

Εικονική Μνήμη (Virtual Memory)

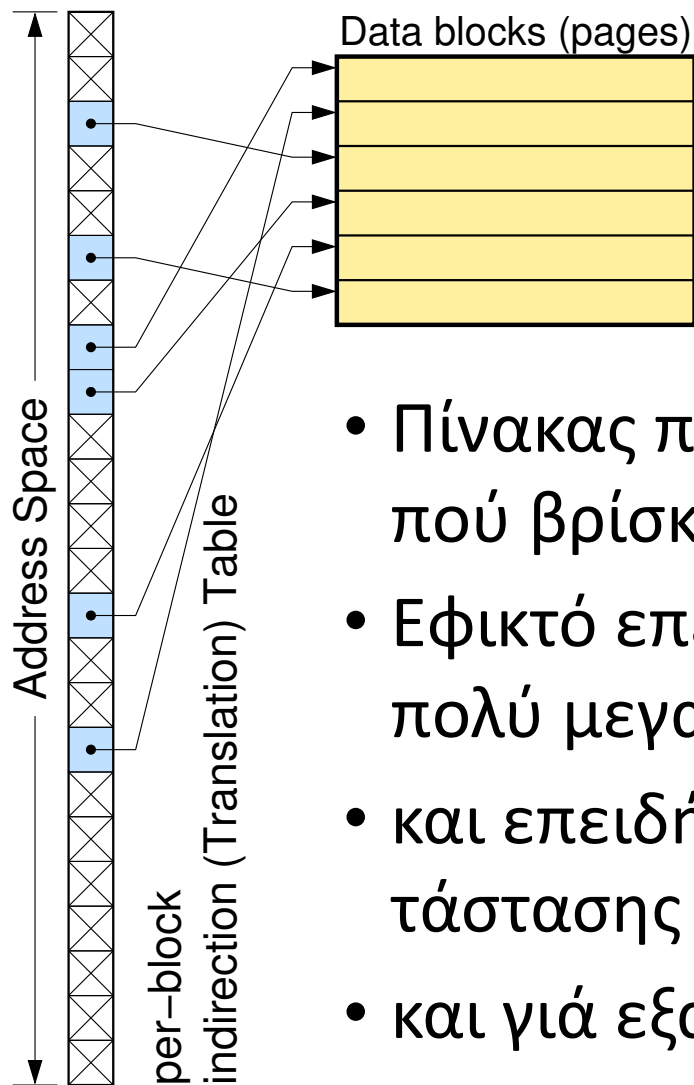
12α (§12.1-7) – 24-26 Απριλίου 2023 – Μανόλης Κατεβαίνης

© copyright University of Crete – <https://www.csd.uoc.gr/~hy225/23a/copyright.html>

except those marked as copyrighted by Morgan Kaufmann

Στόχοι Εικονικής Μνήμης: 3 σε 1

1. Εικονική Μηχανή / Προστασία (Virtual Machine – VM, Protection): κάθε Διεργασία (*Process*) νομίζει ότι έχει όλη τη μηχανή (το χώρο διευθύνσεων) δική της, ανεξάρτητα (προστατευμένη) από τις άλλες
 2. Επίπεδο Ιεραρχίας Μνήμης πριν την Αποθήκευση/Δίσκο
 3. Επίλυση *Fragmentation*: ο διαθέσιμος χώρος μνήμης για νέα διεργασία είναι κομματιασμένος, επειδή άλλες διεργ. μικρές και άλλες μεγάλες (“working set”)
- Διαχείριση από Λειτουργικό Σύσ. με βοήθεια από το Υλικό

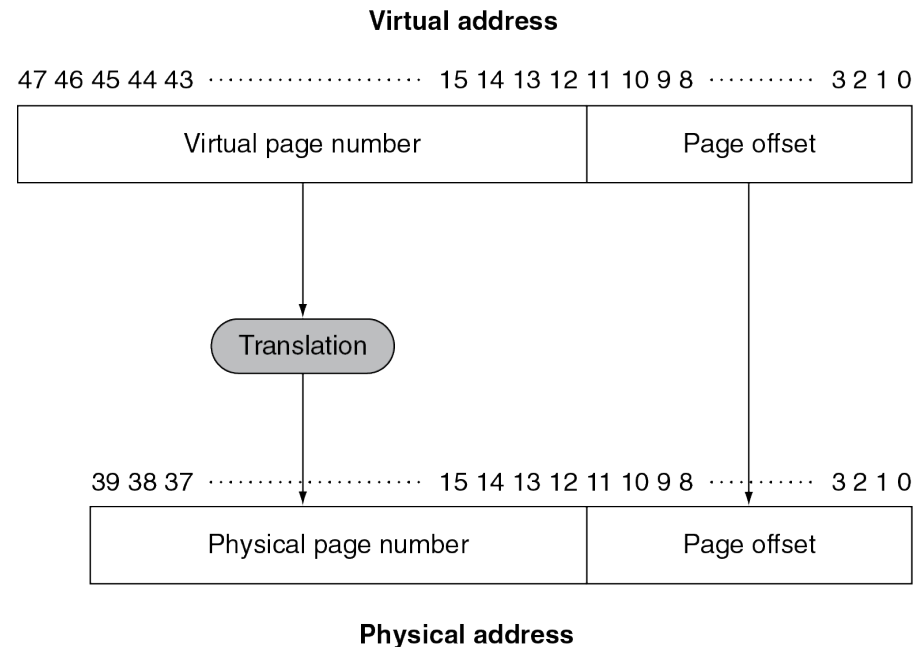
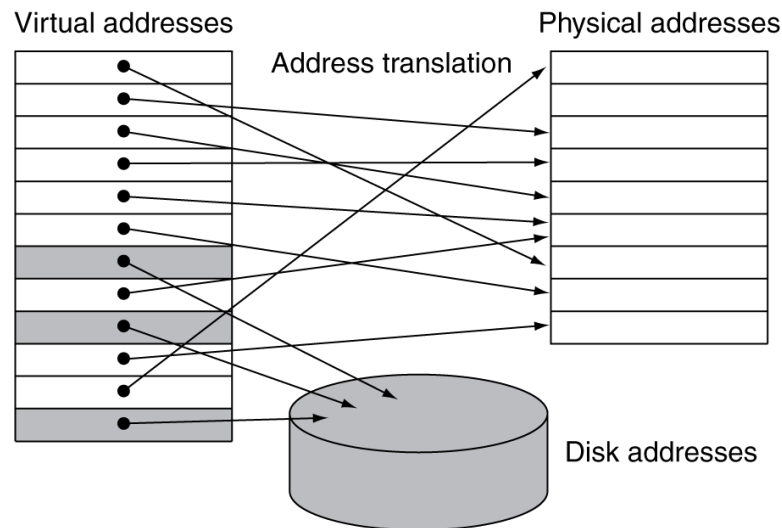


Γενική δομή Εικονικής Μνήμης

- Σε αντίθεση με την οργ. των κρυφών μνημών, όπου ψάχναμε στις υποψήφιας θέσεις για την επιθυμητή γραμμή
- Πίνακας που δείχνει, για κάθε block ("page"), πού βρίσκεται στην (μικρότερη) Φυσική Μνήμη
- Εφικτό επειδή εδώ τα blocks («σελίδες») είναι πολύ μεγαλύτερα από τις γραμμές κρυφών μν.
- και επειδή η διαχείριση τοποθέτησης / αντικατάστασης είναι σε λογισμικό (Λειτουργικό – OS)
- και για εξαιρετ. Μικρό miss rate (~fully assoc.)

Address Translation

- Fixed-size pages (e.g., 4K)

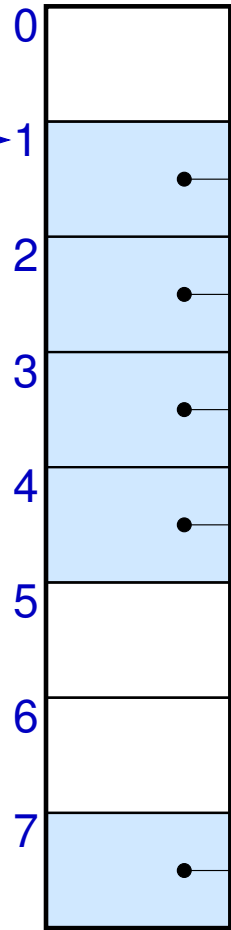


Page size is power of 2, all pages aligned on their natural boundaries
==> page offset = address mod page-size = LS address bits,
page number = integer quotient... = MS address bits

Page Fault Penalty

- On page fault, the page must be fetched from disk
 - Takes millions of clock cycles
 - Handled by OS code
- Try to minimize page fault rate
 - Fully associative placement
 - Smart replacement algorithms

Process A Virtual Memory

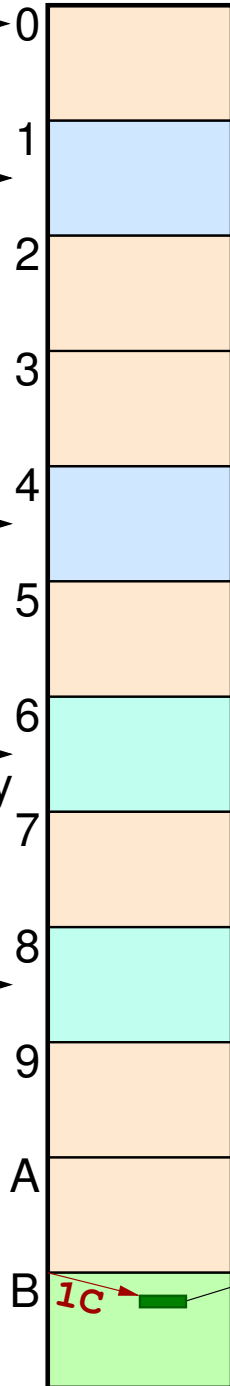


Page Table of A

	V	Prot	D	R	Phy.Pg#
0	0	---			
1	1	--x			8
2	1	rw-			1
3	0	r--			<disk addr>
4	1	-w-			6
5	0	---			
6	0	---			
7	1	rw-			4

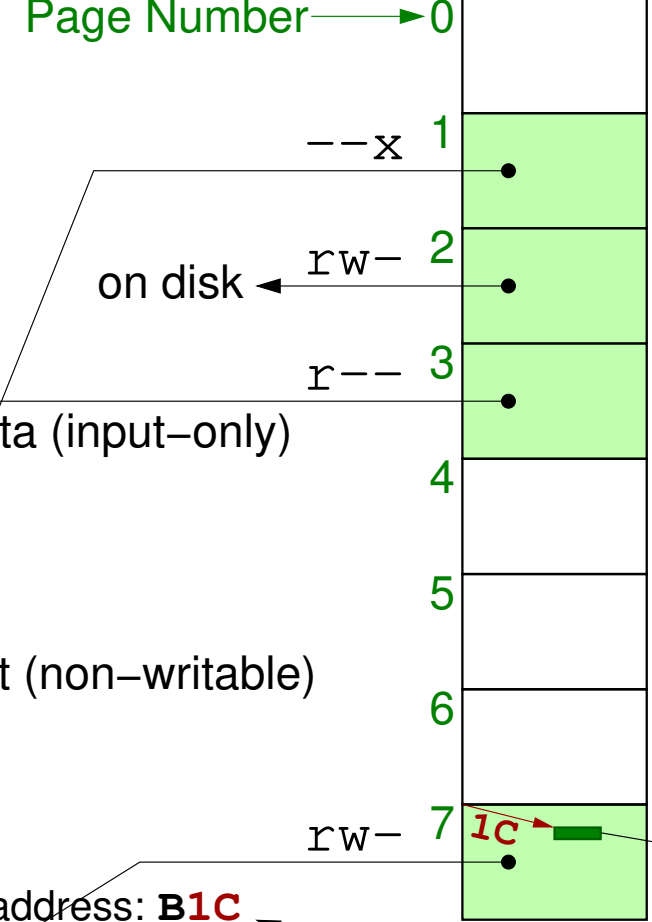
Physical
Page Number

Physical Memory



Two Processes, A and B, instances of a same program, isolated from each other, except for a shared data page

Process B Virtual Memory



physical address: **B1C**

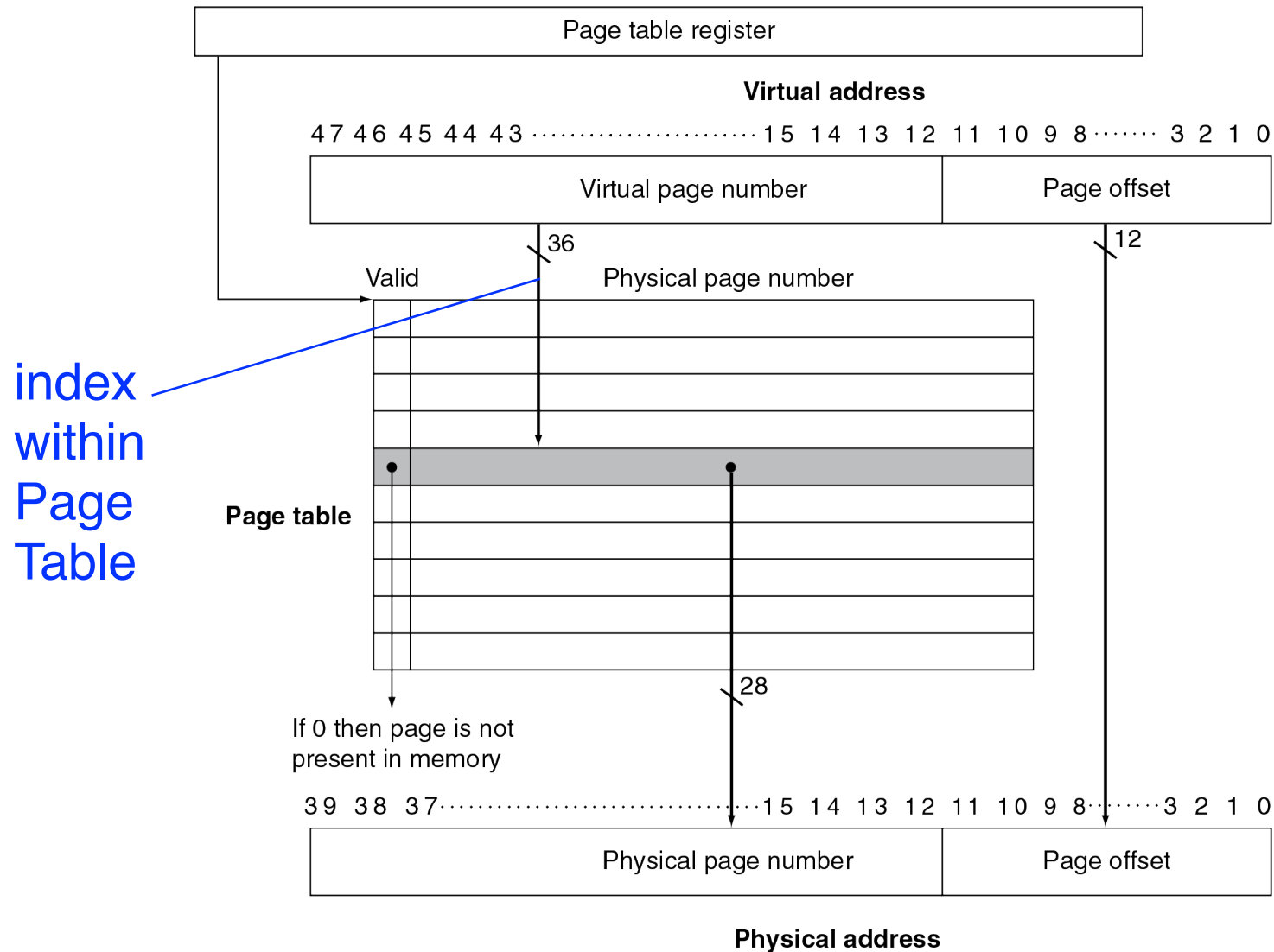
virtual address: **71C**

1C is the offset-within-page

Page Tables (per Process!)

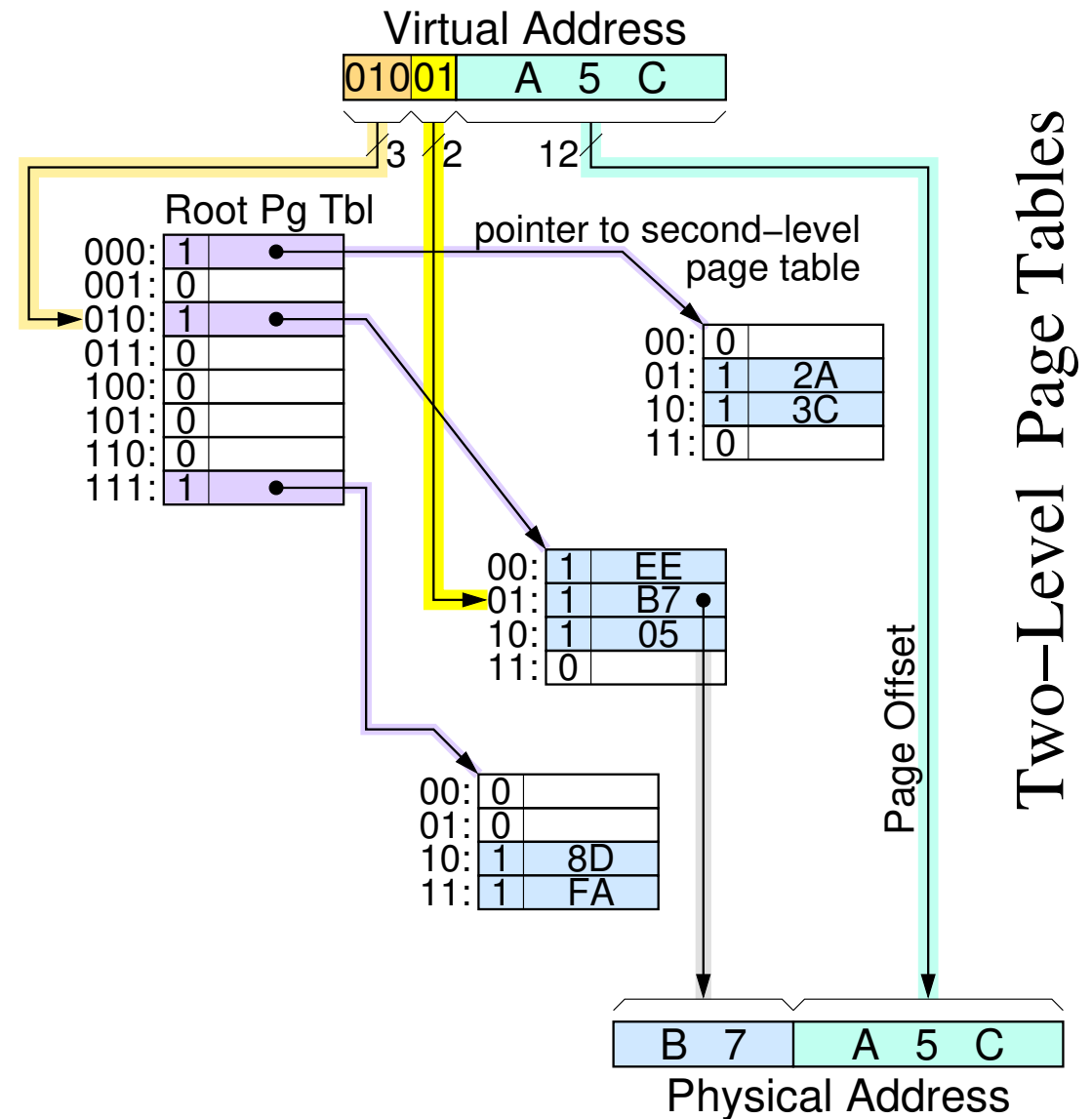
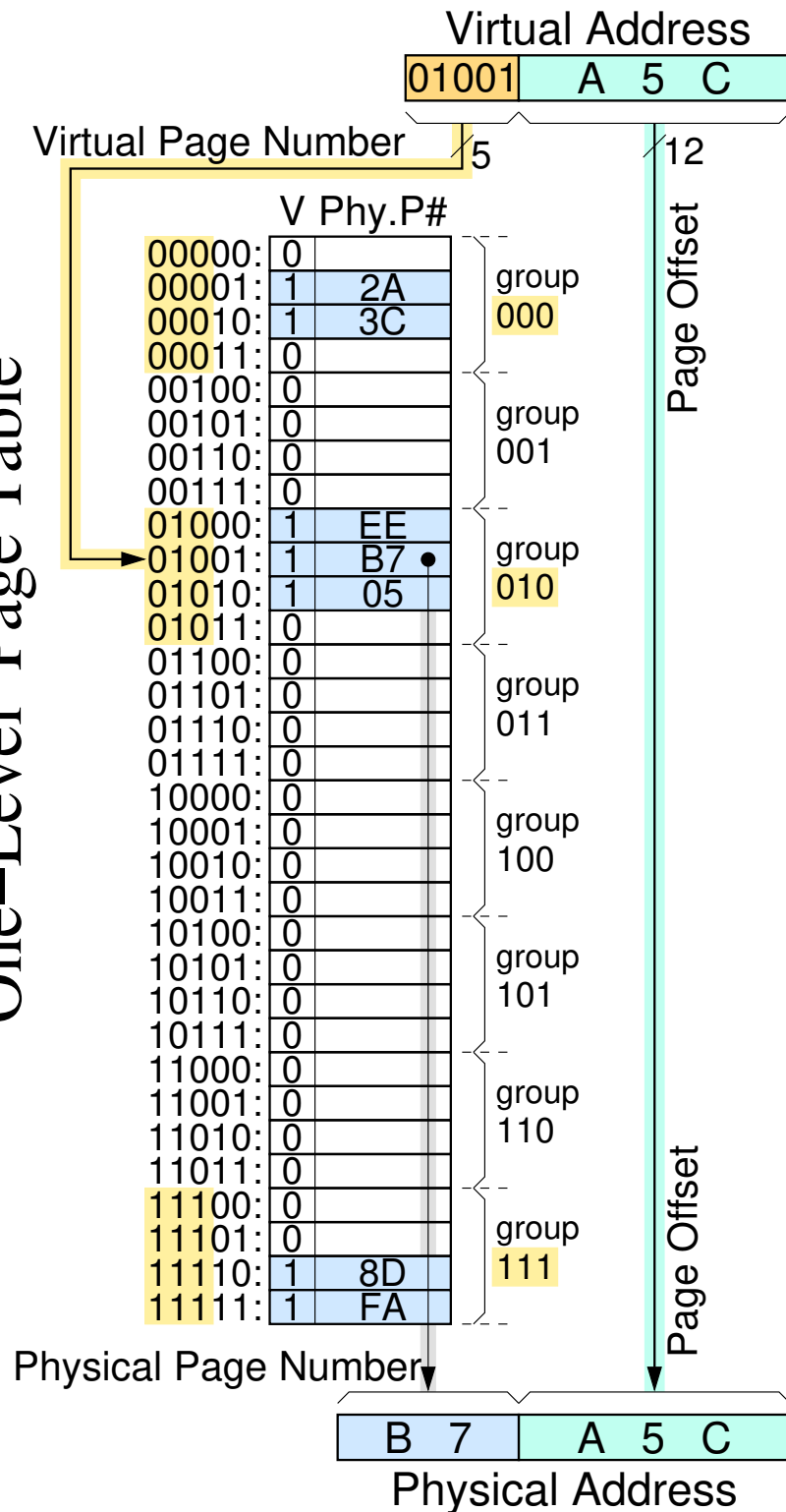
- Stores placement information
 - Array of page table entries, indexed by virtual page number
 - Page table register in CPU points to page table in physical memory (to the page table of the currently running process!)
- If page is present in memory
 - PTE stores the physical page number
 - Plus other status bits (referenced, dirty, ...)
- If page is not present
 - PTE can refer to location in swap space on disk

Translation Using a Page Table



Problem:
Very Large Size
of single-level
Page Table;
Solution:
Multi-Level
Page Tables.

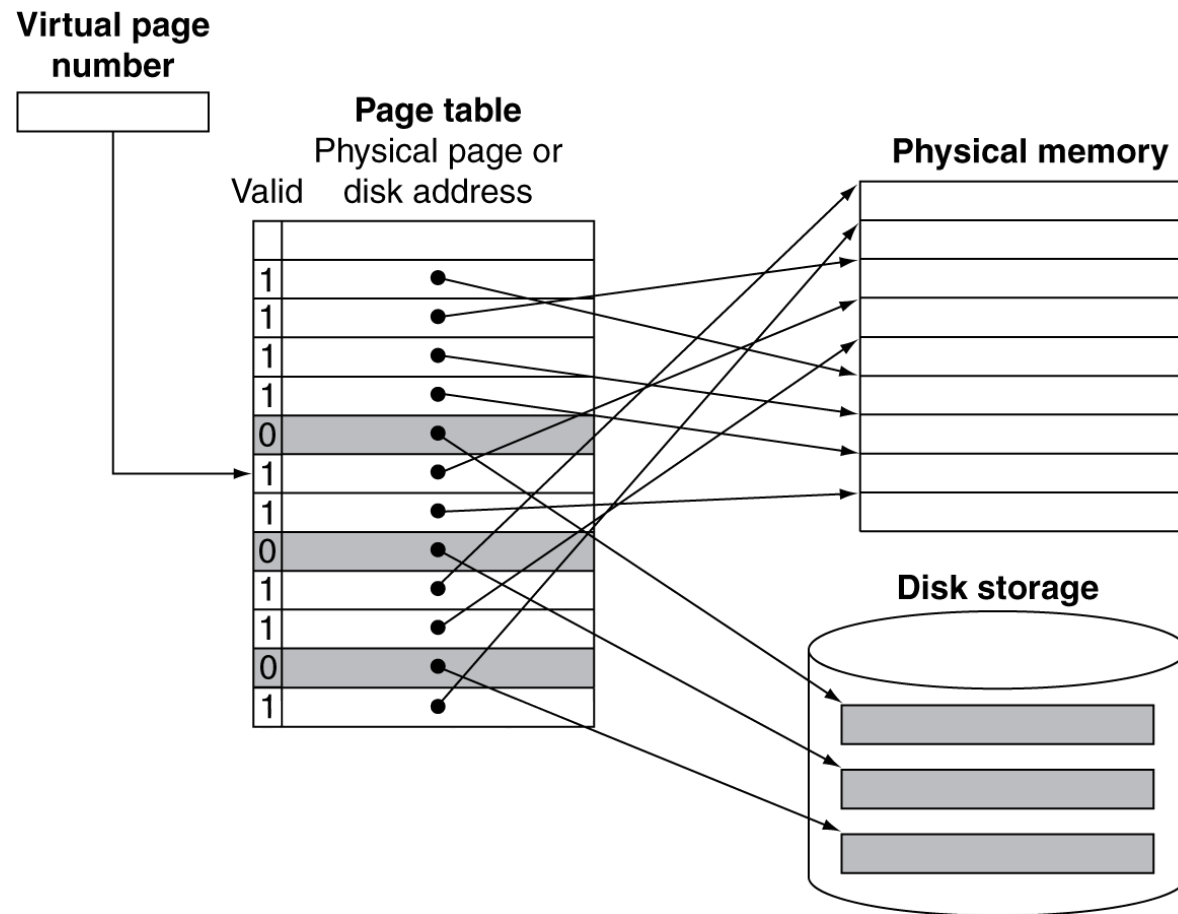
One-Level Page Table



In this example:

- Page size: 4 KBytes
- Virtual Address Space: 128 KBytes
=> 32 virtual pages per process
- Physical Address Space: 1 MByte
=> 256 physical pages

Mapping Pages to Storage



Replacement and Writes

Υπό "Κ.Σ." (δηλ. no "thrashing"), οι σελίδες LRU

- To reduce page fault rate, prefer least-recently used (LRU) replacement που διώχνουμε

- Reference bit (aka use bit) in PTE set to 1 on access to page δεν έχουν χρησιμοποιηθεί εδώ και
- Periodically cleared to 0 by OS λεπτά της ώρας
- A page with reference bit = 0 has not been used recently

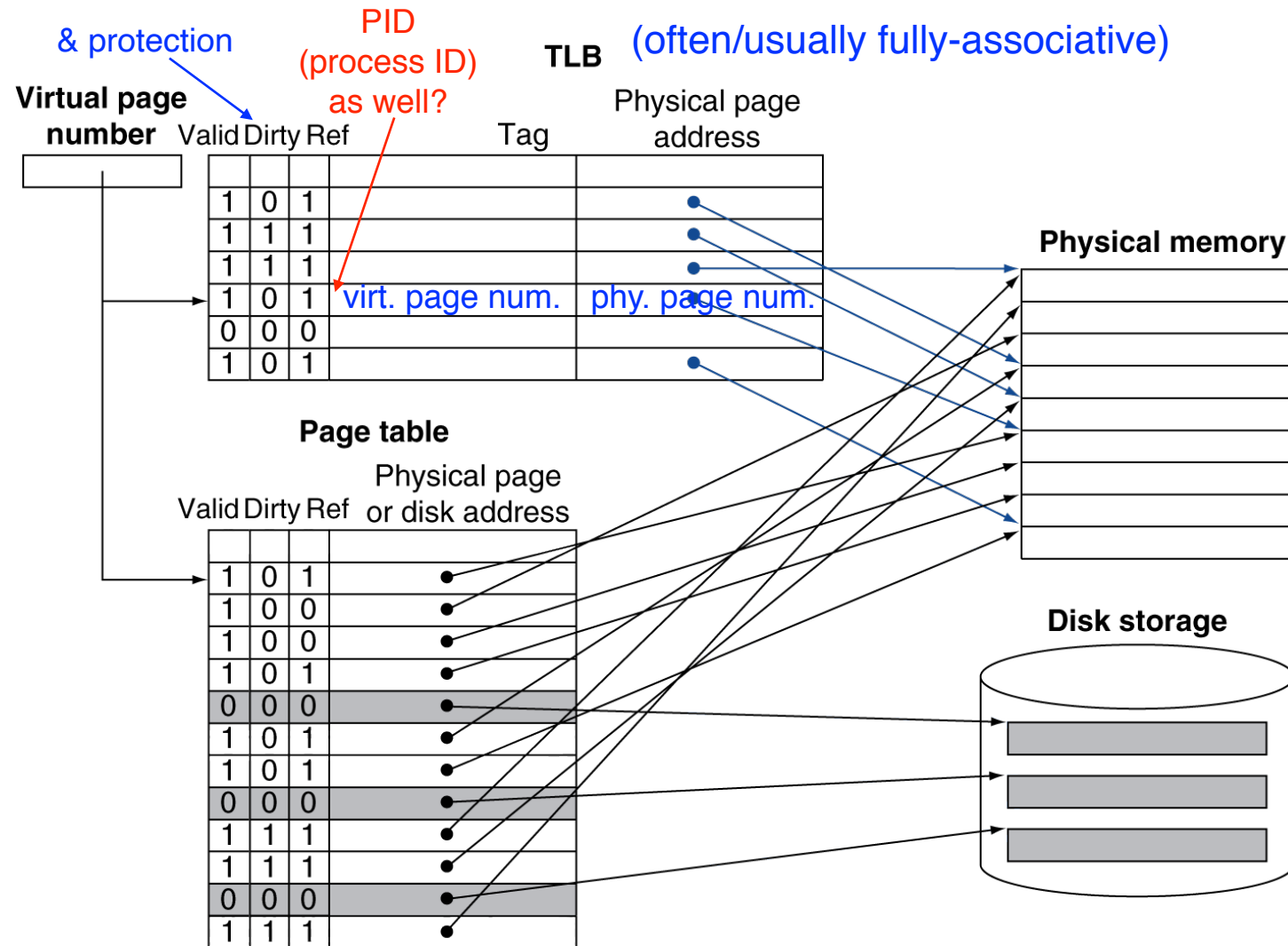
π.χ. κάθε
1 second

- Disk writes take millions of cycles
 - Block at once, not individual locations
 - Write through is impractical
 - Use write-back
 - Dirty bit in PTE set when page is written

Fast Translation Using a TLB

- Address translation would appear to require extra memory references
 - One to access the PTE (and more for multi-level tables)
 - Then the actual memory access
- But access to page tables has good locality
 - So use a fast cache of PTEs within the CPU
 - Called a Translation Look-aside Buffer (TLB)
 - Typical: 16–512 PTEs, 0.5–1 cycle for hit, 10–100 cycles for miss, 0.01%–1% miss rate
 - Misses could be handled by hardware or software

Fast Translation Using a TLB



TLB Misses

- If page is in memory
 - Load the PTE from memory and retry
 - Could be handled in hardware
 - Can get complex for more complicated page table structures
 - Or in software
 - Raise a special exception, with optimized handler
- If page is not in memory (page fault)
 - OS handles fetching the page and updating the page table
 - Then restart the faulting instruction

Page Table structure
fixed in hardware

TLB Miss Handler

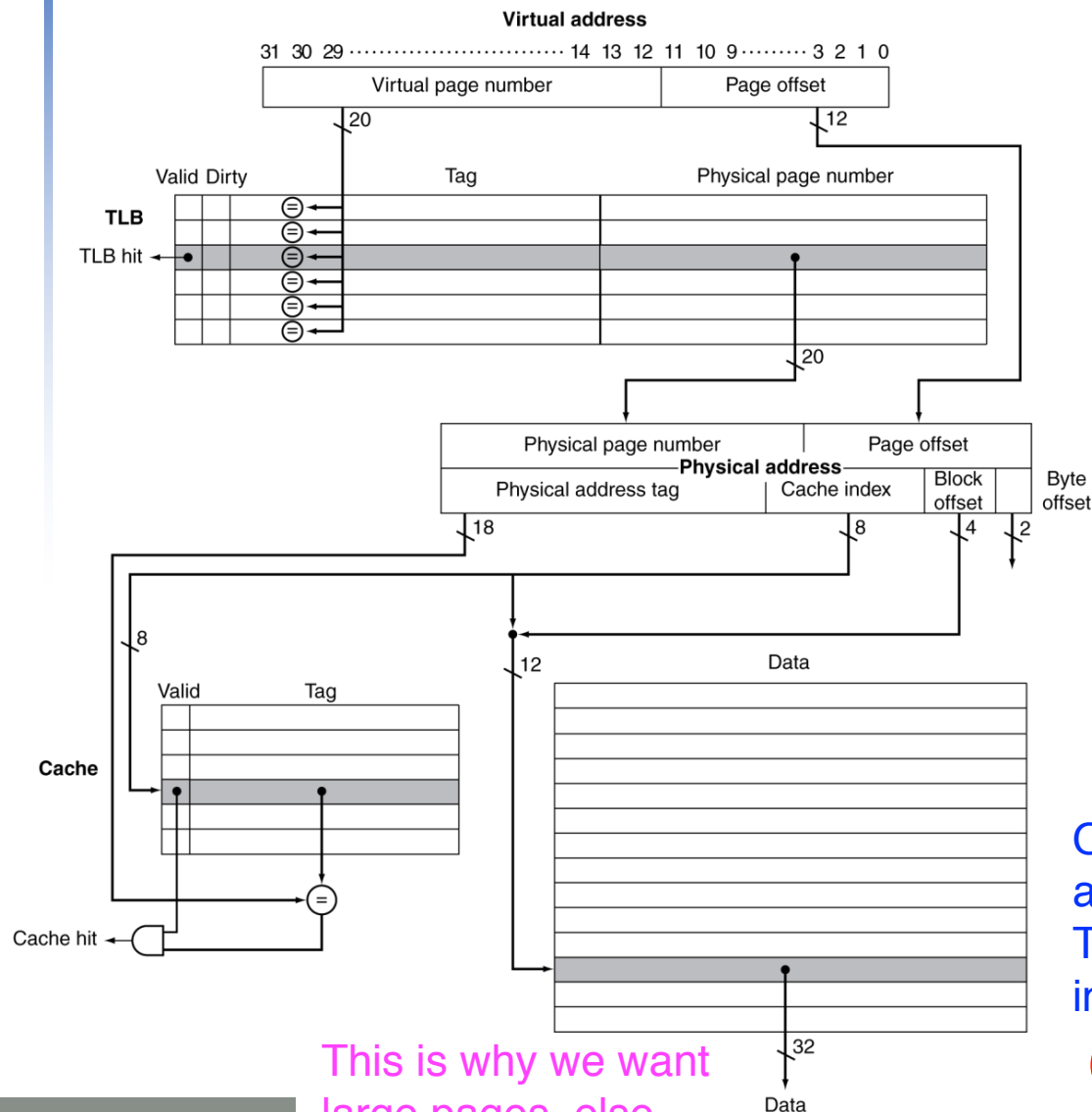
(when TLB misses
are handled in software)

- TLB miss indicates
 - ^{either} Page present, but PTE not in TLB
 - ^{or} Page not present
- Must recognize TLB miss before destination register overwritten ^(in stage 4 of our pipeline, before stage 5)
 - Raise exception
- Handler copies PTE from memory to TLB
 - Then restarts instruction
 - If page not present, page fault will occur

Page Fault Handler

- Use faulting virtual address to find PTE
- Locate page on disk
- Choose page to replace
 - If dirty, write to disk first
- Read page into memory and update page table
- Make process runnable again
 - Restart from faulting instruction

TLB and Cache Interaction



This is why we want large pages, else forced to increase associativity of L1

- If cache tag uses physical address
 - Need to translate before cache lookup
- Alternative: use virtual address tag
 - Complications due to aliasing
 - Different virtual addresses for shared physical address

Often we want: physical addr. cache, and TLB access in parallel with tag read from cache. This requires cache index to be fully contained in page offset bits, which means:

Cache Way Size $\leq$ Page Size