Address Translation

Instructor: Youngjin Kwon

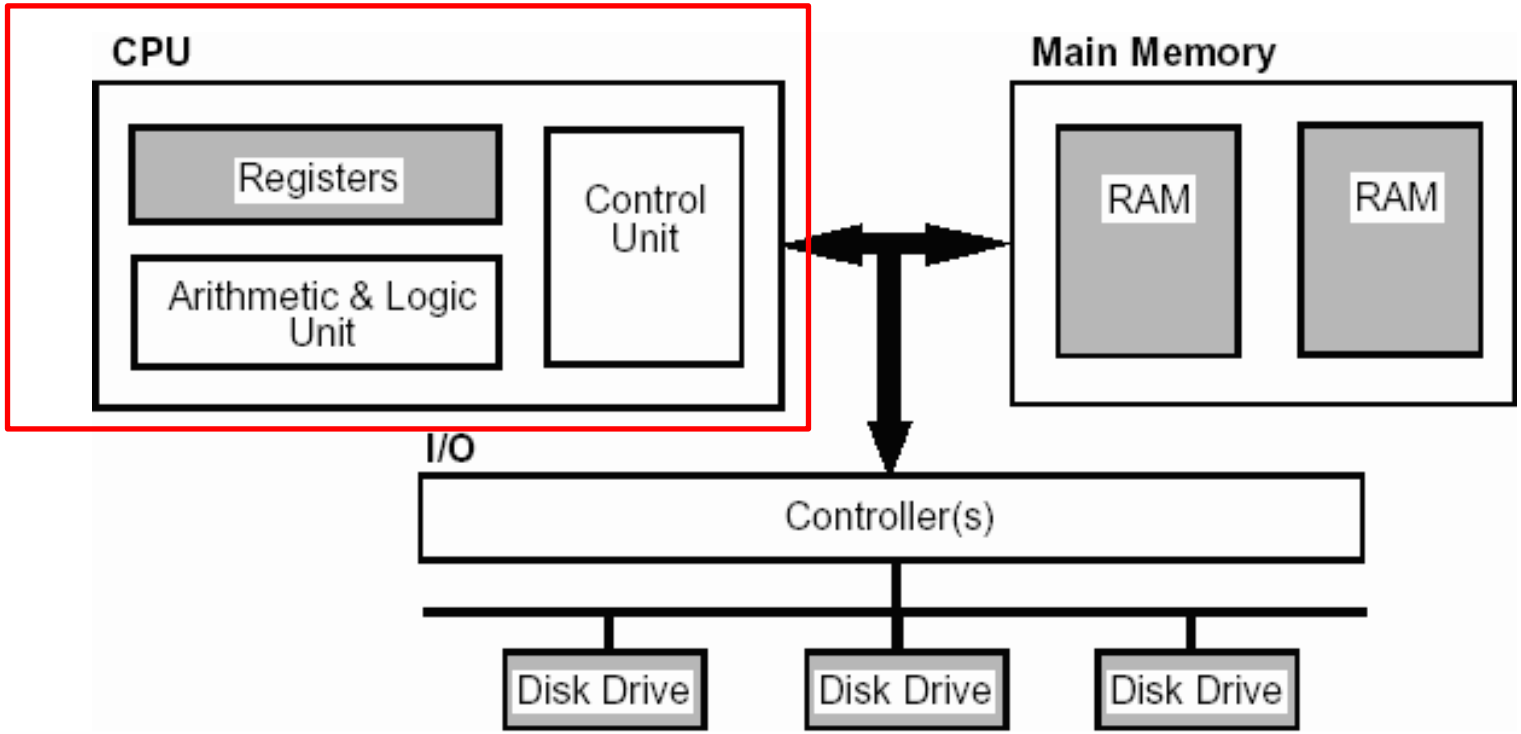
Abstraction, Mechanism, Policy

- Abstraction
  - Thread as an execution unit
- Mechanism
  - Per-processor ready queue, context switch, work stealing, load balancing …
- Policy
  - FIFO, SJF, Round robin, MFQ …

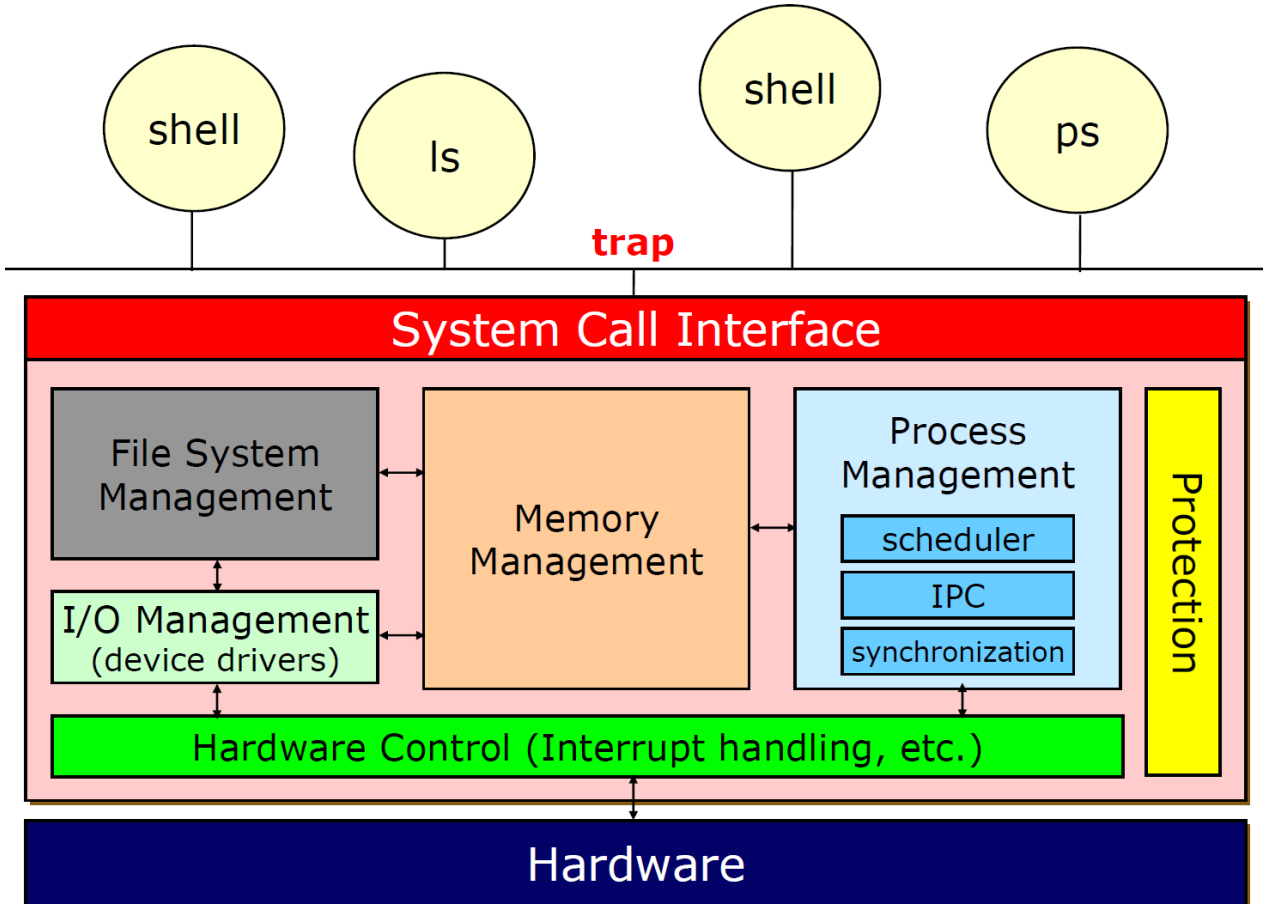
What we've done so far (a resource-oriented view)

Abstractions:

Mechanism and policy for concurrency:



What we've done so far



A new abstraction for memory

Exposing physical memory to processes directly is:

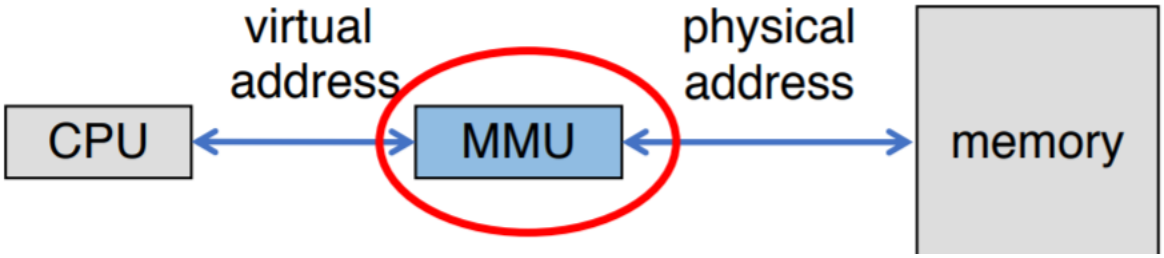
Dangerous!

Making programming hard!

Design a new abstraction (called virtual address) for:

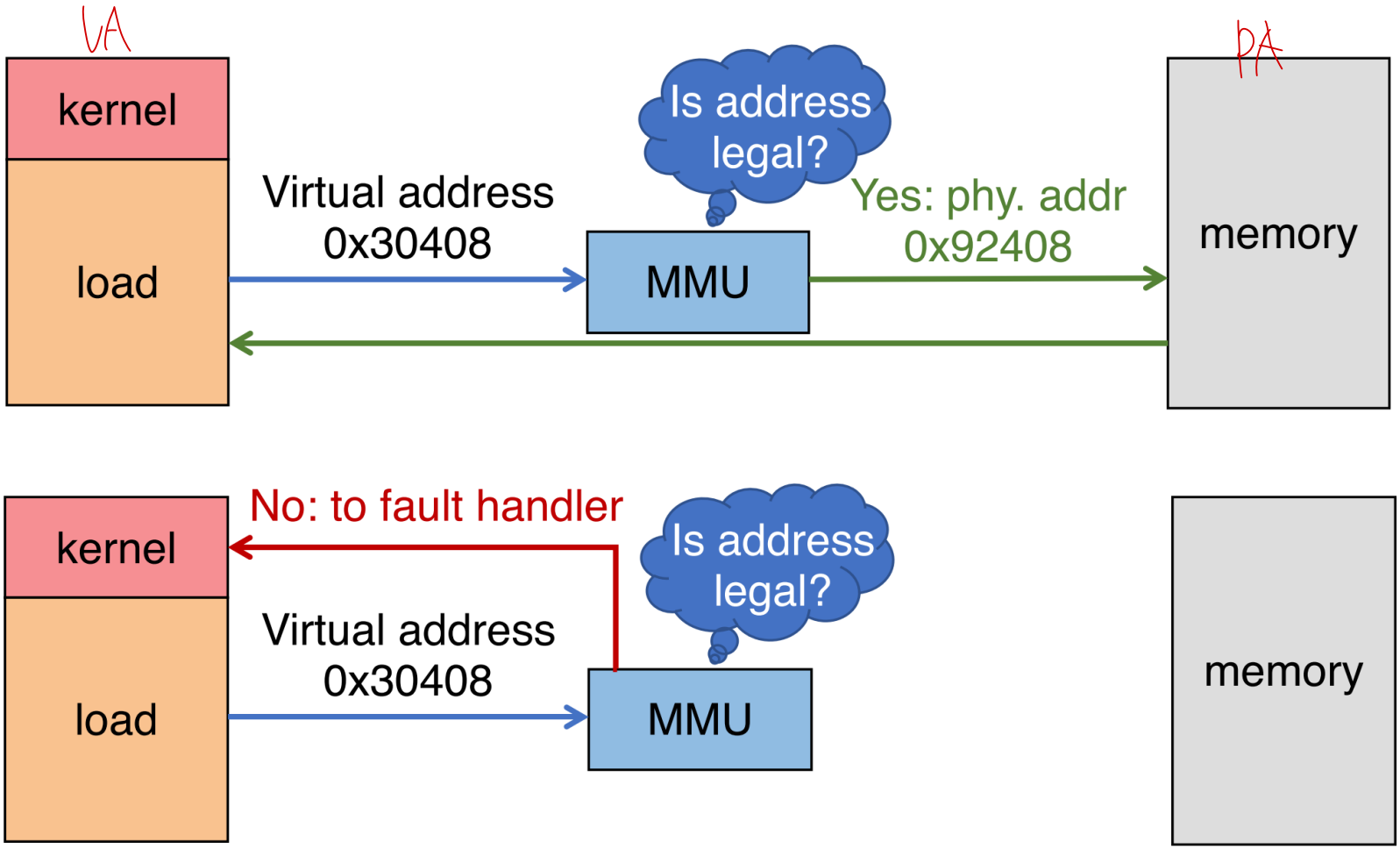
Safety!

Convenience and efficiency!



Address Translation Concept

At runtime, Memory-Management Unit (MMU) relocates each load/store



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
  - Superpages

Design Goals of Address Translation

- Memory protection
  - Prevent one app from messing with another's memory
- Memory sharing
  - Shared libraries, interprocess communication
- Sparse addresses
  - Multiple regions changing size dynamically (heaps/stacks)
- Efficiency: having less cost for translations
  - Memory placement
  - Runtime lookup
  - Compact translation tables

Bonus Features

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

How to archive design goals?

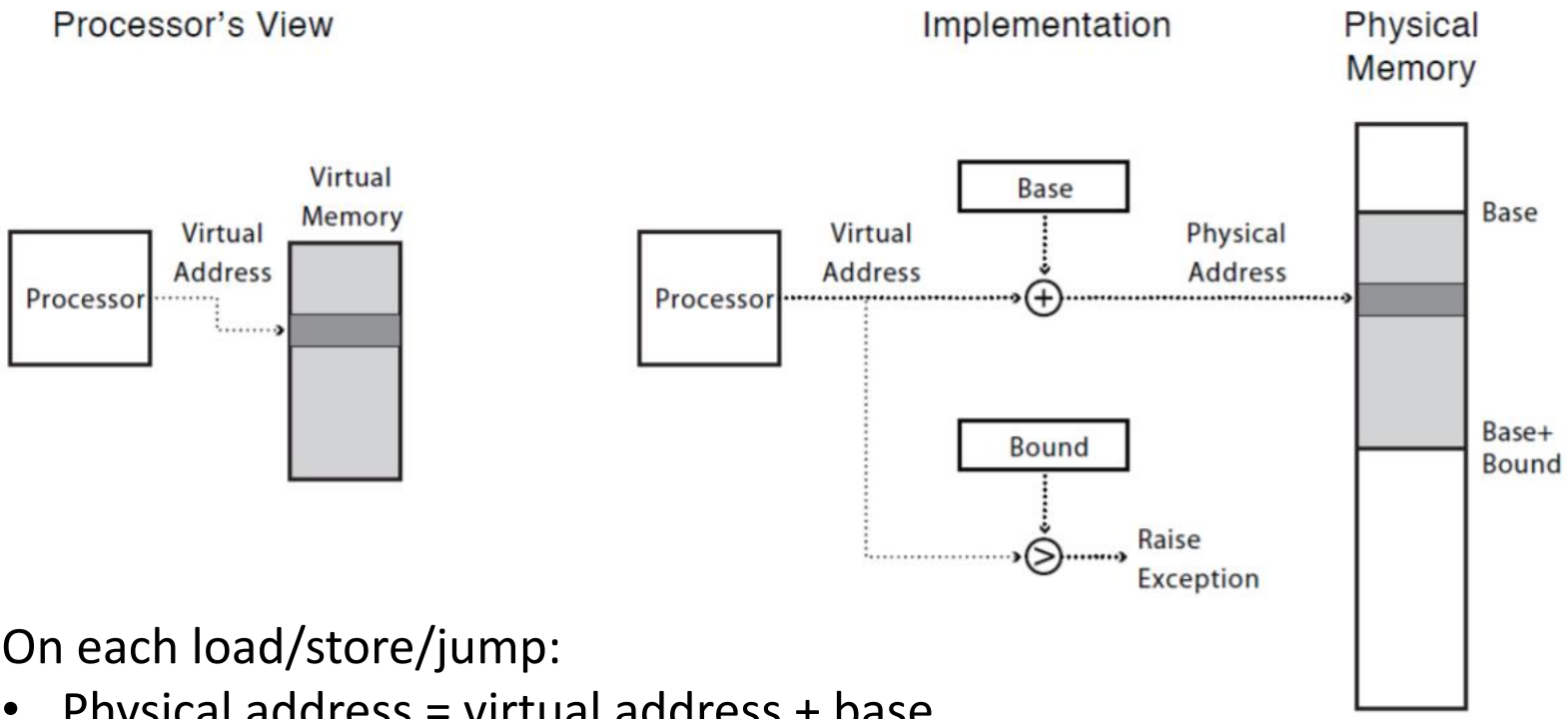
- Memory protection
  - Hardware (MMU) authorizes memory accesses
- Memory sharing
  - By shared mapping
- Sparse addresses
  - By changing mapping (size, base address, adding new mappings etc)
- Efficiency

All the above questions depend on "mapping mechanisms"

Mapping mechanisms

- Base and Bounds
- Segmentation
- Paging

Virtually Addressed Base and Bounds



On each load/store/jump:

- Physical address = virtual address + base
- Check 0 ≤ virtual address < bound, else trap to kernel

- What happens on context switch? Reset values of base & bound

Virtually Addressed Base and Bounds

- Pros?

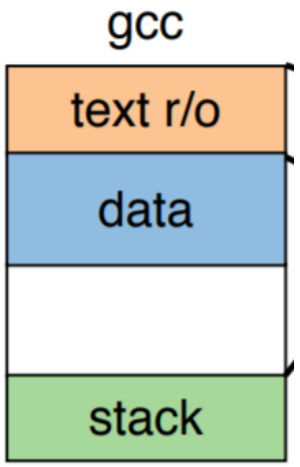
Simple design, Context switch cost ↓

- Cons?

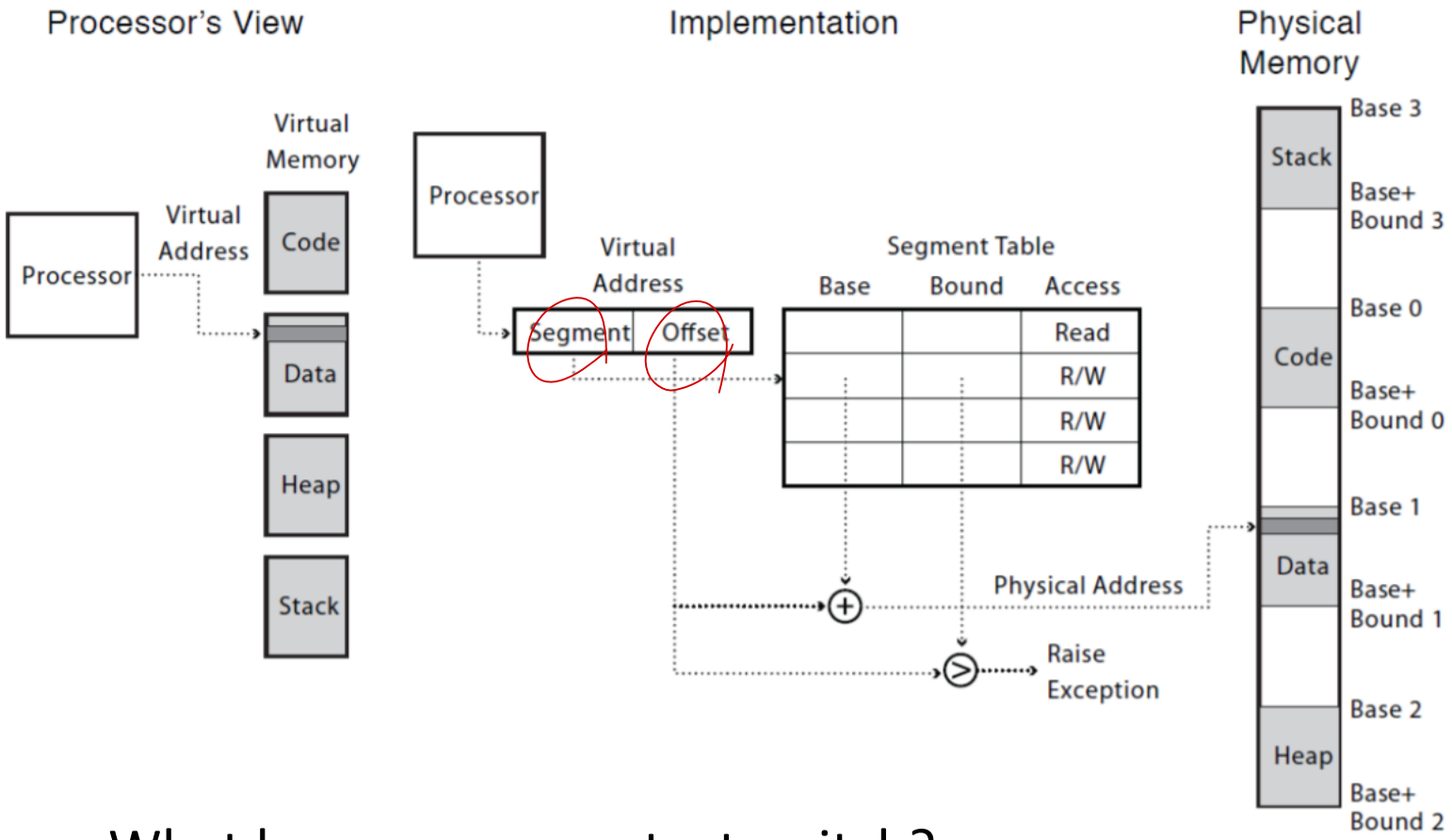
Sparse address advantages X

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

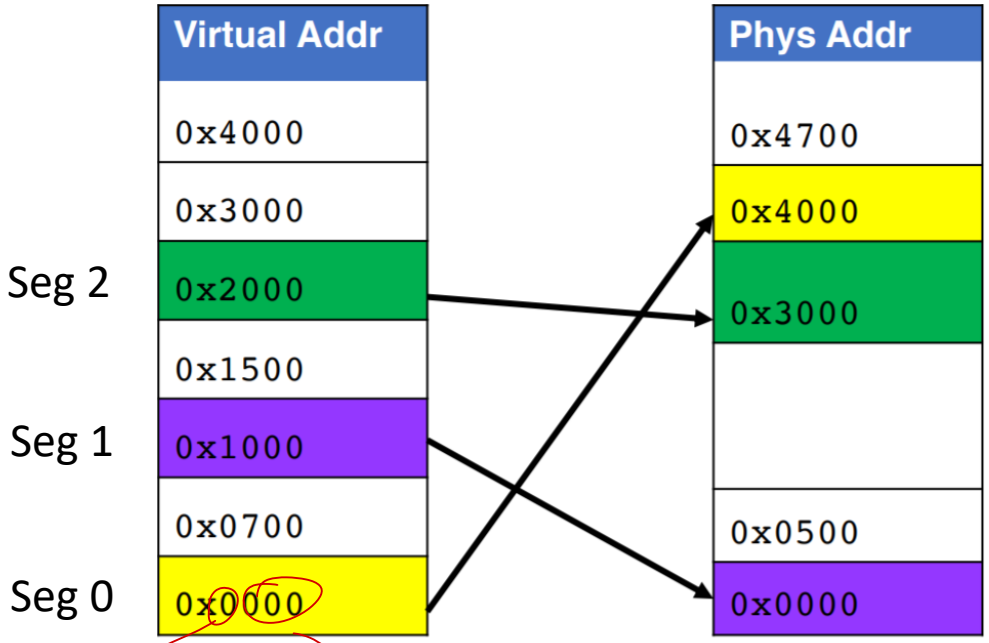


- What happens on context switch?

Reset Segment table → cost ↑

Segmentation Example

Segment	Base	Bound	RW
0	0x4000	0x6ff	10
1	0x0000	0x4ff	11
2	0x3000	0xfff	11
3			00



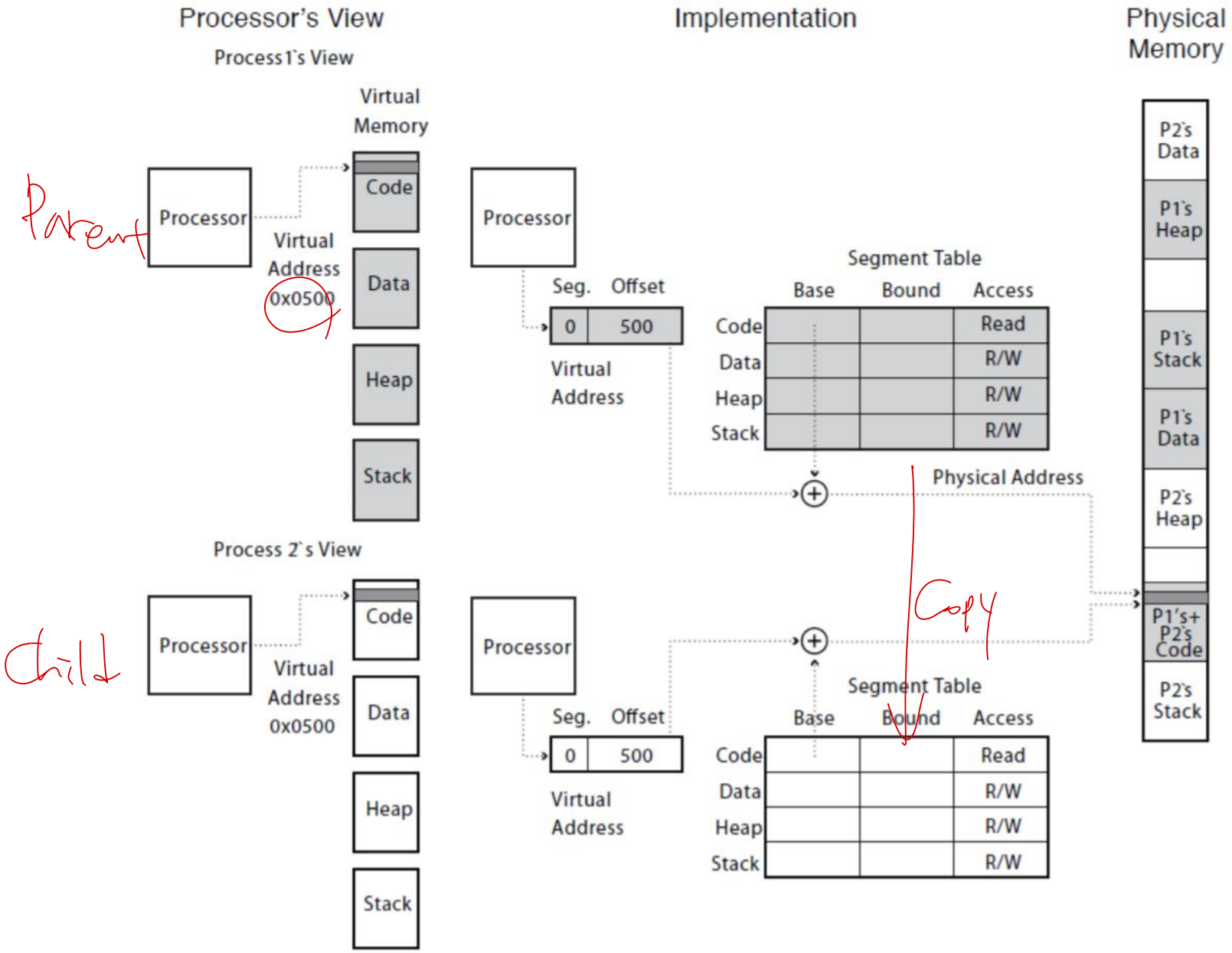
- 2-bit segment number (1st digit), 12 bit offset (last 3)

How to translate virtual address of 0x0240? 0x265c? 0x3002?

0x4240 0x365C Fault

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
Implementation
Physical Memory
Process1's View
Virtual Memory
Processor
Virtual Address 0x0500
Code
Data
Heap
Stack
Parent
Segment Table
Base
Bound
Access
Seg.
Offset
0
500
Virtual Address
Code
Data
Heap
Stack
Read
R/W
R/W
R/W
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
Child
Copy

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
  - Modify segment table
  - Resume process

Segmentation

- Pros?

  No Internal frag (?)

  Efficiently separate Code, Data, Heap, Stack part.

  Memory extension possible

- Cons?

  External fragmentation → due to different size alloc

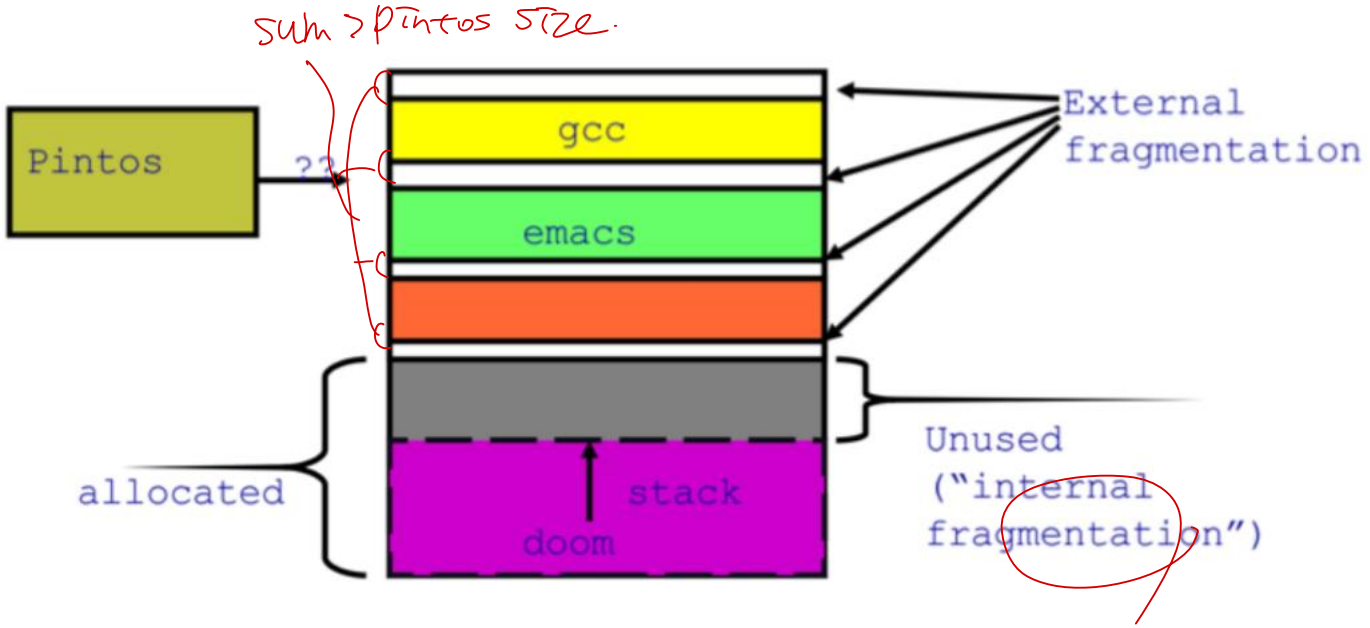
  Context switch cost ↑

  Need 2 accesses to get data (Seg table access + Memory)

Two Types of Fragmentation

- Fragmentation → Inability to use free memory

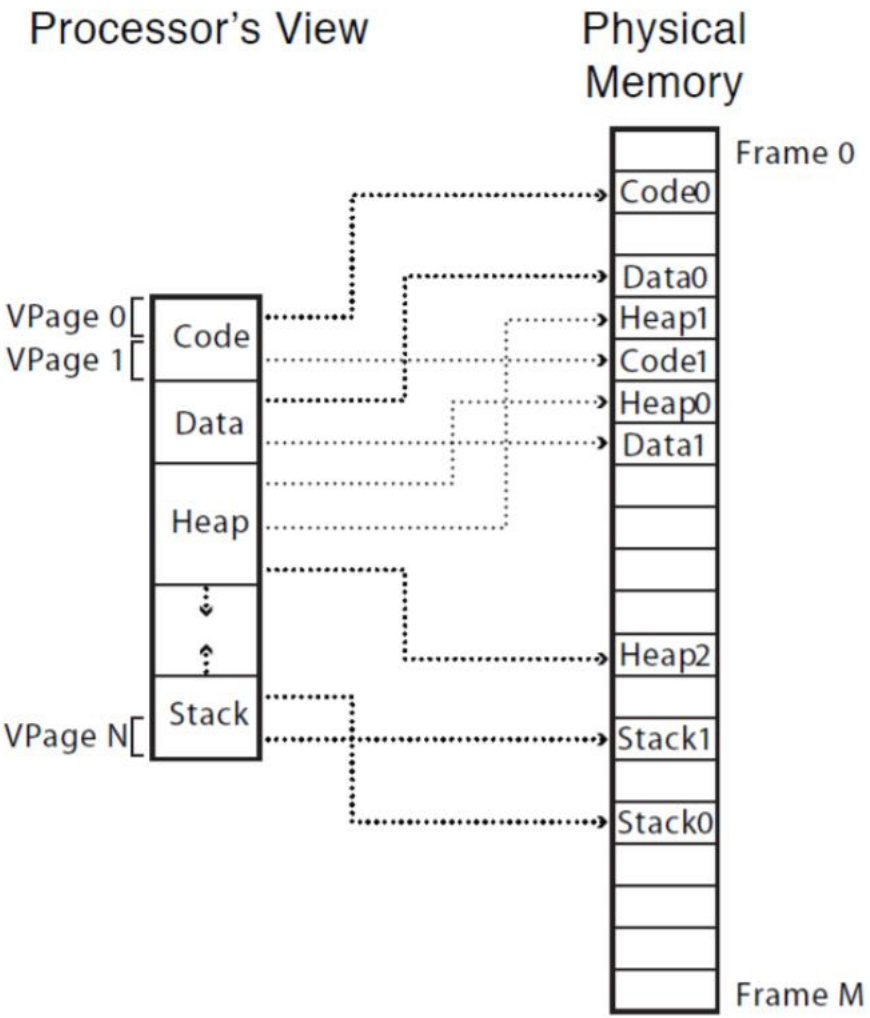
External fragmentation: Variable-sized pieces = many small holes
Internal fragmentation: Fixed-sized pieces = no external holes, but force internal waste



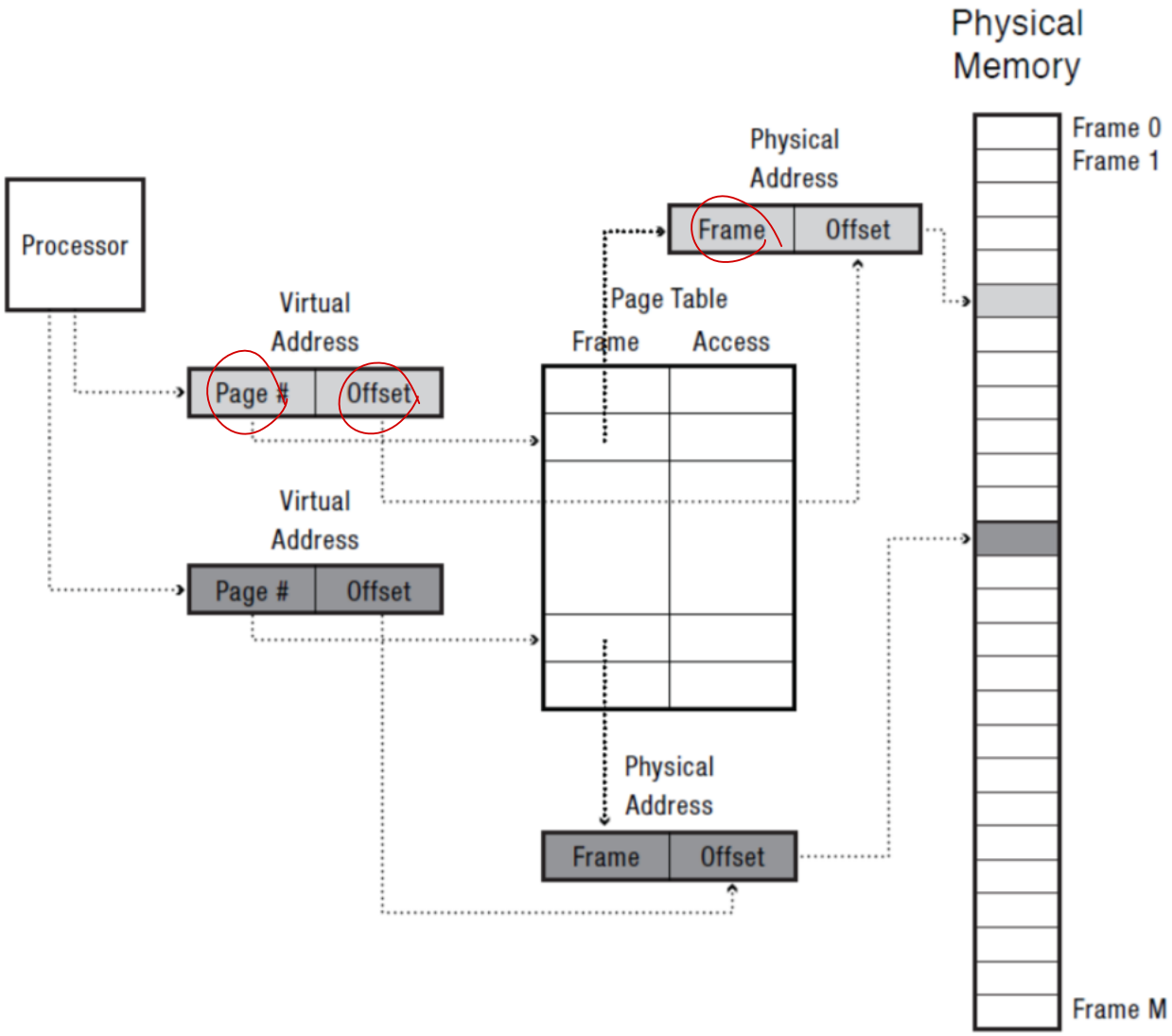
Paged Translation

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

Paged Translation (Abstract)



Paged Translation (Implementation)



Paging example

- Pages are 4KB
  - VPN is 20 bits (2^20 VPNs), offset is 12 bits
- Virtual address is 0x7468
  - Virtual page is 0x7, offset is 0x468
- Page table entry 0x7 contains 0x2
  - Physical page number is 0x2

What is Physical address?

Physical address = 0x2000 + 0x468 = 0x2468

Paging Questions

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

Page table pointer, length, PTEs

- What if page size is very small?

Page fault↑, page demanding overhead↑

- What if page size is very large?

Internal fragmentation↑

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?
  - Page table size is too big → Memory disadv
- 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
  - PG fault into kernel on write (in child or parent)
  - Copy page
  - Mark both as writeable
  - Resume execution

Sparse Address Spaces

- Paging makes dynamic mappings easy
  - Per-process heaps
  - Per-thread stacks
  - Memory-mapped files
  - Dynamically linked libraries
- What if virtual address space is large?
  - 32-bits, 4KB pages => 2^20 page table entries
  - 64-bits, 4KB pages => 2^52 page table entries

↳ 12 offs

How to solve the memory overhead?

A simple data structure question

- Storing numbers
  - Dense space: e.g., all numbers from 0 to 100000
    → Array is enough: No waste of space
  - Sparse space: e.g., 0, 4, 10000, 99998
    → What data structures are preferred?

hash?

Multi-level Translation

- **Tree** of translation tables
  - Paged segmentation
  - Multi-level page tables

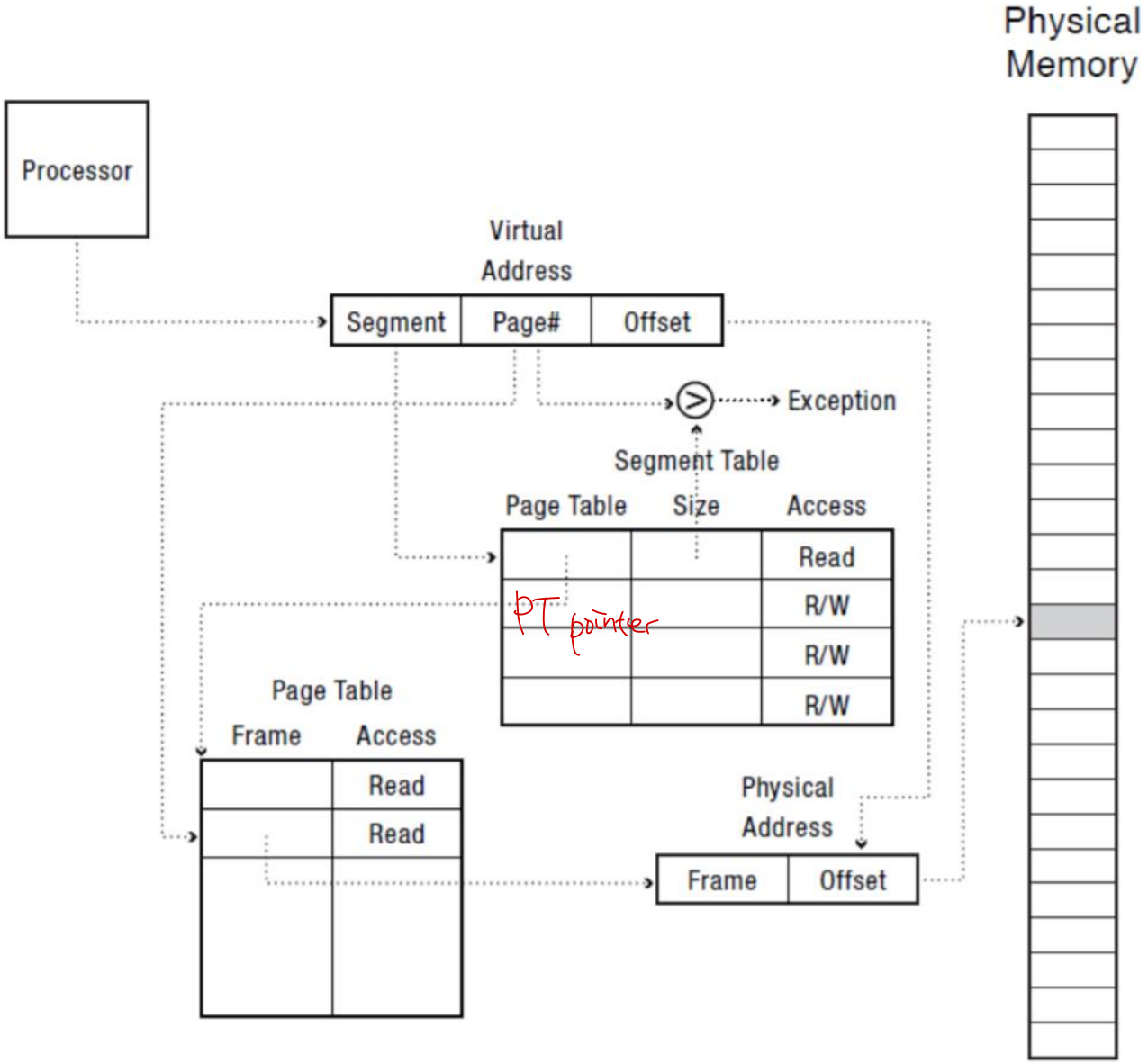
Paged Segmentation

- 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

⊕PGSIZE↓

⊖3 memory accesses

Paged Segmentation (Implementation)



Questions: Paged Segmentation

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

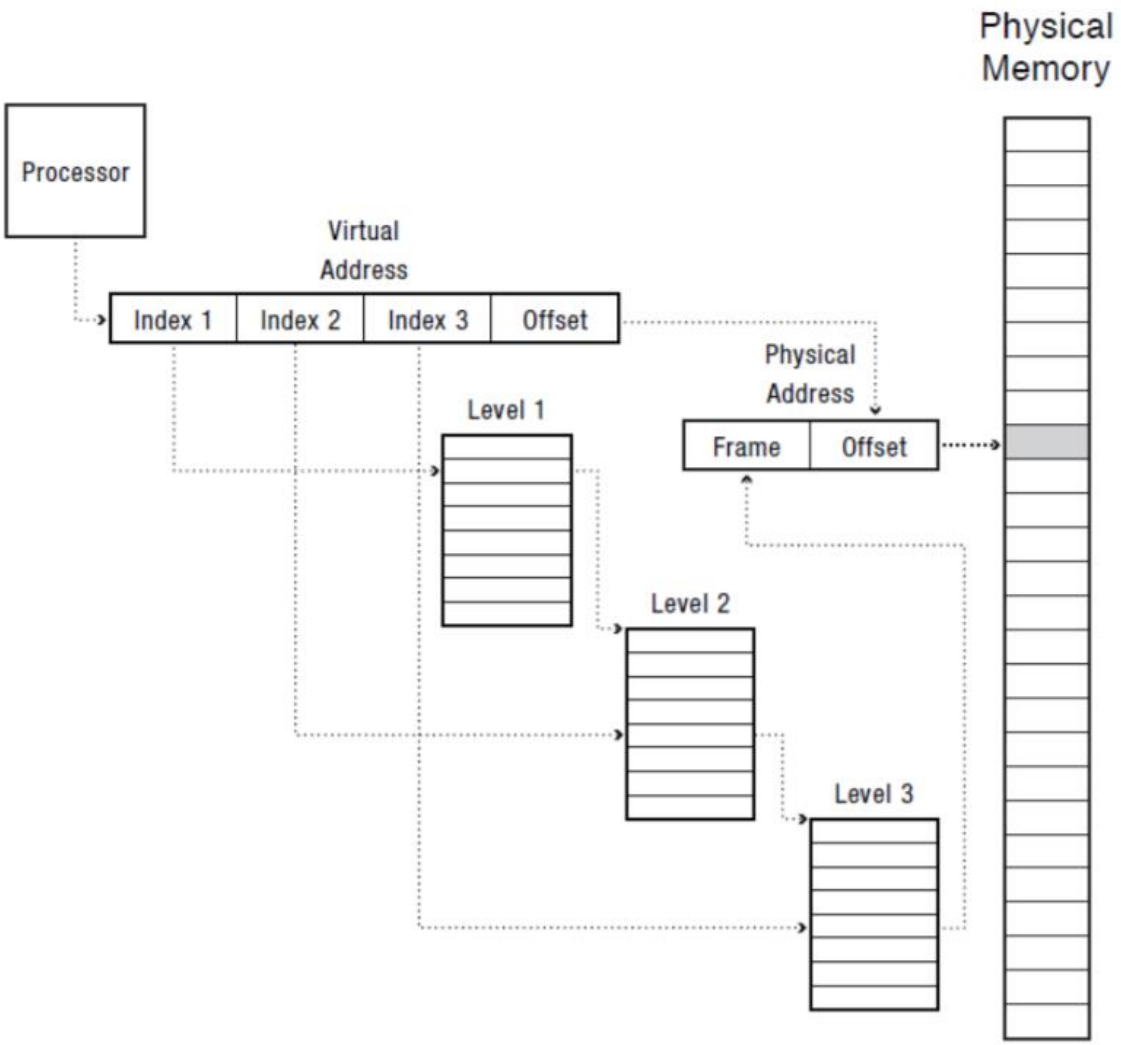
pointers &

Segment table & page table

- How does paged segmentation reduce space overhead compared to flat page table design?

# of PTEs ↓ or PGSIZE ↓

Multilevel Paging



Questions: Multilevel paging

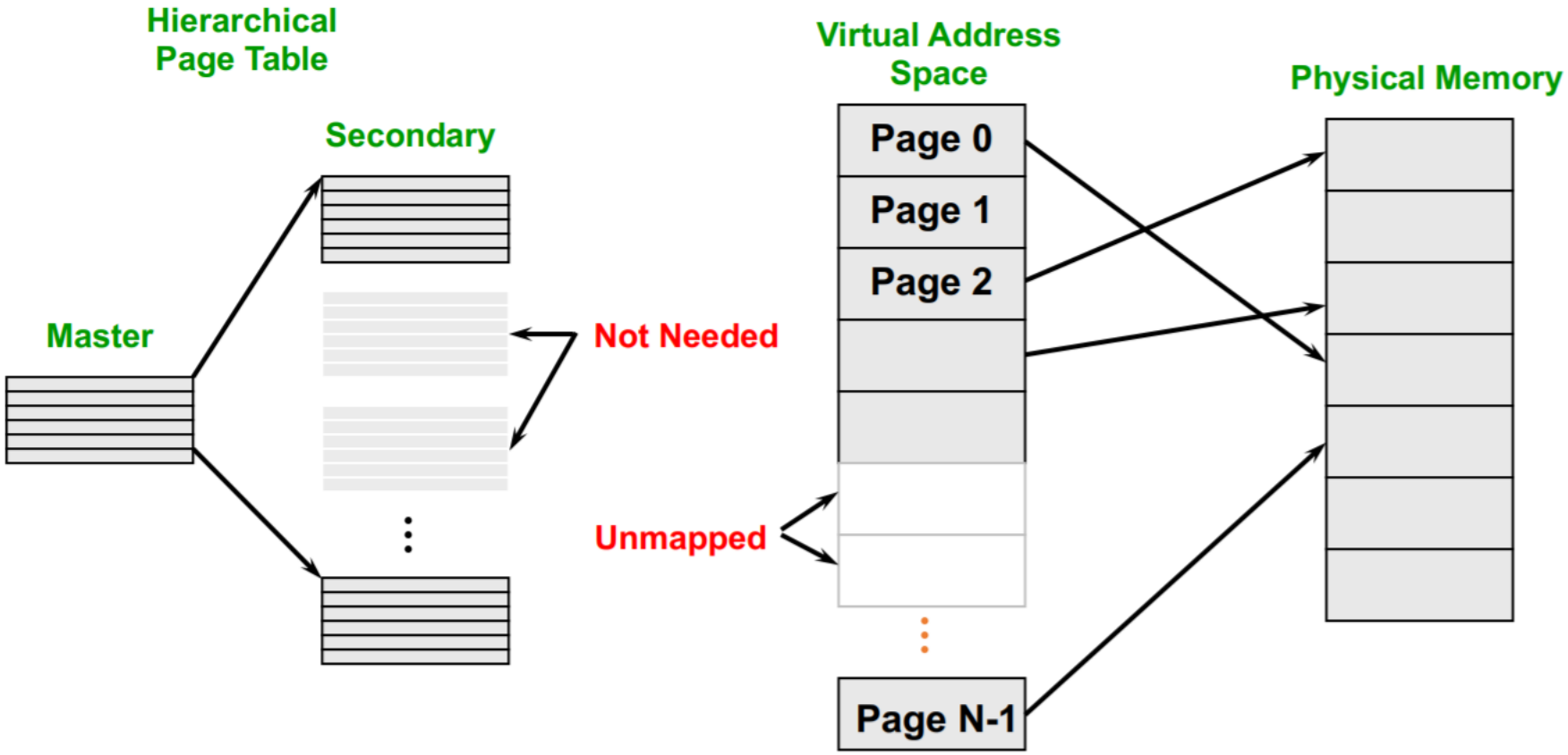
- With multilevel paging, what must be saved/restored across a process context switch?

Page tables

- How does multilevel paging reduce space overhead compared to flat page table design?

PTE ↓

Multilevel paging



Multilevel Translation

- Pros:
  - Allocate/fill only page table entries that are in use
  - Simple memory allocation: physical memory is managed by a fixed size
  - Fine-grained share at segment or page level
- Cons:

#of Memory access ↑ if level goes deeper

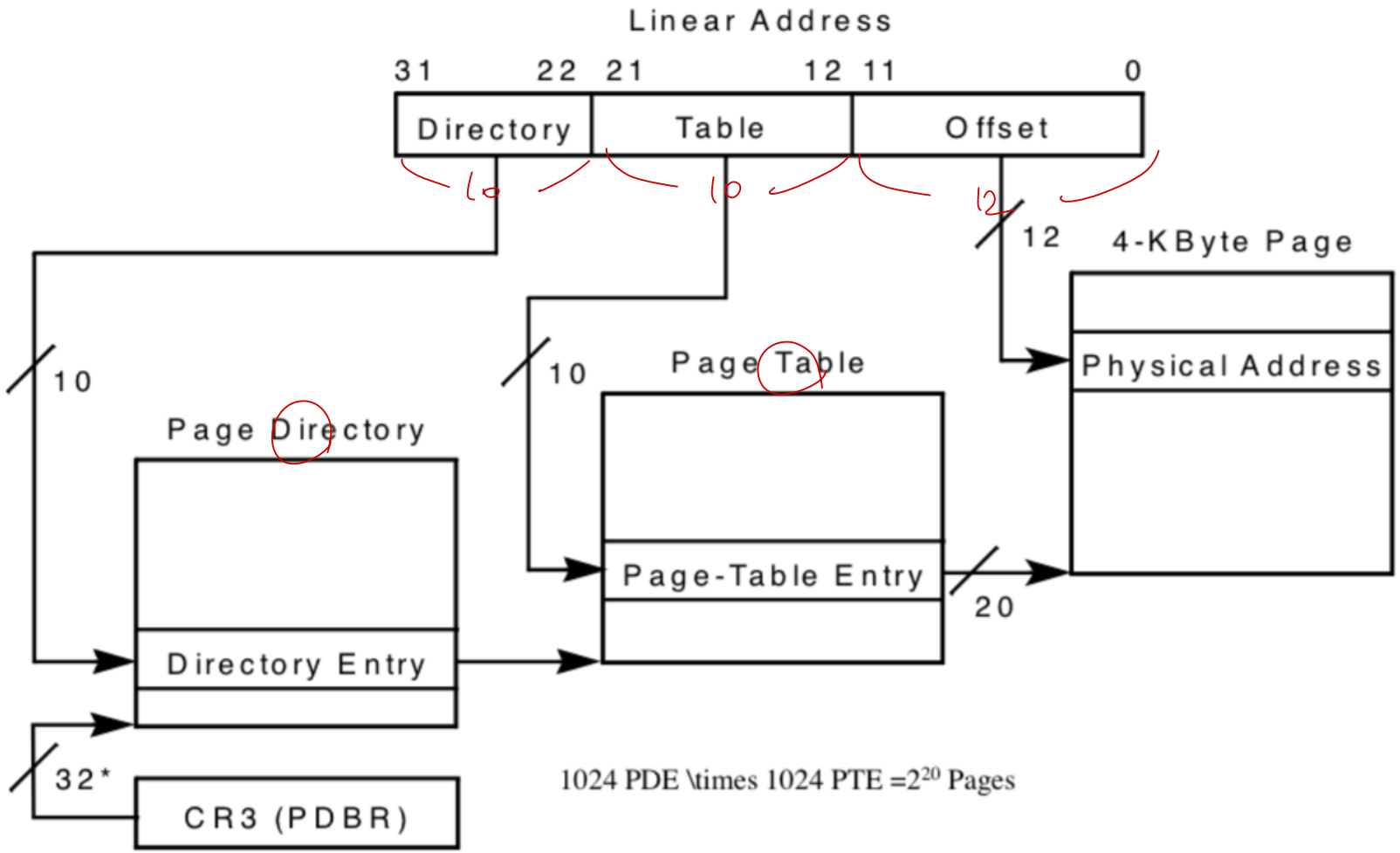
TLB miss logic goes complex

Let’s switch gears to Implementation discussions

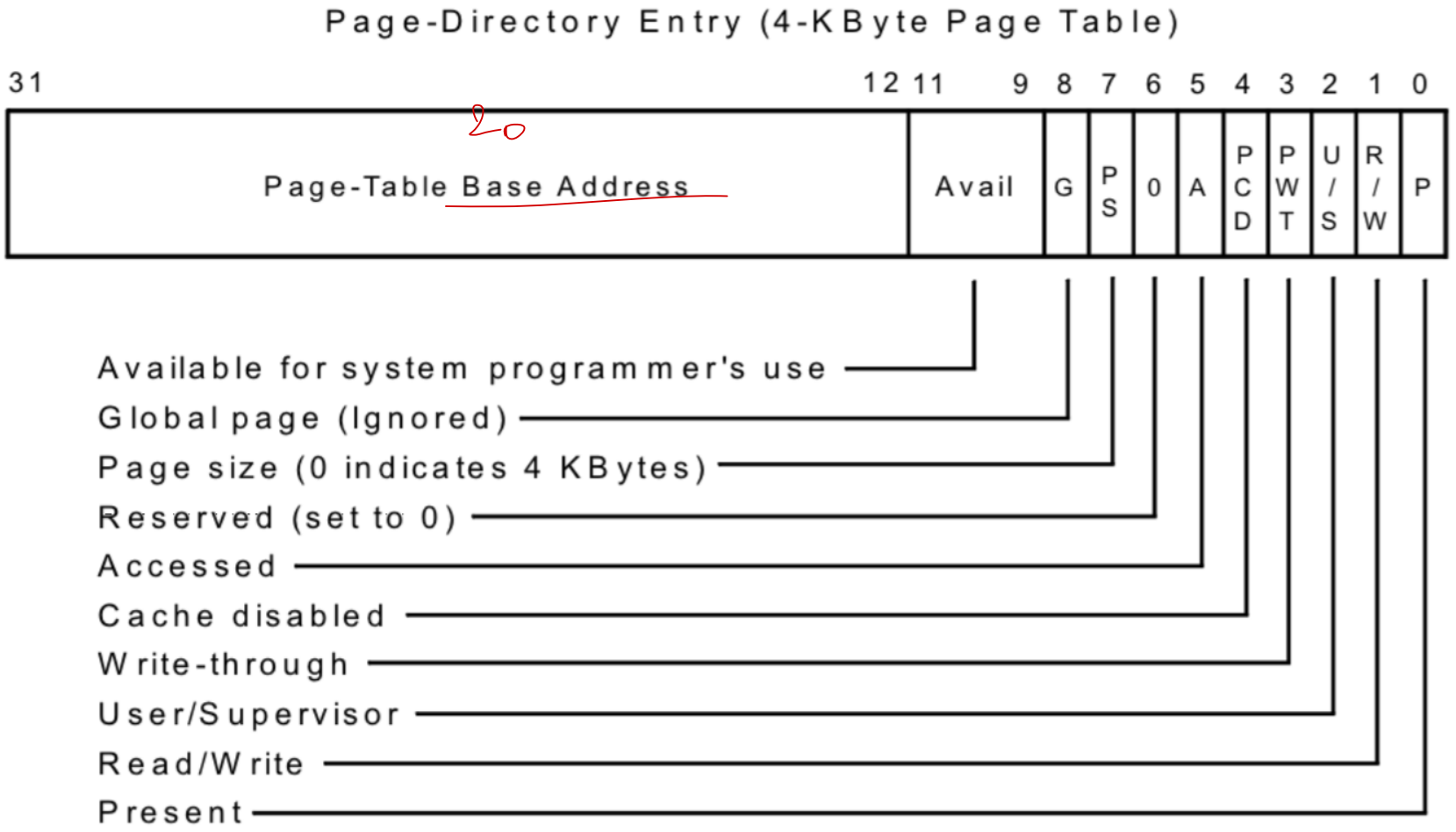
x86 Multilevel Paged Segmentation

- 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

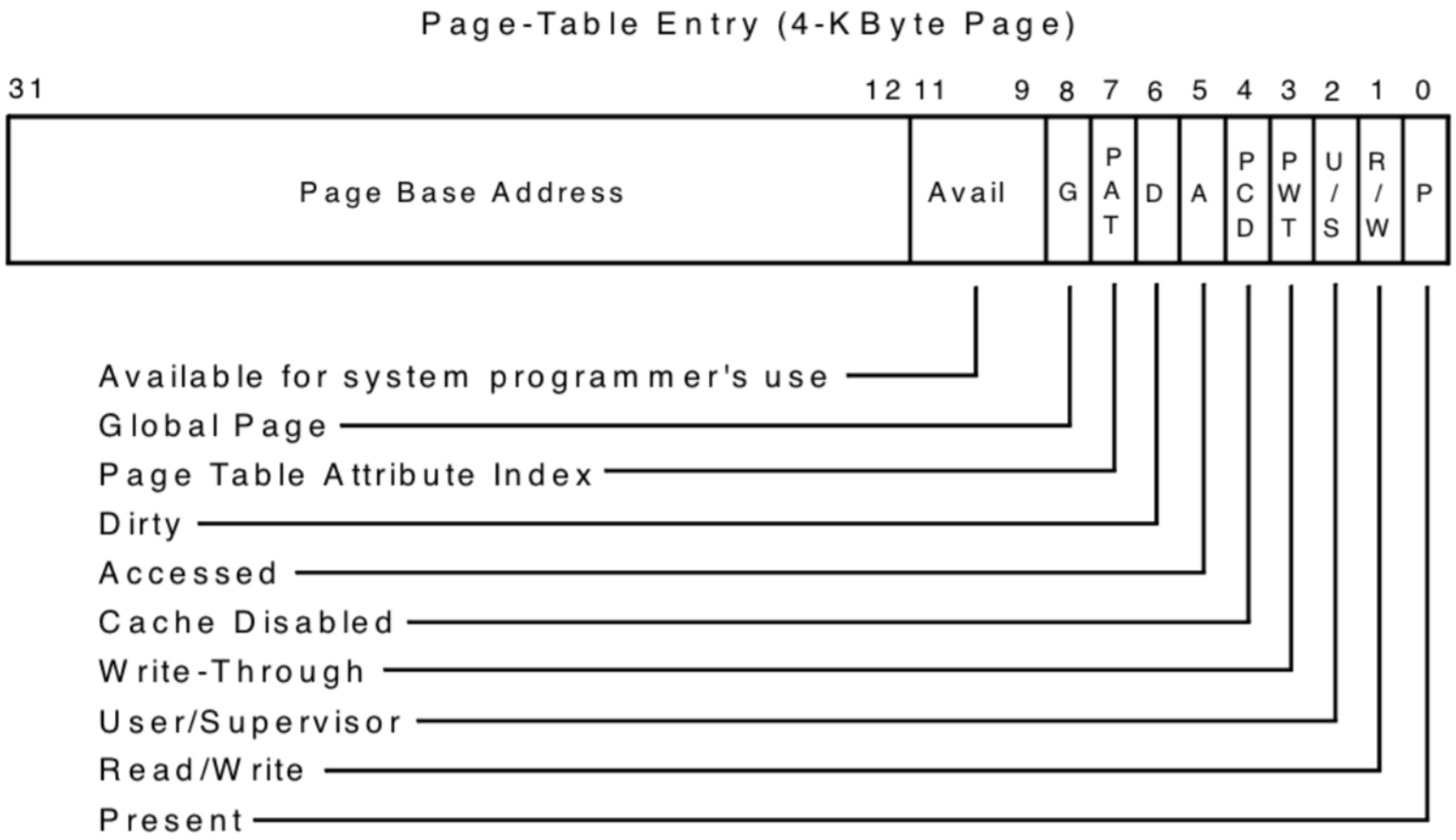
X86 Page Translation



X86 Page directory entry



X86 Page Table Entry



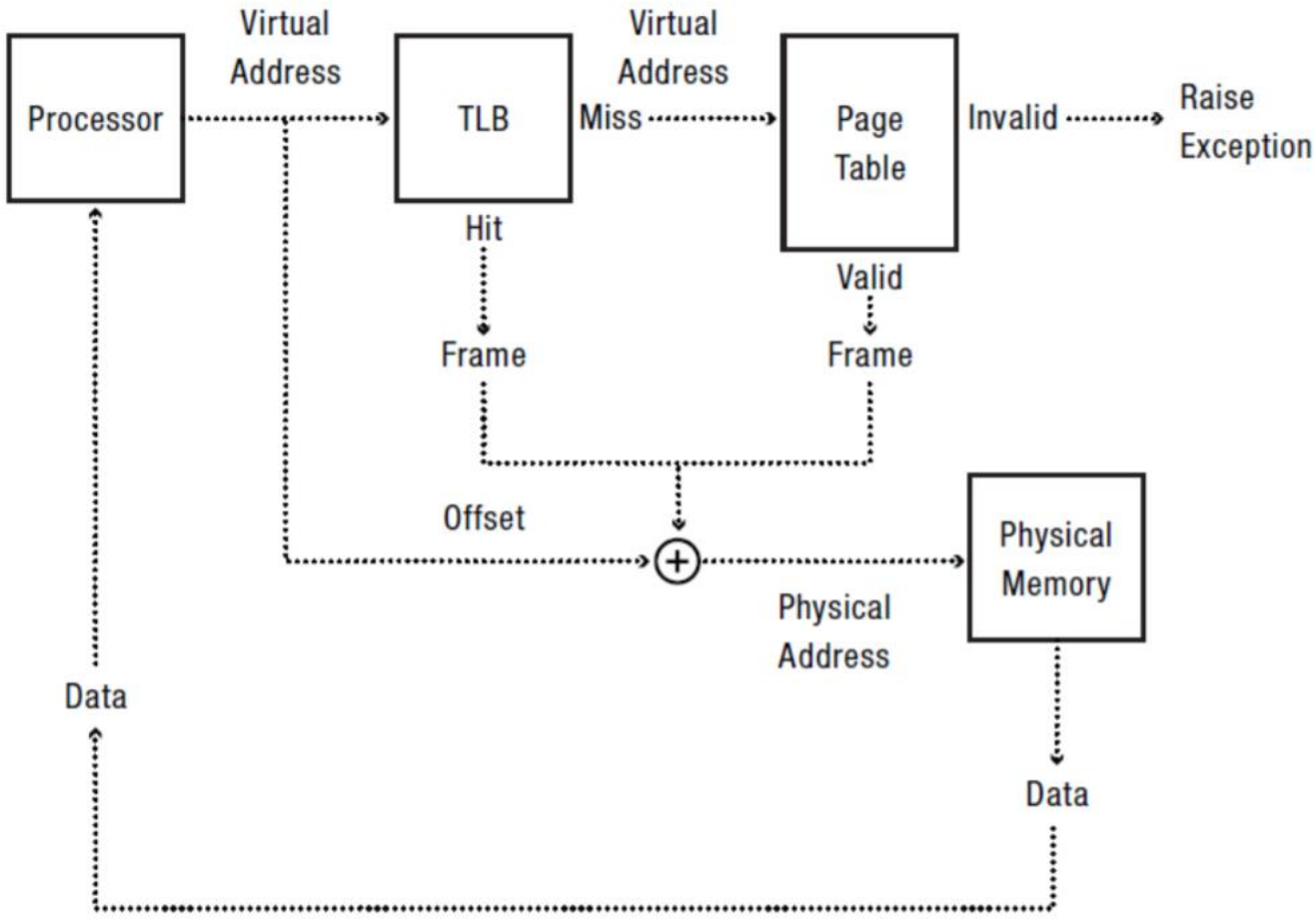
Efficient Address Translation

- 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 +

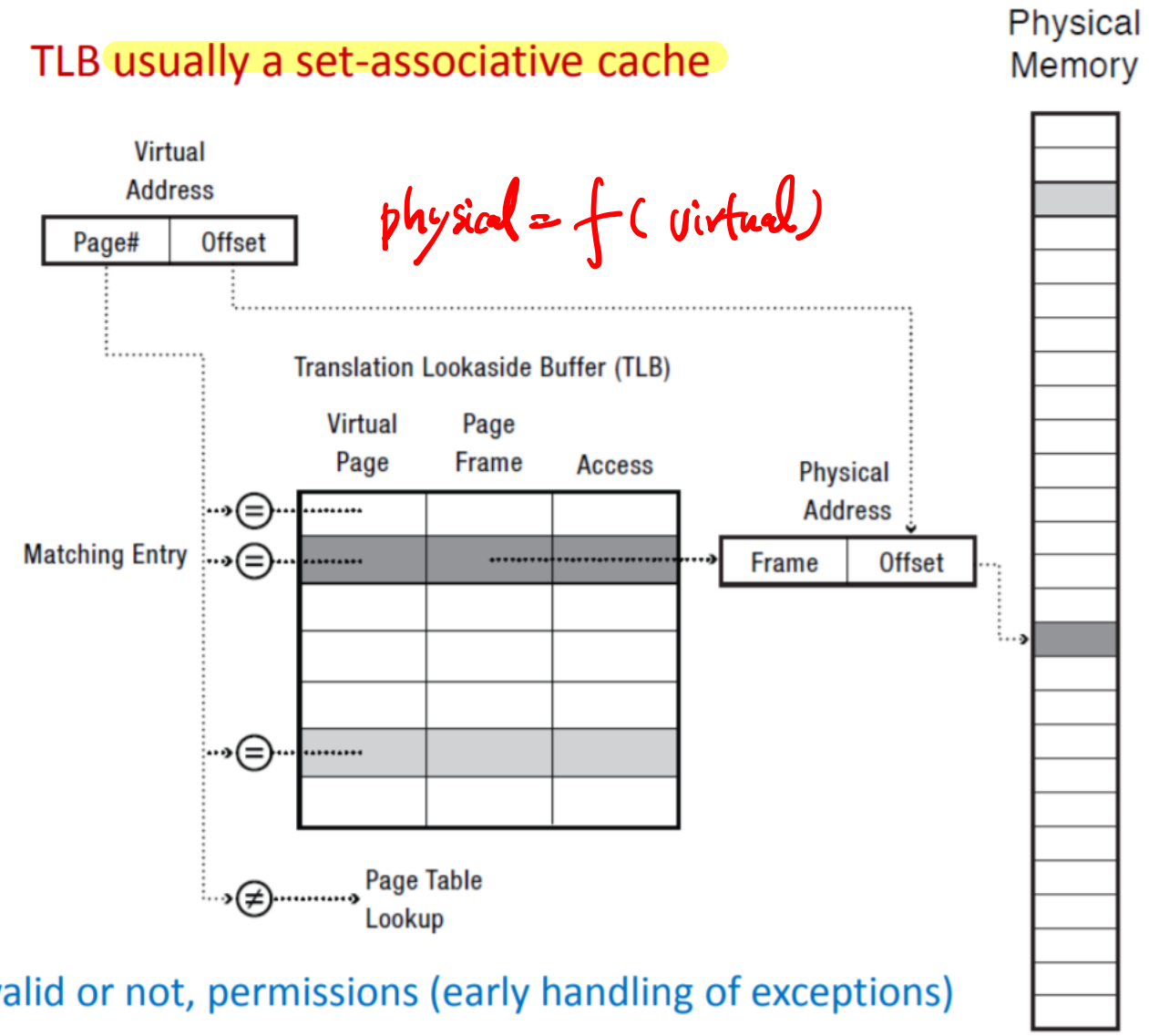
  Prob(TLB miss) * cost of page table lookup

TLB and Page Table Translation



TLB Lookup

TLB usually a set-associative cache

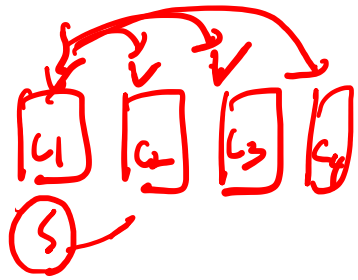


Access: valid or not, permissions (early handling of exceptions)

How TLB works, part 1

- What happens when the OS changes the permissions on a page?
  - For demand paging, copy on write, zero on reference, …

How TLB works, part 1



- 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 flush
invpg
TLB shootdown

How TLB works, part 2

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

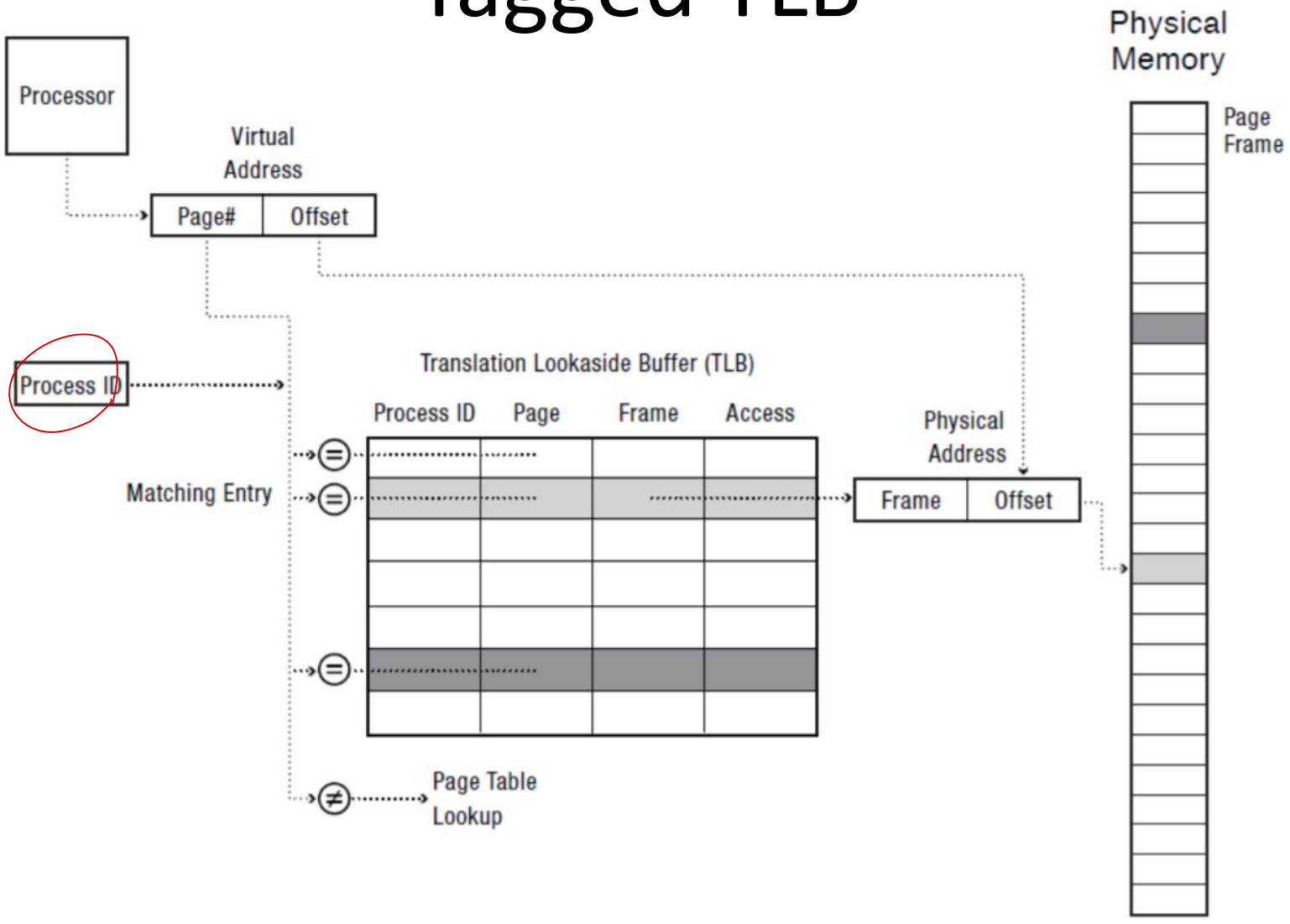
How TLB works, part 2

- 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

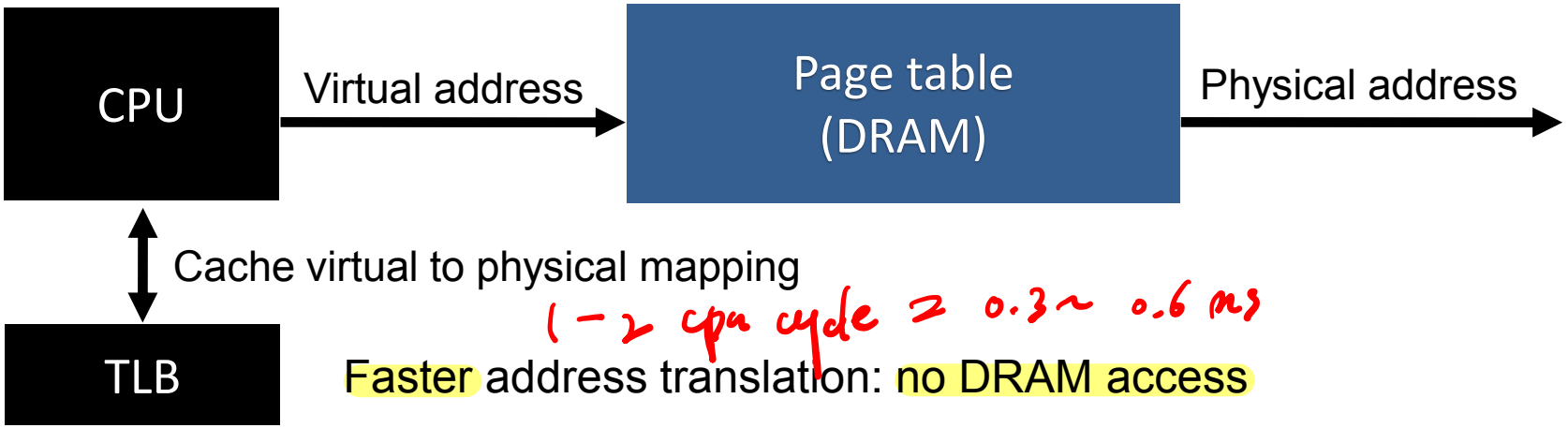
Don't need to flush all TLB entries when CS

Tagged TLB



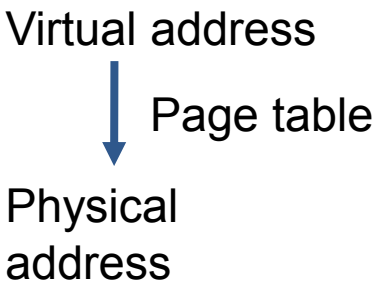
TLB reduces address translation cost

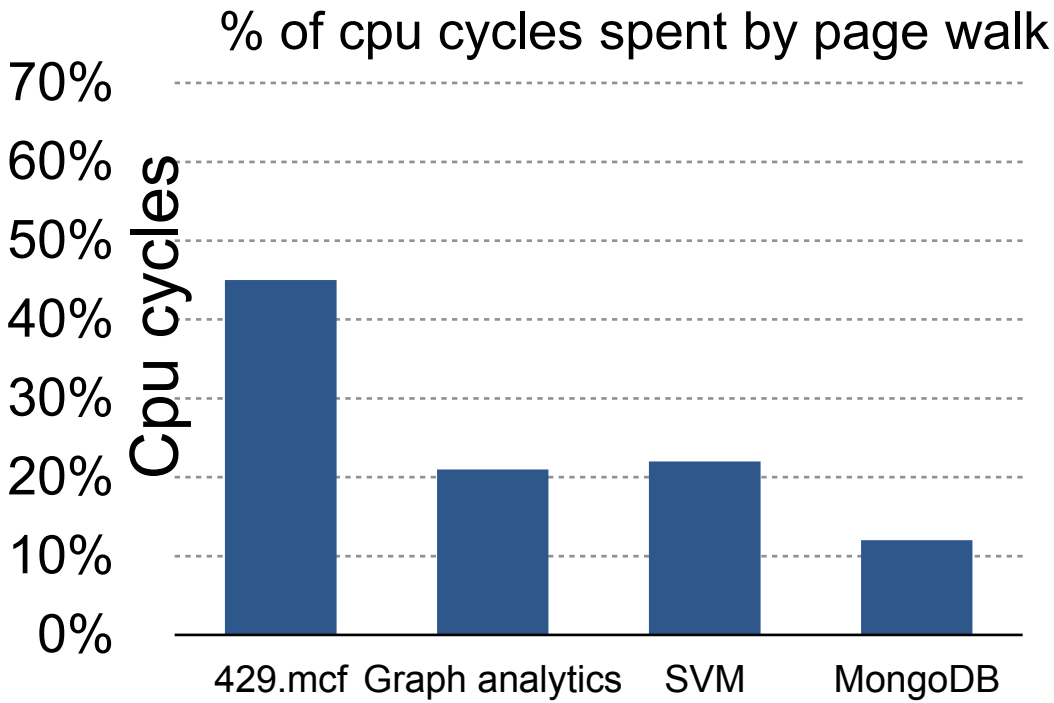
- TLB: caching virtual to physical address translation



High address translation cost

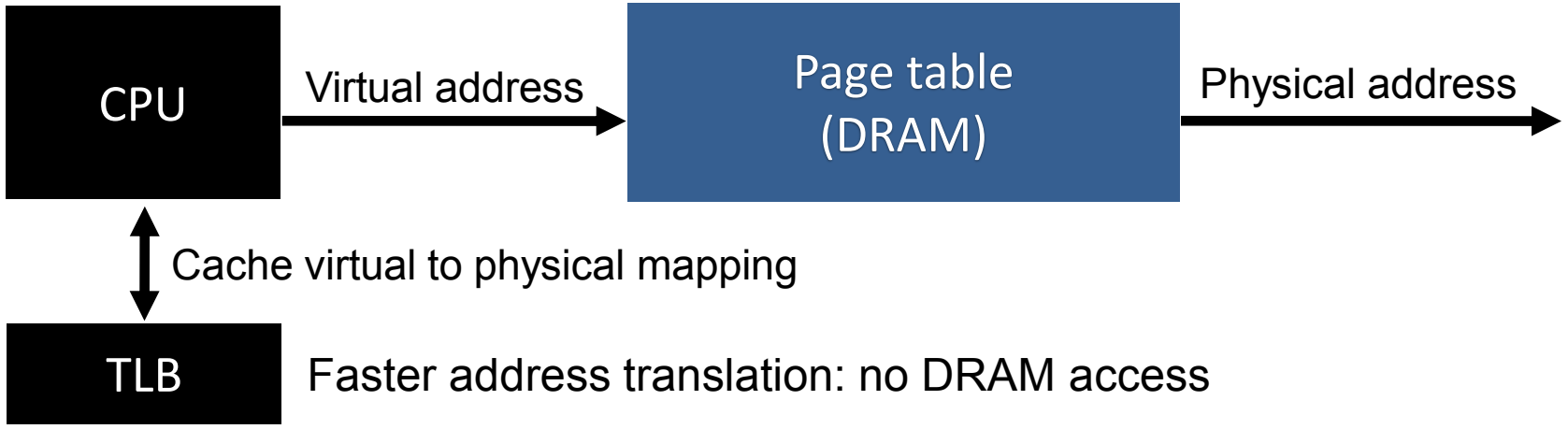
- Modern applications: large memory footprint, low memory access locality
- TLB coverage using base pages is insufficient





Address translation cost in large memory system

- TLB: caching virtual to physical address translation



High address translation cost

- Modern applications: large memory footprint and low access locality
  - **TLB size does not scale with memory size (low TLB coverage)**

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 → Page
    - 2MB
    - 1GB → Superpage

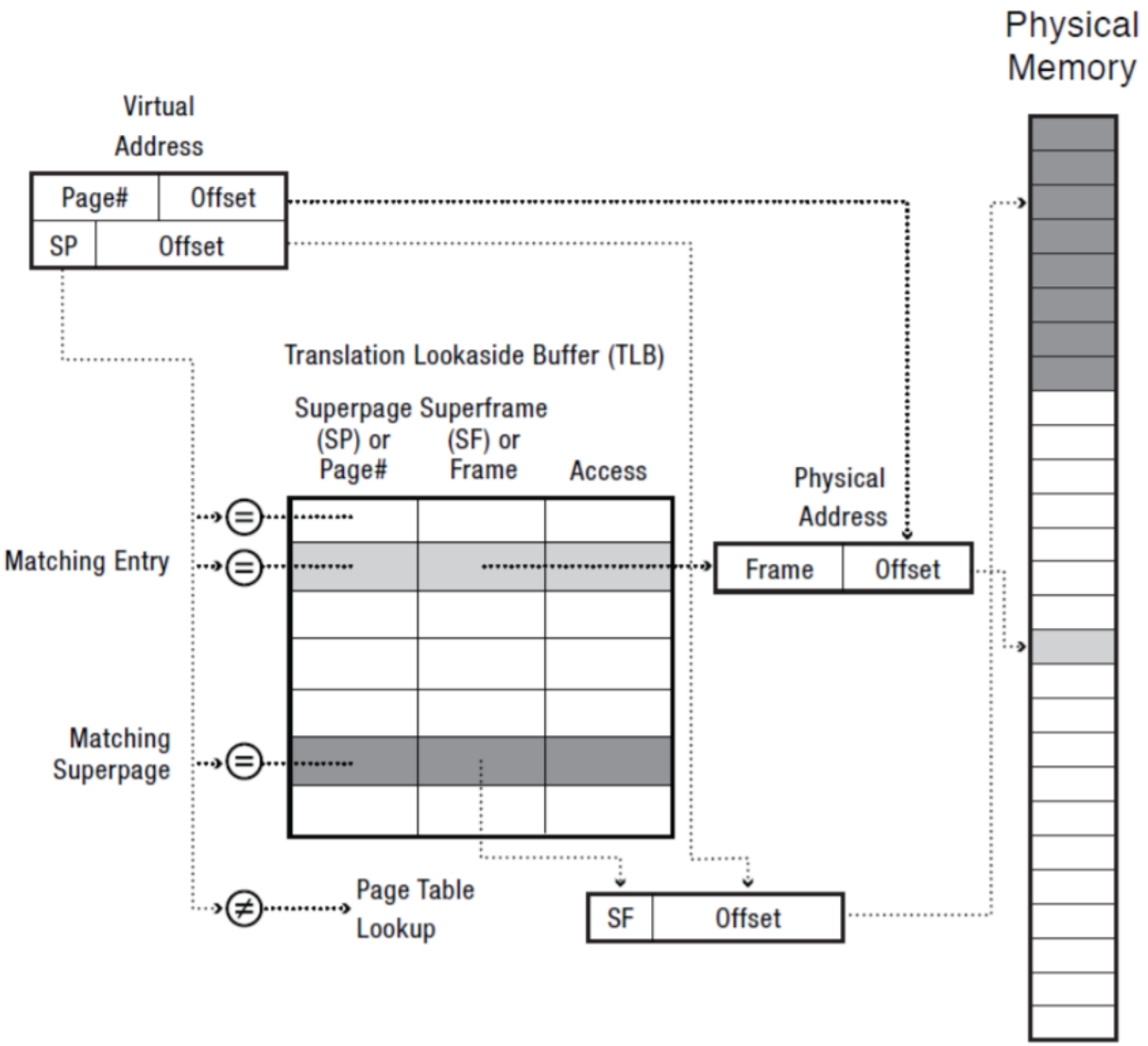
Super pages mitigate address translation costs

완화하다.

How to increase TLB coverage?

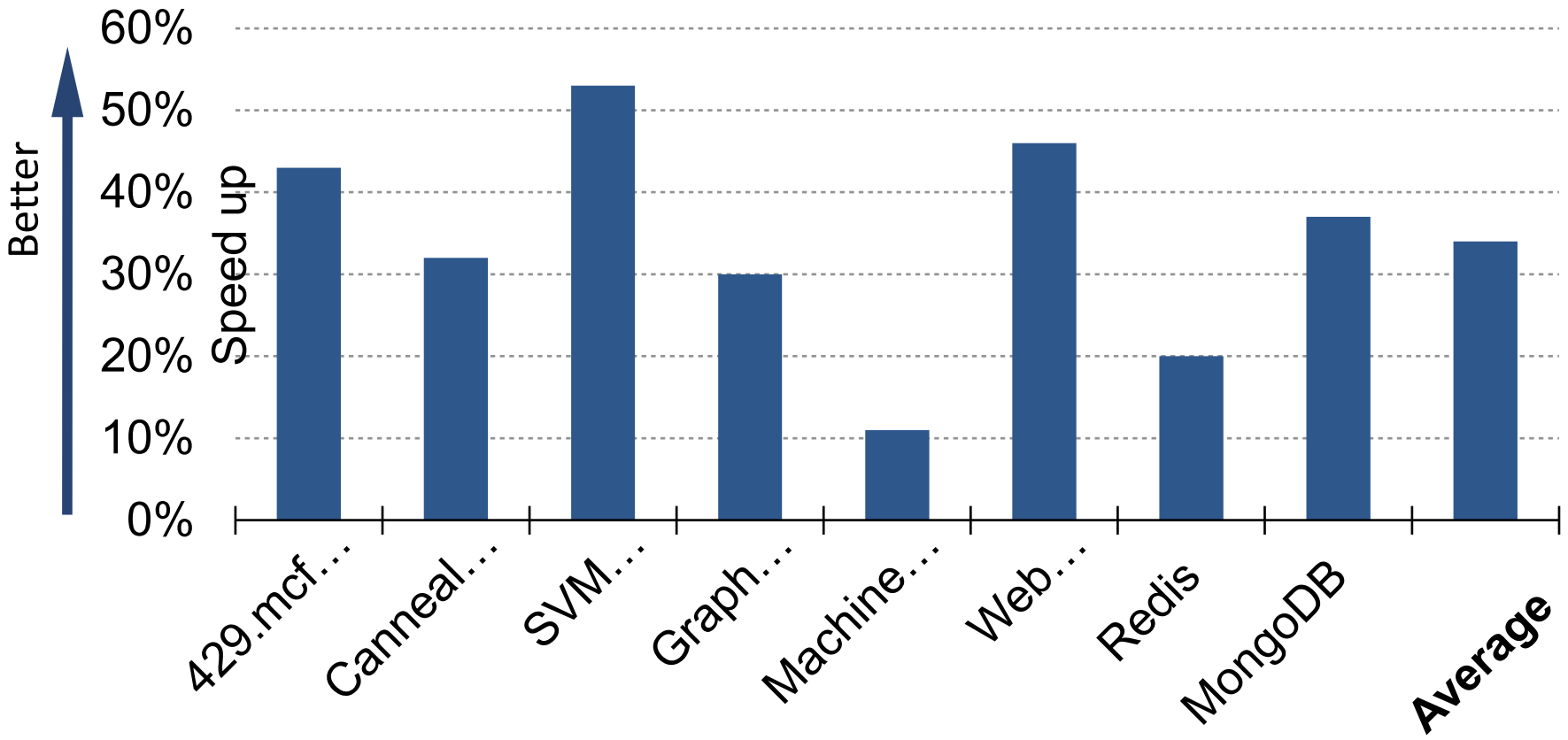
- Architecture supports larger page size (e.g., 2MB page)
- TLB entries for 2 MB pages
  - TLB covers larger address space

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

Super pages improve performance

- Application speed up over using base (4KB) pages only



Are huge pages a free lunch?

Disable Transparent Huge Pages (THP)

2.3.2. Disable Transparent Huge Pages

Tran
map
Volt

Ther

```
$ e
$ e
```

IBM Support

splunk> docs PRODUCTS SOLUTIONS CUSTOMERS COMMUNITY SPLEXICON Support & Services My Account Search D

cloudera

Why Cloudera Products Services & Support Solutions

okta

PRODUCT DOCS DISCUSSION SUPPORT

Transparent Huge Pages: Thanks for your help...please don't help

By the next morning **CPU contention was worse**.

The alarmingly high system CPU usage that we'd seen in the previous 3 months was always due to MySQL using kernel mutex. But sinc
problem, *what the heck was this?*

We discussed turning off TCMalloc, but that would've been a mistake. Implementing TCMalloc was a critical link in the chain of problem
ultimately strengthened our platform.

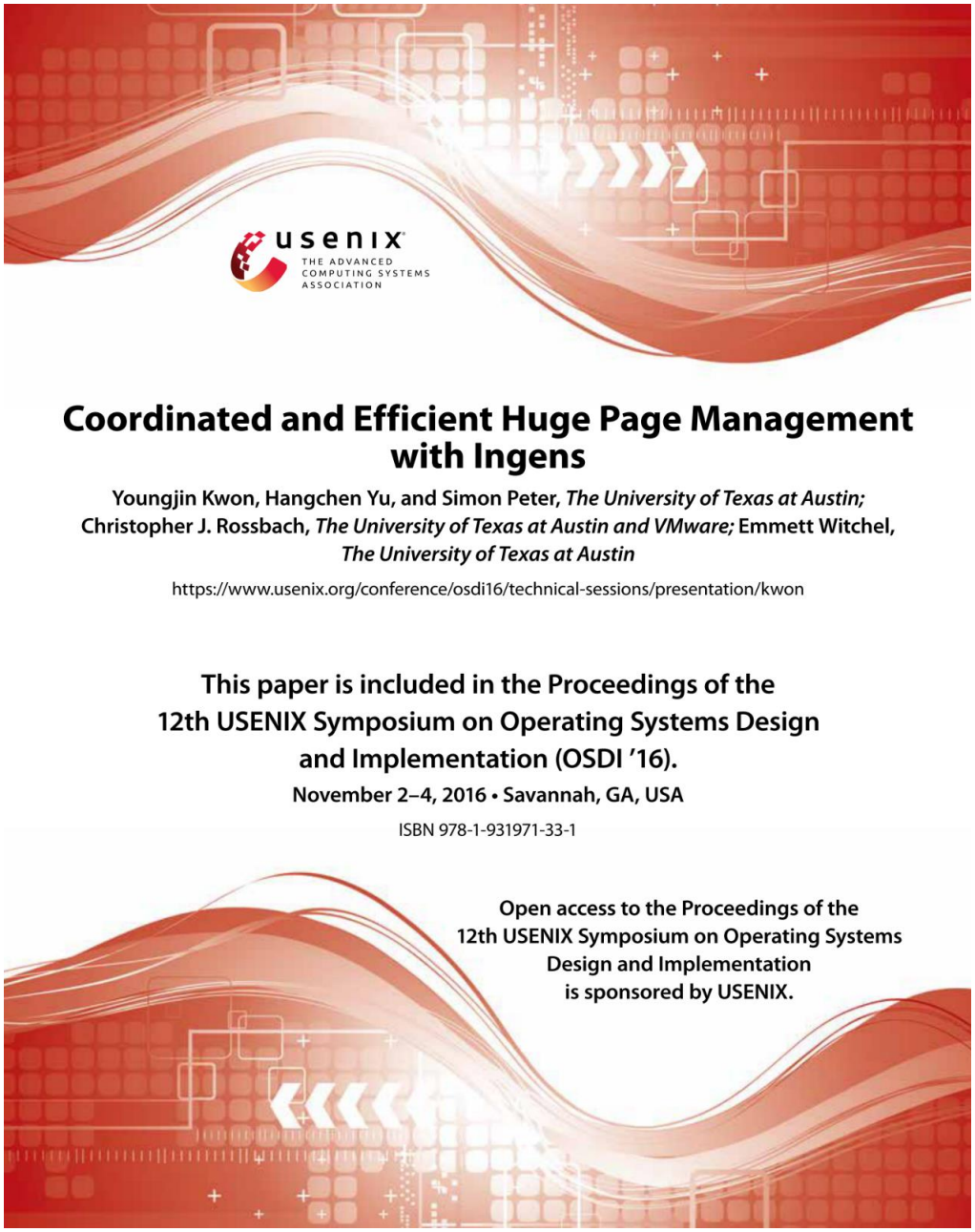
We discovered very quickly that the culprit this time was a ***khugepaged*** enabled by a Linux kernel flag called **Transparent Huge Pages**
default in most Linux distributions). Huge pages are designed to improve performance by helping the operating system manage large a
They effectively increase the page size from the standard 4kb to 2MB or 1Gb (depending on how it is configured).

THP makes huge pages easier to use by, among other things, arranging your memory into larger chunks. It works great for app servers
memory-intensive operations.

- ▸ High Availability
- ▸ Backup and Disaster Recovery
- ▸ Cloudera Manager Administration
- ▸ Cloudera Navigator Data Management Component Administration

Disabling Transparent Hugepage Compaction

Most Linux platforms supported by CDH 5 include a feature called **transparent hugepage compaction** whic
interacts poorly with Hadoop workloads and can seriously degrade performance.



A shameless plug

A research for better super page management