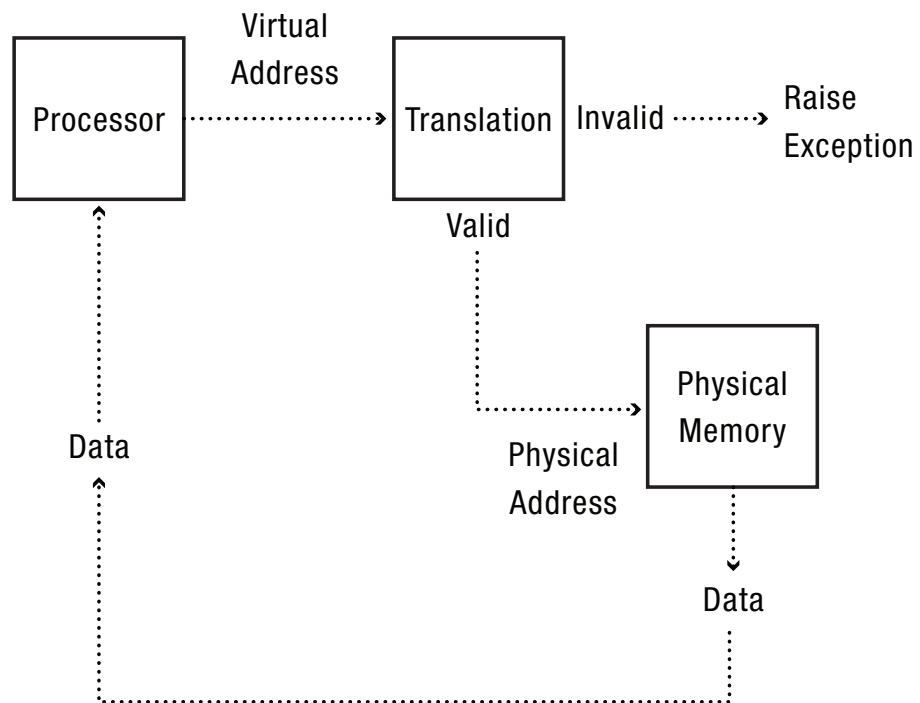# Address Translation

Main Points

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

Address Translation Concept

Processor →(Virtual Address)→ Translation →(Invalid)→ Raise Exception

Translation →(Valid)→ Physical Memory (via Physical Address)

Physical Memory →(Data)→ Processor (Data)

Address Translation Goals

- Memory protection
- Memory sharing
  - Shared libraries, interprocess communication
- Sparse addresses
  - Multiple regions of dynamic allocation (heaps/stacks)
- Efficiency
  - Memory placement
  - Runtime lookup
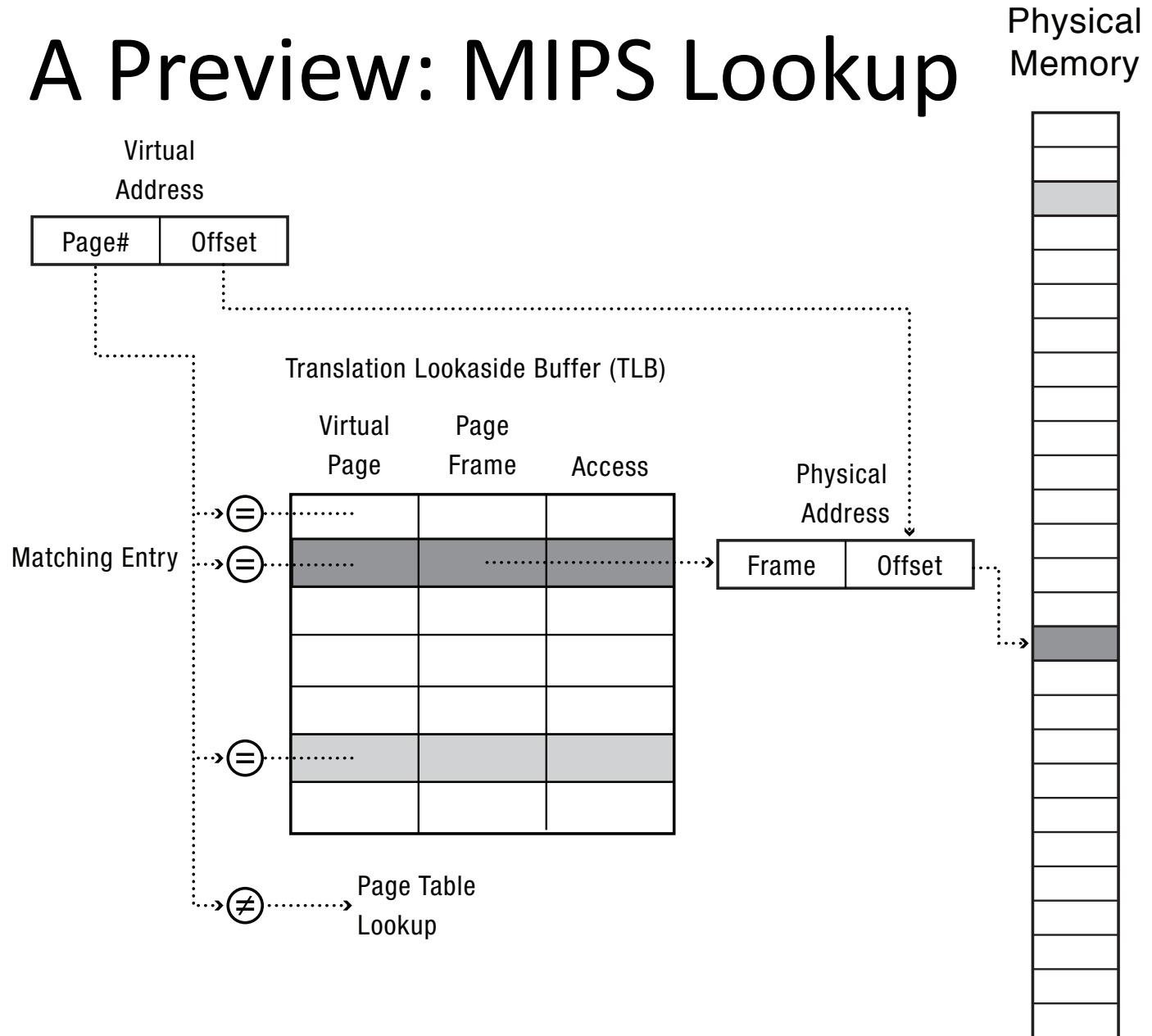 - Compact translation tables
- Portability

Bonus Feature

- What can you do if you can (selectively) gain control whenever a program reads or writes a particular virtual memory location?
- Examples:
  - Copy on write
  - Zero on reference
  - Fill on demand
  - Demand paging
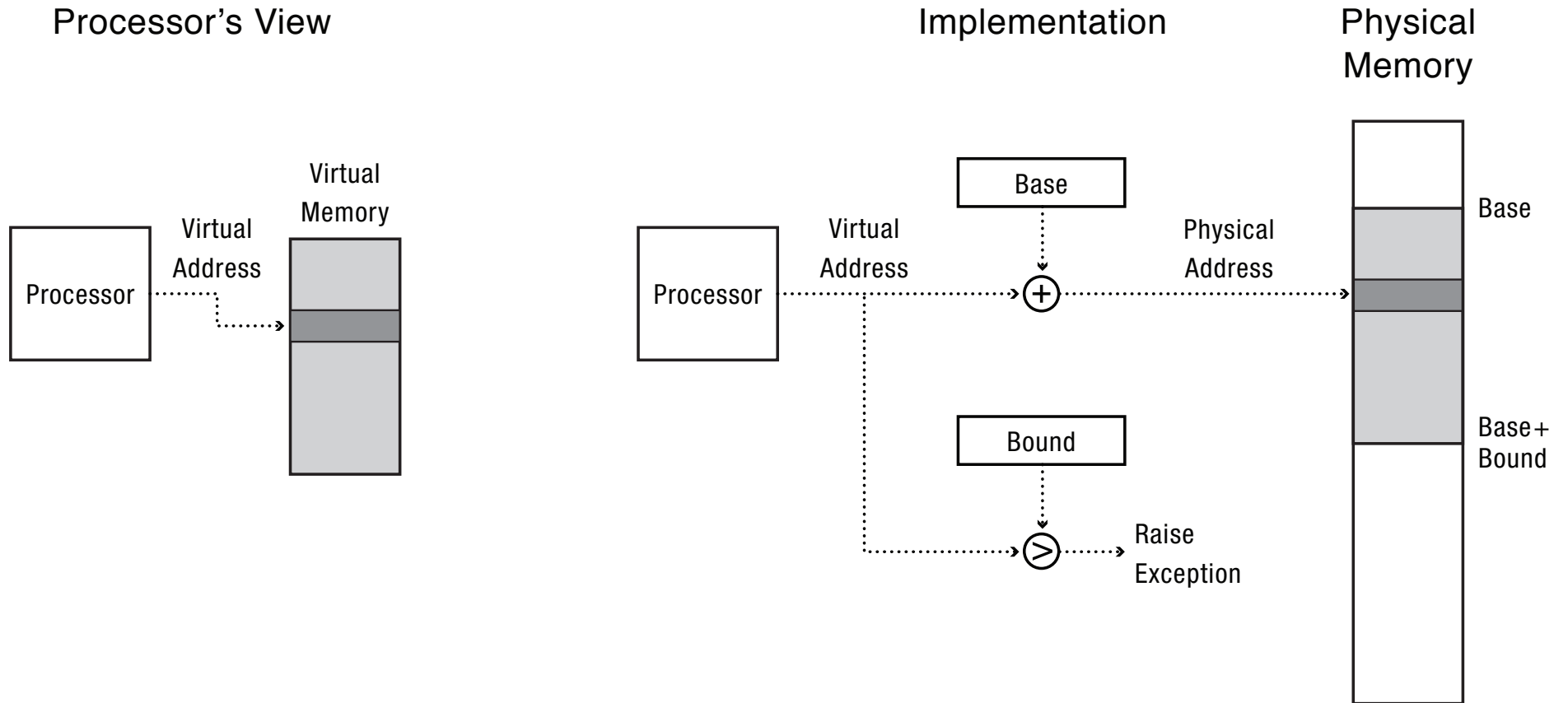 - Memory mapped files
  - …

A Preview: 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
  - …

A Preview: MIPS Lookup

Physical Memory

Virtual Address

Page#	Offset

Translation Lookaside Buffer (TLB)

Virtual Page	Page Frame	Access

Matching Entry

Physical Address

Frame	Offset

Page Table Lookup

Virtually Addressed Base and Bounds

Processor's View

Implementation

Physical Memory

Processor → Virtual Address → Virtual Memory

Base

Processor → Virtual Address → ⊕ → Physical Address → Physical Memory

Bound → ⊗ → Raise Exception

Base

Base+ Bound

Question

- With virtually addressed base and bounds, what is saved/restored on a process context switch?

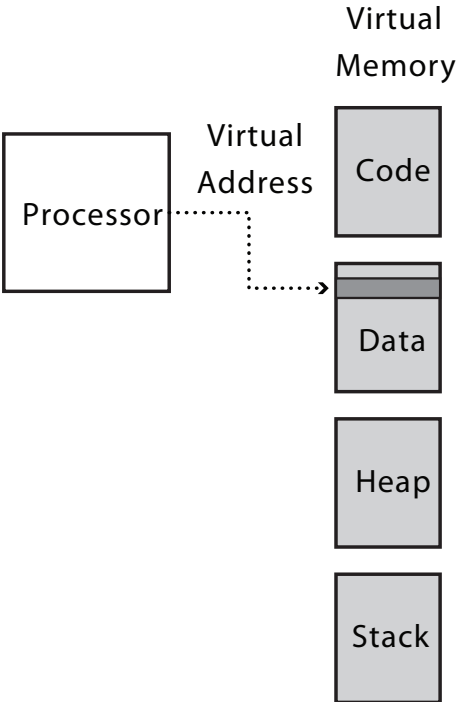# Virtually Addressed Base and Bounds

- 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
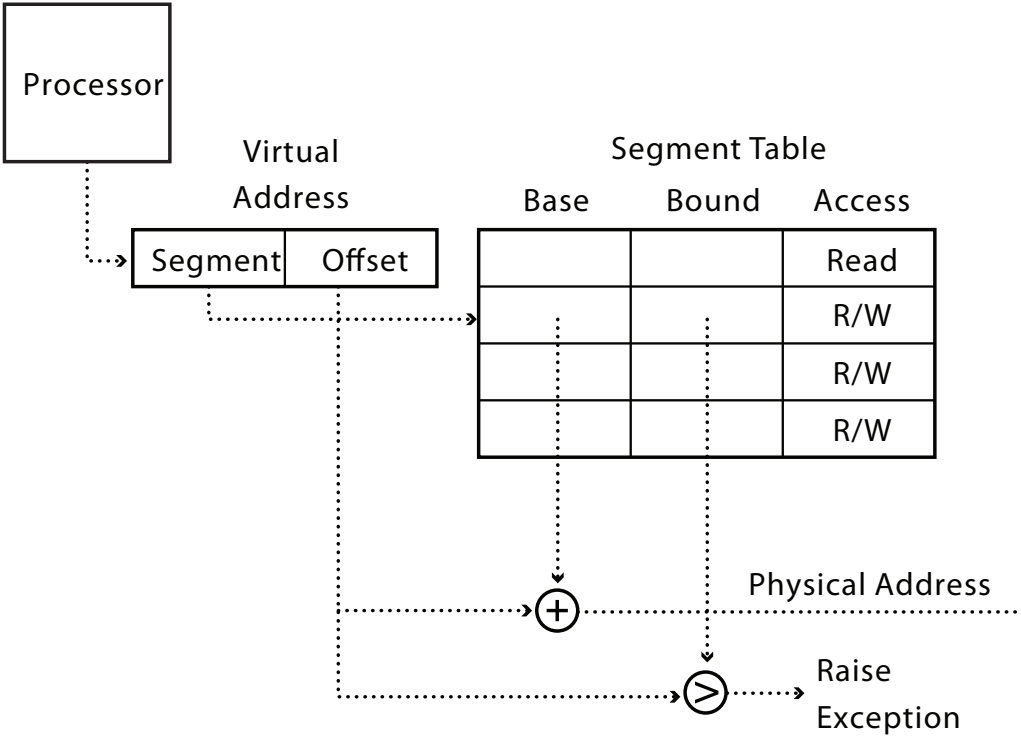 - Can't grow stack/heap as needed

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
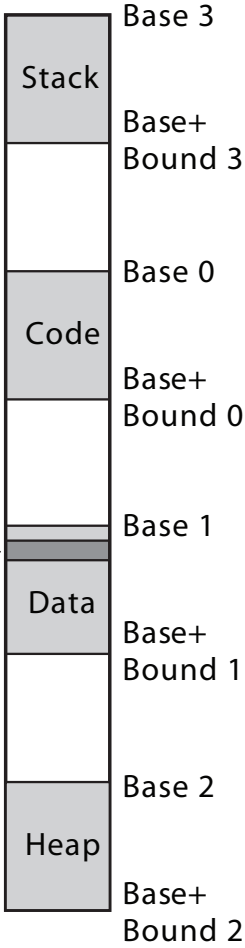 - Same start, length, same/different access permissions

Segmentation

Processor's View

Processor → Virtual Address

Virtual Memory:
- Code
- Data
- Heap
- Stack

Implementation

Processor → Virtual Address

Virtual Address: | Segment | Offset |

Segment Table

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

$+$ → Physical Address

$>$ → Raise Exception

Physical Memory

- Stack — Base 3
- Base+ Bound 3
- Base 0
- Code
- Base+ Bound 0
- Base 1
- Data
- Base+ Bound 1
- Base 2
- Heap
- Base+ Bound 2

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

2 bit segment #
12 bit offset

Virtual Memory

Physical Memory

main: 240	store #1108, r2
244	store pc+8, r31
248	jump 360
24c	
…	
strlen: 360	loadbyte (r2), r3
…	…
420	jump (r31)
…	
x: 1108	a b c \0
…	

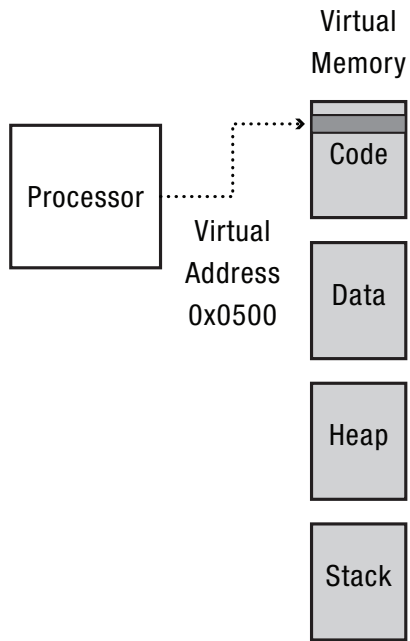|	
x: 108	a b c \0
…	
main: 4240	store #1108, r2
4244	store pc+8, r31
4248	jump 360
424c	
…	…
strlen: 4360	loadbyte (r2),r3
…	
4420	jump (r31)
…	

Question

- With segmentation, what is saved/restored on a process context switch?
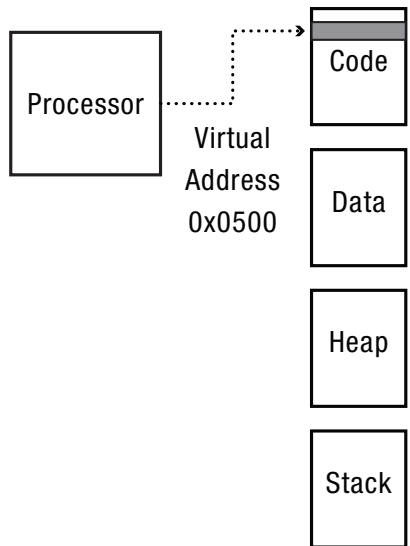
UNIX fork and Copy on Write

- UNIX fork
  - Makes a complete copy of a process
- Segments allow a more efficient implementation
  - Copy segment table into child
  - Mark parent and child segments read-only
  - Start child process; return to parent
  - If child or parent writes to a segment (ex: stack, heap)
    - trap into kernel
    - make a copy of the segment and resume
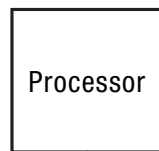
Processor's View

Process 1`s View

Processor

Virtual
Address
0x0500

Virtual
Memory

- Code
- Data
- Heap
- Stack

Process 2`s View

Processor

Virtual
Address
0x0500

- Code
- Data
- Heap
- Stack

Implementation

Processor

Seg.	Offset
0	500

Virtual
Address

Segment Table

	Base	Bound	Access
Code			Read
Data			R/W
Heap			R/W
Stack			R/W

Physical Address

$\oplus$

Processor

$\oplus$

Seg.	Offset
0	500

Virtual
Address

Segment Table

	Base	Bound	Access
Code			Read
Data			R/W
Heap			R/W
Stack			R/W

Physical
Memory

- P2`s Data
- P1`s Heap
-
- P1`s Stack
- P1`s Data
- P2`s Heap
-
- P1's+ P2`s Code
- P2`s Stack
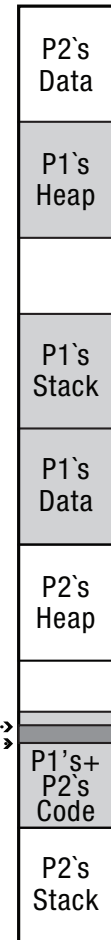
Zero-on-Reference

- How much physical memory is needed for the stack or heap?
  - Only what is currently in use
- When program uses memory beyond end of stack
  - Segmentation fault into OS kernel
  - Kernel allocates some memory
    - How much?
  - Zeros the memory
    - avoid accidentally leaking information!
 - Modify segment table
  - Resume process

Segmentation

- Pros?
  - Can share code/data segments between processes
  - Can protect code segment from being overwritten
  - Can transparently grow stack/heap as needed
  - Can detect if need to copy-on-write
- Cons?
  - Complex memory management
    - Need to find chunk of a particular size
  - May need to rearrange memory from time to time to make room for new segment or growing segment
   - 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

Paged Translation (Abstract)

Processor's View
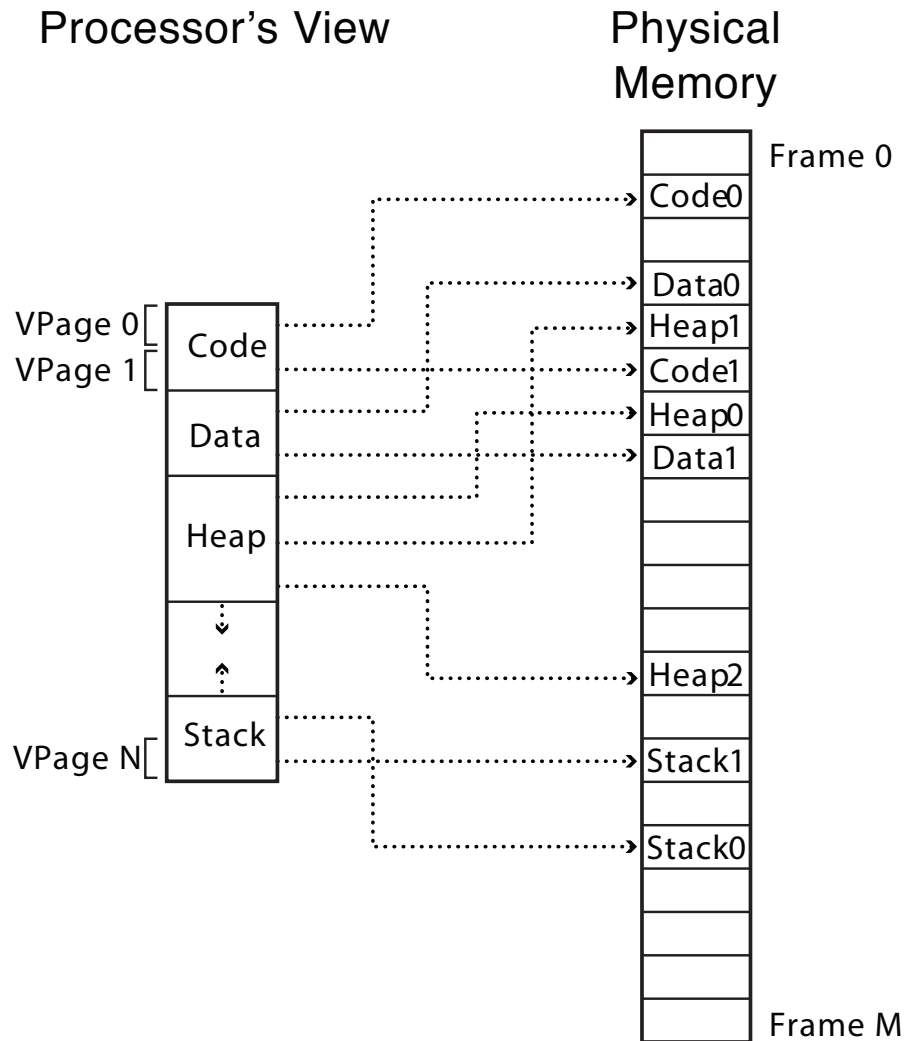
Physical Memory

Paged Translation (Implementation)
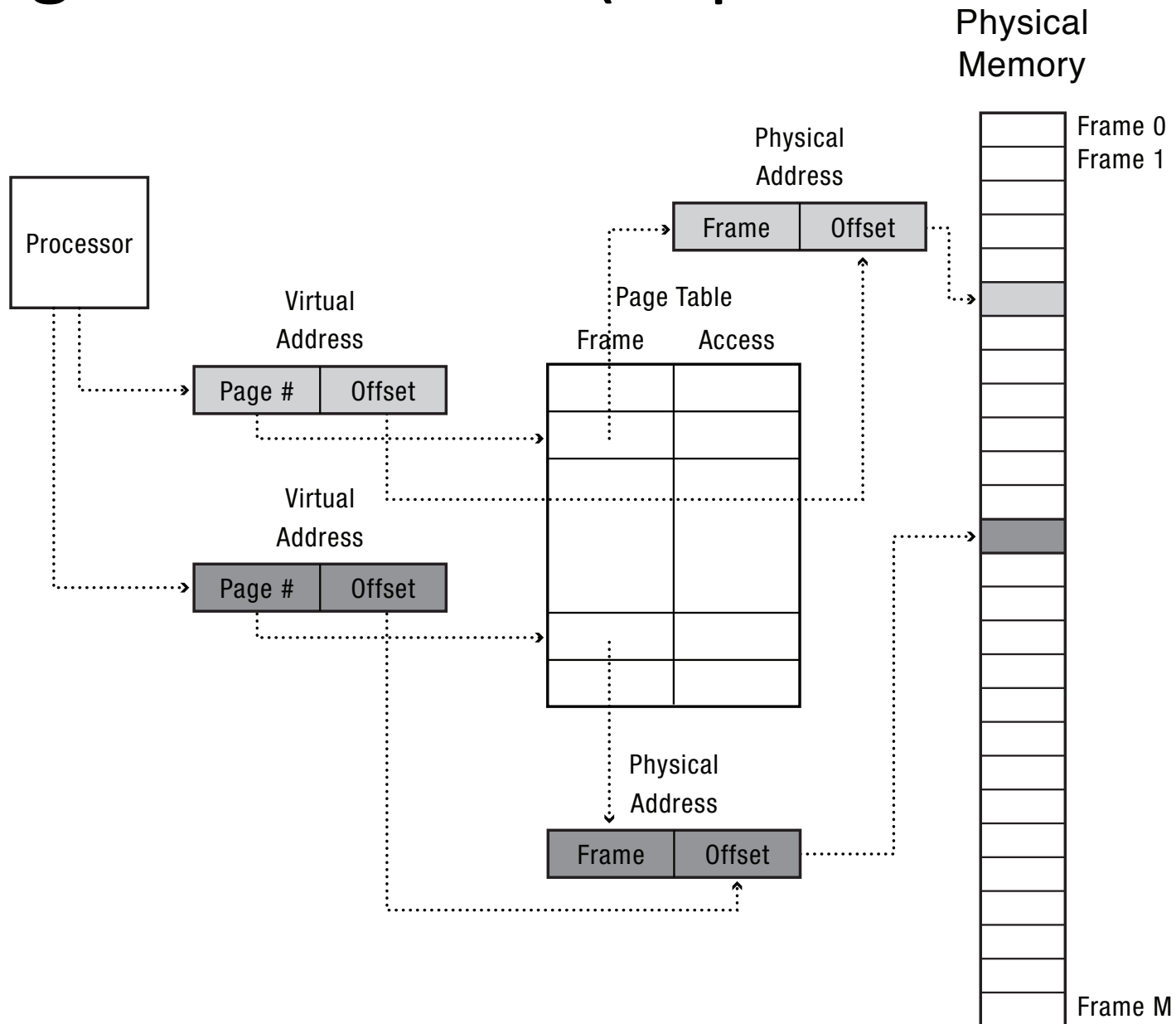
Process View

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 Copy on Write

- Can we share memory between processes?
  - Set entries in both page tables to point to same page frames
  - Need *core map* of page frames to track which processes are pointing to which page frames (e.g., reference count)
- 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

Fill On Demand

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

Sparse Address Spaces

- Might want many separate dynamic 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
- Fixed-size page as lowest level unit of allocation
  - Efficient memory allocation (compared to segments)
  - Efficient for sparse addresses (compared to paging)
  - Efficient disk transfers (fixed size units)
  - Easier to build translation lookaside buffers
  - Efficient reverse lookup (from physical -> virtual)
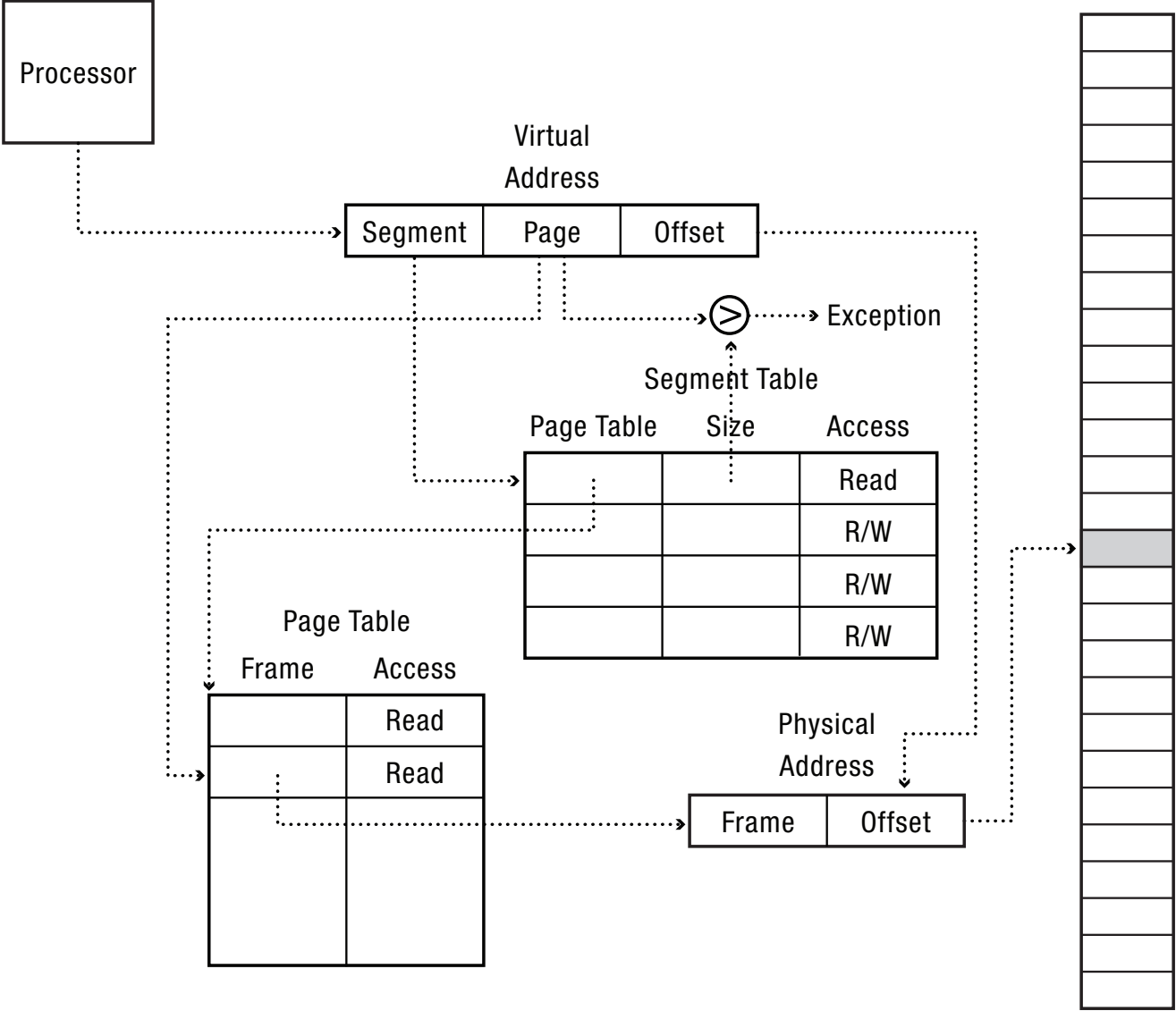 - Variable granularity for protection/sharing

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/protection at either page or segment-level

Paged Segmentation (Implementation)
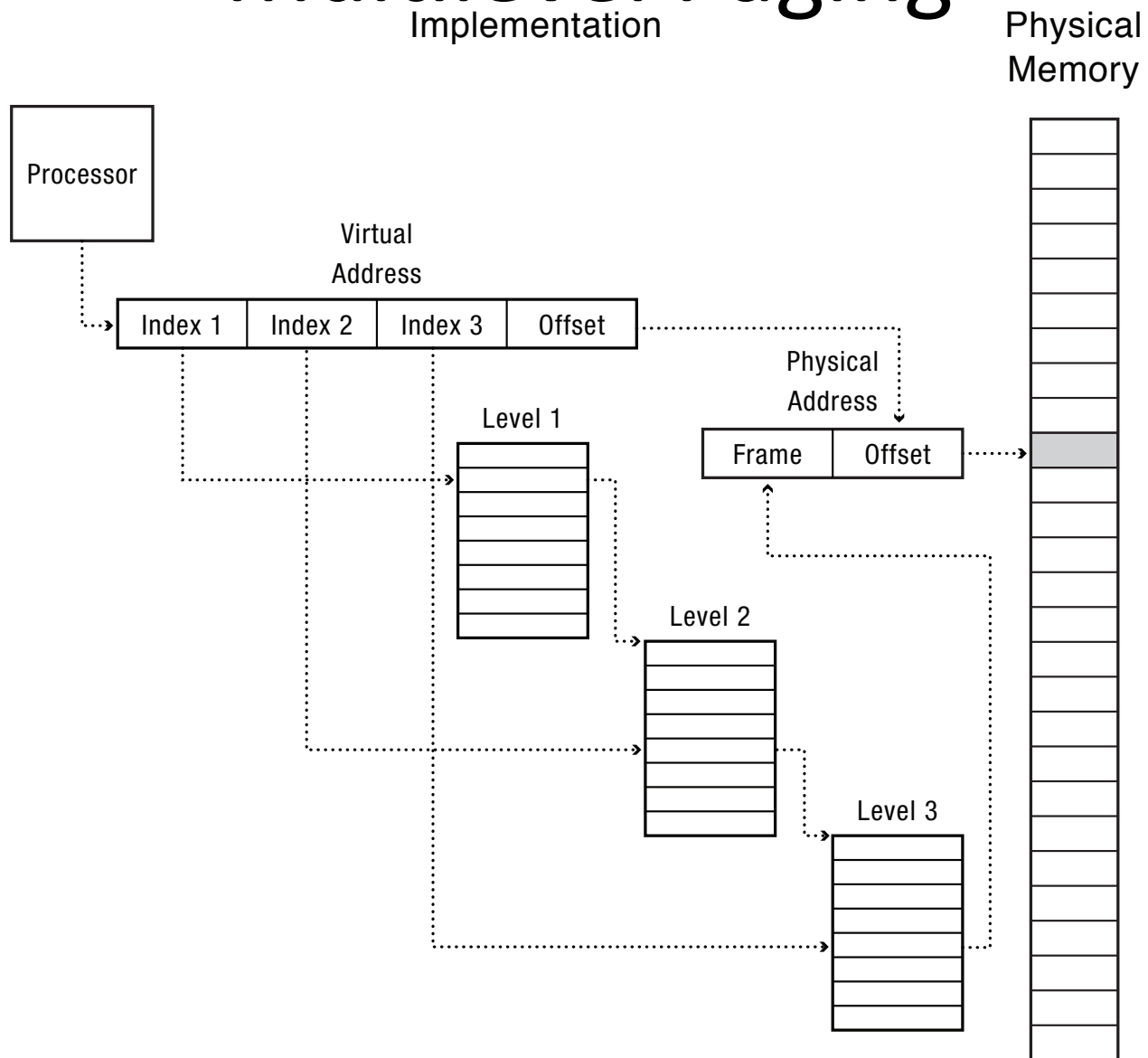
Implementation

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?

Multilevel Paging

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.

x86 Multilevel Paged Segmentation

- Global Descriptor Table (segment table)
  - Pointer to page table for each segment
  - Segment length
  - Segment access permissions
  - Context switch: change global descriptor table register (GDTR, pointer to global descriptor table)
- Multilevel page table
  - 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)
  - Omit sub-tree if no valid addresses

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
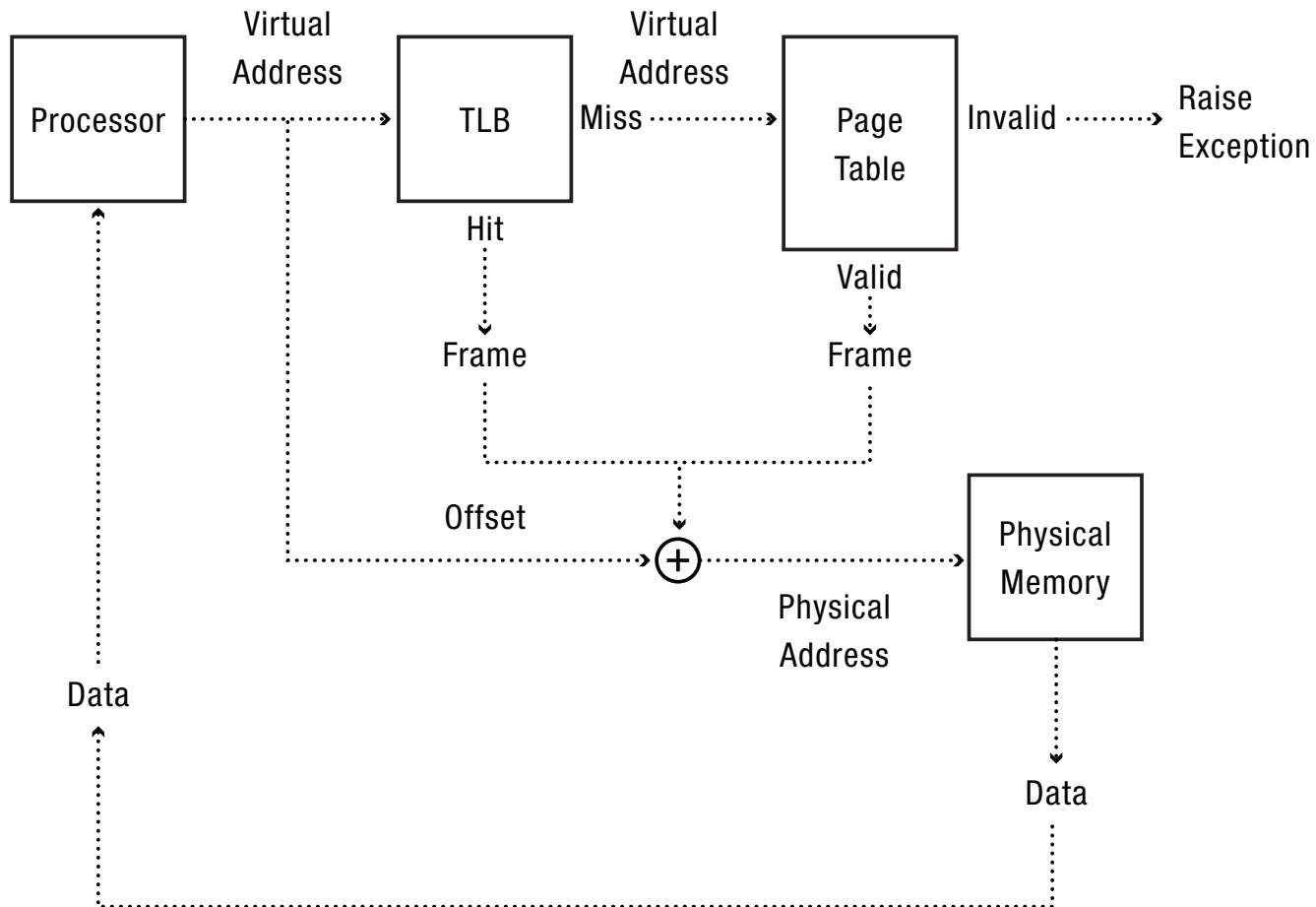 - Two (or more) lookups per memory reference

Portability

- Many operating systems keep their own memory translation data structures
  - List of memory objects (segments)
  - Virtual page -> physical page frame
  - Physical page frame -> set of virtual pages
- One approach: Inverted page table
  - Hash from virtual page -> physical page
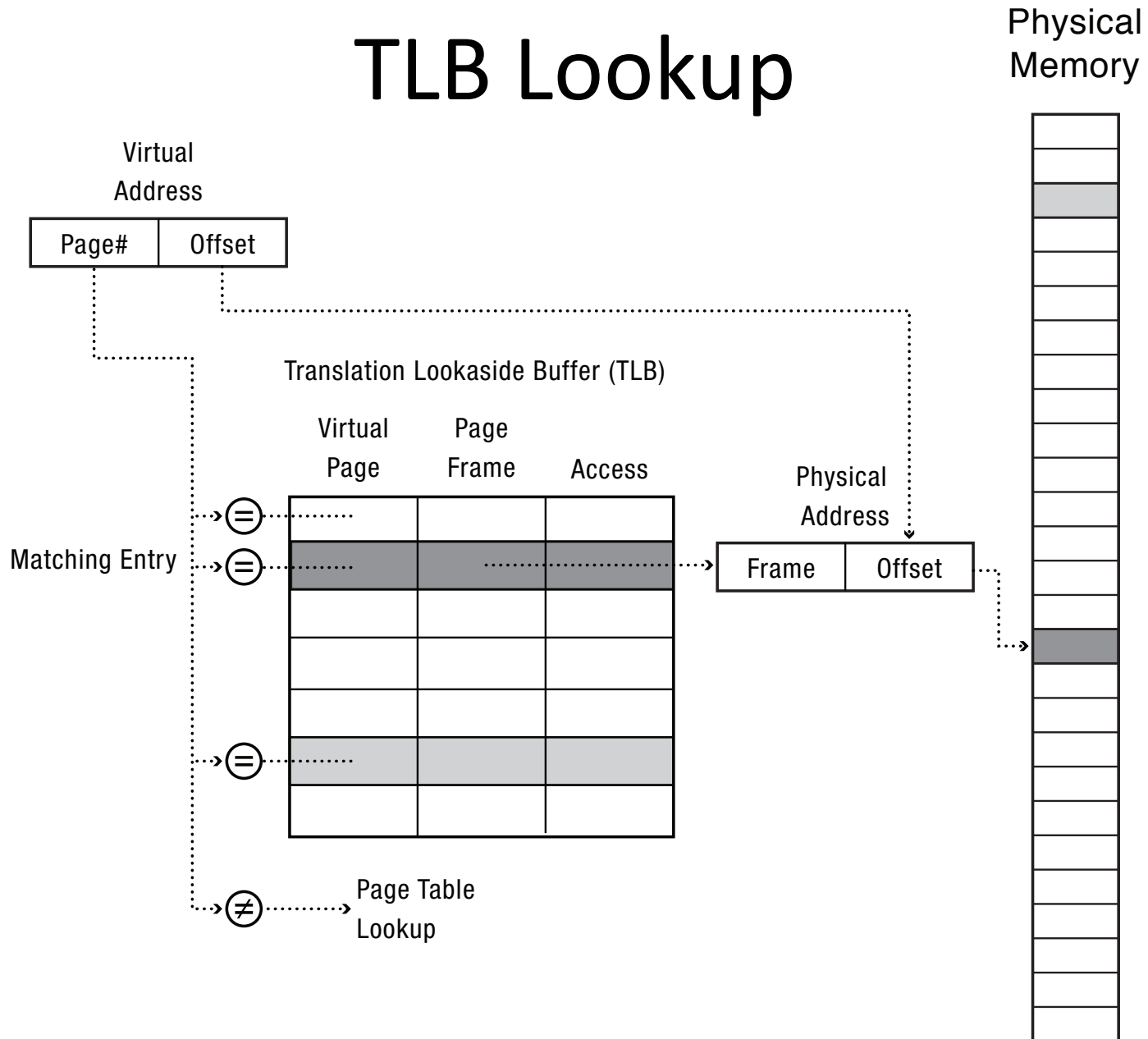 - Space proportional to # of physical pages

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

	Virtual Address		Virtual Address		
Processor	→	TLB	Miss →	Page Table	Invalid ⟶ Raise Exception

Hit

Frame

Valid

Frame

Offset

⊕

Physical
Memory

Physical
Address

Data

Data

TLB Lookup

Virtual
Address

Page#	Offset

Physical
Memory

Translation Lookaside Buffer (TLB)

Virtual Page	Page Frame	Access

Matching Entry

Physical
Address

Frame	Offset

Page Table
Lookup

MIPS Software Loaded TLB

- Software defined translation tables
  - If translation is in TLB, ok
  - If translation is not in TLB, trap to kernel
  - Kernel computes translation and loads TLB
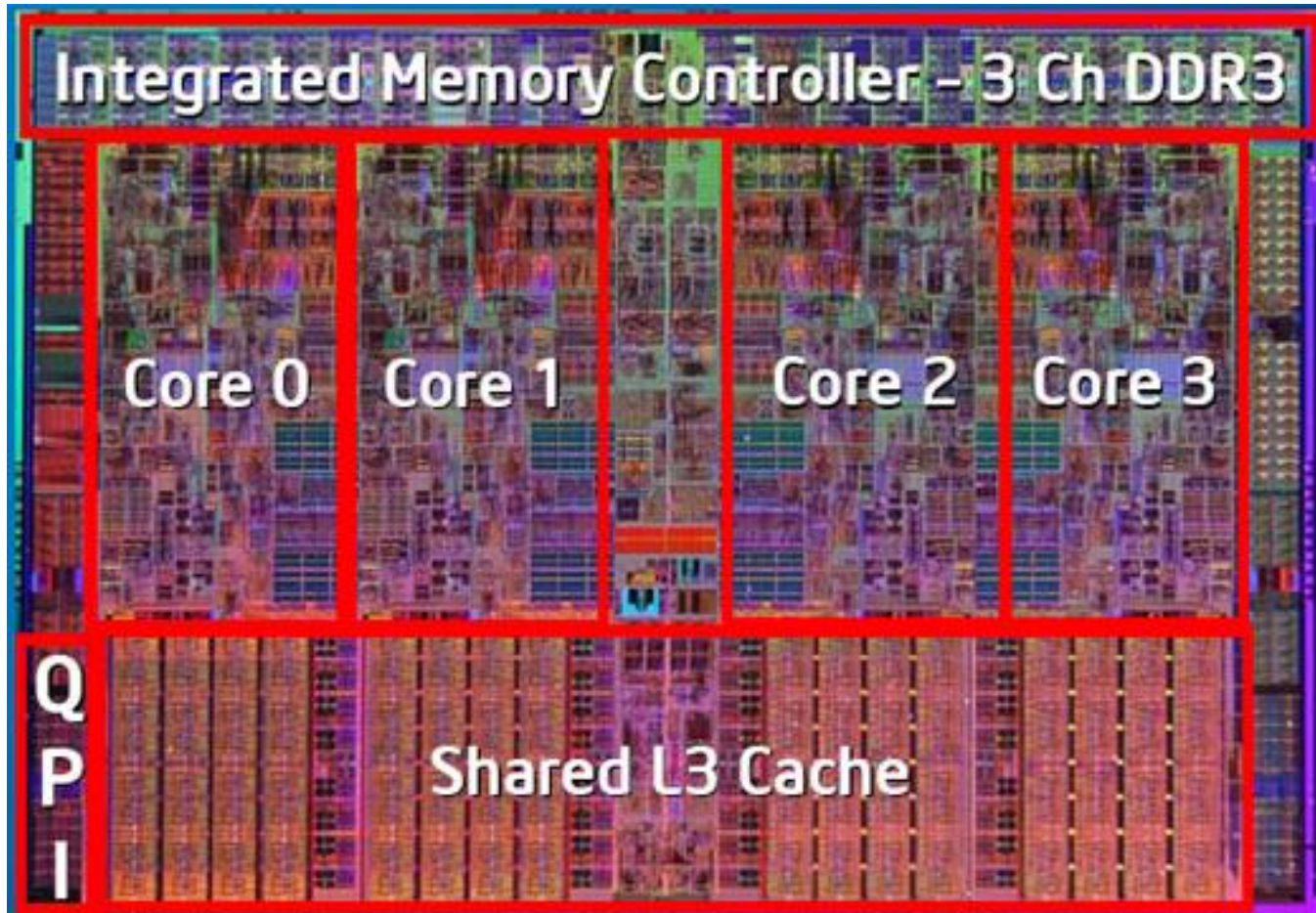 - Kernel can use whatever data structures it wants
- Pros/cons?

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

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?
  - x86: 2-4 level page table walk
- How expensive is a 4-level page table walk on a modern processor?

Virtually Addressed vs. Physically Addressed Caches

- 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, access TLB to generate physical address in case of a cache miss

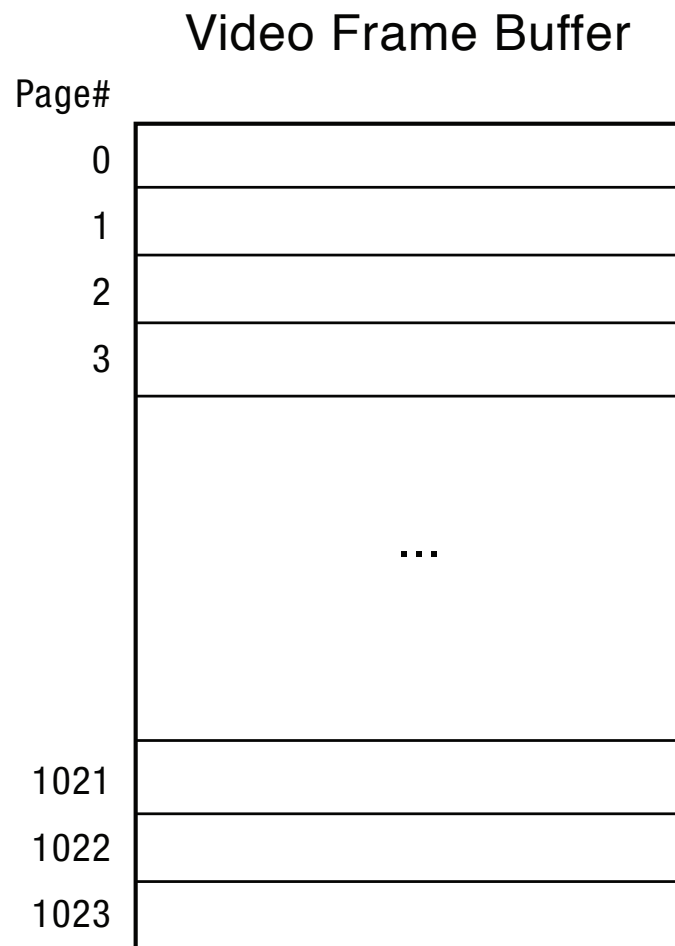# Virtually Addressed Caches
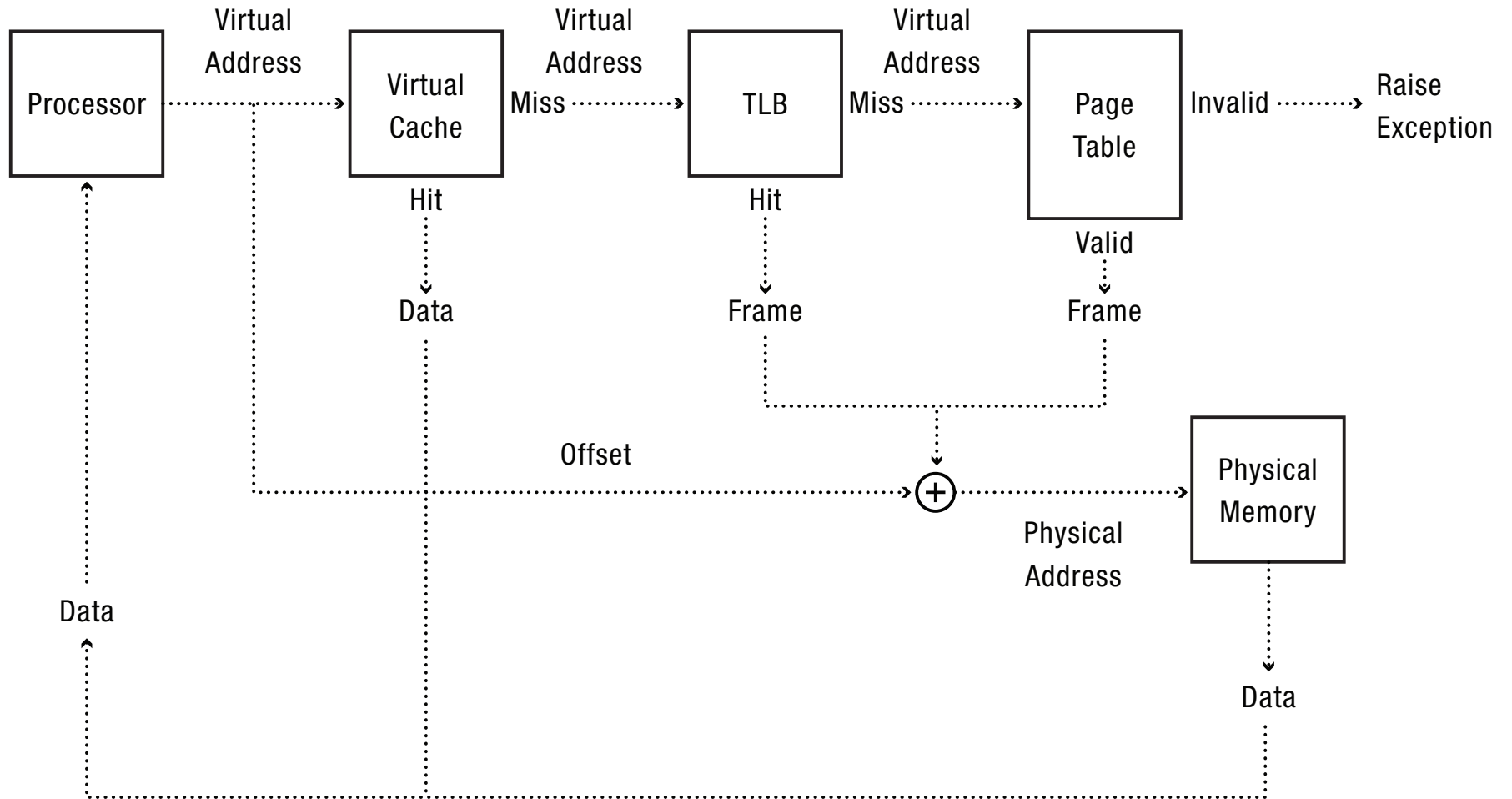
Physically Addressed Cache
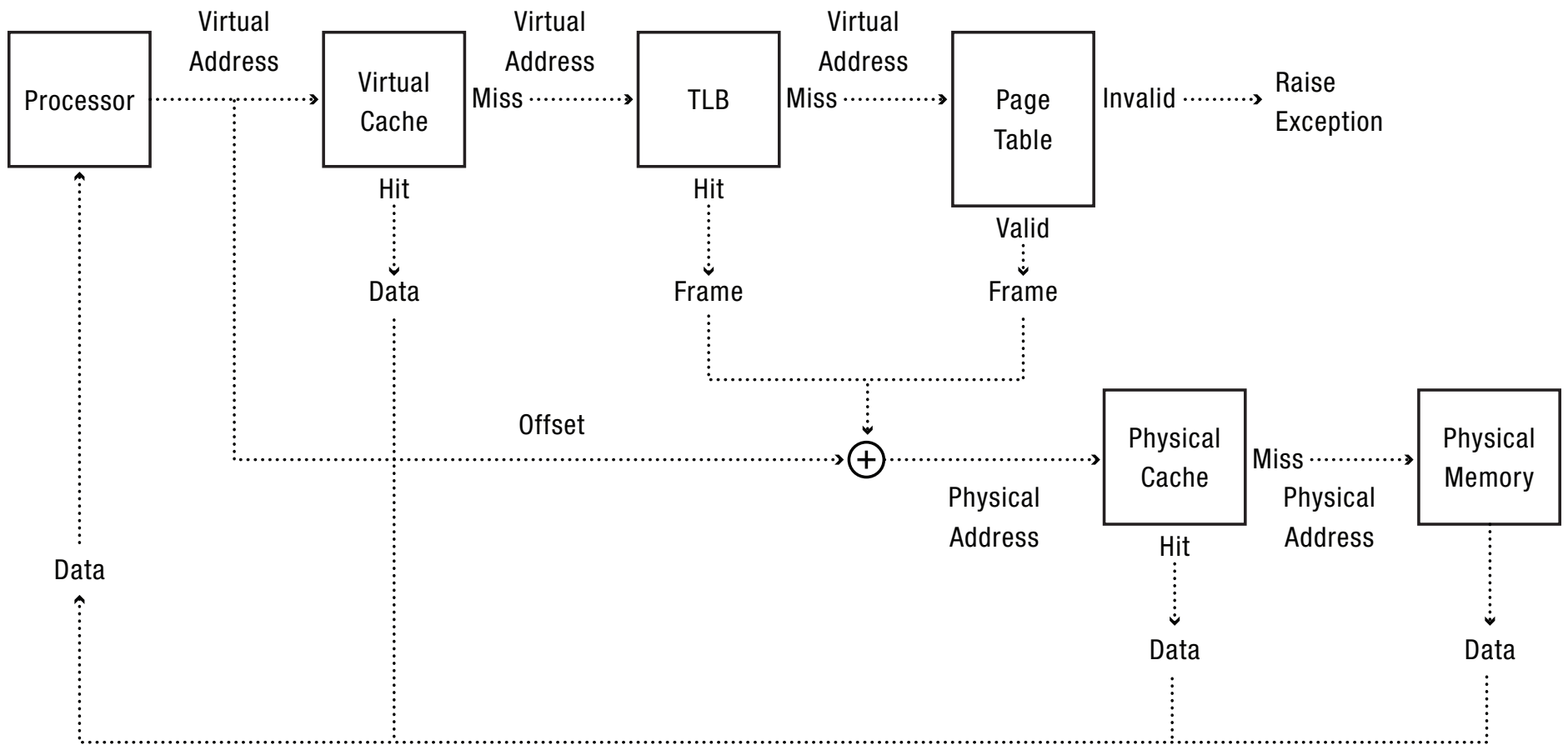
When Do TLBs Work/Not Work?

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

Video Frame Buffer

Page#	
0	
1	
2	
3	
	...
1021	
1022	
1023	

Superpages

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

Superpages

Virtual Address

Page#	Offset
SP	Offset

Physical Memory

Translation Lookaside Buffer (TLB)

Superpage (SP) or Page#	Superframe (SF) or Frame	Access

Matching Entry

Matching Superpage

Physical Address

Frame	Offset

SF	Offset

Page Table Lookup
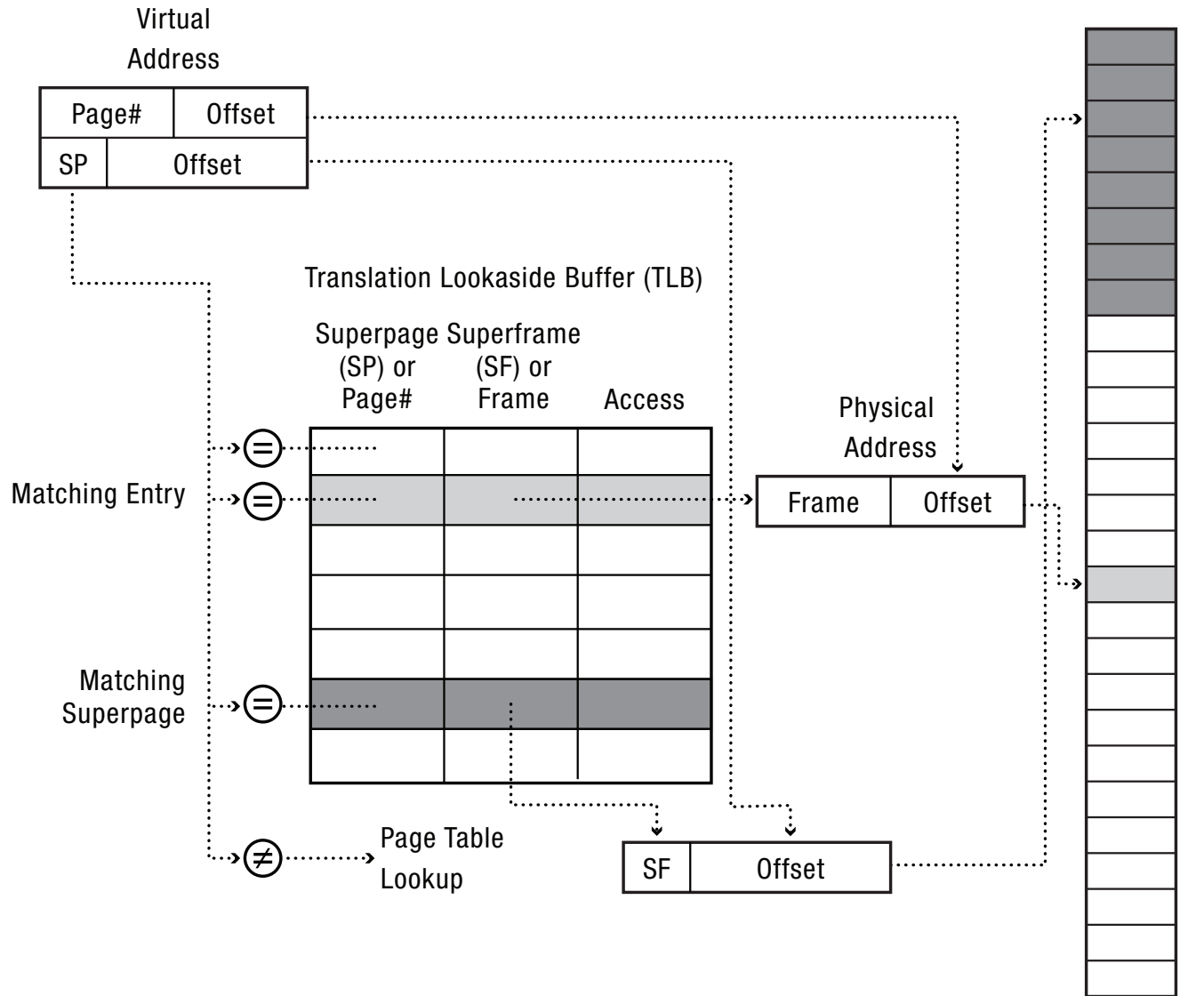
When Do TLBs Work/Not Work, part 2

- 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
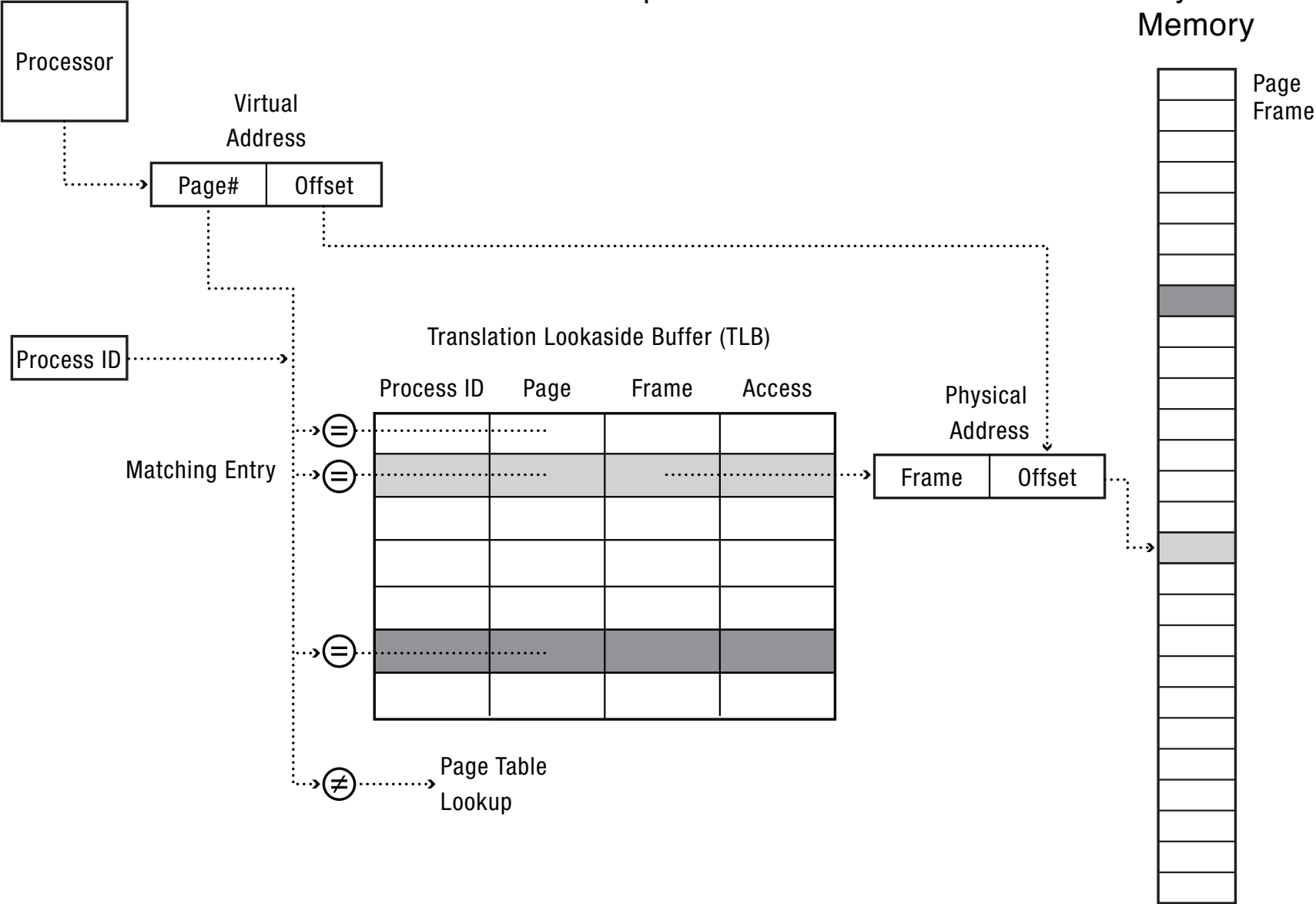 - OS must ask each CPU to purge TLB entry

TLB Shootdown

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

When Do TLBs Work/Not Work, part 3

- What happens on a context switch?
  - Reuse TLB?
  - Discard 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

Process ID

Translation Lookaside Buffer (TLB)

Process ID	Page	Frame	Access

Matching Entry

=

=

=

≠ ········ Page Table Lookup

Physical
Address

Frame	Offset

Page
Frame

Question

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

Aliasing

- 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
- Typical solution
  - Keep both virtual and physical address for each entry in virtually addressed cache
  - Lookup virtually addressed cache and TLB in parallel
  - Check if physical address from TLB matches multiple entries, and update/invalidate other copies

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