National Yang Ming Chiao Tung University
Computer Architecture & System Lab

Virtual Memory

IOC5226 Operating System Capstone

Tsung Tai Yeh
Department of Computer Science
National Yang Ming Chiao Tung University

Acknowledgements and Disclaimer

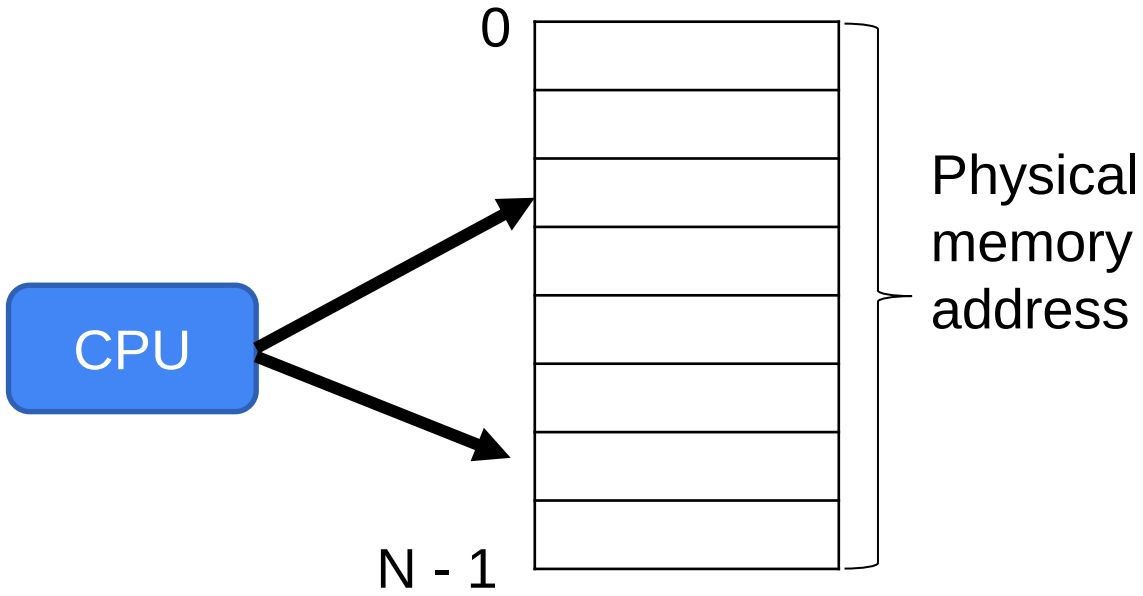
- Slides were developed in the reference with
  - MIT 6.828 Operating system engineering class, 2018
  - MIT 6.004 Operating system, 2018
  - COMP 3710 Computer Microarchitecture, Australian National University
  - Remzi H. Arpaci-Dusseau etl. , Operating systems: Three easy pieces. WISC

Outline

- Virtual memory
  - Address translation
- Paging
  - Page table
  - Translation lookaside buffer (TLB)

A system with physical memory only

- CPU's load or store addresses used directly to access memory



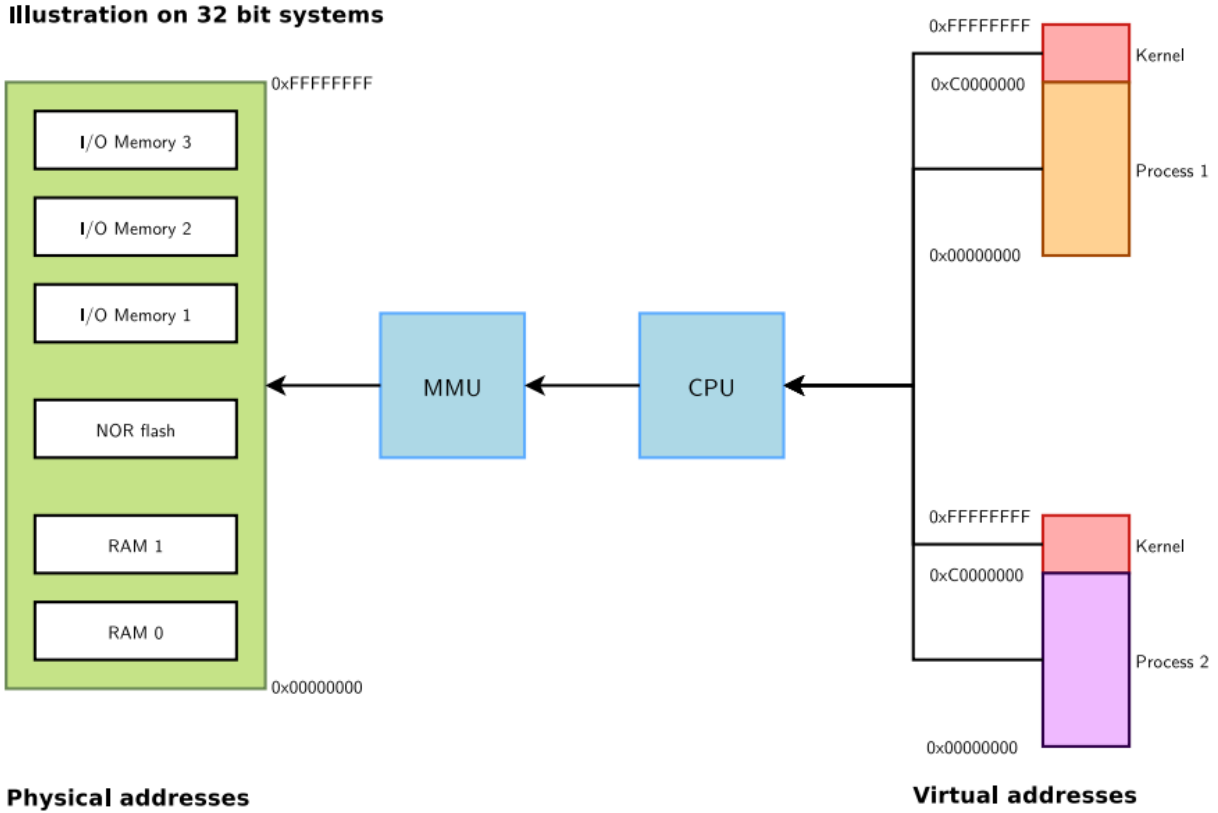
Problems of physical memory addressing

- Physical memory is of **limited size**
- Programmer needs to manage physical memory space
  - Inconvenient & difficult
  - Harder when you have multiple processes
- **Challenges**
  - Code and data relocation
  - Protection and isolation between multiple processes
  - Sharing of physical memory space

Virtual memory

- **Virtual memory**
  - The **illusion of a large address space** while having a small physical memory
  - Only **a portion of the virtual address space** lives in the physical address space at any amount of time
  - Programmer doesn’t worry about managing physical memory
  - Address generated by each instruction in a program is a “virtual address”
- **Virtual memory requires both HW + SW support**
  - Can cached in special hardware structures call translation lookaside buffers (TLBs)

Physical and virtual memory



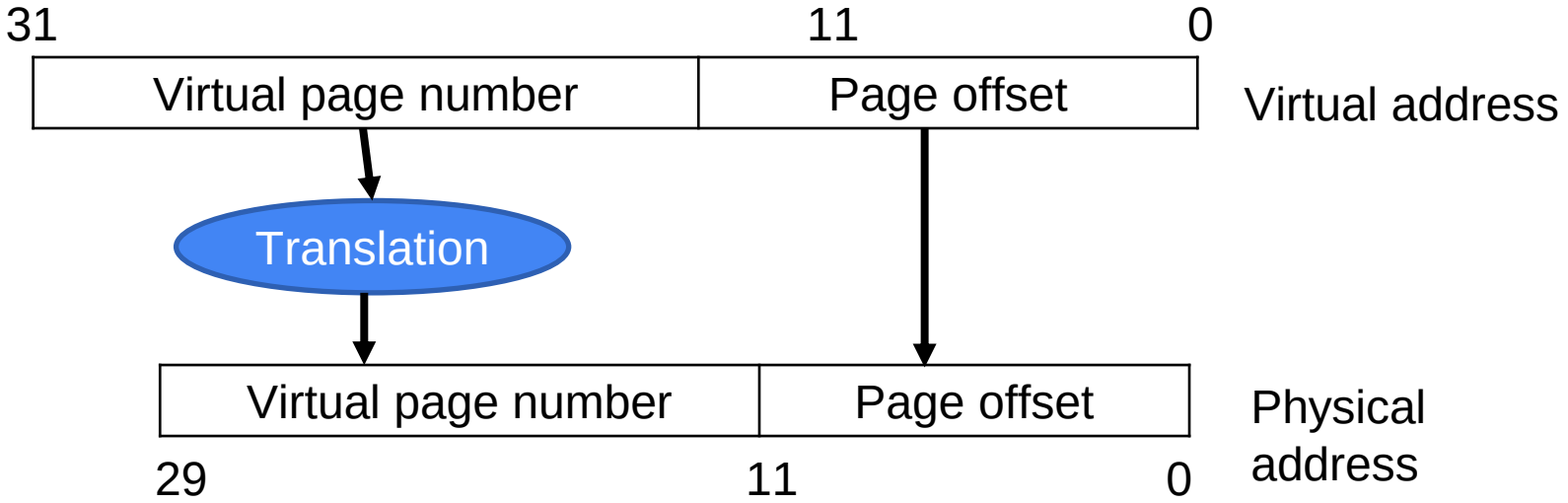
https://bootlin.com/doc/training/linux-kernel/linux-kernel-slides.pdf

Advantages of virtual memory

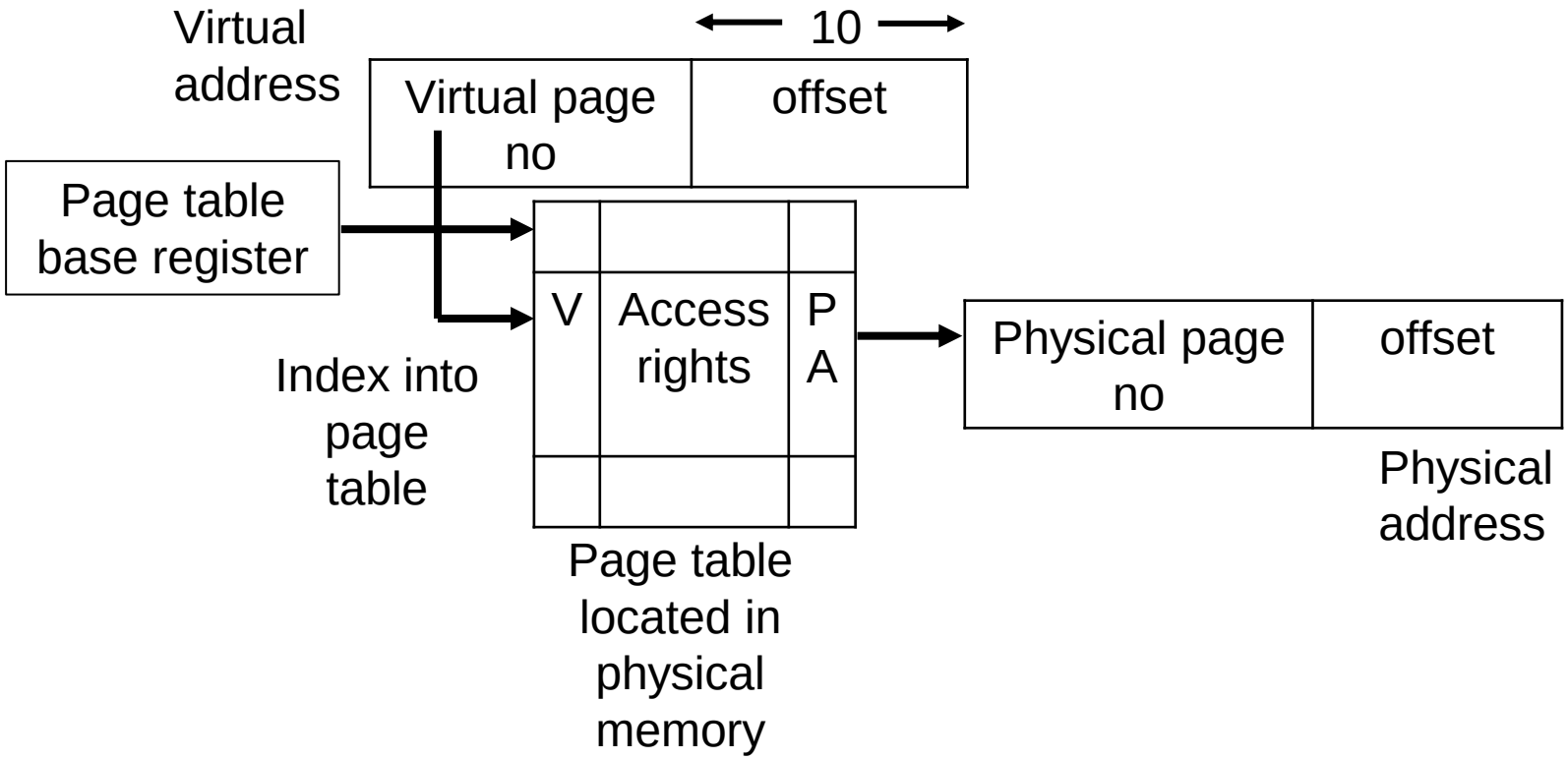
- **Illusion of having more physical memory**
- **Multiple programs share the physical memory**
  - Permit sharing without knowing other programs
  - Division of memory among programs is automatic
- **Program relocation**
  - Program addresses can be mapped to any physical location
  - Physical memory does not have to be contiguous
- **Protection**
  - Per process protection can be enforced on pages

A system with virtual memory (page based)

- **Address translation**
  - 4GB virtual memory, 1GB physical memory, page size is 4KB (2^{12}) with 2^{18} physical pages

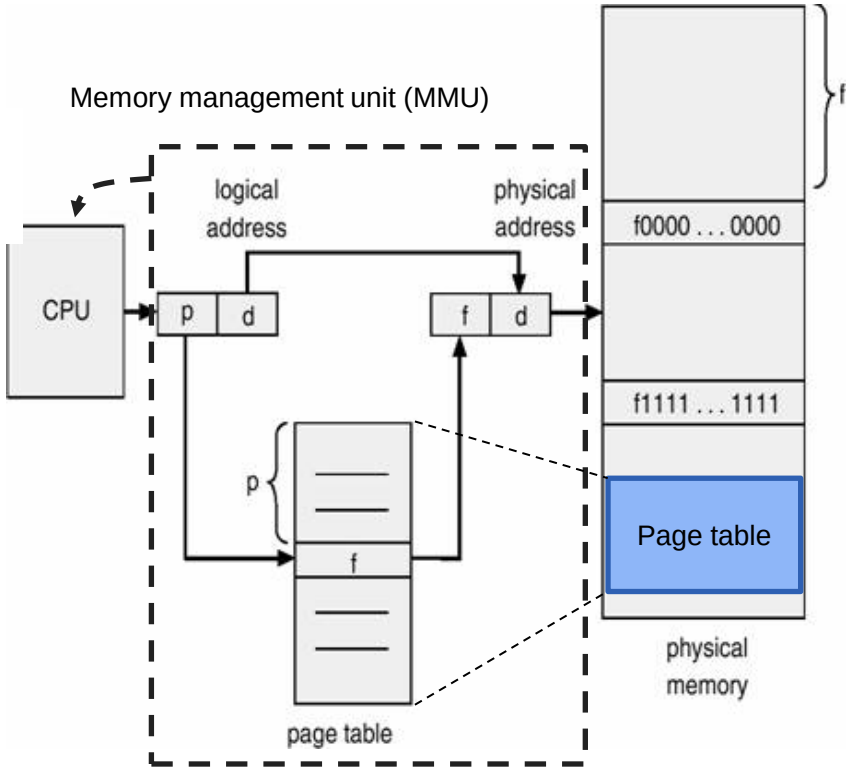


Page tables for address translation



Page memory management

MMU is in the CPU



Source: Operating System Concepts by Abraham Silberschatz, Greg Gagne, Peter B. Galvin

Virtual pages, physical frames

- **Virtual** address space divided into **pages**
- **Physical** address space divided into **frames**
- A virtual page is mapped to
  - A physical frame if the page is in the physical memory
  - A location in the disk, otherwise
- **Page table**
  - Stores the mapping of virtual pages to physical frames
  - Page table is in the main memory

Outline

- Virtual memory
  - Address translation
- Paging
  - Page table
  - Translation lookaside buffer (TLB)

Paging

- Why does data segment cause fragmentation?
  - Variable-sized segments
- Solution -> **paging !**
  - All “chunks” be the **same size** (typically 512 – 8 K bytes)
  - Call chunks be “pages” rather than “segments”
  - Allocation is done in terms of full page-aligned pages -> no bounds
  - MMU maps virtual page numbers to physical page numbers

Paging (cont.)

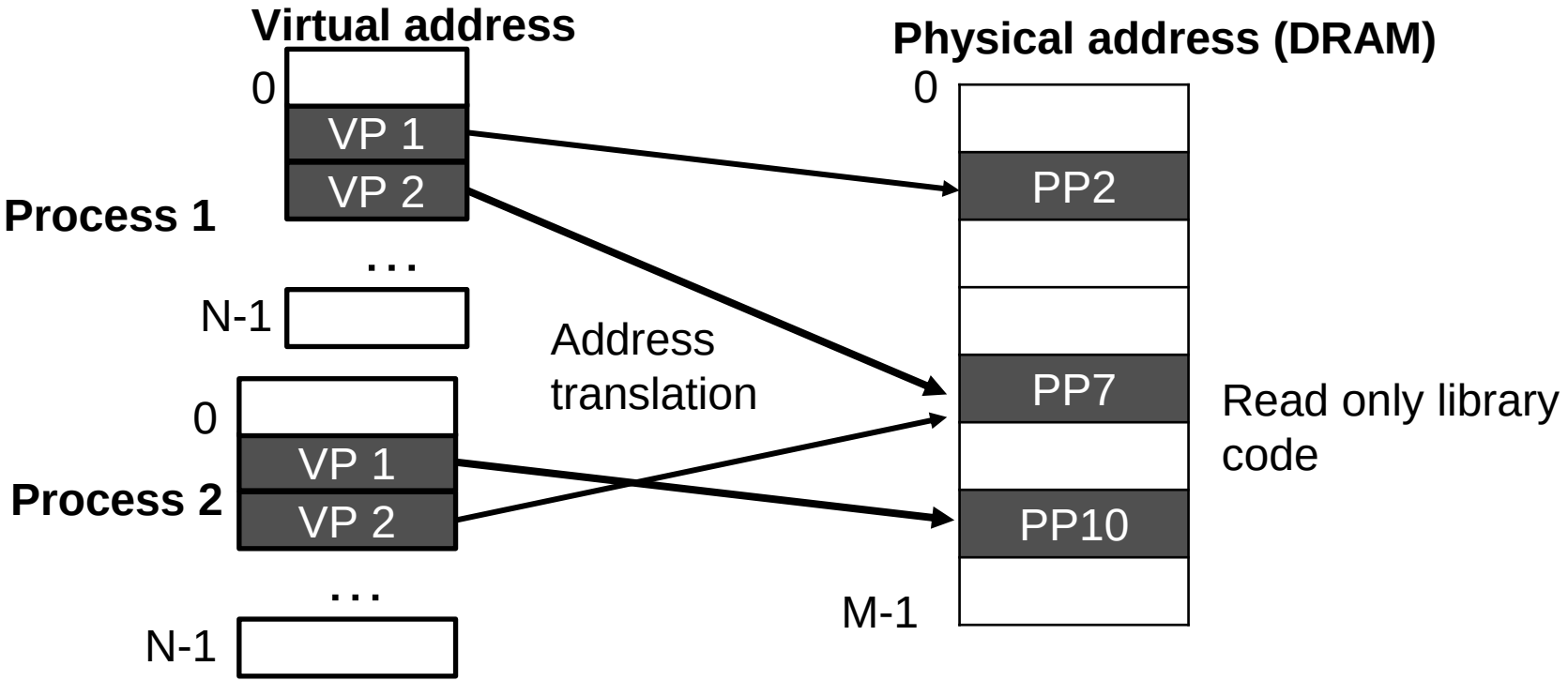
- Modern hardware and OS use paging
- **Pages** are like segments, but **fixed size**
  - The bounds register goes away
  - External fragmentation goes away
- Since pages are small (4 or 8 KB, often), a lot of them
  - So page table has to go into RAM
  - Page table might be huge
  - Accessing the page table requires a memory reference by the hardware to perform the translation

How does the paging help ?

- **How does the paging help ?**
  - No external fragmentation
  - No forced holes in virtual address space
  - Easy translation -> everything aligned on power-of-2 addresses
  - Easy for OS to manage/allocate free memory pool
- **What problems are introduced ?**
  - What if you do not need entire page ? -> internal fragmentation
  - Page table may be large
  - How can we do fast translation if not stored in processor ?
  - How big should you make your pages ?

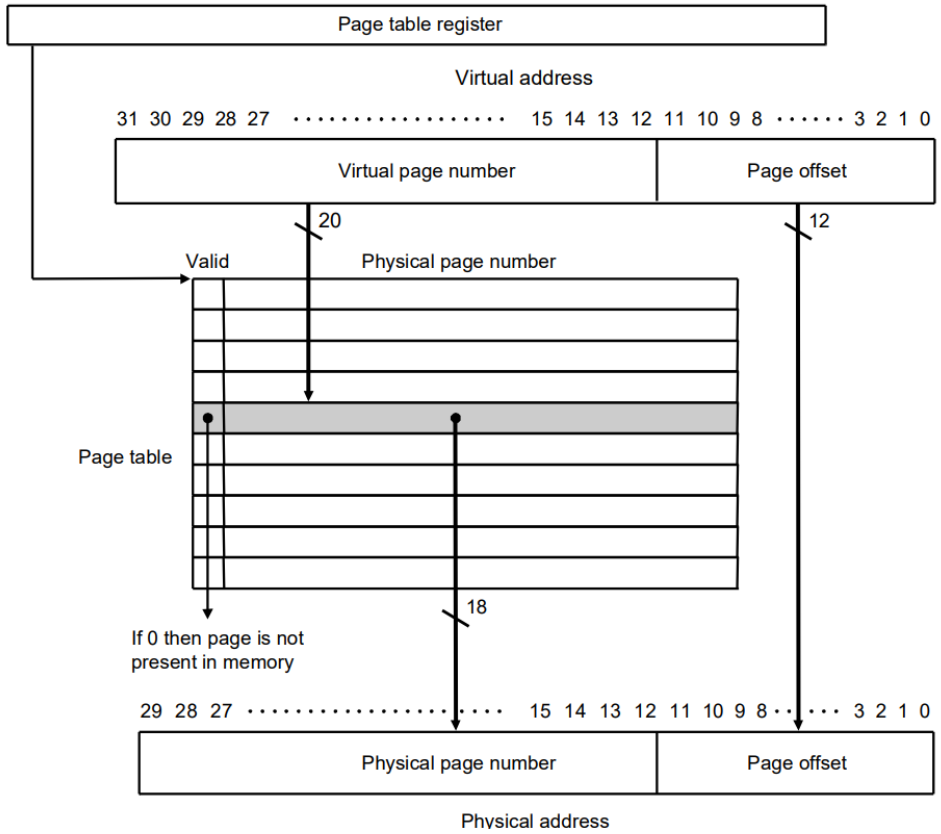
Page table is per process

- Each process has its own virtual address space



The page table

- **Each process has a separate page table**
  - A "page table register" points to the current process's page table
  - The page table is indexed with the **virtual page number (VPN)**
  - Each entry contains a valid bit, and a **physical page number (PPN)**
  - The PPN is concatenated with the page offset to get the physical address

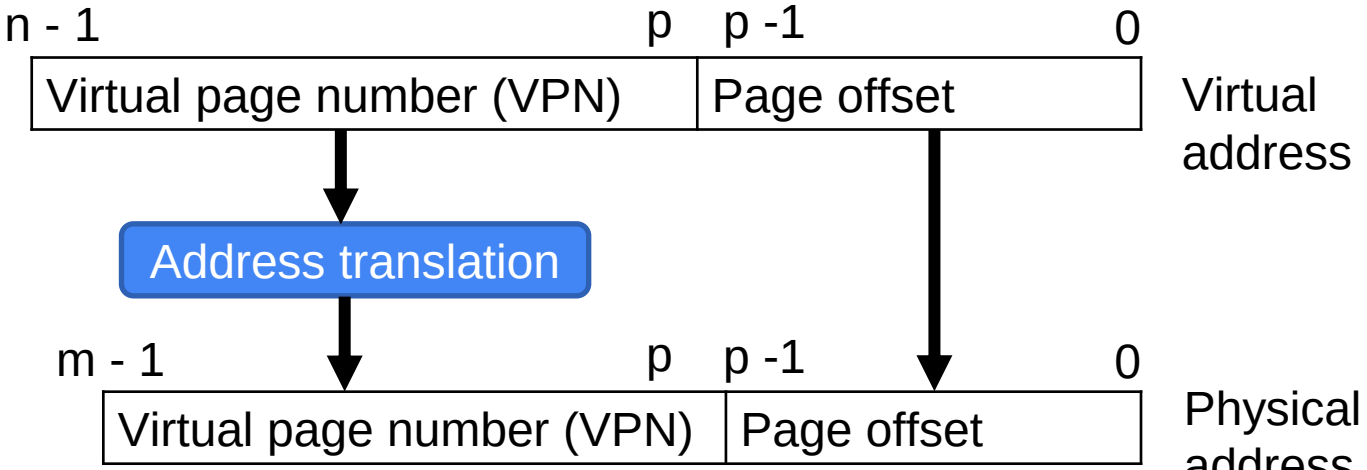


https://courses.cs.washington.edu/courses/cse378/09wi/lectures/lec20.pdf

Page table

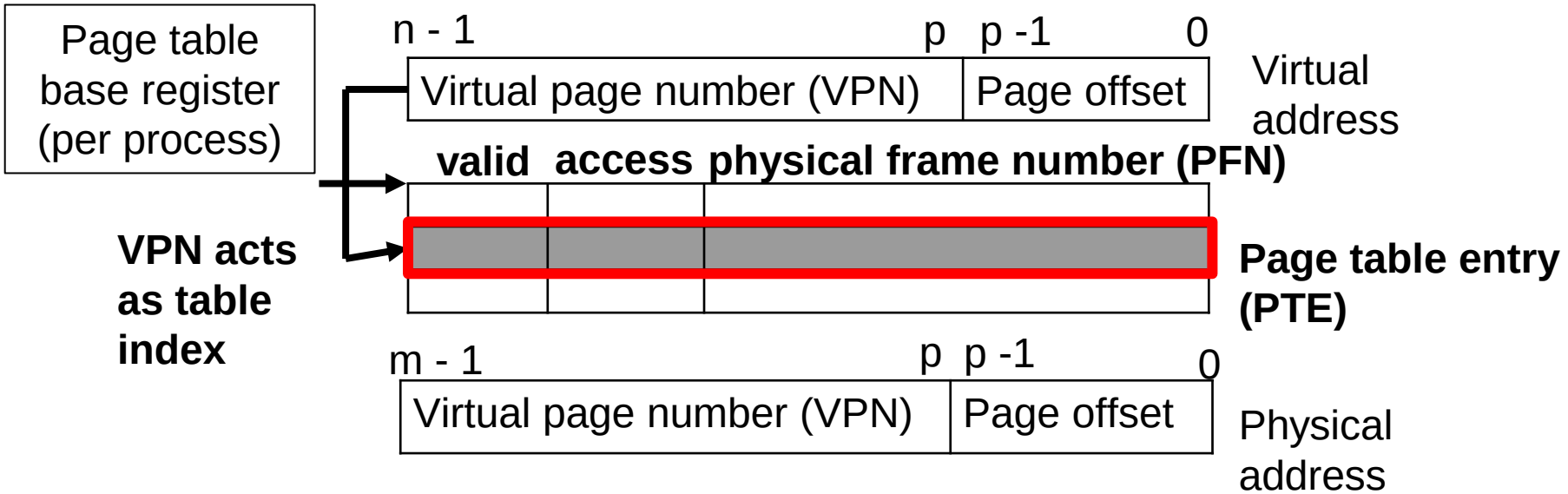
- Parameters
  - P = 2^p = page size (bytes)
  - N = 2^n = virtual-address limit
  - M = 2^m = physical address limit

Page offset bits don't change as a result of translation



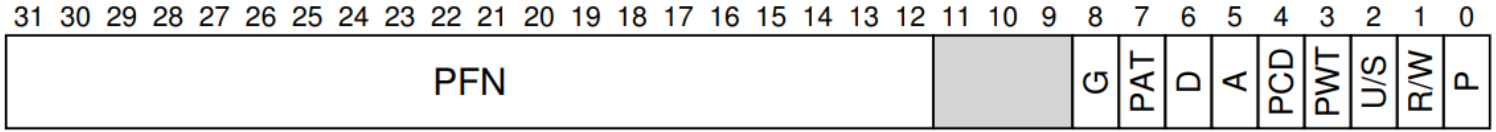
Page table (cont.)

- **Each process has a separate page table**
  - VPN forms index into page table (points to a page table entry)
  - If valid = 0, then page not in memory (page fault)



What is in a page table entry (PTE)?

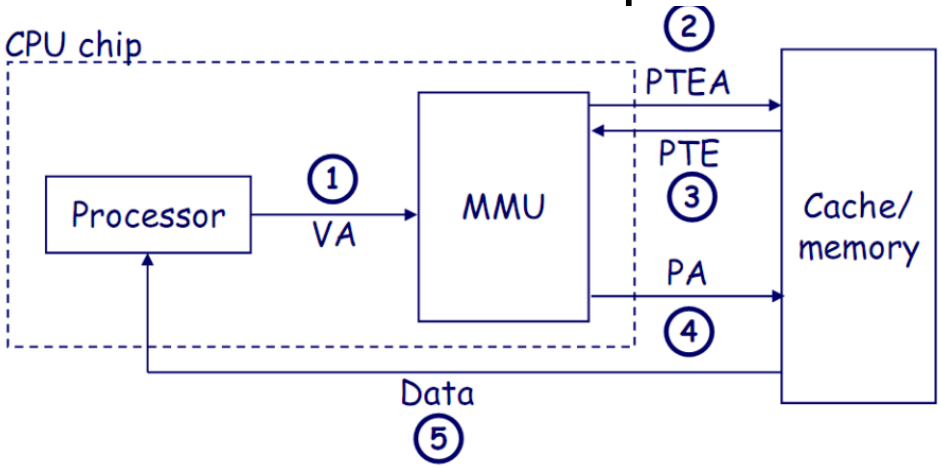
- Page table is the “tag store” for the physical memory
- PTE is the “tag store entry” for a virtual page in memory
  - **A present bit** -> whether this page is in physical memory or on disk
  - **A protection bit** -> enable access control and protection
  - **A dirty bit** -> whether page has been modified since it was brought into memory
  - **A reference bit** -> track whether a page has been accessed



An x86 page table entry (PTE)

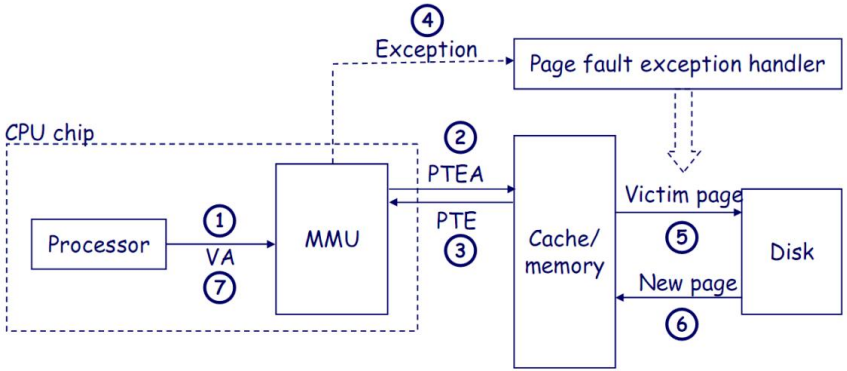
Page hit

- 1) Processor sends virtual address to MMU
- 2 – 3) MMU fetches PTE from page table in memory
- 4) MMU sends physical address to L1 cache
- 5) L1 cache sends data word to processor



Page fault

- 1) Process sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in memory
- 4) Valid bit is 0, so MMU triggers page fault exception
- 5) Handler identifies victim, and if dirty pages it out to disk
- 6) Handler pages in new page and updates PTE in memory
- 7) Handler returns to original process, restarting faulting instruction



Slow paging

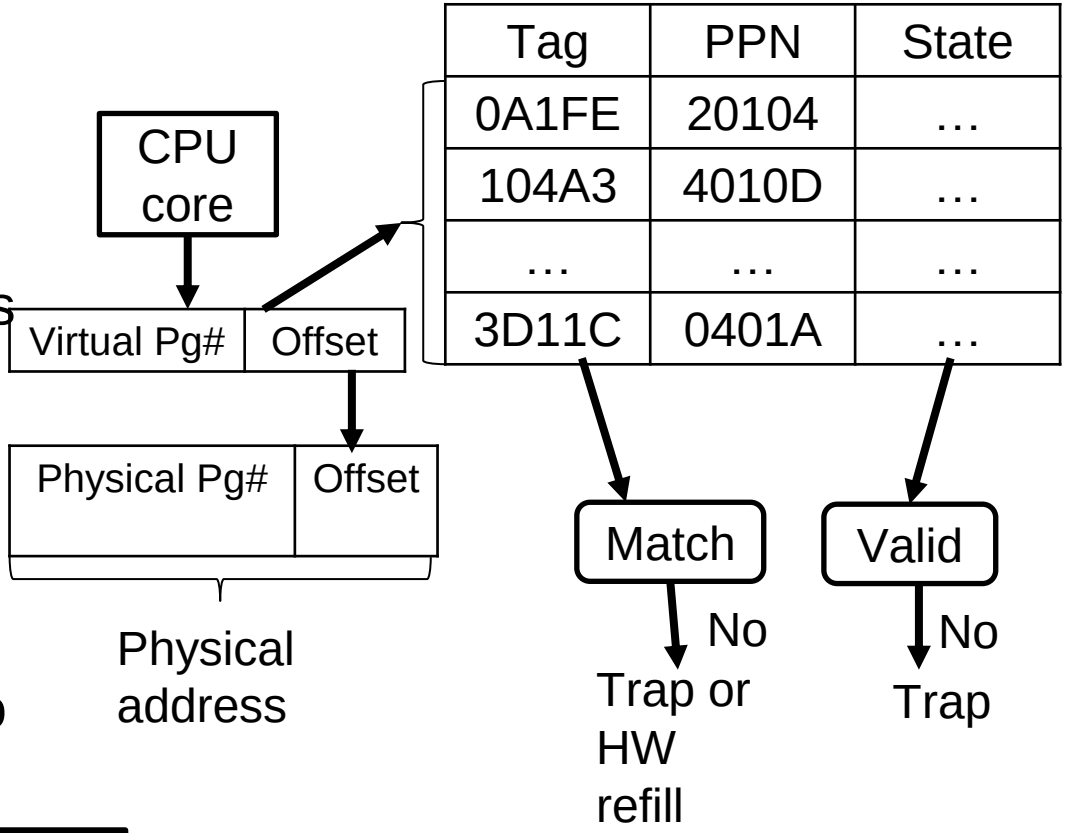
- Require a large amount of mapping information
  - The mapping information is stored in physical memory
  - Paging logically requires an **extra memory lookup for each virtual address** generated by the program
  - Going to memory for translation information before every instruction fetch
  - **Explicit load or store** is prohibitively slow

Paging unit

- CPU issues load/store

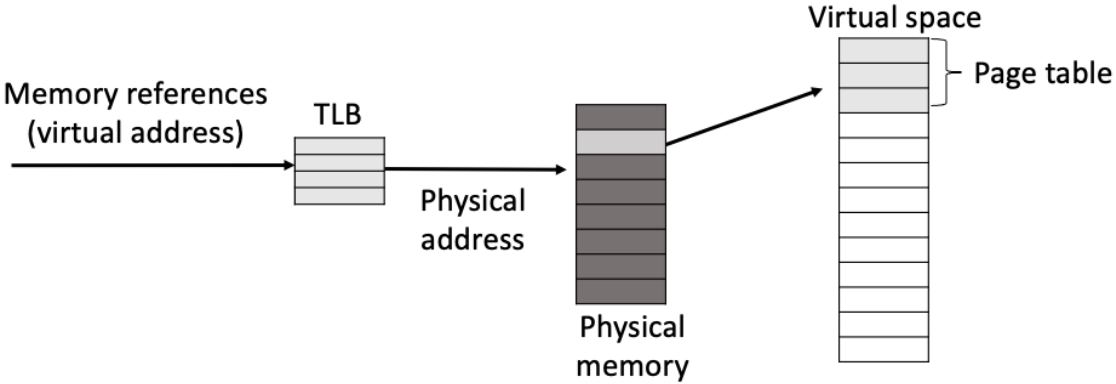
1. Compare VPN to all TLB tags
2. If no match, need TLB refill
   a. SW -> trap to OS
   b. HW -> HW table walker
3. Check if VA is valid
   a. if not, generate trap
4. Concatenate PPN to offset to form physical address

Does all needs to be very fast ?



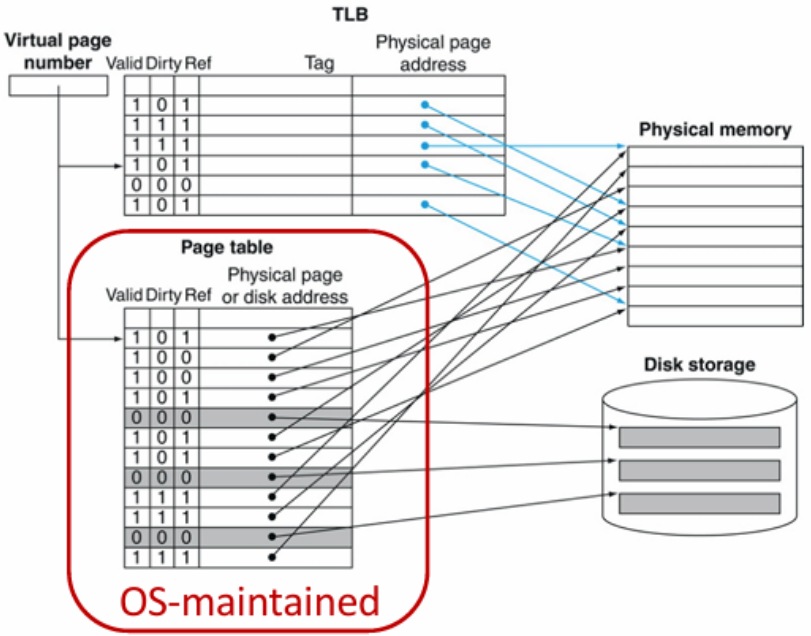
Translation lookaside buffer (TLB)

- **Translation lookaside buffer (TLB)**
  - Reduce memory reference time if page tables stored in hardware
  - A hardware cache of popular virtual-to-physical address translation
  - **Caching of the page table**



TLB (cont.)

- TLB is a dedicated cache for recently accessed page table entries
- Page tables (per process) are stored in physical memory
  - Starting address of the table for the currently executing process is in the page table register (PTR)
  - Memory management unit (MMU) uses the PTR to walk the page table (TLB miss)
- **TLB is exposed to the OS**
  - Many scenarios require the OS to flush an entry (or entries) from the TLB
    - Page replacement
    - Page remapping (in multicores, this results in a *TLB shootdown*)



TLB (cont.)

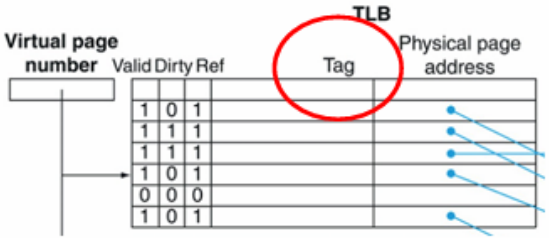
- What are typical TLB sizes and organization ?
  - Usually small: 16 – 512 entries
  - Fully associative, sub-cycle access latency
    - Lookup is by virtual address
    - Return physical address + other info
  - TLB misses take 10-100 cycles
  - Search the **entire TLB** in parallel to find the desired translation
  - Why is TLB often fully associative ?
- What happens when fully-associative is too slow ?
  - Put a small (4 – 16 entry) direct-mapped cache in front
  - Called a “TLB slice”

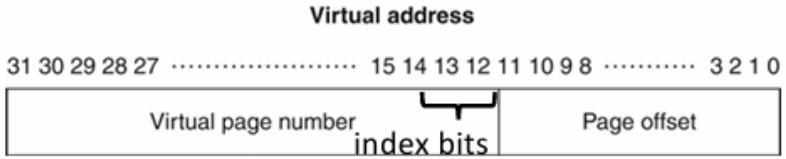
Question

- What happens to the cache contents of that page in the processor cache when we move a page to the swap area on disk?
  - The OS flushes the contents from the cache when it decides to migrate a page to disk
  - The attempt to access any data from that virtual page generate a page fault

Tag Bits in the TLB

- How many tag bits in each TLB entry?
  - 32-bit system, 4 KB page size, 256-entry fully-associative TLB
    - Need to search every entry in the TLB for a virtual page
    - 20 bits for the tag
  - 32-bit system, 4 KB page size, 256-entry direct-mapped TLB
    - 8 bits for indexing the TLB
    - 12 bits for the tag





Note: If the virtual page and the physical frame are both the same size (4 KB), then the low-order 12 bits of the physical address remain unchanged after the translation from the virtual address

TLB organization

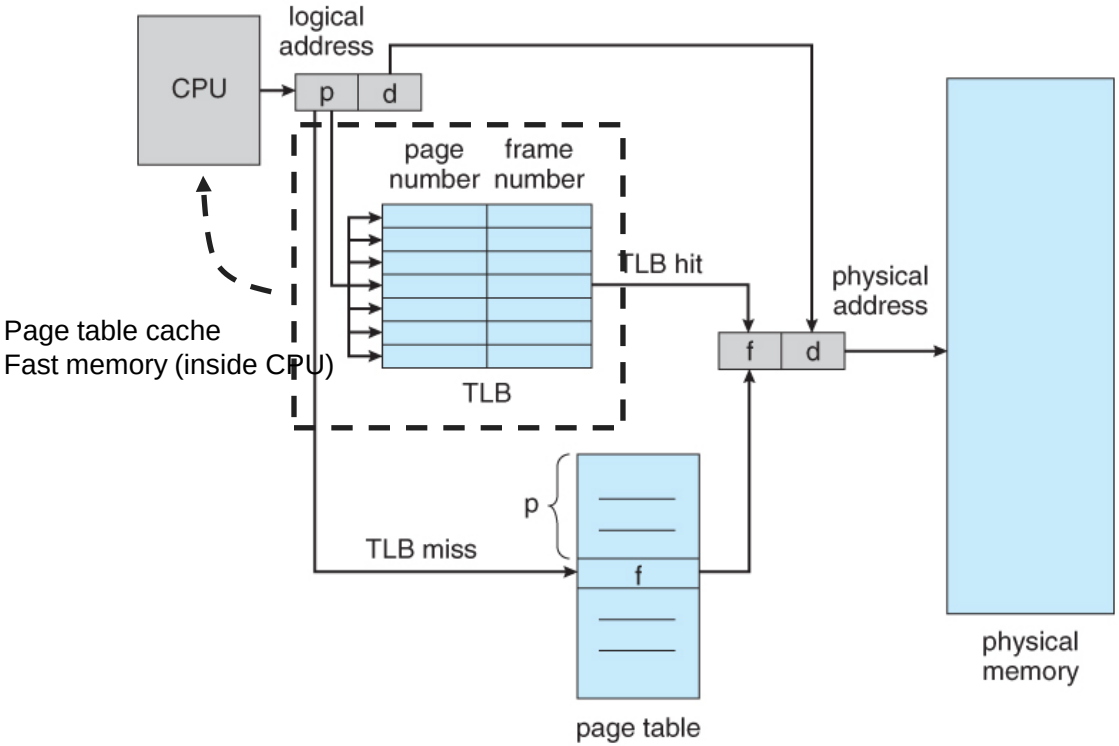
- TLB entry
  - Tag is virtual page and data is PTE for that tag
  - **Dirty** is marked when the page has been written to
  - **Coherence bit** determines how a page is cached by the hardware
  - **Valid bit** tells the hardware if there is a valid translation
  - **Address space identifier (ASID)** as a process identifier (PID)

TLB organization

- TLB entry

Virtual address	Physical address	Dirty	Coherence	Valid	Access	ASID
0xFA00	0x0003	Y	N	Y	R/W	34
0x0040	0x0010	N	Y	Y	R	0

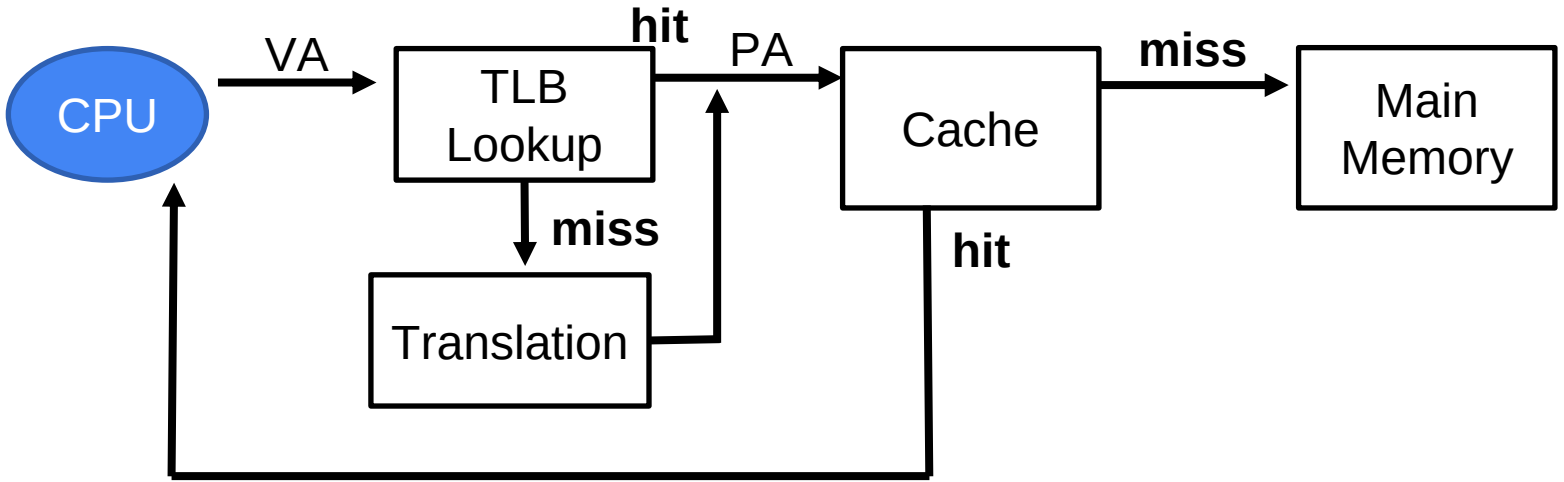
Paging with TLB



Source: Operating System Concepts by Abraham Silberschatz, Greg Gagne, Peter B. Galvin

Translation with a TLB

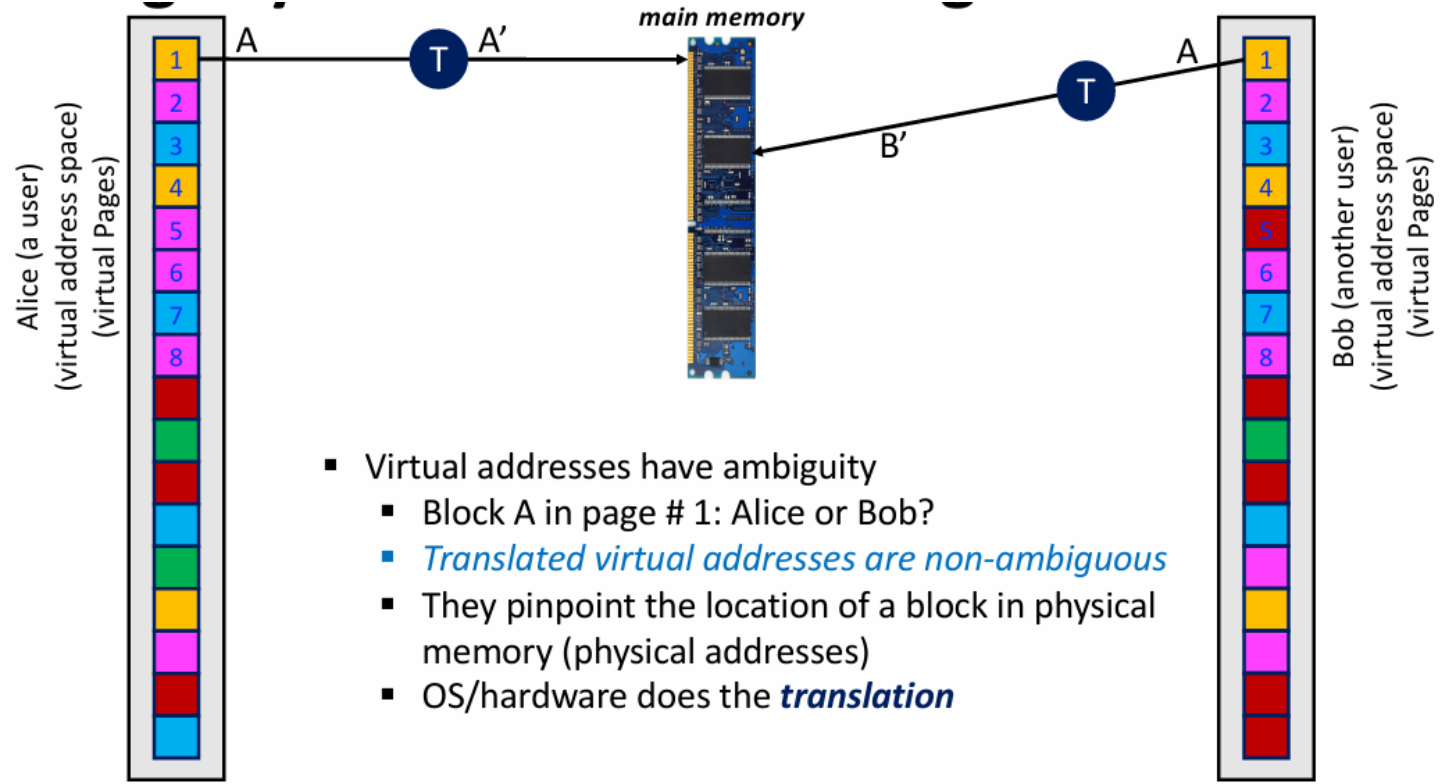
- Overlap the cache access with the TLB access
  - High order bits of the VA are used to look in the TLB
  - Low order bits are used as index into cache



Cache Access with Virtual Memory

- **Question:** To use virtual or physical address to index the cache?
  - Physical Indexed and Physically Tagged cache (PIPT)
    - TLB access is on the critical path
  - Virtually Indexed and Virtually Tagged cache (VIVT)
    - Two virtual pages (different processes, Alice & Bob) mapped to same physical address
    - Two virtual pages (one process, Alice) mapped to the same physical page

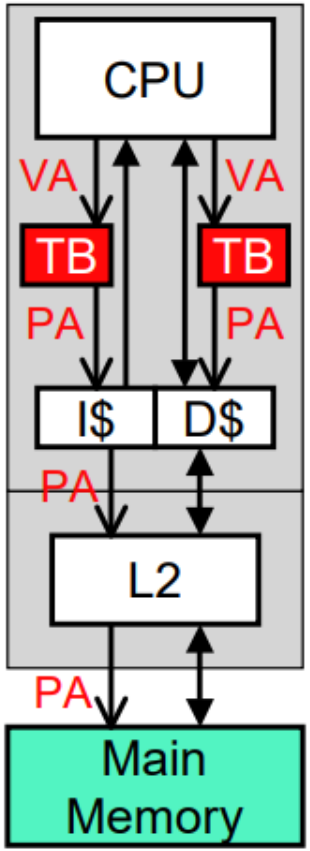
Ambiguity in Virtual Addressing



- Virtual addresses have ambiguity
  - Block A in page # 1: Alice or Bob?
  - *Translated virtual addresses are non-ambiguous*
  - They pinpoint the location of a block in physical memory (physical addresses)
  - OS/hardware does the ***translation***

Physical caches

- **Physical caches**
  - Indexed and tagged by PAs
  - Translate to PA at the outset
  - No need to flush caches on process switches
    - Processes do not share PAs
  - Cached inter-process communication works
    - Single copy indexed by PA
    - Slow: adds 1 cycle to t_{hit}

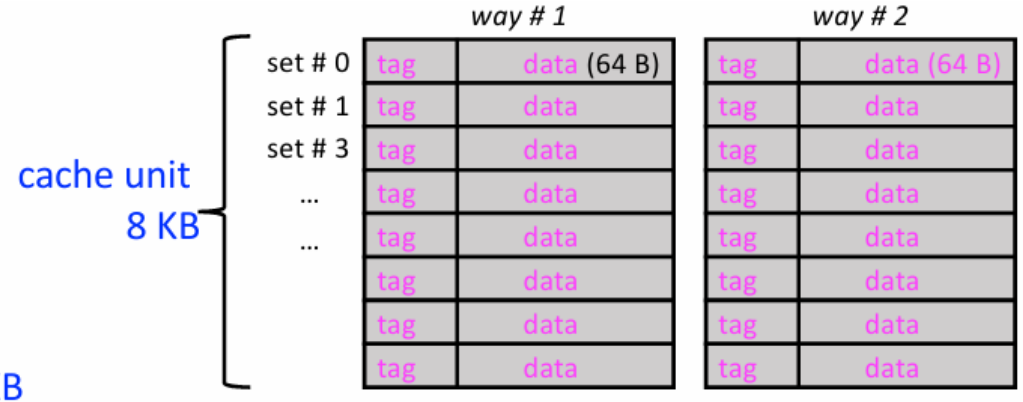


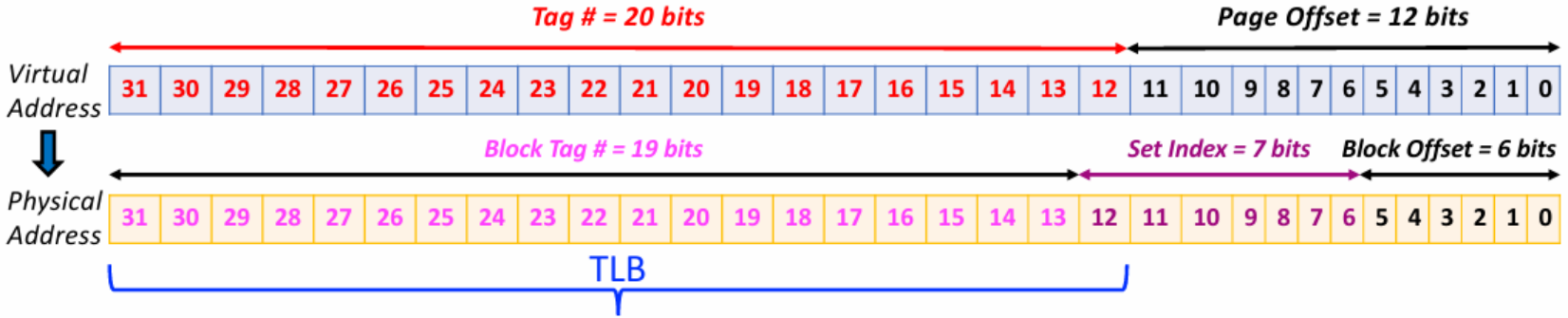
http://people.ee.duke.edu/~sorin/prior-courses/ece152-spring2008/lectures/6.9-memory.pdf

PIPT (2-Way Set Assoc)

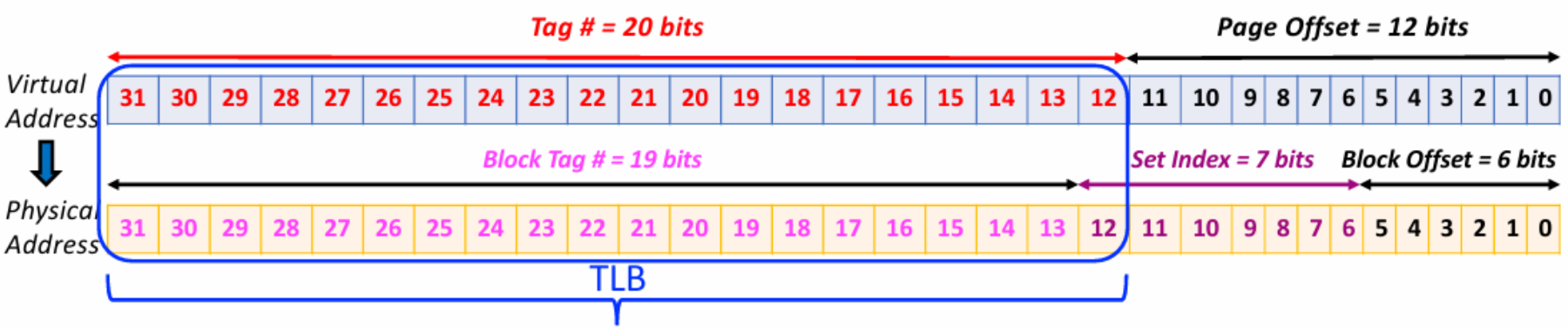
System specs
- 32 bit addresses
- Page size = 4 KB
- 4 GB physcial memory

Hypothetical cache
- Block size = 64 bytes
- # sets = 128
- # sets X block size = 8 KB
- Two pages fit in one unit/way





PIPT-Removing Ambiguity



- Size of virtual and physical page is 4 KB
  - The low-order 12 bits do not change after translation
  - The page offset is used to index into the page to access the word/byte
- We must remove the ambiguity in the 13th bit and wait for the translation
  - Two processes may have the same 13th bit in the virtual addresses but not in their physical addresses

PIPT

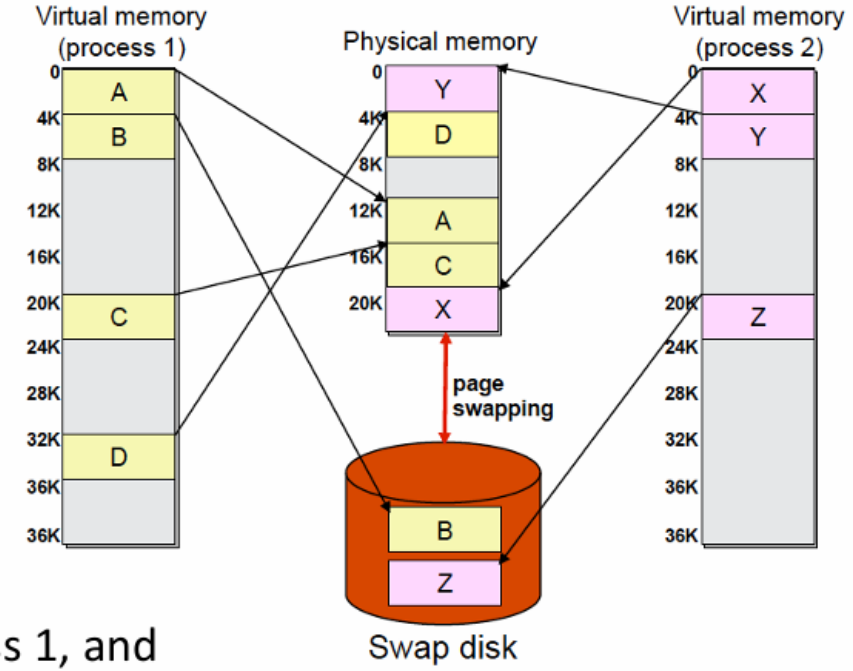
- **Advantage (Will become clear once we see other indexing schemes)**
  - No constraint on the size of the cache unit
    - (# sets X block size) *can exceed the 4 KB page size*
  - Multiple pages can fit inside a single unit
    - An easy way to build large caches without increasing associativity
- **Disadvantage**
  - TLB access and cache access are serialized
  - Must get translation from TLB to find the correct set in the cache

Homonym and Synonym

- **Homonym**
  - Two similar virtual addresses from different processes with different physical addresses
  - Homonyms in English: same spelling, different meaning
- **Synonym**
  - Two different virtual addresses (same process) with the same physical address
  - Also called virtual aliases

Example of Homonym

- Example 4K page size
- Process 1 has pages A, B, C and D
- Page B is held on disk
- Process 2 has pages X, Y, Z
- Page Z is held on disk
- Process 1 cannot access pages X, Y, Z
- Process 2 cannot access page A, B, C, D
- OS can access any page (full privileges)

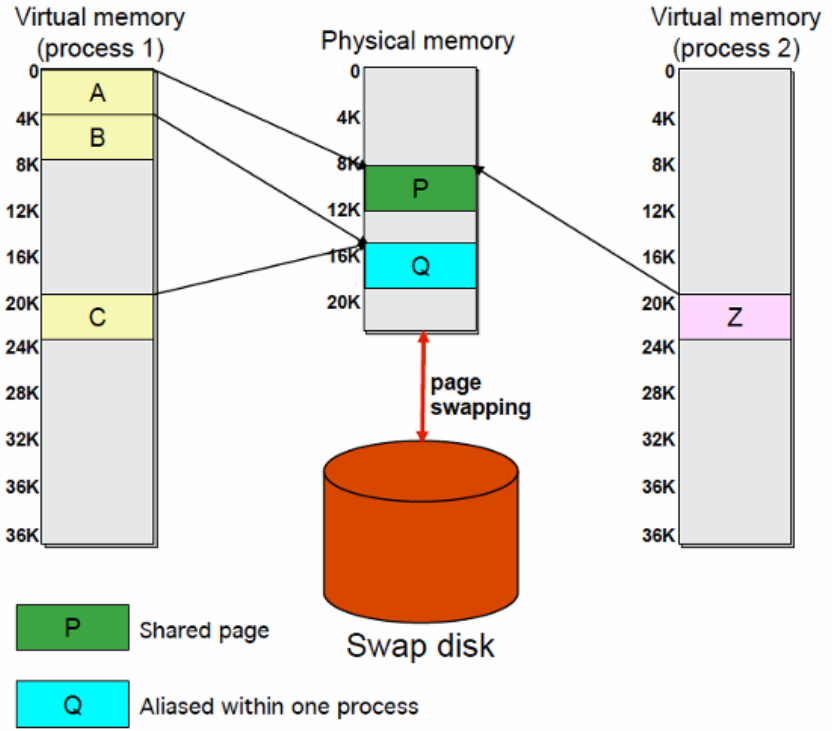


Key idea: Virtual page 0 of process 1, and virtual page 0 of process B are homonyms

Example of Synonym

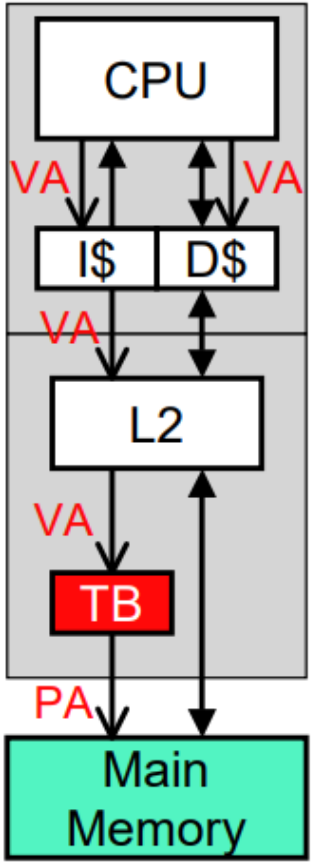
- Process 1 and Process 2 want to share a page of memory
- Process 1 maps virtual page A to physical page P
- Process 2 maps virtual page Z to physical page P
- Permissions can vary between the sharing processors.
- **Note:** Process 1 can also map the same physical page at multiple virtual addresses

Key idea: Virtual pages B and C are **synonyms**



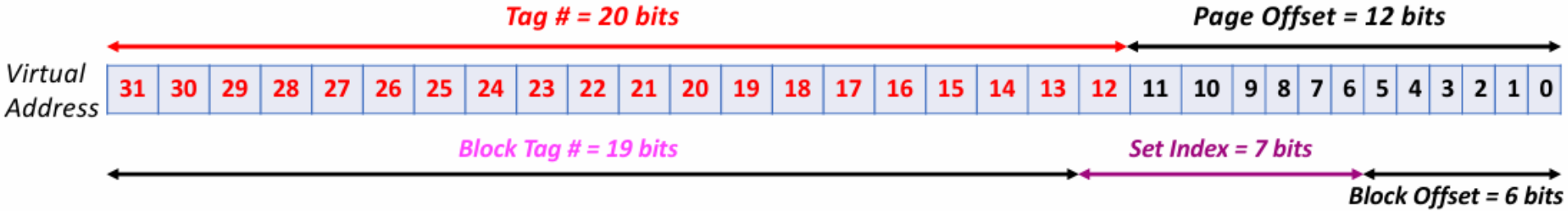
Virtual caches

- **Virtual cache**
  - Tags in cache are virtual addresses
  - Translation only happens on cache misses
- **What to do on process switches ?**
  - Flush caches ? Slow
  - Add process IDs to cache tags
- **Does inter-process communication work ?**
  - Aliasing: multiple VAs map to same PA
  - How are multiple cache copies kept in sync?
  - Disallow caching of shared memory ? Slow



http://people.ee.duke.edu/~sorin/prior-courses/ece152-spring2008/lectures/6.9-memory.pdf

VIVT



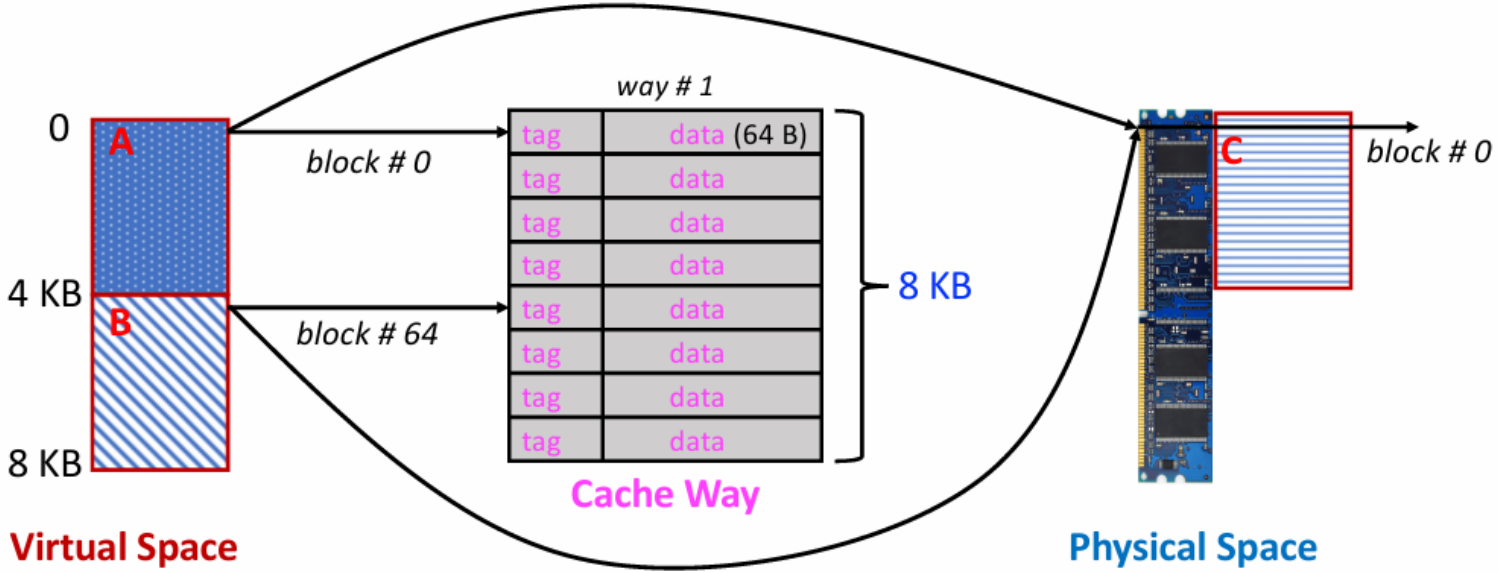
Two problems

- Virtual aliasing due to multiple users
  - *Cache flushing on a context switch solves this problem*
- Virtual aliasing due to multiple virtual pages mapped to a single physical page
  - Two copies of the same physical block in the cache (correctness problem)
  - How can we solve this problem?

Synonyms (Aliases) Problem

Synonyms (aliases)

- When the OS/program uses two virtual addresses for the same physical address
  - Allowed behavior with some practical uses
- Duplicate addresses are aliases of each other
- If one is modified, then the other has wrong data (impossible with a physical cache)

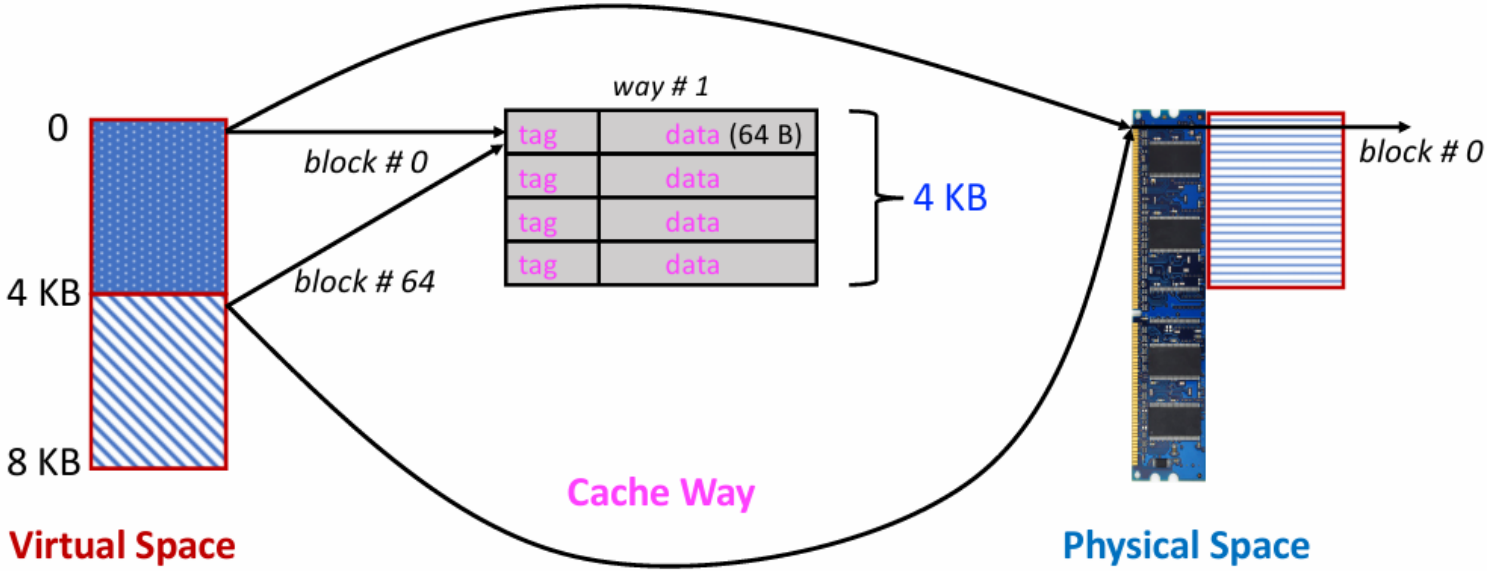


Some solutions to the synonym problem

- Limit cache size to (page size x associativity)
  - Limit the cache unit size (# sets x block size) to be 4KB
- On a write to a block, search all possible indices that can contain the same physical block, and update/invalidate
  - Used in Alpha 21264, MIPS R10K
- Restrict page placement in OS
  - Make sure index(VA) = index(PA)
  - Called page coloring
  - Used in many SPARC processors

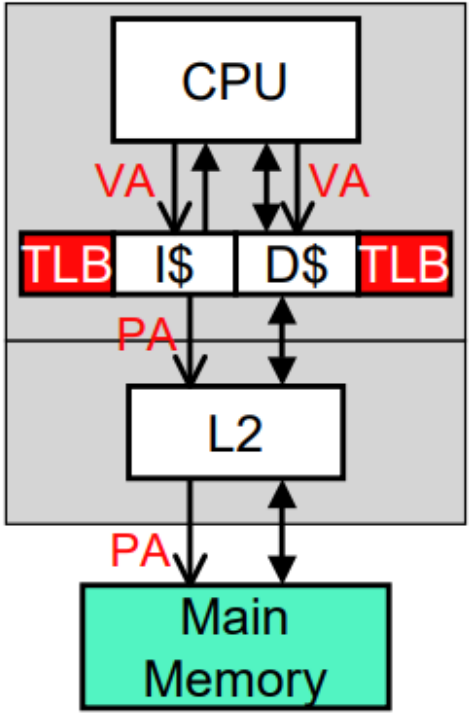
Dealing with Synonyms

Dealing with synonyms comes with a limitation on cache design

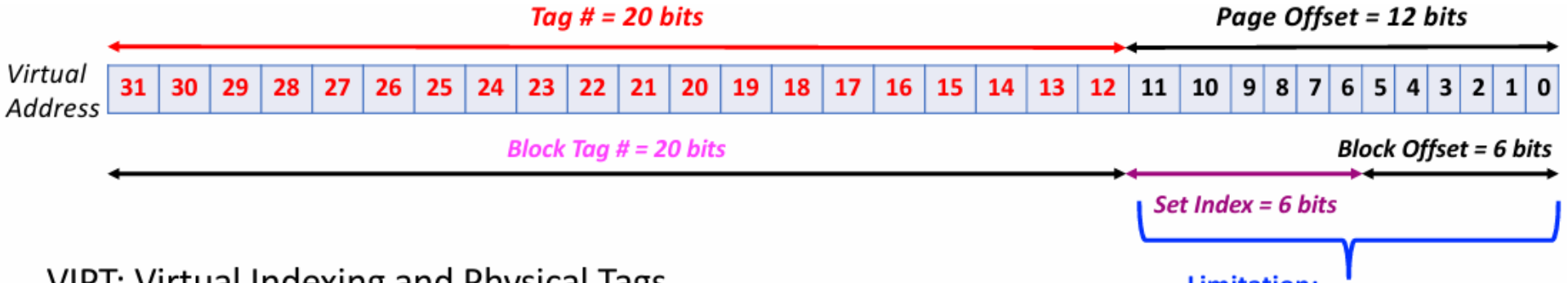


Virtual physical caches

- Virtual-physical caches
  - Indexed by VAs
  - Tagged by PAs
  - Cache access and address translation in parallel
  - No context-switching/aliasing problems
  - Fast: no additional t_{hit} cycles



VIPT

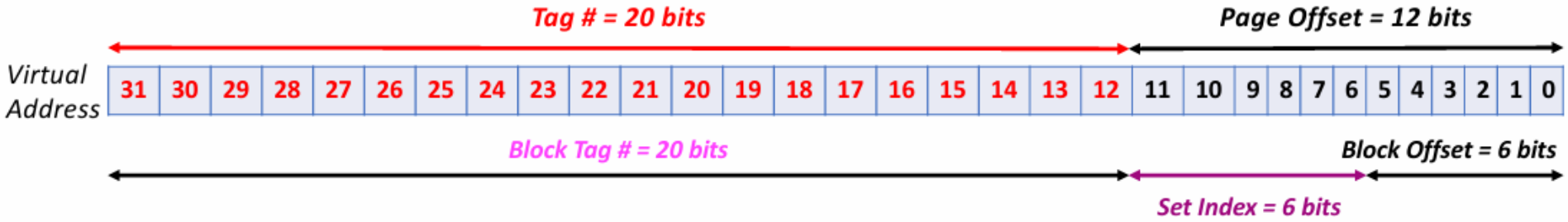


VIPT: Virtual Indexing and Physical Tags

- Use virtual bits of the page offset to index the cache
  - Limitation: Cache unit size cannot exceed 4 KB
- Use physical tags to eliminate ambiguity due to virtual aliases (multiple users)
  - *Better alternative to flushing on context switches*

Limitation:
sets * line-size cannot be greater than page size

VIPT



Operation of a cache with VIPT

- Use the page offset to index the cache (**step # 1**)
- In parallel, get the translation from the TLB (**step # 1**)
- Then, compare the physical tag to the translated portion of the address (**step # 2**)

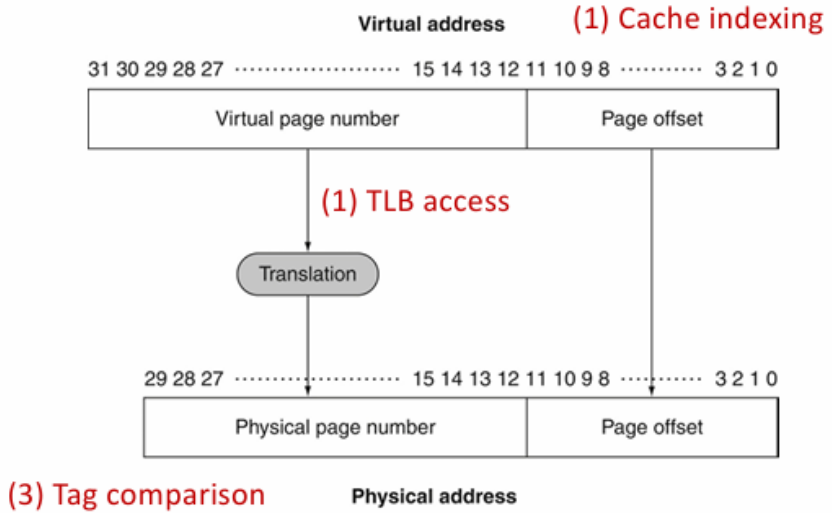
} **Overlapped in time**

VIPT

Operation of a cache with VIPT

- Use the page offset to index the cache (**step # 1**)
- In parallel, get the translation from the TLB (**step # 1**)
- Then, compare the physical tag to the translated portion of the address (**step # 2**)

Overlapped in time



Summary

- Virtual memory
  - Page tables and address translation
  - Page fault and handling
- Translation lookaside buffer (TLB)
  - Reduce the overhead of paging
  - Must be fast
  - Virtual, physical, and virtual-physical caches and TLBs

Takeaway Questions

- Which fragmentation is created by the data segment?
  - (A) External fragmentation
  - (B) Internal fragmentation
  - (C) None of them
- What are benefits of paging?
  - (A) No internal fragmentation
  - (B) No external fragmentation
  - (C) Efficient memory translation

Takeaway Questions

- Where can we find the page table?
  - (A) Data cache
  - (B) Main memory
  - (C) Hard disk drive
- What are inside a TLB?
  - (A) Virtual memory page
  - (B) Page entry table
  - (C) Physical memory frame