

CSE 451: Operating Systems

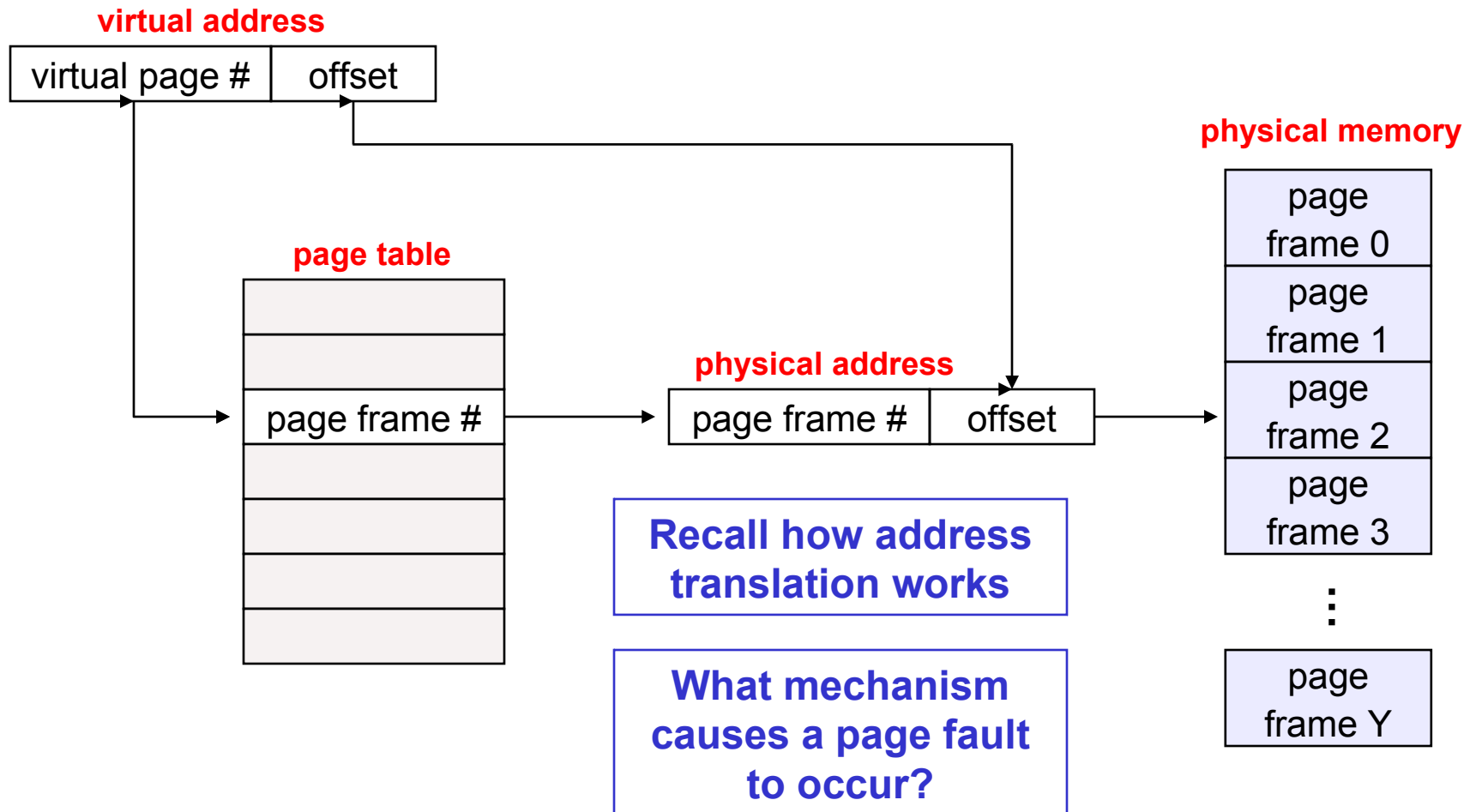
Spring 2010

Module 12

Page Table Management, TLBs, and Other Pragmatics

John Zahorjan
zahorjan@cs.washington.edu
Allen Center 534

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
 - 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
 - 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
 - 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)

“Issues”

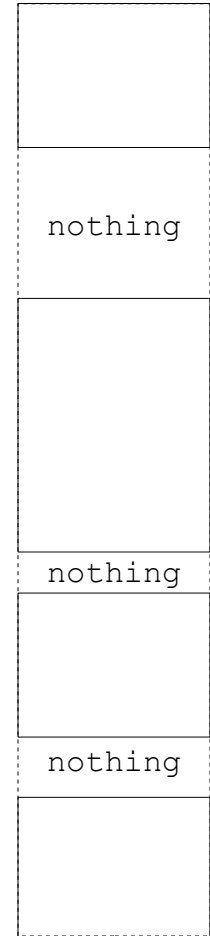
- (1) Memory reference overhead of address translation
 - 2 references per address lookup (page table, then memory)
 - **solution: use a hardware cache to absorb page table lookups**
 - translation lookaside buffer (TLB)
- (2) Memory required to hold page tables can be huge
 - need one PTE per page in 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 1 to (2): Page the page tables

- Simplest notion:
 - Put user page tables in a pageable segment of the OS address space
 - The OS page table maps the portion of the VAS in which the user process page tables live
 - Pin the OS's page table(s) in physical memory
 - never replace them, so you can never fault trying to access them
 - When need a user page table entry:
 - It's in OS virtual 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 the user page table isn't in real memory
 - That's just a regular page fault, handled in the usual way
- This isn't exactly what is done any longer
 - But it's a useful model for what's actually done
 - It's a component of what is actually done

Solution 2 to (2): Segment the VAS

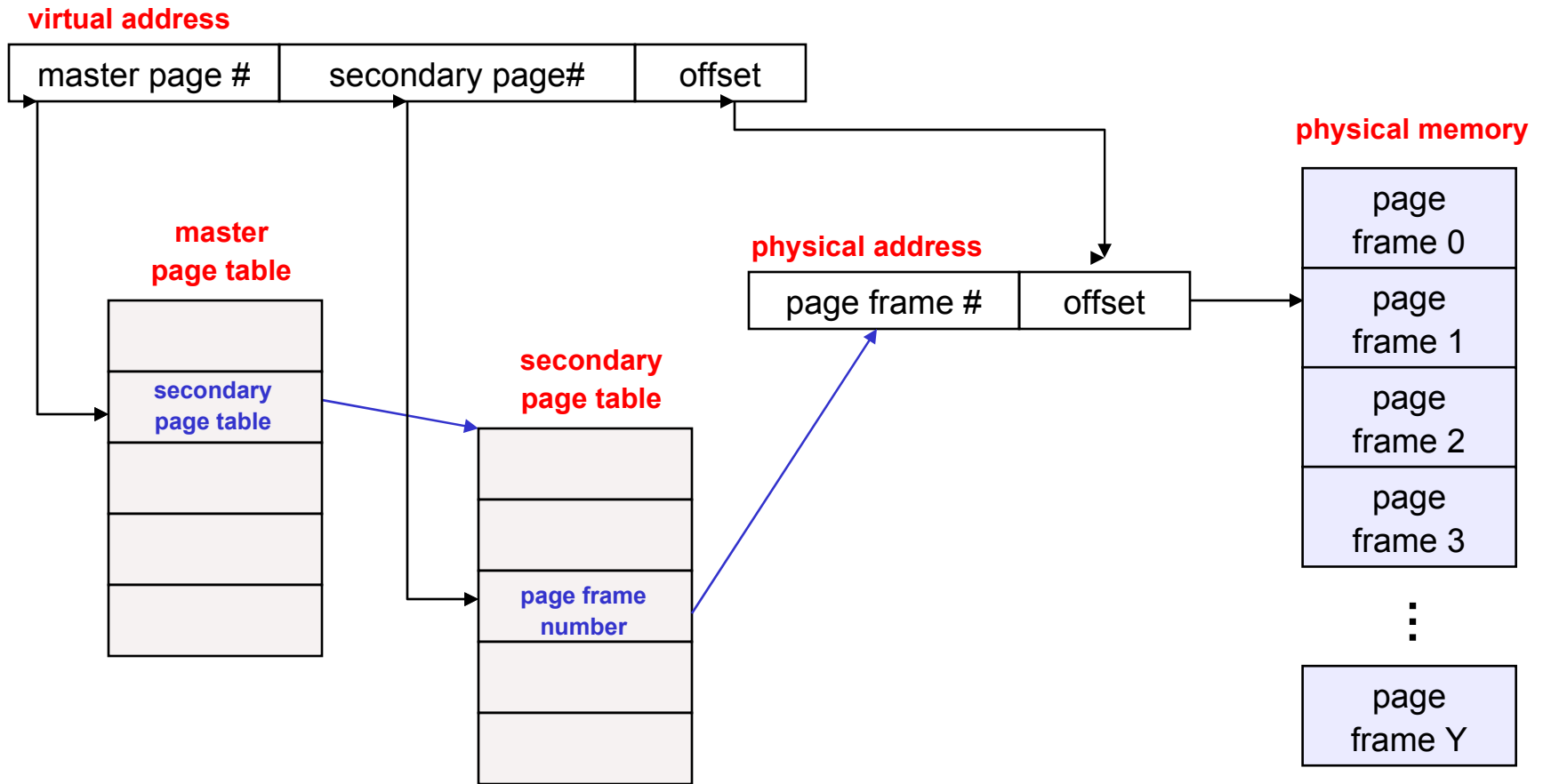
- How can we reduce the physical memory requirements of page tables?
 - observation: only need to map the portion of the address space that is actually being used (often a tiny fraction of the total address space)
 - a process may not use its full 32/48/64-bit address space
 - a process may have unused “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 uses 3-level page tables
- 68030 uses 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!)

Making it all efficient (CSE 378)

- 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

- 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
- 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 of virtualizing the x86...
- Software loaded TLB (OS)
 - TLB miss faults to OS, OS finds right PTE and loads TLB
 - must be fast (but, 20-200 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”
 - implemented in hardware, usually simple (e.g., LRU)

Functionality Enabled by Page Tables

- Code (instructions) is read-only
 - A bad pointer can't change the program code!
- Dereferencing a null pointer is an error
 - Why? Address 0 is a perfectly good address...
 - How? There's a "hole" in the VAS at page 0
 - So, never a valid page table entry for it
- Inter-process memory protection
 - Of course
- Shared libraries
 - All running C programs are using libc
 - Have only one (partial) copy in real memory, not one per process
 - How? All page table entries mapping libc are managed together, to point to same physical frame
 - DLL's in Windows...

More functionality

- Generalizing the use of “shared memory”
 - regions of two separate processes’ address spaces map to the same physical frames
 - Why? Fast inter-process communication
 - Just read/write from/to shared memory
 - Don't have to make a syscall to pass an integer to another process
 - Will have separate PTEs per process, so can give different processes different access privileges
 - E.g., one reader, one writer
 - Must the shared region map to the same VA in each process?
- Copy-on-write (COW), e.g., on fork()
 - instead of copying all pages, created shared mappings of parent pages in child address space
 - make shared mappings read-only in child space
 - when child does a write, a protection fault occurs, OS takes over and can then copy the page and resume client

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
 - 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
- Why (system view)?
 - Use of real memory for paging and use as a file cache (coming soon...) don't conflict
- Why (programmer view)?
 - See next slide...

Memory Mapped Files (cont.)

- Imagine you have some 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 into 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”
 - In place of pointers (addresses in the entire VAS), use offsets (distance from the 0th byte of the region) to link nodes
 - Now just connect the memory region and the file using the memory mapped file facility
 - Normal paging results in any changes made in-memory being pushed to file
 - The file is still there the next time you run. Just map-'n-go.

More “Unusual Uses”

- We saw that page replacement algorithms use the fact that “soft faults” are pretty cheap
 - Soft faults: faults on pages that are in memory, but whose PTE entries have artificially been marked invalid
- That idea can be used whenever it would be useful to trap on a reference to some data item
- Example: debugger watchpoints
 - Debuggers let you set a breakpoint on access to a variable
 - How can they detect an access?
 - Mark the page containing the variable as invalid
 - When a fault on that page occurs, look at the address and see if it corresponds to the variable
 - If so, break on the watchpoint
 - If not, simulate the instruction and continue program execution
- The effectiveness of this idea is limited by the fact that the granularity of detection is the page
 - E.g., if there are a lot of accesses to the other variables that happen to be on the page containing the watchpoint variable, the program will run very slowly

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!