Module 4. Memory Hierarchy
Virtual Memory

Taken from:

Patterson, D., Hennessy, J. L. “Computer Organization and Design. The hardware/software interface” , 5th ed. Section 5.7

Hennessy, J. L., Patterson, D. “Computer Architecture: A Quantitative Approach”, 5th ed. Chapter 2 and Apendix B.4,5

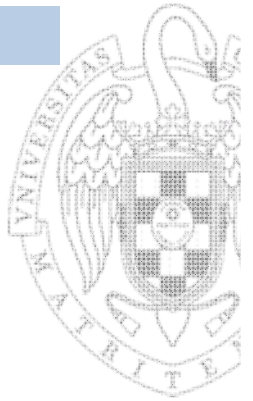
Harris, S.L and Harris, D.M. "Digital Design and Computer Architecture. ARM Edition“, Section 8.4

Bhattacharjee, A. and Lustig, D. “Architectural and Operating System Support for Virtual Memory”, Synthesis Lectures on Computer Architecture

Module 4.2. Virtual memory

- Introduction
  - What is virtual memory?
  - Why virtual memory is used
  - Virtual memory and cache
- Basic VM implementation: paging
  - Address translation
  - Page tables
  - Making address translation fast: TLB
- Integration of VM and caches
  - PIPT
  - VIVT
  - VIPT
- DMA and VM

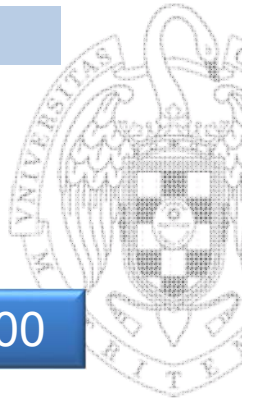
CO



What is virtual memory?

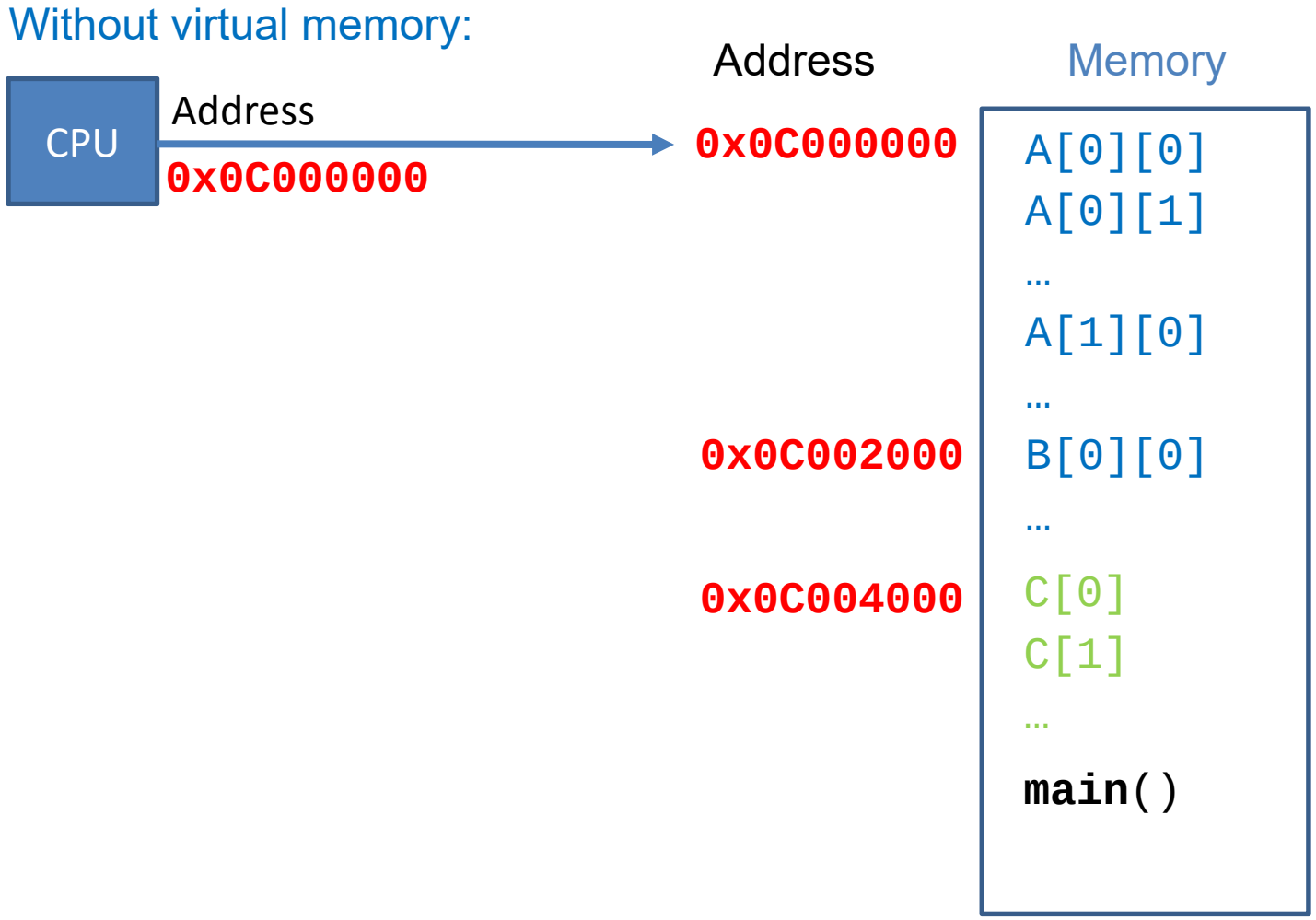
- Virtual memory is an **abstraction of the storage** resources (main memory) that are actually available on a given machine.
- Programs perform memory accesses using only virtual addresses, and the processor and operating system work together to translate those virtual addresses into physical addresses that specify where the data is actually stored.

What is virtual memory? (2)



Reading A[0][0], which is stored in main memory at address 0x0C000000

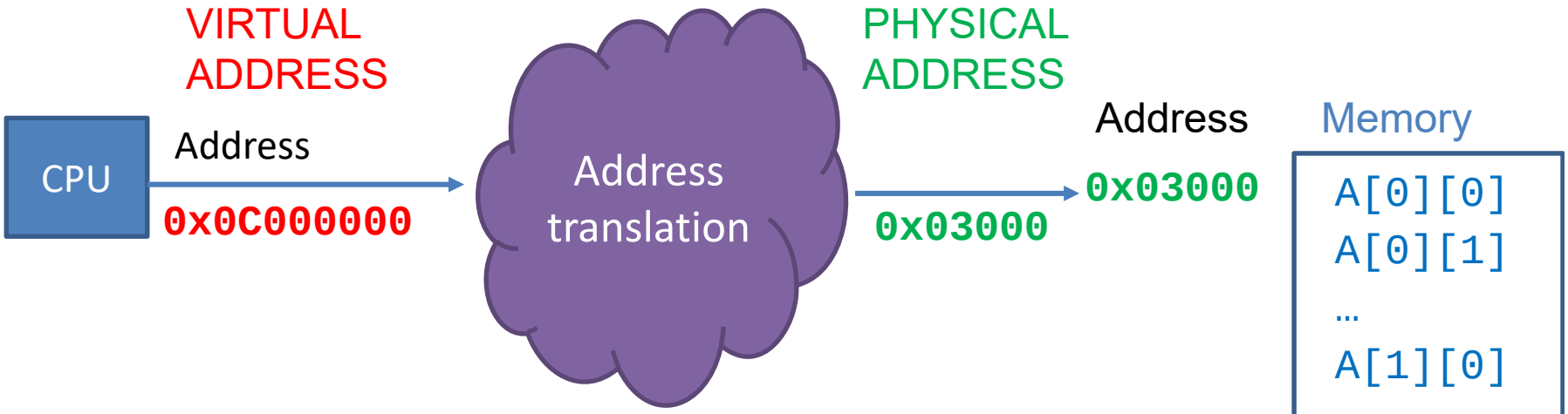
Without virtual memory:



What is virtual memory? (3)

Reading A[0][0], which is stored in main memory at address 0x0C000000

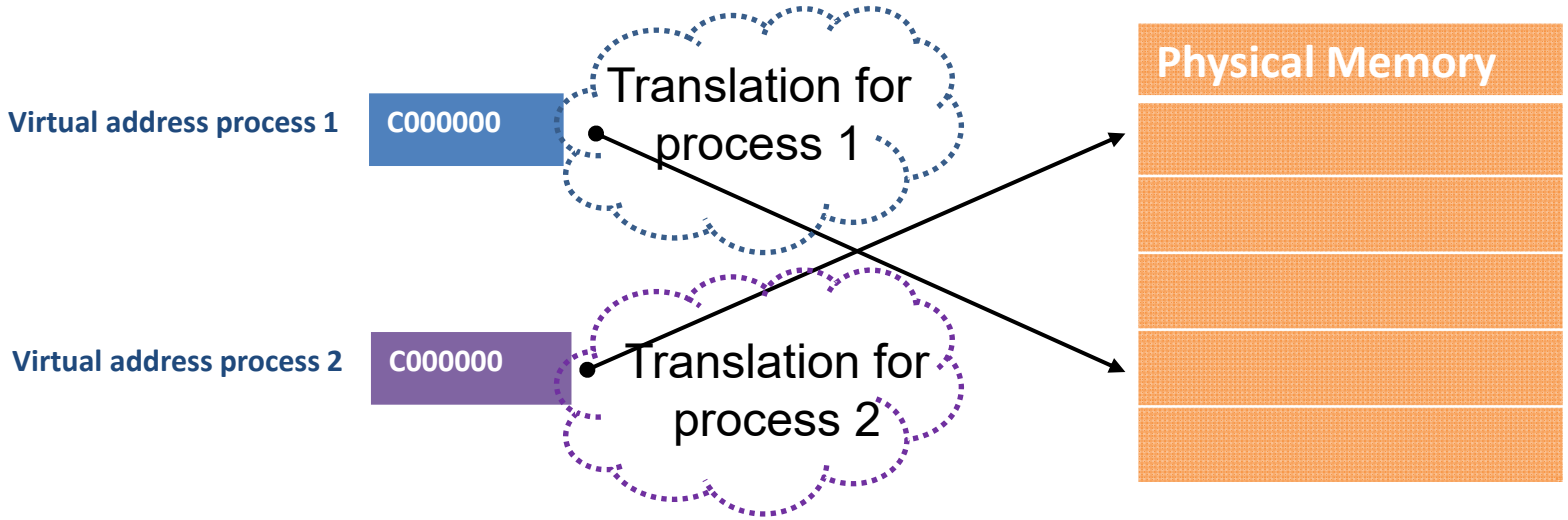
With virtual memory:



Address translation is performed in runtime on every access to memory

Why Virtual Memory is used (1)

1. To allow programs running concurrently to share the memory yet not interfere with each other
   - Each program is compiled in its own address space: a separate range of memory locations accessible only to this program.
     - The same virtual address in two different processes (*) must be assigned to different physical addresses
   - The translation from each process address space to the physical addresses enforces protection between processes



* Process = instance of a program being executed

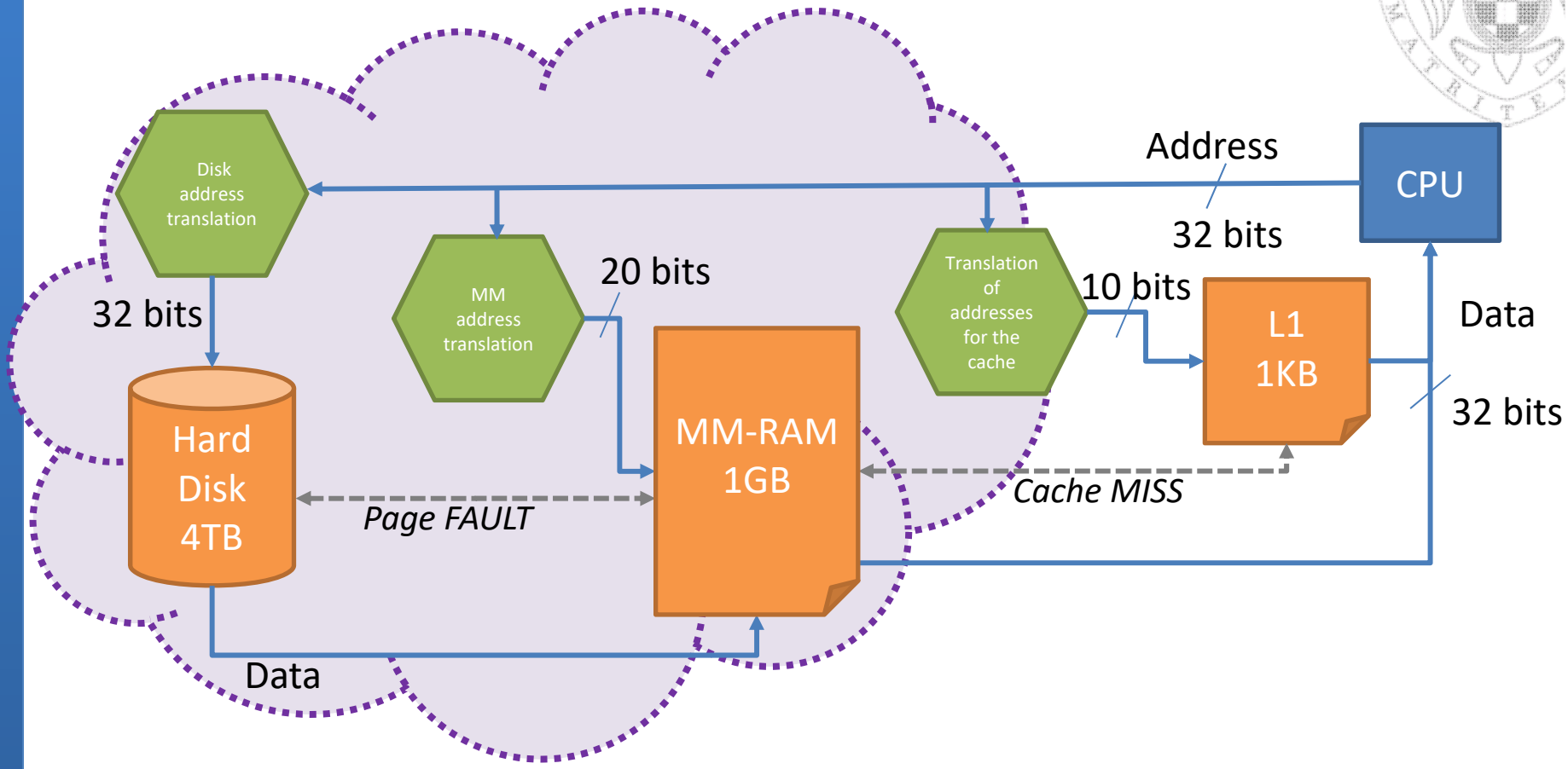
Virtual address space

- Its size only depends on the architecture: number of bits of the memory addresses.
  - For example: x86-32
    - Memory addresses: 32 bits
    - Address space: 2^{32} bytes = 4 GB
      - It's not so big
  - Another example: x86-64
    - Memory addresses: 64 bits
    - Address space: 2^{64} bytes = 16 EB (it is not fully used)
- Each process has its own address space.

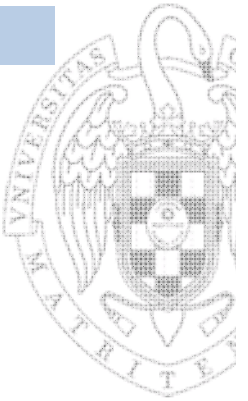
Why Virtual Memory is used (2)

2. To allow a single program to exceed the size of the available memory
   - VM automatically manages the two levels of the hierarchy represented by main memory (physical memory) and secondary storage.
   - Using the principle of locality (similar to cache memory)
     - MM only needs to contain the active portions of each process
     - When the data is not in MM, it will be searched on the Secondary Memory
3. It is not necessary for the same program (or process) to always be stored in the same position on Main Memory (Relocation)

Virtual memory and cache

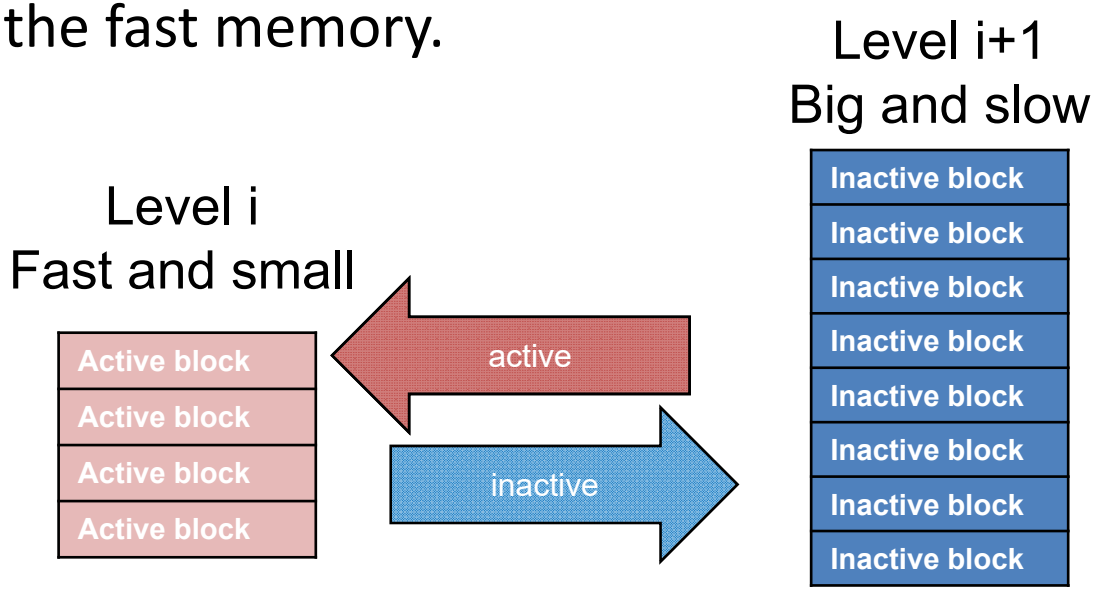


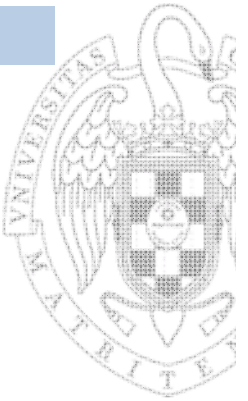
	VIRTUAL	CACHE
BLOCK	Page	Block / line
MISS	Page Fault	Cache Miss



Virtual memory and cache (2)

- Similarities:
  - Using the principle of locality active blocks are stored in the fast (but small) memory and inactive blocks in the slow one.
  - If high HIT rate is achieved, performance will be similar to the one of the fast memory.





Virtual memory and cache (3)

- Differences:
  - Cache memory management is implemented in hardware by the MMU (Memory Management Unit). Virtual memory management is implemented by the cooperation of hardware (MMU) and the operating system.
  - The access time and miss penalty are quite different. So, the miss rate needs to be much smaller.

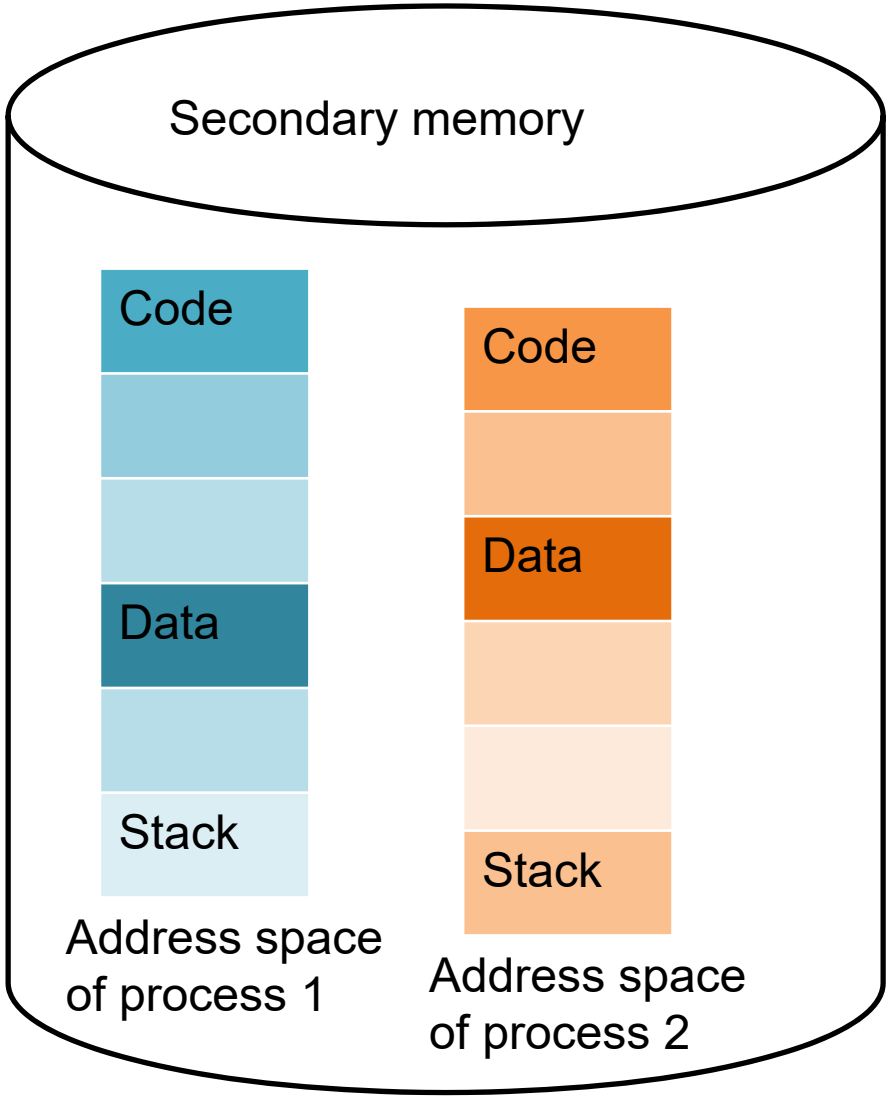
	L1 Cache	VM
Block size	16-128 bytes	4096-65.536 bytes
Hit time	1-3 cycles	100-500 cycles
Miss penalty	8-200 cycles	**1.000.000-10.000.000 cycles**
Access time	6-160 cycles	800.000-8.000.000 cycles
Transfer time	2-40 cycles	200.000-2.000.000 cycles
Miss rate	0,1-10%	0,00001-0,001%

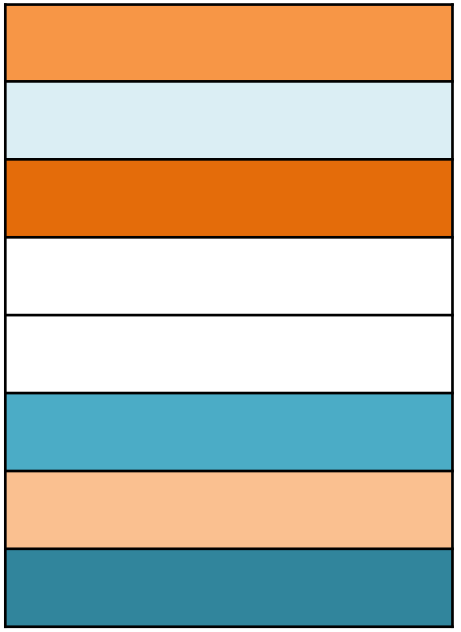
Figure B.20 from Hennessy & Patterson

Basic VM implementation: paging

- How does it work?
  - Main memory is divided into fixed-size blocks named **physical pages or page frames**
  - Every process (instance of a program) is divided in virtual pages of the same size as the physical pages
  - The process is stored in Secondary Memory (usually disk)
  - Only a small set of pages of a process are loaded to MM
  - The remaining pages are keep in SecM
  - New pages will be copied to MM when they are needed, replacing other pages
- Requirements
  - Address translation mechanism
  - Management strategies → Operating System

Basic VM implementation: paging

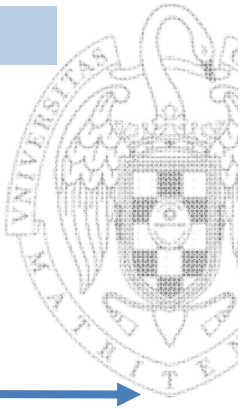




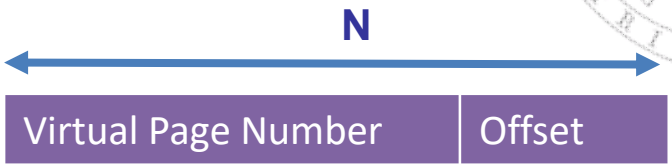
Basic VM implementation: paging

- The enormous miss penalty leads to several decisions:
  - Pages should be large enough to amortize the high access time: 4 KB-16 KB
  - Placement is fully associative to reduce miss rate
  - Page faults can be handled in software (the overhead will be small compared to the disk access time). LRU replacement is used.
  - Update policy is write-back, since writes take too long for write-through.
- How is a block found if it is in main memory?
  - It can be in any page.

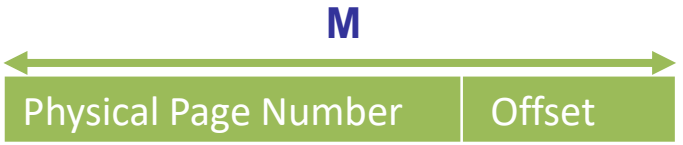
Physical and virtual addresses



- Virtual address
  - Provided by the processor
  - Divided in two fields:
    - Virtual page number
    - Page offset
      - It indicates the byte within the page to access
      - The number of bits of the offset determines the page size

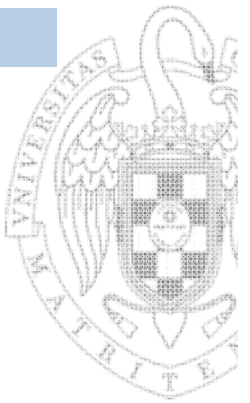


- Physical address
  - Used by Main Memory
  - Divided in two fields:
    - Physical page number
    - Page offset (the **same size and content** as the virtual address)

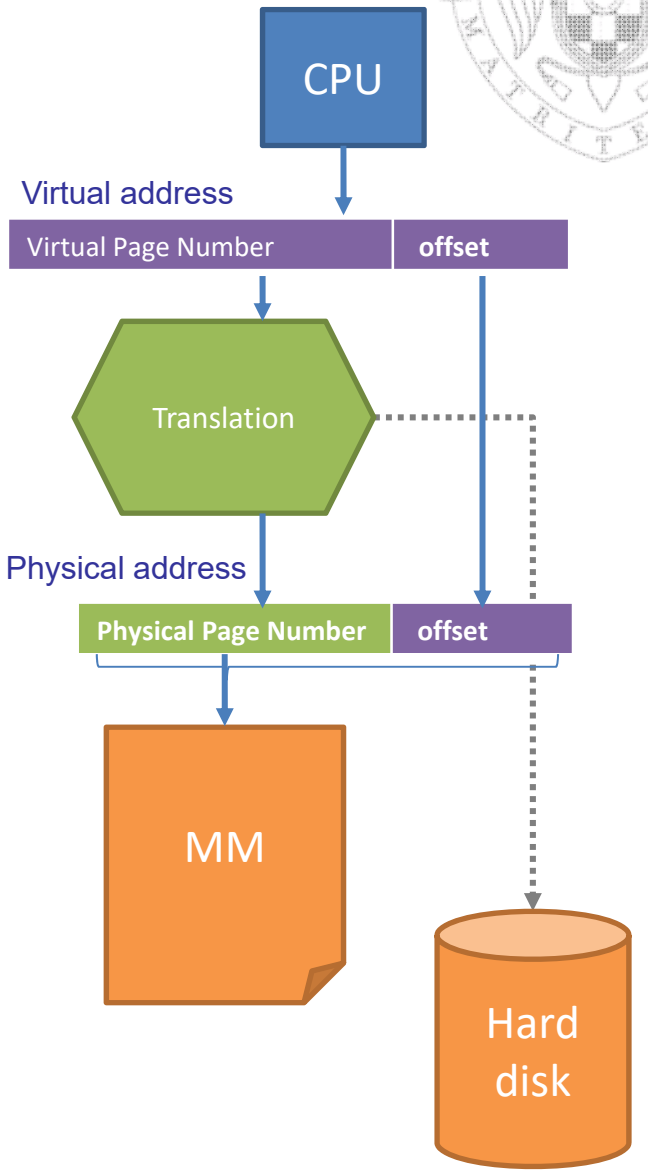


- The number of virtual pages needs not be equal to the number of physical pages.
  - Usually: Size of the Virtual Address >> Size of the Physical address (N>M)

Translation mechanism

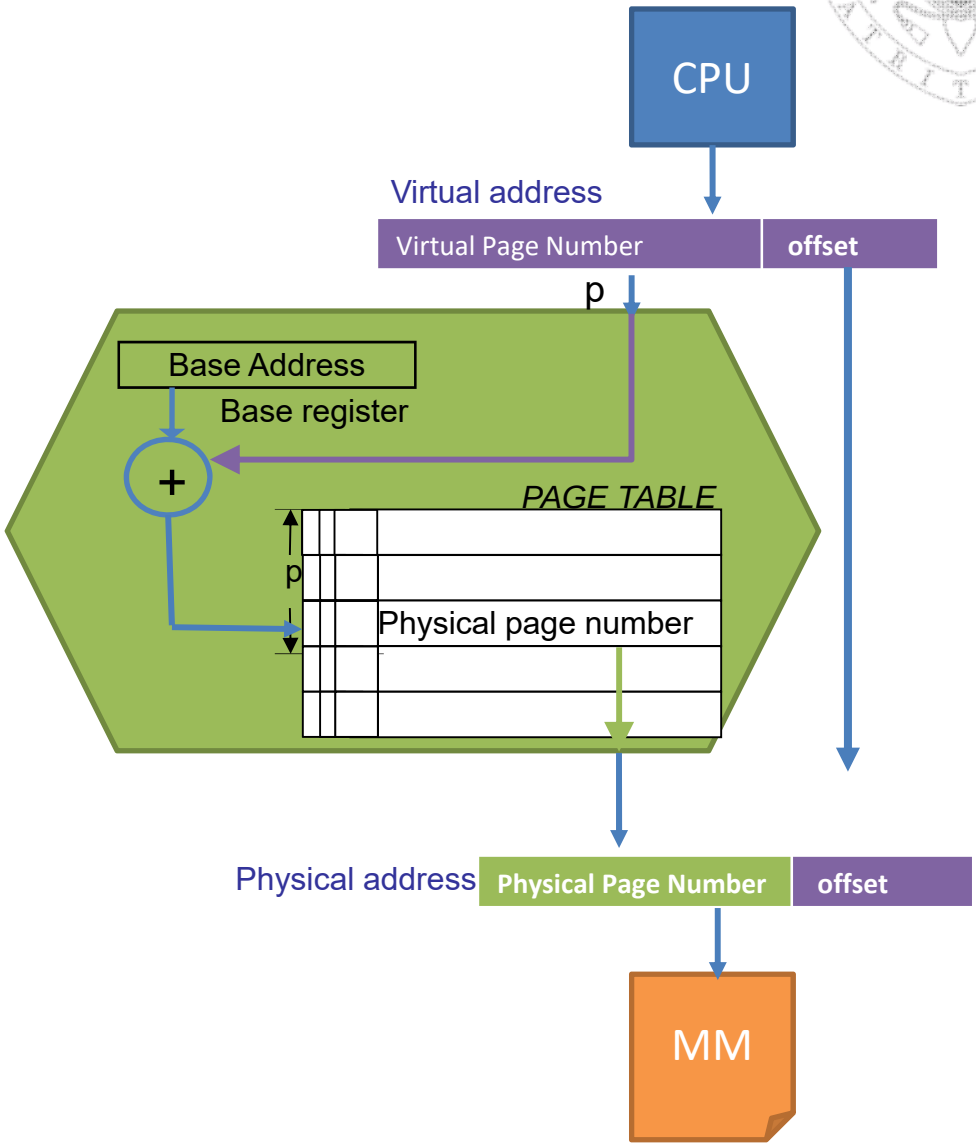


- Translate the virtual address to physical address
  - Translate the virtual page number to a physical page number
  - Keep the same offset for the virtual and physical address
- The translation is dynamic: it is done on every access to memory
- A mechanism must be implemented to address the virtual pages found in SecM

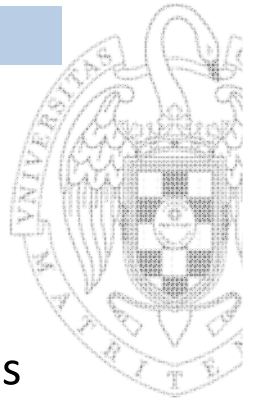


Translation mechanism (2)

- Each process has a page table:
  - To store the translation from virtual to physical page.
- A Base Register is needed
  - Indicates the address of MM where the table starts
- The virtual page number acts as an offset that is added to the register contents to find the entry

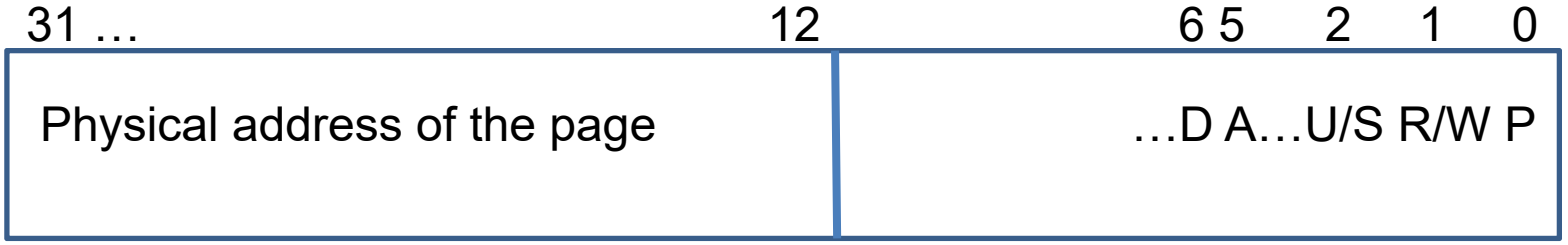


Page Table

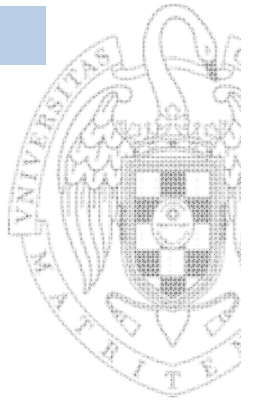


- It is a data structure for translating virtual addresses to physical addresses
  - OS creates this table when the process is created
  - Resides in main memory
- The table will have as many entries as possible virtual pages are available in the architecture
  - From the virtual page number, the physical page number is obtained
- It includes a Valid bit for each entry to indicate whether the virtual page is in MM
  - Bit = 1 →page in MM → entry contains a number of physical page
  - Bit = 0 → page not in MM →page fault
- Other information that the entry may contain
  - Reference bit
  - Dirty bit
  - Security/permission bits (Read/Write, User/Supervisor,…)

Page Table Entry: x86-32 bits

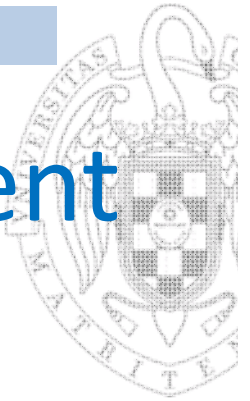


P: present (valid bit)
R/W: read/write
U/S: user/supervisor
A: accessed (reference bit)
D: dirty



Page Miss/Fault

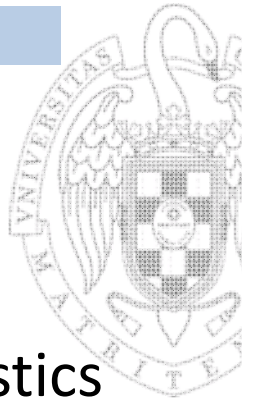
- What is it?
  - The page to access is not in Main Memory
- How can we know that a miss has happened?
  - Valid bit = 0
- What happens then?
  - An exception occurs and the OS takes control
  - Actions:
    - Search page in the hard disk
    - Decide where the new page should be placed in Main Memory
- How can we search pages in the hard drive (HD)?
  - The virtual address does not tell us where this page is in HD
  - OS knows the address of each virtual page in the HD



Page Miss/Fault - LRU replacement

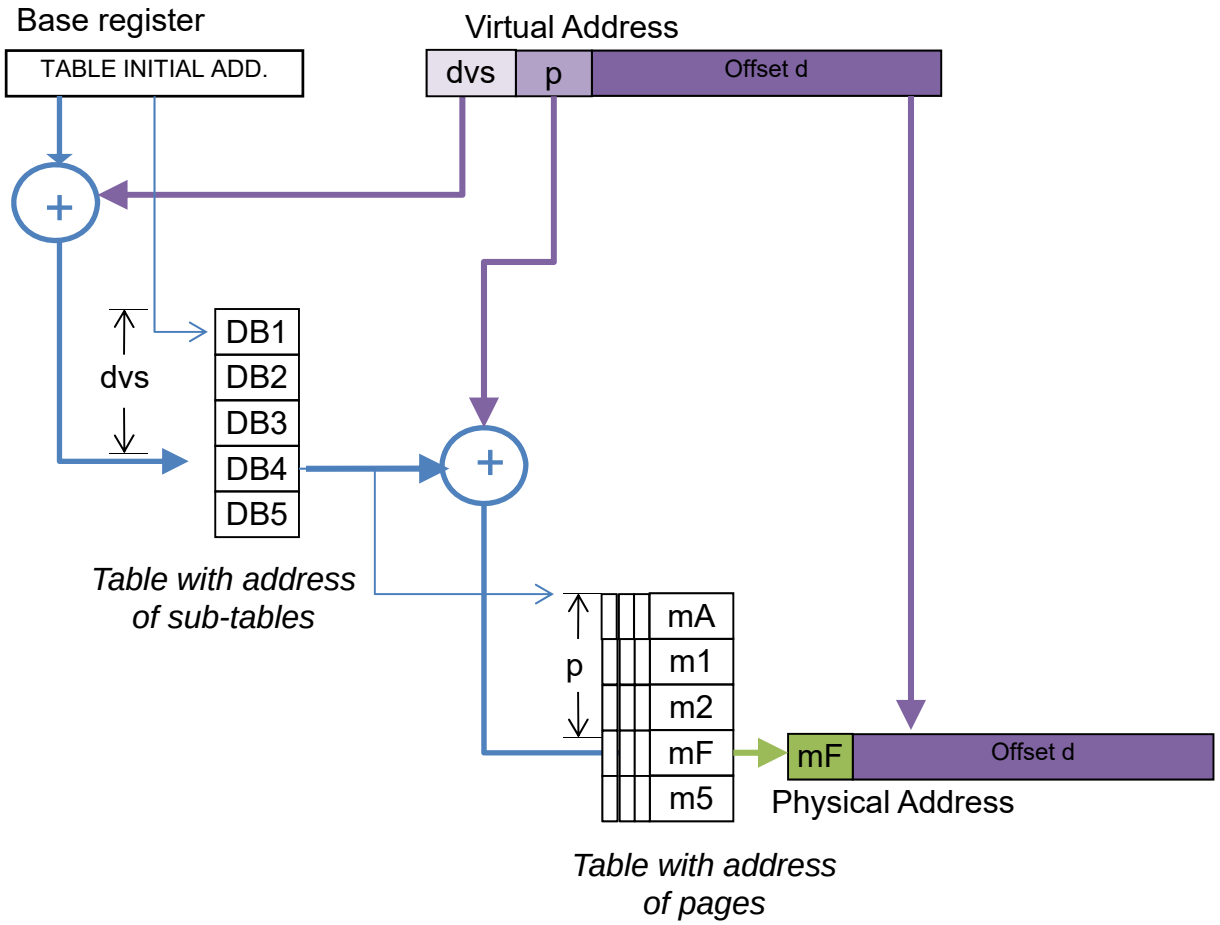
- Replaces the Least Recently Used (LRU) page
  - OS creates a data structure that stores information about page use
- Replaced pages are stored in the swap region
  - Example:
    - First, pages referenced are 10,12,9,7,11,10
    - If MM is full, a new page must replace page 12
- Implementation of exact LRU is very expensive: Usually, approximate implementations

Problems with the page table



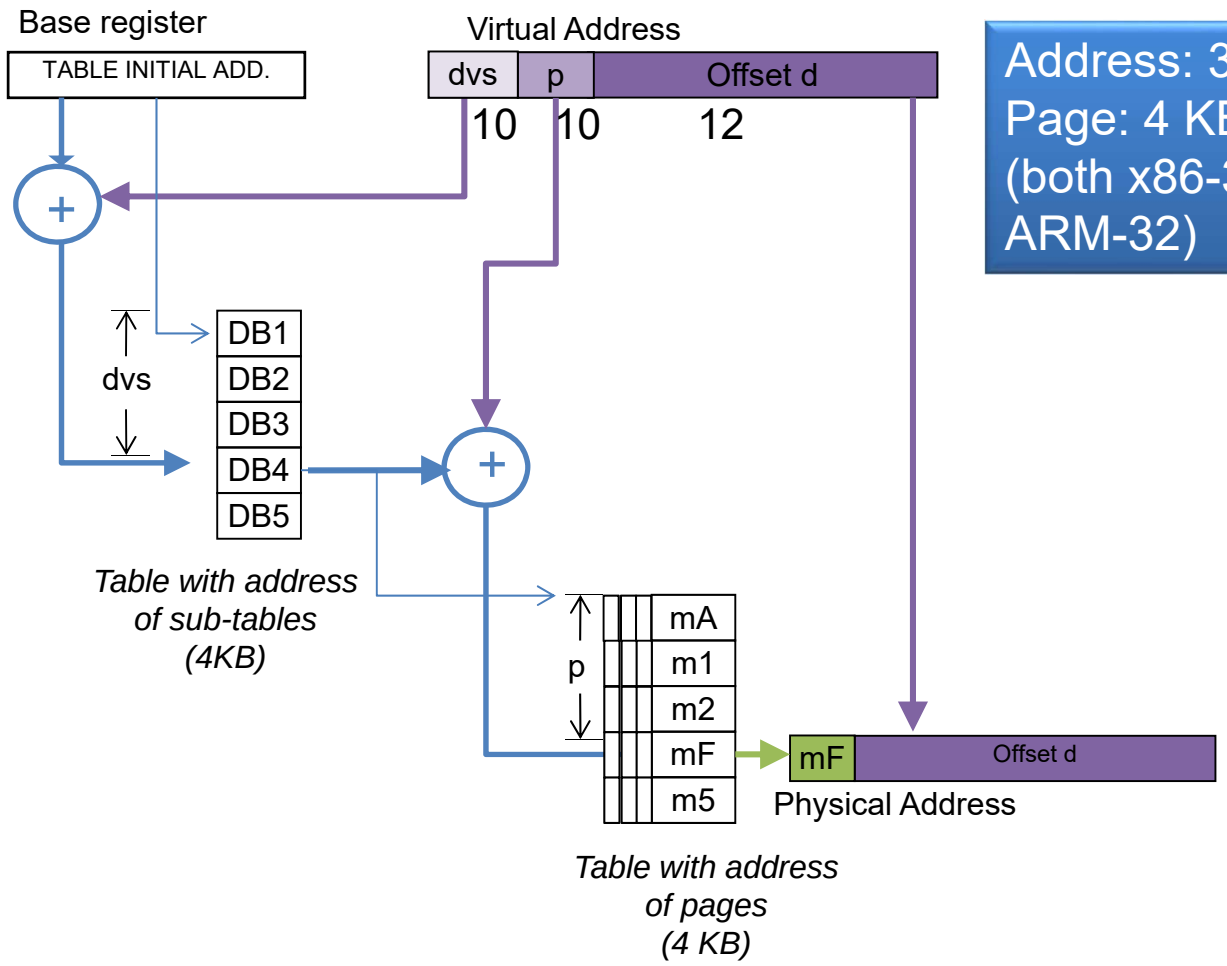
- A virtual memory system with the following characteristics
  - 32-bit virtual address
  - Page size of 4 KB
  - 4 bytes per entry
  - Number of pages = 2^{32} / 2^{12} =2^{20}
  - Page table size = Number of entries x Size = 2^{20} x 2^{2} = 4MB
- Problem:
  - A computer can have tens to hundreds of active processes, thus almost all memory would be used for storing page tables
- Several solutions:
  - Two-level page table
  - Inverted table page

Two-level page table



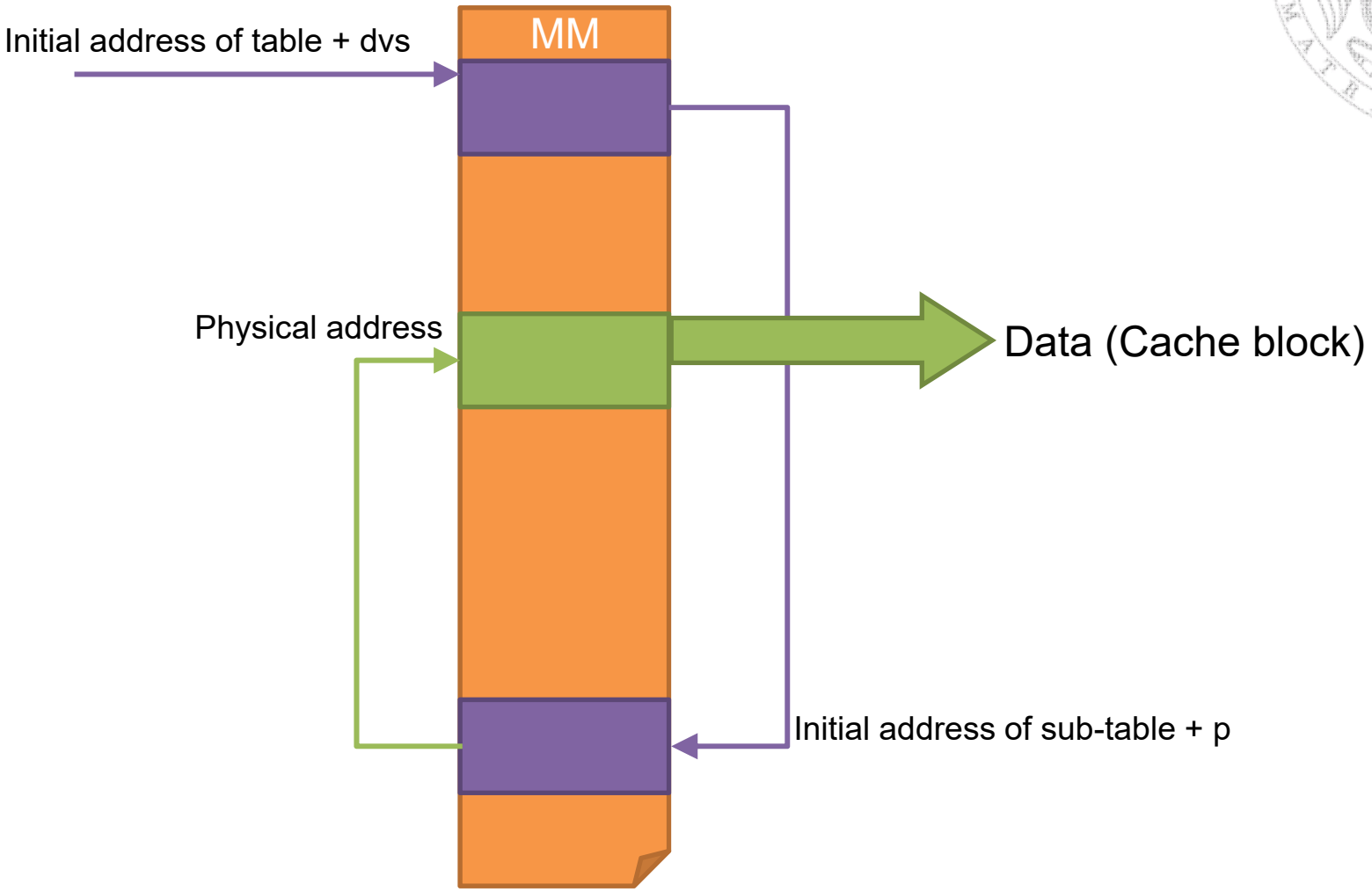
x86-32bits. V.A. 32bits, P.A. 40bits. 2-level table
x86-64bits. V.A. 48bits, P.A. 52bits. 4-level table.

Two-level page table



Address: 32 bits
Page: 4 KB
(both x86-32 and ARM-32)

Two-level page table (2)



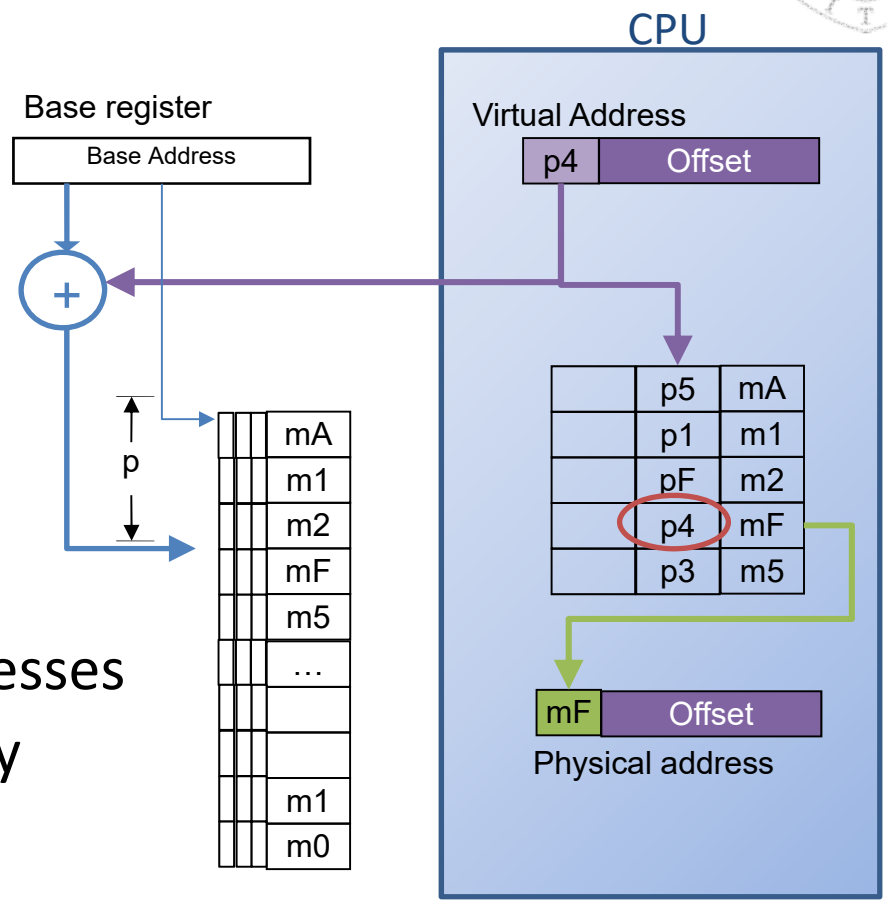
We need 3 memory accesses in a 2-level page table.

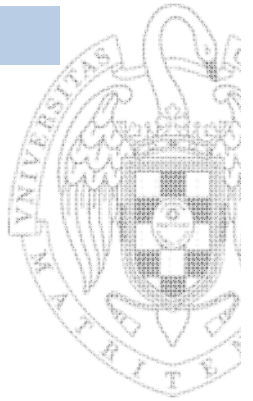
Making address translation fast: TLB

- Problem of page tables: Time to access data
  - Since the tables are stored in MM, two memory accesses are required:
    - Obtain the physical address
    - Obtain data
  - Multilevel tables → Longer time
- Solution: Translation Lookaside Buffer (TLB)
  - Cache that stores information about the recent translations
  - Very small and fast
  - Exploits locality of references
    - If an address is translated, there is a high probability of re-translating the same address in the near future
  - Usually located within the core
  - Tag: stores the virtual page number
  - Data: stores the physical page number
- The TLB also stores other bits of information for management
  - Dirty bit⟶ Has it been modified?
  - Reference bit ⟶ Has it been accessed?
  - Valid bit
  - Permissions (R/W,…)

Translation Lookaside Buffer

- The virtual memory system consists of two tables:
  - TLB
    - It contains the latest translations
    - Located in the core
  - Page table
    - It contains all page addresses
    - Located in Main Memory

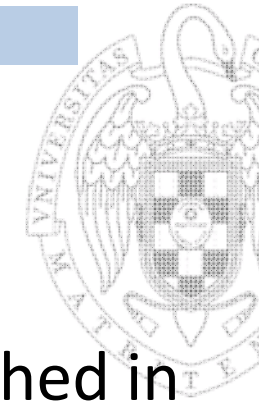




TLB-Alternative approaches

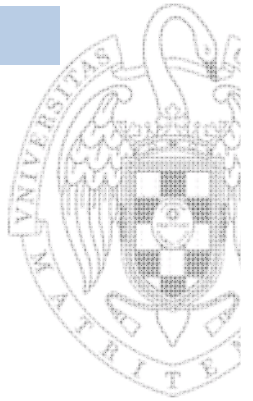
- There are two alternatives in the design of a TLB
  - With process IDs
  - Without process IDs
- TLB without process IDs
  - TLB is accessed only with the virtual page number
  - Whenever there is a process change, the OS must invalidate the TLB, as each process has its own memory map
- TLB with process IDs
  - Access to the TLB with the page number and a process ID
  - This identifier is also stored in each entry of the TLB
  - Whenever there is a process change, there is no need for invalidating the TLB contents

TLB Management



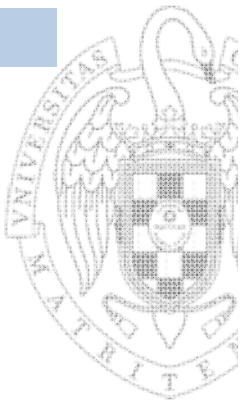
- In each reference, the virtual page number is searched in the TLB
  - Hit → The entry is found in the TLB and its valid bit (V) is set
    - The physical page number from the TLB is used to calculate the physical address
    - The reference bit (R) is set for LRU
    - If it is a write, the dirty bit (D) is set
  - Miss → The entry is not found in the TLB or the valid bit is unset
    - Determine if it is a TLB miss or a Page miss/fault
    - If the page is in Main Memory, then it is a TLB miss
      - Action: Copy the entry into the TLB
    - If the page is not in MM, then it is a true Page Fault
      - Action: OS exception is triggered
    - As the TLB has much less entries than pages in MM, TLB misses will be much more frequent than page faults

TLB Management (2)

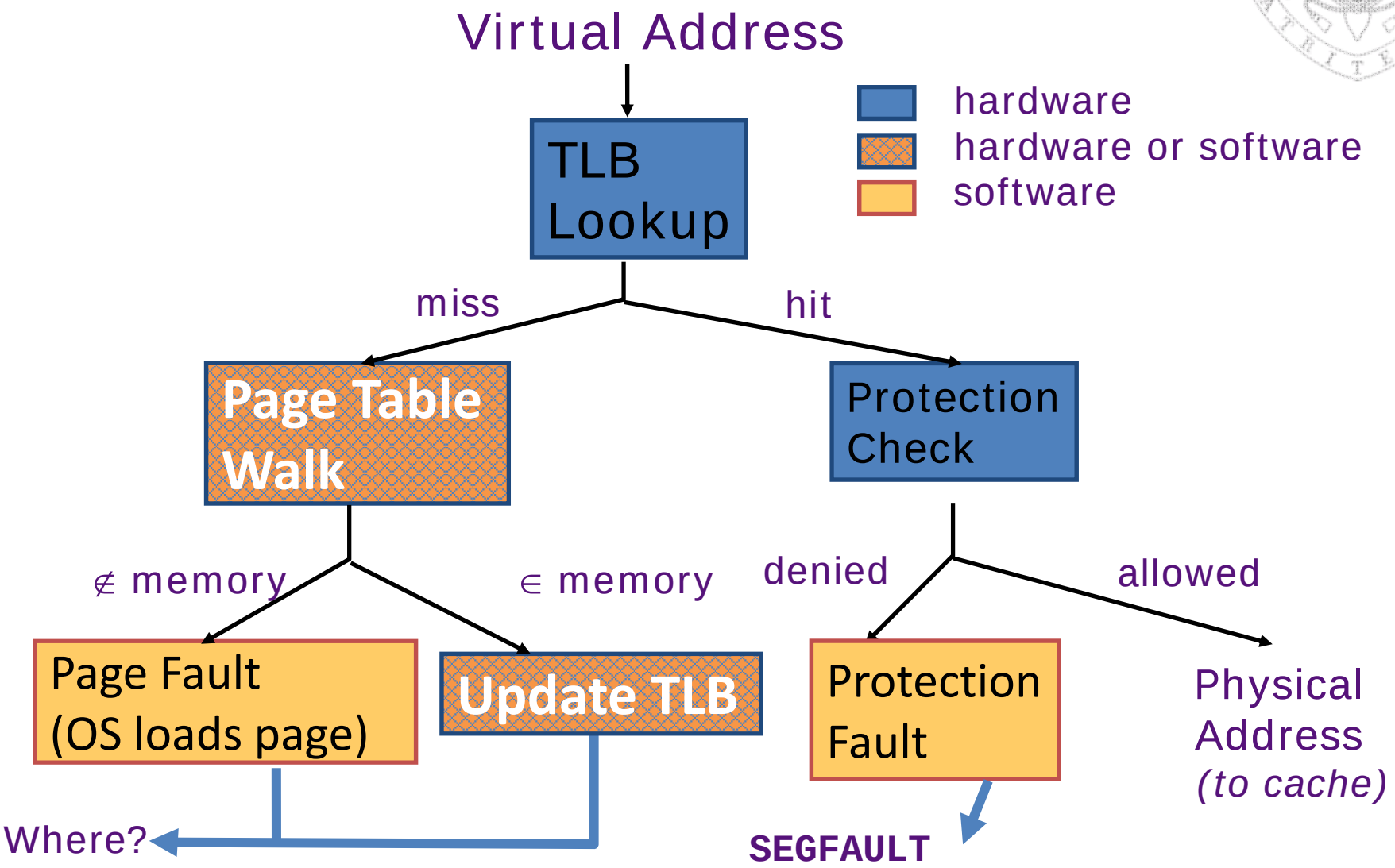


- TLB miss can be handled by
  - SW
  - HW

  Both approaches have a similar performance
- When a TLB miss occurs, the new translation must replace an entry in the TLB
- Typical configuration values
  - Size of the TLB: 16-512 entries
  - Hit time: 0,5 -1 clock cycles
  - Miss penalty: 10-100 clock cycles
  - Miss rate: 0.01-1%

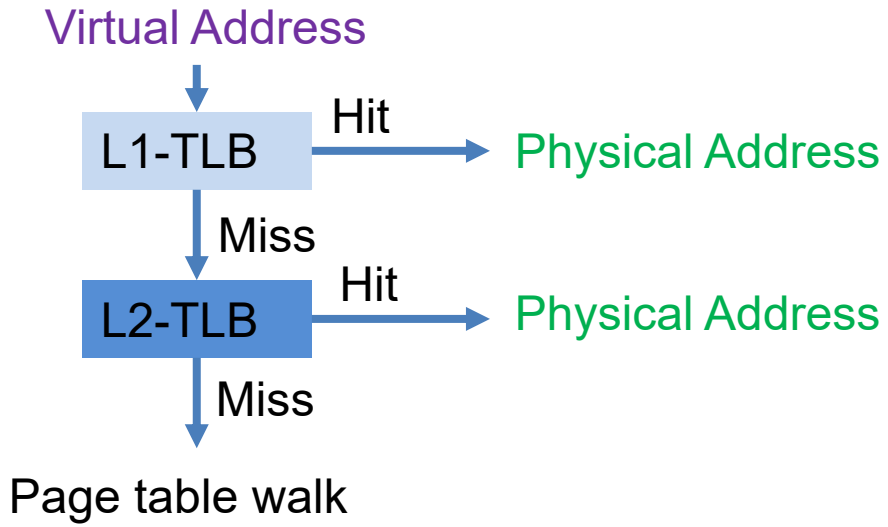


TLB Management



Multi-level TLB

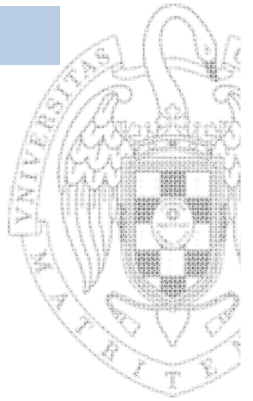
- In order to remain fast, the TLB cannot be very large
  → Processors use a hierarchy of TLBs to cache translations



	Intel Kaby Lake	ARM A73
L1-TLB	64 entries (data), 128 entries (instr.)	48 entries (data), 32 entries (instr.)
L2-TLB	1536 entries	1024 entries

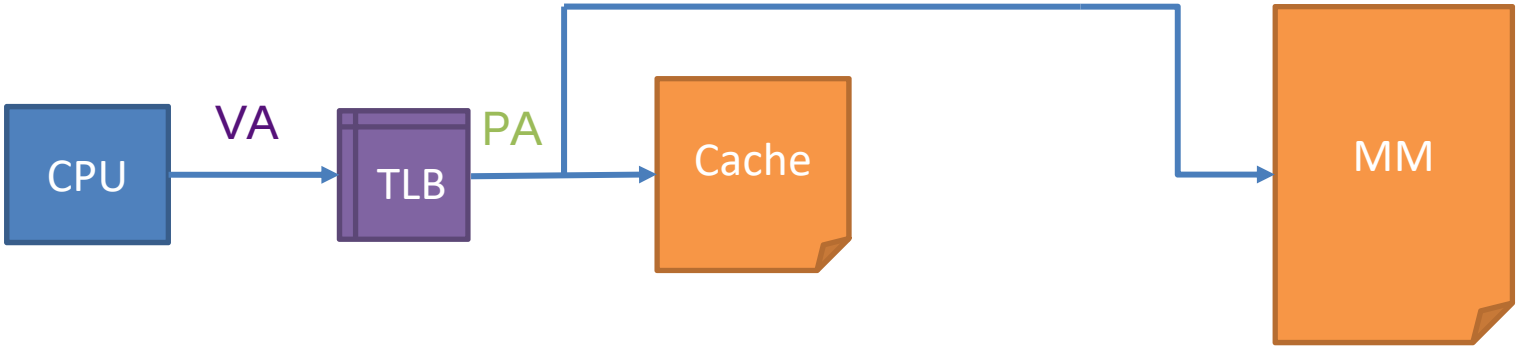
Integration of virtual memory and cache

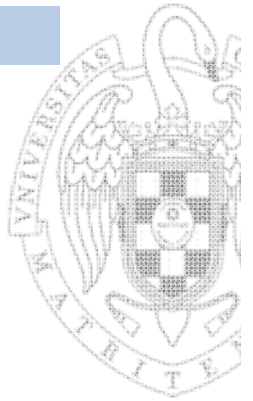
- The virtual memory and cache work together in the hierarchy
  - Data cannot be cached unless it is in MM
- The OS plays an important role in removing pages from MM
  - Cleaning the cache
  - Modifying the page table and the TLB so when trying to access the page a page fault occurs
- Relation between the virtual address and the cache?
  - There are three approaches:
    - Physical cache (PIPT)
    - Virtual cache (VIVT)
    - Virtually accessed - physically tagged cache (VIPT)



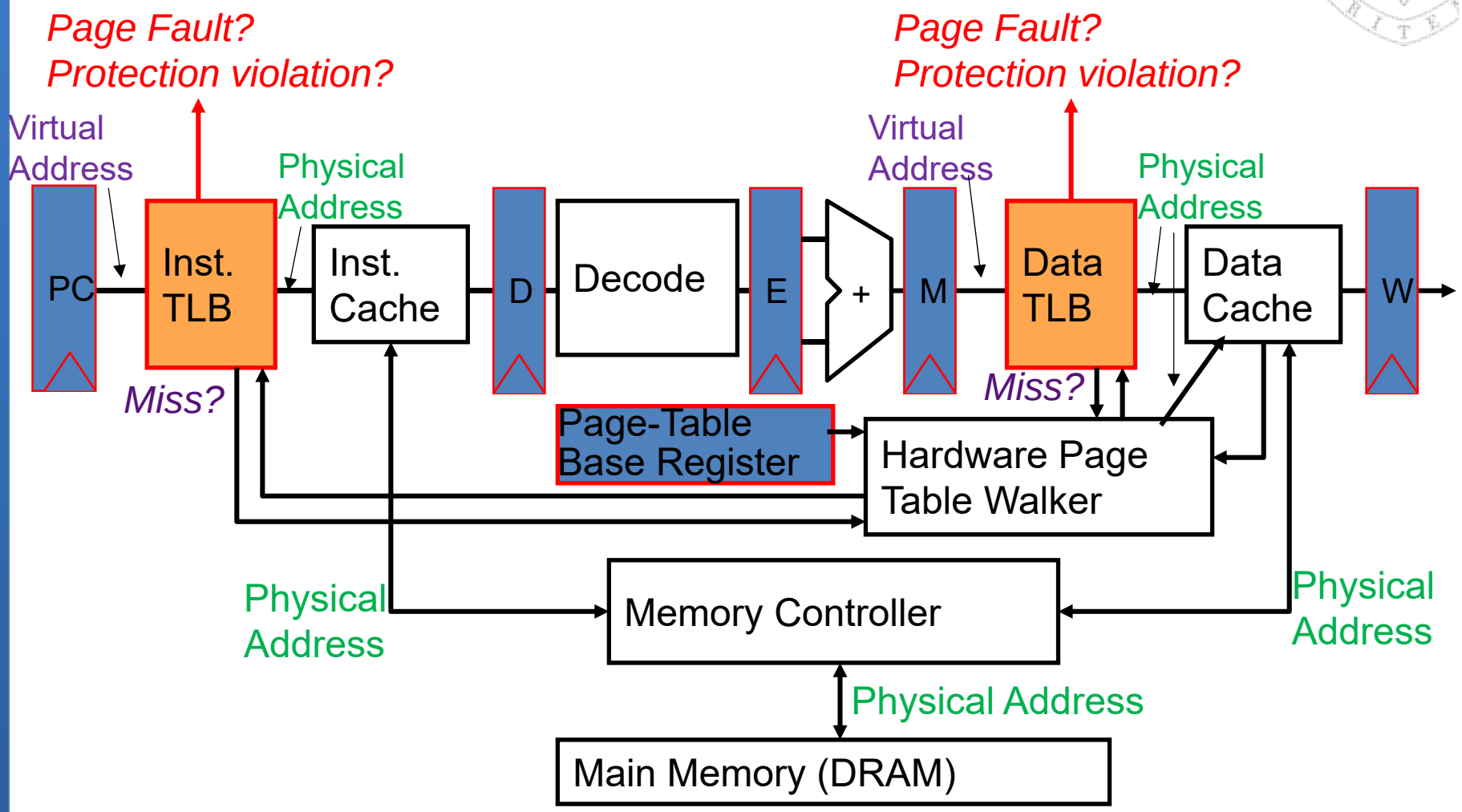
Physical cache

- **Physically Indexed Physically tagged**
- The cache uses only physical addresses:
  - It uses the physical address to index the cache and to compare tags
- The virtual address generated by the processor is always translated through the TLB
- Advantage: It easily allows multiple active processes
  - The same virtual address for different processes directed to different physical addresses
- Problem: High delay in the accesses to cache because the address must be translated (problem for L1-caches)
- Synonyms can be a problem.

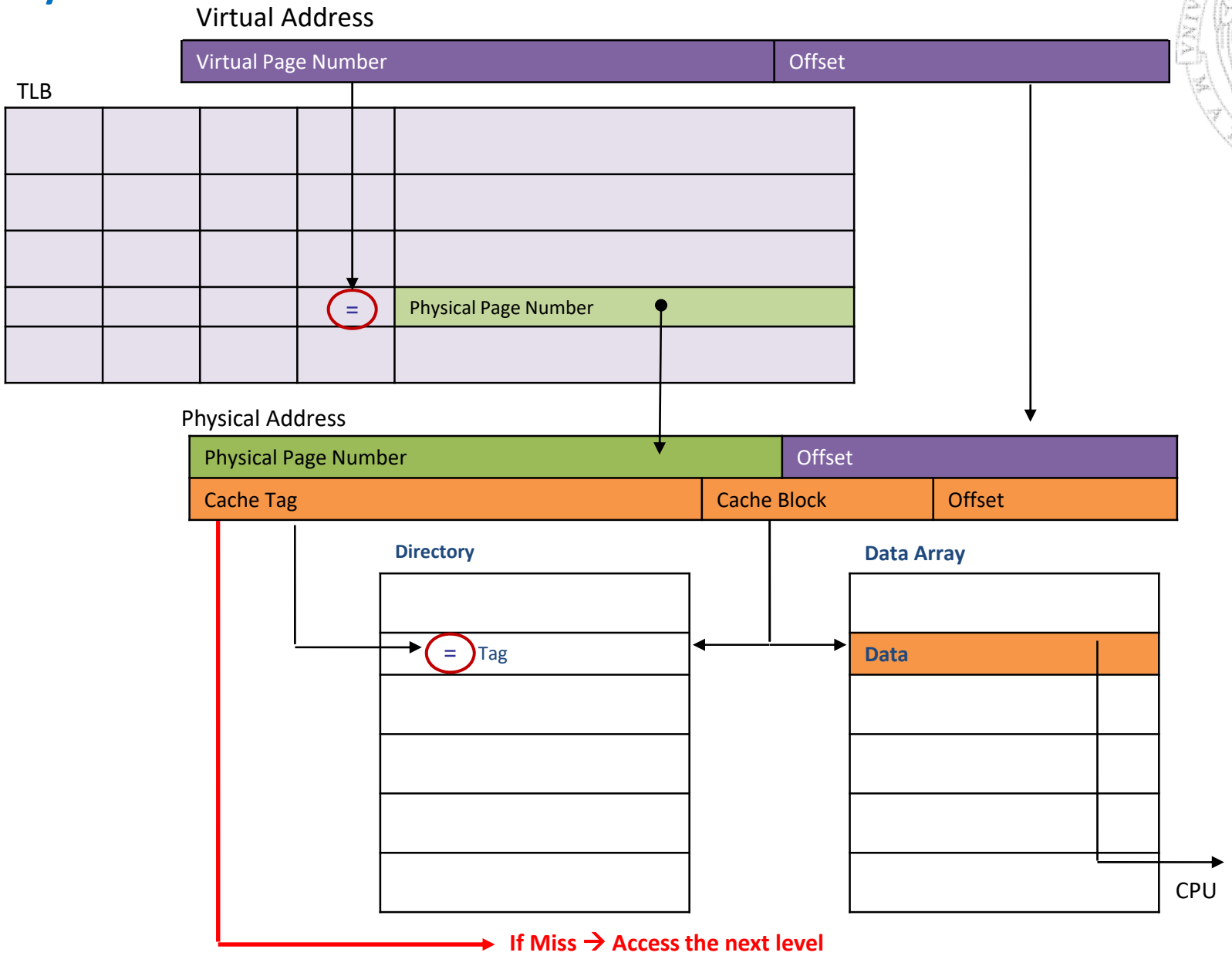




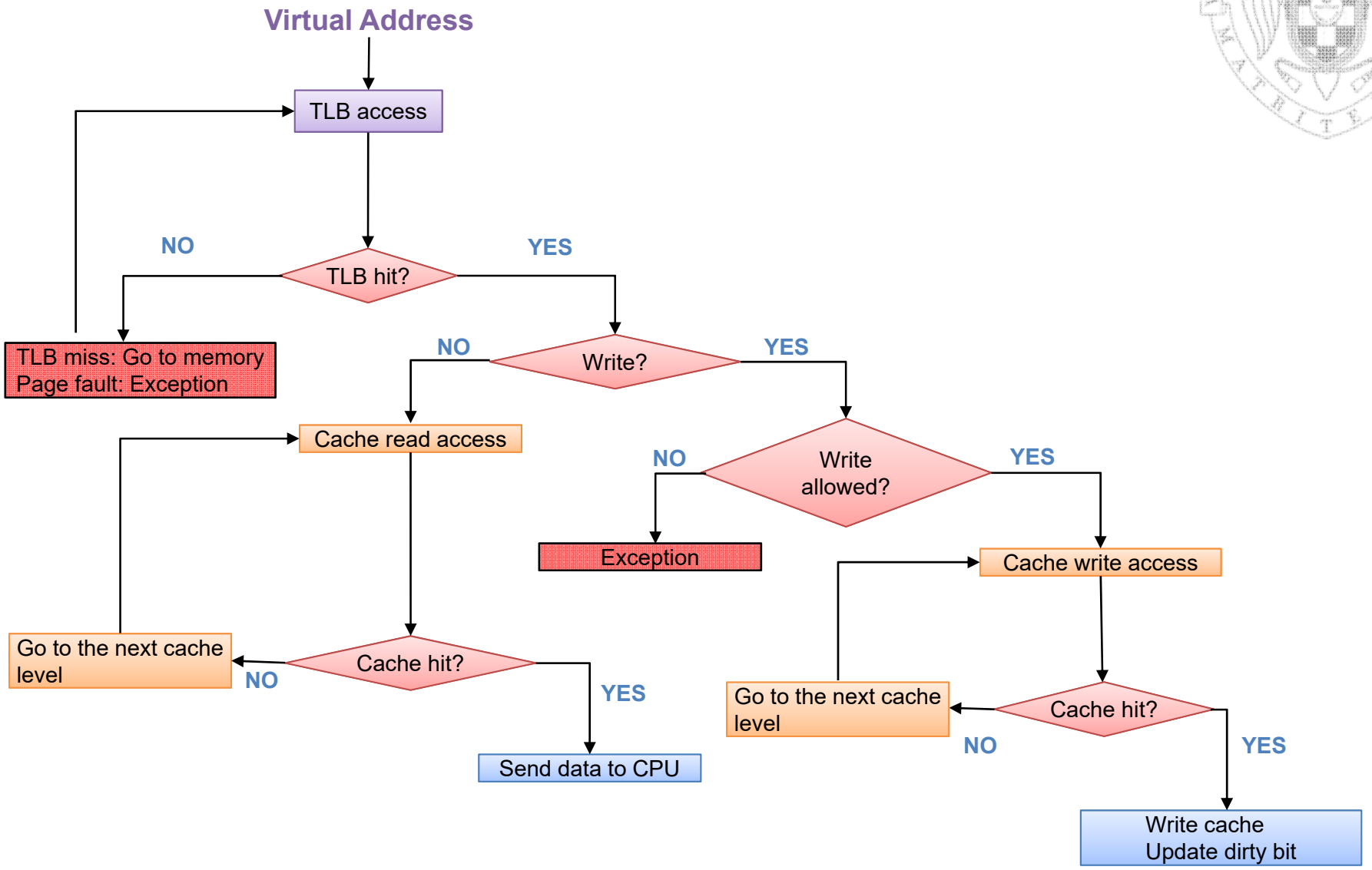
Physical cache in the pipeline



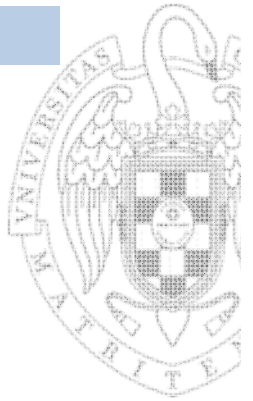
Physical cache: address translation + cache access



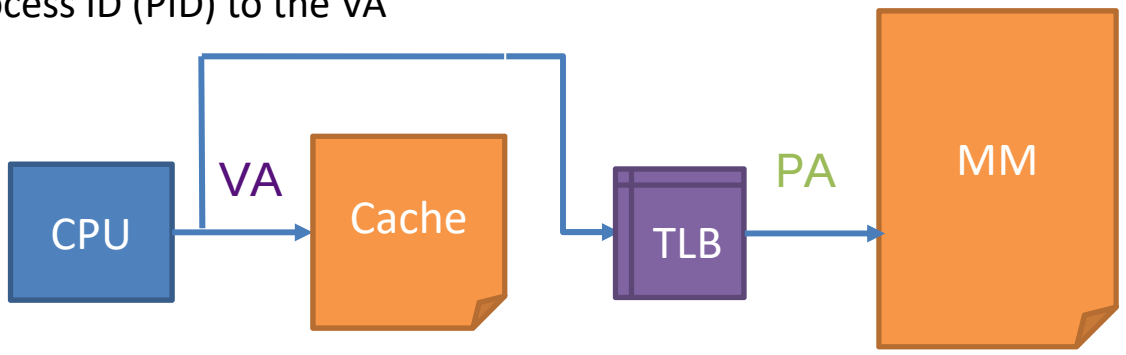
Physical cache



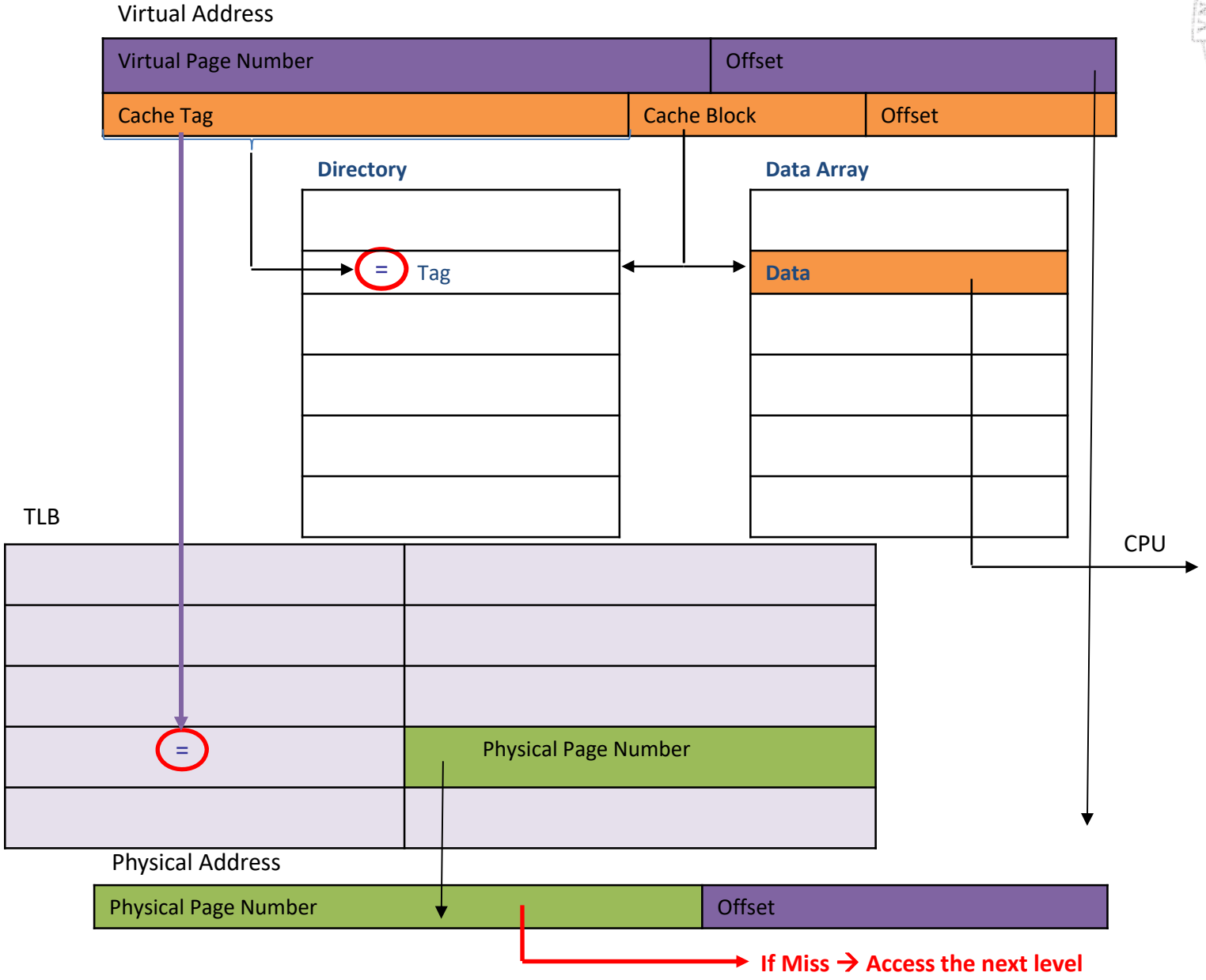
Virtual cache



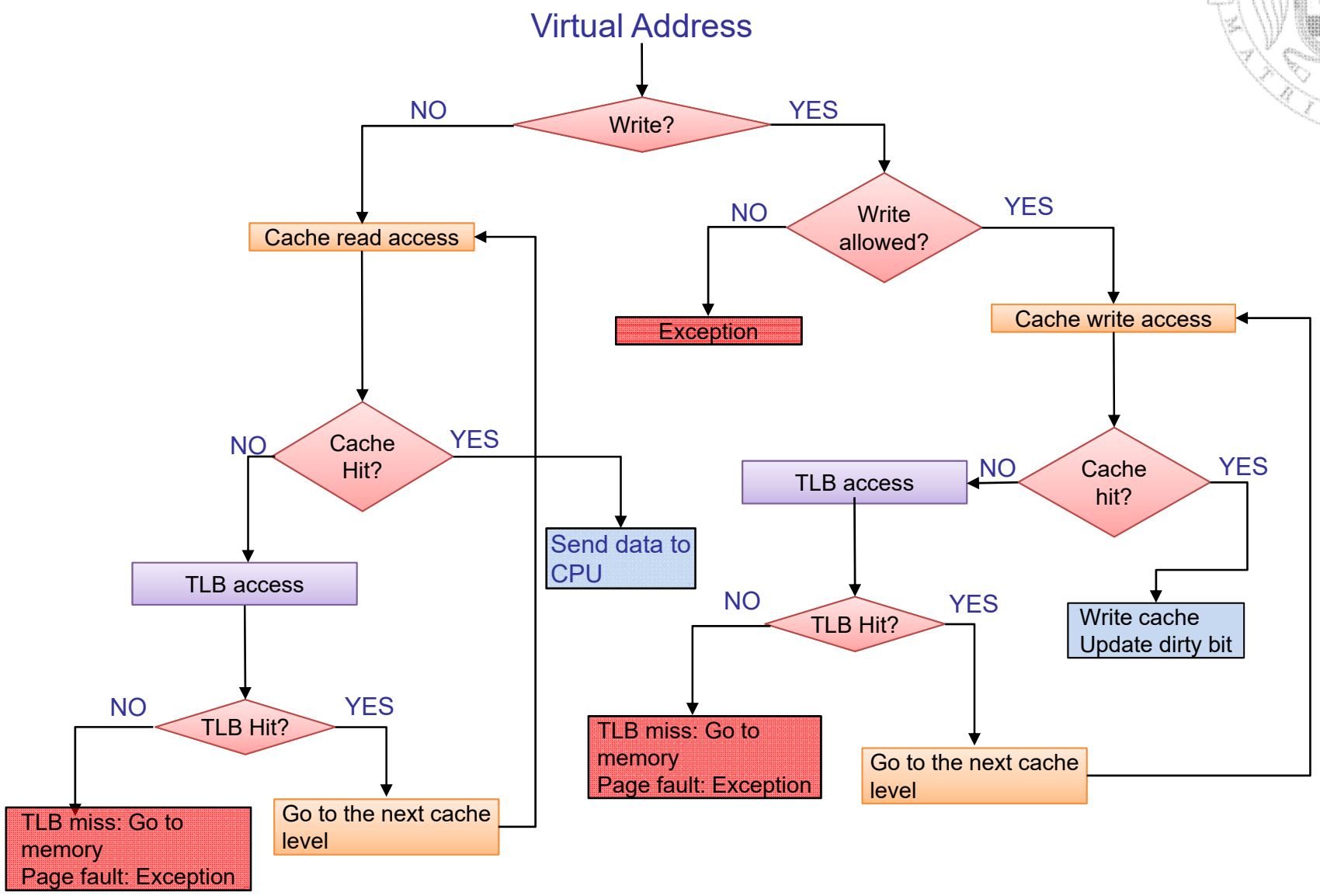
- **Virtually Indexed Virtually tagged**
- The cache uses only virtual address
  - It uses the virtual address to index the cache and to compare tags
- Advantage: fast access to the cache
  - Access the TLB only when there is a cache miss
  - More time required for cache miss management
- Context switching (process change) problem:
  - When a context switch takes place, the cache must be cleaned, otherwise, false hits are possible (Homonyms)
  - Context switching time = cancellation time (flush) + initial misses
- Solution:
  - Add a process ID (PID) to the VA



Virtual cache: cache access + address translation



Virtual cache

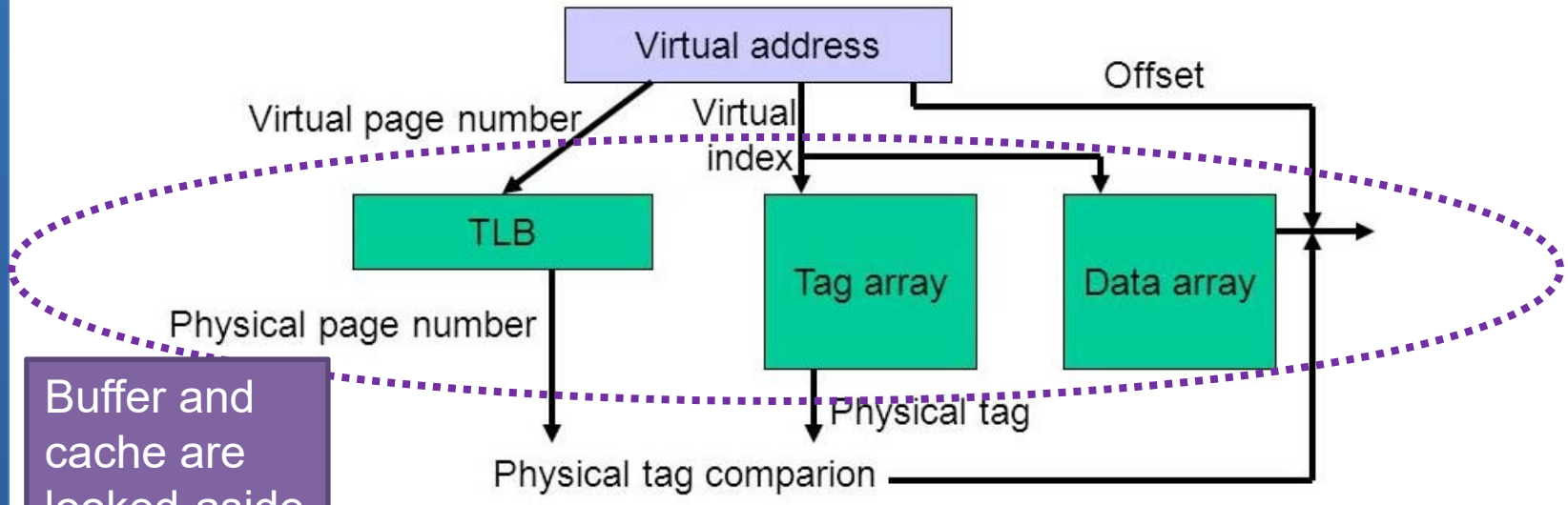


Virtually Accessed - Physically Tagged cache

- It uses the virtual address for indexing the cache
- The following actions are performed in parallel:
  - Reading data using the virtual address
  - The TLB is accessed and then tags are compared
- Advantage:
  - Hide translation latency Virtual Address => Physical Address

Widely used

- Disadvantage:
  - Limits the size of the cache
  - One way to increase the maximum size of the cache is to increase the associativity



Buffer and cache are looked-aside

Virtually Accessed - Physically Tagged cache