

Lectures 22

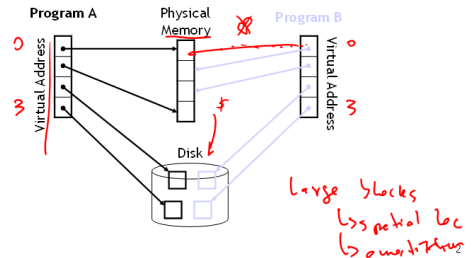
- Virtual Memory cont ☺

1

Virtual Memory

V.A. → P.A.

- Because different processes will have different mappings from virtual to physical addresses, two programs can freely use the same virtual address.
- By allocating distinct regions of physical memory to A and B, they are prevented from reading/writing each others data.

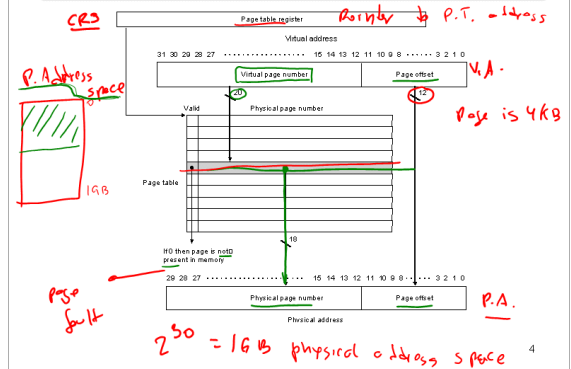


Finding the right page

- If it is fully associative, how do we find the right page without scanning all of memory?
 - Use an index, just like you would for a book. *V.A. → P. bits → P.A.*
- Our index happens to be called the page table: *in memory*
 - Each process has a separate page table
 - A "page table register" points to the current process's page table
 - The page table is indexed with the virtual page number (VPN)
 - The VPN is all of the bits that aren't part of the page offset.
 - Each entry contains a valid bit, and a physical page number (PPN)
 - The PPN is concatenated with the page offset to get the physical address
 - No tag is needed because the index is the full VPN.

3

Page Table picture



4

How big is the page table?

- From the previous slide: *V.A. space = 32 bits*
 - Virtual page number is 20 bits.
 - Physical page number is 18 bits + valid bit → round up to 32 bits. (4 bytes)

$$(2^{20} \times 4) = 4 \text{ MB}$$

p.t. entry

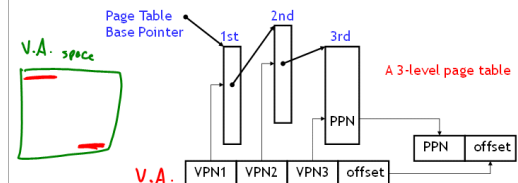
↓ per byte ⇒ Another

In case

- How about for a 64b architecture? *64 bits, P.O. 12 bits*
 $64 - 12 = 52 \text{ bits V.P.N.}$ *(2⁵² × 4) = 16 petabytes*

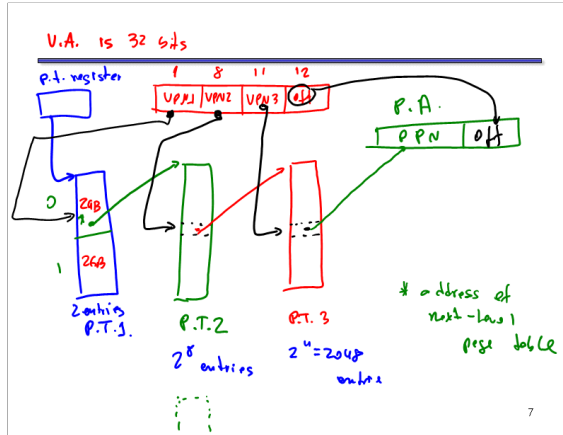
Dealing with large page tables

- Multi-level page tables
 - "Any problem in CS can be solved by adding a level of indirection" or two...



- Why does this work?
 - processes do not entire V.A. space.

6



Waitaminute!

- We've just replaced every memory access $MEM[addr]$ with:

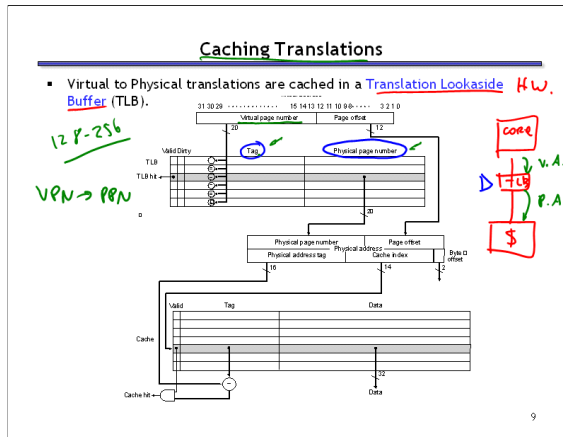
$$MEM[MEM[MEM[MEM[PTBR + VPN1 \ll 2] + VPN2 \ll 2] + VPN3 \ll 2] + offset]$$
 i.e., 4 memory accesses
- And we haven't talked about the bad case yet (i.e., page faults)...

"Any problem in CS can be solved by adding a level of indirection" — except too many levels of indirection...

- How do we deal with too many levels of indirection?

$\$$ Cache VA \rightarrow PA

8



What about a TLB miss?

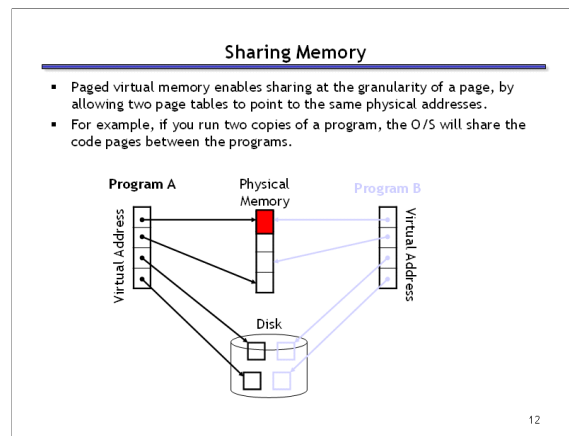
- If we miss in the TLB, we need to "walk the page table"
 - In MIPS, an exception is raised and software fills the TLB
 - In x86, a "hardware page table walker" fills the TLB
- What if the page is not in memory?
 - This situation is called a **page fault**.
 - The operating system will have to request the page from disk.
 - It will need to select a page to replace.
 - The O/S tries to approximate LRU (see CS451)
 - The replaced page will need to be written back if dirty.

10

Memory Protection

- In order to prevent one process from reading/writing another process's memory, we must ensure that a process cannot change its virtual-to-physical translations.
- Typically, this is done by:
 - Having two processor modes: user & kernel.
 - Only the O/S runs in kernel mode
 - Only allowing kernel mode to write to the virtual memory state, e.g.,
 - The page table
 - The page table base pointer
 - The TLB

11



Summary

- Virtual memory is **great**:
 - It means that we don't have to manage our own memory.
 - It allows different programs to use the same memory.
 - It provides protect between different processes.
 - It allows controlled sharing between processes (albeit somewhat inflexibly).
- The key technique is **indirection**:
 - Yet another classic CS trick you've seen in this class.
 - Many problems can be solved with indirection.
- Caching made a few appearances, too:
 - Virtual memory enables using physical memory as a cache for disk.
 - We used caching (in the form of the Translation Lookaside Buffer) to make Virtual Memory's indirection fast.