

# CSE 451: Operating Systems, 26au

# L15: More Virtual Memory

**Instructor:** Naomi Alterman

**TAs:**

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

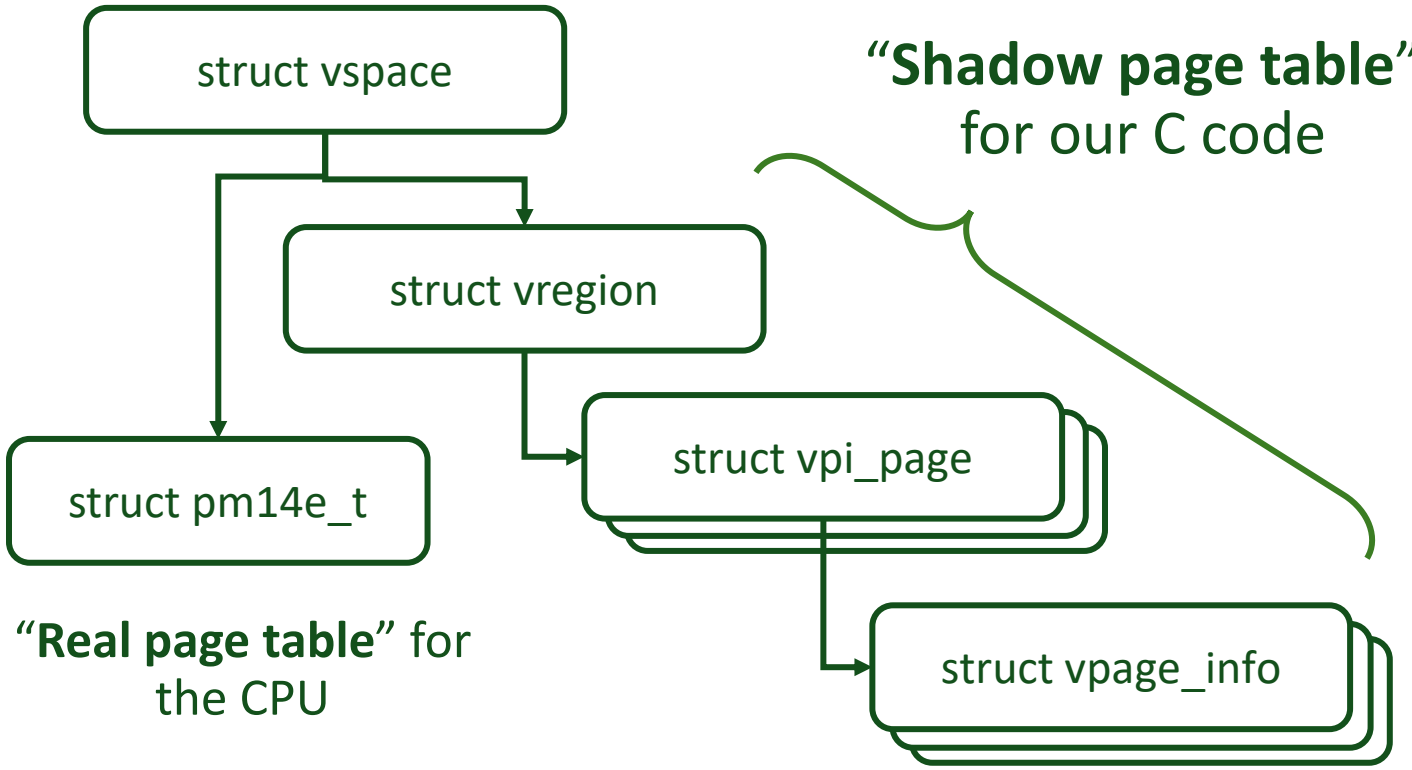
Adapted from slides by Tom Anderson

# Administriviatta

- Midterm review in class this Wednesday
  - Post/heart topics you'd like us to cover in this Ed thread
  - The review will **not be recorded** as we may break into smaller groups to support more personalized studying help.

# Oh hey, it's been a while

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






# 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?

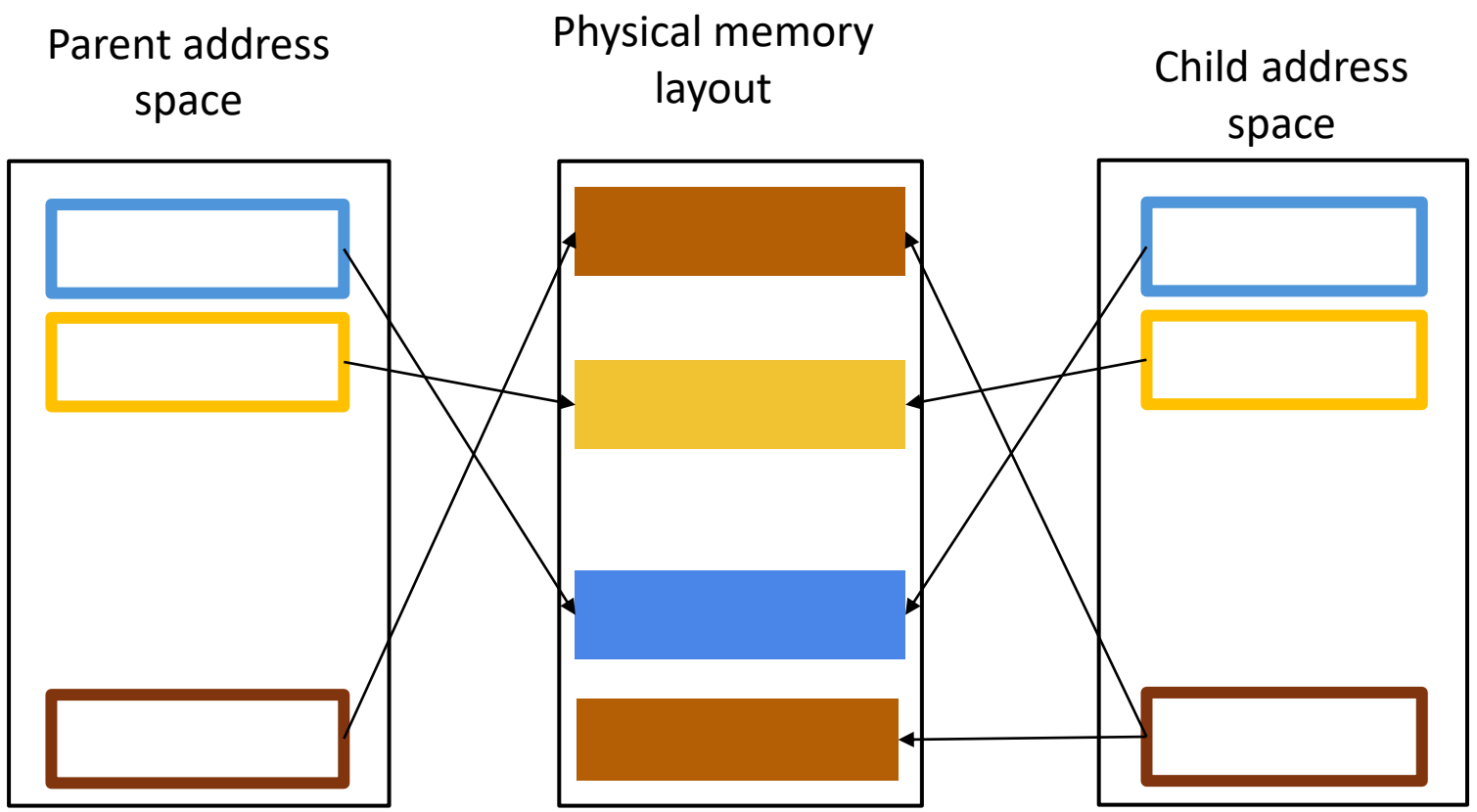
# **Nifty Trick #2**: "Copy on Write"

- *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

Class concluded here on 11/2 and will resume on 11/9 (after the midterm)



# **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

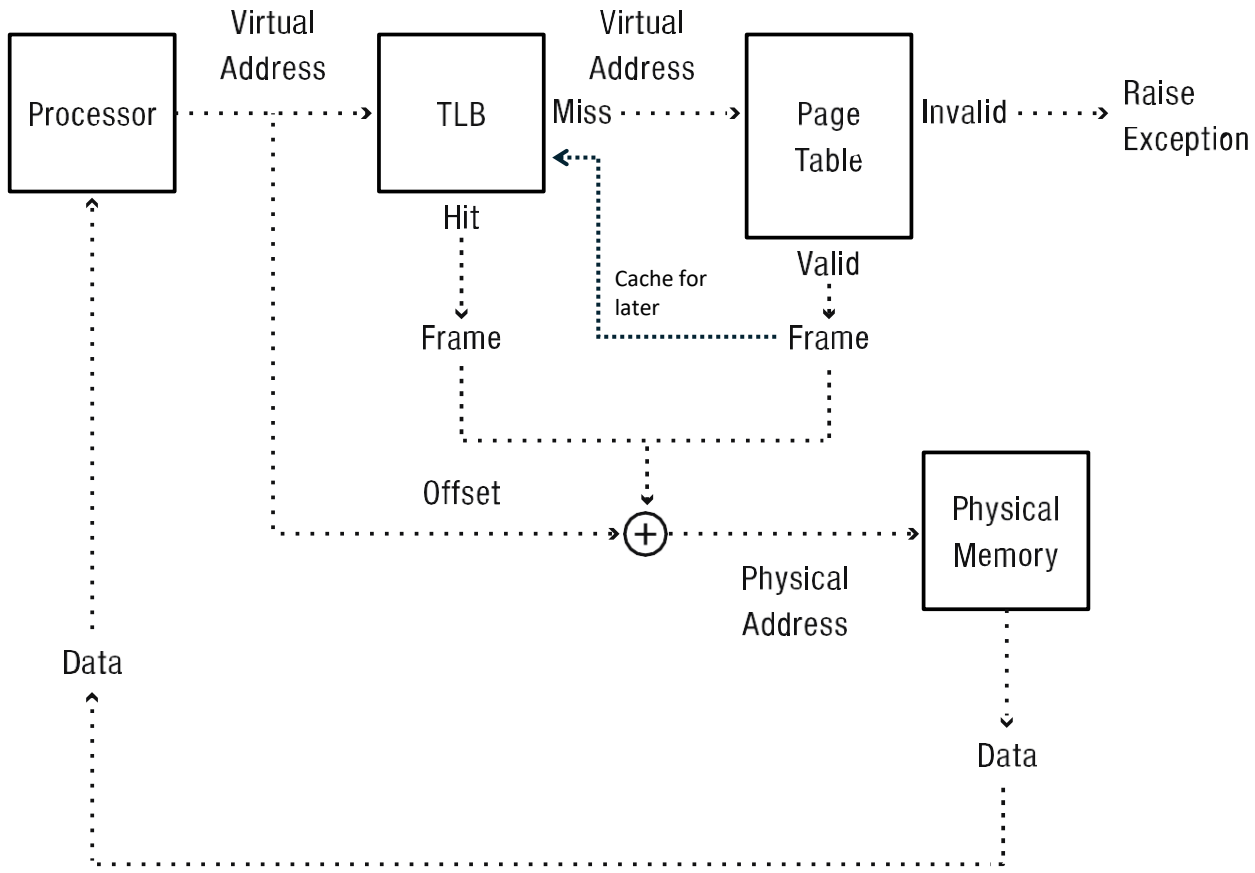
# Caching to the rescue

- 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

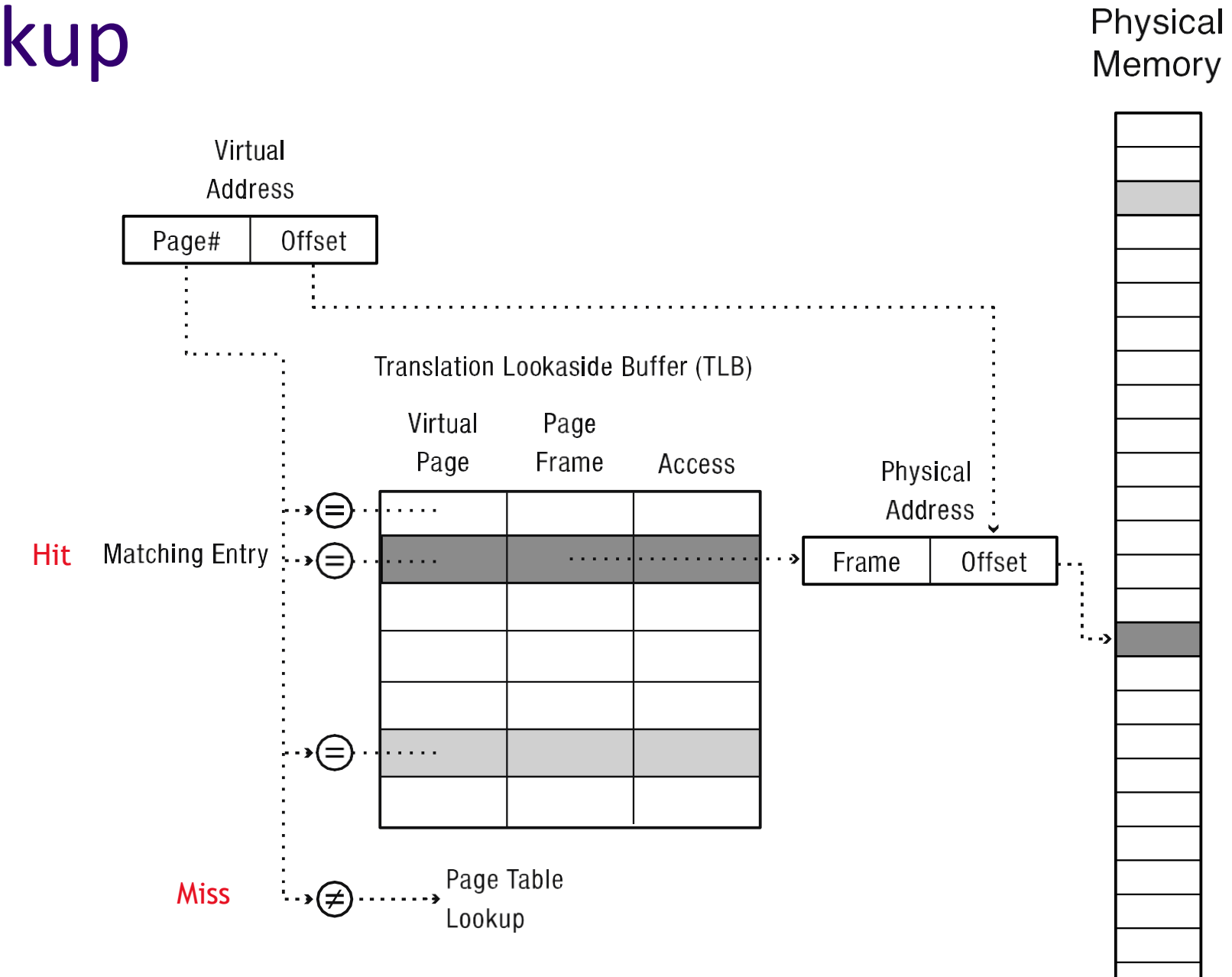
- Cost of translation =
  Cost of TLB lookup + Prob(TLB miss) * cost of page table lookup

- 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

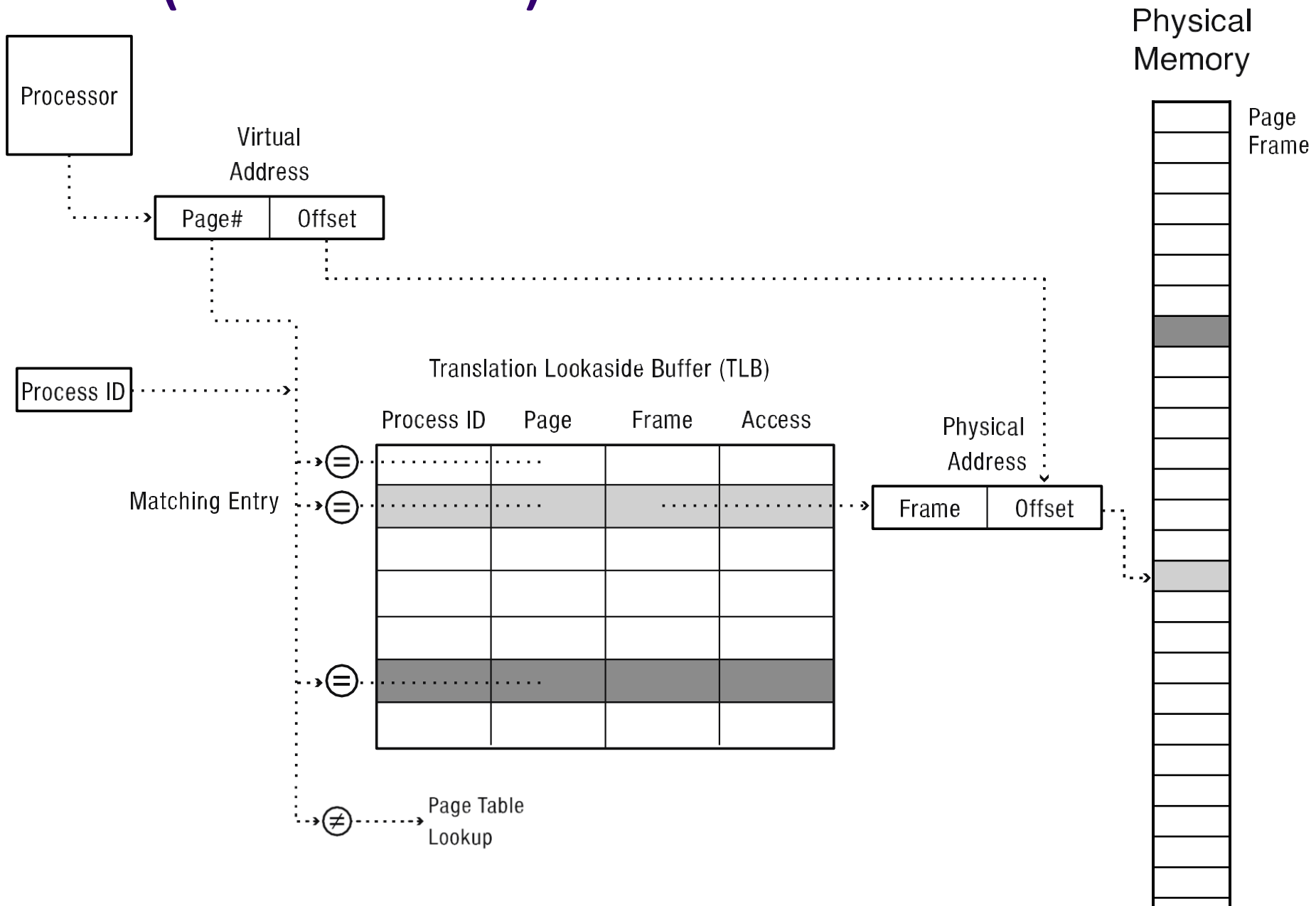
- Can we reuse TLB entries across processes?
  - 🪄OS As Illusionist: **No!** Why?
  - 👩‍⚖️OS As Referee: **No!** Why?

- 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/

himanshu@blog:~$

Home About Show more ▾

## I Missed One TLB Shootdown and Somehow Ended Up Controlling a Page Table

2026-09-13 :: Himanshu Anand :: 29 min read (6017 words)

#security #linux #kernel #kernelctf #exploit-development #ctf #blog

### Table of Contents


- TLDR
- I had some free time and a slightly unreliable memory
- Finding a finish line that could argue back

# TLB miss penalty

- The cost of a standard TLB miss:
  - On x86-64: 4-level page table walk
  - Page table is just regular old bytes in RAM, so PTE reads are just as unpredictable and possibly slow as any other dereference
    - ~2ns (L1 cache)
    - ~20ns (L3 cache)
    - ~80ns (main memory)
  - So, worst case: **~3µs** (just to translate the address!)

- One mitigation: **multi-level TLB**
  - A tiny fast TLB
  - On miss: a larger, slower TLB
  - On second miss: full page table walk
- Hardware Design Principle: The bigger the memory, the slower the memory

# Uh oh, a tradeoff

**Larger memory**



**Faster memory**

# Optimization: Superpages

# Superpages

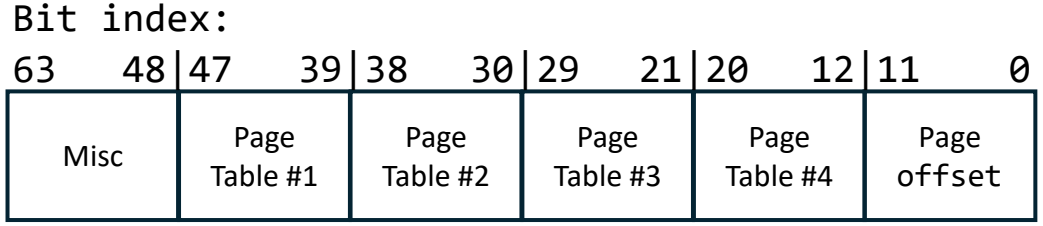
- Goal: allow a mixture of normal-size (4 KB) pages and bigger (2 MB, 1 GB) pages

- With superpages, a TLB with 64 entries can map addresses for up to:
  - `64 * 4KB = 256 KB`
  - `64 * 2MB = 128 MB`
  - `64 * 1GB =  64 GB`

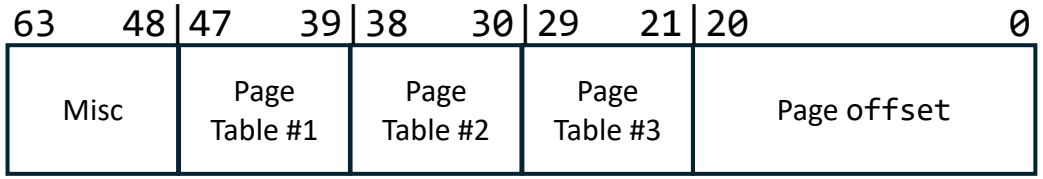
# Superpages

- Implementation:
  - Some page table entries are marked as “super pages”
  - Indicate that they point to a physical frame rather than another level of page table
  - Rest of the address should be interpreted as a larger page offset (needed to index into a larger number of frame bytes)
- Consequences of implementation:
  - Superpage size is dictated by the number of page table entries they’re “skipping”. Eg:
    - skipping table #4 -> (4kb pages * 512 PTEs) -> 2 MB
    - skipping table #3 -> (4kb pages * 512 PTEs * 512 PTEs) -> 1 GB
  - Virtual address of superpage must be aligned to a boundary of the superpage size

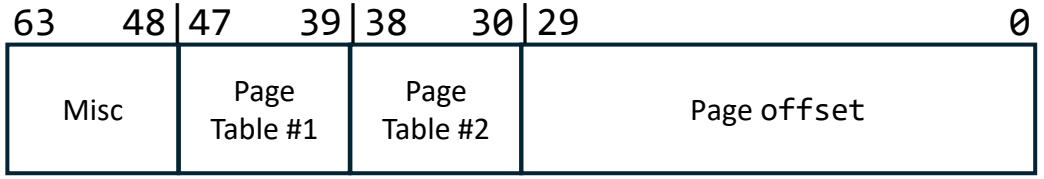
Normal virtual address



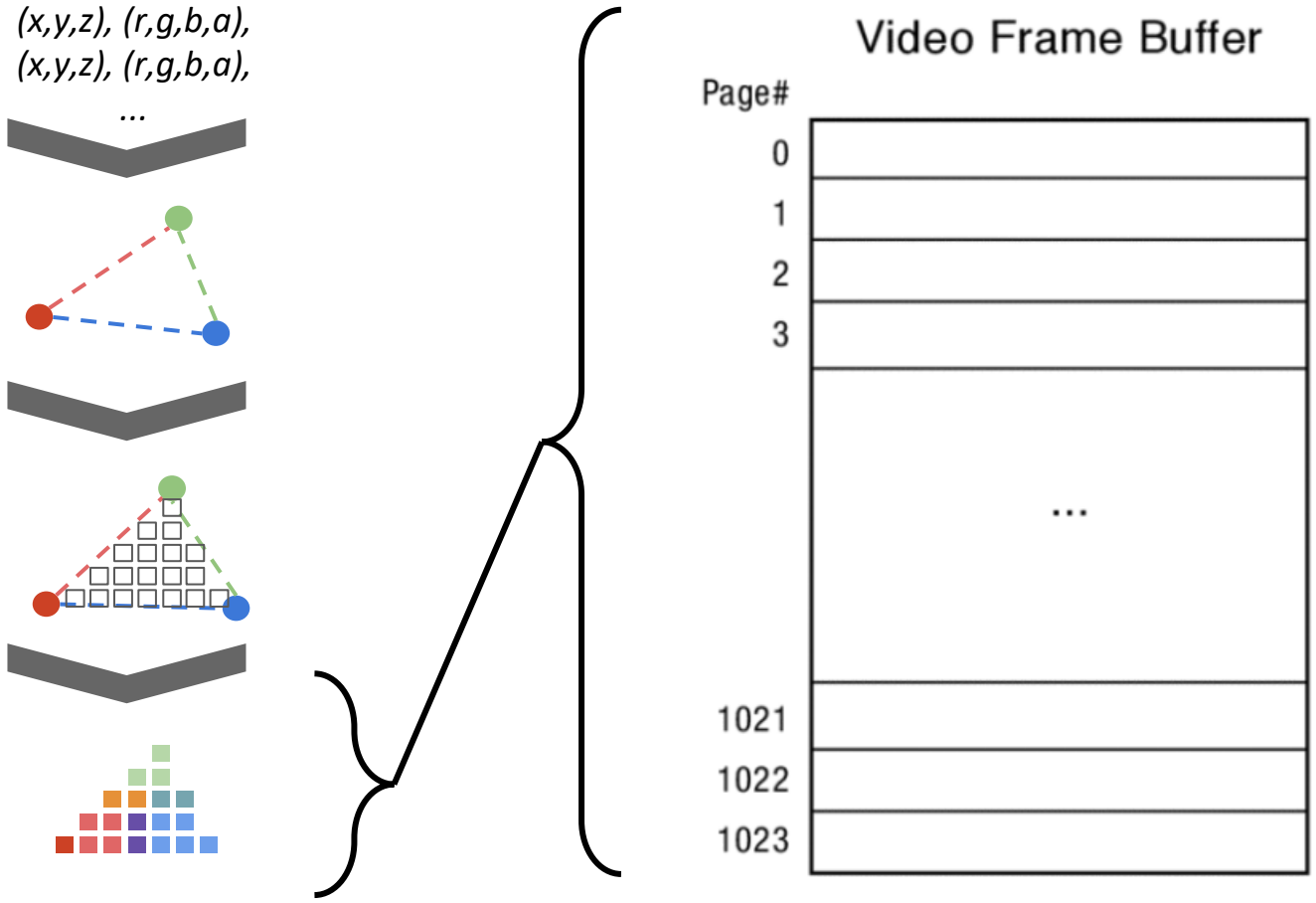

2 MB superpage virtual address




1 GB superpage virtual address



# Superpage use case

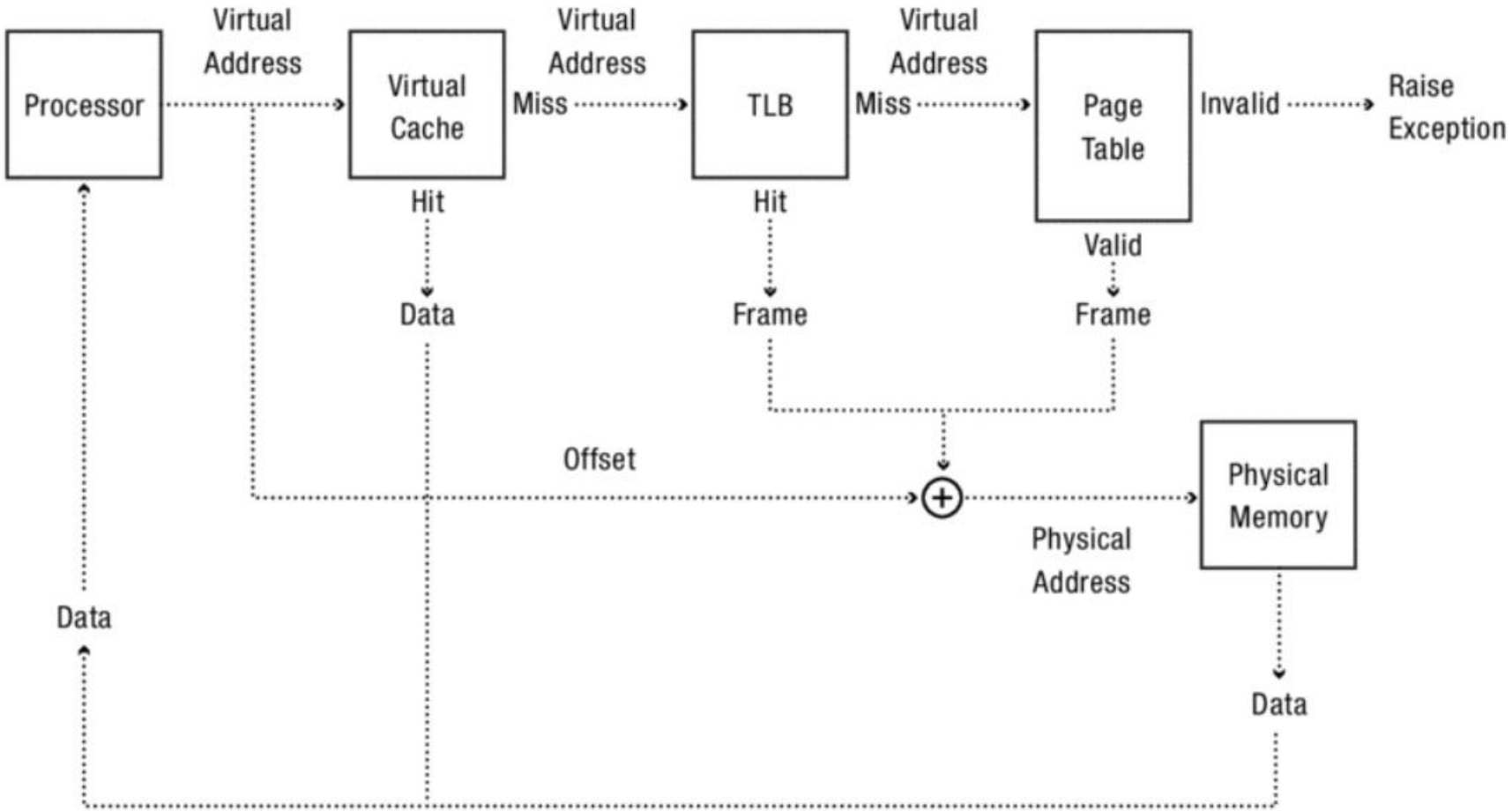


# Optimization: Virtual memory compatible caching

# Cache addressing

- First level cache's performance is critical
  - Any delay slows **every** instruction fetch and data reference!
- Naïve implementation: look up TLB to get physical address, then lookup physical address in the cache
  - Ha, no way! Too slow!
- Mitigation: use virtual addresses to do cache lookups
  - A few different ways to do this

# Virtually addressed cache



# Virtual *and* physical caches

