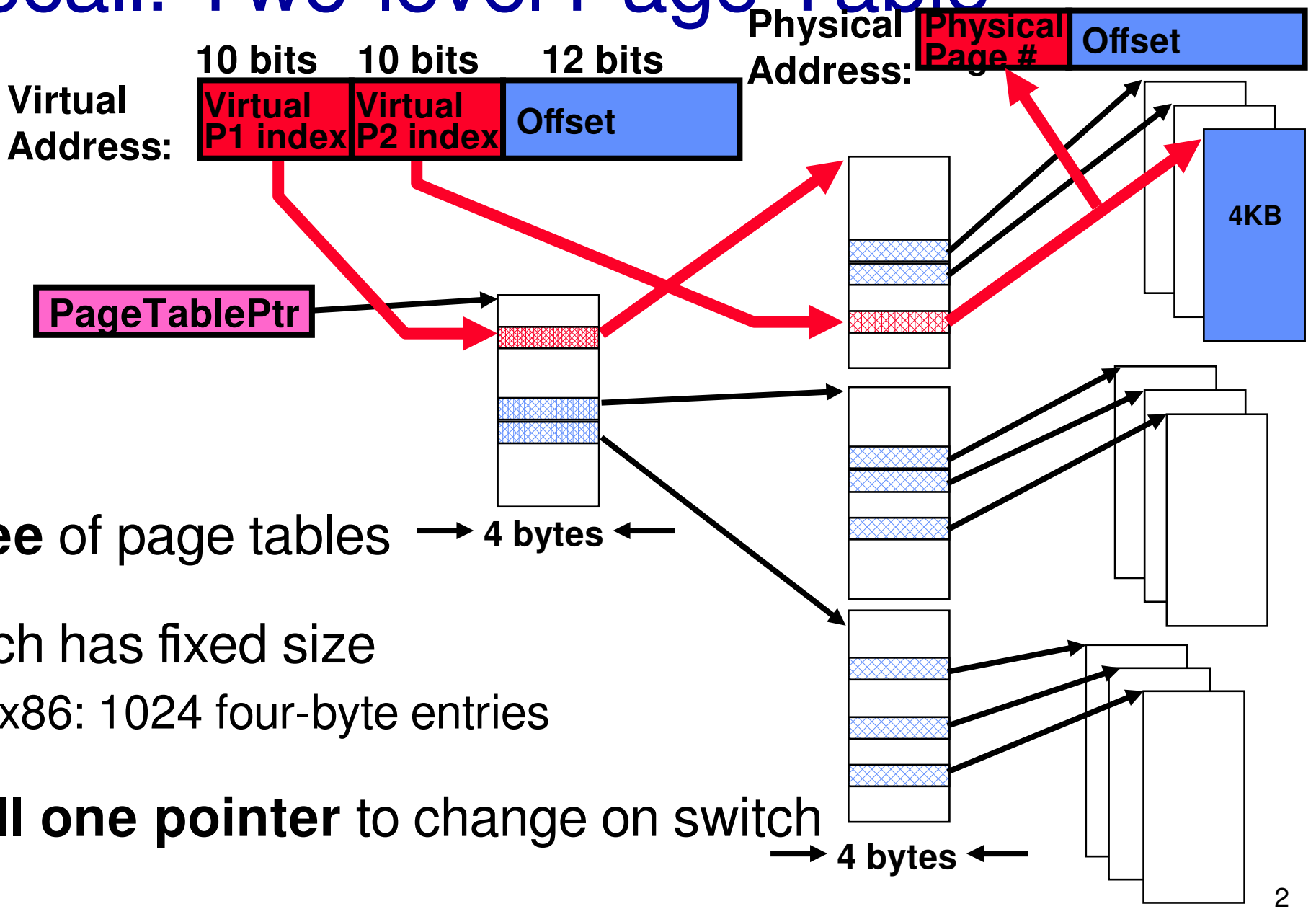
CS162:
Operating Systems and Systems Programming

Lecture 12:
Demand Paging

9 July 2015
Charles Reiss
https://cs162.eecs.berkeley.edu/

Recall: Two-level Page Table



Tree of page tables

Each has fixed size

- x86: 1024 four-byte entries

Still one pointer to change on switch

Recall: Paging Tricks

What does invalid Page Table Entry (PTE) mean?

- Region of address space is invalid *or*
- Page is just somewhere else/not ready yet

When program accesses invalid PTE, OS gets an *exception* (a *page fault* or *protection fault*)

Options:

- Crash program (it's actually invalid)
- **Get page ready and restart instruction**

Recall: Paging Tricks: Examples

Demand Paging

- **Swapping for pages**
- **Keep only active pages in memory**
- **When exception occurs for page not in memory, load from disk and retry**

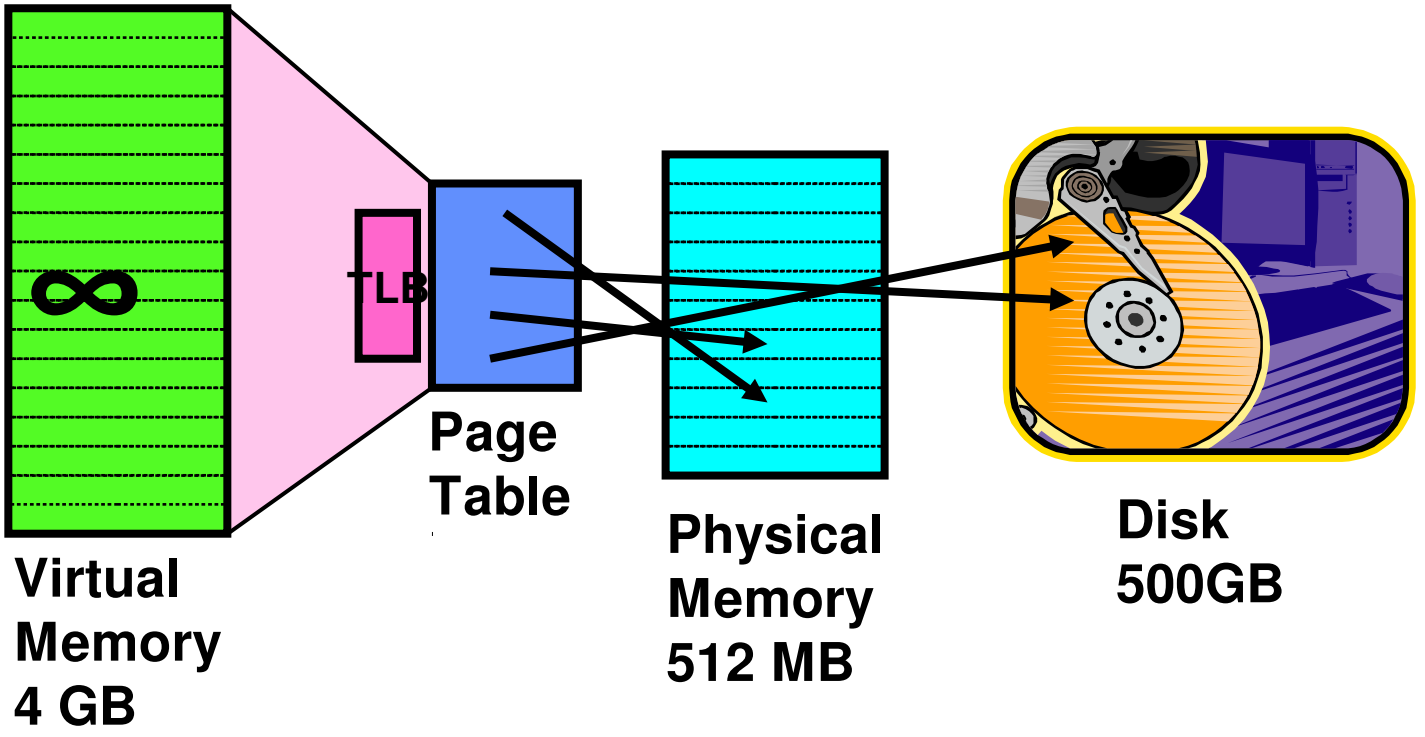
Copy on Write

- Remember fork() – copy of address space
- Instead of real copy, mark pages ***read-only***
- Allocate new pages on protection fault

Zero-Fill On Demand

- New pages should be zeroed out – slow!
- Instead, pages start invalid – create new zero page when accessed

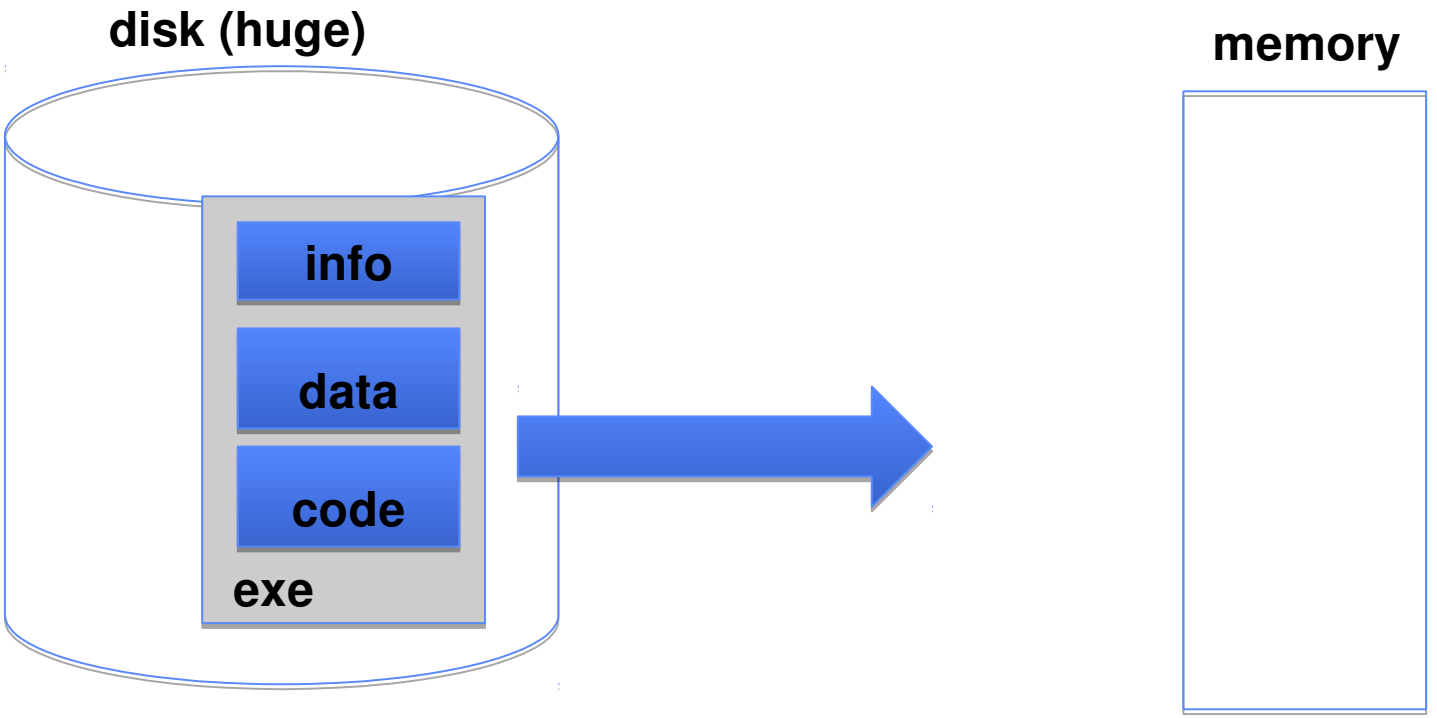
Recall: Illusion of "infinite" memory



How? **Transparent layer of indirection**

- the page table + page fault handlers

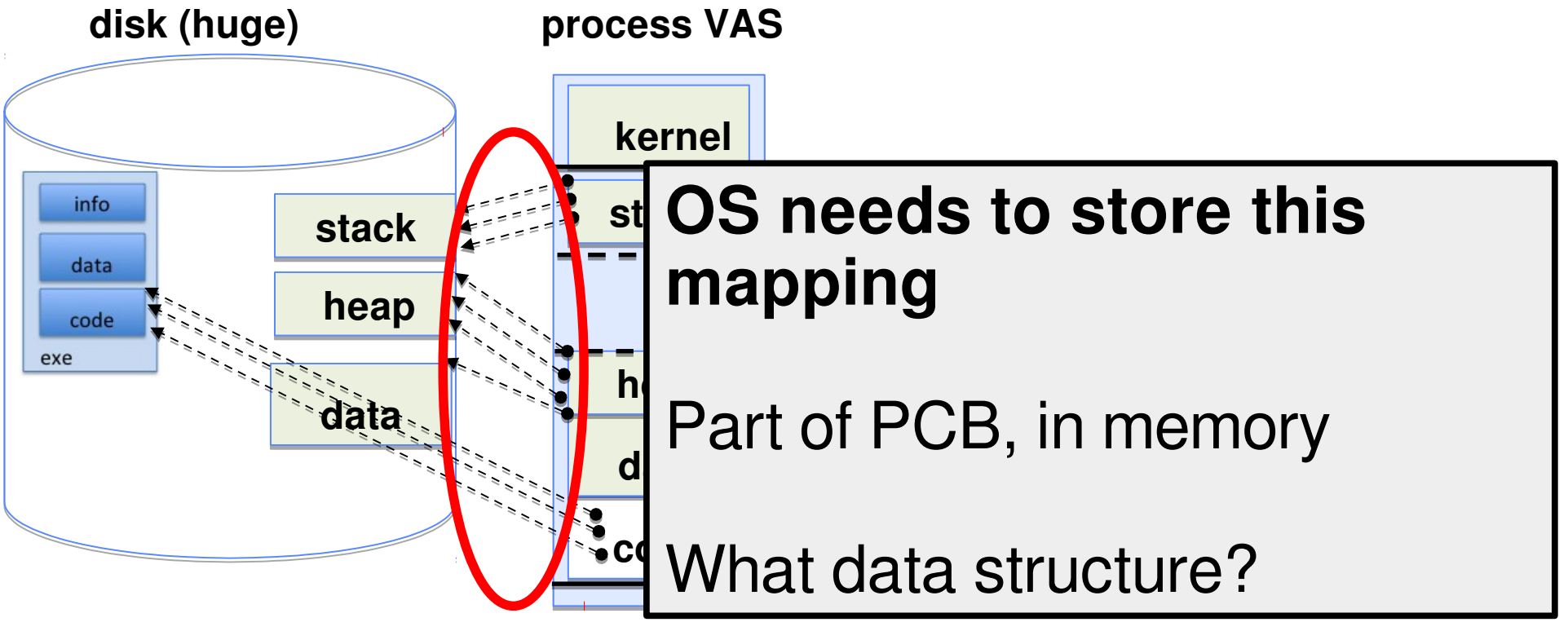
Recall: Loading a Program



View so far: OS copies each segment into memory

Then sets up registers, jumps to start location

New View: Create Address Space

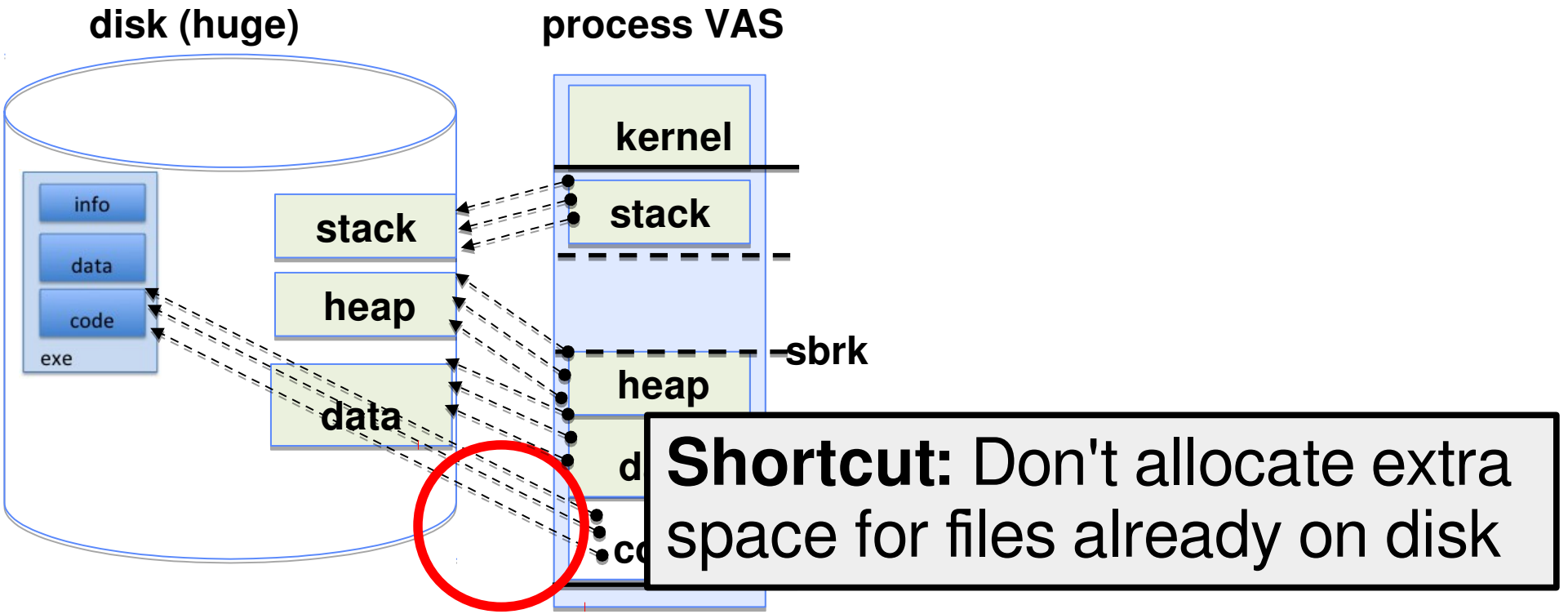


One method: Everything **backed by disk**

Just allocate space on disk

- Let page faults trigger it actually being read from disk

New View: Create Address Space

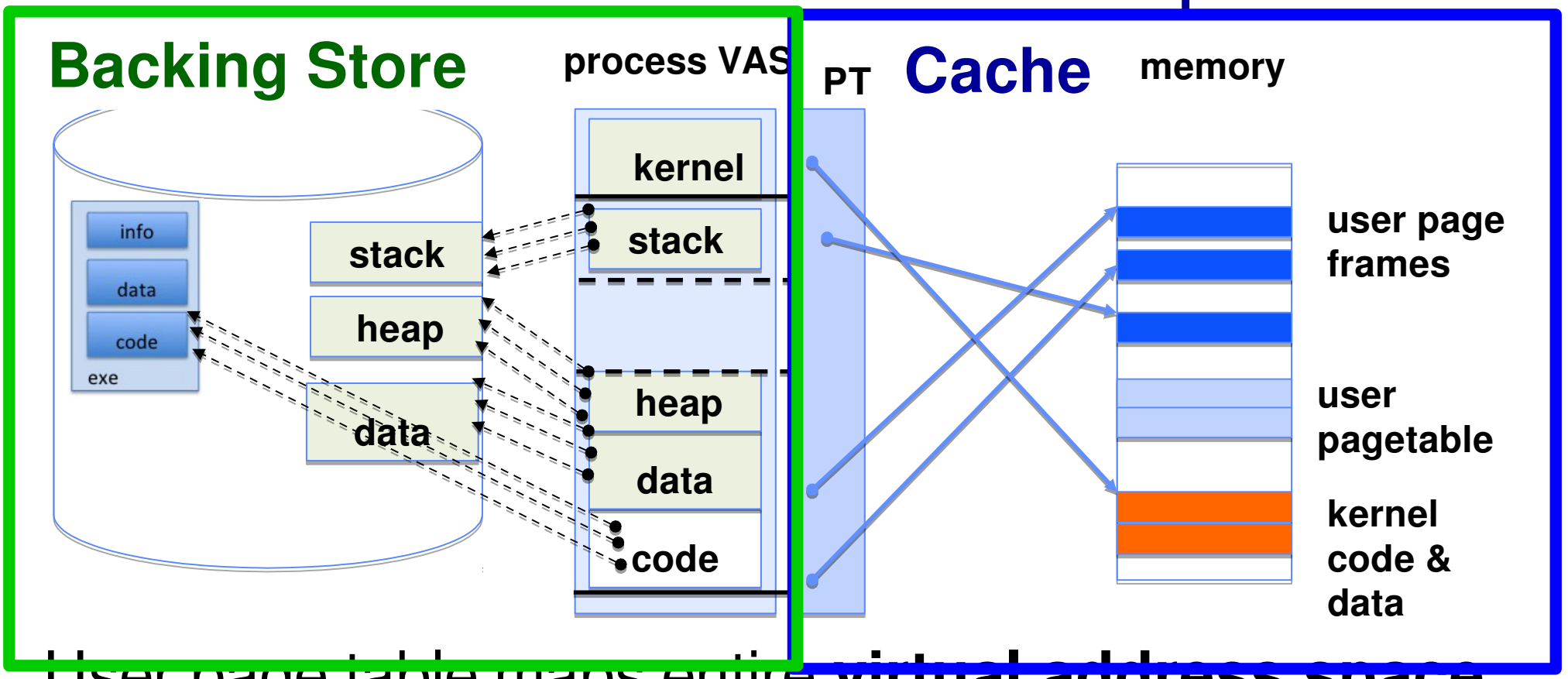


One method: Everything **backed by disk**

Just allocate space on disk

- Let page faults trigger it being read from disk

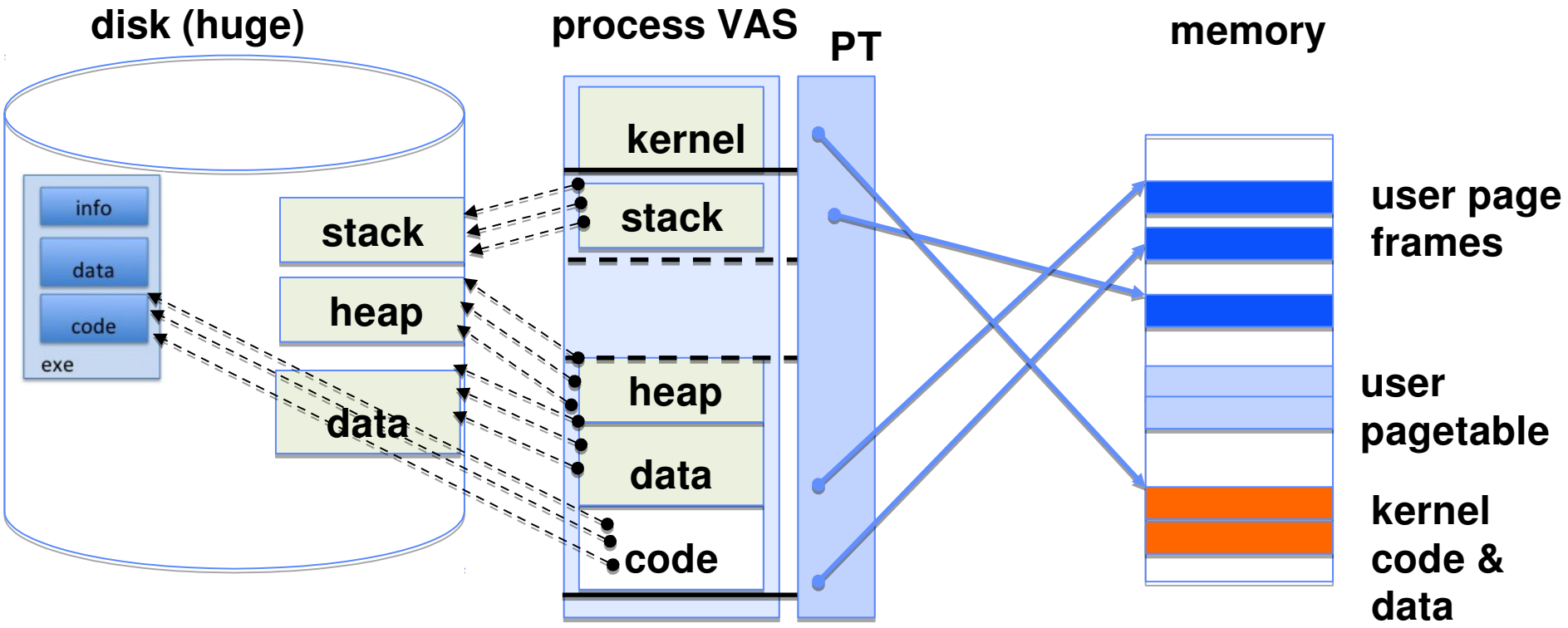
New View: Create Address Space



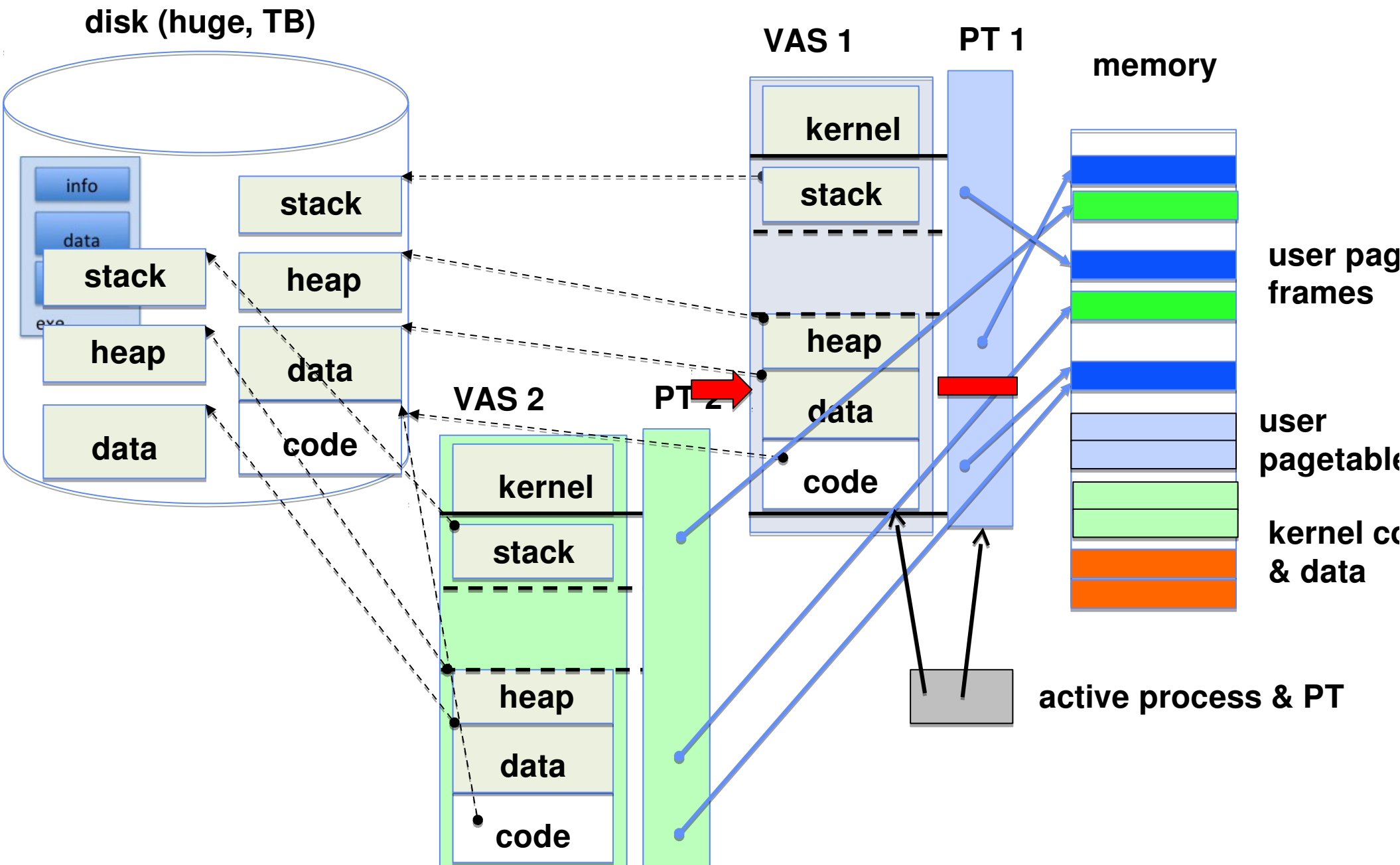
User page table maps entire **virtual address space**

- Only **resident** pages present
- One-to-one correspondence to OS's mapping

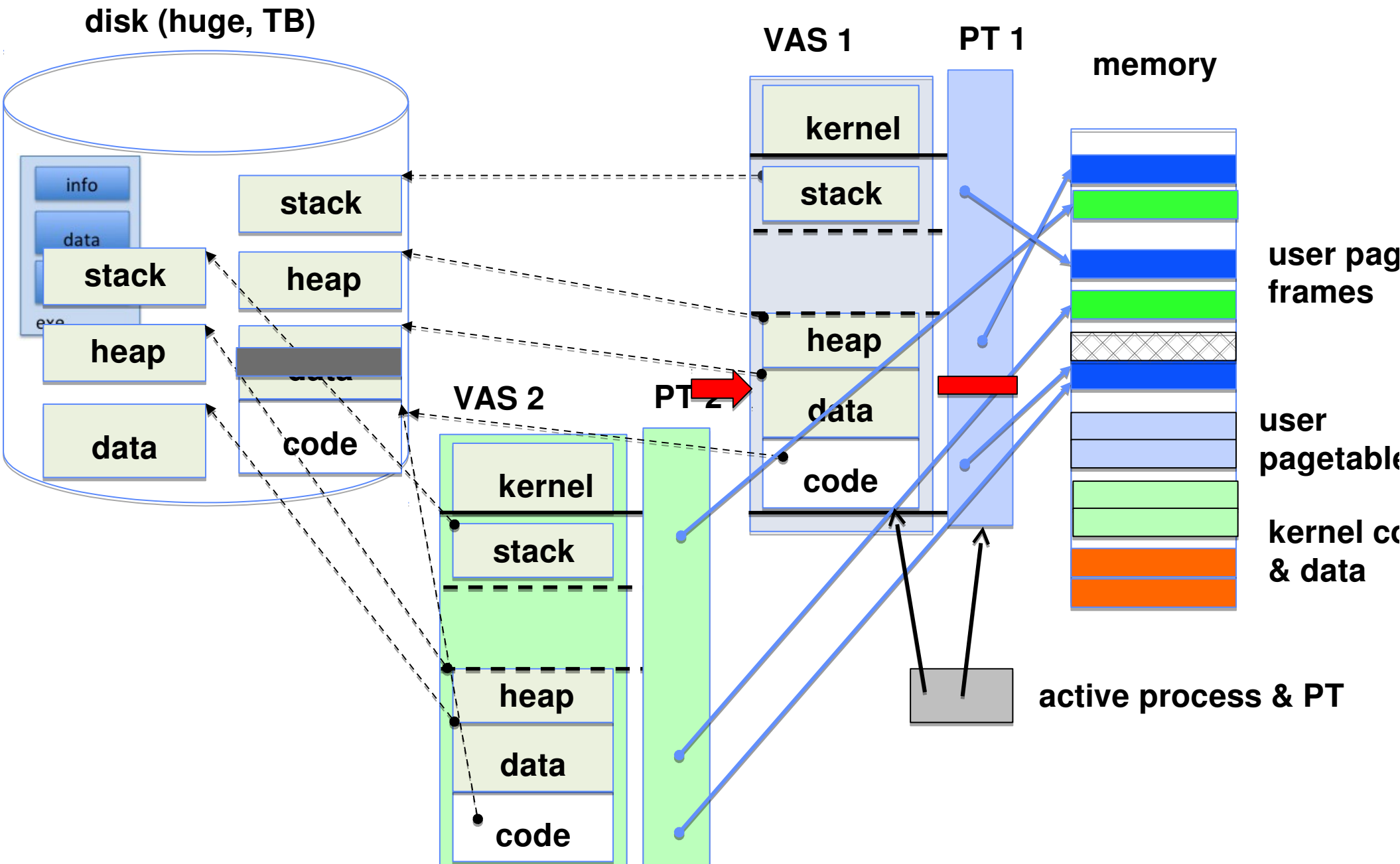
Process Address Space



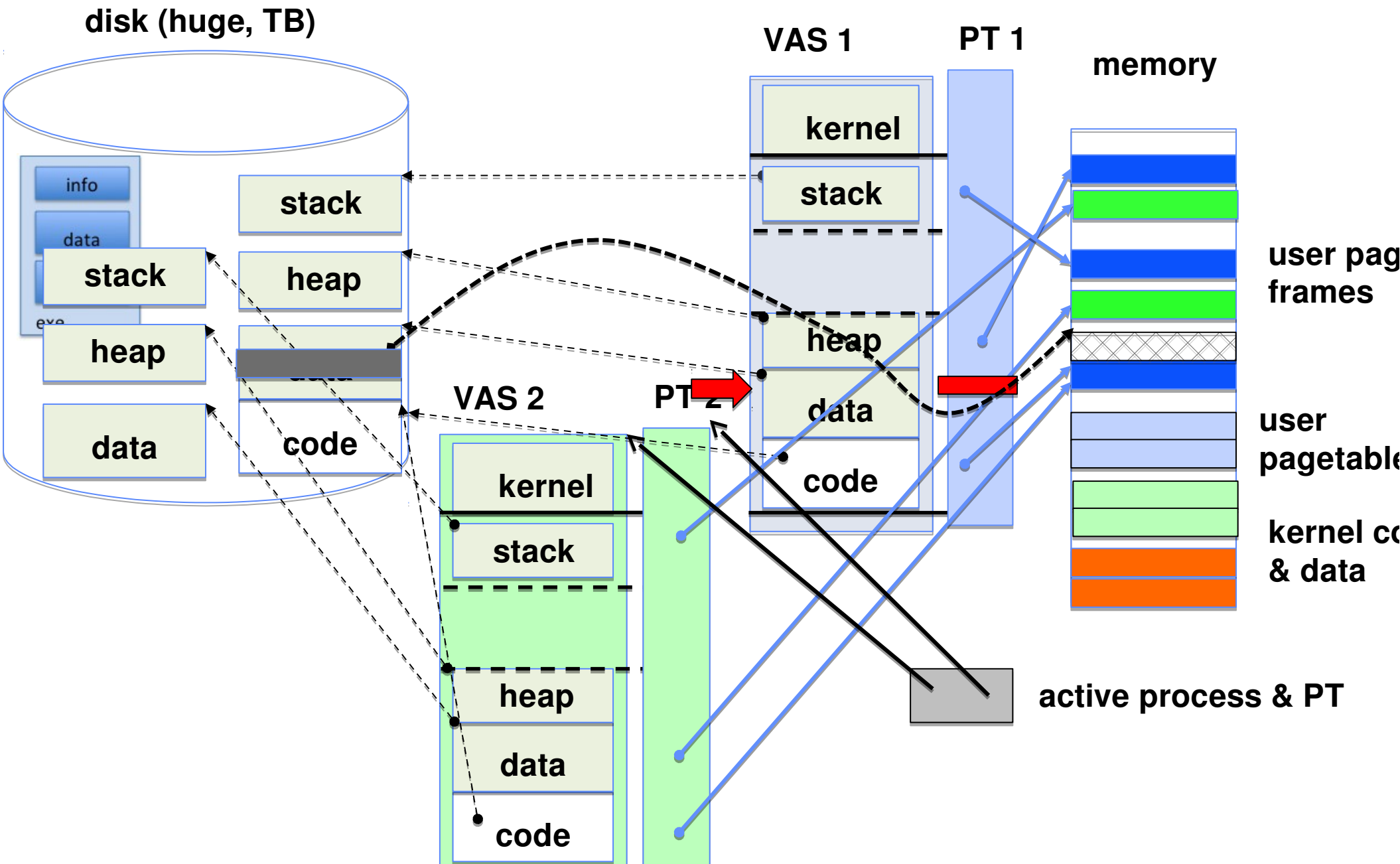
A Page Fault


disk (huge, TB)
info
data
exe
stack
heap
data
data
stack
heap
data
code
VAS 1
PT 1
kernel
stack
heap
data
code
memory
user pag
frames
user
pagetable
kernel co
& data
VAS 2
PT 2
kernel
stack
heap
data
code
active process & PT

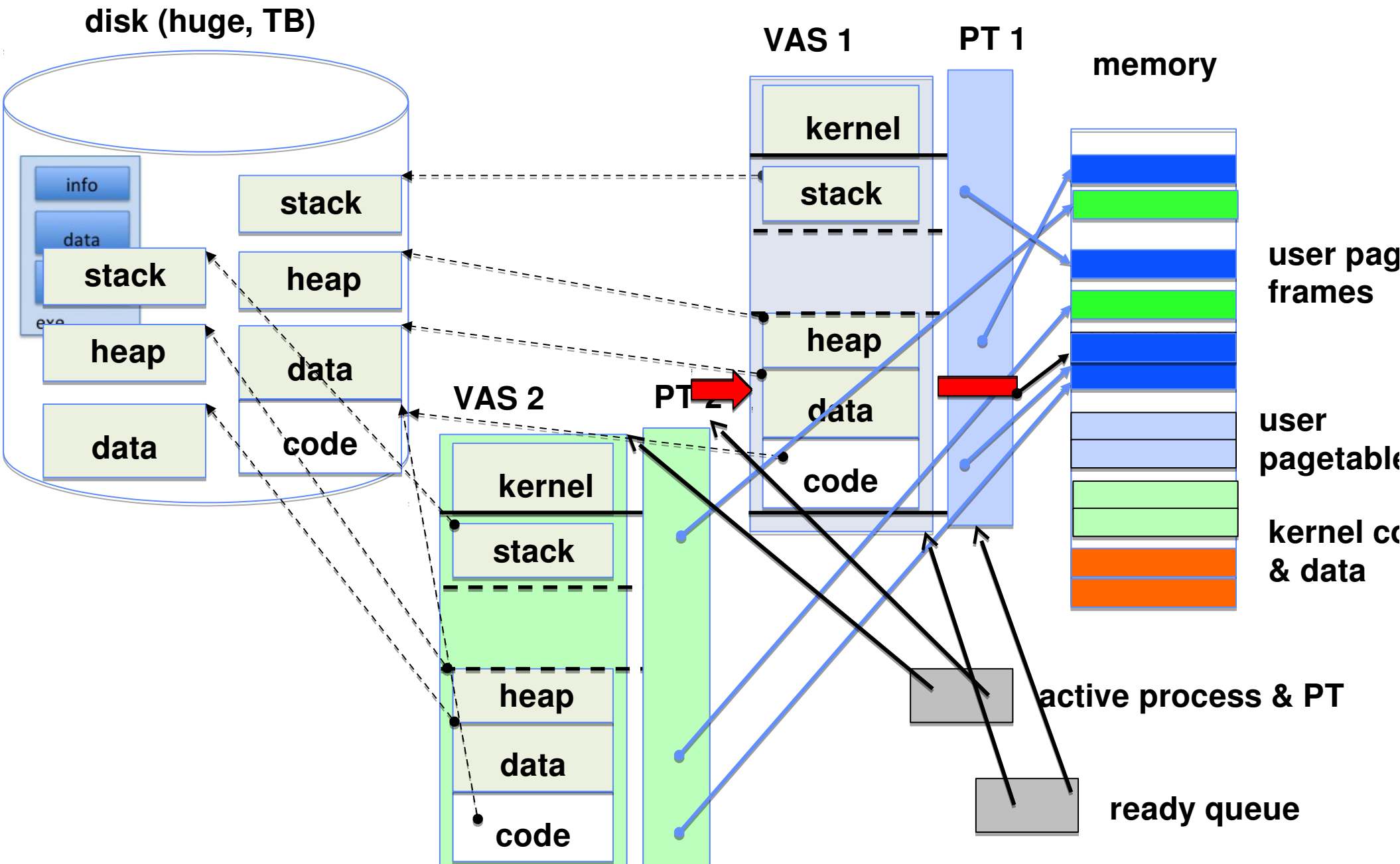
A Page Fault: Find + Start Loading



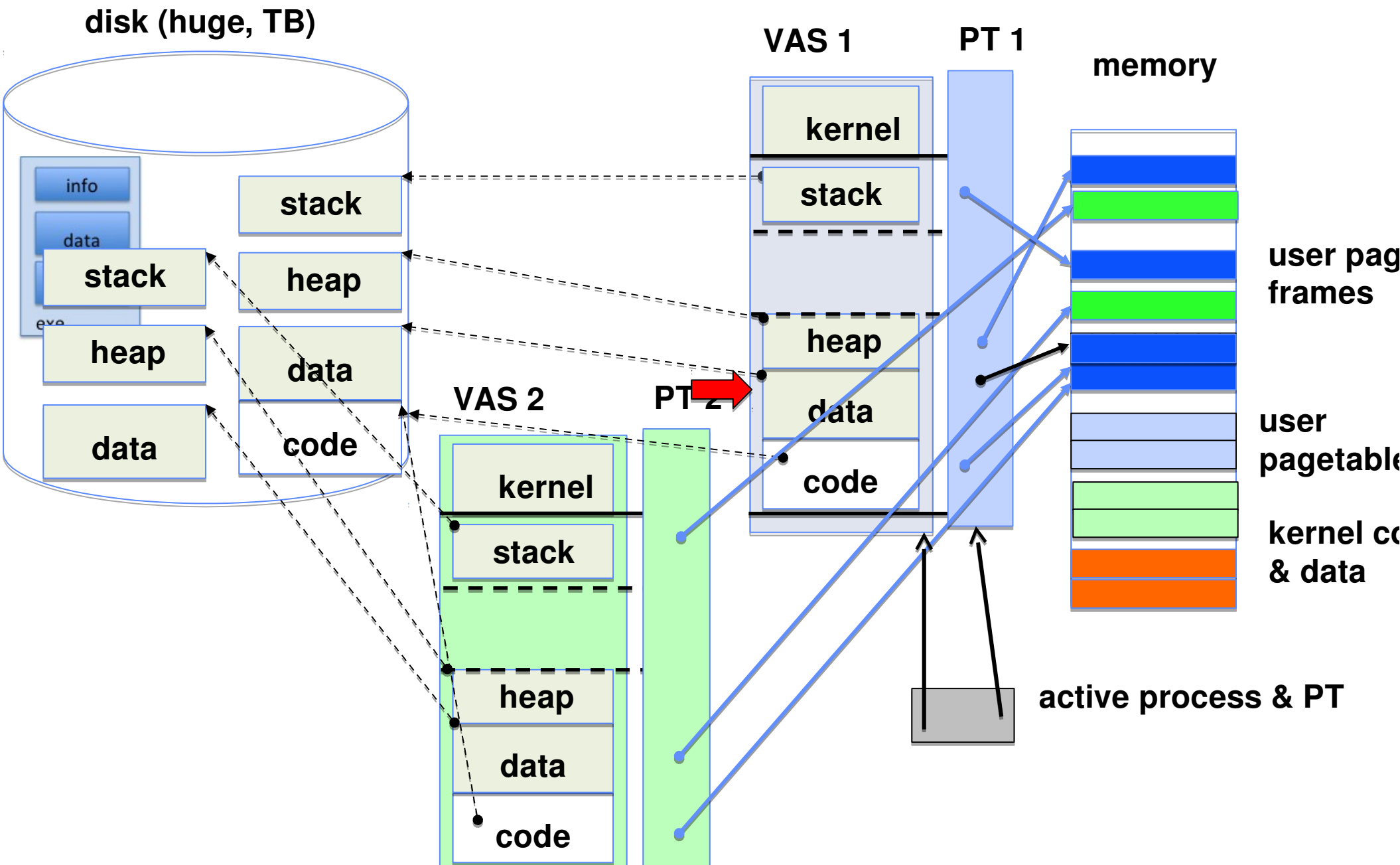
A Page Fault: Switch during IO



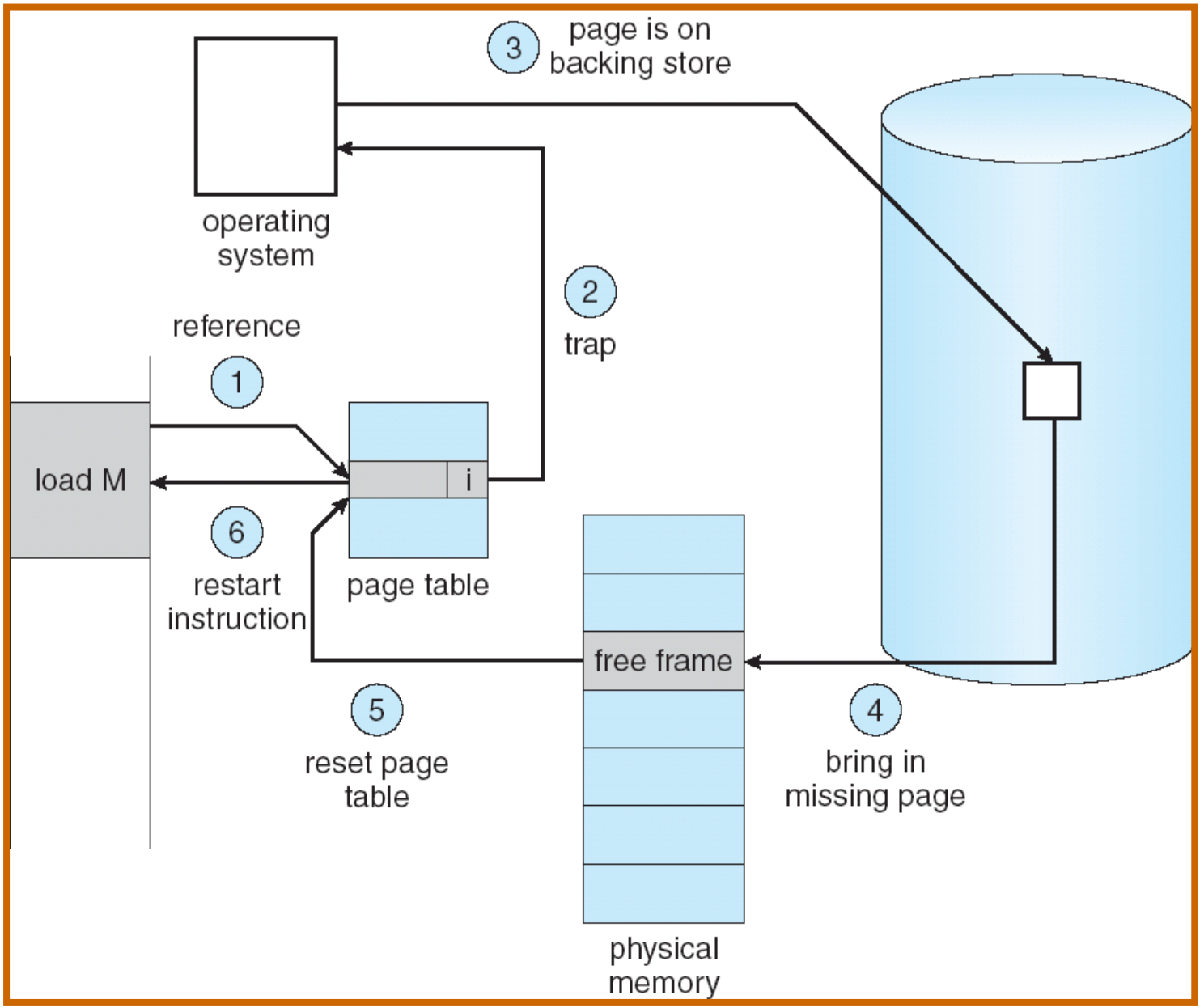
A Page Fault: Update PTE



A Page Fault: Reschedule



A Page Fault: Summary



Page Replacement

Where does free page come from?

- **Free list** (hopefully)
- Evict a page on demand (slow)

Ideally evict pages *in advance* – pool of free pages

What page gets evicted?

Could be another process's

Metrics like scheduling:

- **Utilization**
- **Fairness**
- **Priority**

Effective Access Time

Just like for processor caches:

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

Example:

- *Hit time* = Memory access time = 200 ns
- *Miss penalty* = Page fault service time = 8 ms
- EAT = 200 ns + *miss rate* x 8 ms

Miss rate of 0.1%: EAT = 8200 ns

Want EAT = 220 ns $\rightarrow$ Miss rate = ***0.00025%***

Recall: Types of Cache Misses

Compulsory ("cold start"): first access to a block

- Insignificant in any long-lived program

Capacity: not enough space in cache

Conflict: memory locations map to same cache location

Coherence (invalidation): memory updated externally

- Example: multiple cores, I/O

Eliminating Cache Misses

Compulsory ("cold start"): first access to a block

- Insignificant in any long-lived program

Probably still significant with demand paging

Mitigate: **Prefetching** (load early)

Example: *Load pages around accessed page*

Eliminating Cache Misses

Compulsory ("cold start"): first access to a block

- Insignificant in any long-lived program

Capacity: not enough space in cache

Could take memory from other programs

Otherwise, need to add more DRAM

Eliminating Cache Misses

Compulsory ("cold start"): first access to a block

- Insignificant in any long-lived program

Technically, no conflict misses b/c fully associative

Conflict: memory locations map to same cache location

Coherence (invalidation): memory updated externally

- Example: multiple cores, I/O

Policy Misses

What if our replacement policy always evicted the next block the program would access?

- ~100% miss rate

Obviously, choose a better replacement policy. But how much better?

Page Replacement Ideal

Ideal policy [Belady's] **MIN**:

- Evict the page whose next accessed is furthest in the future

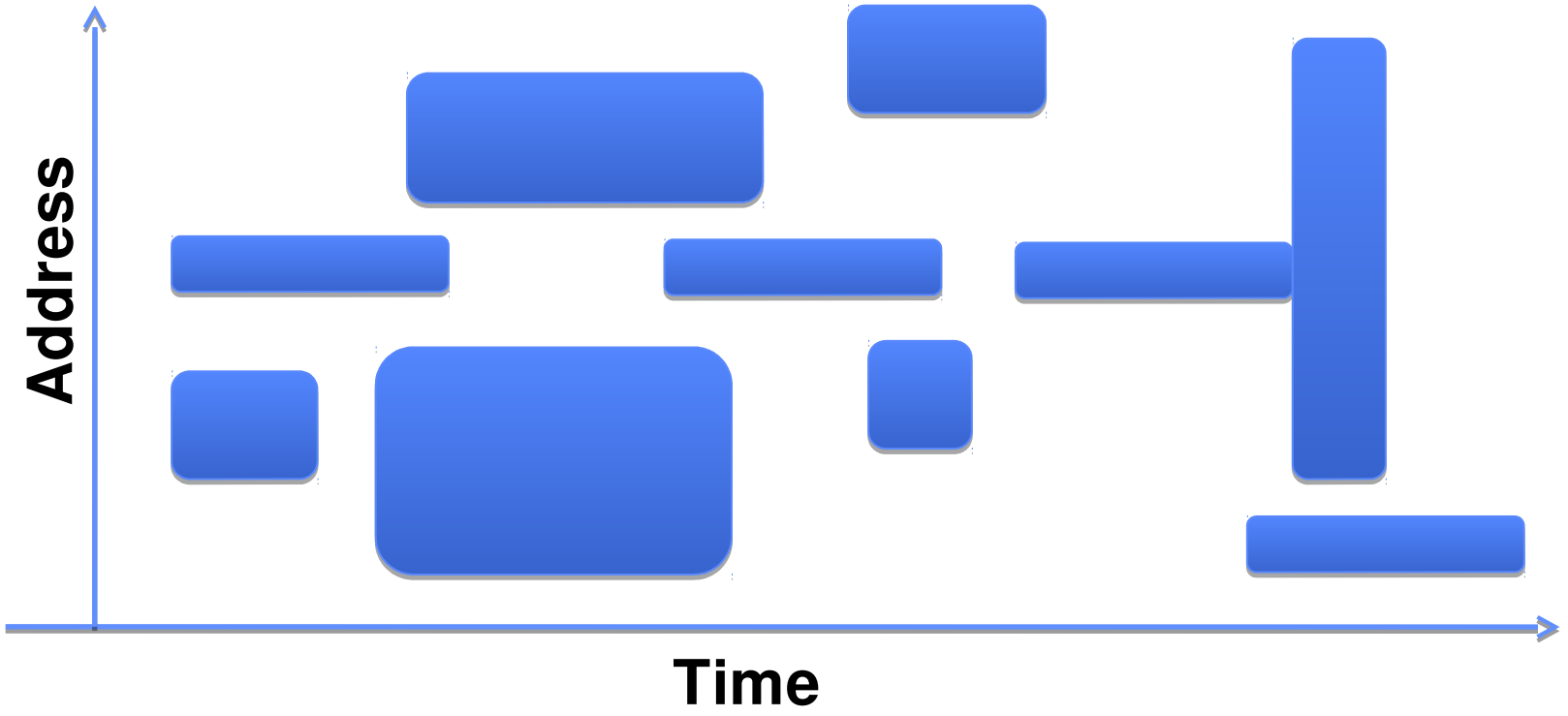
Optimizes for **minimum miss rate**

- N.B. does not deal with fairness, etc.

Problem: ***the future***

Recall: The Working Set Model

Theory: Programs transition through sequence of "working sets" – subsets of their address spaces



Simple Page Replacement Policies

Random

FIFO (First In/First Out)

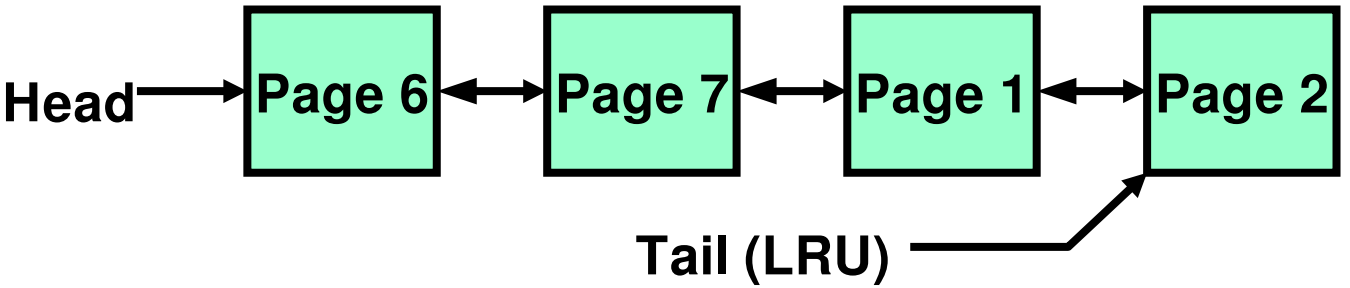
- Throw out oldest page
- **Bad:** heavily used pages more likely to be old

LRU (Least Recently Used)

- Throw out page used longest ago
- Takes advantage of temporal locality

Implementing Exact LRU

Linked list:



Problem: Update list on each use

How expensive is this?

Example: FIFO

3 page frames, 4 virtual pages:

Reference pattern: A B C A B D A D B C B

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 replacements.

When referencing D, replacing A is bad choice, since need A again right away

Example: MIN

3 page frames, 4 virtual pages, same access pattern:

Reference pattern: A B C A B D A D B C B

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

MIN: 5 replacements

LRU: same in this case – not always

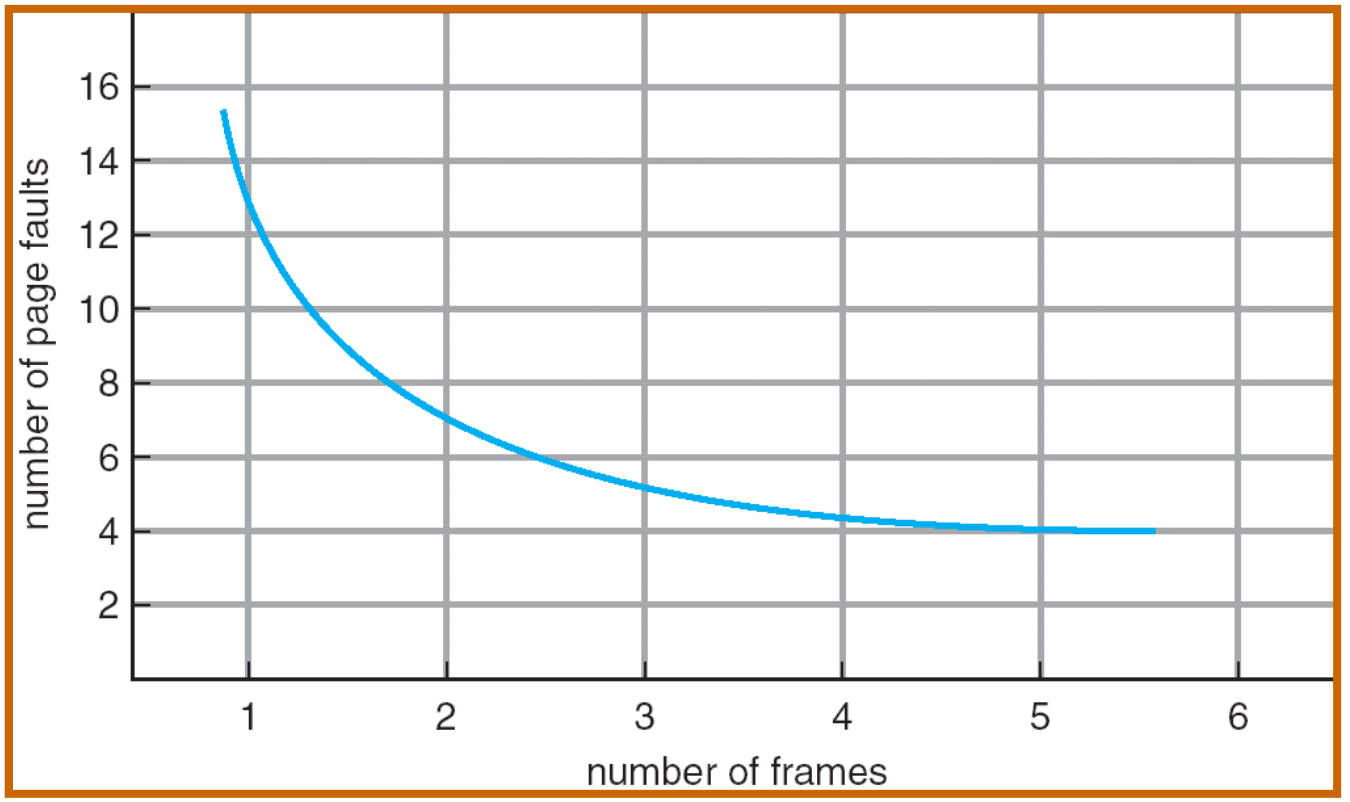
LRU worst cases

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

versus same pattern with MIN:

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

Memory versus Fault Rate



Typical pattern

But adding memory can *increase* fault rate with FIFO

Belady's Anomaly

Adding memory *increases miss rate* with FIFO:

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		

Belady's Anomoly

Never happens with LRU/MIN

- Contents with X pages included if X+1 pages

Logistics

Project 1 Final submission next Monday

Next Tuesday: Midterm review session (instead of lecture)

Midterm next week

Break

Implementing LRU (1)

Could

- update linked list on each access
- keep timestamp of last accesses + scan for highest

But **page fault on every memory access**

- Might as well run an emulator

Implementing LRU (2)

Can't do it exactly (with practical overhead)

- Approximation!

Approximation: **second chance**

Approximation: **clock algorithm**

Intuition: **old** not **oldest**

Intuition: **not recently used**

Clock Algorithm

"Used" bit per page

- Hardware support (but see later)

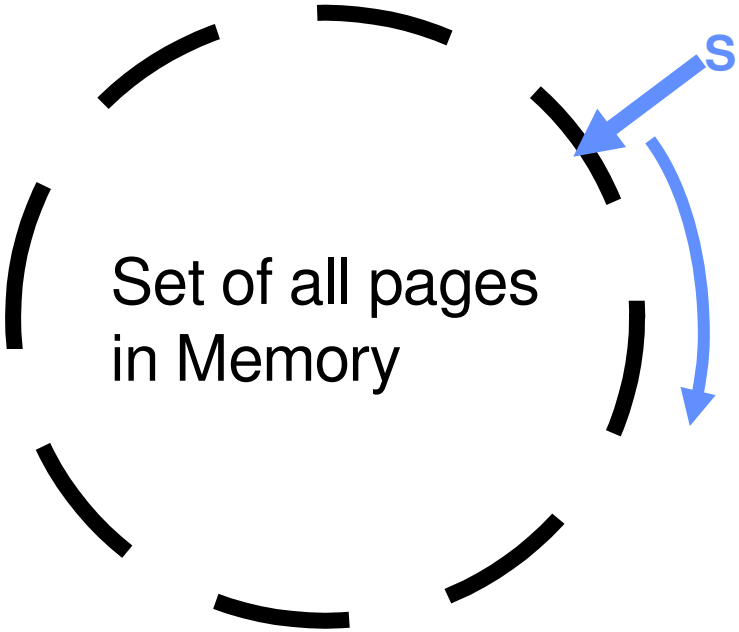
On page fault: scan + reset used bits

- Hand of the "clock"

Replacement candidates are **pages not used since last scan**

No candidate? Revert to FIFO

Clock Algorithm



Single Clock Hand:

Advances only on page fault!
Check for pages not used recently
Mark pages as not used recently

Extra state for each physical page: 1 bit: "used in last cycle"

Basic idea: **Second chance**

- Page wasn't used during last cycle *and* current one

Nth Chance Clock Algorithm

Replace 1 bit "used in last cycle" with a counter "cycles since last use"

- Reset to 0 if used, increment otherwise

Evict if counter > N

Tradeoff:

- Larger N $\rightarrow$ better approximation but more scanning

Nth chance: Dirty pages

What about writeback?

- Expensive – should prefer to keep dirty pages

Allow worse miss rates to avoid writeback

Common approach:

- Give dirty pages a higher # of chances

Hardware support for Clock Alg

Several bits in page table entry, updated by HW:

Used:

- set on reference, cleared by clock algorithm

Modified/Dirty:

- set when page is modified

Do we really need these?

Emulating used and modified bits

Used:

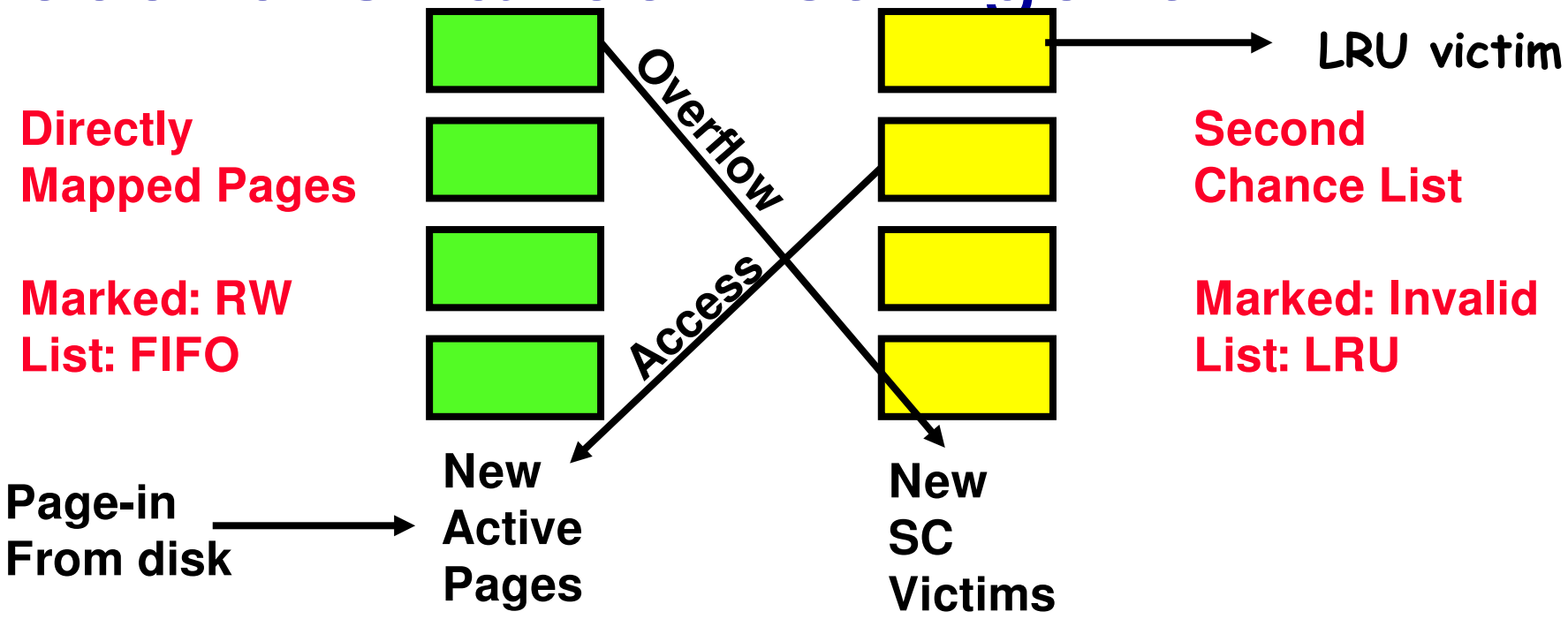
- Mark page as **invalid**
- On page fault, mark page as used
- Then mark page as valid again, restart faulting instruction

Modified/Dirty:

- Mark page as **read-only**
- On *protection fault*, mark page as dirty
- Then mark page as read/write again, restart faulting instruction

No used bit: Second-Chance Lists

Second-Chance List Algorithm



Split memory in two: Active list (RW), SC list (**Invalid**)

Page Fault to access Second Chance List

- Move to front of access list on page fault

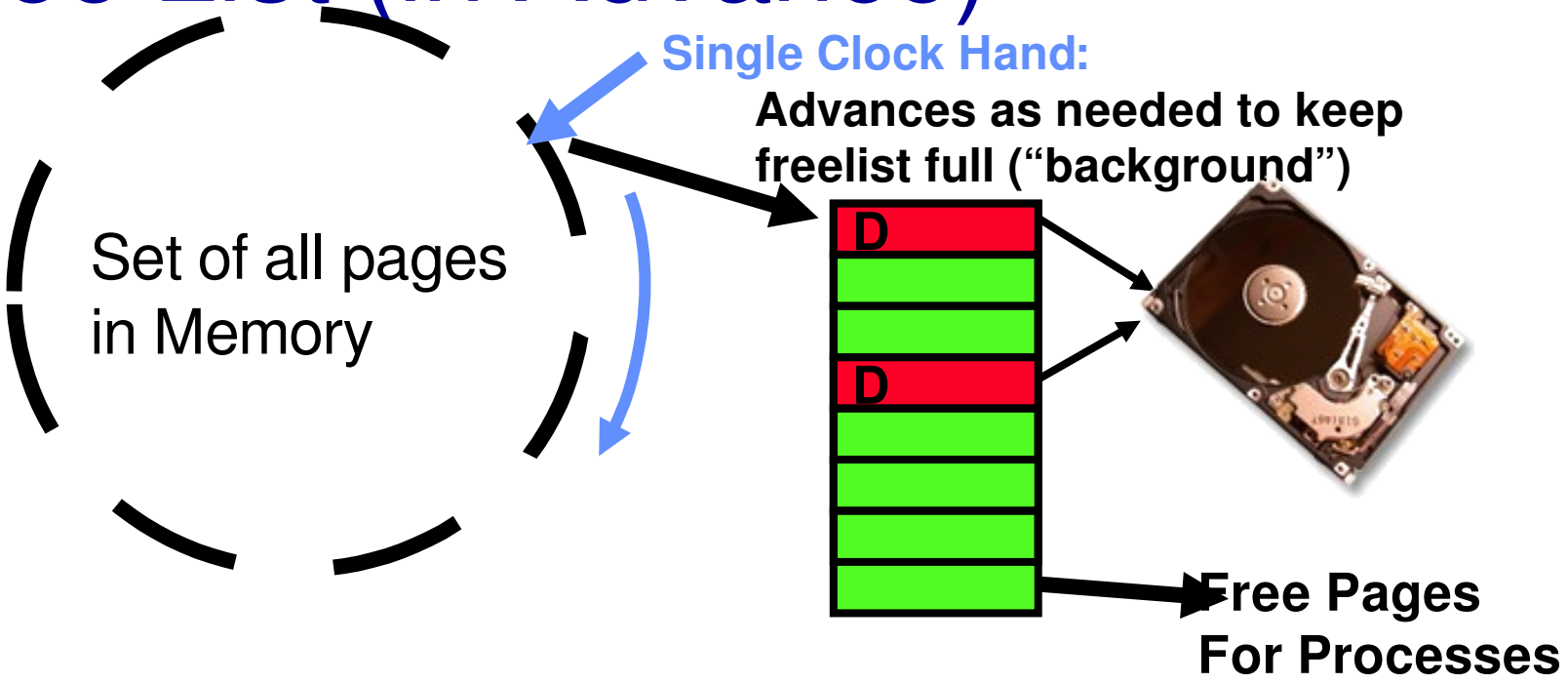
Victims from "front" of second chance list

Second-Chance List Algorithm (con't)

How many pages for second chance list?

- If 0, just FIFO
- If (size of memory), exact LRU (with huge overhead)

Free List (in Advance)



Idea: Avoid scanning pages on page fault

Instead, scan in background, make **free list**

- Write out dirty pages now (instead of stalling page fault)

Page fault: take top of freelist, actually invalidate

Dividing memory among processes

Policy question:

- Fairness
- Starvation

Minimum required to make progress:

- Example: x86 movsw (memory to memory mov word)
  - Instruction itself can span **2 pages**
  - Misaligned source word can span **2 pages**
  - Misaligned destination word can span **2 pages**
- → 2 + 2 + 2 = **6 pages minimums**

Allocation Policies

Equal allocation:

- 100 frames, 5 processes $\rightarrow$ 20 frames/process

Proportional allocation:

- Assign weight to each process
- Allocate weight / (total weight of all processes) to each

Priority allocation:

- Always replace from higher priority process

Allocation Policies

Equal allocation:

- 100 frames, 5 processes → 20 frames/process

Proportional allocation:

- Assign weight to each process
- Allocate weight / (total weight of all processes) to each

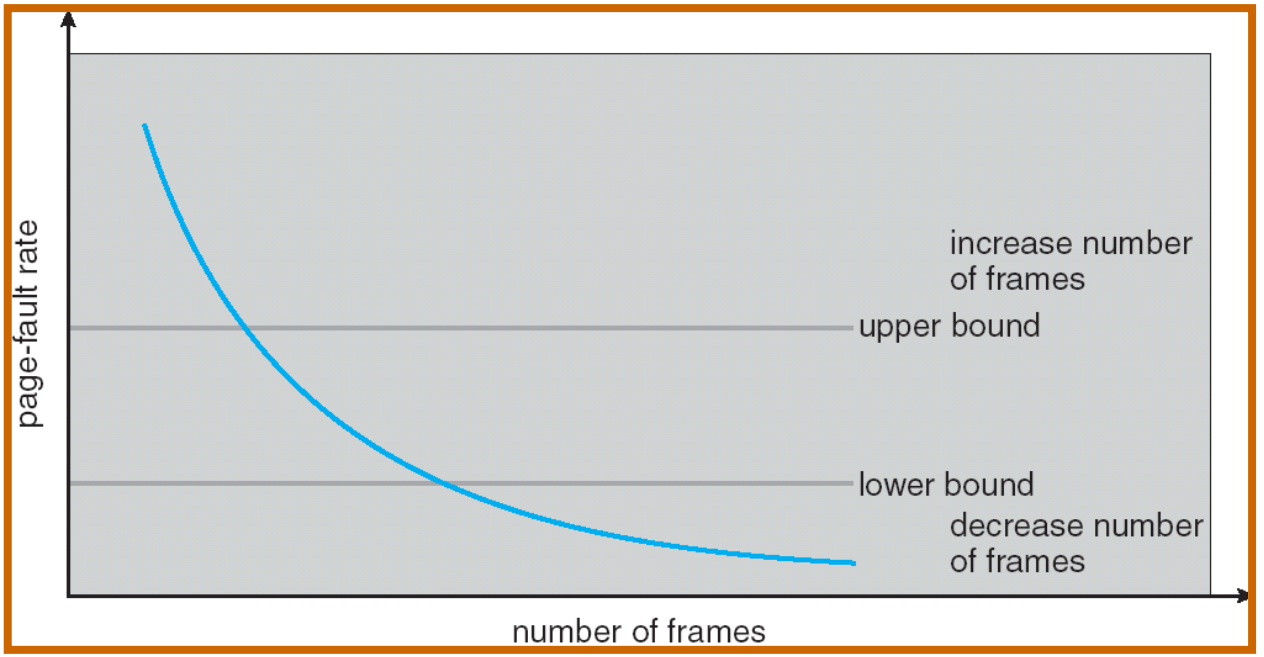
How to implement?

On fault, take from process most over its allocation

On tie, use LRU approximation (probably)

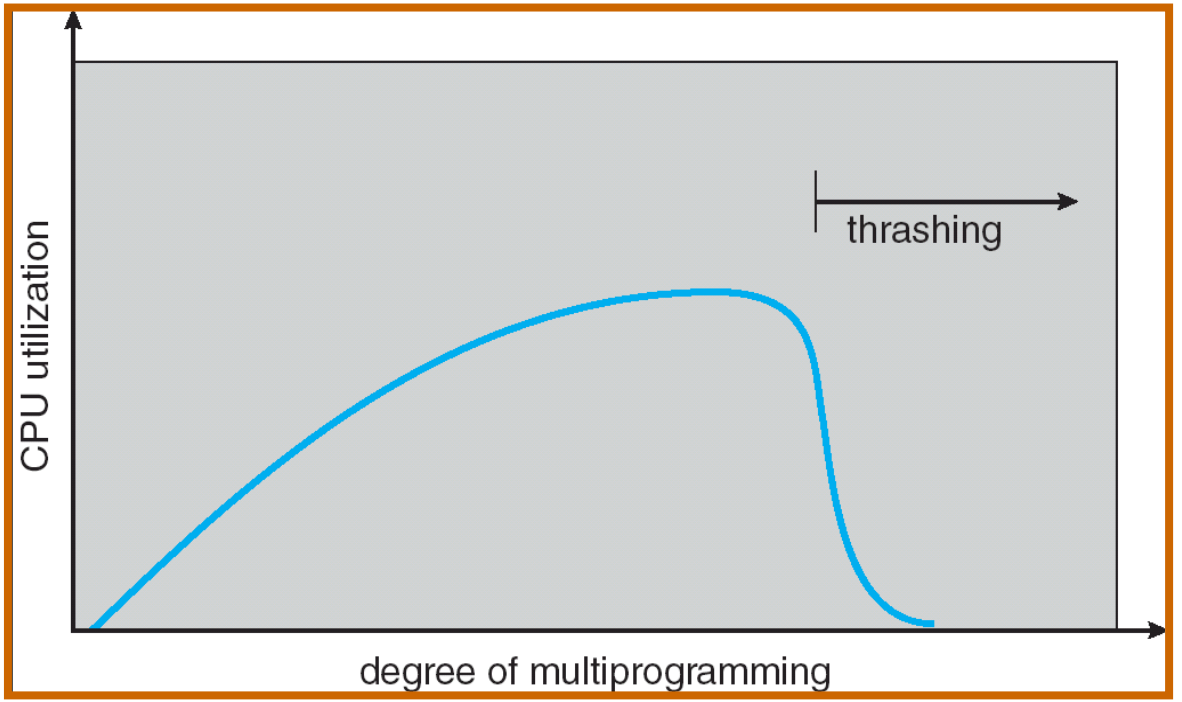
Page-Fault Frequency

Idea: fairness in **amount of swapping**



Evict from process **based on its page fault rate**

Thrashing



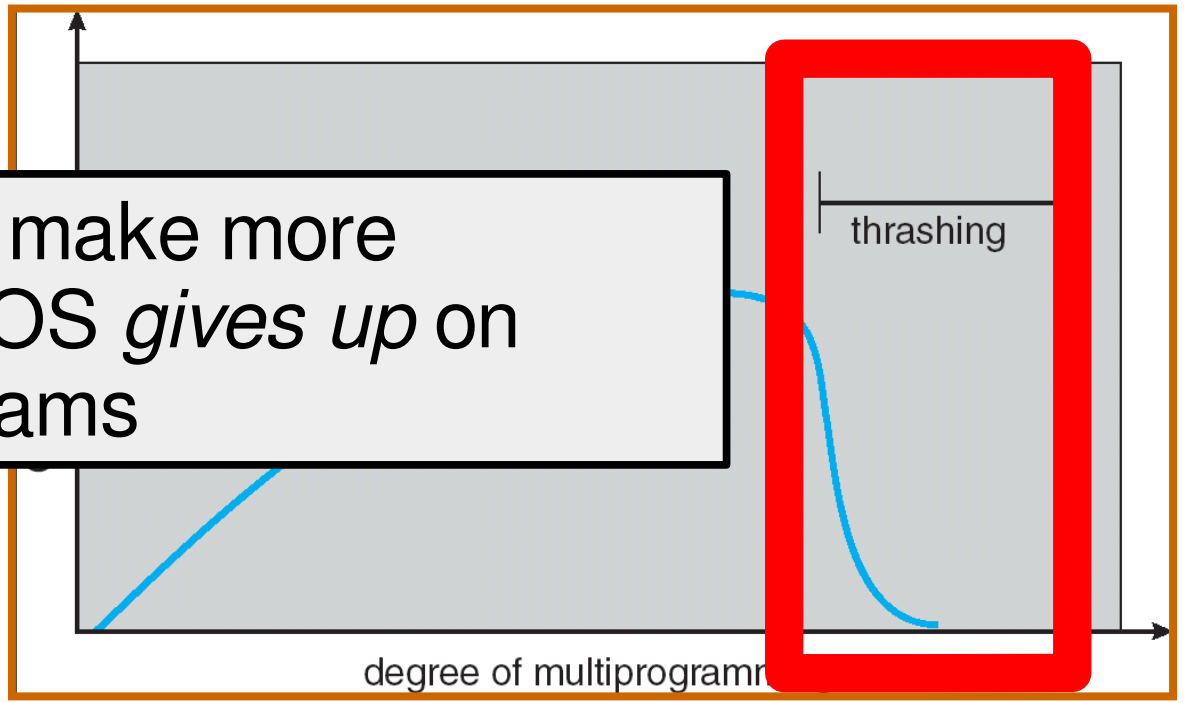
Not enough memory for any process:

- It runs a little and immediately page faults

Result is **low CPU utilization**

- Every process is waiting on IO

Thrashing



System will make more progress if OS *gives up* on some programs

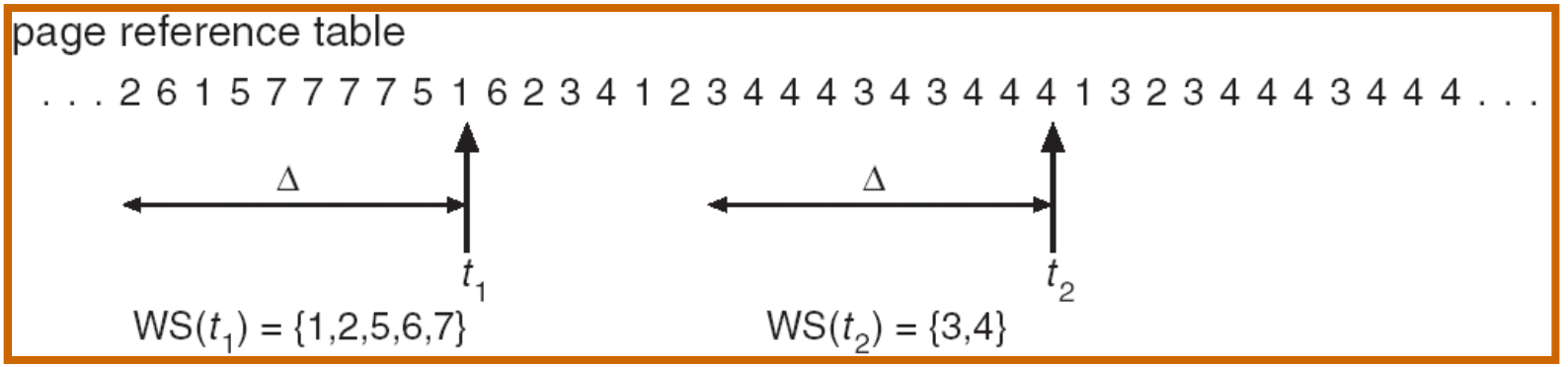
Not enough memory for any process:

- It runs a little and immediately page faults

Result is **low CPU utilization**

- Every process is waiting on IO

Working Set Tracking



Δ = working-set window = fixed number of references or instructions

WS_i (working set of Process P_i) = total set of pages referenced in the most recent Δ (varies in time)

- Δ needs tuned to encompass "current" behavior, not too little/much

If $\Sigma|WS_i|$ = total demand frames > total memory → thrashing

- Policy: **better to suspend a process**

Other Paging Heuristics

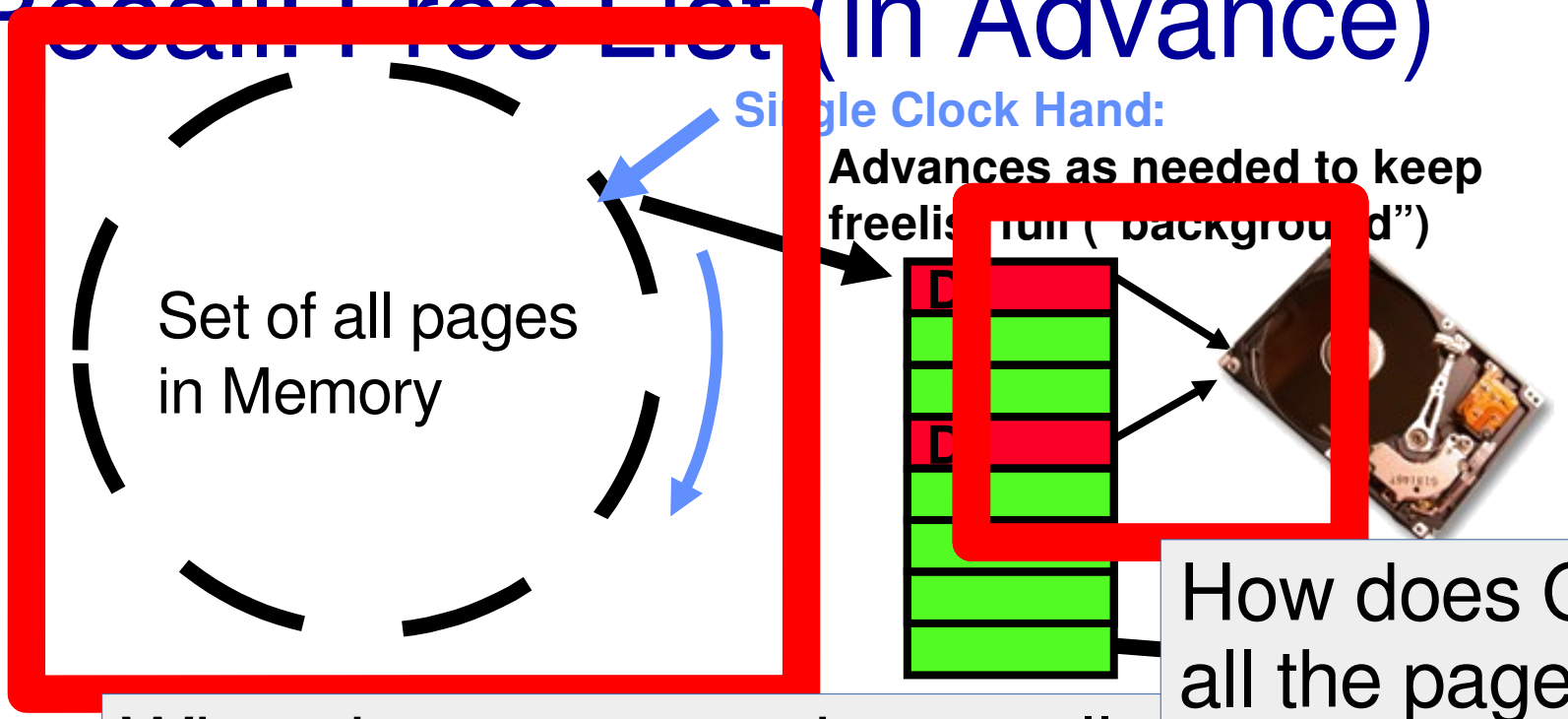
Clustering: Bring in pages "around" faulting page

- One type of prefetching
- Larger reads from disks are more efficient (fixed cost per read of any size)
- Take advantage of spatial locality

Working set tracking:

- Track the "working set" of an application
- When swapping application in, make sure the working set is swapped in

Recall: Free List (in Advance)



How does OS find all the page tables to modify?

What data structure is actually being scanned?

Idea:

Instead, scan in background, make free list

- Write out dirty pages now (instead of stalling page fault)

Page fault: take top of freelist, actually invalidate

Reverse Page Mapping ("coremap")

Mapping from physical page frame # to **all its locations in memory**

Example uses:

- Scanning accessed/dirty bits of **all its PTEs**
- Marking page not present in **all its PTEs** when replacing with another

Linux implementation:

- Linked list of *memory regions* derived from an object (e.g. file)

Summary: Demand Paging

Key idea: **Illusion of infinite memory**

Process's memory lives **on secondary storage**

Memory is **just a cache**

- Block = page
- Fully associative
- Cache replacement = page fault

Summary: Demand Paging Policies

Ideal replacement policy: Belady's MIN

- Impossible to implement, but ideal (access time)
- Similar problems with SRTF

Possible replacement policy: Least Recently Used

- Still impossible to implement – too much overhead to track "uses"

Practical replacement policy: Clock algorithm

- "Not recently used"
- Use HW support of used bit, scan periodically

Practical replacement policy: Second chance list

- "Not recently used"
- Make inactive pages invalid to see if they're really unused