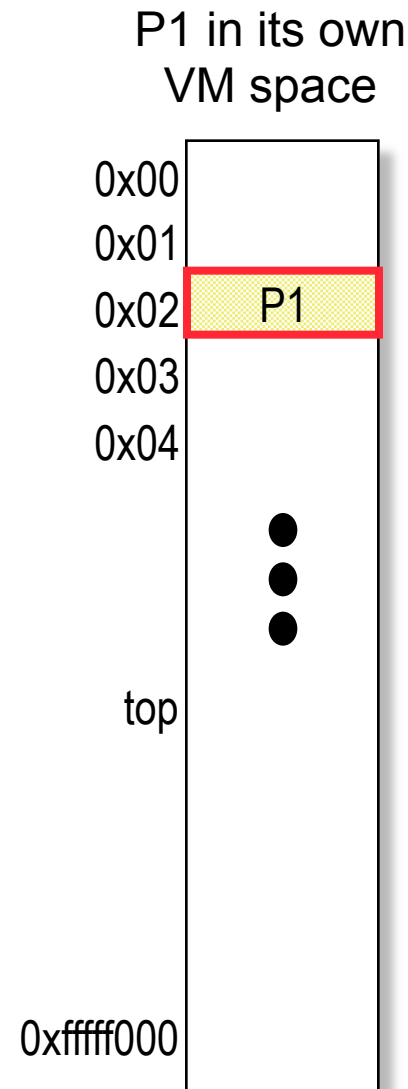
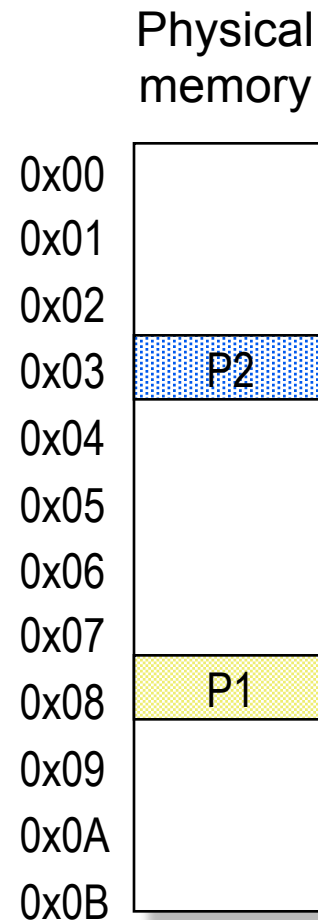


Multiple Virtual Address Spaces (Multiprogramming)

- Load/store to
VA (page 0x02)



- Is it VA from P1 or P2?
- Is it a hit? miss?



Three Solutions to Support Multiple Address Spaces

1. **Keep “process id” with cache block tags**

- ▶ Upon cache lookup check both address tag and process id

2. **Flush cache on context switch**

- ▷ Expensive (lose contents of the cache every switch)

3. **Use single-address space OS's**

- ▷ Nope. Still under research.

Summary

► **Memory access is hard! Complicated!**

- ▷ Speed of CPU core demands very fast memory access. We do cache hierarchy to solve this one. Gives illusion of speed--most of the time. Occasionally slow.
- ▷ Size of programs demands large RAM. We do VM hierarchy to solve this one. Gives illusions of size--most of the time. Occasionally slow.

► **VM hierarchy**

- ▷ Gives illusion of a LARGE physical RAM, even if you have LESS real RAM
- ▷ RAM divided into chunks called pages, typically 1k-8k bytes. “Live” pages are in the physical RAM. Pages that don’t fit are on the disk.
- ▷ Page table serves as translation mechanism from virtual to physical address
 - ▷ Page table lives in physical memory, managed by OS
 - ▷ For 64b addresses, multi-level tables used, some of the table is in VM
- ▷ TLB is yet another cache--caches translated addresses, page table entries.
 - ▷ Saves from having to go to physical memory to do lookup on each access
 - ▷ Usually very small, managed by OS
- ▷ VM, TLB, cache have “interesting” interactions.
 - ▷ Big impacts on speed, pipelining. Big impacts on exactly where the virtual to physical mapping takes place.