

CSE 451: Operating Systems

Spring 2009

Paging & TLBs

Steve Gribble

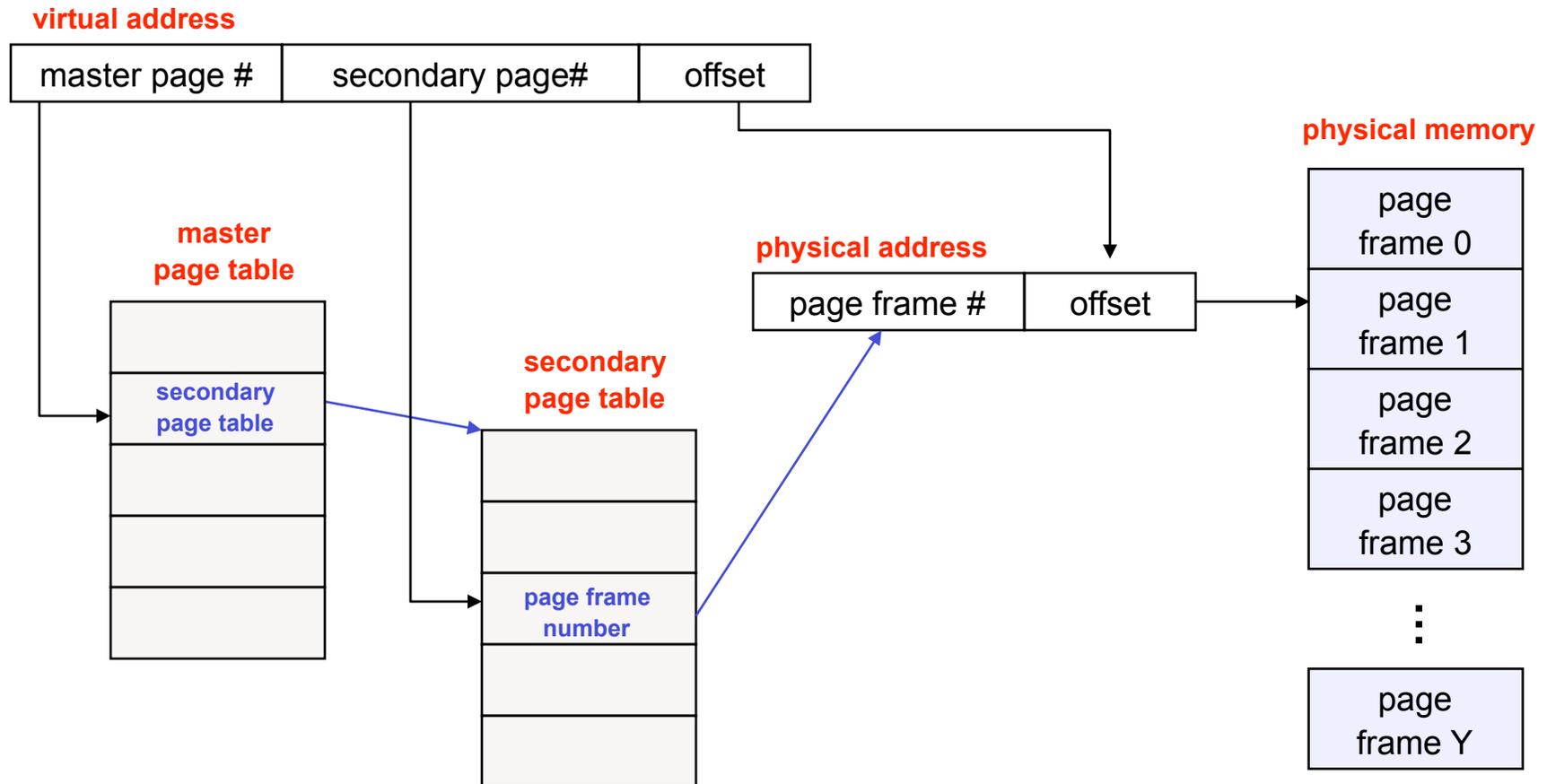
Managing Page Tables

- Last lecture:
 - size of a page table for 32 bit AS with 4KB pages was 4MB!
 - far too much overhead
 - how can we reduce this?
 - observation: only need to map the portion of the address space that is actually being used (tiny fraction of address space)
 - only need page table entries for those portions
 - how can we do this?
 - make the page table structure dynamically extensible...
 - all problems in CS can be solved with a level of indirection
 - two-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 + PFN = physical address
- Example:
 - 4KB pages, 4 bytes/PTE
 - how many bits in offset? need 12 bits for 4KB
 - want master PT in one page: $4\text{KB}/4\text{ bytes} = 1024\text{ PTE}$
 - hence, 1024 secondary page tables
 - so: master page number = 10 bits, offset = 12 bits
 - with a 32 bit address, that leaves 10 bits for secondary PN

Two level page tables



Addressing Page Tables

- Where are page tables stored?
 - and in which address space?
- Possibility #1: physical memory
 - easy to address, no translation required
 - but, page tables consume memory for lifetime of VAS
- Possibility #2: virtual memory (OS's VAS)
 - cold (unused) page table pages can be paged out to disk
 - but, addresses page tables requires translation
 - how do we break the recursion?
 - don't page the outer page table (called **wiring**)
- So, now that we've paged the page tables, might as well page the entire OS address space!
 - tricky, need to wire some special code and data (e.g., interrupt and exception handlers)

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 buffers
 - translates virtual page #s into PTEs (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
 - 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 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, who places translations 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
- 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

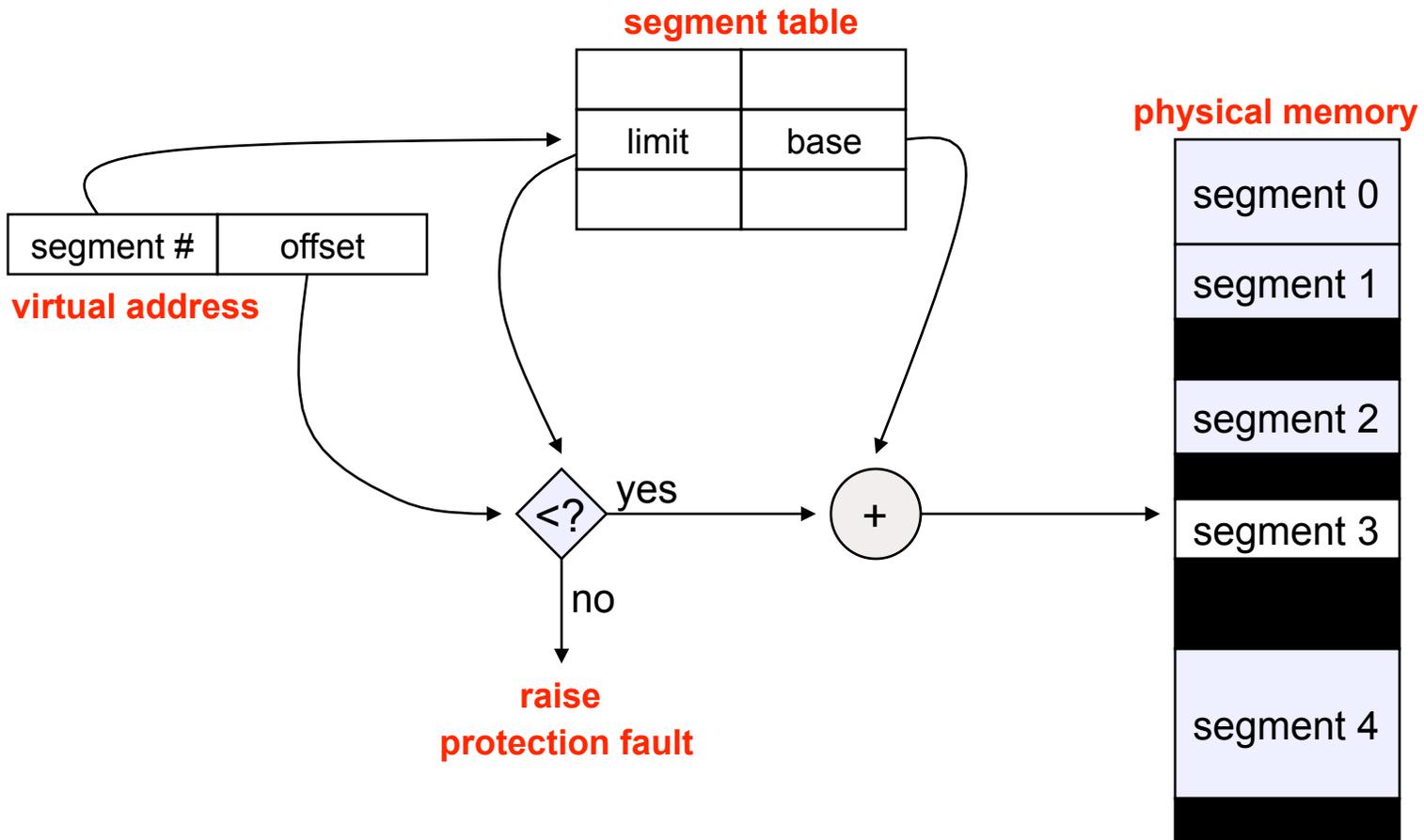
Managing TLBs (2)

- 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)

Segmentation

- A similar technique to paging is segmentation
 - segmentation partitions memory into logical units
 - stack, code, heap, ...
 - on a segmented machine, a VA is **<segment #, offset>**
 - segments are units of memory, from the user's perspective
- A natural extension of variable-sized partitions
 - variable-sized partition = 1 segment/process
 - segmentation = many segments/process
- Hardware support:
 - multiple base/limit pairs, one per segment
 - stored in a segment table
 - segments named by segment #, used as index into table

Segment lookups



Combining Segmentation and Paging

- Can combine these techniques
 - x86 architecture supports both segments and paging
- Use segments to manage logically related units
 - stack, file, module, heap, ...?
 - segment vary in size, but usually large (multiple pages)
- Use pages to partition segments into fixed chunks
 - makes segments easier to manage within PM
 - no external fragmentation
 - segments are “pageable”- don’t need entire segment in memory at same time
- Linux:
 - 1 kernel code segment, 1 kernel data segment
 - 1 user code segment, 1 user data segment
 - N task state segments (stores registers on context switch)
 - 1 “local descriptor table” segment (not really used)
 - all of these segments are paged
 - three-level page tables

Cool Paging Tricks

- Exploit level of indirection between VA and PA
 - shared memory
 - regions of two separate processes' address spaces map to the same physical frames
 - read/write: access to share data
 - execute: shared libraries!
 - will have separate PTEs per process, so can give different processes different access privileges
 - 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

Another great trick

- 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