

Virtual Memory



Today

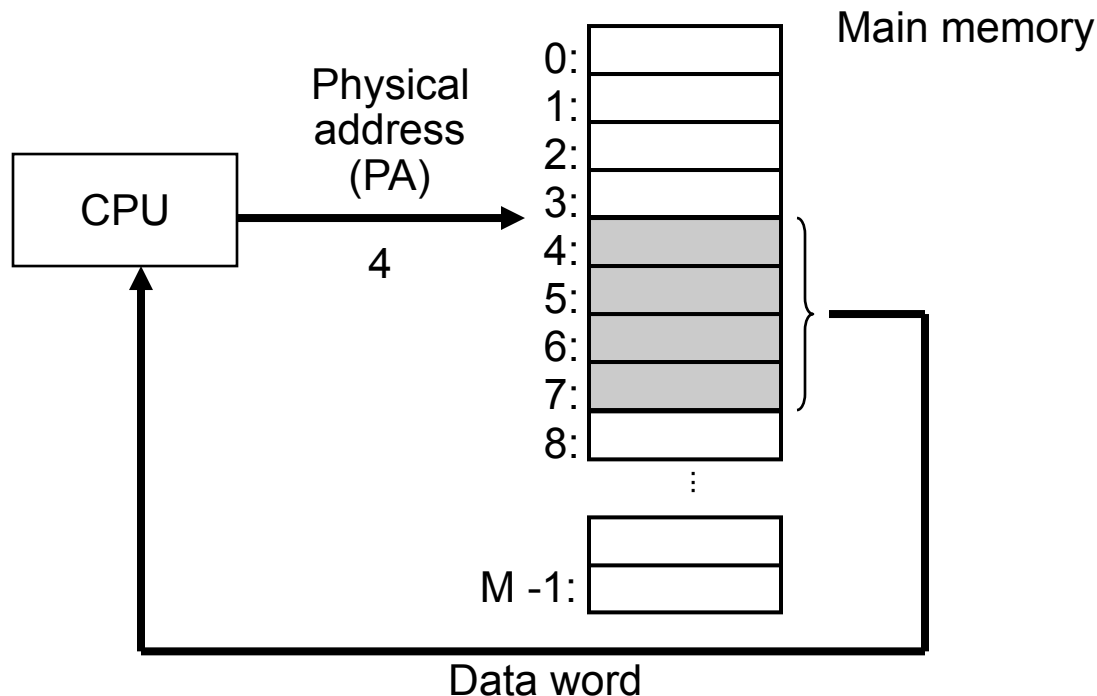
- Motivations for VM
- Address translation
- Accelerating translation with TLBs

Next time

- Linking

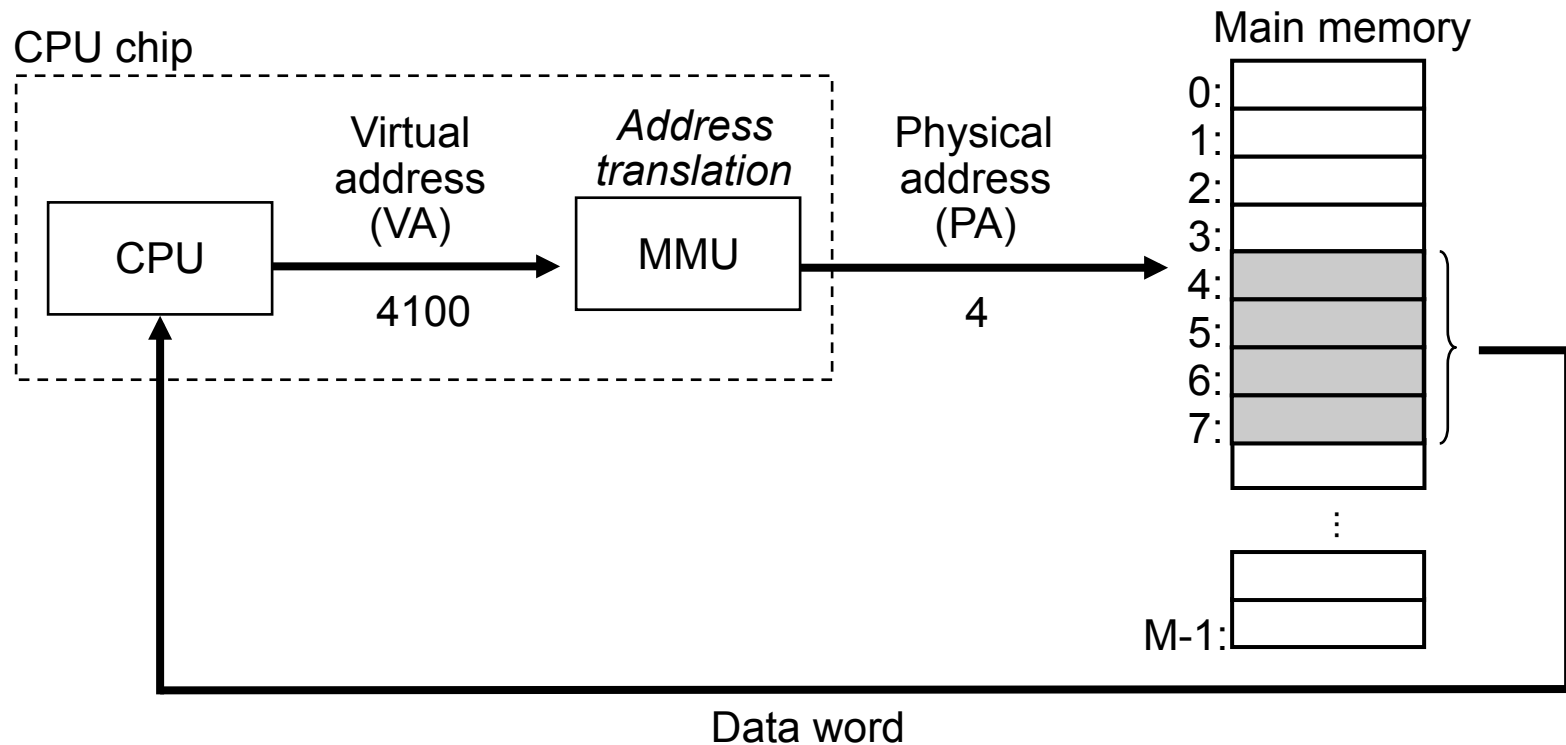
A system with physical addressing

- Main memory - An array of M contiguous byte-sized cells, each with a unique physical address
- Physical addressing
 - Most natural way to access it – Addresses generated by the CPU correspond to bytes in it
 - Used in simple systems like early PCs and embedded microcontrollers (e.g. cars and elevators)



A system with virtual addressing

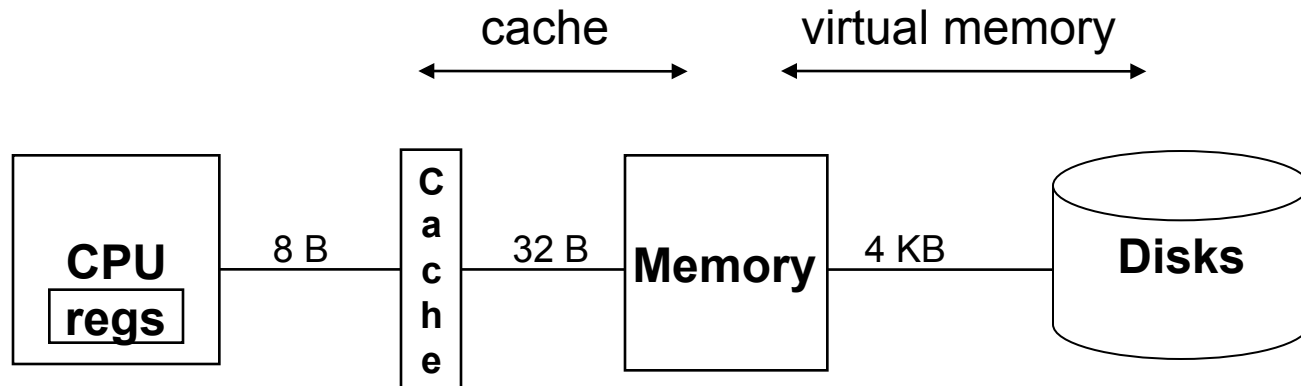
- Modern processors use virtual addresses
- CPU generates virtual address and address translation is done by dedicated hardware (*memory management unit*) via OS-managed lookup table



Motivations for virtual memory

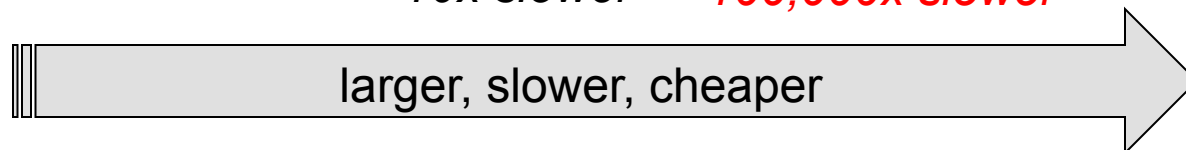
- Use physical DRAM as a cache for the disk
 - Address space of a process can exceed physical memory
 - Sum of address spaces of multiple processes can exceed physical memory
- Simplify memory management
 - Multiple processes resident in main memory
 - Each process with its own address space
 - Only “active” code and data is actually in memory
 - Allocate more memory to process as needed
- Provide protection
 - One process can't interfere with another
 - Because they operate in different address spaces
 - User process cannot access privileged information
 - Different sections of address spaces have different permissions.

Levels in memory hierarchy



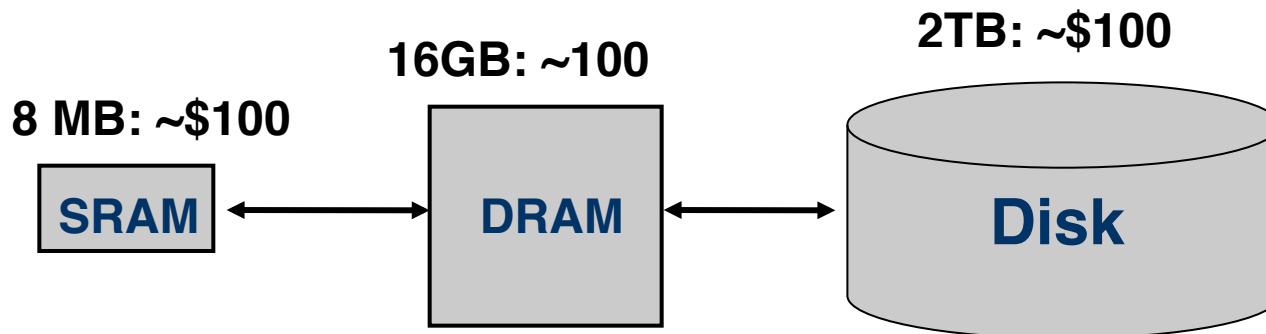
	Register	Cache	Memory	Disk Memory
size:	32 B	32KB-4MB	1024 MB	100 GB
speed:	1 ns	2 ns	30 ns	8 ms
\$/Mbyte:		\$125/MB	\$0.20/MB	\$0.001/MB
line size:	8 B	32 B	4 KB	

10x slower *100,000x slower*



Motivation: DRAM a “cache” for disk

- Full address space is quite large:
 - 32-bit addresses: ~4 Gigabytes (4 billion) bytes
 - 64-bit addresses: ~16 Exabytes (16 quintillion) bytes
- Disk storage is ~100X cheaper than DRAM storage
 - 2TB of DRAM: ~ \$10,000
 - 2TB GB of disk: ~\$100
- To access large amounts of data in a cost-effective manner, the bulk of the data must be stored on disk

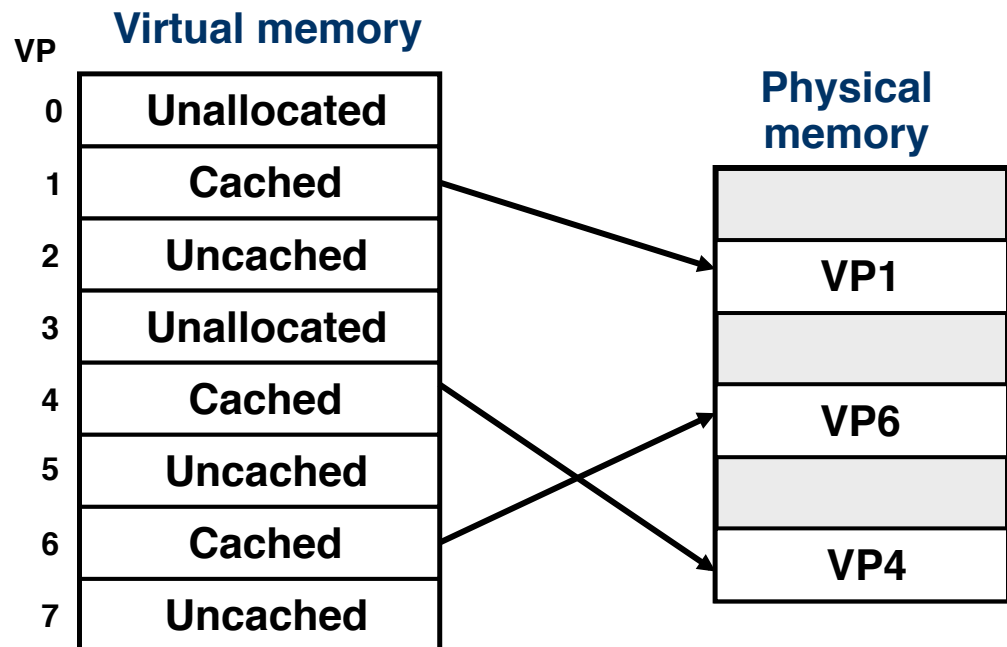


DRAM vs. SRAM as a “cache”

- DRAM vs. disk is more extreme than SRAM vs. DRAM
 - Access latencies:
 - DRAM ~10X slower than SRAM
 - Disk ~**100,000X** slower than DRAM
 - Importance of exploiting spatial locality:
 - First byte is ~**100,000X** slower than successive bytes on disk
 - vs. ~4X improvement for page-mode vs. regular accesses to DRAM
 - Bottom line:
 - Design decisions made for DRAM caches driven by enormous cost of misses

Virtual and physical pages

- Data on disk is partitioned into fixed sized blocks – virtual pages
- Physical memory into blocks of equal size – physical pages or page frames
- At any point, VM pages are partitioned into three sets
 - Unallocated
 - Allocated
 - Cached

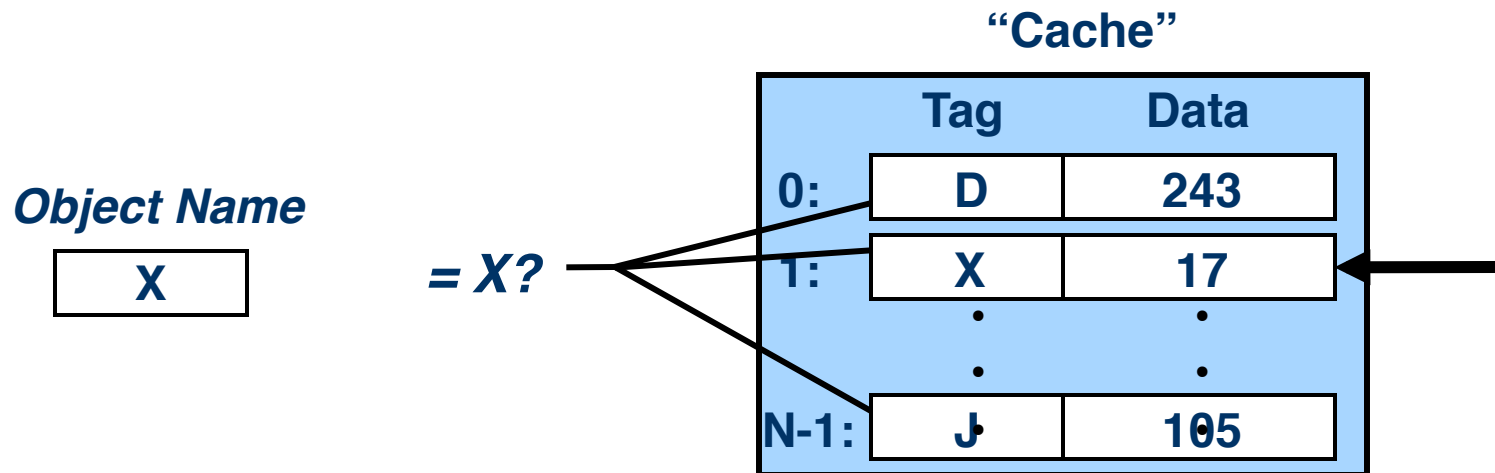


Impact of properties on design

- If DRAM were to be organized similar to an SRAM cache, how would we set the following design parameters?
 - Line size? Large, since disks are better at xferring large blocks
 - Associativity? High, to minimize miss rate
 - Write through/write back? Write back, can't afford to perform small writes to disk
- What would the impact of these choices be on:
 - Miss rate: Extremely low. $\ll 1\%$
 - Hit time: Must match cache/DRAM performance
 - Miss latency: Very high. $\sim 20\text{ms}$
 - Tag storage overhead: Low, relative to block size

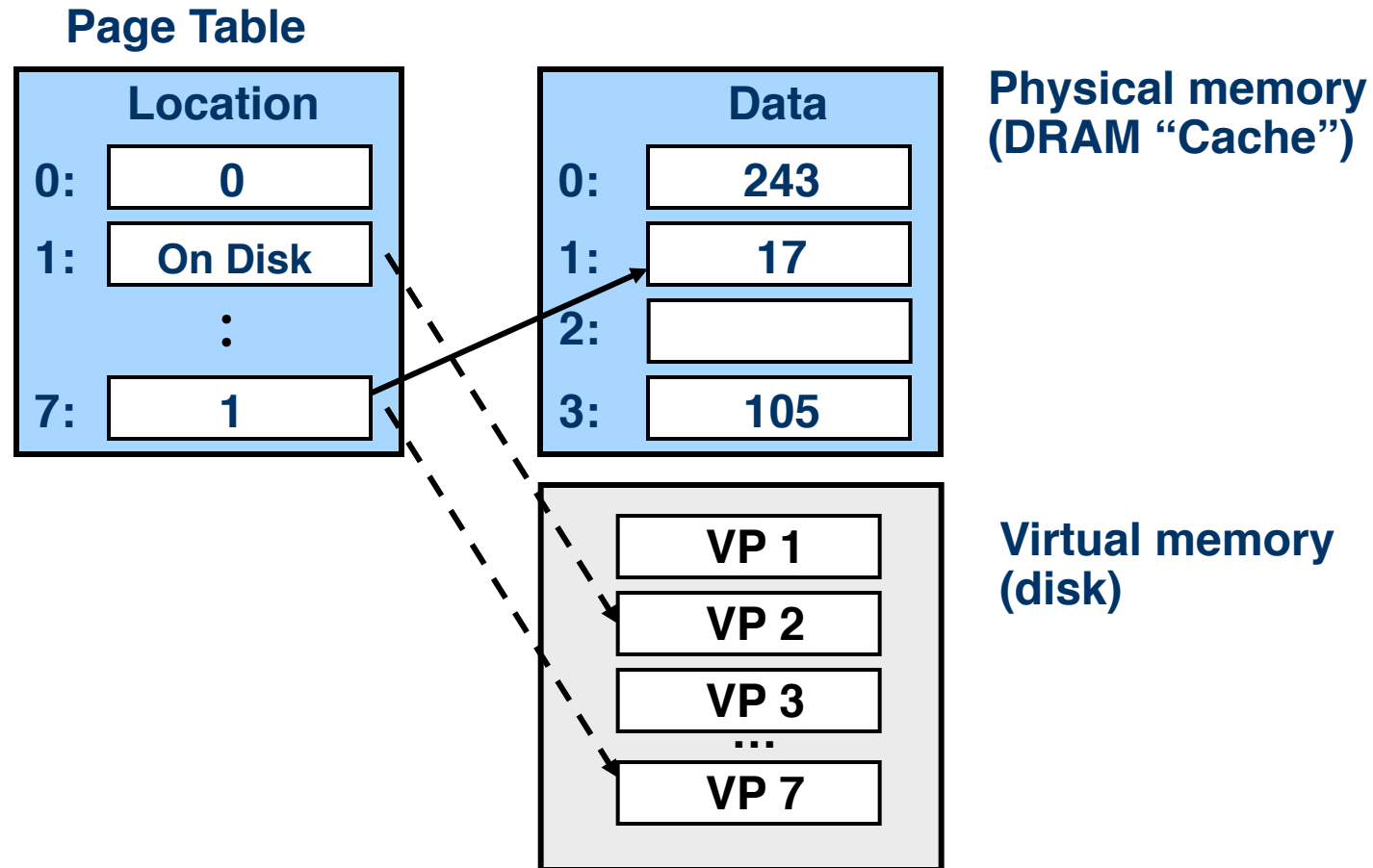
Locating an object in a "Cache"

- SRAM Cache
 - Tag stored with cache line
 - From cached to uncached form
 - Save a few bits by only storing tag
 - No tag for block not in cache
 - Hardware retrieves information
 - Can quickly match against multiple tags



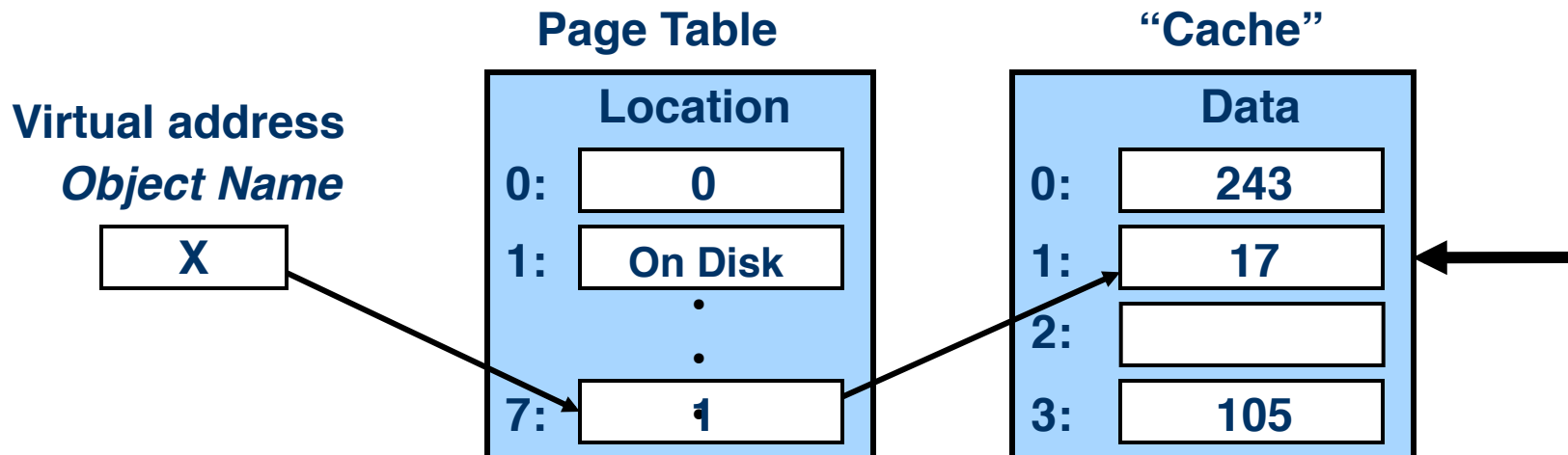
Locating an object in "Cache"

- DRAM Cache
 - Each allocated page of virtual memory has entry in *page table*
 - Mapping from virtual pages to physical pages



Locating an object in "Cache"

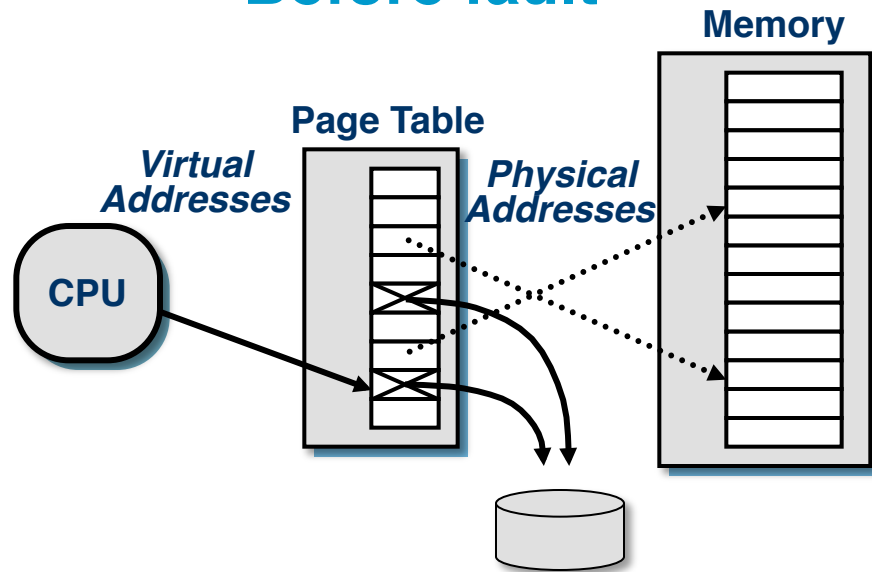
- Mapping from virtual pages to physical pages
- Page table entry even if page not in memory
 - Specifies disk address
 - Only way to indicate where to find page
 - OS retrieves information



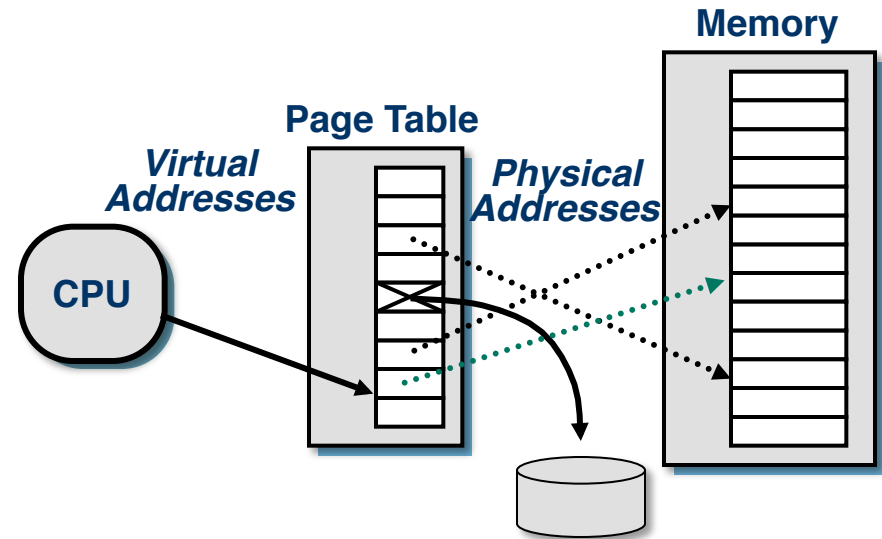
Page faults (like “cache misses”)

- What if an object is on disk rather than in memory?
 - Page table entry indicates virtual address not in memory
 - OS exception handler invoked to move data from disk into memory
 - Current process suspends, others can resume
 - OS has full control over placement, etc.

Before fault

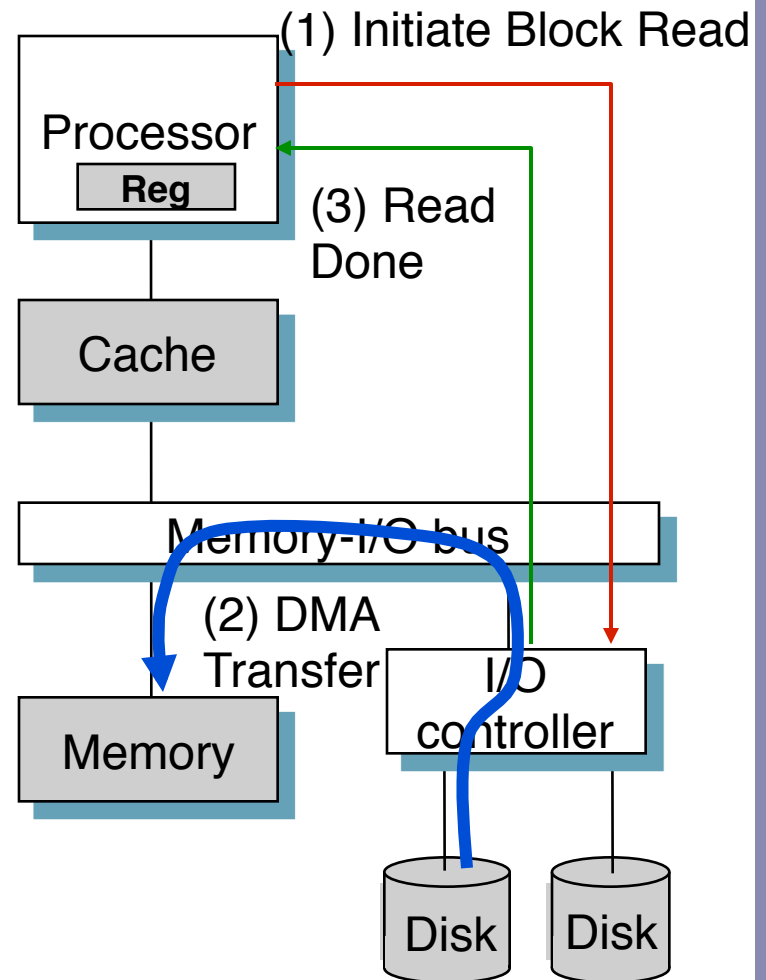


After fault



Servicing a page fault

- Processor signals controller
 - Read block of length P starting at disk address X and store starting at memory address Y
- Read occurs
 - Direct Memory Access (DMA)
 - Under control of I/O controller
- I / O controller signals completion
 - Interrupt processor
 - OS resumes suspended process



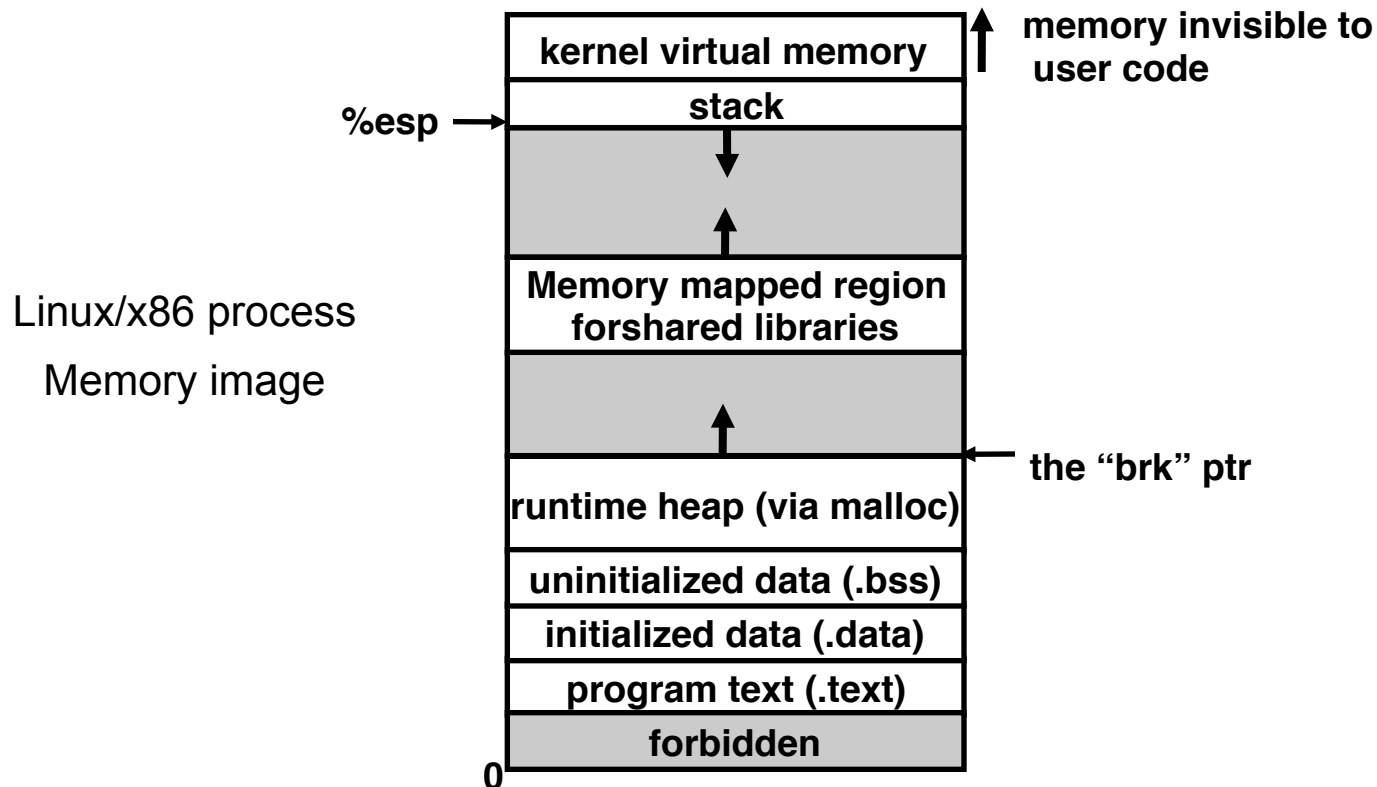
How does anything get done?! Locality to the rescue!

Motivations for virtual memory

- Use physical DRAM as a cache for the disk
 - Address space of a process can exceed physical memory size
 - Sum of address spaces of multiple processes can exceed physical memory
- **Simplify memory management**
 - **Multiple processes resident in main memory**
 - **Each process with its own address space**
 - **Only “active” code and data is actually in memory**
 - **Allocate more memory to process as needed**
- Provide protection
 - One process can't interfere with another
 - Because they operate in different address spaces
 - User process cannot access privileged information
 - Different sections of address spaces have different permissions.

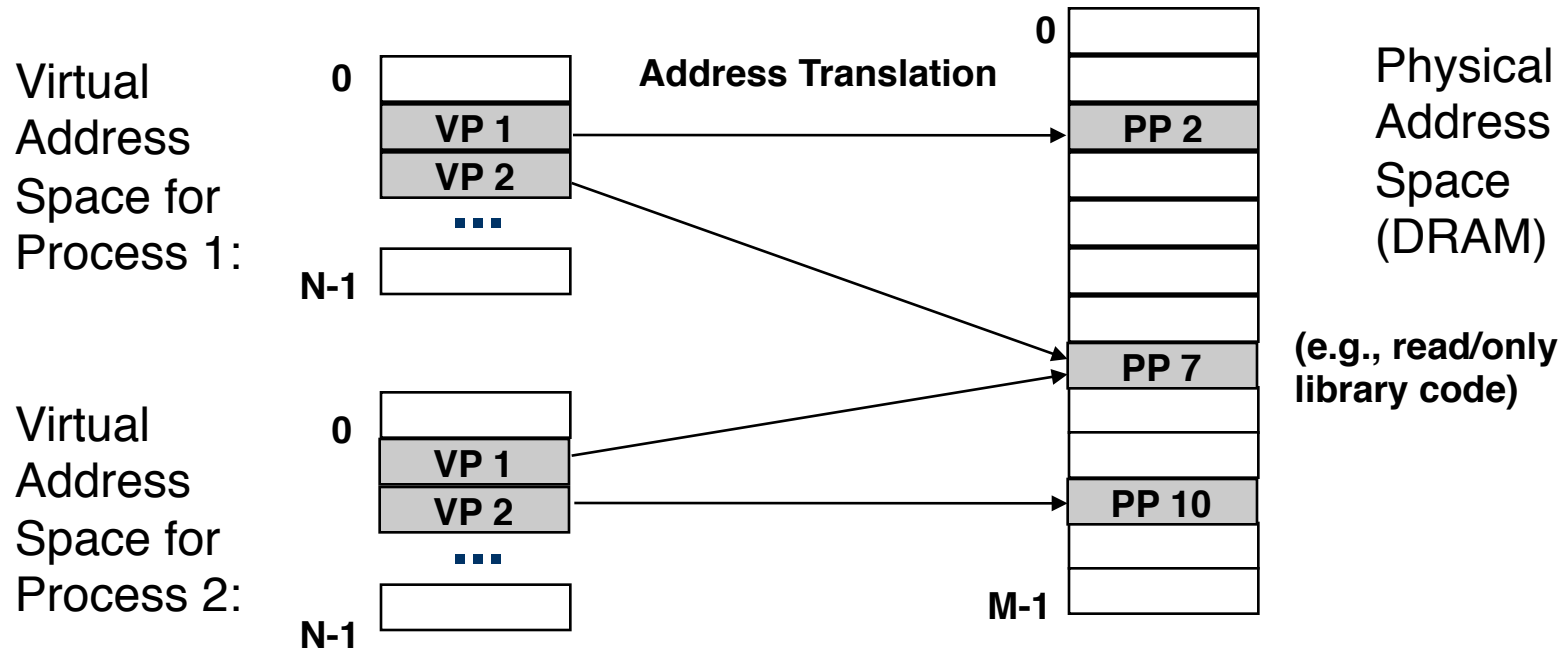
Motivation #2: Memory management

- Multiple processes can reside in physical memory.
- How do we resolve address conflicts?
 - what if two processes access something at the same address?



Solution: Separate virtual addr. spaces

- Each process has its own virtual address space
 - OS controls how virtual pages are assigned to physical mem.

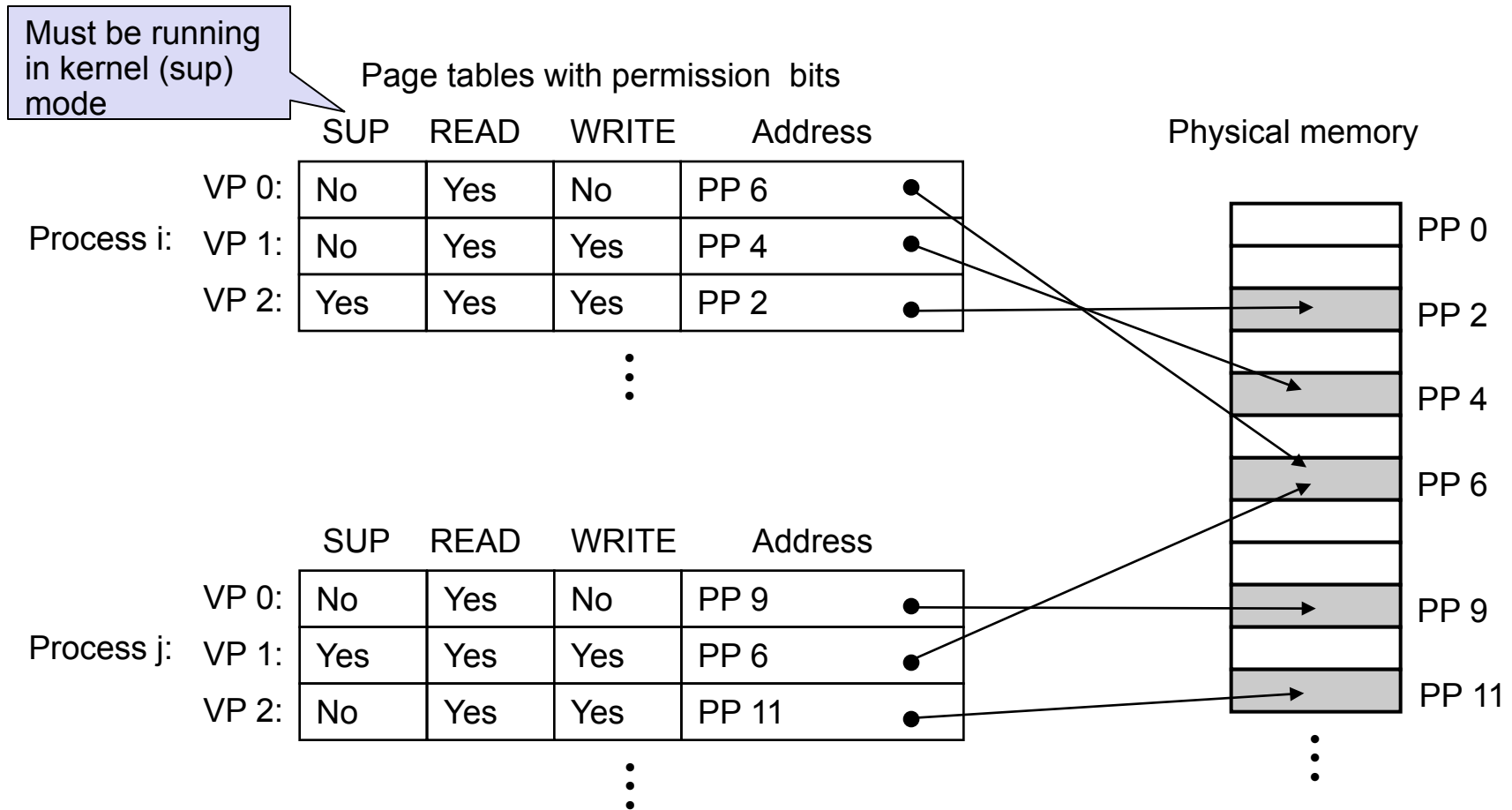


Motivations for virtual memory

- Use physical DRAM as a cache for the disk
 - Address space of a process can exceed physical memory size
 - Sum of address spaces of multiple processes can exceed physical memory
- Simplify memory management
 - Multiple processes resident in main memory
 - Each process with its own address space
 - Only “active” code and data is actually in memory
 - Allocate more memory to process as needed
- Provide protection
 - One process can't interfere with another
 - Because they operate in different address spaces
 - User process cannot access privileged information
 - Different sections of address spaces have different permissions

Motivation #3: Protection

- Page table entry contains access rights information
 - HW enforces this protection (trap into OS if violation occurs)



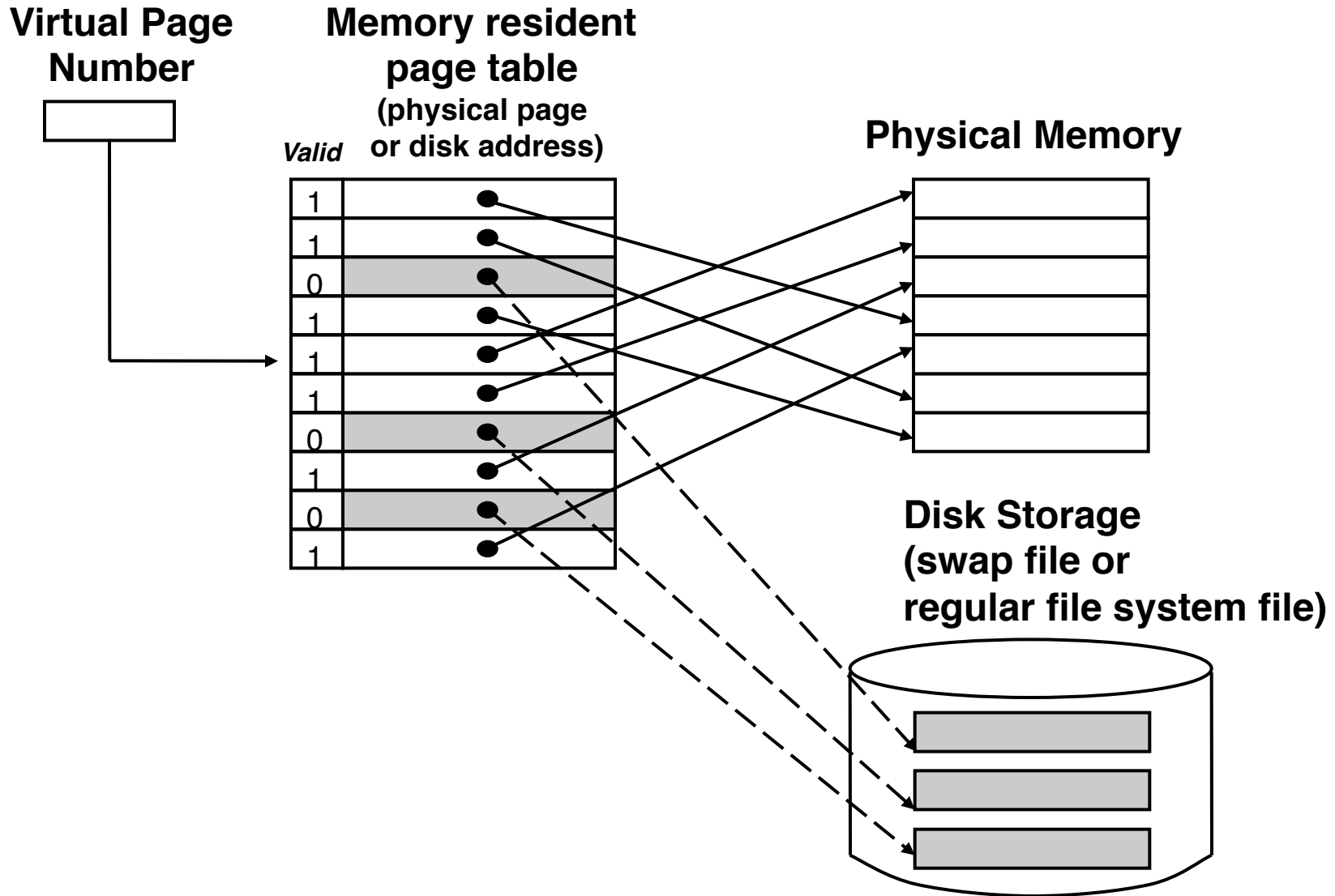
VM address translation

- First some terminology ...
- Virtual Address Space
 - $V = \{0, 1, \dots, N-1\}$ ($N = 2^n$)

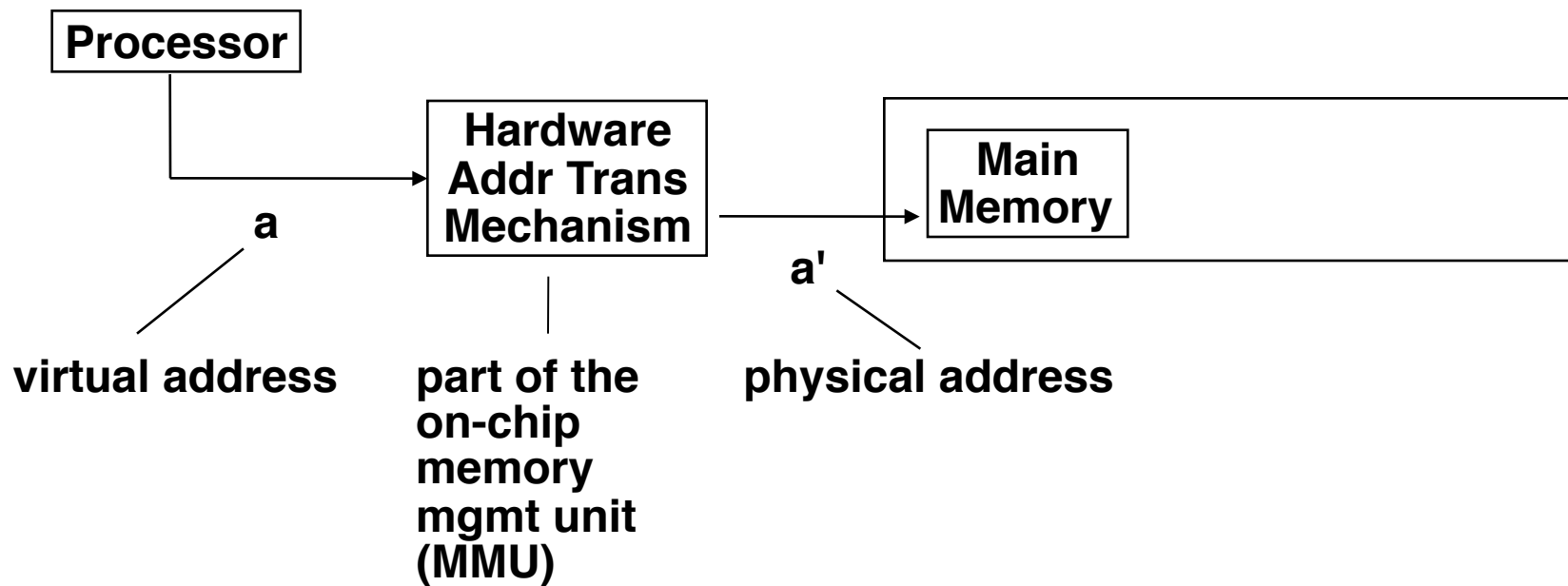
N Number of addresses in virtual address space
- Physical Address Space
 - $P = \{0, 1, \dots, M-1\}$ ($M = 2^m$)
 - $M < N$

M Number of addresses in physical address space
- Address Translation
 - MAP: $V \rightarrow P \cup \{\emptyset\}$
 - For virtual address a :
 - $\text{MAP}(a) = a'$ if data at virtual address a is at physical address a' in P
 - $\text{MAP}(a) = \emptyset$ if data at virtual address a is not in physical memory
 - Either invalid or stored on disk

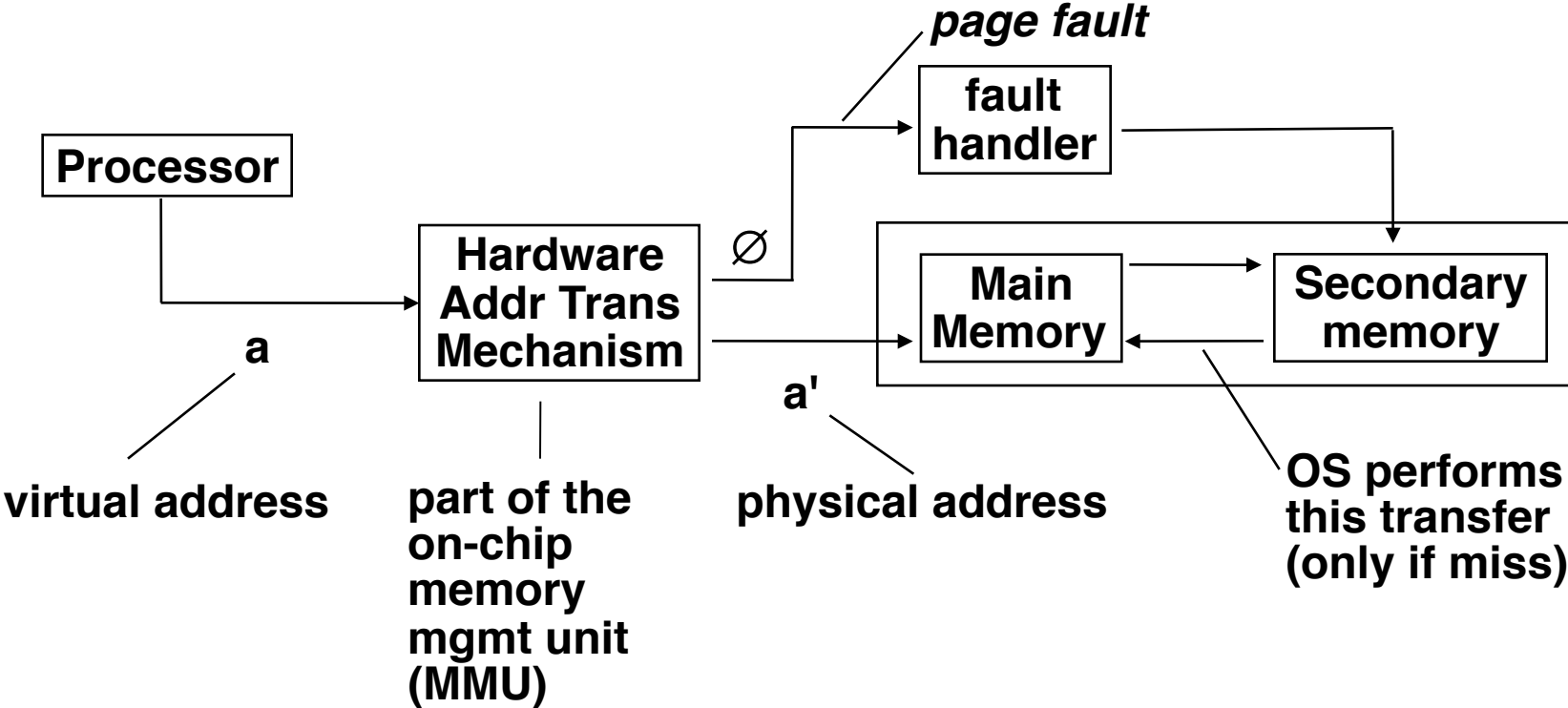
Page tables



VM address translation: Hit



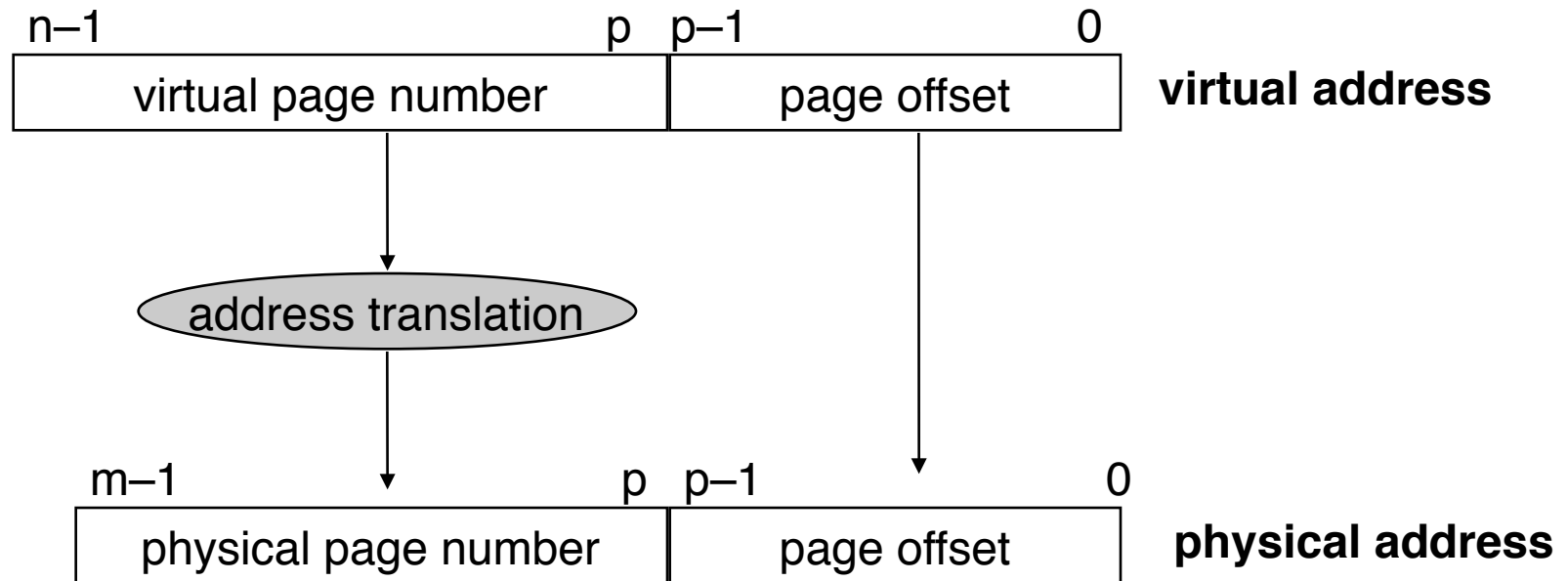
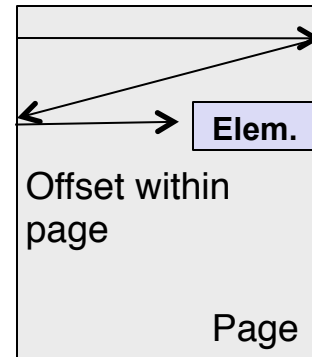
VM address translation: Miss



VM address translation

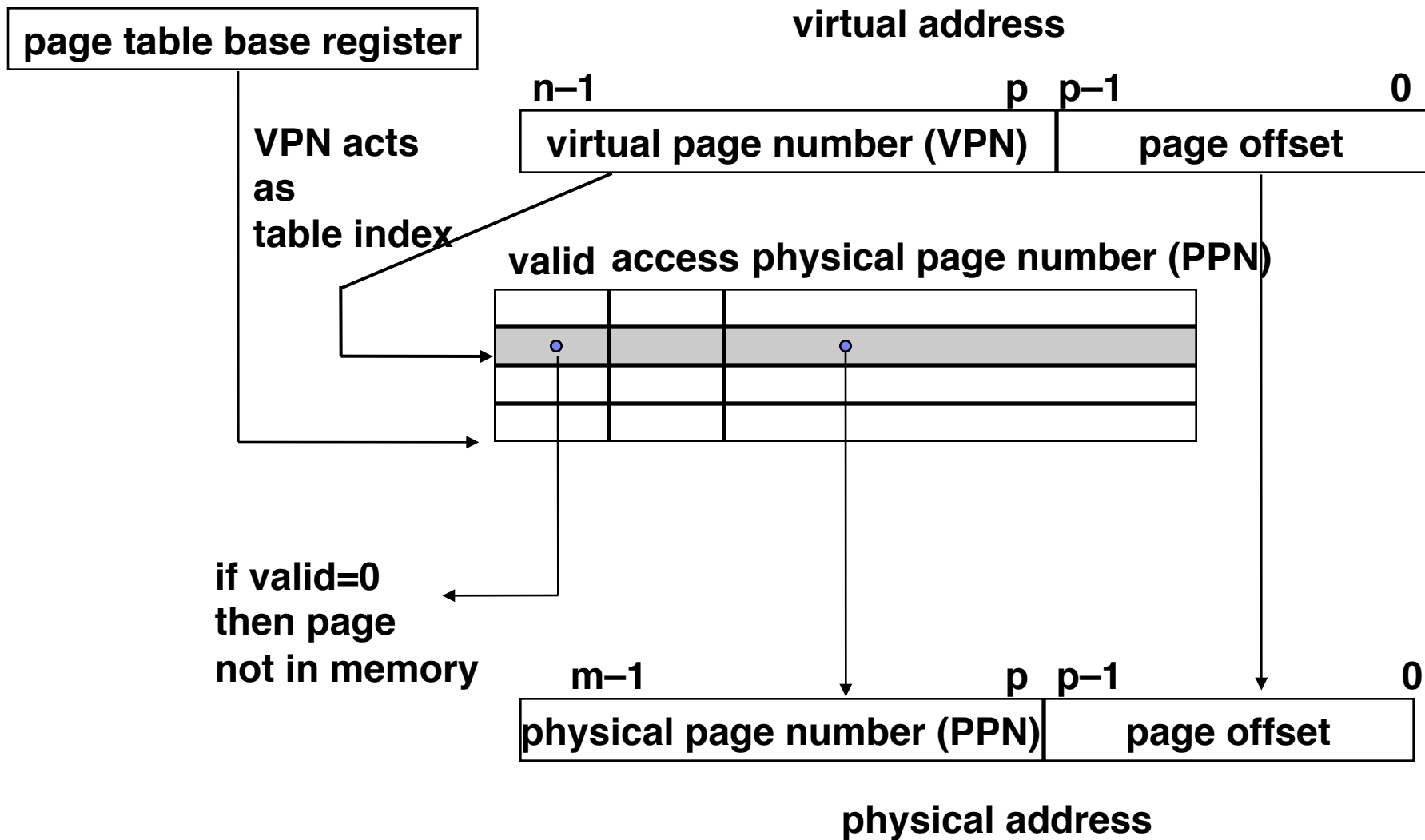
- Parameters

- $P = 2^p =$ page size (bytes).
- $N = 2^n =$ Virtual address limit
- $M = 2^m =$ Physical address limit



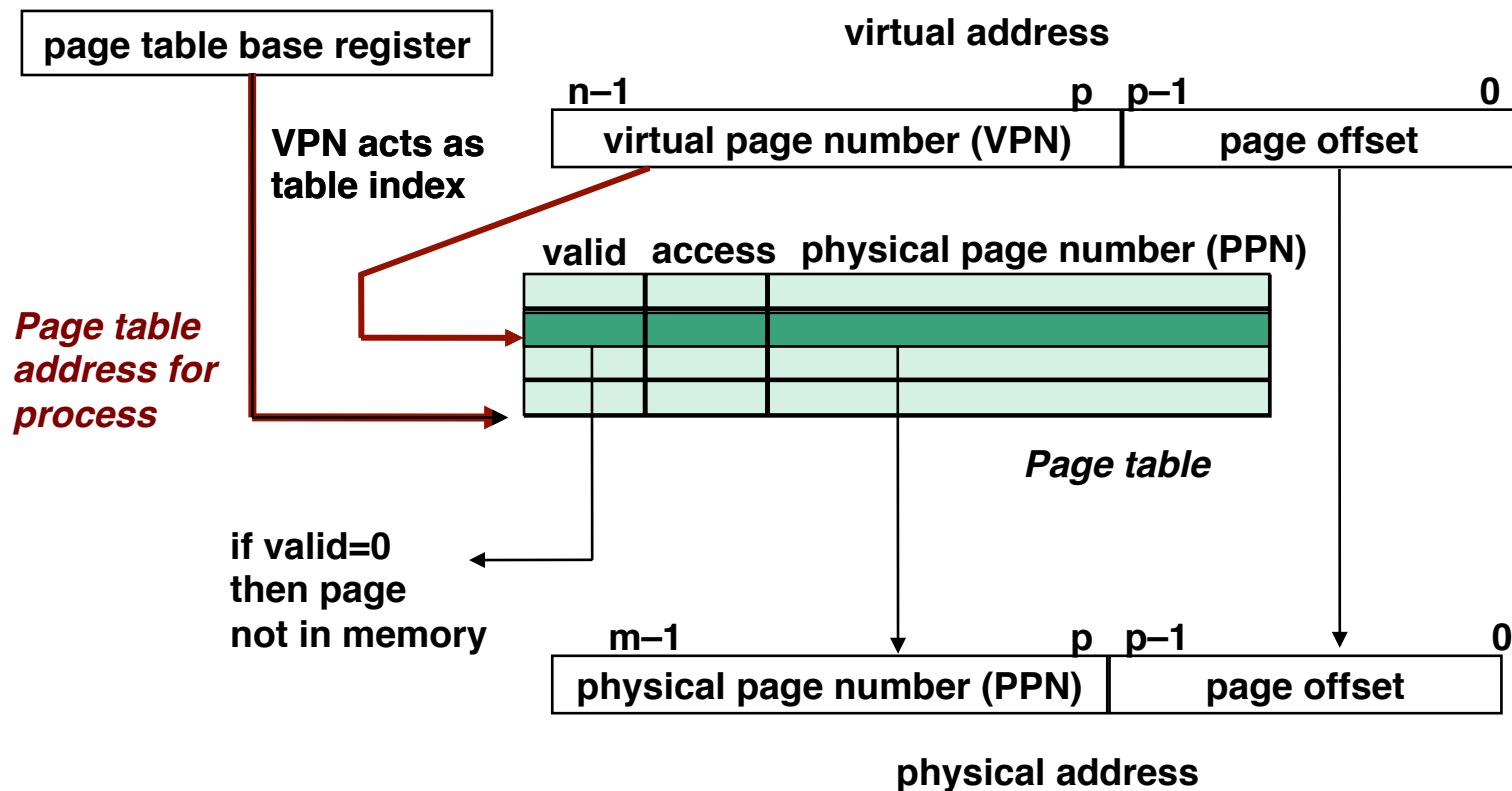
Page offset bits don't change as a result of translation

Address translation via page table



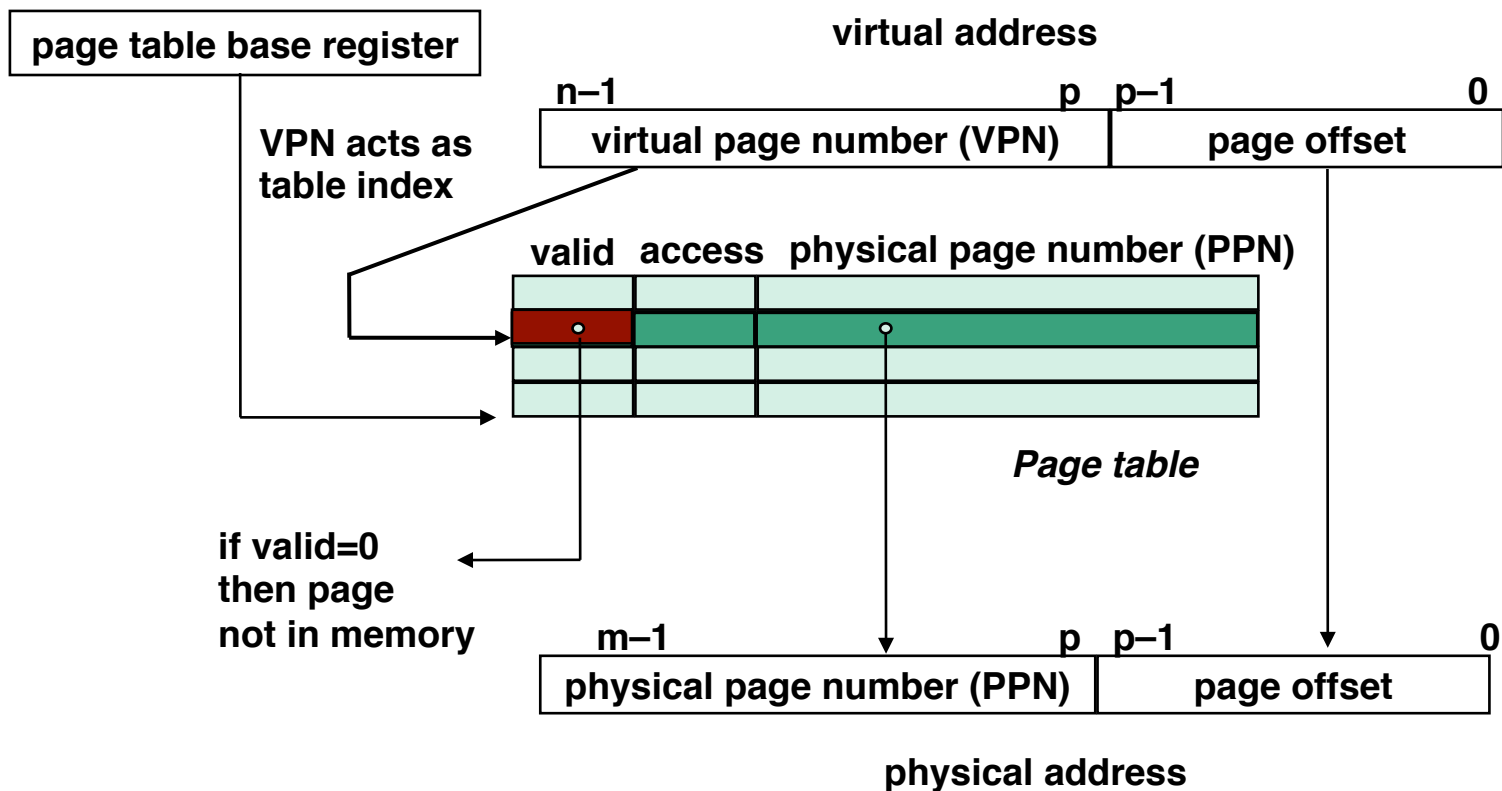
Page table operation

- Translation
 - Separate (set of) page table(s) per process
 - VPN forms index into page table (points to a page table entry)



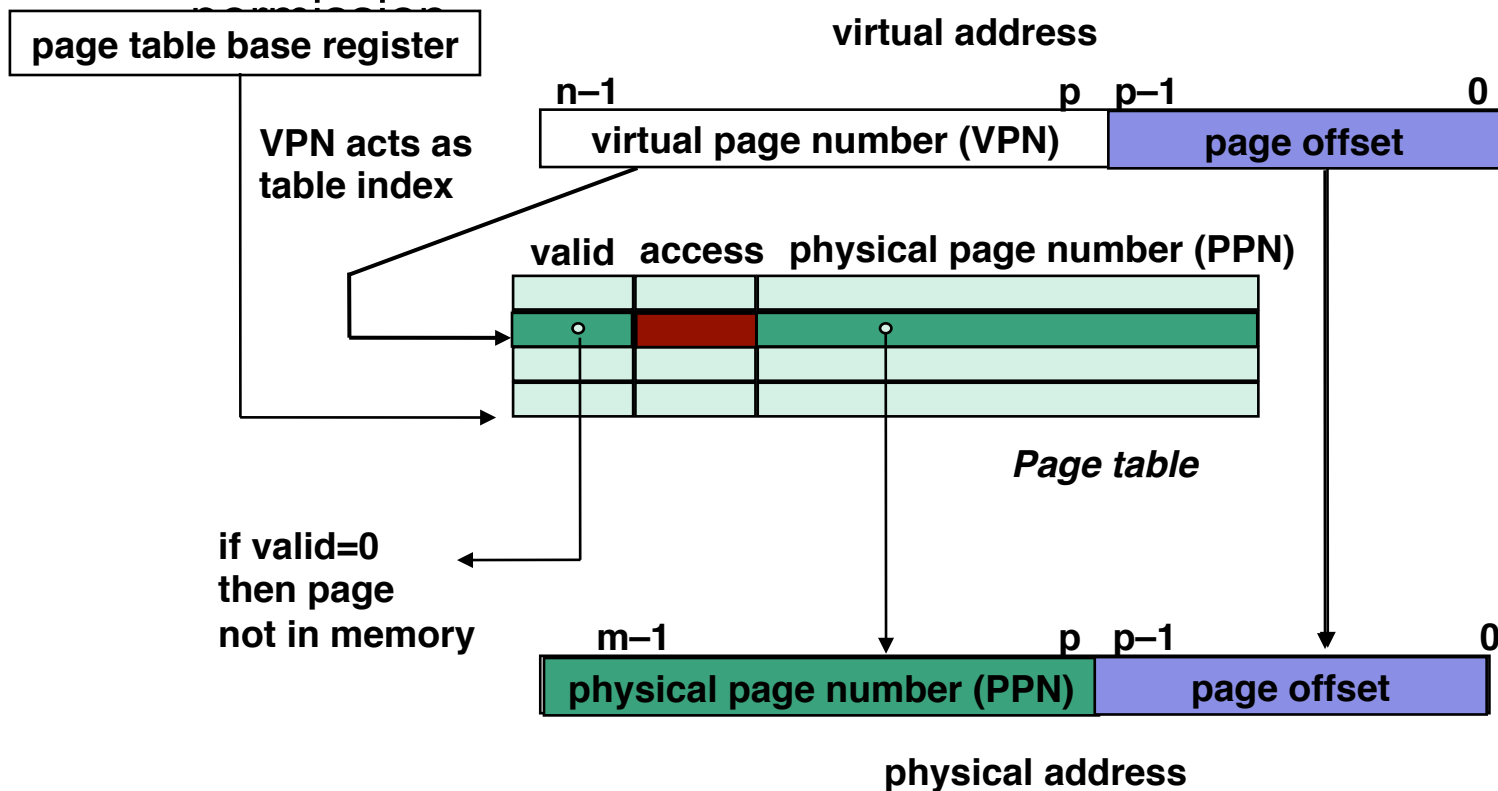
Page table operation

- Computing physical address
 - Page Table Entry (PTE) provides info about page
 - if (valid bit = 1) then the page is in memory.
 - Use physical page number (PPN) to construct address
 - if (valid bit = 0) then the page is on disk - page fault

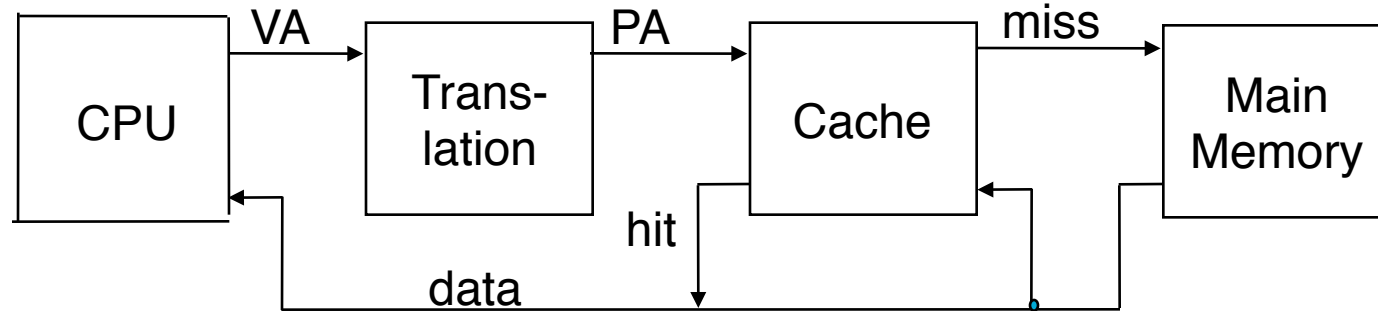


Page table operation

- Checking protection
 - Access rights field indicate allowable access
 - e.g., read-only, read-write, execute-only
 - typically support multiple protection modes
 - Protection violation fault if user doesn't have necessary permissions



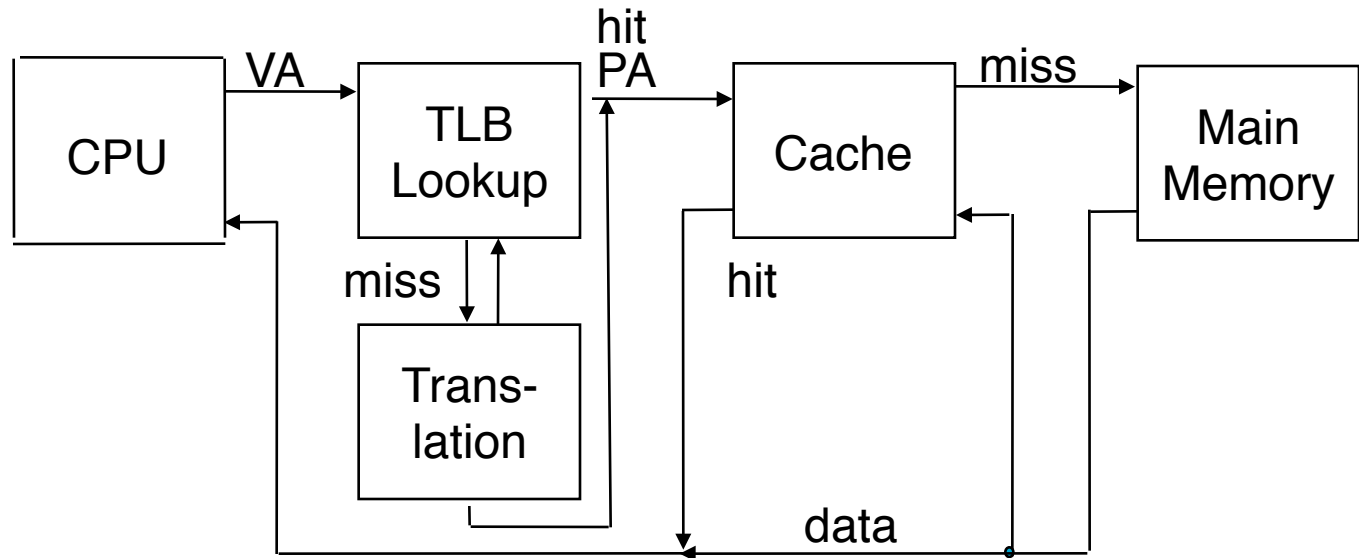
Integrating VM and cache



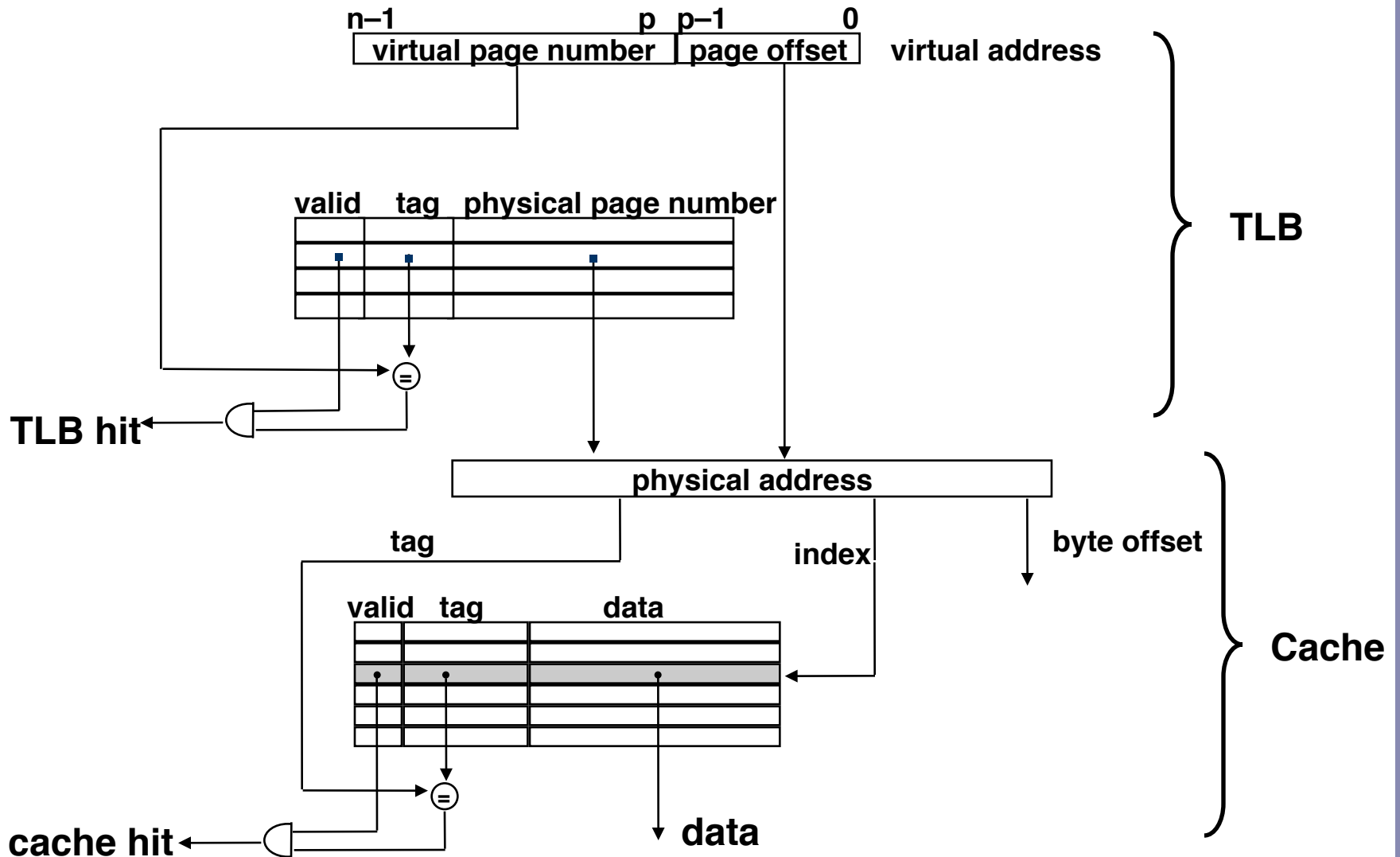
- Most caches “Physically Addressed”
 - Accessed by physical addresses
 - Allows multiple processes to have blocks in cache at a time
 - Allows multiple processes to share pages
 - Cache doesn’t need to be concerned with protection issues
 - Access rights checked as part of address translation
- Perform address translation before cache lookup
 - But this could involve a memory access itself (of the PTE)
 - Of course, page table entries can also become cached

Speeding up translation with a TLB

- Translation Lookaside Buffer (TLB)
 - Small hardware cache in MMU
 - Maps virtual page numbers to physical page numbers
 - Contains complete page table entries for small number of pages

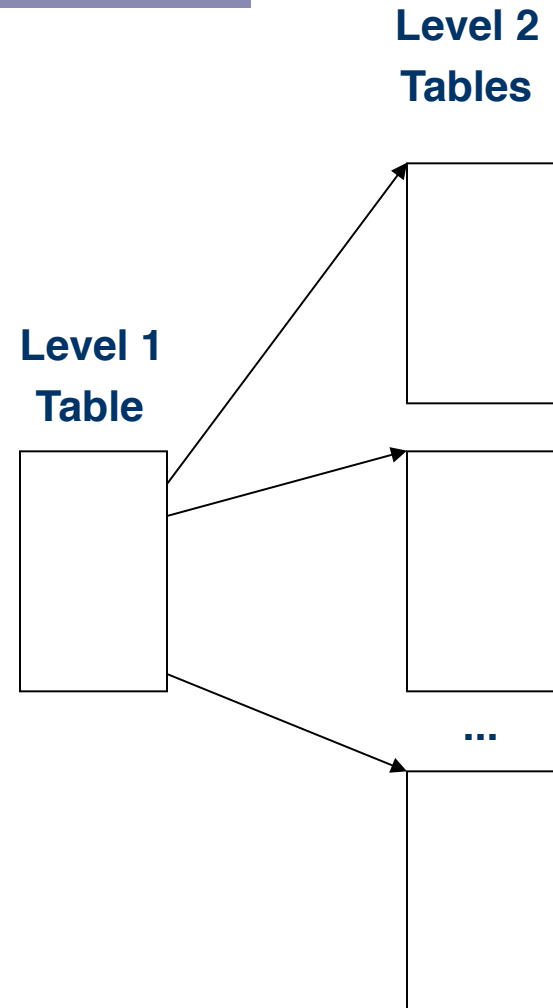


Address translation with a TLB



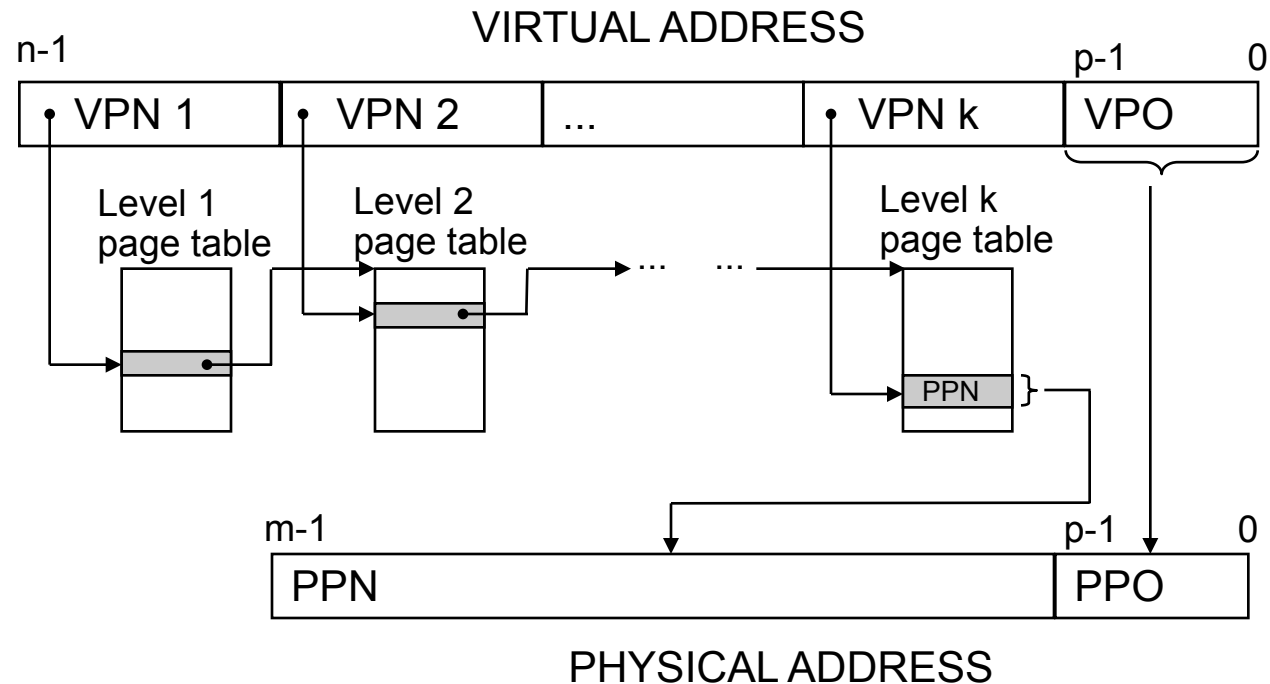
Multi-level page tables

- Given:
 - 4KB (2^{12}) page size
 - 32-bit address space
 - 4-byte PTE
- Problem:
 - Would need a 4 MB page table!
 - $2^{20} * 4$ bytes
- Common solution
 - Paged the page table – multi-level page tables
 - e.g., 2-level table (P6)
 - Level 1 table: 1024 entries, each of which points to a Level 2 page table.
 - Level 2 table: 1024 entries, each of which points to a page



Multi-level page table with k levels

- Virtual address split into k VPNs and a VPO, each VPN i is an index into a page table at level i
- Of course, now you really *really* need a TLB!

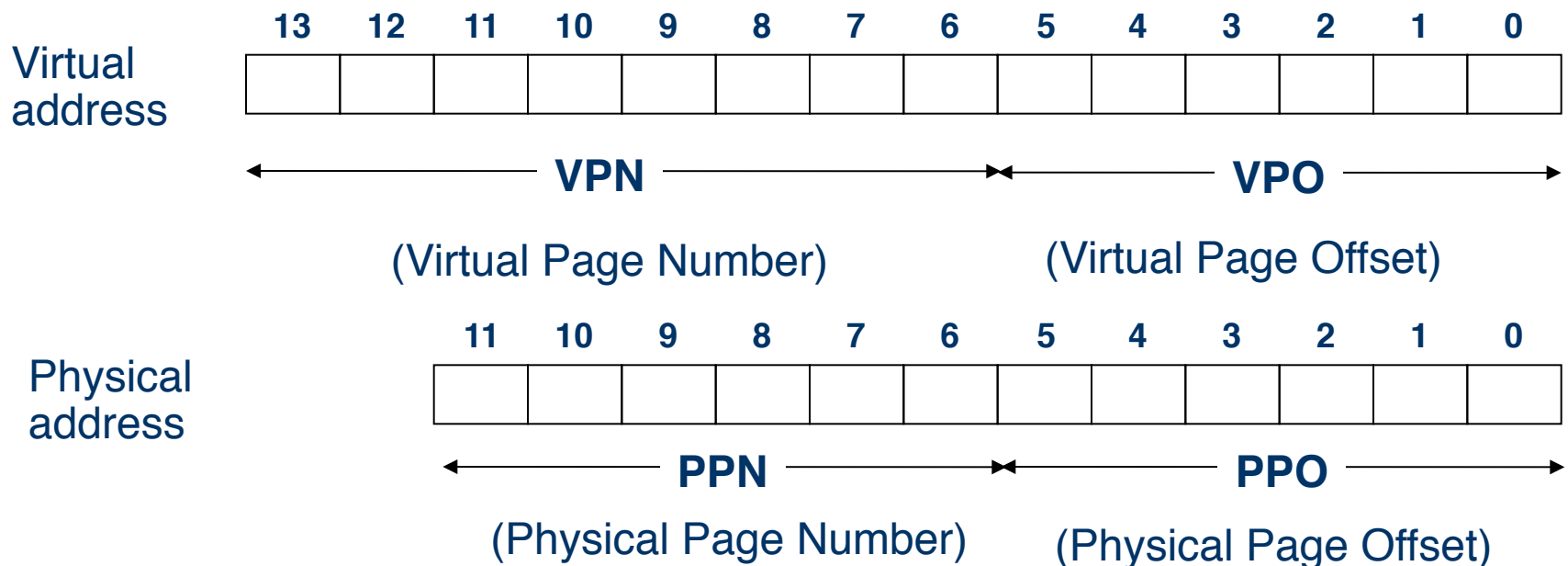


Taking stock – main themes

- Programmer's view
 - Large “flat” address space
 - Can allocate large blocks of contiguous addresses
 - Processor “owns” machine
 - Has private address space
 - Unaffected by behaviour of other processes
- System view
 - Virtual address space created by mapping to set of pages
 - Need not be contiguous
 - Allocated dynamically
 - Enforce protection during address translation
 - OS manages many processes simultaneously
 - Continually switching among processes
 - Especially when one must wait for a resource
 - E.g., disk I/O to handle page fault

Simple memory system

- Memory is byte addressable
- Access are to 1-byte words
- 14-bit virtual addresses, 12-bit physical address
- Page size = 64 bytes (2^6)



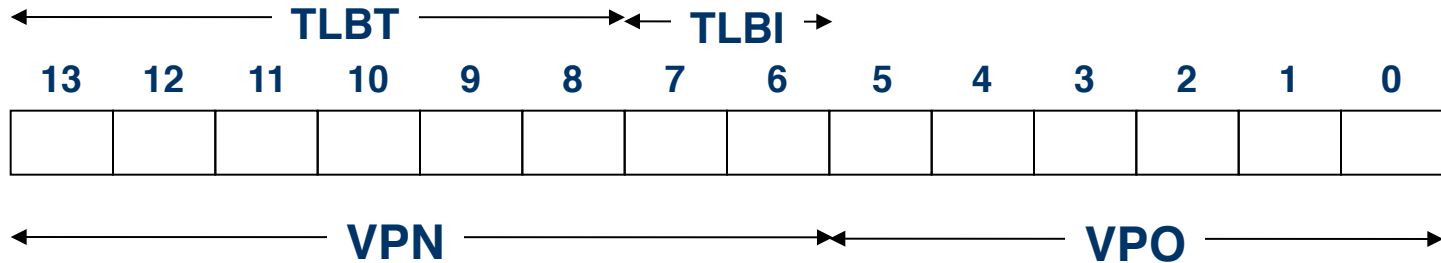
Simple memory system page table

- Only show first 16 entries

VPN	PPN	Valid	VPN	PPN	Valid
00	28	1	08	13	1
01	–	0	09	17	1
02	33	1	0A	09	1
03	02	1	0B	–	0
04	–	0	0C	–	0
05	16	1	0D	2D	1
06	–	0	0E	11	1
07	–	0	0F	0D	1

Simple memory system TLB

- TLB
 - 16 entries
 - 4-way associative

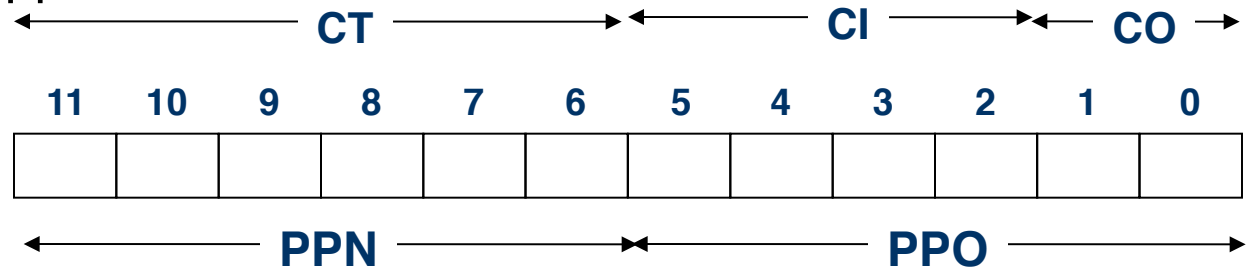


Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

Simple memory system cache

- Cache

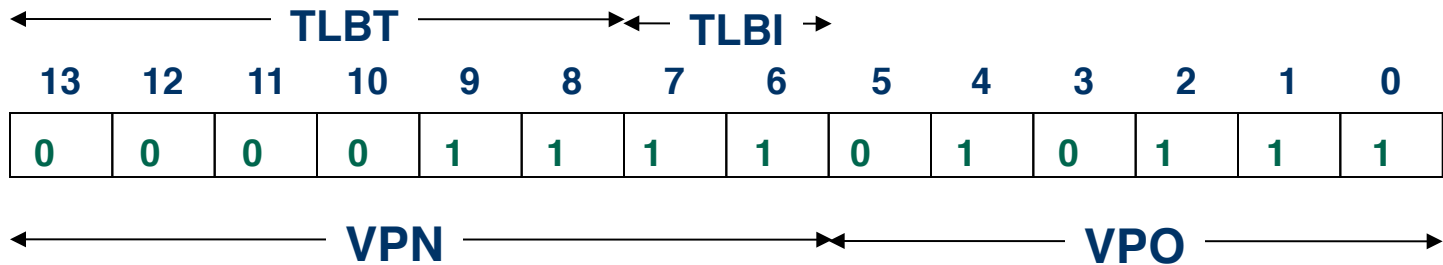
- 16 lines
- 4-byte line size
- Direct mapped



Idx	Tag	Valid	B0	B1	B2	B3	Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11	8	24	1	3A	00	51	89
1	15	0	-	-	-	-	9	2D	0	-	-	-	-
2	1B	1	00	02	04	08	A	2D	1	93	15	DA	3B
3	36	0	-	-	-	-	B	0B	0	-	-	-	-
4	32	1	43	6D	8F	09	C	12	0	-	-	-	-
5	0D	1	36	72	F0	1D	D	16	1	04	96	34	15
6	31	0	-	-	-	-	E	13	1	83	77	1B	D3
7	16	1	11	C2	DF	03	F	14	0	-	-	-	-

Address translation problem 9.4

- Virtual address 0x03d7

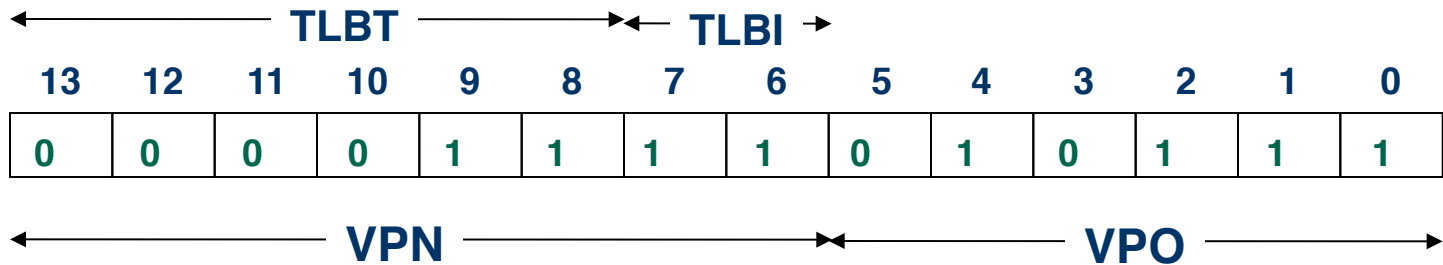


VPN 0xf TLBI 0x3 TLBTag 0x3 TLB Hit? Y Page Fault? N PPN: 0xd

Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

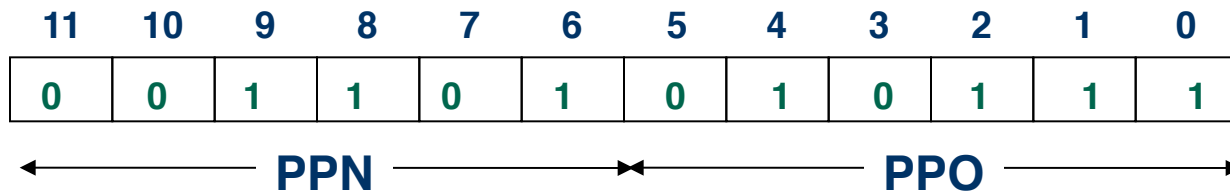
Address translation problem 9.4

- Virtual address 0x03d7



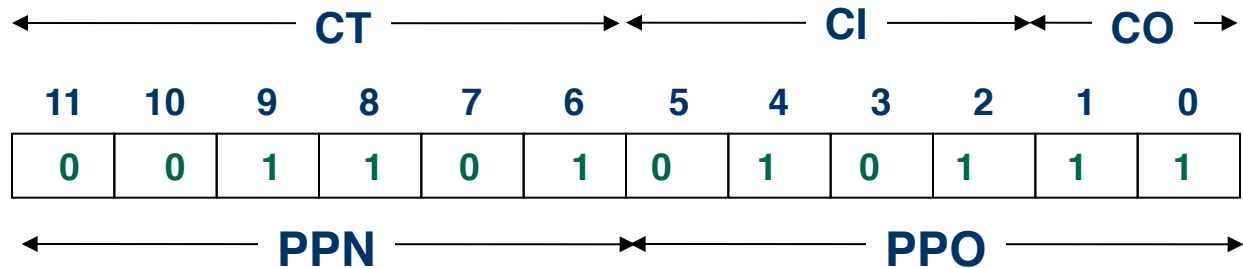
VPN 0xf TLBI 0x3 TLBTag 0x3 TLB Hit? Y Page Fault? N PPN: 0xd

- Physical address



Address translation problem 9.4

- Virtual address 0x03d7
- Physical address



Offset: 0x3 C: 0x5 CT: 0xd Hit? Y Byte returned: 0x1d

Idx	Tag	Valid	B0	B1	B2	B3	Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11	8	24	1	3A	00	51	89
1	15	0	-	-	-	-	9	2D	0	-	-	-	-
2	1B	1	00	02	04	08	A	2D	1	93	15	DA	3B
3	36	0	-	-	-	-	B	0B	0	-	-	-	-
4	32	1	43	6D	8F	09	C	12	0	-	-	-	-
5	0D	1	36	72	F0	1D	D	16	1	04	96	34	15
6	31	0	-	-	-	-	E	13	1	83	77	1B	D3
7	16	1	11	C2	DF	03	F	14	0	-	-	-	-

Summary

- Today
 - VM motivation
 - VM mechanisms
 - Optimizing VM performance
- Next time - Linking