# Virtual Memory & DMA

Ravi Suppiah
Lecturer, NUS SoC

The Architecture for the Digital World®

## Learning Objectives

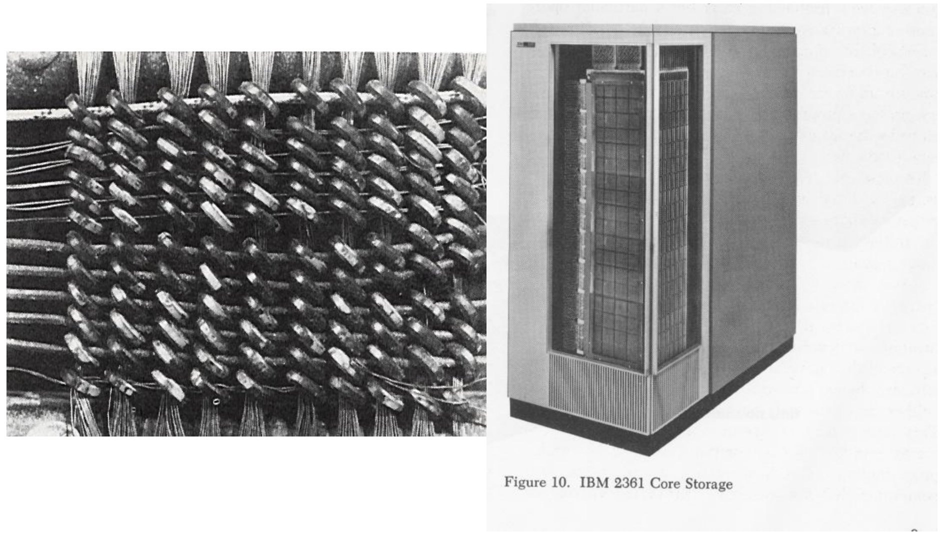
- Motivation for virtual memory from
  - Technology perspective
  - Software perspective

- Memory Hierarchy:
  - Cache, Main Memory, Disk
  - Spatial and Temporal Locality

- Virtual Memory
  - Virtual address space
  - Physical address space
  - Mapping between virtual and physical memory
  - Virtual/Physical Page
  - Page Table
  - Page Fault
  - TLB

## Memory Technology: 1950s

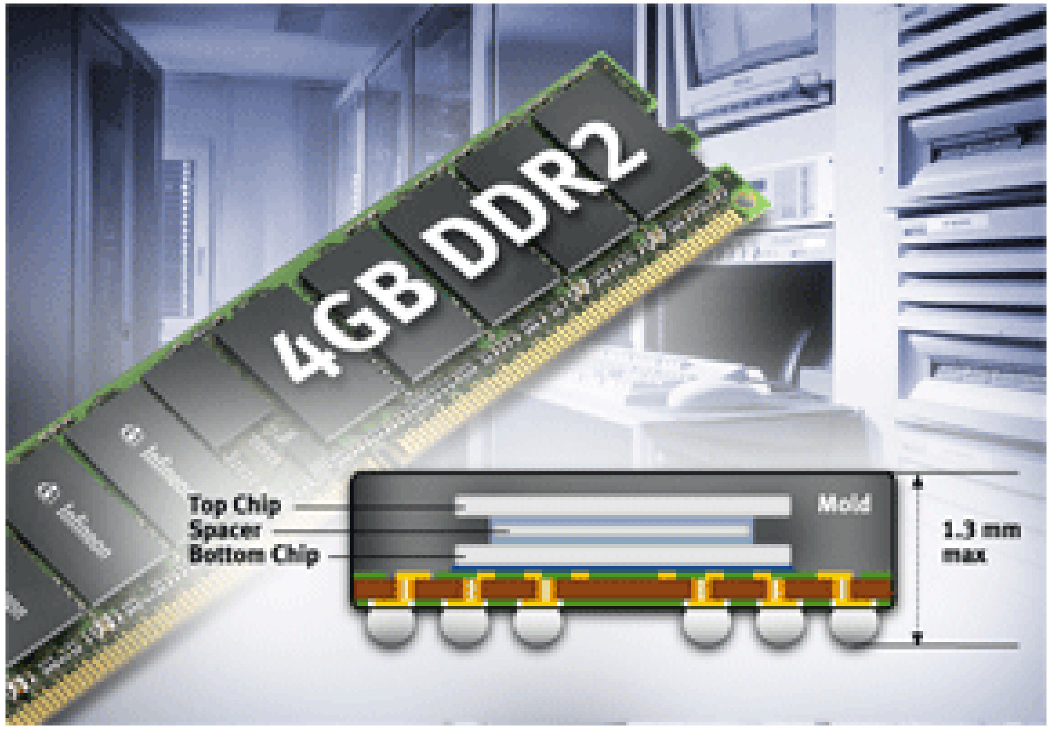
1948: Maurice Wilkes examining
EDSAC's delay line memory tubes
16-tubes each storing 32 17-bit words



1952: IBM 2361 16KB magnetic core memory

Maurice Wilkes: 2005

## Memory Technology today: DRAM

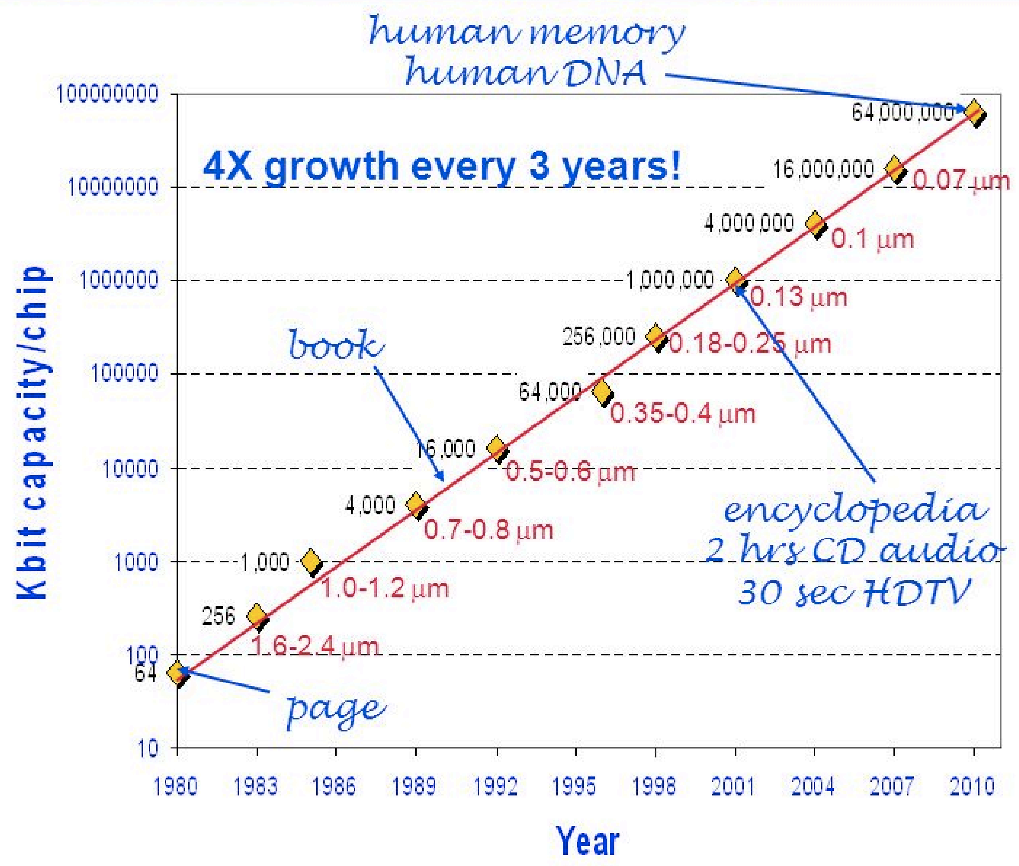


Infineon Technologies: stores data equivalent to 640 books, 32,000 standard newspaper pages, and 1,600 still pictures or 64 hours of sound

# Recent DRAM chip

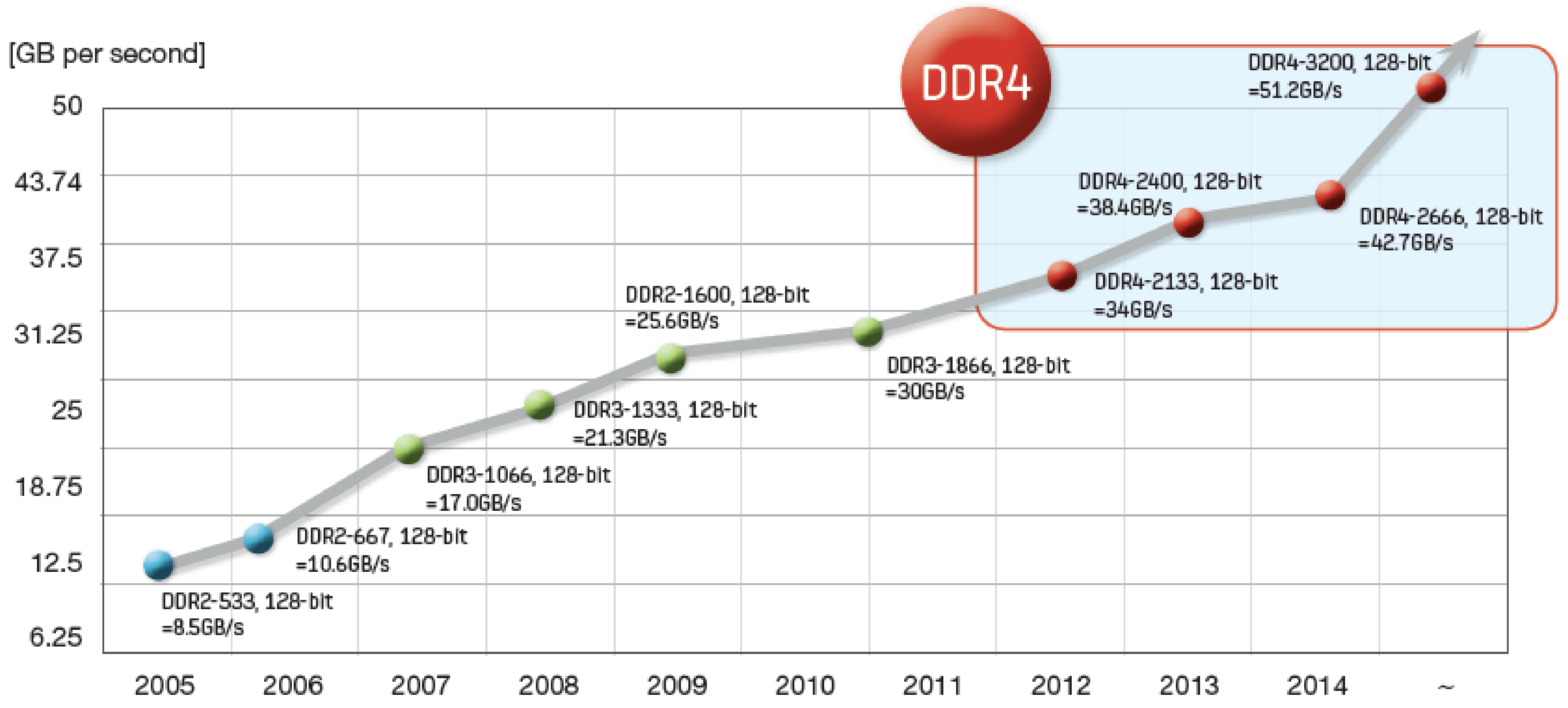
- 128GB DRAM chip from Samsung

# DRAM Capacity Growth



4X capacity increase almost every 3 years, i.e., 60% increase per year for 20 years

## DRAM Bandwidth Growth



DRAM Bandwidth grows by 1.3x per year

## Issue with DRAM

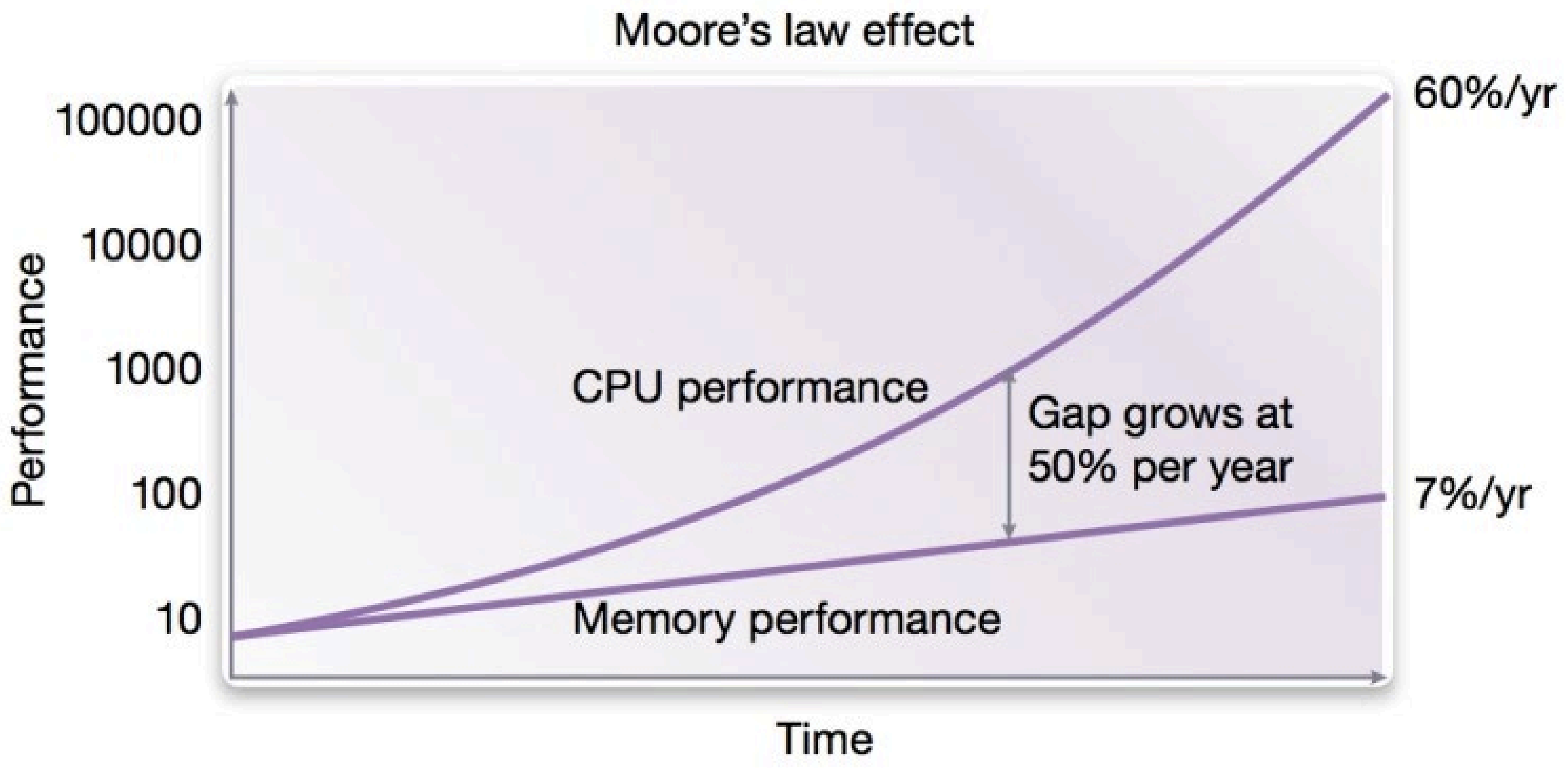
- 1.6x  capacity growth of DRAM per year
- 1.3x bandwidth growth of DRAM per year
- DRAM latency improvement is far behind

# Processor-DRAM Performance Gap

Memory Wall:

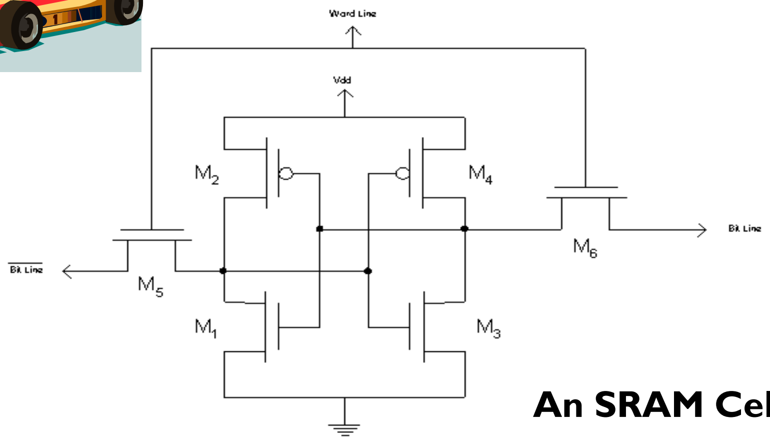
DRAM access latency: 50 ~ 70 ns

1 GHz Processor ➔ 1 ns per clock cycle but 50 ns to access DRAM
50 processor clock cycles per memory access !!

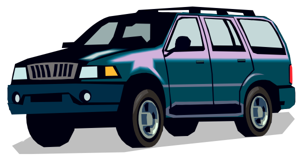


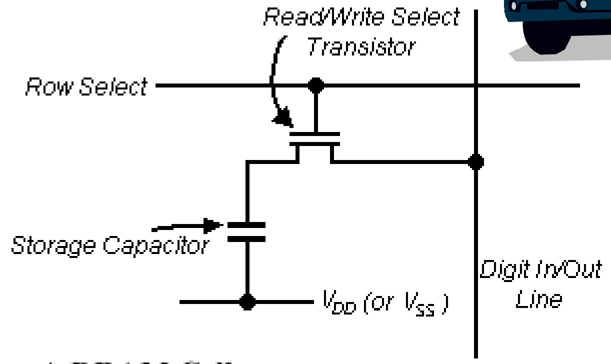
<u>Do we have any faster memory technology?</u>

## Fast Memory Technologies: SRAM



**An SRAM Cell**





**A DRAM Cell**

SRAM

6 transistor per memory cell

→ Low density

Fast access latency of 0.5 – 5 ns

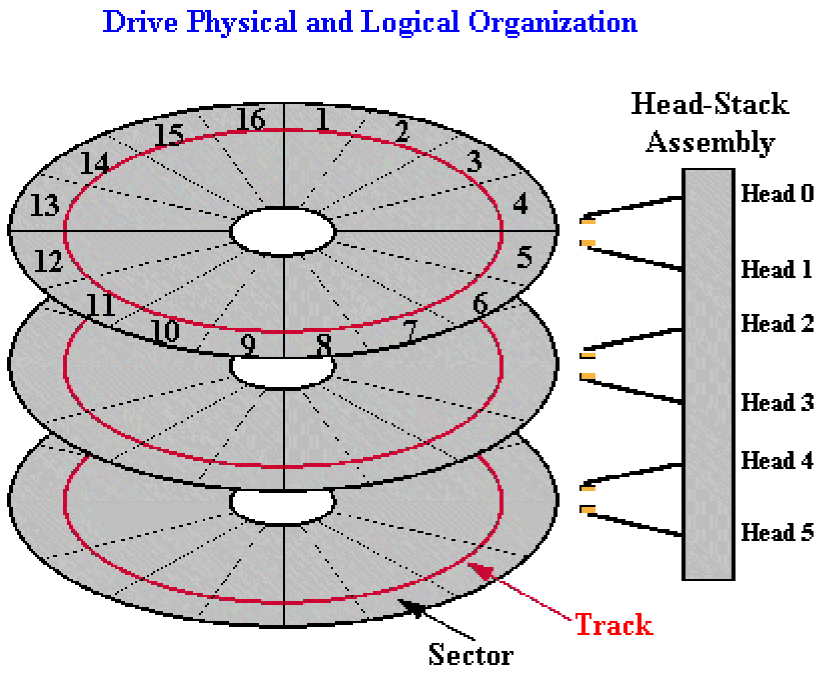
**DRAM**

1 transistor per memory cell

→ High density

Slow access latency of 50-70ns

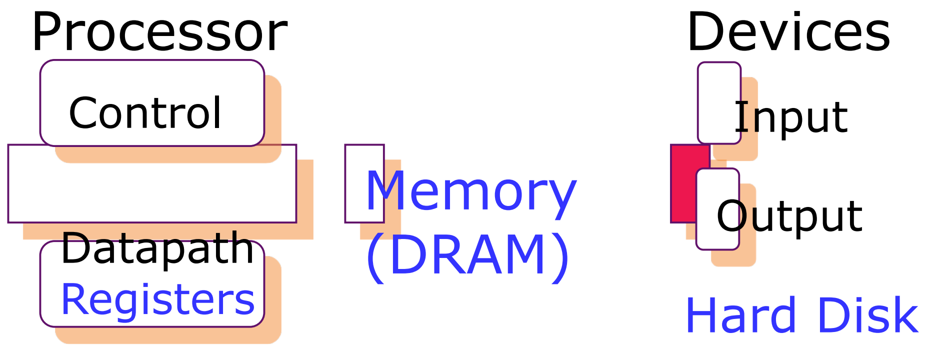
## Slow Memory Technologies: Magnetic Disk

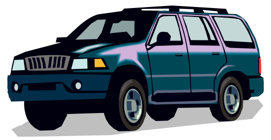


High-end hard disk:

Average latency: 5-20ms
Capacity: Seagate 4TB

# Quality vs. Quantity



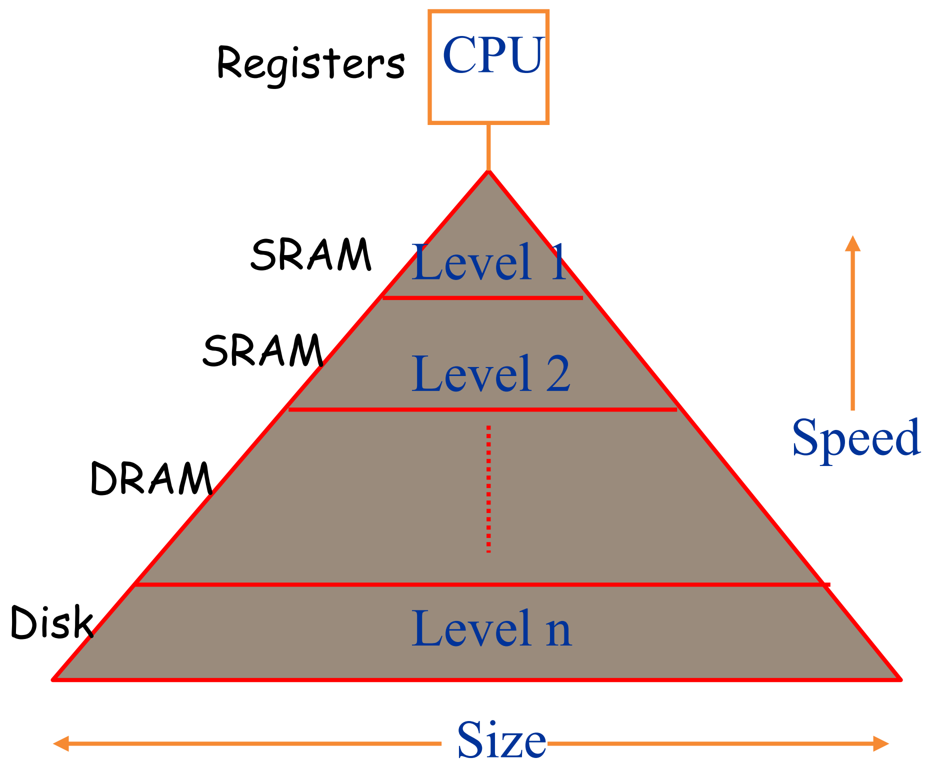


| | Capacity | Latency | Cost/GB |
|---|---|---|---|
| Register | 100s Bytes | 20 ps | $$$$ |
| SRAM | 100s KB | 0.5-5 ns | $$$ |
| DRAM | 10s GB | 50-70 ns | $ |
| Hard Disk | 100s GB | 5-20 ms | Cents |
| Ideal | 1 GB | 1 ns | Cheap |

## Best of Both Worlds

- What we want: A BIG and FAST memory
- Memory system should perform like 1GB of SRAM (1ns access time) but cost like 1GB of slow memory
- Key concept: Use a hierarchy of memory technologies
  - Small but fast memory near CPU
  - Large but slow memory farther away from CPU

## Memory Hierarchy

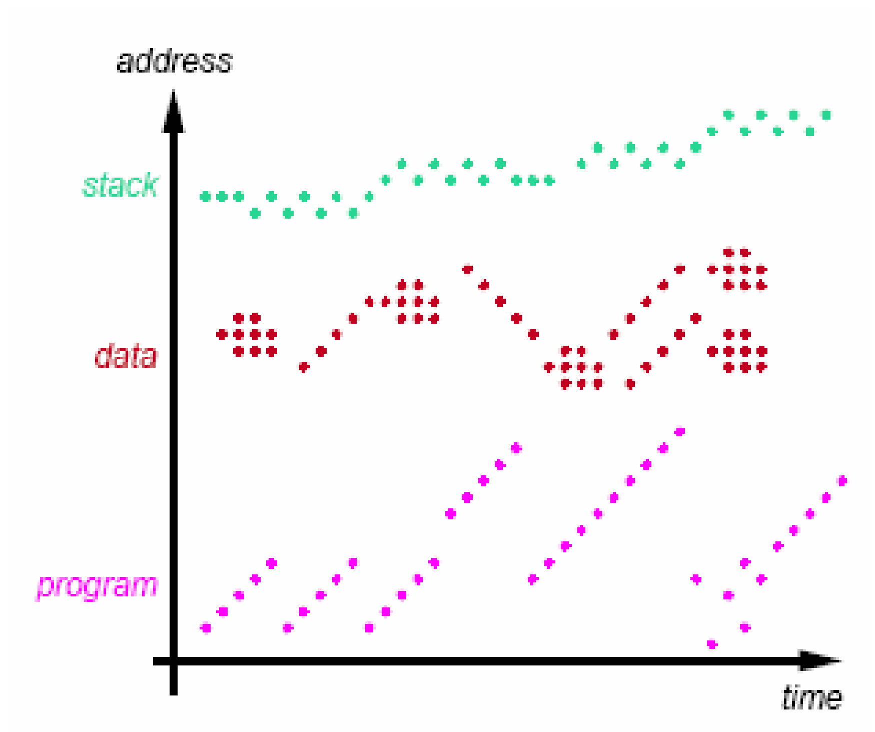


## The Basic Idea

- Keep the frequently and recently used data in smaller but faster memory
- Refer to bigger and slower memory only when you cannot find data/instruction in the faster memory
- Why does it work? Principle of Locality

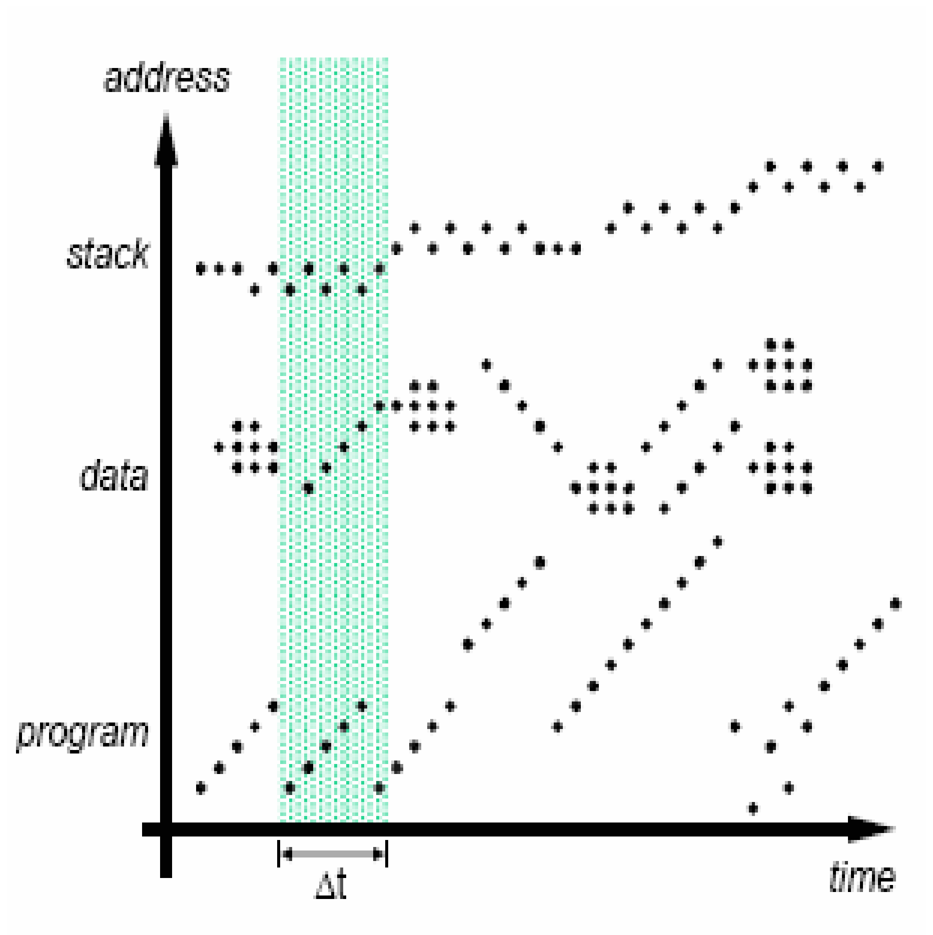
Program accesses only a small portion of the memory address space within a small time interval

# Locality



- Temporal locality
  - If an item is referenced, it will tend to be referenced again soon
- Spatial locality
  - If an item is referenced, nearby items will tend to be referenced soon
- Locality for instruction
- Locality for data

# Working Set



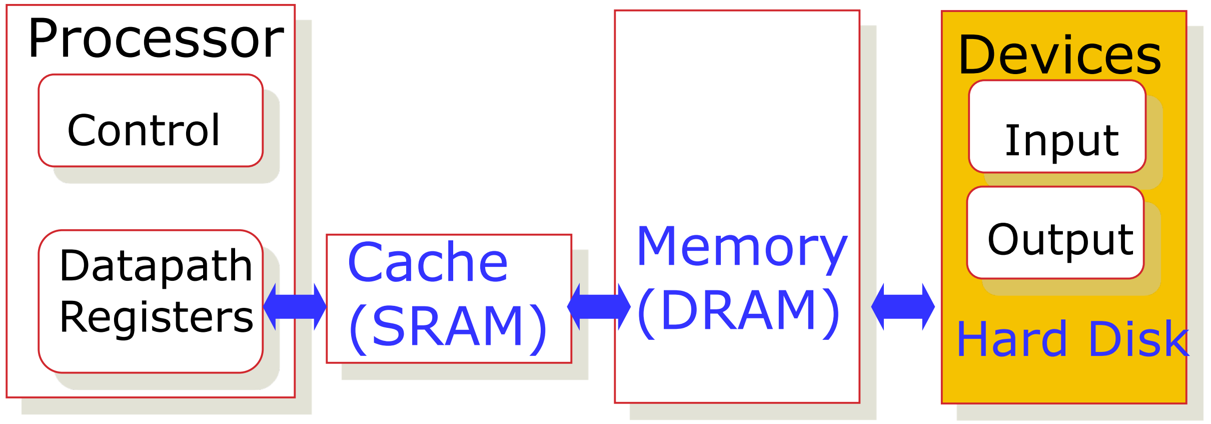
- Set of locations accessed during Δt
- Different phases of execution may use different working sets
- Want to capture the working set and keep it in the memory closest to CPU

# Exploiting Memory Hierarchy (1/2)

- **Visible to Programmer**
  - Various storage alternatives, e.g., register, main memory, hard disk
  - Tell programmer to use them cleverly

- **Transparent to Programmer** **(except for performance)**
  - Single address space for programmer
  - Processor automatically assigns locations to fast or slow memory depending on usage patterns

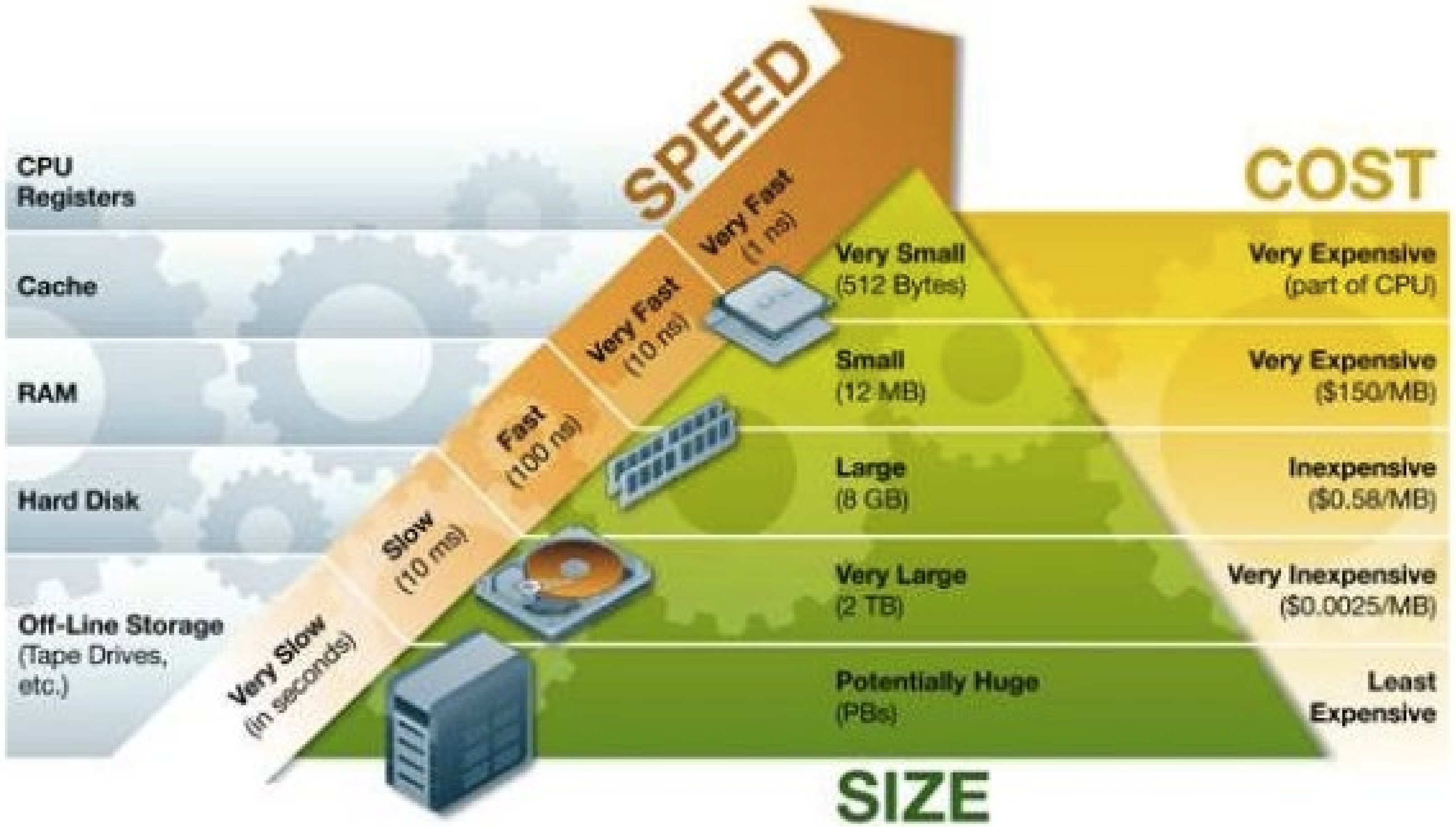
# Exploiting Memory Hierarchy (2/2)



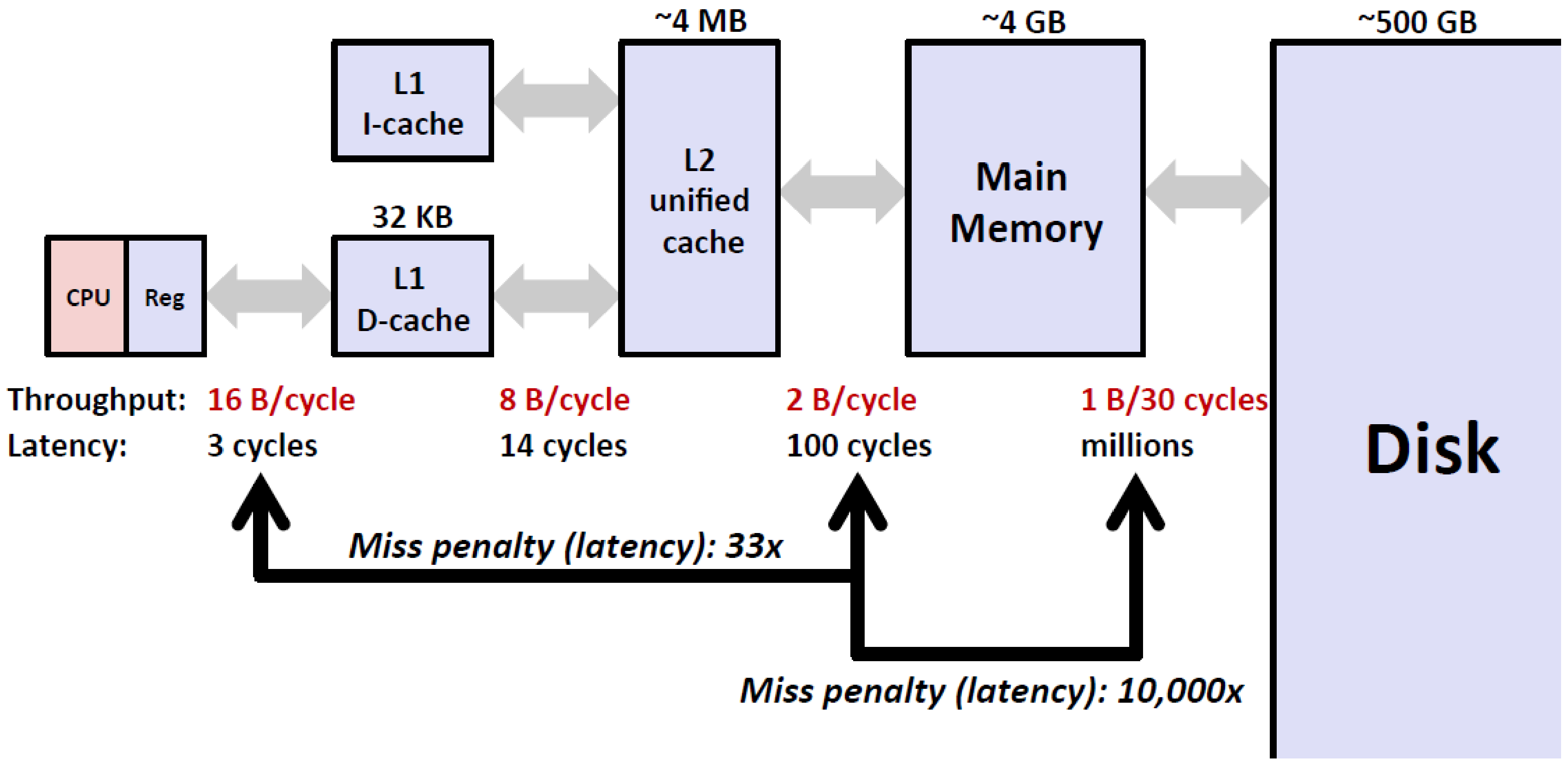
- **How to make SLOW main memory appear faster?**
- **Cache – a small but fast SRAM near CPU**
  - Often in the same chip as CPU
  - Introduced in 1960; almost every processor uses it today
  - Hardware managed: Transparent to programmer

- **How to make SMALL main memory appear bigger than it is? Virtual memory**
  - OS managed: Again transparent to programmer

# Memory Hierarchy Summary



## Memory Hierarchy: Core 2 Duo



## Virtual Memory Motivation Re-iterated

- Hard disk is cheap
- Make a portion of the hard disk *look like* memory to the CPU, so that we can fool the CPU into thinking there's more memory that we actually paid for!
- We can do this by having instructions and data that the CPU is currently interested in, in main memory. Everything else stays on disk
- In this way we can squeeze MUCH MORE instructions and data than otherwise possible!

# Processes

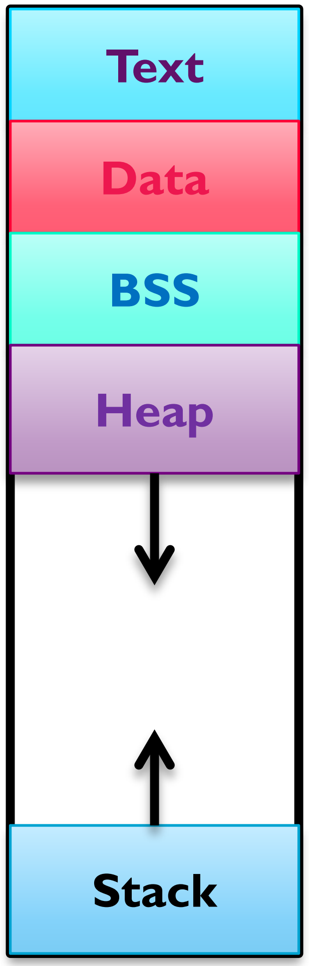
- **A process is an instance of a running program**

- **Process provides each program with two key abstraction**
  - Independent control flow
  - Each process seems to have exclusive use of the CPU
  - Private virtual address space
  - Each process seems to have exclusive use of main memory

- **How are these illusions maintained?**
  - Process executions are interleaved through scheduling
  - Address space managed by virtual memory system

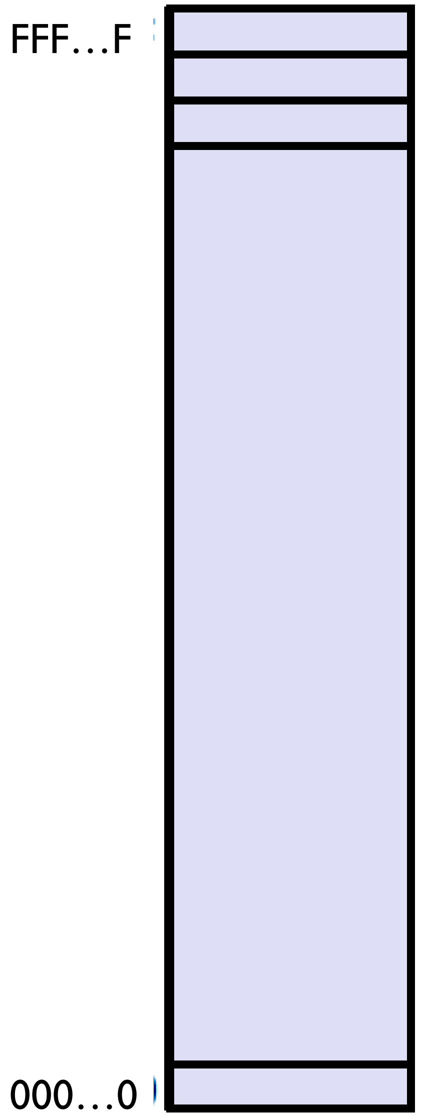
## Process Memory

- **Text:** **Code**
- **Data:** **Initialized global and static variables**
- **BSS:** **Uninitialized global and static variables**
- **Heap:** **Dynamic Memory**
- **Stack:** **Local variables**



# Virtual Memory

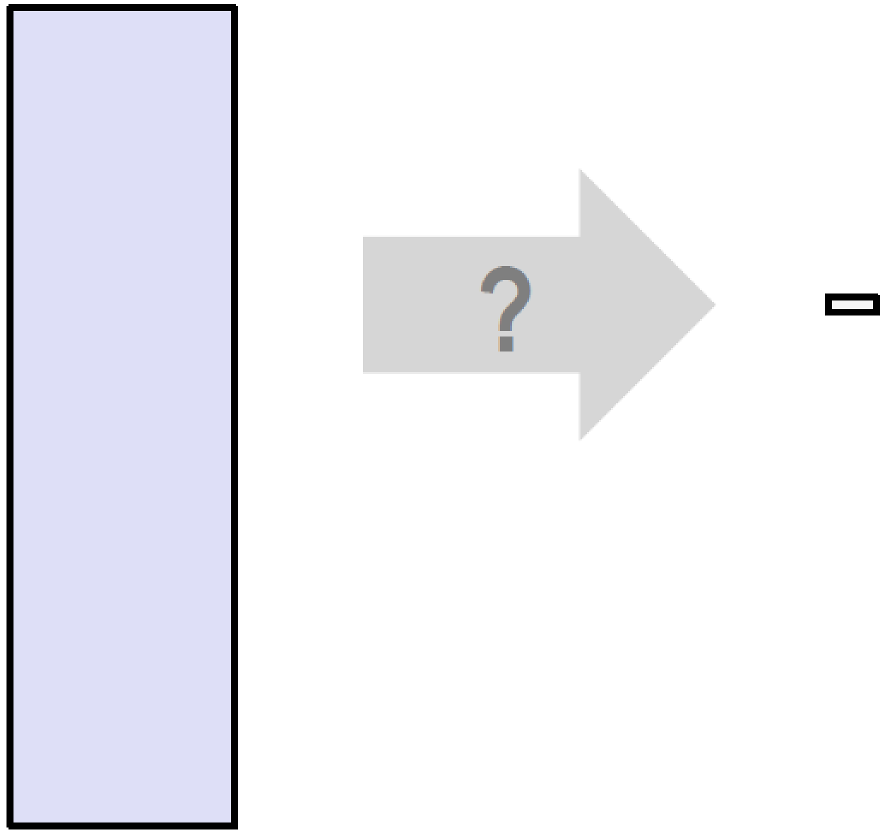
- **Program refers to virtual memory addresses**
  - Conceptually memory is just a very large array of bytes
  - Each byte has its own address
  - System provides address space "private" to process
- **Allocation: Compiler and run-time system**
  - Where different program objects should be stored
  - All allocation within single virtual address space
- **What problems does virtual memory solve?**



## Problem 1: How does Everything Fit?

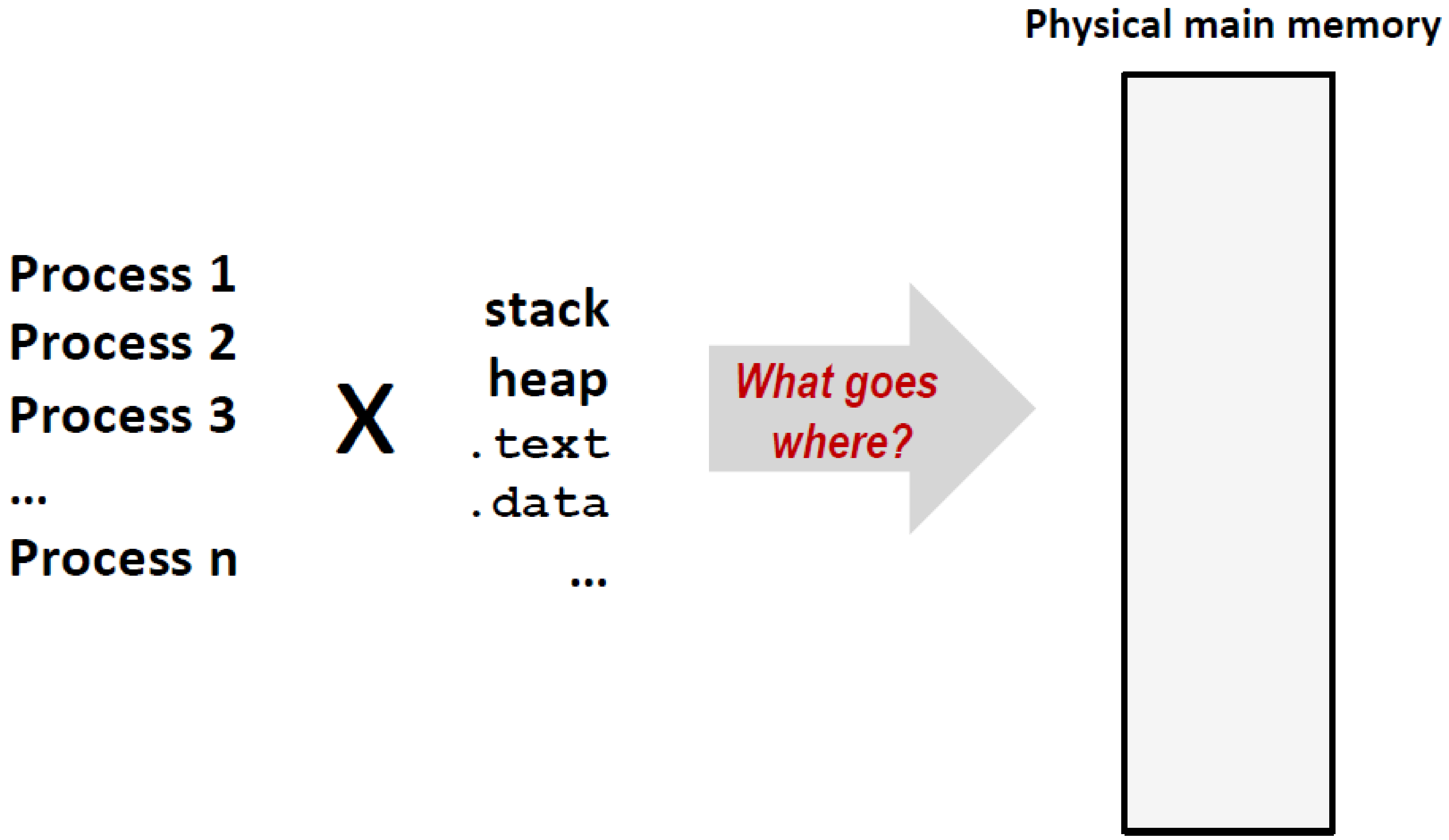
64-bit addresses:
16 Exabyte

Physical main memory:
Few Gigabytes

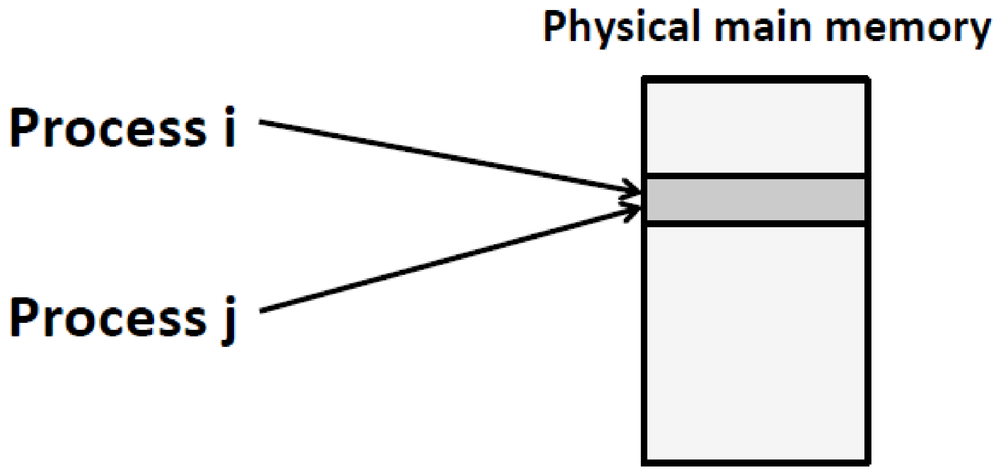


And there are many processes ....

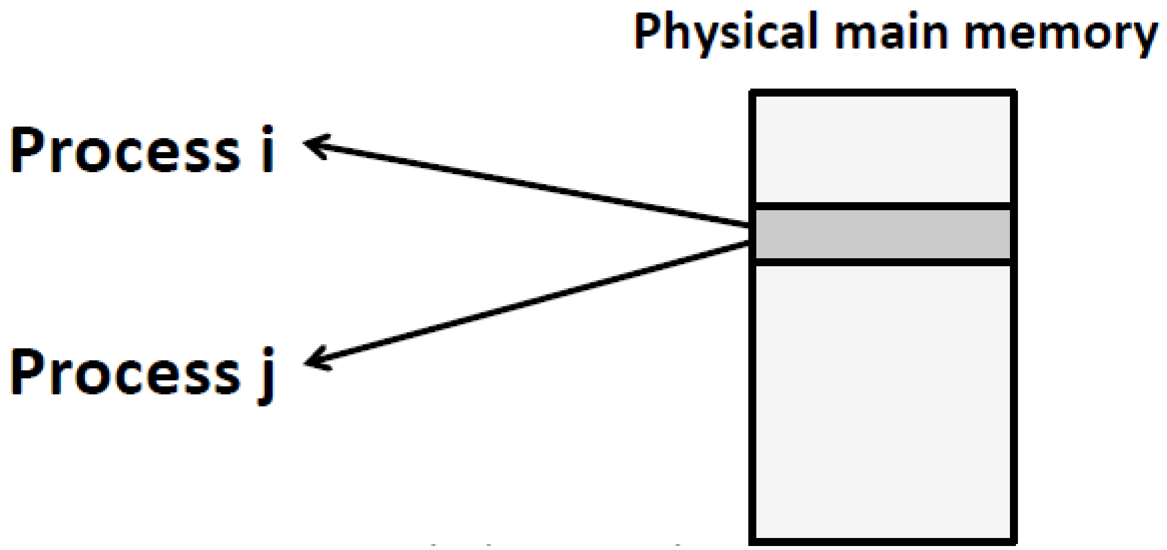
# Problem 2: Memory Management



## Problem 3: How to Protect?



# Problem 4: How to Share?

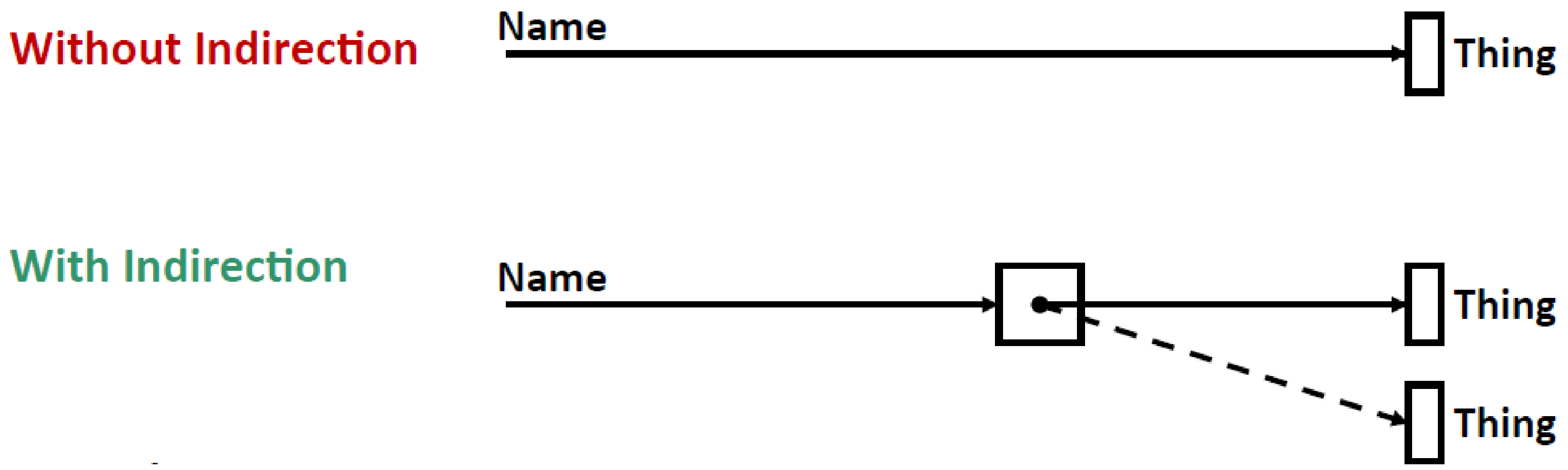


## How would you solve those problems?

- Fitting a huge memory into a tiny physical memory
- Managing the memory space of multiple processes
- Protecting processes from stepping on each other’s memory
- Allowing processes to share common parts of memory

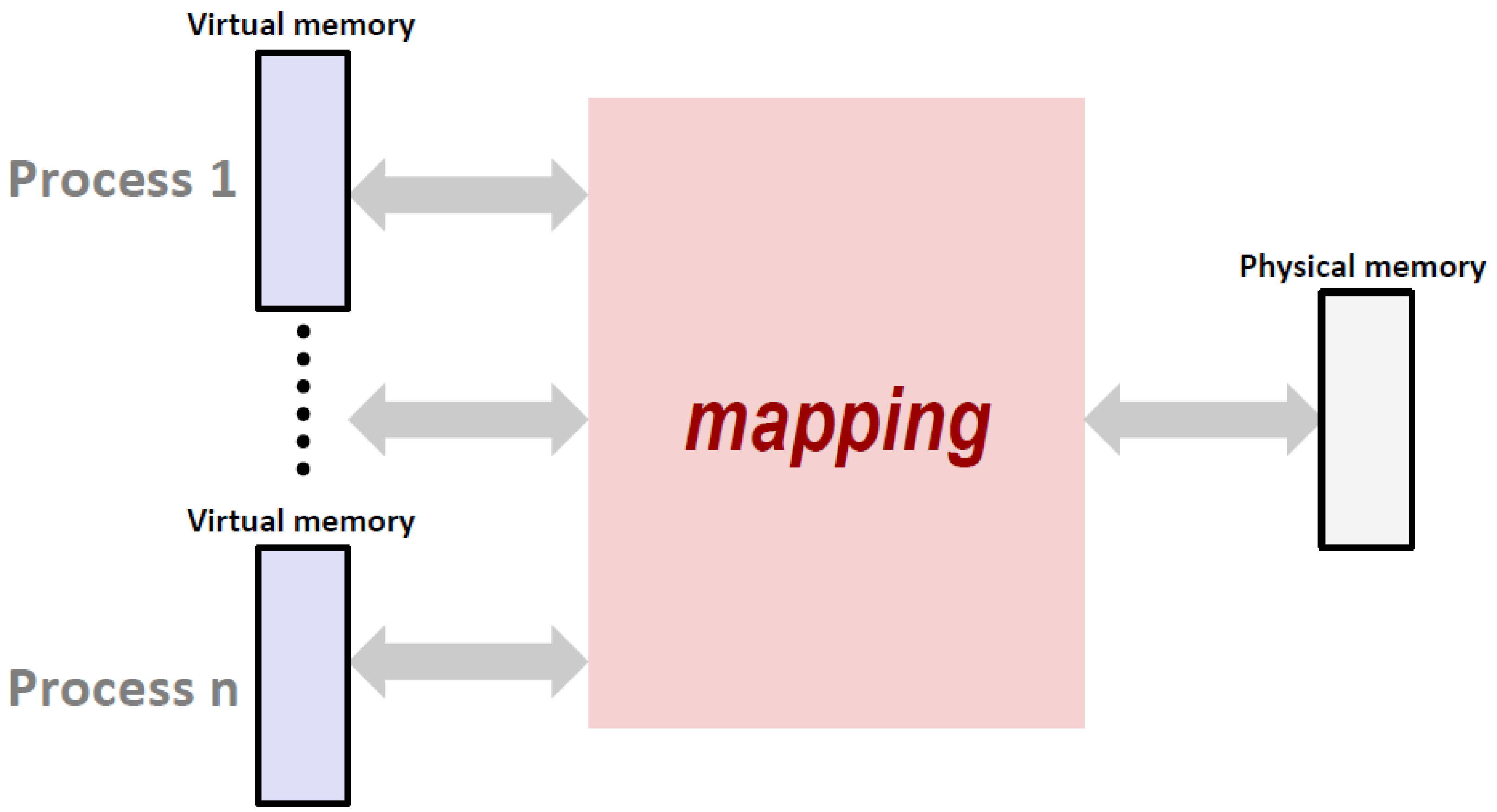
## Indirection

- Any problem in Computer Science can be solved by adding another level of indirection
- The ability to reference something using a name, reference, or container instead of the value itself. A flexible mapping between a name and a thing allows changing the thing without notifying the holder of the name



## Virtual Memory Solution: Indirection

- **Each process gets its own private virtual address space**
- **Virtual addresses are mapped to physical memory**

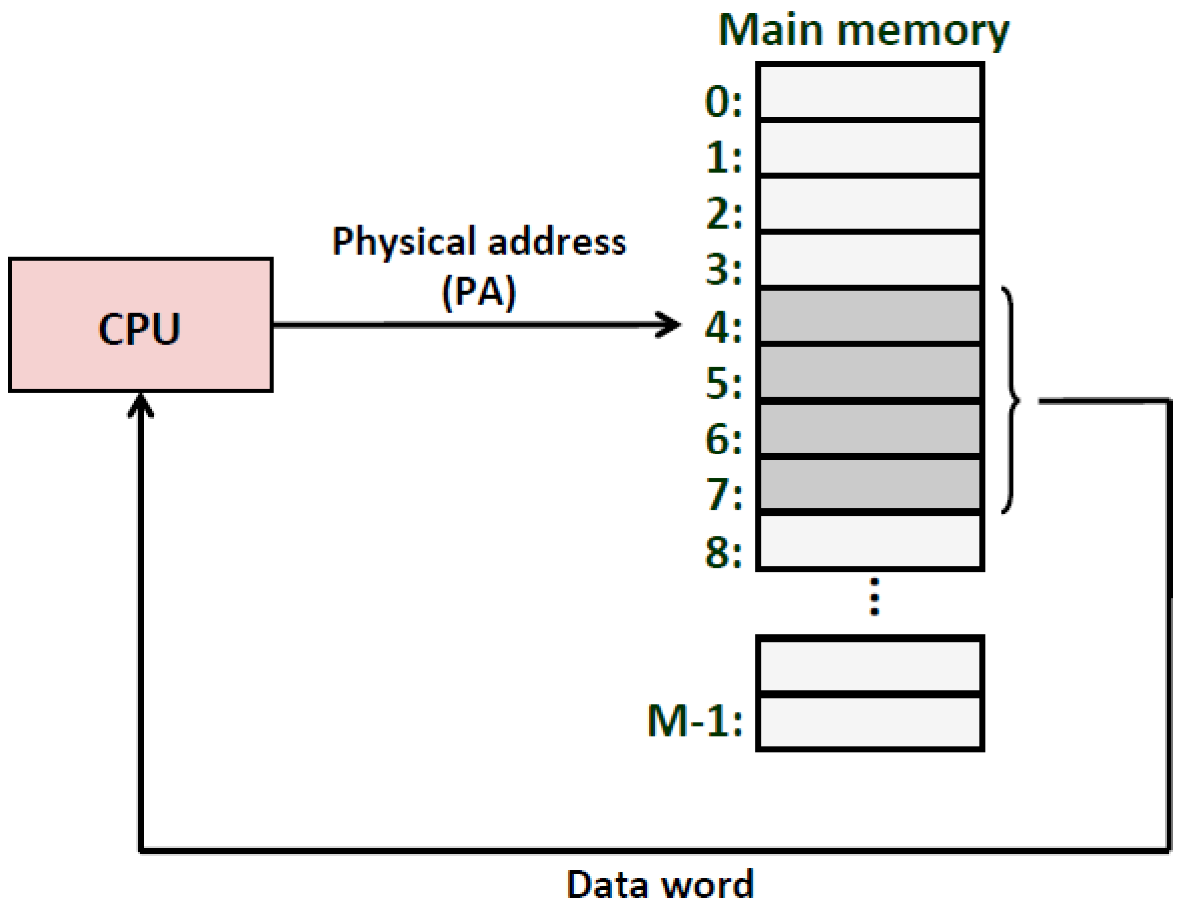


# Address Spaces

- **Virtual Address Space: Set of $N = 2^n$ virtual addresses**
  **{0, 1, 2, 3, ….., N-1}**
- **Physical Address Space: Set of $M = 2^m$ physical addresses ($n > m$)**
  **{0, 1, 2, 3, ….., M-1}**
- **Each byte in physical main memory:**
  - One physical address
  - Zero, one, or more virtual addresses

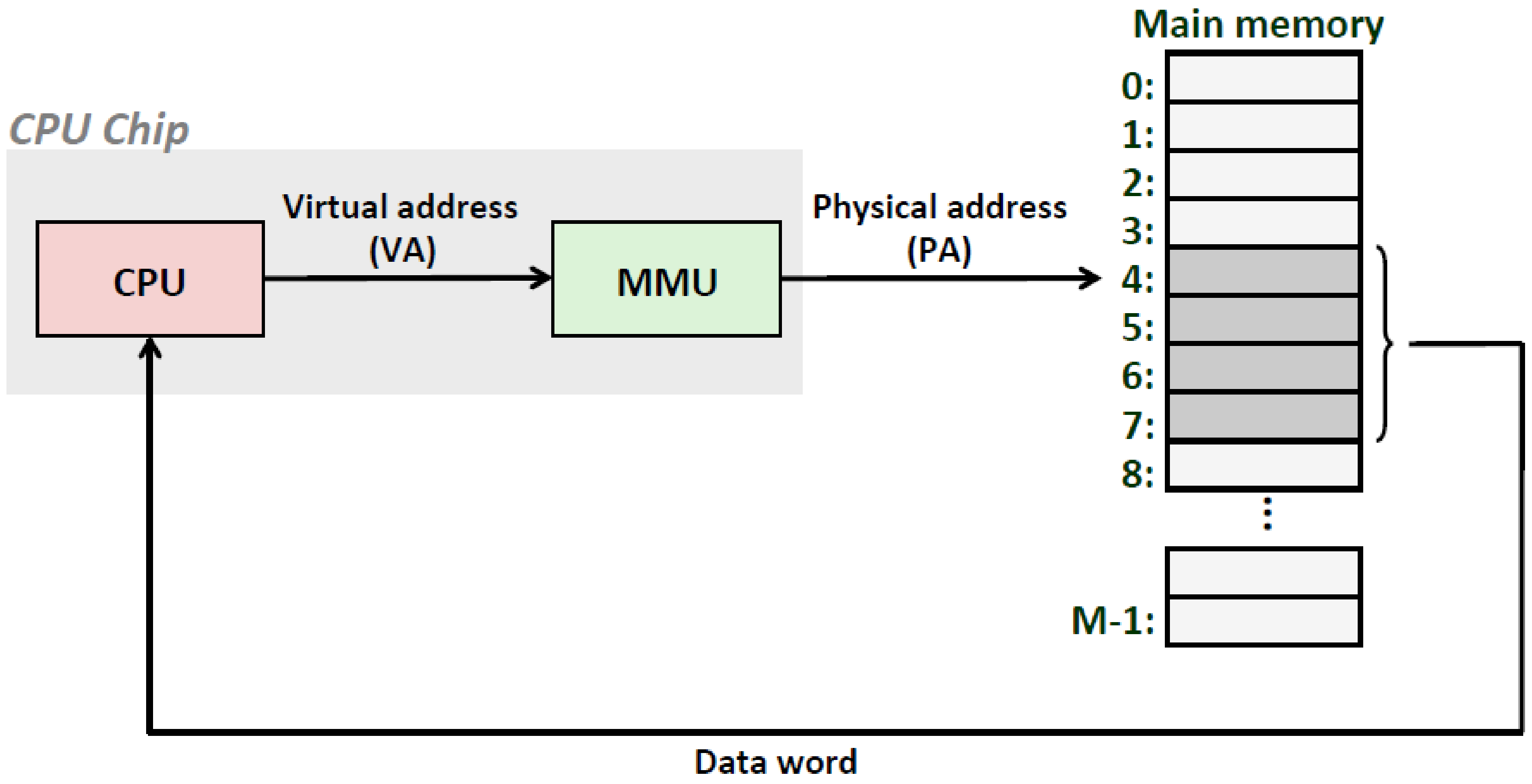
# A System using Physical Addressing

- **Uses in simple systems like embedded micro-controllers**



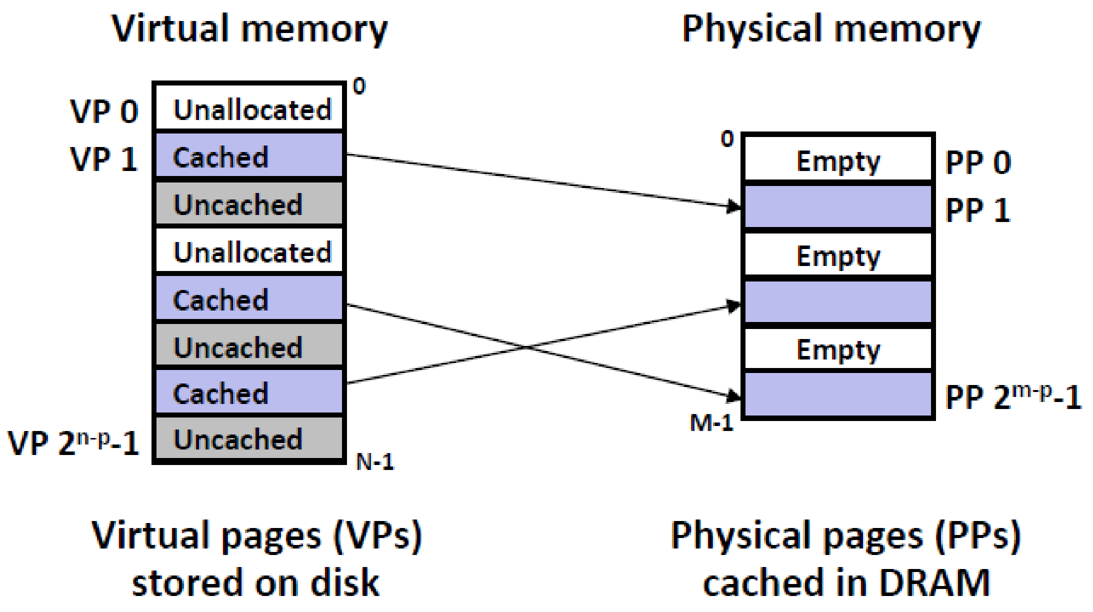
## A System using Virtual Addressing

- **Used in all modern desktops, laptops, servers, smartphones**
- **One of the great ideas in Computer Science**
-



# Virtual Memory Pages

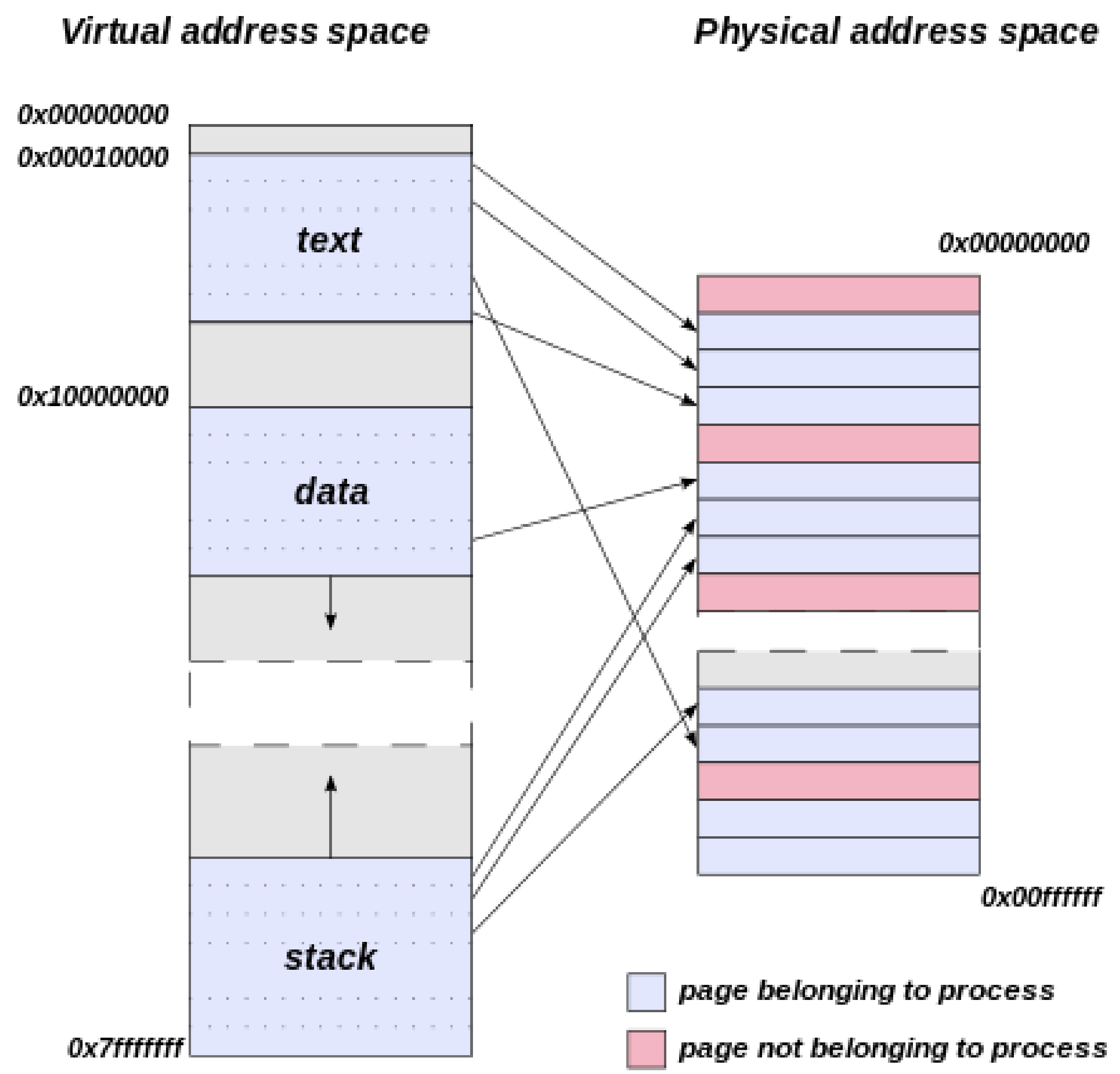
- Virtual memory can be thought of as an array of $N = 2^n$ contiguous bytes stored on a disk
- Virtual memory in partitioned into number of blocks called pages (size of page $P = 2^p$ bytes)
- Then physical main memory (DRAM) is used as a cache (temporary store) for some pages of the virtual memory array



## Status of a Virtual Memory Page

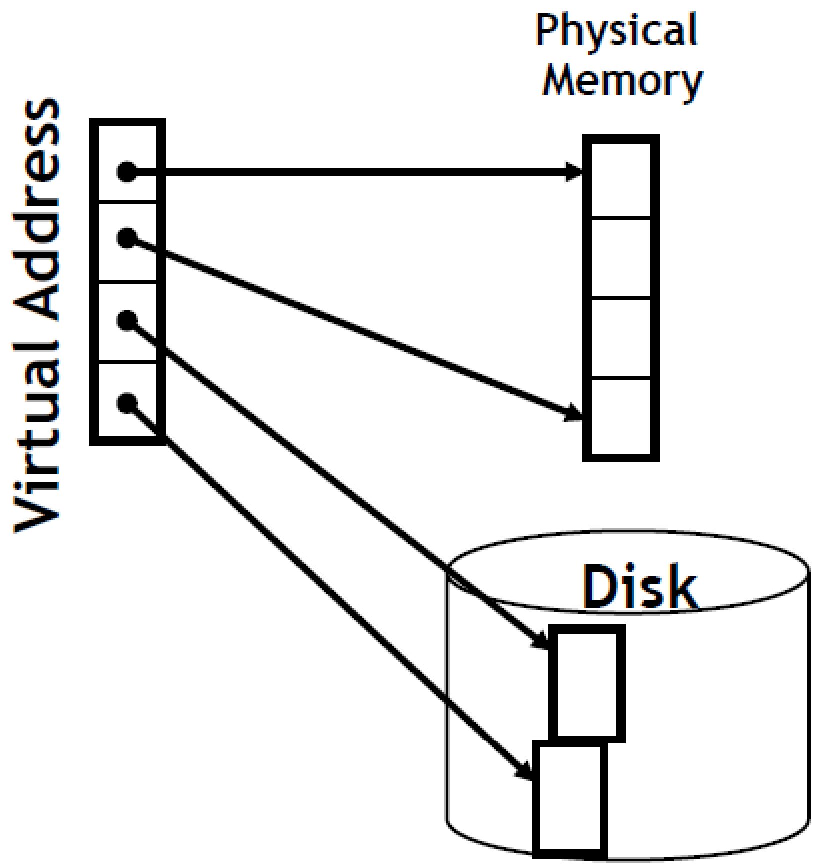
- Unallocated: the page is not being used by the process (e.g., the pages between heap and stack)
- Uncached: the page is being used by the process but it is not cached in DRAM
- Cached: the page is being used by the process and it is cached in DRAM

# Virtual & Physical Memory Pages



## Virtual Address Mapping

- **A virtual address can be mapped to either physical memory or disk**



## Virtual Memory Example (1/3)

- **16KB virtual address space: 14-bit virtual address**
- **4KB physical address space: 12-bit physical address**
- **Page size = 64 bytes**

- **How many virtual pages? 16KB/64B = 256**

- **How many physical pages? 4KB/64B = 64**

# Virtual Memory Example (2/3)

- 16KB virtual address space: 14-bit virtual address
- 4KB physical address space: 12-bit physical address
- Page size = 64 bytes

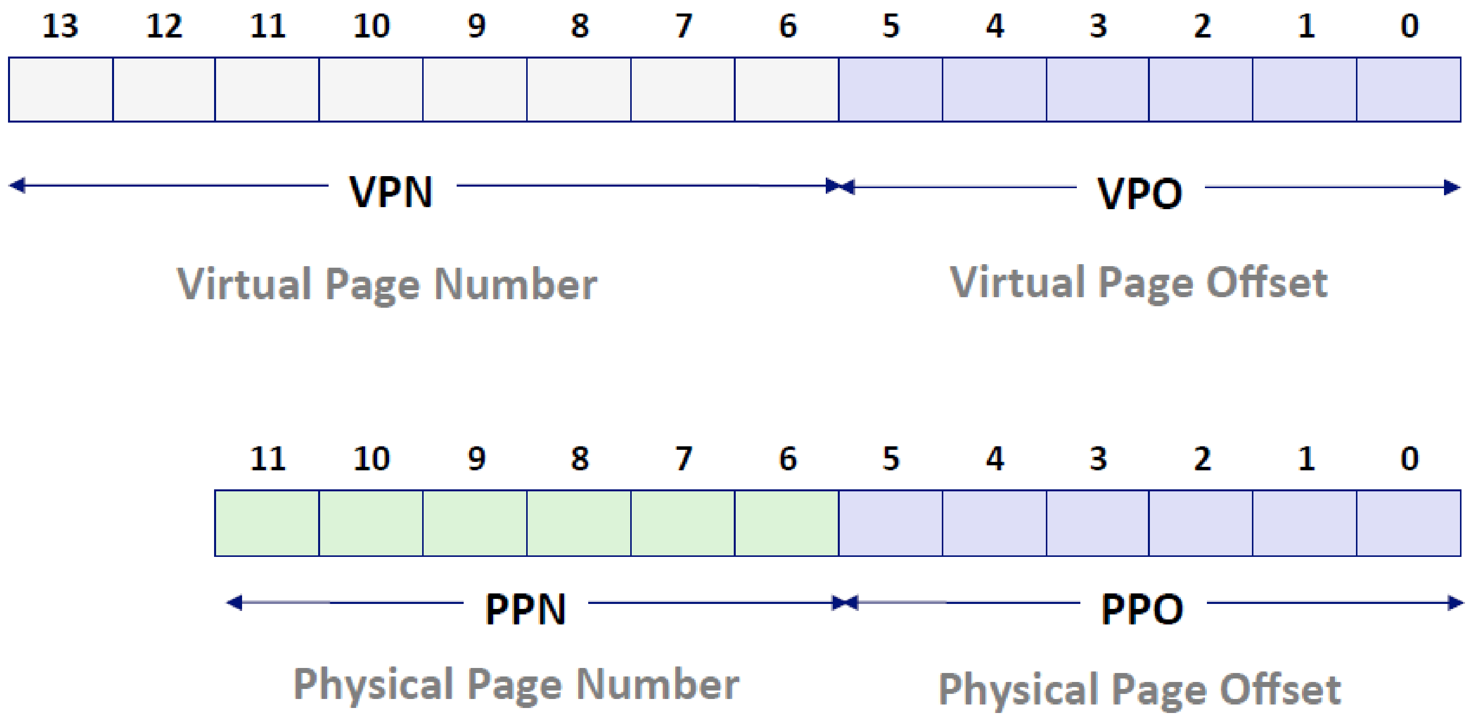
- How many virtual pages? 16KB/64B = 256
- How many bits for virtual page ID? 256 pages = $2^8$ require 8 bits

- How many physical pages? 4KB/64B = 64
- How many bits for physical page ID? 64 pages = $2^6$ require 6 bits

- How many bits to access an address within a page (page offset)?
  64 bytes = $2^6$ require 6 bits

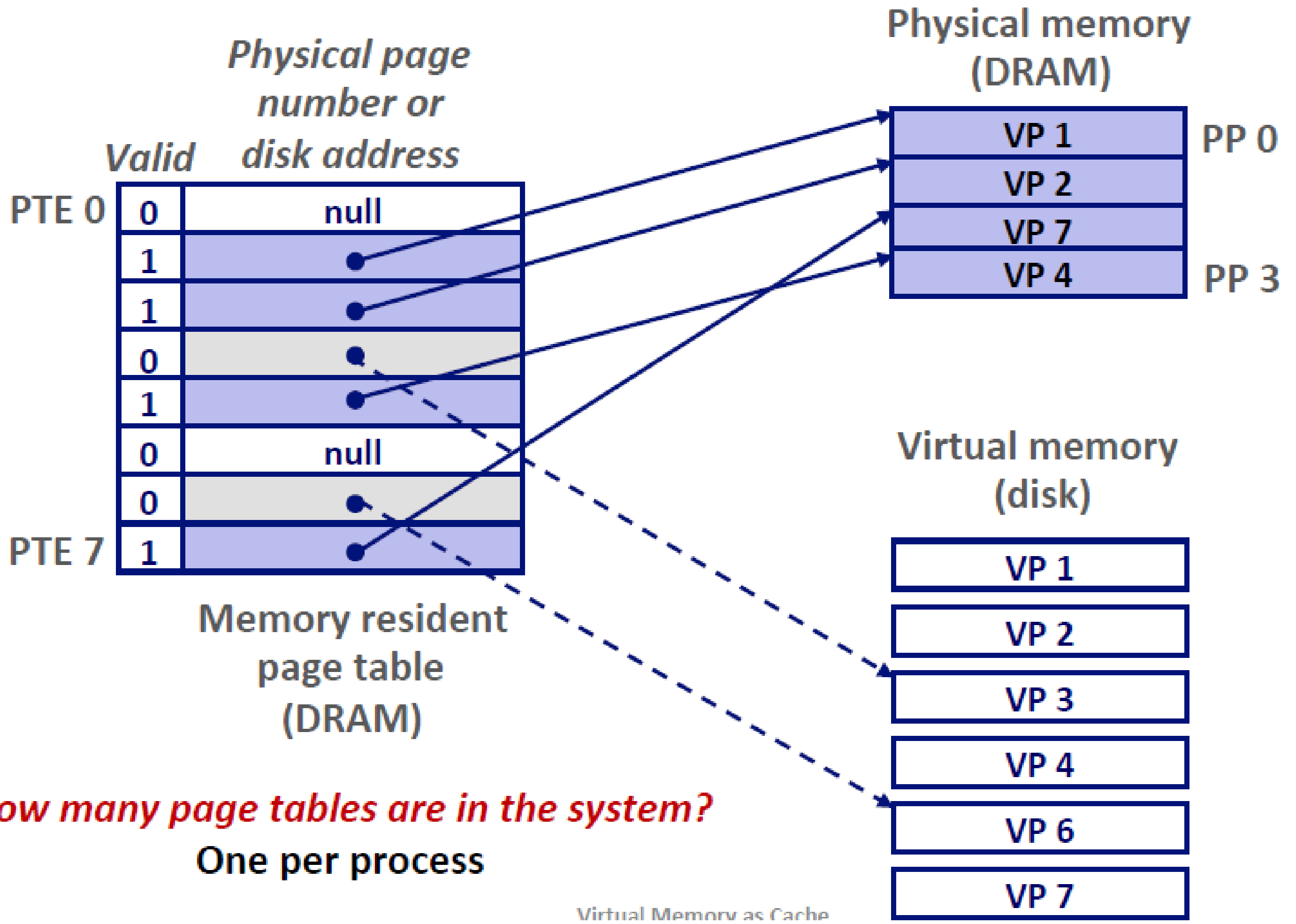
# Virtual Memory Example (3/3)

- 14-bit virtual addresses
- 12-bit physical addresses
- Page size = 64 bytes



## Address Translation: Page Tables

- A **page table** is an array of page table entries (PTE) that maps virtual pages to physical pages



*How many page tables are in the system?*

One per process

## More about Page Table

- **One page table per process**
- **Page tables are stored in main memory (DRAM)**
- **Page table should have one page table entry (PTE) corresponding to each virtual page**
  - 256 virtual pages will need 256 PTE
- **Each PTE stores the status of the corresponding virtual page**
  - 0 implies uncached and 1 implies cached
- **If a virtual page is cached in DRAM, then PTE contains the physical page number**

## Page Table Example

| | Physical Page Number (PP) | Valid Bit |
|---|---|---|
| 0 | 0x10 | 1 |
| 1 | DISK | 0 |
| 2 | 0x23 | 1 |
| 3 | 0x15 | 1 |
| 4 | DISK | 0 |
| 5 | 0x1B | 1 |
| 6 | 0x2E | 1 |
| … | … | … |
| 255 | 0x2A | 1 |

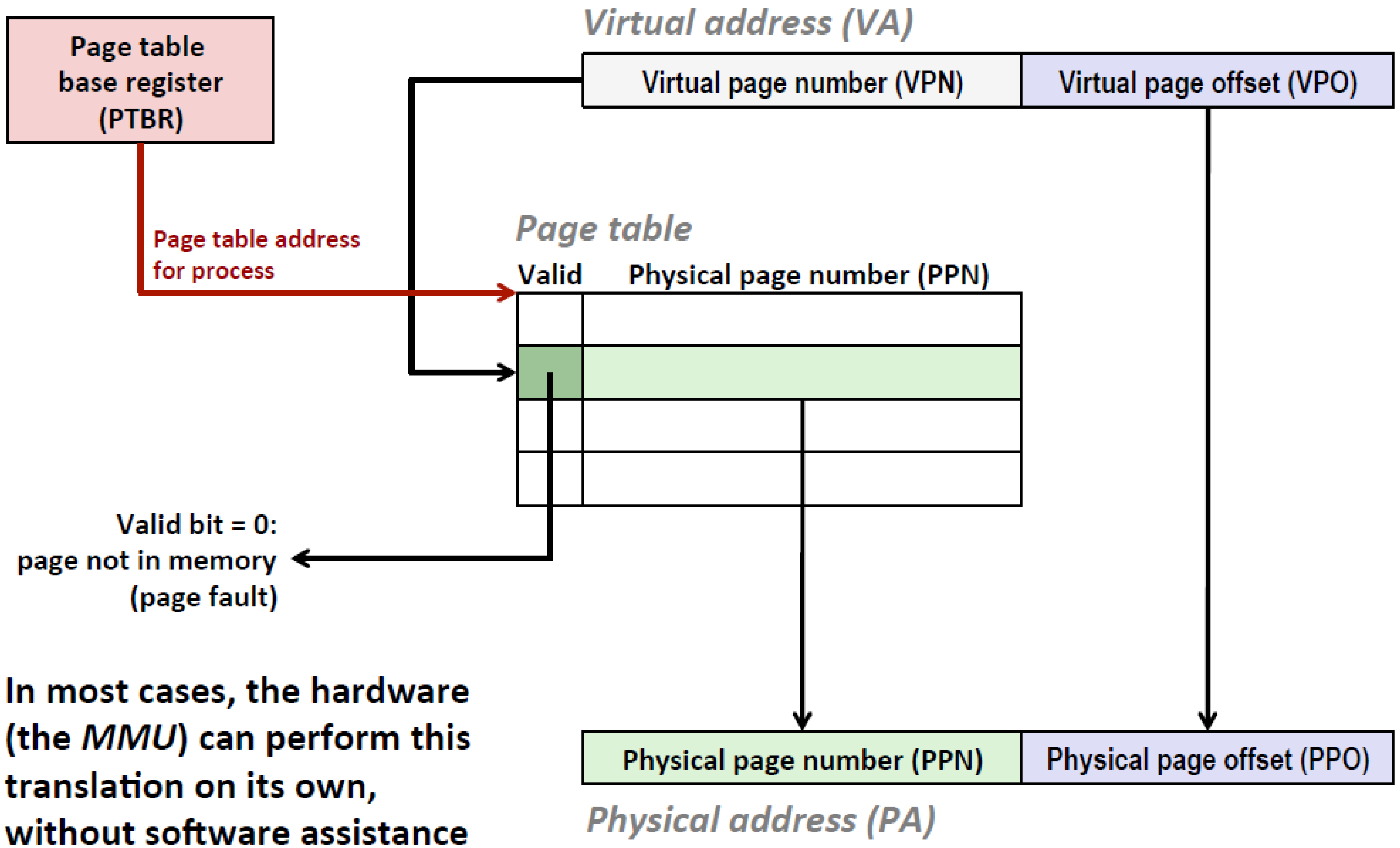
## Page Table Size

- 16KB virtual address space: 14-bit virtual address
- 4KB physical address space: 12-bit physical address
- Page size = 64 bytes

- How many virtual pages? 16KB/64B = 256
- How many physical pages? 4KB/64B = 64
- How many bits for physical page ID? 64 pages = $2^6$ require 6 bits

- Each page table entry: 1 valid bit + 6 physical page no bits = 7 bits
- 256 virtual pages ➔ 256 page table entries
- Total page table size = 256 x 7 bits = 1,792 bits = 224 bytes

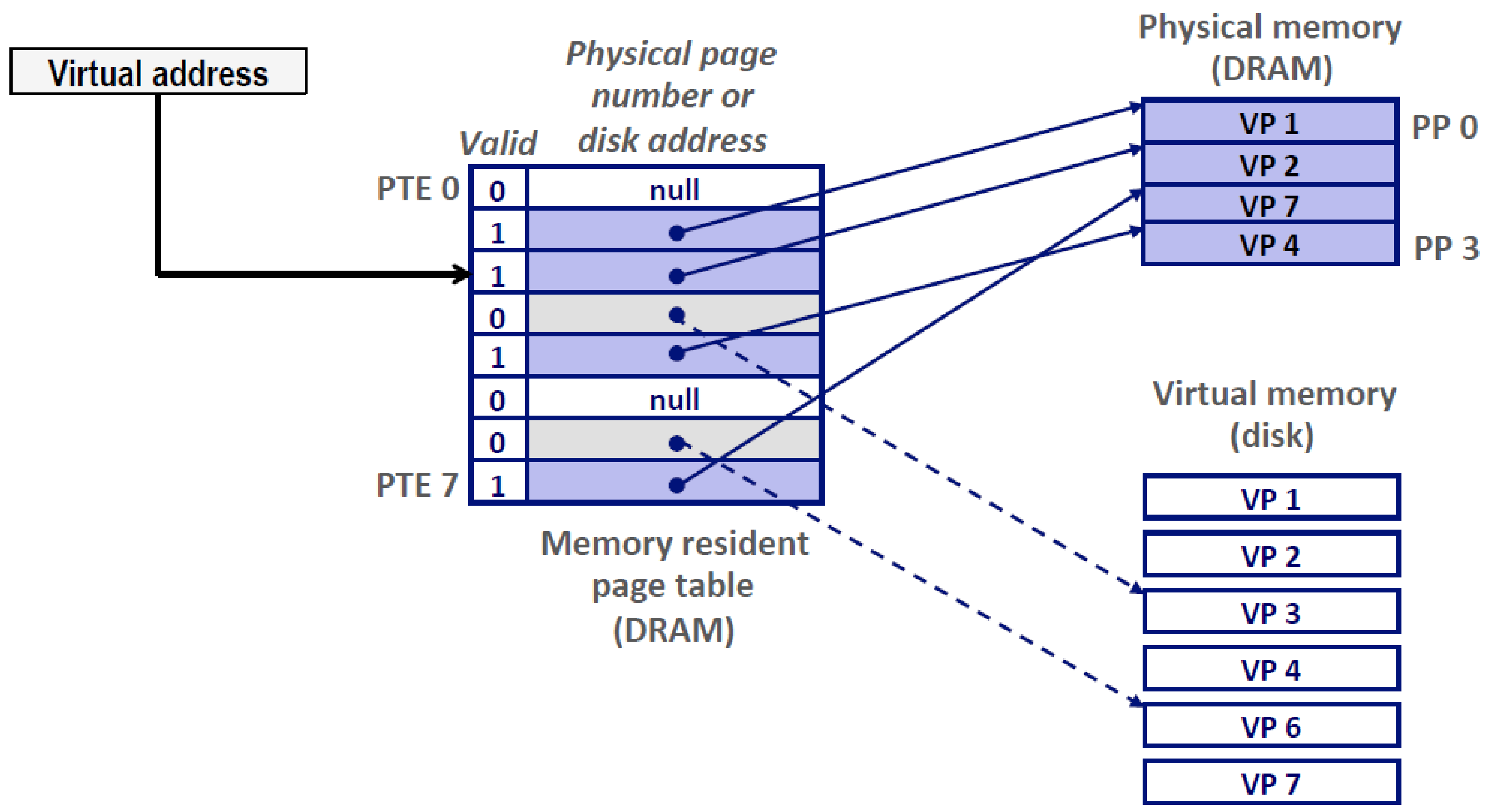
# Address Translation with a Page Table


Page table base register (PTBR)
Page table address for process
Virtual address (VA)
Virtual page number (VPN)
Virtual page offset (VPO)
Page table
Valid
Physical page number (PPN)
Valid bit = 0: page not in memory (page fault)
Physical page number (PPN)
Physical page offset (PPO)
Physical address (PA)

**In most cases, the hardware (the *MMU*) can perform this translation on its own, without software assistance**

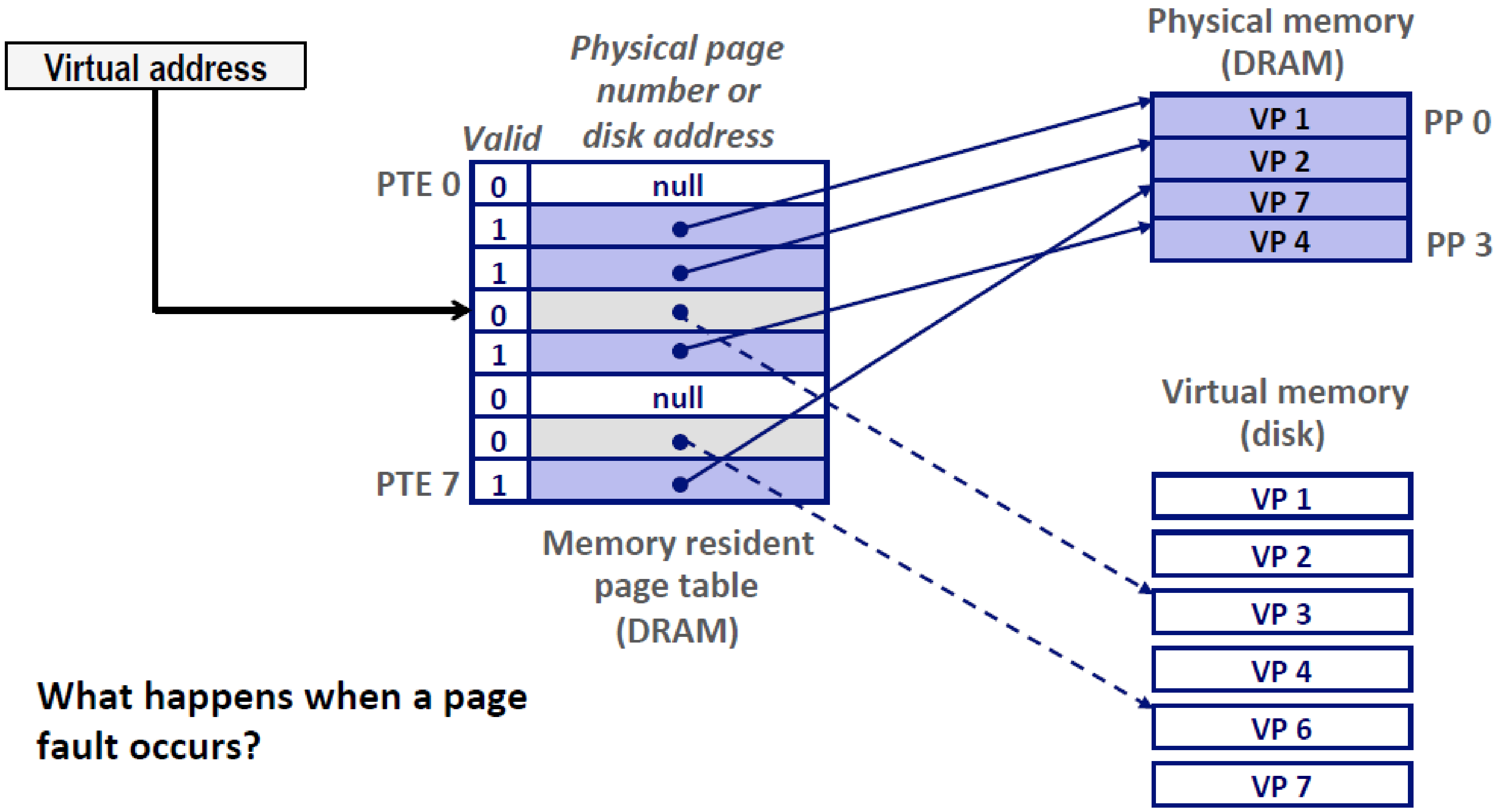
## Page Hit

- **Reference to virtual memory byte that is in physical memory**



# Page Fault

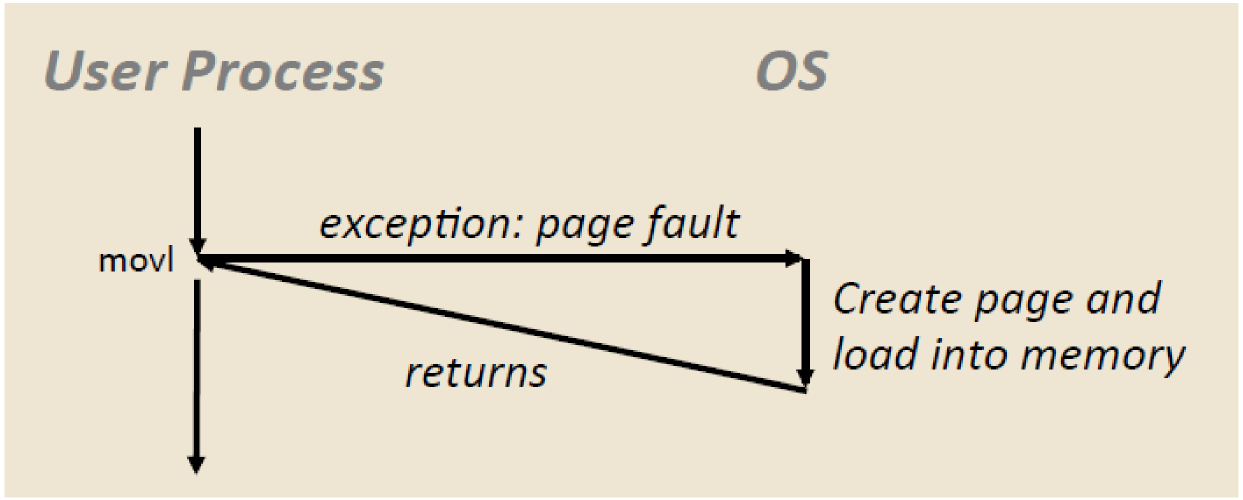
- **Reference to virtual memory byte that is NOT in physical memory**



**What happens when a page fault occurs?**

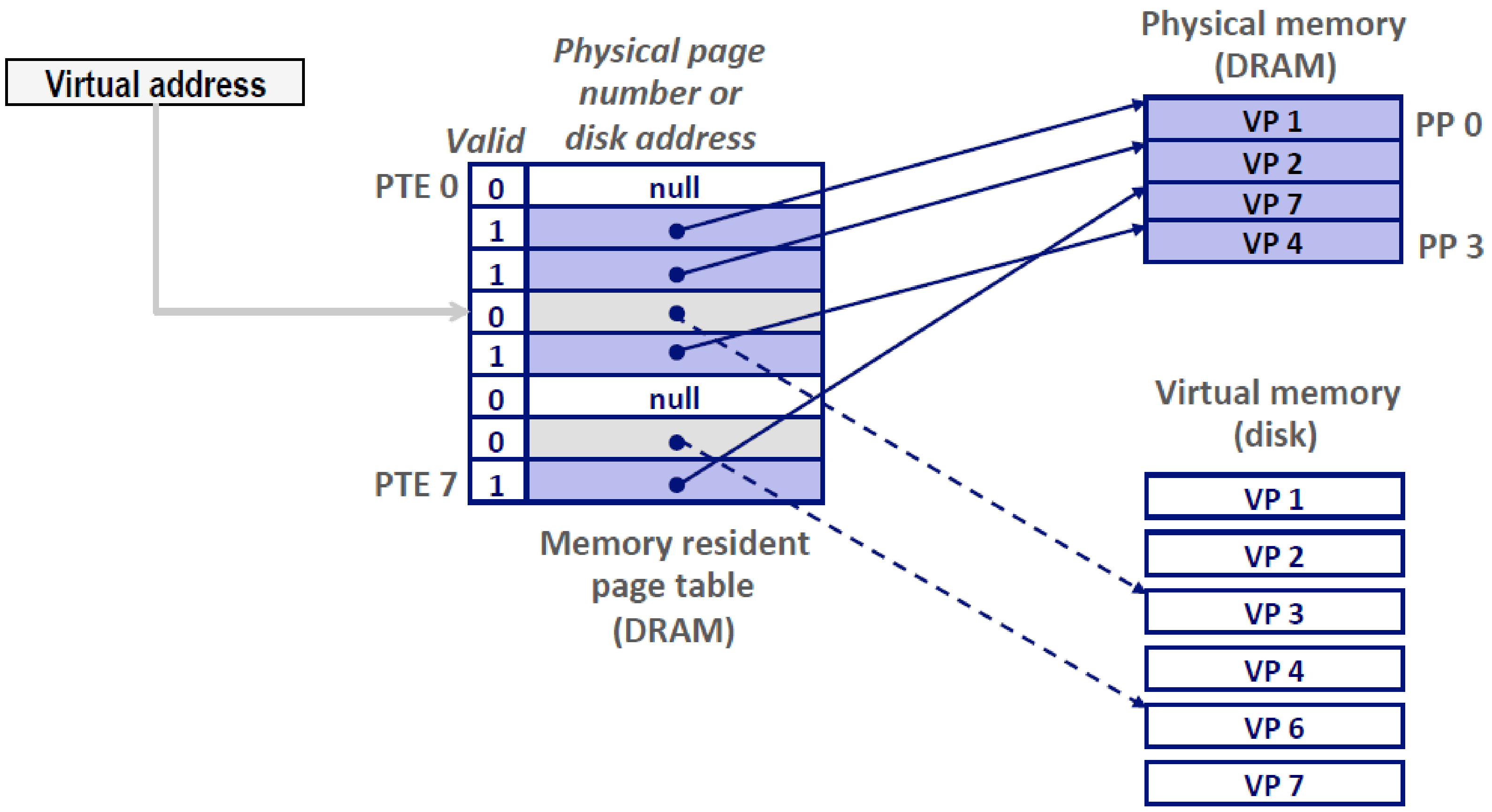
## Handling Page Fault

- User accesses a memory location
- That portion (page) of user's memory is currently on disk
- Page fault (page miss) raises an exception (similar to interrupt)
- OS page handler must load page from disk to physical memory
- Modify page table entry corresponding to the virtual page
- Returns to faulting instruction
- Successful memory access on second try



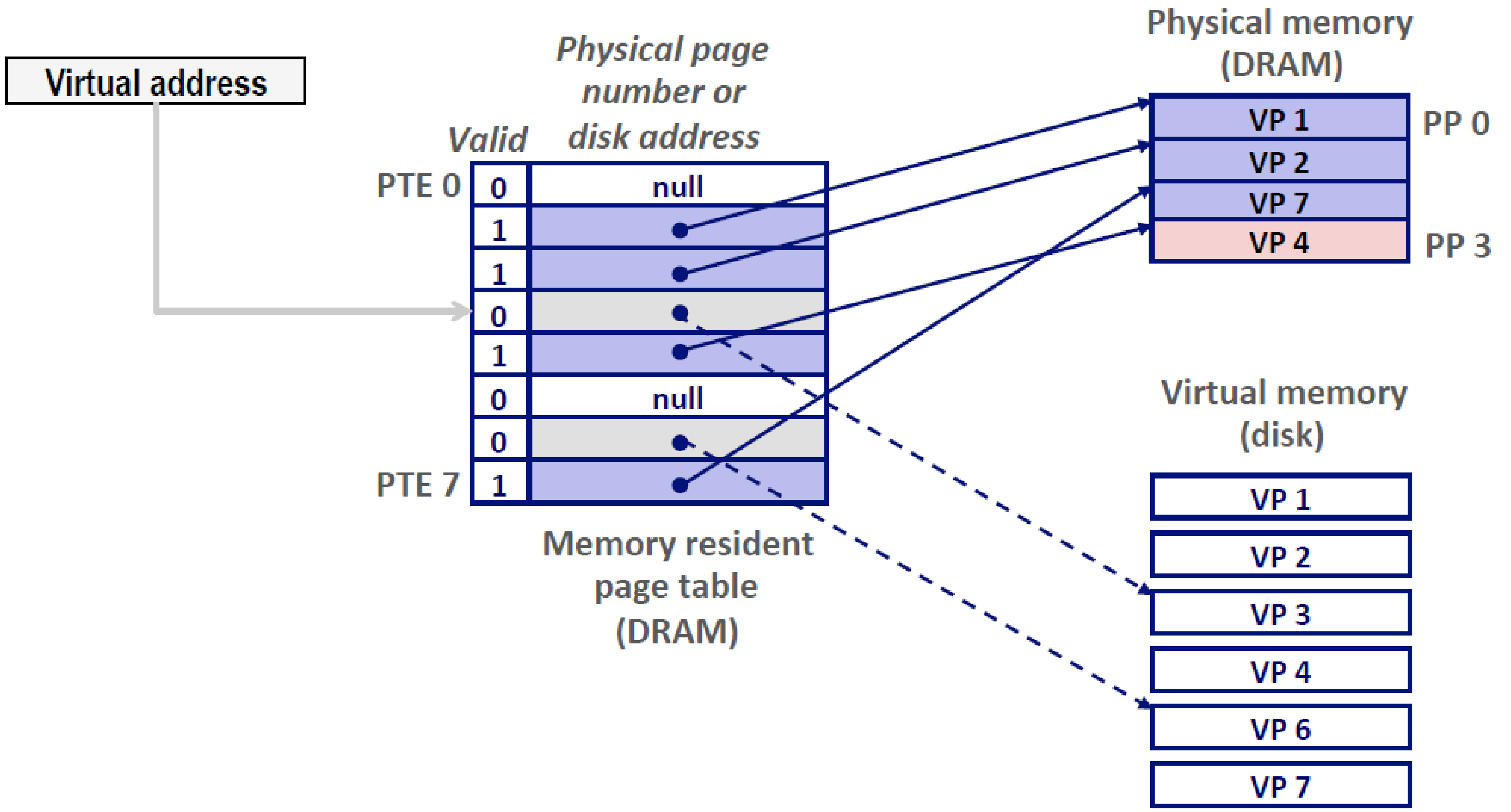
# Page Fault Example

- **Page fault causes an exception for Virtual Page Number (VP) 3**



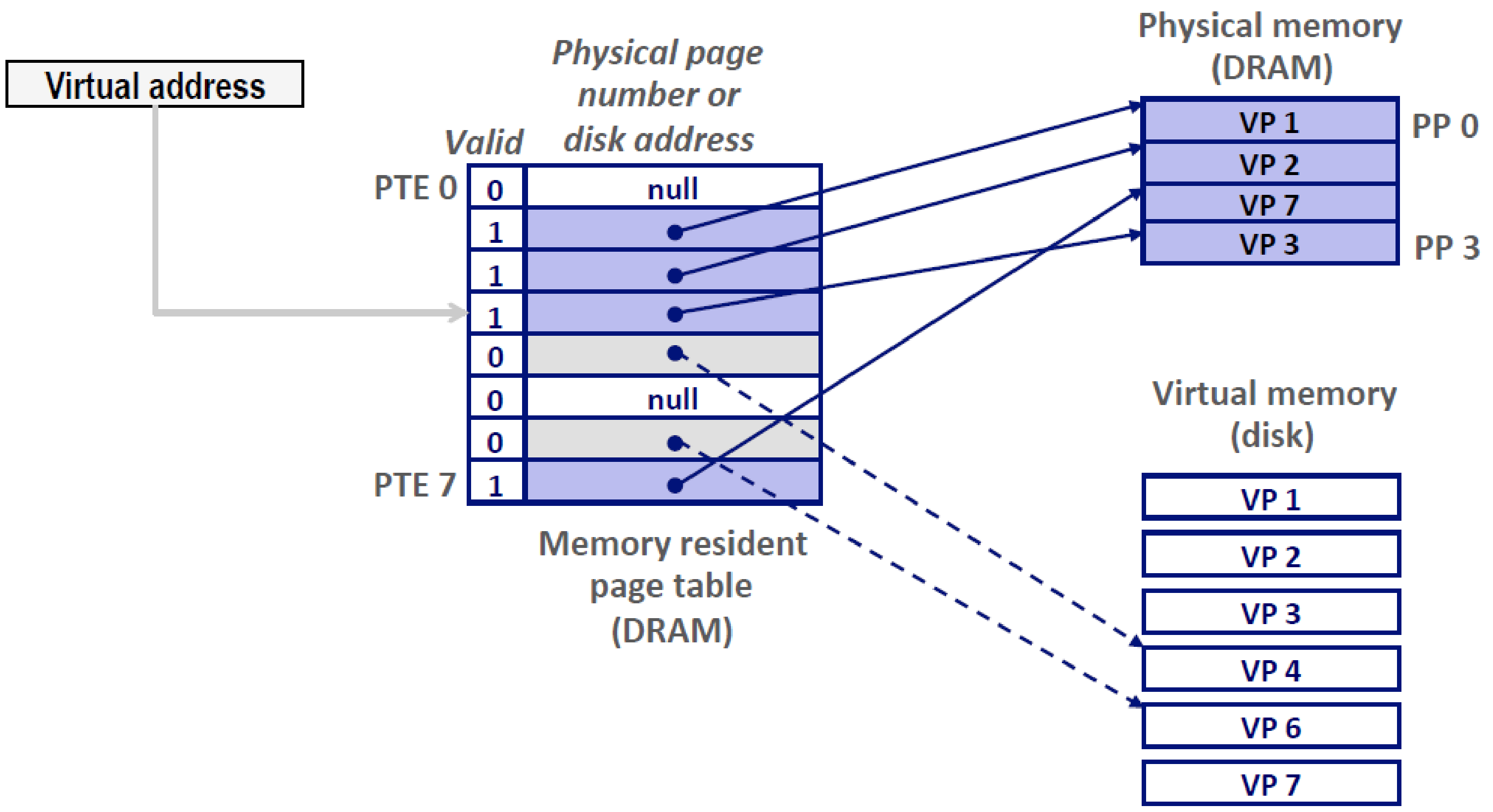
# Page Fault Example

- **Page fault causes an exception**
- **OS page fault handler selects a victim to be evicted (here VP 4)**



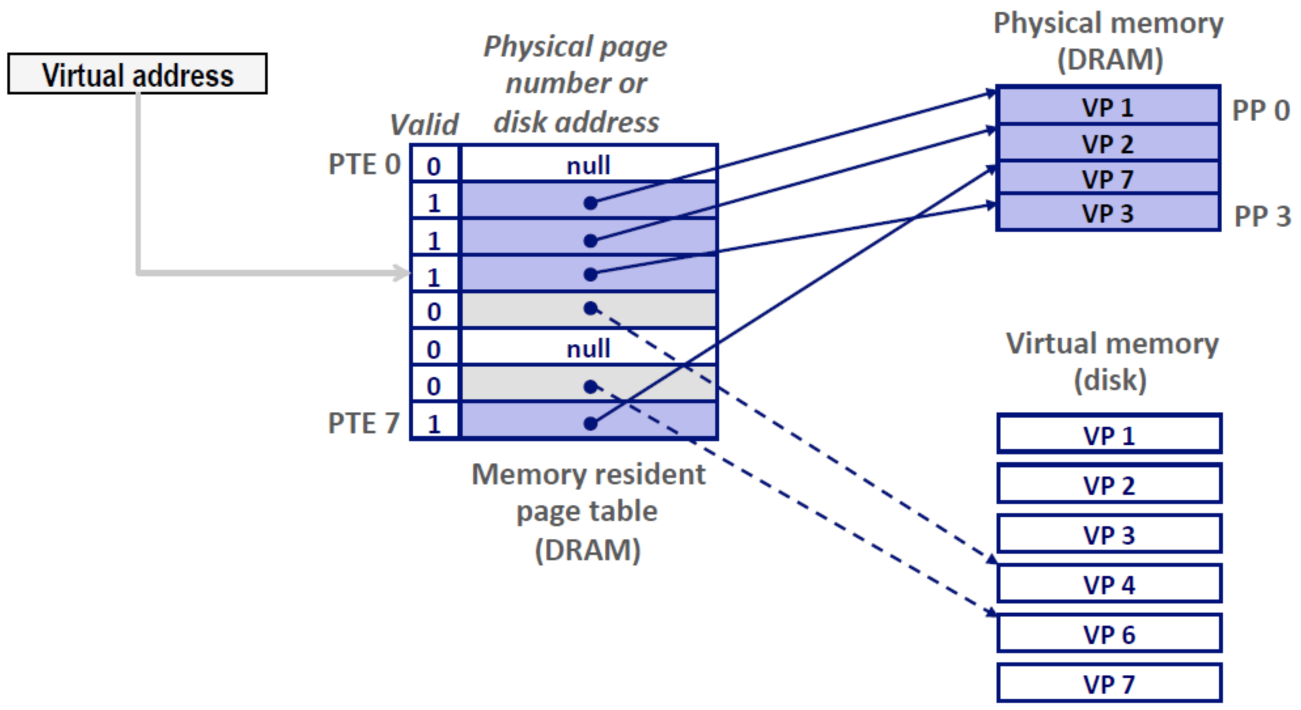
## Page Fault Example

- **Page fault causes an exception**
- **OS page fault handler selects a victim to be evicted (here VP 4)**
- **OS reads in VP3 from disk to memory in physical memory PP3 and modifies page table**



# Page Fault Example

- **Page fault causes an exception**
- **OS page fault handler selects a victim to be evicted (here VP 4)**
- **OS reads in VP3 from disk to memory in physical memory PP3 and modifies page table**
- **Offending instruction is restarted: page hit!**

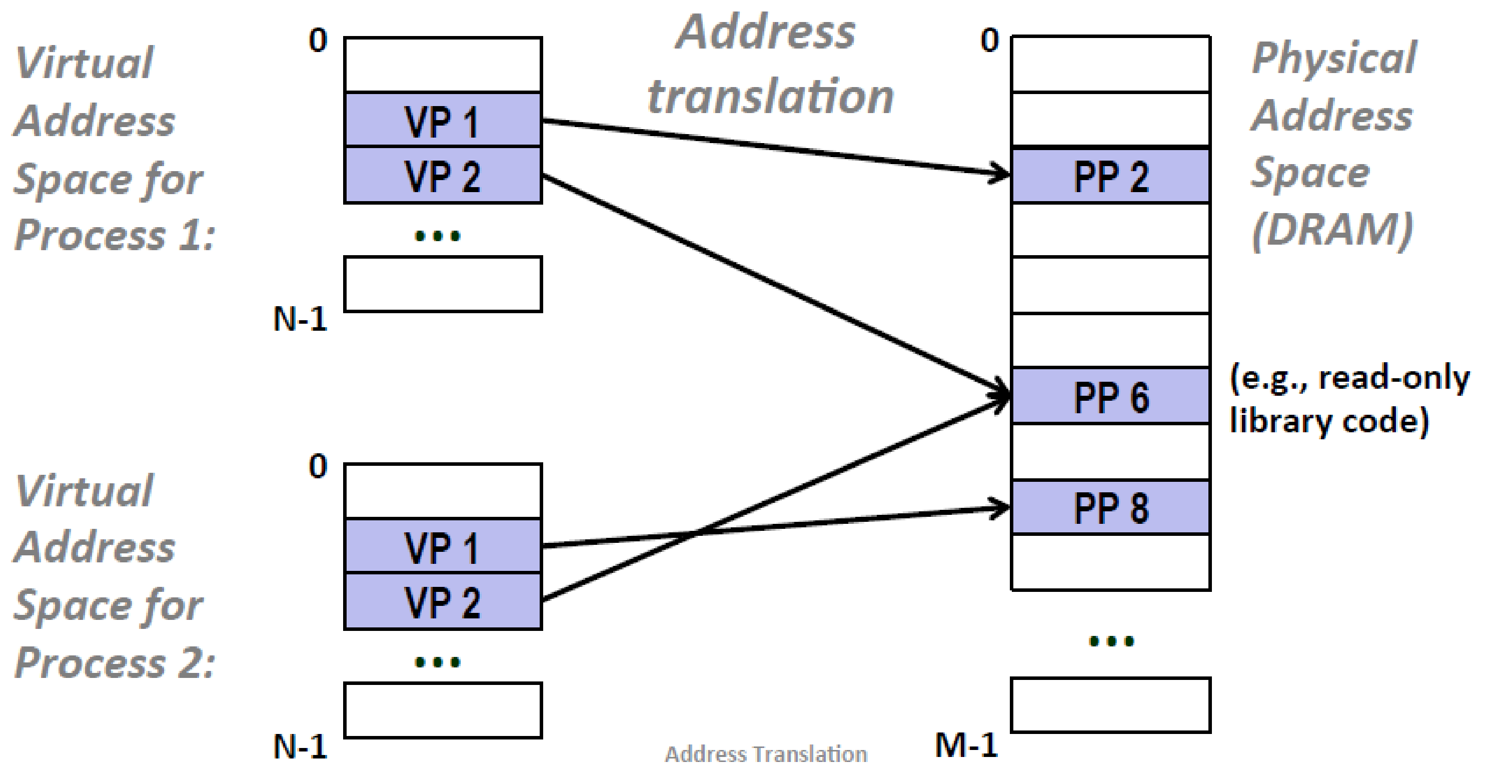


## Why does it work? Locality

- **Virtual memory works well because of locality**
  - **Spatial locality:** once an address is accessed, it is very likely that its neighbors will be accessed in the near future
  - **Temporal locality:** once an address is accessed, it is likely to be accessed again in the near future
- **The set of virtual pages that a program is "actively" accessing at any point in time is called its working set**
- **If (working set size < main memory size)**
  - Good performance for the process after initial compulsory misses
- **If (working set size > main memory size)**
  - **Thrashing:** Performance meltdown where pages are swapped (copied) in and out continuously

## VM for Managing Multiple Processes

- **Key abstraction: Each process has its own virtual address space**
  - It can view memory as a simple linear array
- **With virtual memory, this simple linear virtual address space need not be contiguous in physical memory**



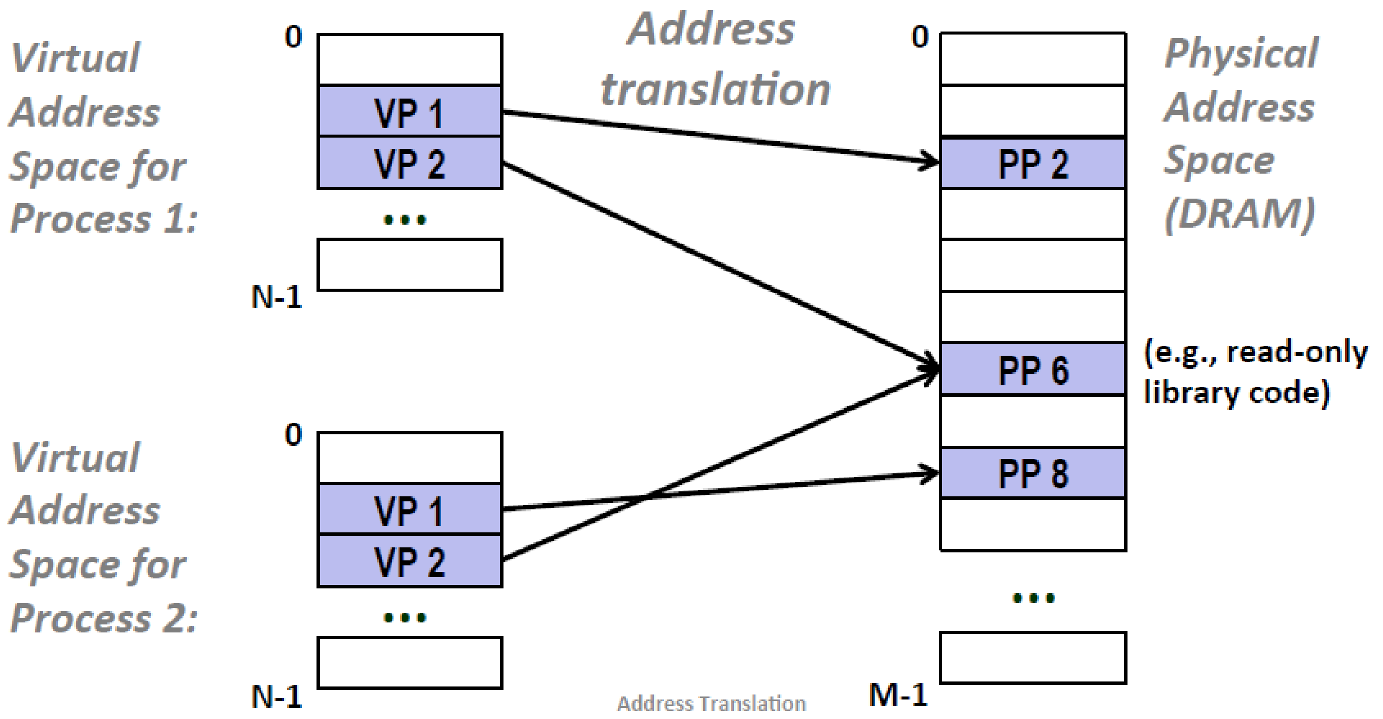
Virtual Address Space for Process 1:
0
VP 1
VP 2
...
N-1
Address translation
0
PP 2
PP 6
PP 8
...
M-1
Physical Address Space (DRAM)
(e.g., read-only library code)
Virtual Address Space for Process 2:
0
VP 1
VP 2
...
N-1
Address Translation

## Virtual Memory Advantages

- Virtual memory solves some of the issues with memory allocation in physical memory
- Virtual memory uses fixed page-size allocation: Internal Fragmentation if the process is not using the entire page
- Virtual memory does not require contiguous allocation in physical memory for a process: No external fragmentation

# VM for Protection and Sharing

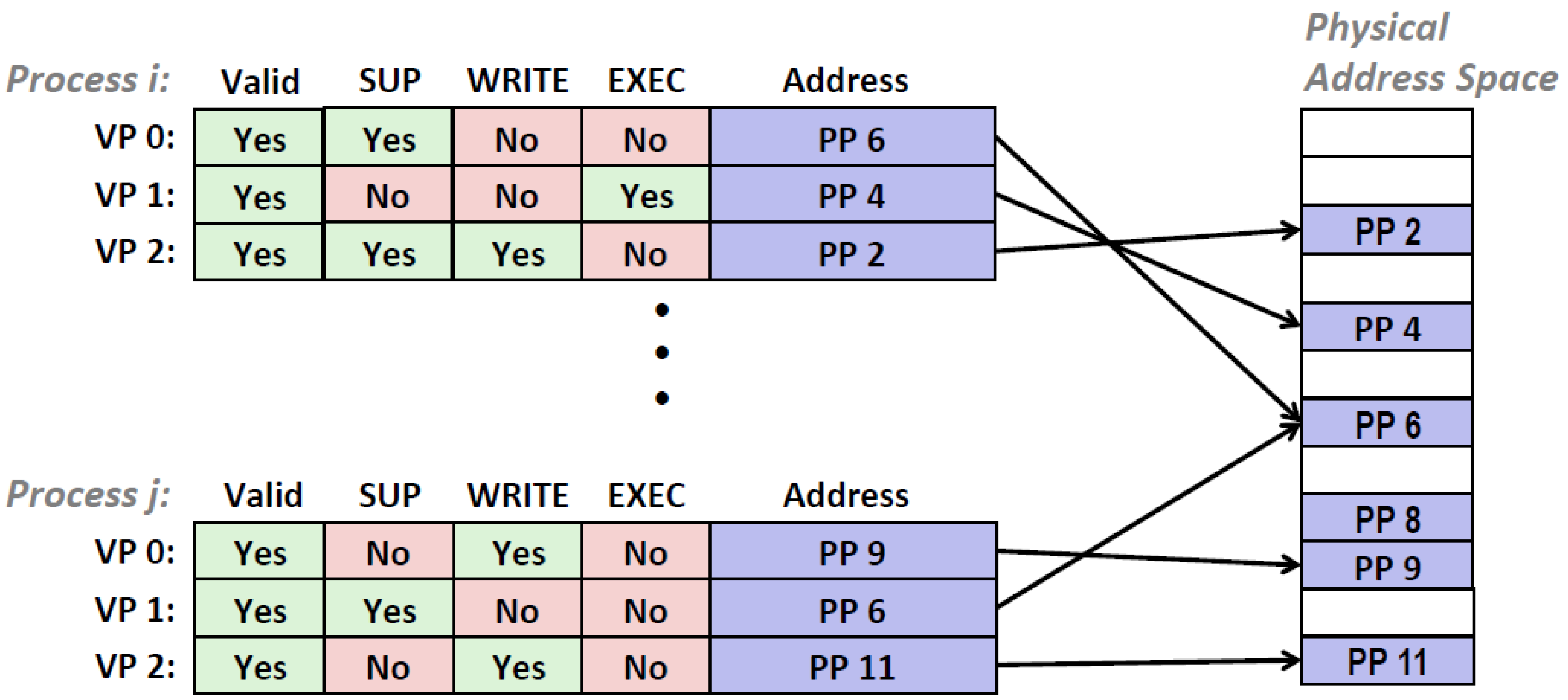
- **Mapping of VPs to PPs provides a simple mechanism for protecting memory and for sharing memory between processes**
  - Sharing: Map virtual pages in separate address spaces to the same physical page (here PP6)
  - Protection: a process cannot access physical pages it does not have mapping for (here Process 2 cannot access PP2)



Virtual Address Space for Process 1:
0
VP 1
VP 2
...
N-1
Address translation
0
PP 2
PP 6
PP 8
...
M-1
Physical Address Space (DRAM)
(e.g., read-only library code)
Virtual Address Space for Process 2:
0
VP 1
VP 2
...
N-1

## Memory Protection within a Single Process

- **Extend page table entries with permission bits**
- **MMU checks these permission bits on every memory access**
    - If violated, raises an exception and OS sends segmentation fault signal to the process



*Process i:*

| | Valid | SUP | WRITE | EXEC | Address |
|---|---|---|---|---|---|
| VP 0: | Yes | Yes | No | No | PP 6 |
| VP 1: | Yes | No | No | Yes | PP 4 |
| VP 2: | Yes | Yes | Yes | No | PP 2 |

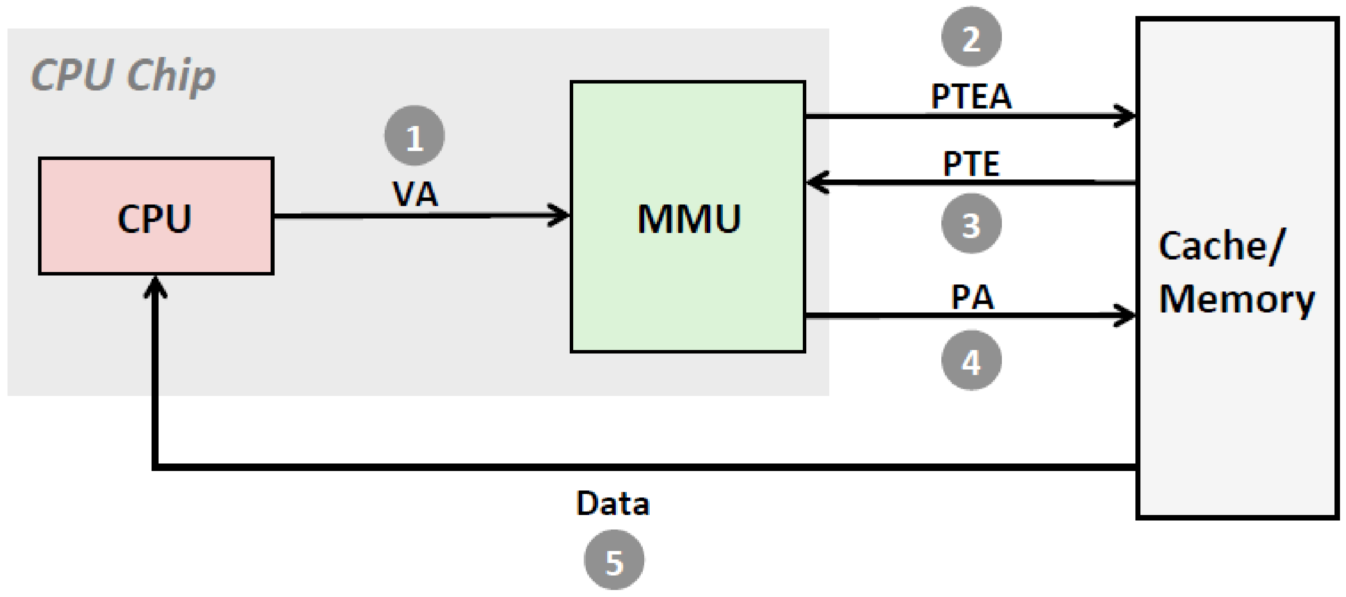
•
•
•

*Process j:*

| | Valid | SUP | WRITE | EXEC | Address |
|---|---|---|---|---|---|
| VP 0: | Yes | No | Yes | No | PP 9 |
| VP 1: | Yes | Yes | No | No | PP 6 |
| VP 2: | Yes | No | Yes | No | PP 11 |

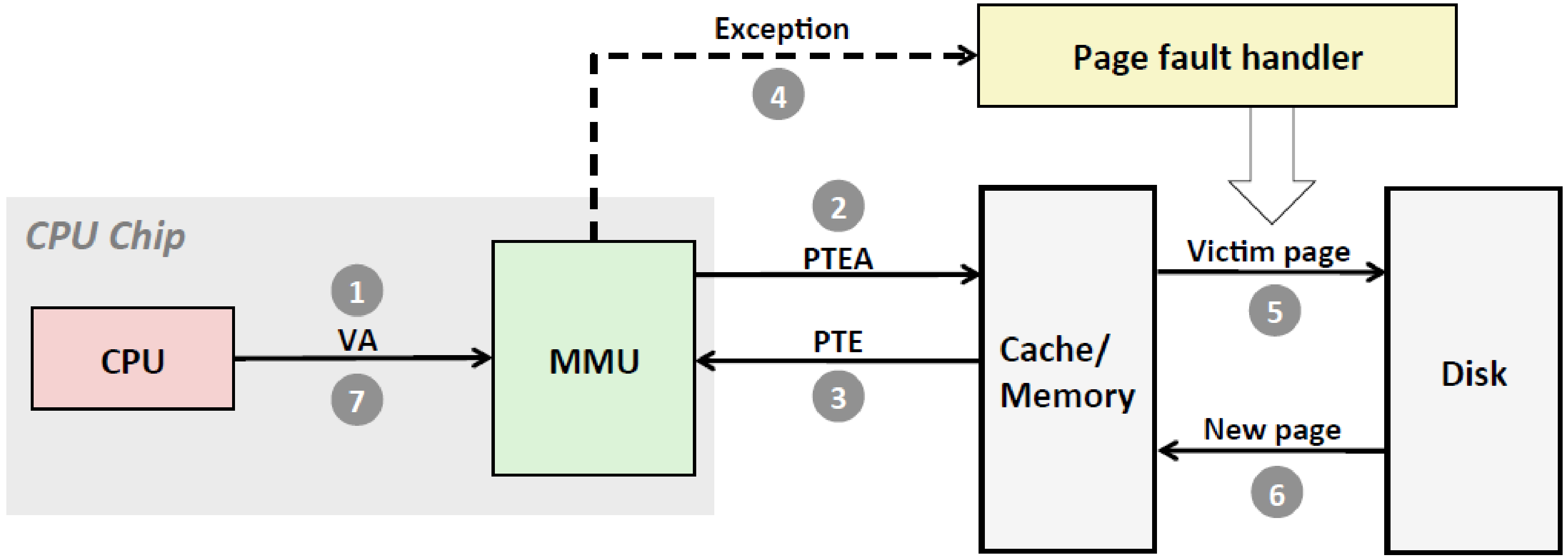
## Address Translation: Page Hit

- (1) Processor sends virtual address to MMU (memory management unit)
- (2-3) MMU fetches page table entry (PTE) by accessing the page table stored in memory
- (4) MMU sends physical address to memory
- (5) Memory sends data word to processor



# Address Translation: Page Fault

- (1) Processor sends virtual address to MMU
- (2-3) MMU fetches PTE from page table
- (4) valid bit is zero, 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 faulty instr.



# Hmm…Translation Sounds Slow!!

- **MMU accesses memory twice: once to first get the PTE for translation and then again for the actual memory request from the CPU**
  - The PTE may be cached like any other memory word
  - But they may be evicted by other data references
  - And hit in the cache still requires 1-3 cycles

- **What can we do to make translation faster?**

## Speeding up Translation with TLB

- **Solution: Add another cache**
- **Translation Lookaside Buffer (TLB)**
  - Small hardware cache inside MMU
  - Maps virtual page numbers to physical page numbers
  - Contains page table entries for small number of virtual pages
  - Modern Intel processors: 128 or 256 entries in TLB
  - Works because of locality

## TLB Example

- TLB contains only a subset of page table entries
- Both Virtual Page number and Physical Page number are maintained
- Search all the TLB entries in parallel (associative search)

| | PP | Valid |
|---|---|---|
| 0 | 0x10 | 1 |
| 1 | DISK | 0 |
| 2 | 0x23 | 1 |
| 3 | 0x15 | 1 |
| 4 | DISK | 0 |
| 5 | 0x1B | 1 |
| 6 | 0x2E | 1 |
| ... | ... | ... |
| 255 | 0x2A | 1 |

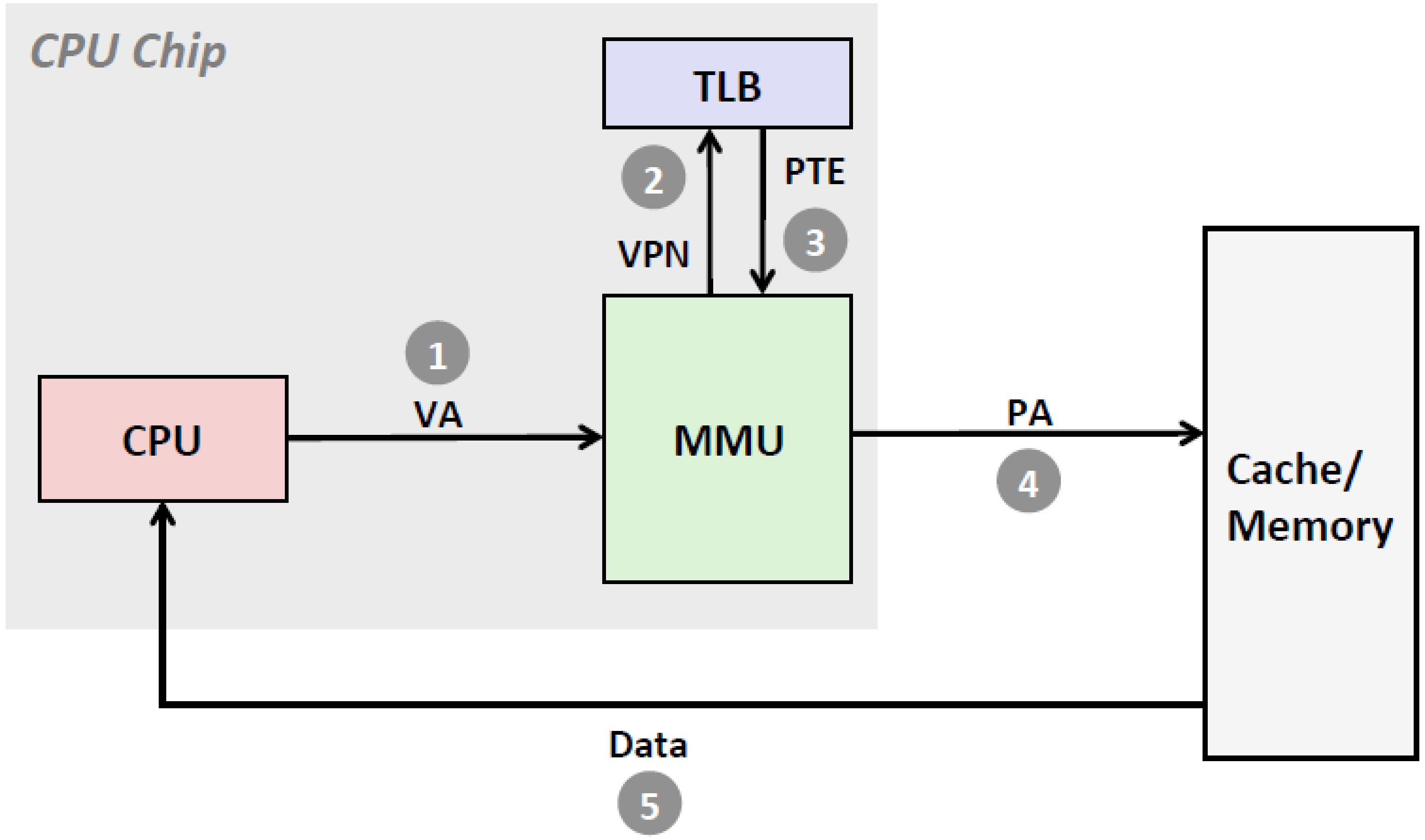
Full Page Table in DRAM

| VP | PP | Valid |
|---|---|---|
| 0x06 | 0x2E | 1 |
| - | - | 0 |
| 0x02 | 0x23 | 1 |
| - | - | 0 |

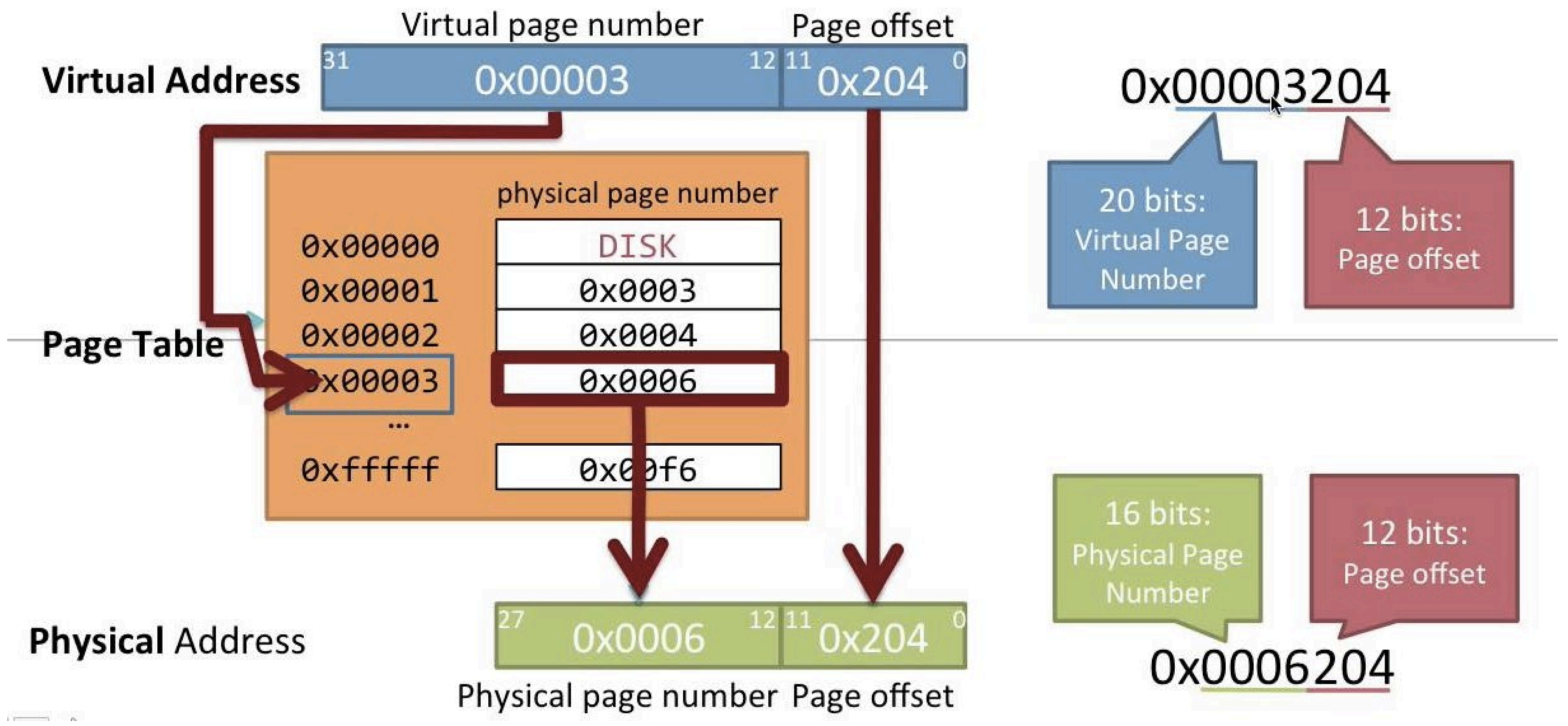
On-chip TLB

## TLB Hit

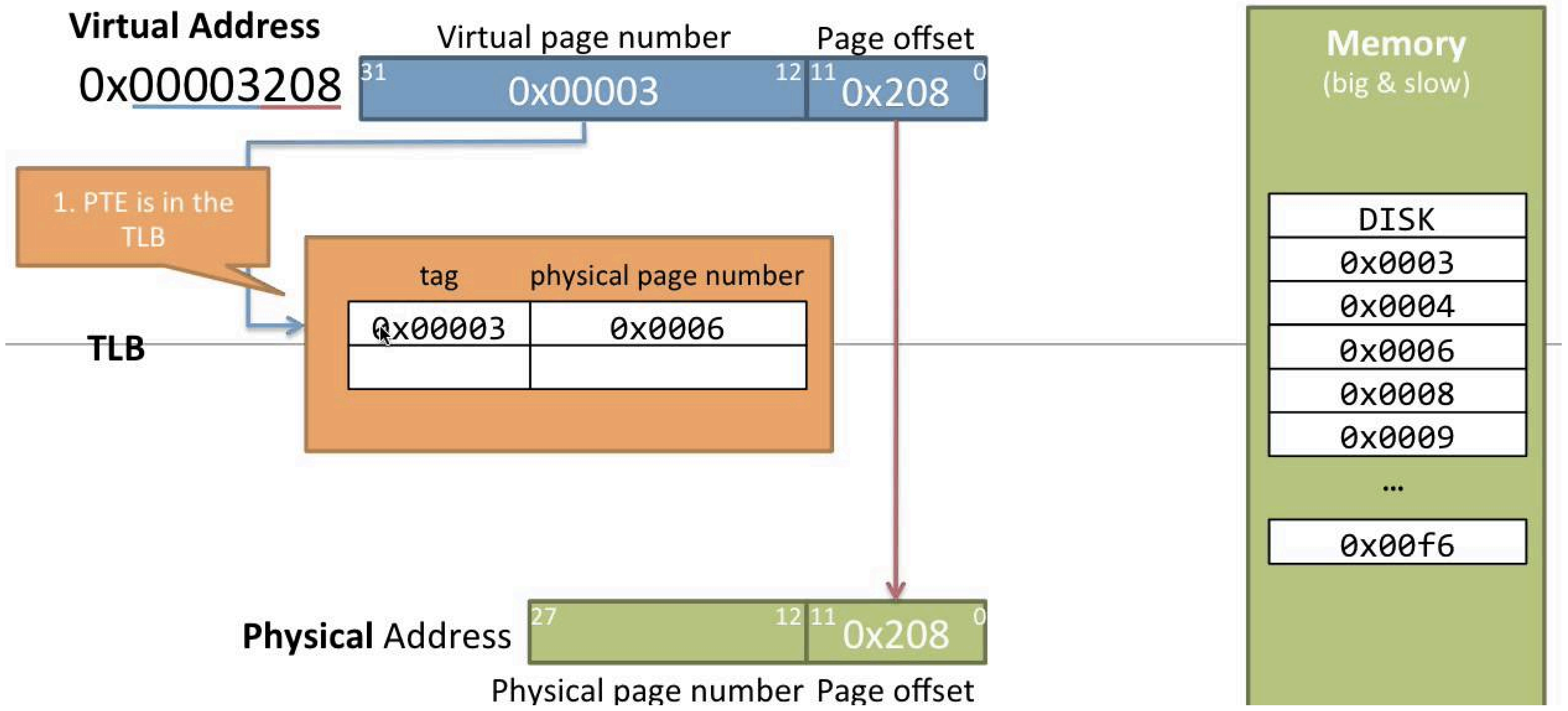
- **A TLB hit eliminates costly memory access for page table entry**



## Translation with Page Table in DRAM

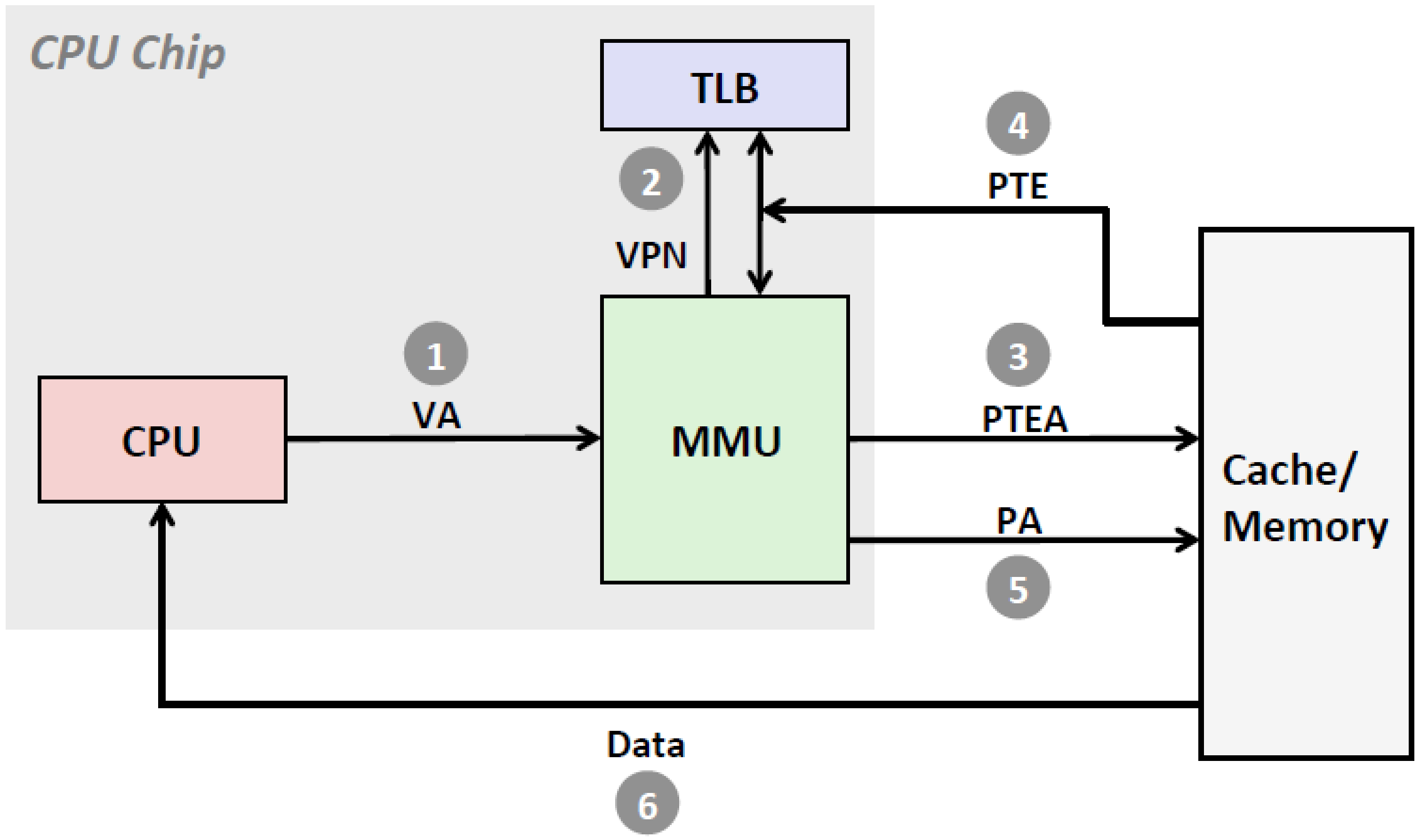


## Translation with TLB Hit

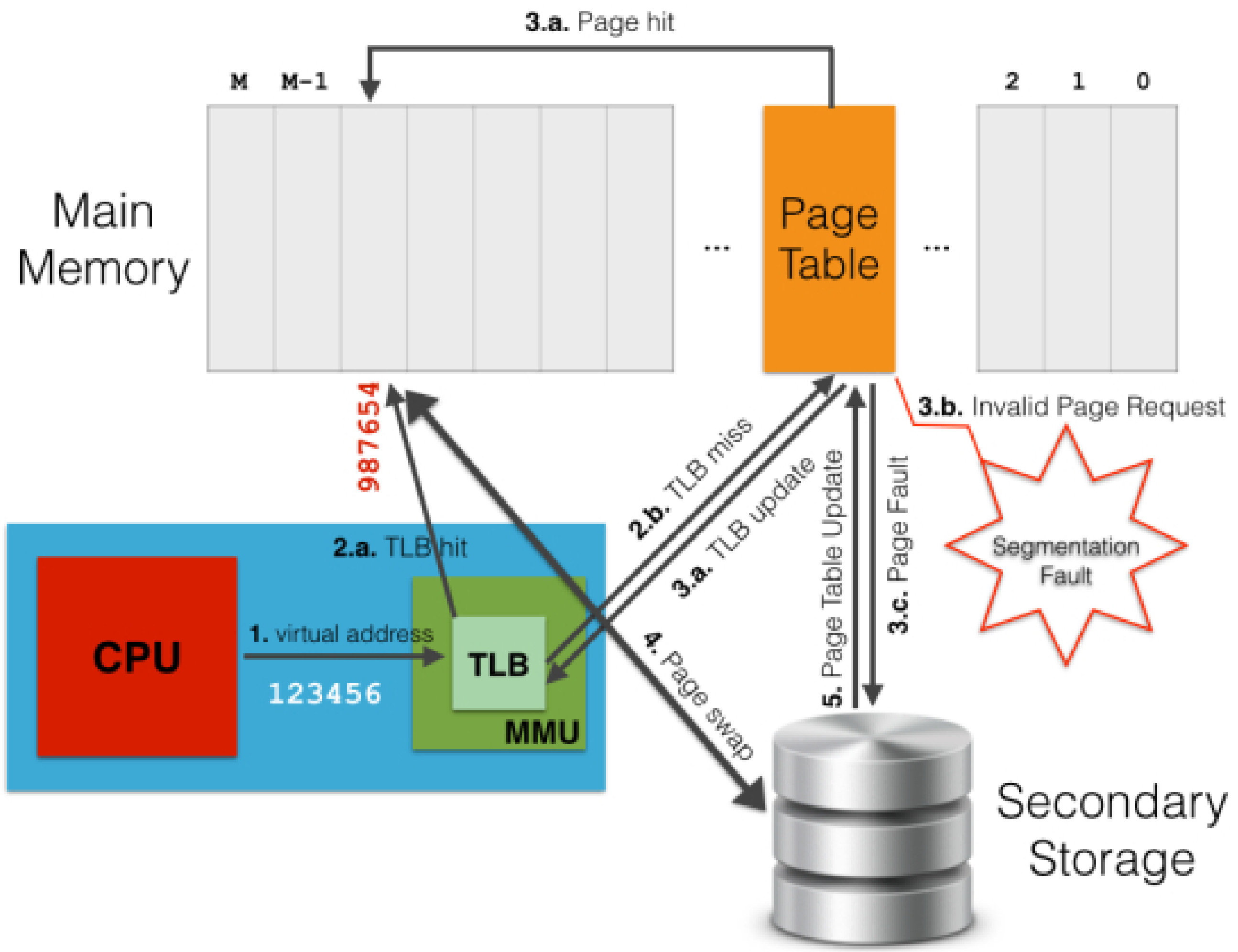


# TLB Miss

- **A TLB miss incurs additional memory access for the PTE**
  - Fortunately TLB misses are rare



# Putting it all together

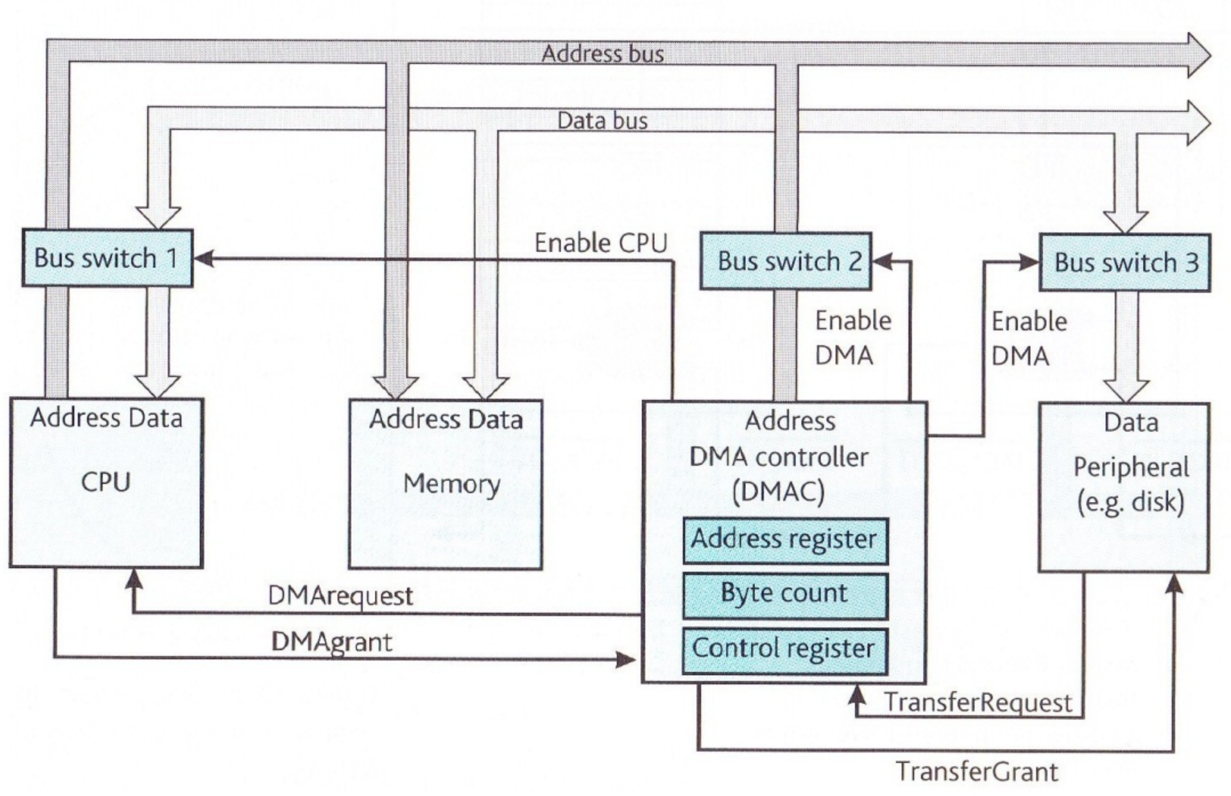


## Direct Memory Access

- Direct Memory Access (DMA) moves data between a peripheral and the CPU's memory without the direct intervention of the CPU itself.
- Provides the fastest possible means of transferring data between an interface and memory.
- It requires no CPU overhead and leaves the CPU free to do useful work.
- Helps to increase data throughput.

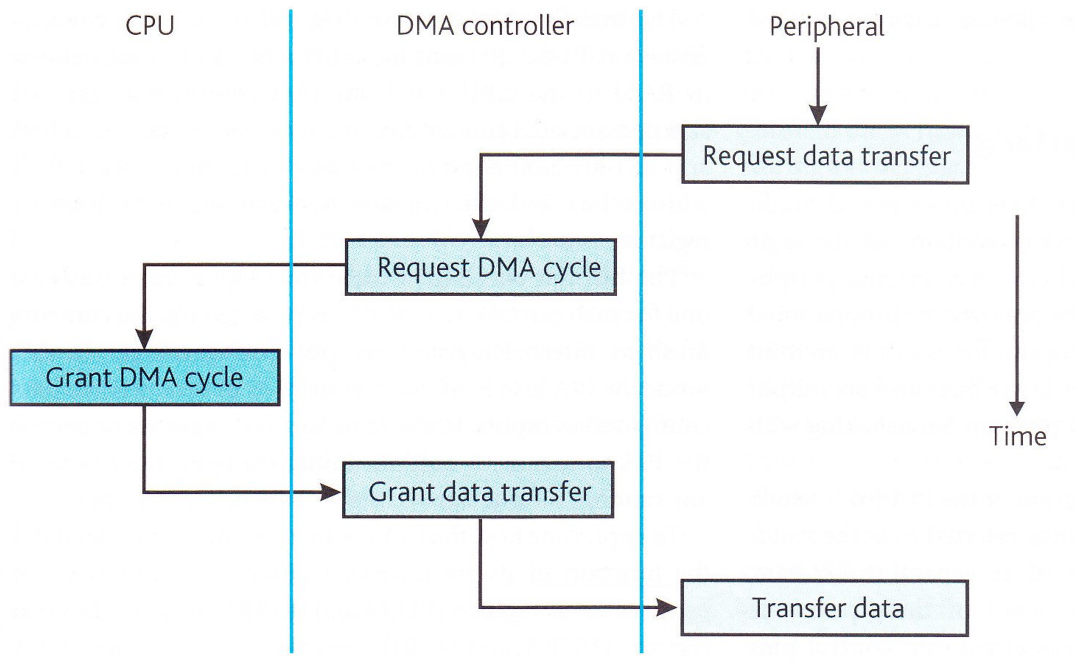
## Direct Memory Access

- During normal operation of the computer, bus switch 1 is closed and bus switches 2 and 3 are open.
- The CPU control the buses, providing an address on the address bus and reading and writing data from memory via the data bus.



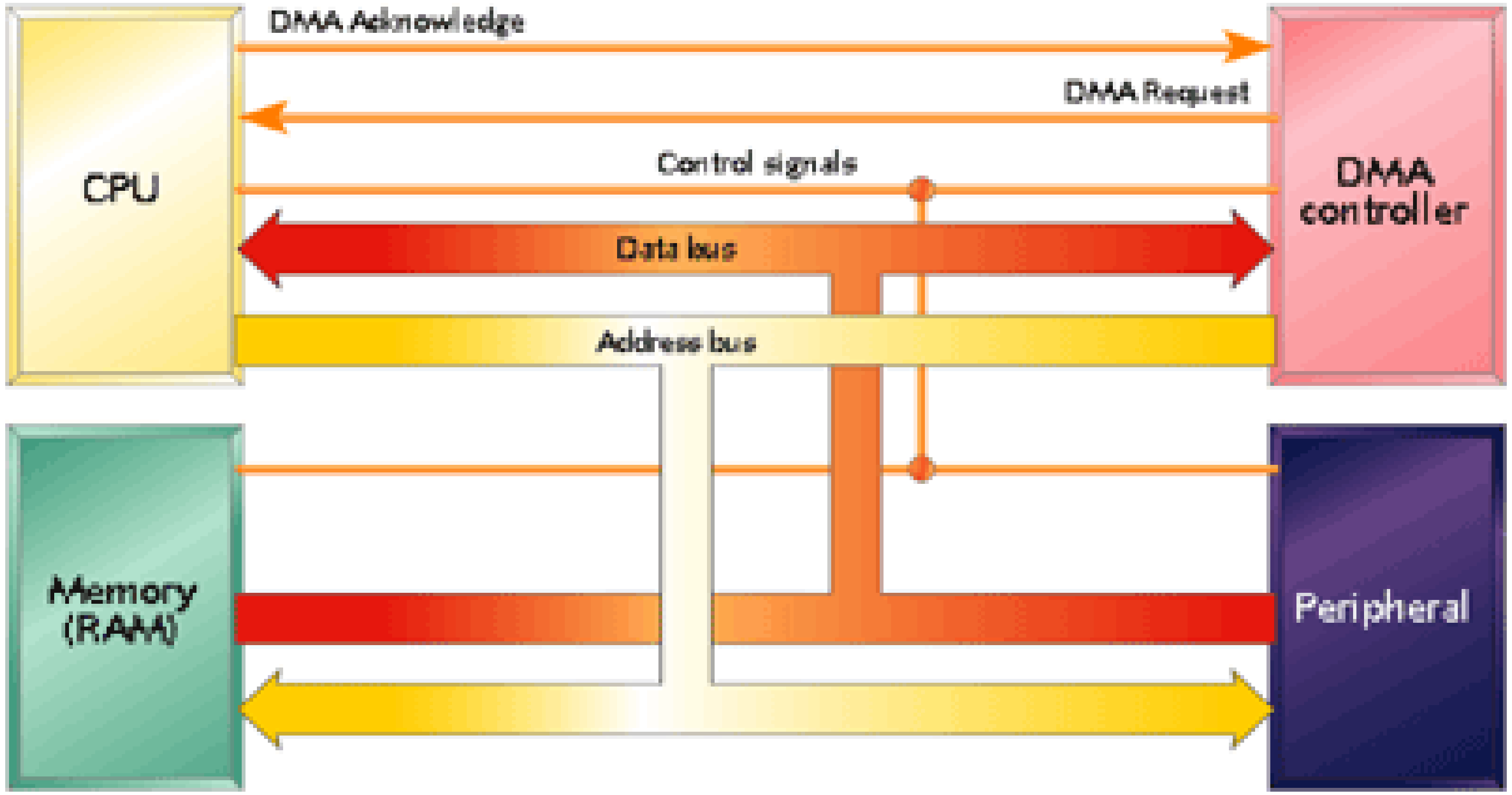
## Direct Memory Access

- When a peripheral wishes to take part in an I/O transaction it asserts the TransferRequest input of the DMA controller (DMAC).
- In turn, the DMAC controller asserts the DMArequest (BusReq) to request control of the buses from the CPU.
- When the CPU returns DMAgrant (BusAck) to the DMAC, a DMA transfer takes place.
- Bus switch 1 is open and switches 2 and 3 are closed.
- The DMAC provides TransferGrant signal to the peripheral.
- The peripheral is able to communicate with the memory using the DMA
- When DMA operation is completed, DMAC hands control of the bus back to the CPU.



## Direct Memory Access

- It is not necessary for the DMA to be initiated by the Peripheral.
- It can be initiated within the Processor itself.

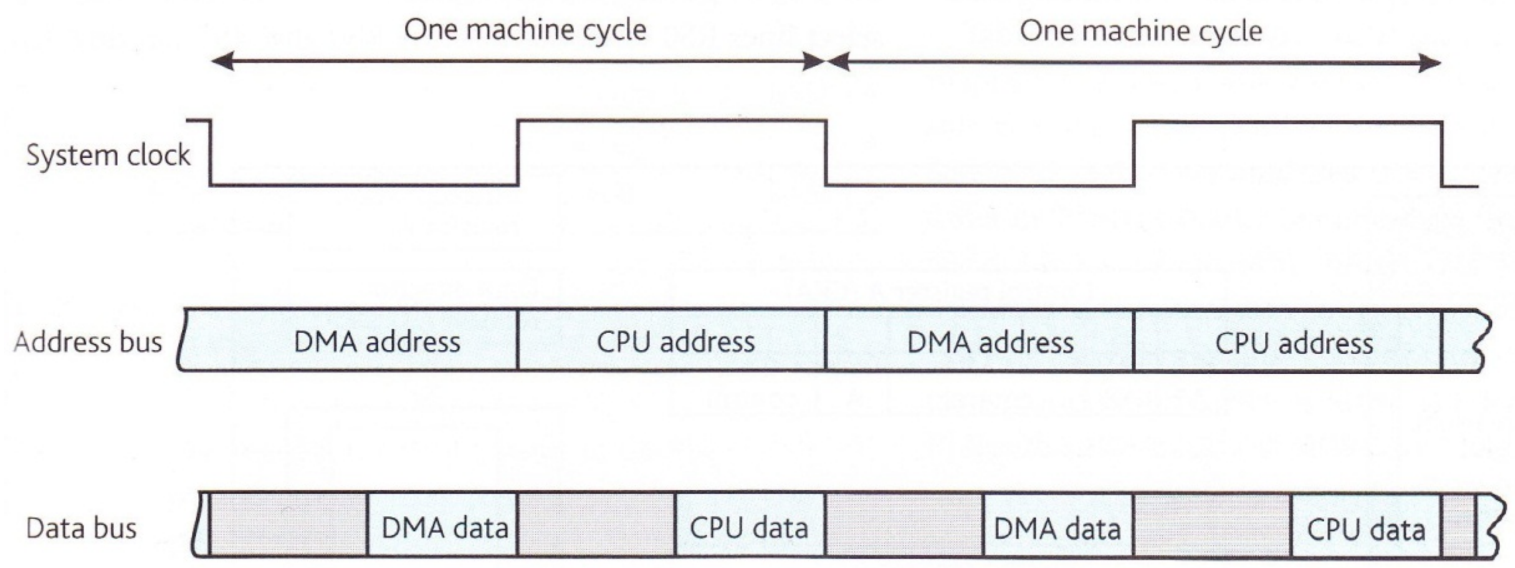


## Direct Memory Access

- Many DMACs are able to handle several interfaces, which requires their registers to be duplicated.
- Each interface is referred to as a channel.
- DMA operates in one of two modes: burst mode or cycle stealing
- Burst Mode
  - DMA controller seizes the system bus for the duration of the data transfer operation.
  - Allows data to be moved at the fastest possible rate within the memory/bus/interface timing requirements
  - CPU is effectively halted in the burst mode because it cannot use its buses.

## Direct Memory Access

- Cycle Steal Mode
  - DMA operations are interleaved with the processor's normal memory accesses.
  - DMA can take place when CPU is busy.
  - Transparent because transfer is invisible to the processor and no processing time is lost.



## Summary

- **Programmer's view of virtual memory**
  - Each process has its own private linear address space
  - Cannot be corrupted by other processes

- **System view of virtual memory**
  - Uses memory efficiently be caching virtual memory pages
  - Efficient only because of locality
  - Simplifies memory management and sharing
  - Simplifies protection by proving a convenient intermediate point to check permission

- **DMA**
  - A means to free up the CPU to allow memory access to be handled by the HW

## The End!

- We have completed the module! Hooray! ☺
- You have obtained extensive experience and knowledge in both Embedded Systems Development together with key RTOS concepts.
- If you are keen to be a TA for this module next semester, please apply and let me know!
- We appreciate your feedback, which helps us improve
- We look forward to not just feedback for this module, but also feedback for the CEG programme.
- Thank You and Good Luck for the Exams! ☺