

CSE 451: Operating Systems, 26au

L14: Virtual Memory

Instructor: Naomi Alterman

TAs:

Jonathan Trinh (Head)
Mahir Emran
Ted Guan
Jai Rungrachakanont
Soham Raut
Sagar Saby
Andrew Wood

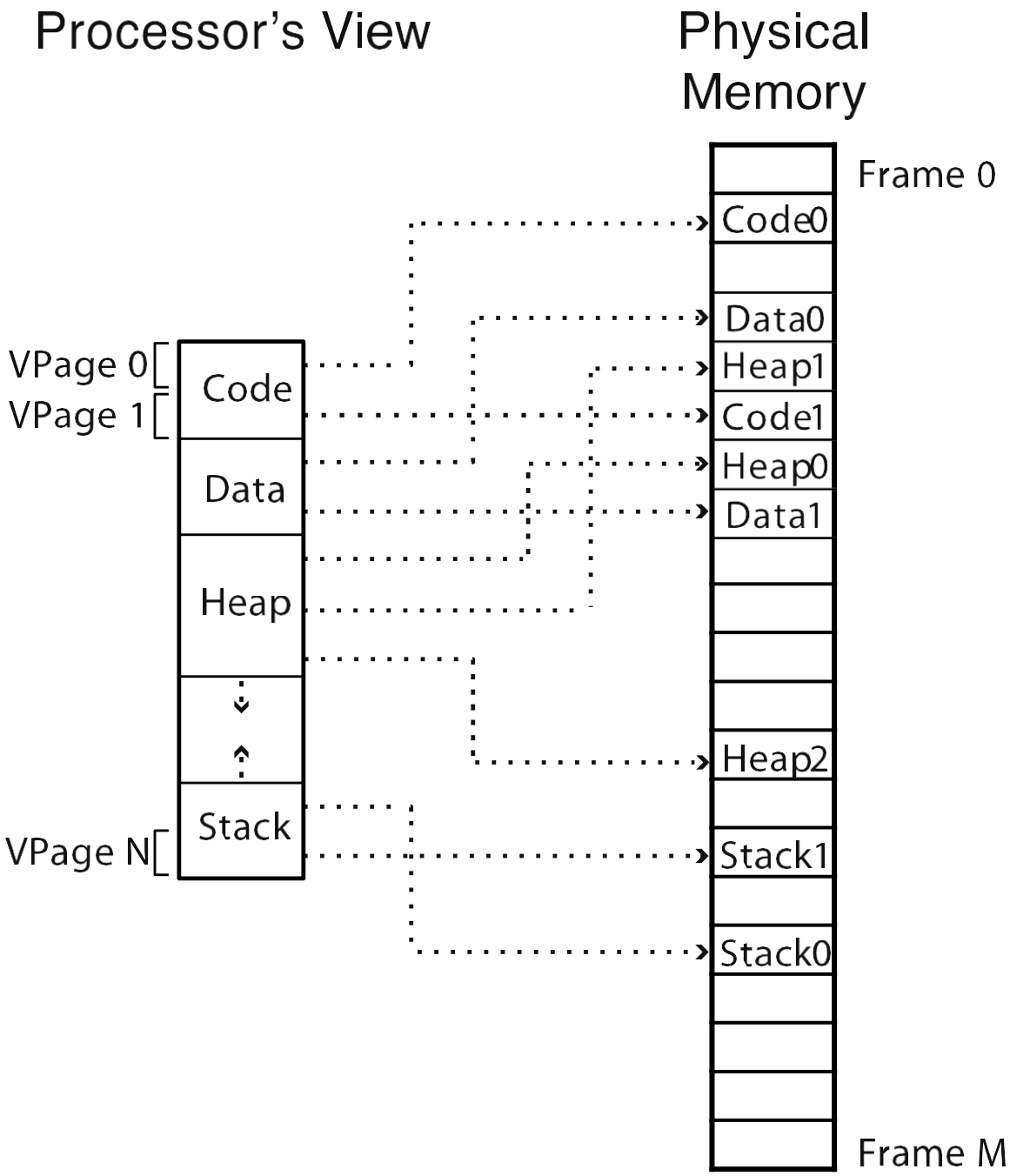
Adapted from slides by Tom Anderson

Address Translation Goals

- Memory protection (🙅 OS as referee)
  - Isolate process to its only memory
- Memory sharing (🧱 OS as glue)
  - Shared libraries, interprocess communication
- Portability (🪄 OS as illusionist)
  - Applications don’t need to be compiled to run in specific locations in memory
- Efficiency (OS as tired college student)
  - Growth of stack and heap don’t need to be contiguous
  - Reduce fragmentation and copying

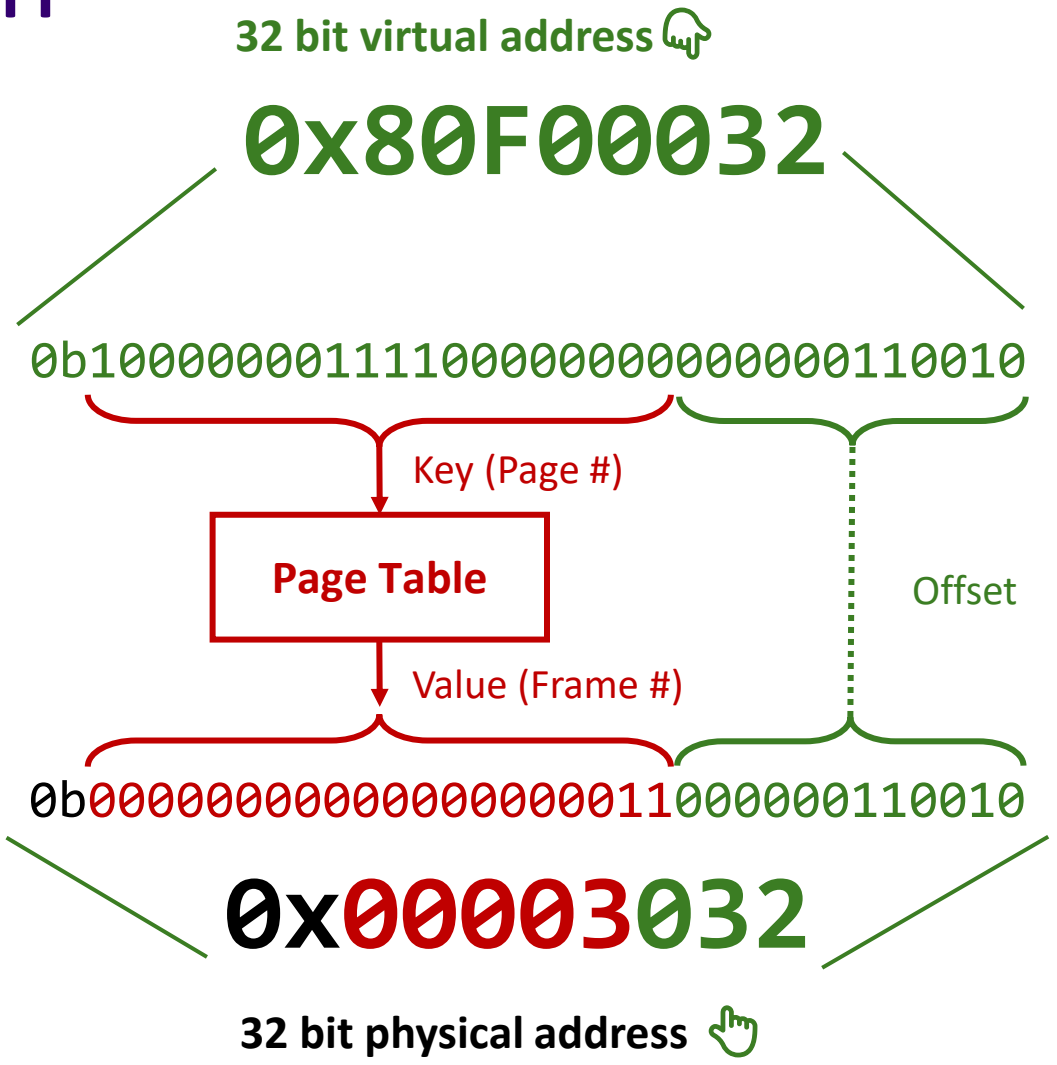
Virtual Memory

In all its glorious, gritty detail


Processor's View
Physical Memory
Frame 0
Code0
Data0
Heap1
Code1
Heap0
Data1
Heap2
Stack1
Stack0
Frame M
VPage 0
VPage 1
VPage N
Code
Data
Heap
Stack

Paged Address Translation

- Split physical memory into groups of N bytes called “**frames**”
- Lower bits of virtual address directly address into the frame (eg, act as an “**offset**”)
- Upper bits are used as a key in a lookup table for the rest of the physical address (the “**frame number**”)
- That lookup table is called the **page table.** The keys are called **page numbers**. The entries are called pages. **Pages map virtual addresses to physical frames.**



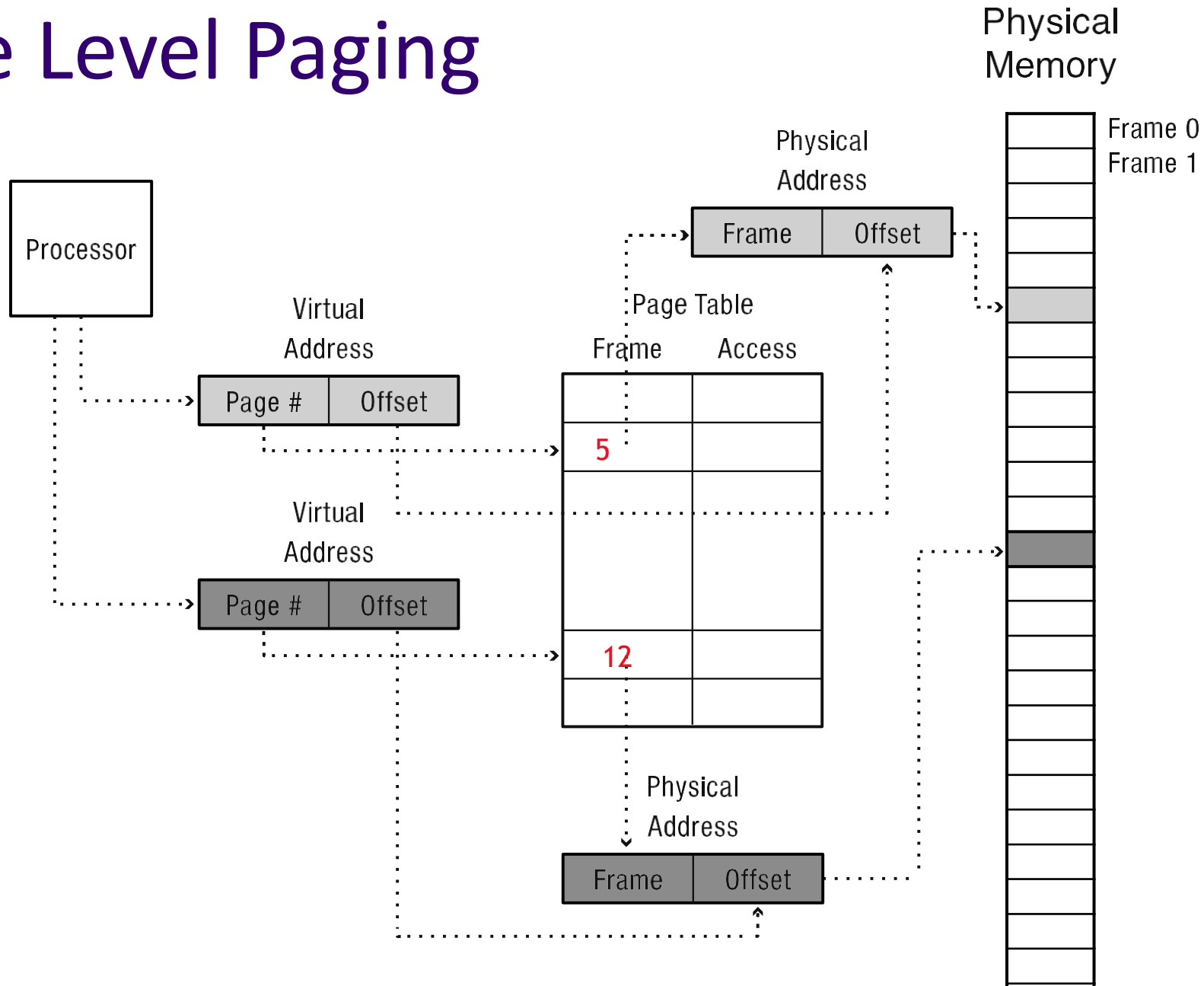
Page table, huh?

- A pretty large data structure (there's a lot of memory out there!)
  - We need a **different page table for each process**, since the page table effectively dictates the virtual address space
  - Page tables *themselves* **live in RAM**
  - **Format** is specified by **CPU architecture**

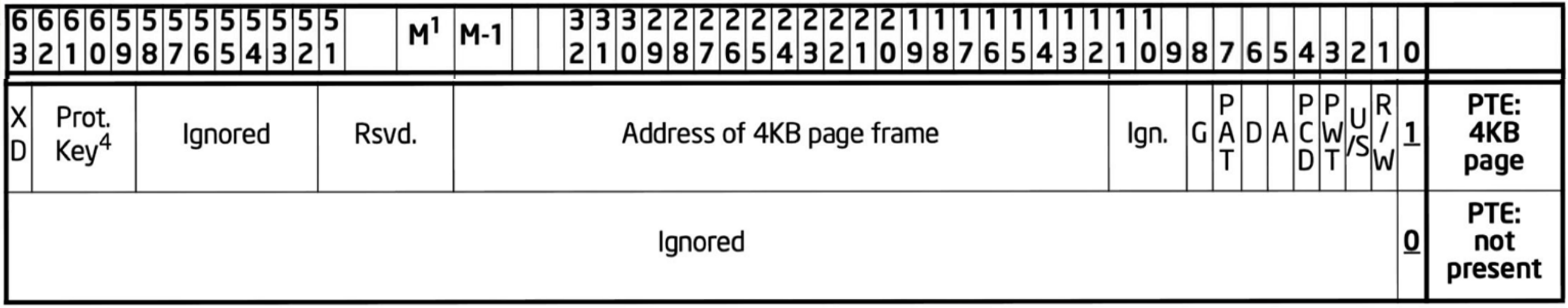
- **OS's job:**
  - **Write page tables** somewhere in physical memory
  - **Point** hardware to base address of our desired page table (on x86, the "**CR3**" **register**)

- **Hardware's job:**
  - Run all memory accesses through page table according to **address translation** algorithm

Single Level Paging



Page Table Entry (Intel 64 bit)



PTE not present: Unused virtual pages do not need a physical page frame

- Trap into kernel if process references memory on that page

Interesting PTE bits

- **0 (“P”):** entry valid?
- **1 (“R/W”):** read-only if bit is 0 (useful for code pages)
- **2 (“U/S”):** permission level (if 0, only kernel is allowed to access the page)
- **5 (“A”):** indicates whether the data in this page has been used/ “accessed” **(set by hardware)**
- **6 (“D”):** indicates whether the data in this page has been *modified* **(set by hardware)**

Finding unused frames

- OS can get away with a brute force implementation:
  - Use a **bitmap** where each bit indicates if a physical frame is in use
  - Can use fancy x86 instructions to search for 0 bits
  - Number of bits scales reasonably with amount of installed physical RAM
    - 8 GB ram, 4kb frame size: 2,000,000 pages
      1 bit/page: 250 KB
    - Just need one frame bitmap for the entire machine
    - That's not too bad

Let’s take a look, shall we?

- On Linux:
  - Process virtual memory maps are inspectable!
  - See “`/proc/[PID]/maps`” and “`/proc/[PID]/pagemaps`”
  - Also see: https://www.kernel.org/doc/html/latest/filesystems/proc.html

Paging Questions

- With paging, what is saved/restored on a process context switch? CR3 register!
- What are the tradeoffs of a small page size?
- What are the tradeoffs of a large page size?

45 bit page # $\Rightarrow 2^{45}$ page entries

8B/entry $\Rightarrow 2^{48}$ B (256 TB) per process

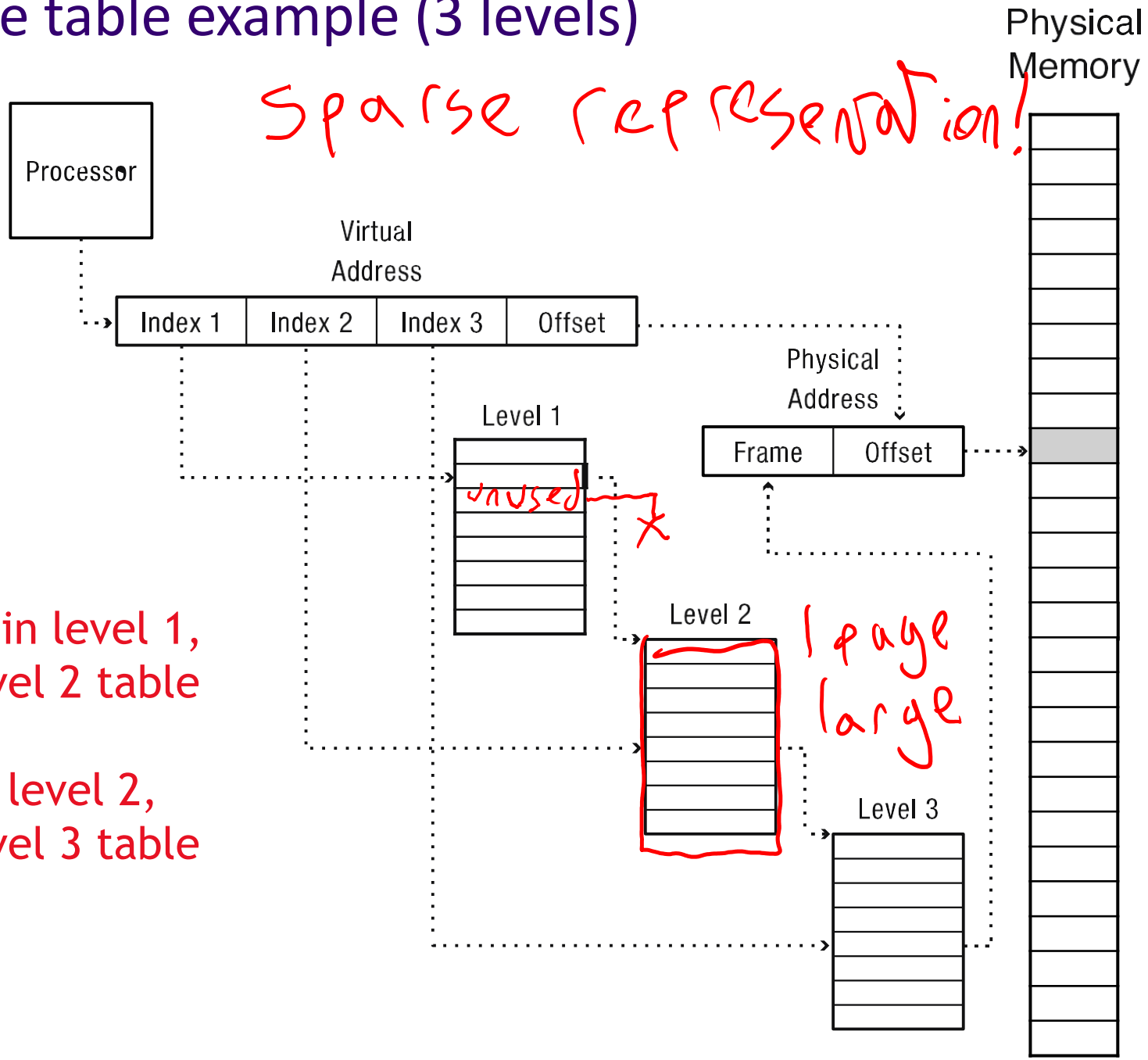
Tradeoffs: Single Level Page Table

- Pros?
  - Simple memory allocation
  - Safe
  - Efficient as long as page size not too large (internal fragmentation)
  - Page permission can prevent overwriting code by accident
  - Can share physical page frames (two copies of same program)
  - Unused virtual pages don’t need to be backed by physical frames
- Cons?
  - An extra memory lookup on every memory reference
  - Need page table to map entire virtual address space
  - Page table itself needs to be contiguous in memory (fragmentation)

x86 Multilevel Paging

- Each level of page table fits in one 4KB page
- 32-bit: **two level** page table
  - Page table entry is 32 bit, so 2^10 entries per page
- 64-bit: **five level** page table
  - Page table entry is 64 bit, so 2^9 entries per page
- Omit sub-tree if page table entry is invalid
  - Good for sparse address space

Multi-level page table example (3 levels)

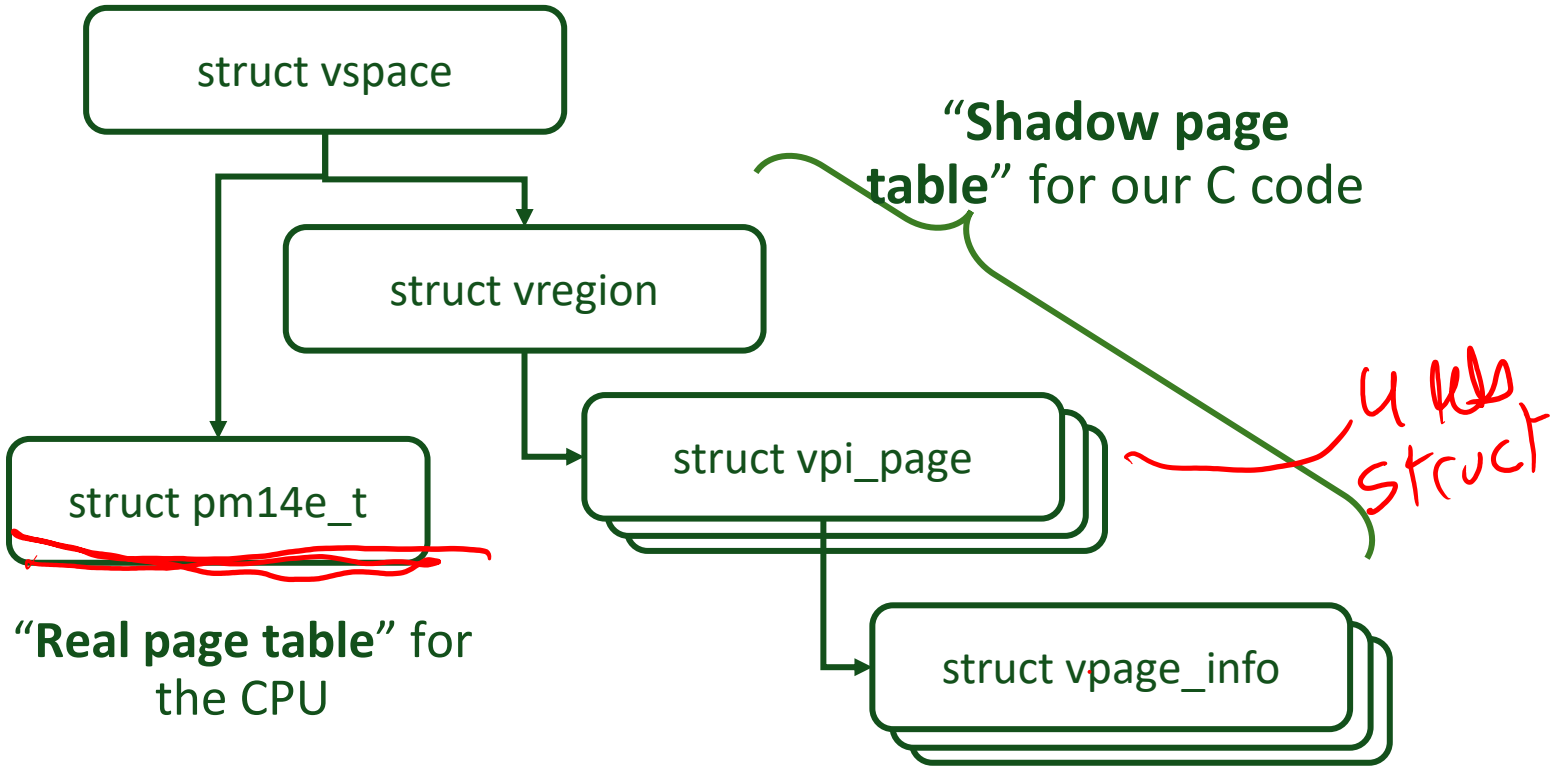


If PTE is invalid in level 1, don't need a level 2 table

If it is invalid in level 2, don't need a level 3 table

Oh hey, it's been a while

- xk: inc/vspace.h, kernel/vspace.c





Class concluded here on 10/30 and will resume on 11/2



Walking the hardware page table in xk

- See:
  - kernel/x86_vm64.c : `walkpm14()`
  - inc/mmu.h
  - inc/memlayout.h



Hmm…



…why do we bother keeping a shadow page table? Why not just use the real one?

Page sharing between processes

- Set page table entry in each process to point to same page frame (4KB)
  - Or pointer to page table (both point to same page table) (2MB)
  - Or pointer to pointer to page tables (both point to same page directory) (1GB)
  - Or pointer to pointer to pointer to page tables (0.5TB)
    - Example: kernel portion of address space

- This means we can't just use a bitmap to track used frames
  - Upgrade to a structure called a "**core map:**" an array of information about each physical page frame
    - Set of processes pointing to that page frame
    - Reference count (when it goes to zero, reclaim!)

Tradeoffs: Multilevel Paging

Pros

- Efficient memory allocation
- Efficient for sparse addresses
- Efficient disk transfers
- Easier to build translation lookaside buffers
- Variable granularity for protection/sharing

Cons

- Every memory access involves 6 memory reads in the general case

Nifty Paging Tricks

Hooray

Side effect of paging: lots of hooks for OS intervention

- Effectively, the kernel can now regain control whenever a program reads or writes a virtual memory location “of interest”

- This lets us do *a lot* of cool stuff. Examples (detail to follow):
  - Zero on reference
  - Copy on write
  - Fill on demand
  - Demand paging
  - Memory mapped files



Animation from [Compact Objects by Swardy]

Nifty Trick 1: "Allocate on Reference"

- We don't know how much stack space a process is going to need
- So, we'll always keep an unmapped page at the bottom of the stack
- When stack grows into this page, the first reference to unmapped memory will generate a **page fault**
  - Transfer is forced through the interrupt vector into OS kernel
  - Allocate an additional page of memory
  - Zero the page to avoid accidentally leaking information!
  - Update the page table
  - **Resume process at the instruction that caused the page fault**

Oh by the way



How should we “zero out” a page? A for loop?

When should we “zero out a page? On allocation? While idle?

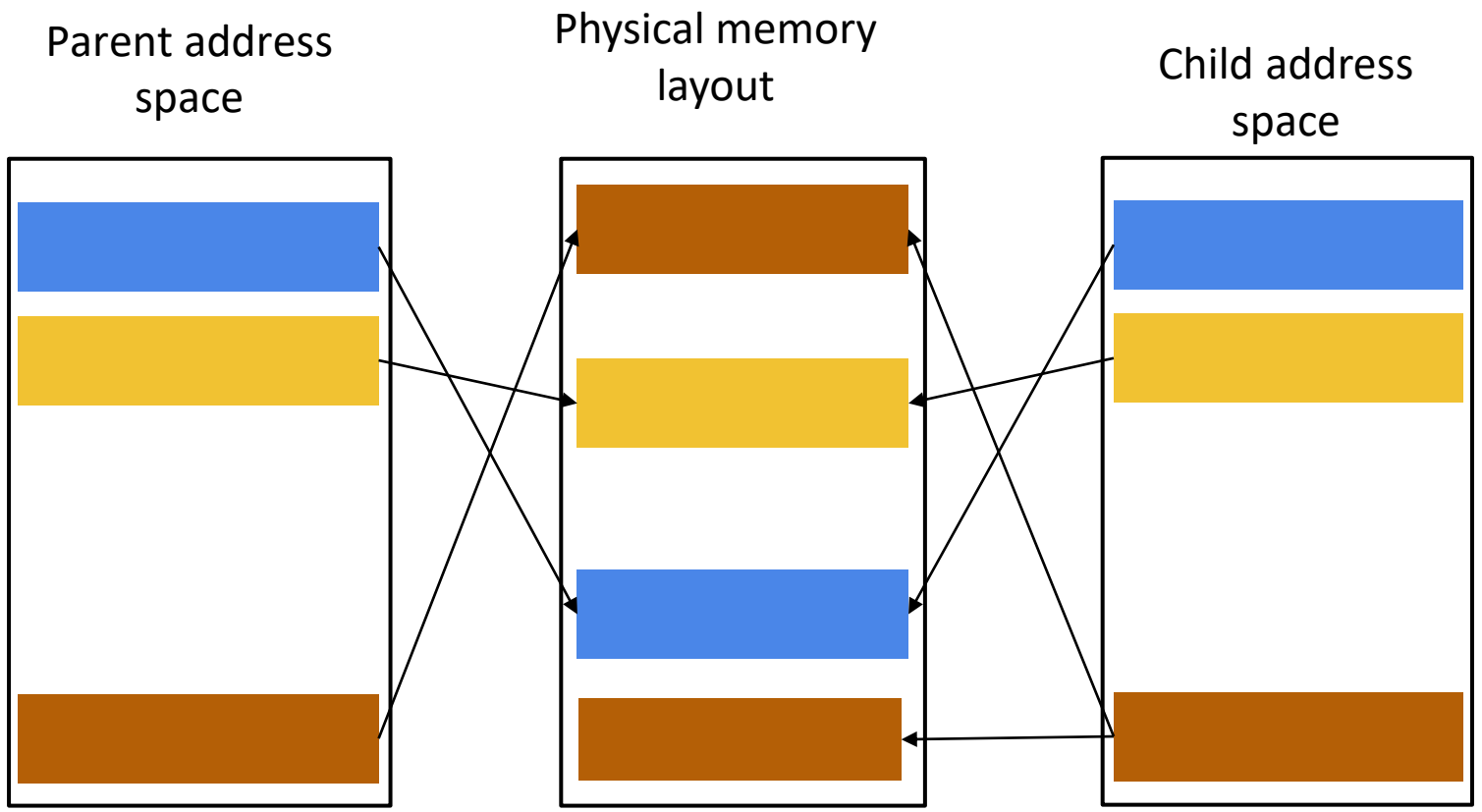
Nifty Trick #2: "Copy on Write"

- *Most* of the time, *most* memory duplicated by a `fork()` call won't ever get used
  - Paging allows an efficient "common case" with elegant fallback to "unusual cases"

- Efficient forking procedure:
  - On fork(), copy page table of parent into child and mark ***all*** pages (in both parent **and** child page tables) as read-only
  - Start child process; resume parent
  - Trap into kernel on write (again, child ***or*** parent)
  - Copy page and install in child's page table
  - Mark both old page and new copy as writeable again
  - Resume execution

- With multilevel page tables, can efficiently mark a whole region (top level page table entry) as read-only

Copy-on-write Illustrated



- Parent initially R+W for all data / stack pages
- After fork(), all pages become “read-only”
- Any pages modified in either cause a “clone” to happen. Both the original and the clone are now R+W again

Nifty Trick 3: “Fill On Demand”

- Loading a program requires the bytes of its ELF file to be copied from disk to RAM
  - Sometimes disks are busy, sometimes ELFS are big
  - And regardless, I want “ls” to start and finish faster than I can perceive

- Trick:
  - Load the first code page and load the rest in the background with a kernel thread
  - On page fault, transfer missing page immediately

Nifty Trick 3.5: "Memory Mapped Files"

- Contents of a file get mapped to be readable/writable at a given region of memory
  - Different from traditional "faucet of bytes" interface that we have with open()/read()/write()
  - Reads and writes happen with standard assembly memory instructions rather than syscalls (yay)
  - On POSIX: `mmap()` syscall

- In reality:
  - The OS is acting as an 🪄 invisible translation layer 🪄 between your userspace load/store instructions and the OS's filesystem code
  - mmap() marks mapped region's pages as invalid
  - Reading from page traps to kernel, which uses FS functions to load the bytes
  - Writing to page causes its "dirty bit" to get set in the page table.
  - When the OS gets a chance (through background threads or on file close), uses FS functions to write contents back to disk

Nifty Trick 4: “Demand Paging”

- We can use virtual memory to give the appearance of more RAM than is actually installed in the machine
  - ….software that conjures more hardware? what?

- ***Much*** more on this in a few lectures ;) ;)

Address Translation Uses (Summary)

- Process isolation
- Shared code segments
- Dynamic memory allocation
- Efficient usage of physical memory
- Early Program initialization
- Memory mapped files
- Demand-paged virtual memory

Address Translation (more)

- Cache management
  - Page coloring
- Program debugging
  - Data breakpoints when address is accessed
- Zero-copy I/O
  - Directly from I/O device into/out of user memory
- Efficient interprocess communication
  - Shared regions of memory between processes
- Checkpointing/restart
  - Transparently save a copy of a process without stopping the program while the save happens

Address Translation (even more)

- Persistent data structures
  - Implement data structures that can survive system reboots
- Process migration
  - Transparently move processes between machines
- Information flow control
  - Track what data is being shared externally
- Distributed shared memory
  - Illusion of memory that is shared between machines

Optimization: TLBs



s 6 memory reads i

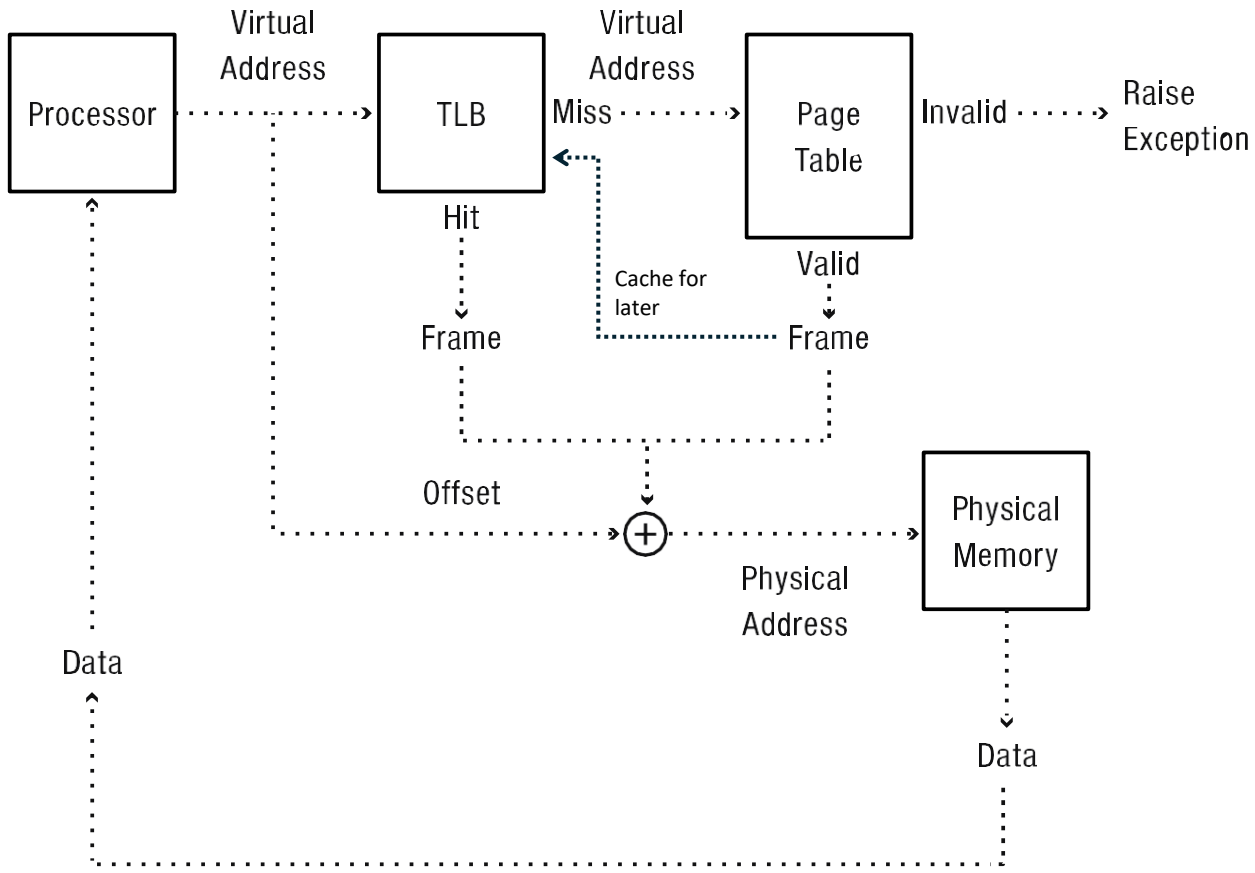
Caching to the rescue

- Translation lookaside buffer (**TLB**)
  - Hardware cache of recent “virtual page -> physical frame” translations
  - Hardware uses cached translation on hit
  - Hardware does the traditional multi-level walk on miss

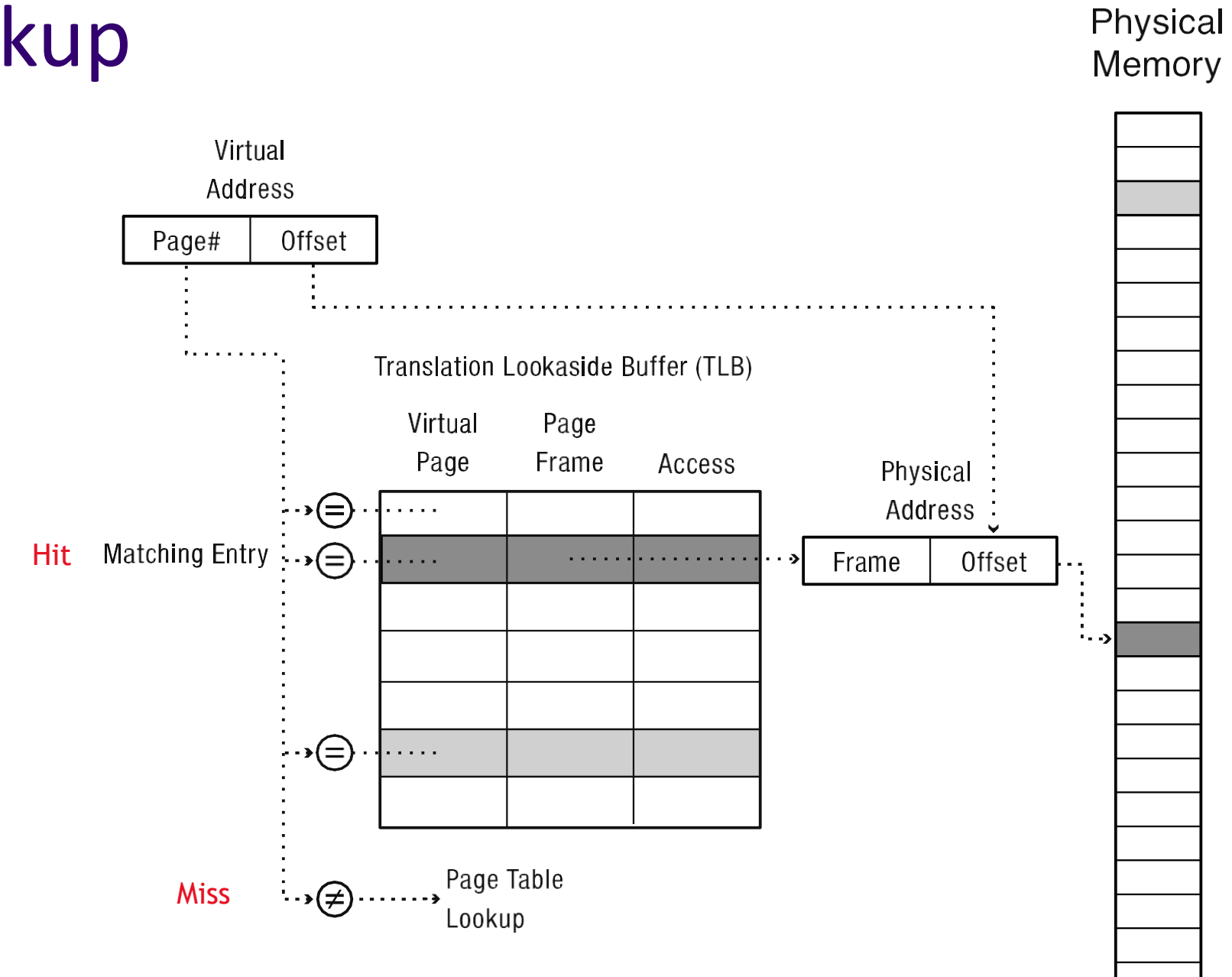
- Cost of translation =
  Cost of TLB lookup + Prob(TLB miss) * cost of page table lookup

- Size varies, usually on the order of ~64 entries

Virtual Memory Diagram (with TLB)



TLB Lookup


Physical Memory
Virtual Address
Page#
Offset
Translation Lookaside Buffer (TLB)
Virtual Page
Page Frame
Access
Physical Address
Frame
Offset
Hit
Matching Entry
Miss
Page Table Lookup

Software intervention #1: context switches

- In most cases, the TLB is “transparent” to software
  ….but not all of them

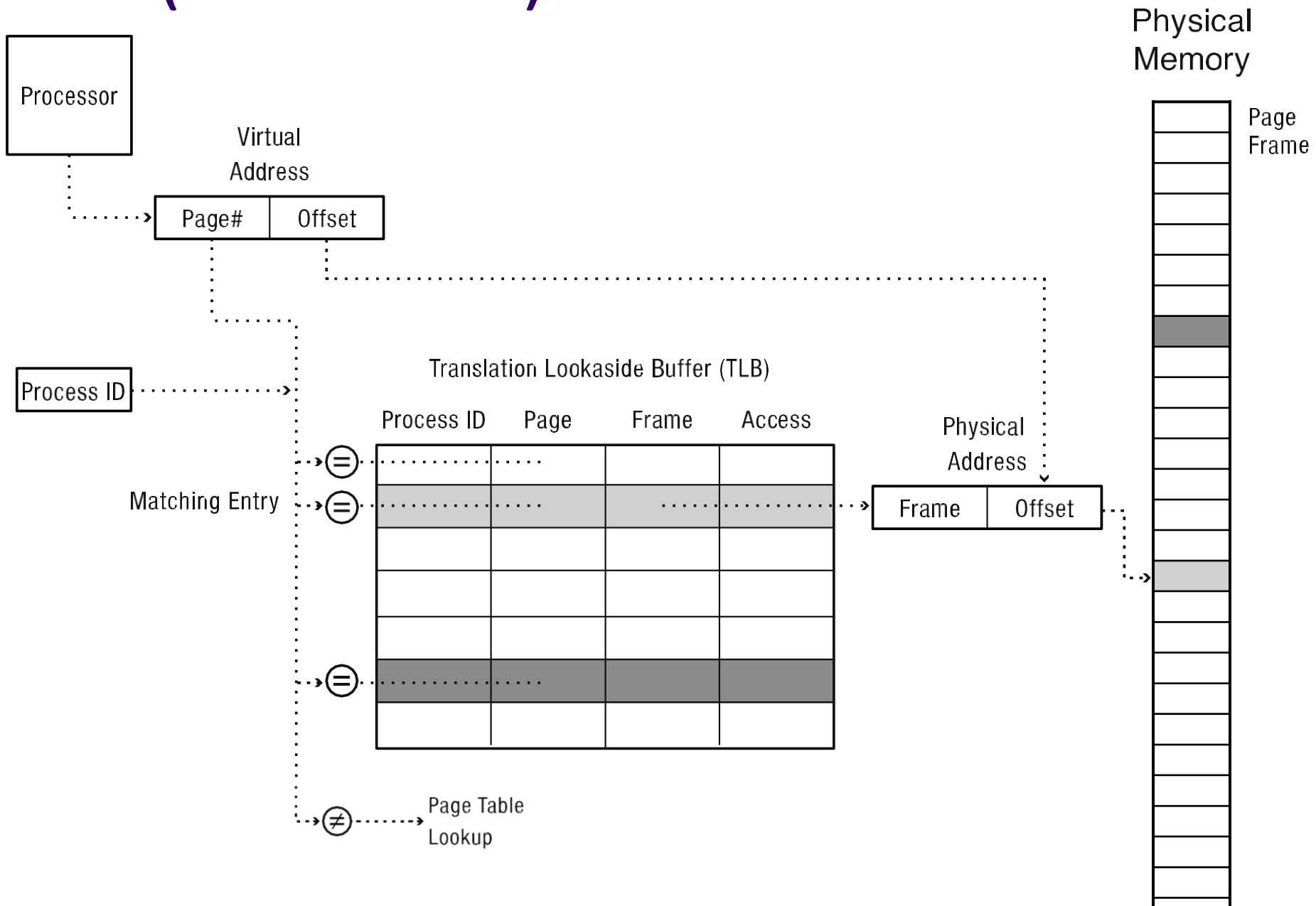
- Can we reuse TLB entries across processes?
  - 🪄OS As Illusionist: **No!** Why?
  - 👩‍⚖️OS As Referee: **No!** Why?

- Brute force solution: flush the TLB
  - This is what we do on x86 processors
  - Writing to CR3 register clears the TLB

Optimization optimization: tagged TLBs

- Mark each TLB entry's corresponding process
  - Cache only hits when there is an address match *and* a PID match
- Hardware told the PID with a special register (the "Process-Context Identifier")
- Implemented by concatenating PID bits to address in TLB key

Tagged TLBs (illustrated)



Software intervention #2: Permission Changes

- We often want to change the read/write/access permissions on a page
  - For copy-on-write, zero-on-reference, …
- TLB may contain old permission bits
  - OS must ask CPU to purge TLB entry
  - On x86: INVLPG instruction
- On a multicore system: “**TLB shootdown**”
  - OS must ask ***each core*** to purge TLB entry

TLB Shootdown



Processor 1 TLB

Processor 2 TLB

Processor 3 TLB

	Process ID	VirtualPage	PageFrame	Access
Processor 1 TLB =	0	0x0053	0x0003	R/W
Processor 1 TLB =	1	0x40FF	0x0012	R/W
Processor 2 TLB =	0	0x0053	0x0003	R/W
Processor 2 TLB =	0	0x0001	0x0005	Read
Processor 3 TLB =	1	0x40FF	0x0012	R/W
Processor 3 TLB =	0	0x0001	0x0005	Read

TLB Shootdown Rabbithole

https://blog.himanshuanand.com/2026/09/i-missed-one-tlb-shootdown/

