

CSE 451: Operating Systems

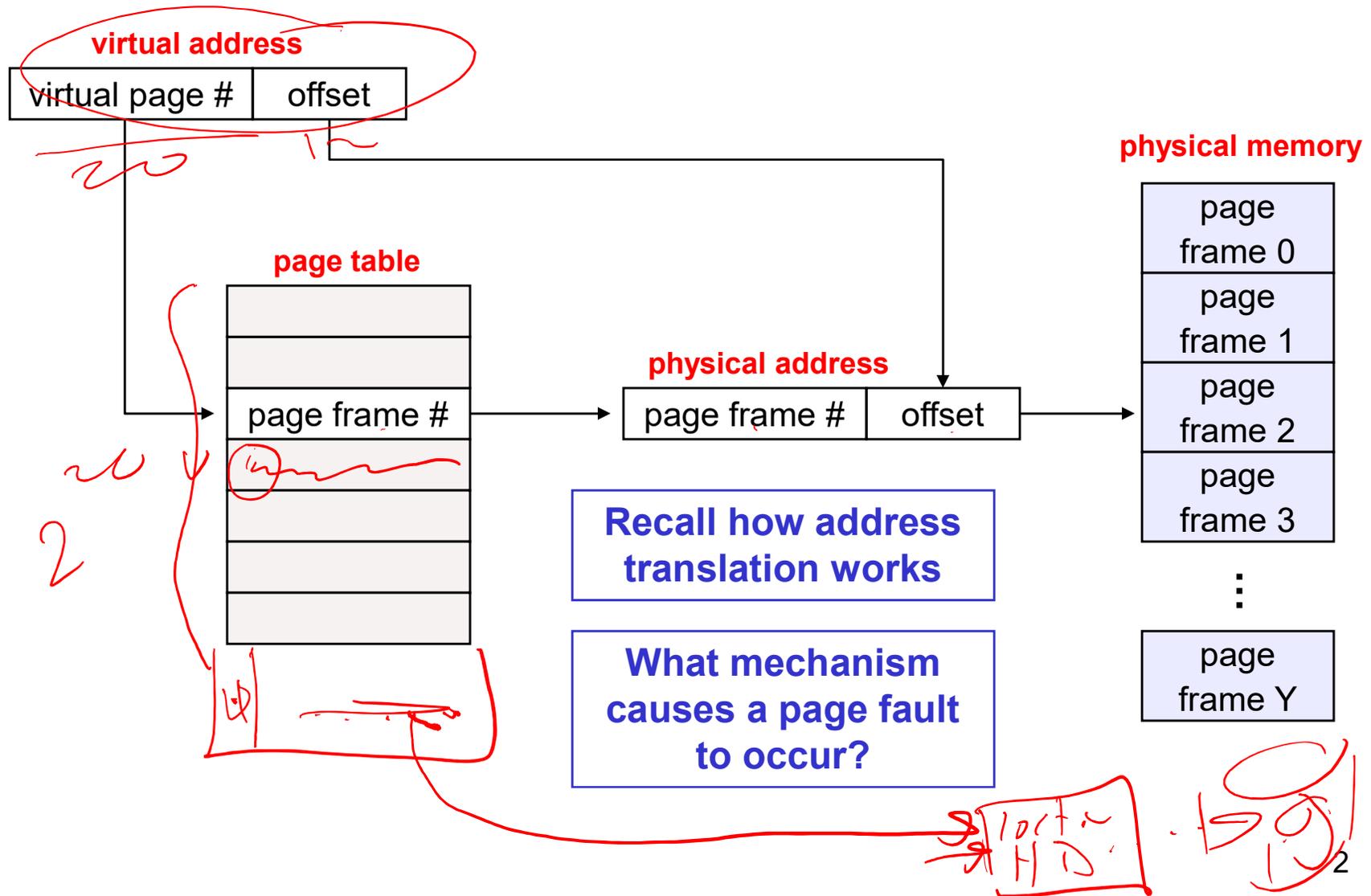
Winter 2022

Module 13

Page Table Management, TLBs, and Other Pragmatics

Gary Kimura

Address translation and page faults (refresher!)



How does OS handle a page fault?

- Interrupt causes system to be entered
- System saves state of running process, then vectors to page fault handler routine

① – find or create (through eviction) a page frame into which to load the needed page (1)

- if I/O is required, run some other process while it's going on

② – find the needed page on disk and bring it into the page frame (2)

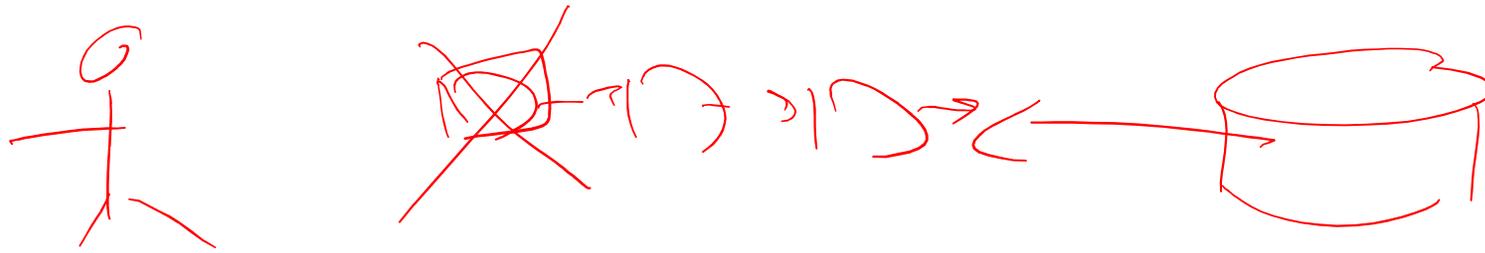
- run some other process while the I/O is going on

– fix up the page table entry PTE

- mark it as “valid,” set “referenced” and “modified” bits to false, set protection bits appropriately, point to correct page frame

– put the process on the ready queue

- (2) Find the needed page on disk and bring it into the page frame
 - processor makes process ID and faulting virtual address available to page fault handler
 - process ID gets you to the base of the page table
 - VPN portion of VA gets you to the PTE
 - data structure analogous to page table (an array with an entry for each page in the address space) contains disk address of page
 - at this point, it's just a simple matter of I/O
 - must be positive that the target page frame remains available!
 - or what?



- **(1)** Find or create (through eviction) a page frame into which to load the needed page
 - run page replacement algorithm
 - free page frame
 - assigned but unmodified (“clean”) page frame
 - assigned and modified (“dirty”) page frame
 - assigned but “clean”
 - find PTE (may be a different process!)
 - mark as invalid (disk address must be available for subsequent reload)
 - assigned and “dirty”
 - find PTE (may be a different process!)
 - mark as invalid
 - write it out

- OS may speculatively maintain lists of clean and dirty frames selected for replacement
 - May also speculatively clean the dirty pages (by writing them to disk)

Bottleneck

"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!

size

Solution 1 to (2): Page the page tables

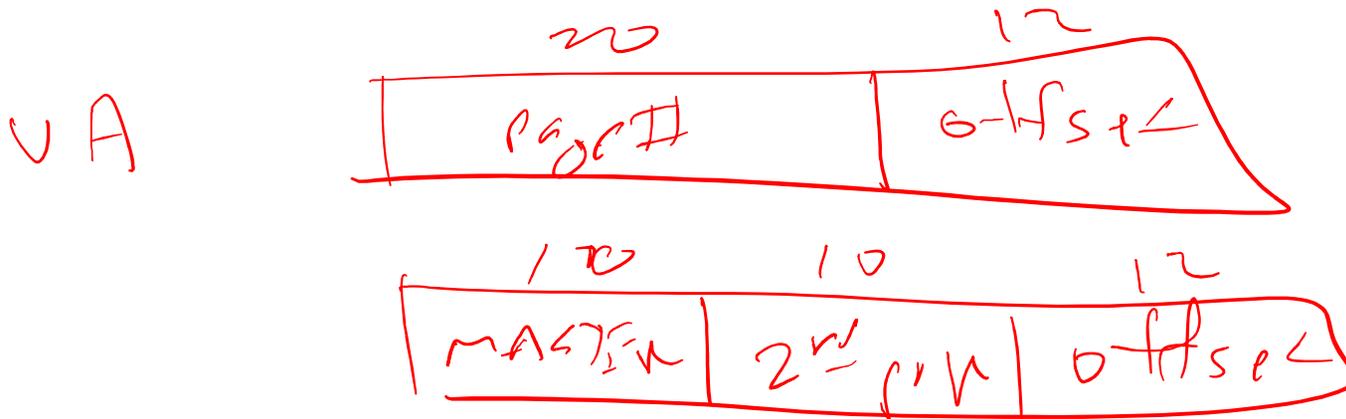
- Simplest notion:
 - Put user page tables in a pageable segment of the system's address space
 - The OS page table maps the portion of the VAS in which the user process page tables live
 - Pin the system's page table(s) in physical memory
 - So you can never fault trying to access them
 - When you need a user page table entry
 - It's in the OS virtual address space, so need the OS page table to translate to a physical address
 - You cannot fault on accessing the OS page table (because it's pinned)
 - The OS page table might indicate that the user page table isn't in physical memory
 - That's just a regular page fault
- This isn't exactly what's done any longer
 - Although it is *exactly* what VAX/VMS did!
 - And it's a useful model, and a component, for what's actually done

Solution 2 to (2): Multi-level page tables

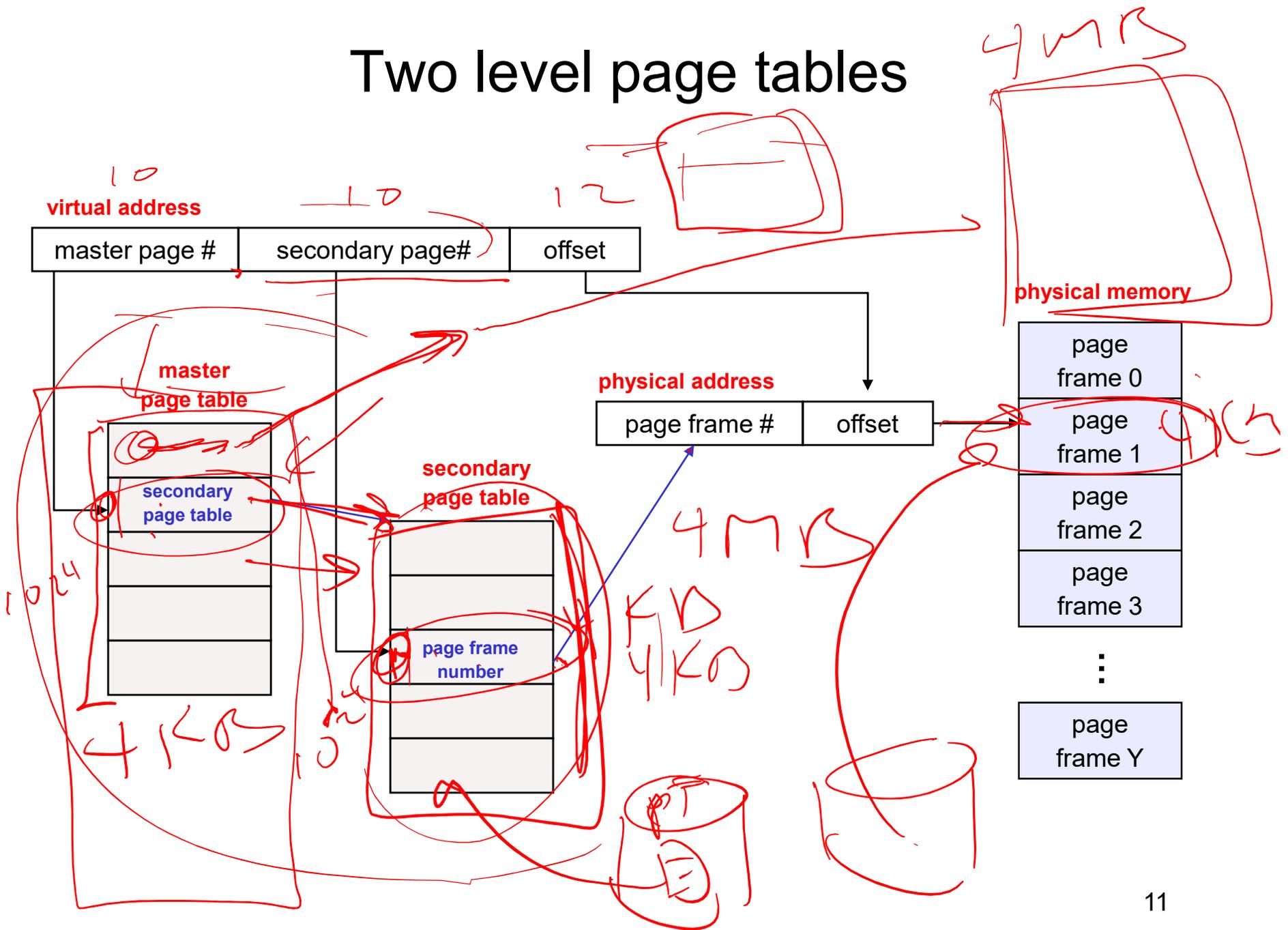
- How can we reduce the physical memory requirements of page tables?
 - observation: only need to map the portion of the address space that is actually being used (often a tiny fraction of the total address space)
 - a process may not use its full 32/48/64-bit address space
 - a process may have unused “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



- Example:
 - 32-bit address space, 4KB pages, 4 bytes/PTE
 - how many bits in offset?
 - need 12 bits for 4KB ($2^{12}=4K$), so offset is 12 bits
 - want master PT to fit in one page
 - 4KB/4 bytes = 1024 PTEs
 - thus master page # is 10 bits ($2^{10}=1K$)
 - and there are 1024 secondary page tables
 - and 10 bits are left (32-12-10) for indexing each secondary page table
 - hence, each secondary page table has 1024 PTEs and fits in one page

Generalizing

- Early architectures used 1-level page tables
- VAX, P-II used 2-level page tables ←
- SPARC used 3-level page tables
- 68030 used 4-level page tables
- Key thing is that the outer level must be wired down (pinned in physical memory) in order to break the recursion – *no smoke and mirrors*

Alternatives

- Hashed page table (great for sparse address spaces)
 - VPN is used as a hash
 - collisions are resolved because the elements in the linked list at the hash index include the VPN as well as the PFN
- Inverted page table (really reduces space!)
 - one entry per page frame
 - includes process id, VPN
 - hard to search! (but IBM PC/RT actually did this!)

- super pages

Making it all efficient

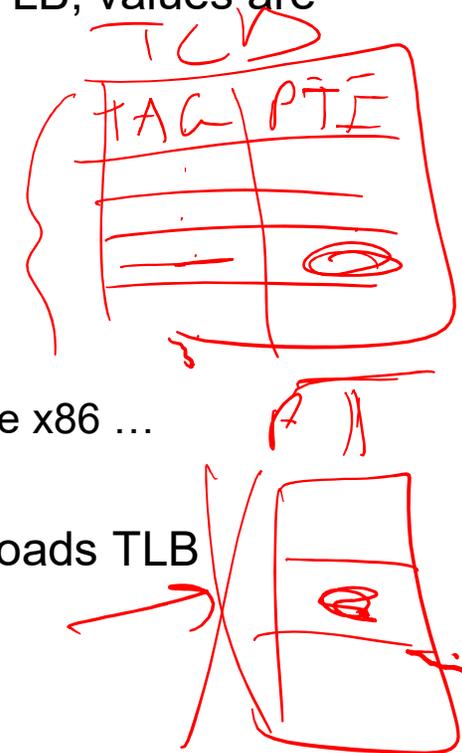
- Original page table scheme doubled the cost of memory lookups
 - one lookup into page table, a second to fetch the data
- Two-level page tables triple the cost!!
 - two lookups into page table, a third to fetch the data
- How can we make this more efficient?
 - goal: make fetching from a virtual address 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 (not physical address)
- 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 (including protection, valid bit!)
 - with PFN(from PTE) + offset, MMU can directly calculate the physical address
- TLBs exploit locality
 - processes only use a handful of pages at a time
 - can hold the “hot set” or “working set” of a process
 - hit rates in the TLB are therefore really important for performance

Managing TLBs

- Address translations are mostly handled by the TLB
 - >99% of translations, but there are **TLB misses** occasionally
 - in case of a miss, translation is placed into the TLB, values are evicted. Selection algorithm is proprietary
- Hardware (memory management unit (MMU))
 - knows where page tables are in memory
 - OS maintains them, HW access them directly
 - tables have to be in HW-defined format
 - this is how x86 works
 - And that was part of the difficulty in virtualizing the x86 ...
- Software loaded TLB (OS) *MIPS*
 - TLB miss faults to OS, OS ~~finds~~ right PTE and loads TLB
 - must be fast (but, 20-1000 cycles typically)
 - CPU ISA has instructions for TLB manipulation
 - OS gets to pick the page table format



Managing TLBs (2)

- 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”

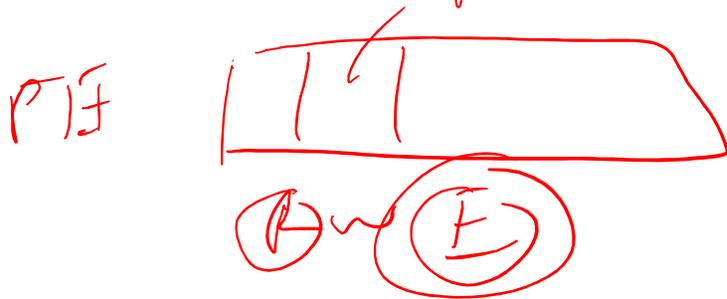
Functionality enhanced by page tables

- Code (instructions) is read-only
 - A bad pointer can't change the program code
- Dereferencing a null pointer is an error caught by hardware
 - Don't use the first page of the virtual address space – mark it as invalid – so references to address 0 cause an interrupt
- Inter-process memory protection
 - My address XYZ is different than your address XYZ
- Shared libraries
 - All running C programs use libc
 - Have only one (partial) copy in physical memory, not one per process
 - All page table entries mapping libc point to the same set of physical frames
 - DLL's in Windows

Illustrations

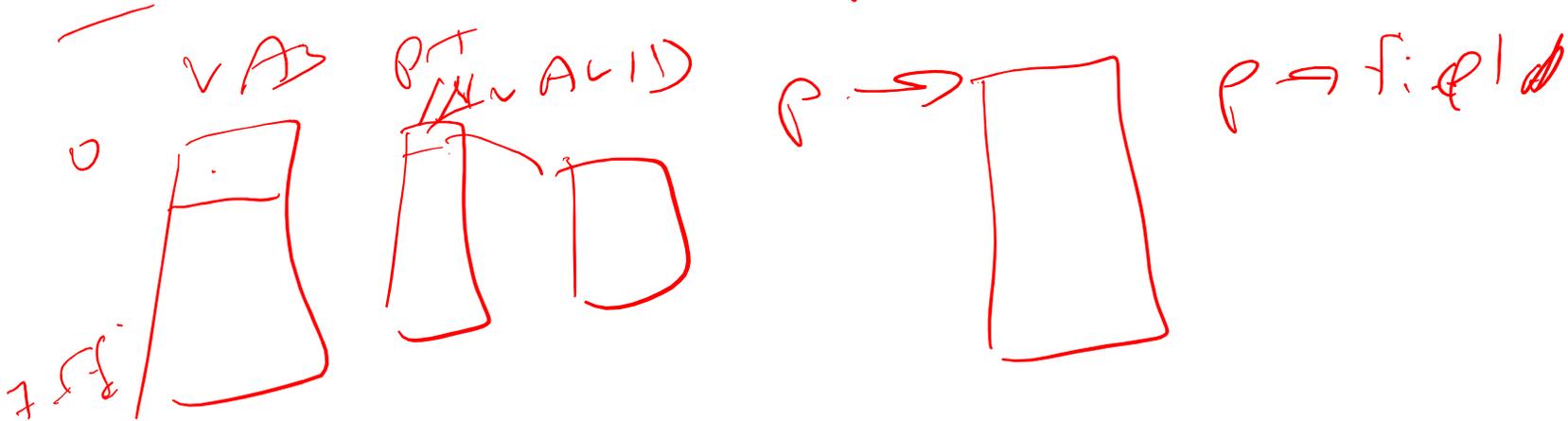
- Read only

ACCESS BITS OR EXECUTE ONLY



- Null Pointer

if (p == NULL)



Shared Libraries

- Shared code? Yes
- Shared data? Yes
- Same virtual address? What???
- Who decides a shared libraries virtual address?

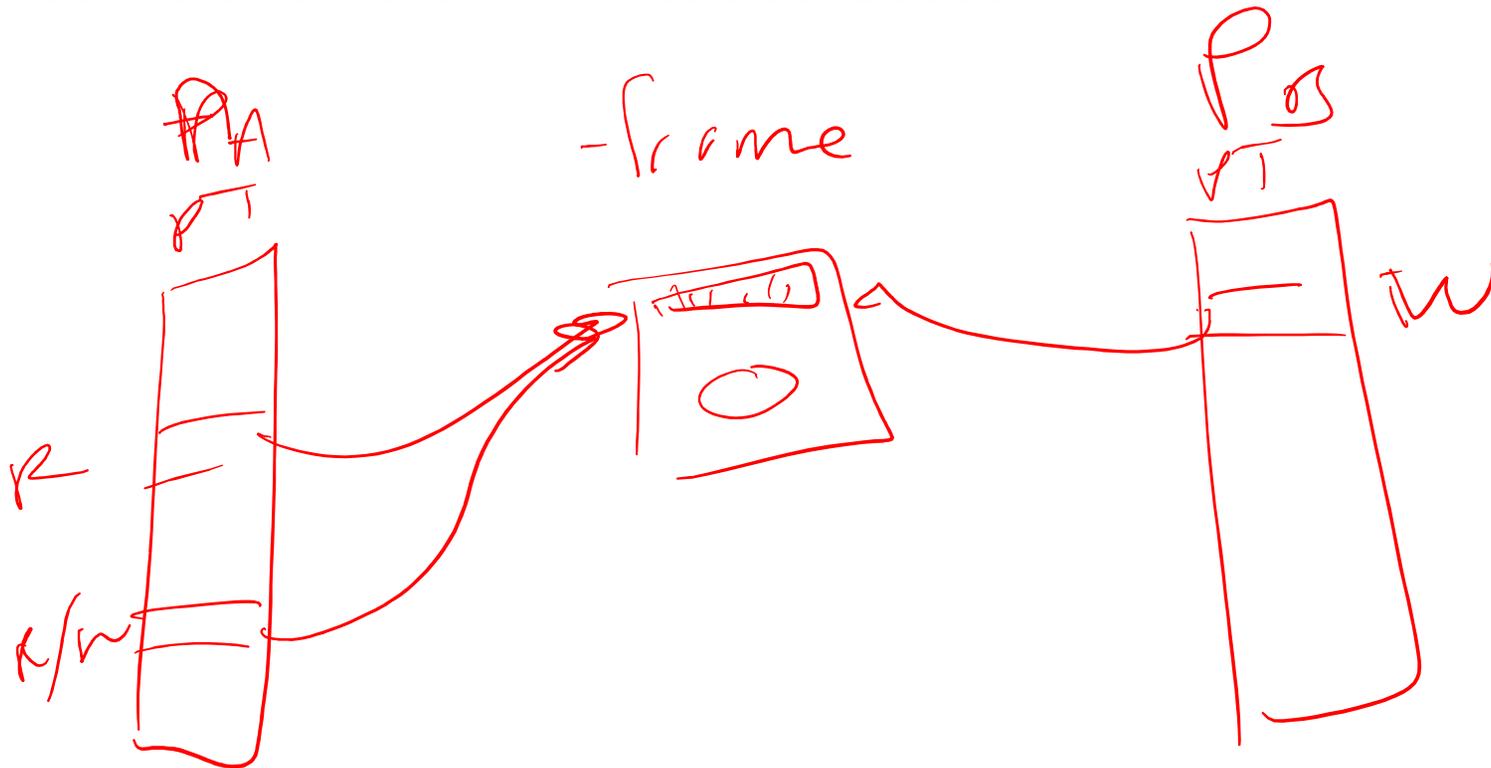


More functionality

- Generalizing the use of “shared memory”
 - Regions of two separate processes’s address spaces map to the same physical frames
 - Why? Faster inter-process communication
 - Just read/write from/to shared memory
 - Don’t have to make a syscall
 - Will have separate PTE’s per process, so can give different processes different access rights
 - E.g., one reader, one writer
- Copy-on-write (CoW), e.g., on fork()
 - Instead of copying all pages, create shared mappings of parent pages in child address space
 - Make shared mappings read-only for both processes
 - When either process writes, fault occurs, OS “splits” the page

Shared Memory

- Read or write access
- Same of different virtual addresses



Less familiar uses

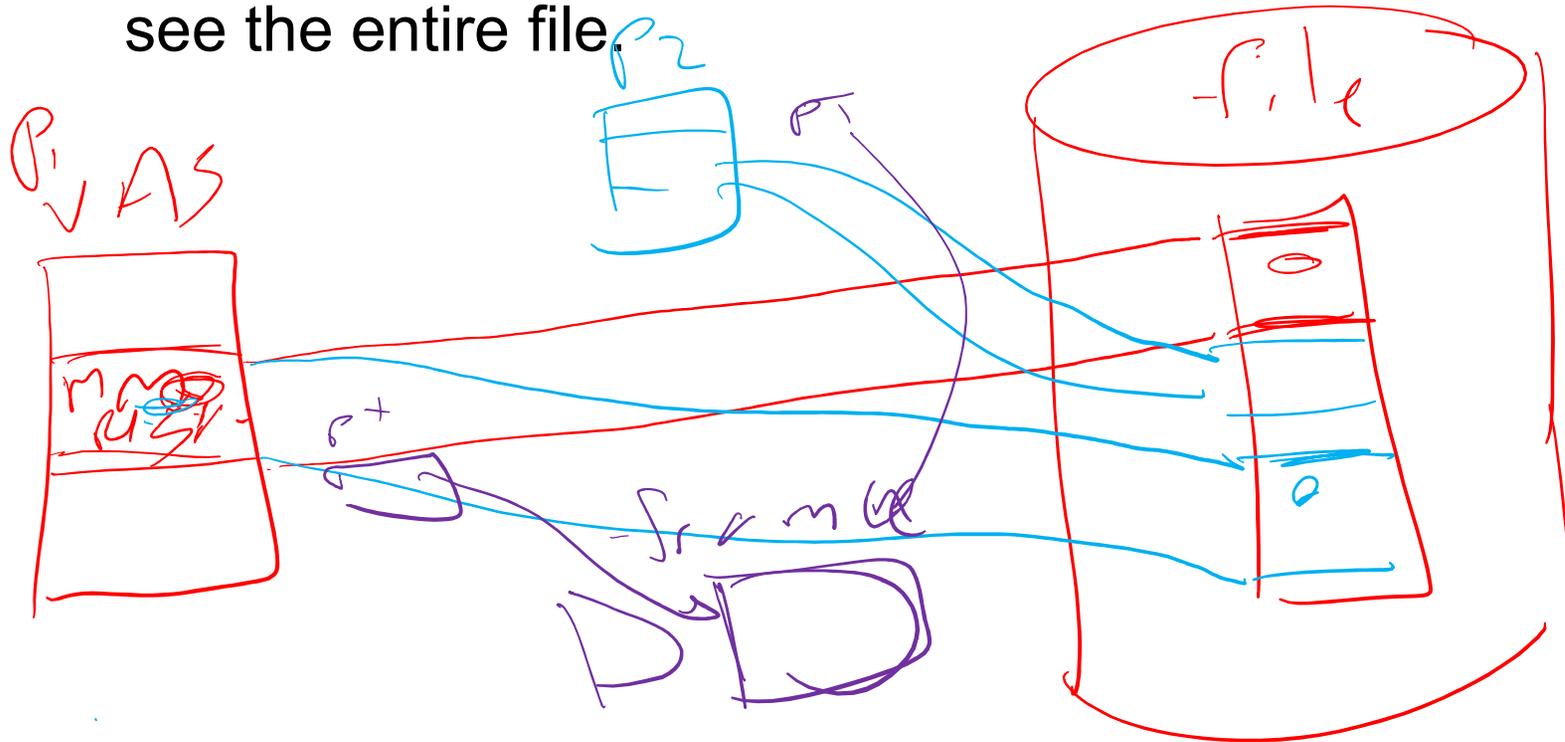
- Memory-mapped files
 - instead of using open, read, write, close
 - “map” a file into a region of the virtual address space
 - e.g., into region with base ‘X’
 - accessing virtual address ‘X+N’ refers to offset ‘N’ in file
 - initially, all pages in mapped region marked as invalid
 - Using that “table that looks like a page table” ...
 - OS reads a page from file whenever invalid page accessed
 - OS writes a page to file when evicted from physical memory
 - only necessary if page is dirty

Memory mapped files

- Imagine you have a pointer-based, in-memory data structure, like a tree
- You want to preserve it across runs
- Usual approach:
 - **Serialize** on way from memory to a disk file, **deserialize** on way from file back to memory
 - E.g., to serialize, perform a depth-first traversal, writing each node to disk as you go; to deserialize, do the opposite
- Potentially easier
 - Allocate tree nodes in a “region”
 - Treat the memory region as a file, using the memory-mapped file facility
 - Normal paging causes changes to be pushed to disk; the file is still there next time you run
 - What happens if you crash? Uh oh...

Mapped view of file

- Analogous to a visual text editor scrolling through a file. Like reading a book using a magnifying glass
- In Windows you create a region of memory to map a part of a file. Then move the view as necessary to see the entire file.

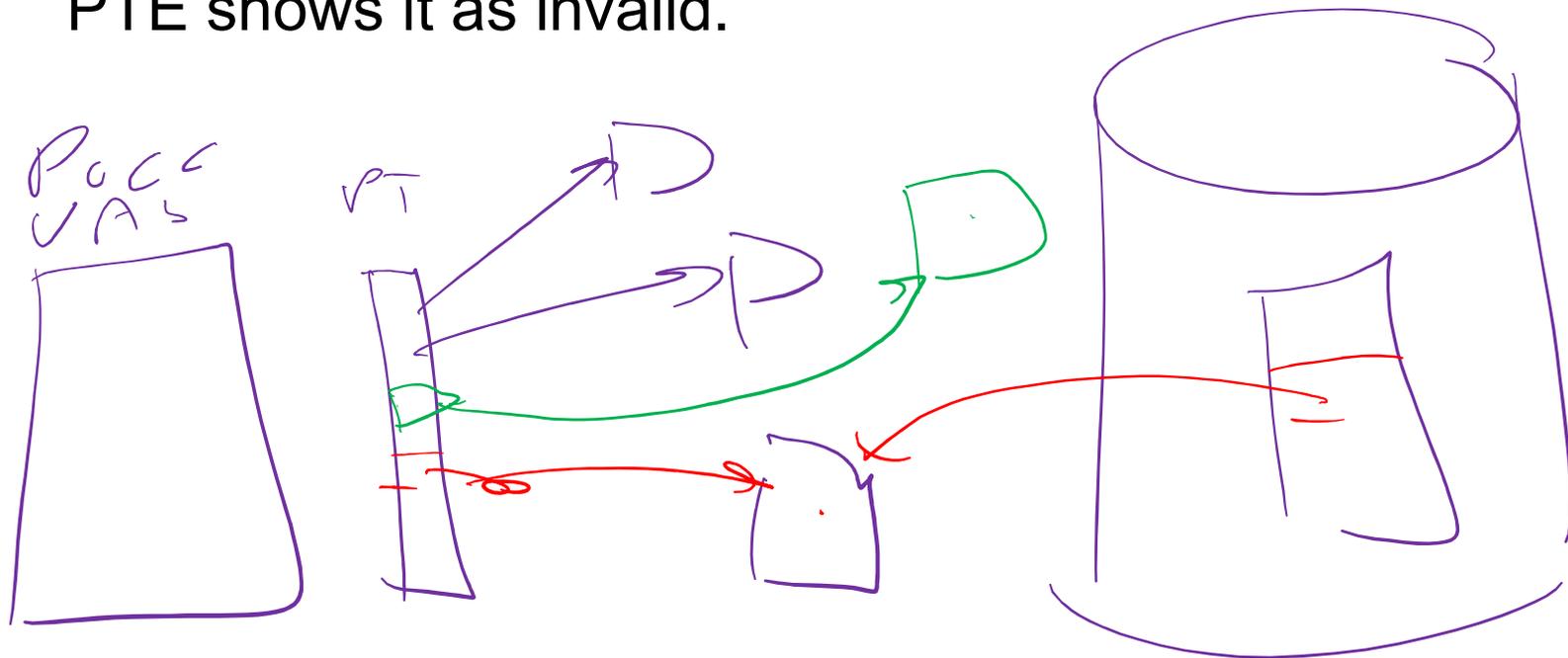


More unusual uses

- We saw that page replacement algorithms use the fact that “soft faults” are relatively cheap
 - Soft faults: faults on pages that are actually in memory, but whose PTE entries have artificially been marked as invalid
- That idea can be used whenever it would be useful to trap on a reference to some data item
- Example: debugger watchpoints
 - How?
- (The utility of this idea is limited by the fact that the granularity of detection is the page)

Hard versus soft faults

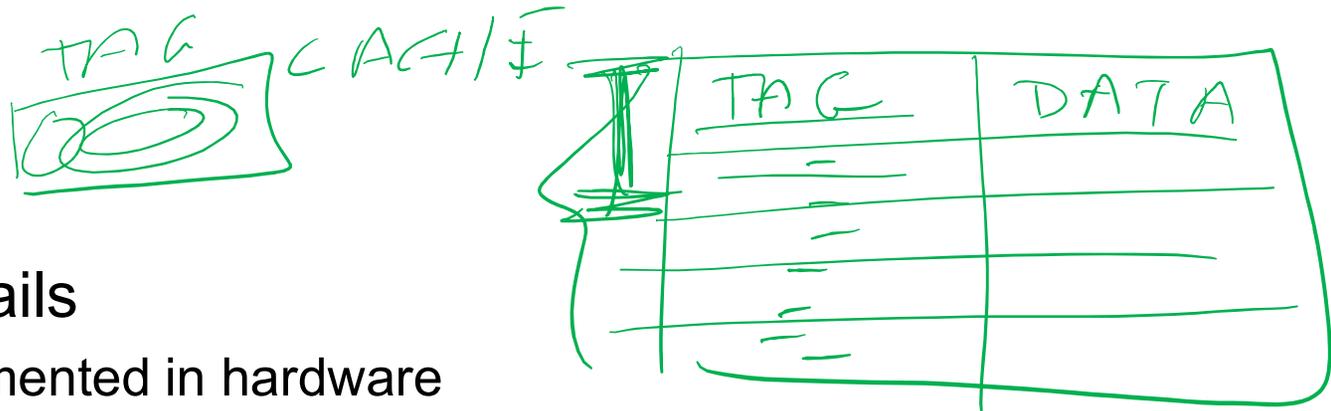
- **Hard fault** – the needed page is on backend store and needs to be brought into memory
- Soft fault – the needed page is in memory, but the PTE shows it as invalid.



Summary

- We know how address translation works in the “vanilla” case (single-level page table, no fault, no TLB)
 - hardware splits the virtual address into the virtual page number and the offset; uses the VPN to index the page table; concatenates the offset to the page frame number (which is in the PTE) to obtain the physical address
- We know how the OS handles a page fault
 - find or create (through eviction) a page frame into which to load the needed page
 - find the needed page on disk and bring it into the page frame
 - fix up the page table entry
 - put the process on the ready queue

- We're aware of two "gotchas" that complicate things in practice
 - the memory reference overhead of address translation
 - the need to reference the page table doubles the memory traffic
 - solution: use a hardware cache (TLB = translation lookaside buffer) to absorb page table lookups
 - the memory required to hold page tables can be huge
 - solution: use multi-level page tables; can page the lower levels, or at least omit them if the address space is sparse
 - this makes the TLB even more important, because without it, a single user-level memory reference can cause two or three or four page table memory references ... and we can't even afford one!



- TLB details

- Implemented in hardware

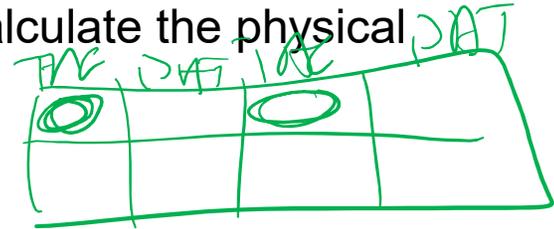
- **fully associative cache** (all entries searched in parallel)
- cache **tags** are virtual page numbers
- cache **values** are page table entries (page frame numbers)
- with PTE + offset, MMU can directly calculate the physical address

- Can be small because of locality

- 16-48 entries can yield a 99% hit ratio

- Searched *before* the hw or OS walks the page table(s)

- **hit**: address translation does not require an extra memory reference (or two or three or four) – “free”
- **miss**: walk the page table(s) to translate the address; this translation is put into the TLB, evicting some other translation; typically managed 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
 - Dereferencing a null pointer is an error
 - Inter-process memory protection
 - Shared libraries
 - Inter-process communication
 - Shared memory
 - Copy-on-write
 - Memory-mapped files
 - Soft faults (e.g., debugger watchpoints)