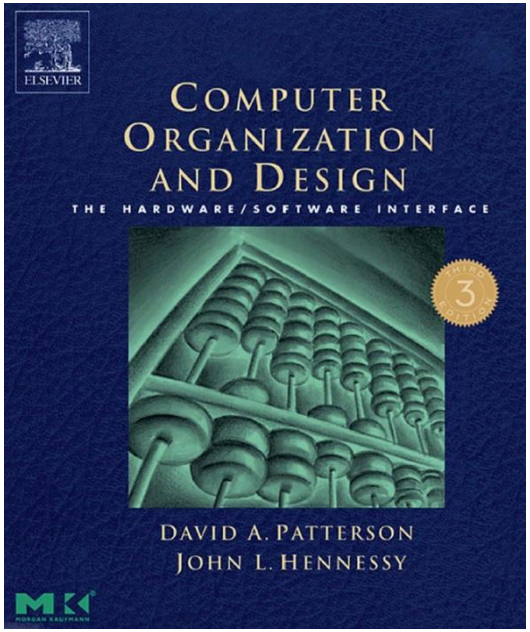
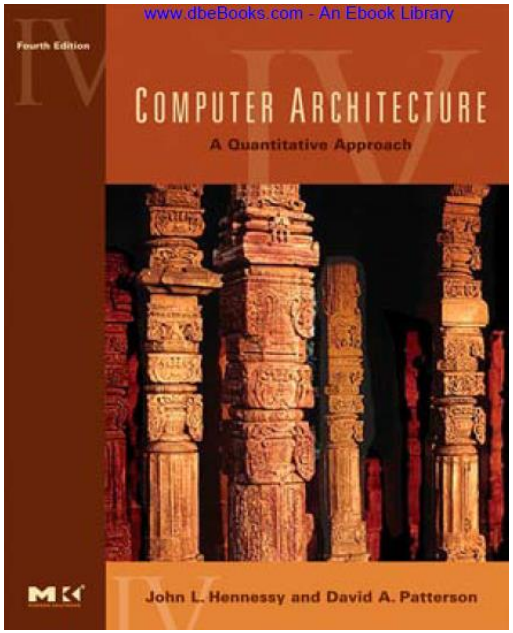
Virtual Memory

prepared and instructed by
Shmuel Wimer
Eng. Faculty, Bar-Ilan University





Motivation

Virtual memory (VM): A technique using the main memory as a **"cache"** for secondary storage (disk).

A **collection of programs** running simultaneously on a computer requires memory that is much larger than the main memory.

VM allows a single user's program to **exceed** the size of main memory.

Main memory needs contain only the **active portions** of the programs, possible by the **principle of locality**.

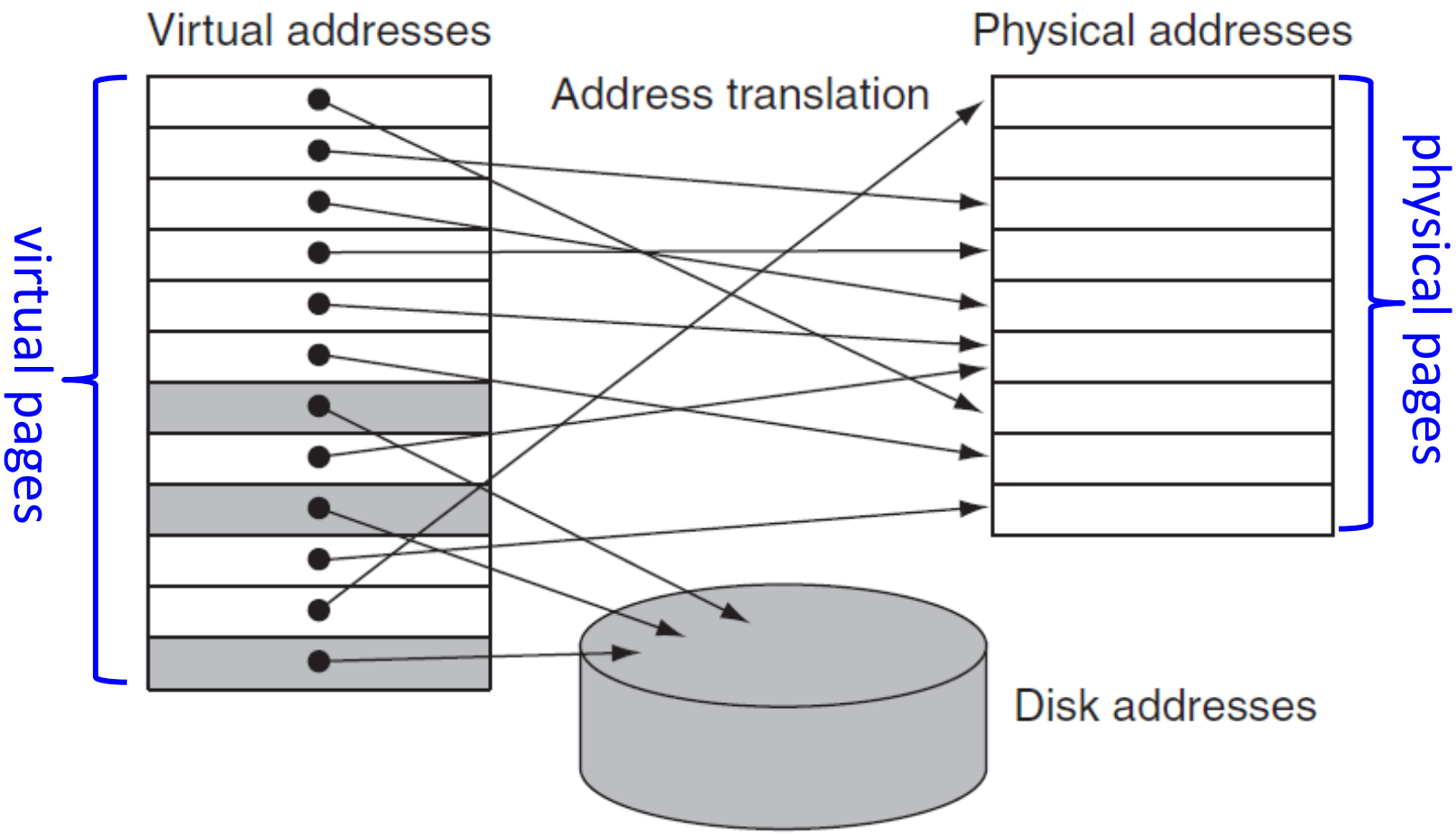
Compiler allocates each program its own **address space without awareness** of where, when and how they will run.

VM **protects** the programs from each other using the portions of main memory that have been assigned to it.

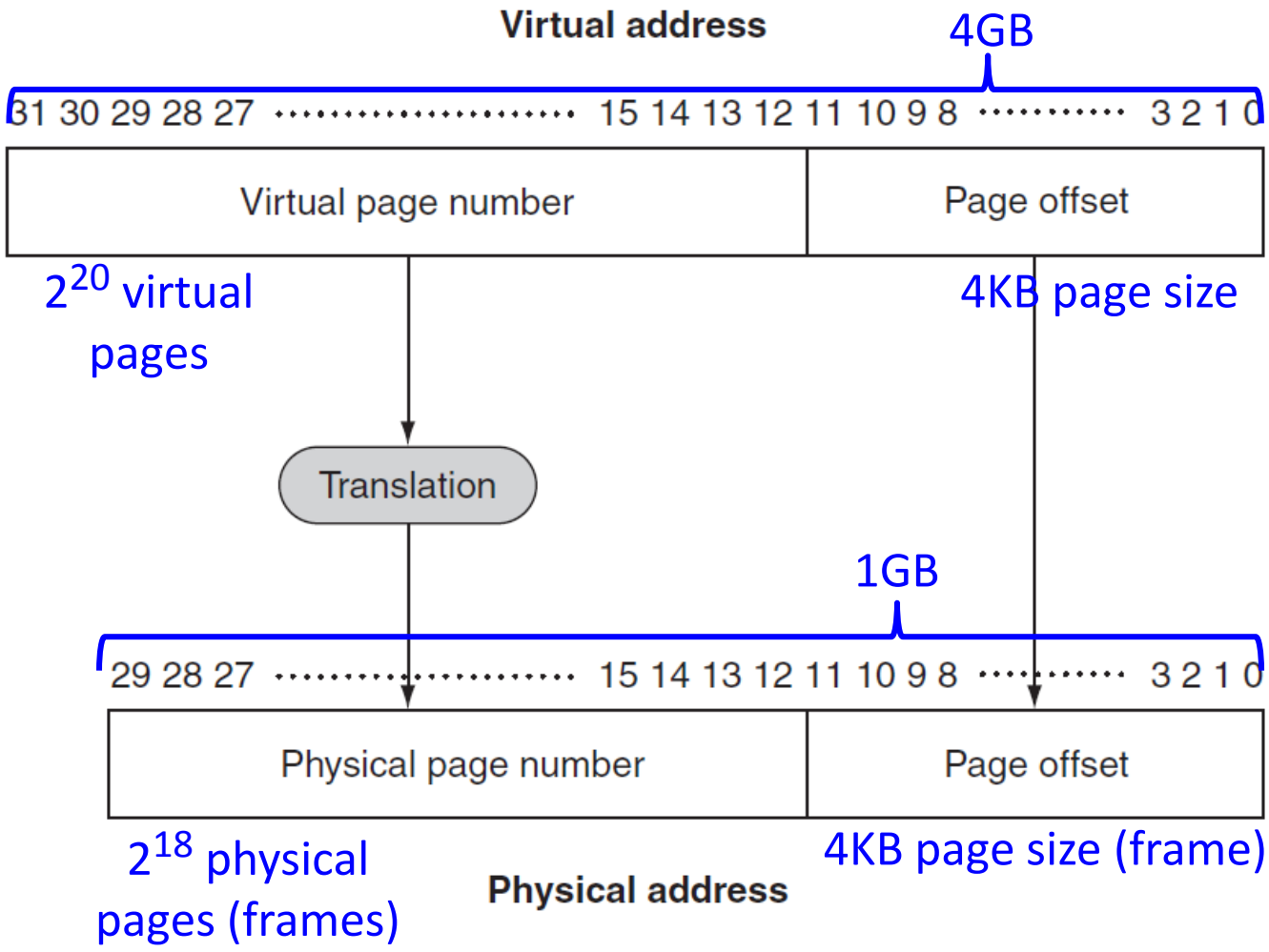
VM translates the program's address space to **physical addresses**, enforcing **protection** of a program's address space from other programs.

A VM block is called a **page**, and a virtual memory miss is called a **page fault**.

VM produces **virtual address**, translated by a SW and HW combination to **physical address**.



Virtual to Physical Address Mapping



Illusion of **unbounded** amount of VM.

Design Choices in VM

Disk access takes **millions** of clock cycles! Pages should be **large** to amortize disk access time.

1 KB (embedded), 16 KB (PC) to 64 KB (severs).

Minimize page faults by fully associative placement of pages in memory.

Handle page faults and placement by **SW algorithms** since overhead is small compared to disk access time.

Write-through does not work for VM, since writes take too long. Uses **write-back**.

Page Table: Placing and Finding Pages

Reducing page fault frequency is critical.

Operating system (OS) maps **VM page** to **physical frame** (associative placement) with smart algorithms and complex data structures tracking page usage.

The **page table** is stored **in memory**.

Contains the **virtual to physical** address translations.

Indexed by the virtual page address.

Table's entry is the frame address if the page is presently in memory.

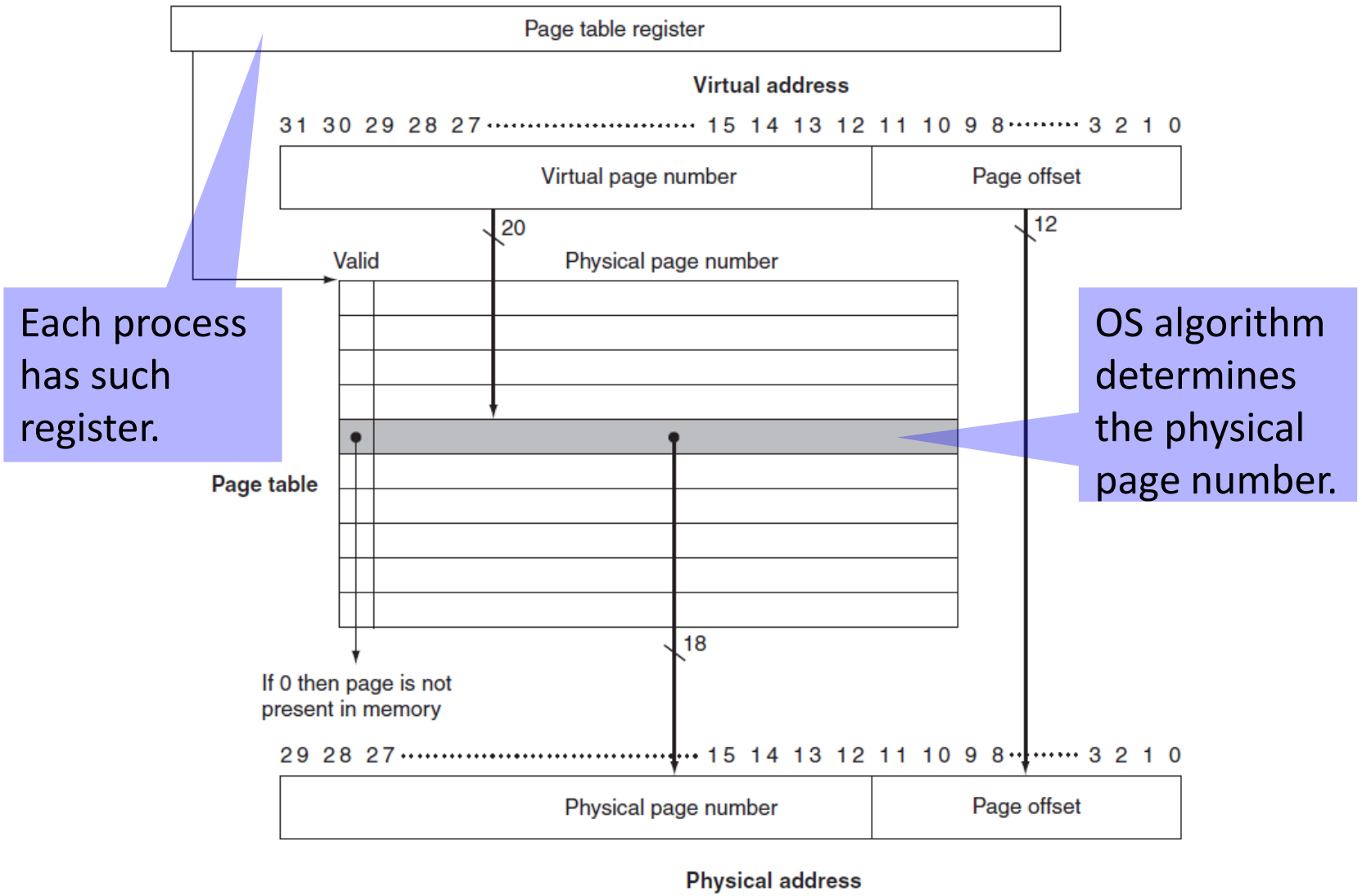
Program's page table location in memory is stored in **page table register** (HW) pointing its start address.

Page table + PC + ISA Regs specify the **state** of a program (process).

When OS executes another program on CPU (**active process**) the state of former program (**inactive process**) is saved.

Enables the suspended program to resume execution by loading its state in CPU, including PC.

OS is handling the program swap.



Each process has such register.

OS algorithm determines the physical page number.

The process's main memory address space is defined by its page table, also residing in main memory.

OS loads only the page table register, pointing the process's associated page table.

OS allocates the physical memory and updates the page tables.

It avoids **collision** of the virtual address spaces of different processes.

Separate page tables provide **protection** of one process from another.

OS creates the space on **disk** for all the pages of a process when it creates the process.

This disk space is called the **swap space**.

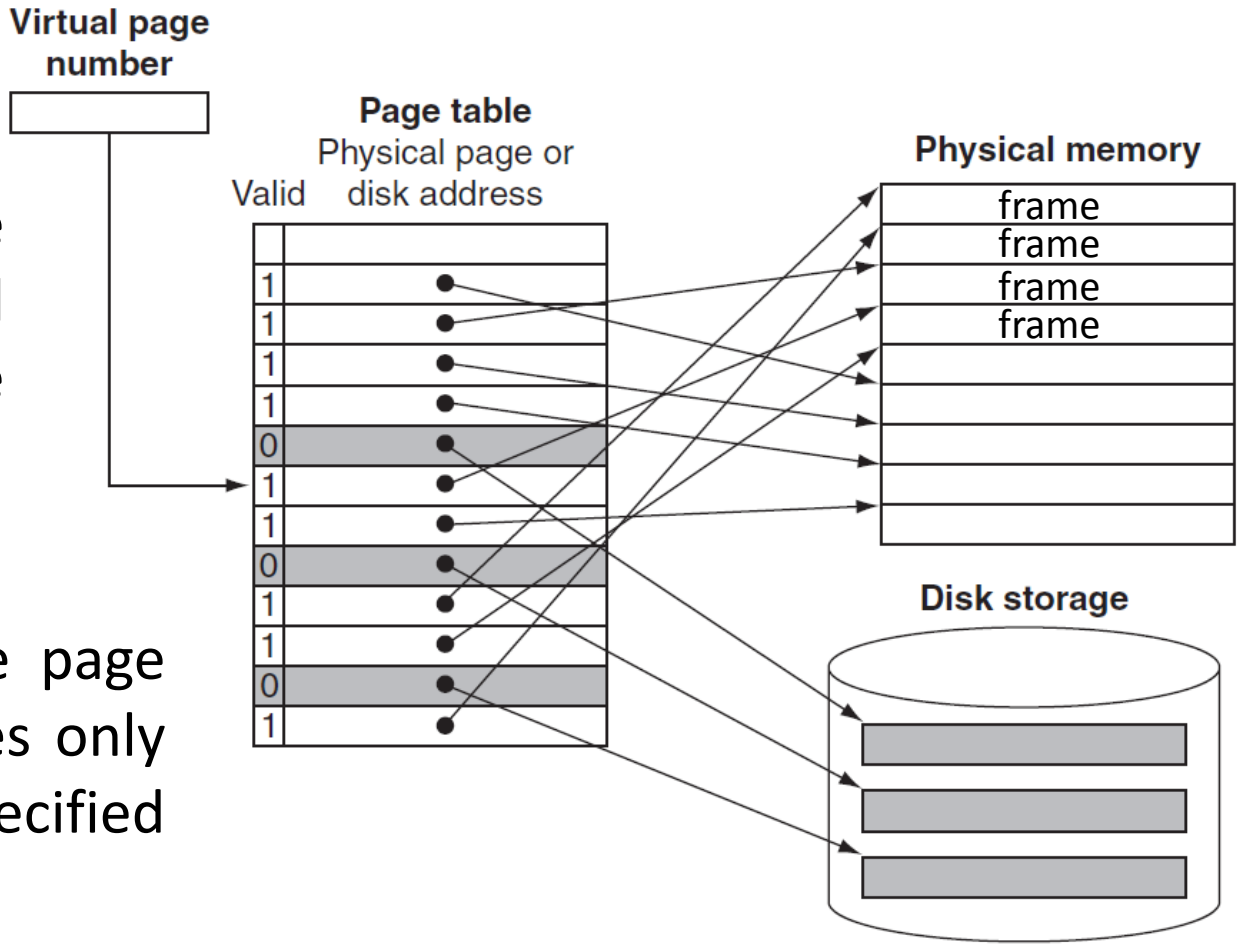
OS also creates a data structure to record where each virtual page is **stored on disk**.

It may be part of the page table or an auxiliary data structure indexed in the same way as the page table.

The page table maps each VM page to either a page in main memory or a page stored on disk.

Virtual page number is used to index the page table.

Valid bit 0: the page currently resides only on disk, at a specified disk address.



OS handles data structure to track which processes and which virtual addresses use each physical page.

OS is another process, and these tables also reside in main memory.

When all the pages in main memory are in use, the OS chooses a page to replace. Replaced pages are written to swap space on the disk.

LRU minimizes page faults but too expensive, requiring update of data structure on **every memory reference**.

Approximation is used instead.

Set a **Use bit** (**reference bit**) (in HW) whenever a page is accessed.

OS periodically clears use bits, determining which pages were touched during a certain time period.

OS then evicts a page whose reference bit is off.

With 32-bit virtual address, 4 KB page size, and 4 bytes per page table entry, the total page table size is: # page table entries = $2^{32} / 2^{12} = 2^{20}$.

Size of page table = 2^{20} x 4bytes = 4MB **(for every process! There may be 100's!)**

Writes in Virtual Memory

Access time between cache and main memory is 10s – 100s cycles. Write-through with write buffer hided this latency.

In a VM writes to disk take millions CPU cycles, so write-through is impractical.

Write-back, called **copy back**, is copying the page back to disk when it is replaced in the memory.

Disk **transfer time** is small compared to **access time**, so copy back is far more efficient than write-through.

A write-back is still costly.

We would like to know whether at a replacement the page needs to be copied back.

A **dirty bit** is added to the page table, being set when any word in a page is written.

Dirty bit indicates whether the page should be written to disk before its location in memory is given to another page.

Hierarchical Paging

Modern computer support logical address space of 2^{32} to 2^{64}, making page table excessively large.

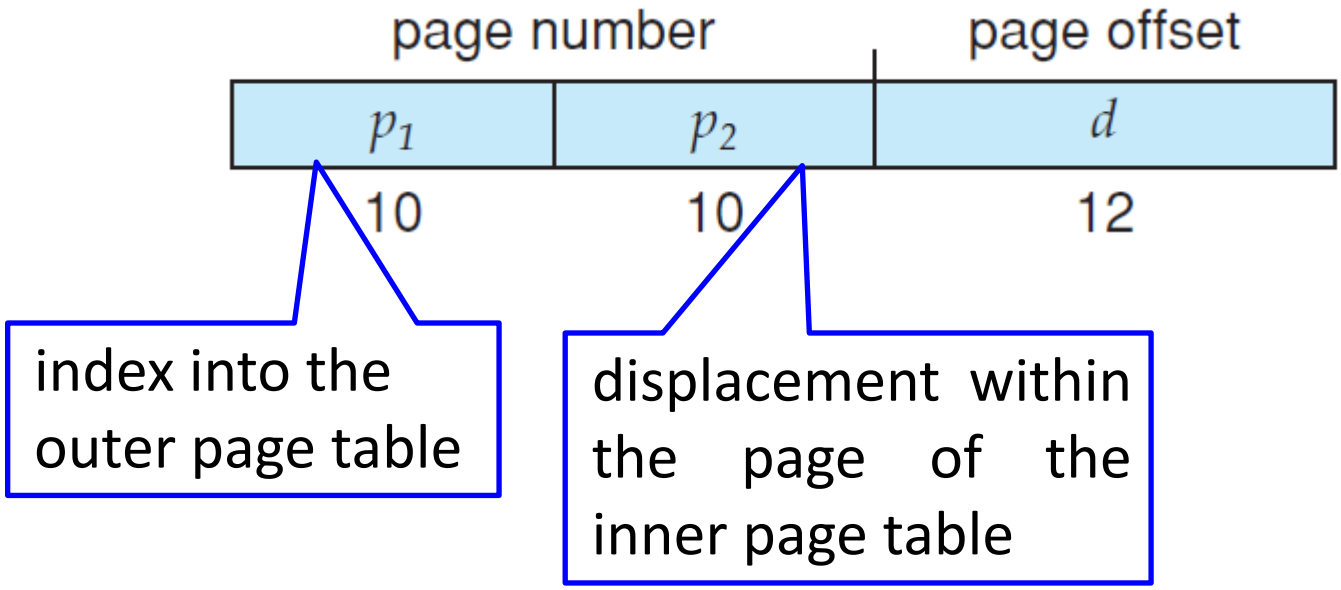
Example: In 32-bit address with 4KB (2^{12}) page size, page table may have $2^{32}/2^{12} = 1\text{M}$ entries, 4byte each, yielding 4MB physical space for a single page table. ■

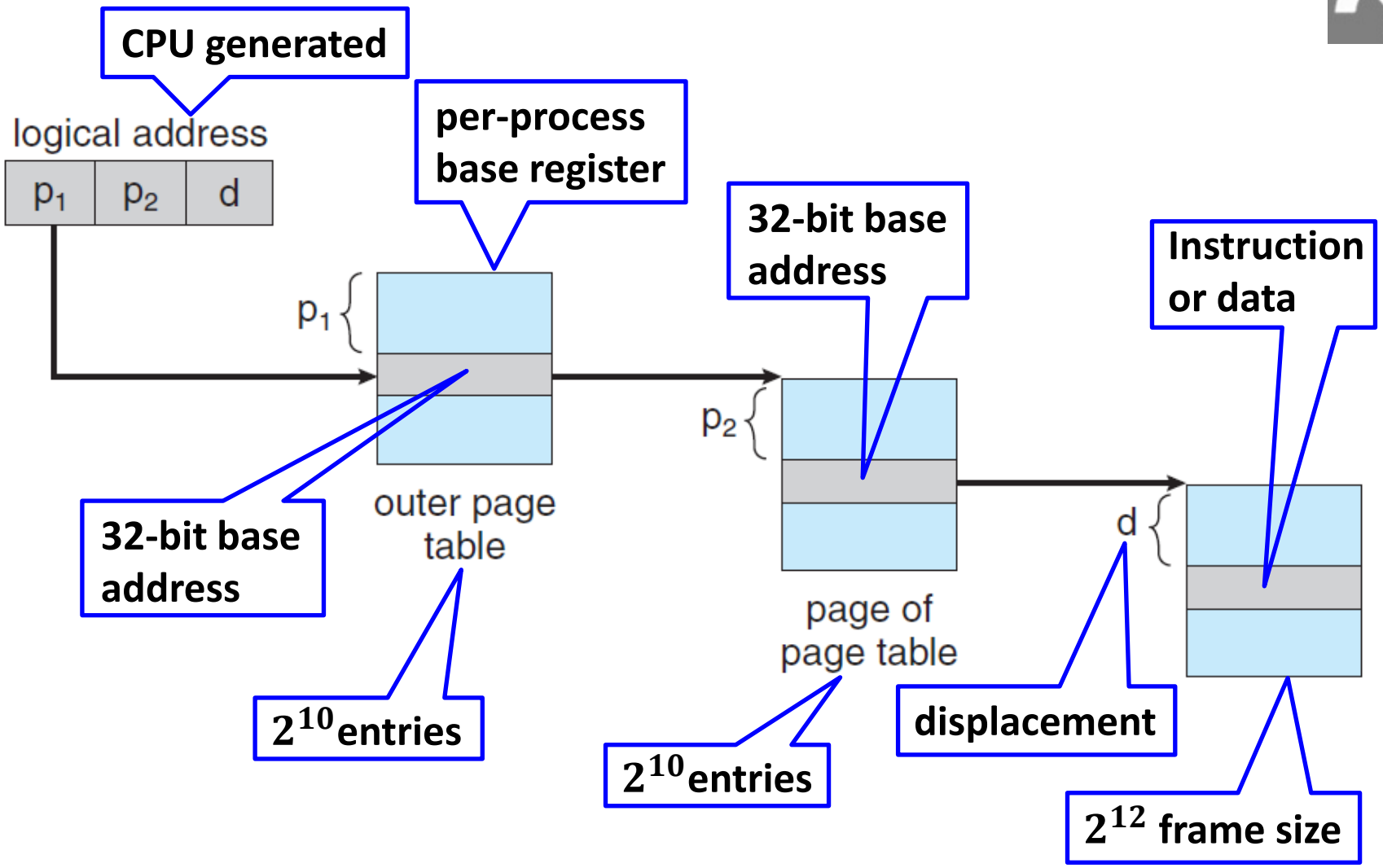
We do not want to allocate the page table **contiguously** in main memory.

A solution is to use a two-level paging algorithm, in which the page table itself is also paged.

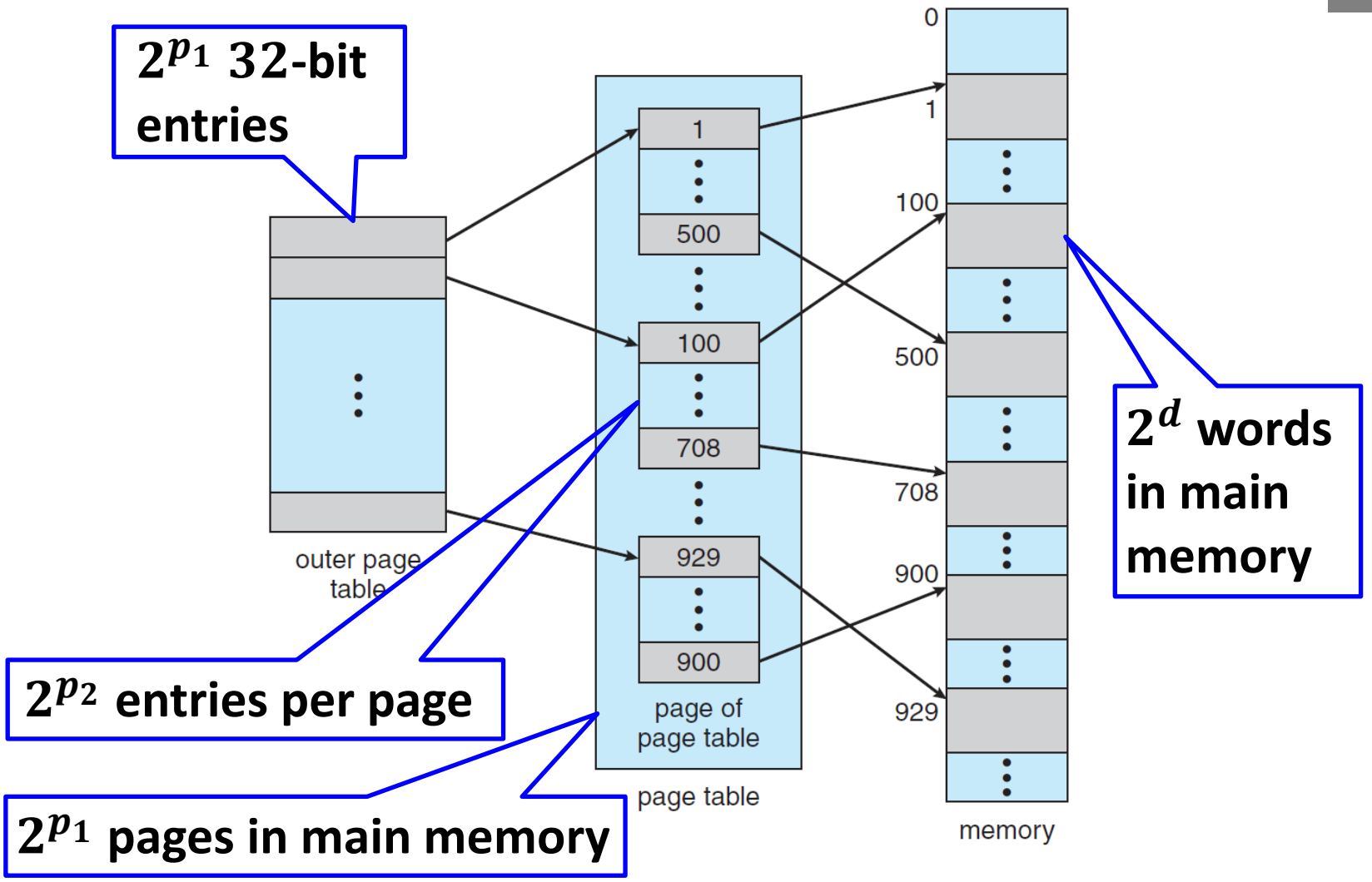
Example: 32-bit logical address and 4KB page size. Logical address is divided into 20-bit page # and 12-bit page offset.

The page number is further divided into 10-bit **outer part** and 10-bit **inner part**. ■





Address translation for a two-level 32-bit paging.



Two-level page-table (forward-mapped page table).

TLB: Fast Address Translation

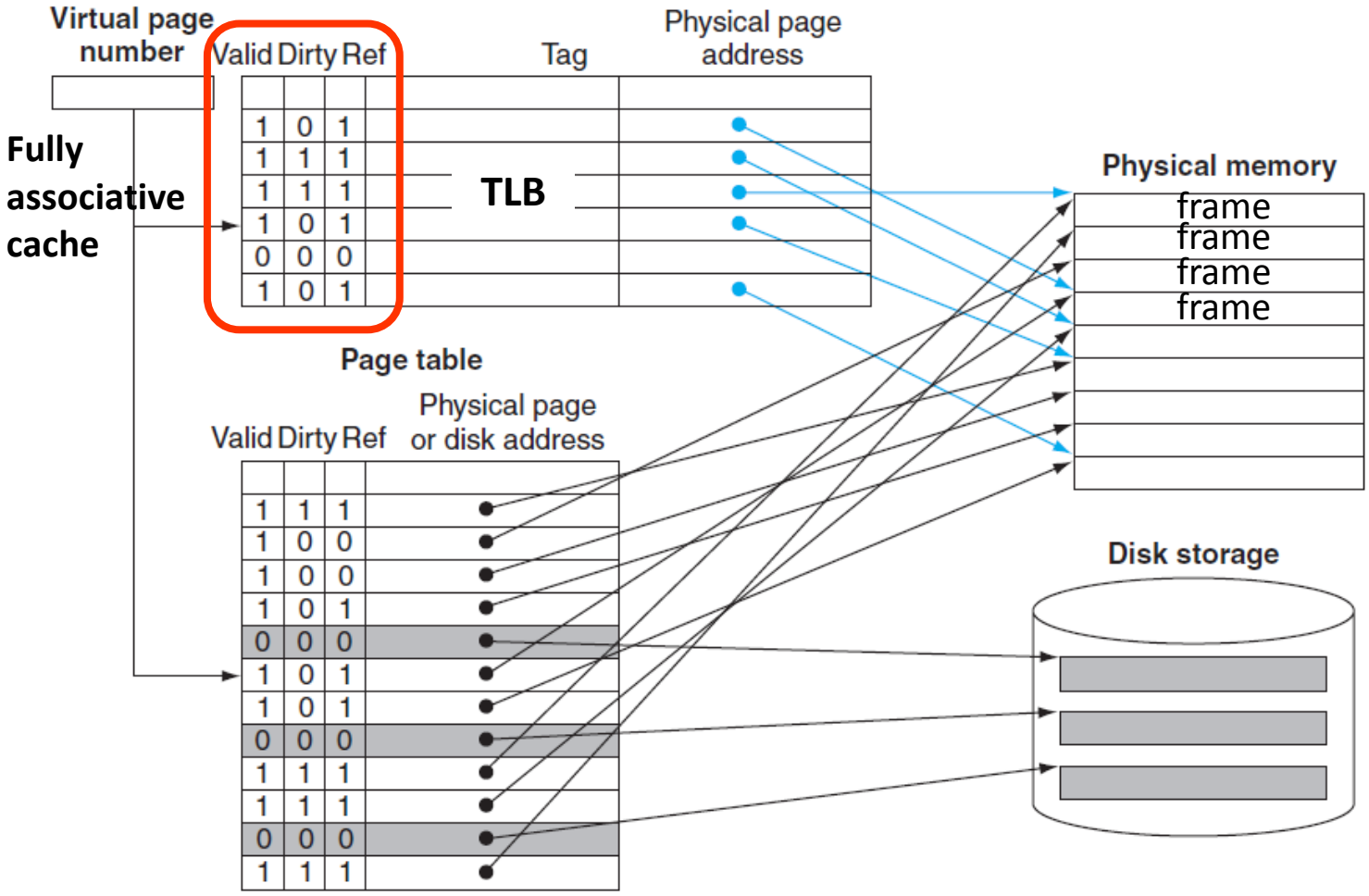
Since page tables are stored in main memory, every memory access by a program is twice long:

- access to get physical address (from page table),
- access to get the data (elsewhere in memory).

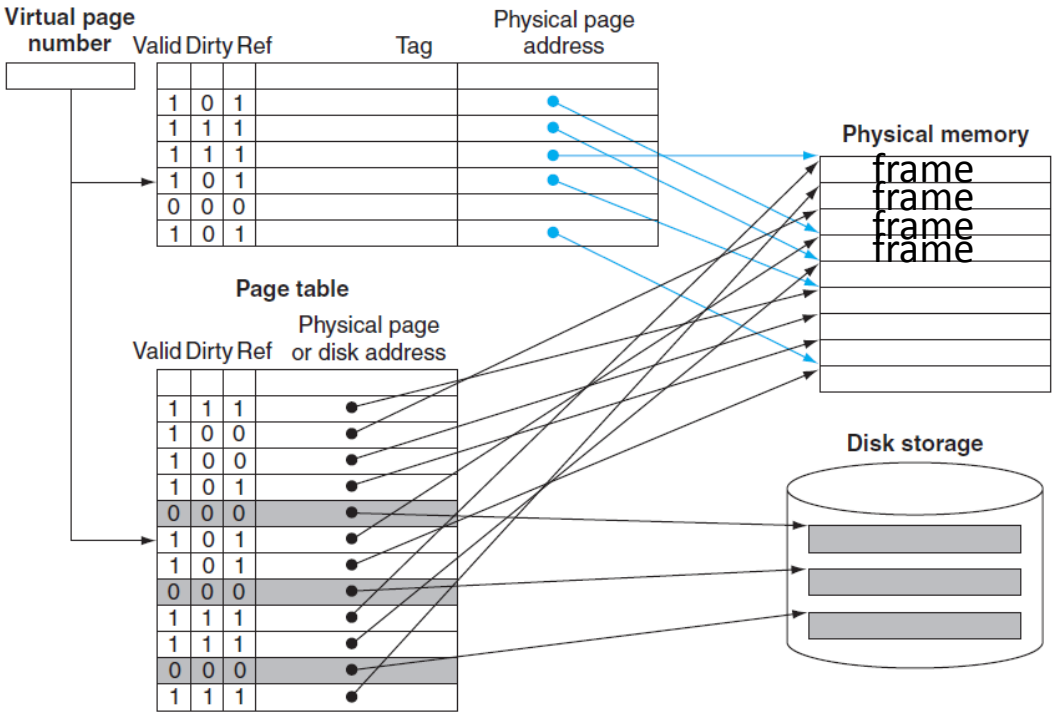
Locality of reference to page table can help:

- Since words of that page have **temporal** and **spatial** locality, same virtual-physical translation will be needed soon.

A special **cache**, called **Translation Look-aside Buffer** (**TLB**), keeps track of recently used translations.



Access TLB on every reference instead of page table. TLB must therefore include the **valid**, **dirty** and the **reference** bits.



Every reference looks up the virtual page # in TLB.

TLB hit uses the physical page # to form the address and turns on the reference bit. Write turns on the dirty bit too.

TLB miss can be either a **true** page fault or just a **TLB miss**.

If the page exists in memory, the processor loads the translation from the page table into the TLB and tries the reference again.

True page fault is handled by the OS (exception).

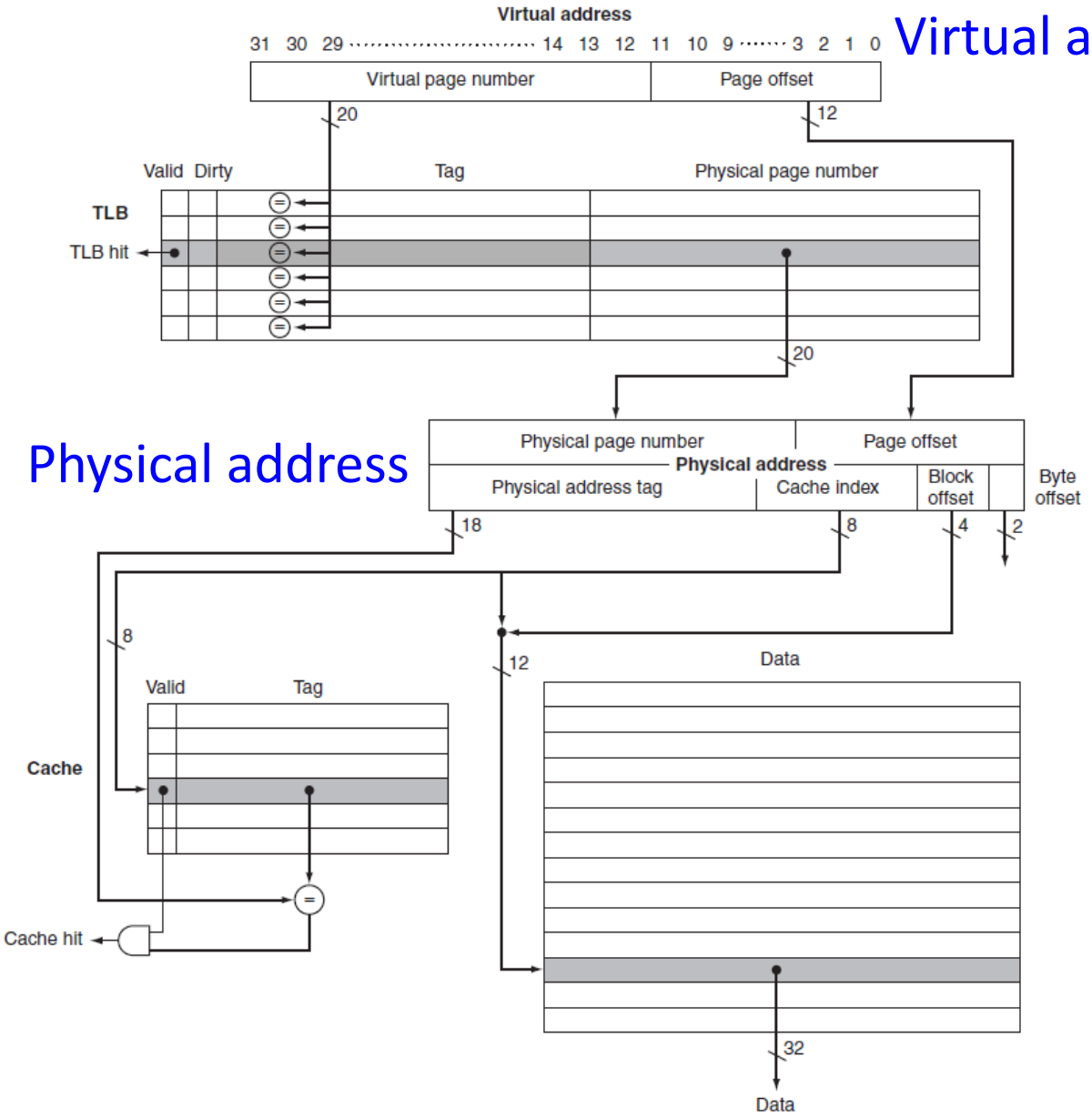
TLB has far fewer entries than pages are in main memory. Hence TLB misses are much more frequent than page faults.

TLB miss need to select a TLB entry to replace and copy the **reference** and **dirty** bits back to the page table entry.

These are the only TLB entry portion that can be changed.

Typical TLB parameters

- TLB size: 16–512 entries
- Block size: 1–2 page table entries (4–8 bytes)
- Hit time: 0.5–1 clock cycle
- Miss penalty: 10s–100s clock cycles
- Miss rate: 0.01%–1%



Virtual address

TLB

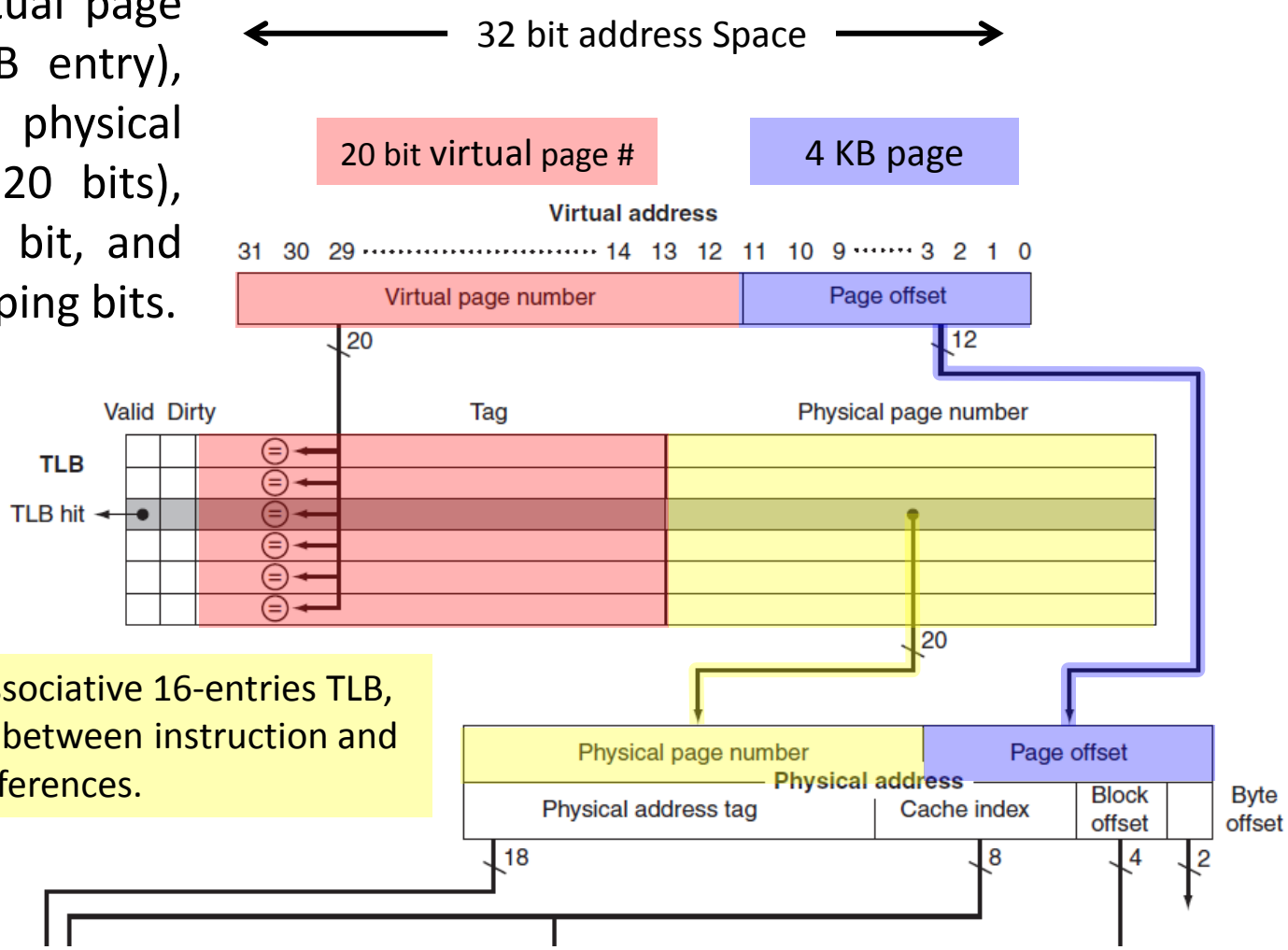
Physical address

Real TLB Read

cache

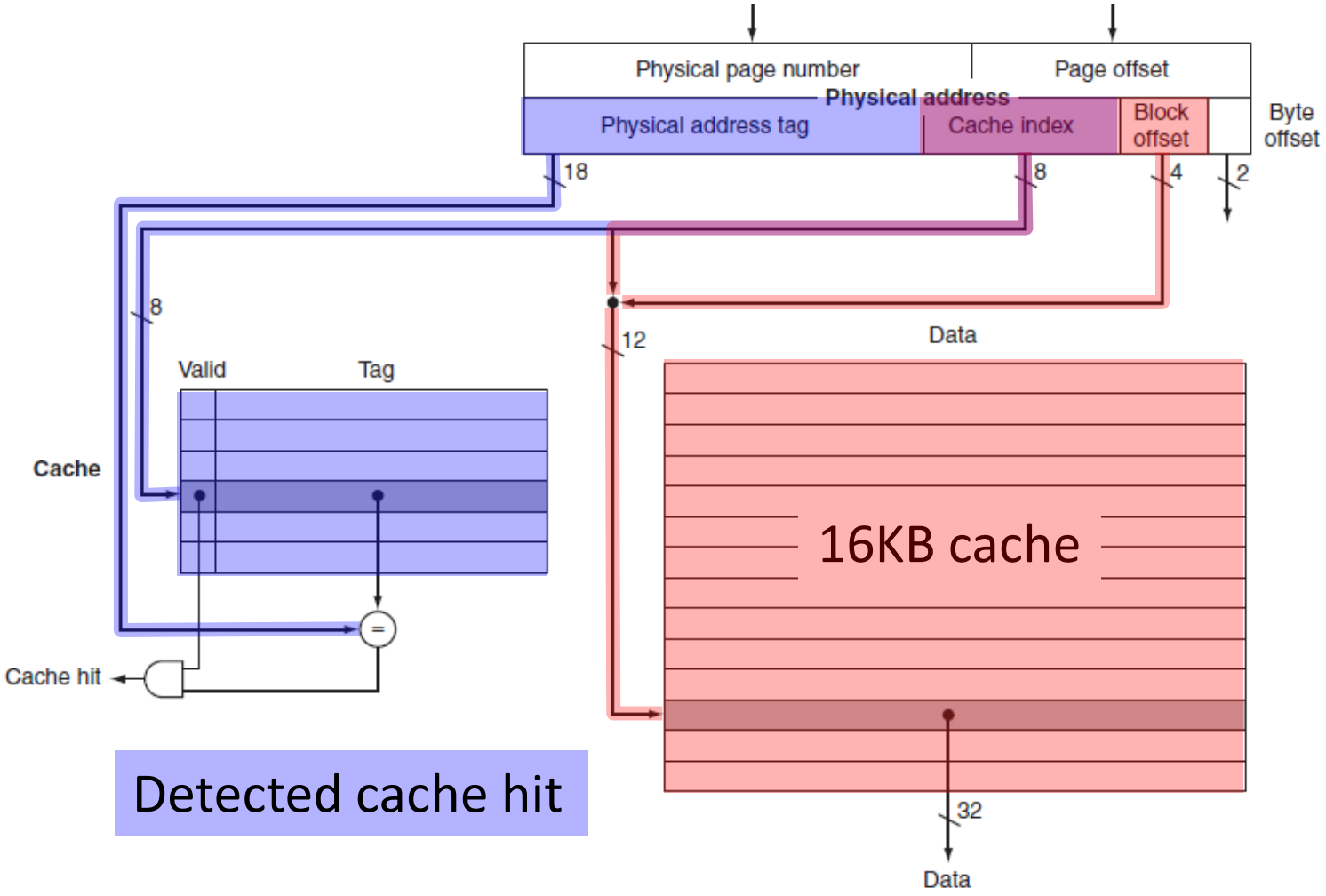
TLB entry is 64 bits:

20-bit tag (virtual page # for that TLB entry), corresponding physical page # (also 20 bits), valid bit, dirty bit, and other bookkeeping bits.



Fully associative 16-entries TLB, shared between instruction and data references.

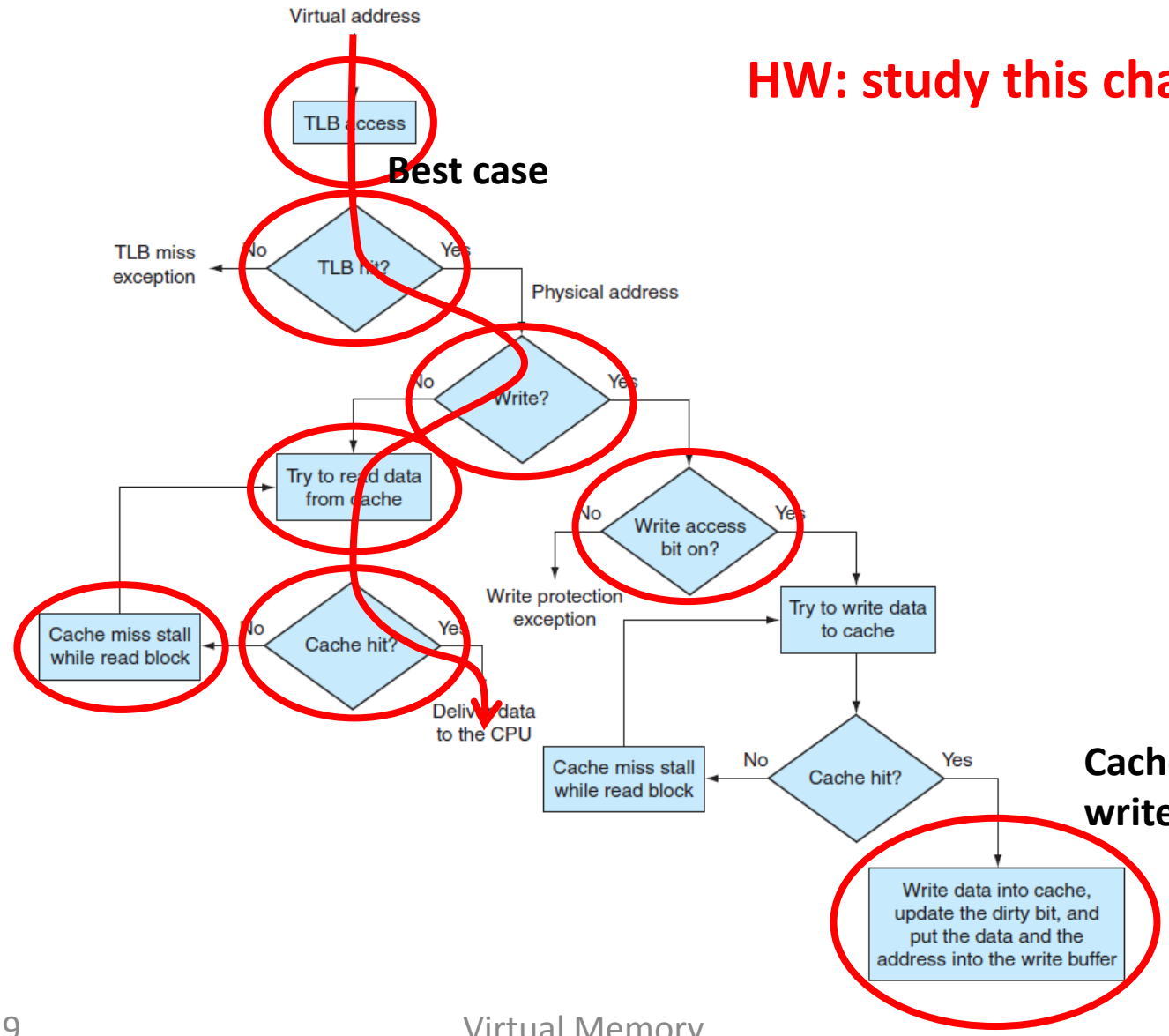
physical address generation



Detected cache hit

read from cache

Read and Cache Write-Through



HW: study this chart.

Cache write-through

VM, TLBs and Caches Integration

VM and cache work together as a hierarchy. Data must be in main memory if it is in the cache.

OS maintains this hierarchy by flushing the contents of any page from the cache migrating it to disk.

OS modifies the page tables and TLB accordingly.

In **best case**, a virtual address is translated by the TLB and sent to the cache where the appropriate data is found, retrieved, and sent back to the processor.

In the **worst case**, a reference can miss all: the TLB, the page table, and the cache.

Consider all the seven combinations of the three events. State for each whether it can actually occur and under what circumstances.

HW: all the cases below.

TLB	Page table	Cache	Possible? If so, under what circumstance?
hit	hit	miss	Possible, although the page table is never really checked if TLB hits.
miss	hit	hit	TLB misses, but entry found in page table; after retry, data is found in cache.
miss	hit	miss	TLB misses, but entry found in page table; after retry, data misses in cache.
miss	miss	miss	TLB misses and is followed by a page fault; after retry, data must miss in cache.
hit	miss	miss	Impossible: cannot have a translation in TLB if page is not present in memory.
hit	miss	hit	Impossible: cannot have a translation in TLB if page is not present in memory.
miss	miss	hit	Impossible: data cannot be allowed in cache if the page is not in memory.