COS 318: Operating Systems

Virtual Memory Design Issues: Address Translation

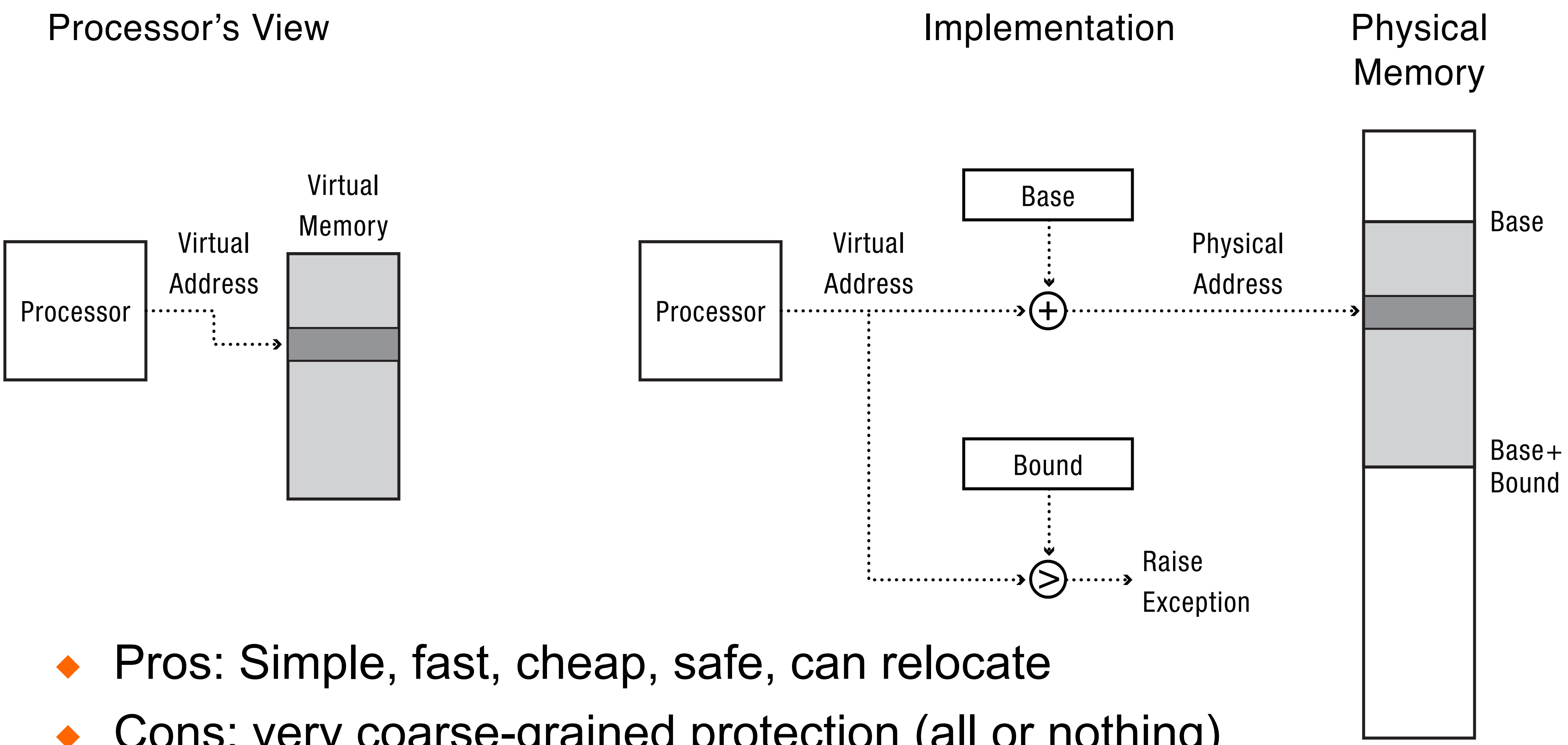
Virtual Memory Design Issues

- Protection granularity
- Enabling memory sharing (code, libraries, communication)
- Flexibility and growth/shrinking of processes
- Efficiency
  - Translation efficiency (TLB as cache)
  - Access efficiency
    - Access time = h · memory access time + (1 - h) · disk access time
    - E.g. Suppose memory access time = 100ns, disk access time = 10ms
    - If h = 90%, VM access time is 1ms!
- Process forking and copy on write

Process Forking and Copy on Write

- Idea of Copy-on-Write
  - Child process inherits copy of parent’s address space on fork
  - But don’t really want to make a copy of all data upon fork
  - Would like to share as far as possible and make own copy only “on-demand”, i.e. upon a write
- A way to do this is to protect data as read-only in both parent and child on fork
  - When a write is done by either, a protection fault occurs and a copy is made

Recall Address translation: Base and Bound



- Pros: Simple, fast, cheap, safe, can relocate
- Cons: very coarse-grained protection (all or nothing)
  - Can't keep program from accidentally overwriting its own code
  - Can't share subsets of code/data with other processes (all or nothing)
  - Can't grow stack/heap as needed (stop program, change reg, …)

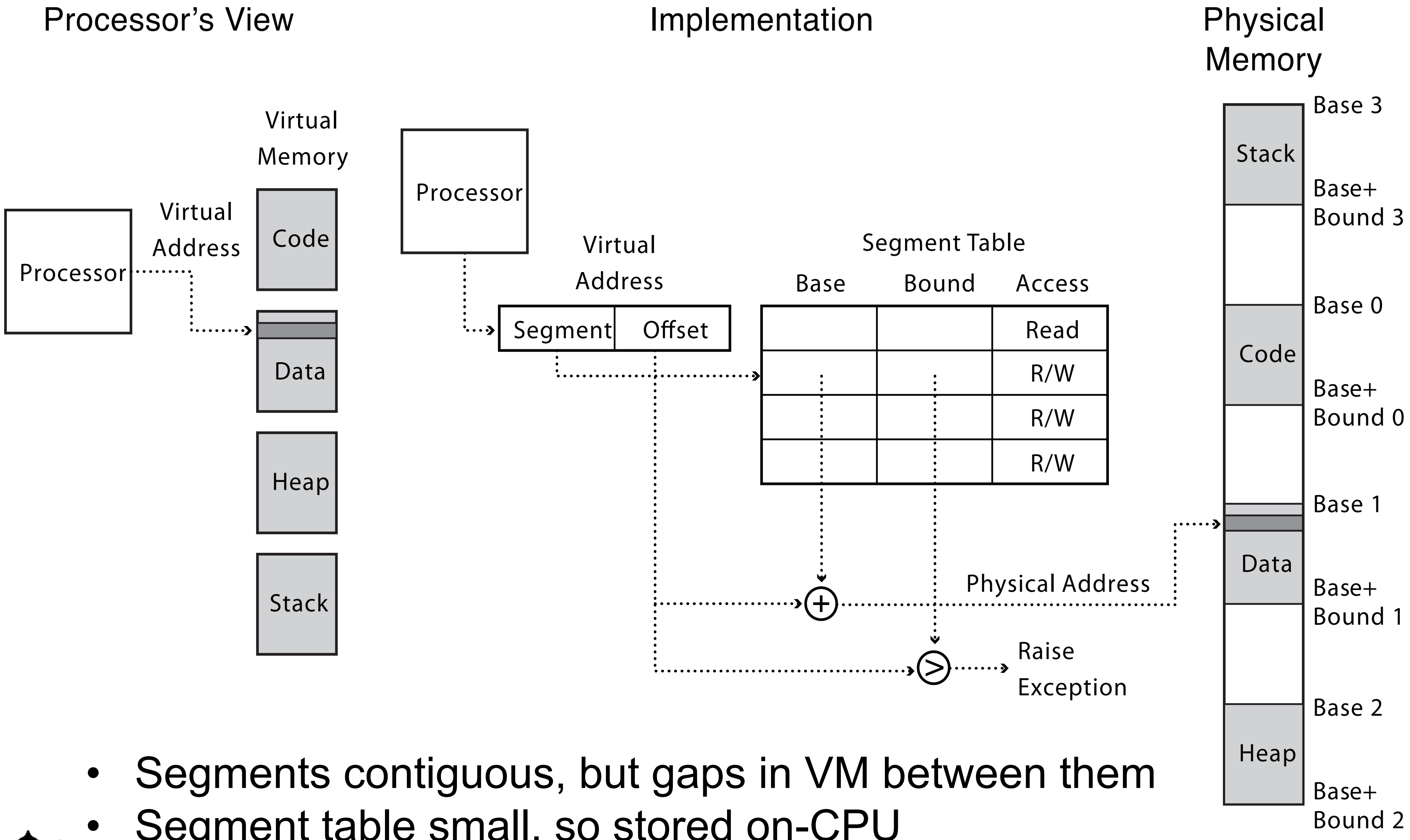
Base and Bound

- Protection granularity: Entire process space (code+data)
  - Can’t keep program from accidentally overwriting its own code
- Sharing
  - Can’t share subsets of code/data with other processes (all or nothing)
- Growth/shrinking of processes
  - Can’t grow stack/heap as needed (stop program, change reg, …)
- Efficiency
  - Translation: fast (simple and cheap)
  - Access
    - External fragmentation leads to inefficient use of physical memory and hence high miss rates
- Process forking and copy on write
  - Protection granularity is entire process space: no benefit from copy on write

Segmentation

- A segment is a contiguous region of *virtual* memory
- Every process has a segment table (in hardware)
  - Entry in table per segment
- Segment can be located anywhere in physical memory
  - Each segment has: start, length, access permission
- Protection is at granularity of segments

Segmentation



Segment Table

Base	Bound	Access
		Read
		R/W
		R/W
		R/W

- Segments contiguous, but gaps in VM between them
- Segment table small, so stored on-CPU
- Access control on per-segment basis

Segmentation

- Protection granularity: A (user-defined) segment
  - Protects code separately from data
- Sharing
  - Processes can share segments: Same start, length, same/different access permissions
- Growth/shrinking of processes
  - Can grow segments independently, may need to relocate
- Efficiency
  - Translation: fast (few segments so table can be in hardware)
  - Access
    - Better than base+bound, but still external fragmentation due to holes
- Process forking and copy on write
  - Can do on a segment granularity: copy entire segment on first write to it

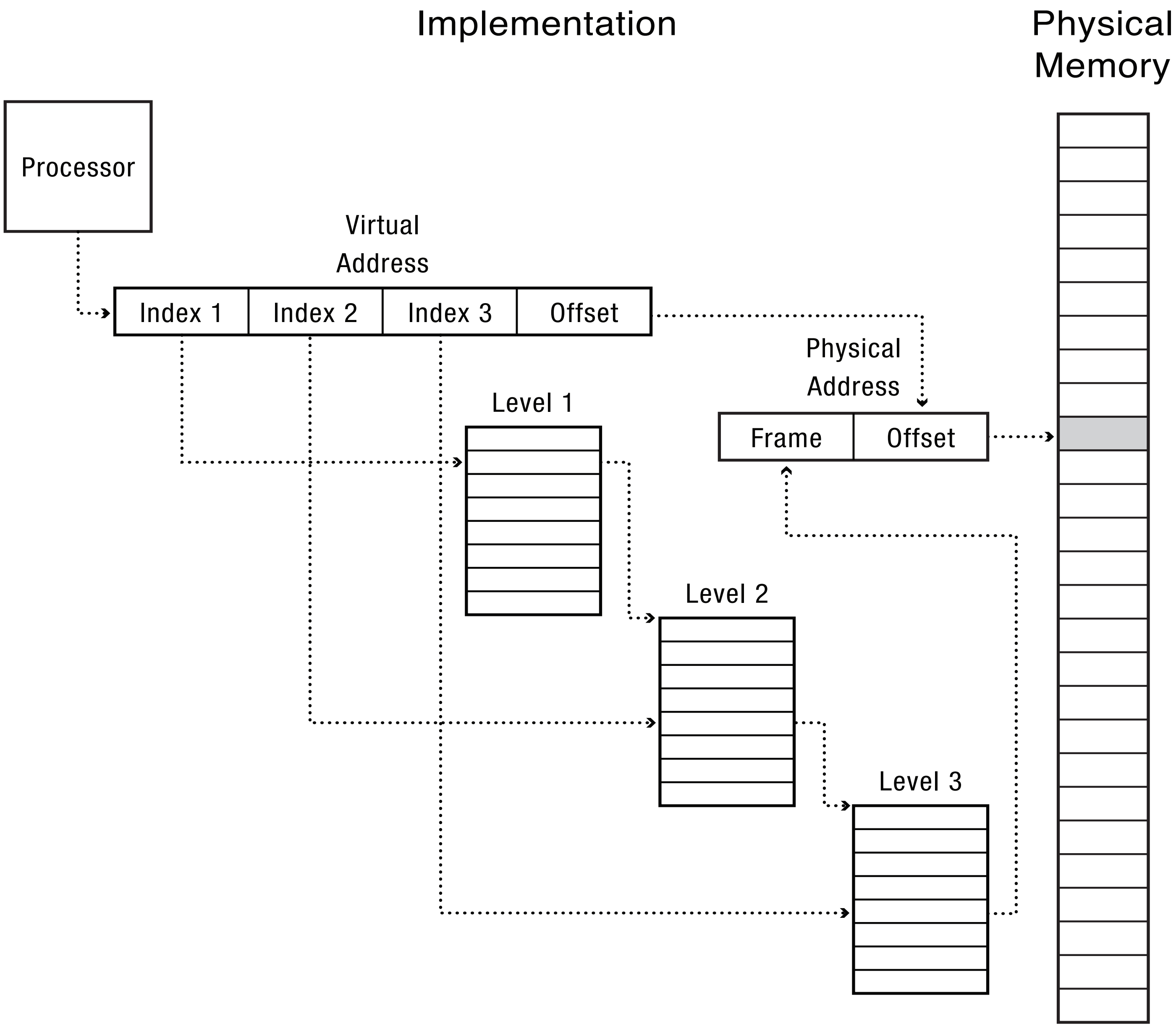
Segments Enable Copy-on-Write

- To an extent …
  - Copy segment table into child, not entire address space
  - Mark all parent and child segments read-only
  - Start child process; return to parent
  - If child or parent writes to a segment (e.g. stack, heap)
    - Trap into kernel
    - At this point, make a copy of th
- But segmentation has other problems too:
  - Complex memory management due to external fragmentation
    - Need to find chunk of particular size
    - Wasted space between chunks/segments
    - May need to rearrange memory from time to time to make room for new segment or to grow segment

Paging

- Manage memory in fixed size units, or pages
- Finding a free page is easy
  - Effectively a bitmap allocation: 0011111100000001100
  - Every bit represents one physical page frame
- Every process has its own page table
  - Stored in physical memory
  - Supported by a couple of hardware registers:
    - Pointer to start of page table
    - Page table length
- Recall fancier structures: segmentation+paging, multi-level PT
  - Better for sparse virtual address spaces
  - E.g. per-processor heaps, per-thread stacks, memory mapped files, dynamically linked libraries, …
  - Eliminate need for page table entries for address space “holes”

Multilevel Page Table

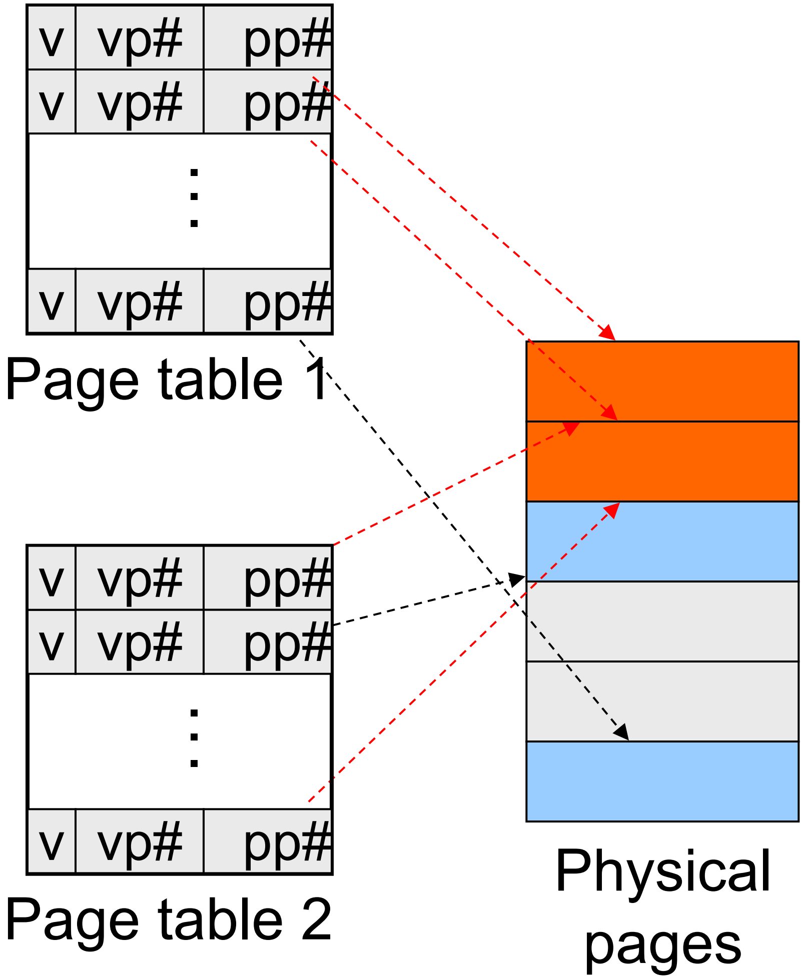


Copy on Write with Paging

- UNIX fork with copy on write
  - Copy page table of parent into child process
  - Mark all pages (in new and old page tables) as read-only
  - Trap into kernel on write (in child or parent)
  - Copy page
  - Mark both as writeable
  - Resume execution
  - Finer grained than with segments

Shared Pages

- PTEs from two processes share the same physical pages
  - Entries in both page tables to point to same page frames
  - What use cases?
- Implementation issues
  - What if you terminate a process with shared pages
  - Paging in/out shared pages
  - Deriving the working set for a process with shared pages
  - Pinning/unpinning shared pages



Pinning (or Locking) Page Frames

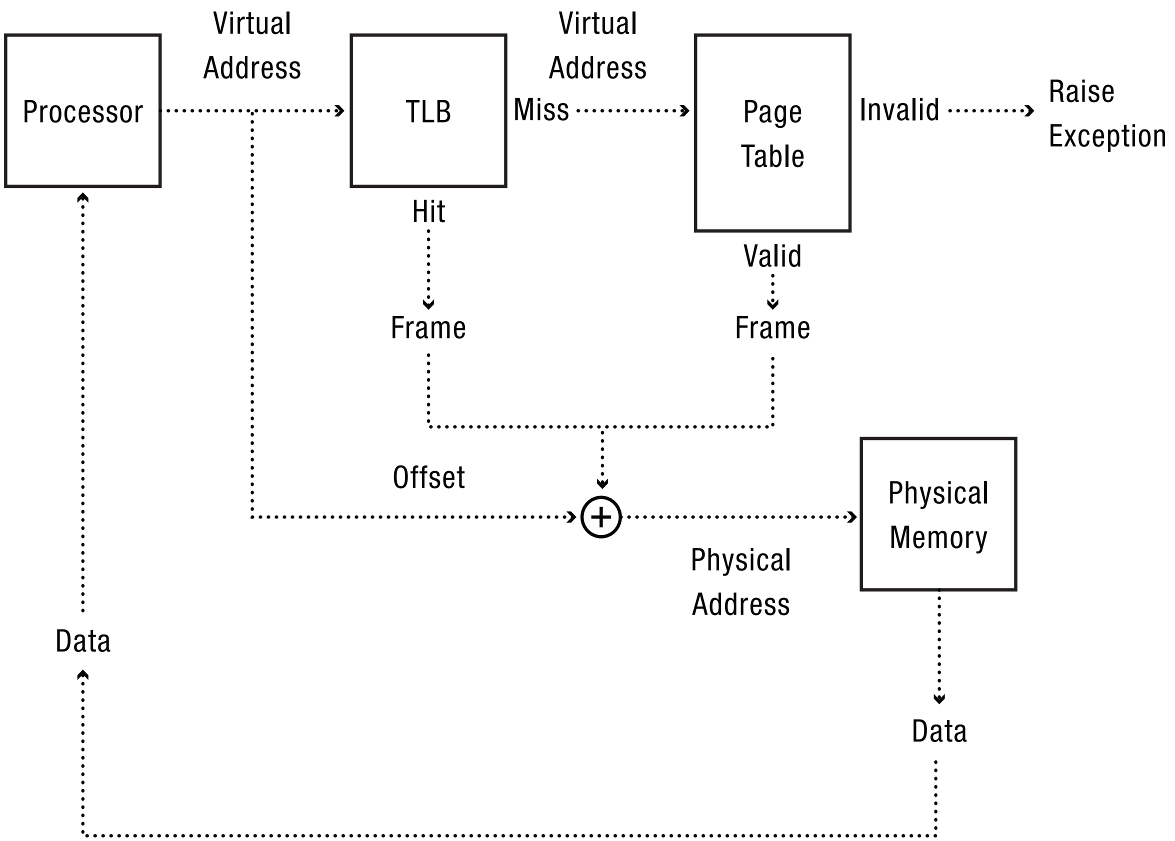
- When do you need it?
  - When DMA is in progress, you don’t want to page the pages out to avoid CPU from overwriting the pages
- Mechanism?
  - A data structure to remember all pinned pages
  - Paging algorithm checks the data structure to decide on page replacement
  - Special calls to pin and unpin certain pages

Zeroing Pages

- Initialize pages to all zero values
  - Heap and static data are initialized
- Q: State two ways to implement page zero’ing

Efficient address translation

- Recall translation lookaside buffer (TLB)
  - Cache of recent virtual page -> physical page translations
  - If cache hit, use translation
  - If cache miss, walk (perhaps multi-level) page table



TLB Performance

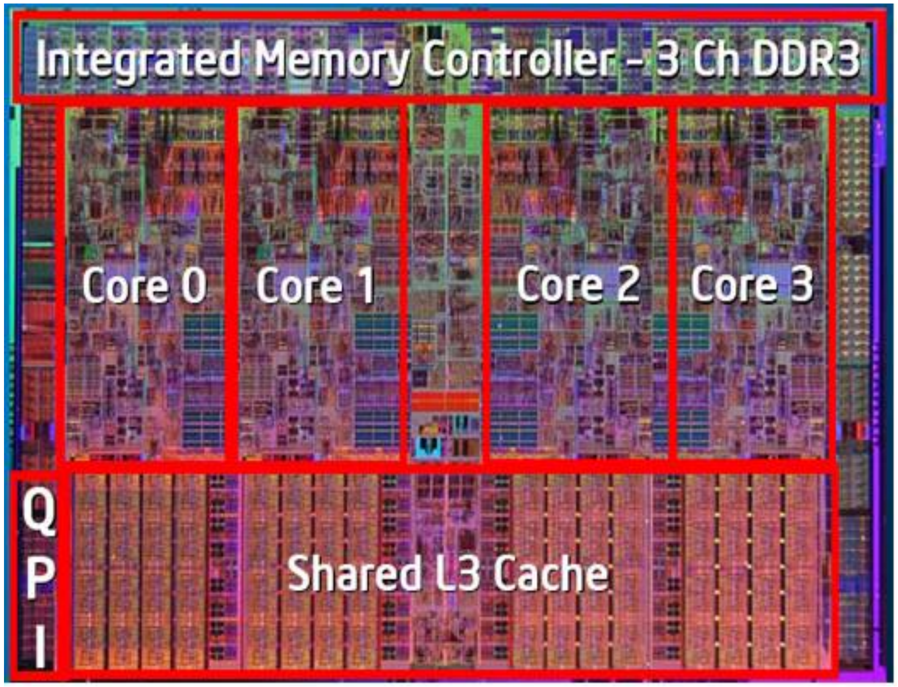
- Cost of translation =

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

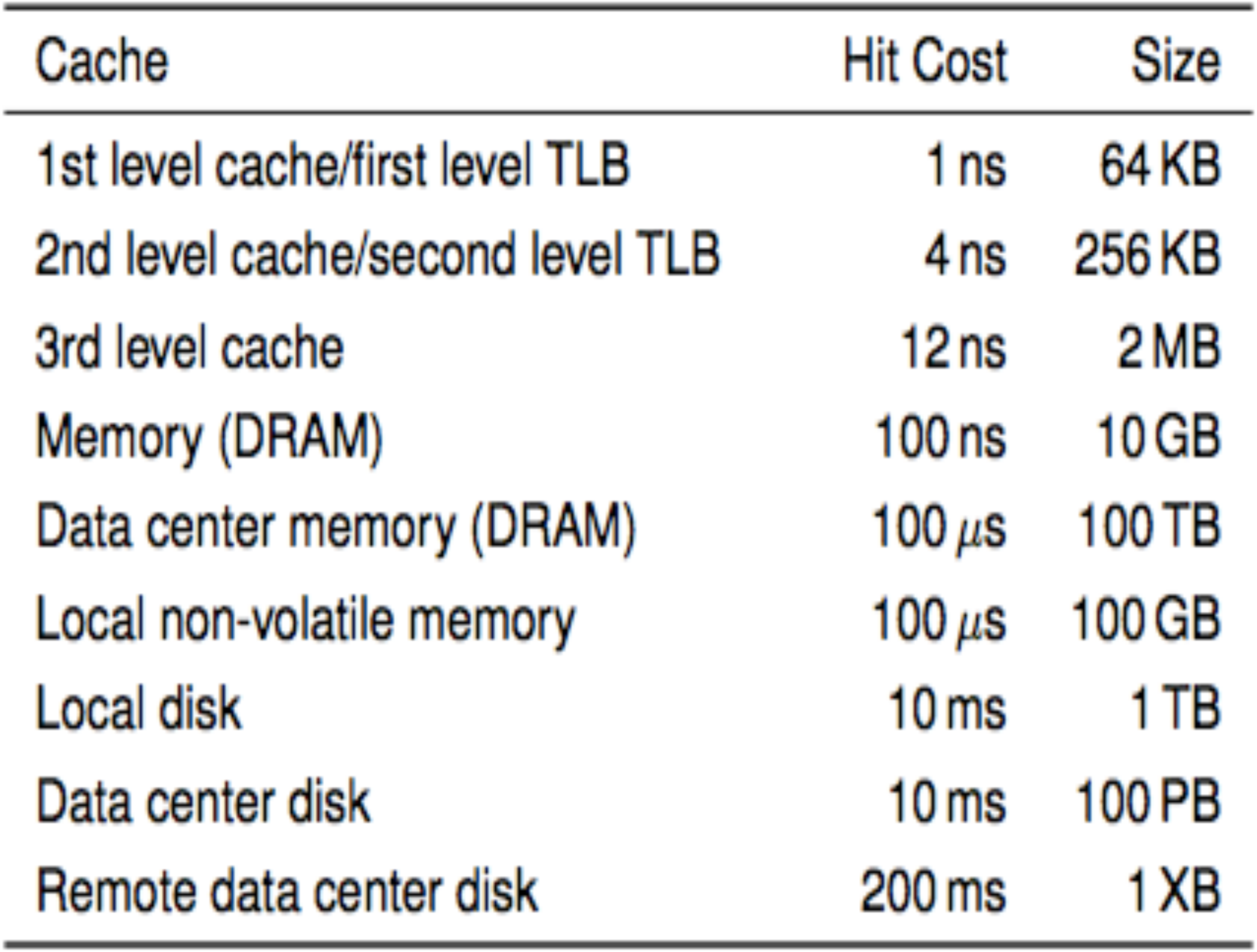
- Cost of a TLB miss on a modern processor?
  - Cost of multi-level page table walk
  - Software-controlled: plus cost of trap handler entry/exit
  - Use additional caching principles: multi-level caching, etc

TLB is important:

Intel i7 Processor Chip



Intel i7 Memory hierarchy



Cache	Hit Cost	Size
1st level cache/first level TLB	1 ns	64 KB
2nd level cache/second level TLB	4 ns	256 KB
3rd level cache	12 ns	2 MB
Memory (DRAM)	100 ns	10 GB
Data center memory (DRAM)	100 μs	100 TB
Local non-volatile memory	100 μs	100 GB
Local disk	10 ms	1 TB
Data center disk	10 ms	100 PB
Remote data center disk	200 ms	1 XB

i7 has 8MB as shared 3rd level cache; 2nd level cache is per-core

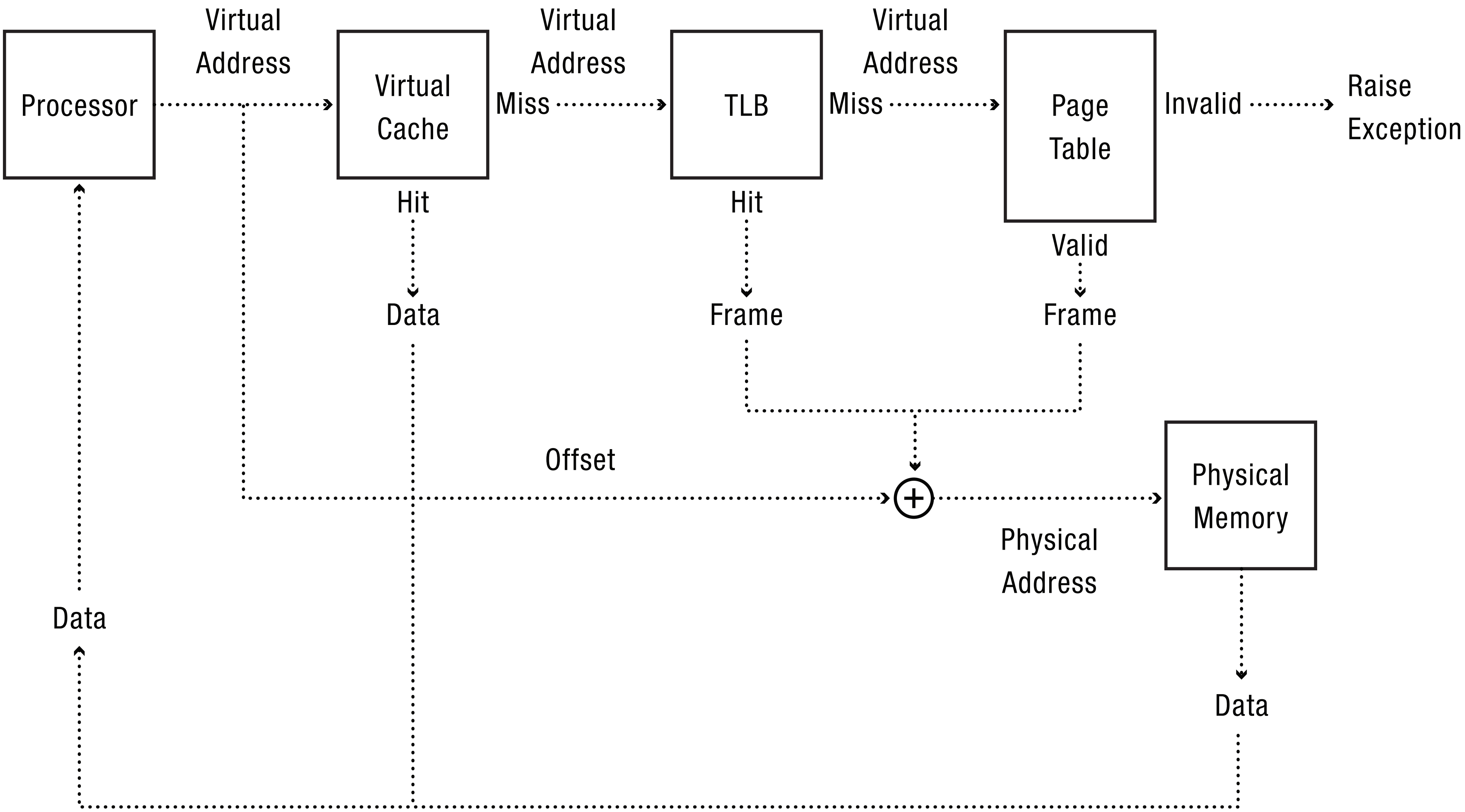
Problem with Translation Slowdown

- What is the cost of a first level TLB miss?
  - Second level TLB lookup
- What is the cost of a second level TLB miss?
  - x86: 2-4 level page table walk
- Problem: Do we need to wait for the address translation in order to look up the caches (for code and data)?

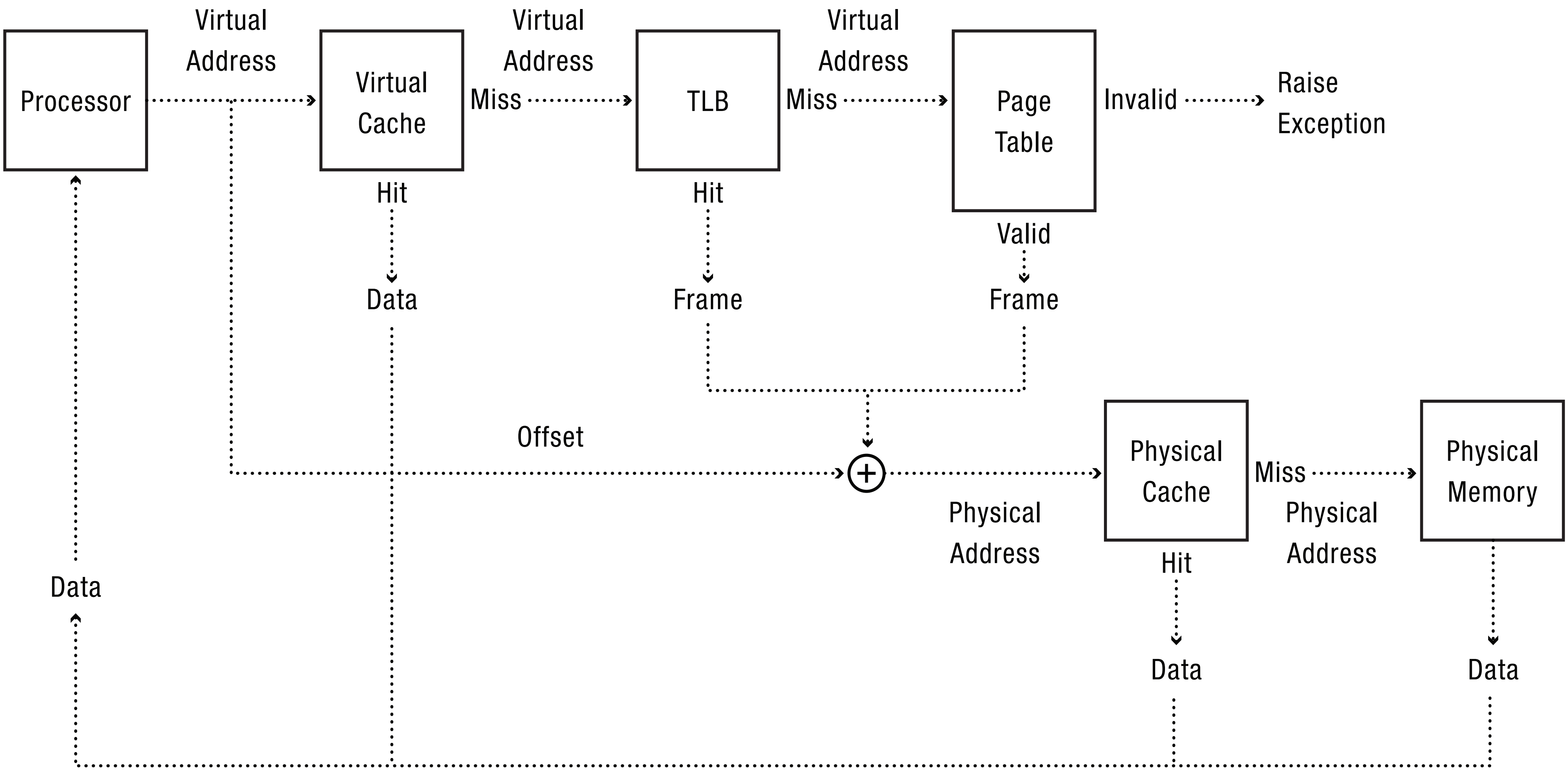
Virtually vs. Physically Addressed Caches

- It can be too slow to first access TLB to find physical address, then look up address in the cache
- Instead, first level cache is virtually addressed
- In parallel with cache lookup using virtual address, access TLB to generate physical address in case of a cache miss

Virtually addressed caches

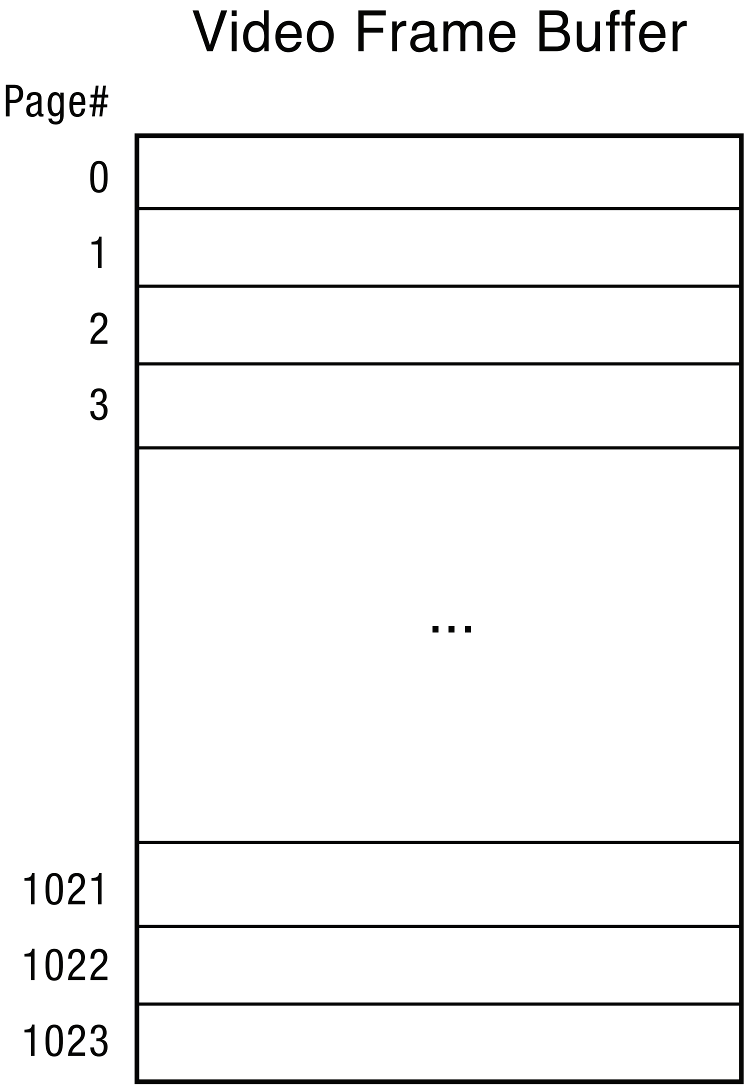


Physically addressed cache



When do TLBs work/not work?

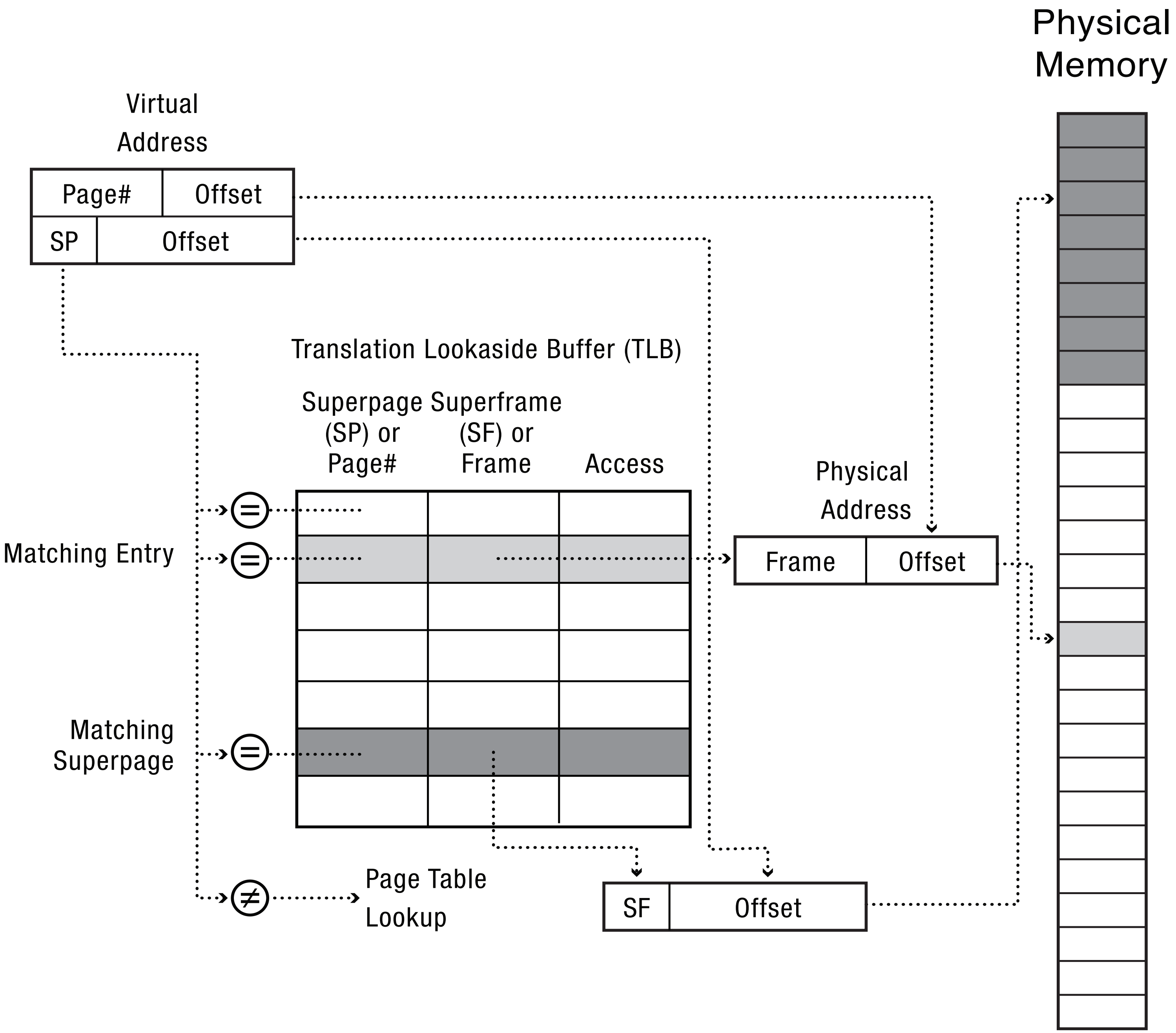
- Video Frame Buffer: 32 bits x 1K x 1K = 4MB



Superpages

- On many systems, TLB entry can be
  - A page
  - A superpage: a set of contiguous pages
- x86: superpage is a set of pages with one PTE
  - x86 TLB entries
    - 4KB
    - 2MB
    - 1GB

Superpages

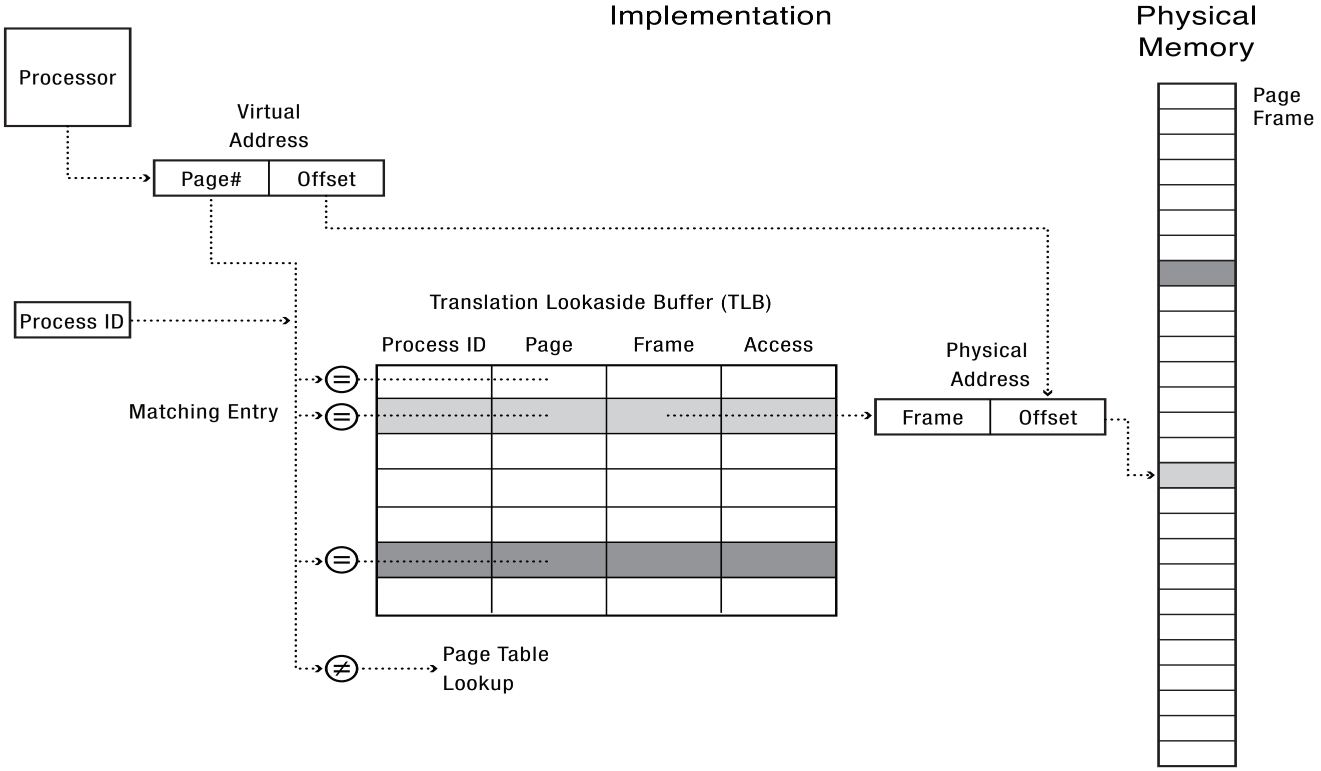

Physical Memory
Virtual Address
Page#
Offset
SP
Offset
Translation Lookaside Buffer (TLB)
Superpage (SP) or Page#
Superframe (SF) or Frame
Access
Physical Address
Frame
Offset
Matching Entry
Matching Superpage
=
=
=
≠
Page Table Lookup
SF
Offset

When do TLBs Work/Not Work, Part 2

- What happens when the OS changes the permissions on a page?
  - For demand paging, copy on write, zero on reference, …
- TLB may contain old translation
  - OS must ask hardware to purge TLB entry
- On a multicore: TLB shootdown
  - OS must ask each CPU to purge TLB entry

When do TLBs Work/Not Work, Part 3

- What happens on a context switch?
  - Keep using TLB?
  - Flush TLB?
- Solution: Tagged TLB
  - Each TLB entry has process ID
  - TLB hit only if process ID matches current process



TLB Consistency Issues

- "Snoopy" cache protocols (hardware)
  - Maintain consistency with DRAM, even when DMA happens
- Consistency between DRAM and TLBs (software)
  - You need to flush related TLBs whenever changing a page table entry in memory
- TLB "shoot-down"
  - On multiprocessors/multicore, when you modify a page table entry, need to flush all related TLB entries on all processors/cores

Summary

- Must consider many issues
  - Global and local replacement strategies
  - Management of backing store
  - Primitive operations
    - Pin/lock pages
    - Zero pages
    - Shared pages
    - Copy-on-write
- Real system designs are complex