

CSE 451: Operating Systems, 26au

L16: Virtual Memory Optimizations

Instructor: Naomi Alterman

TAs:

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

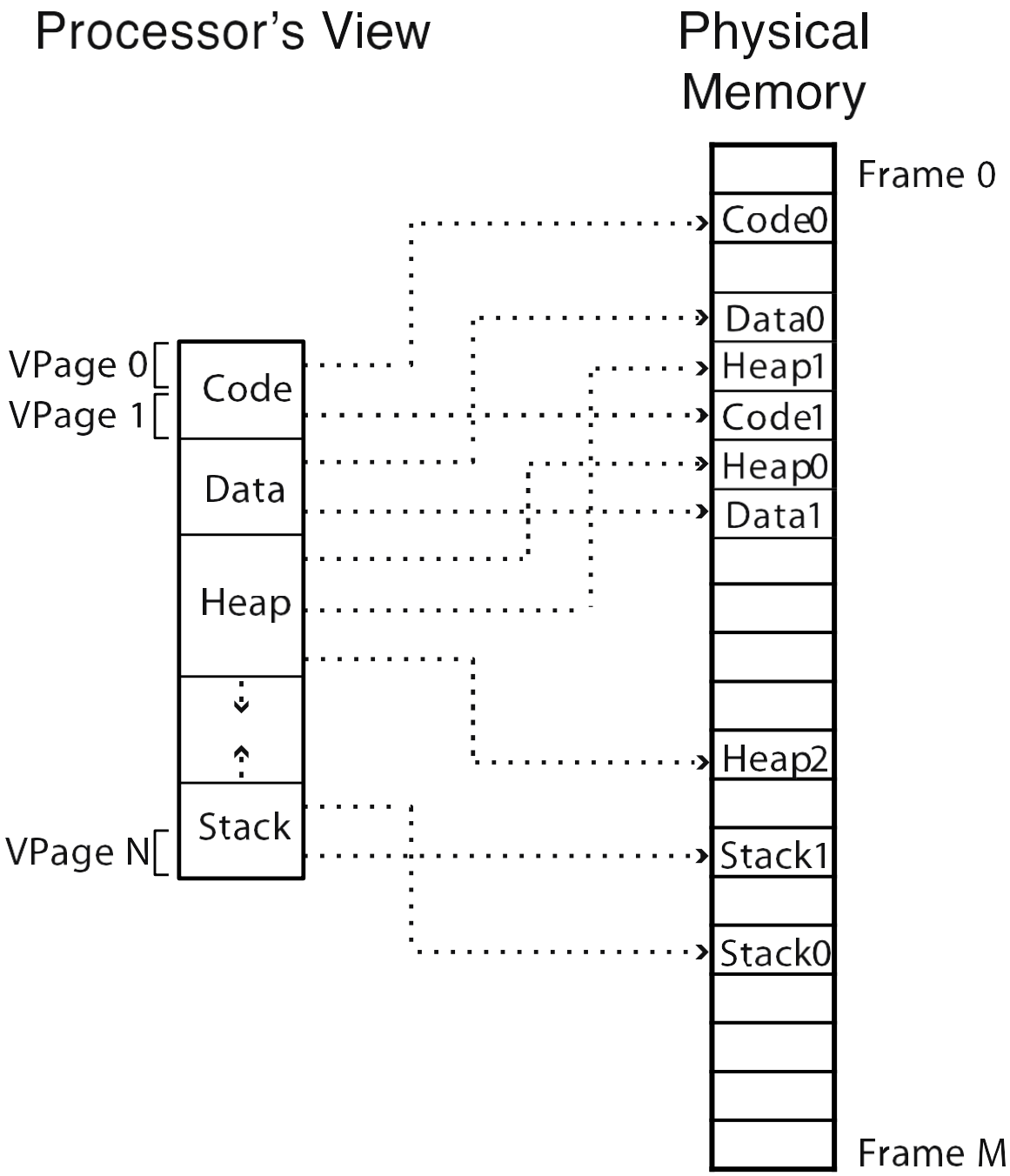
Adapted from slides by Tom Anderson

Administriviatta

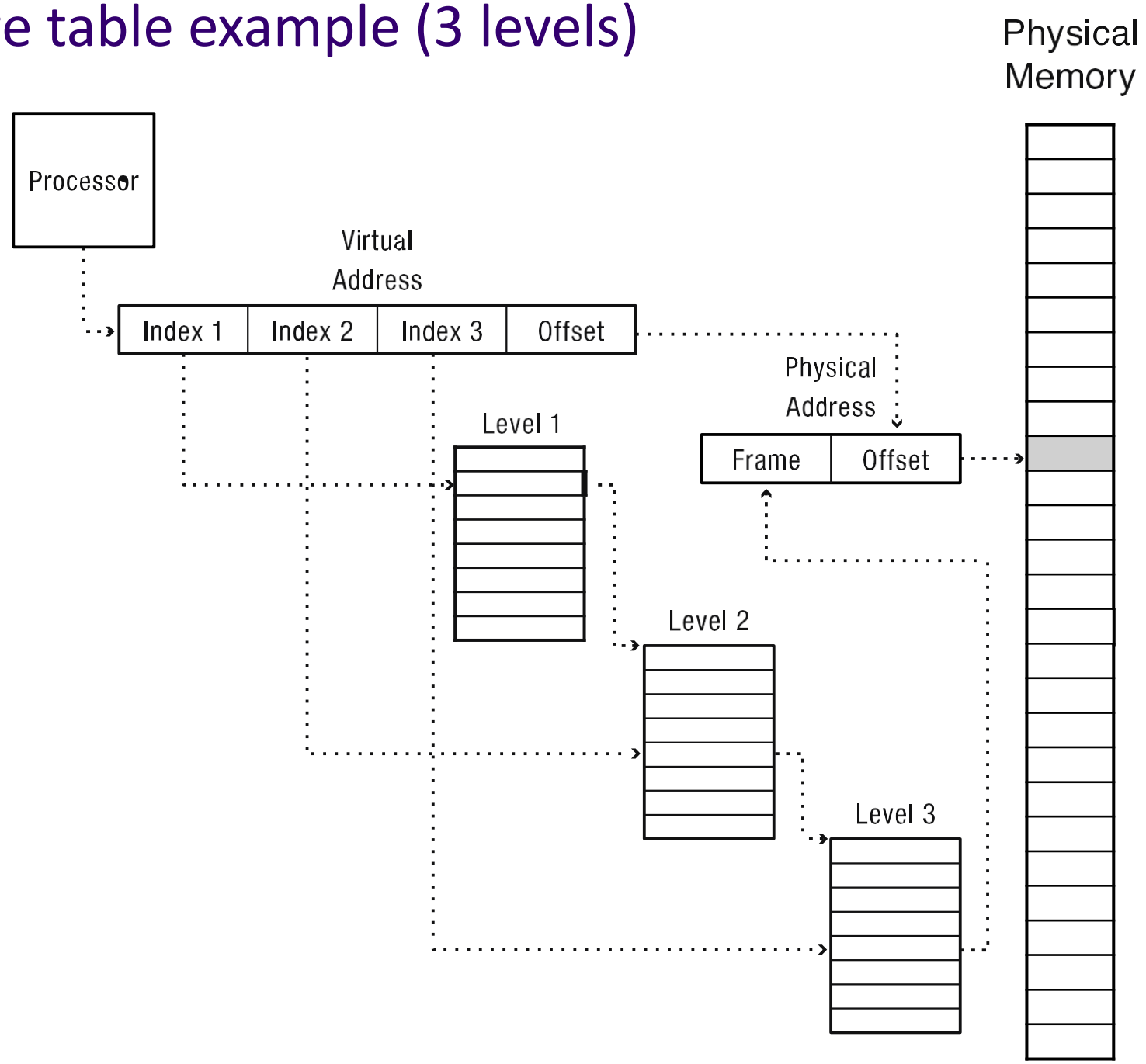
- Pset 3 out today
  - Due Friday 11/20
  - On scheduling and virtual memory (still some material left to cover)

- Lab 3 design docs due tomorrow at midnight
  - + lab 2 pt 2 DD revision

Where were we?


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

Multi-level page table example (3 levels)



The paging game

Can you translate the virtual address to find the treasure?

https://courses.cs.washington.edu/courses/cse451/26au/lectures/paging-game.html



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 soon ;) ;)

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

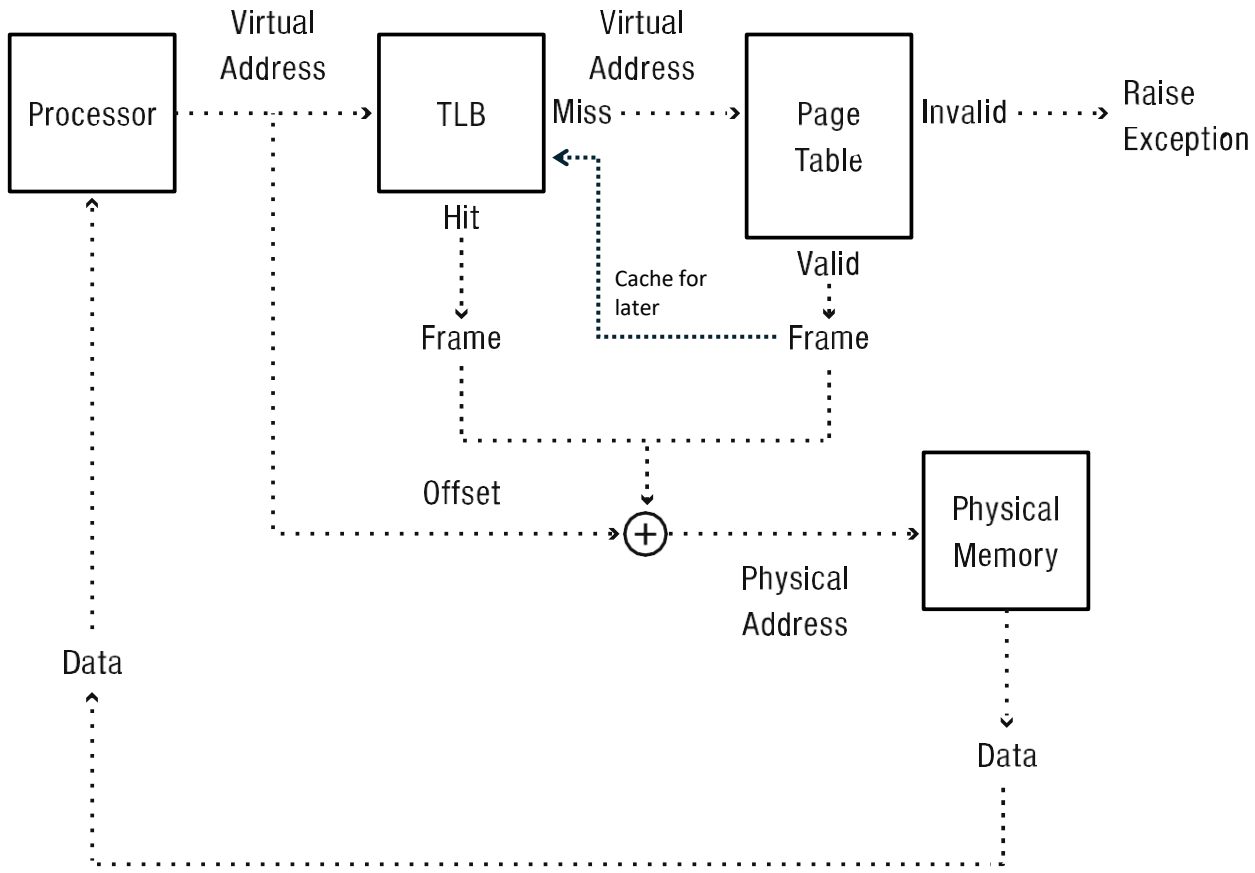
Caching to the rescue

- 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

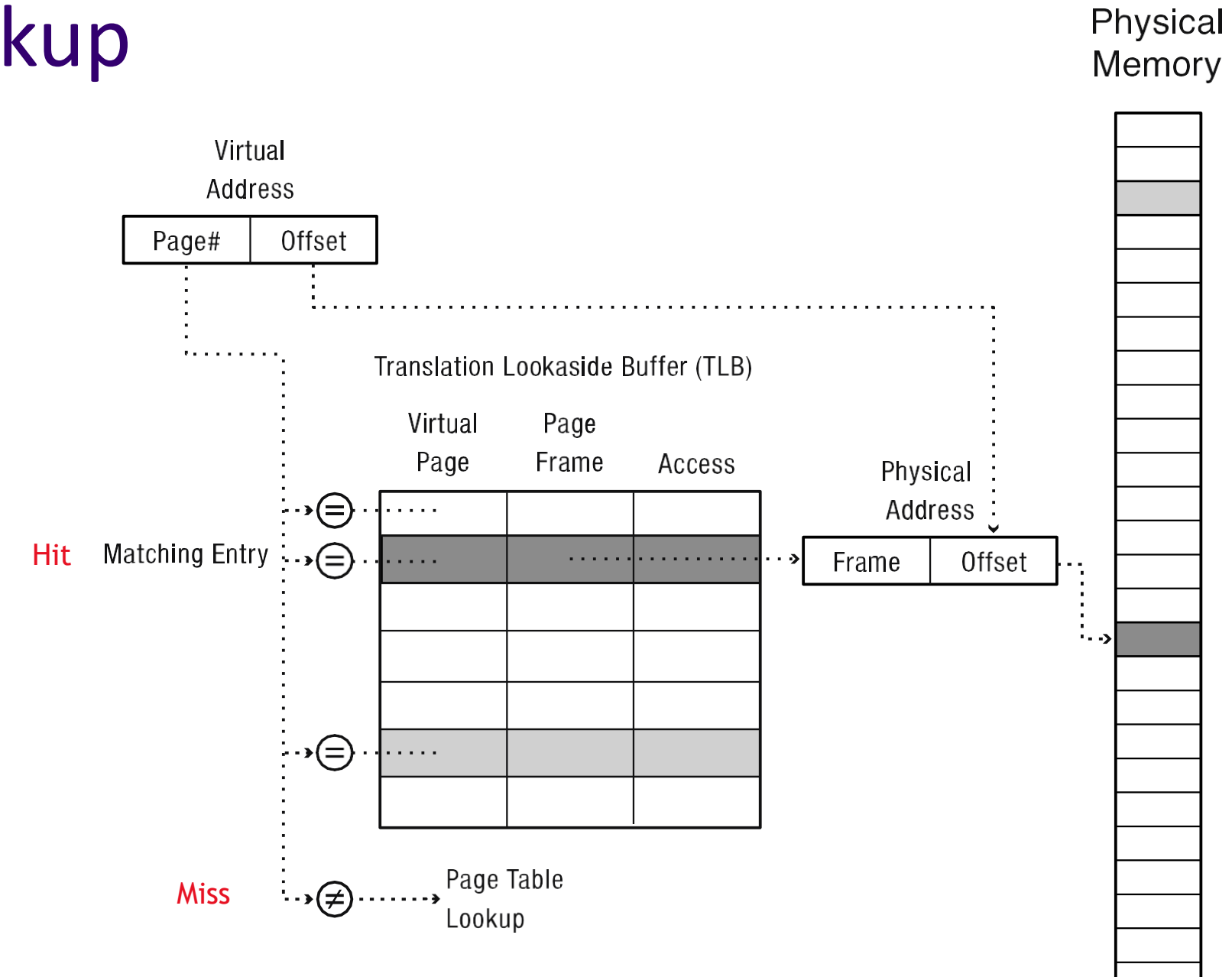
- Cost of translation =
  Cost of TLB lookup + Prob(TLB miss) * cost of page table lookup

- 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

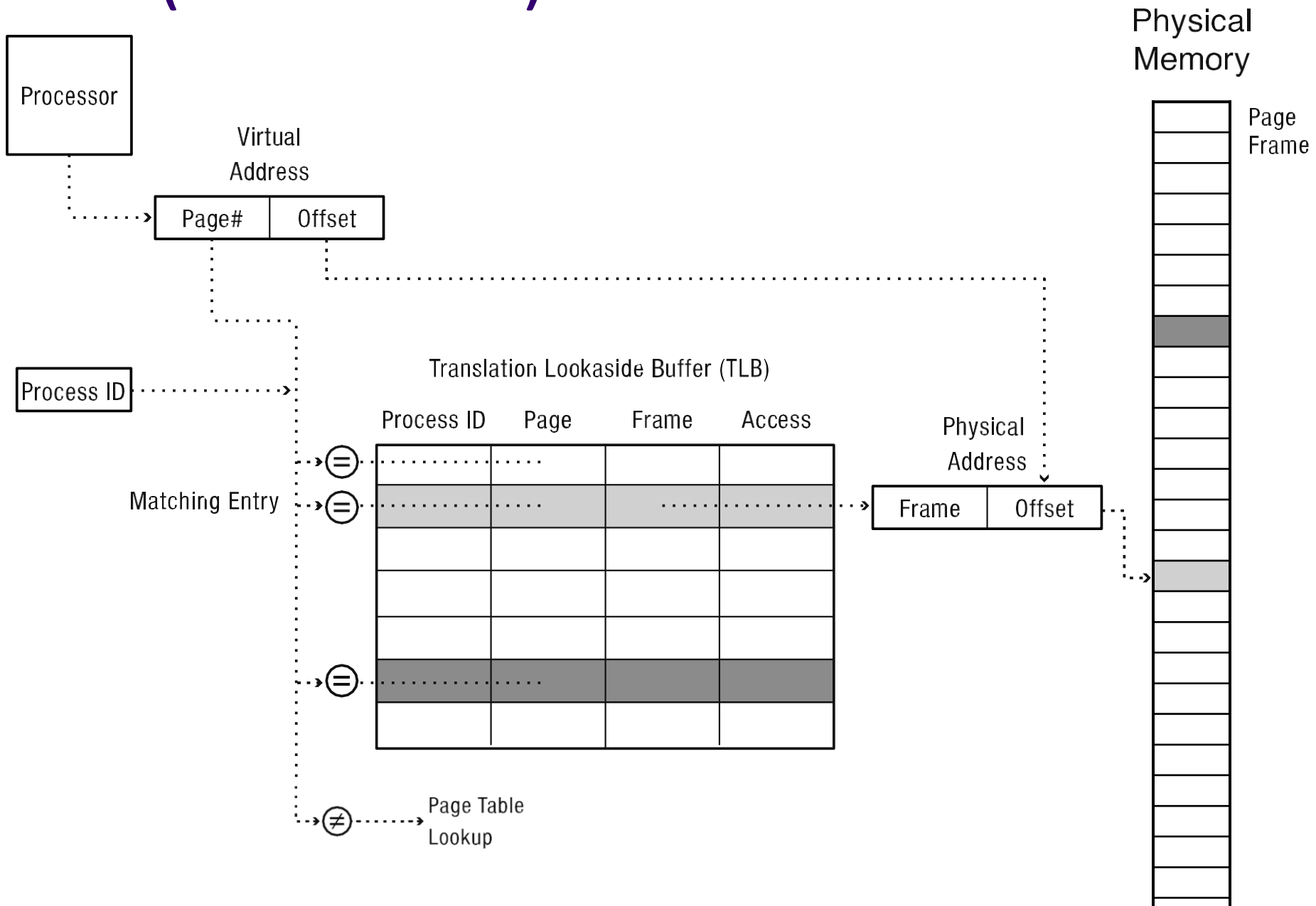
- Can we reuse TLB entries across processes?
  - 🪄OS As Illusionist: **No!** Why?
  - 👩‍⚖️OS As Referee: **No!** Why?

- 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 given 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

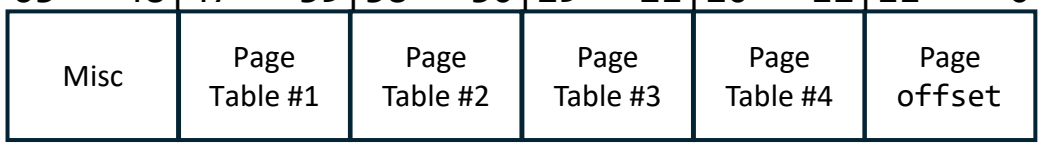
- Goal: allow a mixture of normal-size (4 KB) pages and bigger (2 MB, 1 GB) pages

- 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`

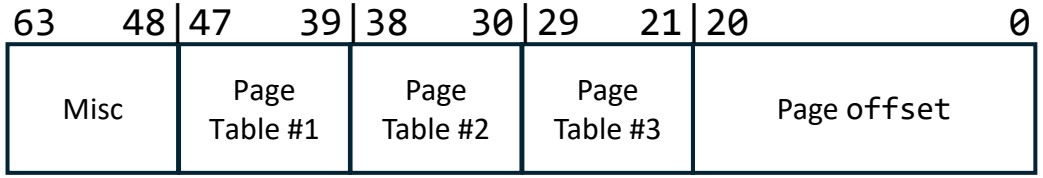
Superpages

- 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

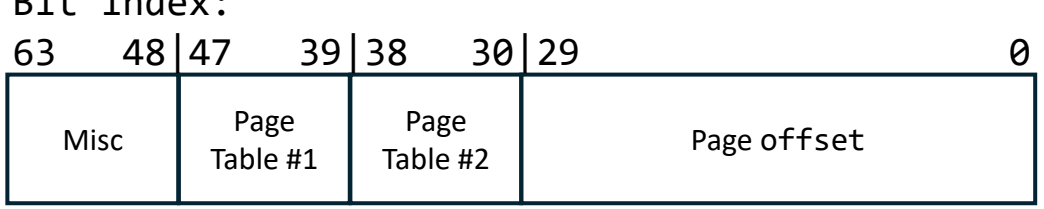
Normal virtual address



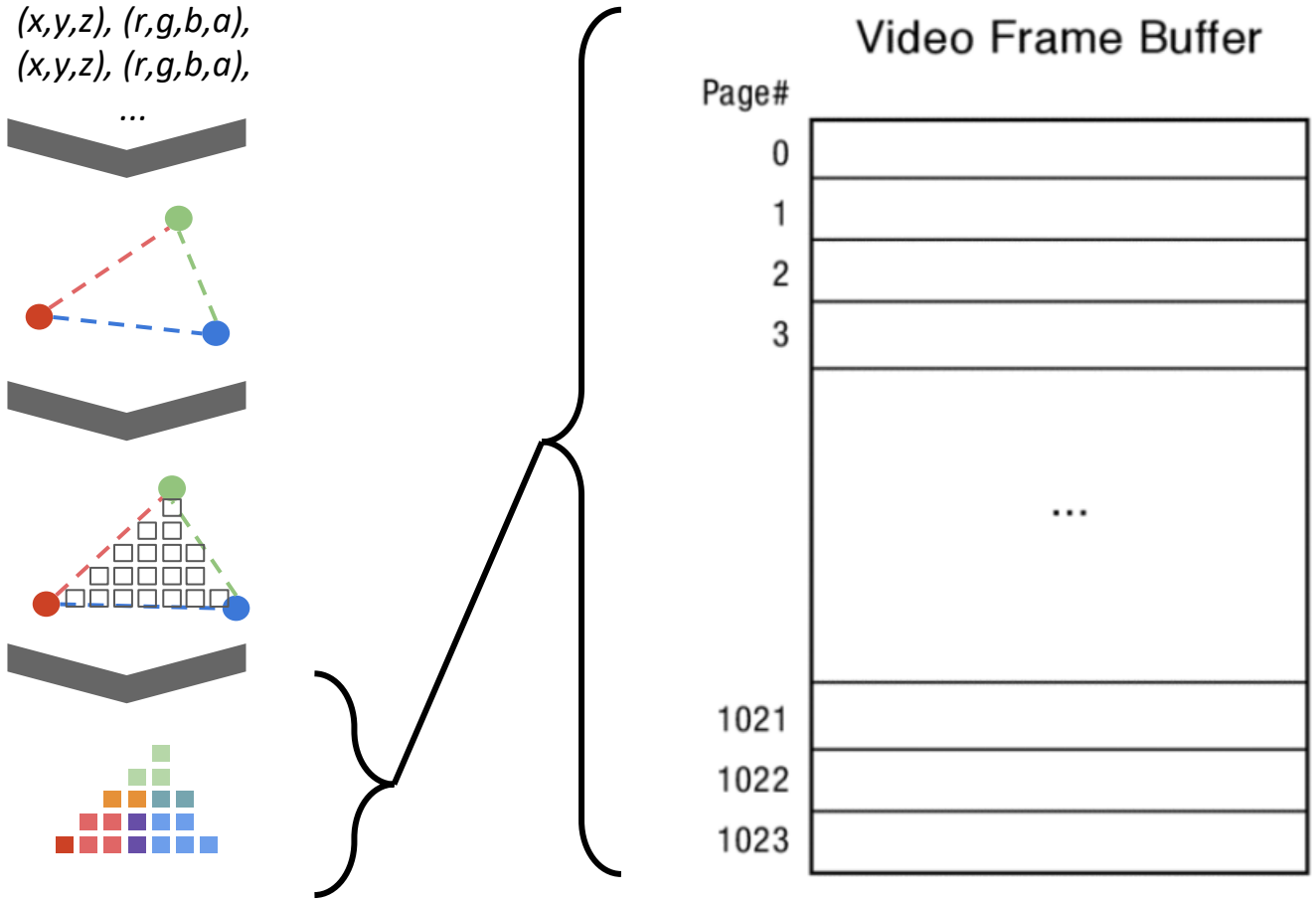
2 MB superpage virtual address



1 GB superpage virtual address



Superpage use case

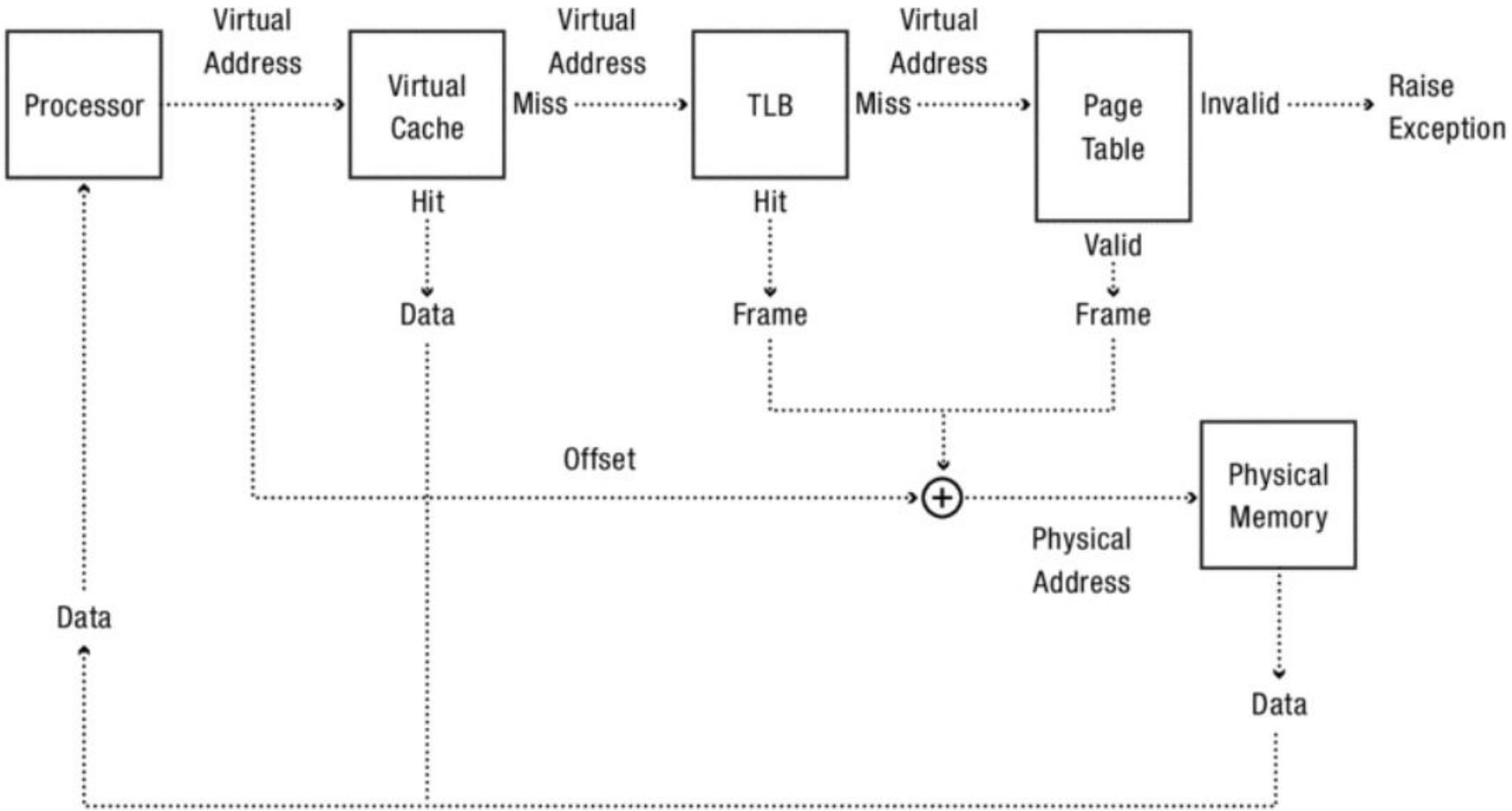


Optimization: Virtual memory compatible caching

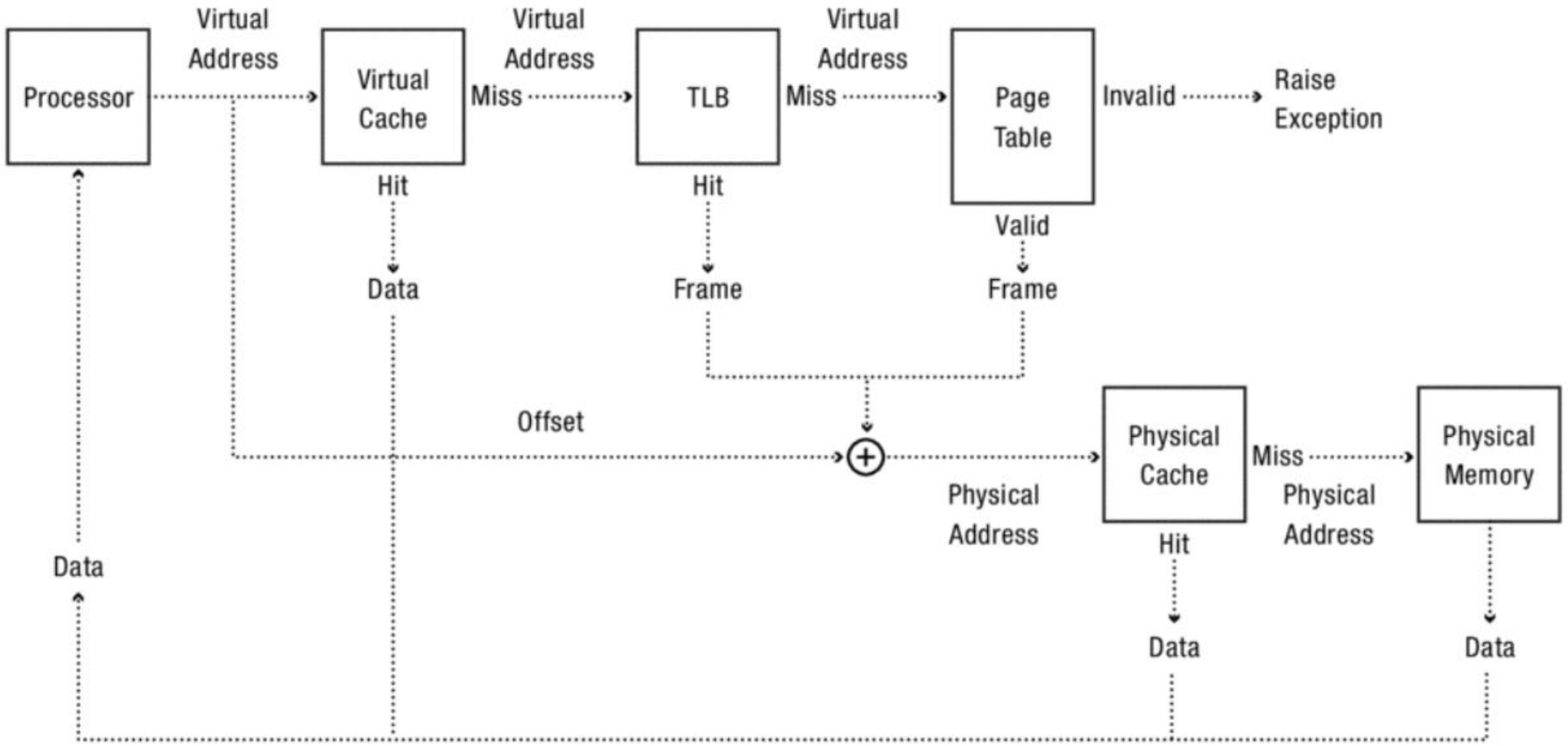
Cache addressing

- 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
  - But if we're not careful this can get *really complicated*

Virtually addressed cache



Virtual *and* physical caches



Virtually ***indexed***, physically ***tagged*** cache

- Solution for *fast* and *coherent* (but small ☹) caches
  - Position in cache (the “index”) is composed entirely of bits from the page offset
  - Each cache line stores the physical address as a piece of metadata (the “tag”)
  - On a cache hit, we check the physical address against the stored tag and only consider it a true hit if they match
- The idea:
  - While we’re doing our cache lookup, we’ll also do the TLB lookup and have a physical address by the time we have a tag to check
  - Only works up to the size of a page

Virtually ***indexed***, physically ***tagged*** cache

