Memory 3: Virtual Memory

Sam Kumar

CS 162: Operating Systems and System Programming

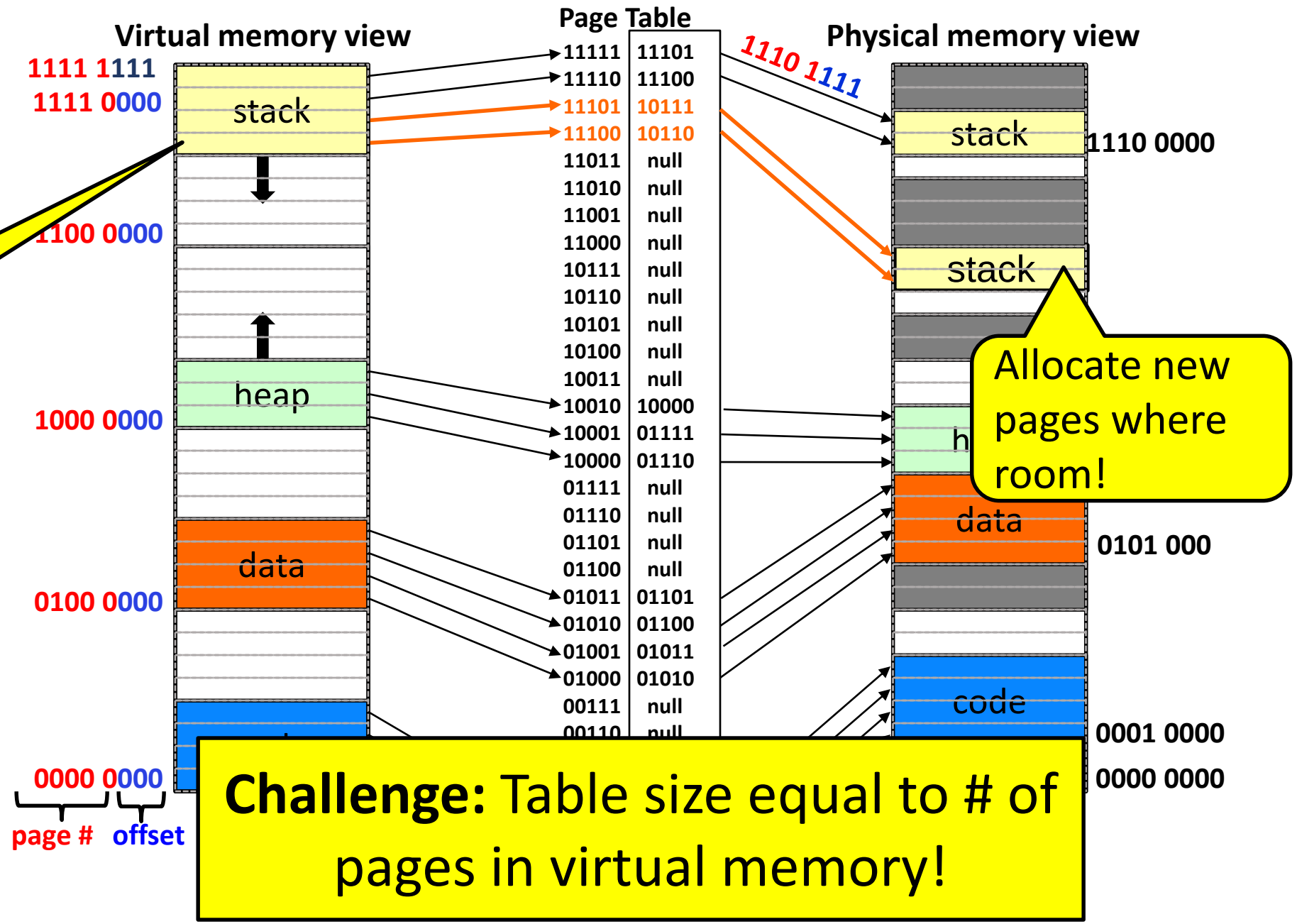
Lecture 17

https://inst.eecs.berkeley.edu/~cs162/su20

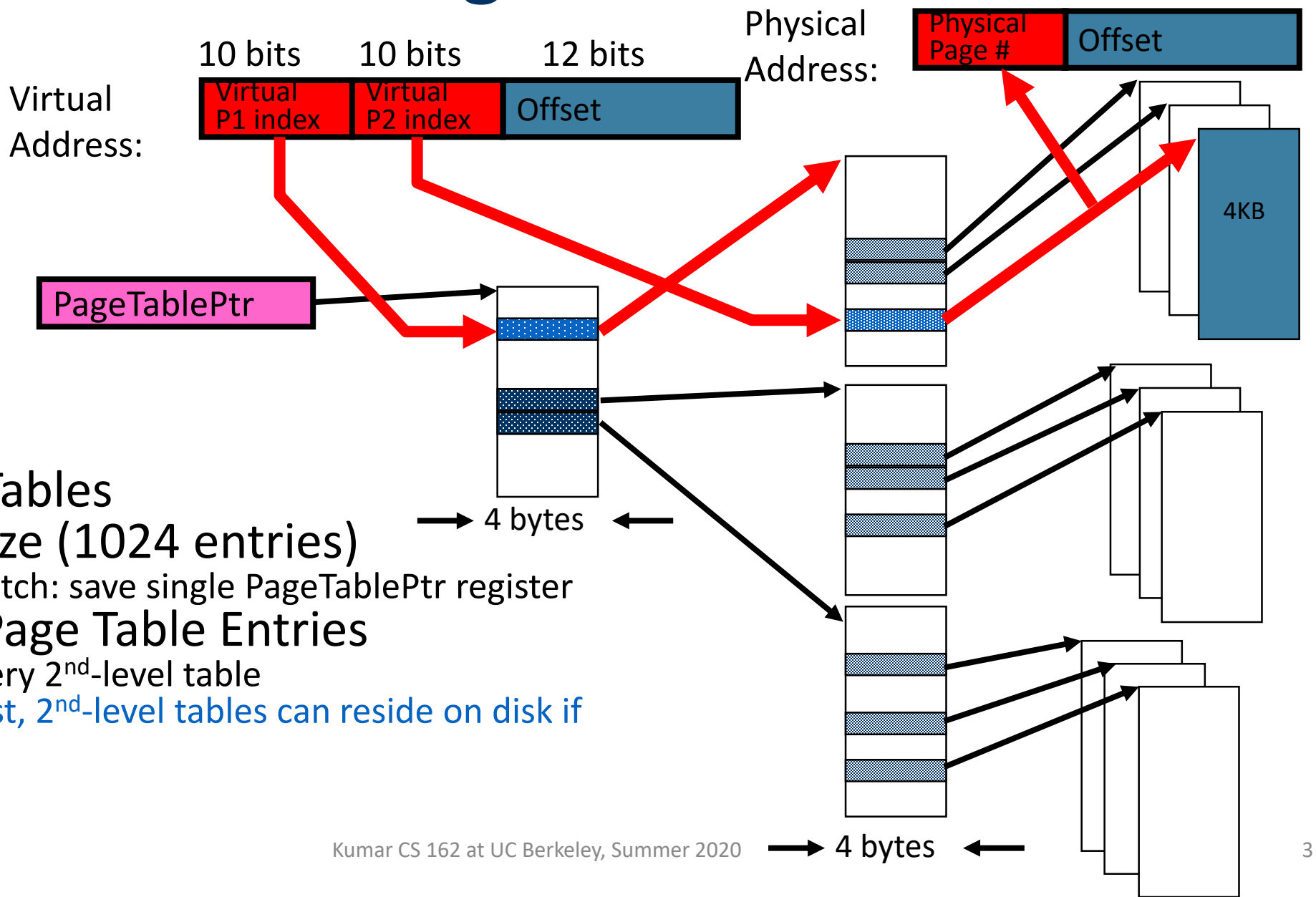
Read: A&D Ch 9.5-8

Recall: Paging

What happens if stack grows to 1110 0000?

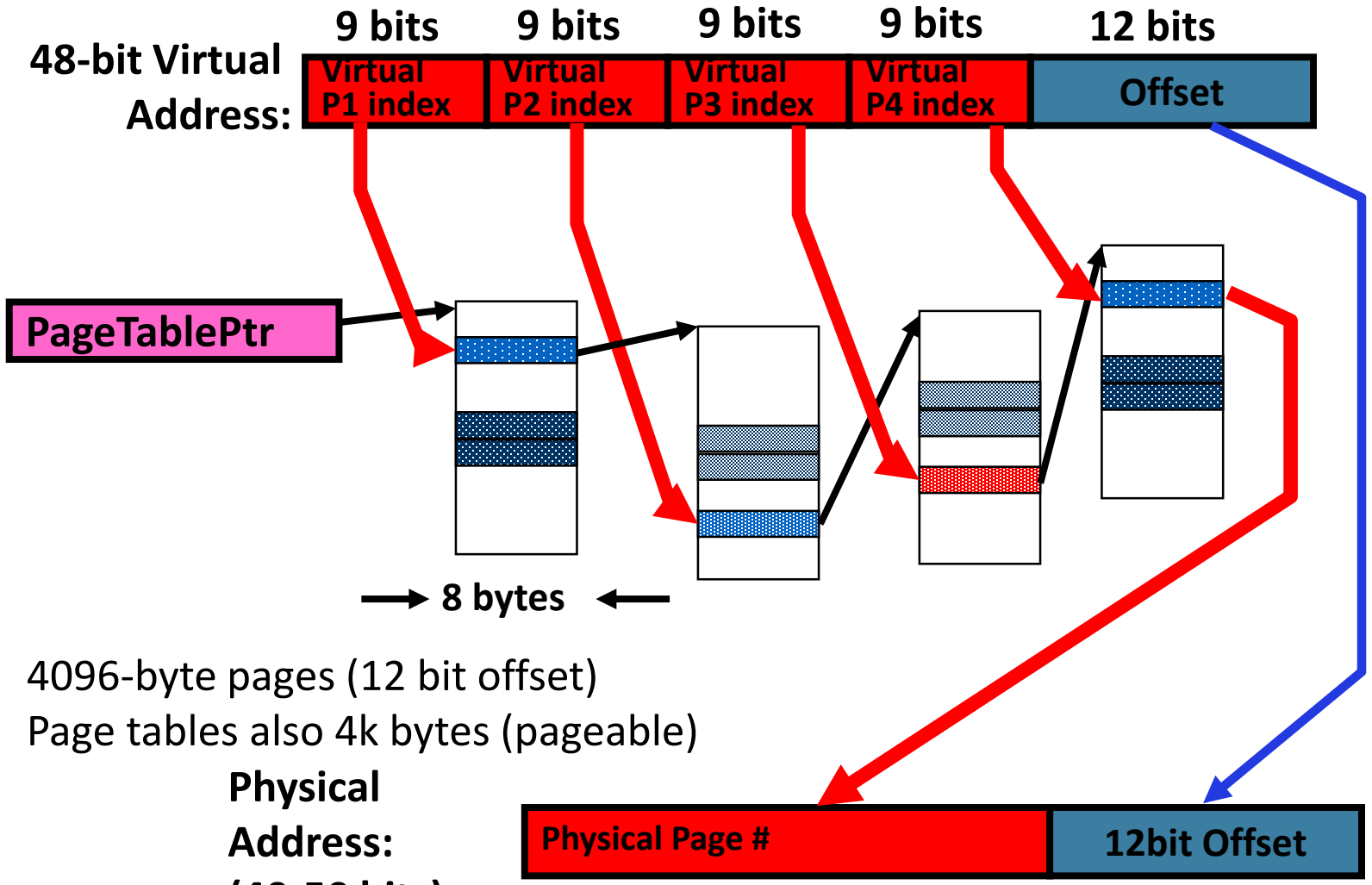


Recall: Two-Level Page Table



- Tree of Page Tables
- Tables fixed size (1024 entries)
  - On context-switch: save single PageTablePtr register
- Valid bits on Page Table Entries
  - Don't need every 2nd-level table
  - Even when exist, 2nd-level tables can reside on disk if not in use

Recall: x86-64, Four-Level Page Table



4096-byte pages (12 bit offset)

Page tables also 4k bytes (pageable)

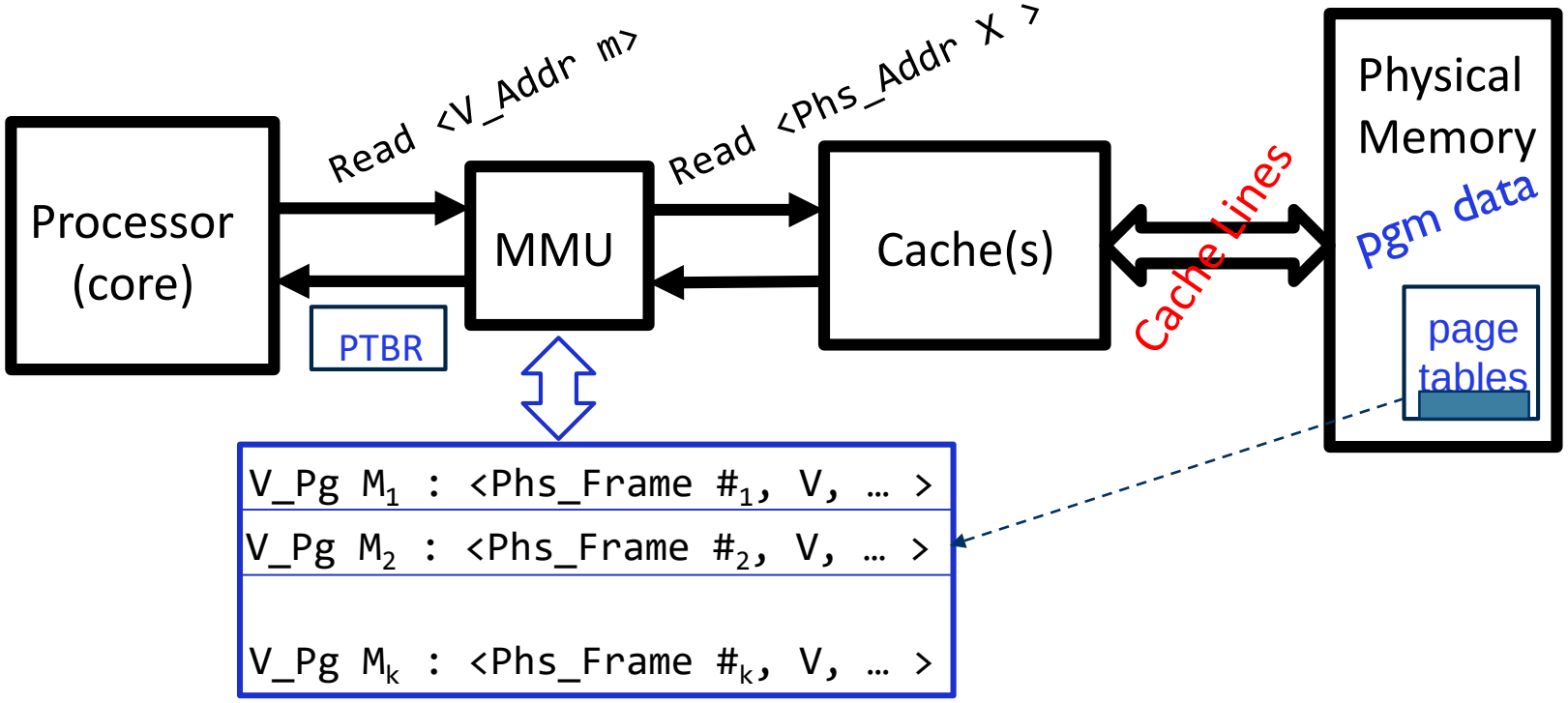
Physical Address: (40-50 bits)

Address Translation Comparison

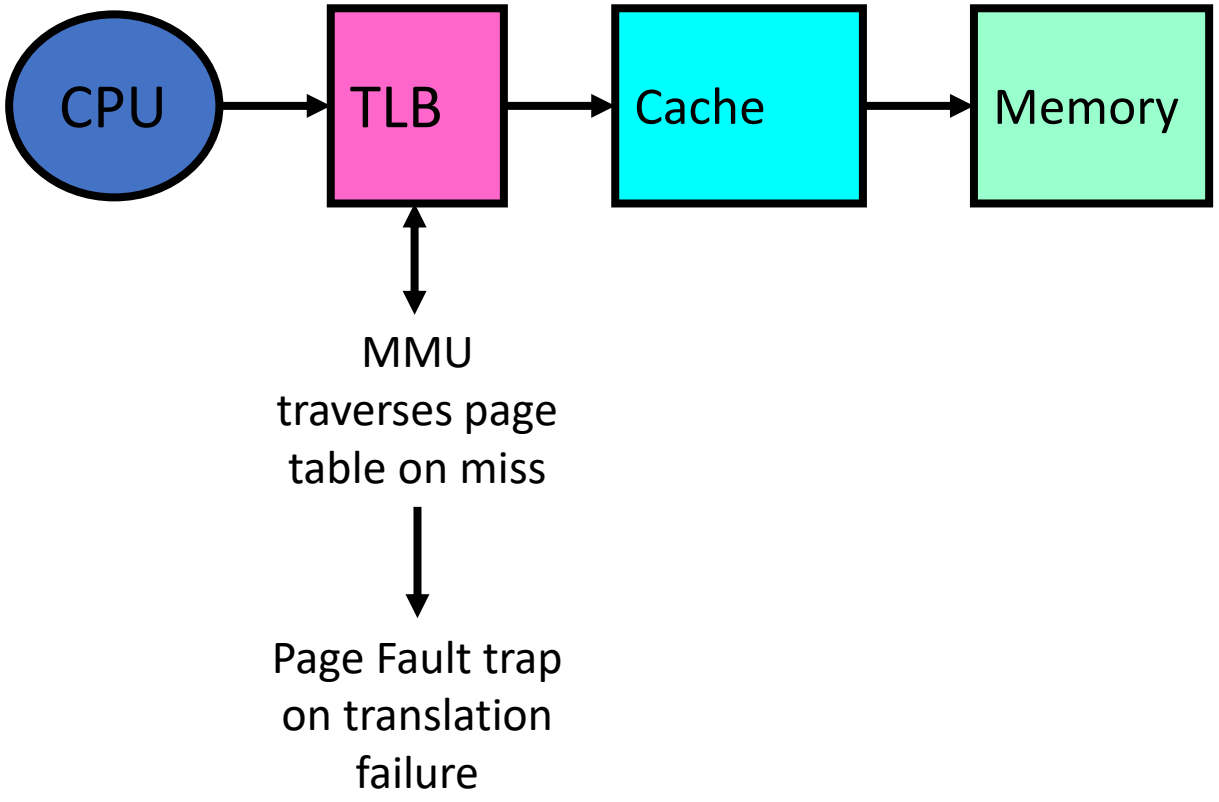
	Advantages	Disadvantages
Simple Segmentation	Fast context switching (segment map maintained by CPU)	External fragmentation
Paging (Single-Level)	No external fragmentation Fast and easy allocation	Large table size (~ virtual memory) Internal fragmentation
Paged Segmentation	Table size ~ # of pages in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

Recall: Making Address Translation Fast

- Cache results of recent translations
  - Separate from memory cache
  - Cache PTEs using Virtual Page Number as the key



Recall: The Big Picture



Recall: TLB Organization

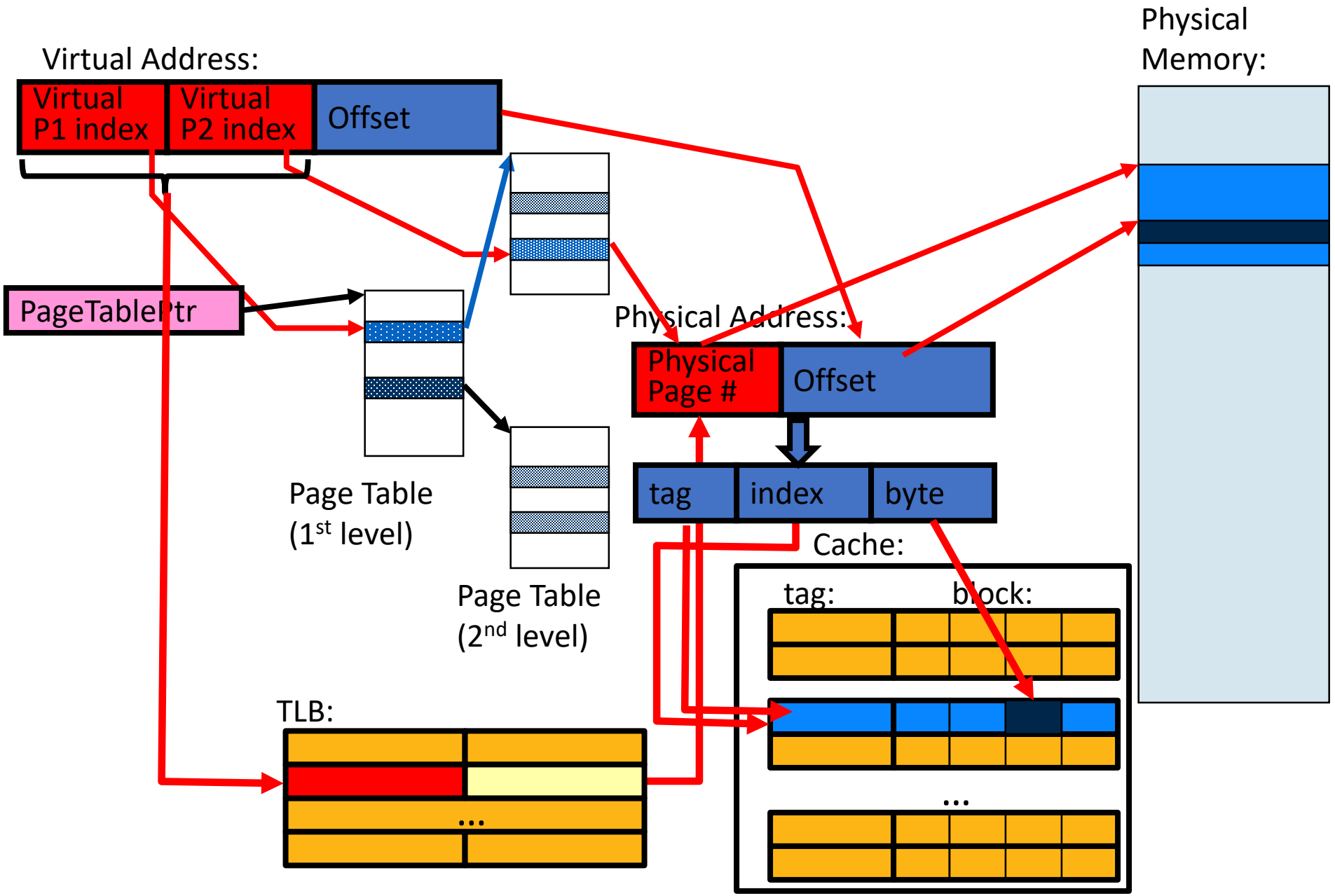
- How big does TLB actually have to be?
  - Usually fewer entries than the cache (why?)
  - Not very big, can support higher associativity
- Small TLBs usually organized as fully-associative cache
  - Lookup is by Virtual Address
  - Returns Physical Address + other info
- What happens when fully-associative is too slow?
  - Put a small (4-16 entry) direct-mapped cache in front
  - Called a "TLB Slice"
- Example for MIPS R3000:

Virtual Address	Physical Address	Dirty	Ref	Valid	Access	ASID
0xFA00	0x0003	Y	N	Y	R/W	34
0x0040	0x0010	N	Y	Y	R	0
0x0041	0x0011	N	Y	Y	R	0

Recall: Current Example

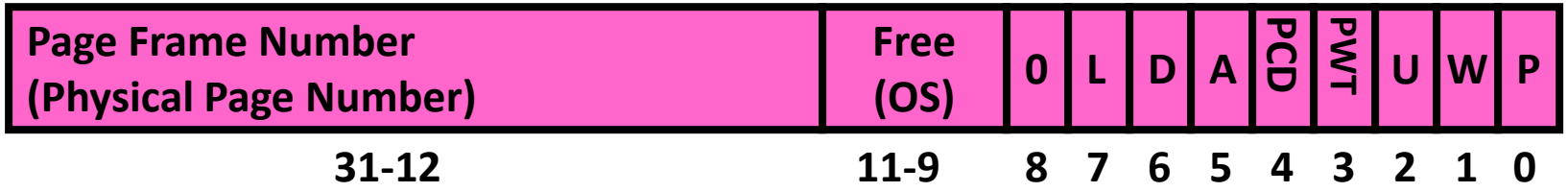
- Caches (all 64 B line size)
  - L1 I-Cache: 32 KiB/core, 8-way set assoc.
  - L1 D Cache: 32 KiB/core, 8-way set assoc., 4-5 cycles load-to-use, Write-back policy
  - L2 Cache: 1 MiB/core, 16-way set assoc., Inclusive, Write-back policy, 14 cycles latency
  - L3 Cache: 1.375 MiB/core, 11-way set assoc., shared across cores, Non-inclusive victim cache, Write-back policy, 50-70 cycles latency
- TLB
  - L1 ITLB, 128 entries; 8-way set assoc. for 4 KB pages
    - 8 entries per thread; fully associative, for 2 MiB / 4 MiB page
  - L1 DTLB 64 entries; 4-way set associative for 4 KB pages
    - 32 entries; 4-way set associative, 2 MiB / 4 MiB page translations
    - 4 entries; 4-way associative, 1G page translations
  - L2 STLB: 1536 entries; 12-way set assoc. 4 KiB + 2 MiB pages
    - 16 entries; 4-way set associative, 1 GiB page translations

Putting Everything Together



What's in a Page Table Entry (PTE)?

- What is in a Page Table Entry (or PTE)?
  - "Pointer to" (address of) next-level page table or to actual page
  - Permission bits: valid, read-only, read-write, write-only
- Example: Intel x86 architecture PTE:



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 (PTE only): page has been modified recently
L: L=1⇒4MB page (directory only).
Bottom 22 bits of virtual address serve as offset

What to do if the Translation Fails?

- **Page Fault**
  - PTE marked invalid, Priv. Level Violation, Access violation, or does not exist
  - Causes a Fault / Trap, ***allowing the OS to run***
  - May occur on instruction fetch or data access

Recall: Interposing on Process Behavior

- OS interposes on process' I/O operations
  - How? All I/O happens via syscalls.

- OS interposes on process' CPU usage
  - How? Interrupt lets OS preempt current thread

- **Question: How can the OS interpose on process' memory accesses?**
  - Too slow for the OS to interpose *every* memory access
  - Translation: hardware support to accelerate the common case
  - Page fault: uncommon cases trap to the OS to handle

What might the OS do on a page fault?

- If the access is right below the stack…
  - OS might allocate a new stack page and retry the instruction
- If the access is a write to a page after fork()…
  - OS might copy the page, mark as writable, and retry the instruction
- If the access is one that the process has no good reason to make…
  - OS typically terminates the process (segmentation fault)
  - (e.g., for page marked kernel only)
- If access is to a page whose contents are in secondary storage…
  - OS brings in page from secondary storage to memory (demand paging)

How to Use a PTE

- Usage Example: Demand Paging
  - Keep only active pages in memory
  - Place others on disk and mark their PTEs invalid
- Usage Example: Copy on Write
  - UNIX fork gives *copy* of parent address space to child
    - Address spaces disconnected after child created
  - How to do this cheaply?
    - Make copy of parent's page tables (point at same memory)
    - Mark entries in both sets of page tables as read-only
    - Page fault on write creates two copies
- Usage Example: Zero Fill On Demand
  - New data pages must carry no information (say be zeroed)
  - Mark PTEs as invalid; page fault on use gets zeroed page
  - Often, OS creates zeroed pages in background

How does the OS know what to do?

- A page fault could mean a variety of things…
- OS keeps track of a *memory map* for each process
- OS needs to store additional info about each page to know what to do
  - Can use extra bits in the PTE
  - Typically, OS keeps additional information about pages in a data structure called the *supplemental page table*, which it consults on page faults

Inversion of the Hardware/Software Boundary

- For an instruction to complete the OS software must intervene
- Receive the page fault, remedy the situation
  - Load the page, create the page, copy-on-write, …
  - Update the PTE entry so the translation will succeed
- Restart (or resume) the instruction
  - This is one of the huge simplifications in RISC instructions sets
  - Can be very complex when instruction modify state (x86)

How to just “Restart” after a Page Fault?

- Modern processors exploit Instruction-Level Parallelism
  - Pipelining, out-of-order execution, etc.

- At the time the hardware recognizes an instruction as a page fault:
  - Prior instructions in that thread may not have been issued
  - Future instructions in that thread may have been completed
  - Some instructions may be partially done

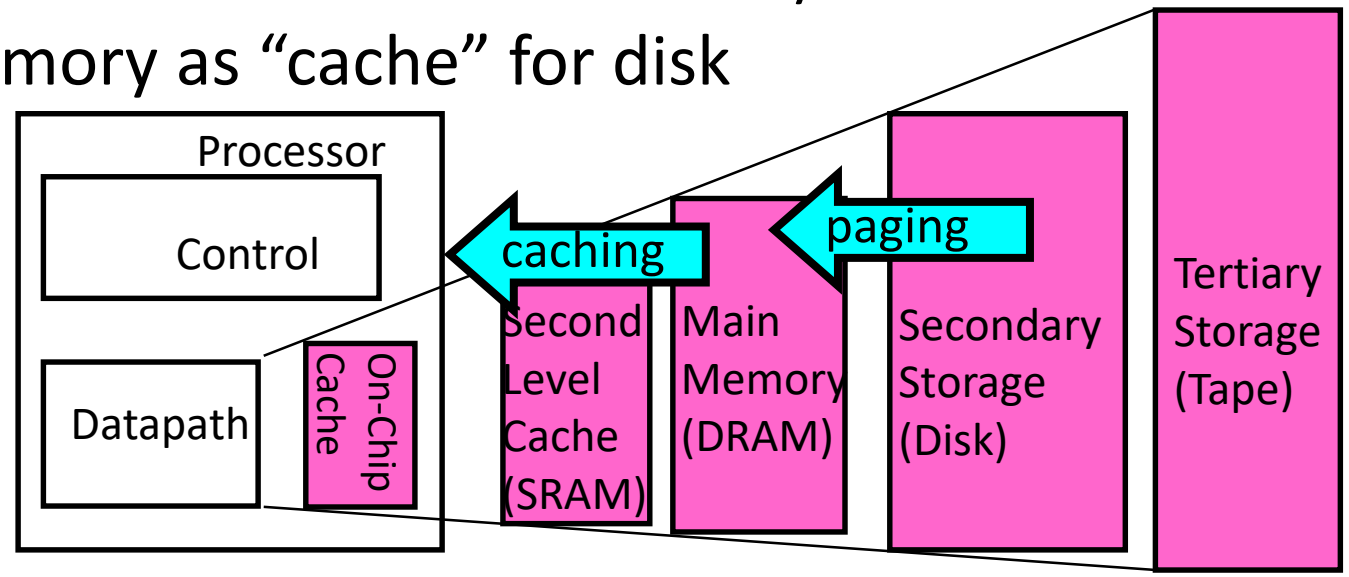
- How can the OS deal with this?

Precise Exceptions

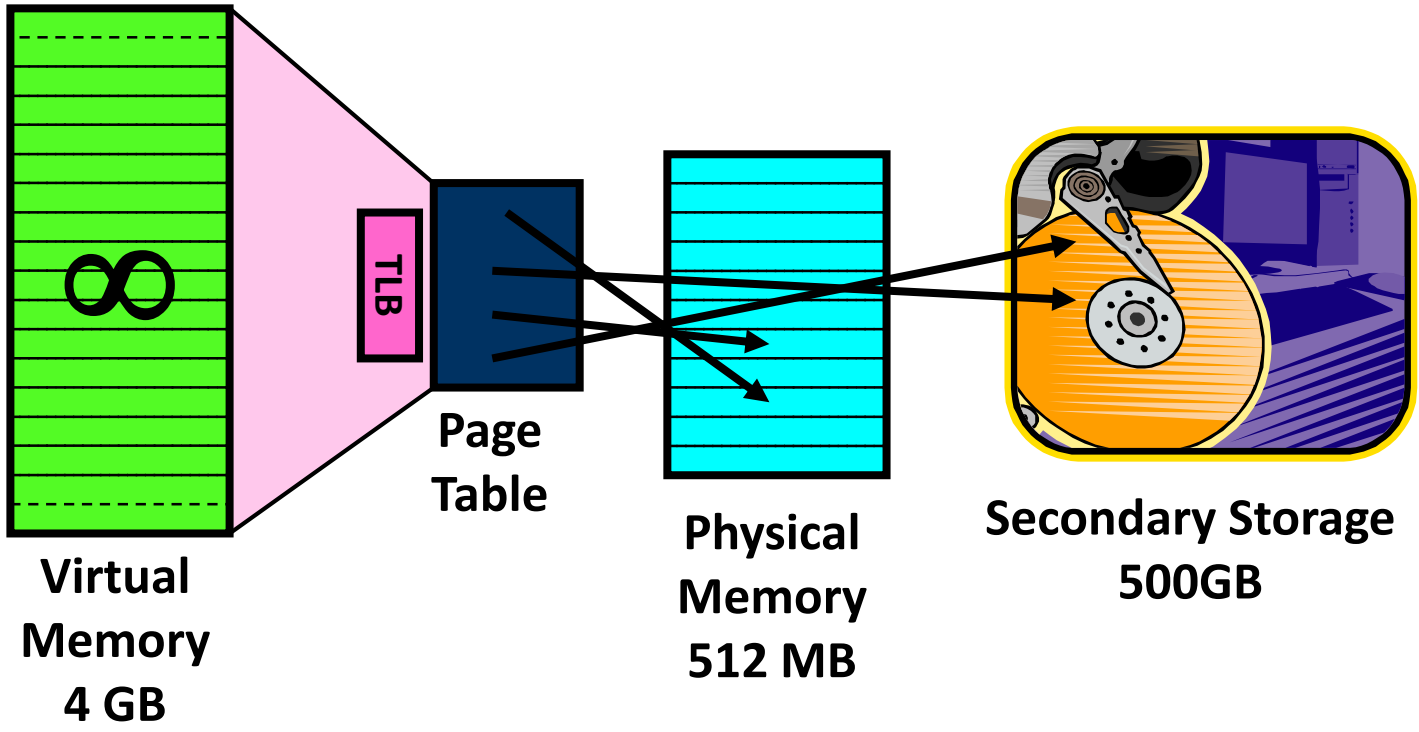
- Precise ⇒ state of the machine is preserved as if program executed up to the offending instruction
  - All previous instructions completed
  - Offending instruction and all following instructions act as if they have not even started
  - Same system code will work on different implementations
  - Difficult in the presence of pipelining, out-of-order execution, ...
- Imprecise ⇒ system software has to figure out what is where and put it all back together
- Modern techniques for out-of-order execution and branch prediction support precise interrupts
- Architectural support for OS is hard
  - Original M68000 had paging, but didn't save fault address properly
  - Original Sun Unix workstation used two, running one-cycle apart!

Demand Paging

- Modern programs require a lot of physical memory
  - Memory per system growing faster than 25%-30%/year
- But they don't use all their memory all of the time
  - 90-10 rule: programs spend 90% of their time in 10% of their code
  - Wasteful to require all of user's code to be in memory
- Solution: use main memory as "cache" for disk



Illusion of Infinite Memory



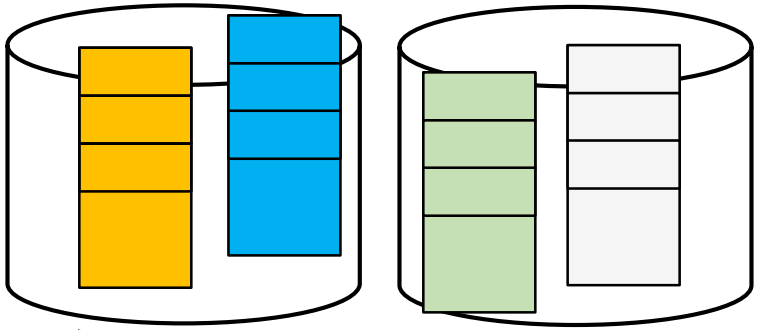
Secondary Storage is larger than physical memory ⇒

- In-use virtual memory can be bigger than physical memory
- More programs fit into memory, allowing more concurrency

- Principle: Transparent Level of Indirection (page table)
  - Supports flexible placement of physical data
    - Data could be on disk or somewhere across network
  - Variable location of data transparent to user program
    - Performance issue, not correctness issue

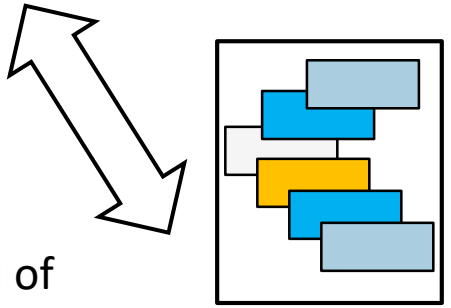
Origins of Paging

Keep most of the address space on disk



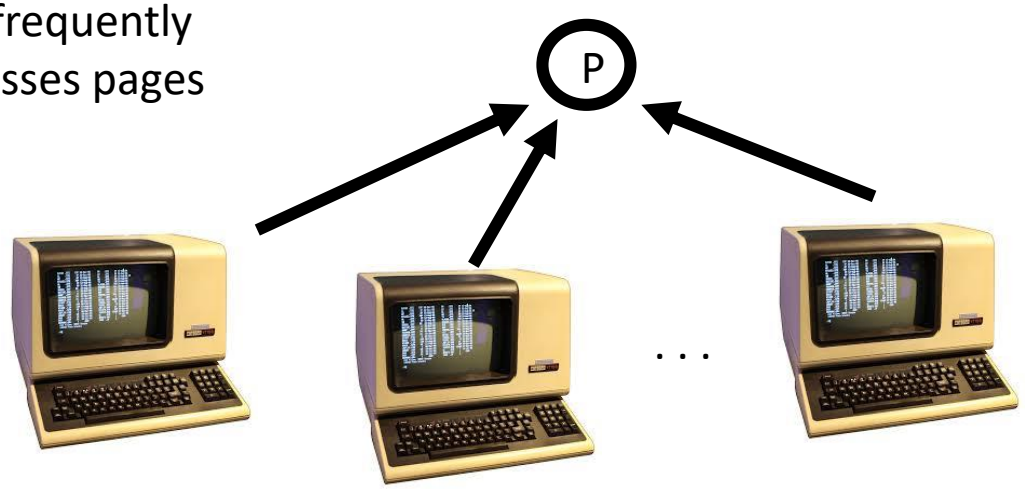
Disks provide most of the storage

Actively swap pages to/from



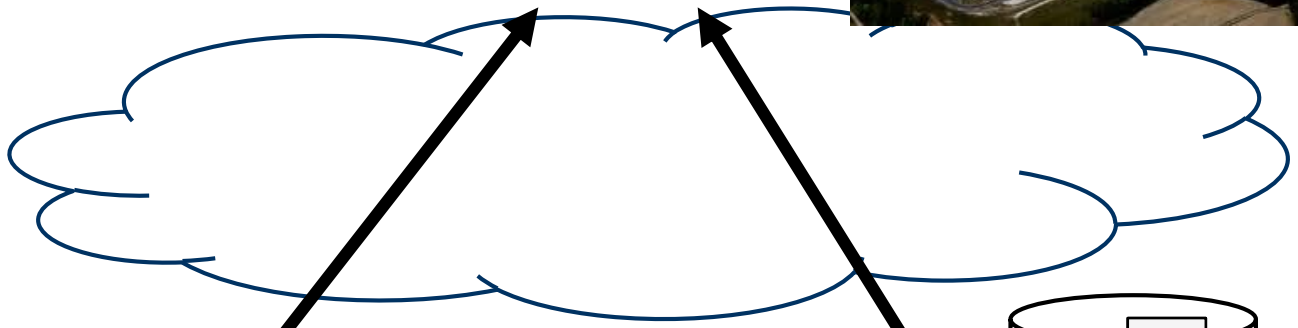
Relatively small memory, for many processes

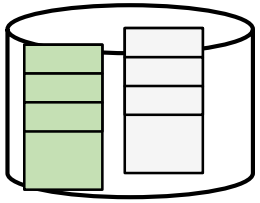
Keep memory full of the frequently accesses pages

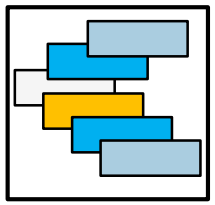


Many clients on dumb terminals running different programs

Very Different Situation Today

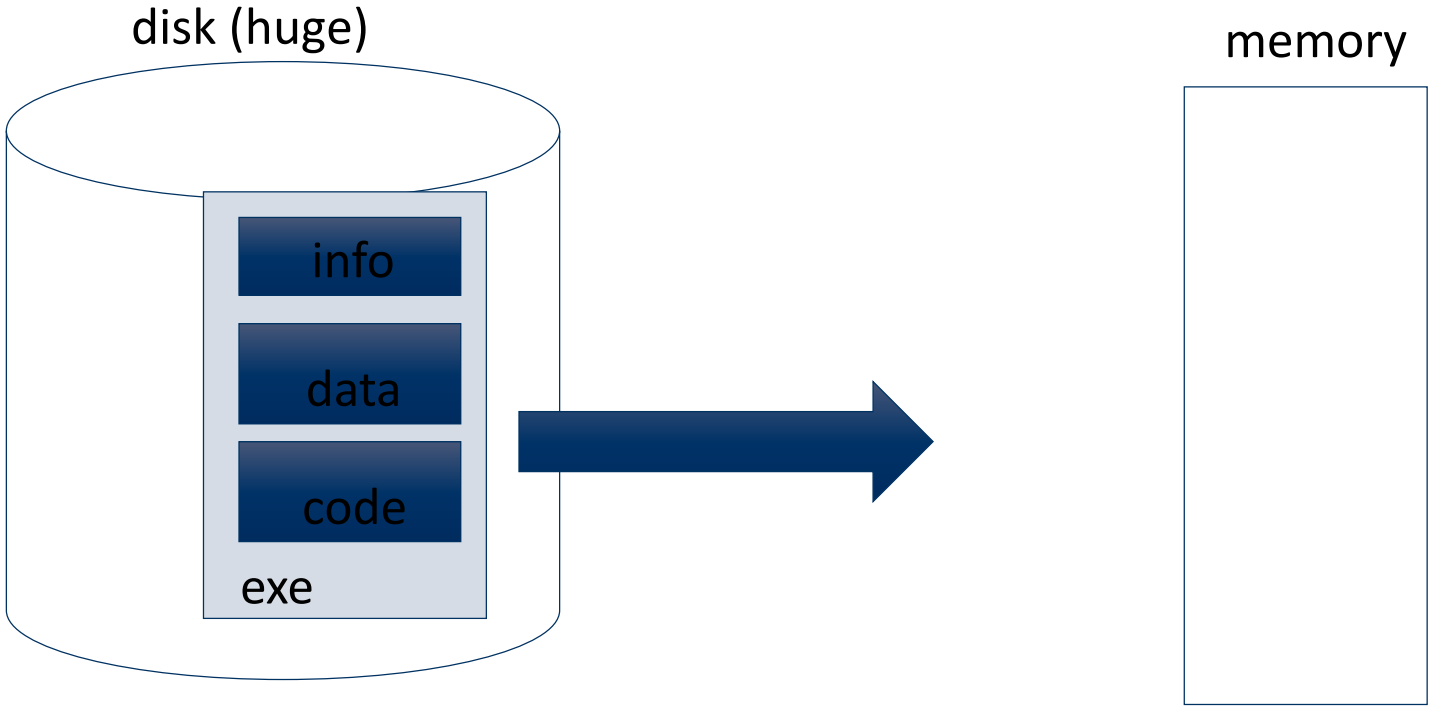






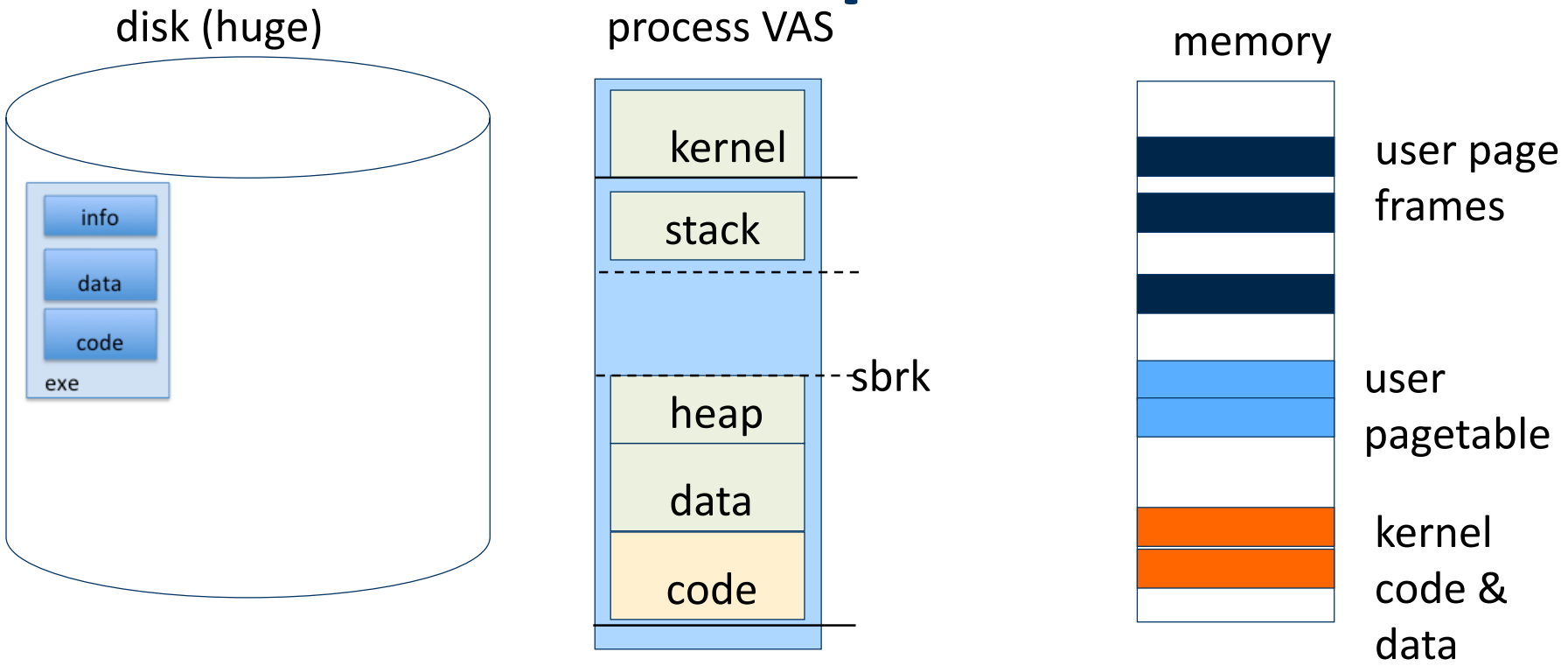
Powerful system
Huge memory
Huge disk
Single user

Classic: Loading an Executable into Memory



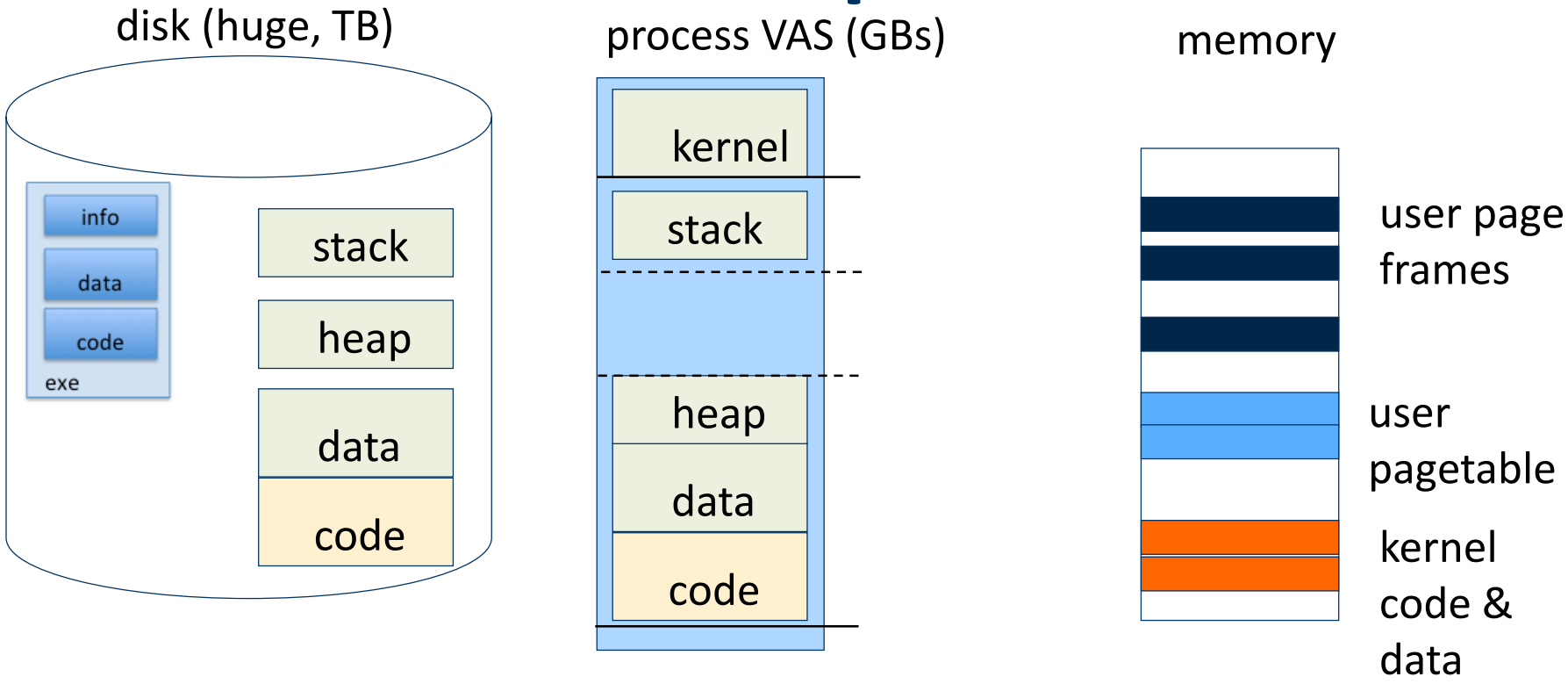
- .exe
  - lives on disk in the file system
  - contains contents of code & data segments, relocation entries and symbols
  - OS loads it into memory, initializes registers (and initial stack pointer)

Create Virtual Address Space of the Process



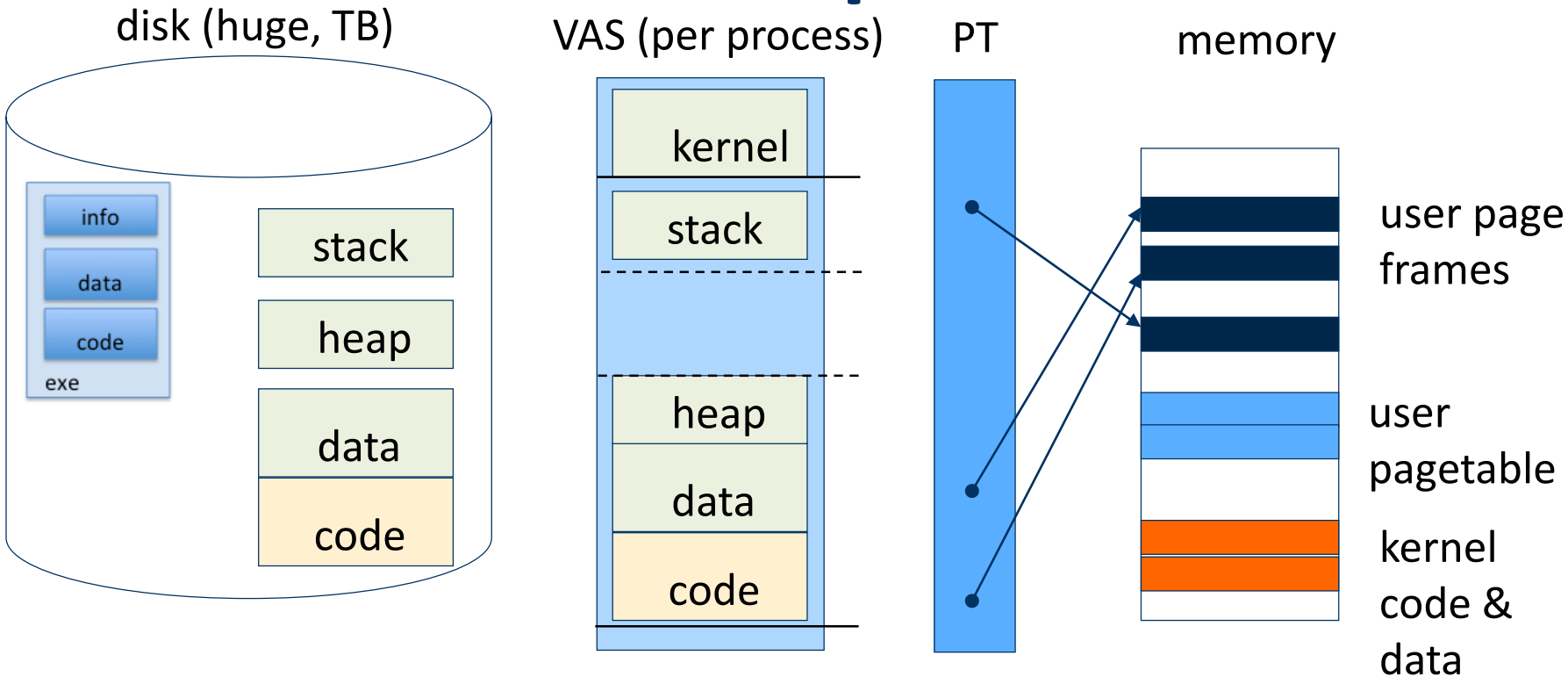
- Utilized pages in the VAS are backed by a page block on disk
  - Called the backing store or swap file
  - Typically in an optimized block store, but can think of it like a file

Create Virtual Address Space of the Process



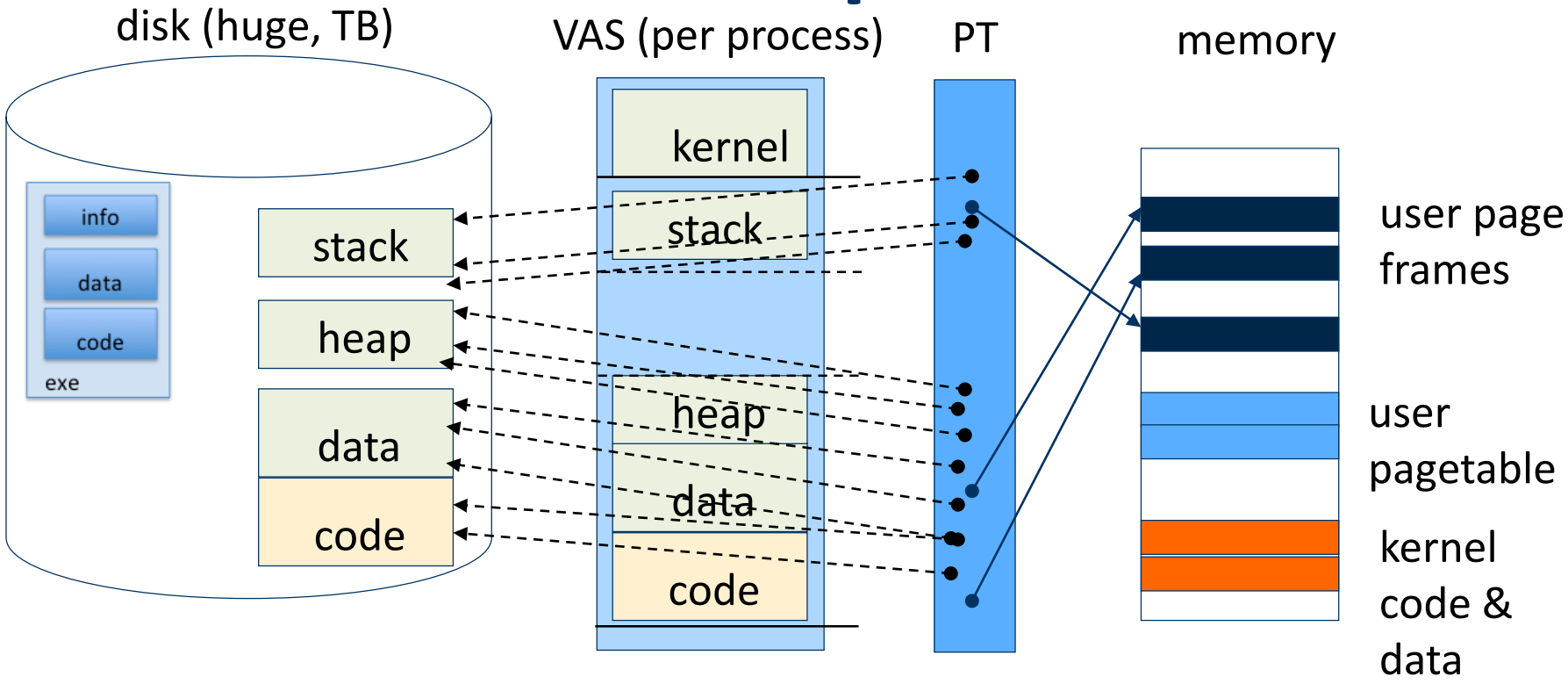
- User Page table maps entire VAS
- All the utilized regions are backed on disk
  - swapped into and out of memory as needed
- For *every* process

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

Create Virtual Address Space of the Process



- 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?

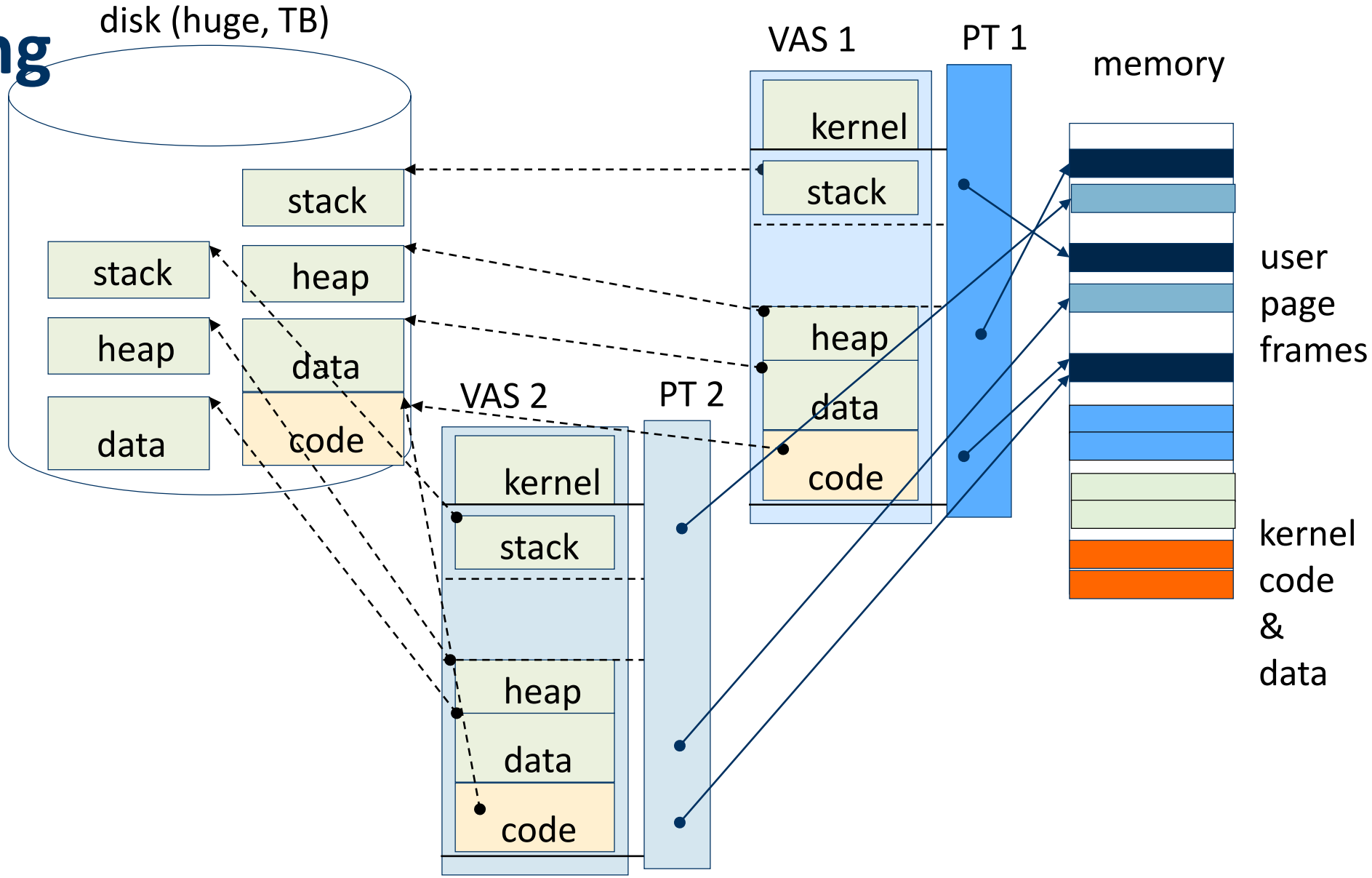
- `FindBlock(PID, page#)` → `disk_block`
  - Some OSs utilize spare space in PTE for paged blocks
  - Like the PT, but purely software

- Where to store it?
  - *Supplemental Page Table*
  - In memory – can be compact representation if swap storage is contiguous on disk
  - Could use hash table (like Inverted PT)

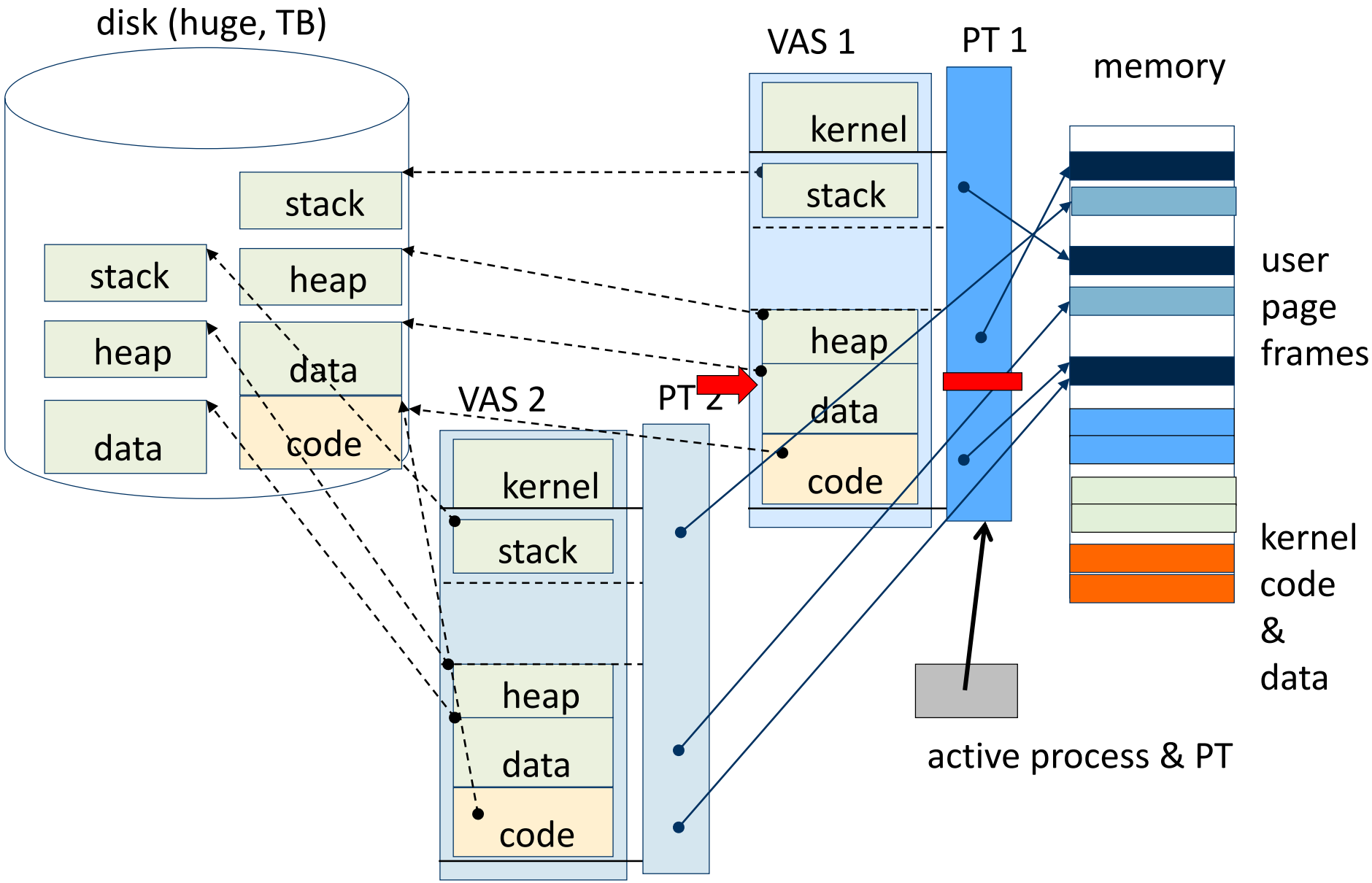
- May map code segment directly to on-disk image
  - Saves a copy of code to swap file

- May share code segment with multiple instances of the program

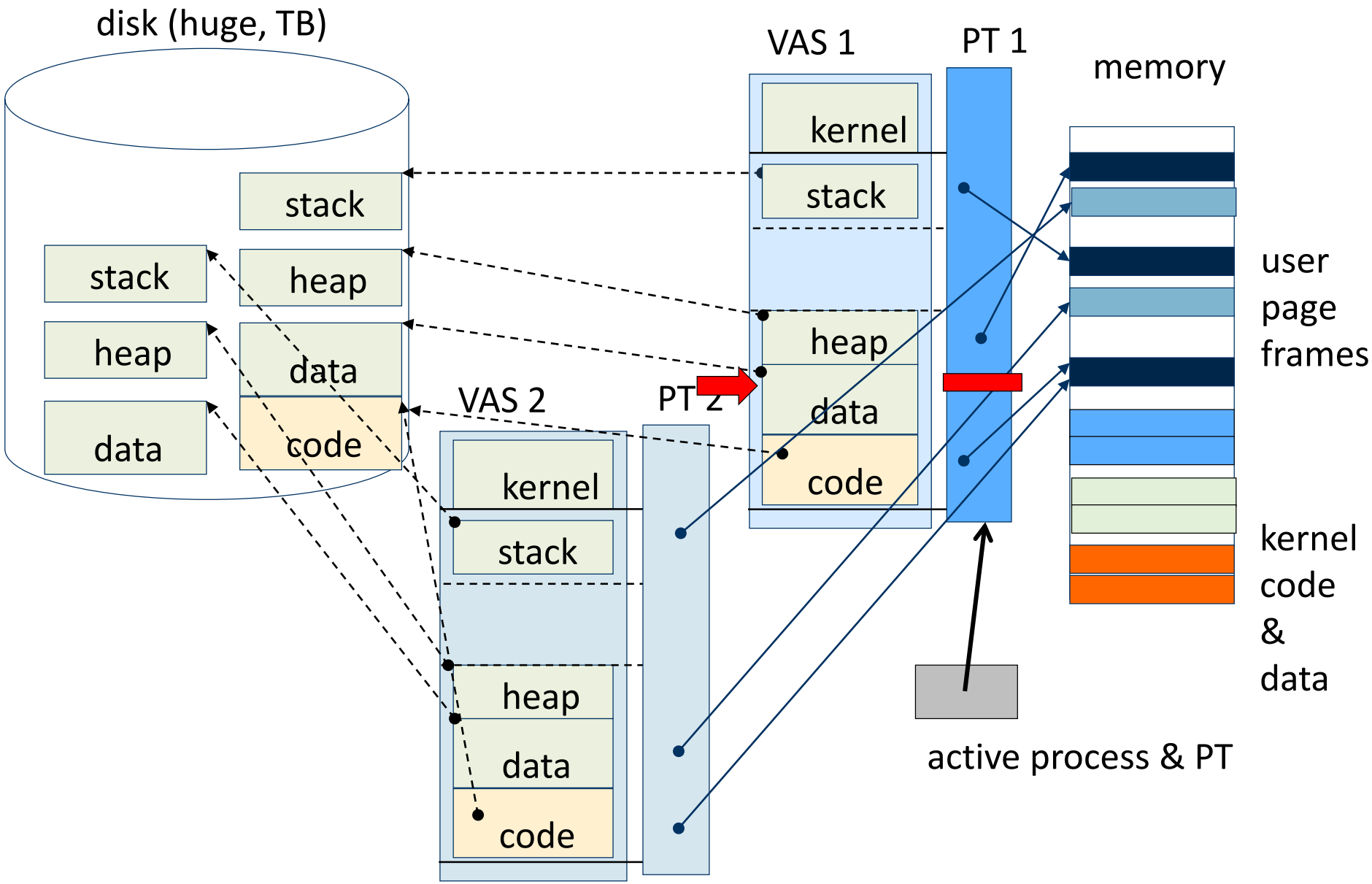
Provide Backing Store for VAS



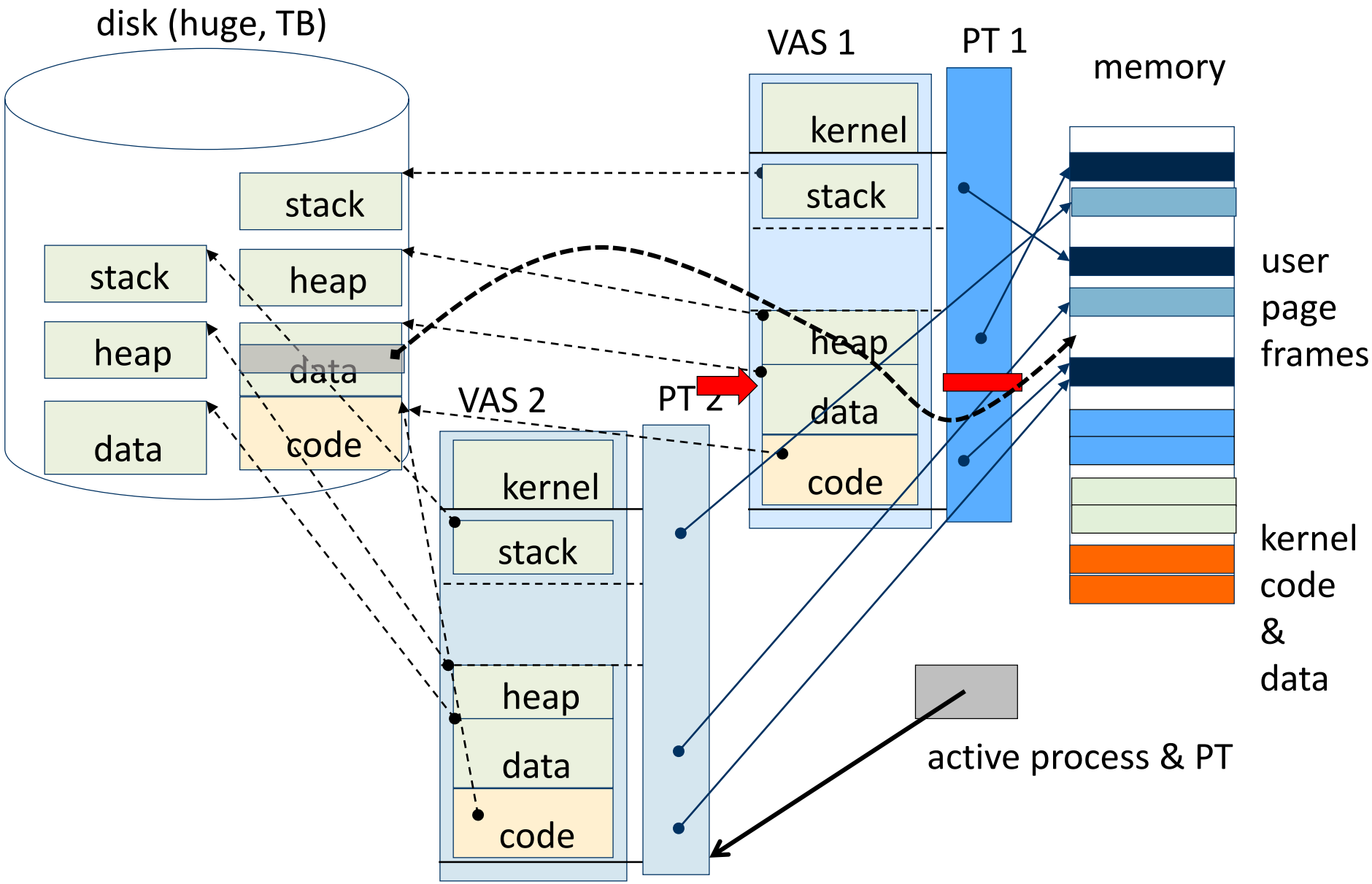
On Page Fault…



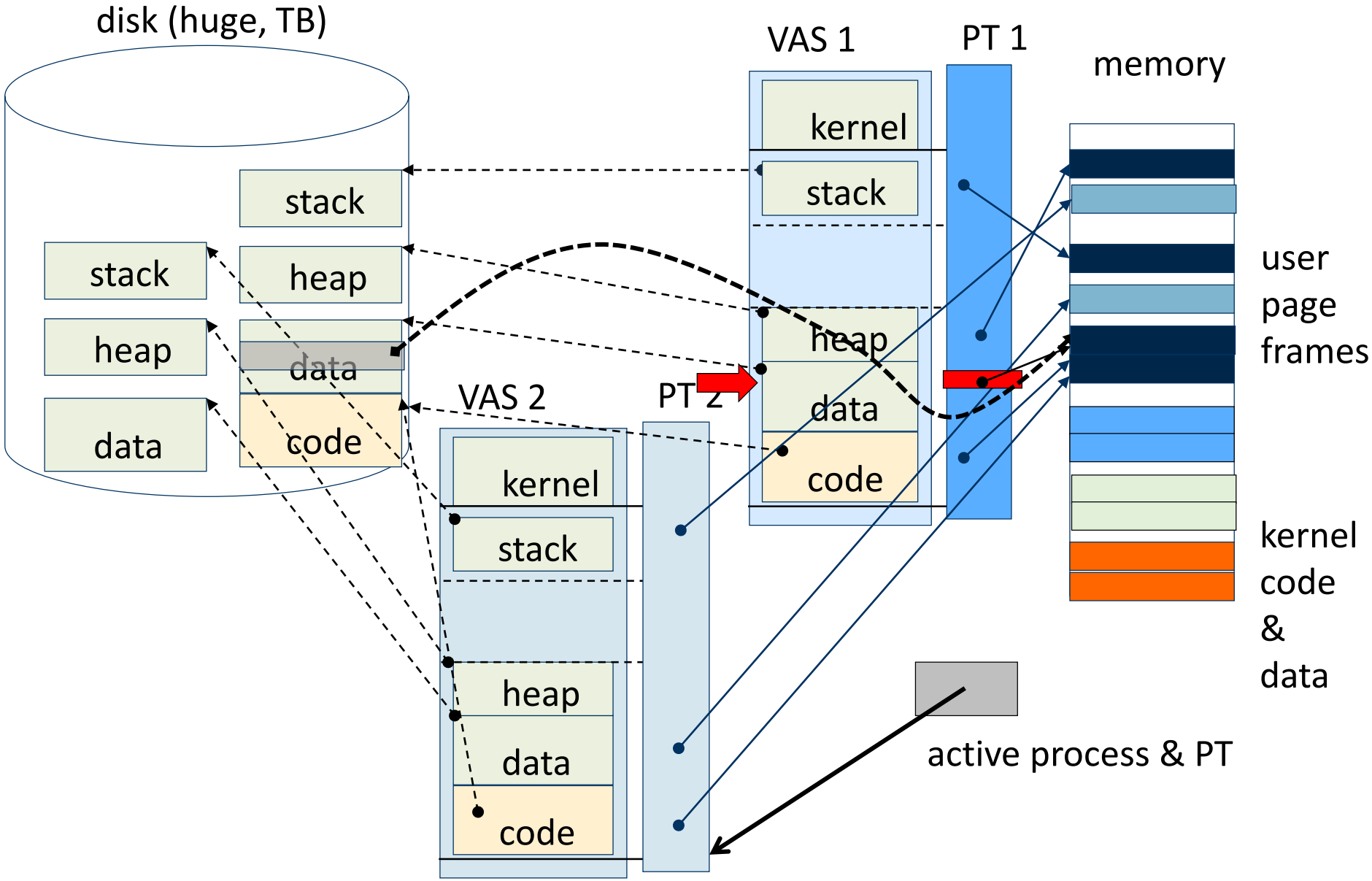
On Page Fault… find page & start load



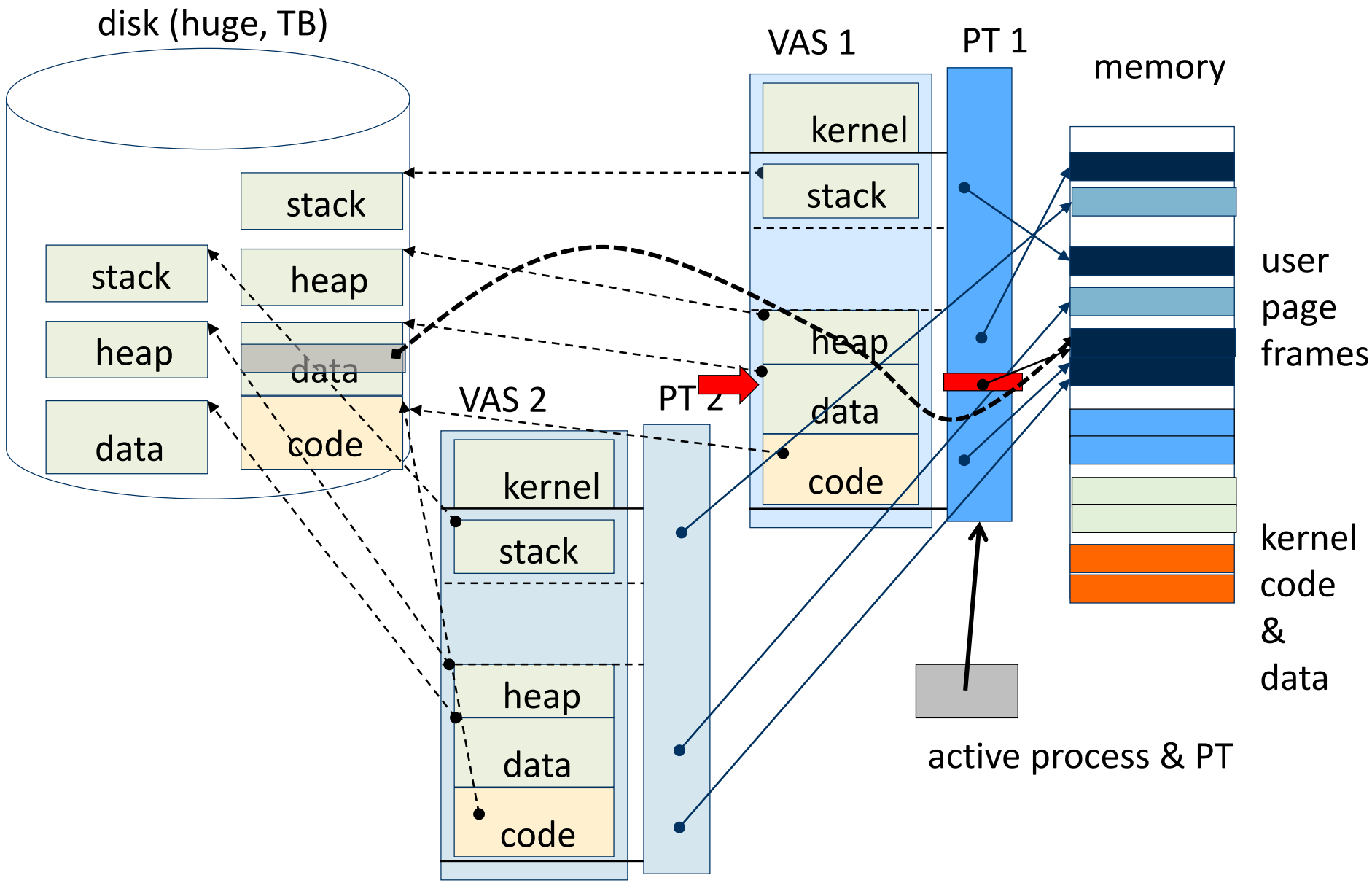
Schedule other Process or Thread



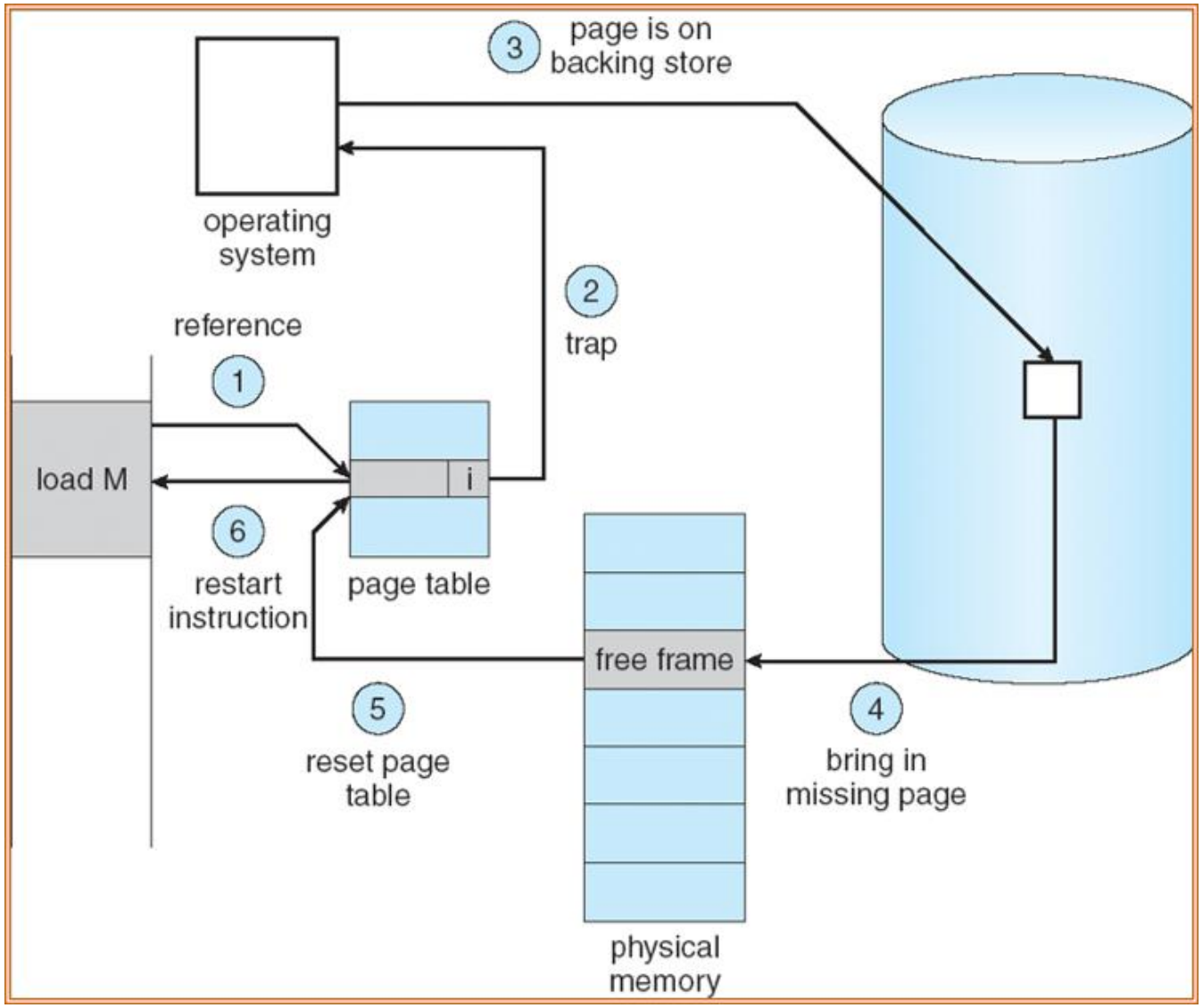
Update PTE


disk (huge, TB)
stack
heap
data
code
stack
heap
data
VAS 1
PT 1
kernel
stack
heap
data
code
memory
user
page
frames
VAS 2
PT 2
kernel
stack
heap
data
code
kernel
code
&
data
active process & PT

Eventually faulting thread is rescheduled



Steps in Handling a Page Fault (for Demand Paging)



Demand Paging Mechanisms

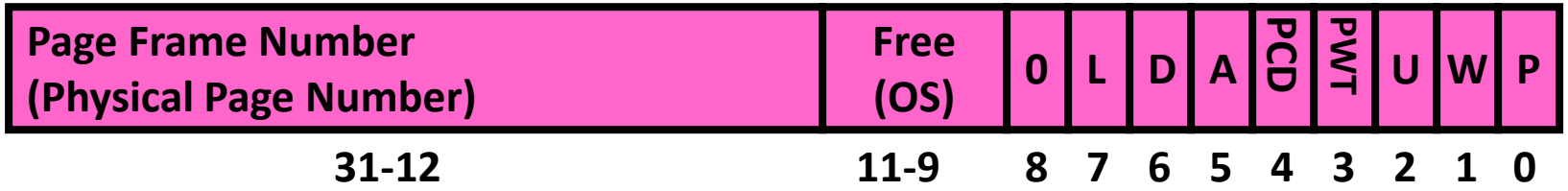
- PTE helps us implement demand paging
  - Valid ⇒ Page in memory, PTE points at physical page
  - Not Valid ⇒ Page not in memory; use info in PTE (or other) to find it on disk
- Suppose user references page with invalid PTE?
  - Memory Management Unit (MMU) traps to OS
    - Resulting trap is a "Page Fault"
  - 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
  - TLB for new page will be loaded when thread is continued!
  - While pulling pages off disk for one process, OS runs another process
    - Suspended process sits on wait queue

Demand Paging as a Form of Caching

- What is block size?
  - 1 page
- What is organization of this cache (i.e. direct-mapped, set-associative, fully-associative)?
  - Fully associative: arbitrary virtual→physical mapping
- How do we find a page in the cache when look for it?
  - First check TLB, then page-table traversal
- What is page replacement policy? (i.e. LRU, Random…)
  - This requires more explanation… (kinda LRU)
- What happens on a miss?
  - Go to lower level to fill miss (i.e. disk)
- What happens on a write? (write-through, write back)
  - Definitely write-back. Need dirty bit!

What's in a Page Table Entry (PTE)?

- What is in a Page Table Entry (or PTE)?
  - "Pointer to" (address of) next-level page table or to actual page
  - Permission bits: valid, read-only, read-write, write-only
- Example: Intel x86 architecture PTE:



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 (PTE only): page has been modified recently
L: L=1⇒4MB page (directory only).
Bottom 22 bits of virtual address serve as offset

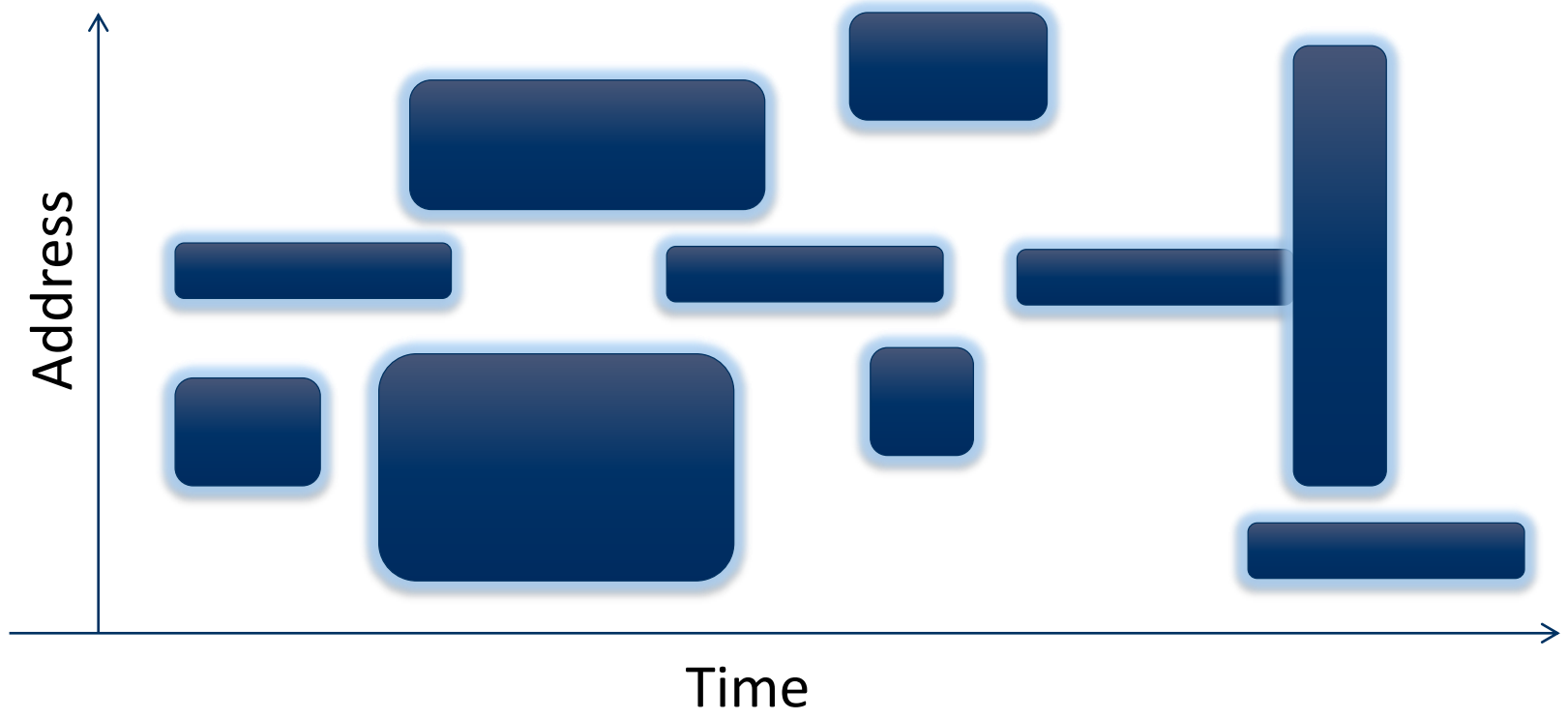
Caching in Operating Systems

- Direct use of caching techniques
  - TLB (cache of PTEs)
  - Paged virtual memory (memory as cache for disk)
  - File systems (cache disk blocks in memory)
  - DNS (cache hostname => IP address translations)
  - Web proxies (cache recently accessed pages)

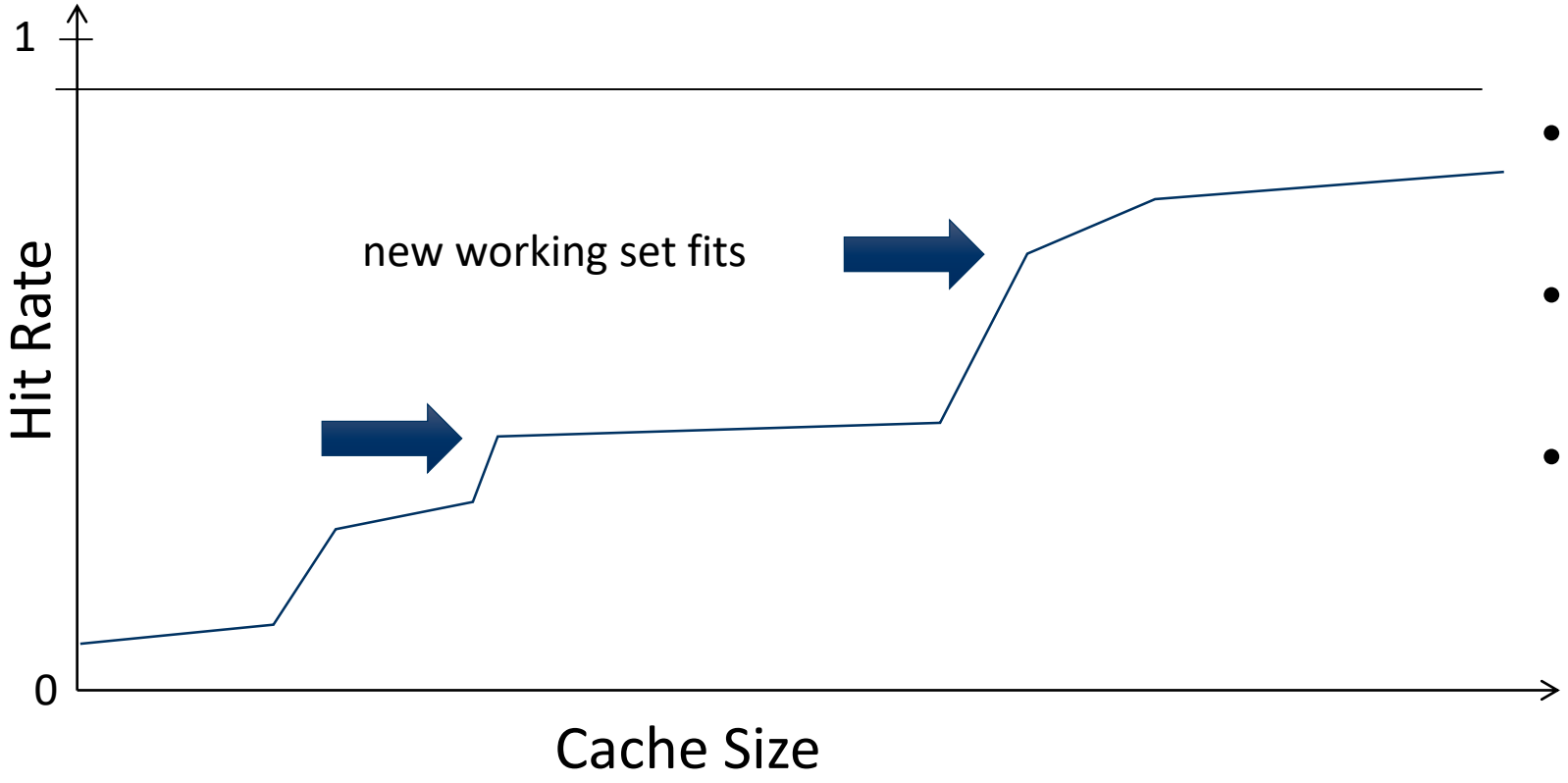
- Which pages to keep in memory?
  - All-important “Policy” aspect of virtual memory

Recall: Working Set Model

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

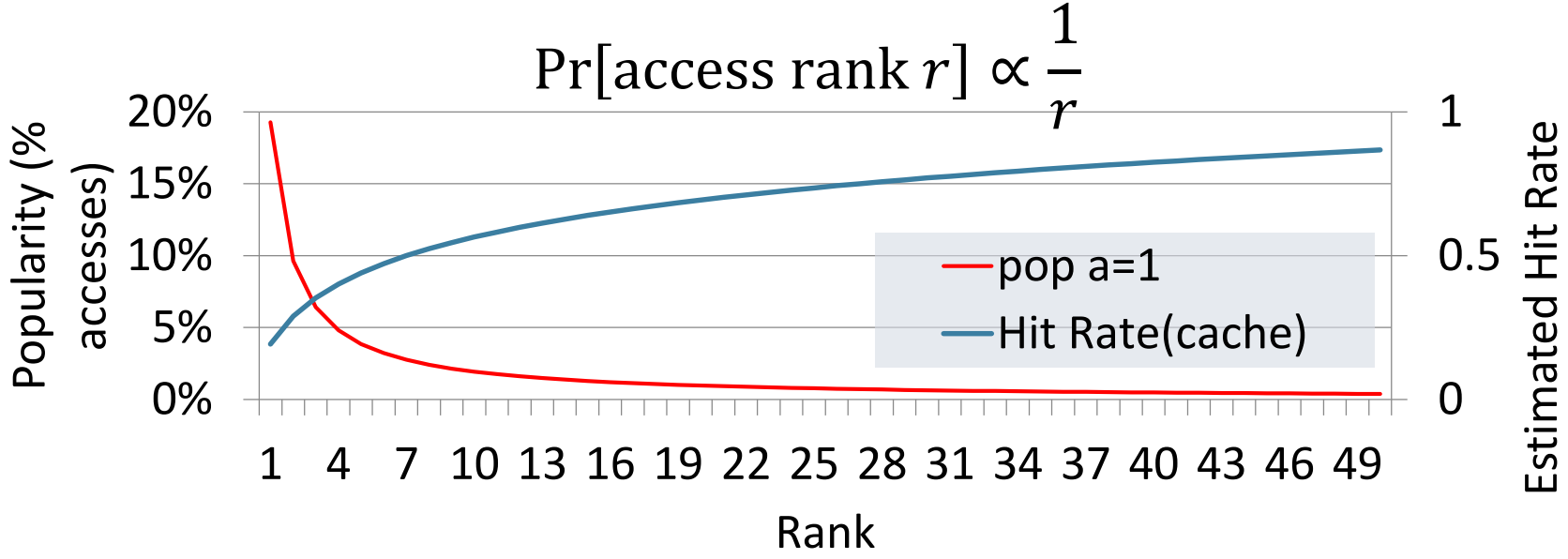


Cache Behavior under Working Set Model



- Amortized by fraction of time the Working Set is active
- Transitions from one WS to the next
- Applicable to memory caches and pages. Others ?

Another Model of Locality: Zipf



- Zipf: likelihood of accessing item of rank r is $\propto {}^{1}/_{r^a}$
- Although rare to access items below the top few, there are so many that it yields a "heavy tailed" distribution
  - Substantial value from even a tiny cache
  - Substantial misses from even a very large cache

Demand Paging Cost Model

- Demand Paging like caching, can compute average access time!
  - AMAT = Hit Rate x Hit Time + Miss Rate x Miss Time
  - AMAT = 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

    AMAT = 200ns + p x 8 ms
    = 200ns + p x 8,000,000ns

- If one access out of 1,000 causes a page fault, then AMAT = 8.2 µs:
  - This is a slowdown by a factor of 40!
- What if want slowdown by less than 10%?
  - 200ns x 1.1 < AMAT $\Rightarrow$ p < 2.5 x 10^{-6}
  - This is about 1 page fault in 400,000!

Announcements

- Project 2 design reviews today
- Homework 4 (Page Walk) released today

Recall: Sources of Cache Misses

- Compulsory (cold start or first reference): first access to a block
  - "Cold" fact of life: not a whole lot you can do about it
  - Note: If you are going to run "billions" of instruction, Compulsory Misses are insignificant
- Capacity:
  - Cache cannot contain all blocks access by the program
  - Solution: increase cache size
- Conflict (collision):
  - Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity
- Coherence (Invalidation): other process (e.g., I/O) updates memory

Why might we miss in the Page Cache?

- Compulsory Misses: Pages that have never been paged into memory before
  - Prefetching: loading them into memory before needed
  - Need to predict future somehow! More later
- Capacity Misses: Not enough memory.
  - One fix: Increase amount of DRAM (not quick fix!)
  - Another option: If multiple processes in memory: adjust percentage of memory allocated to each one!
- Conflict Misses:
  - Technically, conflict misses don't exist in virtual memory, since it is a "fully-associative" cache
- Policy Misses:
  - Caused when pages were in memory, but kicked out prematurely because of the replacement policy
  - How to fix? Better replacement policy

Page Replacement Policies

- Why do we care about Replacement Policy?
  - Replacement is an issue with any cache, but particularly important with pages
    - The cost of being wrong is high: must go to disk
    - Must keep important pages in memory, not toss them out
- FIFO (First In, First Out)
  - Throw out oldest page. Be fair – let every page live in memory for about the 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

Example: FIFO

- Suppose we have 3 page frames, 4 virtual pages, and following reference stream:
  - A B C A B D A D B C B
- Consider FIFO Page replacement:

Ref: / Page:	A	B	C	A	B	D	A	D	B	C	B
1	A					D				C	
2		B					A				
3			C						B		

- FIFO: 7 faults
- When referencing D, replacing A is bad choice, since need A again right away

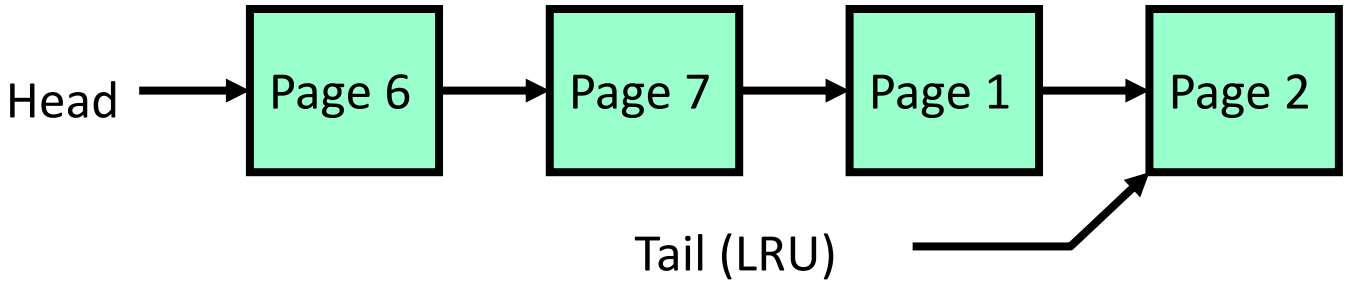
Page Replacement Policy: MIN

- MIN (Minimum):
  - Replace page that won't be used for the longest time
  - Great (provably optimal), but can't really know future…
    - Clairvoyant algorithm
  - Also called Belady's Algorithm of Belady's Theoretically Optimal Paging

- ***But past is a good predictor of the future …***

Page Replacement Policy: LRU

- LRU (Least Recently Used):
  - Replace page that hasn't been used for the longest time
  - Relies on temporal locality
- How to implement LRU? Use a list!



- Approximates MIN based on temporal locality

Example: MIN/LRU

- Suppose we have the same reference stream:
  - A B C A B D A D B C B
- Consider MIN Page replacement:

Ref: / Page:	A	B	C	A	B	D	A	D	B	C	B
1	A									C	
2		B									
3			C			D					

- MIN: 5 faults
- What will LRU do?
  - Same decisions as MIN here, but not true in general!

Is LRU guaranteed to perform well?

- Consider the following: A B C D A B C D A B C D
- LRU Performs as follows (same as FIFO here):

Ref: / Page:	A	B	C	D	A	B	C	D	A	B	C	D
1	A			D			C			B		
2		B			A			D			C	
3			C			B			A			D

  - Every reference is a page fault!
- Example of “Sequential Flooding”

Is LRU guaranteed to perform well?

- LRU Performs as follows (same as FIFO here):

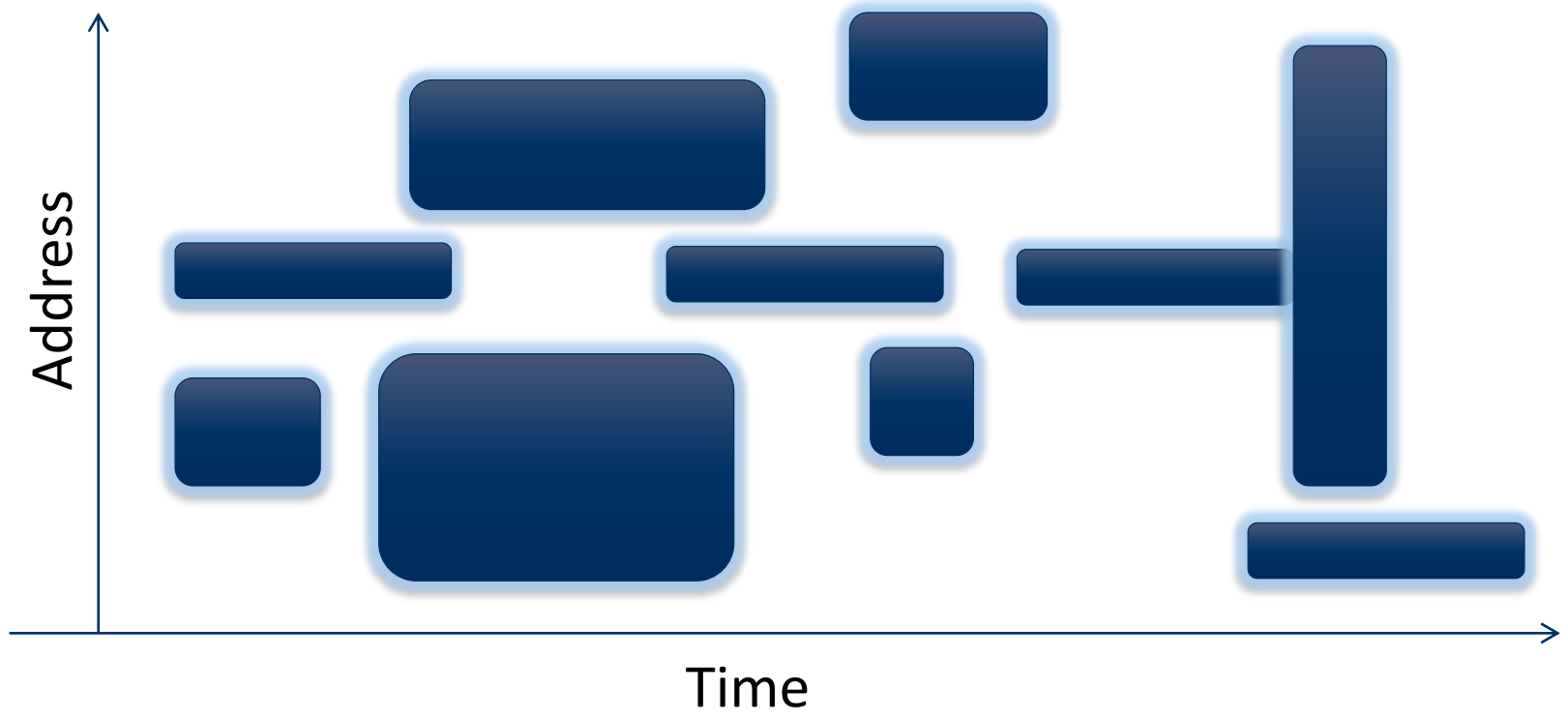
Ref: / Page:	A	B	C	D	A	B	C	D	A	B	C	D
1	A			D			C			B		
2		B			A			D			C	
3			C			B			A			D

- MIN does much better!

Ref: / Page:	A	B	C	D	A	B	C	D	A	B	C	D
1	A									B		
2		B					C					
3			C	D								

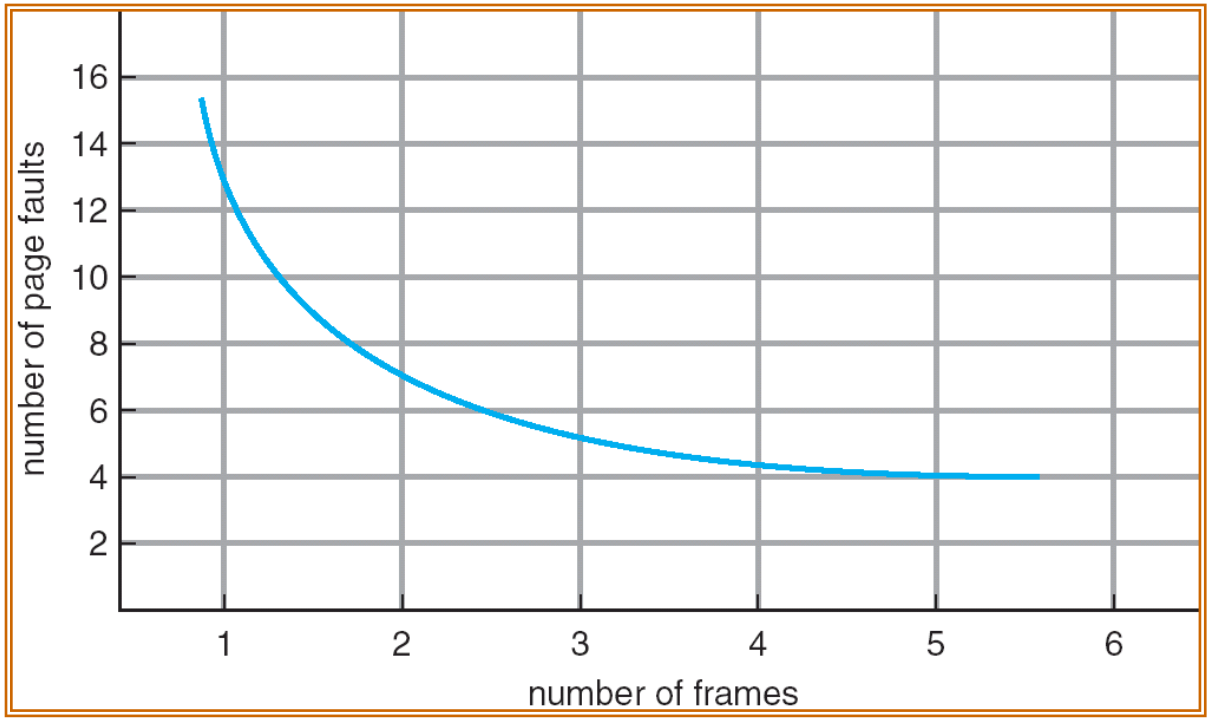
Why LRU Often Works Well: Working Sets!

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



Increasing the Memory Size

- One desirable property: When you add memory the miss rate drops
  - Called the stack property
- Surprisingly, certain replacement algorithms don't have this property!
  - Called Bélády's Anomaly



Bélády's Anomaly

- FIFO example:

- After adding memory:
  - Resident pages could be totally different
  - With LRU/MIN, always a superset

Ref: Page:	A	B	C	D	A	B	E	A	B	C	D	E
1	A			D			E					
2		B			A					C		
3			C			B					D	

Ref: Page:	A	B	C	D	A	B	E	A	B	C	D	E
1	A						E				D	
2		B						A				E
3			C						B			
4				D						C		

Problems with LRU

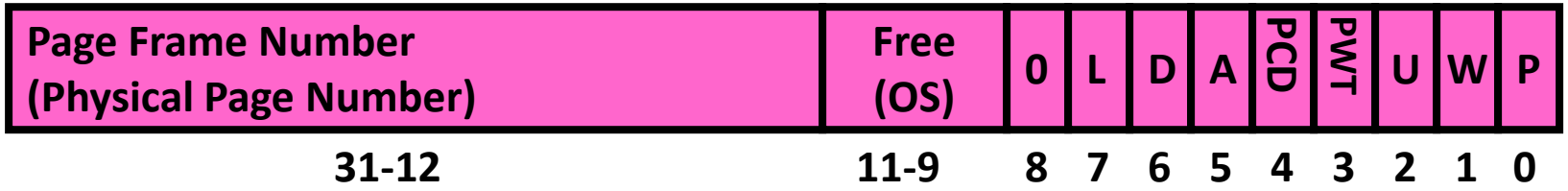
- Not optimal (to be expected)

- How to implement LRU?
  - Requires mutating linked list on every memory access
  - Trap to OS on every memory access?
    - Way too slow
  - Have hardware manipulate a linked list?
    - Too complex

- We will use hardware support to *approximate* LRU

What's in a Page Table Entry (PTE)?

- What is in a Page Table Entry (or PTE)?
  - "Pointer to" (address of) next-level page table or to actual page
  - Permission bits: valid, read-only, read-write, write-only
- Example: Intel x86 architecture PTE:

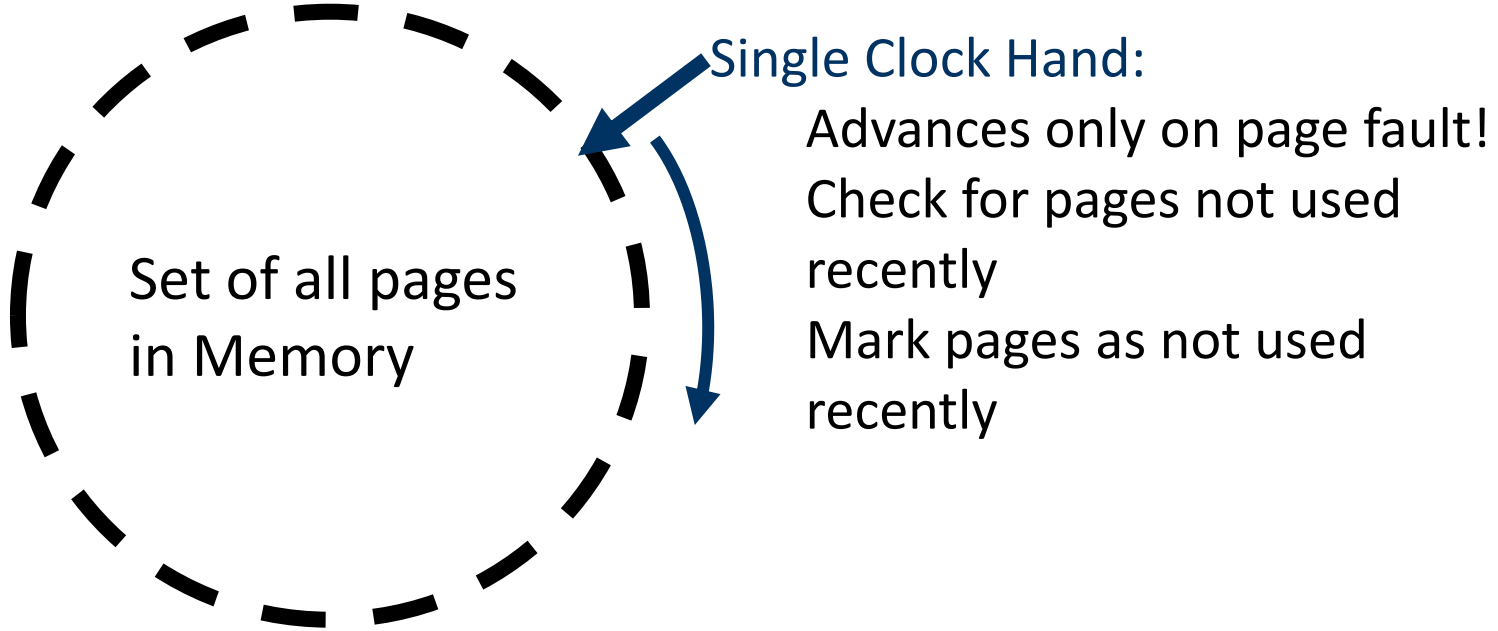


P: Present (same as "valid" bit in other architectures)
W: Writeable
U: User accessible
PWT: Page write transparent: external cache write-through
PCD: Page cache disabled (page cannot be cached)
A: Accessed: page has been accessed recently
D: Dirty (PTE only): page has been modified recently
L: L=1⇒4MB page (directory only).
Bottom 22 bits of virtual address serve as offset

Approximating LRU: Clock Algorithm

- Clock Algorithm (NRU): Arrange physical pages in circle with single clock hand
  - Approximate LRU (*approximation to approximation to MIN*)
  - Replace an old page, not the oldest page
- Details:
  - Hardware sets "use" bit ("accessed" bit) in PTE on each reference
    - Some hardware sets use bit in the TLB, with writeback to PTE
  - On page fault:
    - Advance clock hand (not real time)
    - Check use bit: 1→used recently; clear use bit and continue advancing clock hand
    - 0→not used recently; choose this page for replacement

Clock Algorithm: Not Recently Used



- What if hand moving slowly?
  - Good sign or bad sign?
    - Not many page faults
    - Or find page quickly
- What if hand is moving quickly?
  - Lots of page faults and/or lots of reference bits set
- One way to view clock algorithm:
  - Crude partitioning of pages into two groups: young and old
  - Why not partition into more than 2 groups?

N^{th} Chance Version of Clock Algorithm

- N^{th} chance algorithm: Give page N chances
  - OS keeps counter per page: # sweeps
  - On page fault, OS checks use bit:
    - 1 $\rightarrow$ clear use and also clear counter (used in last sweep)
    - 0 $\rightarrow$ increment counter; if count=N, replace page
  - Means that clock hand has to sweep by N times without page being used before page is replaced
- How do we pick N?
  - Why pick large N? Better approximation to LRU
  - Why pick small N? More efficient
- What about dirty pages?
  - Takes extra overhead to replace a dirty page, so give dirty pages an extra chance before replacing?
  - One approach:
    - Clean pages, use N=1
    - Dirty pages, use N=2 (and write back to disk when N=1)

Clock-Based Algorithms

- Which bits of a PTE entry are useful to us?
  - Use: Set when page is referenced; cleared by clock algorithm
  - Modified: set when page is modified, cleared when page written to disk
  - Valid: ok for program to reference this page
  - Read-only: ok for program to read page, but not modify
    - For example for catching modifications to code pages!
- We rely on hardware support via the “use” bit and “modified” bits

Discussion: Hardware Support

- Do we really need hardware support? No!
  - Can emulate “use” and “modified” bits by marking all pages invalid and trapping to OS
  - On use, set use bit and then mark page as “read-only”
  - On write, set use/modified bits and then mark page as “read-write”

- Given that, without hardware support, we have to take some extra page faults, is there a better approximation of LRU we can use?
  - Second-Chance List

Summary

- Replacement policies
  - FIFO: Place pages on queue, replace page at end
  - MIN: Replace page that will be used farthest in future
  - LRU: Replace page used farthest in past
- Clock Algorithm (NRU): Approximation to LRU
  - Arrange all pages in circular list
  - Sweep through them, marking as not “in use”
  - If page not “in use” for one pass, than can replace
- N^{th}-chance clock algorithm: Another approximate LRU
  - Give pages multiple passes of clock hand before replacing
- Second-Chance List algorithm: Yet another approximate LRU
  - Divide pages into two groups, one of which is truly LRU and managed on page faults.
- Working Set:
  - Set of pages touched by a process recently