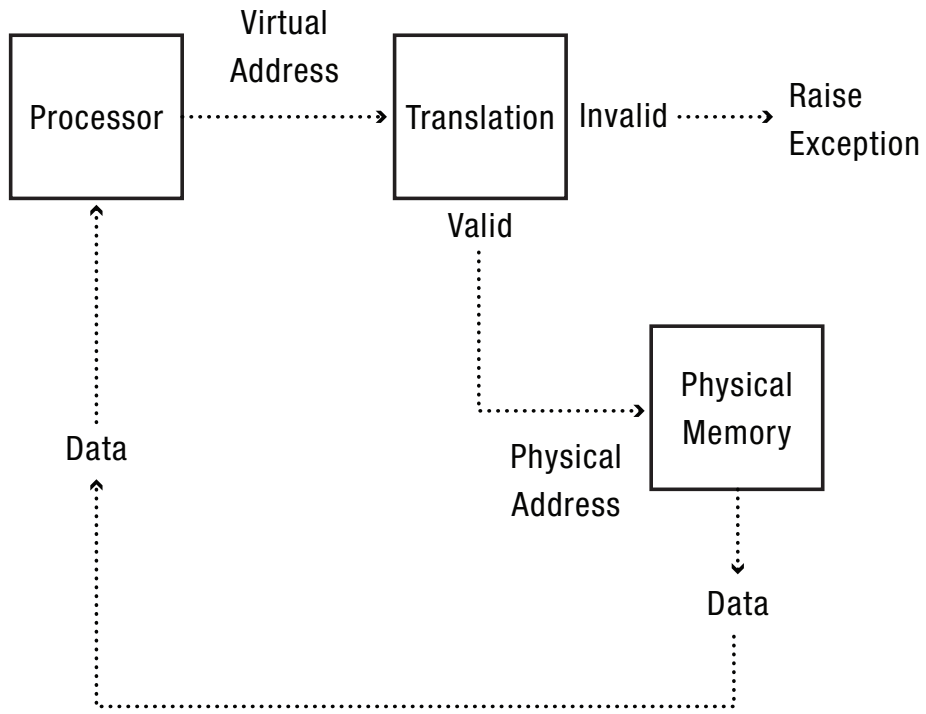
Address Translation

Main Points

- Address Translation Concept
  - How do we convert a virtual address to a physical address?
- Flexible Address Translation
  - Segmentation
  - Paging
  - Multilevel translation
- Efficient Address Translation
  - Translation Lookaside Buffers
  - Virtually and physically addressed caches

Address Translation Concept



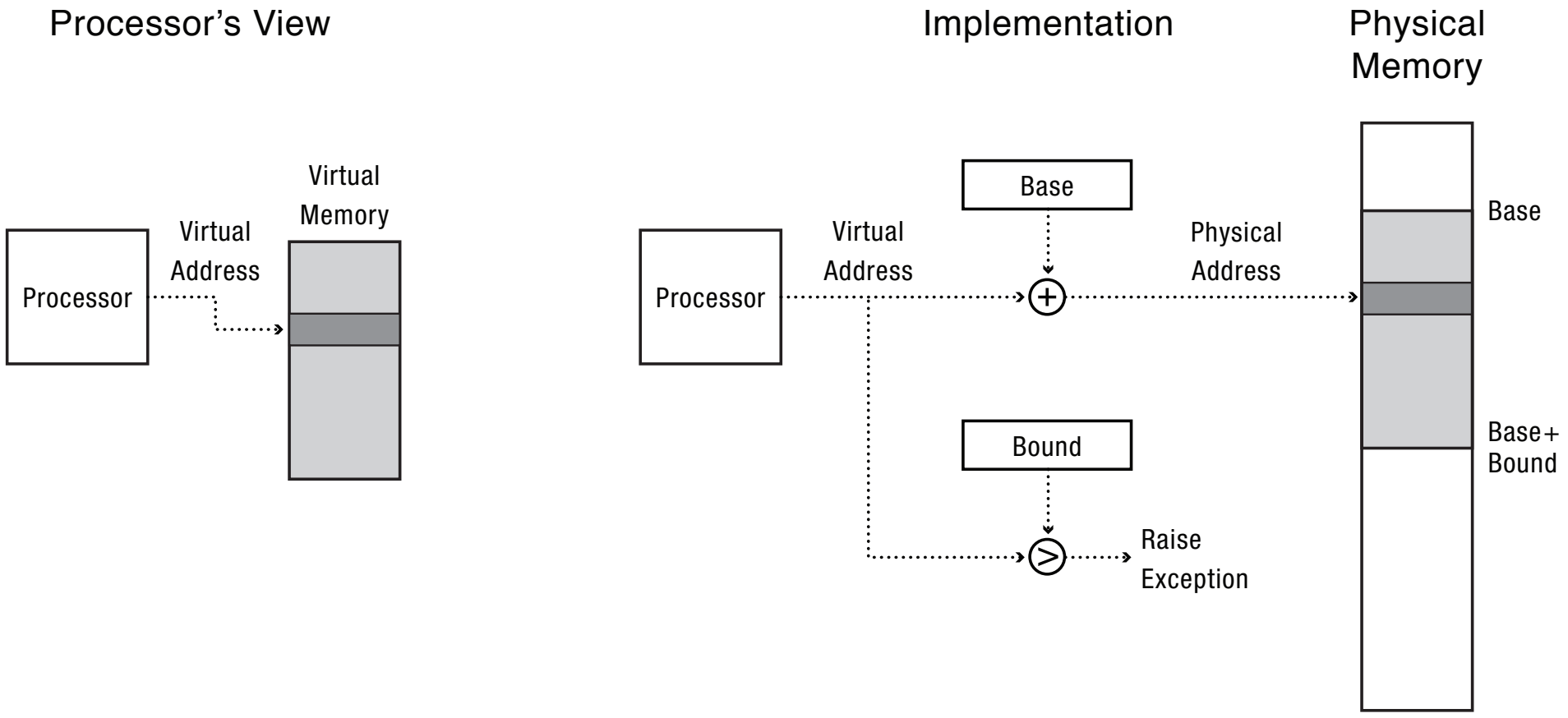
Address Translation Goals

- Memory protection
  - Isolate process to its only memory
  - Prevent virus from re-writing machine instructions
- Memory sharing
  - Shared libraries, interprocess communication
- Sparse addresses
  - Dynamically allocated regions: heaps, stacks, mmap
- Efficiency
  - Reduce fragmentation and copying
  - Runtime lookup cost and TLB hit rate
  - Translation table size
- Portability

Bonus Feature

- What if the kernel can regain control whenever a program reads or writes a particular virtual memory location?
- Examples:
  - Copy on write
  - Zero on reference
  - Fill on demand
  - Demand paging
  - Memory mapped files
  - …

Virtually Addressed Base and Bounds



Virtually Addressed Base and Bounds

- Pros?
  - Simple
  - Fast (2 registers, adder, comparator)
  - Safe
  - Can relocate in physical memory without changing process
- Cons?
  - Can't keep program from accidentally overwriting its own code
  - Can't share code/data with other processes
  - Can't grow stack/heap as needed

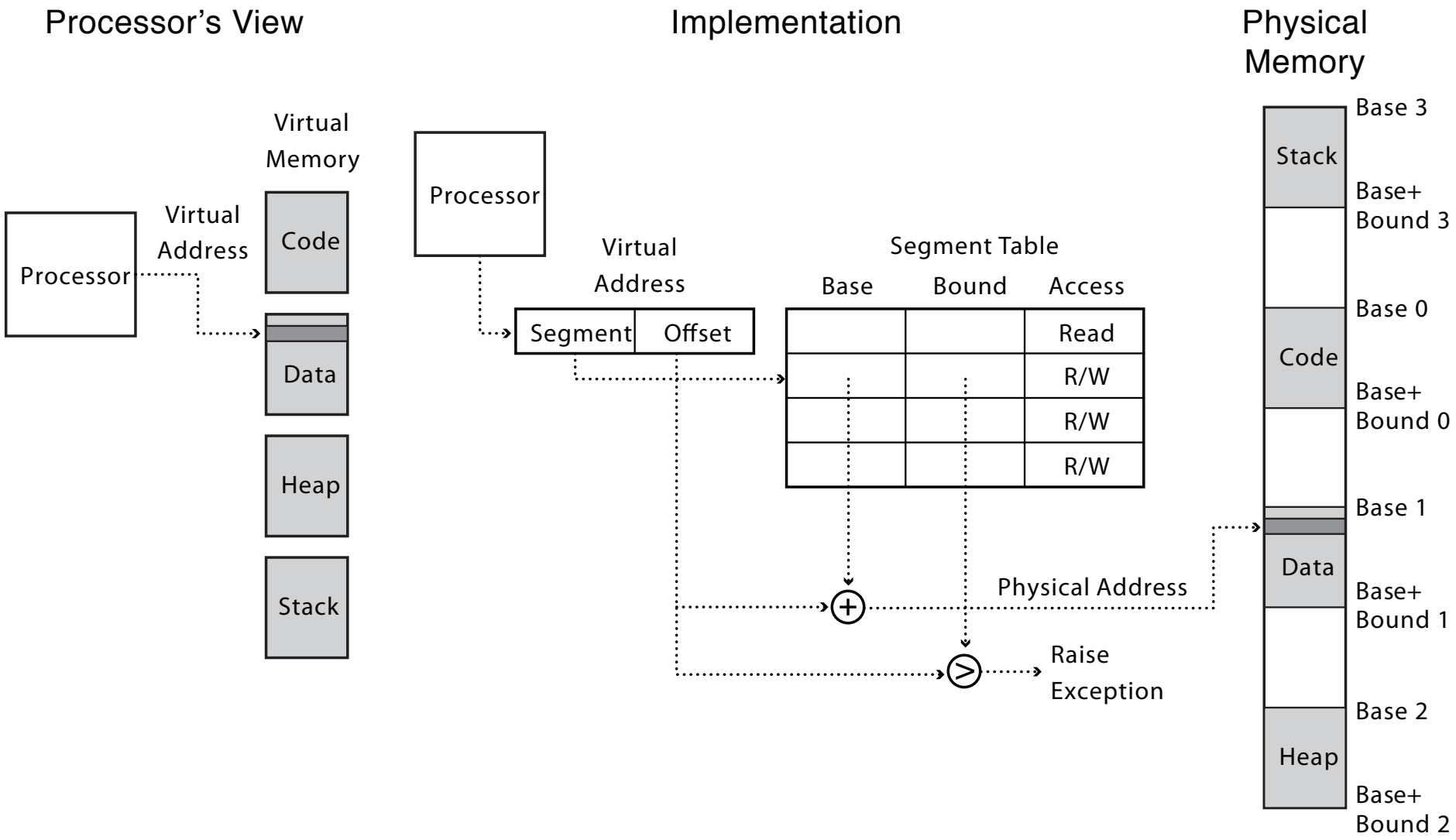
Process Regions or Segments

- Every process has logical regions or segments
  - Contiguous region of process memory
- Code, data, heap, stack, dynamic library (code, data), memory mapped files, …
- Each with its own
  - protection: read-only, read-write, execute-only
  - sharing: code vs. data
  - access pattern: code vs. mmap file

Segmentation

- Segment is a contiguous region of *virtual* memory
- Each process has a segment table (in hardware)
  - Entry in table = segment
- Segment can be located anywhere in physical memory
  - Each segment has: start, length, access permission
- Processes can share segments
  - Same start, length, same/different access permissions

Segmentation



2 bit segment #
12 bit offset

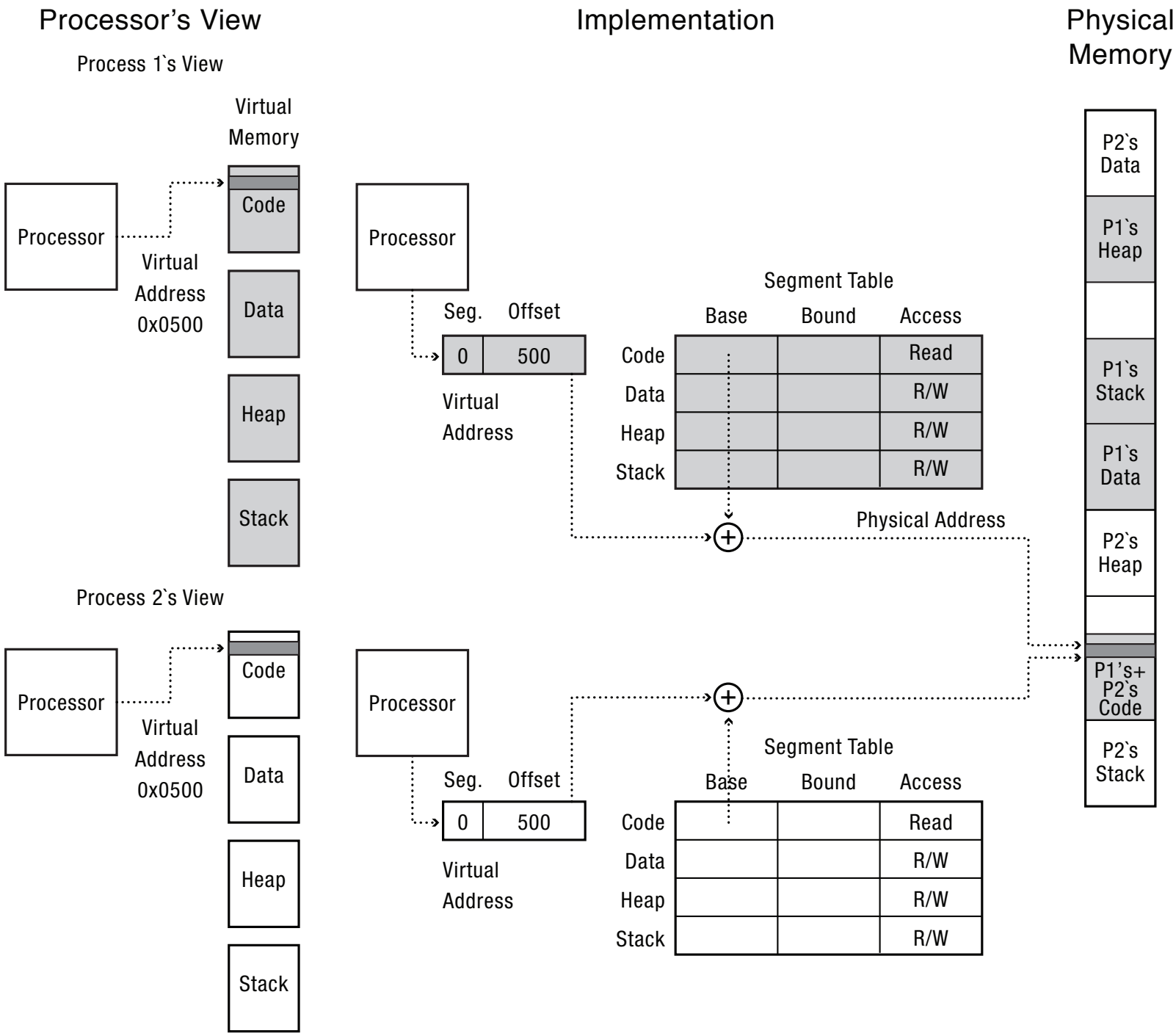
	Segment start	length
code	0x4000	0x700
data	0	0x500
heap	-	-
stack	0x2000	0x1000

Virtual Memory

main: 0:240	store #108, r2
0:244	store pc+8, r31
0:248	jump 360
0:24c	
…	
strlen: 0:360	loadbyte (r2), r3
…	…
0:420	jump (r31)
…	
x: 1:108	a b c \0
…	

Physical Memory

x: 108	a b c \0
…	
main: 4240	store #1108, r2
4244	store pc+8, r31
4248	jump 360
424c	
…	…
strlen: 4360	loadbyte (r2),r3
…	
4420	jump (r31)
…	


Processor's View
Implementation
Physical Memory
Process 1`s View
Virtual Memory
Processor
Code
Data
Heap
Stack
Virtual Address 0x0500
Segment Table
Base
Bound
Access
Seg.
Offset
0
500
Code
Data
Heap
Stack
Read
R/W
R/W
R/W
Virtual Address
Physical Address
P2`s Data
P1`s Heap
P1`s Stack
P1`s Data
P2`s Heap
P1's+ P2`s Code
P2`s Stack
Process 2`s View

Question

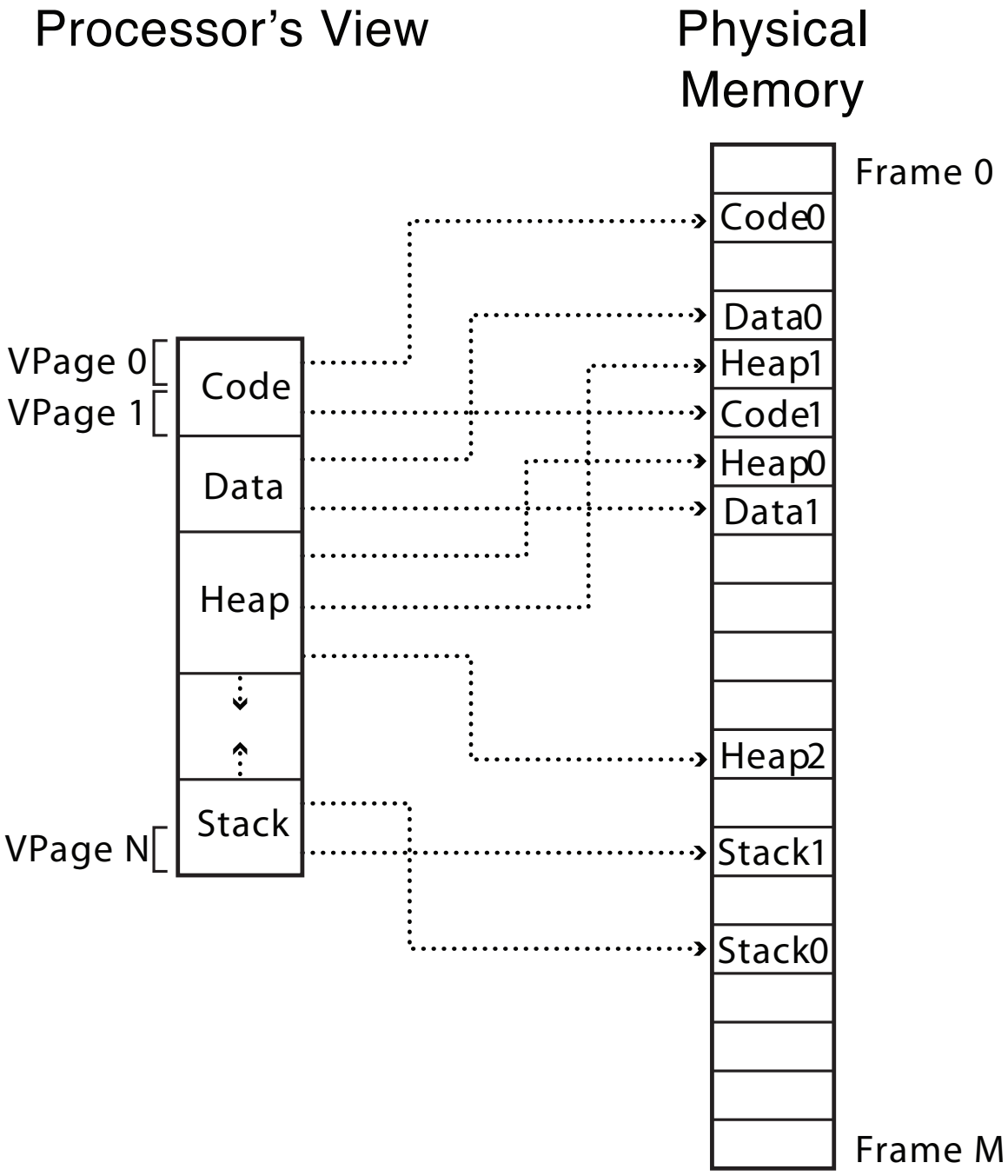
- With segmentation, what is saved/restored on a process context switch?

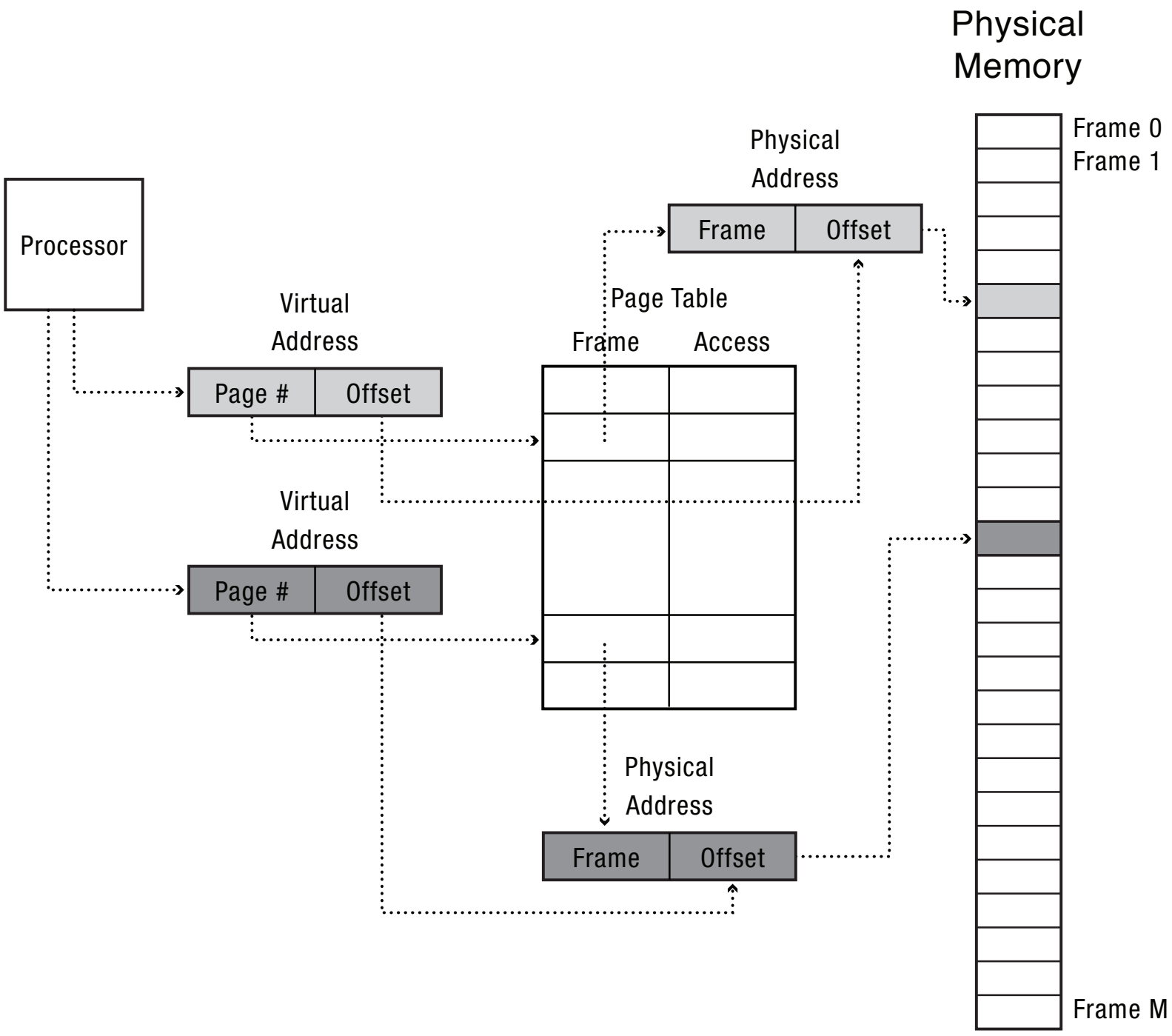
Segmentation

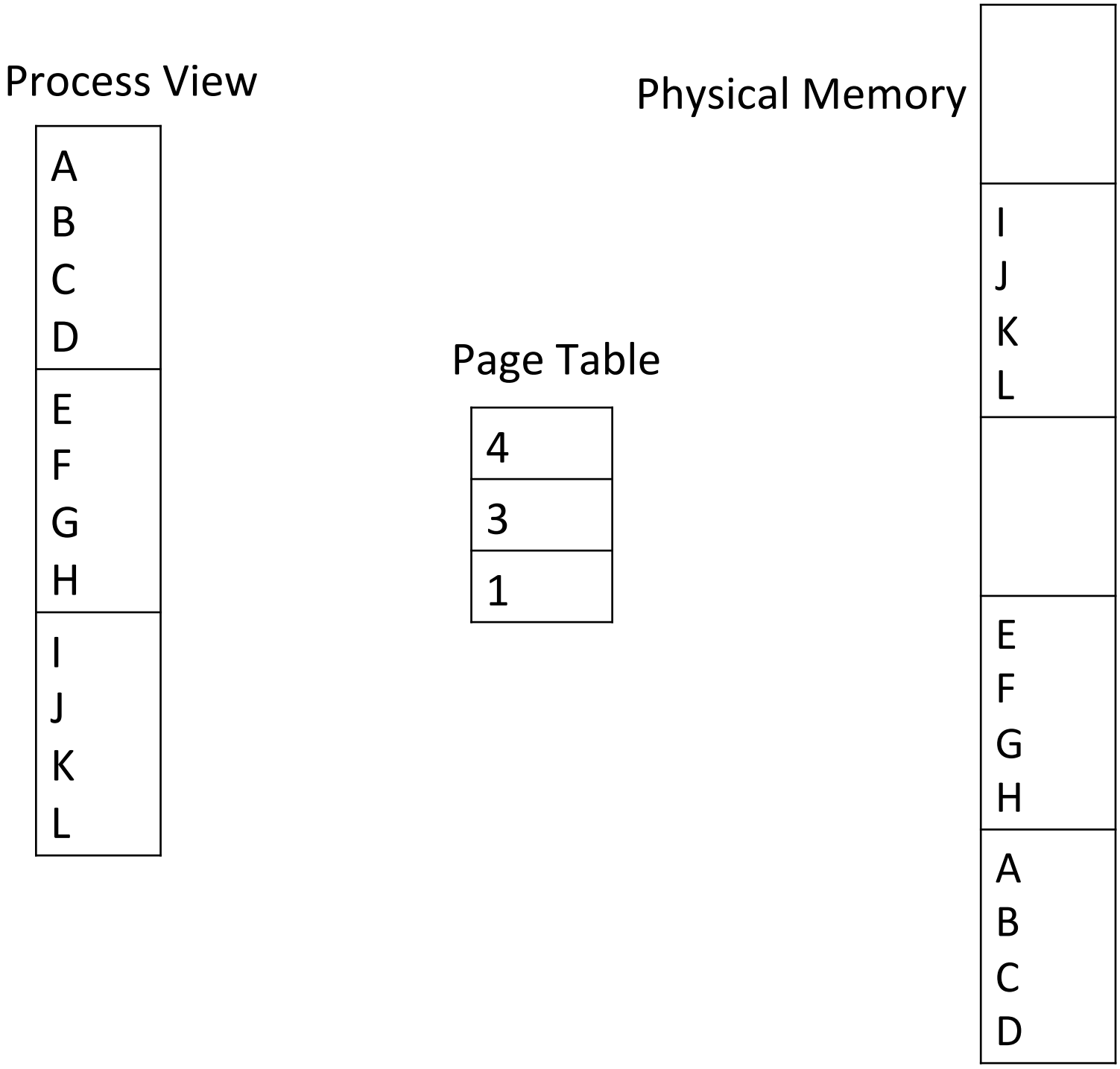
- Pros?
  - Can share code/data segments between processes
  - Can protect code segment from being overwritten
- Cons? Complex memory management
  - Need to find chunk of a particular size
  - May need to rearrange memory to make room for new segment or growing segment (e.g., sbrk)
  - External fragmentation: wasted space between chunks

Paged Translation

- Manage memory in fixed size units, or pages
- Finding a free page is easy
  - Bitmap allocation: 0011111100000001100
  - Each bit represents one physical page frame
- Each process has its own page table
  - Stored in physical memory
  - Hardware registers
    - pointer to page table start
    - page table length


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


Physical
Memory
Frame 0
Frame 1
Frame M
Processor
Physical
Address
Frame
Offset
Page Table
Frame
Access
Virtual
Address
Page #
Offset
Virtual
Address
Page #
Offset
Physical
Address
Frame
Offset


Process View
A
B
C
D
E
F
G
H
I
J
K
L
Page Table
4
3
1
Physical Memory
I
J
K
L
E
F
G
H
A
B
C
D

Paging Questions

- With paging, what is saved/restored on a process context switch?
  - Pointer to page table, size of page table
  - Page table itself is in main memory
- What if page size is very small?
- What if page size is very large?
  - Internal fragmentation: if we don't need all of the space inside a fixed size chunk

Paging and Sharing

- Can we use page tables to share memory between processes?

- Set page tables to point to same page frame
- Need *core map*
  - Array of information about each physical page frame
  - Set of processes pointing to that page frame
  - When reference count goes to zero, can reclaim!

Question

- How big a user stack should I allocate?
- What if some programs need a large stack and others need a small one?

Expand Stack on Reference

- When program references memory beyond end of stack
  - Page fault into OS kernel
  - Kernel allocates some additional memory
    - How much?
  - Remember to zero the memory to avoid accidentally leaking information!
  - Modify page table
  - Resume process

UNIX fork seems inefficient

- Makes a complete copy of process
- Throw copy away on exec
- Do we need to make the copy?
  - One solution: change the syscall interface!

Copy on Write

- Paging allows an efficient fork
  - Copy page table of parent into child
  - Mark all pages (in new/old page tables) as read-only
  - Start child process; restart parent
  - Trap into kernel on write (in child or parent)
  - Copy page
  - Mark both as writeable
  - Resume execution

Question

- Can I start running a program before all of its code is in memory?

Fill On Demand

- Set all page table entries to invalid
- When a page is referenced for first time, kernel trap
- Kernel brings page in from disk
- Resume execution
- Remaining pages can be transferred in the background while program is running

Beyond Paging: Sparse Address Spaces

- Might want many separate segments
  - Per-processor heaps
  - Per-thread stacks
  - Memory-mapped files
  - Dynamically linked libraries
- What if virtual address space is large?
  - 32-bits, 4KB pages => 500K page table entries
  - 64-bits => 4 quadrillion page table entries

Multi-level Translation

- Tree of translation tables
  - Paged segmentation
  - Multi-level page tables
  - Multi-level paged segmentation
- All have pages as lowest level; why?

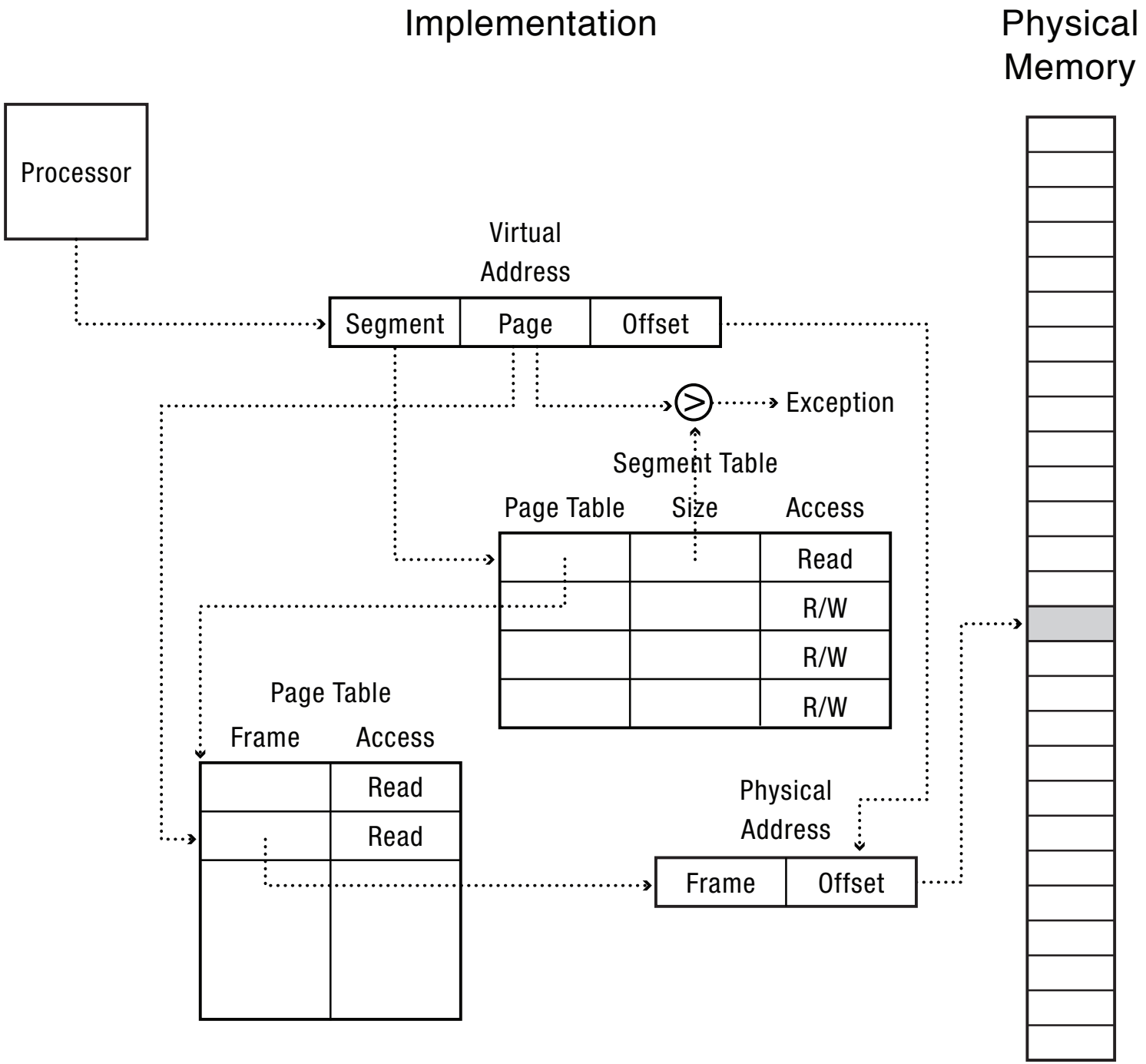
Multilevel Translation with Pages at Lowest Level

- Efficient memory allocation (vs. segments)
- Efficient for sparse addresses (vs. 1 level paging)
- Efficient disk transfers (fixed size units)
- Easier to build translation lookaside buffers
- Efficient reverse lookup (from physical -> virtual)
- Variable granularity for protection/sharing

- Except: see discussion of superpages

Paged Segmentation

- Process memory is segmented
- Segment table entry:
  - Pointer to page table
  - Page table length (# of pages in segment)
  - Access permissions
- Page table entry:
  - Page frame
  - Access permissions
- Share memory or set access permissions at either page or segment-level

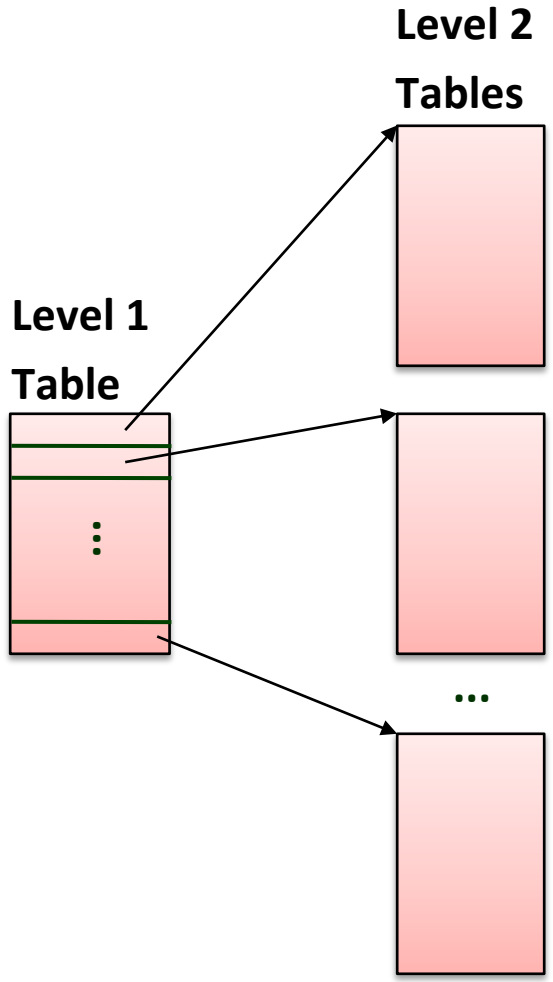

Implementation
Physical Memory
Processor
Virtual Address
Segment
Page
Offset
Exception
Segment Table
Page Table
Size
Access
Read
R/W
R/W
R/W
Page Table
Frame
Access
Read
Read
Physical Address
Frame
Offset

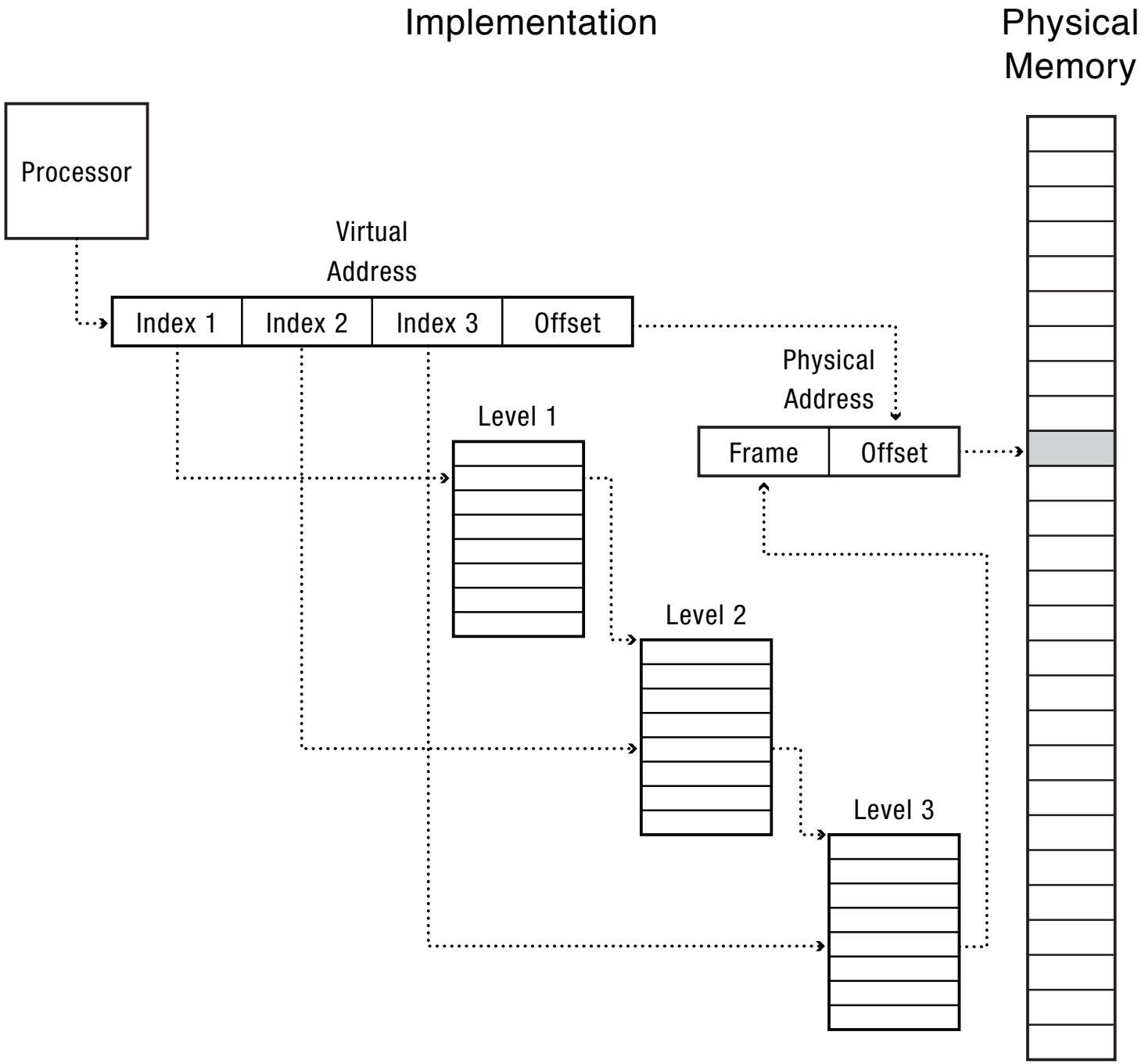
Question

- With paged segmentation, what must be saved/restored across a process context switch?

Multi-level or hierarchical page tables

- Ex: 2-level page table
  - Level 1 table: each PTE points to a page table
  - Level 2 table: each PTE points to a page
- Can share/protect/page in/out at either level 1 or level 2




Implementation
Physical Memory
Processor
Virtual Address
Index 1
Index 2
Index 3
Offset
Physical Address
Frame
Offset
Level 1
Level 2
Level 3

Question

- Write pseudo-code for translating a virtual address to a physical address for a system using 3-level paging, with 8 bits of address per level

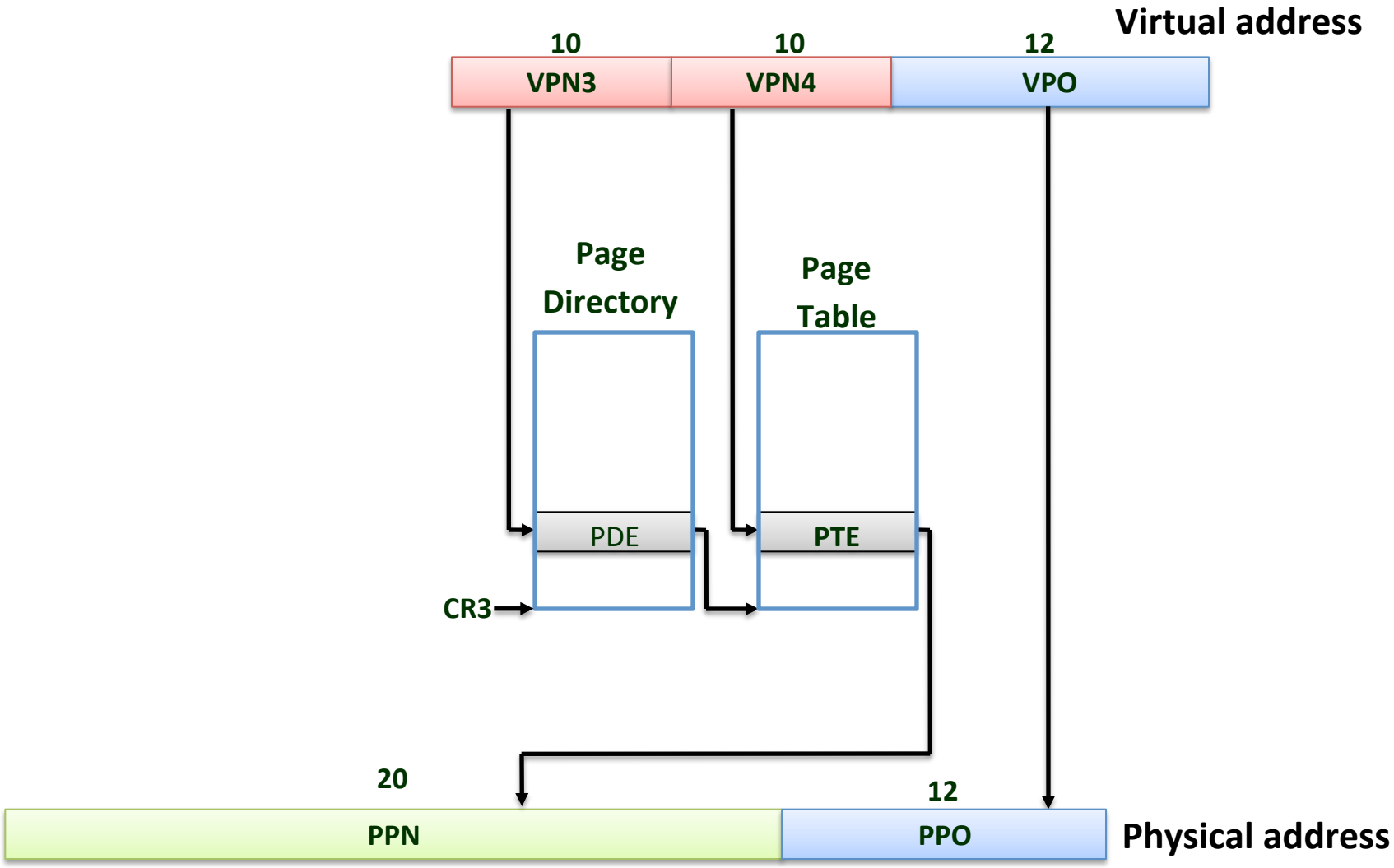
x86 Multilevel Paged Segmentation

- Global Descriptor Table (segment table)
- Each segment descriptor
  - Pointer to (multilevel) page table
  - Segment length
  - Segment access permissions
- Context switch
  - change global descriptor table register (GDTR), pointer to global descriptor table
  - Side effect: invalidates TLB

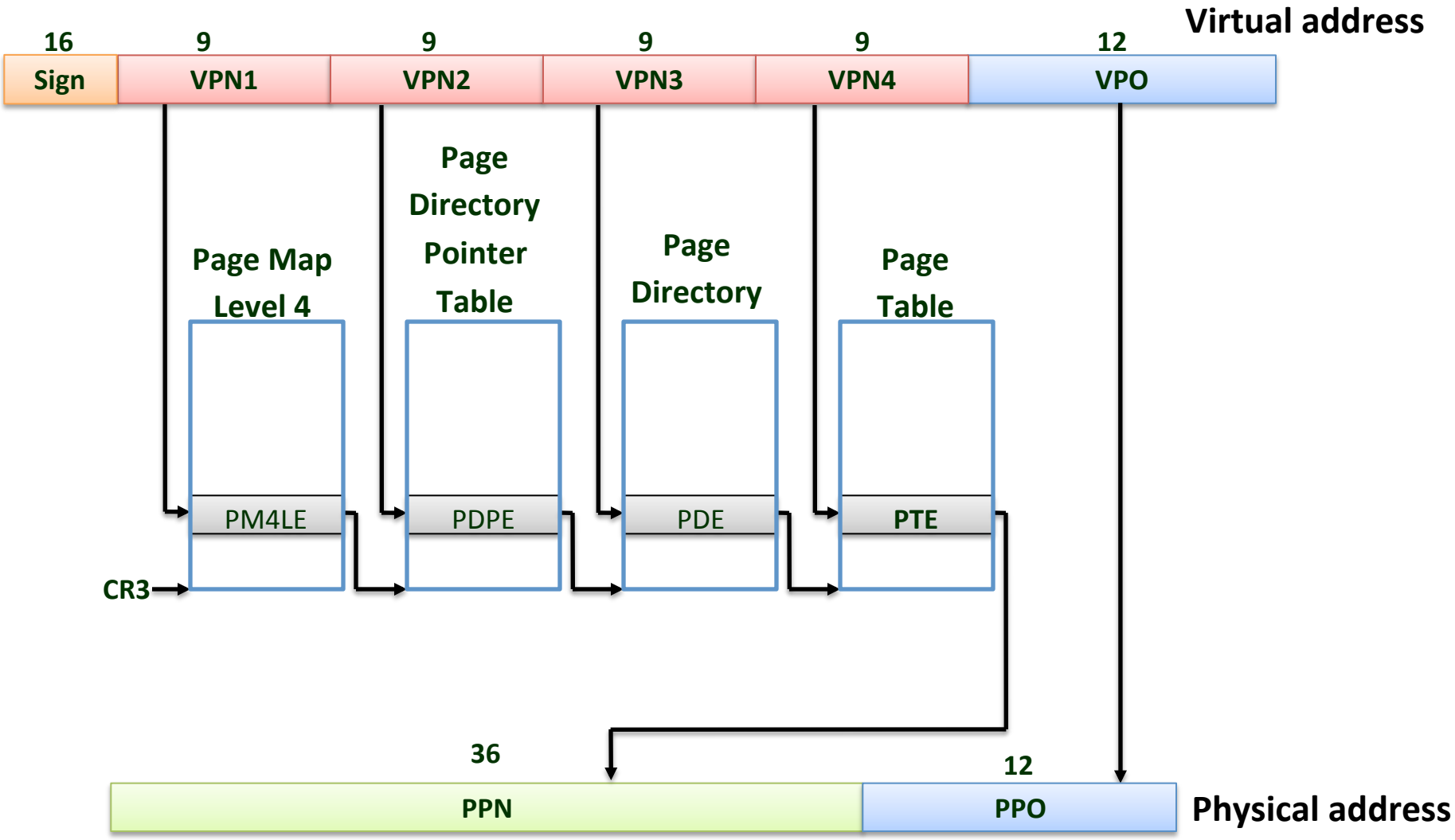
x86 Multilevel Paging

- Omit sub-tree if no valid addresses
  - Good for sparse address space
- 4KB pages
- Each level of page table fits in one page
- 32-bit: two level page table (per segment)
- 64-bit: four level page table (per segment)

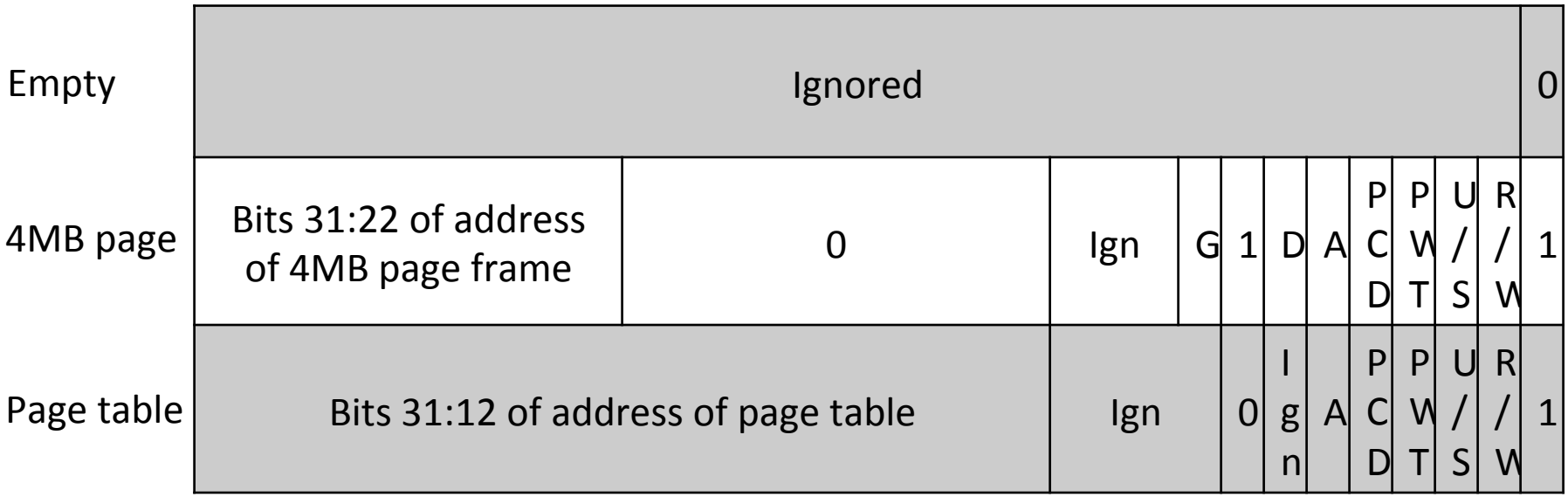
x86-32 Paging



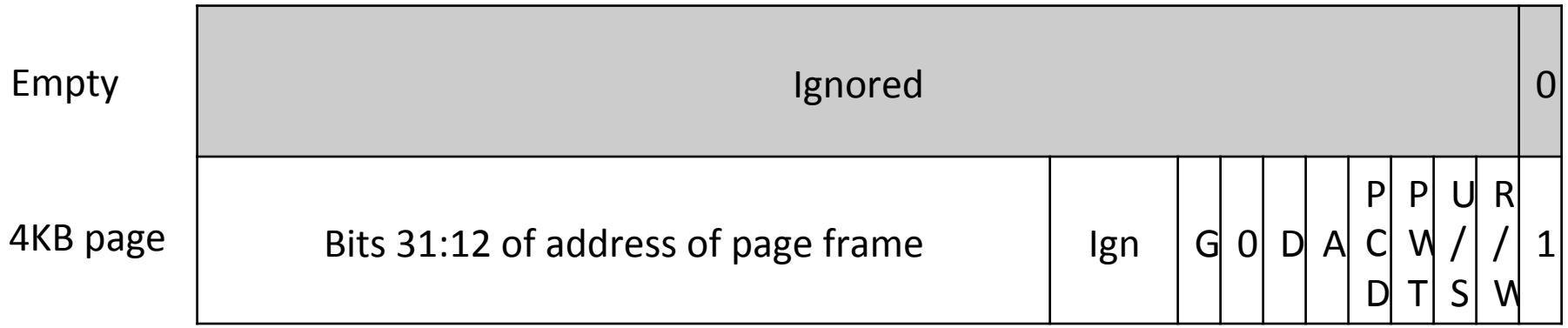
x86-64 Paging



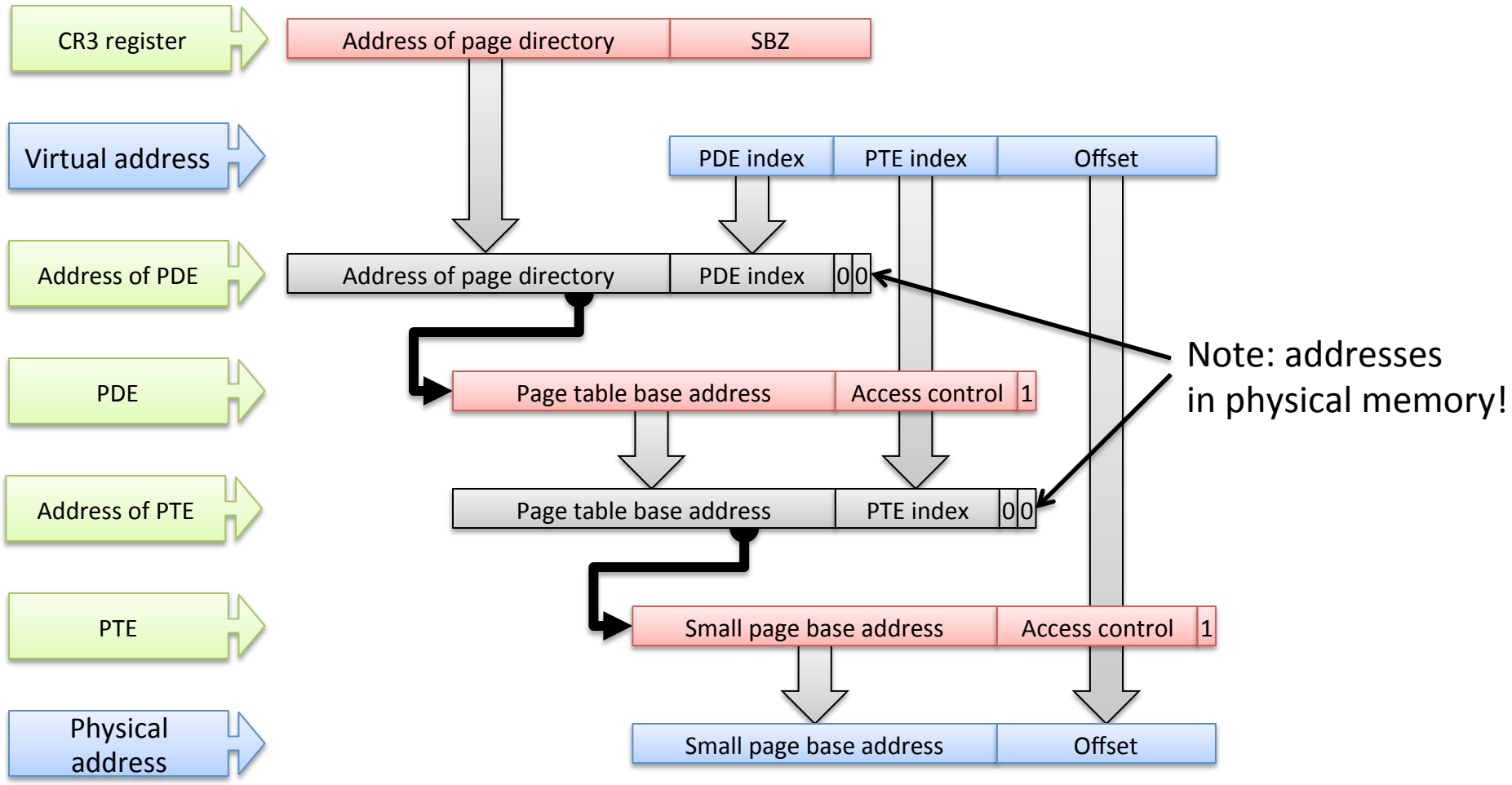
Page directory entries (x86 32 bit)



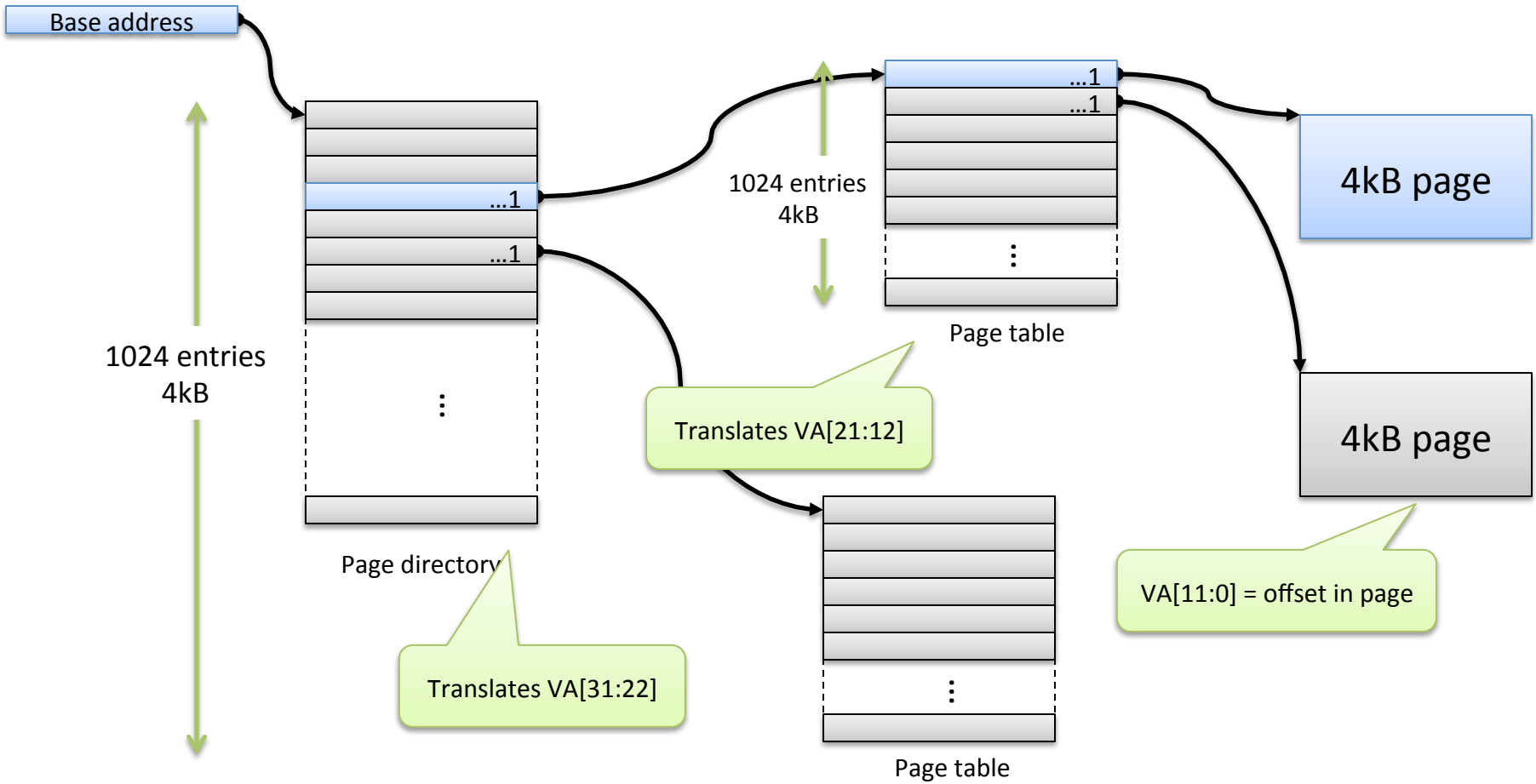
Page table entries (x86 32 bit)



Small page translation



Page table for small pages



Multilevel Translation

- Pros:
  - Allocate/fill only page table entries that are in use
  - Simple memory allocation
  - Share at segment or page level
- Cons:
  - Space overhead: one pointer per virtual page
  - Multiple lookups per memory reference

Page Translation in the OS

- OS's need to keep their own data structures
  - List of memory objects (segments)
  - Virtual page -> physical page frame
  - Physical page frame -> set of virtual pages
  - Keep track of copy on write, load on demand, …
- Why not just use the hardware page tables?

Kernel Page Translation

- Kernel maintains its own page translation data structures
  - Portable, flexible
  - Copy changes down into hardware page tables
- Example: Inverted page table
  - Hash from virtual page -> physical page
  - Space proportional to # of physical pages
- Example: virtual/shadow page table
  - Linux kernel tables mirror x86 structure, even on ARM

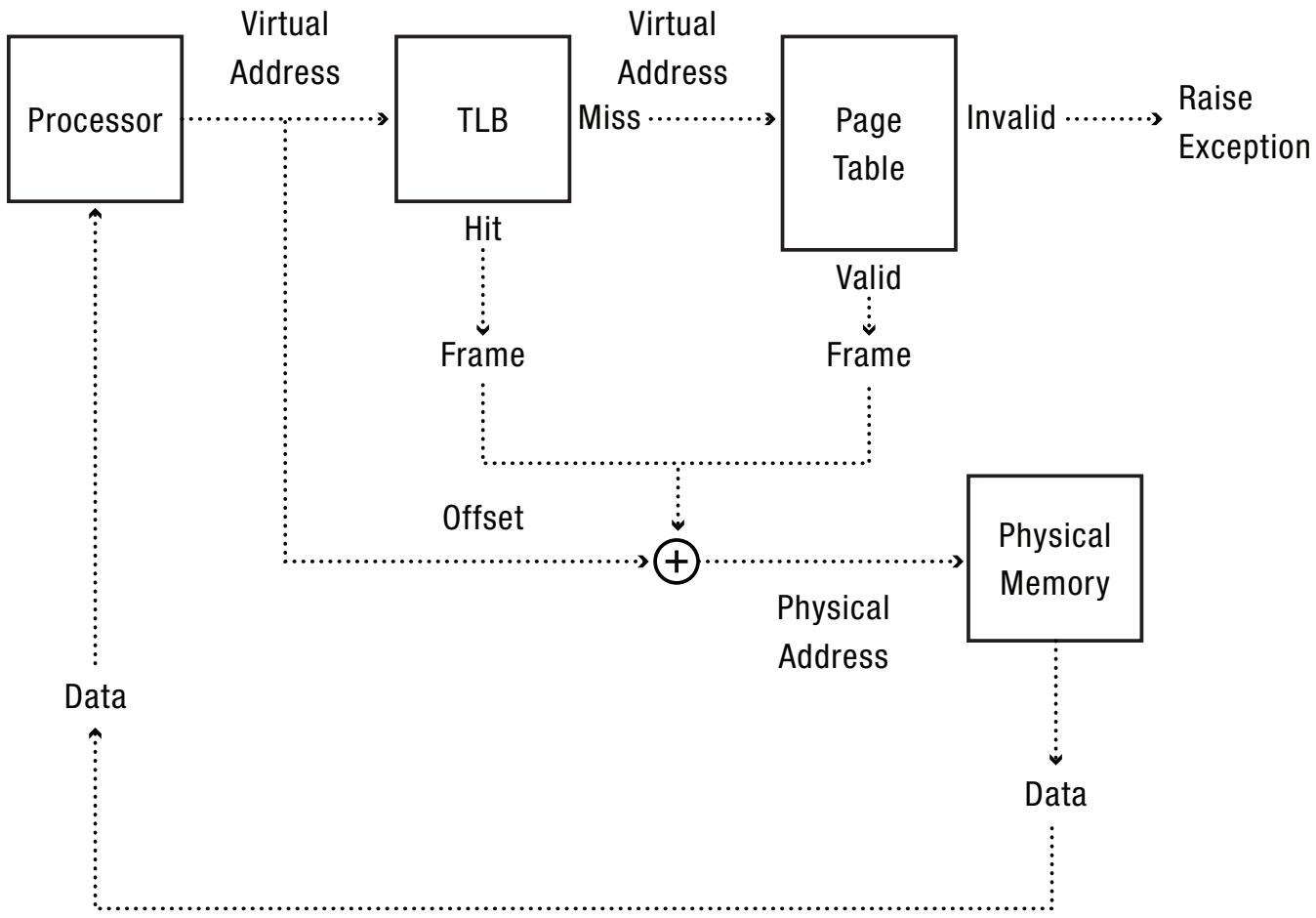
Efficient Address Translation

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

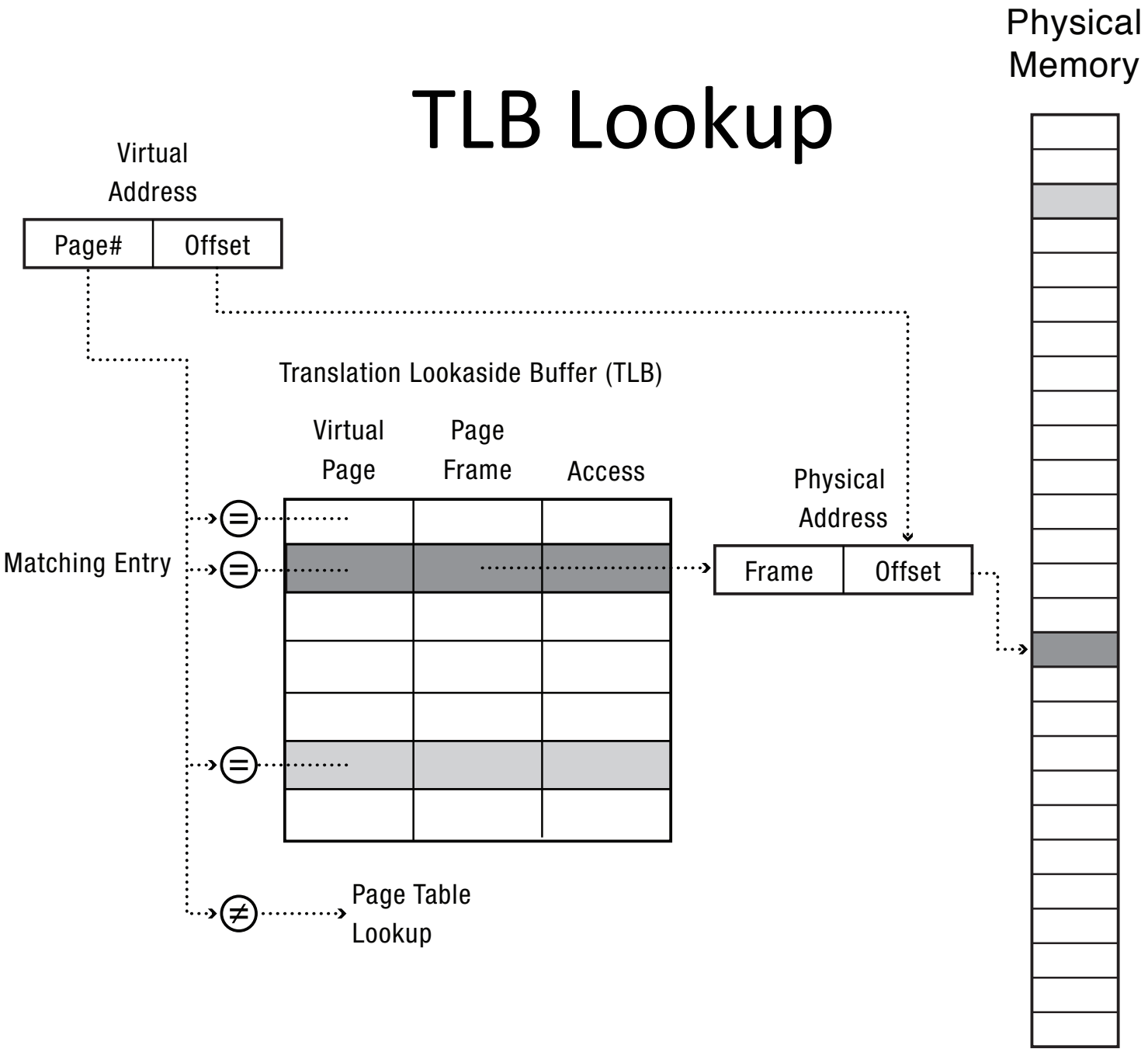
  Cost of TLB lookup +

  Prob(TLB miss) * cost of page table lookup

TLB and Page Table Translation



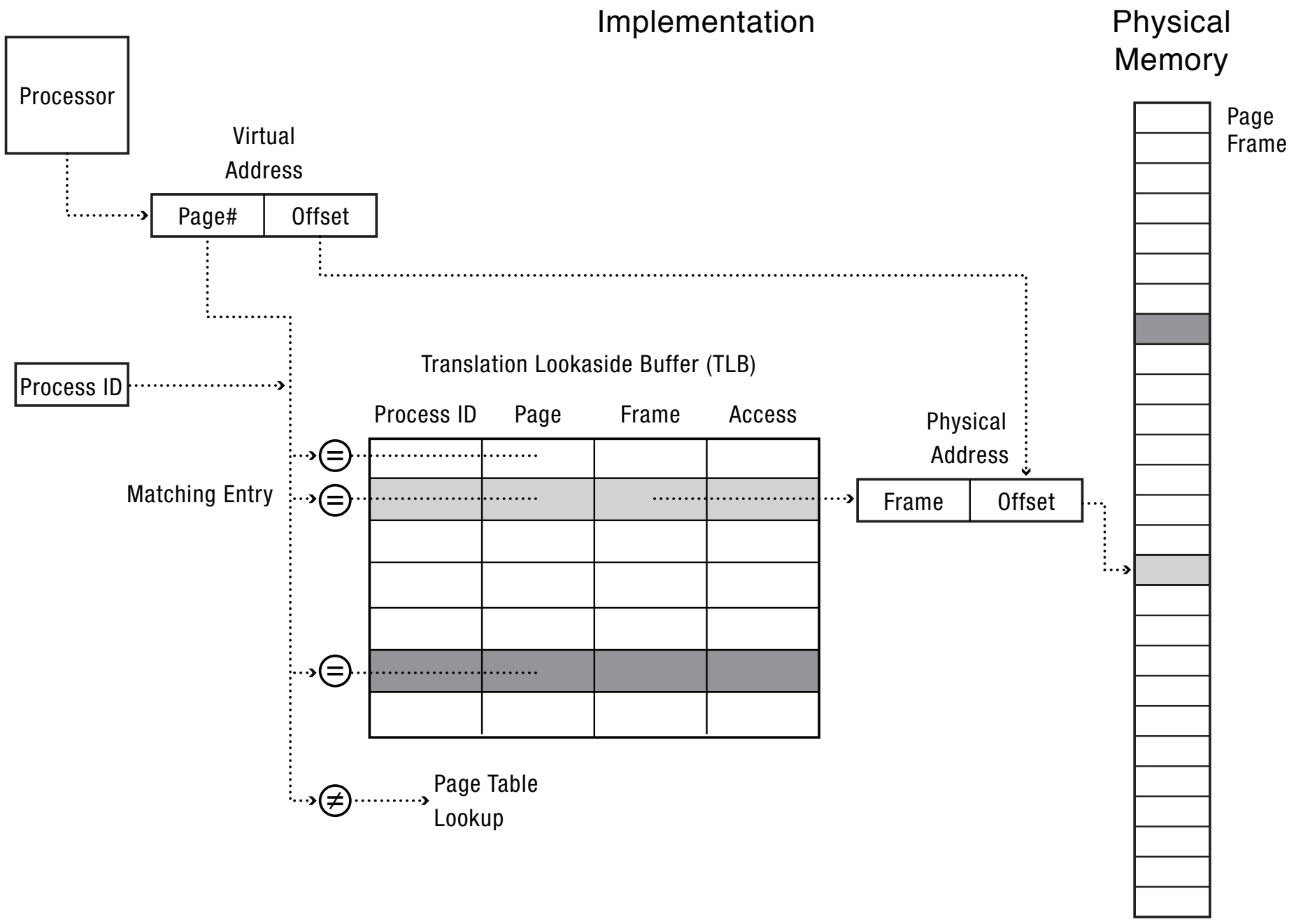
TLB Lookup


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

Question

- What happens on a context switch?
  - Reuse TLB?
  - Discard TLB? (xk resets TLB)

- Solution: Tagged TLB
  - Each TLB entry has process ID
  - TLB hit only if process ID matches current process


Implementation
Physical Memory
Processor
Virtual Address
Page#
Offset
Page Frame
Process ID
Translation Lookaside Buffer (TLB)
Process ID
Page
Frame
Access
Physical Address
Matching Entry
Frame
Offset
Page Table Lookup

MIPS Address Translation

- Software-Loaded Translation lookaside buffer (TLB)
  - Cache of virtual page -> physical page translations
  - If TLB hit, physical address
  - If TLB miss, trap to kernel
  - Kernel fills TLB with translation and resumes execution
- Kernel can implement *any* page translation
  - Page tables
  - Multi-level page tables
  - Inverted page tables
  - …

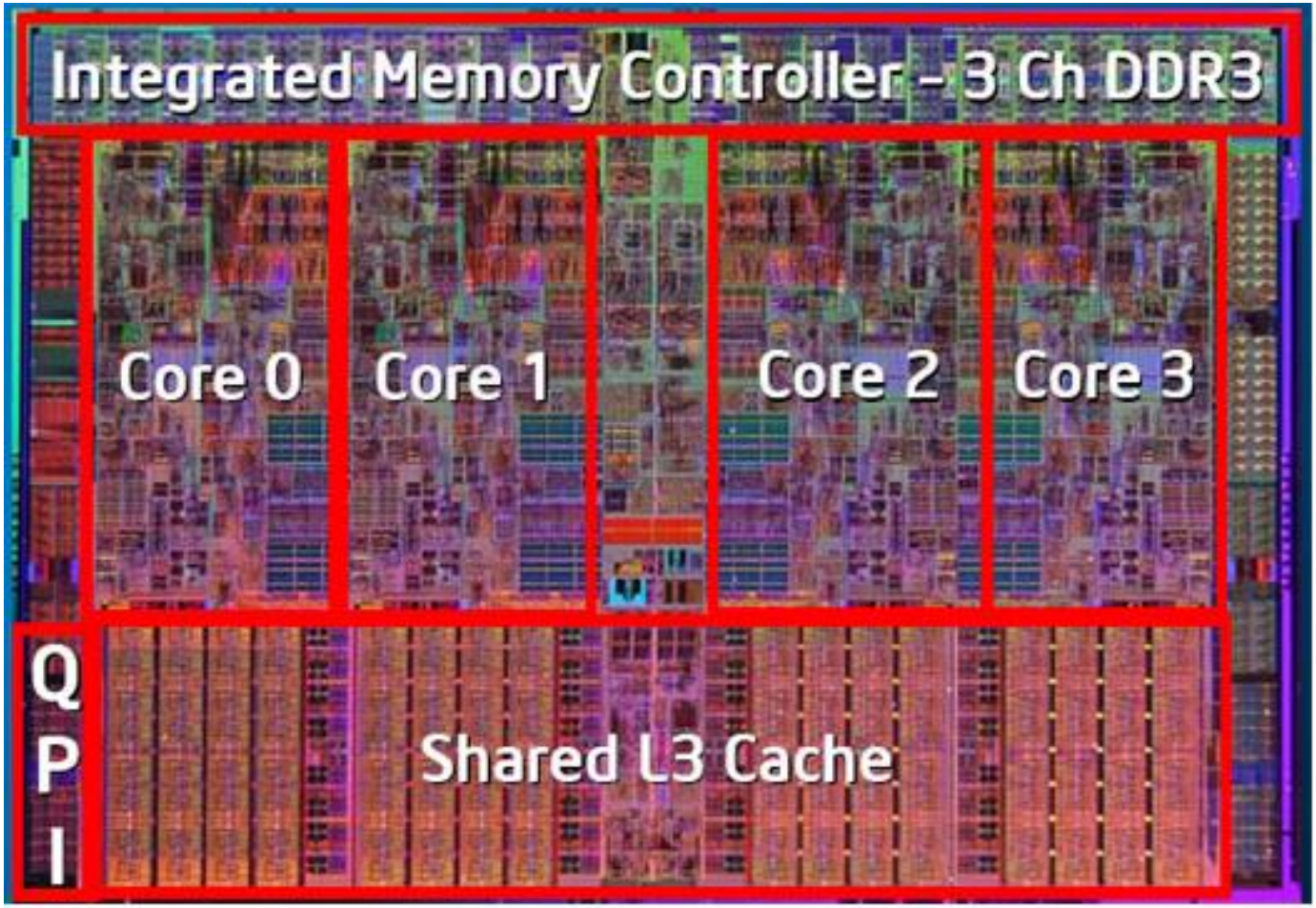
Question

- What is the cost of a TLB miss on a modern processor?
  - Cost of multi-level page table walk
  - MIPS: plus cost of trap handler entry/exit

Hardware Design Principle

The bigger the memory, the slower the memory

Intel i7


Integrated Memory Controller – 3 Ch DDR3
Core 0
Core 1
Core 2
Core 3
QPI
Shared L3 Cache

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

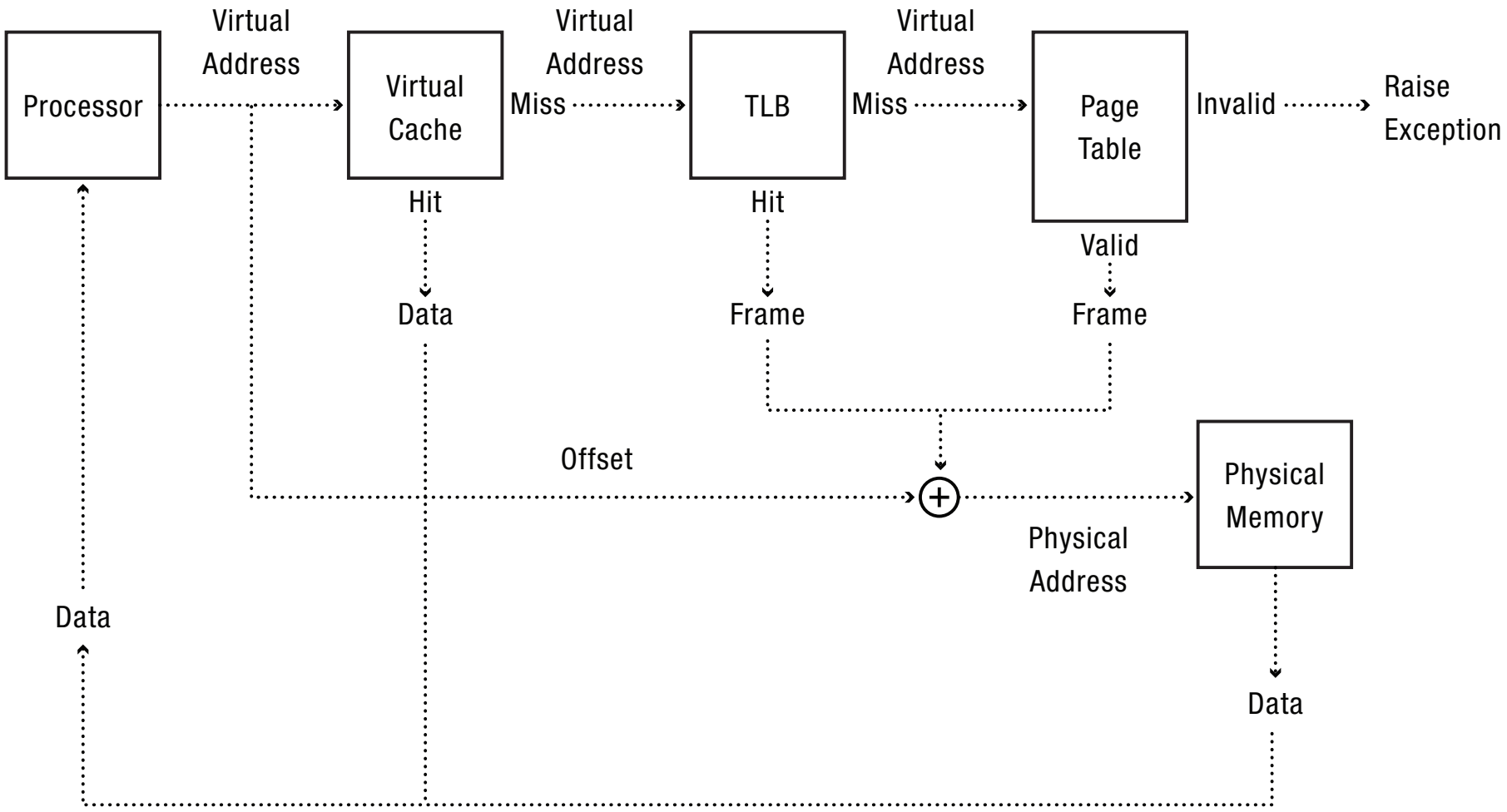
Question

- What is the cost of a first level TLB miss?
  - Second level TLB lookup
- What is the cost of a second level TLB miss?
  - 64 bit x86: 4-level page table walk
- How expensive is a 4-level page table walk?

Virtually Addressed vs. Physically Addressed Caches

- First level cache has at most a few cycles
  - Delays every instruction fetch and data reference
- Lookup TLB to get physical address, then lookup physical address in the cache?
  - Too slow!
- Instead, lookup virtual address in cache
- In parallel, lookup TLB in case of a cache miss

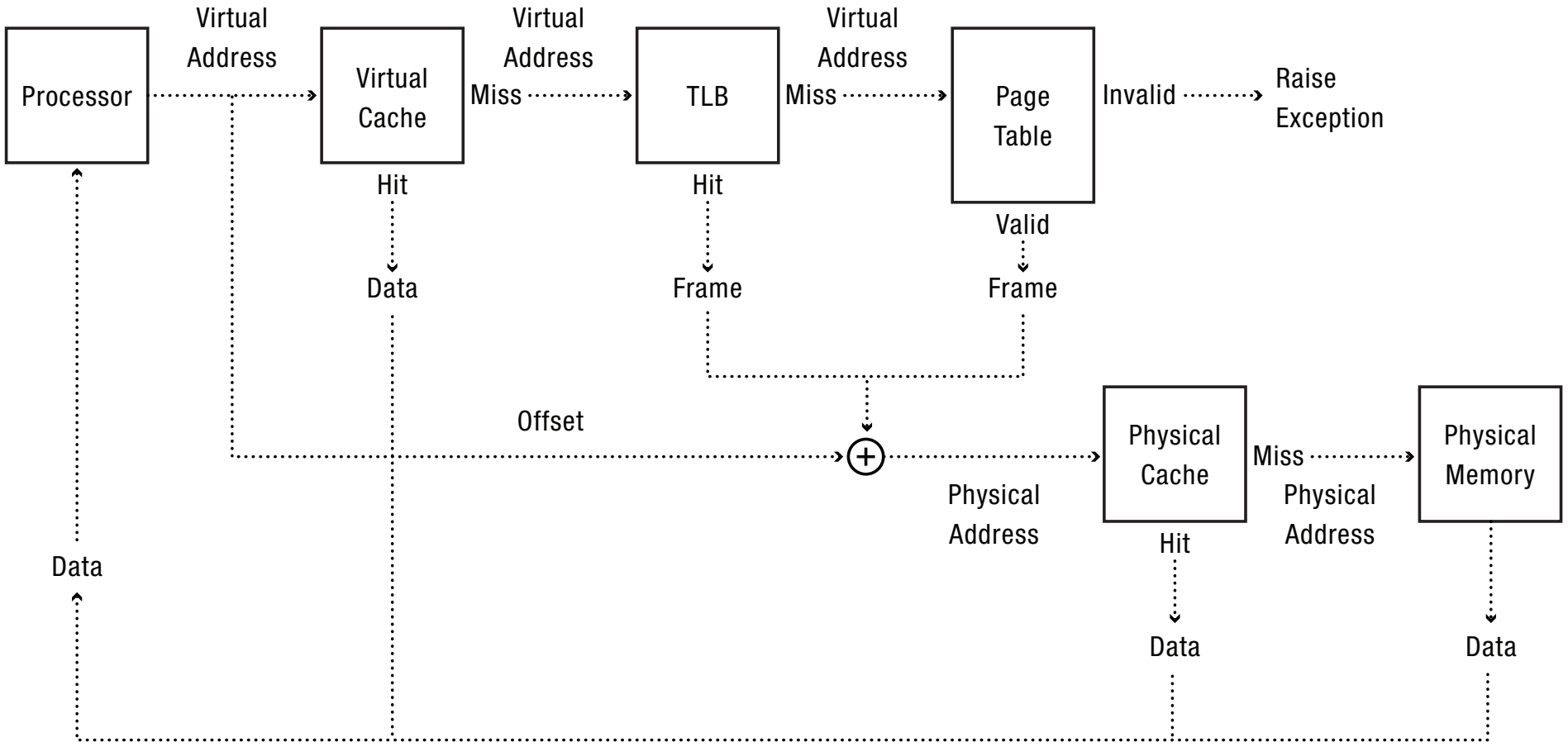
Virtually Addressed Caches



Question

- With a virtual cache, what do we need to do on a context switch?

Physically Addressed Cache



TLB Size (Intel Kaby Lake, 2017)

First level TLB

- Instruction: 128 entries
- Data: 64 entries

Second level TLB

- 1536 entries

Modern server can have 10 TB (!) of DRAM

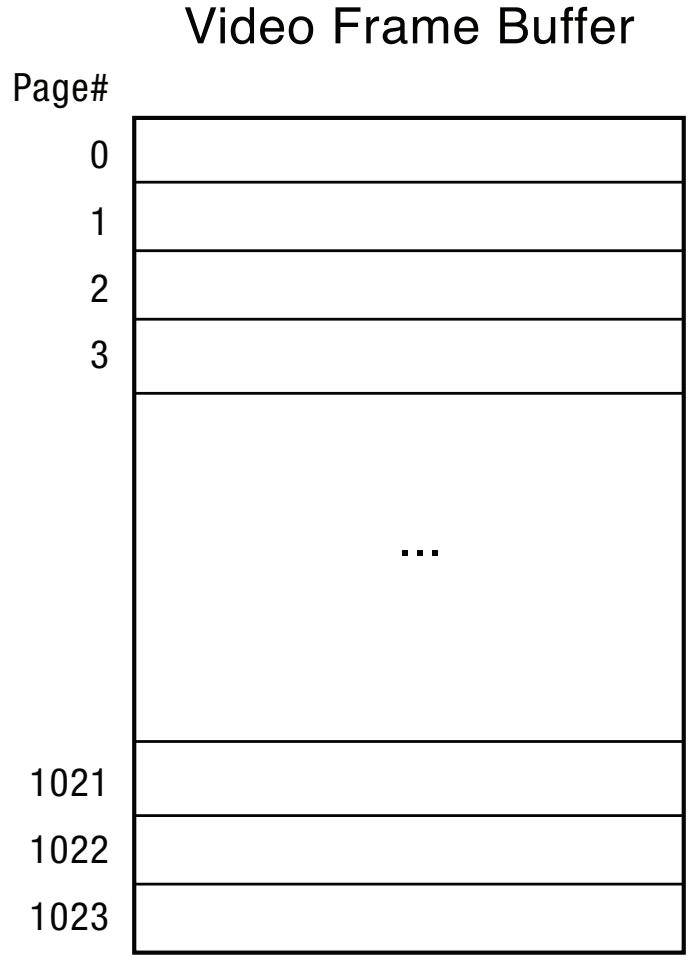
- 10-20% of server CPU time spent in TLB misses

When Do TLBs Work/Not Work?

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

2017 laptop: 3K x 2K =
24MB

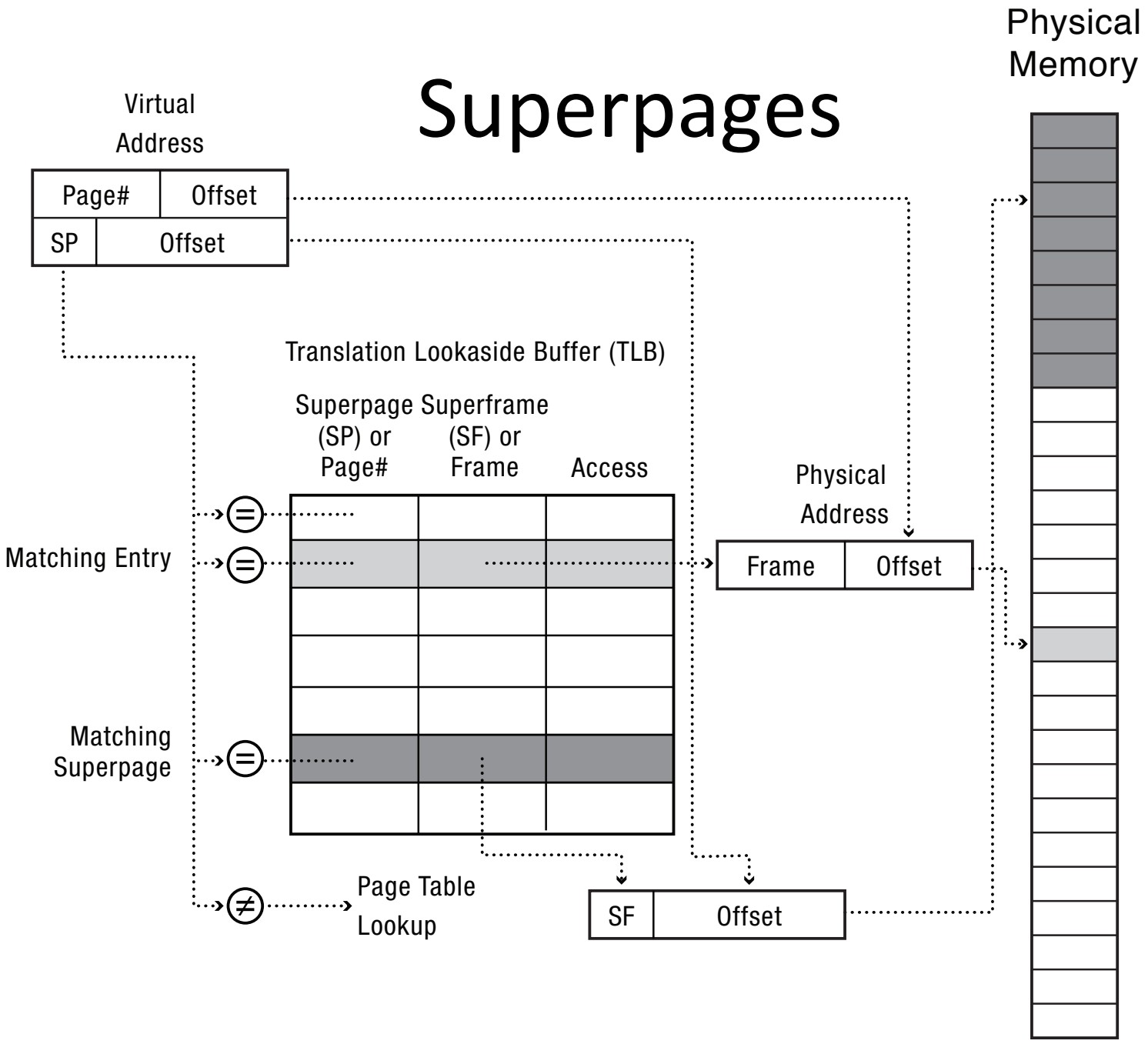
4K display: 4K x 3K =
48MB



Superpages

- On x86 and ARM, TLB entry can be
  - A page
  - A superpage: a set of contiguous, aligned pages
- x86: superpage is set of pages in one page table
  - One page: 4KB
  - One page table: 2MB
  - One page table of page tables: 1GB
  - One page table of page tables of page tables: 0.5TB

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

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

Virtual Cache Shootdown

- When permissions change for a page, we must shoot down the TLB entry on every CPU
- What about the contents of the virtual cache?
- Lazy shootdown of the virtual cache:
  - Lookup virtually addressed cache and TLB in *parallel*
  - Use the TLB to verify virtual address is still valid!
  - Evict entry from cache if not

Virtual Cache Aliases

- Alias: two (or more) virtual cache entries that refer to the same physical memory
  - A consequence of a tagged virtually addressed cache!
  - A write to one copy needs to update all copies
- Solution:
  - Virtual cache keeps both virtual and physical address for each entry
  - Lookup virtually addressed cache and TLB in *parallel*
  - Check if physical address from TLB matches any other entries, and update/invalidate those copies

x86 caches

- 64 byte line size
- Physically indexed
- Physically tagged
- Write buffer

Multicore and Hyperthreading

- Modern CPU has several functional units
  - Instruction decode
  - Arithmetic/branch
  - Floating point
  - Instruction/data cache
  - TLB
- Multicore: replicate functional units (i7: 4)
  - Share second/third level cache, second level TLB
- Hyperthreading: logical processors that share functional units (i7: 2)
  - Better functional unit utilization during memory stalls
- No difference from the OS/programmer perspective
  - Except for performance, affinity, …

Address Translation Uses

- Process isolation
  - Keep a process from touching anyone else's memory, or the kernel's
- Efficient interprocess communication
  - Shared regions of memory between processes
- Shared code segments
  - E.g., common libraries used by many different programs
- Program initialization
  - Start running a program before it is entirely in memory
- Dynamic memory allocation
  - Allocate and initialize stack/heap pages on demand

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
- Memory mapped files
  - Access file data using load/store instructions
- Demand-paged virtual memory
  - Illusion of near-infinite memory, backed by disk or memory on other machines

Address Translation (even more)

- Checkpointing/restart
  - Transparently save a copy of a process, without stopping the program while the save happens
- 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