

Speeding Up Address Translation

Why Does it Work?

- ⦿ Spatial locality
 - program is likely to access memory locations close to each other in VA space
 - ▶ only first access to a page causes a TLB miss and Page Table access
- ⦿ Temporal locality
 - program is likely to quickly access again the same memory locations
 - ▶ if new access happens while translation still in TLB, no need to access Page Table

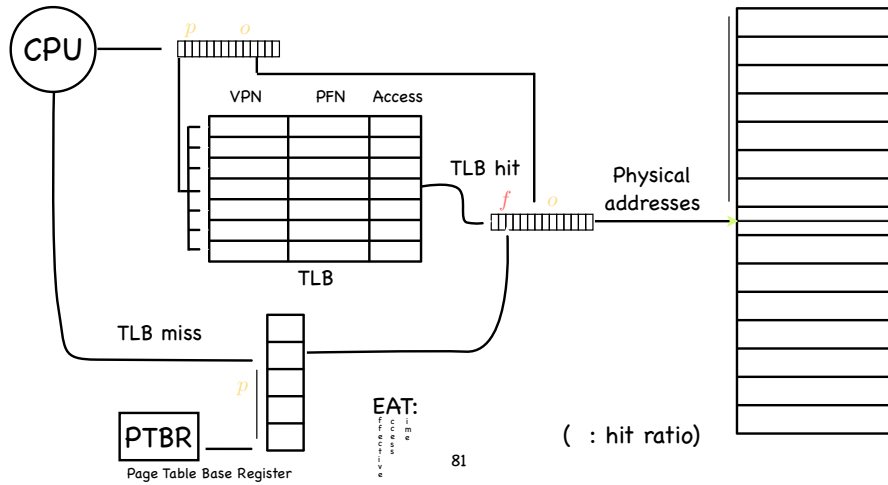
Guess What? Caching!

- ⦿ Translation-Lookaside Buffer (TLB)
 - stores for future use a successful translation between a Virtual Page Number (VPN) and a Physical Frame Number (PFN)
 - on a TLB hit, translation is achieved without accessing PT
 - on a TLB miss
 - ▶ Page Table is accessed
 - ▶ if VA is invalid, exception (segmentation fault)
 - ▶ if VA is valid, but page is not present, load page (we'll talk about it soon)
 - ▶ if VA is valid and page is present, update TLB and retry op

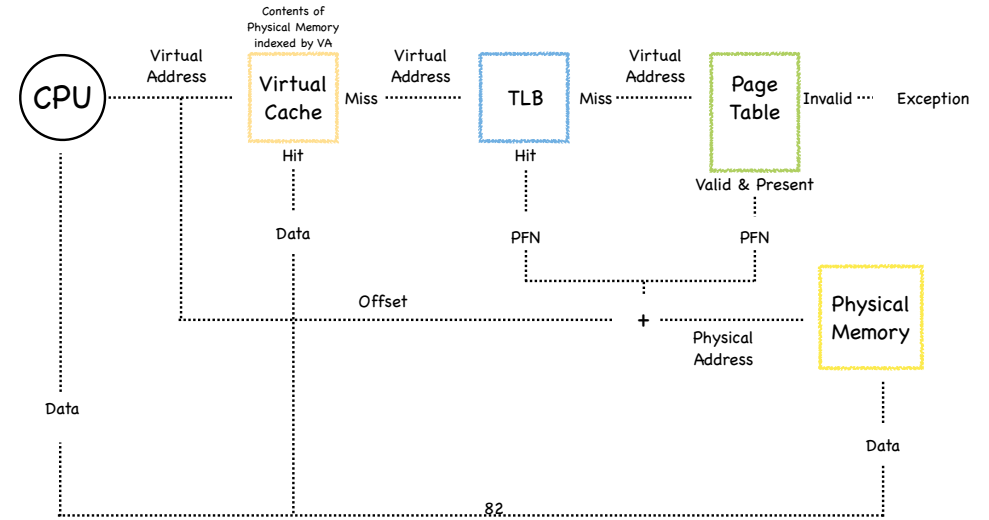
So Sorry I Missed You

- ⦿ TLB Misses can be handled in Hardware
 - HW updates TLB and retries instruction
 - HW uses PTBR to find Page Table (PT)
 - HW performs full address translation
- ⦿ TLB misses can be handled in Software
 - TLB miss causes a trap
 - HW raises an exception, moves to kernel mode, and jumps to trap handler
 - Handler goes through PT and updates TLB
 - ▶ better not trigger a TLB miss while running the handler!
 - HW returns from serving the miss with PC pointing to the instruction that caused the trap, so it is re-executed

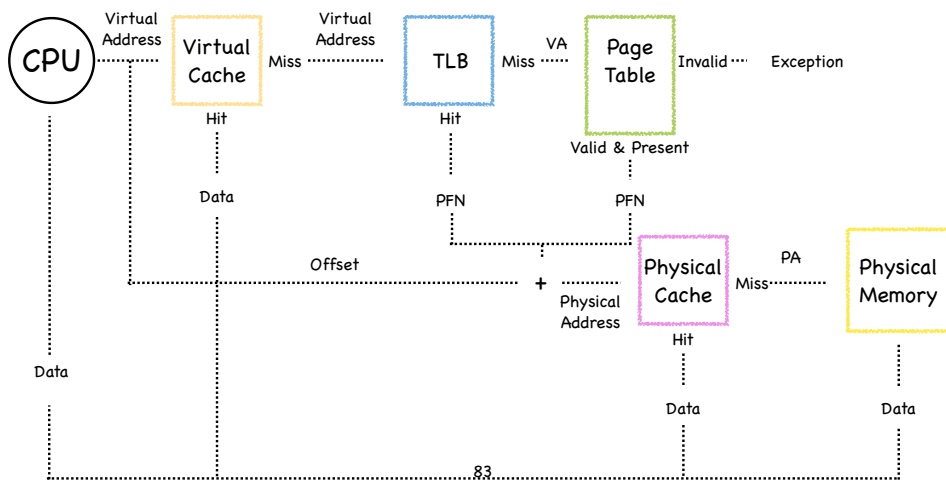
Speeding things up: The TLB



Virtually Addressed Caches



Physically Addressed Caches



TLB Consistency - I

- On context switch
 - VAs of old process should no longer be valid
 - Change PTBR – but what about the TLB?

TLB Consistency - I

- On context switch
 - VAs of old process should no longer be valid
 - Change PTBR — but what about the TLB?
 - Option 1: Flush the TLB
 - Option 2: Add pid tag to each TLB entry

	PID	VPN	PFN	Access
TLB Entry	1	0x0053	0x0012	R/W

Ignore entries with wrong PIDs

85

TLB Consistency - II

- What if OS changes permissions on a page?
 - What if permissions are reduced?
 - OS must purge affected TLB entries (e.g., on copy-on-write)
 - What if permissions are expanded?
 - either cause hardware to load new entries
 - or cause an exception, so handler can update TLB entry as necessary

86

What if we miss in the TLB?

- Suppose a 64-bit VAS, with 4KB page and a 512MB physical memory
 - Page table has 2^{52} entries
 - At 4 bytes/PTE, Page Table is 16 Petabytes!
 - per process!
 - For Page Table at each level to fit in a single page, each level should span at most 10 bits
 - 6 levels of paging!!
 - But frames are few... only $2^{29}/2^{12} = 128K$

87



A different approach

- What if mapping size were proportional to the number of frames, instead of pages?
 - If PTE = 16 bytes, Page table size = 2MB
 - And since all processes share the same physical frames, just one global page table!

Inverted page tables

88

Page Registers (a.k.a. Inverted Page Tables)

- ⊗ For each frame, a register containing
 - ▢ Residence bit
 - is the frame occupied?
 - ▢ Page number of the occupying page
 - ▢ Id of the process currently mapping the frame
 - ▢ VAS of different processes may map the same page number to different frames!
 - ▢ Protection bits
- ⊗ Searched by page number

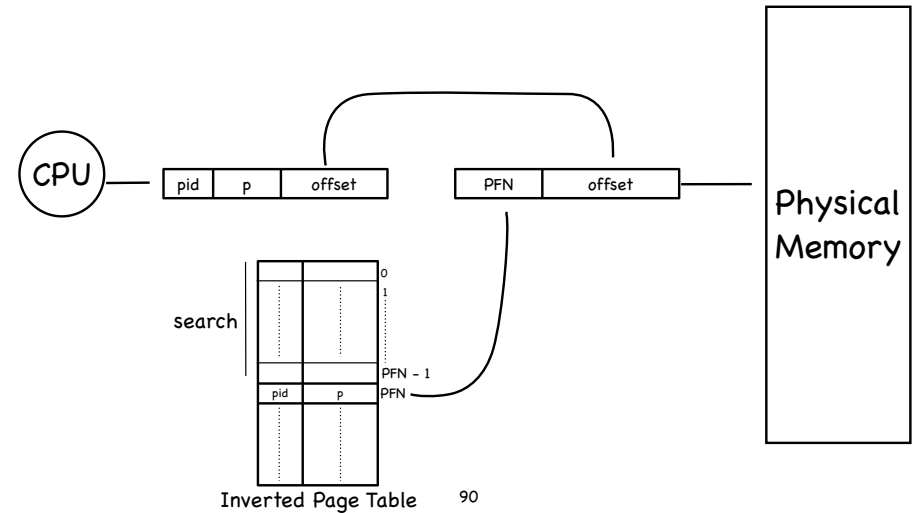
89

Where have all the pages gone?

- ⊗ Searching 128KB of registers on every memory reference is not fun
- ⊗ If the number of frames is small, the page registers can be placed in an associative memory – but...
 - ▢ Large associative memories are expensive
 - ▢ hard to access in a single cycle
 - ▢ consume lots of power

91

Basic Inverted Page Table Architecture



Hashed Inverted Page Tables

- ⊗ Hash Anchor Table maps <pid, VPN> to an entry of the Inverted Page Table
- ⊗ Collisions handled by chaining

