

CSE 451: Operating Systems, 26au

L13: Multi-core Scheduling, Virtual Memory

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

Scheduler Metrics

- We care about optimizing:
  - **Latency/response time**: how long does a task to complete?
    - **Tail latency**: what is the *worst case* response time?
  - **Throughput:** how many tasks can we do per unit of time?
  - **Overhead:** How much extra work/time is spent in the scheduler itself?
  - **Fairness:** Do all tasks get an equal share of the CPU? Should they?
  - **Predictability**: How consistent is the system's performance over time?
  - **Robustness to gaming:** like "gaming the system", not "playing Zelda"

Fair Queueing

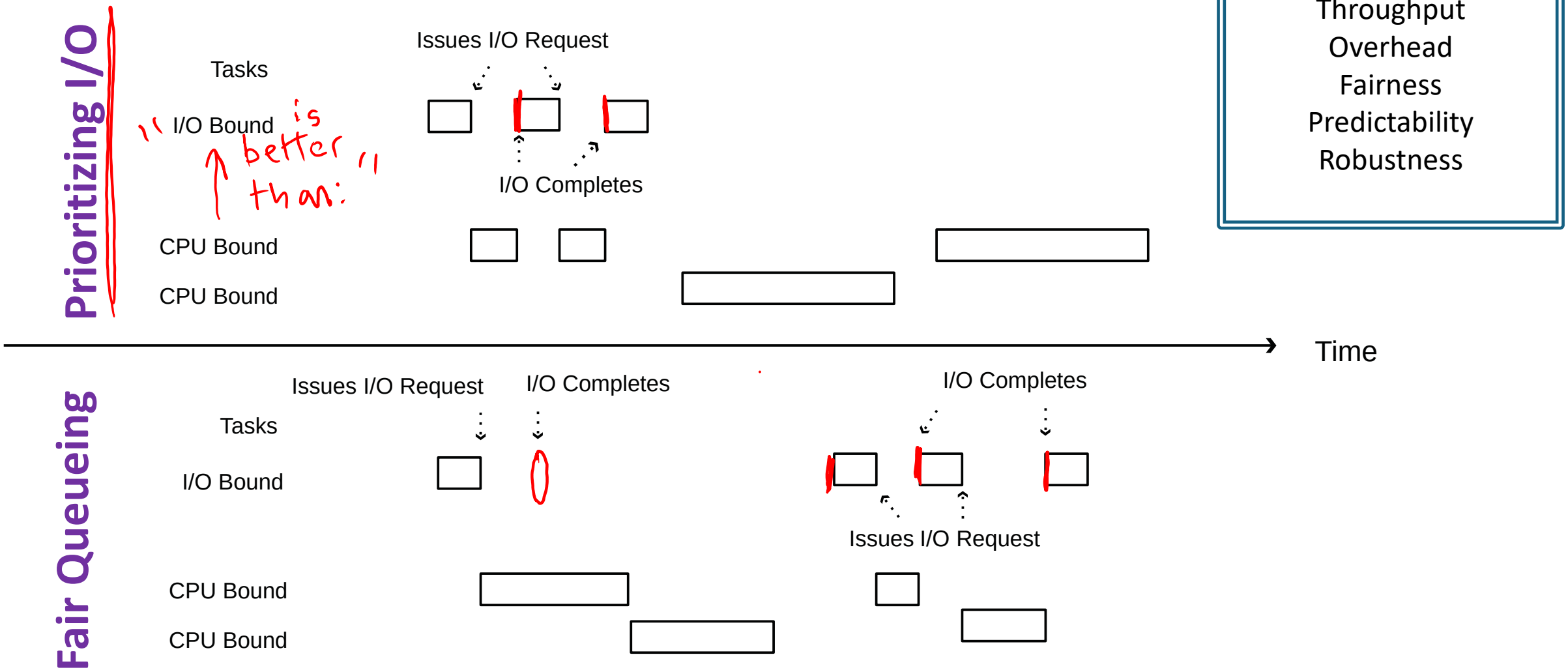
Metrics:
Avg. response time
Max response time
Throughput
Overhead
Fairness
Predictability
Robustness

- **Max-Min Fairness**
  - Maximize the minimum allocation to any task
  - Good for repetitive tasks (comes from computer networking, where workloads can be very predictable)
  - Tasks that needs less than a “fair share” get everything they need. For everyone else, we minimize their disappointment.

- Implementation:
  - Keep track of CPU usage over time (including time spent waiting for I/O)
  - Schedule the task with the smallest accumulated CPU time, +/- time quantum
  - Scheduling is recursively repeated until left-over resources are divided evenly among remaining tasks

Comparisons

Metrics:
Avg. response time
Max response time
Throughput
Overhead
Fairness
Predictability
Robustness



Uniprocessor Summary (1)

- FIFO
  - Simple and minimizes overhead
  - Variable size tasks experience poor average response time
  - Equal size tasks, optimal average response time
- SJF
  - Considering only CPU usage (not I/O blocking), SJF is optimal for average response time
  - SJF is *pessimal* for variance in response time.

Uniprocessor Summary (2)

- Round Robin
  - For variable size tasks, Round Robin approximates SJF
  - For equal size tasks, very poor average response time
- Tasks that intermix processor and I/O
  - Benefit from SJF
  - Do poorly under Round Robin, especially with a long quantum
- MFQ attempts to prioritize latency sensitive tasks
  - Approximates SJF (high variance for long jobs)
  - Under high load, MFQ approximates round robin

Multi-processor Scheduling Policies

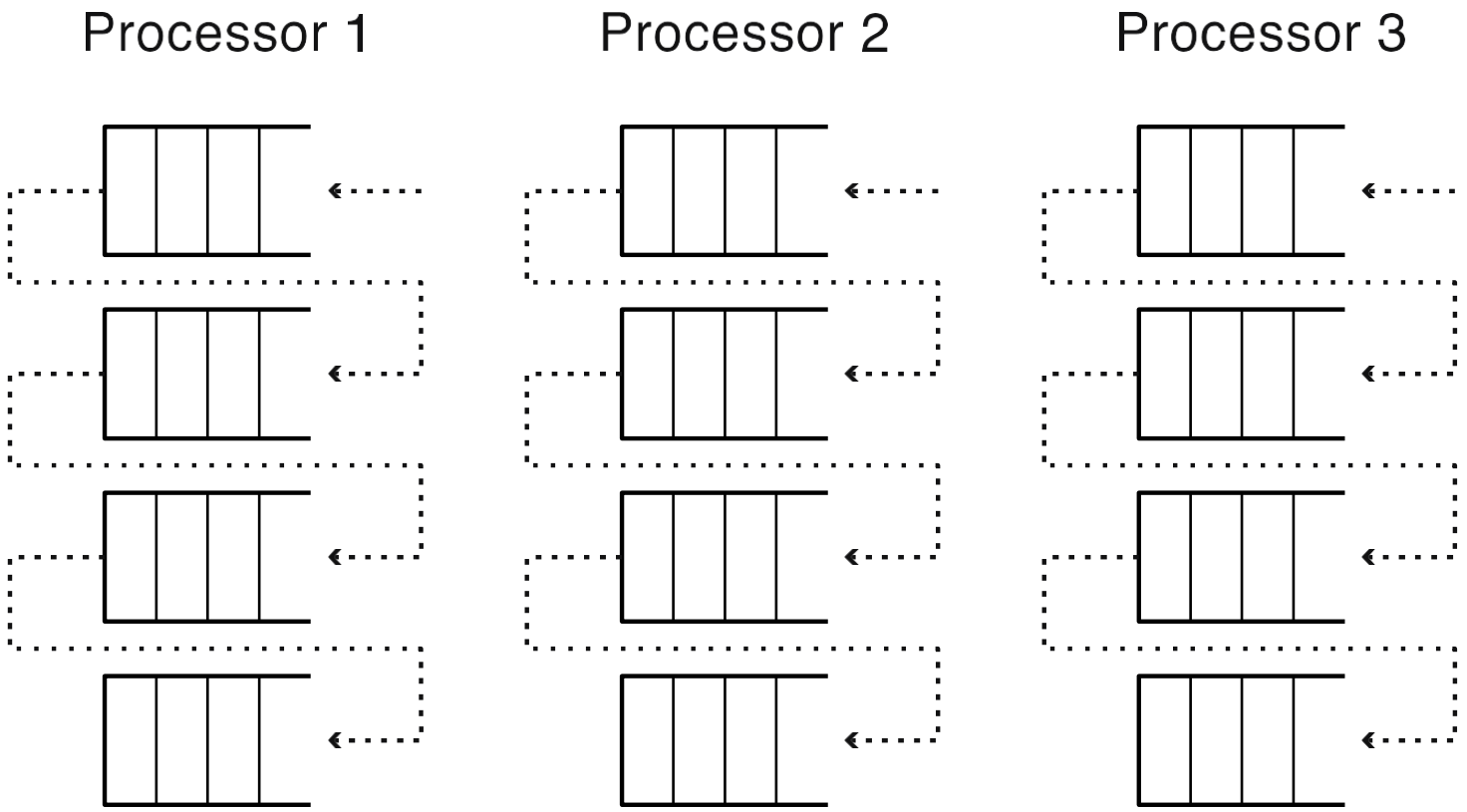
More processors, more problems

- Tasks that move to a different core will start with a cold cache and might take longer than just waiting longer for their current core
- Kernel structures need to be synchronized across cores
  - Necessitates a scheduler spinlock ← bottleneeeeck :)
  - On multi-*processor* systems, high-traffic data like the process table can ping-pong between caches

Mitigation Technique 1: Per-Processor Affinity Scheduling

- Each processor has its own ready list
  - Protected by a per-processor spinlock
- Put threads back on the ready list where it had most recently run
  - Ex: when I/O completes, or on Condition->signal
- Idle processors can steal work from other processors

Per-Processor MFQ w/ Affinity Scheduling



Tradeoff:

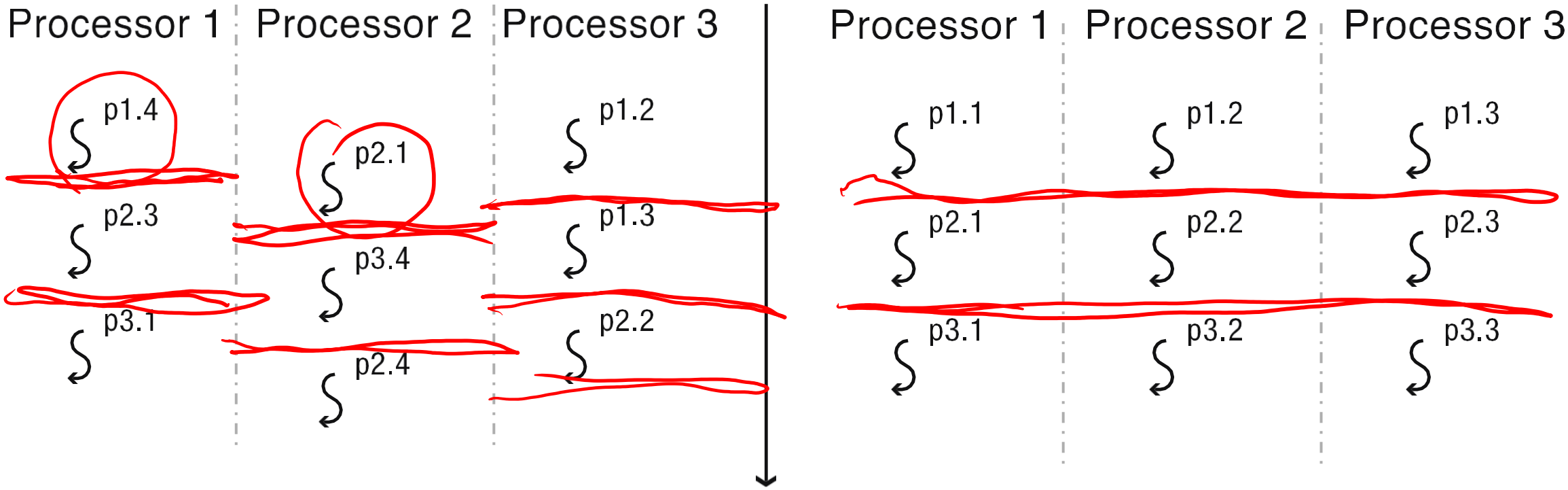
Needs rebalancing in case a lot of high-priority tasks end up on one processor

…But this requires a global view of all queues, and so halts progress on entire machine

Mitigation Technique 2: Gang Scheduling

"Oblivious scheduling": each CPU time-slices its ready list independently. Process threads are scheduled on an "as-available" basis.

"Gang scheduling": CPU time-slices are synchronized. One process is given time on all cores, and then evicted from all cores.

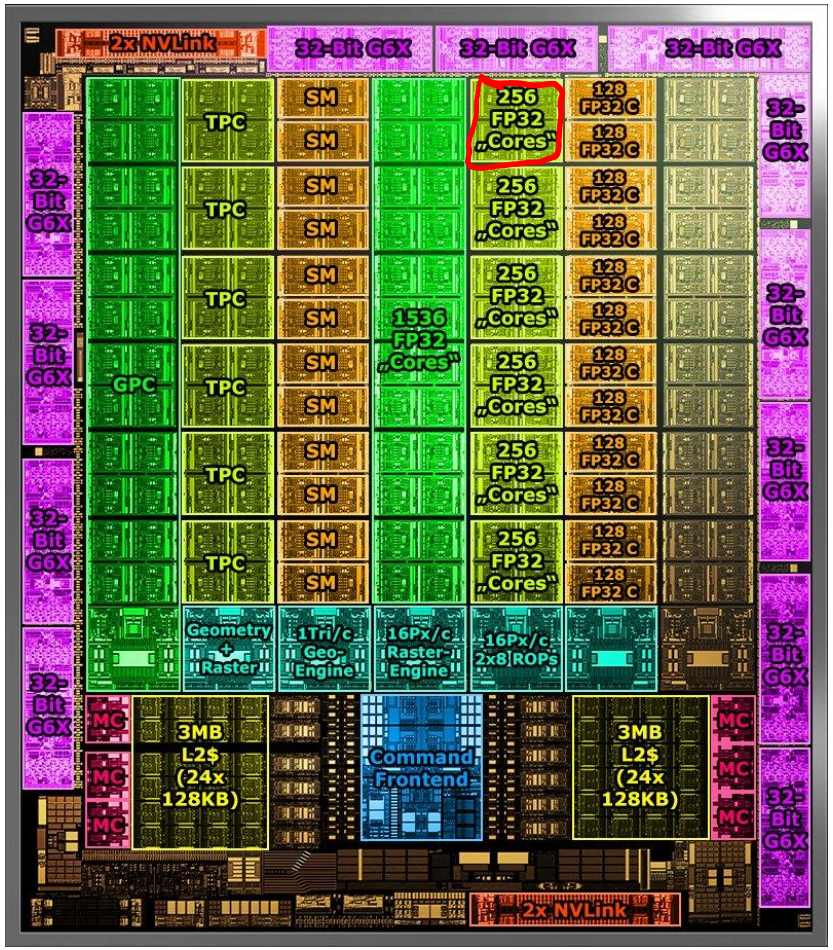


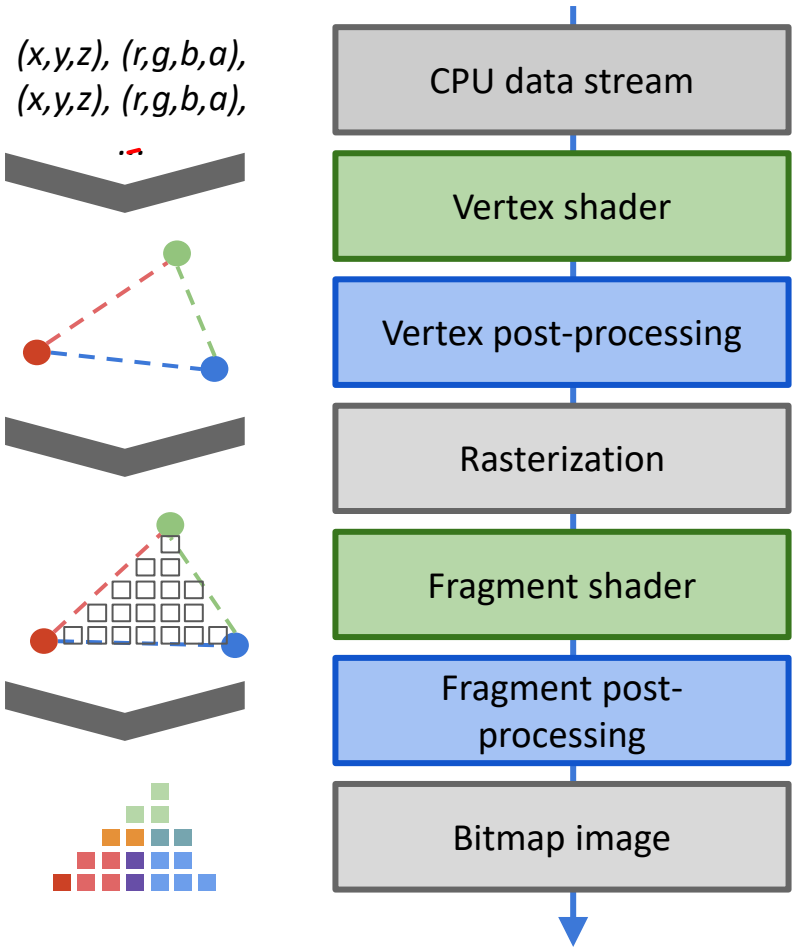
px.y = Thread y in process x

Mitigation Technique 3: Bulk Synchronous Model of Parallelism

- Idea:
  - Farm out small parcels of work to each core
  - Cores work in parallel, but without communication
  - Wait for all cores to finish their work parcels (a "**barrier**")
  - Exchange relevant data and outputs between cores
  - Repeat
- Examples:
  - Graphical rendering
  - MapReduce
  - Machine learning training and inference
  - Physical model simulations (fluid flow over a wing)

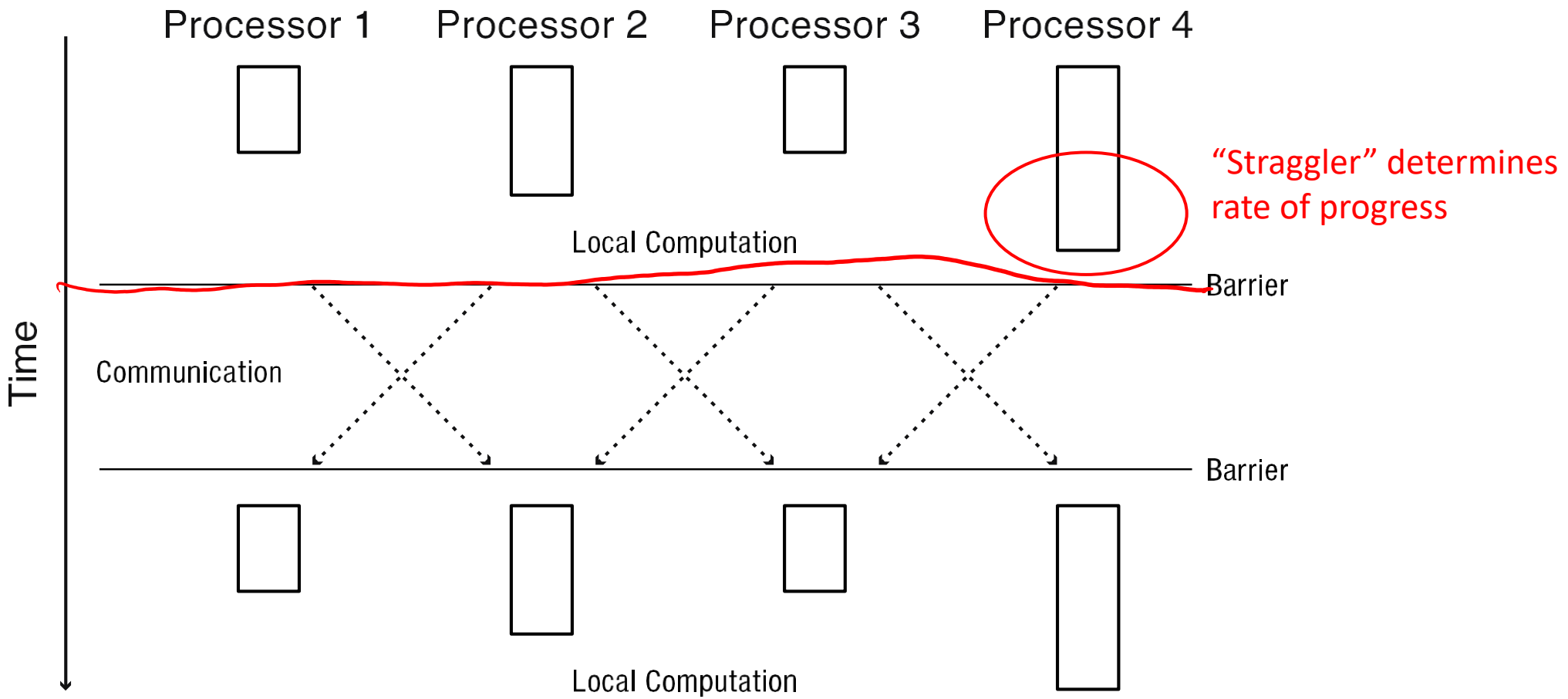
GPU as bulk synchronous processor



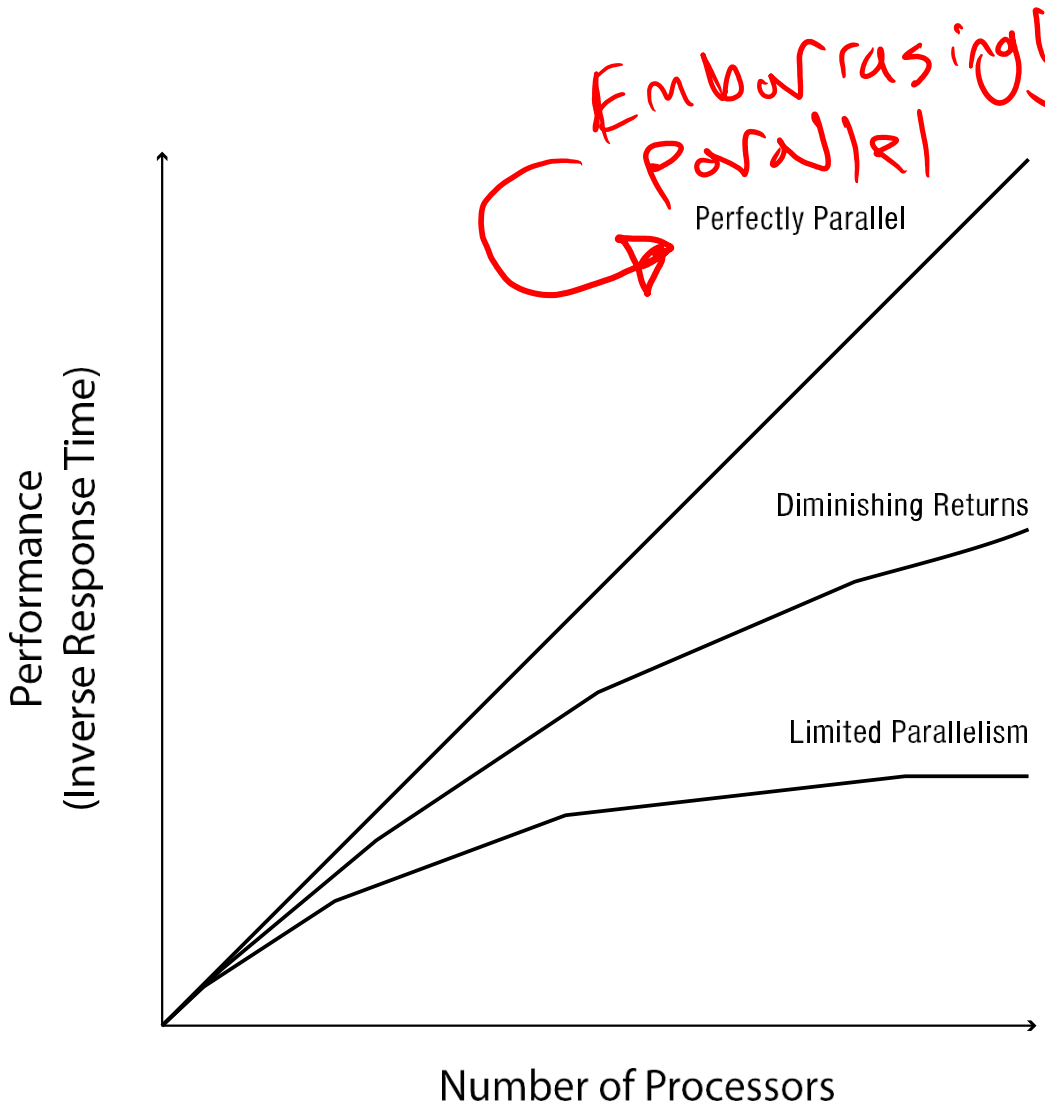


https://www.tomshardware.com/news/infrared-photographer-photos-nvidia-ga102-ampere-silicon

Bulk Synchronous, Abstractly



Parallelism isn't a magic solution



Little to no overhead to running in parallel, all cpus used

Some Overhead, some cpus underused

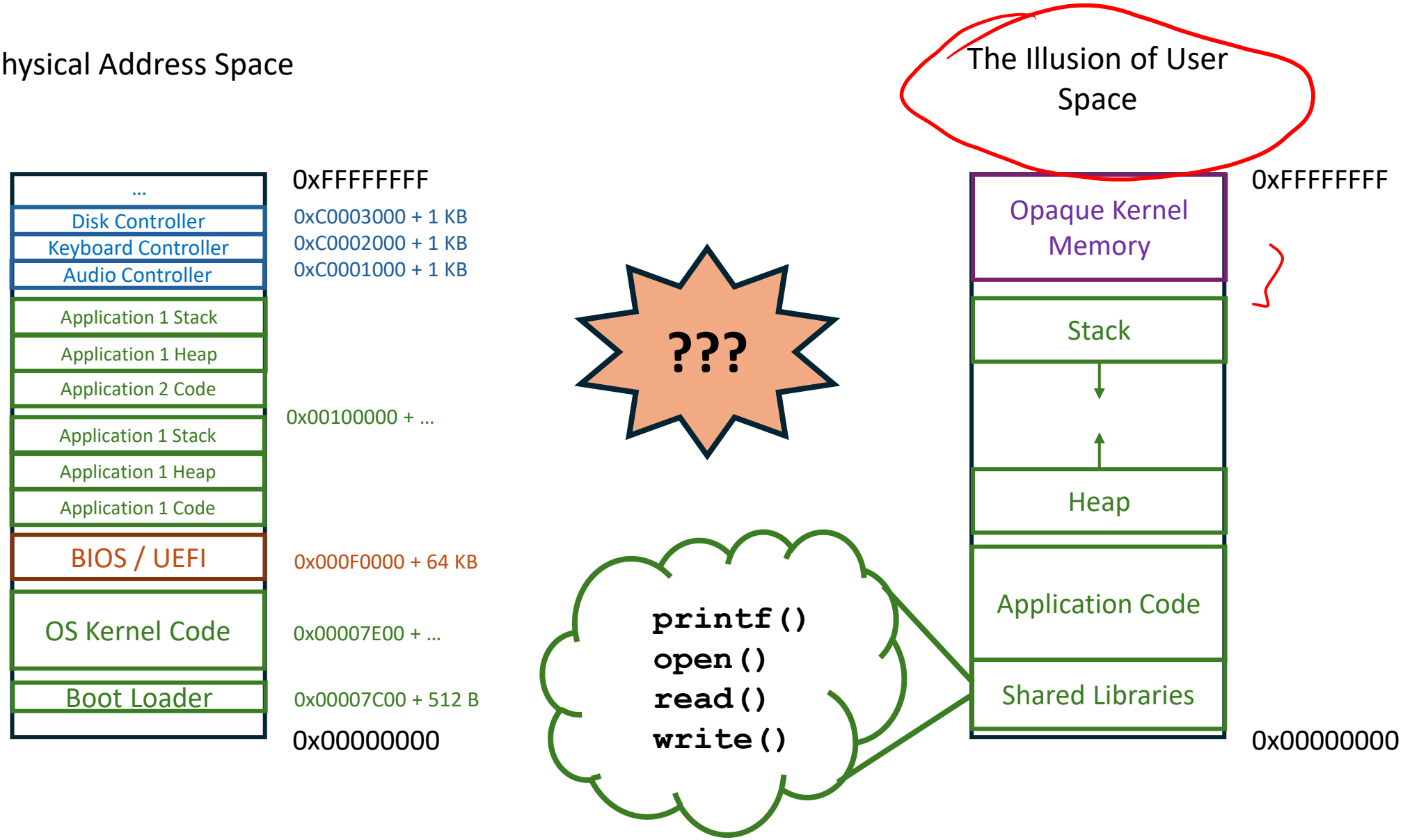
Can make use of additional cpus, but only up to a point (lock contention, tail latency)

This is the end of the content covered on the midterm!

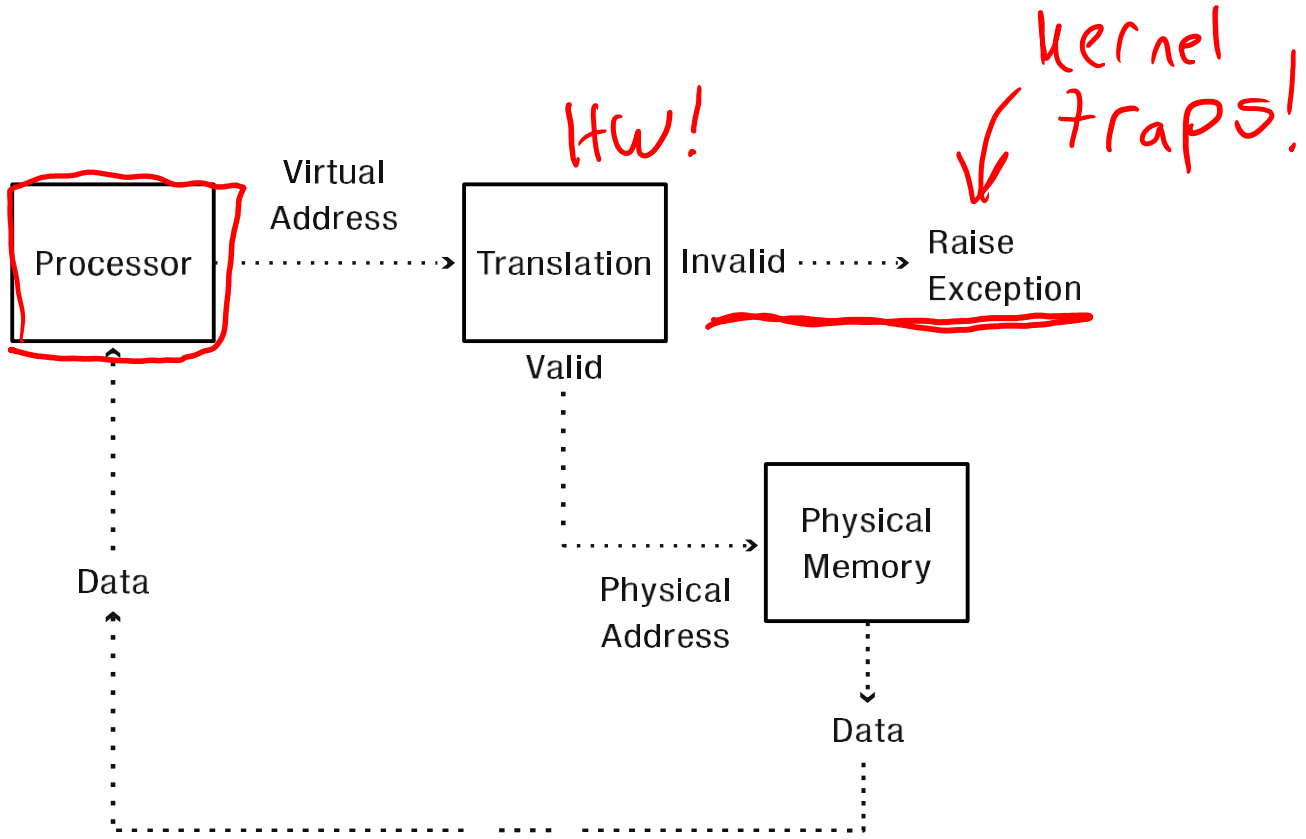
Virtual Memory

In all its glorious, gritty detail

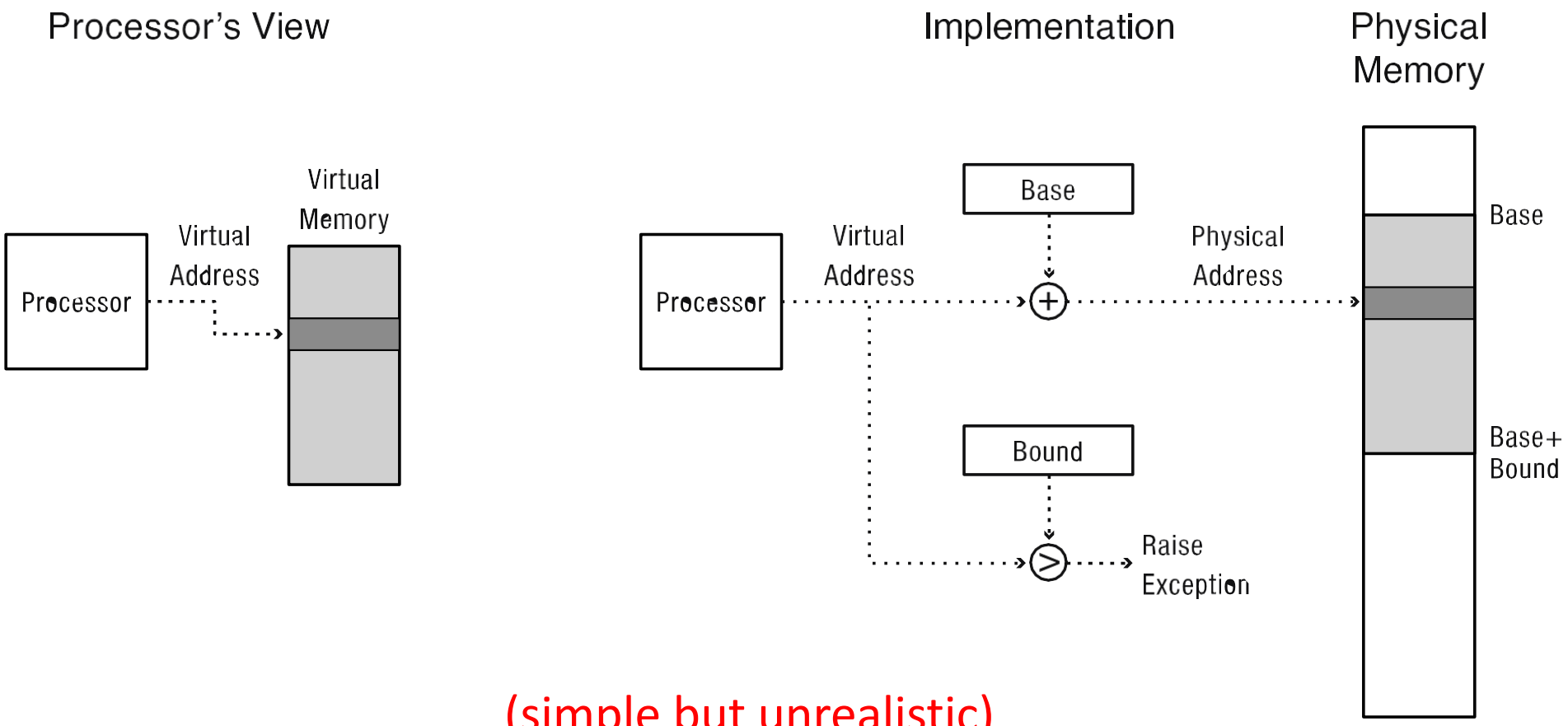
Physical Address Space



Virtual Memory, maximally abstract



Toy Example: Base and Bounds



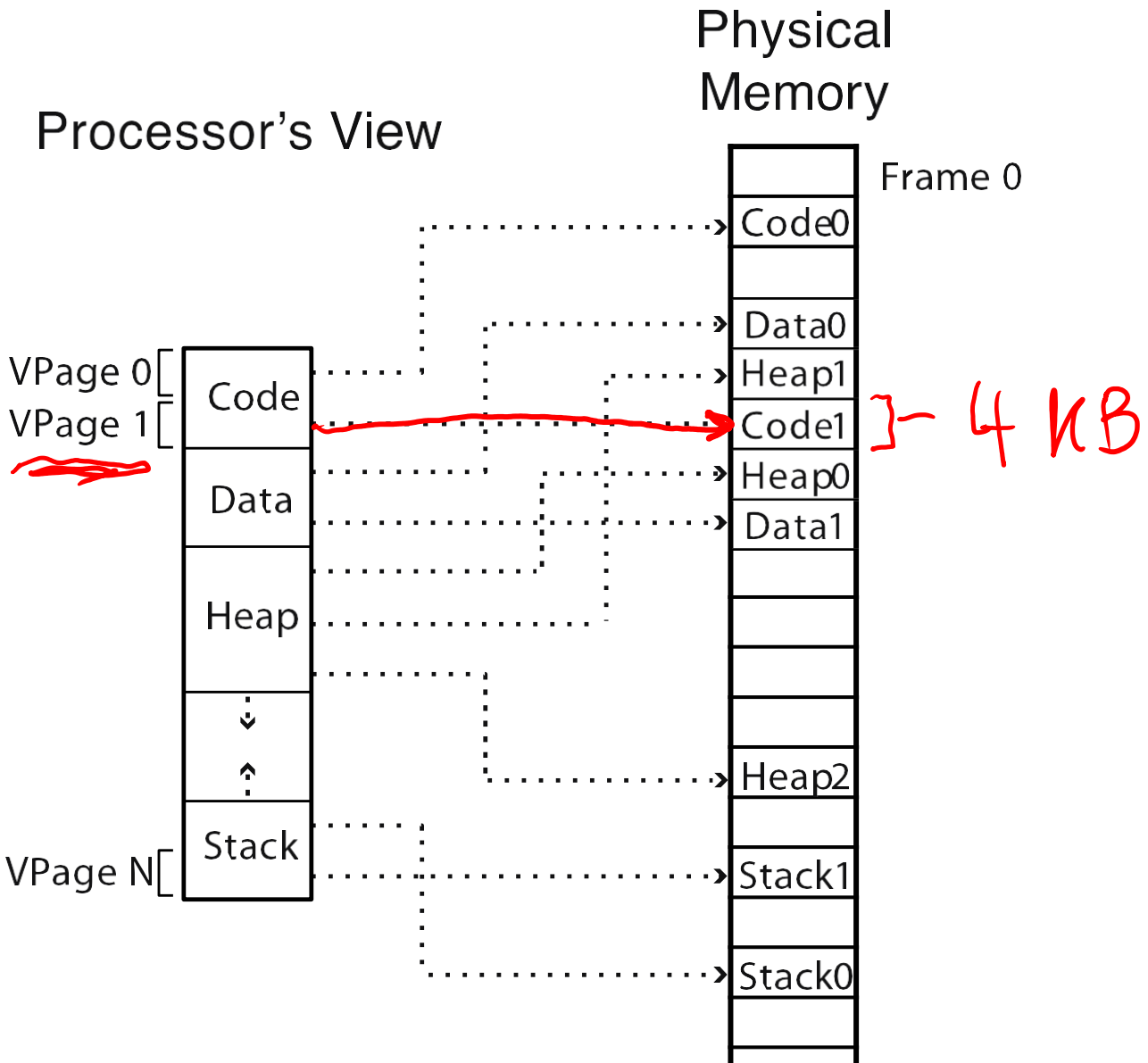
(simple but unrealistic)

Toy Example: Base and Bounds

- Pros?
  - Simple
  - Fast (2 registers, adder, comparator)
  - 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
  - Fragmentation – hard to efficiently use memory if each program is variable size
  - What if virtual address space > physical address space?

In Real Life: Paging

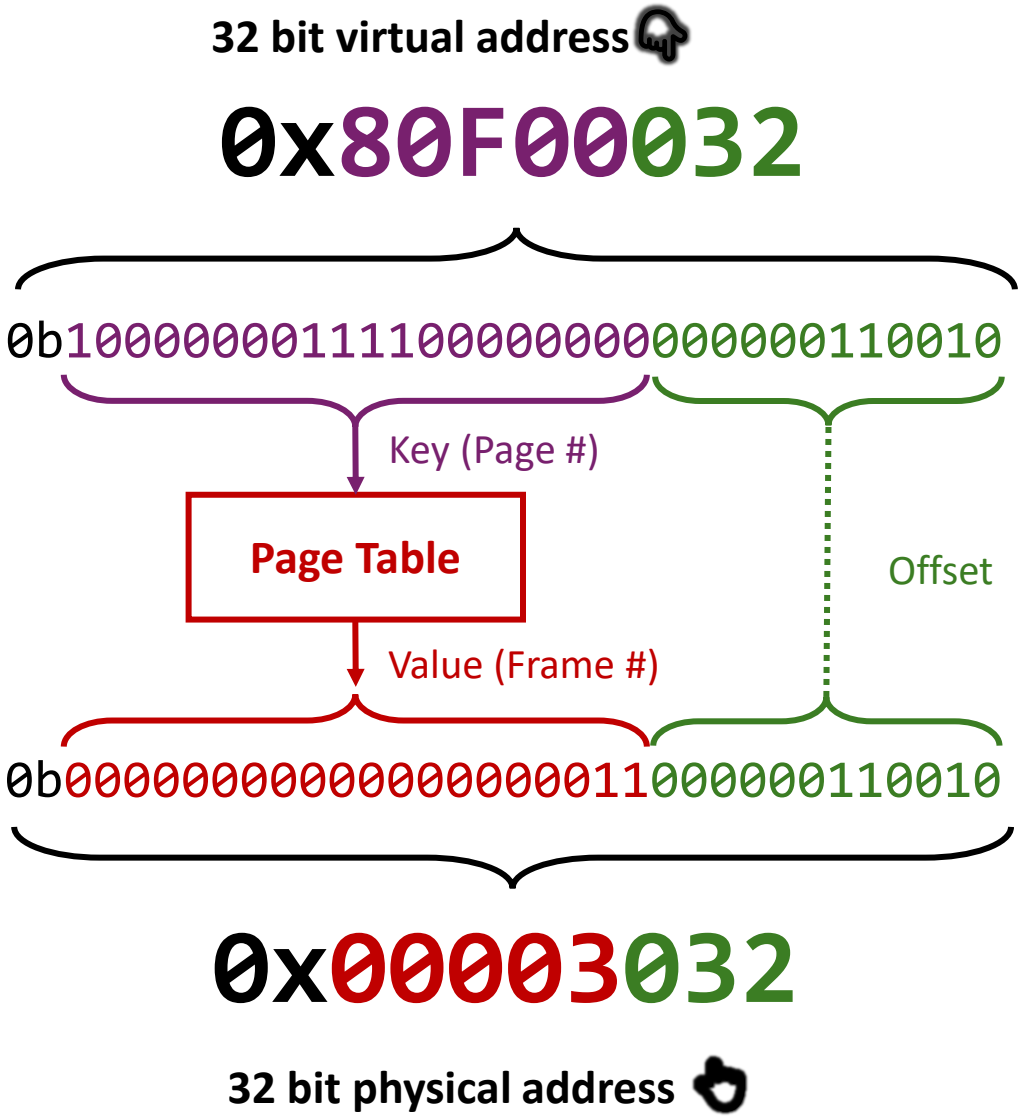


Class concluded here on 10/28 and will resume on 10/30



Paged Address Translation

- Split physical memory into groups of N bytes called “**frames**”
- Lower bits of virtual address directly address into the frame (eg, act as an “**offset**”)
- Upper bits are used as a key in a lookup table for the rest of the physical address (the “**frame number**”)
- That lookup table is called the **page table.** The keys are called **page numbers**. The entries are called pages. **Pages map virtual addresses to physical frames.**



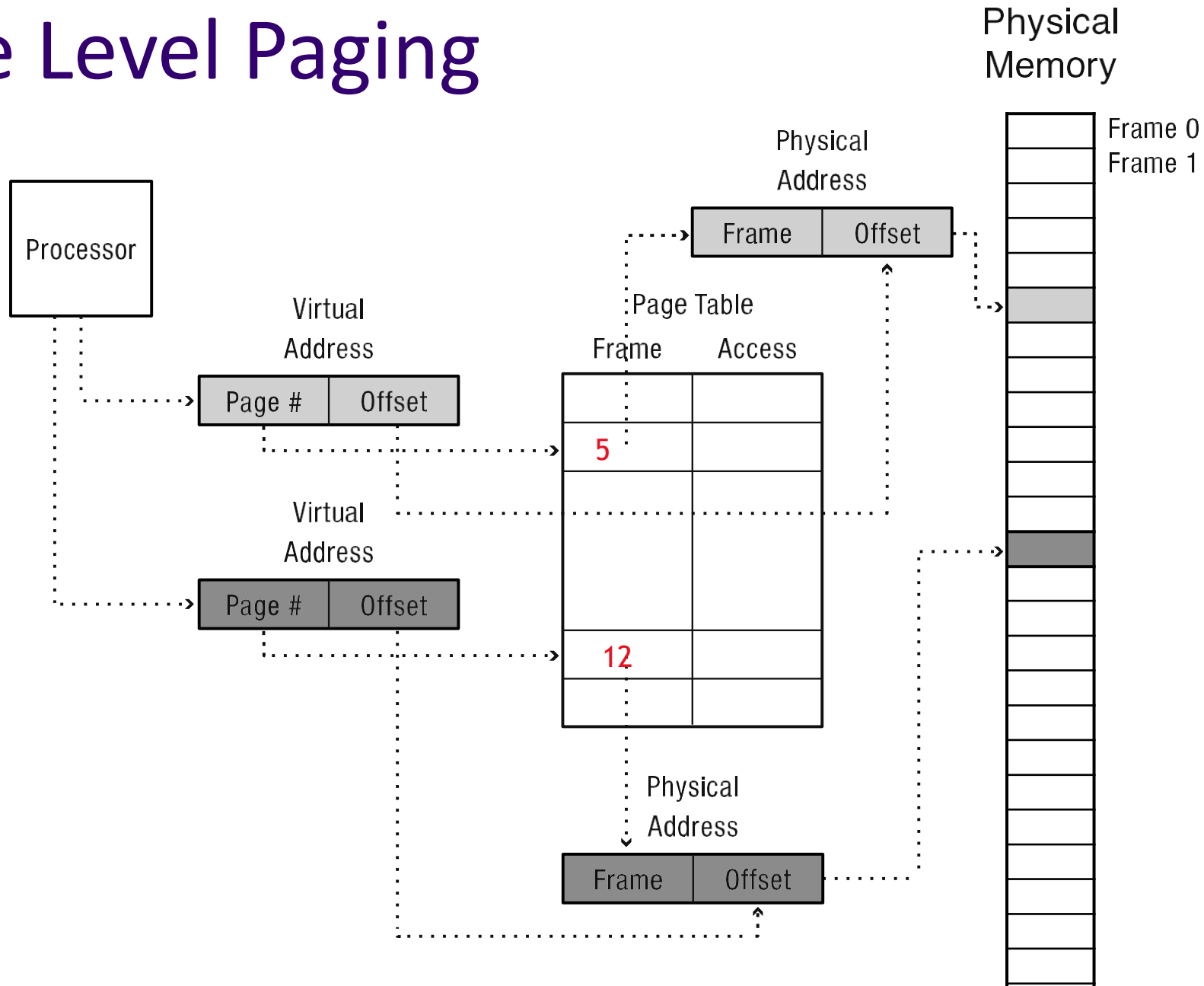
Page table, huh?

- A pretty large data structure (there's a lot of memory out there!)
  - We need a **different page table for each process**, since the page table effectively dictates the virtual address space
  - Page tables *themselves* **live in RAM**
  - **Format** is specified by **CPU architecture**

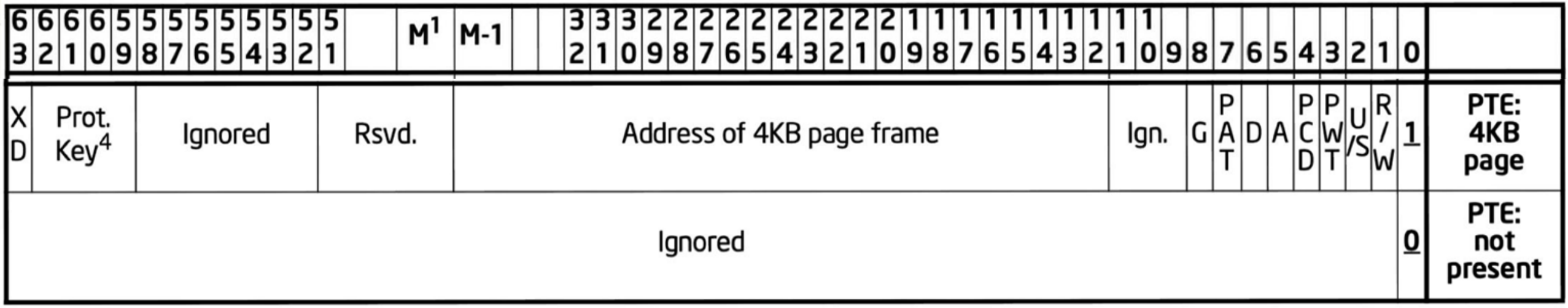
- **OS's job:**
  - **Write page tables** somewhere in physical memory
  - **Point** hardware to base address of our desired page table (on x86, the "**CR3**" **register**)

- **Hardware's job:**
  - Run all memory accesses through page table according to **address translation** algorithm

Single Level Paging



Page Table Entry (Intel 64 bit)



PTE not present: Unused virtual pages do not need a physical page frame

- Trap into kernel if process references memory on that page

Interesting PTE bits

- **0 (“P”):** entry valid?
- **1 (“R/W”):** read-only if bit is 0 (useful for code pages)
- **2 (“U/S”):** permission level (if 0, only kernel is allowed to access the page)
- **5 (“A”):** indicates whether the data in this page has been used/ “accessed” **(set by hardware)**
- **6 (“D”):** indicates whether the data in this page has been *modified* **(set by hardware)**

Finding unused frames

- OS can get away with a brute force implementation:
  - Use a **bitmap** where each bit indicates if a physical frame is in use
  - Can use fancy x86 instructions to search for 0 bits
  - Number of bits scales reasonably with amount of installed physical RAM
    - 8 GB ram, 4kb frame size: 2,000,000 pages
      1 bit/page: 250 KB
    - Just need one frame bitmap for the entire machine
    - That's not too bad

Let’s take a look, shall we?

- On Linux:
  - Process virtual memory maps are inspectable!
  - Check out “`/proc/[PID]/maps`” and “`/proc/[PID]/pagemaps`”
  - See also:
    - man proc_pid_pagemap
    - Kernel docs: https://www.kernel.org/doc/html/latest/filesystems/proc.html

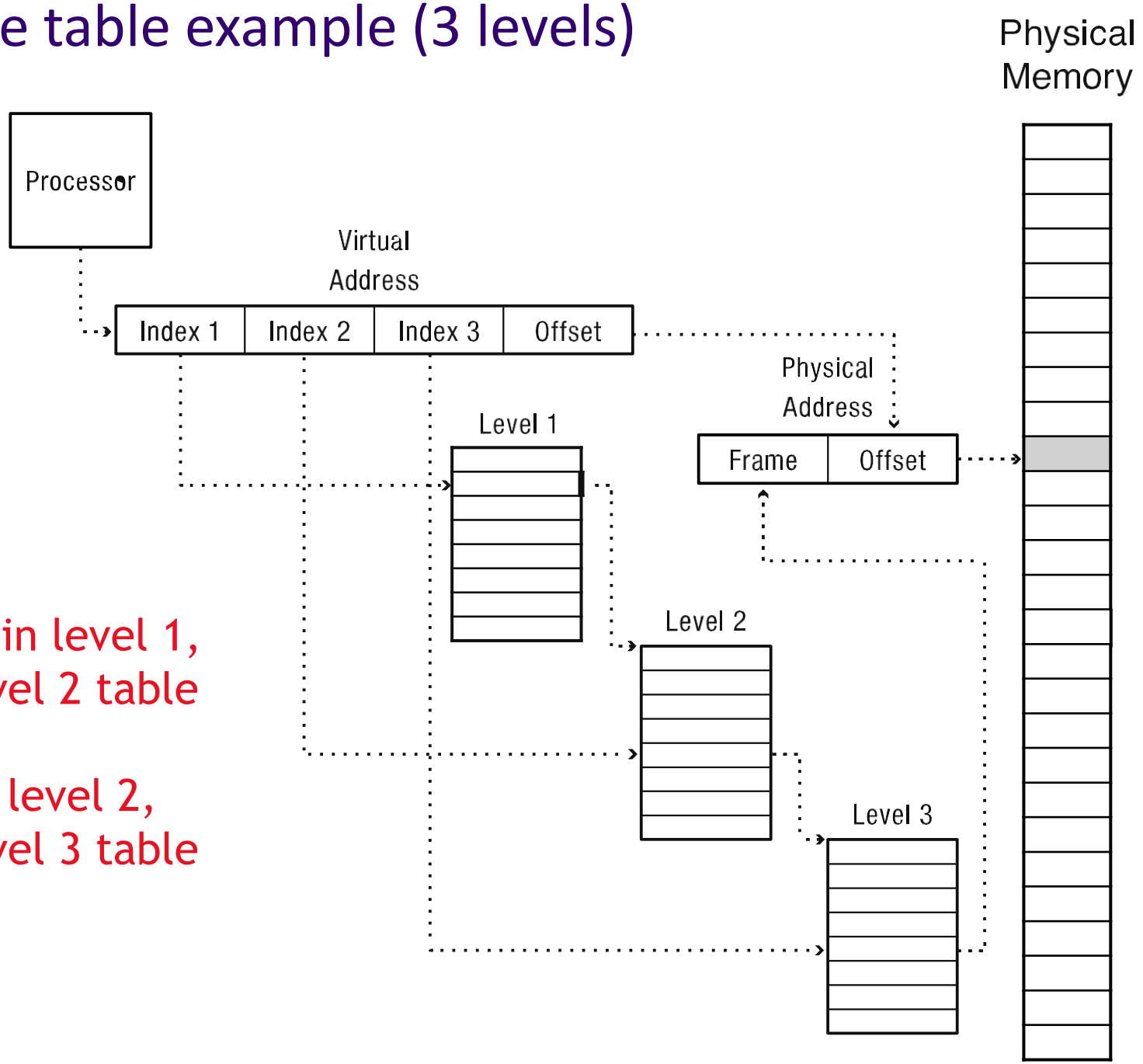
Paging Questions

- With paging, what is saved/restored on a process context switch?
- What are the tradeoffs of a small page size?
- What are the tradeoffs of a large page size?

Tradeoffs: Single Level Page Table

- Pros?
  - Simple memory allocation
  - Safe
  - Efficient as long as page size not too large (internal fragmentation)
  - Page permission can prevent overwriting code by accident
  - Can share physical page frames (two copies of same program)
  - Unused virtual pages don’t need to be backed by physical frames
- Cons?
  - An extra memory lookup on every memory reference
  - Need page table to map entire virtual address space
  - Page table itself needs to be contiguous in memory (fragmentation)

Multi-level page table example (3 levels)



If PTE is invalid in level 1, don't need a level 2 table

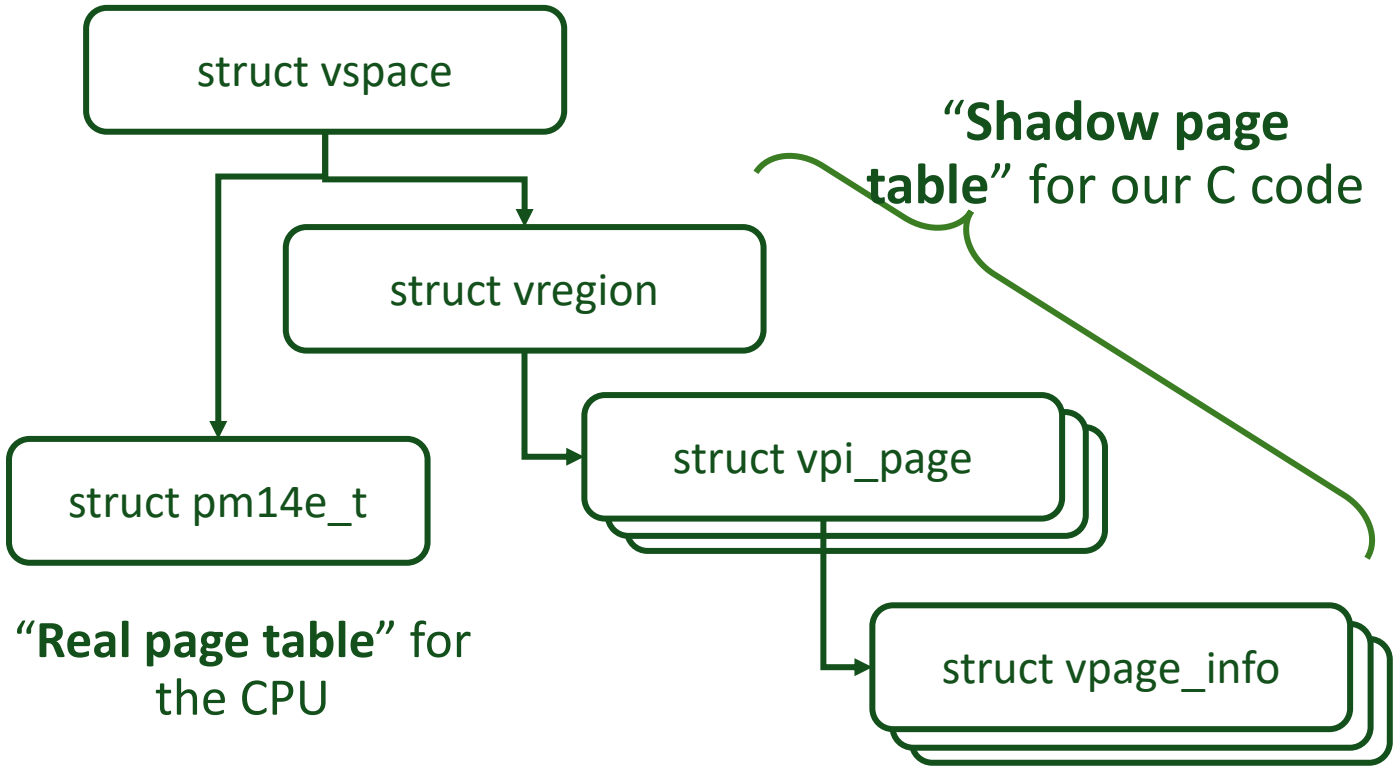
If it is invalid in level 2, don't need a level 3 table

x86 Multilevel Paging

- Each level of page table fits in one 4KB page
- 32-bit: **two level** page table
  - Page table entry is 32 bit, so 2^10 entries per page
- 64-bit: **five level** page table
  - Page table entry is 64 bit, so 2^9 entries per page
- Omit sub-tree if page table entry is invalid
  - Good for sparse address space

Oh hey Gritty, it's been a while

- xk: inc/vspace.h, kernel/vspace.c





Walking the hardware page table in xk

- See:
  - kernel/x86_vm64.c : `walkpm14()`
  - inc/mmu.h
  - inc/memlayout.h



Hmm…



…why do we bother keeping a shadow page table? Why not just use the real one?

Page sharing between processes

- Set page table entry in each process to point to same page frame (4KB)
  - Or pointer to page table (both point to same page table) (2MB)
  - Or pointer to pointer to page tables (both point to same page directory) (1GB)
  - Or pointer to pointer to pointer to page tables (0.5TB)
    - Example: kernel portion of address space

- This means we can't just use a bitmap to track used frames
  - Upgrade to a structure called a "**core map:**" an array of information about each physical page frame
    - Set of processes pointing to that page frame
    - Reference count (when it goes to zero, reclaim!)

Tradeoffs: Multilevel Paging

Pros

- Efficient memory allocation
- Efficient for sparse addresses
- Efficient disk transfers
- Easier to build translation lookaside buffers
- Variable granularity for protection/sharing

Cons

- Every memory access involves 6 memory reads in the general case