COS 140: Foundations of Computer Science

Virtual Memory: Translation Lookaside Buffer

Fall 2018

Homework, announcements

- ☐ Reading: Chapter "Virtual Memory: Translation Lookaside Buffer" (Ch. 21)
- ☐ Homework: Exercises at end of chapter
- ☐ Due Wednesday, 11/14 (later than usual)
- ☐ **NOTE:** Prelim II on 11/14
- ☐ Reminder: Advising!

Virtual Memory

Virtual memory

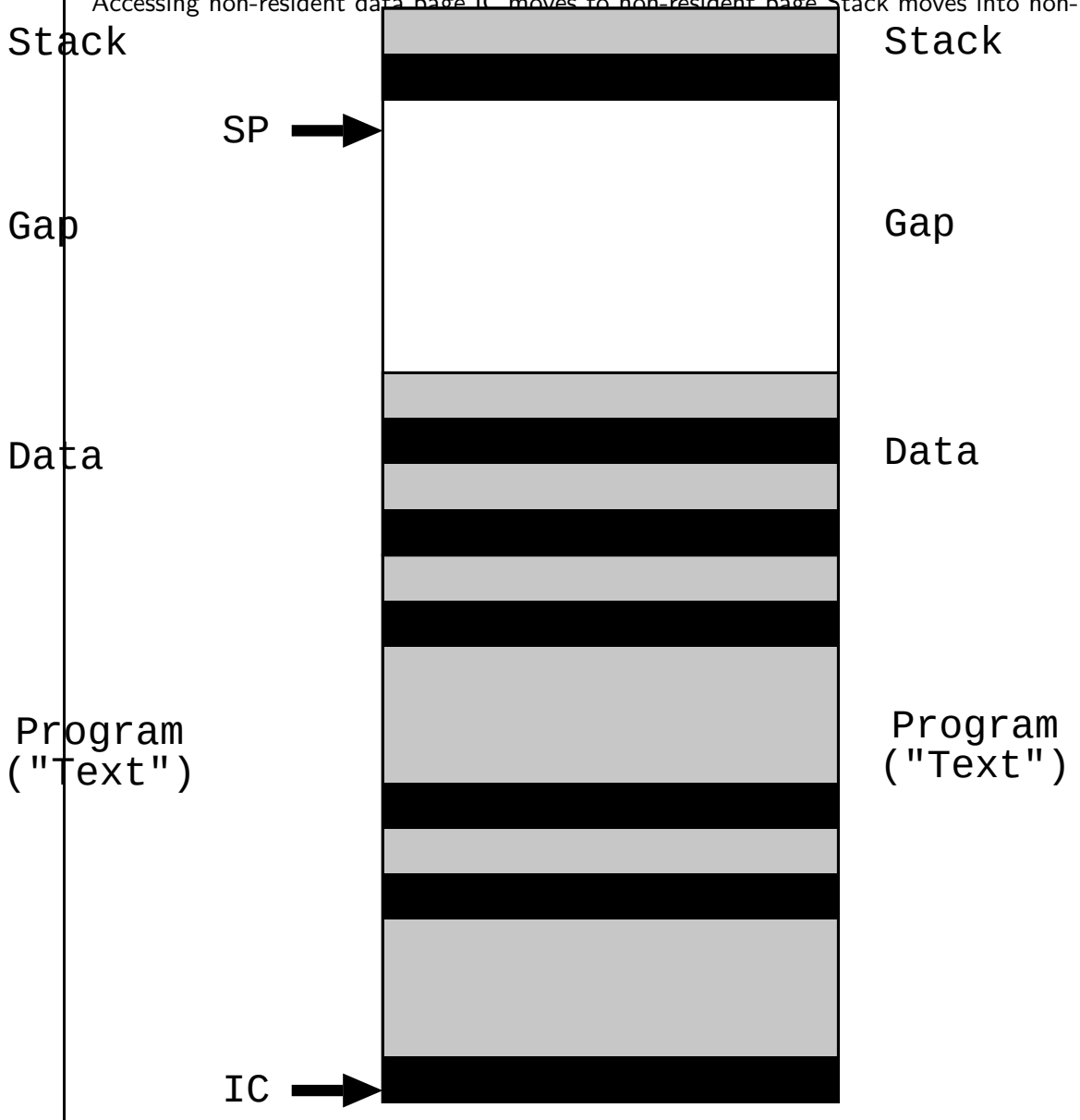
- ☐ Goals:
  - Allow more processes to run simultaneously (increase *degree of multiprogramming*)
  - Allow very large processes to run
- ☐ Approach:
  - Keep most of each process on disk (in paging/swap area)
  - Only keep that part of each process in memory that is actually in use

Pages and page frames

- Divide process' address space into *pages* of some fixed size – usually 2Kb–4Kb
- Divide (physical) memory into *page frames* of same size
- Put needed pages of process into page frames: Need not be contiguous
- Move pages back and forth between disk and memory as needed: *demand paging*

When will a process need a new page?

Accessing non-resident data page IC moves to non-resident page Stack moves into non-resident-page



Page faults

- If page is *resident* – use it
- If page is not resident ⇒ *page fault*
- A page fault is a type of interrupt
- Operating system wakes up, tries to bring in the needed page
  - What if there are free page frames?
  - What if there are no free page frames?

Page Tables

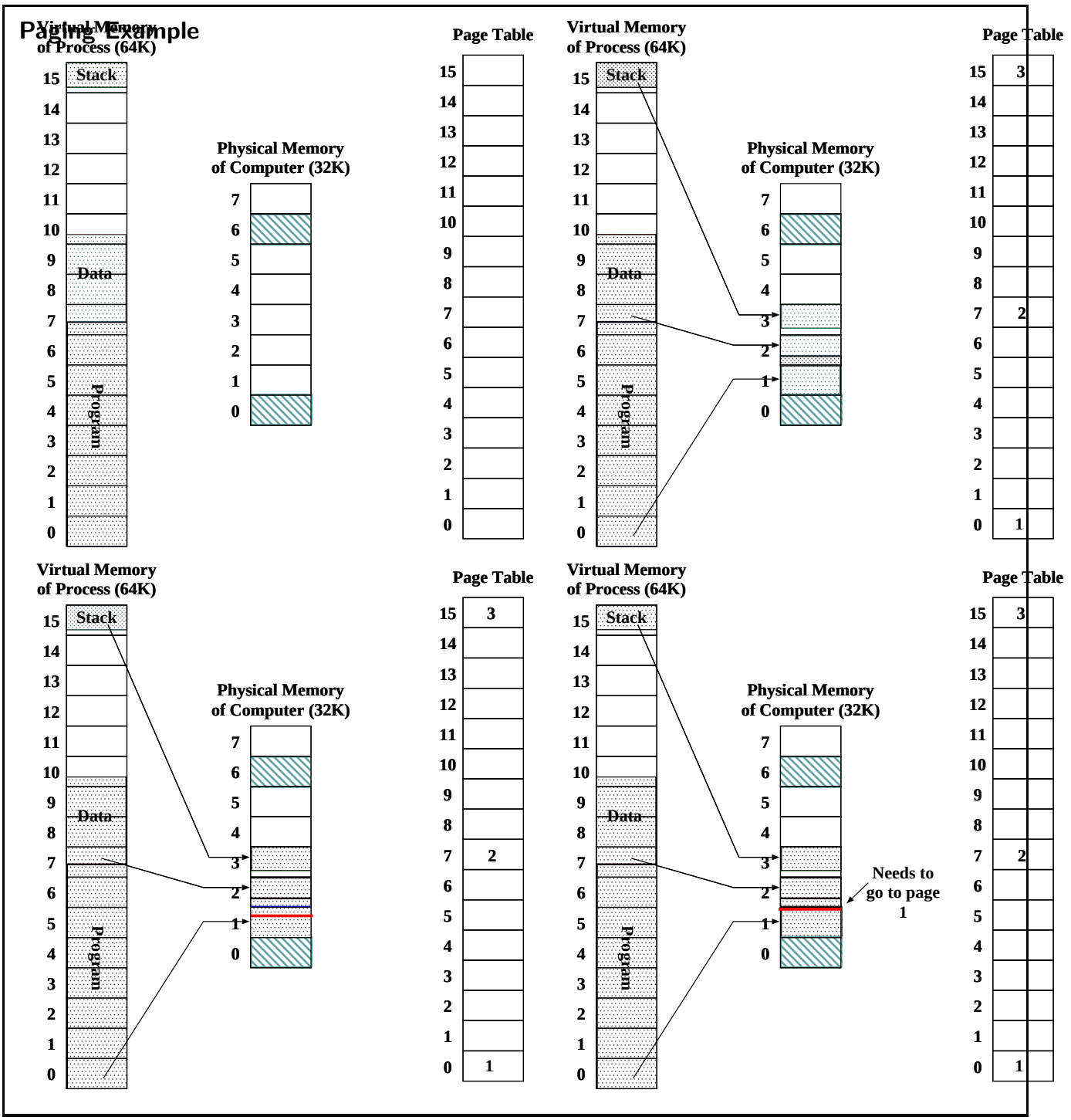
Page⇔frame mapping

- At any given time, need to keep track of where a process' resident pages are
- Also need to keep track of which pages are not resident
- Use a *page table* for this
- *Page table entries* (*PTEs*) map from pages to where those pages live in memory

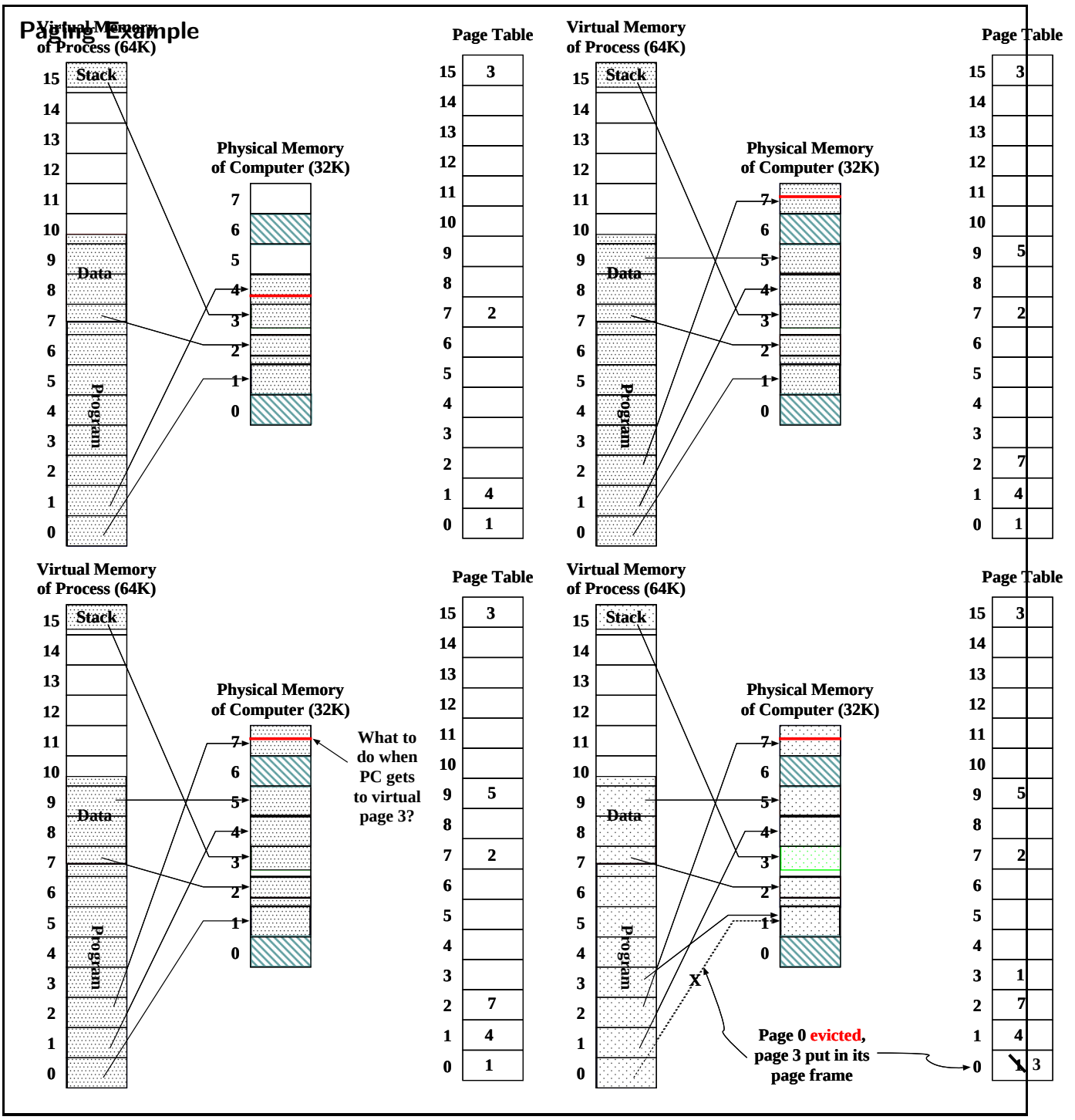
Page tables

- One page table per process
- One page table entry per page in virtual memory (address space)
- Each entry contains:
  - Present/absent bit
  - Page frame number
  - Dirty bit (M bit)
  - Reference bit (R bit)
  - Maybe other things

Paging Example

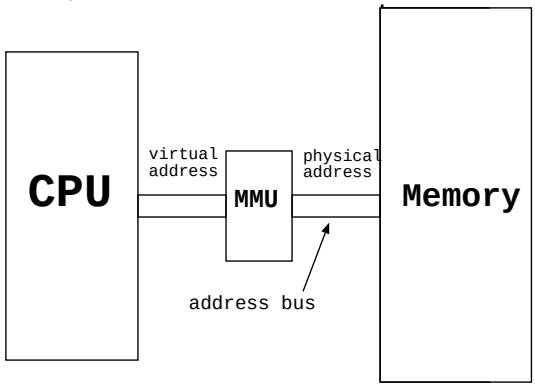


Paging Example



Memory management unit

- How does a virtual address ⇒ a physical address?
- *Memory management unit* (MMU): piece of hardware that sits between the CPU and memory:

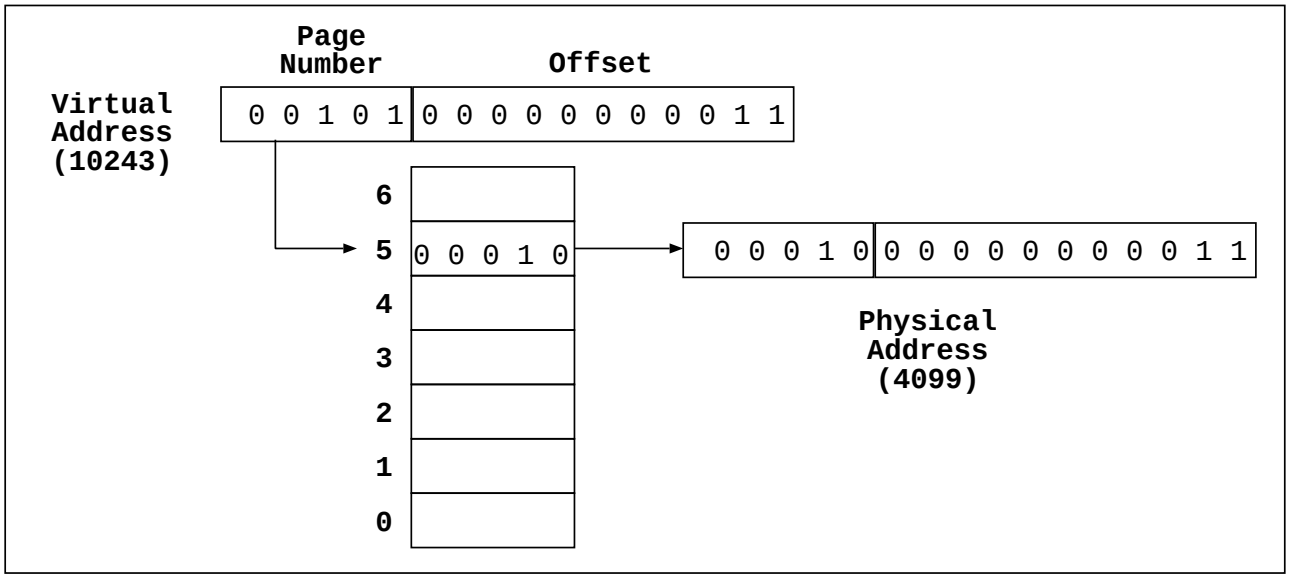


- These days: MMU is on CPU chip

Address translation

- How does the virtual address get translated to a physical address?
- Suppose we have 2KB pages, 16-bit machine:
  - 2KB $= 2^{11}$bytes – need 11 bits to address each byte on a page
  - Divide virtual address into 5-bit page number, 11-bit offset

Address translation



Problems with page tables

- Speed of access:
  - If table lives in memory, then for every memory access, have to look in page table to find address—a memory access itself—then do memory access
  - *Effective memory access time* is doubled
- Possible solution: Use very fast memory in MMU

Problems with page tables

- But page tables can be very large:
  - One entry per page in virtual address
  - With 2KB (2^{11} bytes) pages, 1 word/entry:
    - 16-bit machine: $2^{16}/2^{11} = 2^5$ entries $= 2^5 \times 2 = 64$ bytes
    - 32-bit machines: $2^{32}/2^{11} = 2^{21}$ entries $= 2^{21} \times 2^2 = 8$Mb (!)
    - 64-bit machines: $2^{64}/2^{11} \times 2^3 = 2^{56}$ bytes (!!)$= 67,108,864$ GB
- Realistically, can't afford that much fast memory!
  At $6/GB, $402,653,184...
  ...*per process*!

Solution: Caching

- A *cache* is fast memory that holds *part* of what's in slower memory
- Idea: prevent accessing slower memory by keeping in cache what will be needed soon
- Cache according to (e.g.):
  - recency
  - frequency
  - predicted next use

TLB

Translation Look-Aside Buffer

- TLB is a cache of page table entries
- TLB lives in MMU and is composed of very fast registers
- Special kind of registers: *associative* – MMU can look up page table entry corresponding to page number in one step

How it works

- When process starts – TLB empty, address 0 CPU $\Rightarrow$ MMU:
  - PTE for page 0 not in TLB; read in from page table in memory
  - Page 0 not mapped $\Rightarrow$ page fault
  - Page in page 0 into frame i, update PTE in TLB
- Next time some address on page 0 referenced:
  - Use PTE in TLB to find frame i
  - No memory access for PTE
- When TLB full:
  - Have to eject some PTE from cache
  - Write it to page table first

How well does it work?

- Assume memory access time $t_m = 100$ns; no TLB: 200ns/access
- With TLB:

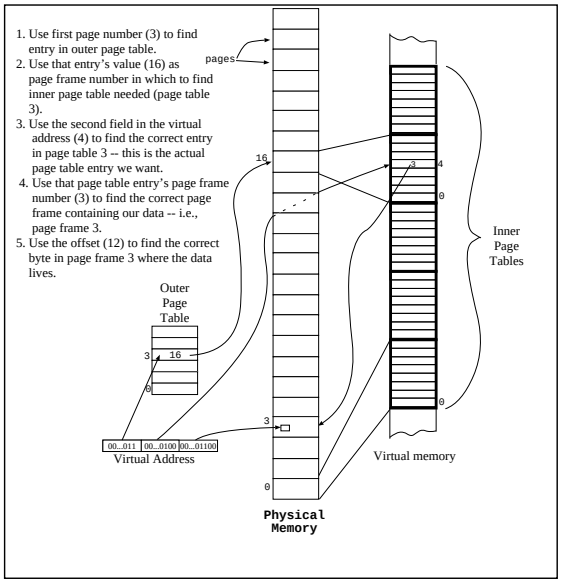
$$T_{eff} = p(t_m + t_r) + (1 - p)(2t_m + t_r)$$

  where T_{eff} is *effective access time*, p is probability of PTE being in the cache (*hit ratio*), and t_r is time to look something up in TLB
- With $t_r = 20ns$, hit ratio 0.5: $T_{eff} = 170$ns
- Hit ratio 0.8: $T_{eff} = 140$ns
- Hit ratio 0.98: $T_{eff} = 122$ns

Multiple-level paging

- Problem: With large address sizes, page tables too big to keep in memory
- Would like to page out the page tables themselves!
- Can't, with monolithic tables
- Solution: multi-level page tables
- "Outer tables" act as page tables for "inner" pages – only outer needs to be resident in memory
- Price: ≥ 2 memory accesses/access in worst case

Multi-level page tables



Does TLB still help?

- Suppose we have 4-level paging, 100 ns memory access time, 20 ns TLB time
- Worst case: 500ns/access (4 page table accesses + desired access)
- With TLB, hit rate 0.98: $T_{\mathit{eff}} = 188$ ns
- More realistic numbers: $t_r = 1$ ns, $t_m = 60$ ns; 4-level paging: $T_{\mathit{eff}} = 68.8$ ns
- In case you're interested, the effective memory access time for n-level paging is:

$$T_{\mathit{eff}_n} = nt_r + t_m + n(1-p)t_m$$

Caching

Other kinds of caching

- Caching shows up many places in OS, elsewhere
- Processor caching physical memory
- Disk block caching for files
- Network file systems
- Virtual memory itself