Scalable systems?!?

How does this apply to ANYTHING we are doing?!

Understanding scalability means…

- Process
- Threads
- Containers… Isolation!?!
- Load balancing
- Stateful/sticky functionality (THIS IS BIG!)
- Microservices
- Consistent Hashing
- Cache design
- CQRS design pattern
- Sharding
- Workflows
- Token buckets

FOR US?!

- Reinforces system level policy and mechanism!

- Questions…
  - What are microservices and workflows?
  - What is the whole point of that architectural approach?
  - How does this help mitigate Brooks' Law?!

- WHAT ARE YOU DOING FOR YOUR PROJECT?!
  - Also, XN course!? https://forms.gle/LUH3iPHDUhw5WQNQ9

TODAY!

- Memory management and Virtual memory
  - Paging
  - Caching
  - Understanding!
- QEMU
- File systems are coming…
  - MicroSD!

Goals of memory management

- Allocate memory resources among competing processes, maximizing memory utilization and system throughput
- Provide isolation between processes
  - We have come to view "addressability" and "protection" as inextricably linked, even though they're really orthogonal
- Provide a convenient abstraction for programming (and for compilers, etc.)

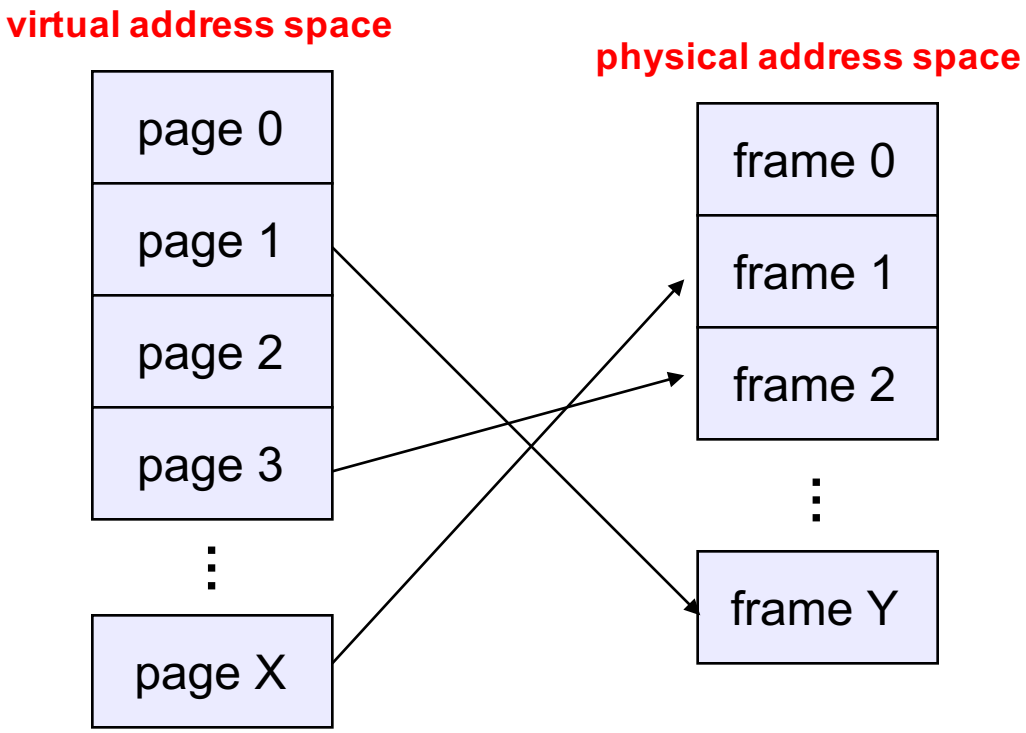
5

VM requires hardware and OS support

- MMU's, TLB's, page tables, page fault handling, …
- Typically accompanied by swapping, and at least limited segmentation

Modern technique: Paging

- Solve the external fragmentation problem by using fixed sized units in both physical and virtual memory
- Solve the internal fragmentation problem by making the units small



Life is easy …

- For the programmer …
  - Processes view memory as a contiguous address space from bytes 0 through N – a virtual address space
  - N is independent of the actual hardware
  - In reality, virtual pages are scattered across physical memory frames – not contiguous as earlier
    - Virtual-to-physical mapping
    - This mapping is invisible to the program
- For the memory manager …
  - Efficient use of memory, because very little internal fragmentation
  - No external fragmentation at all
    - No need to copy big chunks of memory around to coalesce free space

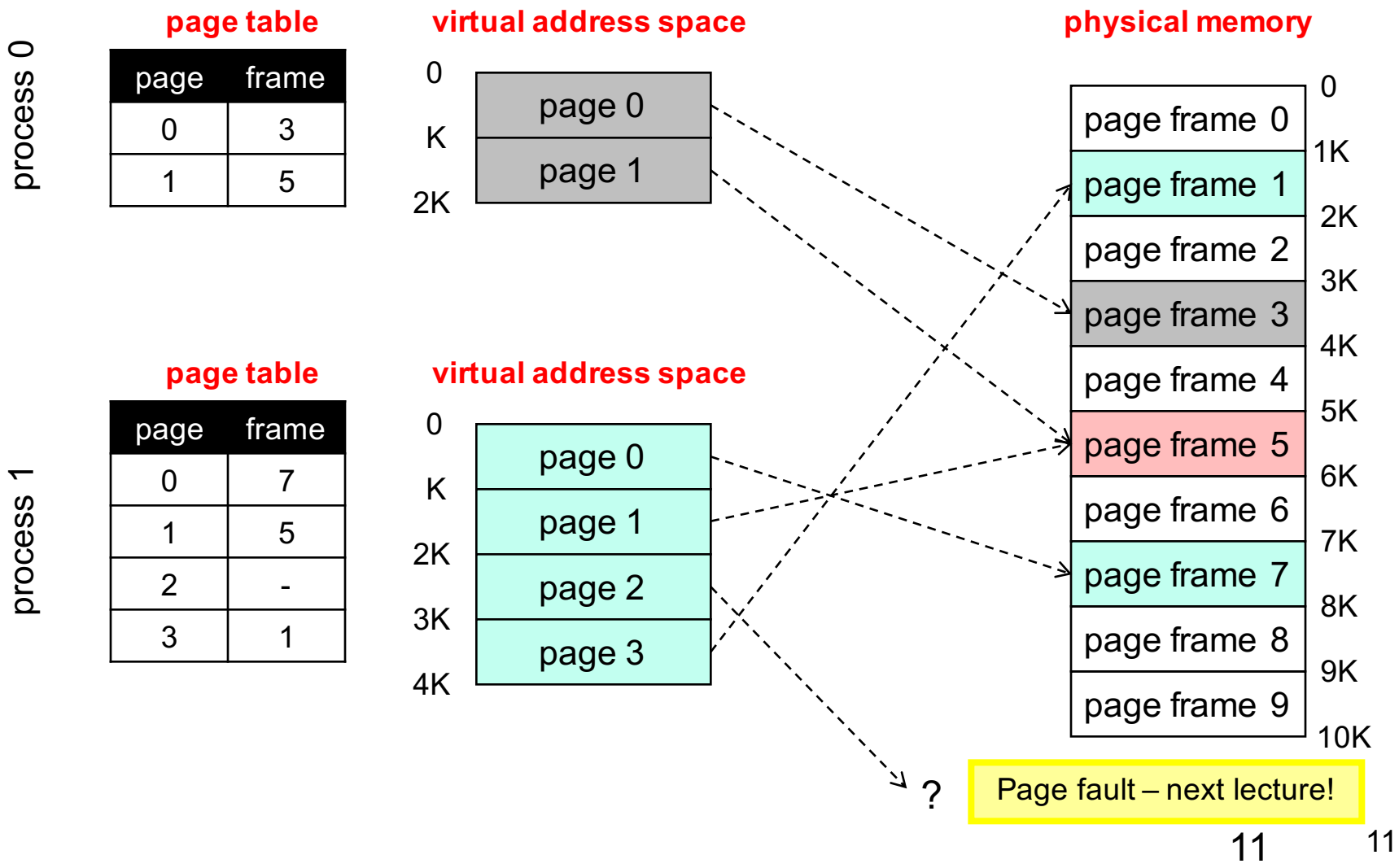
- For the protection system
  - One process cannot "name" another process's memory – there is complete isolation
    - A virtual address 0xFACECAFE maps to different physical addresses for different processes

Note: Assume for now that all pages of the address space are resident in memory – no "page faults"

Address translation

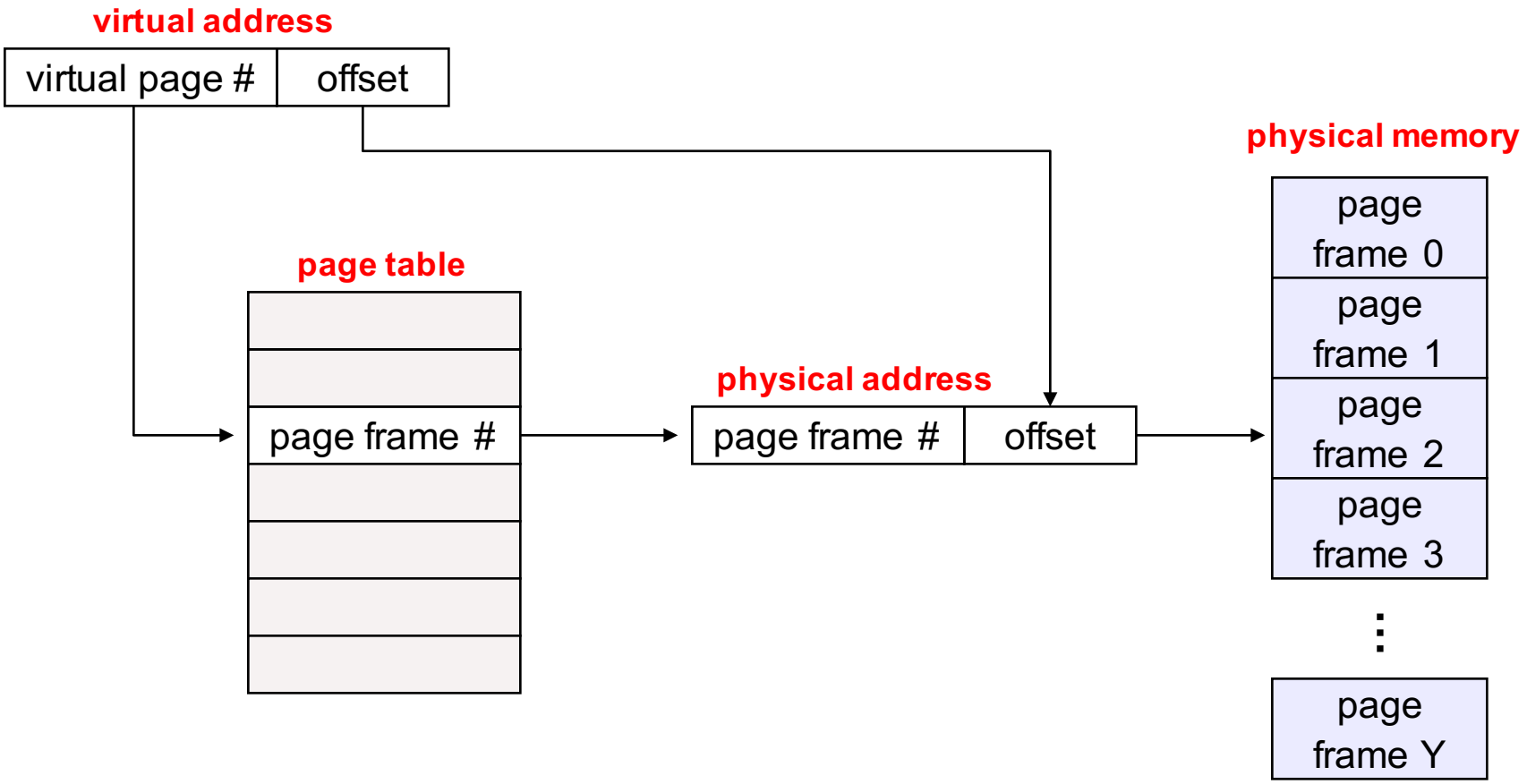
- Translating virtual addresses
  - a virtual address has two parts: virtual page number & offset
  - virtual page number (VPN) is index into a page table
  - page table entry contains page frame number (PFN)
  - physical address is PFN::offset
- Page tables
  - managed by the OS
  - one page table entry (PTE) per page in virtual address space
    - i.e., one PTE per VPN
  - map virtual page number (VPN) to page frame number (PFN)
    - VPN is simply an index into the page table

Paging (K-byte pages)



11

Mechanics of address translation



Example of address translation

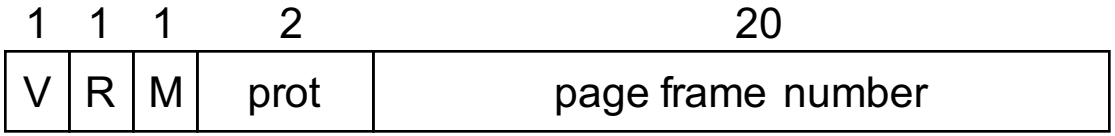
- Assume 32 bit addresses
  - assume page size is 4KB (4096 bytes, or 2^{12} bytes)
  - VPN is 20 bits long (2^{20} VPNs), offset is 12 bits long

- Let's translate virtual address 0x13325328
  - VPN is 0x13325, and offset is 0x328
  - assume page table entry 0x13325 contains value 0x03004
    - page frame number is 0x03004
    - VPN 0x13325 maps to PFN 0x03004
  - physical address = PFN::offset = 0x03004328

Page Table Entries – an opportunity!

- As long as there's a PTE lookup per memory reference, we might as well add some functionality
  - We can add protection
    - A virtual page can be read-only, and result in a fault if a store to it is attempted
    - Some pages may not map to anything – a fault will occur if a reference is attempted
  - We can add some "accounting information"
    - Can't do anything fancy, since address translation must be fast
    - Can keep track of whether or not a virtual page is being used, though
      - This will help the paging algorithm, once we get to paging

Page Table Entries (PTE's)



- PTE's control mapping
  - the valid bit says whether or not the PTE can be used
    - says whether or not a virtual address is valid
    - it is checked each time a virtual address is used
  - the referenced bit says whether the page has been accessed
    - it is set when a page has been read or written to
  - the modified bit says whether or not the page is dirty
    - it is set when a write to the page has occurred
  - the protection bits control which operations are allowed
    - read, write, execute
  - the page frame number determines the physical page
    - physical page start address = PFN

Paging advantages

- Easy to allocate physical memory
  - physical memory is allocated from free list of frames
    - to allocate a frame, just remove it from the free list
  - external fragmentation is not a problem
    - managing variable-sized allocations is a huge pain in the neck
      - "buddy system"
- Leads naturally to virtual memory
  - entire program need not be memory resident
  - take page faults using "valid" bit
  - all "chunks" are the same size (page size)
  - but paging was originally introduced to deal with external fragmentation, not to allow programs to be partially resident

Paging disadvantages

- Can still have internal fragmentation
  - Process may not use memory in exact multiples of pages
  - But minor because of small page size relative to address space size
- Memory reference overhead
  - 2 references per address lookup (page table, then memory)
  - Solution: use a hardware cache to absorb page table lookups
    - translation lookaside buffer (TLB) – next class
- Memory required to hold page tables can be large
  - need one PTE per page in virtual address space
  - 32 bit AS with 4KB pages = 2^{20} PTEs = 1,048,576 PTEs
  - 4 bytes/PTE = 4MB per page table
    - OS's have separate page tables per process
    - 25 processes = 100MB of page tables
  - Solution: page the page tables (!!!)
    - (ow, my brain hurts…more later)

"Issues"

- Memory reference overhead of address translation
  - 2 references per address lookup (page table, then memory)
  - solution: use a hardware cache to absorb page table lookups
    - translation lookaside buffer (TLB)
- Memory required to hold page tables can be huge
  - need one PTE per page in the virtual address space
  - 32 bit AS with 4KB pages = 2^{20} PTEs = 1,048,576 PTEs
  - 4 bytes/PTE = 4MB per page table
    - OS's typically have separate page tables per process
    - 25 processes = 100MB of page tables
  - 48 bit AS, same assumptions, 64GB per page table!

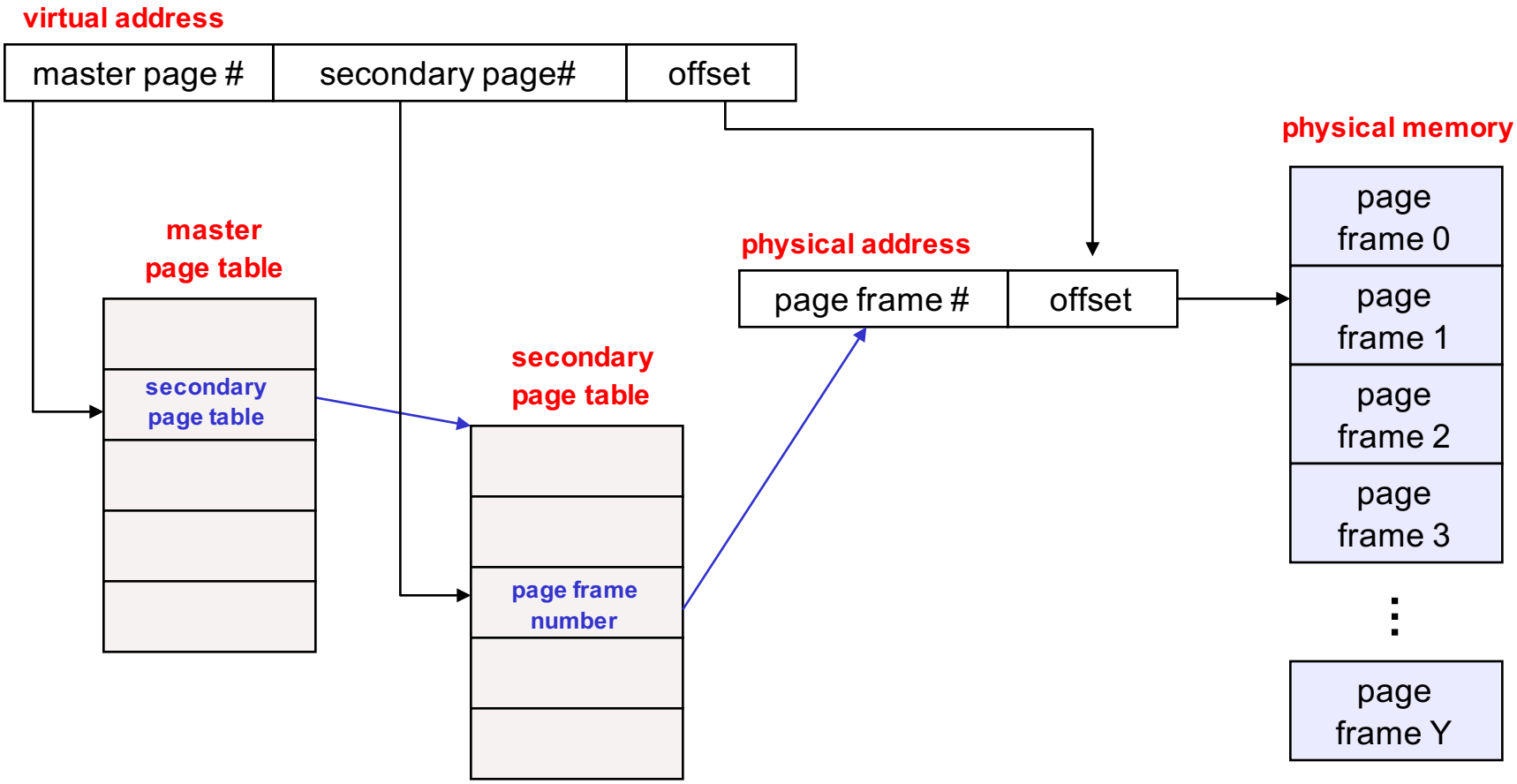
Solution: Multi-level page tables

- How can we reduce the physical memory requirements of page tables?
  - observation: only need to map the portion of the address space that is actually being used (often a tiny fraction of the total address space)
    - a process may not use its full 32/48/64-bit address space
    - a process may have unused "holes" in its address space
    - a process may not reference some parts of its address space for extended periods
  - all problems in CS can be solved with a level of indirection!
    - two-level (three-level, four-level) page tables

Two-level page tables

- With two-level PT's, virtual addresses have 3 parts:
  - master page number, secondary page number, offset
  - master PT maps master PN to secondary PT
  - secondary PT maps secondary PN to page frame number
  - offset and PFN yield physical address

Two level page tables



Making it all efficient

- Original page table scheme doubled the cost of memory lookups
  - one lookup into page table, a second to fetch the data
- Two-level page tables triple the cost!!
  - two lookups into page table, a third to fetch the data
- How can we make this more efficient?
  - goal: make fetching from a virtual address about as efficient as fetching from a physical address
  - solution: use a hardware cache inside the CPU
    - cache the virtual-to-physical translations in the hardware
    - called a translation lookaside buffer (TLB)
    - TLB is managed by the memory management unit (MMU)

TLBs

- Translation lookaside buffer
  - translates virtual page #s into PTEs (page frame numbers) (not physical addrs)
  - can be done in single machine cycle
- TLB is implemented in hardware
  - is a fully associative cache (all entries searched in parallel)
  - cache tags are virtual page numbers
  - cache values are PTEs (page frame numbers)
  - with PTE + offset, MMU can directly calculate the PA
- TLBs exploit locality
  - processes only use a handful of pages at a time
    - 16-48 entries in TLB is typical (64-192KB)
    - can hold the "hot set" or "working set" of a process
  - hit rates in the TLB are therefore really important

Managing TLBs

- OS must ensure TLB and page tables are consistent
  - when OS changes protection bits in a PTE, it needs to invalidate the PTE if it is in the TLB
- What happens on a process context switch?
  - remember, each process typically has its own page tables
  - need to invalidate all the entries in TLB! (flush TLB)
    - this is a big part of why process context switches are costly
  - can you think of a hardware fix to this?
- When the TLB misses, and a new PTE is loaded, a cached PTE must be evicted
  - choosing a victim PTE is called the "TLB replacement policy"
  - implemented in hardware, usually simple (e.g., LRU)

  - On context switch
    - TLB must be purged/flushed (using a special hardware instruction) unless entries are tagged with a process ID
      - otherwise, the new process will use the old process's TLB entries and reference its page frames!

- Cool tricks
  - Read-only code
  - Inter-process memory protection
  - Shared libraries
  - Inter-process communication
  - Shared memory
  - Copy-on-write
  - Memory-mapped files
  - Soft faults (e.g., debugger watchpoints)