Lecture 13:
Virtual Memory

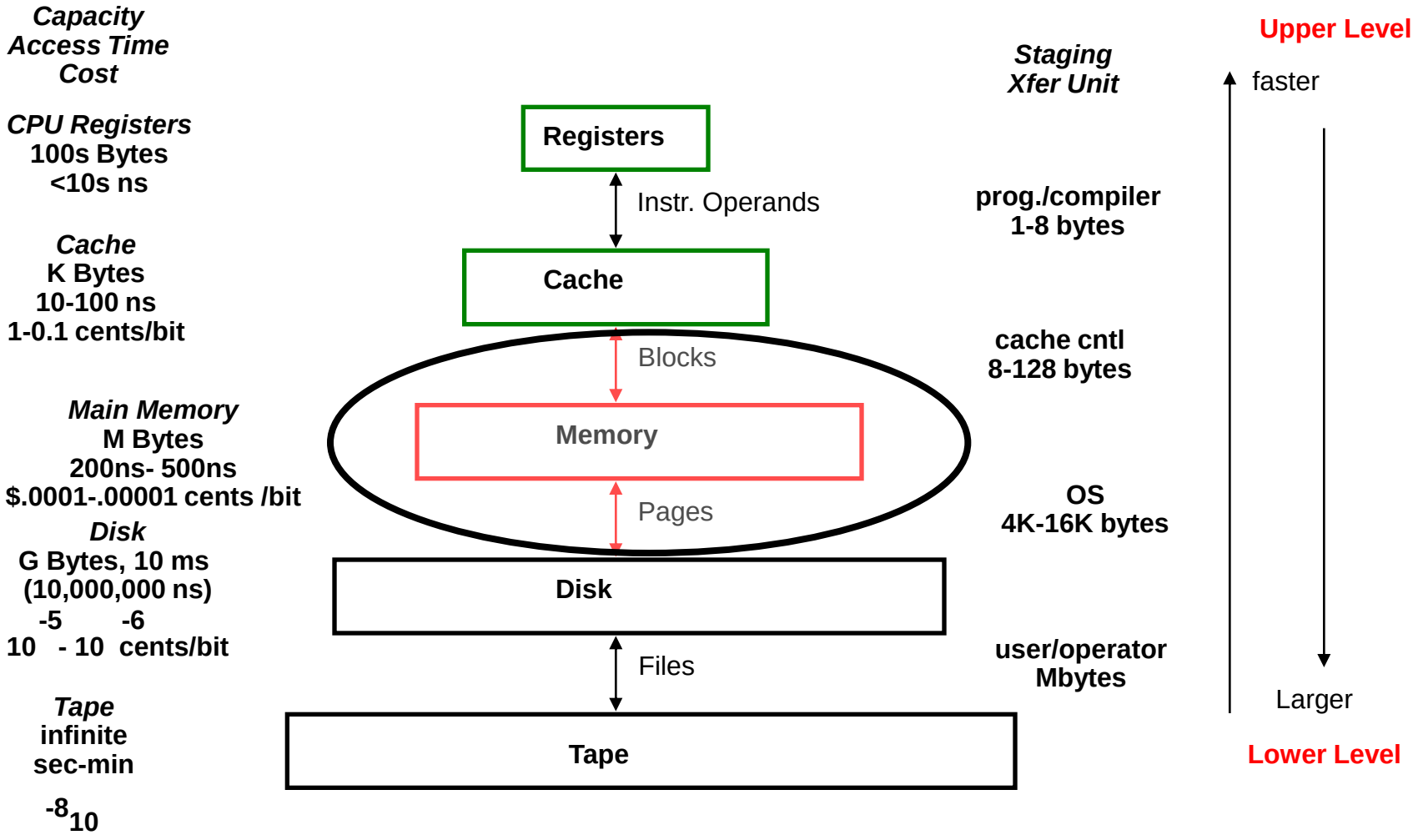
Iakovos Mavroidis

Computer Science Department
University of Crete

Outline

- **Virtual Memory**
  - **Basics**
  - **Address Translation**
  - **Cache vs VM**
  - **Paging**
  - **Replacement**
  - **TLBs**
  - **Segmentation**
  - **Page Tables**

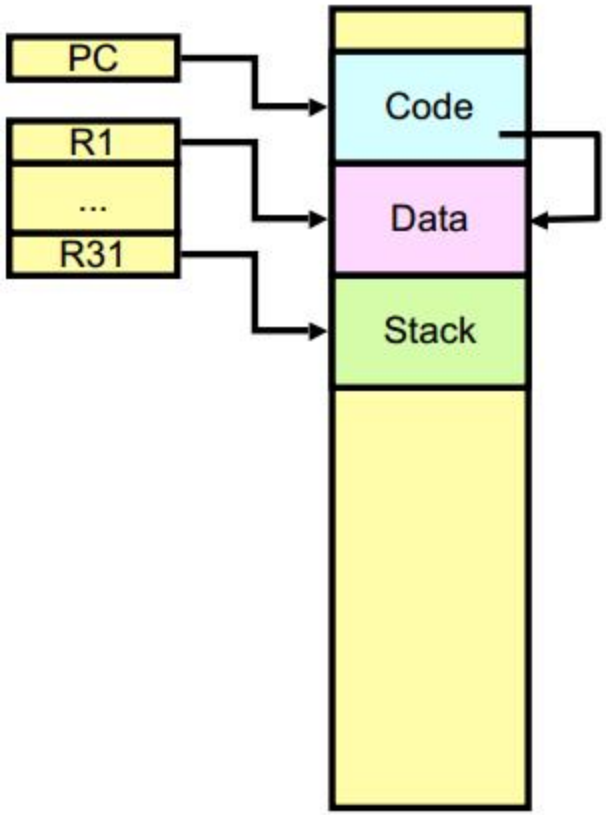
Memory Hierarchy



Virtual Memory

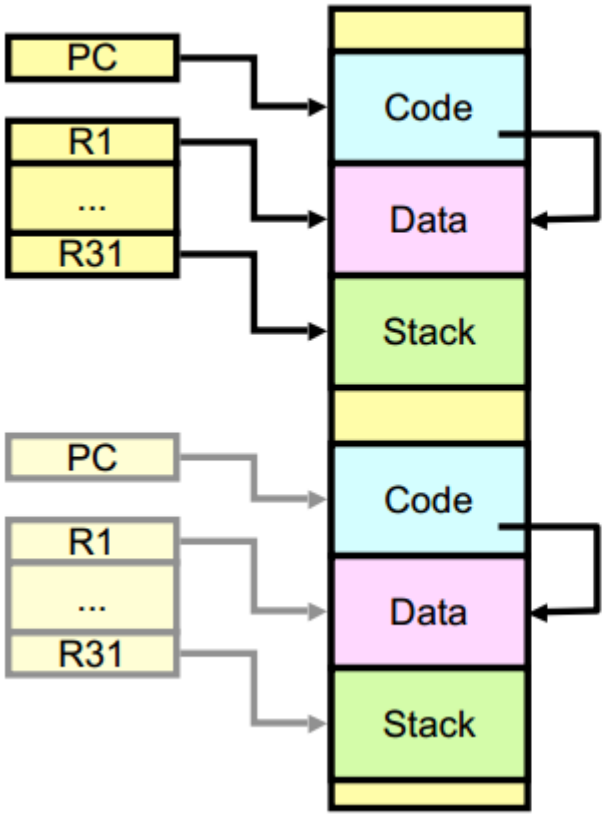
- **Some facts of computer life…**
  - **Computers run lots of processes simultaneously**
  - **No full address space of memory for each process**
  - **Must share smaller amounts of physical memory among many processes**
- **Virtual memory is the answer!**
  - **Divides physical memory into blocks, assigns them to different processes**
  - **Virtual memory (VM) allows main memory (DRAM) to act like a cache for secondary storage (magnetic disk).**
  - **VM address translation a provides a mapping from the virtual address of the processor to the physical address in main memory or on disk.**

Simple View of Memory



- Single *program* runs at a time
- Code and static data are at fixed locations
  - code starts at fixed location, e.g., 0x100
  - subroutines may be at fixed locations (absolute jumps)
- data locations may be *wired* into code
- Stack accesses relative to stack pointer.

Running Two Programs (Relocation) No Protection



- Need to relocate *logical* addresses to *physical* locations
- Stack is already relocatable
  - all accesses relative to SP
- Code can be made relocatable
  - allow only relative jumps
  - all accesses relative to PC
- Data segment
  - can calculate all addresses relative to a DP
    - expensive
  - faster with hardware support
    - base register

Three Advantages of Virtual Memory

- **Translation:**
  - **Program can be given consistent view of memory, even though physical memory is scrambled**
  - **Makes multithreading reasonable (now used a lot!)**
  - **Only the most important part of program ("Working Set") must be in physical memory.**
  - **Contiguous structures (like stacks) use only as much physical memory as necessary yet still grow later.**
- **Protection:**
  - **Different threads (or processes) protected from each other.**
  - **Different pages can be given special behavior**
    - » **(Read Only, Invisible to user programs, etc).**
  - **Kernel data protected from User programs**
  - **Very important for protection from malicious programs**
- **Sharing:**
  - **Can map same physical page to multiple users ("Shared memory")**

Protection with Virtual Memory

- Virtual memory allows protection without the requirement that pages be pre-allocated in contiguous chunks

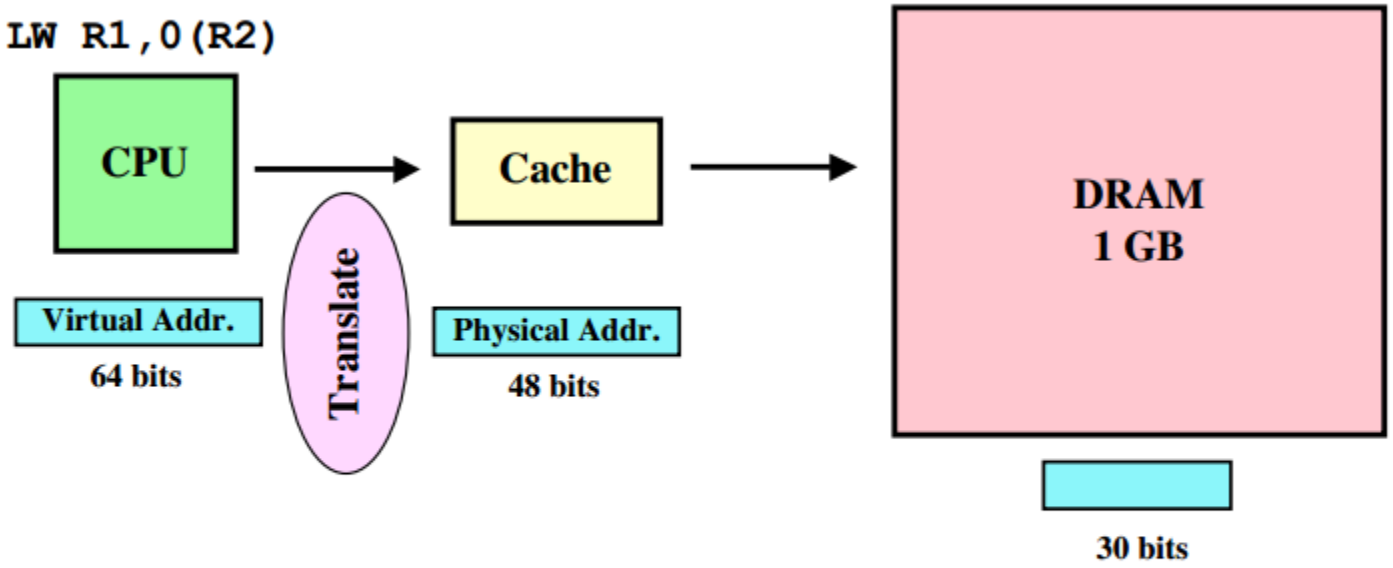
- Physical pages are allocated based on program needs and physical pages belonging to different processes may be adjacent – efficient use of memory

- Each page has certain read/write properties for user/kernel that is checked on every access
  - a program's executable can not be modified
  - part of kernel data cannot be modified/read by user
  - page tables can be modified by kernel and read by user

Basics

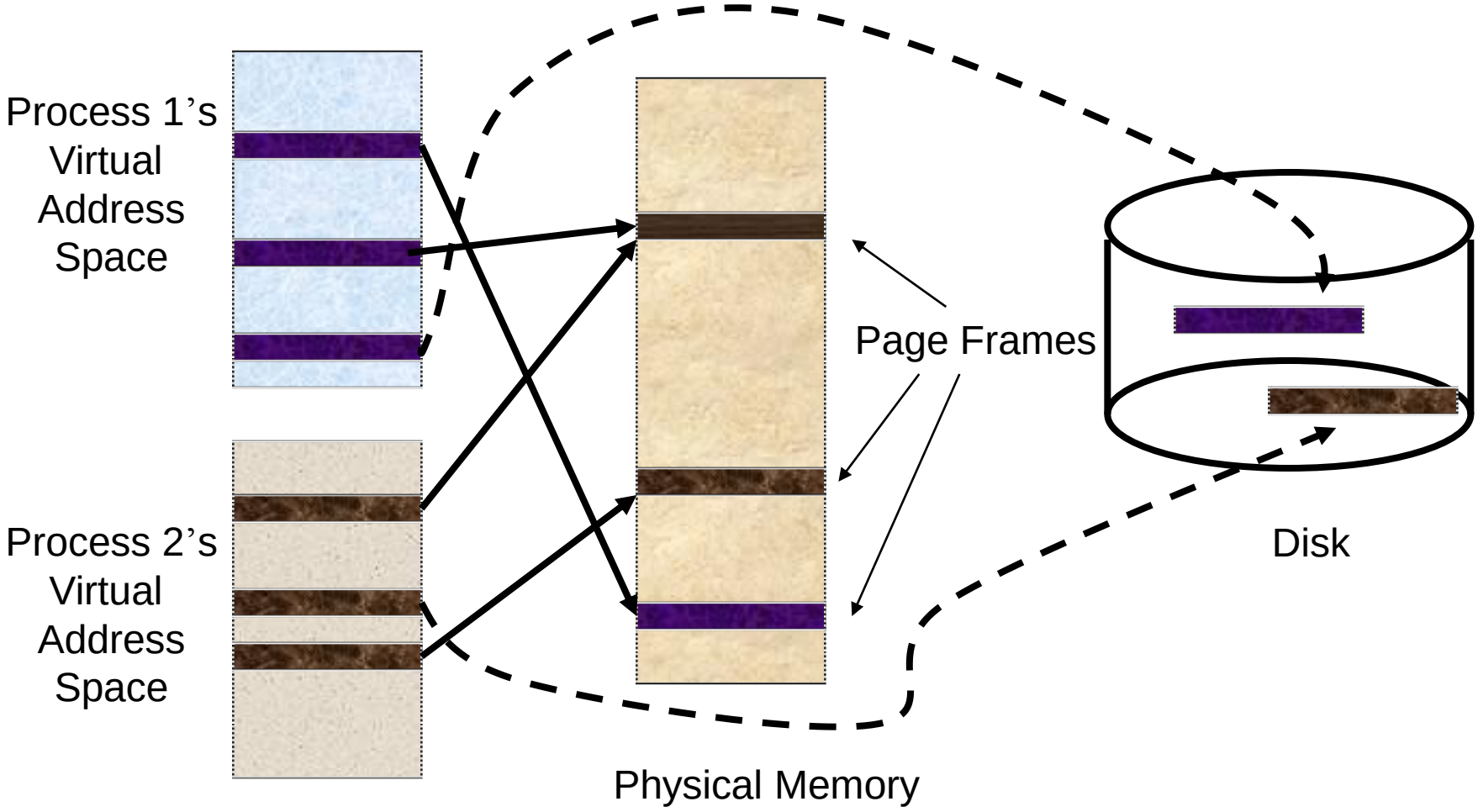
- Programs reference “virtual” addresses in a non-existent memory
  - These are then translated into real “physical” addresses
  - Virtual address space may be bigger than physical address space
- Divide physical memory into blocks, called pages
  - Anywhere from 512B to 16MB (4k typical)
- Virtual-to-physical translation by indexed table lookup
  - Add another cache for recent translations (the TLB)
- Invisible to the programmer
  - Looks to your application like you have a lot of memory!

A Load to Virtual Memory

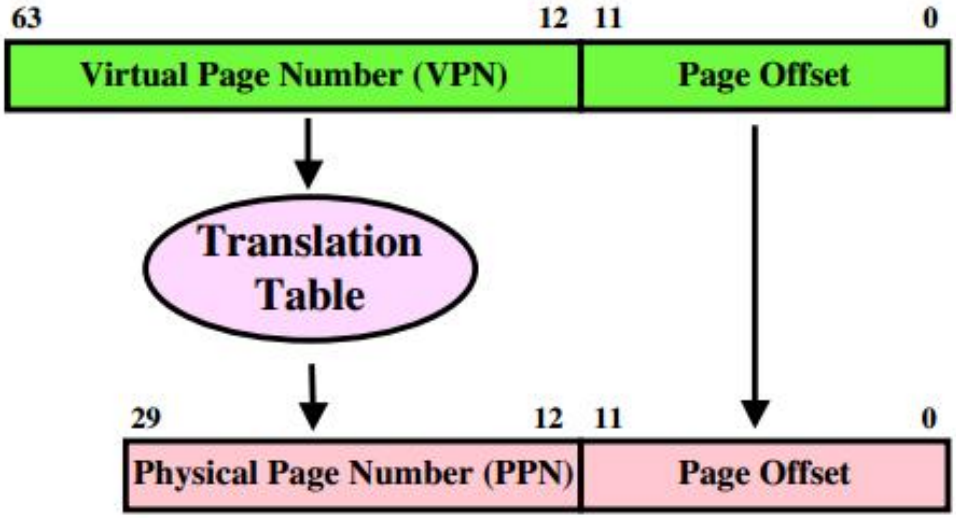


- Translate from virtual space to physical space
  - VA ⇒ PA
  - May need to go to disk

VM: Page Mapping



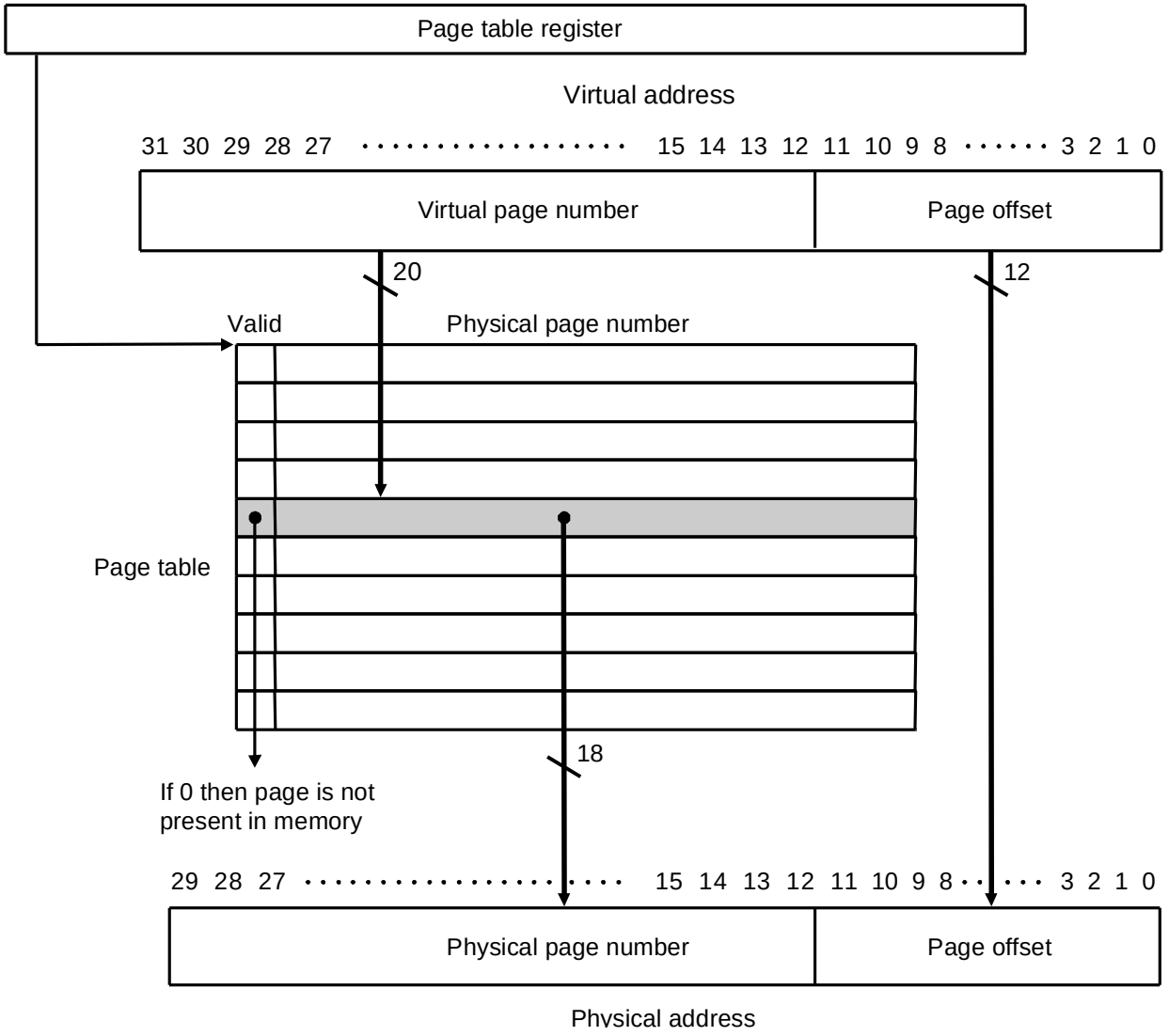
Virtual Address Translation



- Main Memory = 1 GB
- Page Size = 4KB
- VPN = 52 bits
- PPN = 18 bits

- Translation table
  - aka "Page Table"

Virtual Address Translation



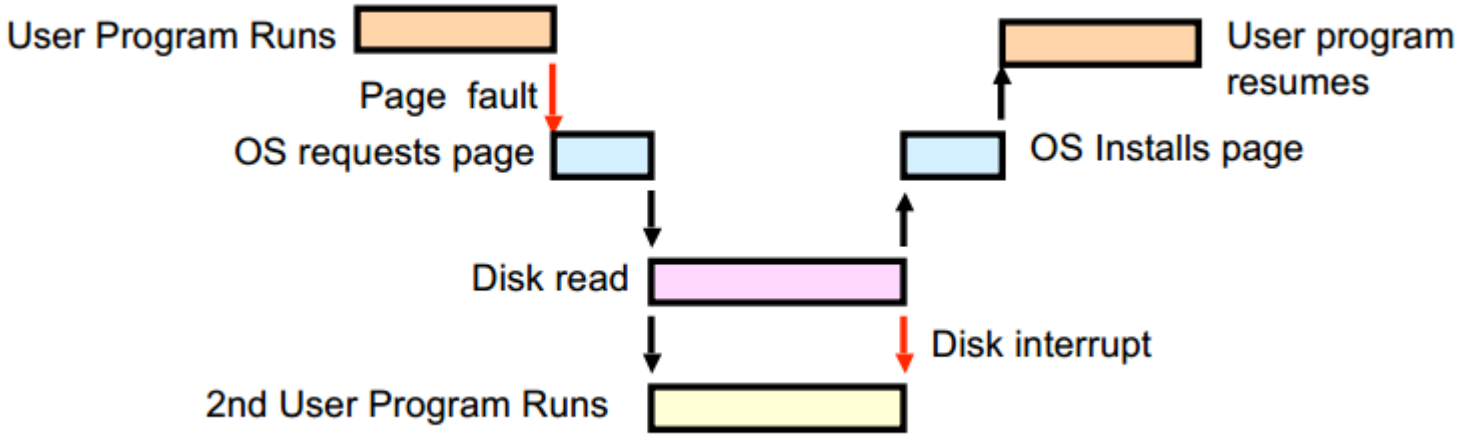
Cache terms vs. VM terms

"real"/physical memory

So, some definitions/"analogies"

- A "*page*" or "*segment*" of memory is analogous to a "block" in a cache
- A "*page fault*" or "*address fault*" is analogous to a cache miss

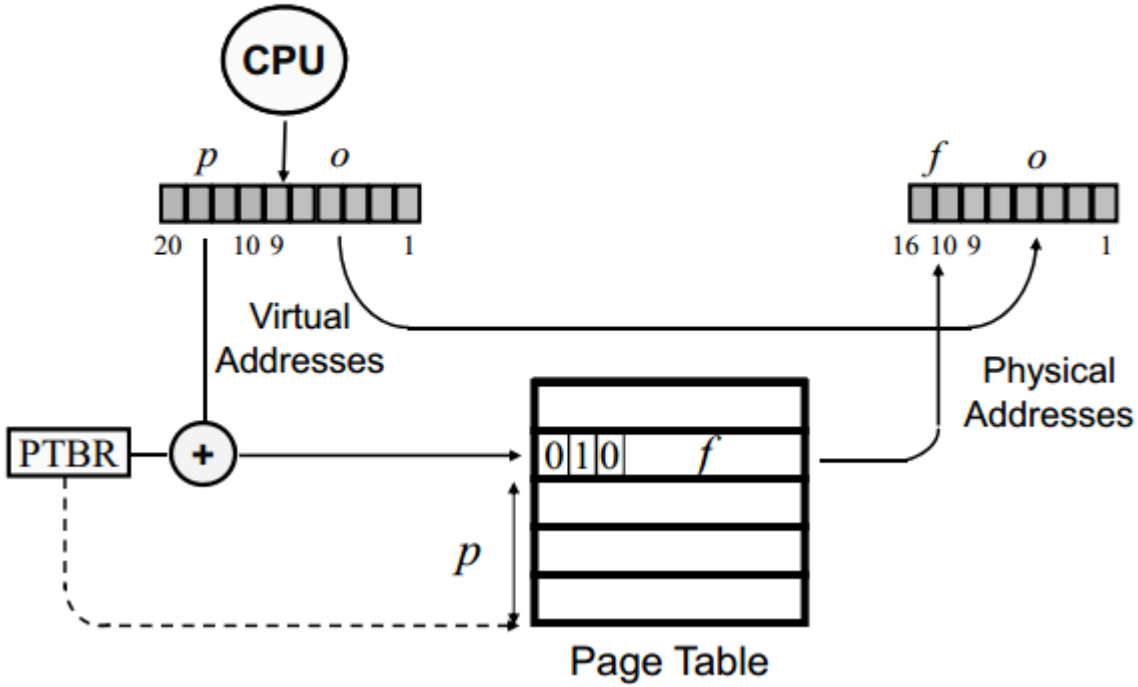
so, if we go to main memory and our data isn't there, we need to get it from disk…



Virtual Address Translation Details

1 table per process
Part of process's state

- Contents:
  - Flags — dirty bit, resident bit, clock/reference bit
  - Frame number



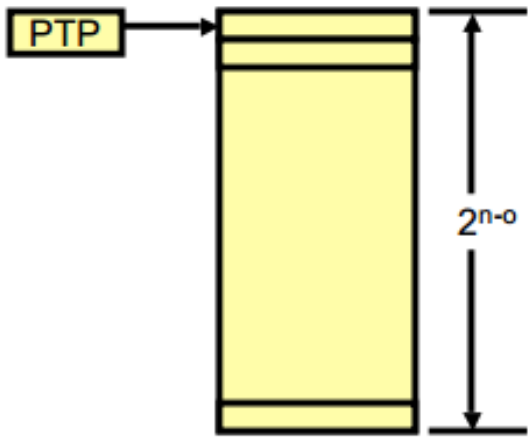
PTBR: Page Table Base Register

Cache VS VM

Parameter	First-level cache	Virtual memory
Block (page) size	12-128 bytes	4096-65,536 bytes
Hit time	1-2 clock cycles	40-100 clock cycles
Miss penalty (Access time) (Transfer time)	8-100 clock cycles (6-60 clock cycles) (2-40 clock cycles)	700,000 – 6,000,000 clock cycles (500,000 – 4,000,000 clock cycles) (200,000 – 2,000,000 clock cycles)
Miss rate	0.5 – 10%	0.00001 – 0.001%
Data memory size	0.016 – 1 MB	4MB – 4GB

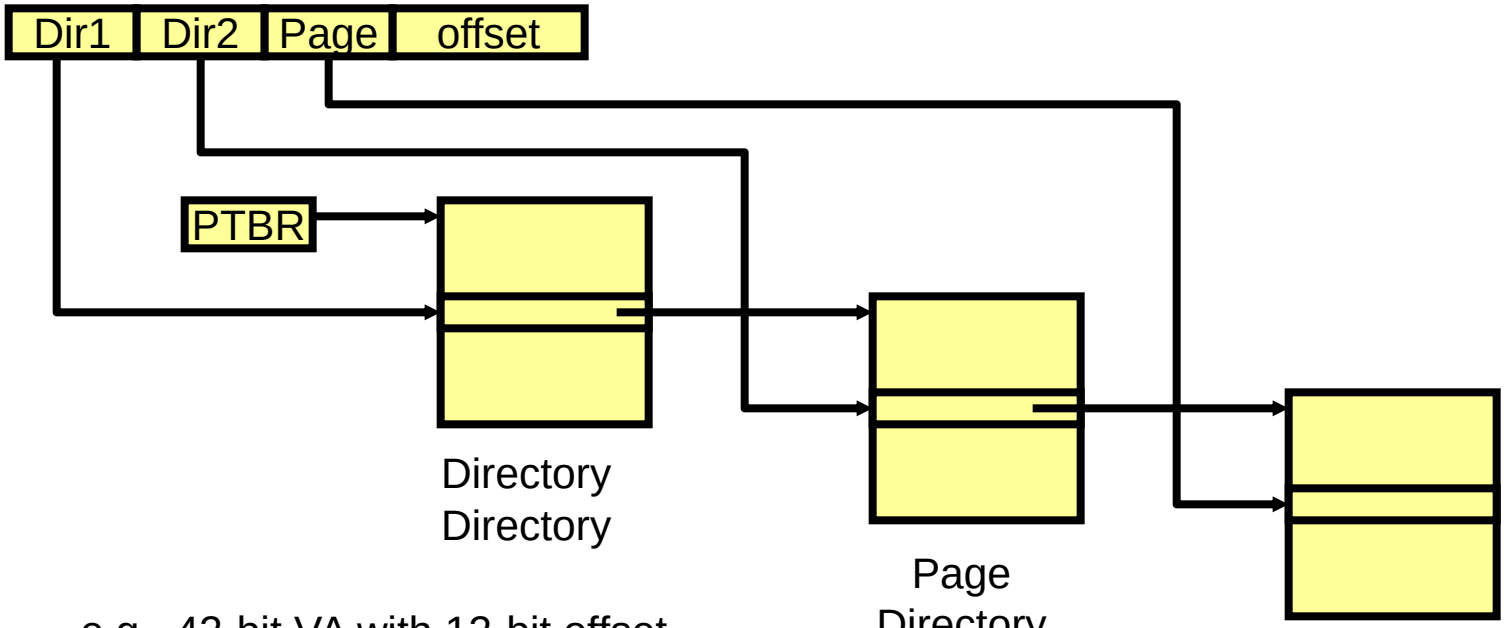
- Replacement on cache misses is primarily controlled by hardware
- The size of the processor address determines the size of virtual memory
- Secondary storage is also used for the file system

Page Table Organization



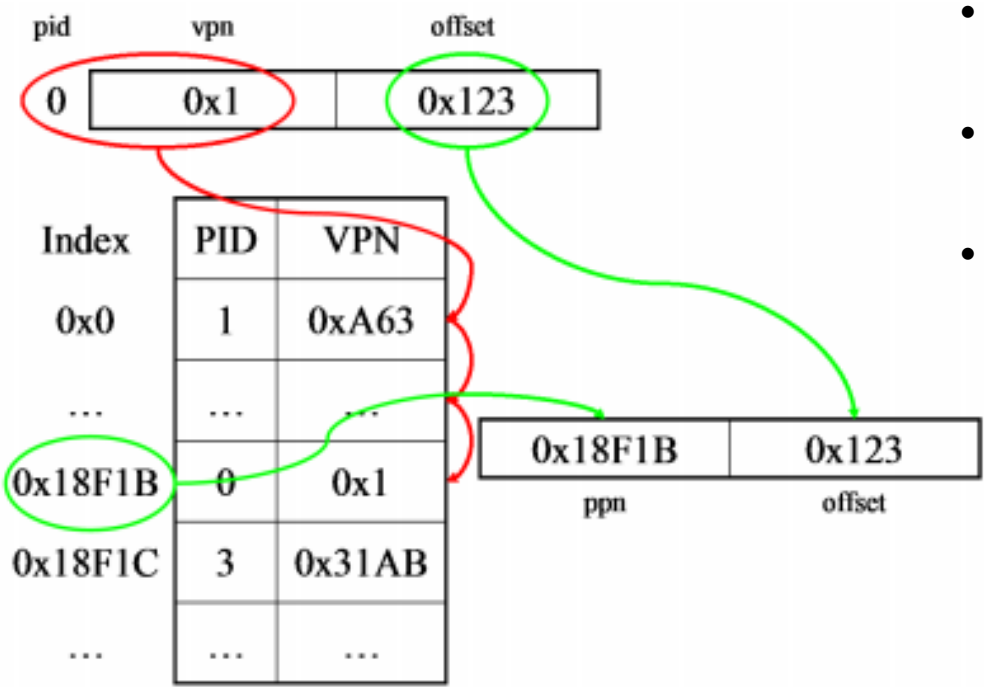
- Flat page table has size proportional to size of *virtual* address space
  - can be very large for a machine with 64-bit addresses and several processes
- Three solutions
  - page the page table (fixed mapping)
    - what really needs to be *locked down?*
  - multi-level page table (lower levels paged - Tree)
  - inverted page table (hash table)

Multi-Level Page Table



e.g., 42-bit VA with 12-bit offset
10-bits for each of three fields
1024 4-byte entries in each table (one page)

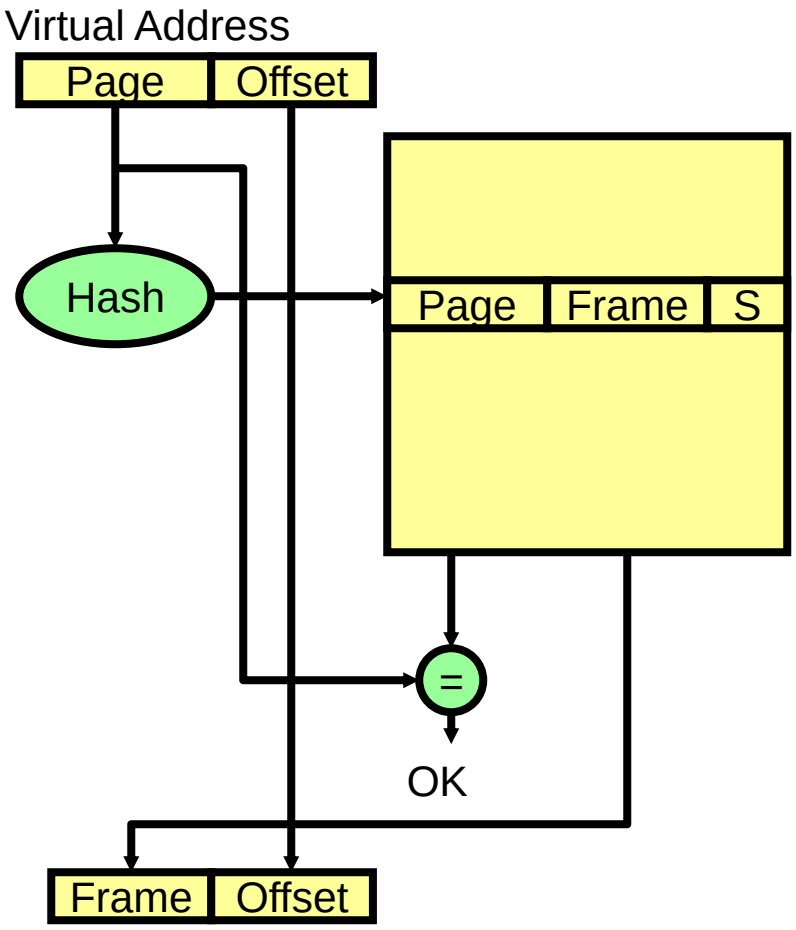
Inverted Page Tables

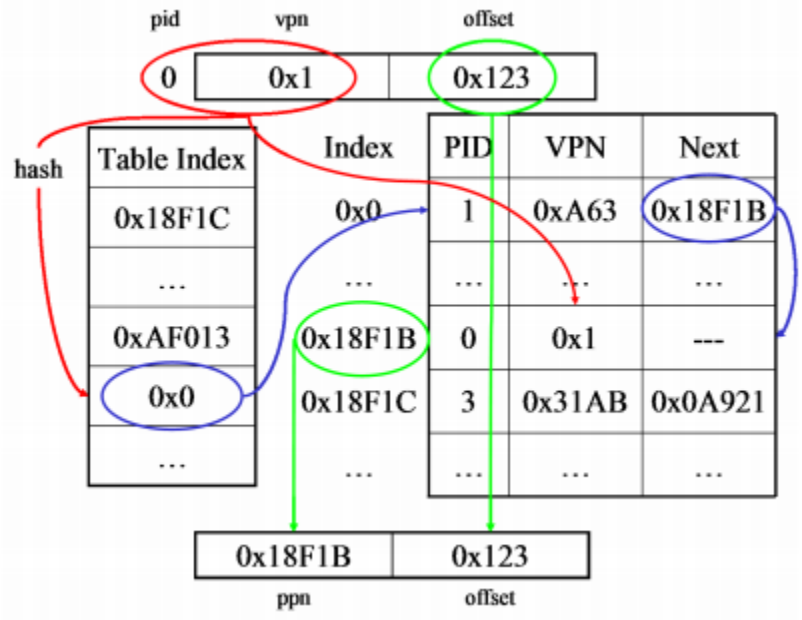


- **Store only PTEs for pages *in* physical memory**
- **Miss in page table implies page is on disk**
- **Need KP entries for P page frames (usually K > 2)**

Requires a large CAM

Hashed Inverted Page Tables





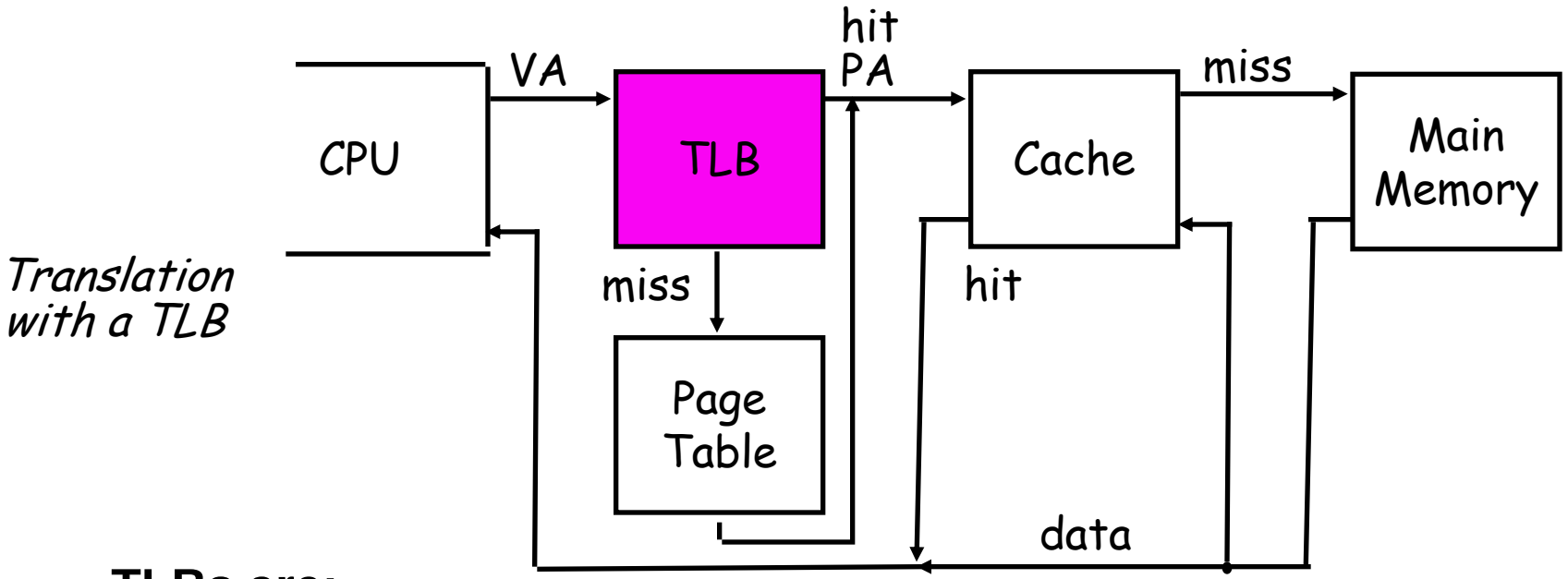
- **Chaining in order to solve collisions**
- **Chain is exhausted by hitting an invalid next pointer => page fault**

Virtual Address Translation - TLB

- What happens during a memory access?
  - map virtual address into physical address using page table
  - If the page is in memory: access physical memory
  - If the page is on disk: page fault
    - » Suspend program
    - » Get operating system to load the page from disk
- Page table is in memory - this slows down access!
- Translation lookaside buffer (TLB) special cache of translated addresses (speeds access back up)

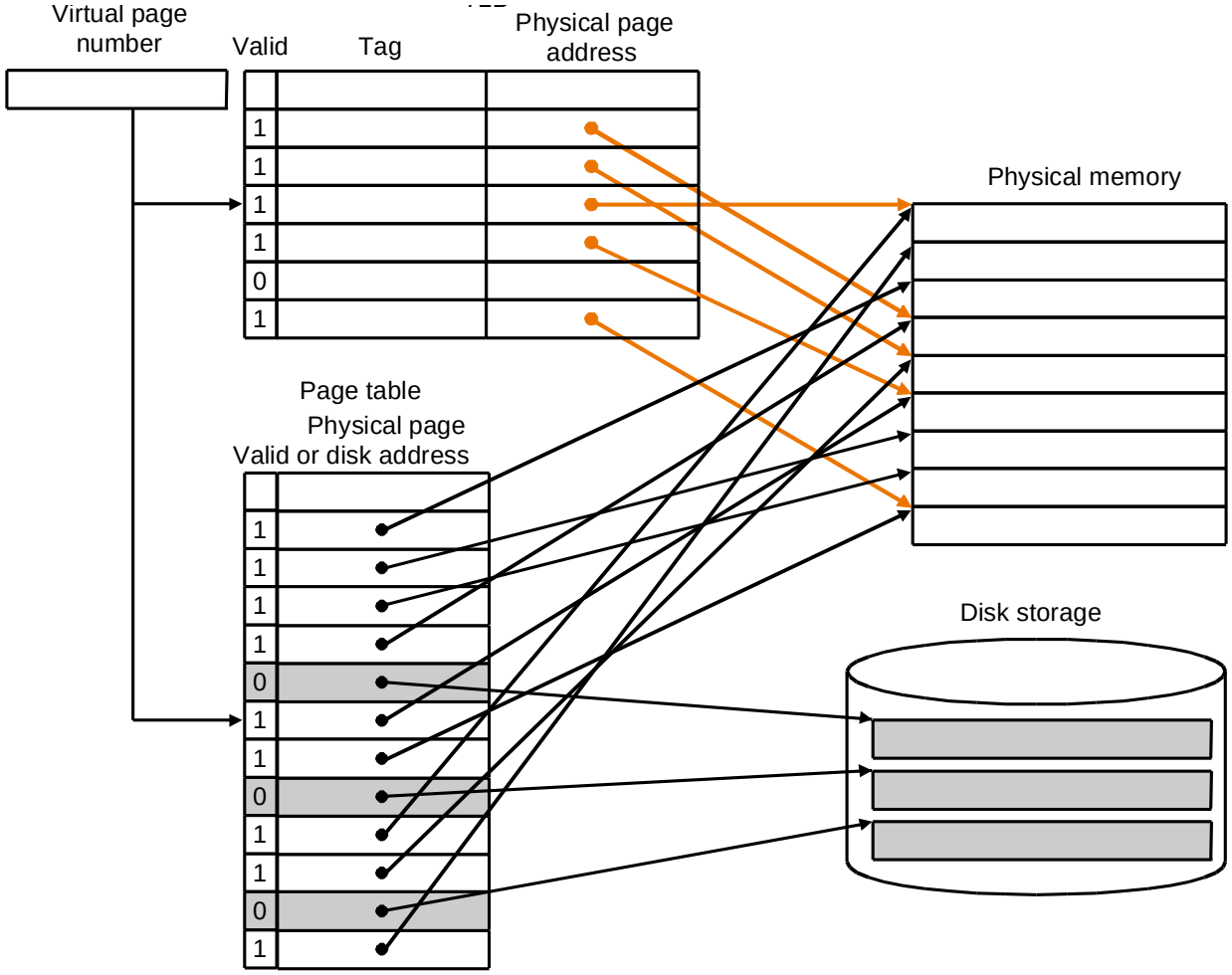
Translation Look-Aside Buffers

- **Translation Look-Aside Buffers (TLB)**
  - **Cache on translations**
  - **Fully Associative, Set Associative, or Direct Mapped**



- **TLBs are:**
  - **Small – typically not more than 128 – 256 entries**
  - **Fully Associative**

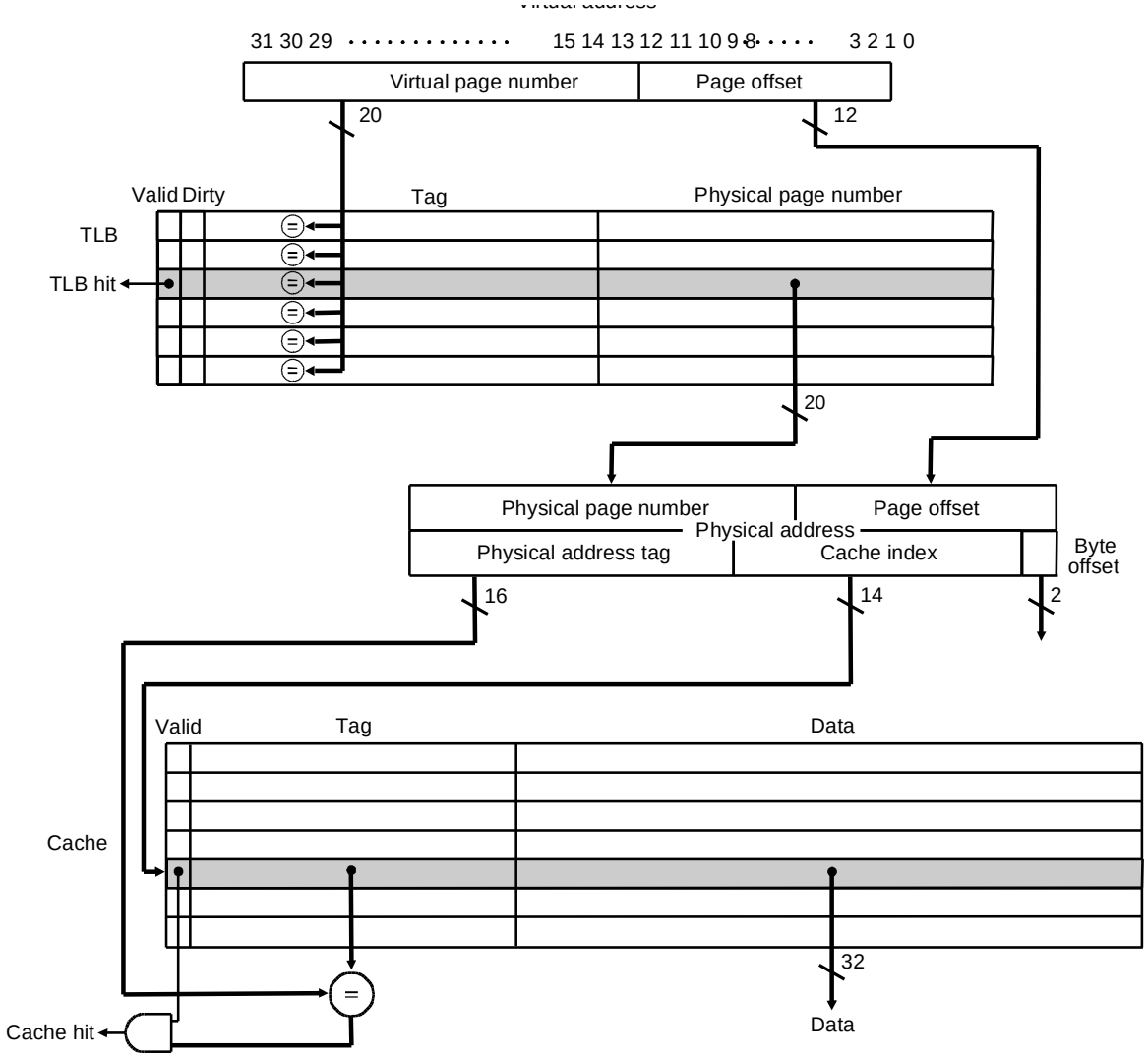
TLB Structure



What Actually Happens on a TLB Miss?

- Hardware traversed page tables:
  - On TLB miss, hardware in MMU looks at current page table to fill TLB (may walk multiple levels)
    - If PTE valid, hardware fills TLB and processor never knows
    - If PTE marked as invalid, causes Page Fault, after which kernel decides what to do afterwards
- Software traversed Page tables (like MIPS)
  - On TLB miss, processor receives TLB fault
  - Kernel traverses page table to find PTE
    - If PTE valid, fills TLB and returns from fault
    - If PTE marked as invalid, internally calls Page Fault handler
- Most chip sets provide hardware traversal
  - Modern operating systems tend to have more TLB faults since they use translation for many things
  - Examples:
    - shared segments
    - user-level portions of an operating system

TLB – Cache Interaction



TLB and Cache

- Is the cache indexed with virtual or physical address?
  - To index with a physical address, we will have to first look up the TLB, then the cache → longer access time
  - Multiple virtual addresses can map to the same physical address – can we ensure that these different virtual addresses will map to the same location in cache? Else, there will be two different copies of the same physical memory word

- Does the tag array store virtual or physical addresses?
  - Since multiple virtual addresses can map to the same physical address, a virtual tag comparison can flag a miss even if the correct physical memory word is present

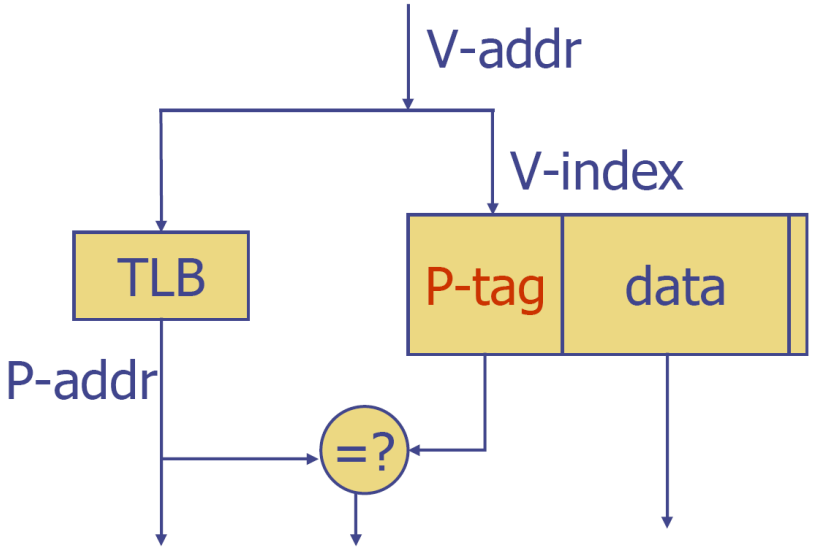
Virtual Indexed, Virtually Tagged Cache

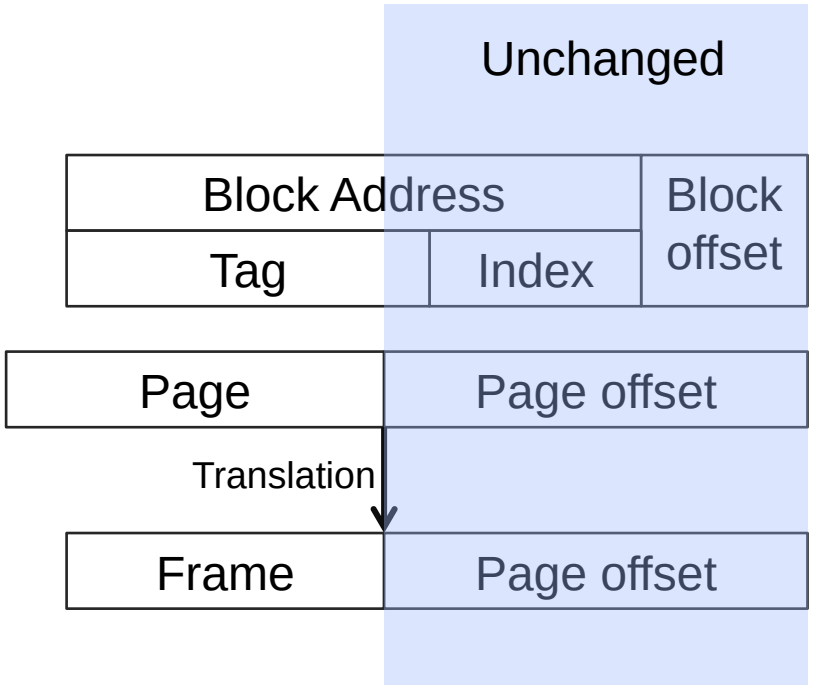
- Protection bits in cache
- Cache flushing on process switch or use Process-identifier tag (PID)
- Aliasing problem: Two different virtual addresses sharing same physical
  - Page coloring: Forces aliases to share same cache block, thus aliases cannot co-exist in the cache

Better Alternative: Virtually Indexed, Physically Tagged Cache

What motivation?

- Fast cache hit by parallel TLB access
- No virtual cache shortcomings

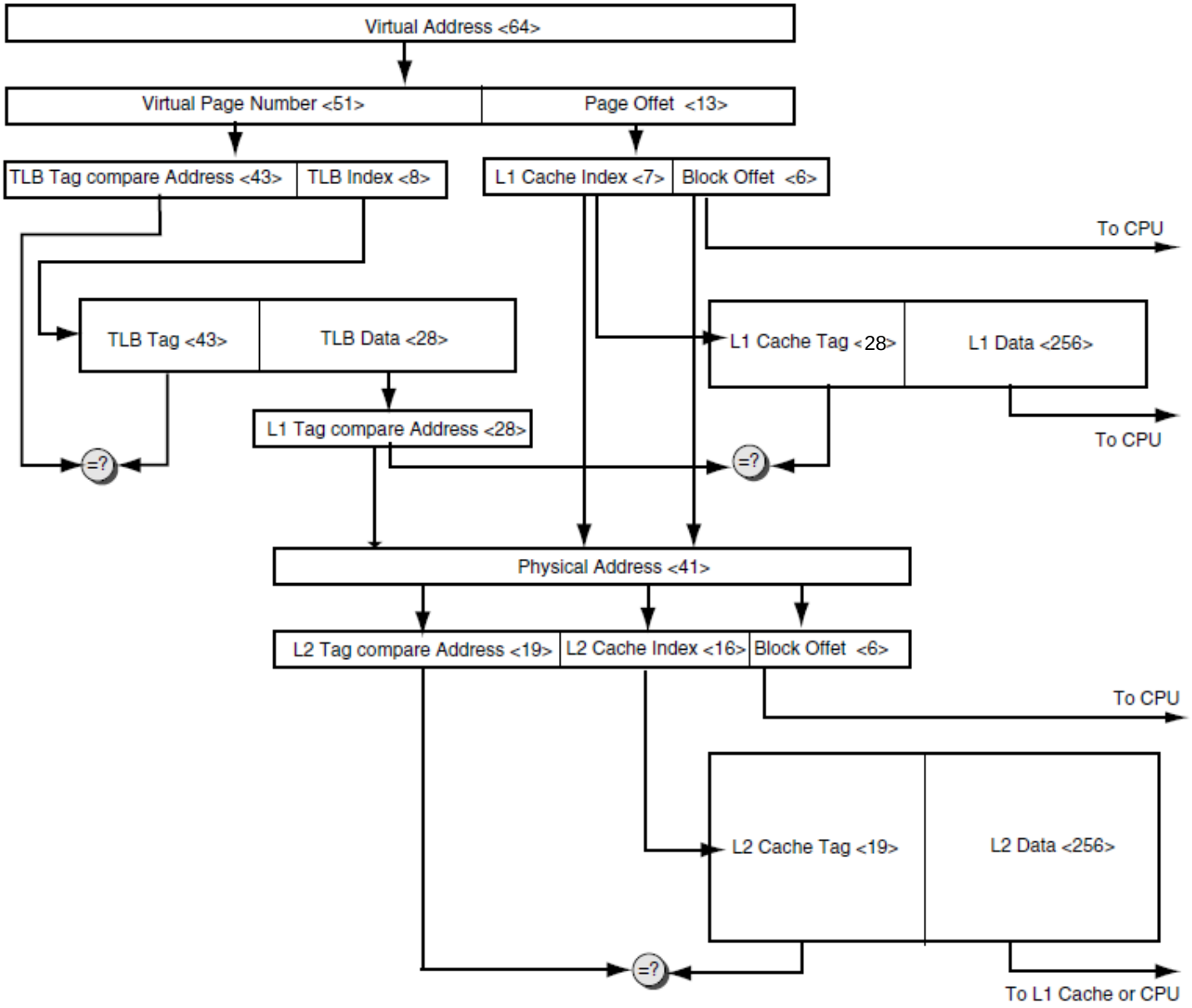




How could it be correct?

- Require #cache set * block size <= page size ⇒ physical index is from page offset
- Then virtual and physical indices are identical ⇒ works like a physically indexed cache!

Virtually Indexed, Physically Tagged Cache



Superpages

- If a program's working set size is 16 MB and page size is 8KB, there are 2K frequently accessed pages – a 128-entry TLB will not suffice
- By increasing page size to 128KB, TLB misses will be eliminated – disadvantage: memory wastage, increase in page fault penalty
- Can we change page size at run-time?
- Note that a single page has to be contiguous in physical memory

Superpages Implementation

- At run-time, build superpages if you find that contiguous virtual pages are being accessed at the same time
- For example, virtual pages 64-79 may be frequently accessed – coalesce these pages into a single superpage of size 128KB that has a single entry in the TLB
- The physical superpage has to be in contiguous physical memory – the 16 physical pages have to be moved so they are contiguous