Operating System Design and Implementation

Lecture 11: Virtual memory

Tsung Tai Yeh

Tuesday: 3:30 – 5:20 pm

Classroom: ED-302

Acknowledgements and Disclaimer

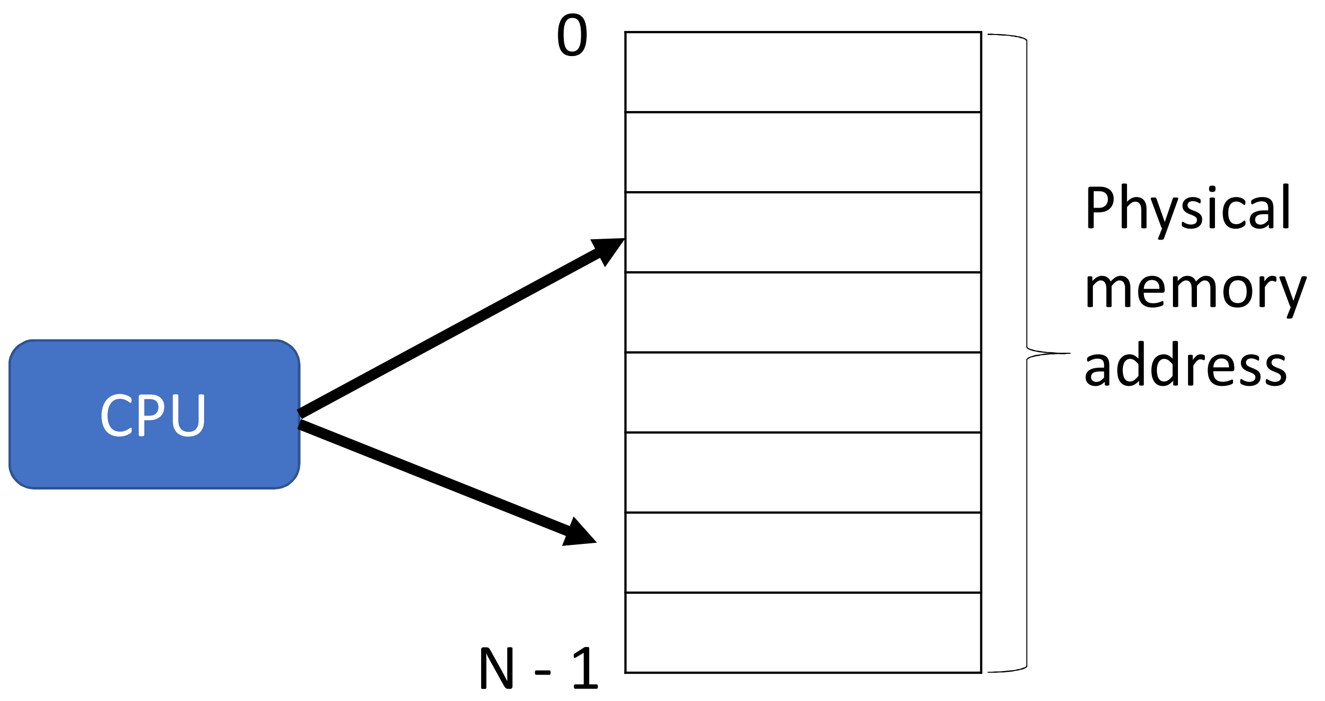
- Slides was developed in the reference with
  MIT 6.828 Operating system engineering class, 2018
  MIT 6.004 Operating system, 2018
  Remzi H. Arpaci-Dusseau etl. , Operating systems: Three easy pieces. WISC
  Onur Mutlu, Computer architecture, ece 447, Carnegie Mellon University

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



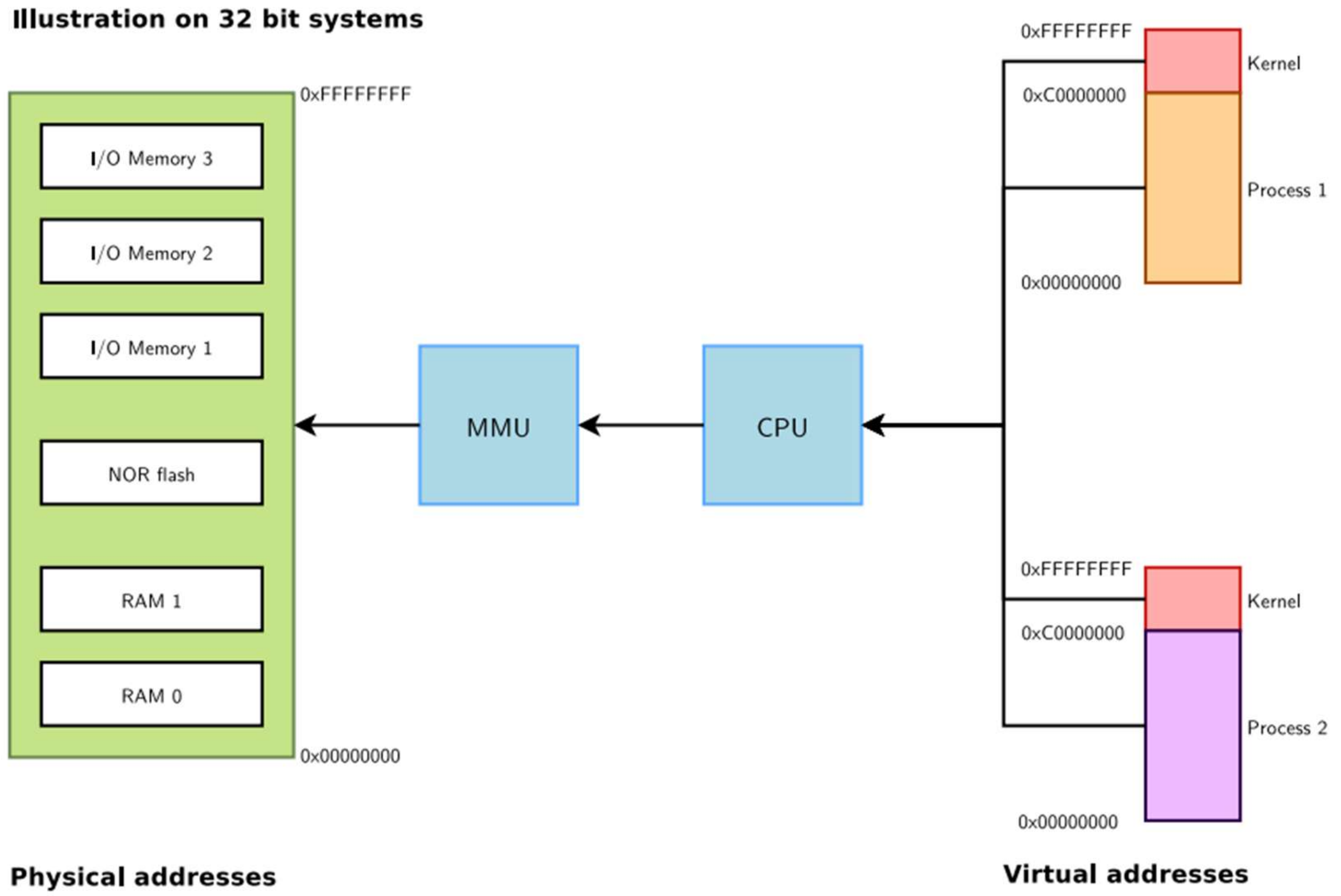
Problems of physical memory addressing

- 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



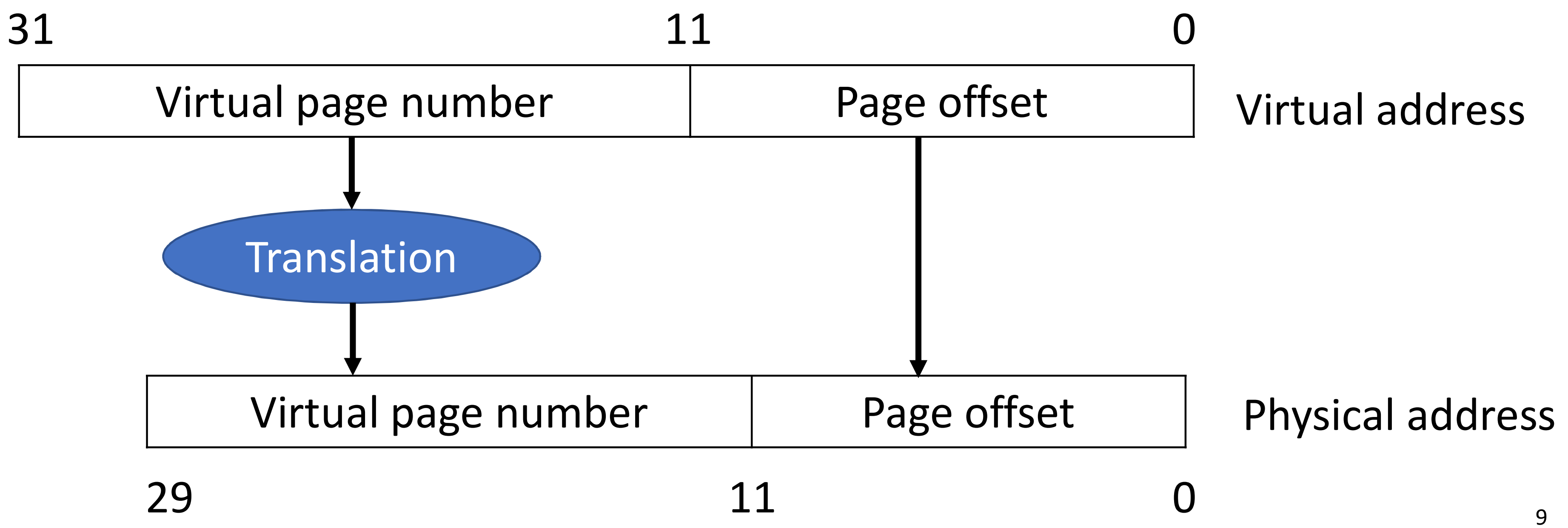
https://bootlin.com/doc/training/linux-kernel/linux-kernel-slides.pdf

Advantages of virtual memory

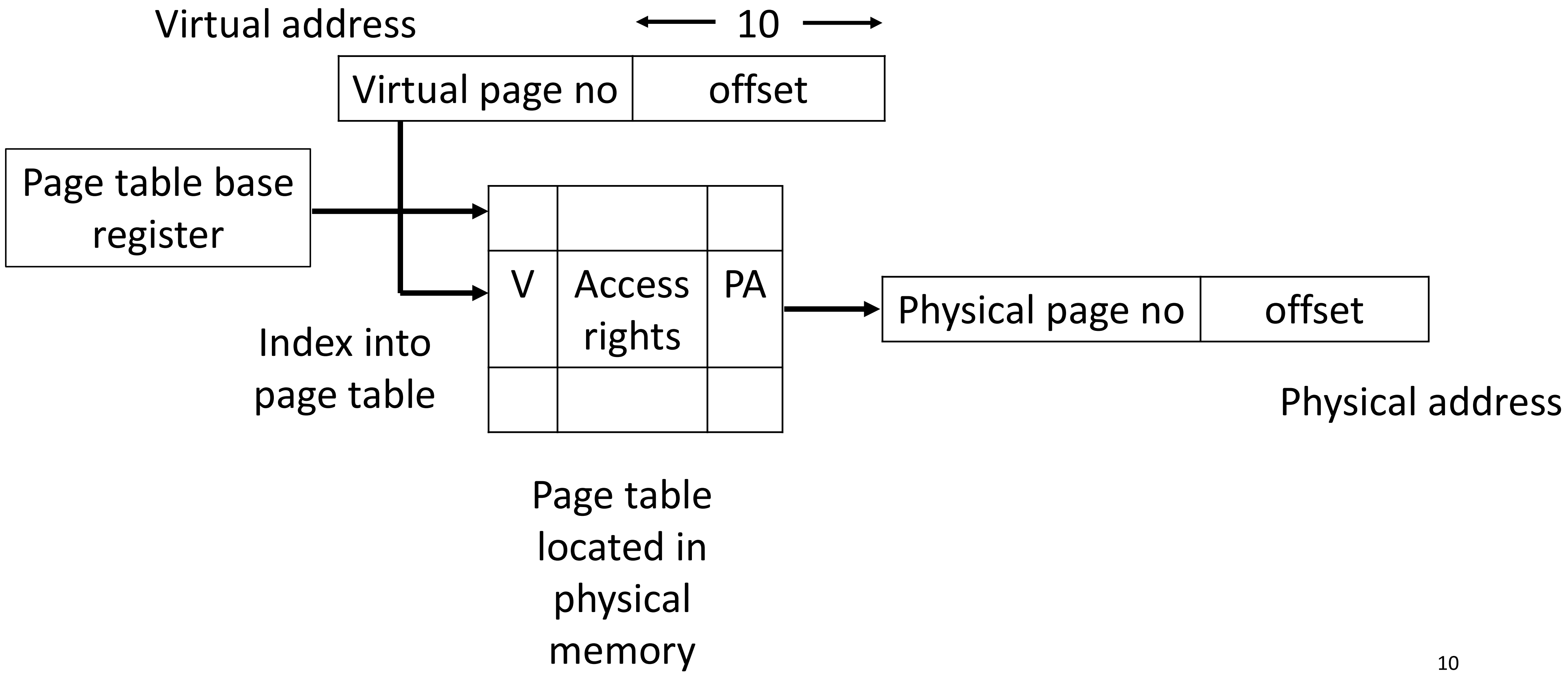
- **Illusion of having more physical memory**
- **Multiple programs share the physical memory**
  - Permit sharing without knowing other programs
  - Division of memory among programs is automatic
- **Program relocation**
  - Program addresses can be mapped to any physical location
  - Physical memory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

A system with virtual memory (page based)

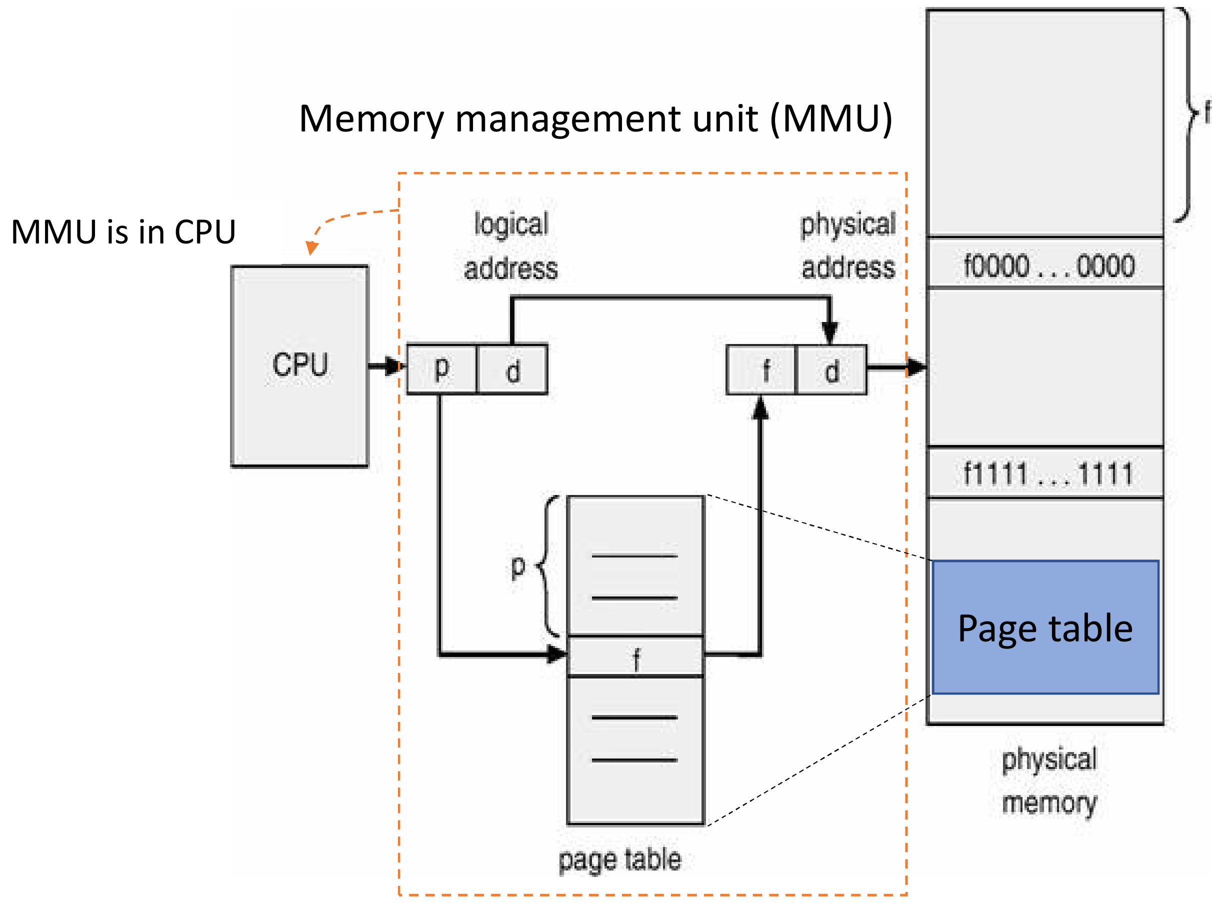
- **Address translation**
  - 4GB virtual memory, 1GB physical memory, page size is 4KB (2^{12}) with 2^{18} physical pages



Page tables for address translation



Page memory management



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 disk, otherwise
- **Page table**
  - Stores the mapping of virtual pages to physical frames
  - Page table is in memory

Paging

- Why does segmentation 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.)

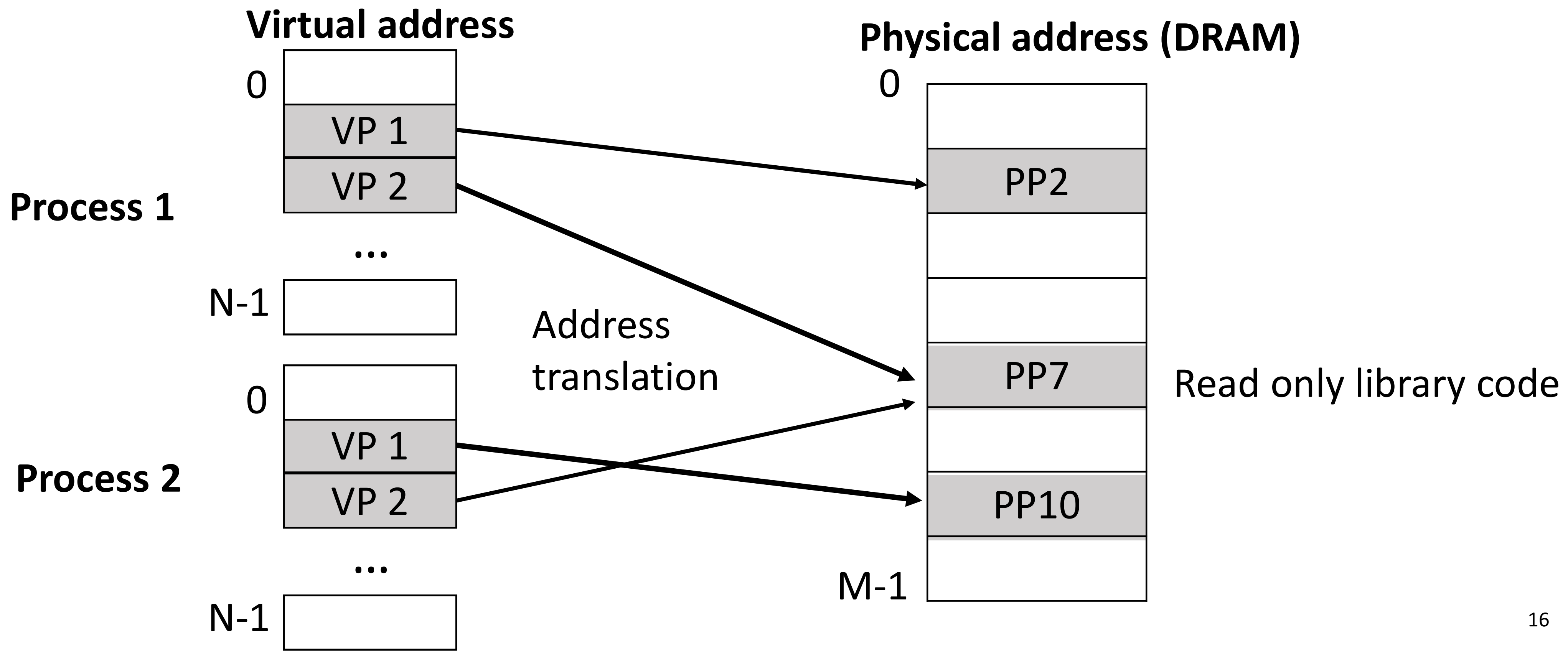
- Modern hardware and OS use paging
- **Pages** are like segments, but **fixed size**
  - The bounds register goes away
  - External fragmentation goes away
- Since pages are small (4 or 8 KB, often), a lot of them
  - So page table has to go into RAM
  - Page table might be huge
  - Accessing the page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

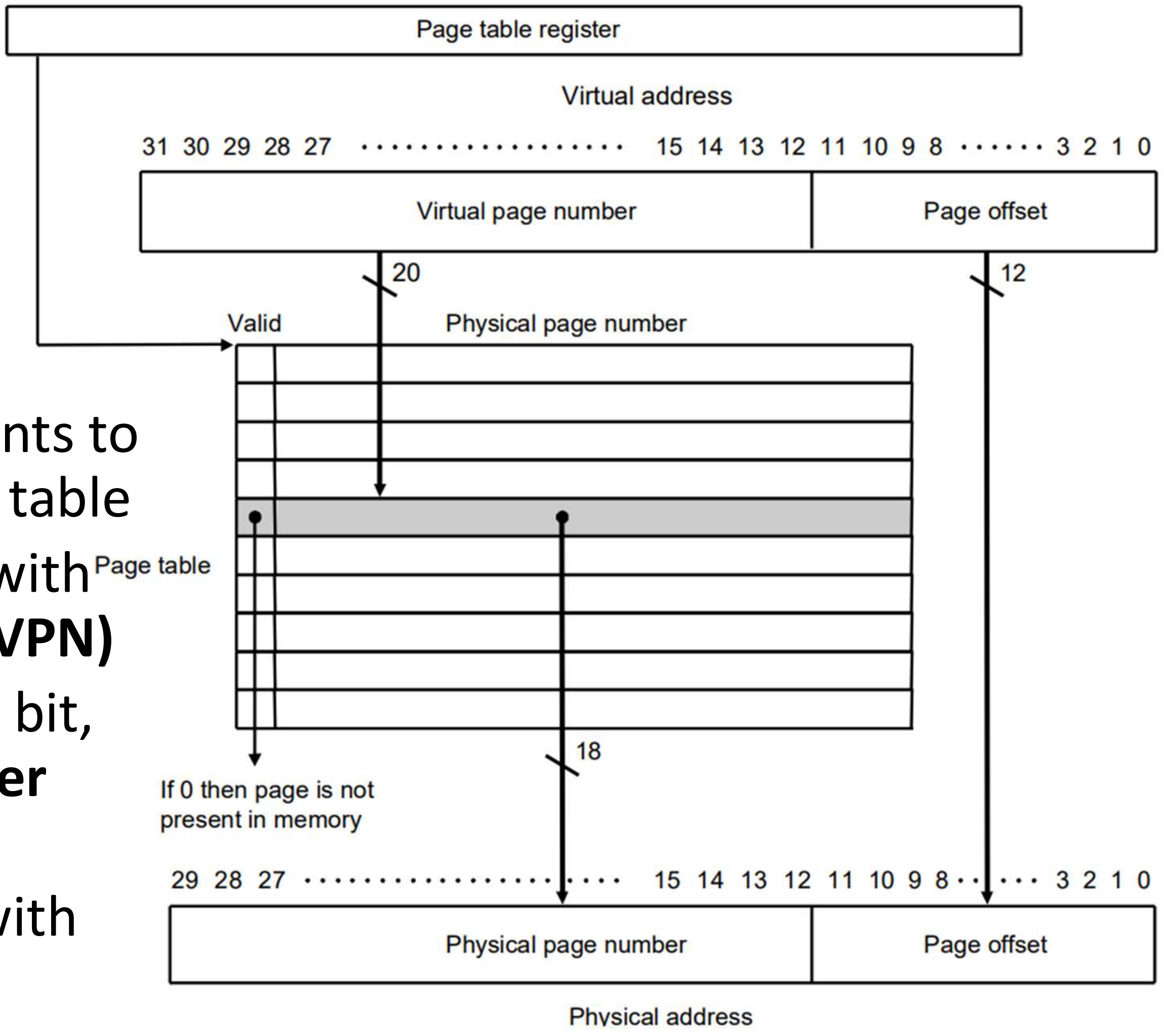
Page table is per process

- 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

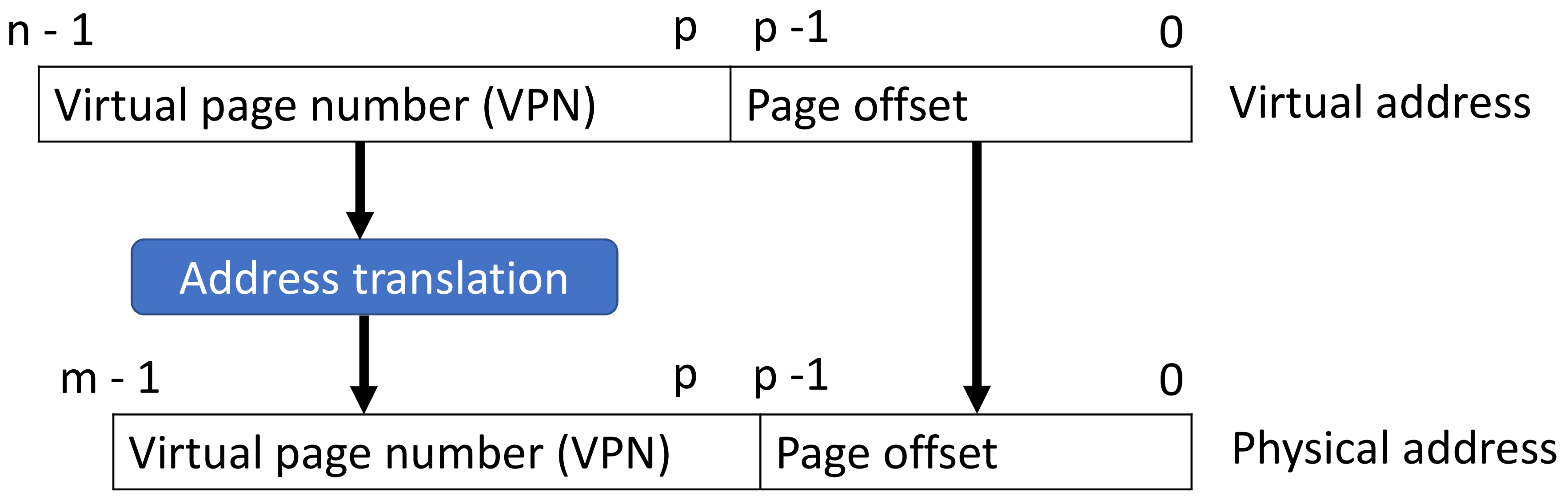


https://courses.cs.washington.edu/courses/cse378/09wi/lectures/lec20.pdf

Page table

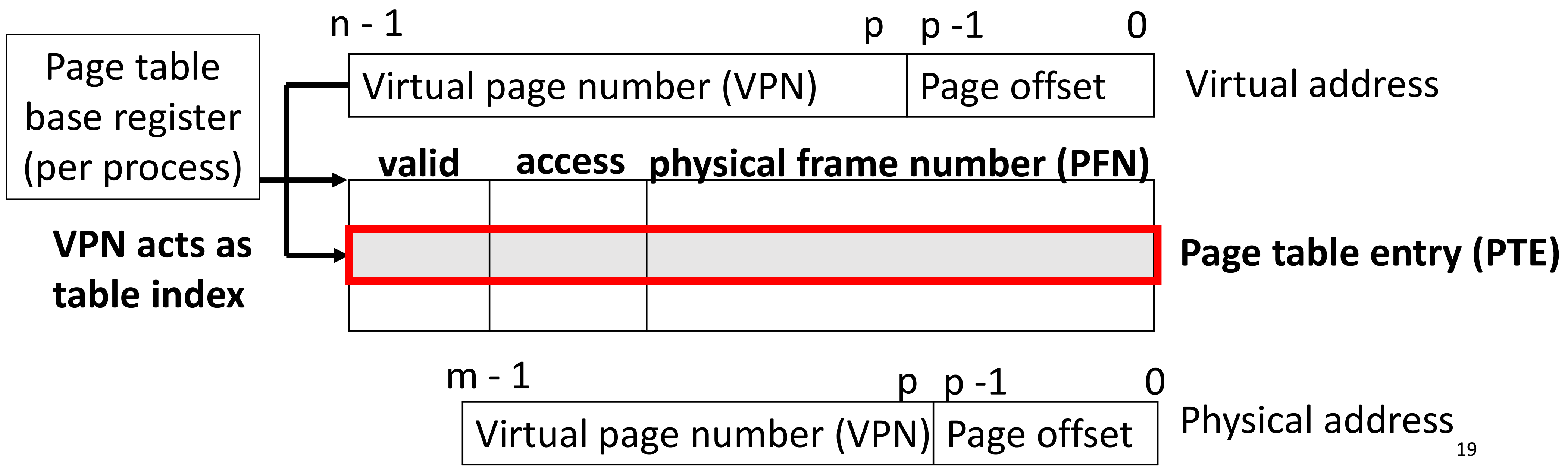
- Parameters
  - $P = 2^p$ = page size (bytes)
  - $N = 2^n$ = virtual-address limit
  - $M = 2^m$ = physical address limit

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

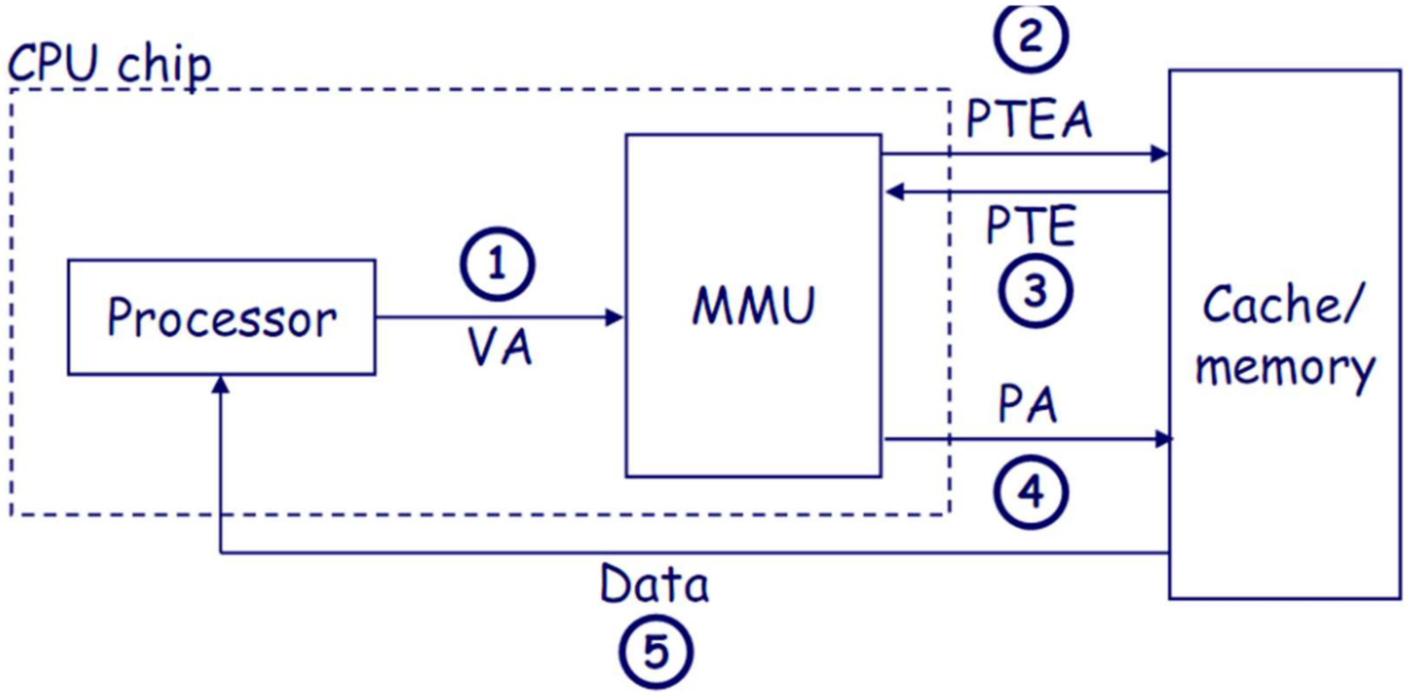


Page table (cont.)

- **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)

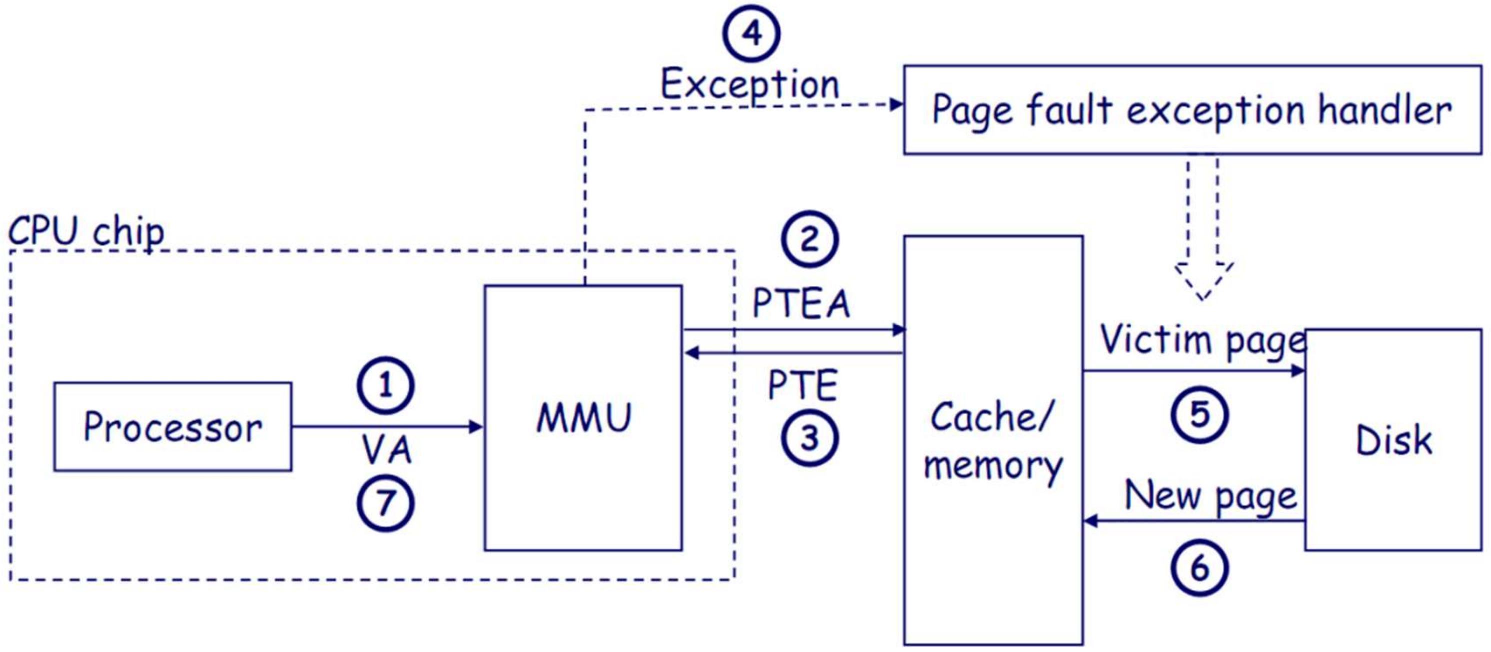


Page hit



- 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

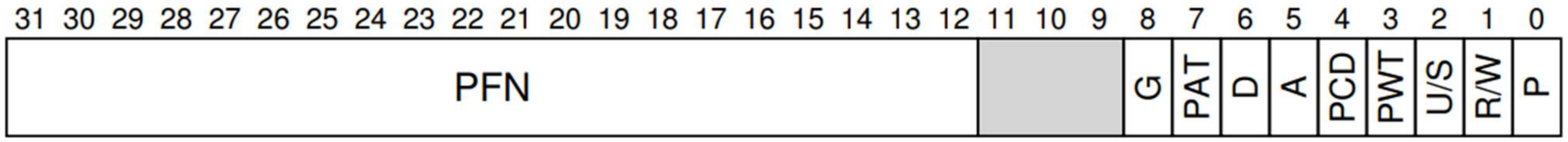
Page fault



- 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

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)

Slow paging

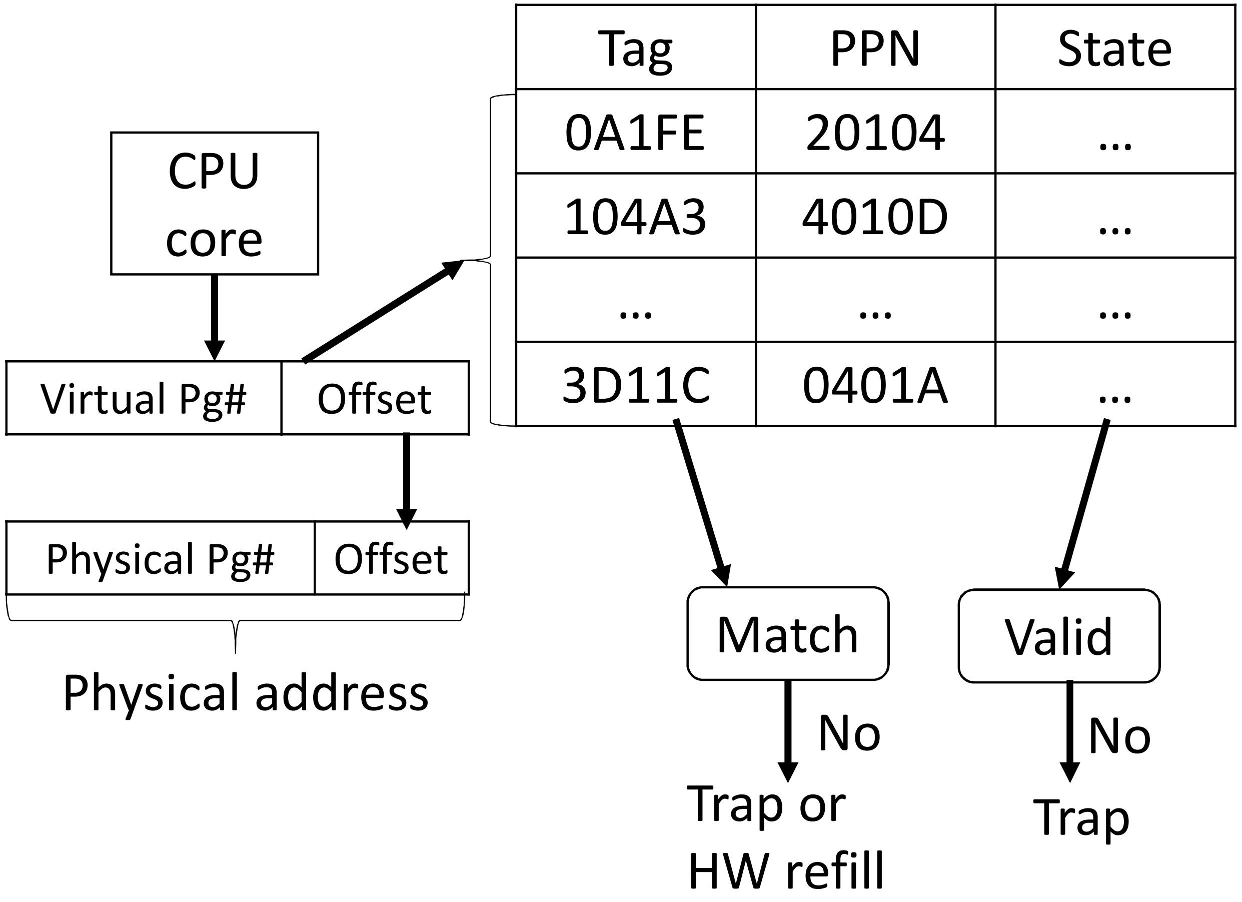
- Require a large amount of mapping information
  - The mapping information is stored in physical memory
  - Paging logically requires an **extra memory lookup for each virtual address** generated by the program
  - Going to memory for translation information before every instruction fetch
  - **Explicit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

Paging unit

- CPU issues load/store

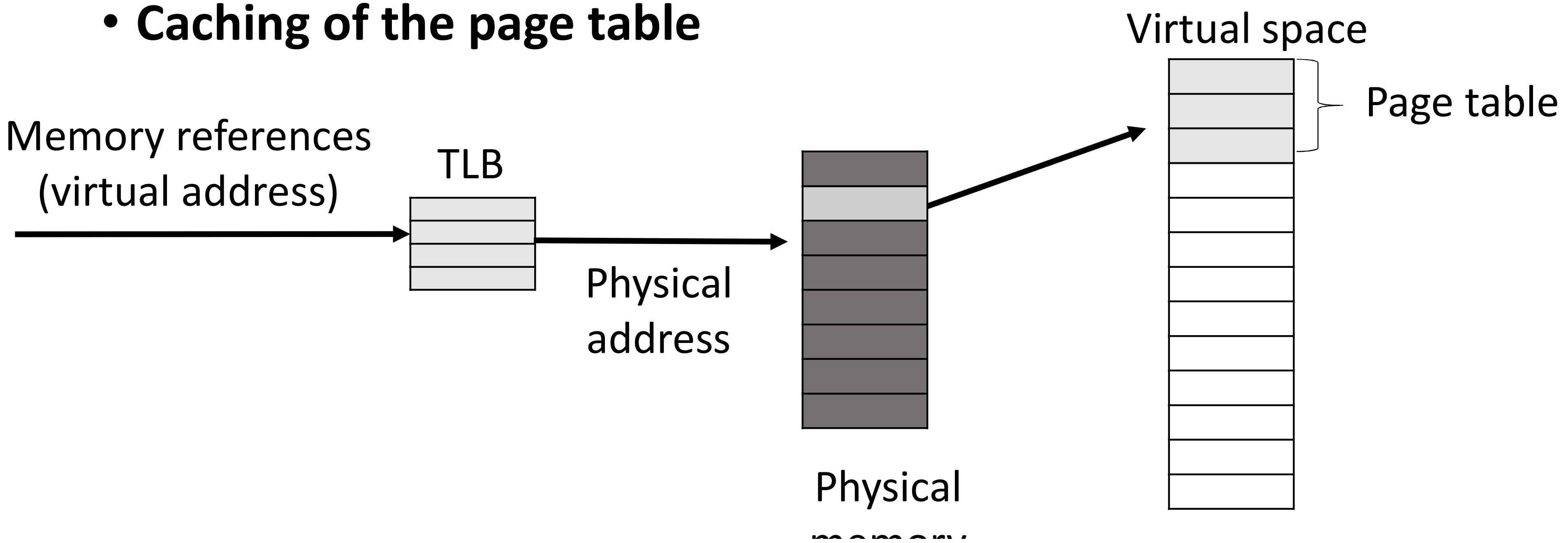
1. Compare VPN to all TLB tags
2. If no match, need TLB refill
   a. SW -> trap to OS
   b. HW -> HW table walker
3. Check if VA is valid
   a. if not, generate trap
4. Concatenate PPN to offset to form physical address

Does all needs to be very fast ?

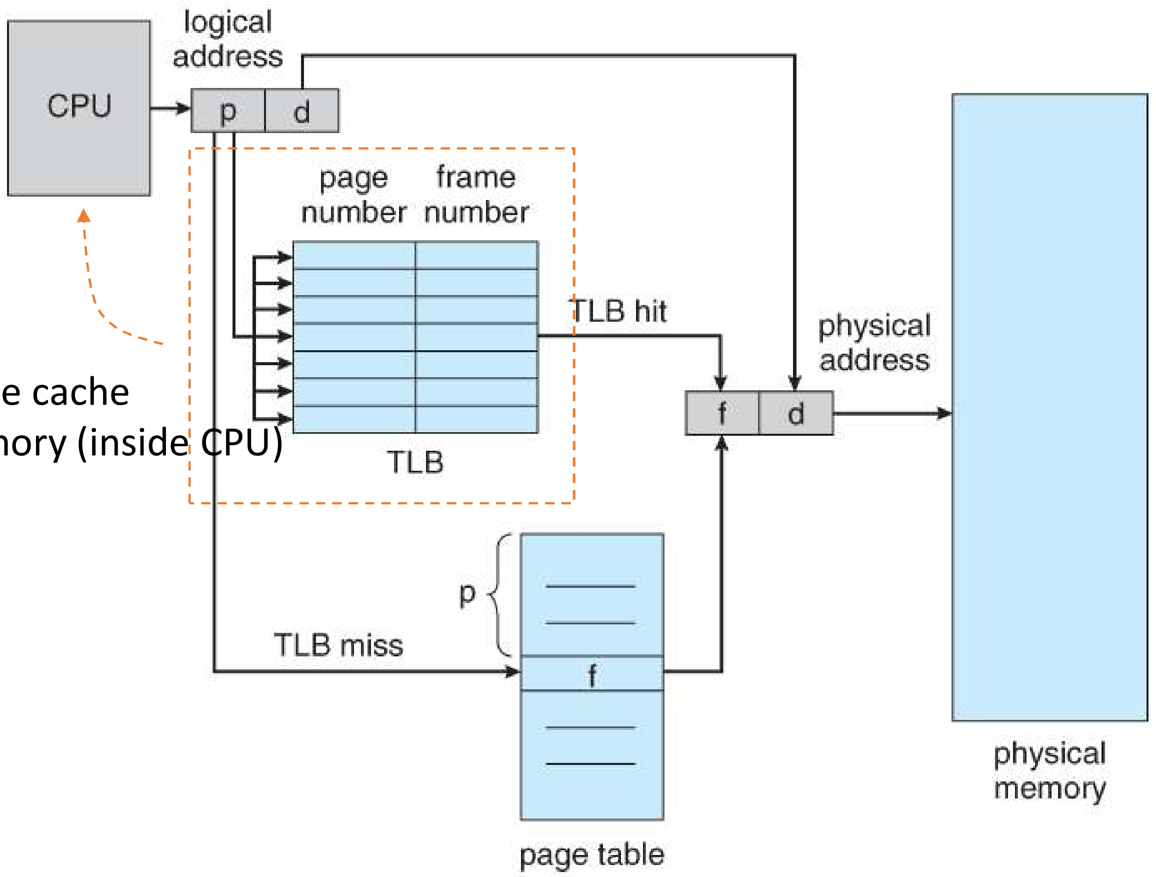


Translation lookaside buffer (TLB)

- **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**



Paging with TLB



Page table cache
Fast memory (inside CPU)

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

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”

TLB organization

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

- 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)

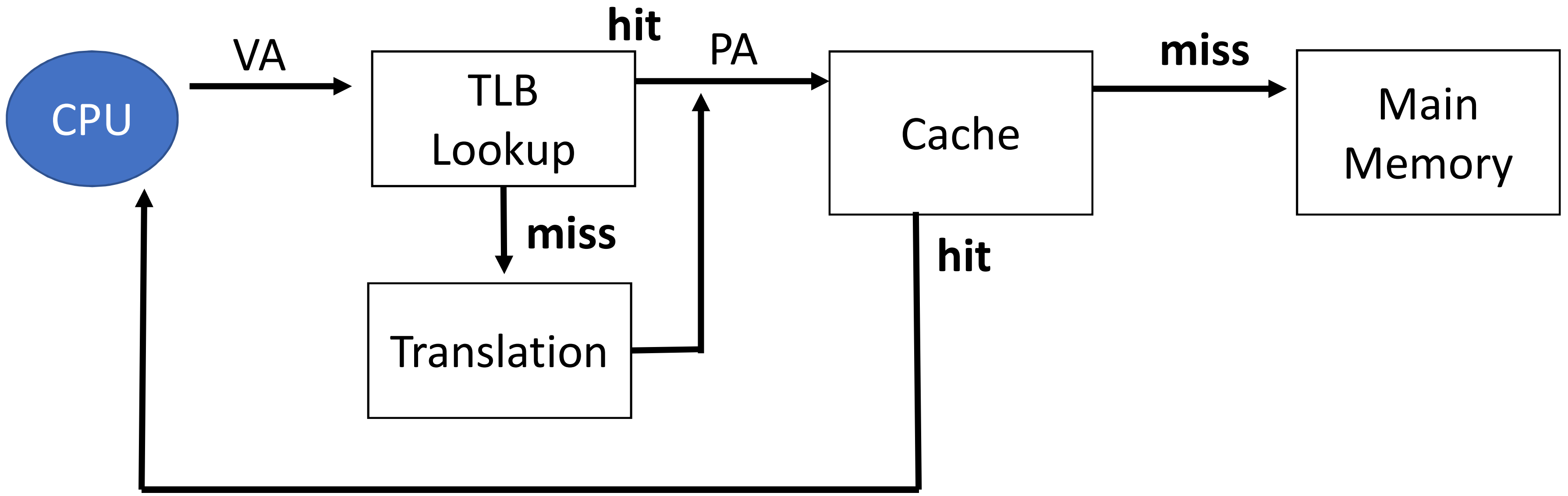
TLB control flow algorithm

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True)    // TLB Hit
    if (CanAccess(TlbEntry.ProtectBits) == True)
        Offset   = VirtualAddress & OFFSET_MASK
        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
        Register = AccessMemory(PhysAddr)
    else
        RaiseException(PROTECTION_FAULT)
else                    // TLB Miss
    PTEAddr = PTBR + (VPN * sizeof(PTE))
    PTE = AccessMemory(PTEAddr)
    if (PTE.Valid == False)
        RaiseException(SEGMENTATION_FAULT)
    else if (CanAccess(PTE.ProtectBits) == False)
        RaiseException(PROTECTION_FAULT)
    else
        TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
        RetryInstruction()
```

PTBR (Page Table Base Register)

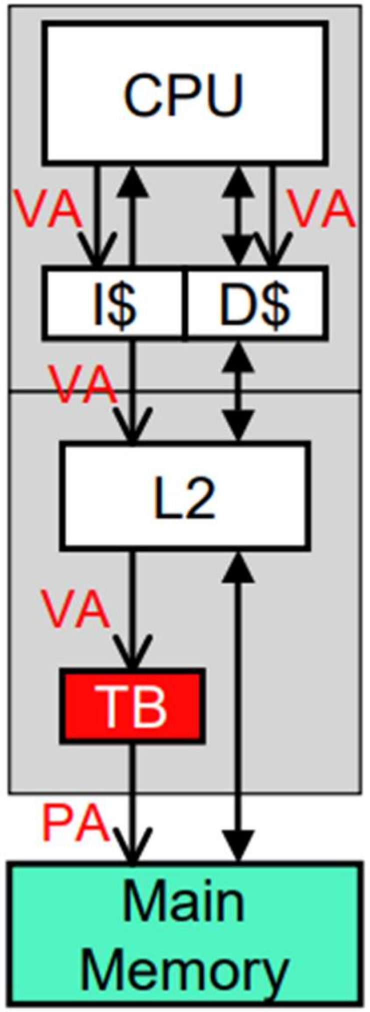
Translation with a TLB

- 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



Virtual caches

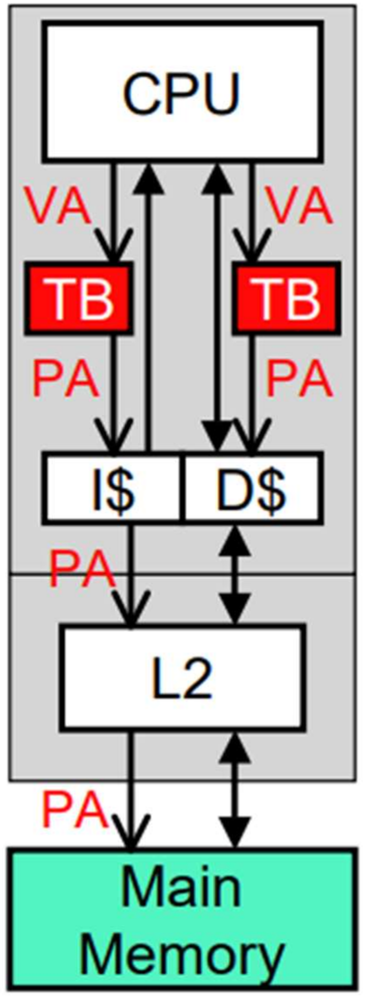
- **Virtual cache**
  - Tags in cache are virtual addresses
  - Translation only happens on cache misses
- **What to do on process switches ?**
  - Flush caches ? Slow
  - Add process IDs to cache tags
- **Does inter-process communication work ?**
  - Aliasing: multiple VAs map to same PA
  - How are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

Physical caches

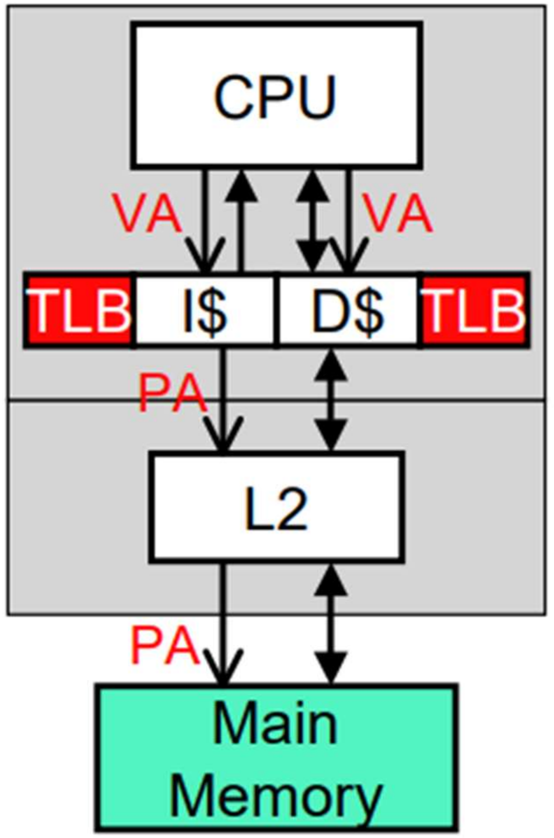
- **Physical caches**
  - Indexed and tagged by PAs
  - Translate to PA at the outset
  - No need to flush caches on process switches
    - Processes do not share PAs
  - Cached inter-process communication works
    - Single copy indexed by PA



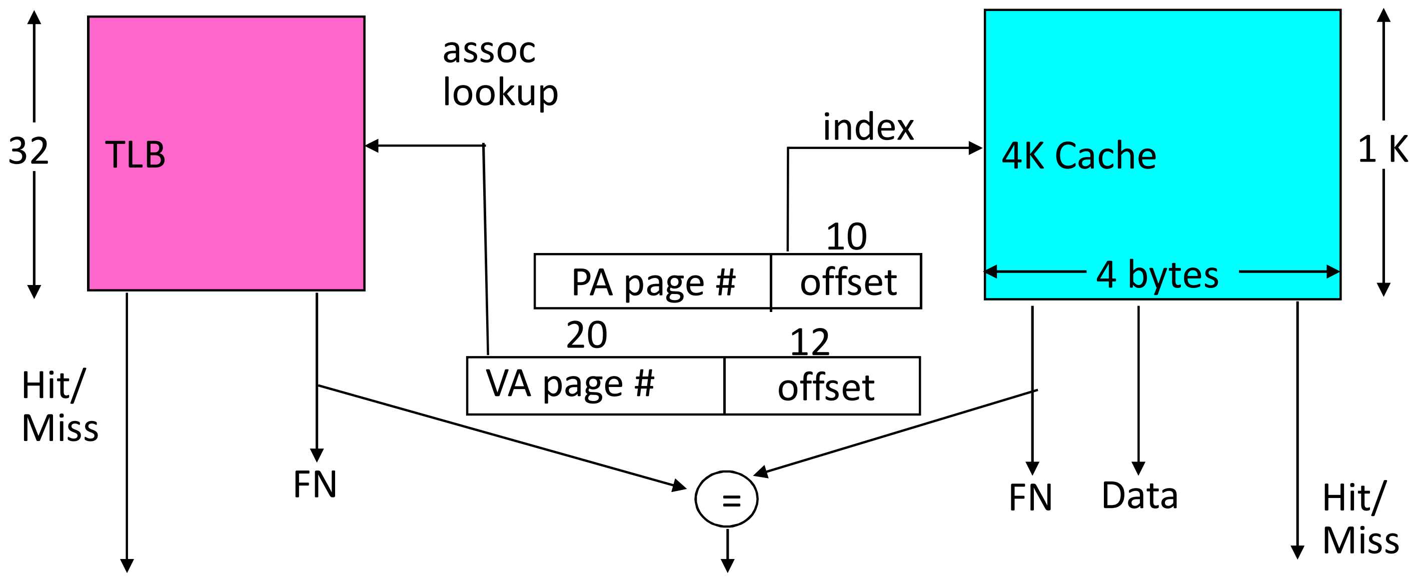
http://people.ee.duke.edu/~sorin/prior-courses/ece152-spring2008/lectures/6.9-memory.pdf

Virtual physical caches

- Virtual-physical caches
  - Indexed by VAs
  - Tagged by PAs
  - Cache access and address translation in parallel
  - No context-switching/aliasing problems
  - A TB that acts in parallel with a cache is TLB



Overlapped cache & TLB access



What if the cache size is increased to 8 KB ?

IF TLB hit and cache hit **and (cache tag = PA)** then deliver data to CPU
ELSE IF TLB hit and cache miss or cache tag != PA **THEN**
        access memory with the PA from the TLB
ELSE do standard VA translation

Some solutions to the synonym problem

- Limit cache size to (page size x associativity)
  - Get index from page offset
- 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

Handling TLB misses

- The TLB is small; it cannot hold all PTEs
  - Some translations will inevitably miss in the TLB
  - Must access memory to find the appropriate PTE
    - Called **walking** the page directory/table
    - Large performance penalty
- Who handles TLB misses ? Hardware or software ?

Handling TLB misses (cont.)

- **Hardware-managed** (e.g. x86)
  - The hardware does the page walk
  - The hardware fetches the PTE and inserts it into the TLB
    - If the TLB is full, the entry **replaces** another entry
  - Transparently to system software
- **Software-managed** (e.g. MIPS)
  - The hardware raises an exception
  - The operating system does the page walk
  - The operating system fetches the PTE
  - The operating system inserts/evicts entries in the TLB

Handling TLB misses (cont.)

- **Hardware-managed TLB**
  - Pro: No exception on TLB miss. Instruction just stalls
  - Pro: Independent instructions may continue
  - Pro: No extra instructions/data brought into caches
  - Con: page directory/table organization is etched into the system, OS has **little flexibility in deciding these**
- **Software-managed TLB**
  - Pro: The OS can define page table organization
  - Pro: More sophisticated TLB replacement policies are possible
  - Con: Need to generate an exception -> performance overhead due to pipeline flush, exception handler execution, extra instruction brought to caches

Summary

- Translation lookaside buffer (TLB)
  - Reduce the overhead of paging
  - Must be fast