

# Virtual Memory

## From CS:APP by Bryant and O'Hallaron

### Topics

- Benefits of Virtual Memory
- Address translation
- Accelerating translation with a TLB  
(Translation Lookaside Buffer)
- Example VM system in action

# Virtual Memory Benefits

- 1. Allows physical DRAM to act as a cache for the disk**
  - Logical address space of a process can now exceed the physical memory size
  - The sum of the logical address spaces of multiple processes can exceed the physical memory size
- 2. Simplifies memory management**
  - Multiple processes resident in main memory
    - Each process with its own separate address space
  - Only “active” code and data is actually in memory
    - Allocate more memory to a process as needed – demand paging
- 3. Provides protection**
  - One process can't interfere with the data of another
    - As they operate in different and distinct address spaces
  - User processes cannot access privileged information
    - Different sections of address spaces have different permissions

# Benefit #1: DRAM a Cache for Disk

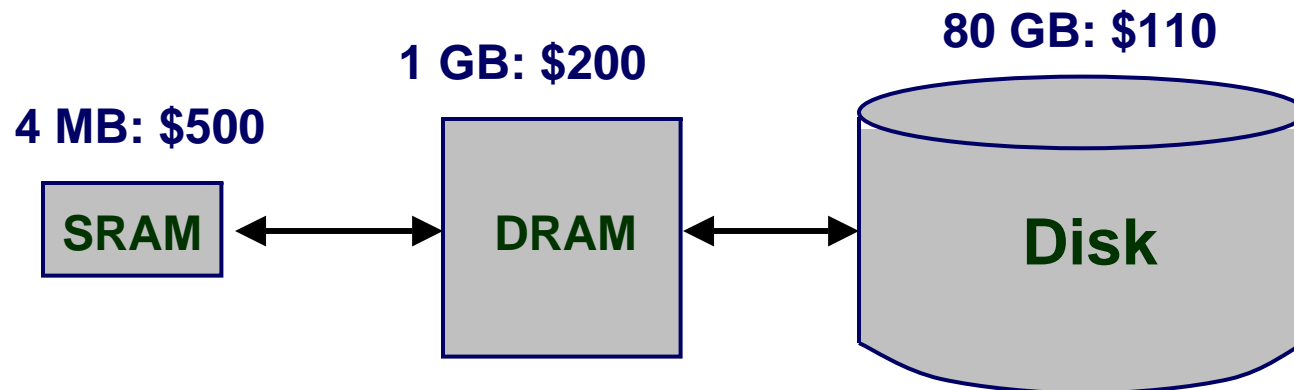
Full address space is quite large:

- 32-bit addresses:  $4 \times 10^9$  (4 billion) bytes (or 4GB)
- 64-bit addresses:  $16 \times 10^{18}$  (16 quintillion) bytes

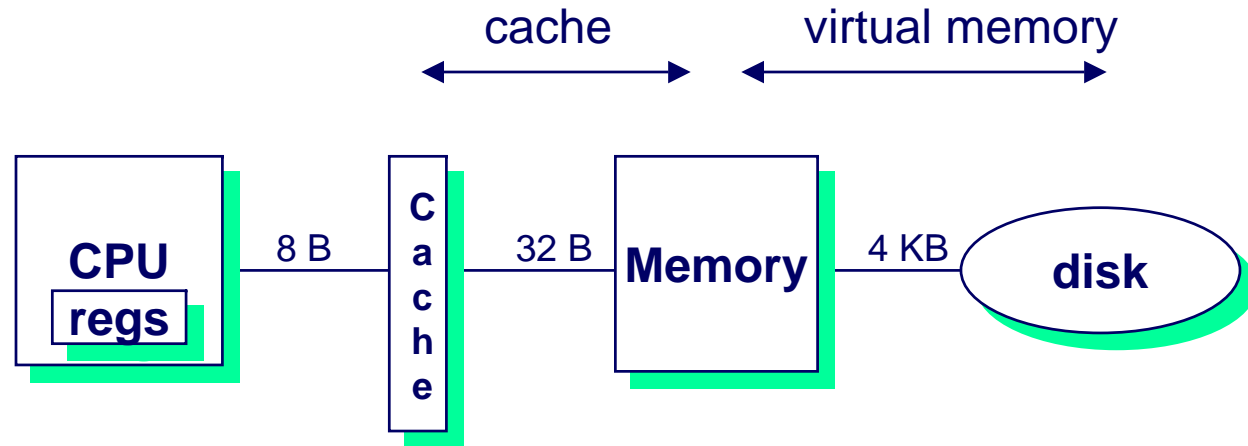
Disk storage is approximately 300X cheaper than DRAM storage (although these numbers are a little dated)

- 80 GB of DRAM: \$33,000
- 80 GB of disk: \$110

To access large amounts of data in a cost-effective manner, the bulk of the data must be stored on disk



# Levels in Memory Hierarchy



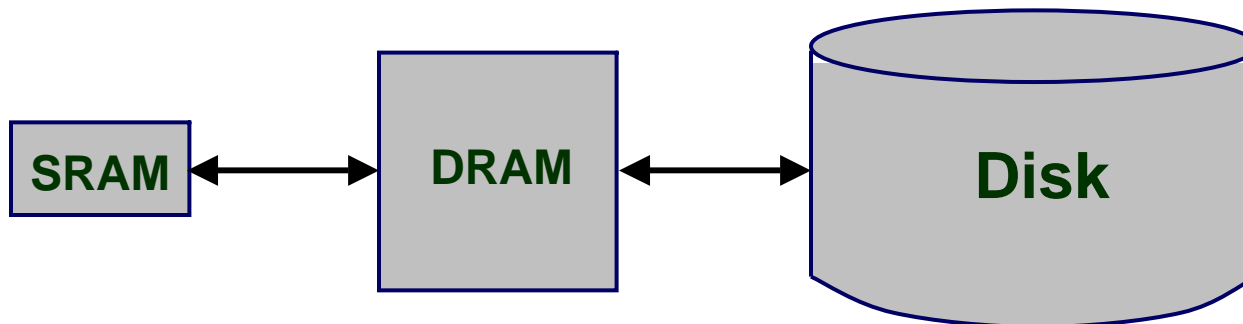
	Register	Cache	Memory	Disk Memory
size:	32B	32KB-4MB	1024MB	100GB
speed:	1ns	2ns	30ns	8ms
\$/MB:		\$125/MB	\$0.20/MB	\$0.001/MB
line size:	8B	32B	4KB	

larger, slower, cheaper 

# 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 since (since we get the successive bytes brought “for free” into DRAM)
- Bottom line:
  - Design decisions made for DRAM caches driven by enormous cost of misses



# Impact of Properties on Design

If DRAM was to be organized similar to an SRAM cache, how would we set the following design parameters?

- **Line size?**
  - Large, since disk most efficient at transferring large blocks
- **Associativity?**
  - High, in order to minimize the miss rate
- **Write through or write back?**
  - Write back, since we can't afford to perform many small writes to disk

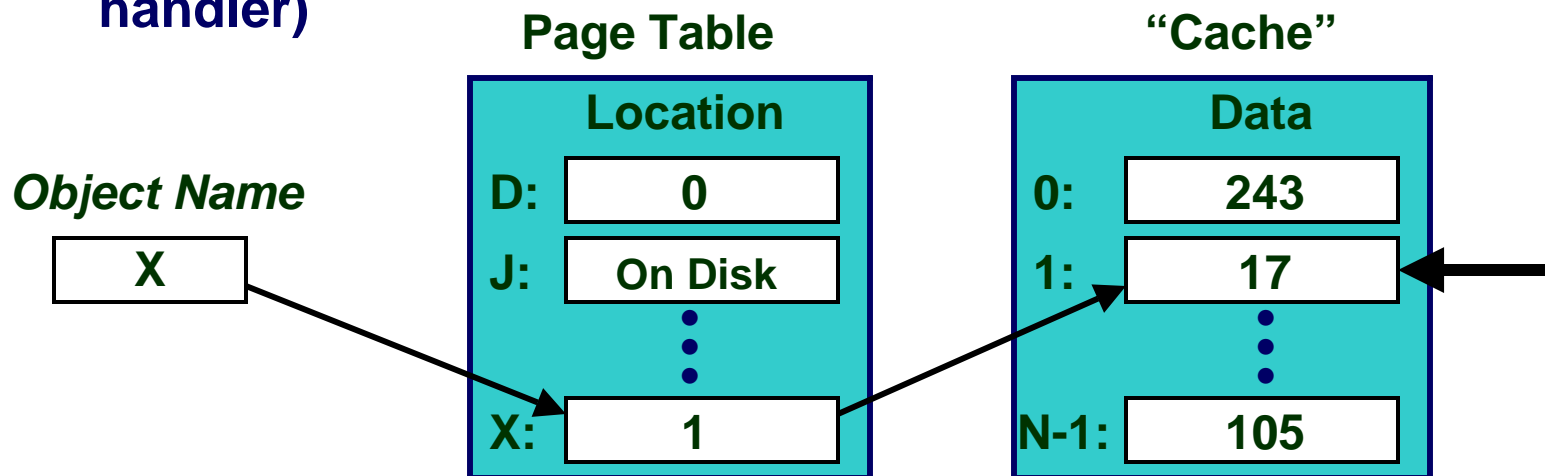
What would the impact of these choices be on:

- **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 DRAM “Cache”

## DRAM Cache

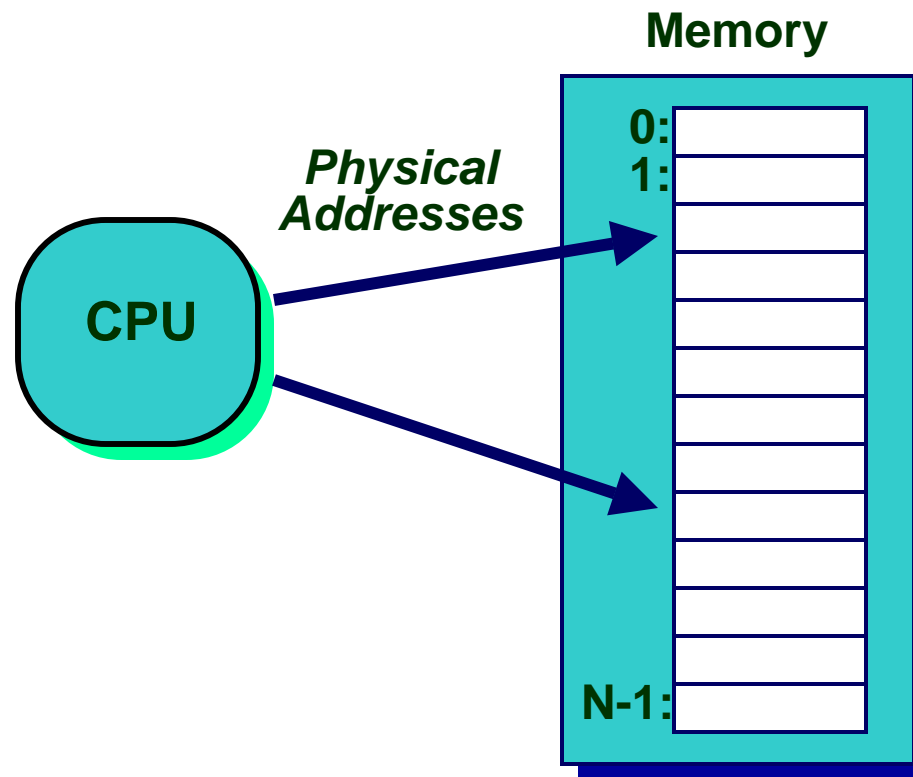
- Each allocated page of virtual memory has an associated entry in the *page table*
- Maps virtual pages to physical pages
- Page table entry exists even if the page is not in memory
  - Specifies disk address
  - Only way to indicate where to find page
- Operating system retrieves this information (in the page fault handler)



# A System with Physical Memory Only

## Examples:

- Most Cray machines, early PCs, nearly all embedded systems...

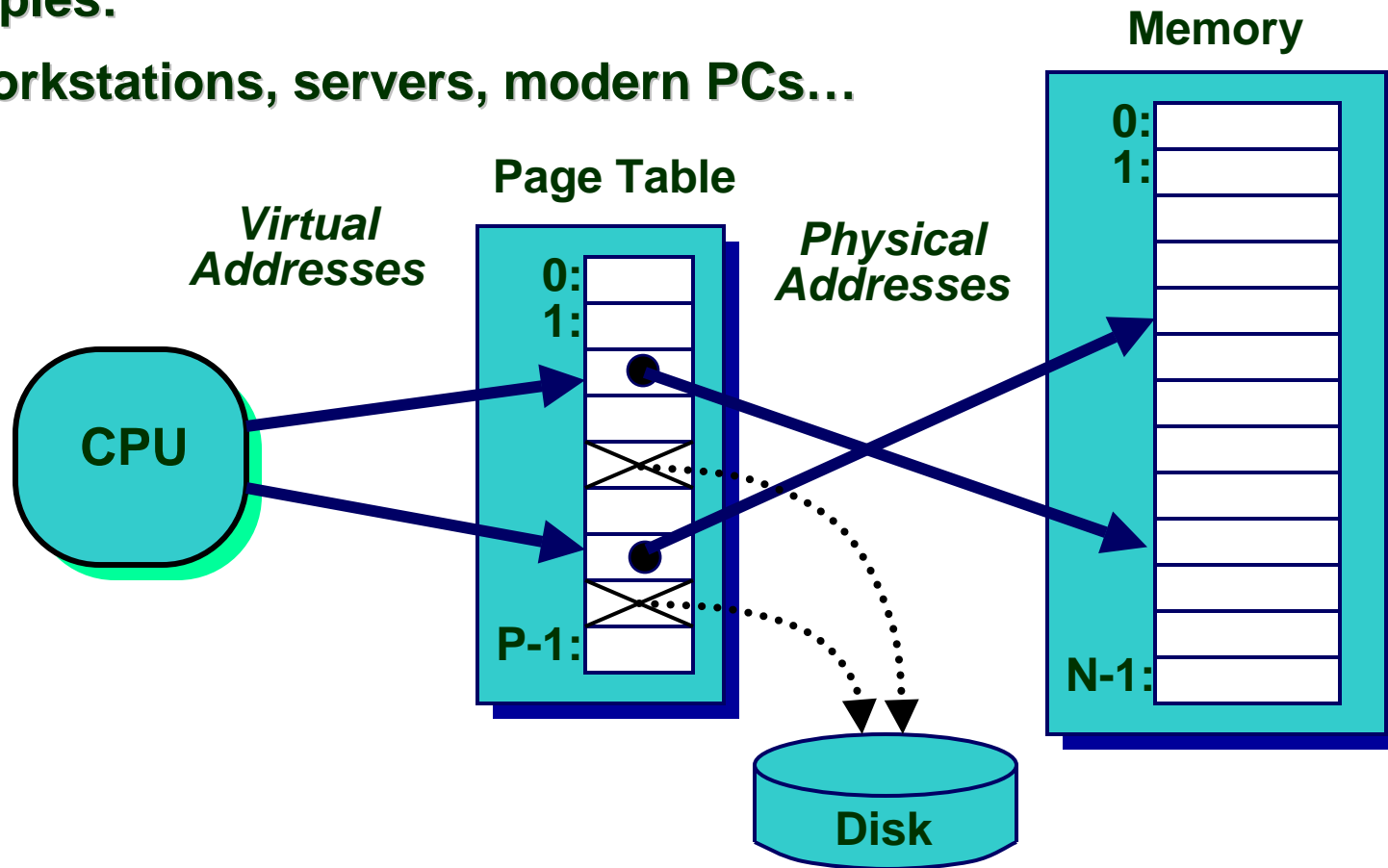


- Addresses generated by the CPU correspond directly to bytes in physical memory: addresses are real addresses in physical memory

# A System with Virtual Memory

## Examples:

- Workstations, servers, modern PCs...



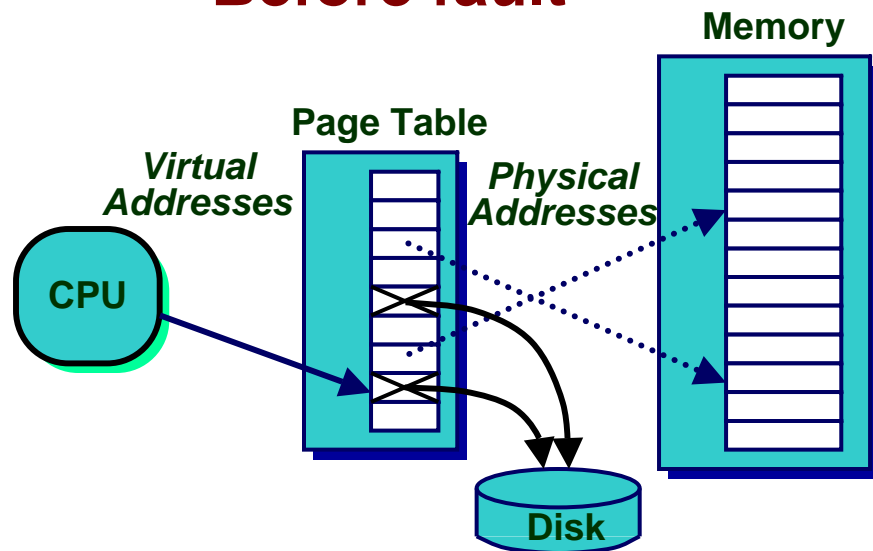
- **Address Translation:** Hardware converts virtual addresses to physical addresses via OS-managed lookup table (page table)

# Page Faults (like “Cache Misses”)

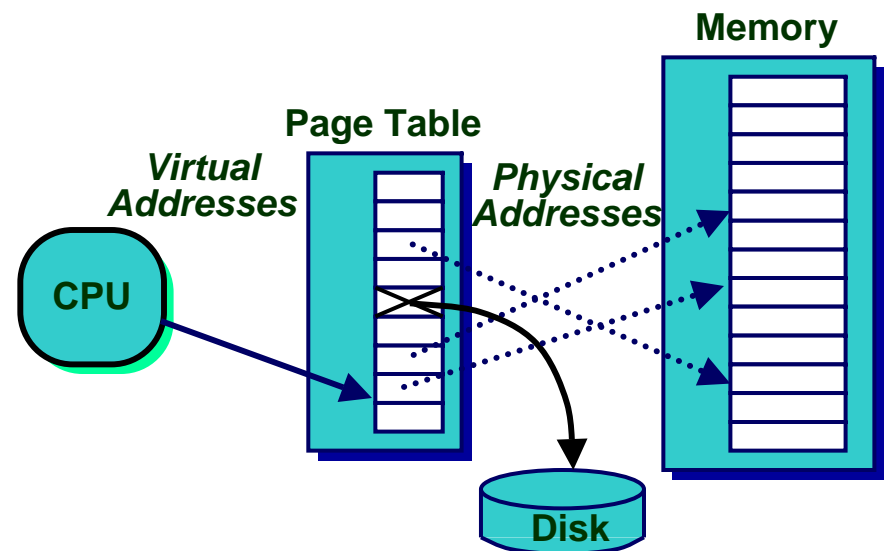
What if an object is on disk rather than in memory?

- 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 frame placement, etc.

**Before fault**



**After fault**



# Servicing a Page Fault

## Processor Signals Controller

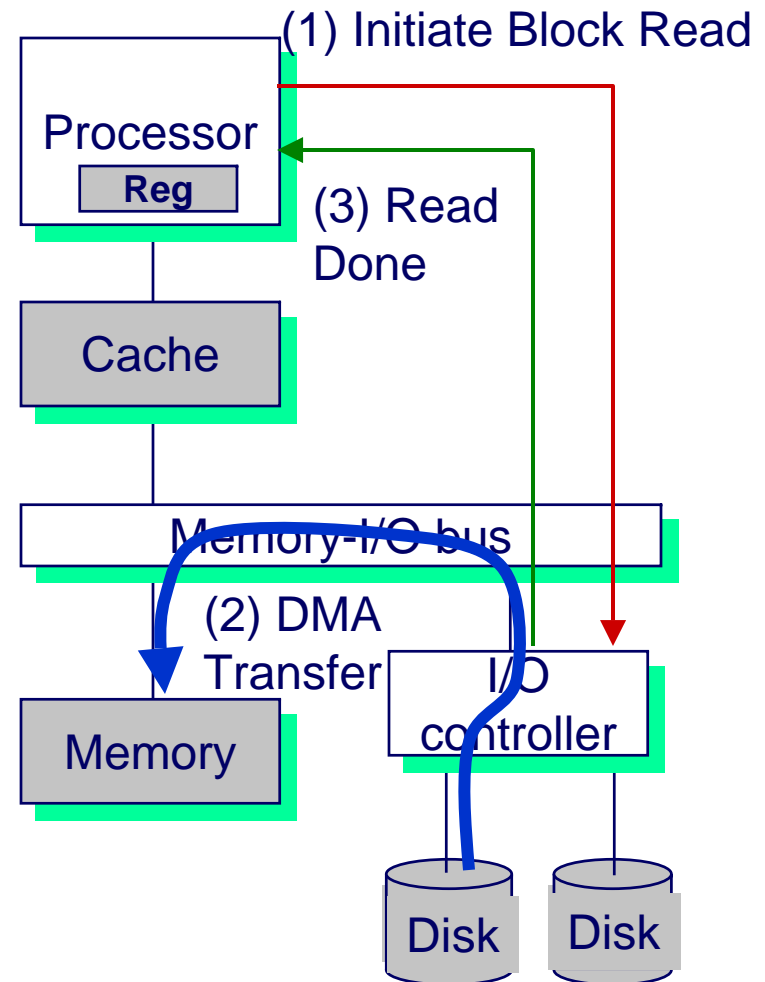
- Read block of length P starting at disk address X and store starting at memory address Y

## Read Occurs

- Direct Memory Access (DMA)
- Under control of I/O controller

## I/O Controller Signals Completion

- Interrupts processor
- OS resumes suspended process and restarts the faulting instruction



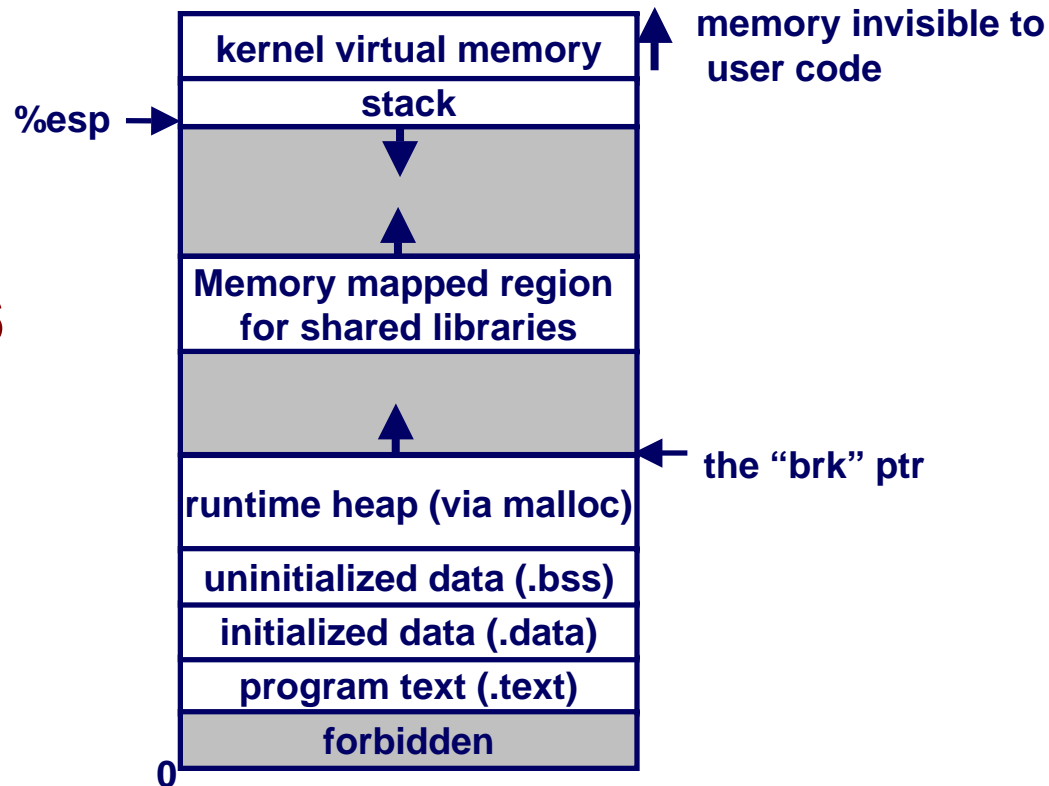
# Benefit #2: Simplifies Memory Management

Multiple processes can reside in physical memory so...

1. How do we keep processes separate?

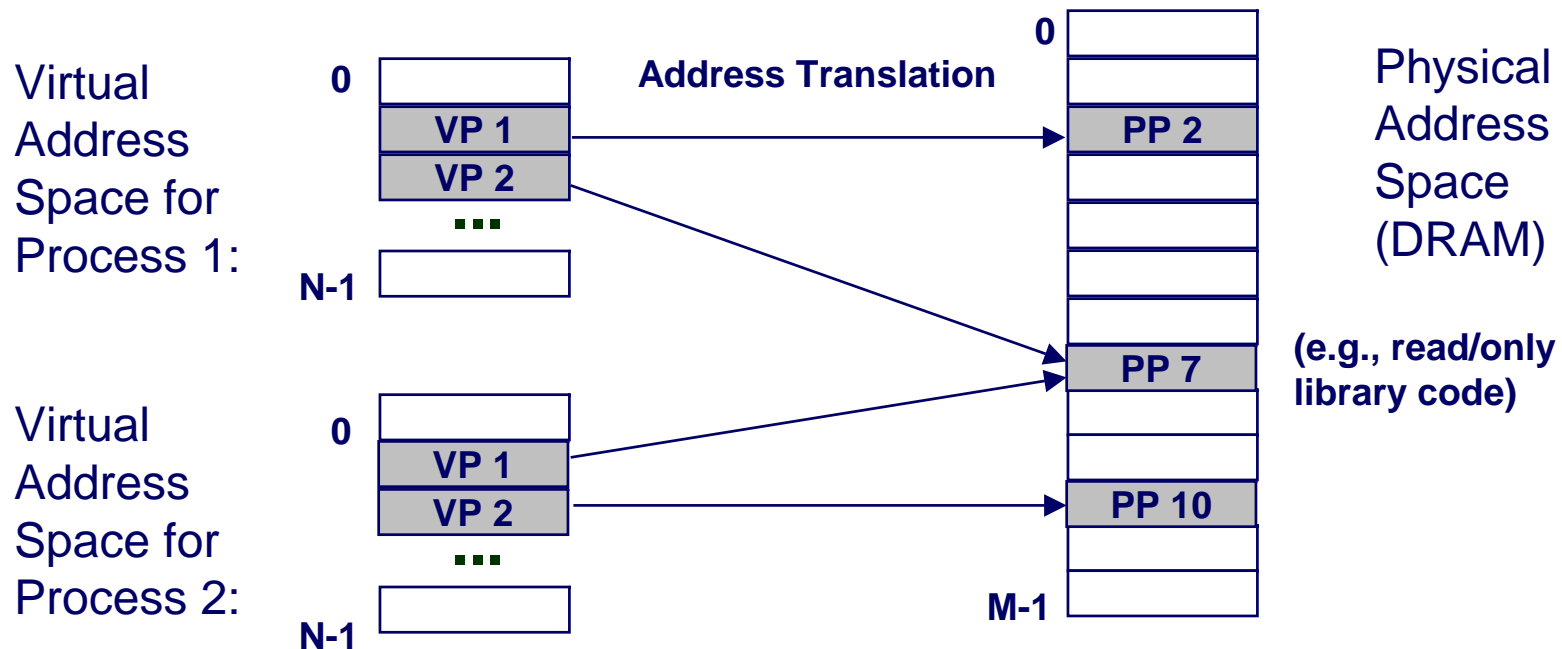
2. How do we find room for new processes?

**Linux/x86  
process  
memory  
image**



# Sol'n: Separate Virtual Address Spaces

- Virtual and physical address spaces divided into equal-sized blocks
  - Blocks are called “pages” (both virtual and physical)
- Each process has its own virtual address space
  - Operating system controls how virtual pages as assigned to physical memory



# VM-Based Memory Management

## Allocating, freeing, and moving memory:

- Simple thanks to page table indirection

## Block sizes:

- Blocks of memory are fixed size
  - Size is equal to *one page* (4KB on x86 Linux systems)

## Allocating contiguous chunks of memory:

- We can map a contiguous range of virtual addresses to disjoint ranges of physical addresses, again thanks to page table indirection (nice when looking for space to hold a new process)

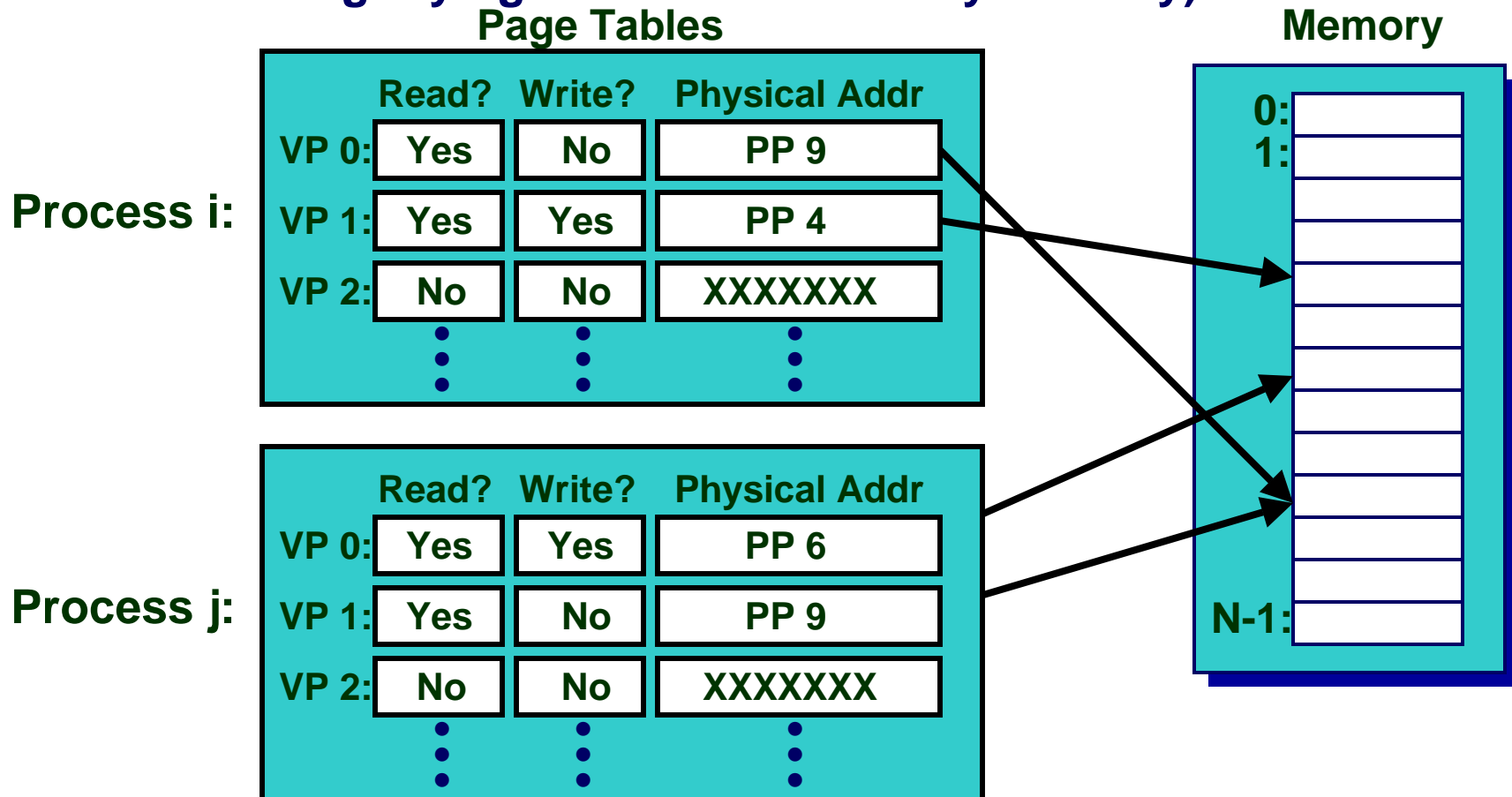
## Protection

- All memory access is through page tables and thus no process can corrupt another's data

# Benefit #3: Protection

## Page table entry contains access rights information

- Hardware enforces this protection (trap into OS if violation occurs e.g. trying to write to read-only memory)



# VM Address Translation

## Virtual Address Space

- $V = \{0, 1, \dots, N-1\}$

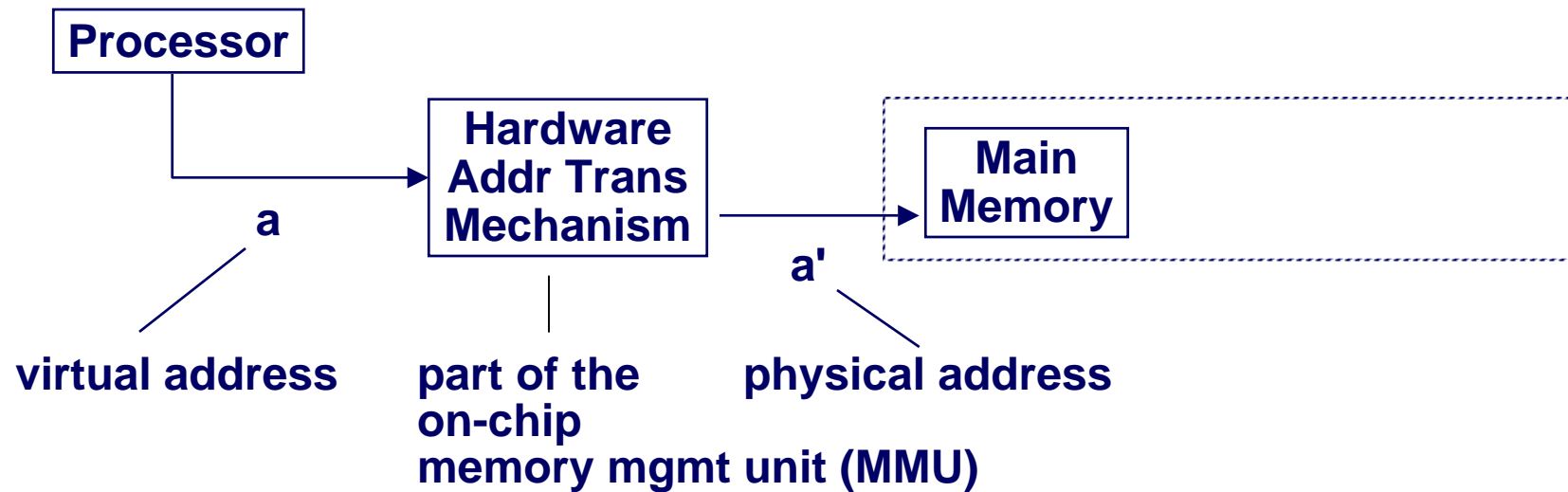
## Physical Address Space

- $P = \{0, 1, \dots, M-1\}$
- $M < N$  (i.e. as usual the physical address space is smaller than the virtual address space)

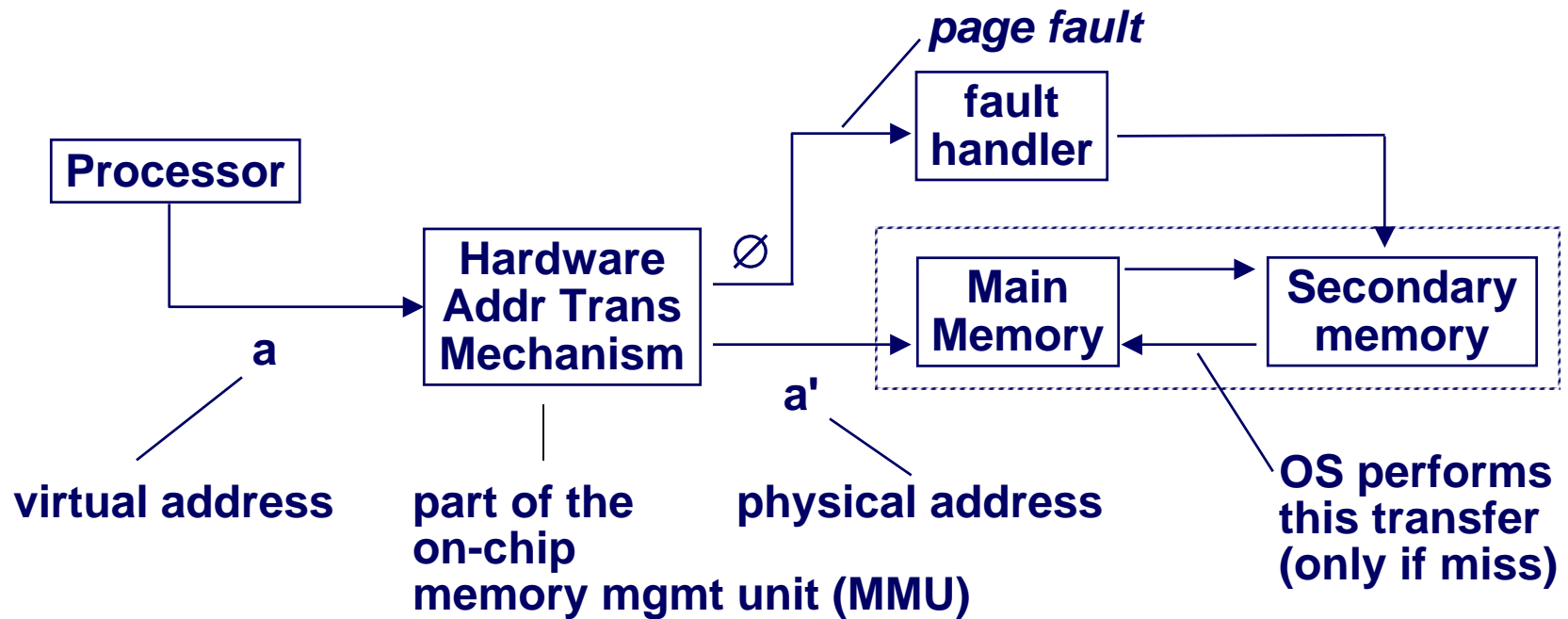
## Address Translation

- $\text{MAP}: V \rightarrow P \cup \{\emptyset\}$
- For virtual address  $a$ :
  - $\text{MAP}(a) = a'$  if data at virtual address  $a$  at physical address  $a'$  in  $P$
  - $\text{MAP}(a) = \emptyset$  if data at virtual address  $a$  not in physical memory
    - » Either invalid or stored on disk

# VM Address Translation: Hit



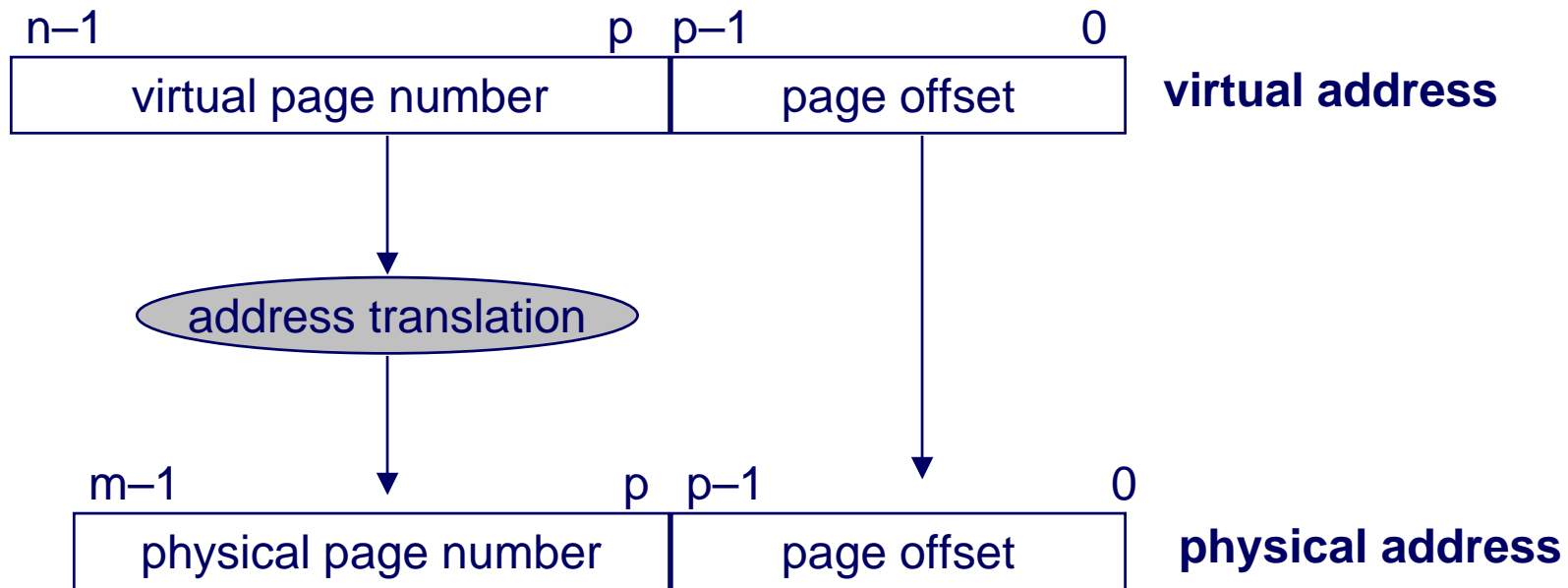
# VM Address Translation: Miss



# VM Address Translation

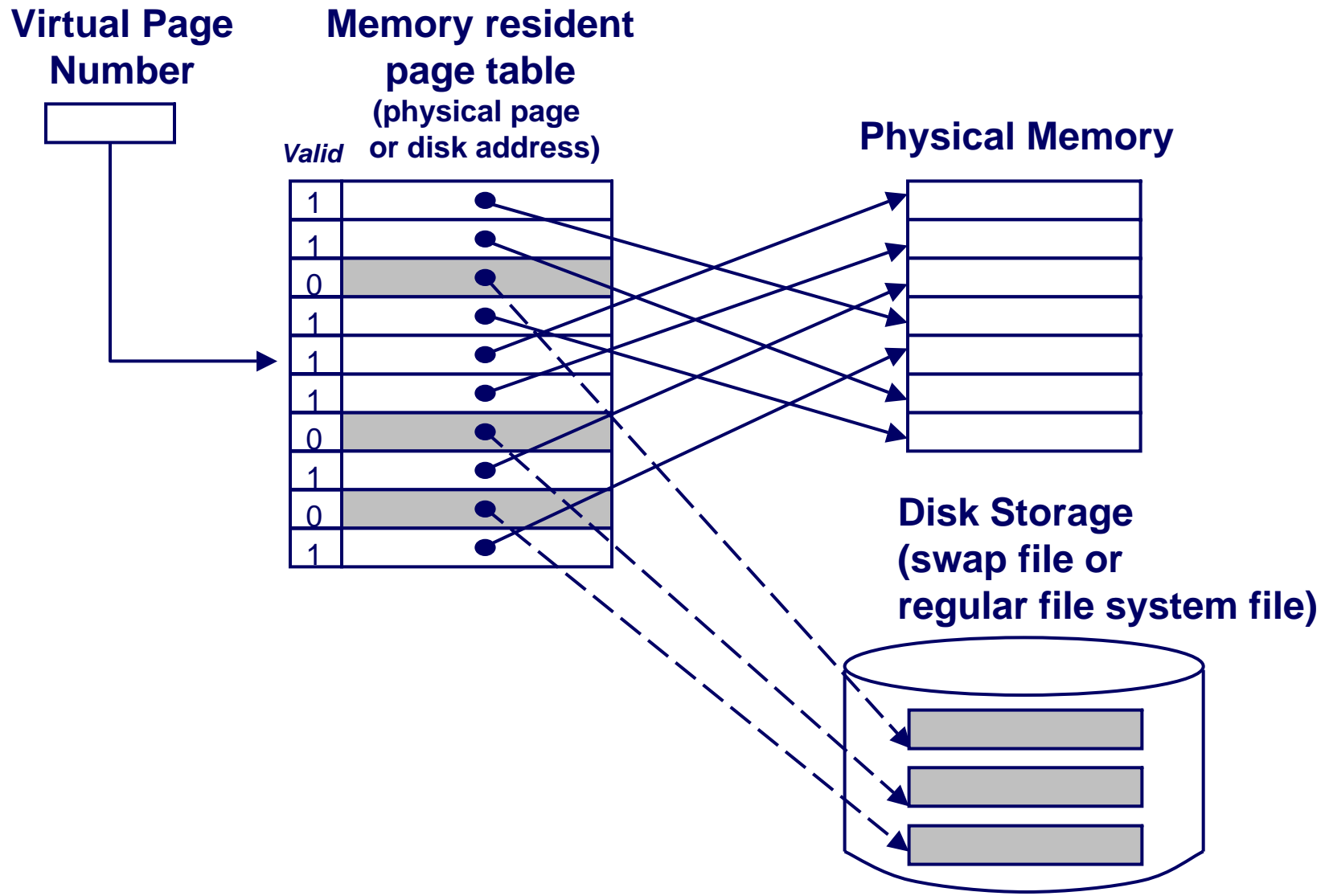
## Parameters

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

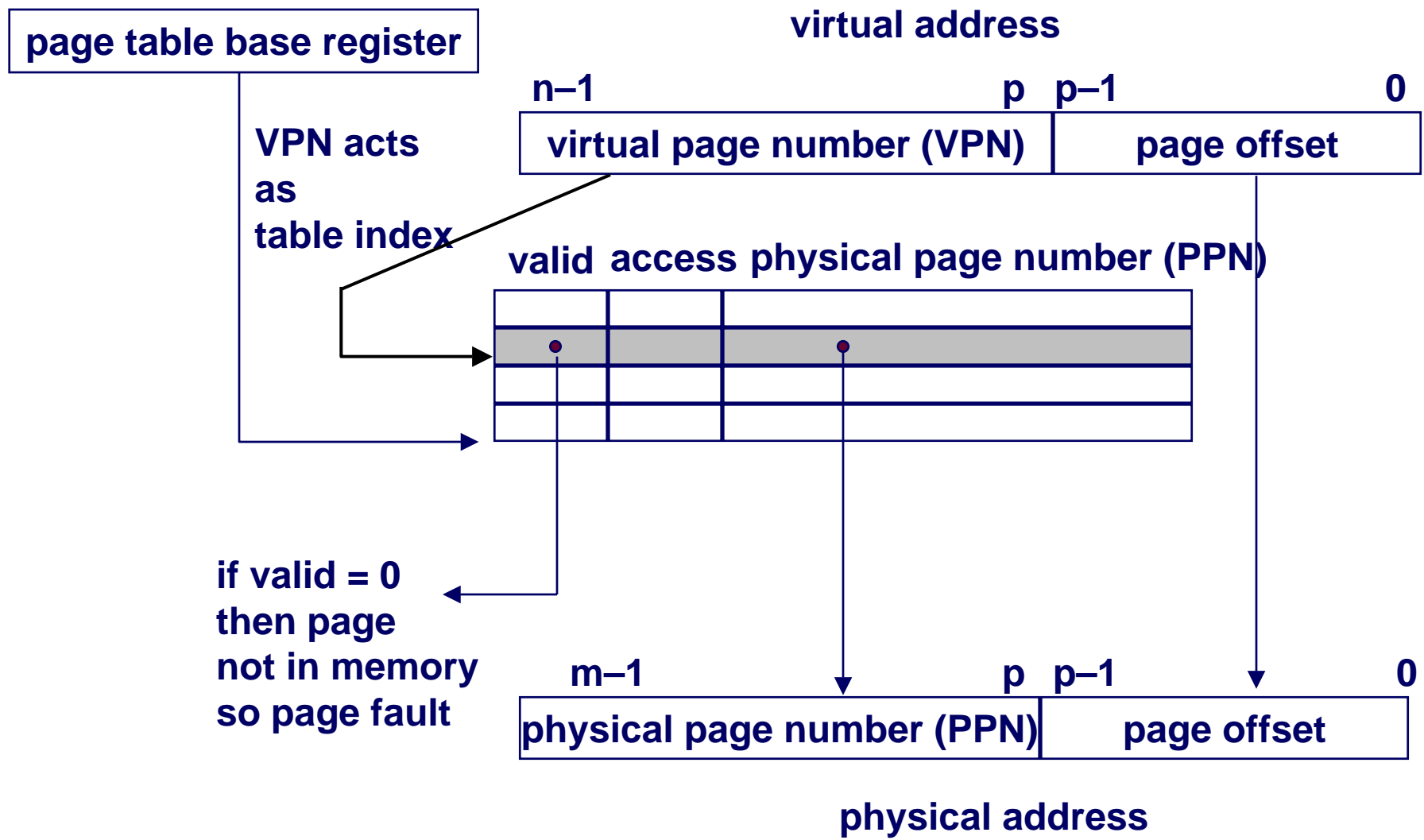


**Page offset bits don't change as a result of translation**

# Page Tables



# Address Translation via Page Table

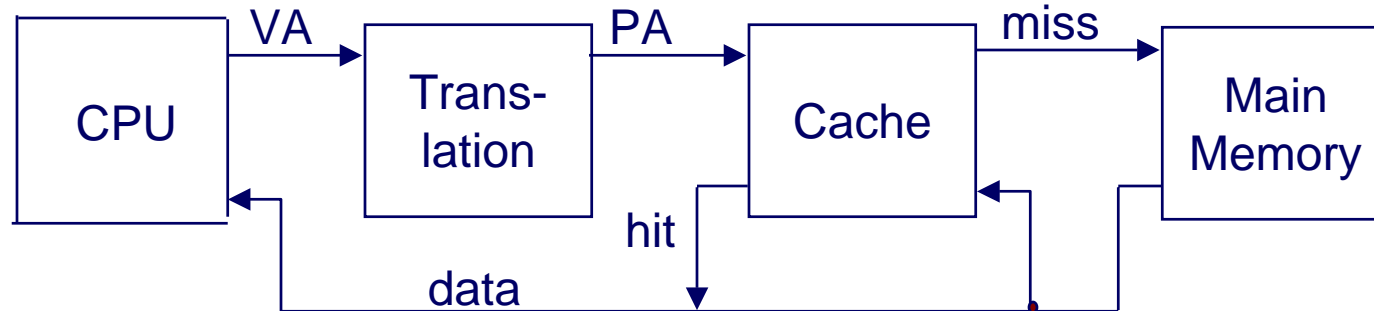


# Page Table Operation

## Checking Protection

- Access rights field indicates allowable access modes
  - E.g., read-only, read-write, execute-only
  - Typically supports multiple protection modes (e.g., kernel vs. user)
- Protection violation fault occurs if user doesn't have the necessary permissions

# Integrating VM and Cache



## Most Caches “Physically Addressed”

- Accessed by physical addresses
- Allows multiple processes to have blocks in cache at same time
- Allows multiple processes to share pages
- Cache need not be concerned with protection issues
  - Access rights checked as part of address translation

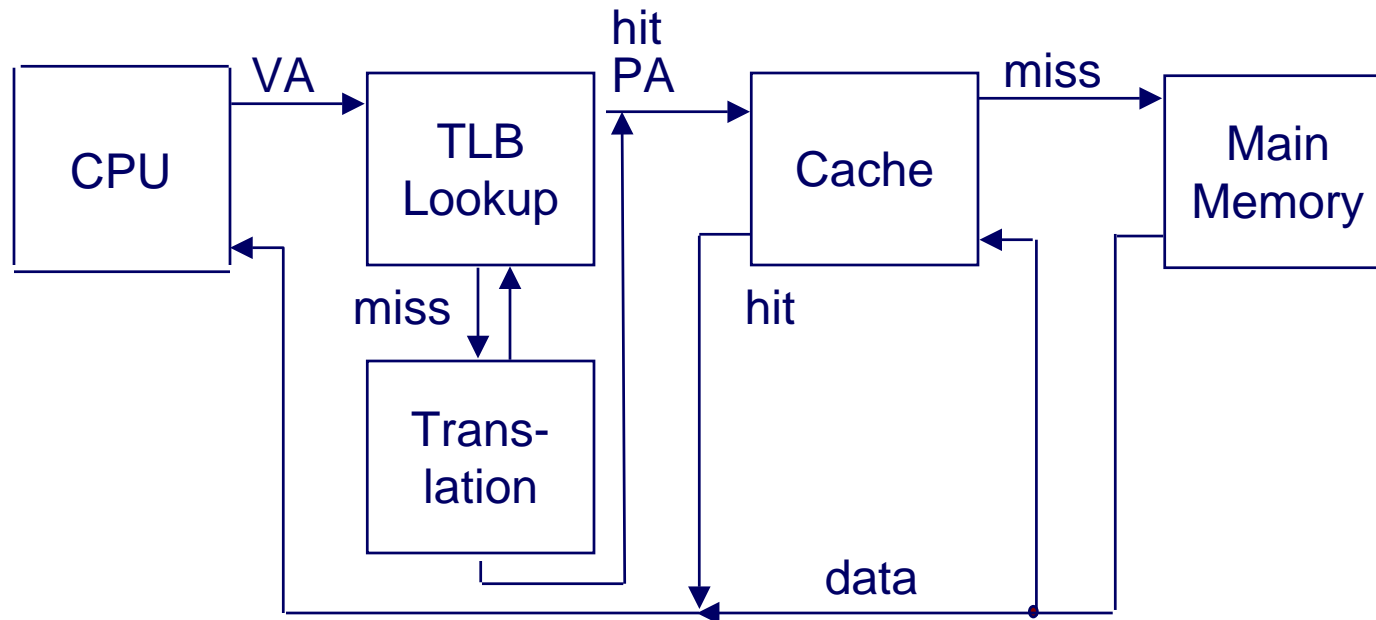
## Perform Address Translation Before Cache Lookup

- But this could involve a memory access itself (of the PTE)
- So page table entries can also be cached (in a TLB)

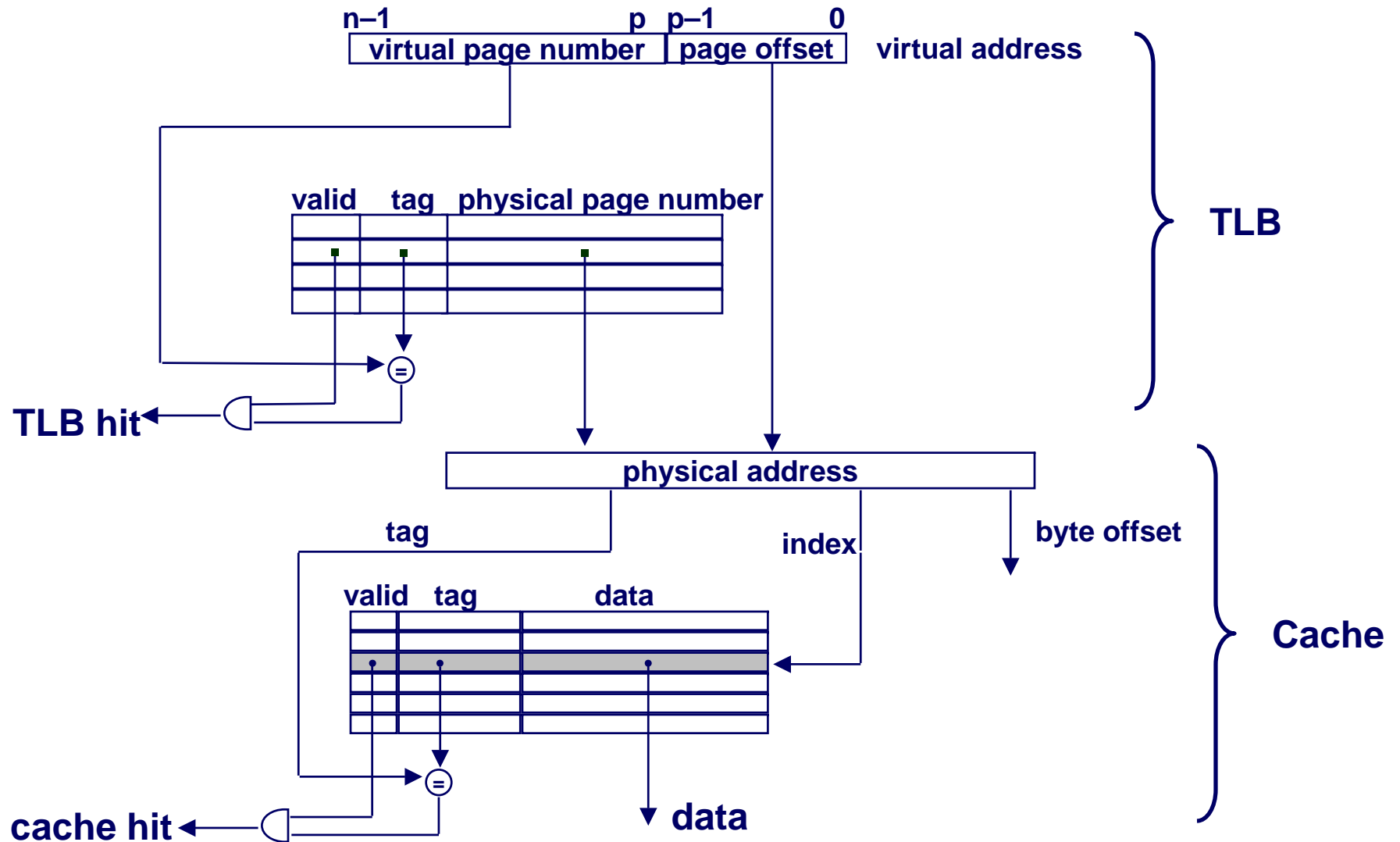
# Speeding up Translation with a TLB

## “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



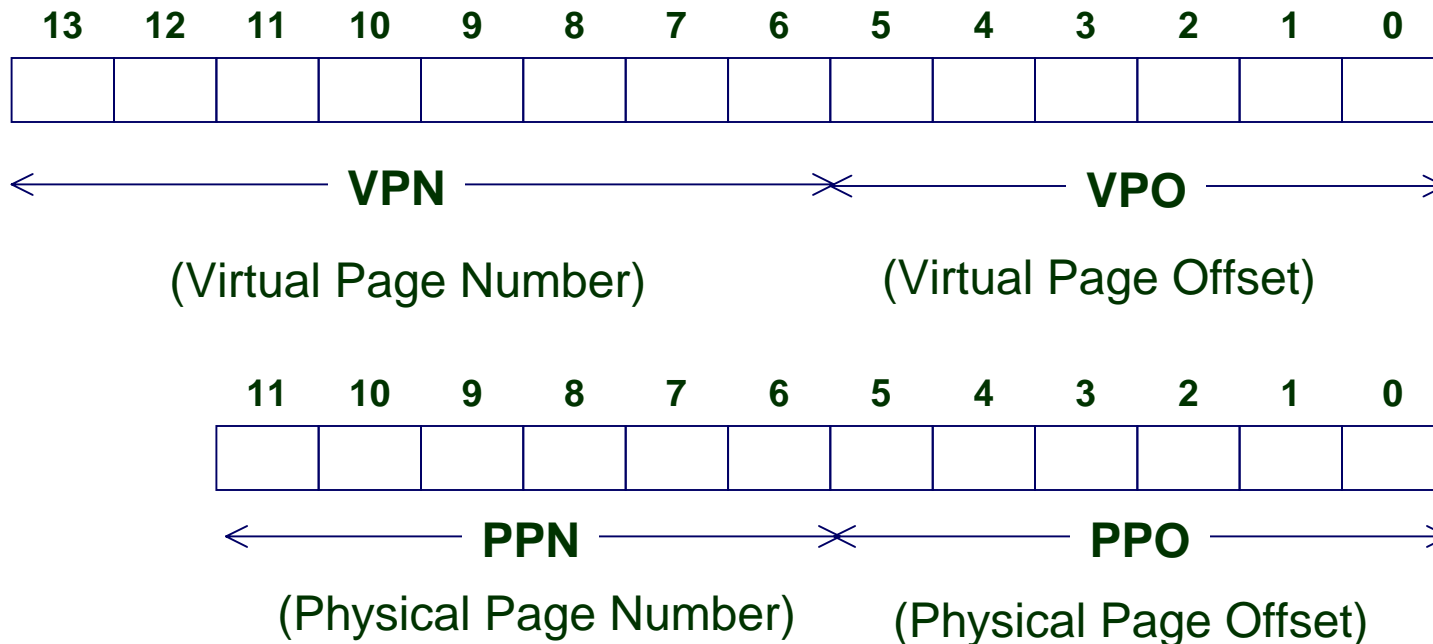
# Address Translation with a TLB



# Simple Memory System Example

## Addressing

- 14-bit virtual addresses (16384 bytes) (256 pages)
- 12-bit physical address (4096 bytes) (64 pages)
- Page size = 64 bytes ( $2^6$  so 6 bits form VPO and PPO)



# Simple Memory System Page Table

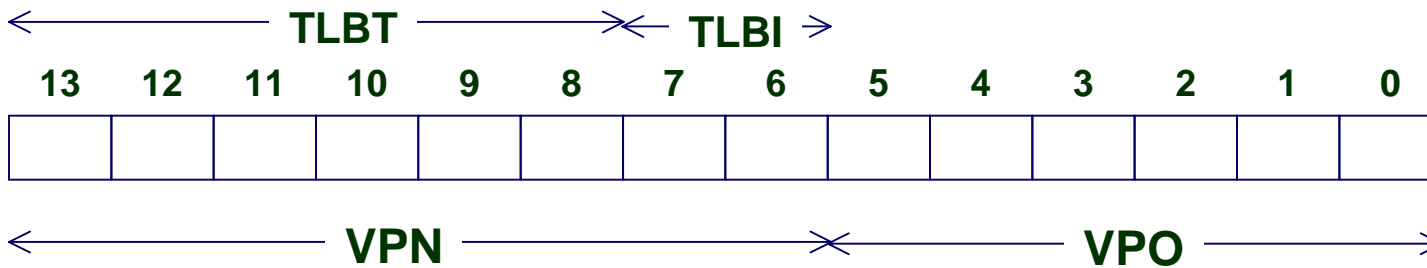
- Only showing first 16 of 256 entries

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

- 4 sets
- 4-way associative (4 lines per set)
- Implements napping from VPN to PPN
- 2 bits of VPN index into cache, remainder acts as the tag

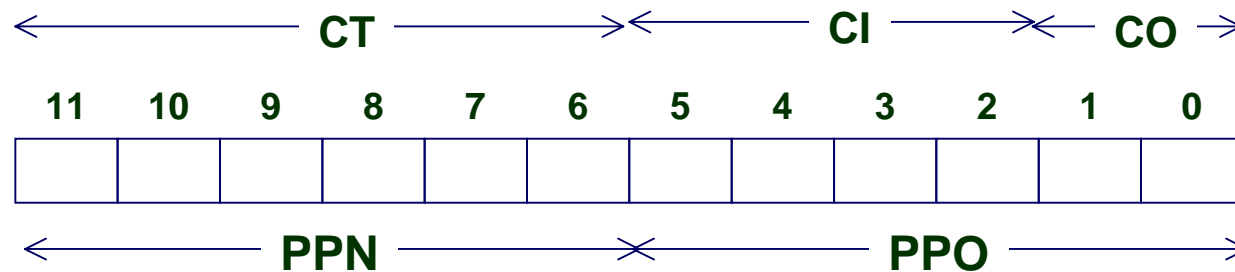


Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	-	0	09	0D	1	00	-	0	07	02	1
1	03	2D	1	02	-	0	04	-	0	0A	-	0
2	02	-	0	08	-	0	06	-	0	03	-	0
3	07	-	0	03	0D	1	0A	34	1	02	-	0

# Simple Memory System Cache

## Cache

- 16 lines, 4-byte block, direct mapped, physically addressed
- 4 bytes per block so 2 bits ( $2^2 = 4$ ) required for CO, 16 lines so 4 bits ( $2^4 = 16$ ) required for CI, remainder is CT



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 Example #1

<b>TLBT</b>						<b>TLBI</b>		<b>VA: 0x03d4</b>						
<b>0x03</b>						<b>0x03</b>								
<b>13</b>	<b>12</b>	<b>11</b>	<b>10</b>	<b>9</b>	<b>8</b>	<b>7</b>	<b>6</b>	<b>5</b>	<b>4</b>	<b>3</b>	<b>2</b>	<b>1</b>	<b>0</b>	<b>TLB hit?</b>
<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>0</b>	<b>1</b>	<b>0</b>	<b>1</b>	<b>0</b>	<b>0</b>	<b>Page fault?</b>
<b>VPN</b>								<b>VPO</b>						<b>PPN?</b>
<b>0x0f</b>								<b>0x14</b>						

<b>CT</b>						<b>CI</b>				<b>CO</b>		<b>Cache hit?</b>
<b>0x0d</b>						<b>0x05</b>				<b>0x0</b>		
<b>11</b>	<b>10</b>	<b>9</b>	<b>8</b>	<b>7</b>	<b>6</b>	<b>5</b>	<b>4</b>	<b>3</b>	<b>2</b>	<b>1</b>	<b>0</b>	<b>Cache byte?</b>
<b>0</b>	<b>0</b>	<b>1</b>	<b>1</b>	<b>0</b>	<b>1</b>	<b>0</b>	<b>1</b>	<b>0</b>	<b>1</b>	<b>0</b>	<b>0</b>	
<b>PPN</b>						<b>PPO</b>						
<b>0x0d</b>						<b>0x14</b>						

# Address Translation Example #2

<b>TLBT</b>						<b>TLBI</b>		<b>VA: 0x03d7</b>						
<b>0x03</b>						<b>0x03</b>								
<b>13</b>	<b>12</b>	<b>11</b>	<b>10</b>	<b>9</b>	<b>8</b>	<b>7</b>	<b>6</b>	<b>5</b>	<b>4</b>	<b>3</b>	<b>2</b>	<b>1</b>	<b>0</b>	<b>TLB hit?</b>
<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>0</b>	<b>1</b>	<b>0</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>Page fault?</b>
<b>VPN</b>								<b>VPO</b>						<b>PPN?</b>
<b>0x0f</b>								<b>0x17</b>						

<b>CT</b>						<b>CI</b>				<b>CO</b>		<b>Cache hit?</b>
<b>0x0d</b>						<b>0x05</b>				<b>0x03</b>		
<b>11</b>	<b>10</b>	<b>9</b>	<b>8</b>	<b>7</b>	<b>6</b>	<b>5</b>	<b>4</b>	<b>3</b>	<b>2</b>	<b>1</b>	<b>0</b>	<b>Cache byte?</b>
<b>0</b>	<b>0</b>	<b>1</b>	<b>1</b>	<b>0</b>	<b>1</b>	<b>0</b>	<b>1</b>	<b>0</b>	<b>1</b>	<b>1</b>	<b>1</b>	
<b>PPN</b>						<b>PPO</b>						
<b>0x0d</b>						<b>0x17</b>						

# Address Translation Example #3

<b>TLBT</b>						<b>TLBI</b>		<b>VA: 0x027c</b>						
<b>0x2</b>						<b>0x1</b>								
<b>13</b>	<b>12</b>	<b>11</b>	<b>10</b>	<b>9</b>	<b>8</b>	<b>7</b>	<b>6</b>	<b>5</b>	<b>4</b>	<b>3</b>	<b>2</b>	<b>1</b>	<b>0</b>	<b>TLB hit?</b>
<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>1</b>	<b>0</b>	<b>0</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>0</b>	<b>0</b>	<b>Page fault?</b>
<b>VPN</b>								<b>VPO</b>						<b>PPN?</b>
<b>0x9</b>								<b>0x3c</b>						

<b>CT</b>						<b>CI</b>				<b>CO</b>		<b>Cache hit?</b>
<b>0x17</b>						<b>0xf</b>				<b>0x0</b>		
<b>11</b>	<b>10</b>	<b>9</b>	<b>8</b>	<b>7</b>	<b>6</b>	<b>5</b>	<b>4</b>	<b>3</b>	<b>2</b>	<b>1</b>	<b>0</b>	<b>Cache byte?</b>
<b>0</b>	<b>1</b>	<b>0</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>0</b>	<b>0</b>	
<b>PPN</b>						<b>PPO</b>						
<b>0x17</b>						<b>0x3c</b>						

# Multilevel Page Tables

## Given:

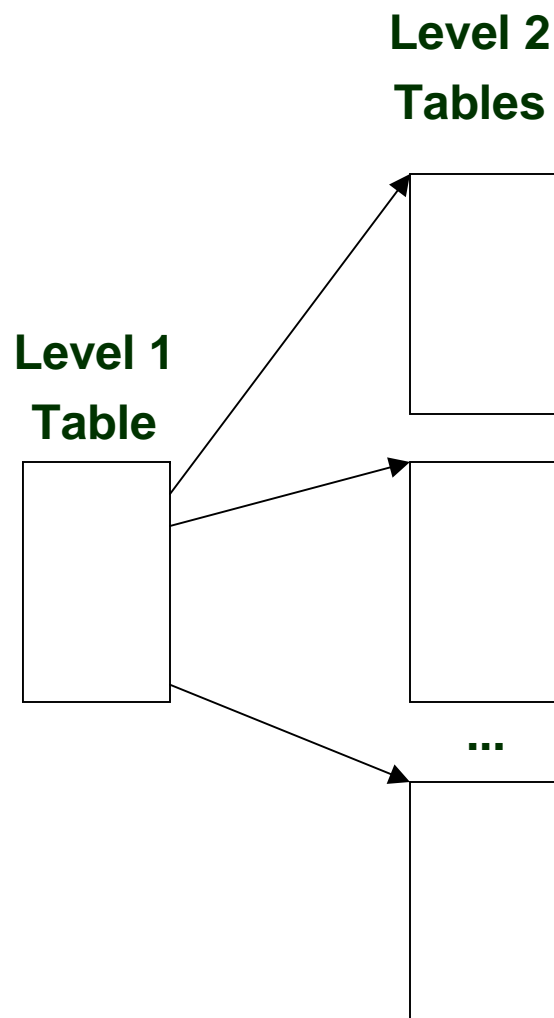
- 4KB ( $2^{12}$ ) page size
- 32-bit address space ( $2^{20}$  pages)
- 4-byte PTE

## Problem:

- Would need a 4 MB page table!
  - $2^{20} * 4$  bytes (with many being null)

## Common solution

- Multilevel 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
  - $2^{10} * 2^{10} * 2^{12} = 2^{32}$  address space



# Summary

## 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

- User virtual address space created by mapping to set of pages
  - Need not be contiguous
  - Allocated dynamically
  - Protection enforced during address translation process
- OS manages many processes simultaneously
  - Continuously switching among processes
  - Especially when one must wait for resource
    - » E.g., disk I/O to handle page fault