

CS / EE 320
**Computer Organization and
Assembly Language**
Spring 2024
Lecture 27 & 28*

Shahid Masud

**Topics: Direct Memory Access (DMA), Virtual Memory (VM),
Cache Coherence in Multicores**

- Direct Memory Access DMA
 - Concept
 - Controller
 - Configurations
- Introduction to Virtual Memory
 - Mapping Physical to Virtual
 - Page Tables
 - Role of Hard Disk
- Cache Coherence in Multicore processors
 - Snoopy Protocols

QUIZ 6
Today's
LECTURE

DIRECT MEMORY ACCESS DMA

Input / Output and DMA

- I/O Modules
- Programmed I/O
- Interrupt-Driven I/O
- **Direct Memory Access✓**



Review

Generic I/O Module

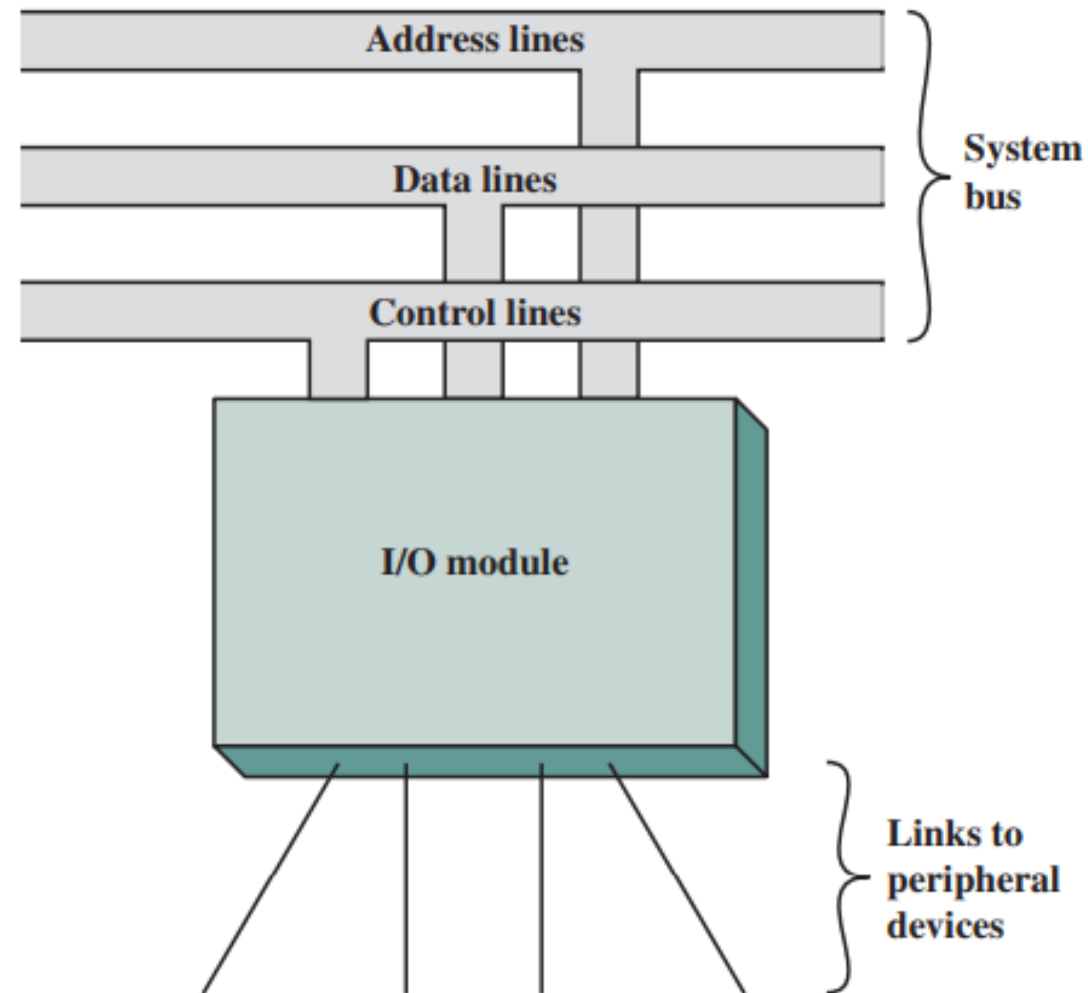


Figure 7.1 Generic Model of an I/O Module

Block Diagram of a Typical I/O Module

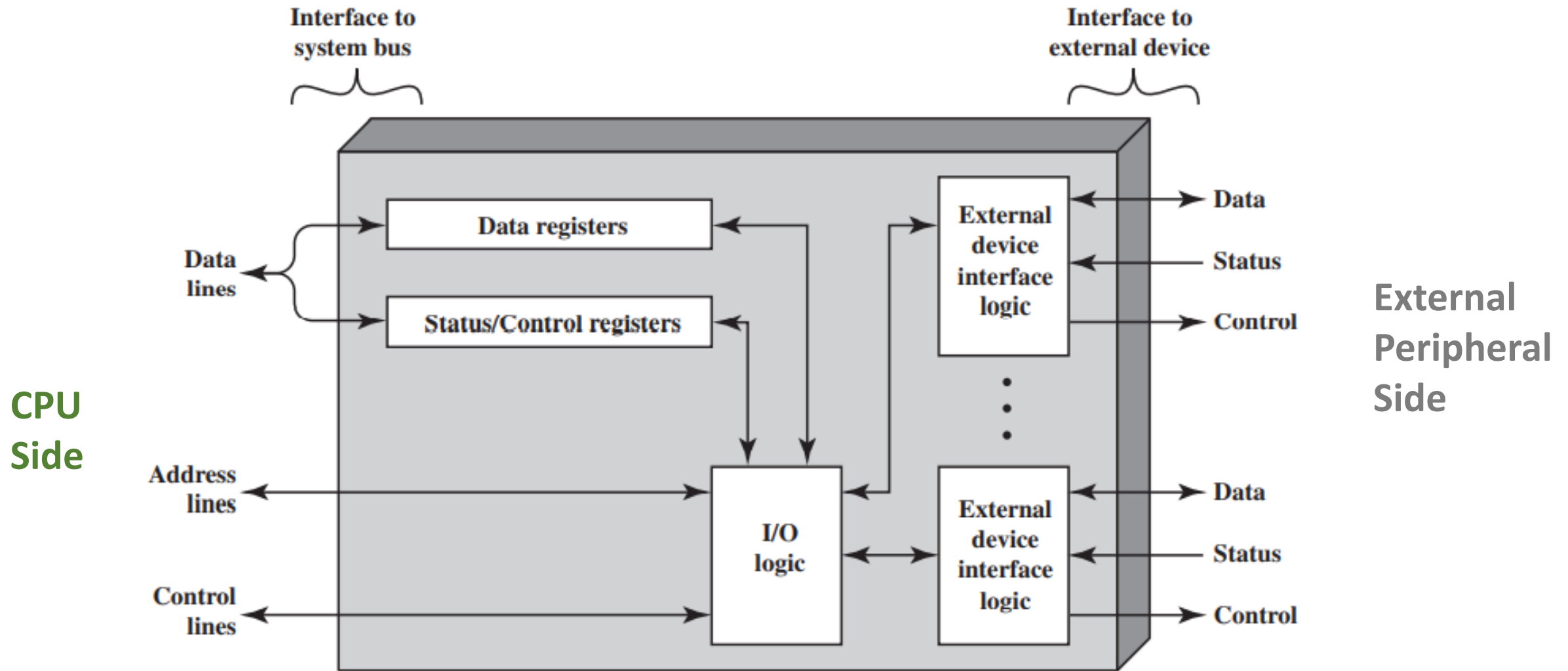


Figure 7.3 Block Diagram of an I/O Module

I/O Channel vs I/O Controllers

- An I/O Module that can take on most of the detailed processing burden
- This presents a high-level interface to the processor
- A primitive or simple I/O module is called 'I/O Controller' or 'Device Controller'
- Personal Computers have I/O Controllers and Main Frames have I/O Channels Or I/O Processors

Three types of Input Output Techniques

1. Programmed I/O
2. Interrupt driven I/O
3. Direct Memory Access (DMA)

	No Interrupts	Use of Interrupts
1,2→ I/O-to-memory transfer through processor	Programmed I/O	Interrupt-driven I/O
3→ Direct I/O-to-memory transfer		Direct memory access (DMA)

Drawback in I/O Techniques

- Interrupt Driven I/O is more efficient than Simple programmed I/O, but still **requires active intervention** of the processor to transfer data between memory and an I/O module, and **any data transfer traverses a path through the processor.**

Both have **two drawbacks**:

1. The I/O Transfer rate is limited by the speed with which the processor can test and service a device.
1. The processor is tied up in managing an I/O transfer, a number of Instructions must be executed for each I/O transfer.

What is Direct Memory Access?

DMA is a capability provided by some computer bus architectures, including PCI, PCMCIA and CardBus, which allows data to be sent directly from an attached device to the memory on the host, freeing the CPU from involvement with the data transfer and thus improving the host's performance.

DMA Programming

The programming of a device's DMA controller is hardware specific. Normally, the OS needs to have the local device address, the physical memory address on the PC, and the size of the memory block to transfer. Then the register that initiates the transfer is set.

Typical DMA Block

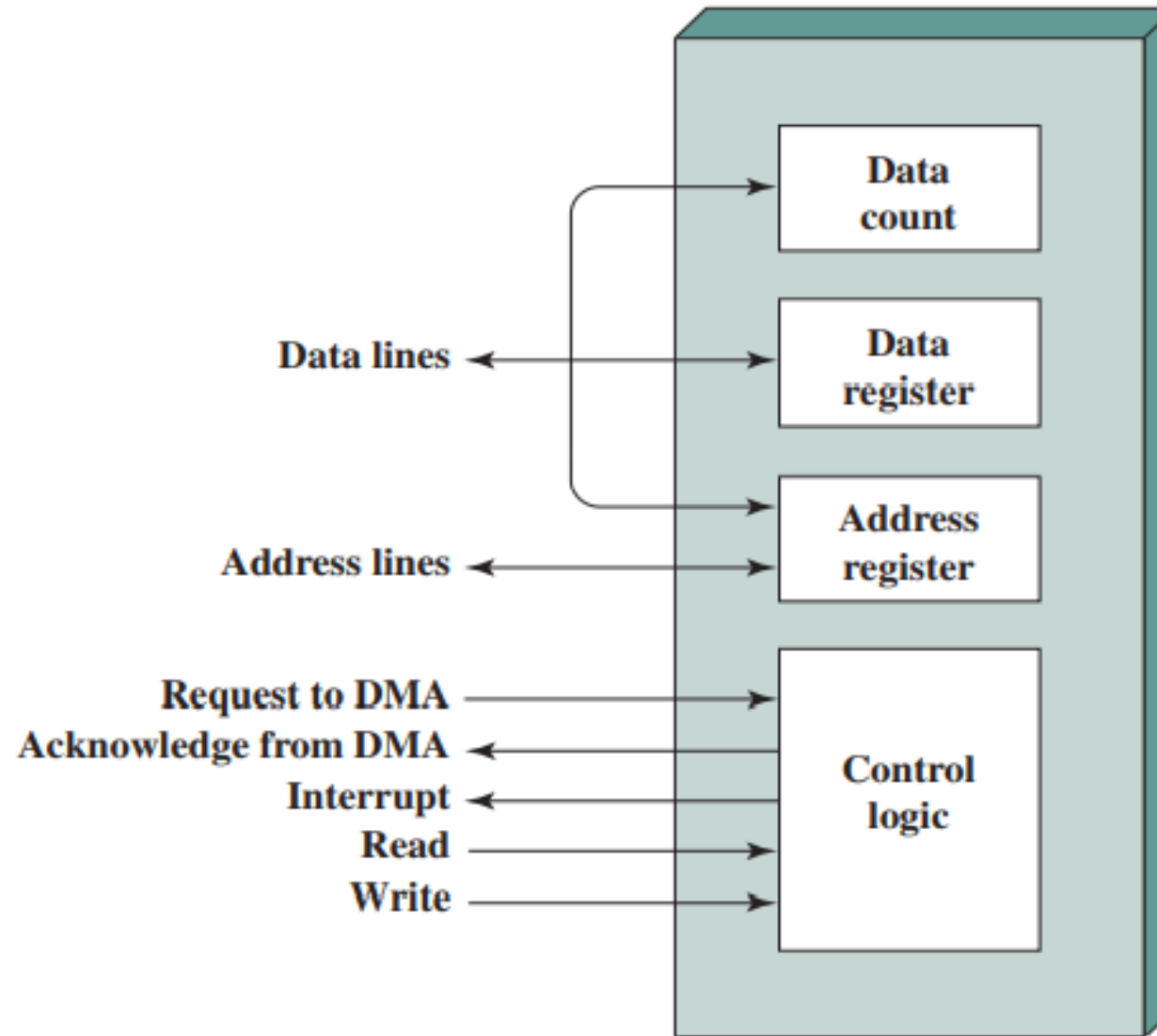
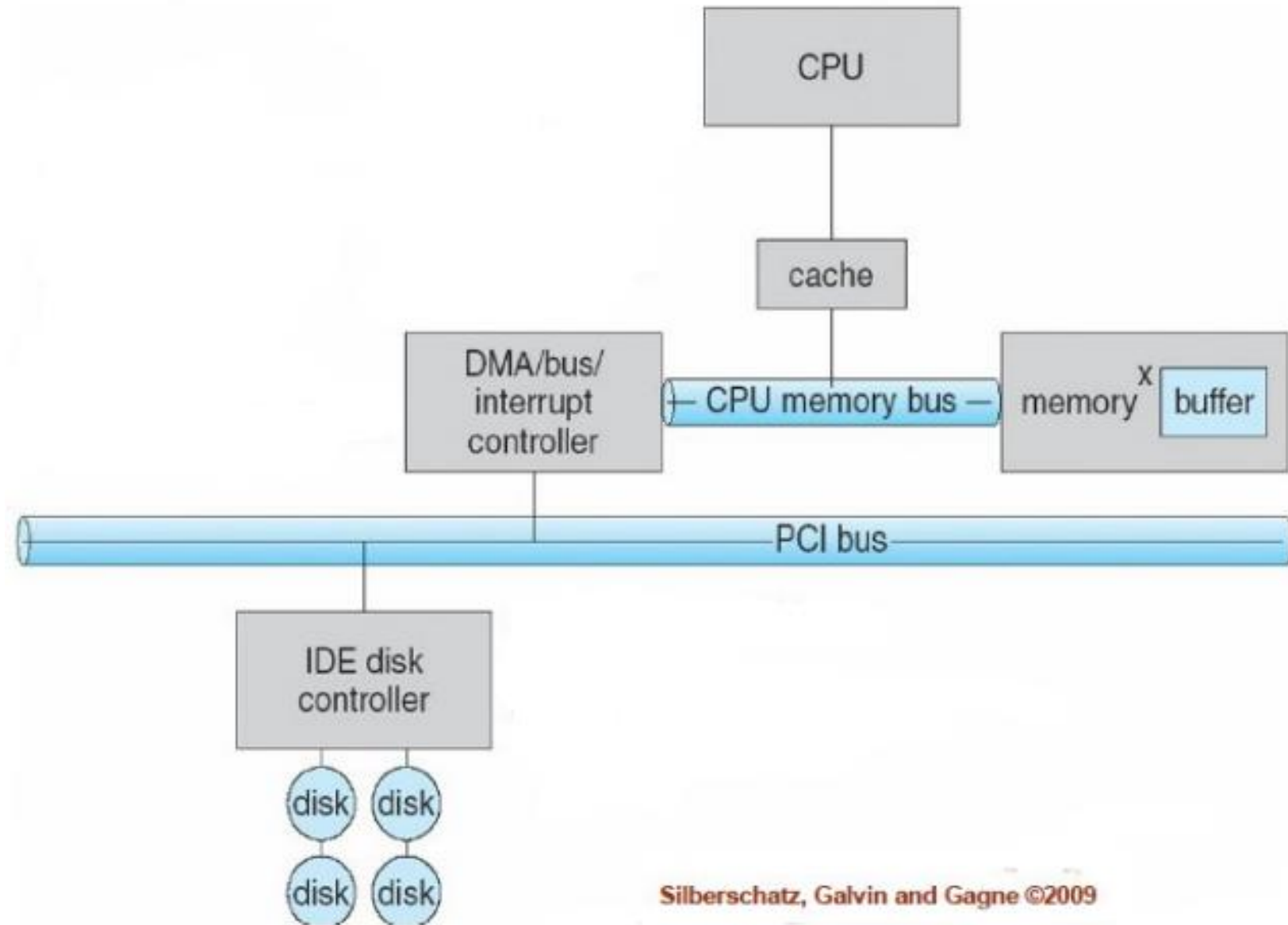


Figure 7.12 Typical DMA Block Diagram

How to DMA?

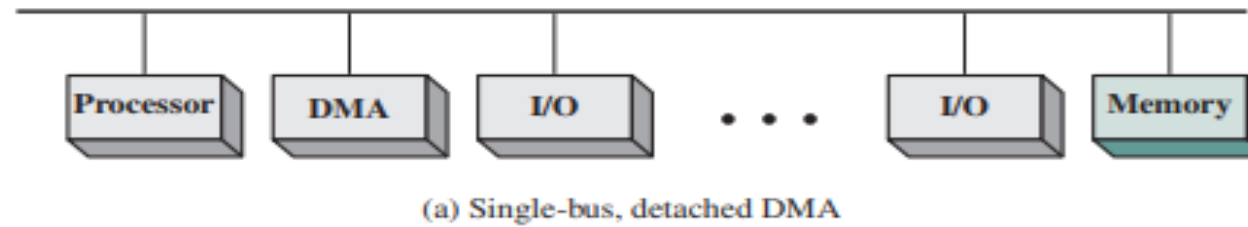
- The starting location in memory to read from, or write to, is communicated on data bus and stored by the DMA module in its address registers.
- The number of words to be read or written is communicated via the data bus and stored in data count register of DMA.
- The processor then continues with other work.
- The DMA module transfers entire block of data, one word at a time, directly to or from memory without going through the processor.
- When the transfer is complete, the DMA module sends an interrupt to the processor.
- The processor is involved only at the beginning and end of data transfer operation.

Basic DMA Architecture

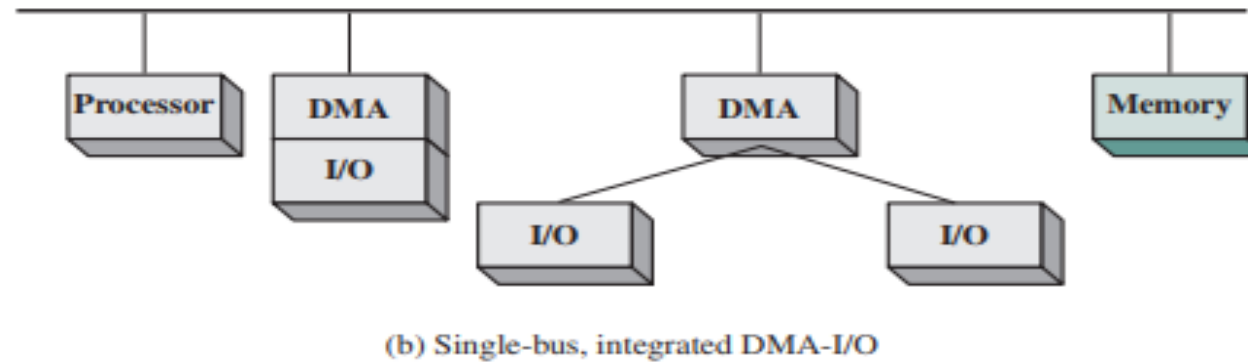


DMA Configurations

Single Bus Detached DMA



Single Bus Integrated DMA-I/O



DMA through I/O Bus

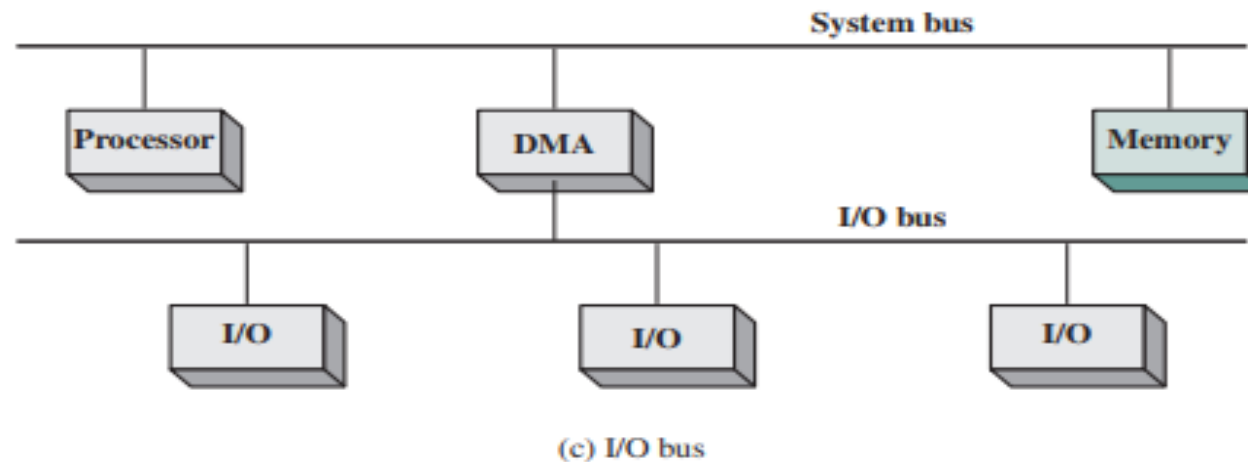
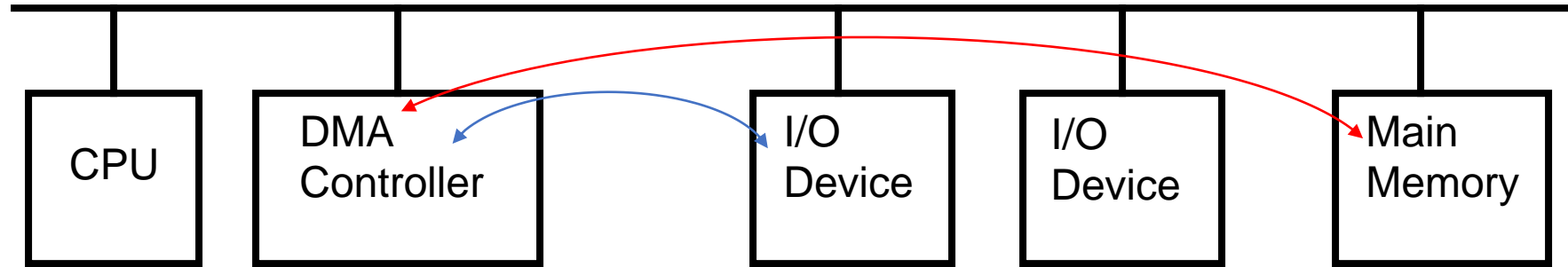


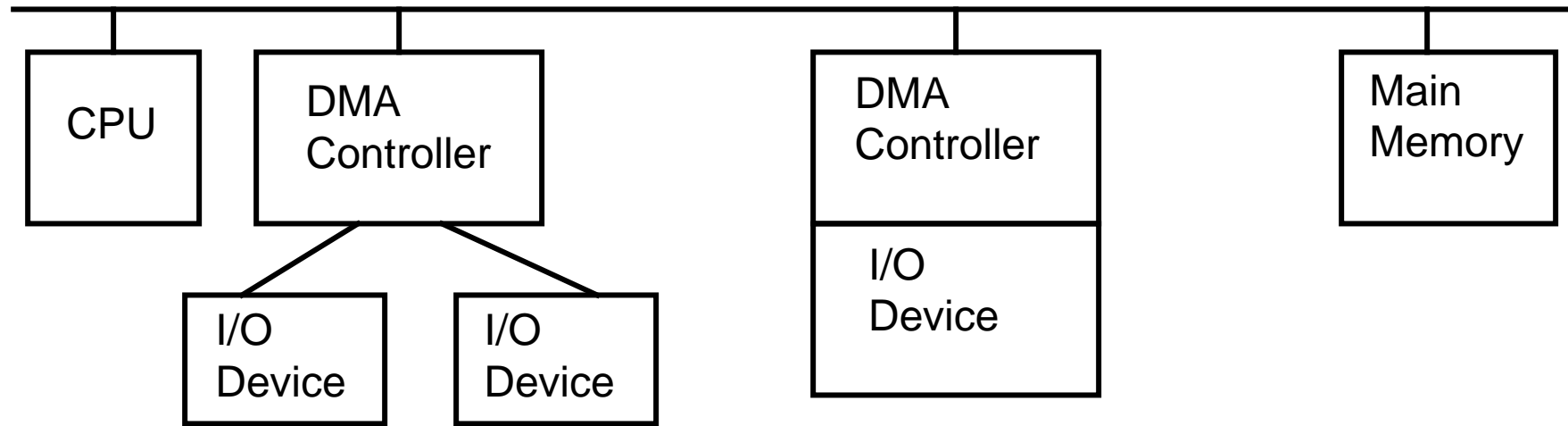
Figure 7.14 Alternative DMA Configurations

DMA Configurations (1)



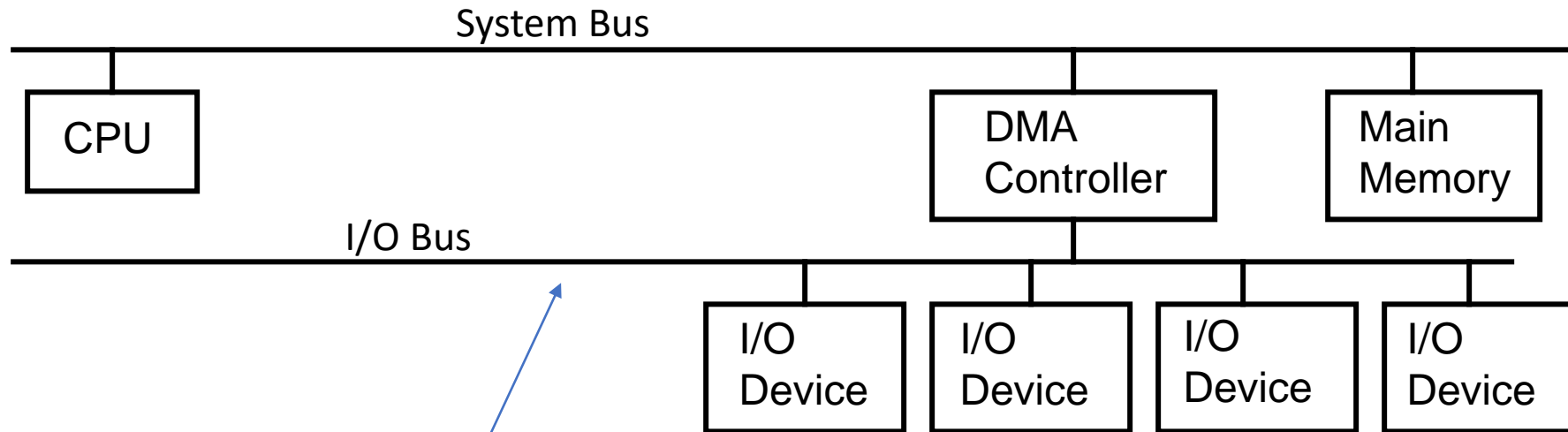
- Single Bus, Detached DMA controller
- Each transfer uses bus twice
 - I/O to DMA then DMA to memory
- CPU is suspended twice

DMA Configurations (2)



- Single Bus, Integrated DMA controller
- Controller may support >1 device
- **Each transfer uses bus once**
 - DMA to memory
- CPU is suspended once

DMA Configurations (3)



- Separate I/O Bus
- Bus supports all DMA enabled devices
- Each transfer uses Bus only once
 - DMA to memory
- CPU is suspended once

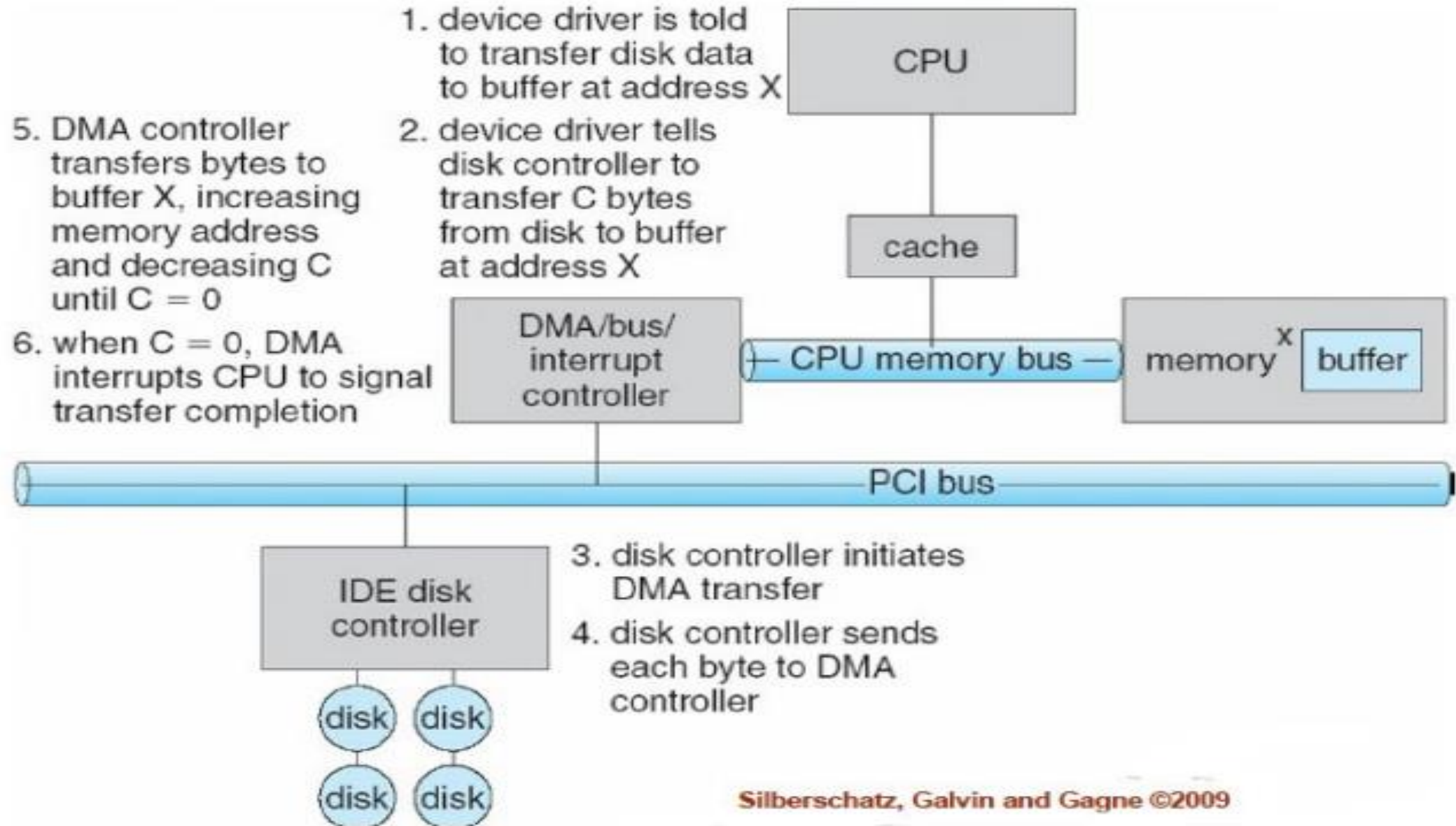
Example of Input

1. Device driver allocates a DMA Buffer, sends a signal to device indicating where to send data, then sleeps.
2. Device writes data to DMA Buffer, raises interrupt when Finished.
3. Interrupt handler gets data from DMA Buffer, acknowledges Interrupt, awakens software to process the data.

Data Arrived for DMA

1. Hardware raises an interrupt to announce that new data has arrived
2. Interrupt handler associates a buffer, tells the hardware where to transfer the data
3. Device writes the data to the buffer, raises another interrupt when transfer is done
4. Interrupt handler dispatches the new data, awakens any relevant process and takes care of housekeeping

DMA Transfer - Graphically



Silberschatz, Galvin and Gagne ©2009

E.g. Intel 8237 DMA Controller Chip

-IOR	1*	40	A7
-IOW	2	39	A6
-MEMR	3	38	A5
-MEMW	4	37	A4
(High or VCC)	5	36	-EOP
READY	6	35	A3
HLDA	7	34	A2
ADSTB	8	33	A1
AEN	9	32	A0
HRQ	10	31	VCC (+5V)
-CS	11	30	DB0
CLK	12	29	DB1
RESET	13	28	DB2
DACK2	14	27	DB3
DACK3	15	26	DB4
DREQ3	16	25	DACK0
DREQ2	17	24	DACK1
DREQ1	18	23	DB5
DREQ0	19	22	DB6
(GND) VSS	20	21	DB7



The 8237 Features:

- Single - One DMA cycle, one CPU cycle interleaved until address counter reaches zero
- Block - Transfer progresses until the word count reaches zero or the EOP signal goes active
- Demand - Transfers continue until TC or EOP goes active or DRQ goes inactive. The CPU is permitted to use the bus when no transfer is requested.
- Cascade - Used to cascade additional DMA controllers. DREQ and DACK is matched with HRQ and HLDA from the next chip to establish a priority chain. Actual bus signals is executed by cascaded chip.
- Memory-to-memory transfer can be performed. The channel 0 Current Address register is the source for the data transfer and channel 1 and the transfer terminates when Current Word Count register becomes 0.

DMA Units of Transfer

- **Single**

A single byte (or word) is transferred. The DMA must release and re-acquire the bus for each additional byte. This is commonly-used by devices that cannot transfer the entire block of data immediately. The peripheral will request the DMA each time it is ready for another transfer.

- **Block/Demand**

Once the DMA acquires the system bus, an entire block of data is transferred, up to a maximum of 64K. If the peripheral needs additional time, it can assert the READY signal to suspend the transfer briefly.

DMA Advantages

- DMA is fast because a dedicated piece of hardware transfers data from one computer location to another and only one or two bus read/write cycles are required per piece of data transferred.
- DMA is usually required to achieve maximum data transfer speed, and thus is useful for high-speed data acquisition devices.
- DMA also minimizes latency in servicing a data acquisition device because the dedicated hardware responds more quickly than interrupts, and transfer time is short.

DMA Disadvantages

- Cost of DMA hardware.
- DMA is useful only for DATA commands. All non-data commands have to be executed by CPU.
- Data has to be stored in continuous locations in memory.
- CPU's intervention is required for initializing DMA logic for every continuous data block transfer.

- During the DMA Transfer, CPU can perform only those operation in which it doesn't require the access of System Bus which means mostly CPU will be in blocked state.
- For how much time CPU remains in the blocked state will depend upon the following modes of DMA Transfer and after that CPU will take back control of system buses from DMAC.

There are three different modes of DMA operations:

- Burst Mode
- Cycle stealing Mode
- Transparent Mode

Burst Mode of DMA (Demand Mode)

- In burst mode, the DMA controller gets the system bus charge, after transfer completion, it will release system bus.
- In burst mode, a whole block of data is shared in one contiguous sequence. Since the DMA controller is allowed access to the system buses by the CPU, it sends all bytes of data in the data block and then yield control of the system buses back to the CPU.
- This mode is beneficial for loading programs or data records into memory.
- The CPU has to wait till that process finishes.
- This is the quickest mode of DMA Transfer since at once a huge amount of data is being transferred.

Cycle Stealing Mode of DMA (Byte Mode)



- In cycle stealing mode, the CPU gets forced by DMA controller to stop the operation and withdraw the control from bus for a short period of time.
- The DMA will release the bus and will request the system bus, like this, DMA will use the cycle clock for sending data bytes.
- It already issues requests via BR, sharing one byte of information per request
- Slow IO device will take some time to prepare data (or word) and within that time CPU keeps the control of the buses.
- Once the data or the word is ready, CPU gives back control of system buses to DMA Controller for 1-cycle in which the prepared word is transferred to memory.
- As compared to Burst mode, this mode is slow since it requires some time which is actually consumed by I/O device while preparing the data.
- Once the DMA controller is granted access to the system bus by the CPU, it transfer one byte of data then it releases the memory access to CPU. Again, for another byte of transfer, it must acquire bus access by CPU via BR and BG signal (BUS REQUEST and BUS GRANT). For Every Byte of Transfer, it acquires bus access and releases it until entire Block of Data Transferred.
- DMA controllers can operate in a cycle stealing mode in which they take over the bus for each byte of data to be transferred and then return control to the CPU.

Comparing Burst Mode and Cycle Stealing Mode of DMA

Parameter	Burst Mode of DMA	Cycle Stealing Mode of DMA
Definition	It is the DMA data transfer technique in which no. of data words are transferred continuously until whole data is not transferred.	It is the data transfer technique in which one data word is transferred and then control is returned to CPU.
Data Transfer	Data transfer Continues until whole data is not transferred.	Data is transferred Only when CPU is idle.
Speed	This is very fast data transfer technique and is used to transfer data for fast speed devices.	It is the slow data transfer technique as data is transferred only when CPU is idle
CPU Utilization	Low CPU Utilization because CPU remains idle until whole data is not transferred.	High CPU utilization because data is transferred when CPU has no task to perform.
Extra Overhead	No need to check CPU idleness	Extra Overhead because every time CPU has to be monitored for idle periods or slots.

Transparent (Interleaving) Mode of DMA

- In transparent mode, the DMA controller will use system bus when the processor does not need system bus.
- Whenever CPU does not require the system buses then only control of buses will be given to DMA Controller.
- In this mode, CPU will not be blocked due to DMA at all.
- This is the slowest mode of DMA Transfer since DMA Controller has to wait for so long time to just even get the access of system buses from the CPU itself.
- The primary advantage of transparent mode is that the CPU never stops executing its programs and the DMA transfer is free in terms of time.
- While the disadvantage is that the hardware needs to determine when the CPU is not using the system buses, which can be complex.

VIRTUAL MEMORY

Three problems in Memory

- **Not Enough RAM in computer**. We want to be able to run programs even if we don't have enough physical memory installed
- **Holes in Address Space** – Run multiple programs, then close some of them. Chunks of distributed un-used memory spaces are apparent
- **Programs overwriting each other** – how to provide Security
- **VIRTUAL MEMORY** uses Indirection – Addresses that program use are artificially MAPPED to real address in Memory. We can control where the memory goes and how to use it
- **PAGE TABLE and TRANSLATIONS** – how the mappings are done and stored
- Also need to worry **how Virtual Memory interacts with Caches**

Problems in using Shared Address Space

1 GB	Physical RAM
2	
3	
4	
5	
6	
7	
8	

Program 1

Program 2

Program 3

Depending upon how programs were called and removed, we may run out of continuous RAM space. Though smaller spaces may exist. This is called **Holes**. Same Physical Spaces maybe Overwritten causing Security Issues.

Idea of Virtual Memory

- Give each program its own Virtual Memory Space
- Separately map each program's virtual memory space to RAM's physical memory space
- Use Disk space if RAM space runs out
- Mapping gives us flexibility how we use Physical RAM space

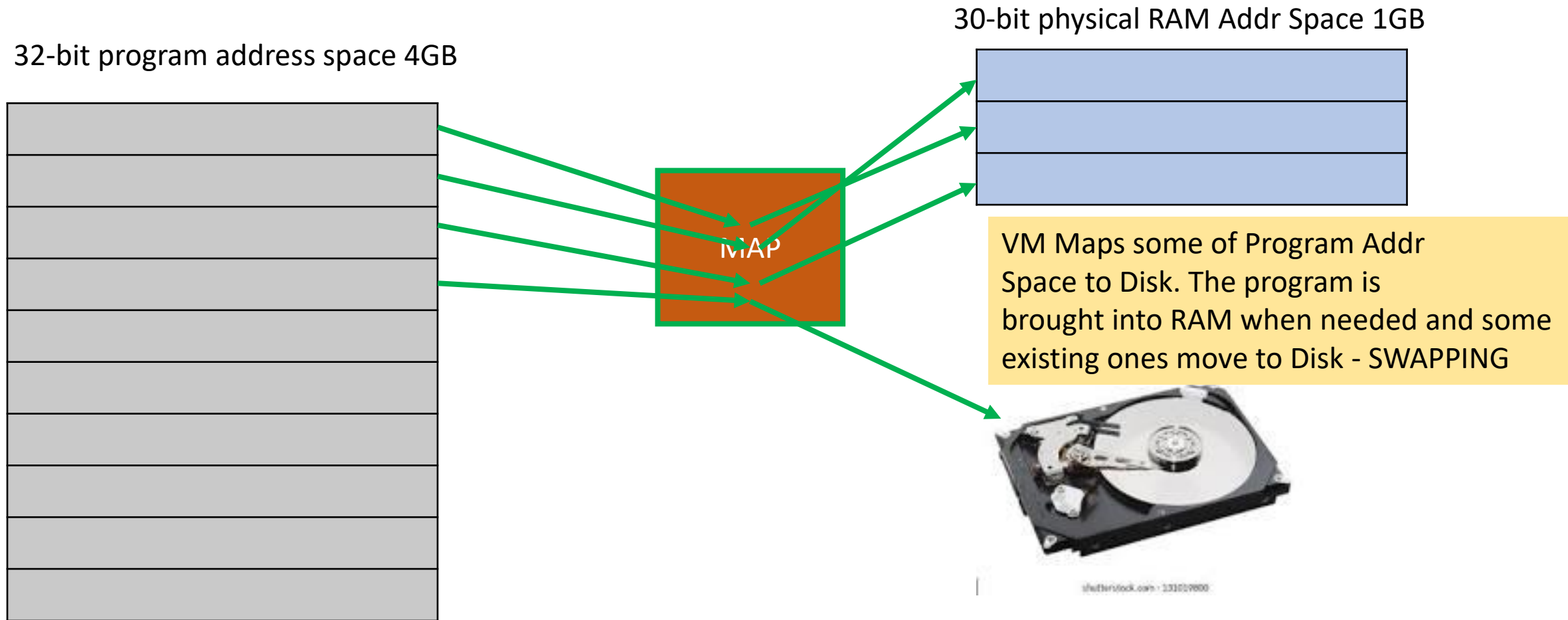
How does Virtual Memory work?

- **Basic idea: separate memory spaces**
 - **Virtual memory**: what the **program** sees
 - e.g., `ld R4, 1024(R0)` accesses **virtual address** $R0 + 1024 = 1024$
 - **Physical memory**: the **physical RAM** in the computer
 - e.g., if you have 2GB of **RAM** installed, you have **physical addresses** 0 to $2^{31}-1$
- **Virtual Addresses (VA)**
 - What the program uses
 - In MIPS, this is the full 32-bit address space: 0 to $2^{32}-1$
- **Physical Addresses (PA)**
 - What the hardware uses to talk to the RAM
 - Address space determined by how much RAM is installed

- Virtual Memory is Implemented using hard disk drive
 - Each virtual page has a place on the disk
 - Some of these may also be in the main memory
 - The subset present in the main memory changes from time to time as per the need
- ISA must support large address spaces
- Hardware + software take care of the rest

Indirection in Virtual Memory

With Virtual Memory, Program Address MAPS to RAM Address

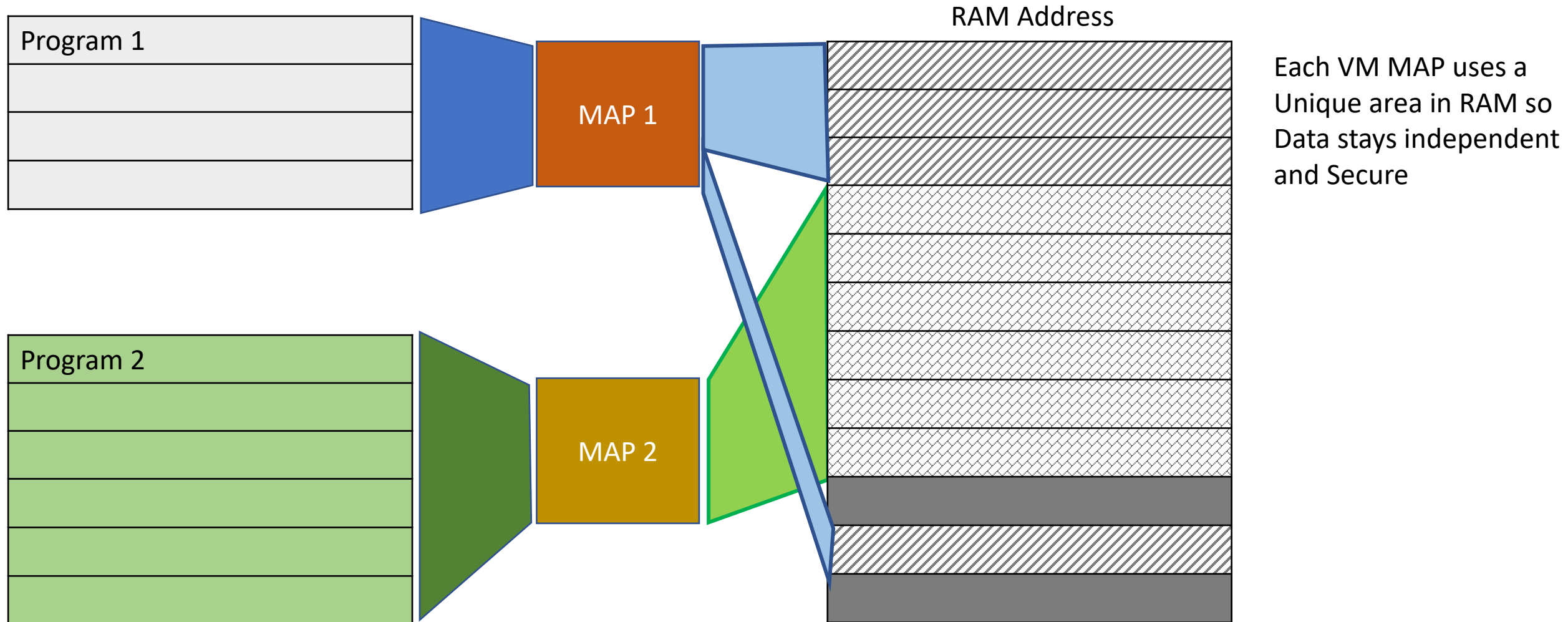


Solving problem of holes in memory and security

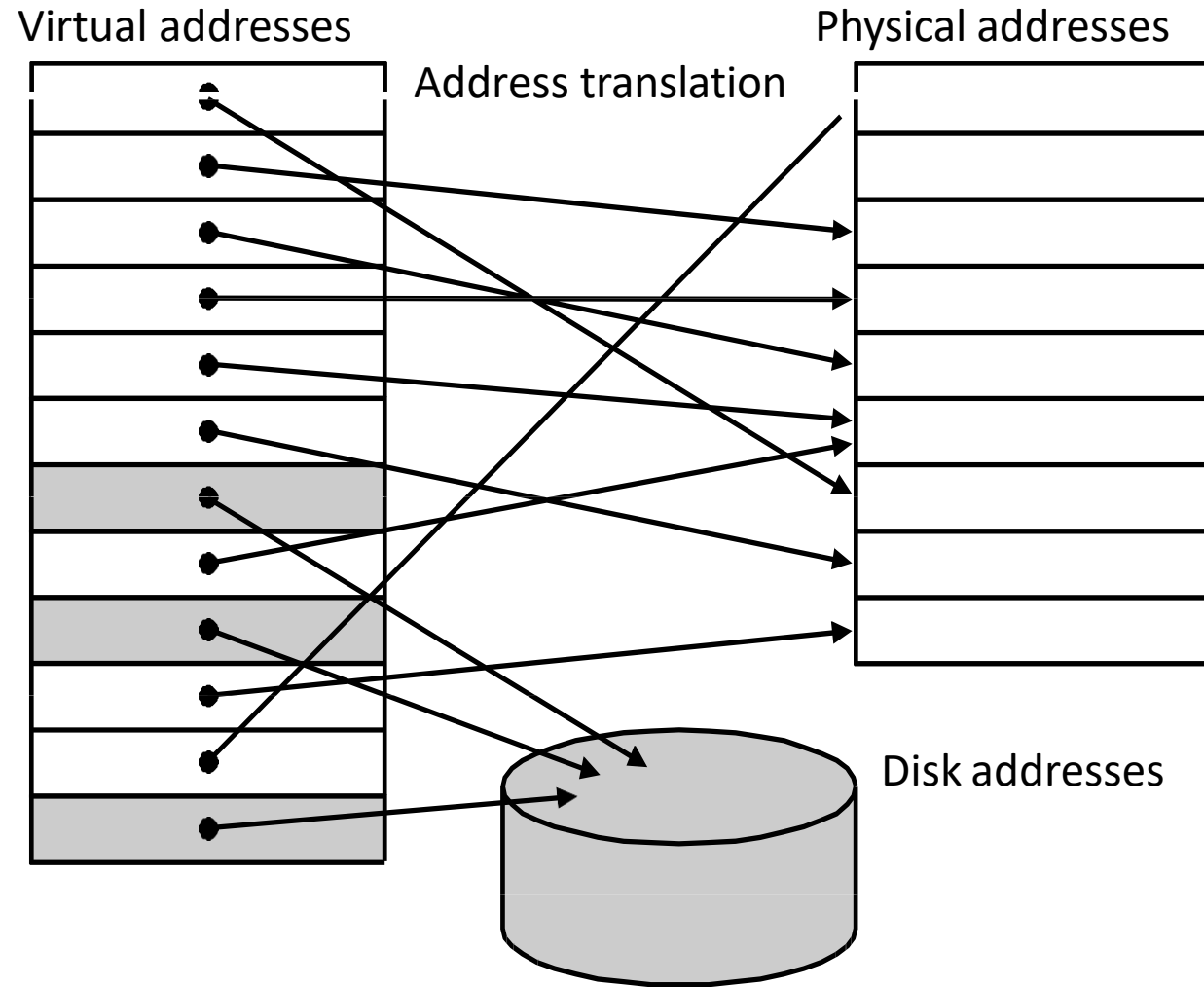


Each Program has its own MAPPING

We can map a program's address to RAM address in any way



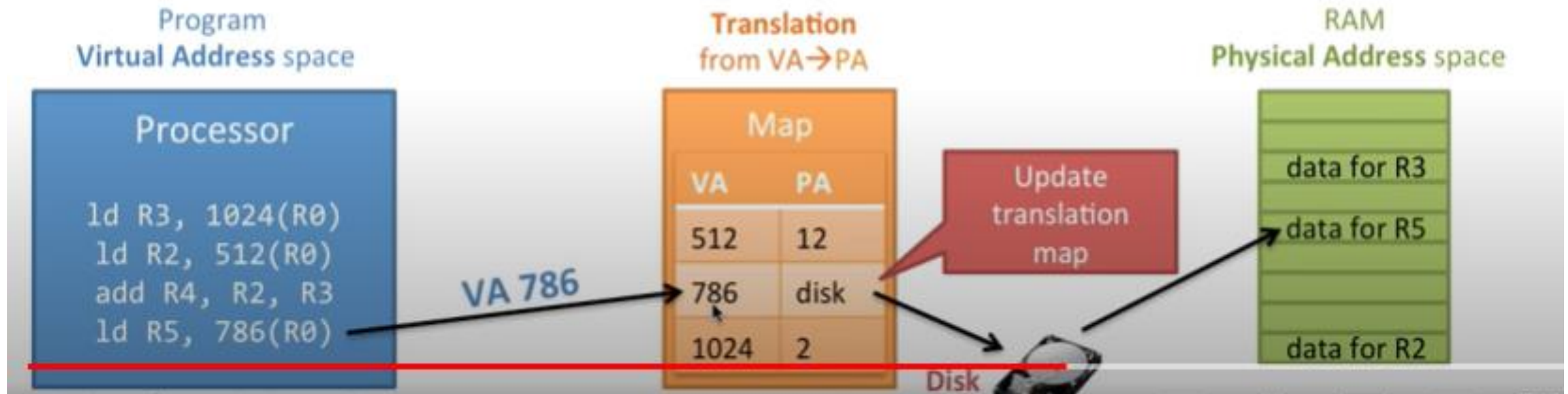
Virtual to Physical mapping



Address Update in Map Table

How does a program access memory?

1. Program executes a load with a **virtual address (VA)**
2. Computer **translates** the address to the **physical address (PA)** in memory
3. (If the **physical address (PA)** is not in memory, the operating system **loads it in from disk**)
4. The computer then **reads the RAM** using the **physical address (PA)** and returns the data to the program



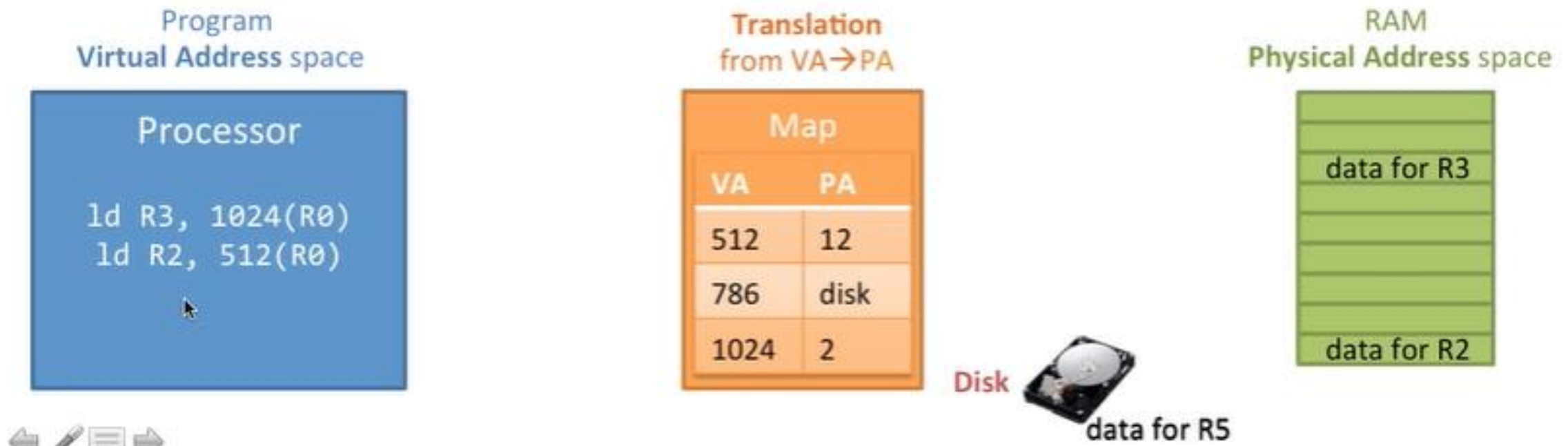
Virtual Memory Concept

- Use main memory as a “cache” for secondary (disk) storage
 - Managed jointly by CPU hardware and the operating system (OS)
- Programs share main memory
 - Each gets a private virtual address space holding its frequently used code and data
 - Protected from other programs
- CPU and OS translate virtual addresses to physical addresses
 - **VM “block” is called a page**
 - **VM translation “miss” is called a page fault**

Making VM Work through Translation

How does a program access memory?

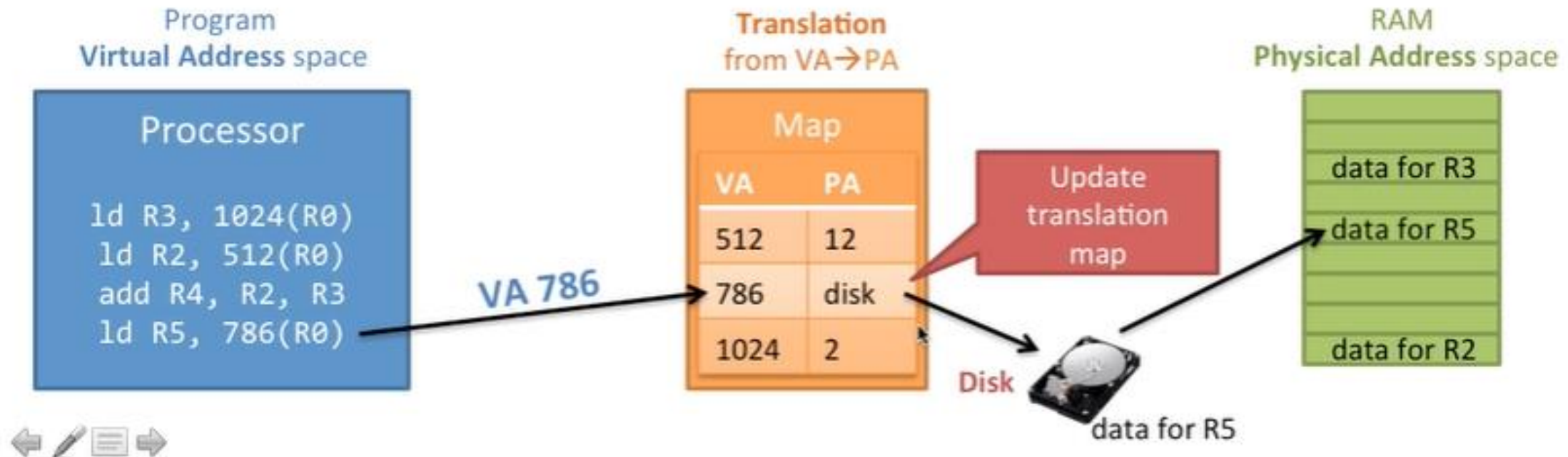
1. Program executes a load with a **virtual address (VA)**
2. Computer **translates** the address to the **physical address (PA)** in memory
3. (If the **physical address (PA)** is not in memory, the operating system **loads it in from disk**)
4. The computer then **reads the RAM** using the **physical address (PA)** and returns the data to the program



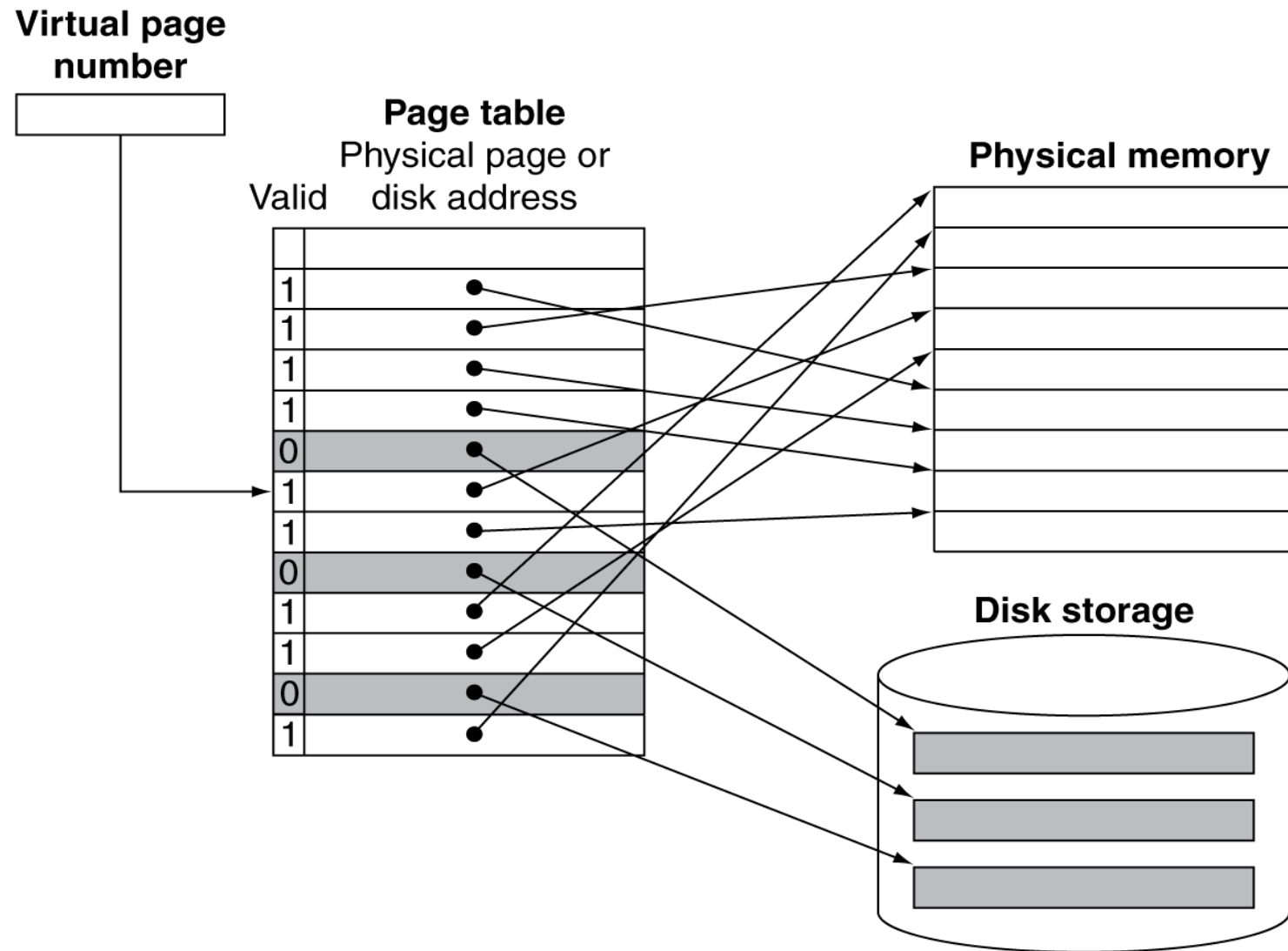
VM Translation and Disk

How does a program access memory?

1. Program executes a load with a **virtual address (VA)**
2. Computer **translates** the address to the **physical address (PA)** in memory
3. (If the **physical address (PA)** is not in memory, the operating system **loads it in from disk**)
4. The computer then **reads the RAM** using the **physical address (PA)** and returns the data to the program



Mapping Pages to Storage



Page Table Size Management

We need to translate every possible address in 32-bit Virtual Address Space

This is 1 Giga **Word** (32 bit) Locations or 4 Giga **Byte** (8 bits) locations

We need extra 1 Giga locations to store Page Tables !!!

We divide memory into PAGES instead of WORDS

Fine Grain – VA to PA MAP

Maps each VA address location to PA

Fine
Grain

VA	PA
512	12
786	3
1024	2

Coarse Grain – VA to PA MAP

Maps each 4KB chunk 'page' to PA

Coarse
Grain

VA	PA
0 - 4095	4096 - 8191

Use Page Address Instead of Word Address

Coarse Grained Addressing

- The **Page Table** manages larger chunks (**pages**) of data:
 - Fewer **Page Table Entries** needed to cover the whole address space
 - But, less flexibility in how to use the RAM (have to move a page at a time)
- Today:
 - Typically 4kB pages (1024 words per page)
 - Sometimes 2MB pages (524,288 words per page)

Q: How many entries do we need in our Page Table with 4kB pages on a 32-bit machine?

- ☐ 1 for every word = $2^{30} = 1$ billion
- ☐ 1 for every 1024 words = 1 million
- ☐ 1 for every 4096 words = 262,144

A: 1 for every 1024 words = 1 million
With 4kB pages we have 1024 words per page. That means we need 1 Page Table Entry for every 1024 words. In total we have 1 billion words, so $1 \text{ billion} \div 1024$ is 1 million Page Table Entries. This is *much* more manageable.

Coarse-grain:
maps chunks
of address

Page Table
translates VA \rightarrow PA

Map	
VA	PA
0-4095	4096-8191



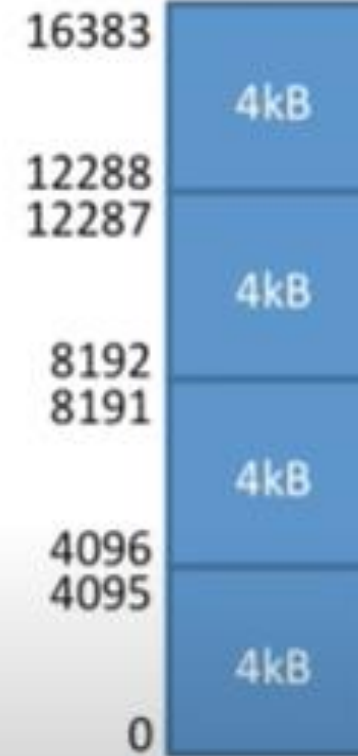
How to Map Address with Pages?

Page Table
translates VA→PA

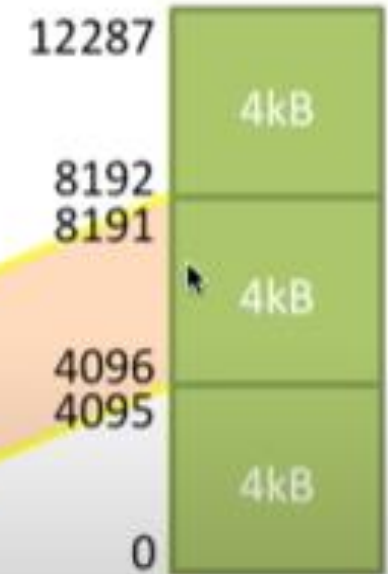
Coarse-grain:
maps chunks
of address

Map	
VA	PA
0-4095	4096-8191

**Virtual Address
Space**

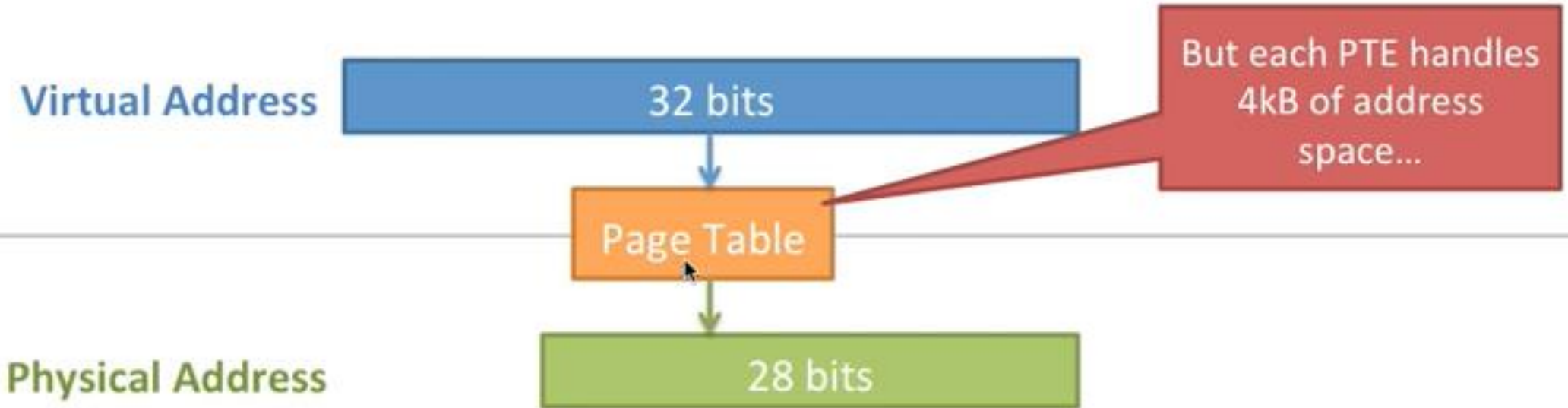


**Physical Address
Space**

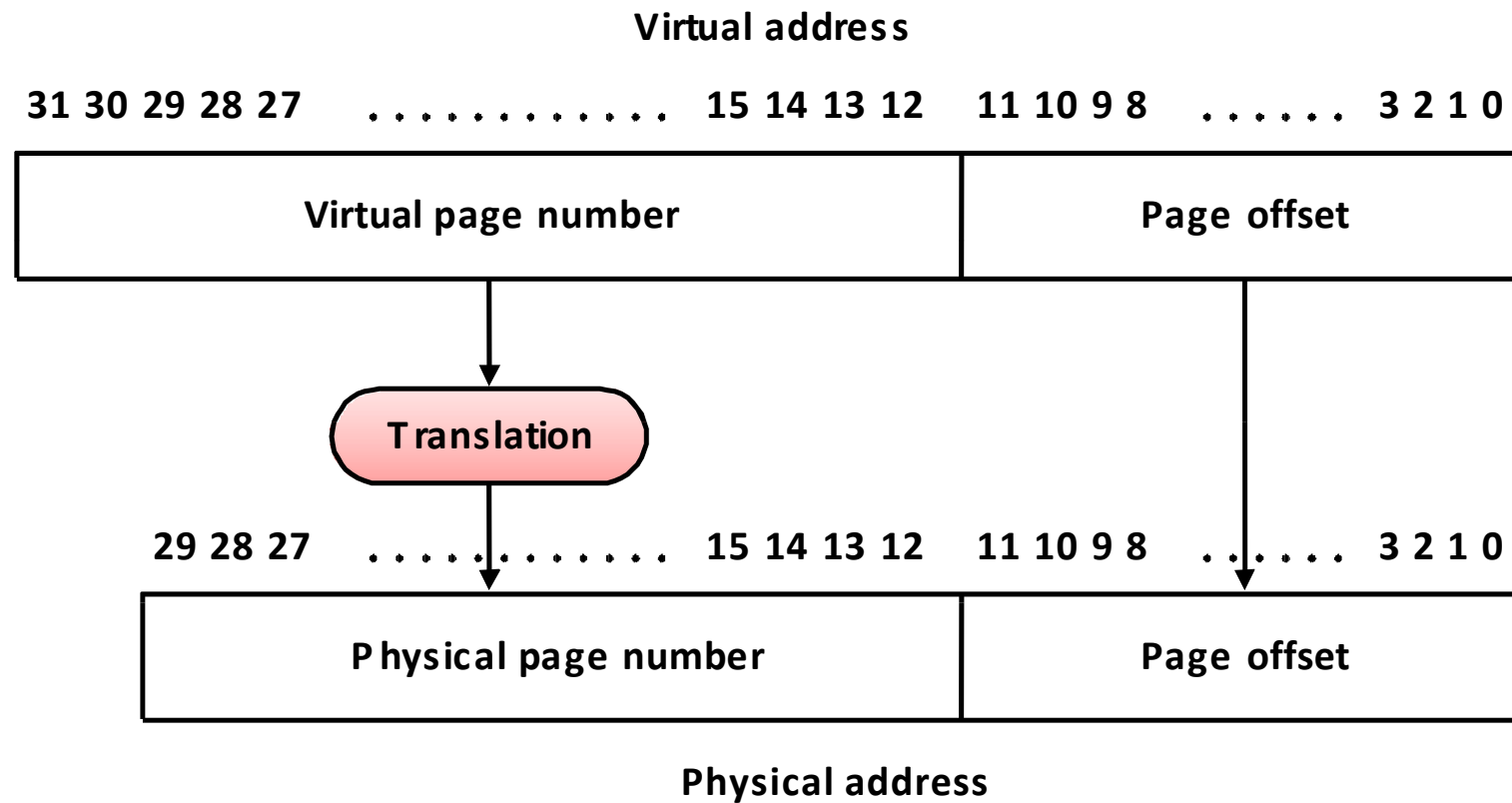


Address Translation - Concept

- What happens on a 32-bit machine with 256MB of RAM and 4kB pages?
 - 32-bit Virtual Addresses
 - 28-bit Physical Addresses



Address translation



Where is page table stored?



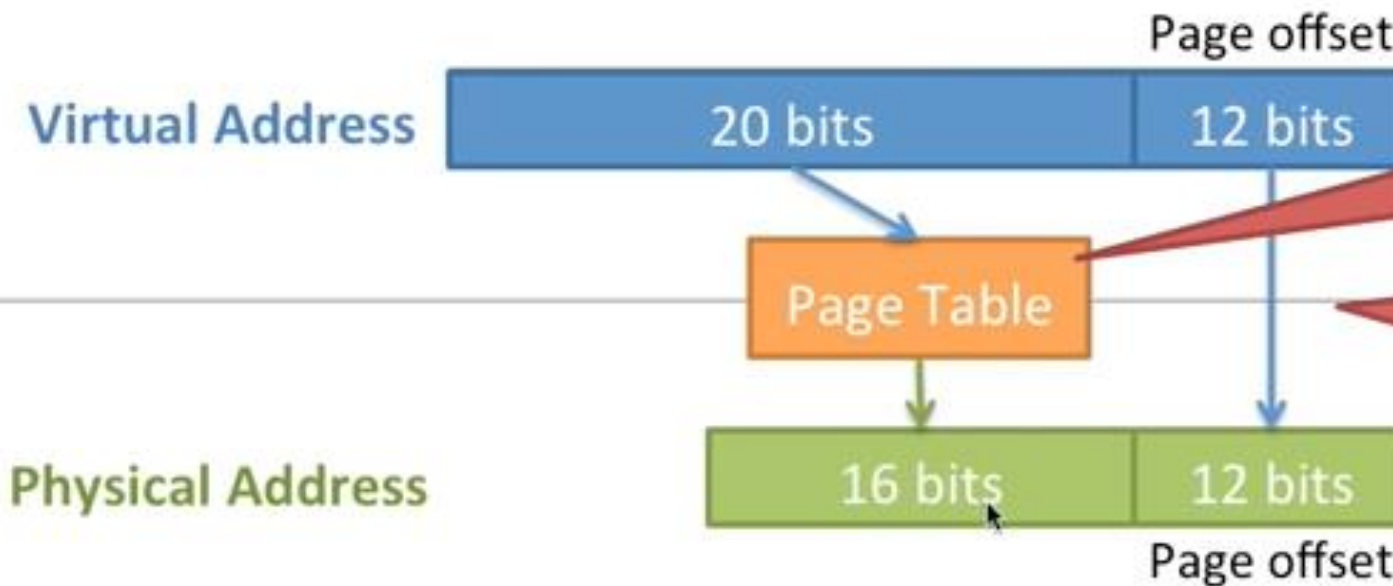
How big is Page Table?

- Suppose virtual address = 32 bits, page size = 4 KB, page table entry = 4 B
- then number of page table entries = number of pages
 $= 2^{32} / 2^{12} = 2^{20}$
- page table size = $2^{20} \times 2^2 = 4 \text{ MB}$
- hundreds / thousands of processes

Store in dedicated memory? main memory?

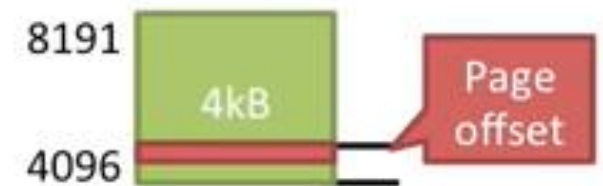
Address Translation in Page Tables

- What happens on a 32-bit machine with 256MB of RAM and 4kB pages?
 - 32-bit Virtual Addresses
 - 28-bit Physical Addresses



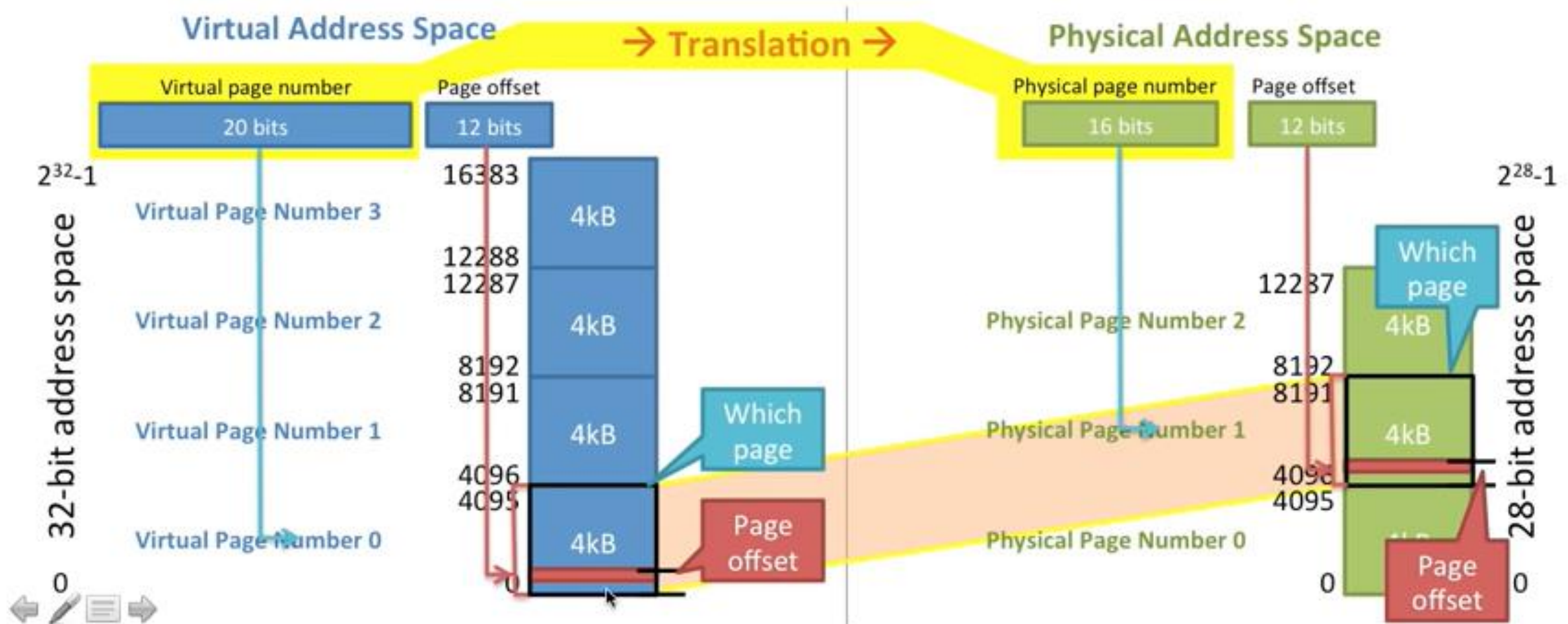
But each PTE handles 4kB of address space...

For every page we have 4096 addresses (12 bits) that don't get translated

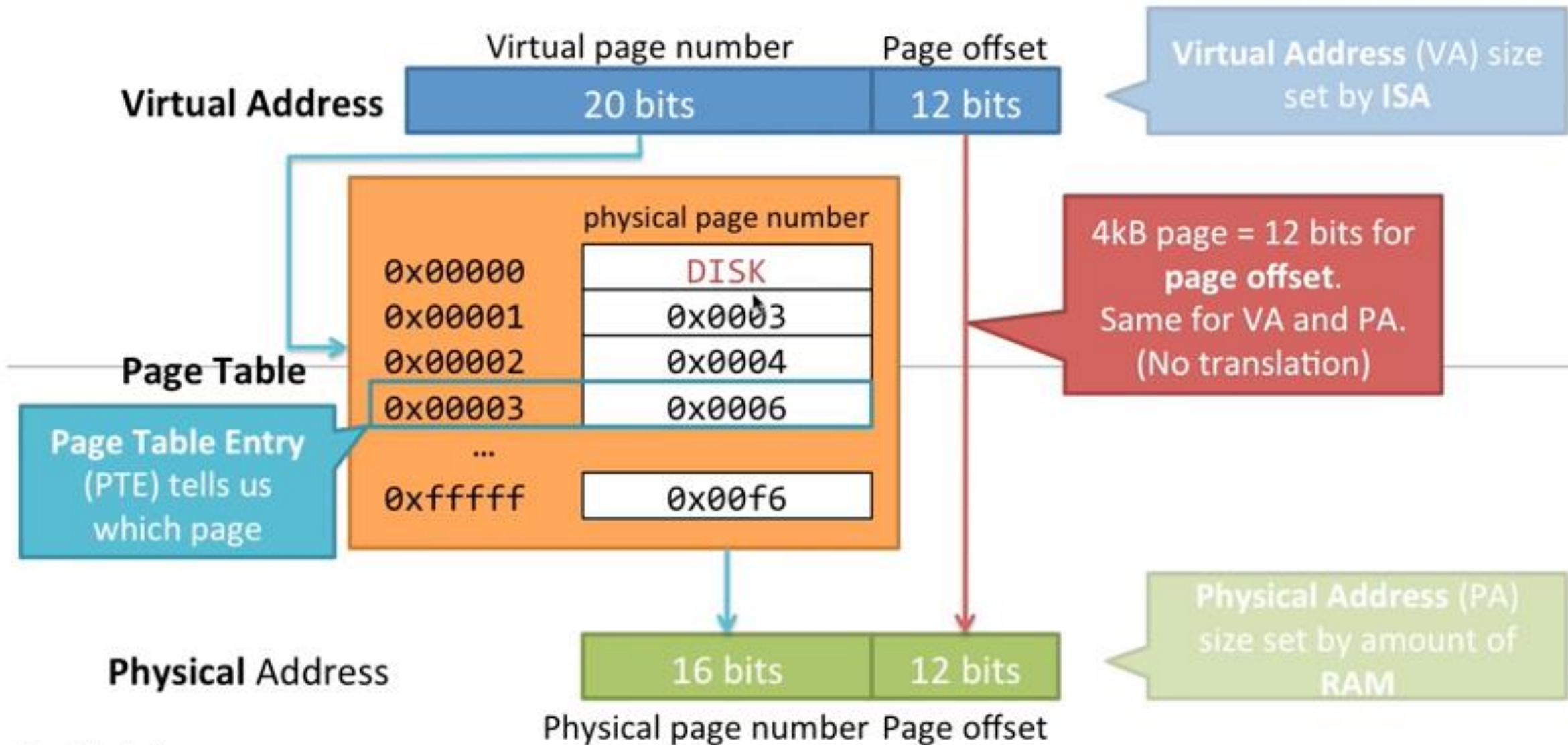


Pages, Offset and Translation

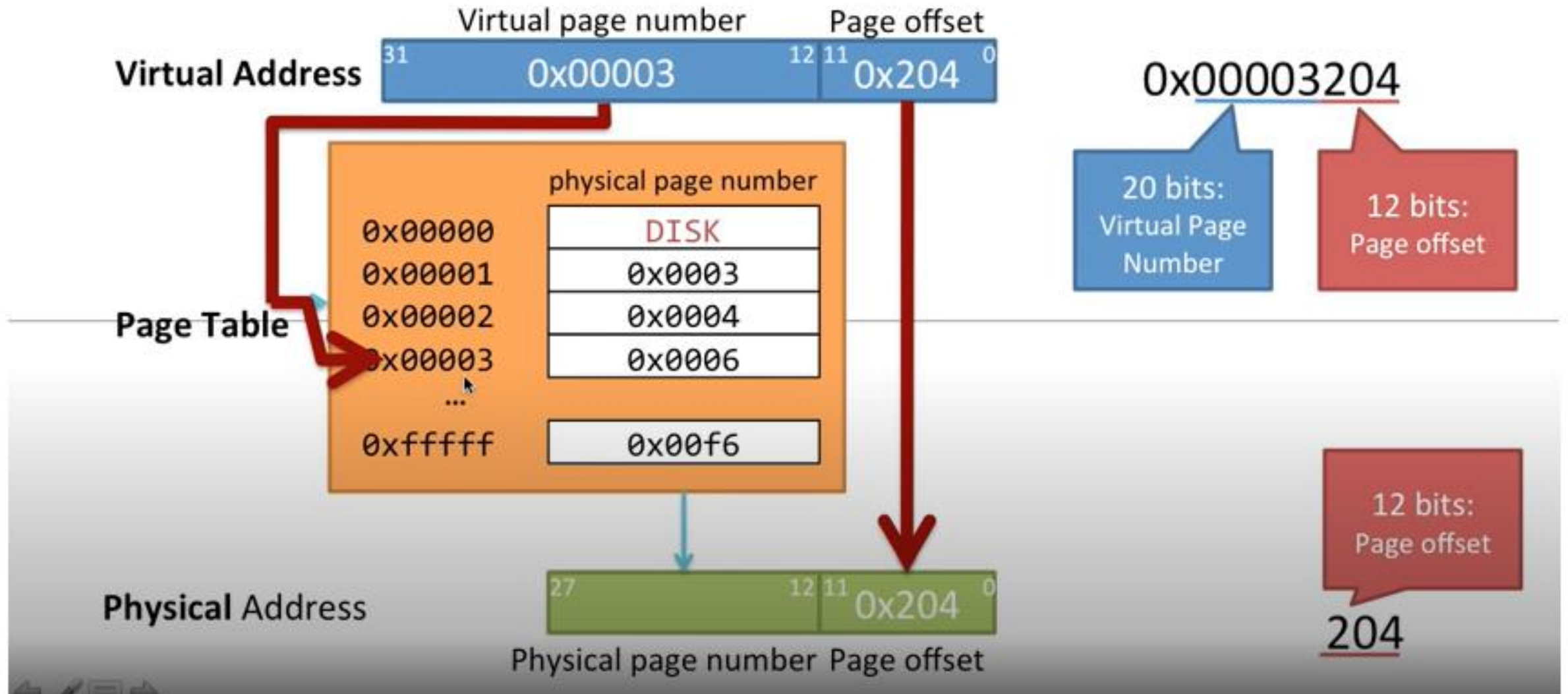
- What happens on a 32-bit machine with 256MB of RAM and 4kB pages?



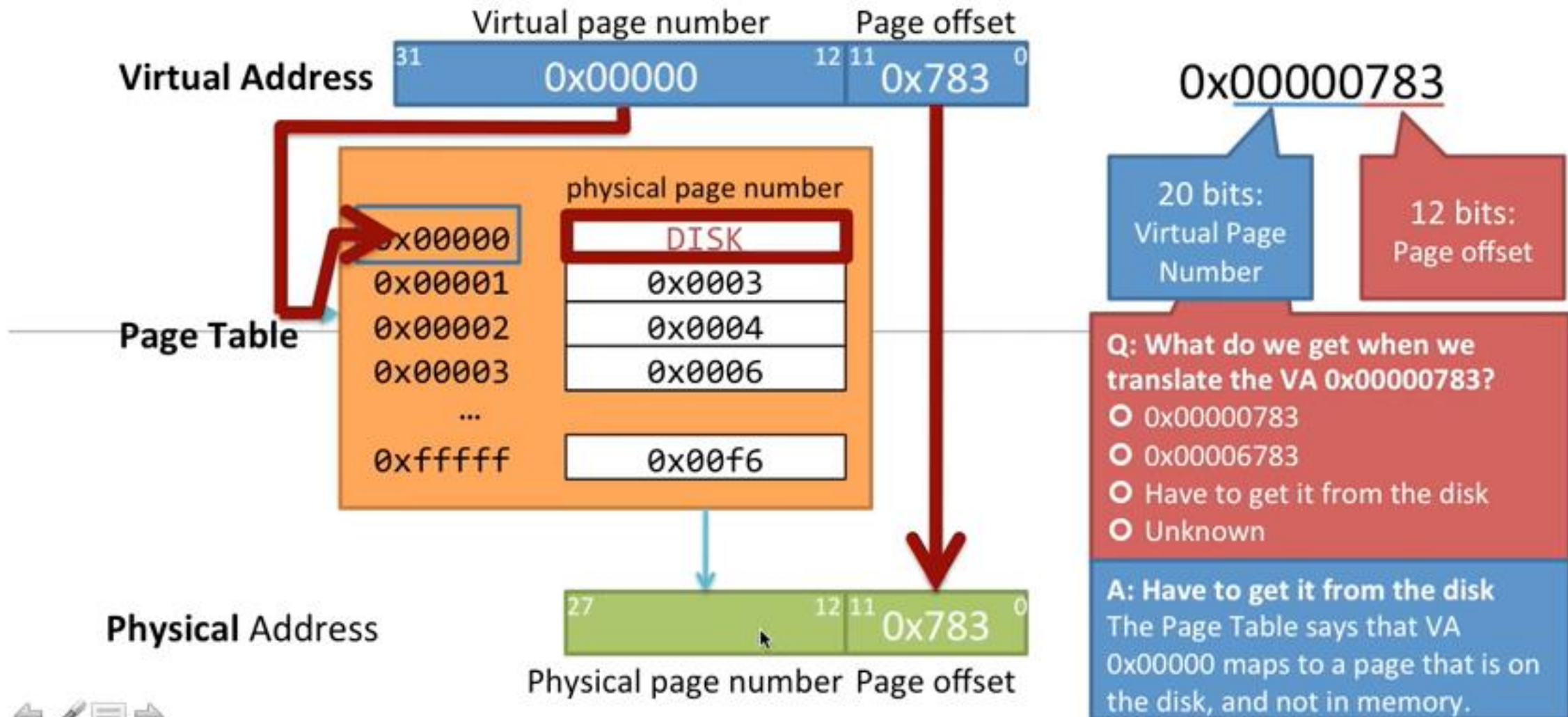
How to do a Page Table Lookup



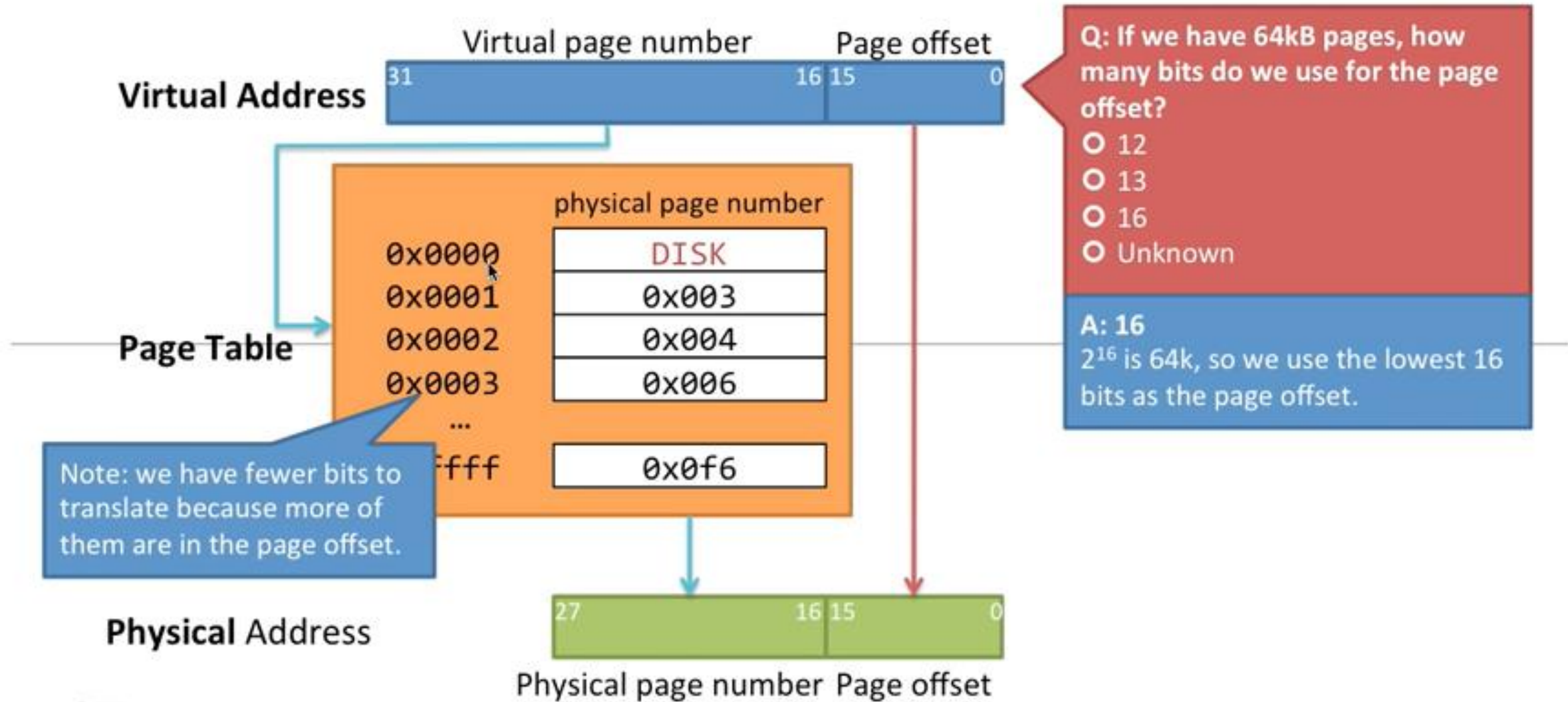
Example Translation (1)



Example Translation (2)



Example Translation for 64KB Pages



What happens if a Page is not in RAM - Fault

- **Page Table Entry** says the page is on **disk**
- Hardware (CPU) generates a **page fault exception**
- The hardware jumps to the OS page fault handler to clean up
 - The OS chooses a page to evict from **RAM** and write to **disk**
 - If the page is **dirty**, it needs to be written back to disk first
 - The OS then reads the page from disk and puts it in **RAM**
 - The OS then changes the **Page Table** to map the new page
- The OS jumps back to the instruction that caused the page fault.
 - (This time it won't cause a page fault since the page has been loaded.)

"Dirty" means the data has been changed (written). If the page has not been written since it was loaded from disk, then it doesn't have to be written back.

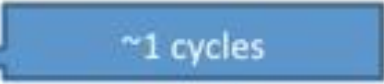






Q: How long does this take?

- ☐ No time
- ☐ A short time
- ☐ A long time
- ☐ An amazingly, incredibly, painfully long time

A: An amazingly, incredibly, painfully long time

Disks are *much* slower than RAM, so every time you have a page fault it takes an amazingly, incredibly, painfully long time.

How long does a Page Fault Take?

- **Page Table Entry** says the page is on **disk** 
- Hardware (CPU) generates a **page fault exception** 
- The hardware jumps to the OS page fault handler to clean up 
 - The OS chooses a page to evict from **RAM** and write to **disk** 
 - If the page is **dirty**, it needs to be written back to disk first 
 - The OS then reads the page from disk and puts it in **RAM** 
 - The OS then changes the **Page Table** to map the new page
- The OS jumps back to the instruction that caused the page fault.
 - (This time it won't cause a page fault since the page has been loaded.) 

In the time it takes to do handle one page fault
you could execute 80 million cycles on a modern CPU.

Page faults are the **SLOWEST** possible thing that can happen to a computer
(except for human interaction).

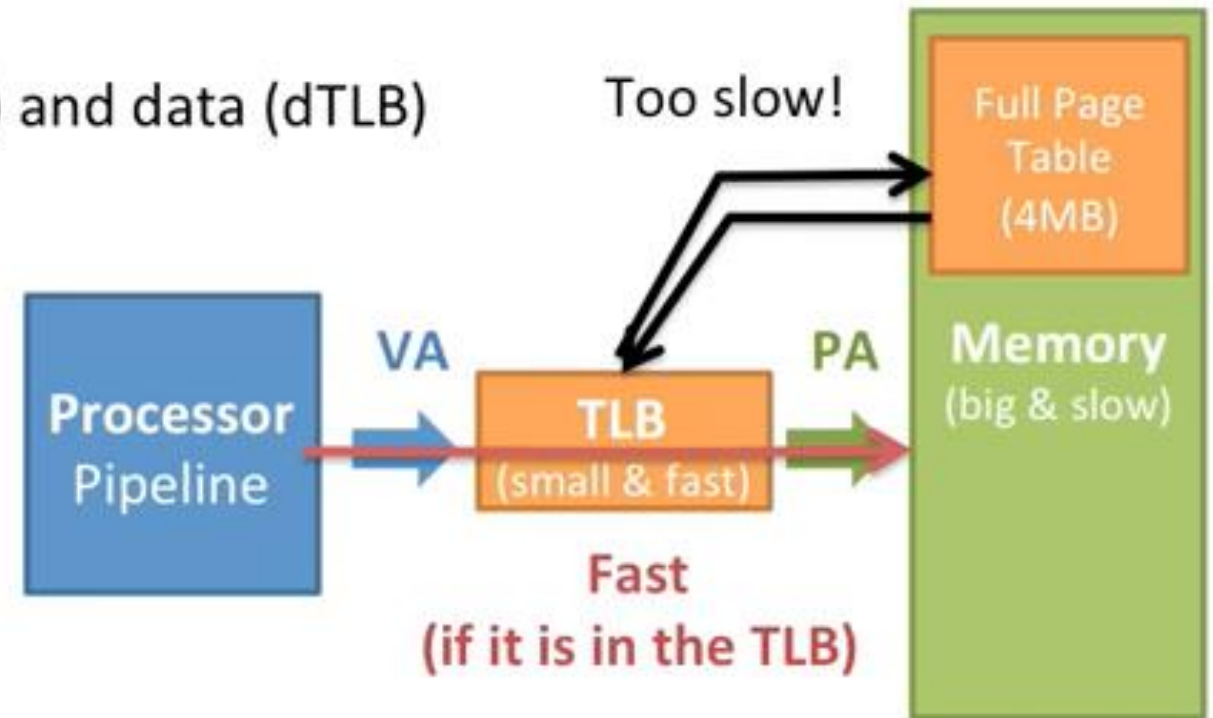
Virtual Memory in Practice

- VM is great:
 - Unlimited programs/memory, protection, flexibility, etc.
- But it comes at a high cost:
 - *Every* memory operation has to look up in the page table
 - Need to access **1) the page table** and **2) the memory address** (2x memory accesses)
(Remember, 1.33 memory accesses per instruction. This is going to hurt.)
- How can we make a page table look up *really really* fast?
 - Software would be far too slow
(e.g., an extra 5 instructions for every memory access would kill performance)
- Perhaps a hardware page table **cache**?

Making VM Fast through the TLB

- To make VM fast we add a special **Page Table cache**: the **Translation Lookaside Buffer (TLB)**
 - Fast: less than 1 cycle (have to do it for every memory access)
 - Very similar to a cache
- To be fast, TLBs must be small:
 - Separate TLBs for instructions (iTLB) and data (dTLB)
 - 64 entries, 4-way (4kB pages)
 - 32 entries, 4-way (2MB pages)
(Page Table is 1M entries)

Lots of locality!
Miss rates are typically
only a few percent.



What can happen when we access memory?

Good: Page in RAM

- **PTE in the TLB**
 - **Excellent**
 - <1 cycle to translate, then go to RAM (or cache)
- **PTE not in the TLB**
 - **Poor**
 - 20-1000 cycles to load PTE from RAM, then go to RAM

With 1.33 memory accesses per instruction we can't afford 20-1000 cycles very often.

Bad: Page not in RAM

- **PTE in the TLB** (unlikely)
 - **Horrible**
 - 1 cycle to know it's on disk
 - ~80M cycles to get it from disk
- **PTE not in the TLB**
 - **(ever so slightly more) horrible**
 - ~80M cycles to get it from disk



MULTI PROCESSORS AND MULTI CORES

Superscalar and Instruction Level Parallelism

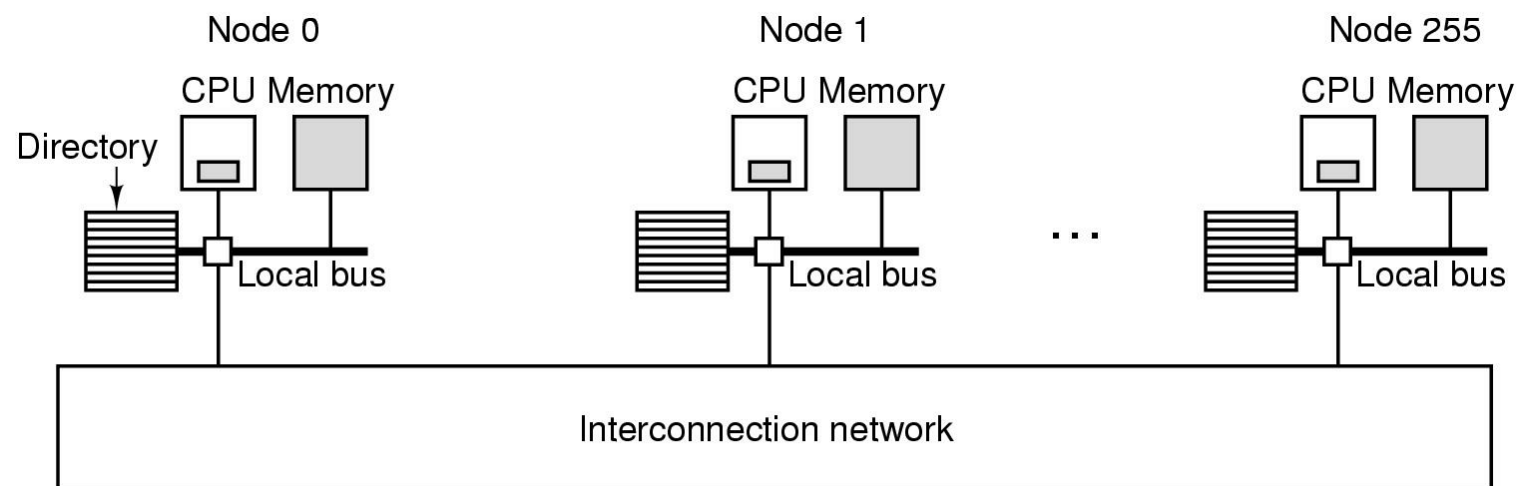
- In Superscalar architectures, multiple Instruction Fetch in Pipeline
- and Multiple Integer and Floating-Point units process multiple pipelines simultaneously.

NUMA Architectures

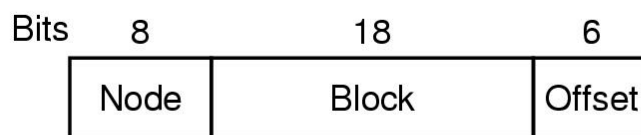
- Non-Uniform Memory Architectures – use Message Passing or Shared Disk for Synchronization

Modern Computer Architectures

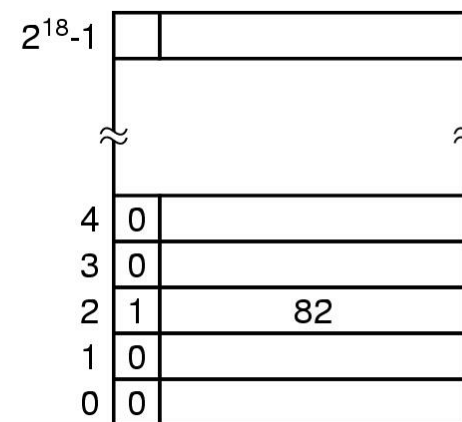
- DSA – **Domain Specific Architectures**, Example is Google Tensor Processor, Tensor Flow Software customized language for ML, AI
- **Reconfigurable Architectures** – Based on run-time hardware configuration using FPGA (Field Programmable Logic Arrays)
- **Hybrid Architectures** – Combination of multiple type of Processor Cores, DSA and Reconf. All combined into one Chip.



(a)



(b)



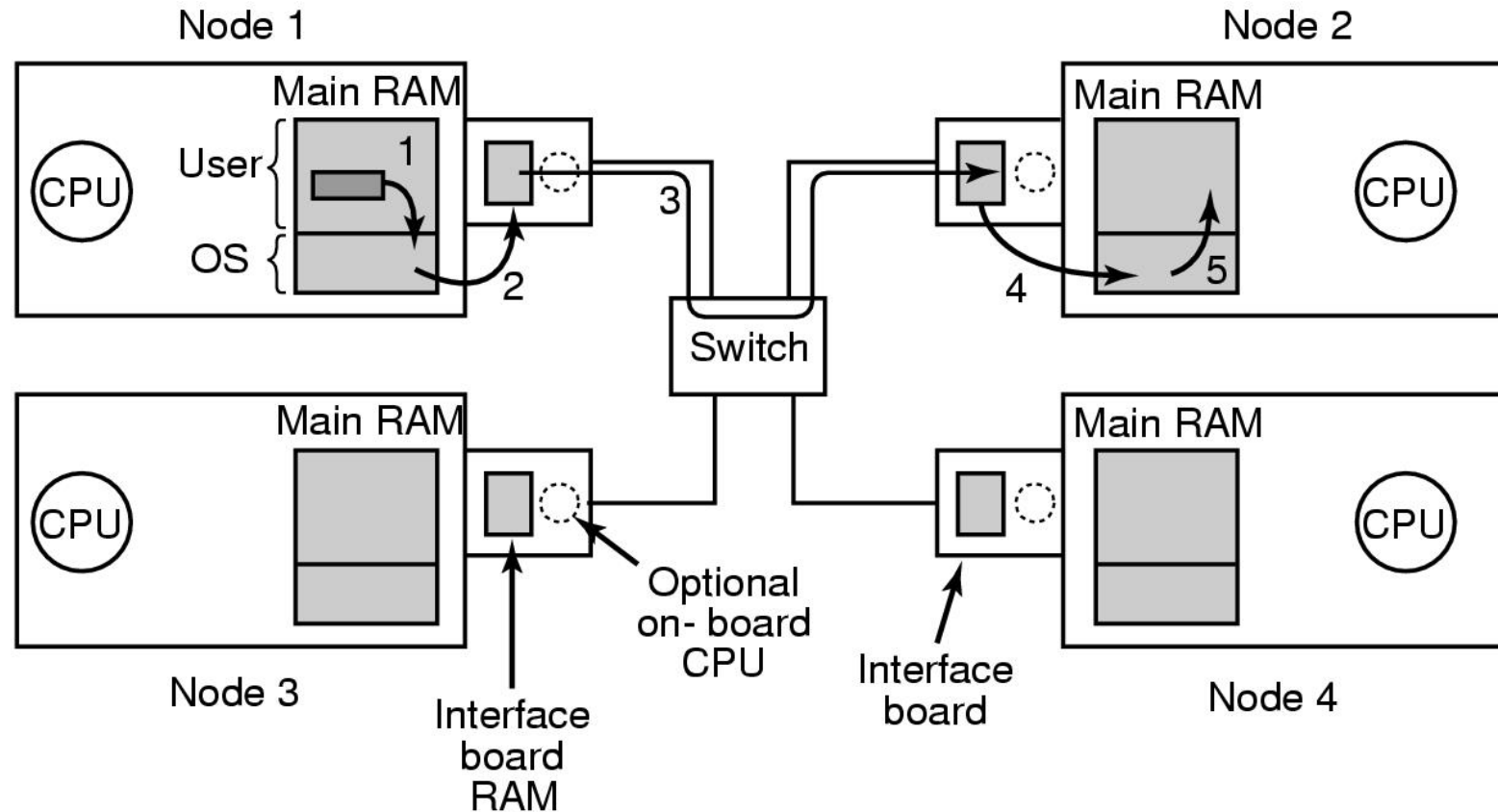
(c)

(a) 256-node directory-based multiprocessor

(b) Fields of 32-bit memory address

(c) Directory at node 36

Multicomputer Hardware (Cluster Computing)



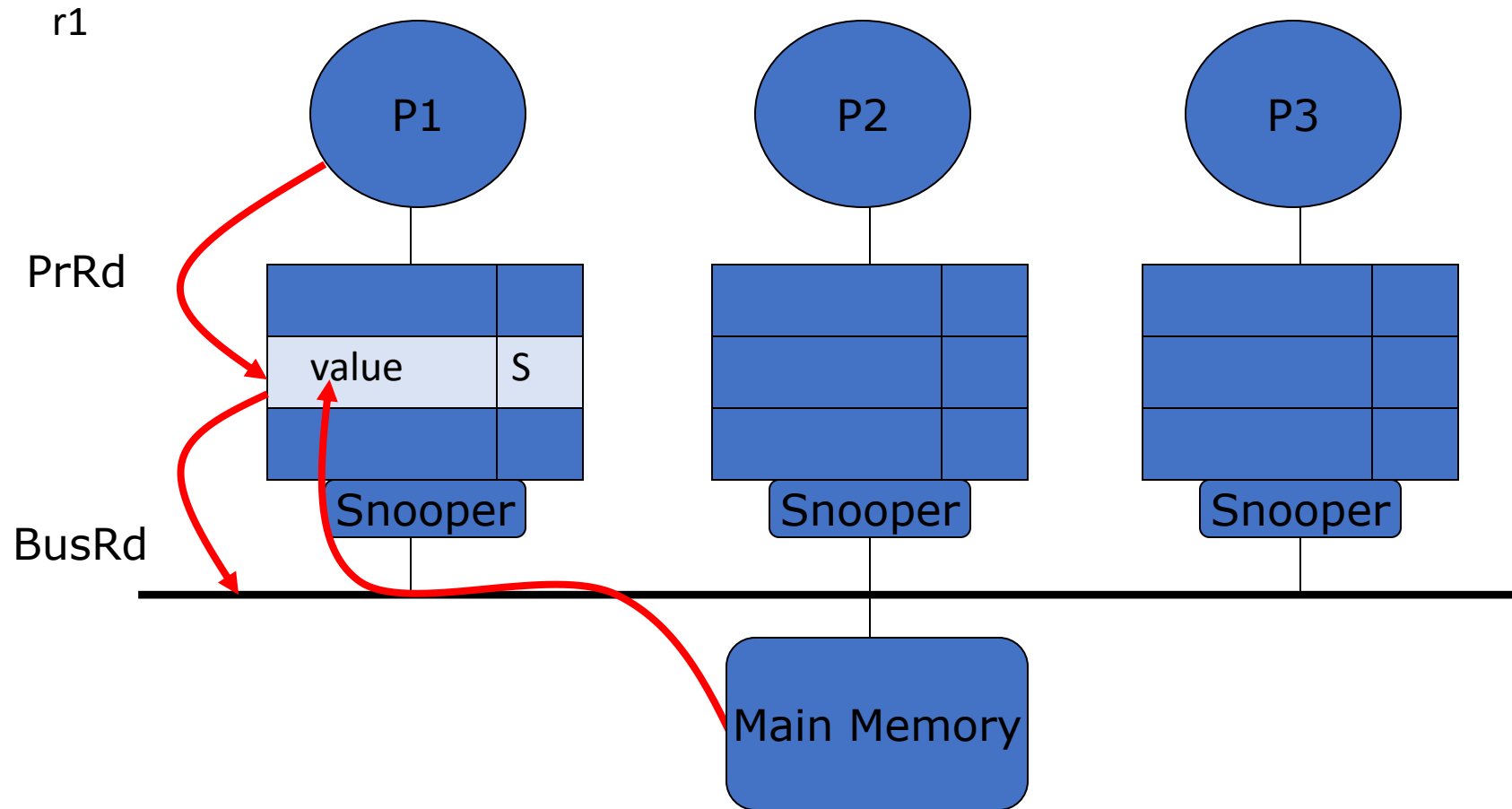
Network interface boards in a multicomputer

Multi Core Synchronization Through Cache Coherence Protocols

SNOOPY BUS BASED CACHE COHERENCE PROTOCOLS

- There are three processors.
- Each is reading/writing the same value from memory where r1 means a read by processor 1 and w3 means a write by processor 3.
- For simplicity sake, the memory location will be referred to as “value.”
- The memory access stream is:
r1, r2, w3, r2, w1, w2, r3, r2, r1

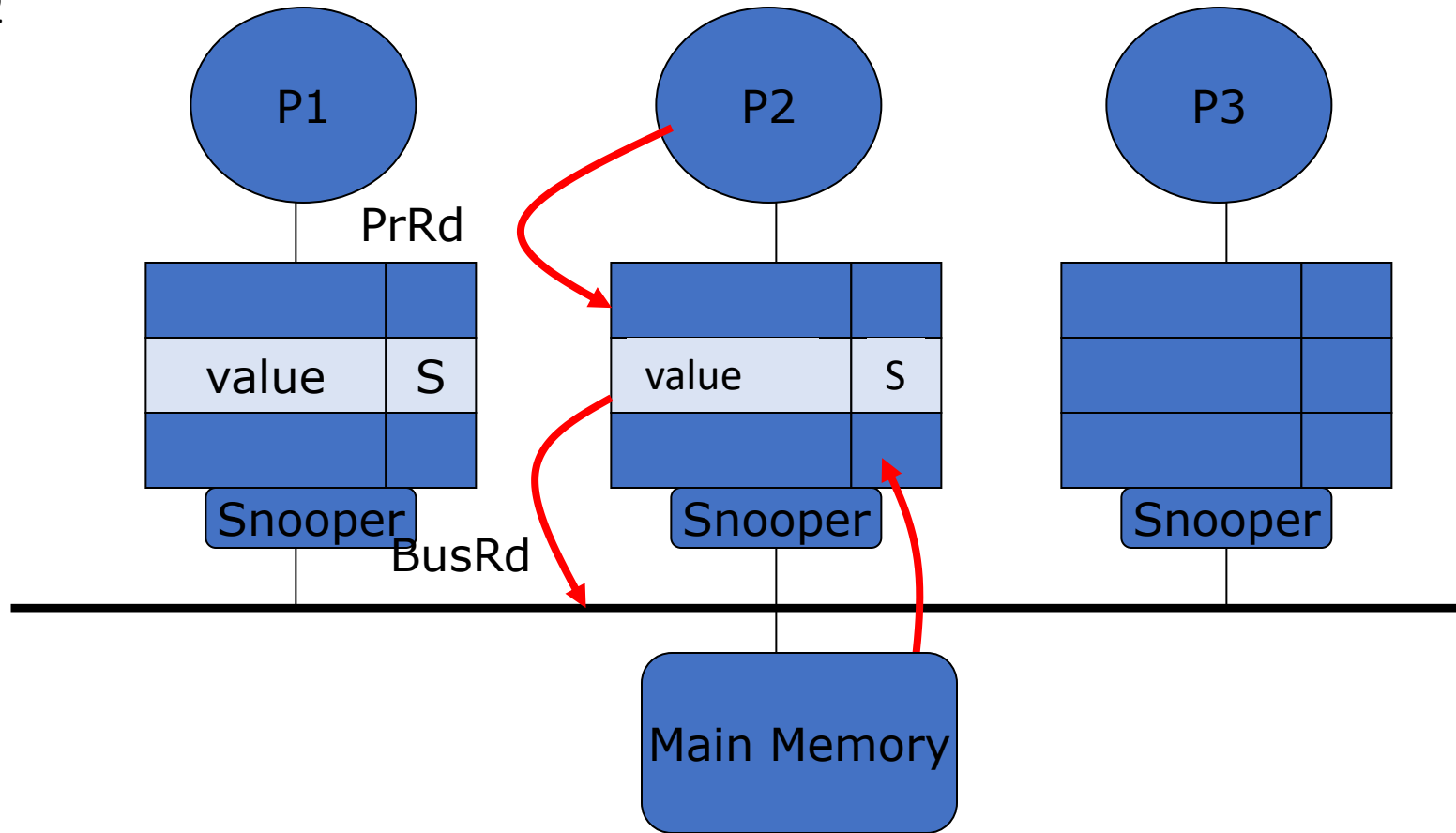
Read in 1



P1 wants to read the value. The cache does not have it and generates a BusRd for the data. Main memory controller provides the data. The data goes into the cache in the shared state.

r2

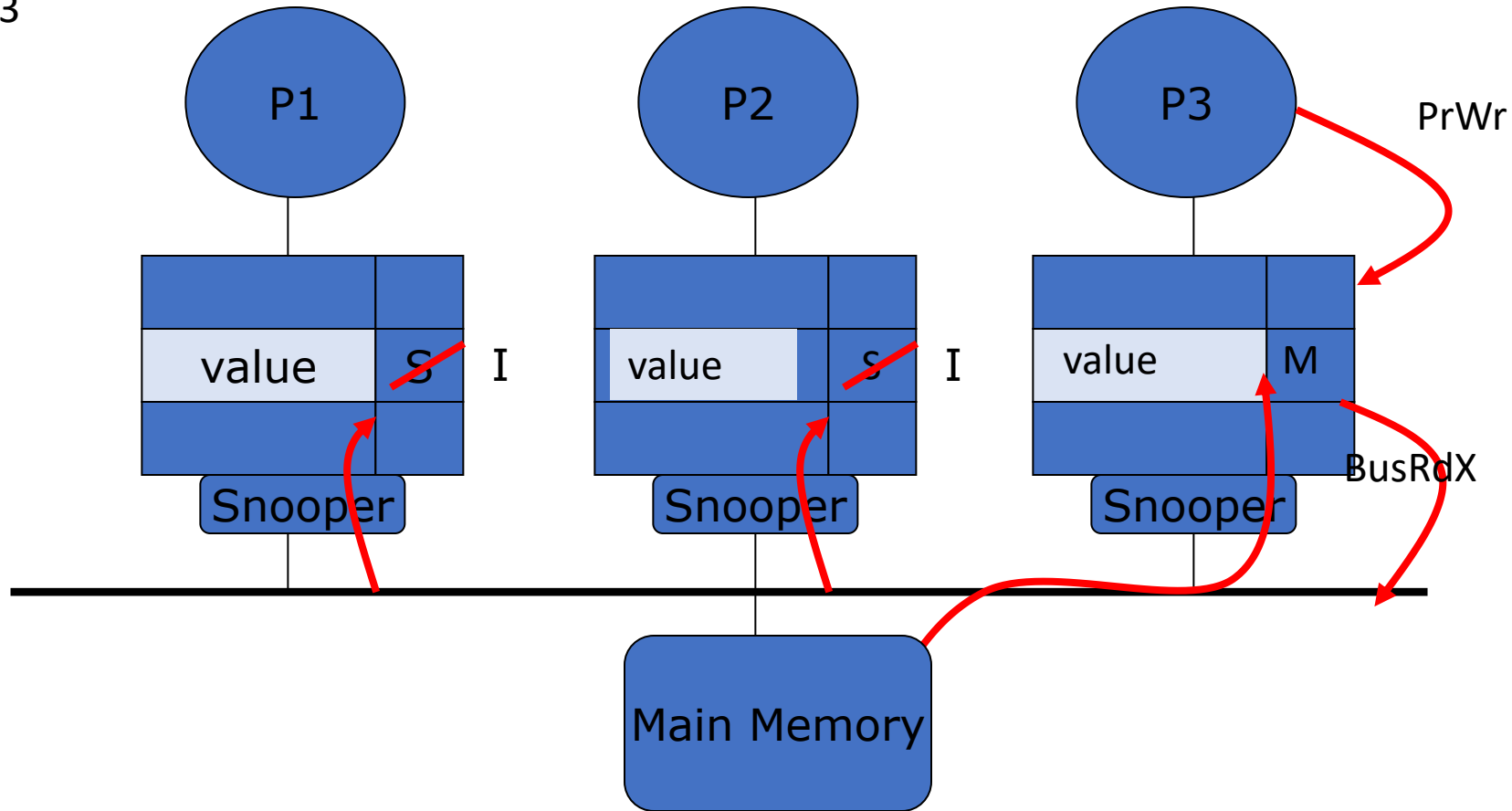
Read in 2



P2 wants to read the value. Its cache does not have the data, so it places a BusRd to notify other processors and ask for the data. The main memory controller provides the data.

w3

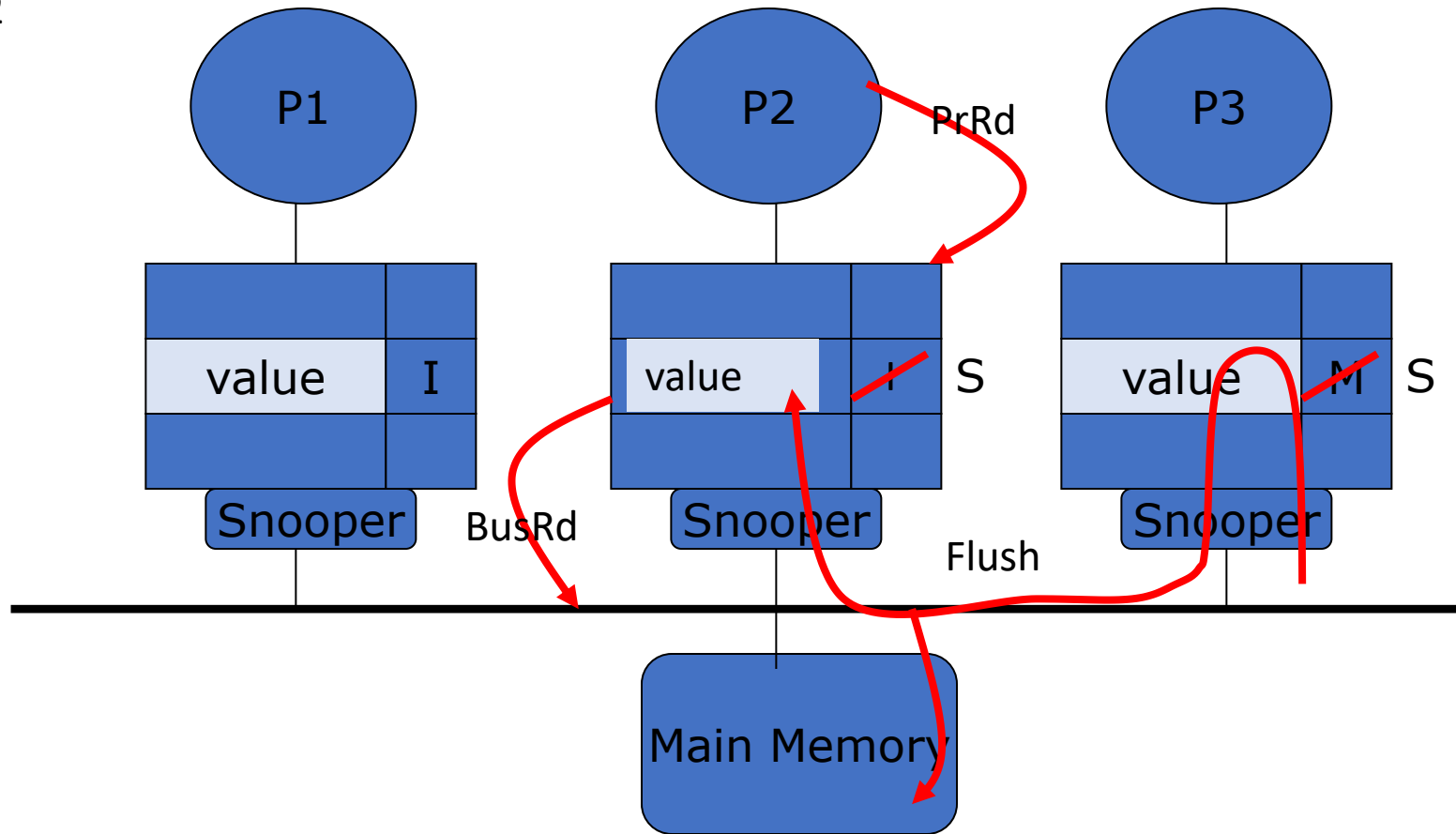
Write in 3



P3 wants to write the value. It places a BusRdX to get exclusive access and the most recent copy of the data. The caches of P1 and P2 see the BusRdX and invalidate their copies. Because the value is still up-to-date in main memory, memory provides the data.

r2

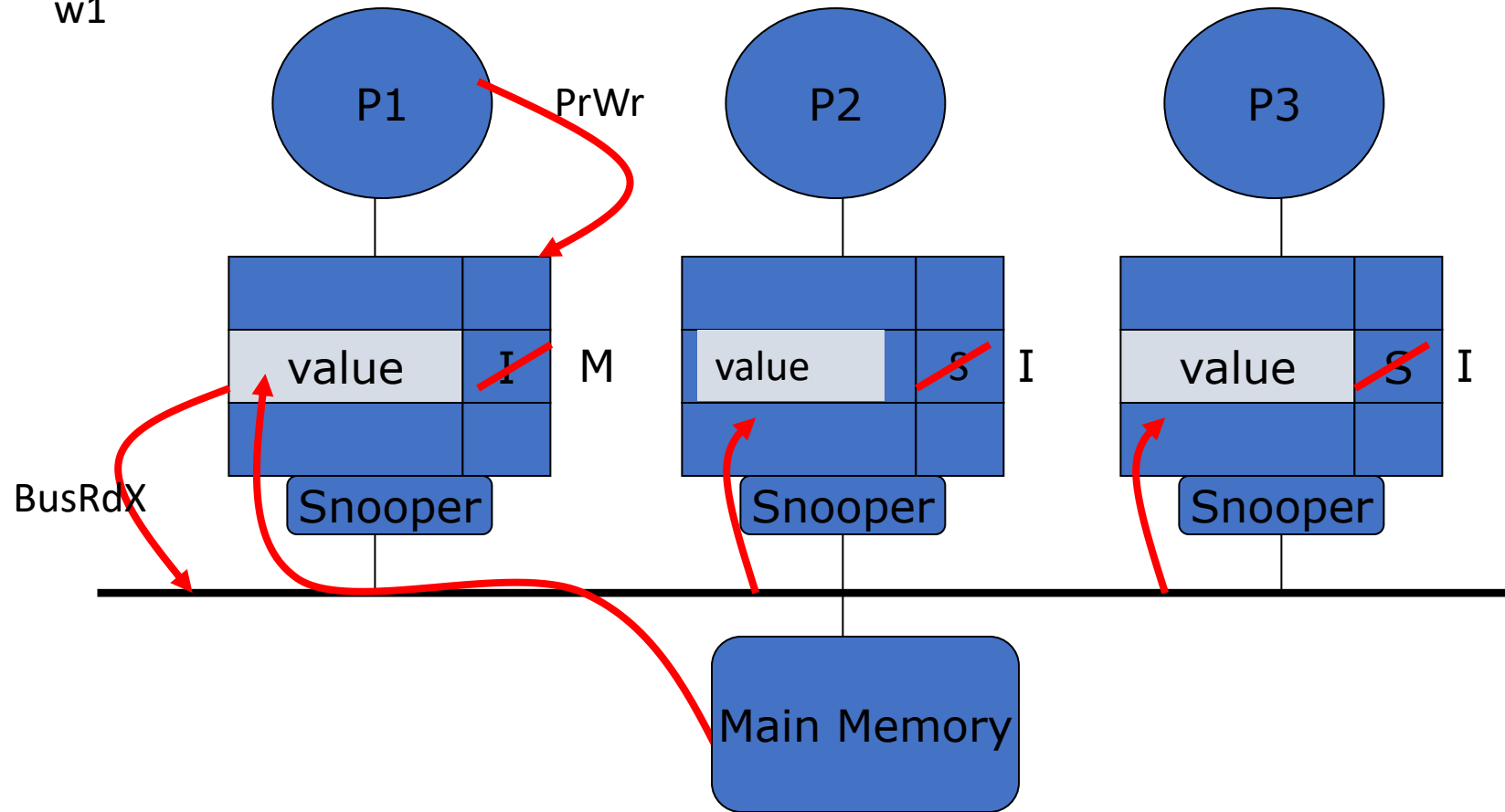
Read in 2



P2 wants to read the value. P3's cache has the most up-to-date copy and will provide it. P2's cache puts a BusRd on the bus. P3's cache snoops this and cancels the main memory access because it will provide the data. P3's cache flushes the data to the bus.

w1

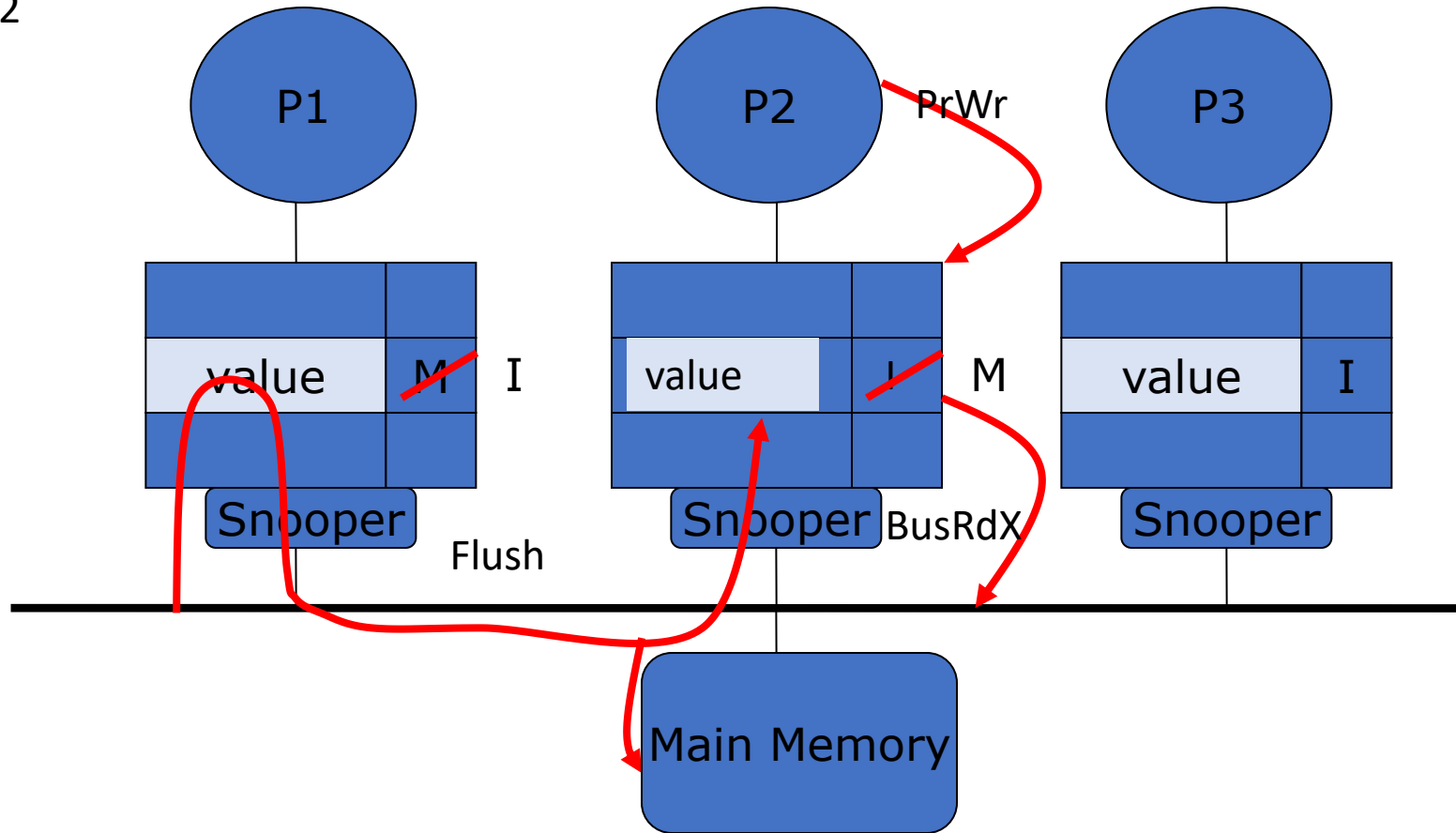
Write in 1



P1 wants to write to its cache. The cache places a BusRdX on the bus to gain exclusive access and the most up-to-date value. Main memory is not stale so it provides the data. The snoopers for P2 and P3 see the BusRdX and invalidate their copies in cache.

w2

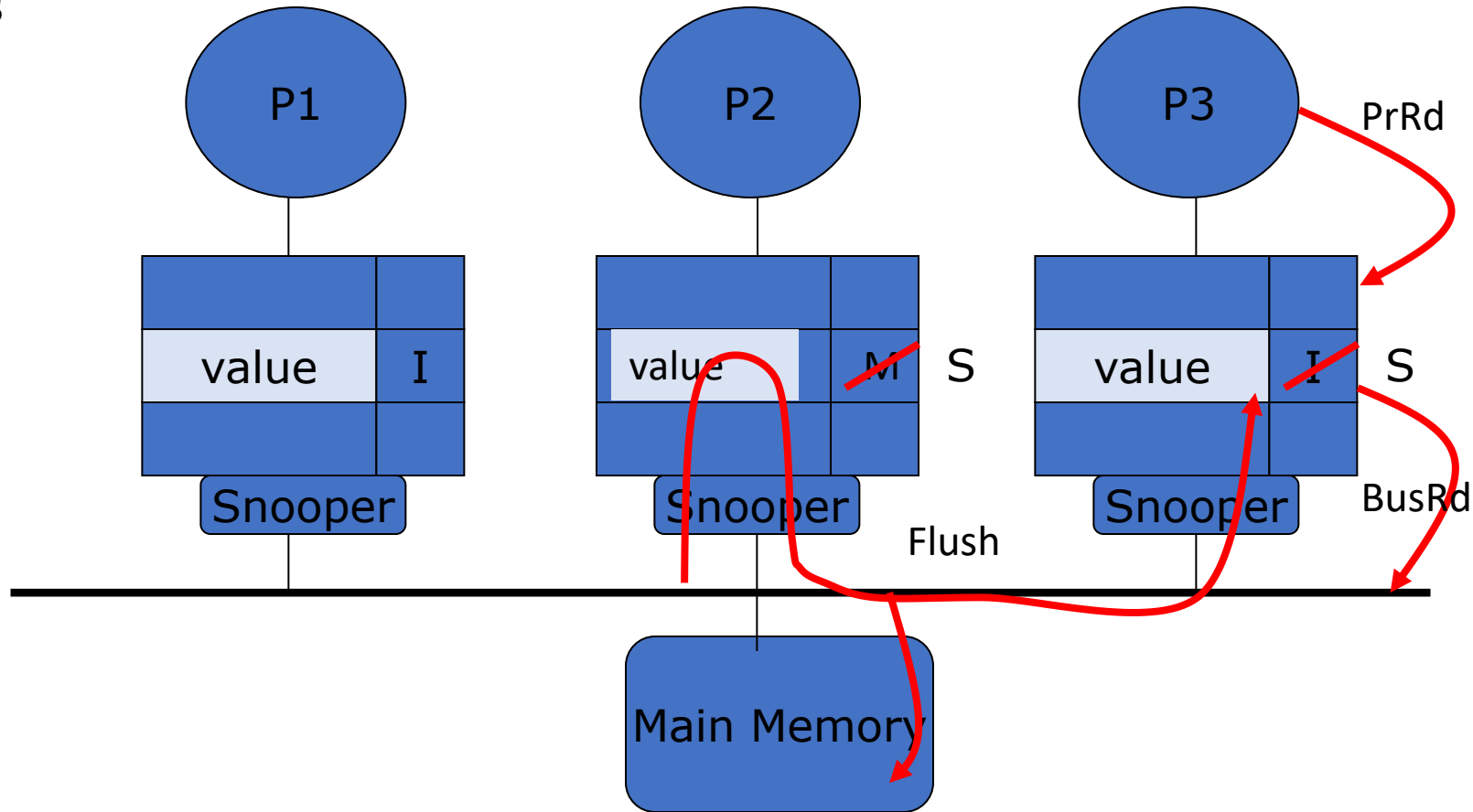
Write in 2



P2 wants to write the value. Its cache places a BusRdX to get exclusive access and the most recent copy of the data. P1's snoopers see the BusRdX and flushes the data to the bus. Also, it invalidates the data in its cache and cancels the main memory access.

r3

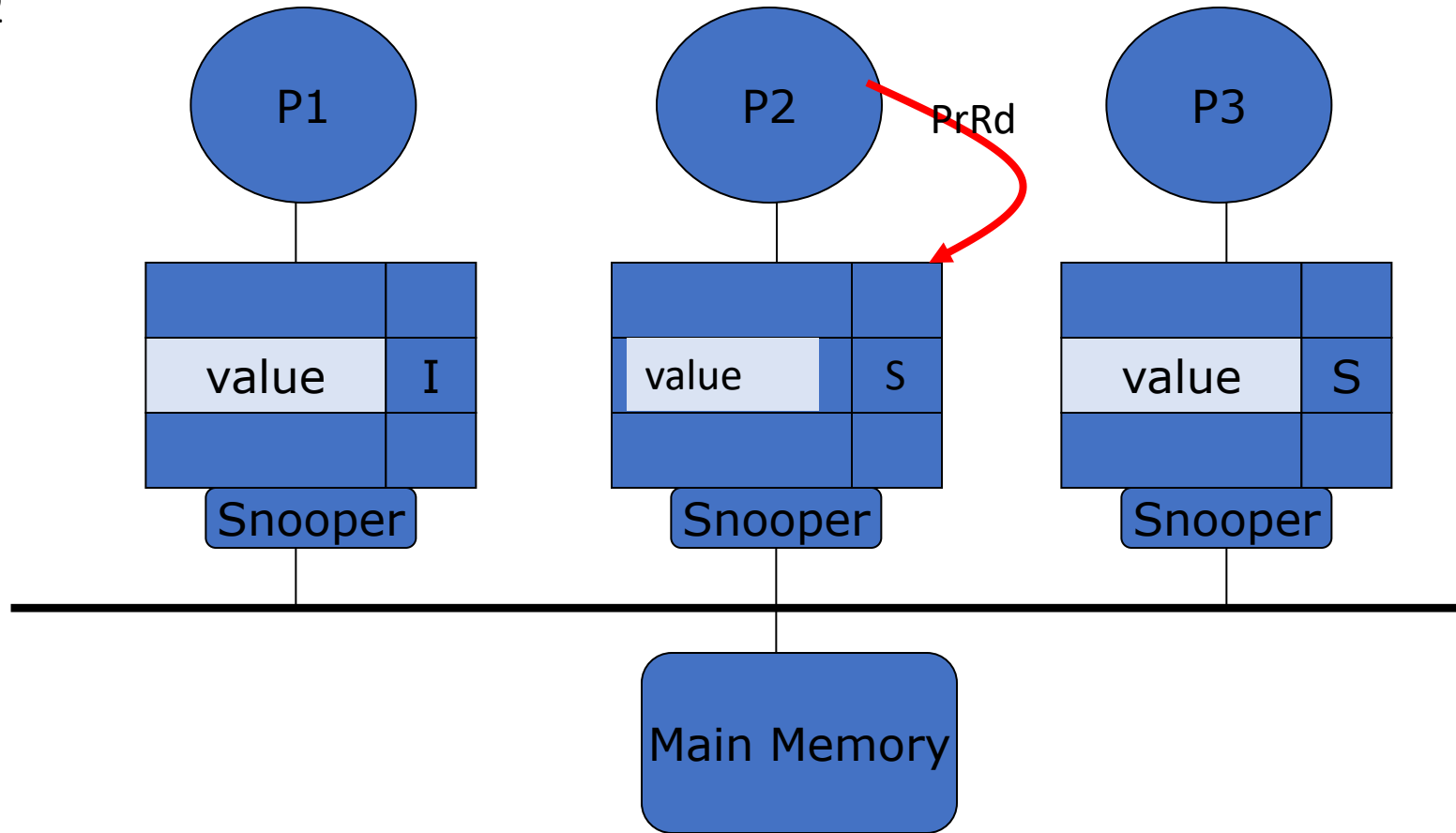
Read in 3



P3 wants to read the value. Its cache does not have a valid copy, so it places a BusRd on the bus. P2 has a modified copy, so it flushes the data on the bus and changes the status of the cache data to shared. The flush cancels the main memory access and updates the data in memory as well.

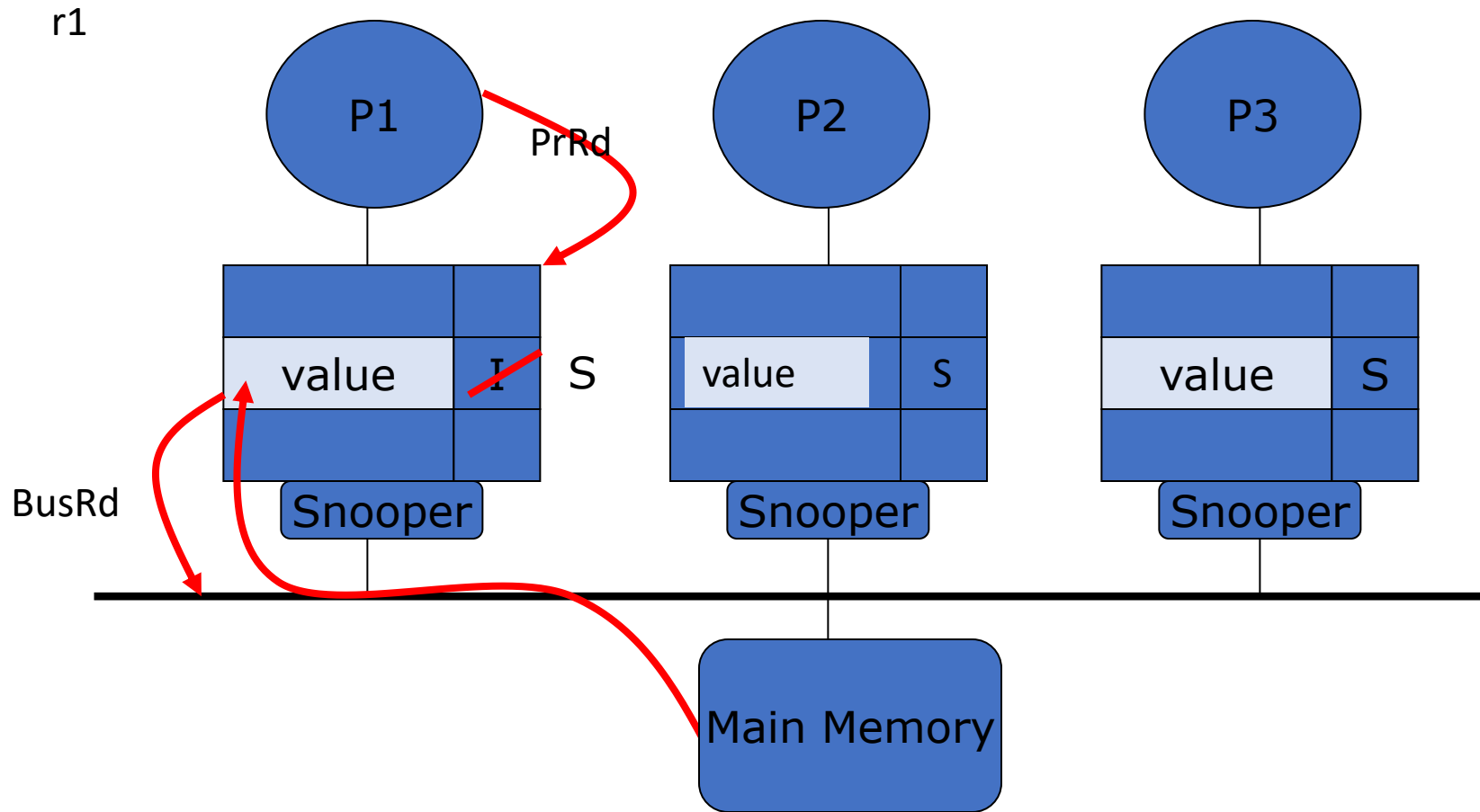
r2

Read in 2



P2 wants to read the value. Its cache has an up-to-date copy. No bus transactions need to take place as there is no cache miss.

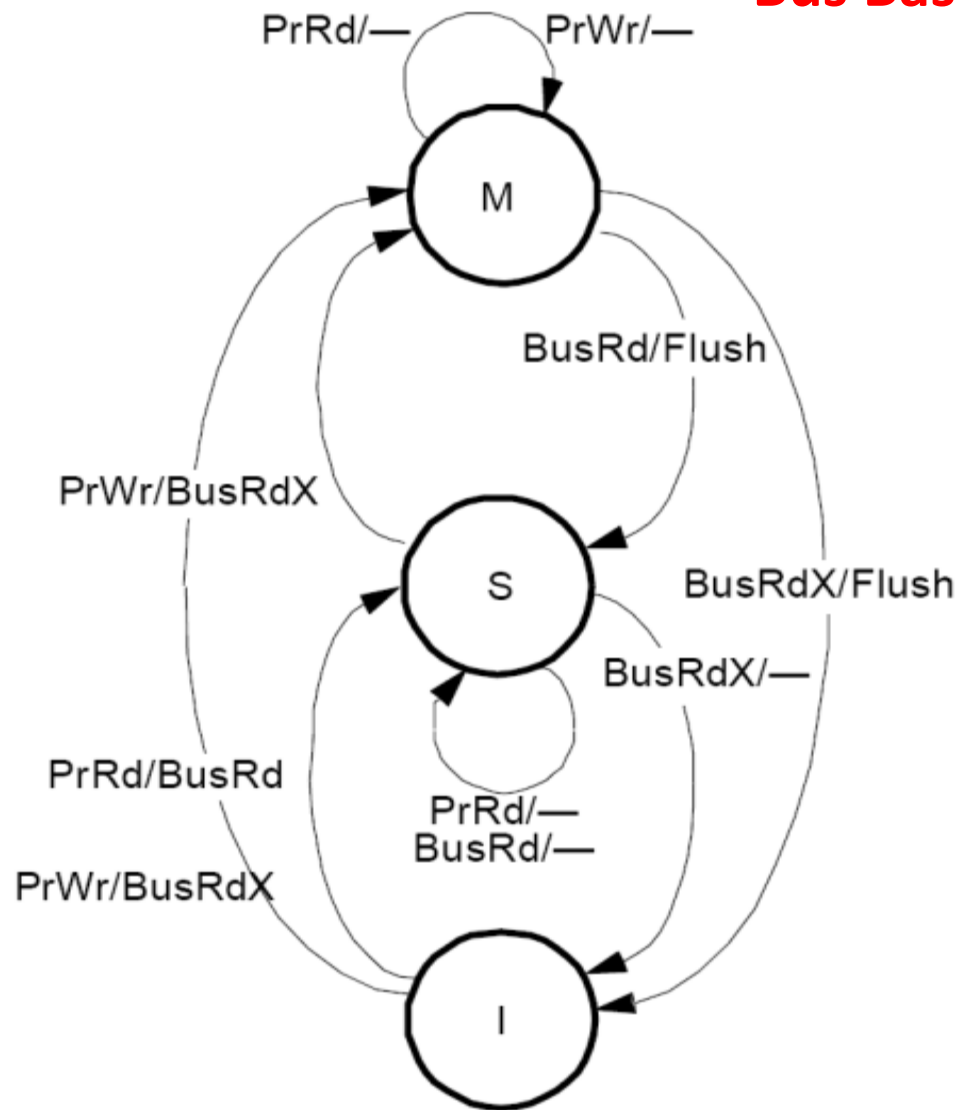
Read in 1



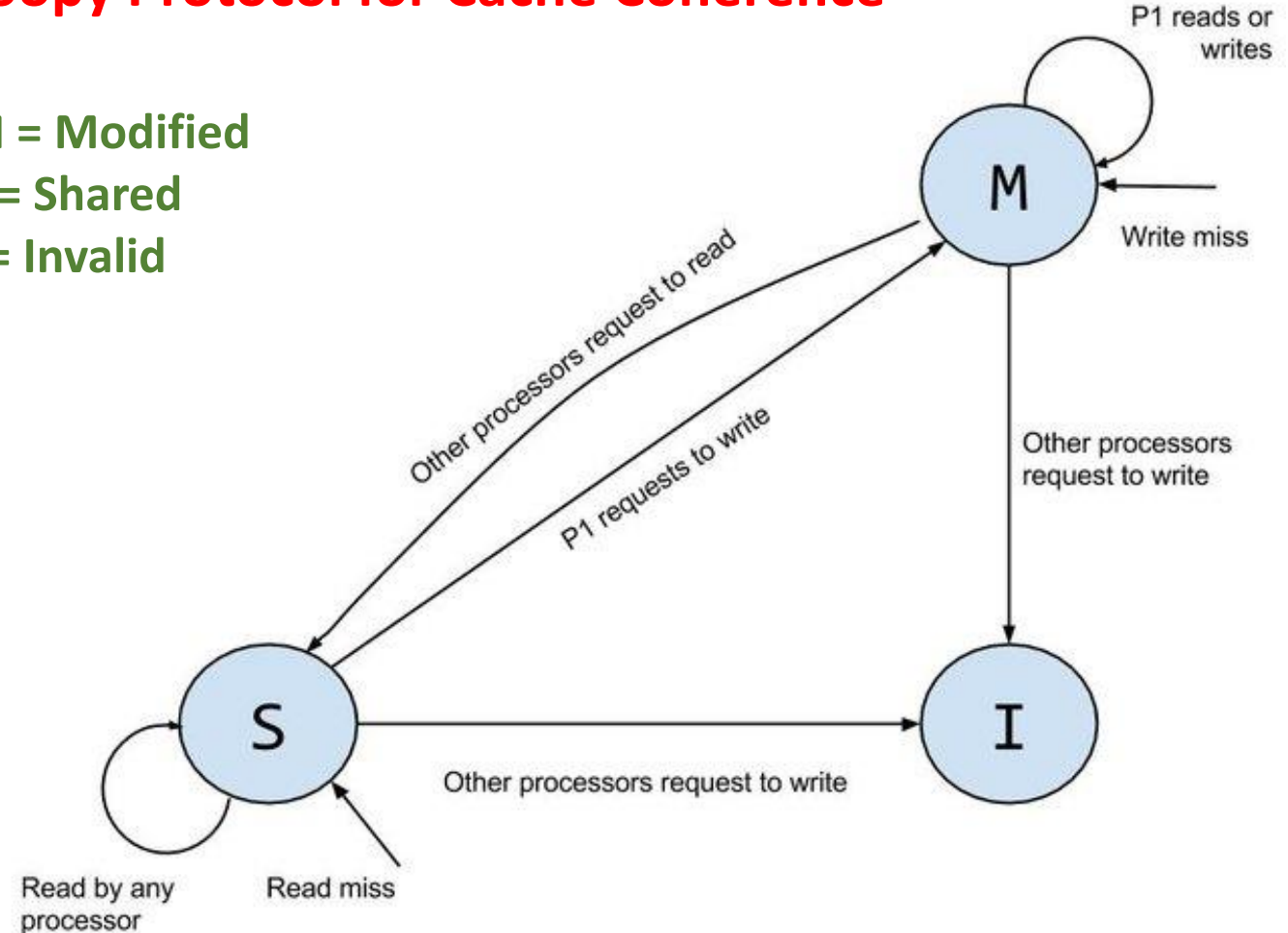
P1 wants to read the value. The cache does not have it, so it places a BusRd onto the bus for the data. The main memory controller provides the data as it has an up-to-date copy. The data goes into the cache in the shared state.

FSM of MSI for Proc Rd and Proc Wr

Bus Based Snoopy Protocol for Cache Coherence



M = Modified
S = Shared
I = Invalid



Readings

- Chap 5, 6 of P&H Textbook
- Image Credits: Youtube channel of David Schaffer