CS162
Operating Systems and Systems Programming
Lecture 16

Memory 3: Demand Paging

Professor Natacha Crooks & Matei Zaharia
https://cs162.org/

Slides based on prior slide decks from David Culler, Ion Stoica, John Kubiatowicz, Alison Norman and Lorenzo Alvisi

Are we done?

How big can a page table get on x86 (32 bits)?

4KB page => 2^12
2^32/2^12 => 2^20 pages
2^20 * 4 bytes = 4 MB (approx.)
That's (not) a lot per process!!

How big can a page table get on x86 (64 bits)?

4KB page => 2^12
2^64/2^12 => 2^52 pages
2^20 * 8 bytes = 36 petabytes (approx.)
That's a lot per process!!

Limitations of paging

Space overhead

With a 64-bit address space, size of page table can be huge

Time overhead

Accessing data now requires two memory accesses must also access page table, to find mapped frame

Internal Fragmentation

4KB pages

The Secret to the Whole of CS

Batching

Caching

Indirection

Specialised Hardware

Sparsity

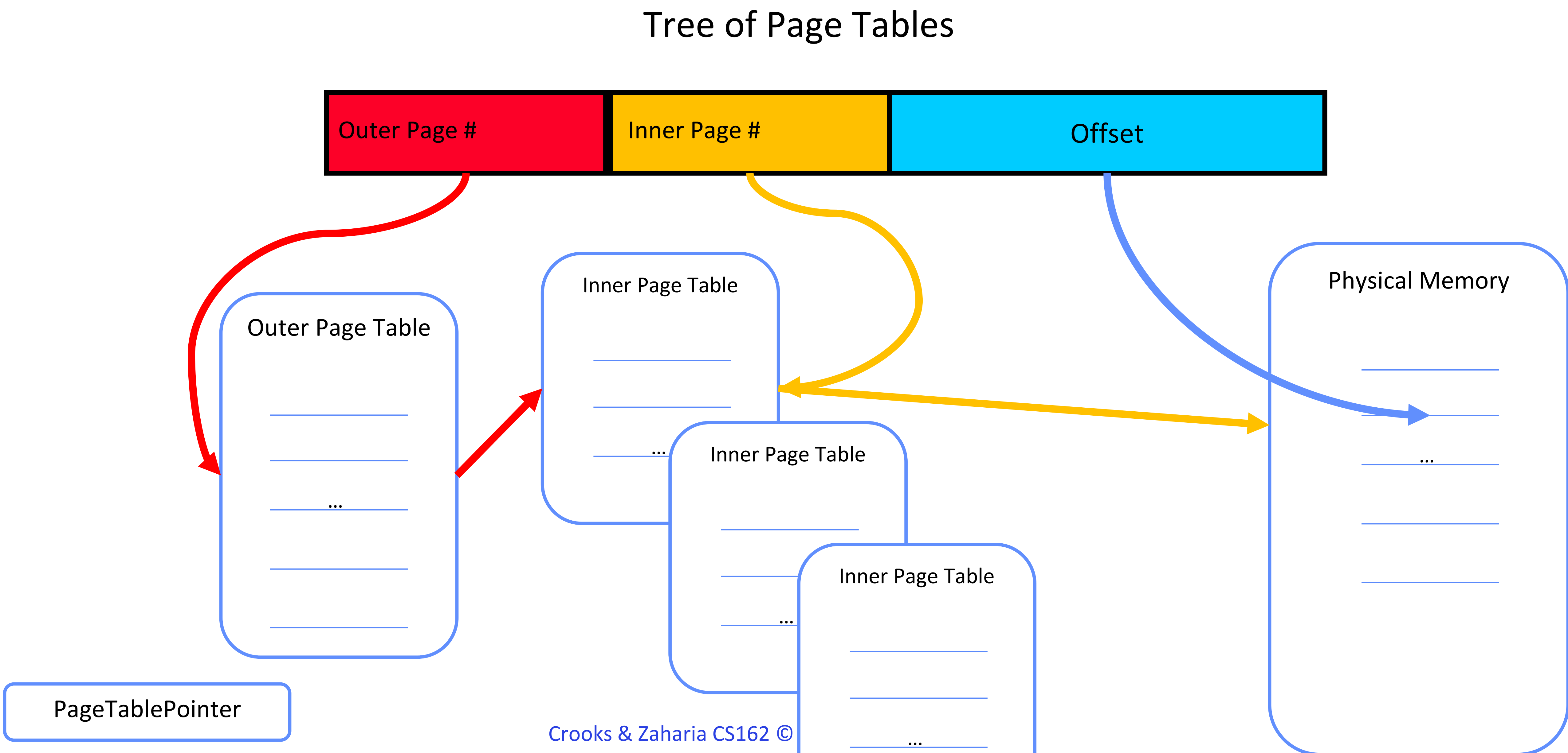
Address space is sparse, i.e. has holes that are not mapped to physical memory

Most this space is taken up by page tables mapped to nothing

Process has access to full 2^64 bytes (virtually)

Physically, that would be 17,179,869,184 gigabytes

Paging the page table: 2-level paging



V2: What is a page table entry? (32 bits)

Inner Page Table VAddress or PFN	Free (OS)	0	PS	D	A	PCD	PWT	U	W	P
31-12	11-9	8	7	6	5	4	3	2	1	0

P: Present (same as "valid" bit in other architectures)

W: Writeable

U: User accessible

PWT: Page write transparent: external cache write-through

PCD: Page cache disabled (page cannot be cached)

A: Accessed: page has been accessed recently

D: Dirty: page has been modified recently

PS: Page Size

Paging the page table: 2-level paging

Tree of Page Tables

Outer Page #	Inner Page #	Offset
Number of top-level pages	Ensure that fits on a single page	Defines size of a page

Paging the page table: 2-level paging

Tree of Page Tables

Outer Page #	Inner Page #	Offset
10 bits	10 bits	4 KB 12 bits

Want to make sure that inner page table fits in a page!

2^12/2^2 = 2^10

Example: x86 classic 32-bit address translation

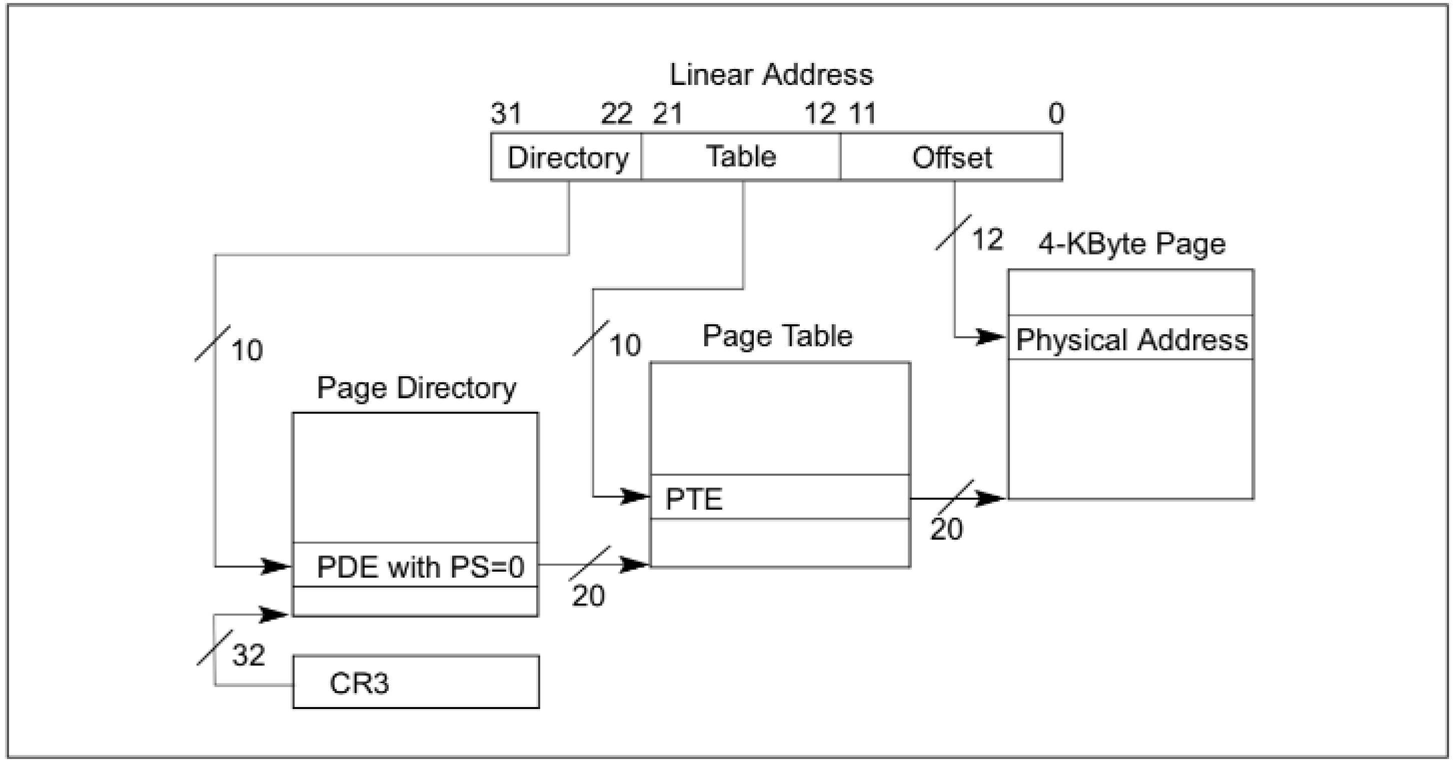
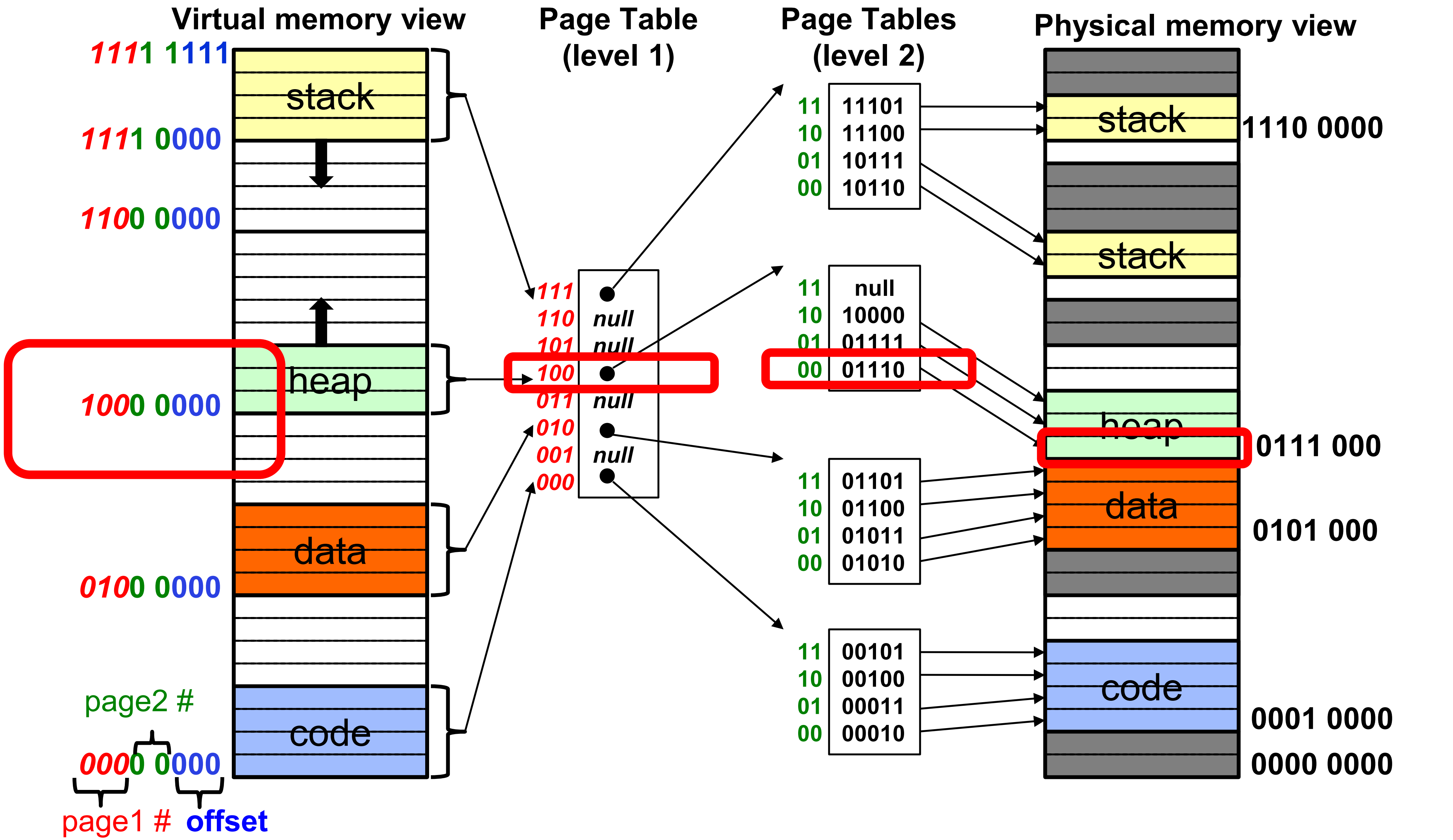


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

Top-level page-table: Page Directory

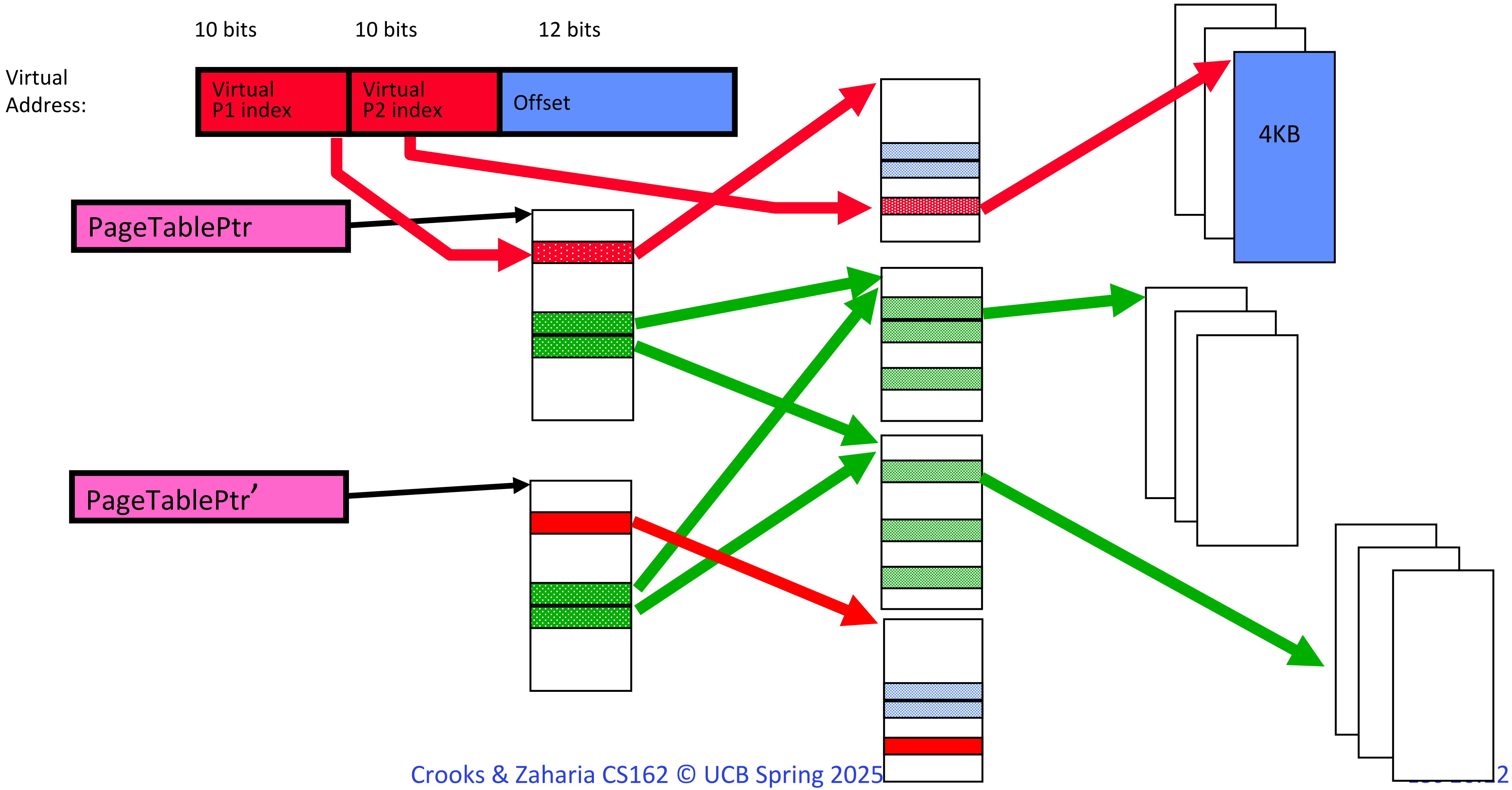
Inner page-table: Page Directory Entries

Example Address Space View



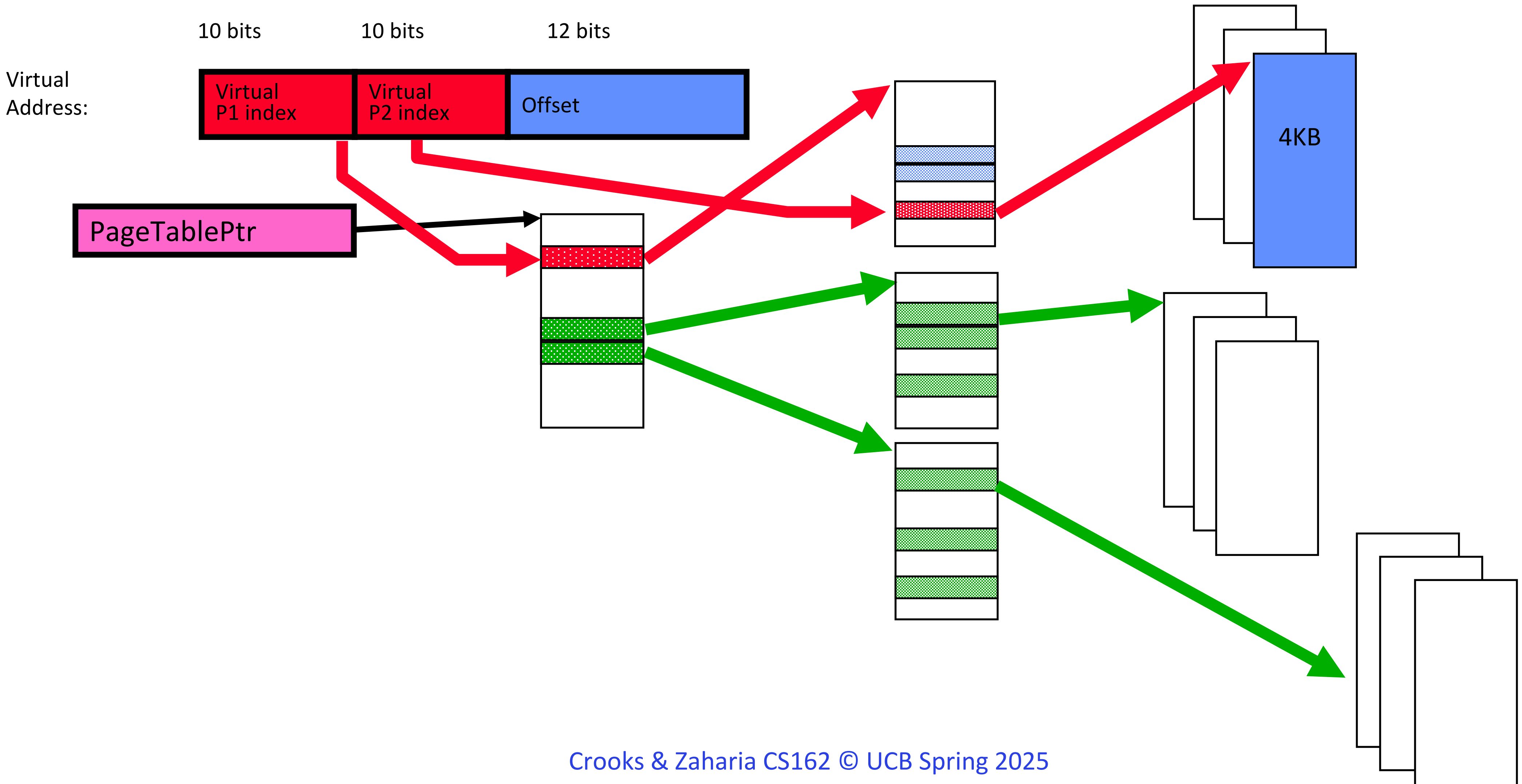
Sharing with multilevel page tables

Entire regions of the address space can be efficiently shared



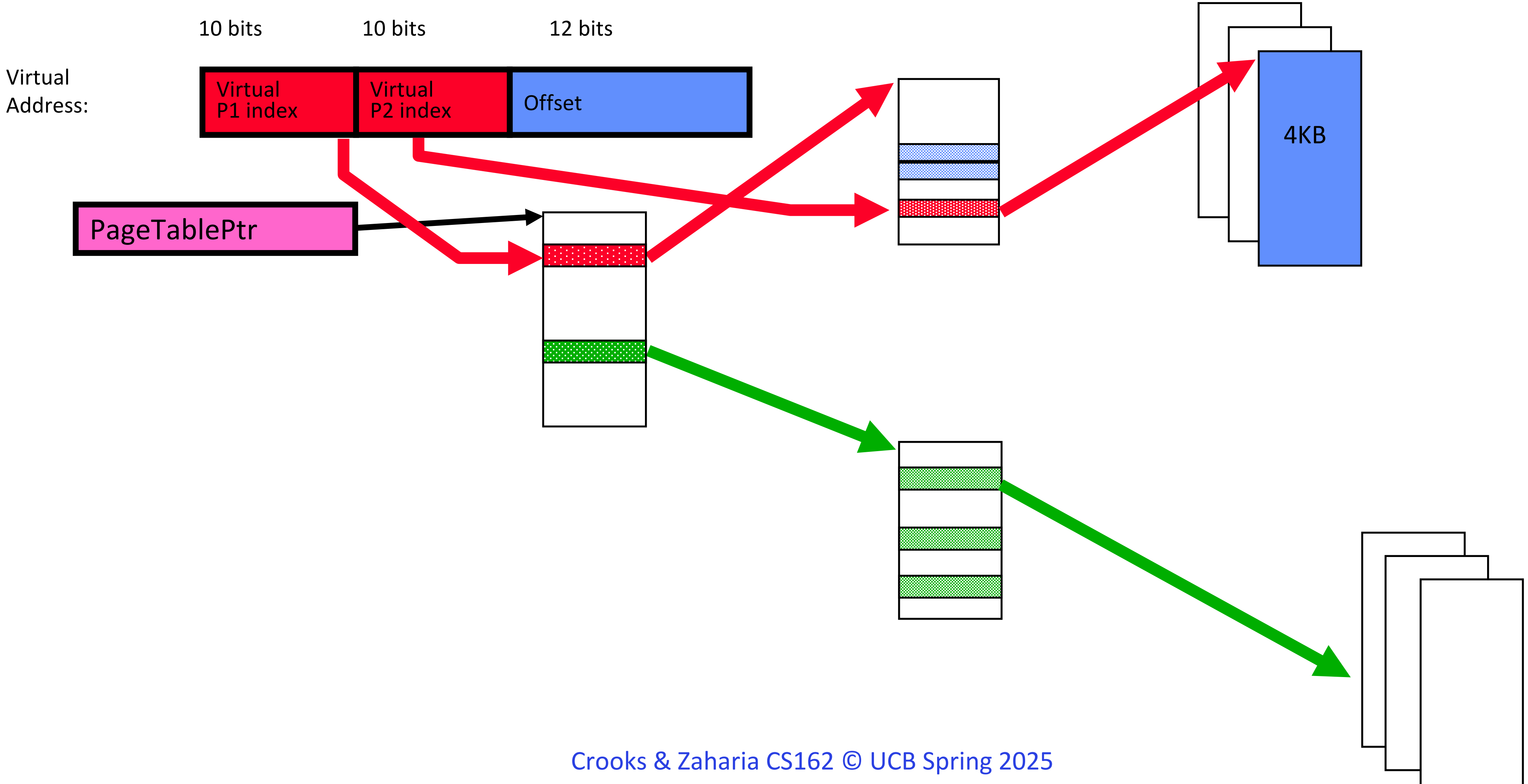
Marking entire regions as invalid!

If region of address space unused, can mark entire inner region as invalid



Marking entire regions as invalid!

If region of address space unused, can mark entire inner region as invalid



Has this helped?

Assuming 10/10/12 split:

Size of Page Table

Outer: (2^10 * 4 bytes) +
Inner: 2^10 * (2^10 * 4 bytes)

Overhead of indirection! BUT Marking inner pages as invalid helps when address spaces are sparse

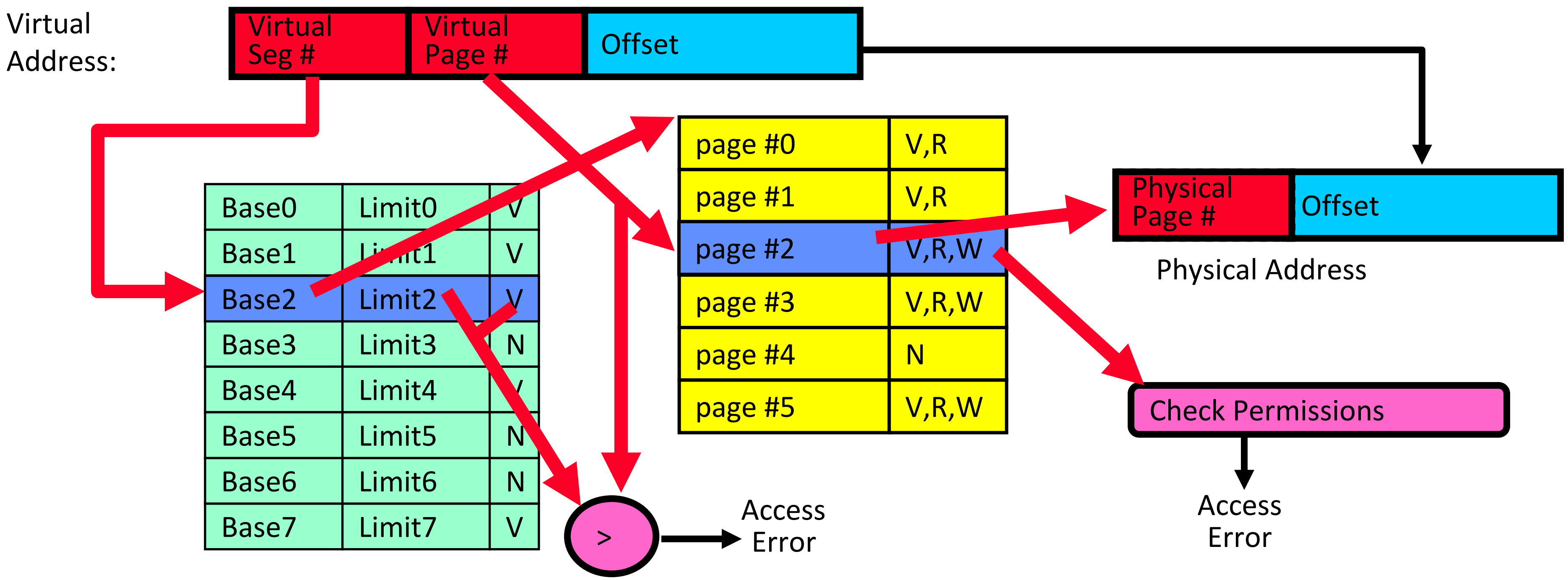
Downside: now have to do

two memory accesses for translation

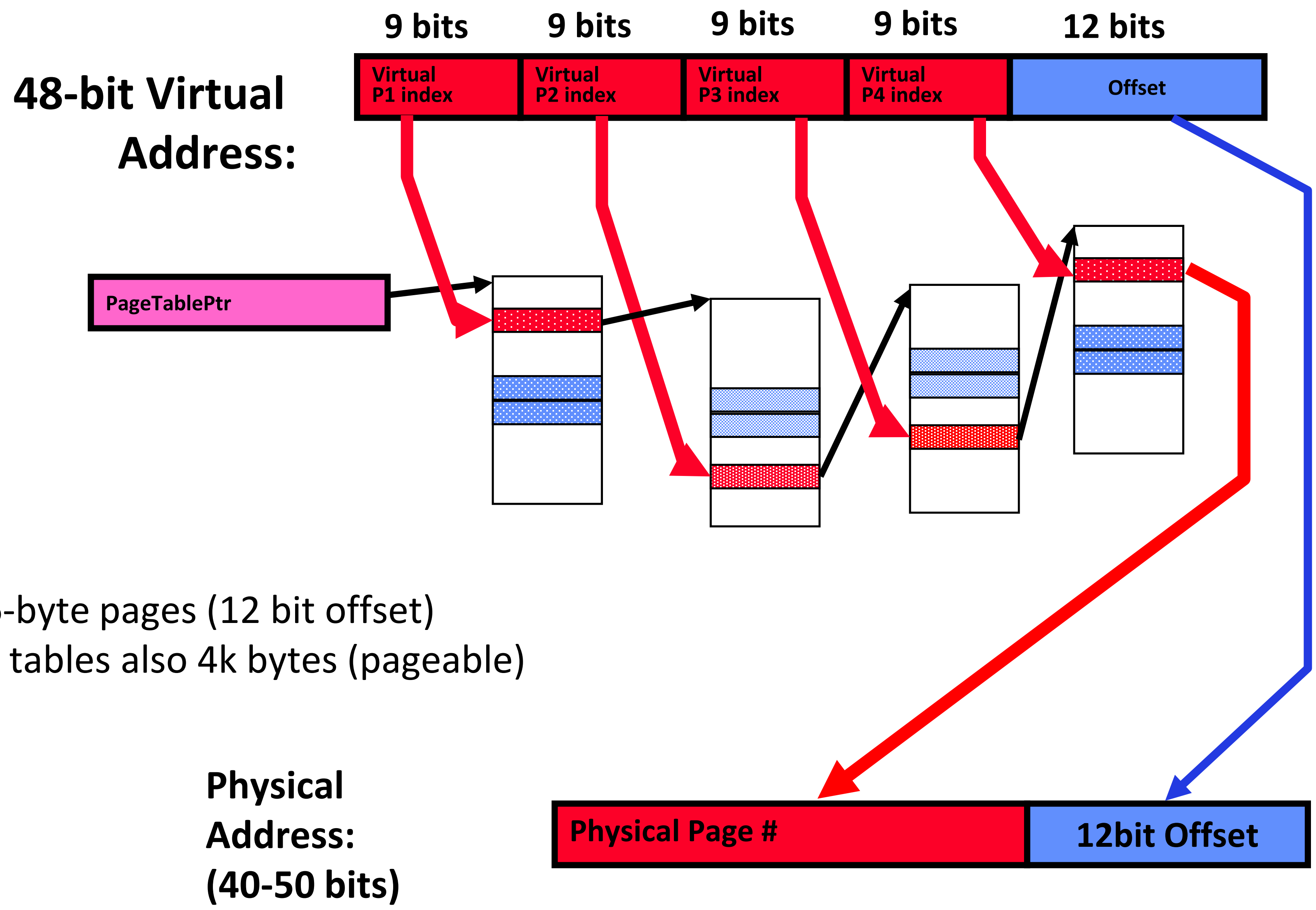
Paged Segmentation

Use segments for top level. Paging within each segment.

Used in x86 (32 bit).
Code Segment, Data Segment, etc.



X86 64 bits has a four-level page table!



4096-byte pages (12 bit offset)
Page tables also 4k bytes (pageable)

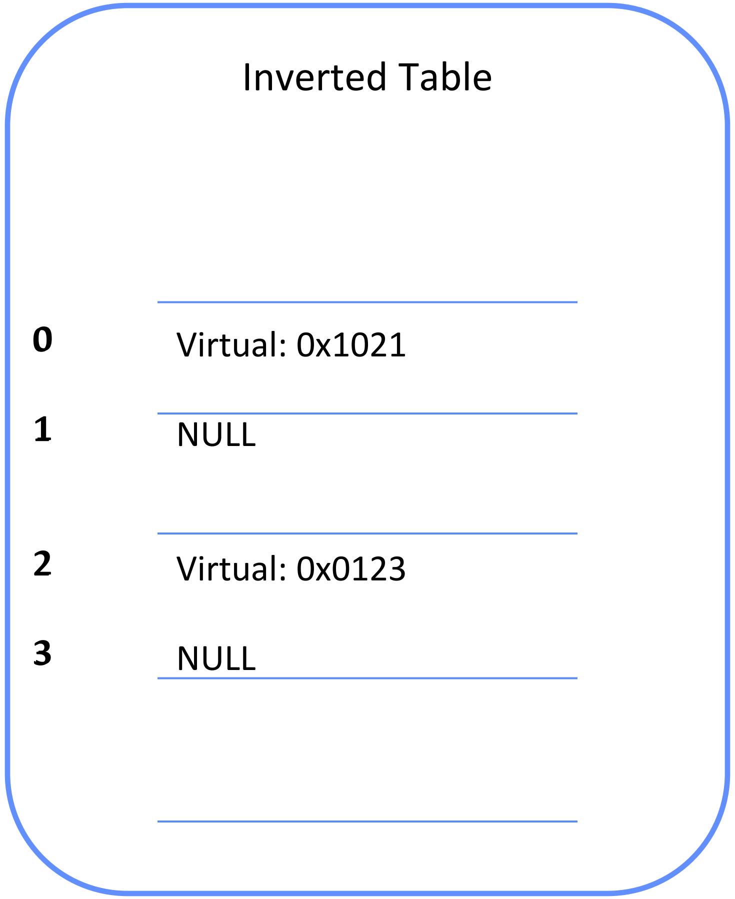
Inverted Page Table

A single page table that
has an entry for each physical page of the system

Each entry contains process ID + which virtual page maps to physical page

Physical memory much smaller than virtual memory

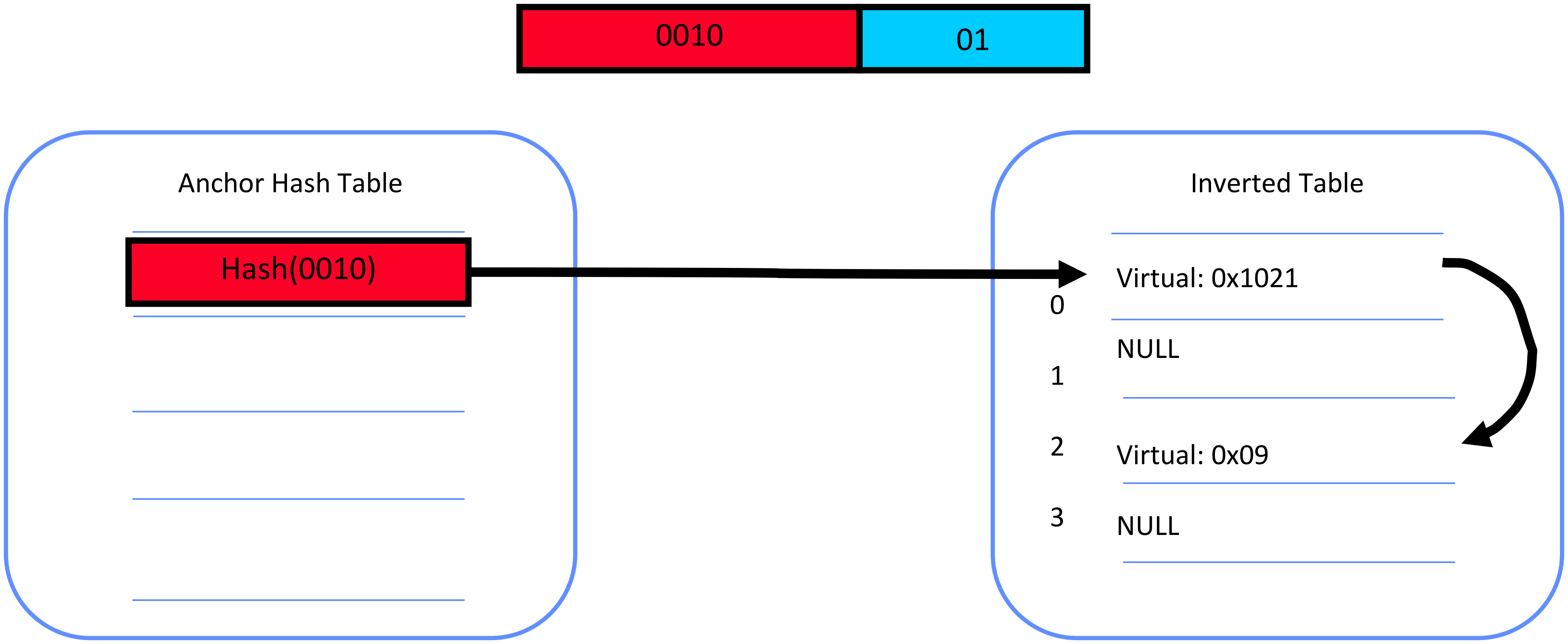
Size proportional to size of physical memory



Inverted Page Table

Don’t we have it backwards?

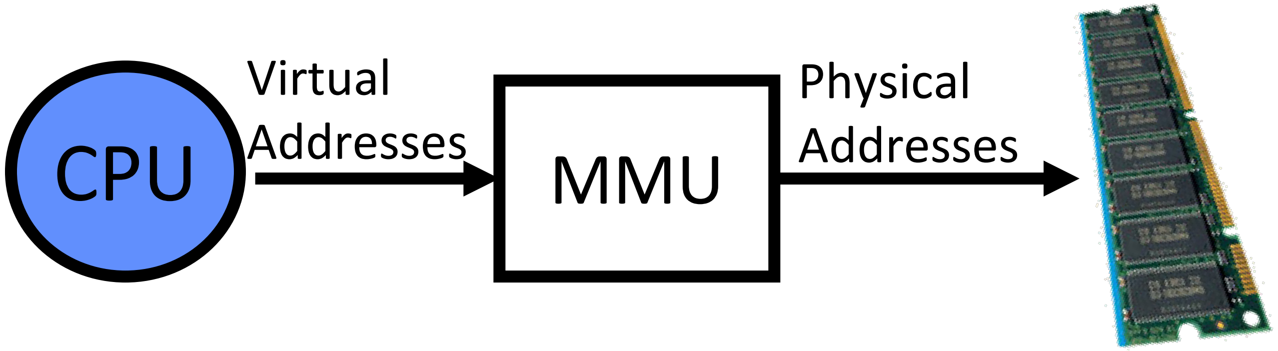
Add a hash table. Virtual memory can only map to specific physical frames



Address Translation Comparison

	Advantages	Disadvantages
Simple Segmentation	Fast context switching (segment map maintained by CPU)	External fragmentation
Paging (Single-Level)	No external fragmentation Fast and easy allocation	Large table size (~ virtual memory) Internal fragmentation
Paged Segmentation Multi-Level Paging	Table size ~ # of pages in virtual memory Fast and easy allocation	Multiple memory references per page access
Inverted Page Table	Table size ~ # of pages in physical memory	Hash function more complex No cache locality of page table

How is the Translation Accomplished?



MMU must translate virtual address to physical address on every instruction fetch, load or store

What does the MMU need to do to translate an address?
Read, check, and update PTE
(set accessed bit/dirty bit on write)

How can we speedup translation?

MMU must make at least 2 memory reads to walk page table. Slow!

Use specialized hardware to
cache virtual-physical memory translations!

Introducing the Translation Lookaside Buffer (TLB)

Recall: CS61c Caching Concept

Cache: a repository for copies that can be accessed more quickly than the original

Only good if:

Frequent case frequent enough and

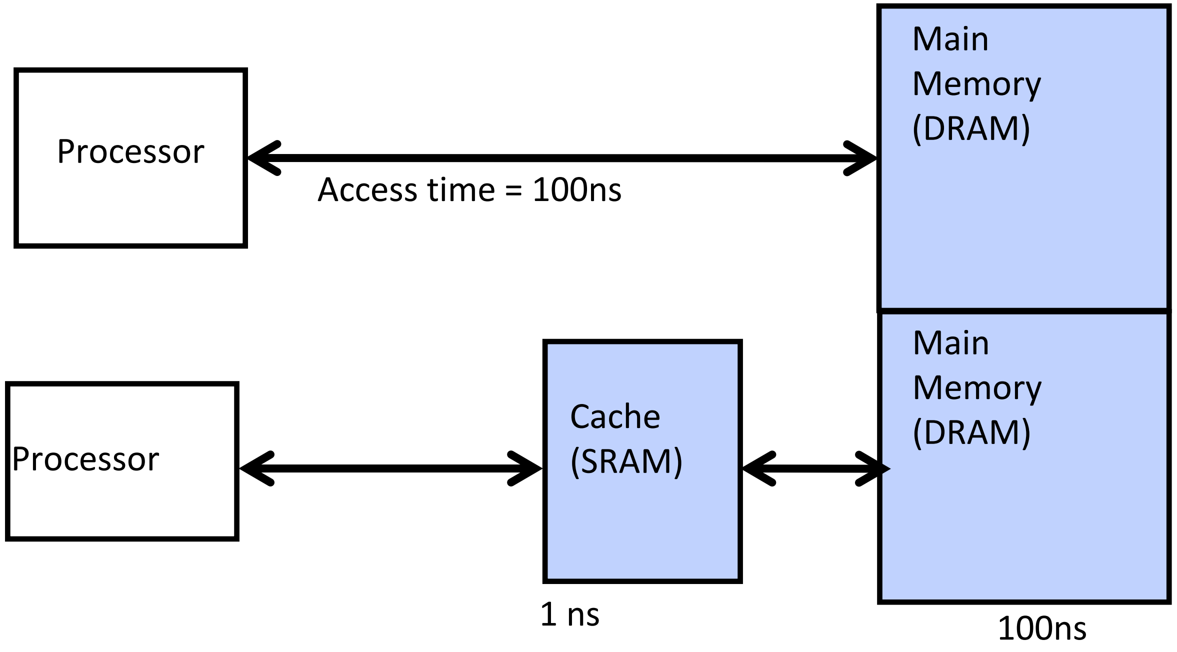
Infrequent case not too expensive

Important measure

Average Access time =
(Hit Rate x Hit Time) + (Miss Rate x Miss Time)

Recall: In Machine Structures (eg. 61C) …

Caching is the key to memory system performance



Recall: In Machine Structures (eg. 61C) …

Average Memory Access Time (AMAT)

= (Hit Rate x HitTime) + (Miss Rate x MissTime)

Where HitRate + MissRate = 1

HitRate = 90% => AMAT = (0.9 x 1) + (0.1 x 101)=11 ns

HitRate = 99% => AMAT = (0.99 x 1) + (0.01 x 101)=2.01 ns

MissTime_{L1} includes

$\text{HitTime}_{L1}+\text{MissPenalty}_{L1} \equiv \text{HitTime}_{L1} + \text{AMAT}_{L2}$

Why Does Caching Help? Locality!

Temporal Locality (Locality in Time):

Keep recently accessed data items closer to processor

Spatial Locality (Locality in Space):

Move contiguous blocks to the upper levels

Recall: Memory Hierarchy

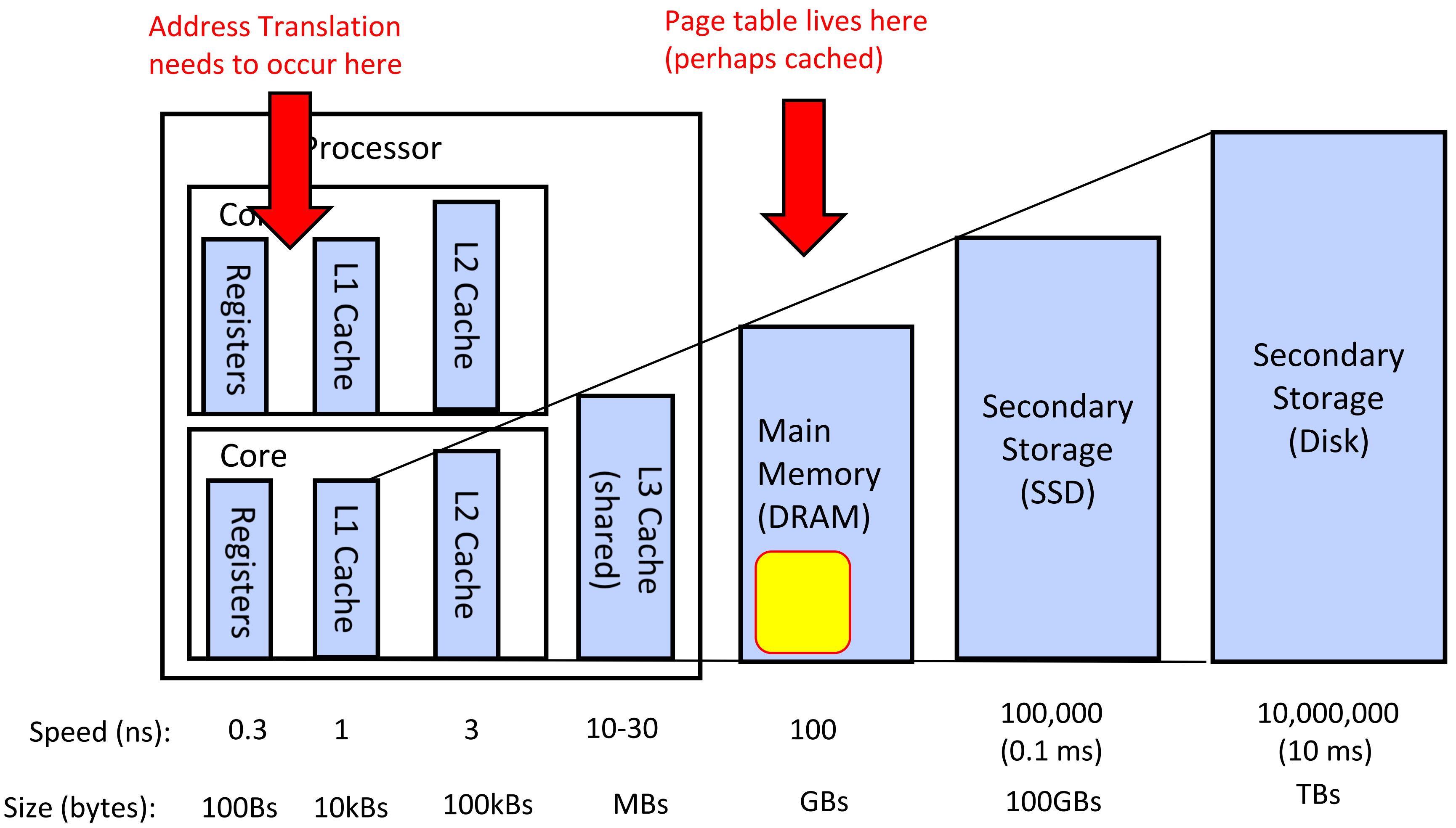
Take advantage of the principle of locality to:

1) Present the illusion of having as much memory as in the cheapest technology

2) Provide average speed similar to that offered by the fastest technology

Recall: fast but small/expensive. Slow but large!

Recall: Memory Hierarchy



Translation Look-Aside Buffer

Record recent Virtual Page # to Physical Frame # translation

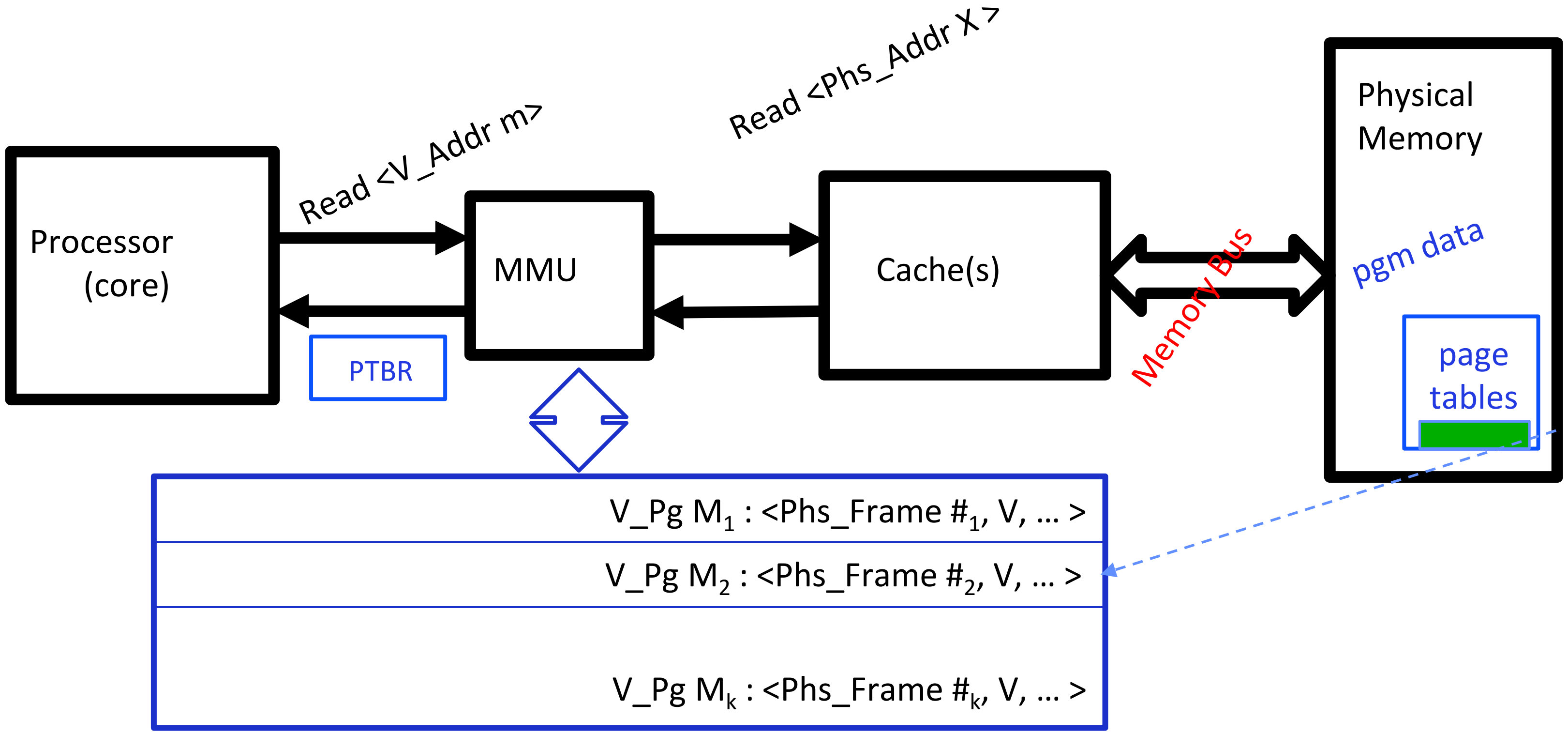
If present, have the physical address without reading any of the page tables !!!

Caches the end-to-end result

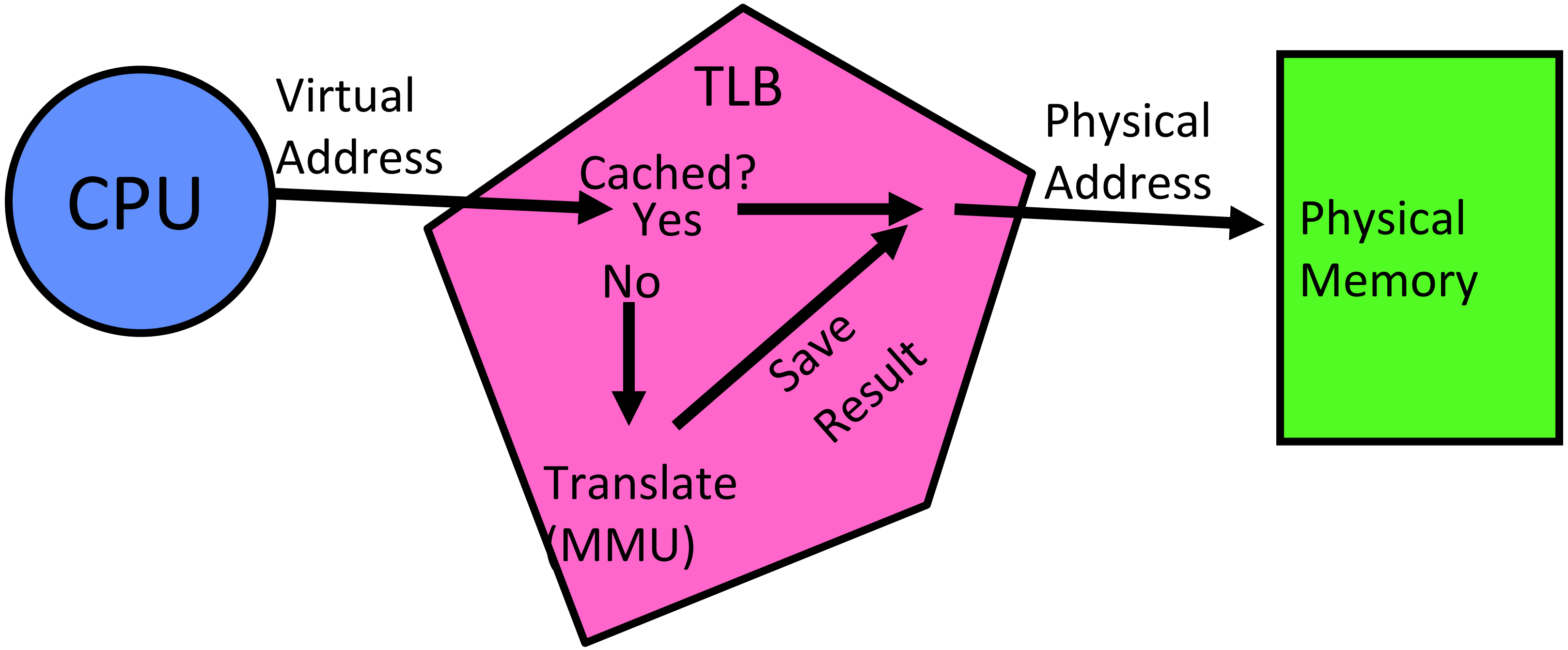
How do we make Address Translation Fast?

Cache results of recent translations !

Cache Page Table Entries using Virtual Page # as the key



Caching Applied to Address Translation



Does page locality exist?

Instruction accesses spend a lot of time on the same page (since accesses sequential)

Stack accesses have definite locality of reference

What happens on a Context Switch?

Need to do something, since TLBs map virtual addresses to physical addresses

- – Address Space just changed, so TLB entries no longer valid!

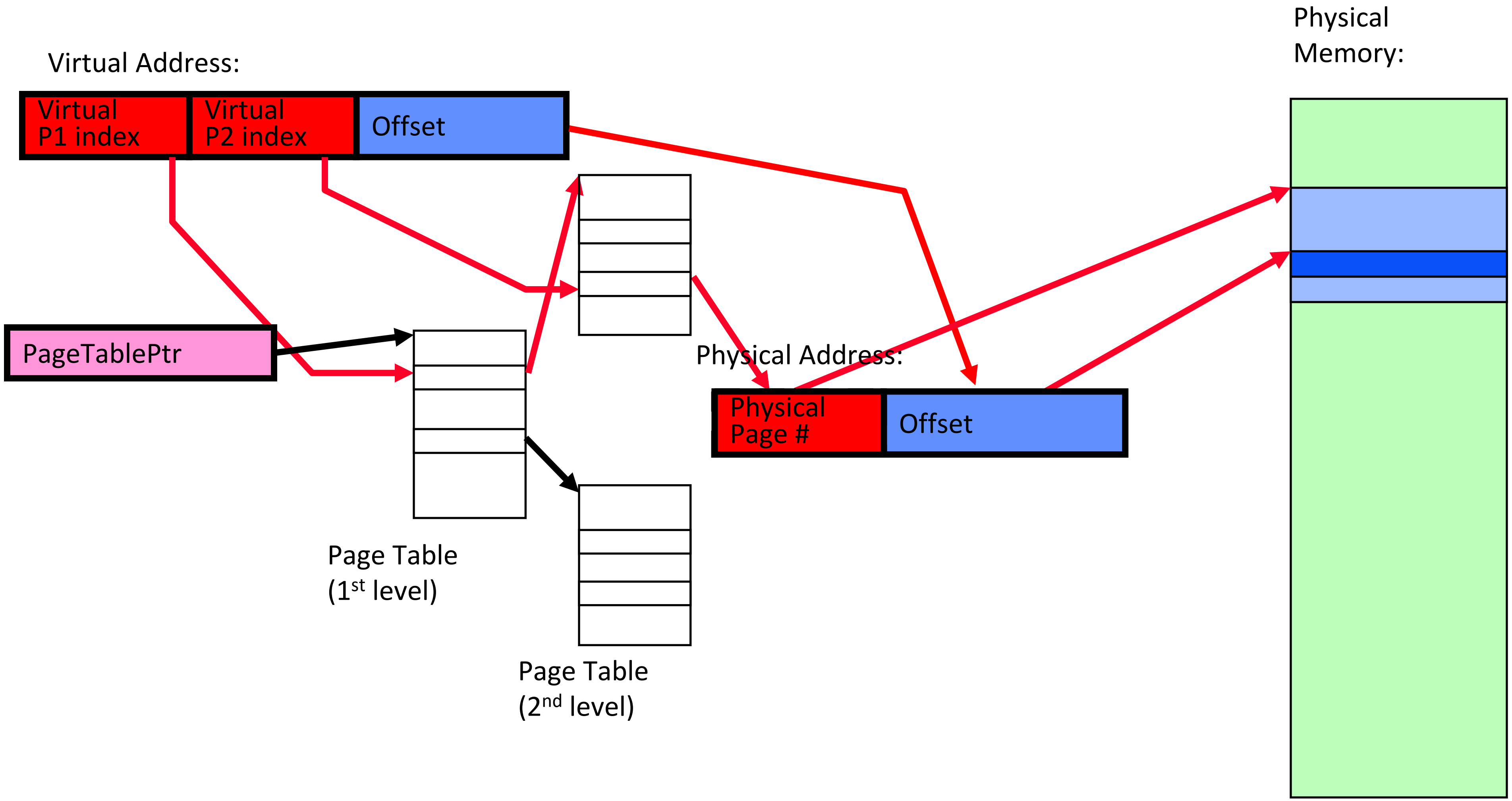
Options?

- Invalidate (“Flush”) TLB: simple but might be expensive
  - What if switching frequently between processes?
- Include ProcessID in TLB
  - This is an architectural solution: needs hardware

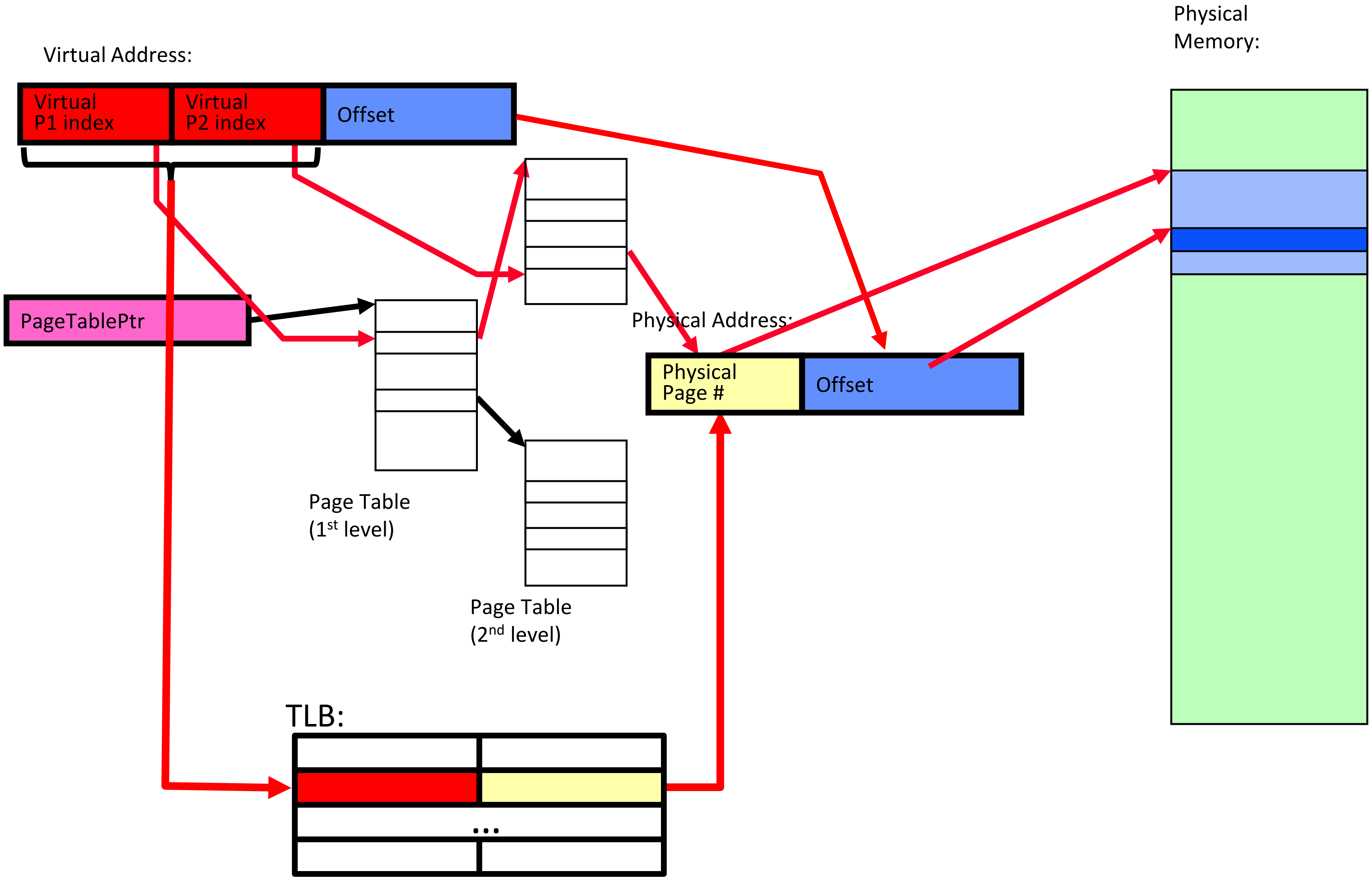
What if translation tables change?

- For example, to move page from memory to disk or vice versa…
- Must invalidate TLB entry!
  - Otherwise, might think that page is still in memory!
- Called “TLB Consistency”

Putting Everything Together: Address Translation



Putting Everything Together: TLB



Page Fault Handling

The Virtual-to-Physical Translation fails

– PTE marked invalid, Privilege Level Violation, Access violation, or does not exist

Causes a Fault / Trap

May occur on instruction fetch or data access

Other Page Faults engage operating system to fix the situation and retry the instruction

– Allocate an additional stack page, or

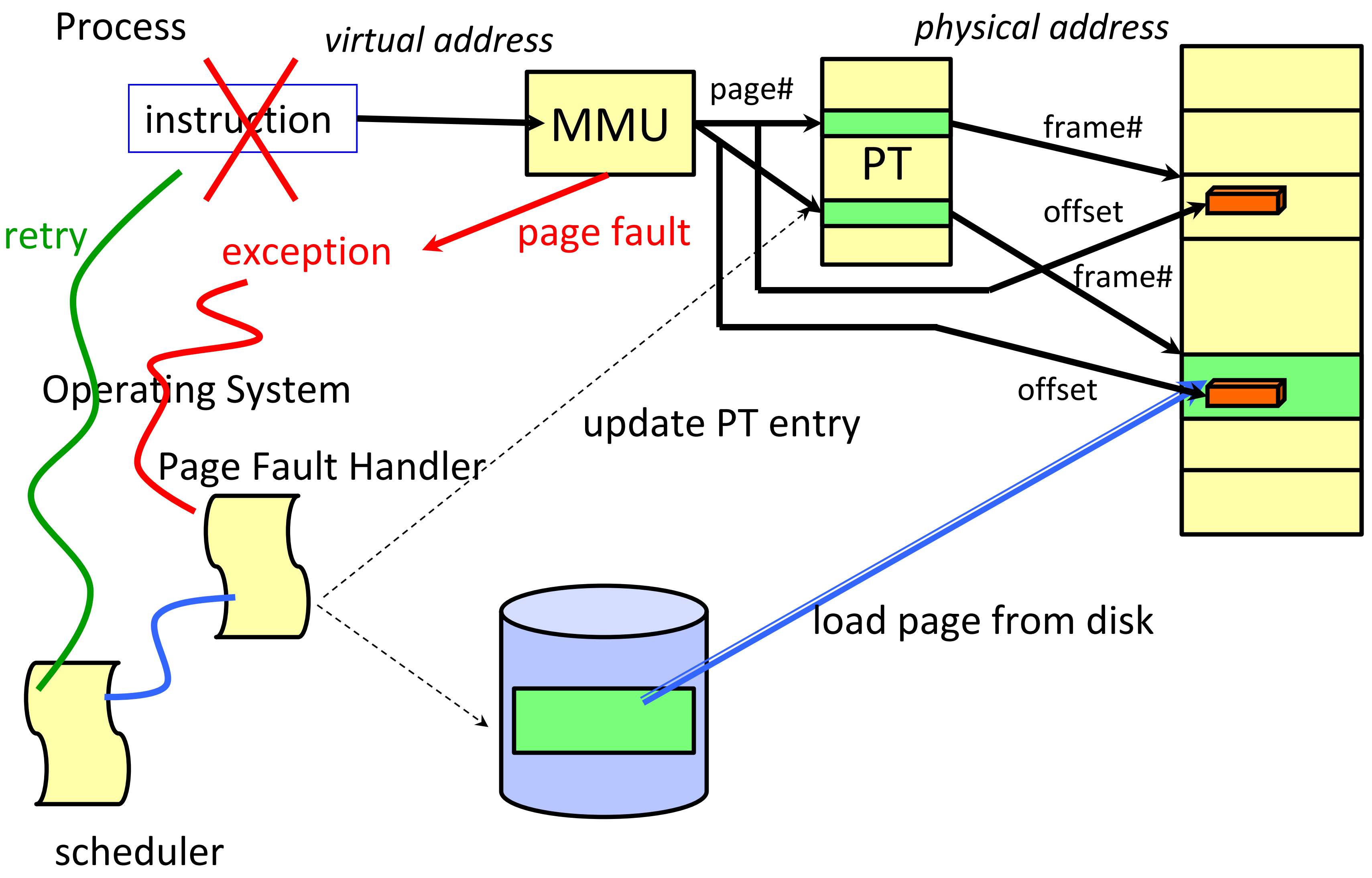
– Make the page accessible – (Copy on Write),

– Bring page in from secondary storage to memory – demand paging

Fundamental inversion of the hardware / software boundary

– Need to execute software to allow hardware to proceed!

Page Fault ⇒ Demand Paging



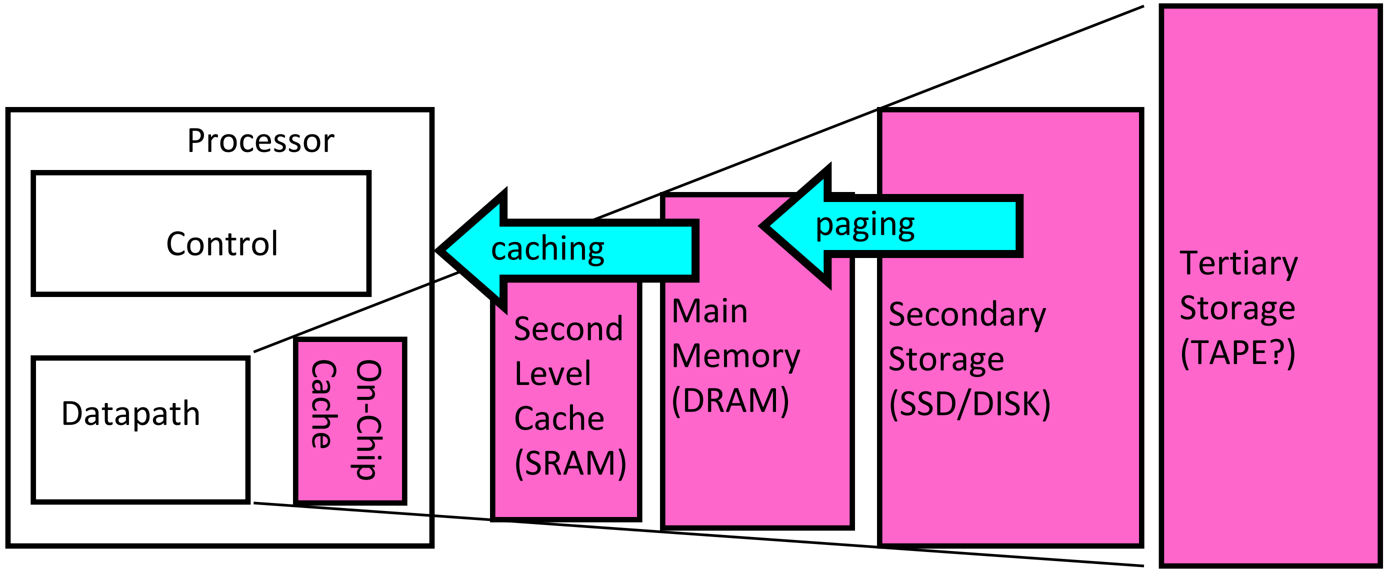
Demand Paging

Modern programs require a lot of physical memory

BUT they don't use all their memory all of the time

Wasteful to require all of user's code to be in memory

Solution: use main memory as "cache" for disk



Illusion of Infinite Memory

Disk is larger than physical memory

In-use virtual memory can be bigger than physical memory
Combined memory of running processes much larger than physical memory

More programs fit into memory, allowing more concurrency

Principle: Transparent Level of Indirection (page table)
Supports flexible placement of physical data

Performance issue, not correctness issue

Review: What is a page table entry? (32 bits)

Inner Page Table VAddress or PFN	Free (OS)	0	PS	D	A	PCD	PWT	U	W	P
31-12	11-9	8	7	6	5	4	3	2	1	0

P: **Present (same as "valid" bit in other architectures)**

D: **Dirty: page has been modified recently**

PTE makes demand paging implementable

– Valid ⇒ Page in memory, PTE points at physical page

– Not Valid ⇒ Page not in memory; use info in PTE to find it on disk when necessary

What happens on an invalid PTE?

1) Memory Management Unit (MMU) traps to OS
- » Resulting trap is a "Page Fault"

2) What does OS do on a Page Fault?:
- » Choose an old page to replace
- » If old page modified ("D=1"), write contents back to disk
- » Change its PTE and any cached TLB to be invalid
- » Load new page into memory from disk
- » Update page table entry, invalidate TLB for new entry
- » Continue thread from original faulting location

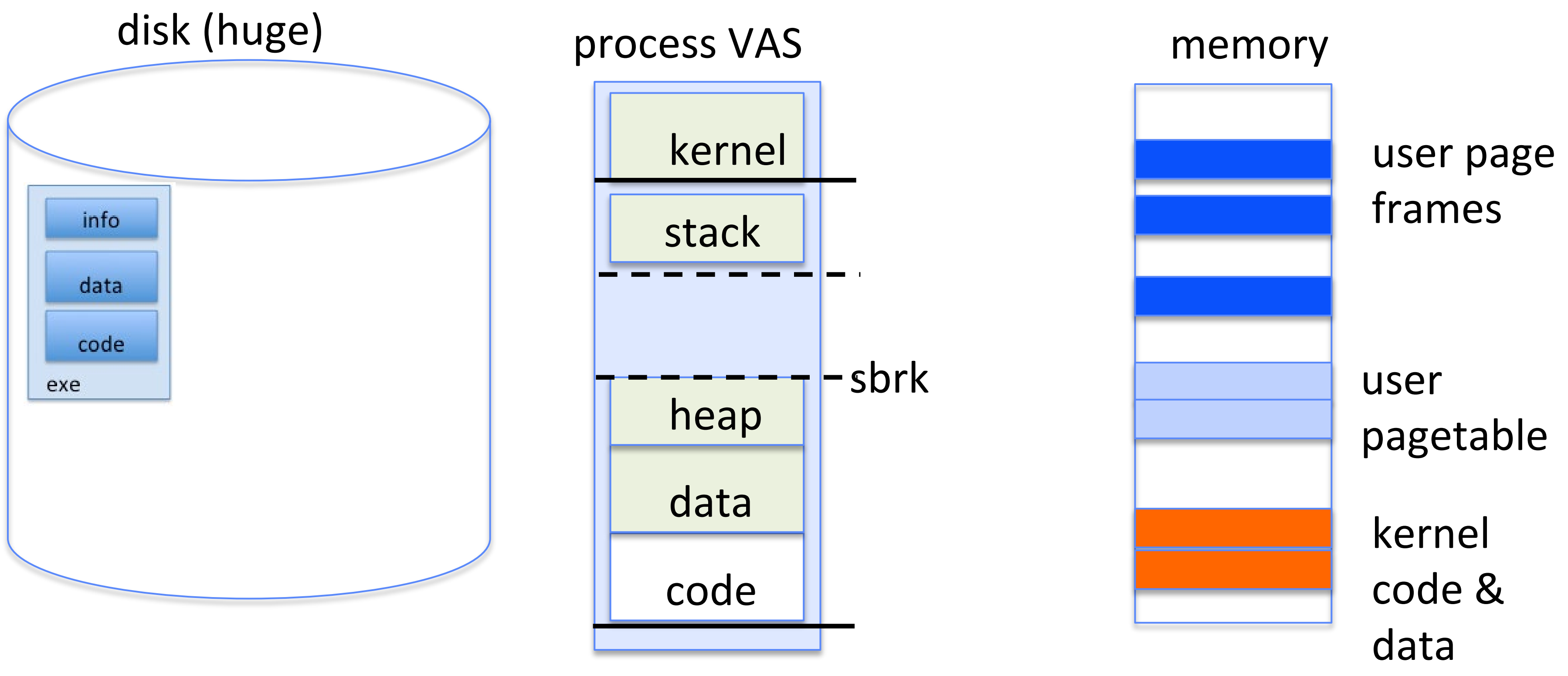
3) TLB for new page will be loaded when thread continues!

4) While pulling pages off disk for one process, OS runs another process from ready queue
- » Suspended process sits on wait queue

Many Uses of Virtual Memory and "Demand Paging" …

- Extend the stack
  - Allocate a page and zero it
- Extend the heap (sbrk of old, today mmap)
- Process Fork
  - Create a copy of the page table
  - Entries refer to parent pages – NO-WRITE
  - Shared read-only pages remain shared
  - Copy page on write
- Exec
  - Only bring in parts of the binary in active use
  - Do this on demand
- MMAP to explicitly share region (or to access a file as RAM)

Create Virtual Address Space of the Process

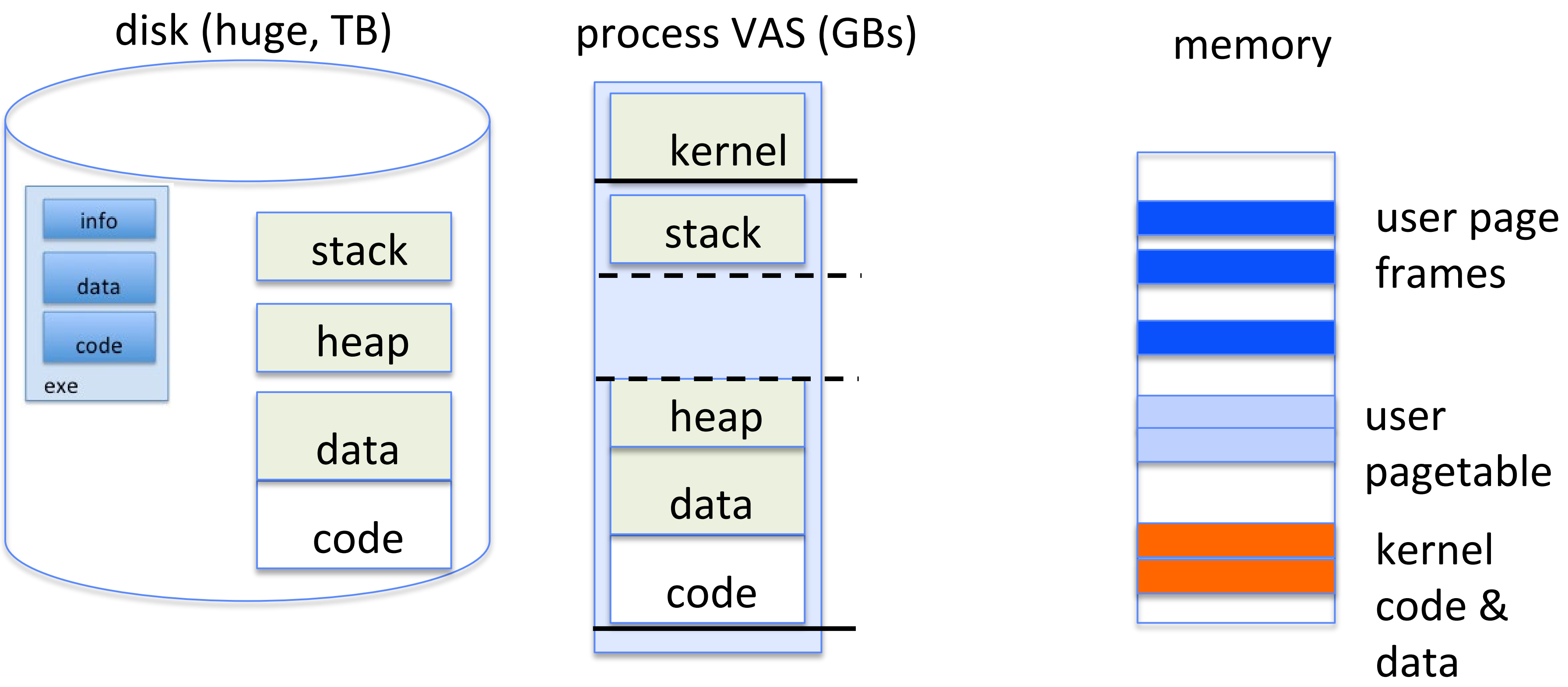


Utilized pages in the virtual address space (VAS) are backed by a page block on disk

Called the backing store or swap file

Typically, in an optimized block store, but can think of it like a file

Create Virtual Address Space of the Process



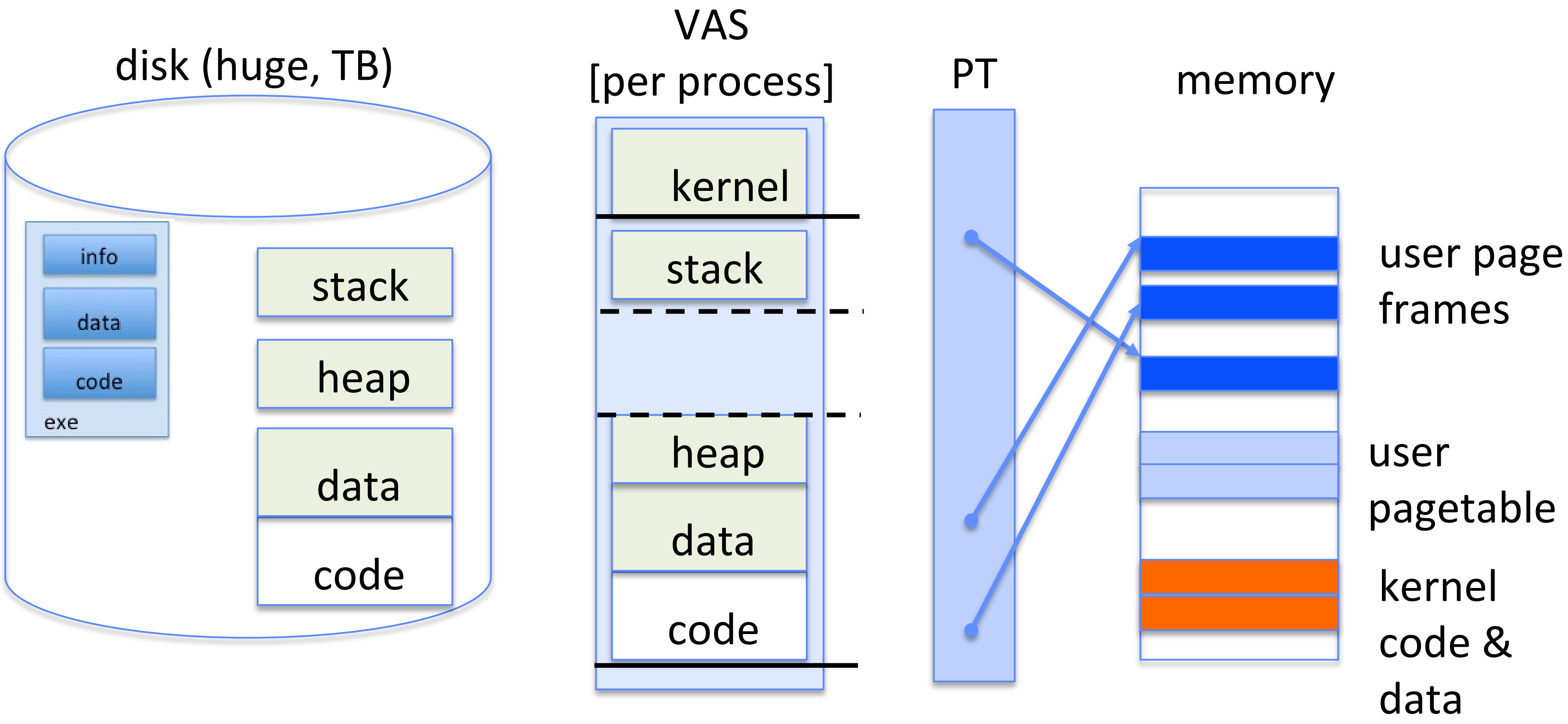
User Page table maps entire VAS

All the utilized regions are backed on disk

swapped into and out of memory as needed

For *every* process

Create Virtual Address Space of the Process

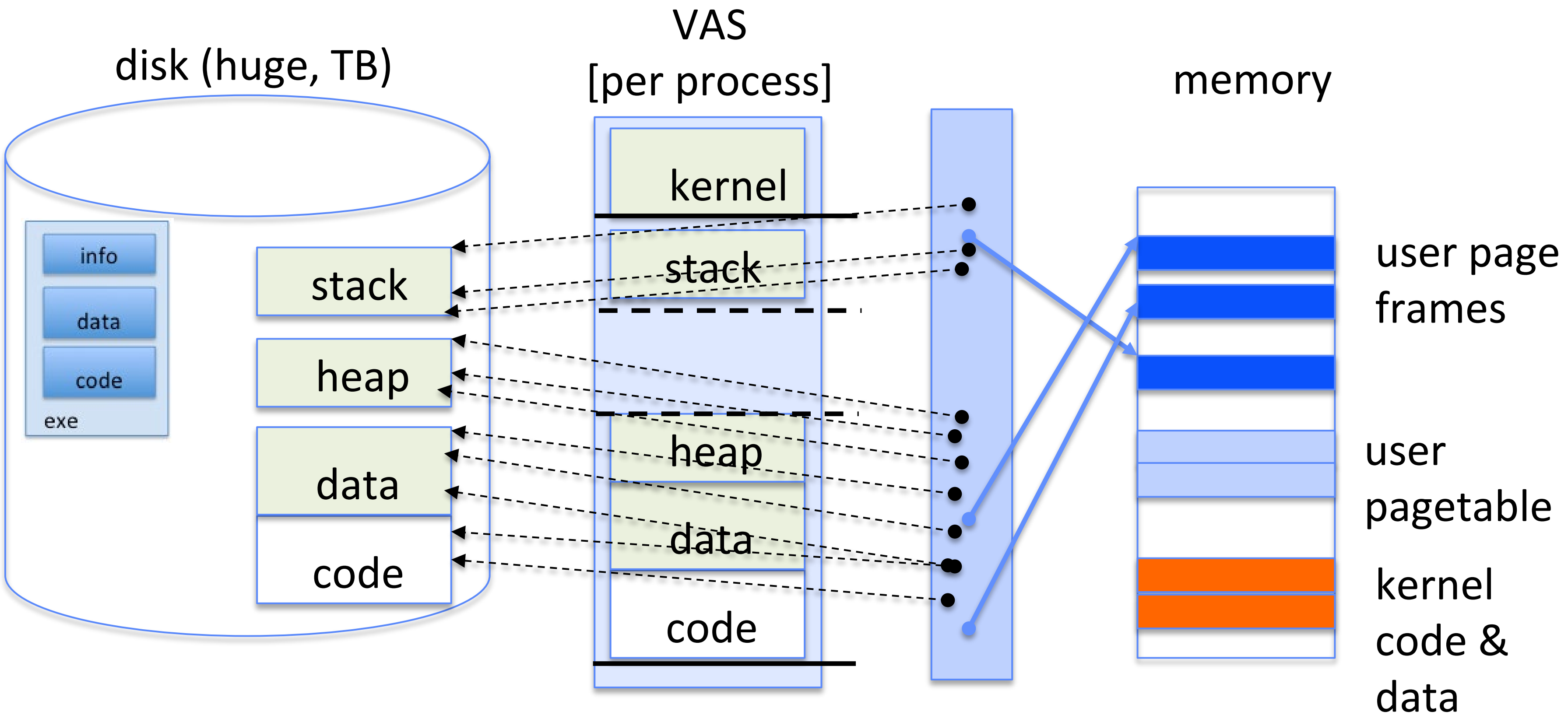


User Page table maps entire VAS

Resident pages to the frame in memory they occupy

The portion of it that the HW needs to access must be resident in memory

Provide Backing Store for VAS



User Page table maps entire VAS

Resident pages mapped to memory frames

For all other pages, OS must record where to find them on disk

What Data Structure Maps / Non-Resident Pages to Disk?

`FindBlock(PID, page#) → disk_block`

– Some OSs utilize spare space in PTE for paged blocks

– Like the PT, but purely software

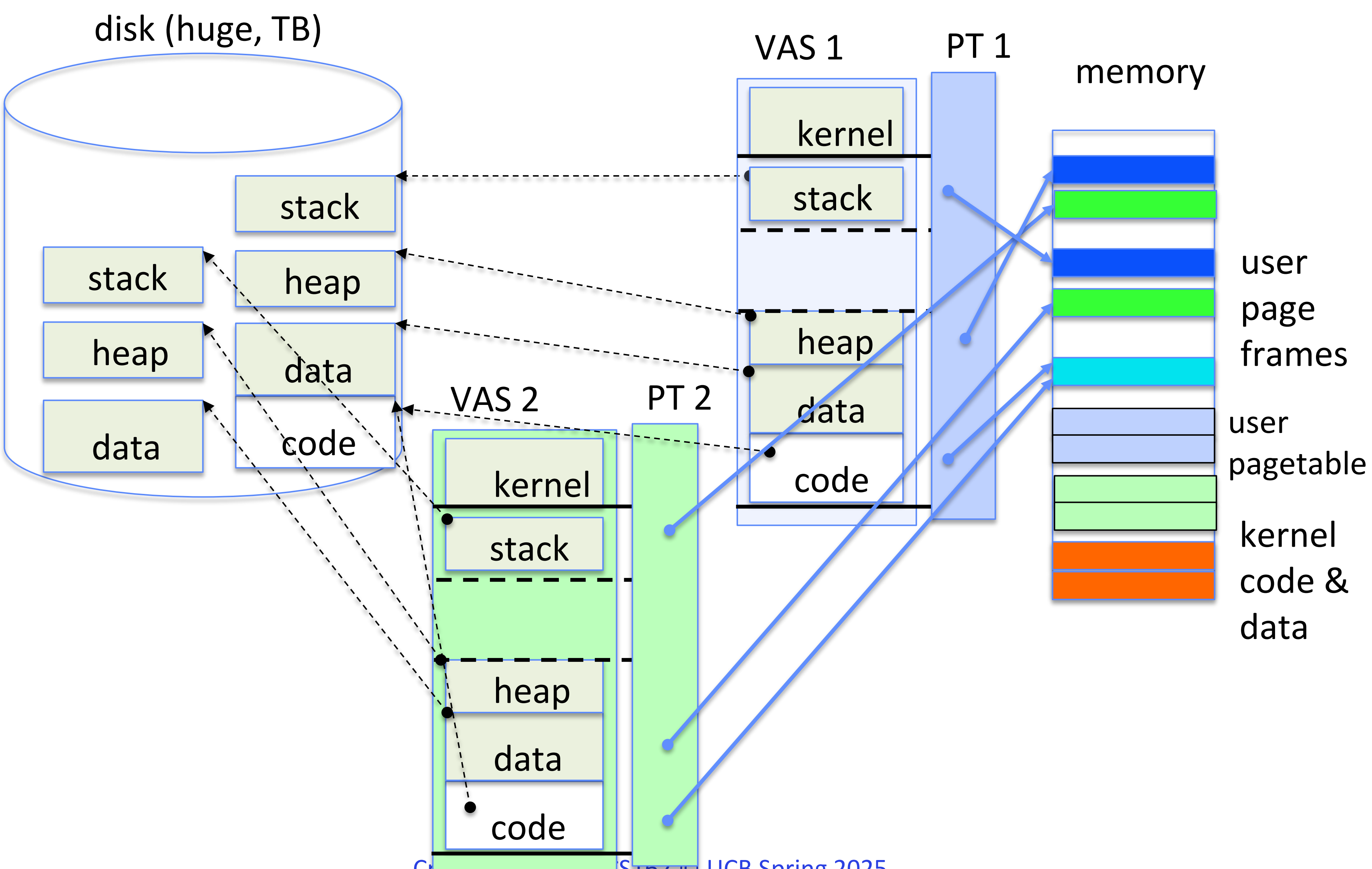
Where to store it?

– In memory – can be compact representation if swap storage is contiguous on disk

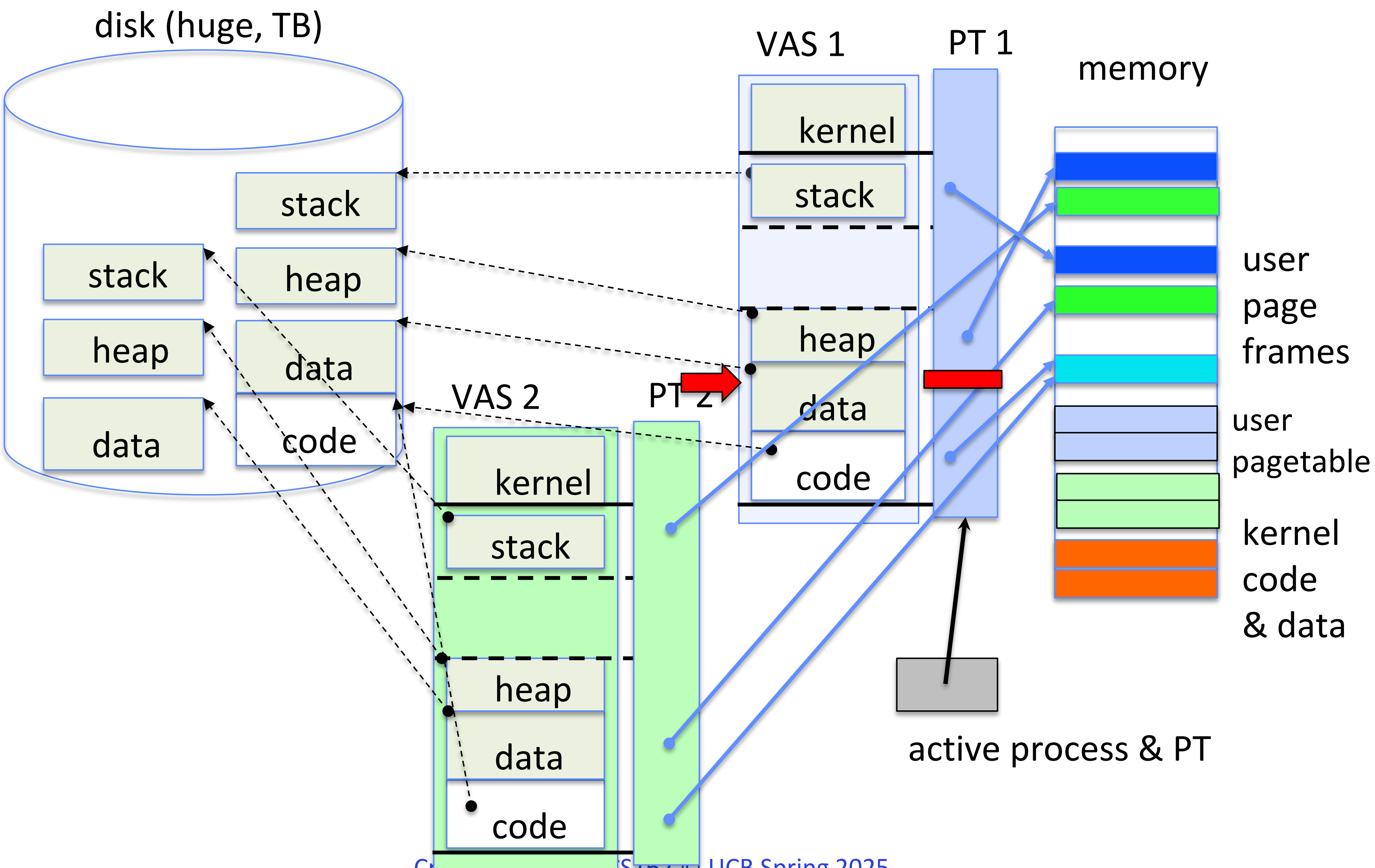
– Could use hash table (like Inverted PT)

Usually want backing store for resident pages too

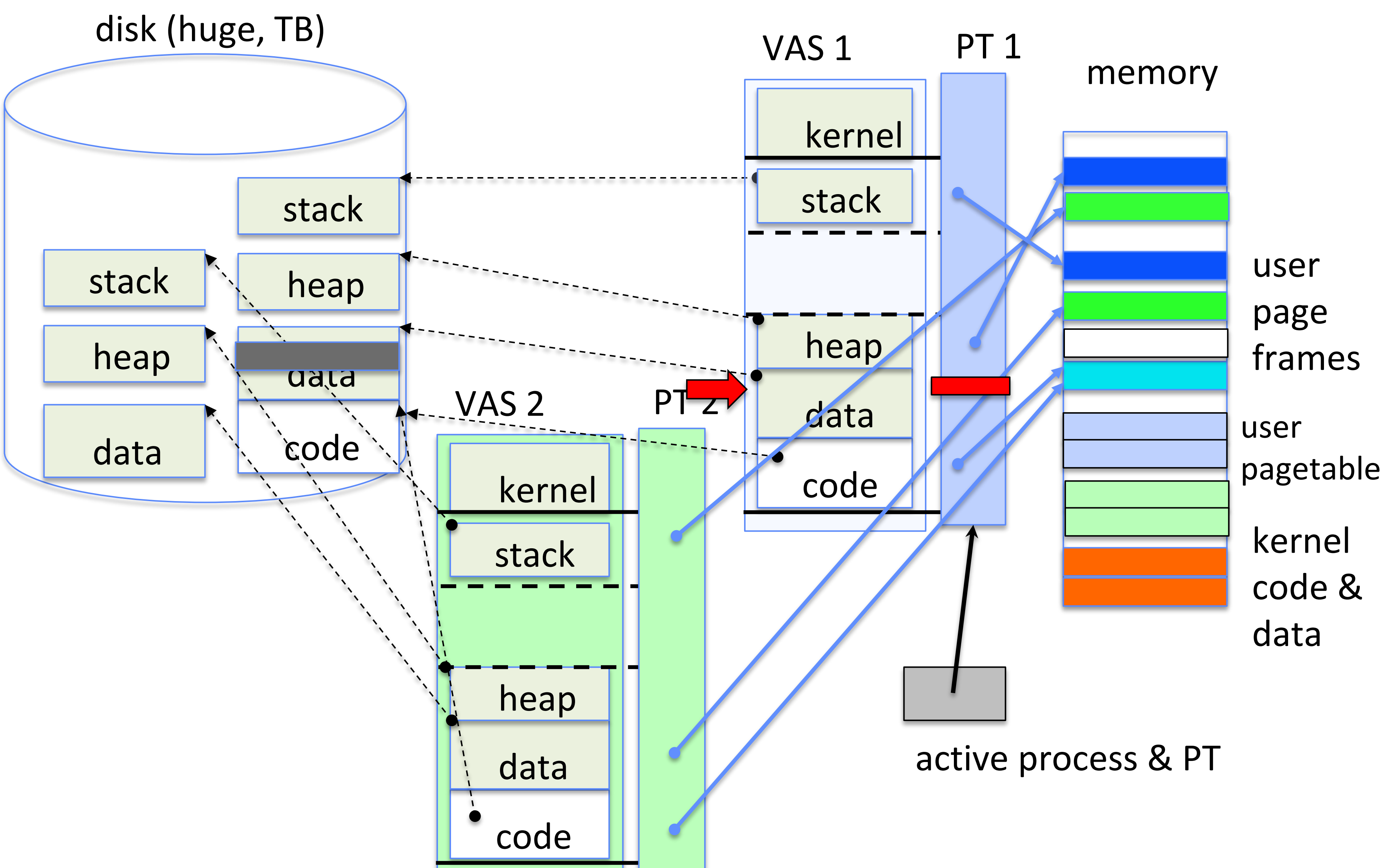
Provide Backing Store for VAS



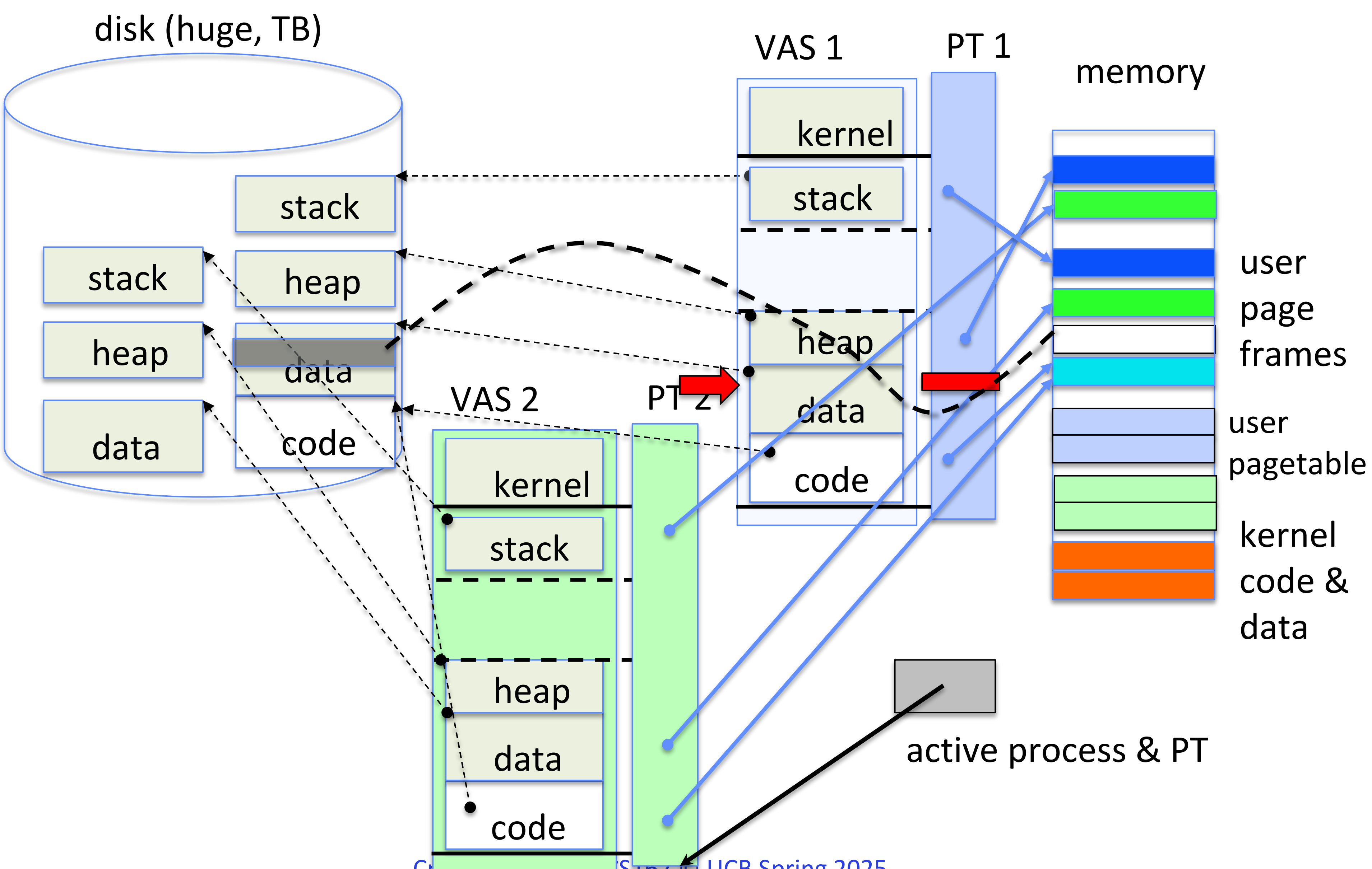
On page Fault …



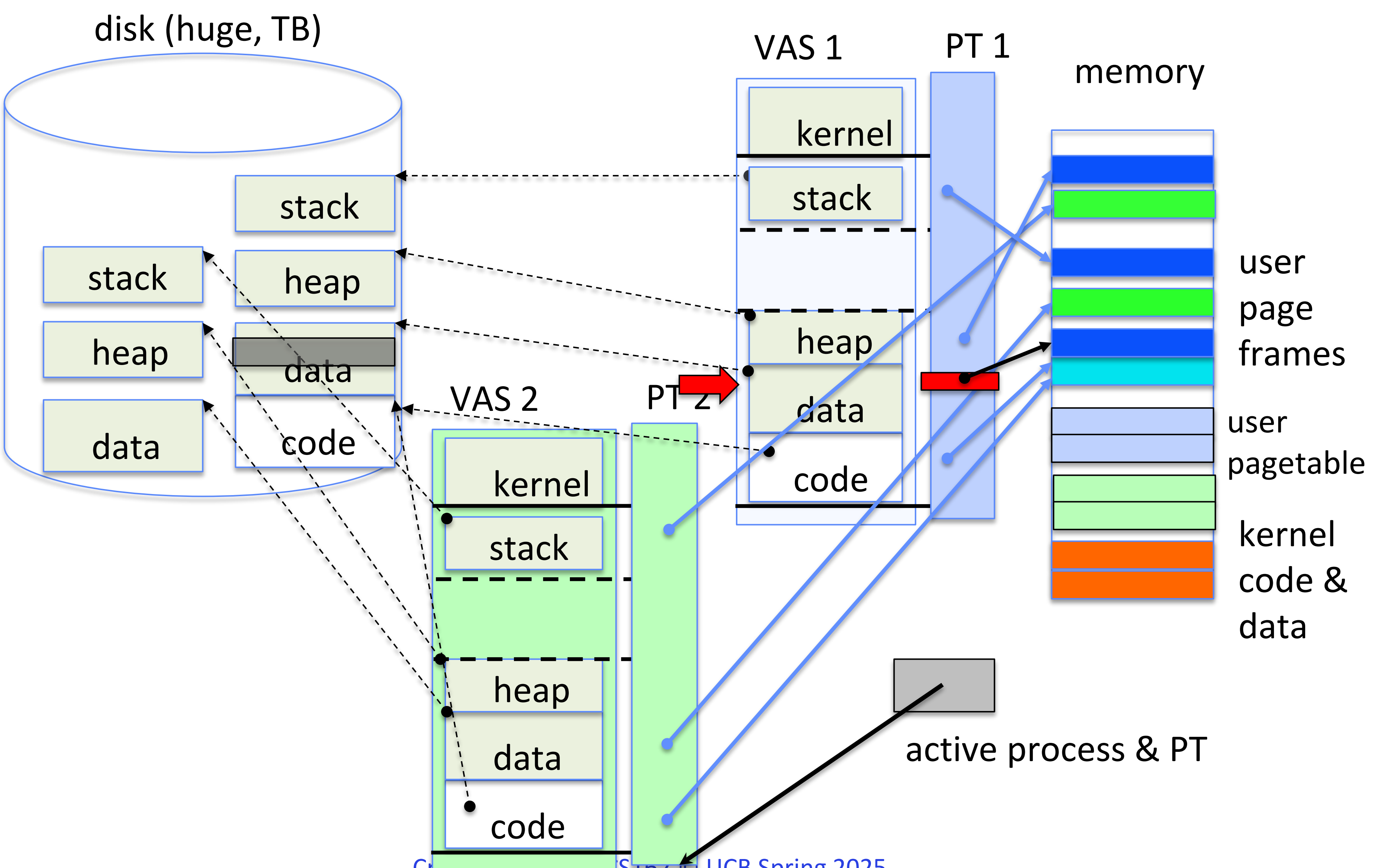
On page Fault … find & start load



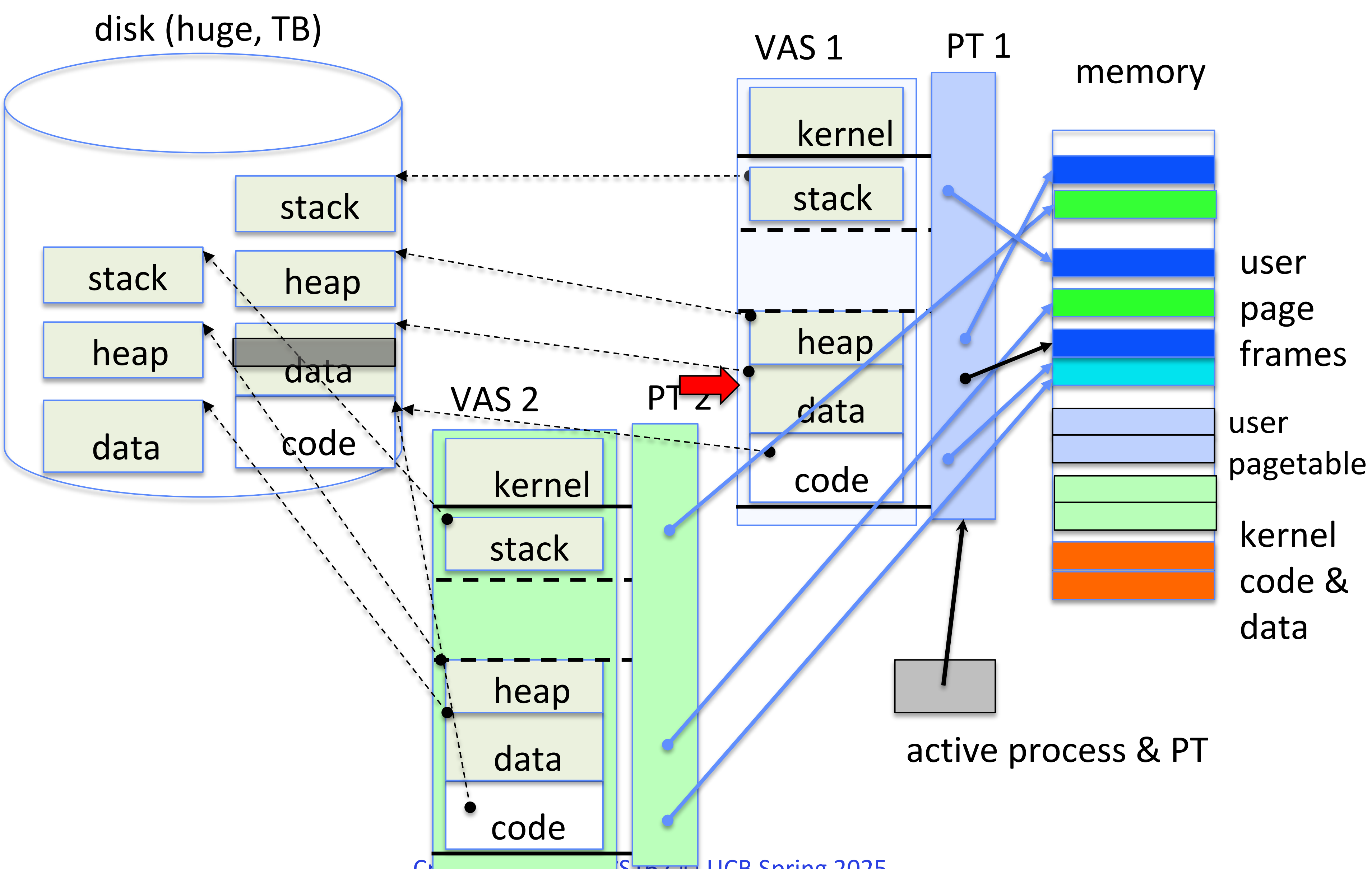
On page Fault … schedule other P or T



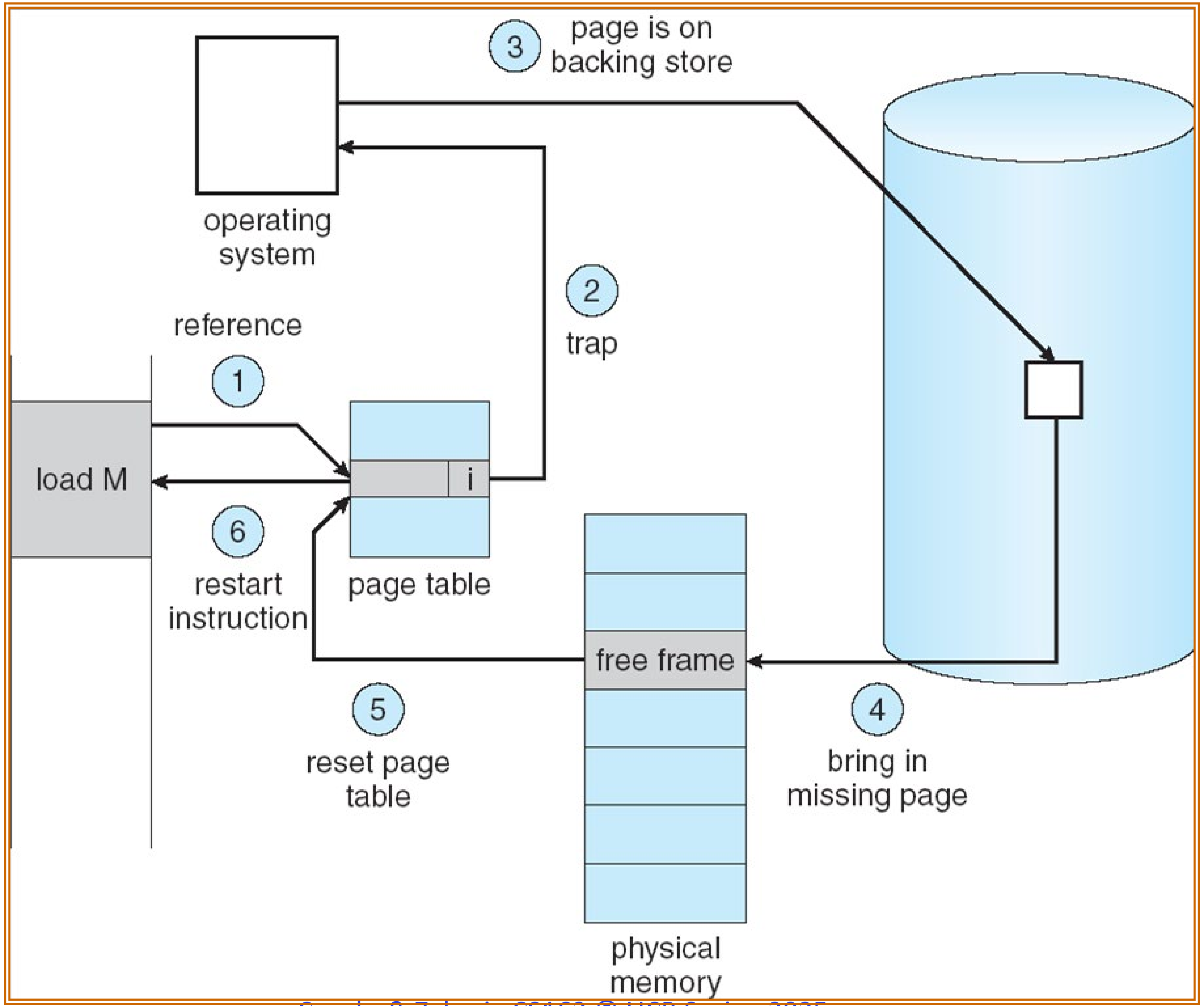
On page Fault … update PTE



Eventually reschedule faulting thread



Summary: Steps in Handling a Page Fault



Some questions we need to answer!

During a page fault, where does the OS get a free frame?

- Keeps a free list
- Unix runs a “reaper” if memory gets too full
  - » Schedule dirty pages to be written back on disk
  - » Zero (clean) pages which haven’t been accessed in a while
- As a last resort, evict a dirty page first

How can we organize these mechanisms?

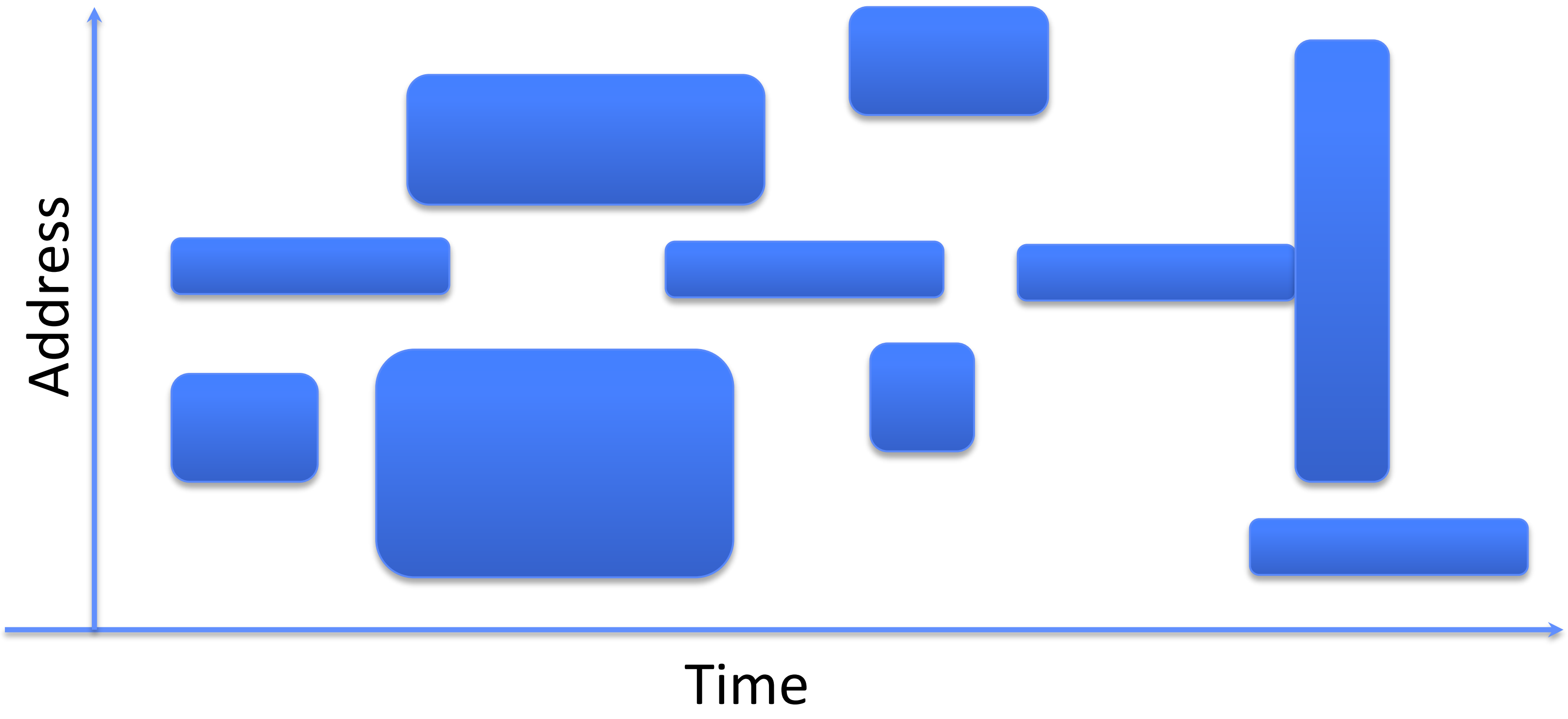
- Work on the replacement policy

How many page frames/process?

- Like thread scheduling, need to “schedule” memory resources:
  - » Utilization? fairness? priority?
- Allocation of disk paging bandwidth

Working Set Model

As a program executes it transitions through a sequence of "working sets" consisting of varying sized subsets of the address space



Demand Paging Cost Model

Since Demand Paging like caching, can compute average access time! ("Effective Access Time")

- EAT = Hit Rate x Hit Time + Miss Rate x Miss Time
- EAT = Hit Time + Miss Rate x Miss Penalty

Example:

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- Suppose p = Probability of miss, 1-p = Probably of hit
- Then, we can compute EAT as follows:

$$EAT = 200ns + p \times 8\ ms$$
$$= 200ns + p \times 8{,}000{,}000ns$$

If one access out of 1,000 causes a page fault, then EAT = 8.2 μs:

- This is a slowdown by a factor of 40!

What if want slowdown by less than 10%?

- $EAT < 200ns \times 1.1 \Rightarrow p < 2.5 \times 10^{-6}$
- This is about 1 page fault in 400,000!

What Factors Lead to Misses in Page Cache?

Compulsory Misses:

Pages that have never been paged into memory before

Capacity Misses:

Not enough memory. Must somehow increase available memory size.

Policy Misses:

Caused when pages were in memory, but kicked out prematurely because of the replacement policy

Page Replacement Policies

Why do we care about Replacement Policy?

Replacement is an issue with any cache

Particularly important with pages

The cost of being wrong is high: must go to disk

Page Replacement Policies

FIFO (First In, First Out)

– Throw out oldest page. Be fair – let every page live in memory for same amount of time.

  – Bad – throws out heavily used pages instead of infrequently used

RANDOM:

  – Pick random page for every replacement

  – Typical solution for TLB's. Simple hardware

  – Pretty unpredictable – makes it hard to make real-time guarantees

MIN (Minimum):

  – Replace page that won't be used for the longest time

  – Great (provably optimal), but can't really know future…

  – But past is a good predictor of the future …