

CSI62

Operating Systems and Systems Programming

Lecture 15

Demand Paging

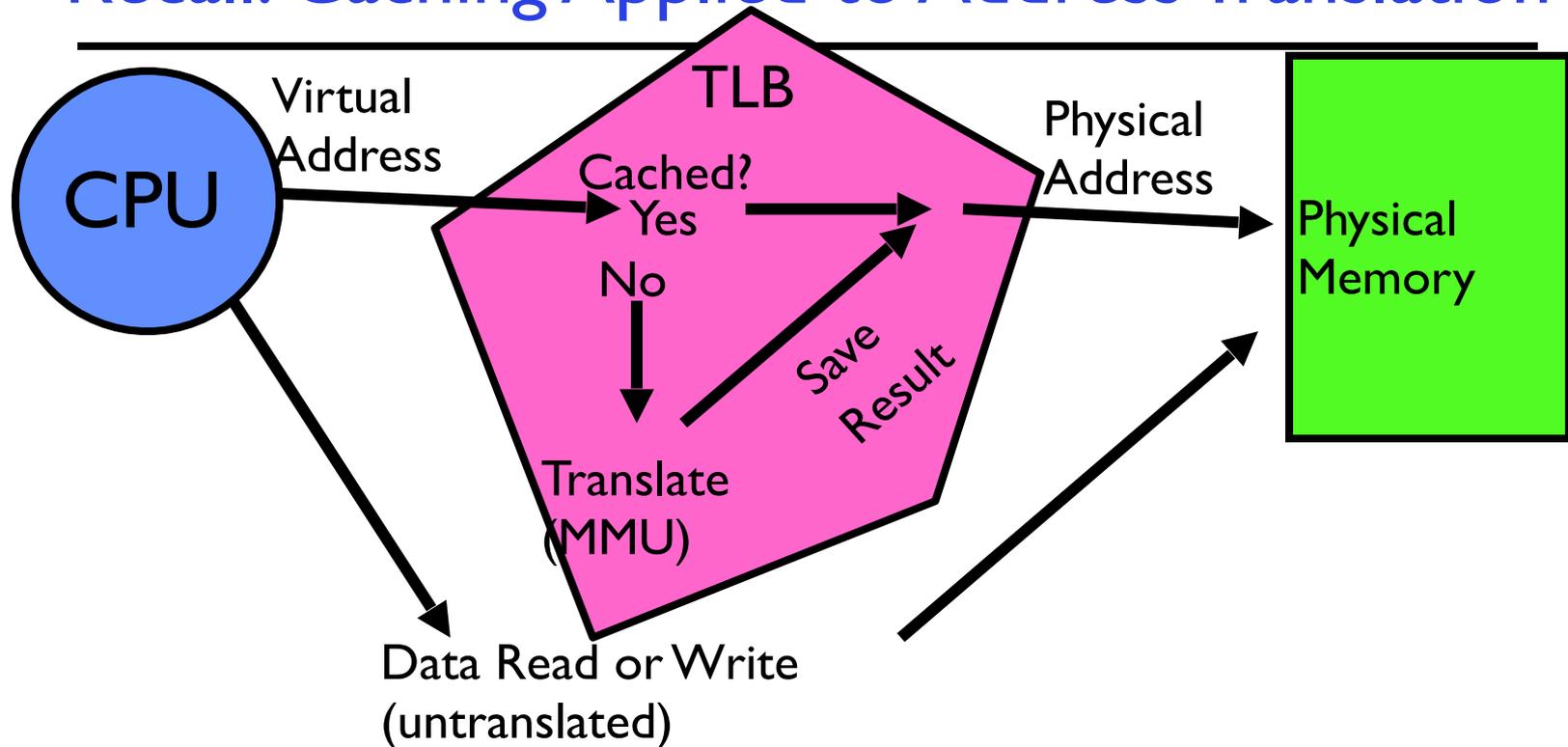
March 17th, 2020

Prof. John Kubiawicz

<http://cs162.eecs.Berkeley.edu>

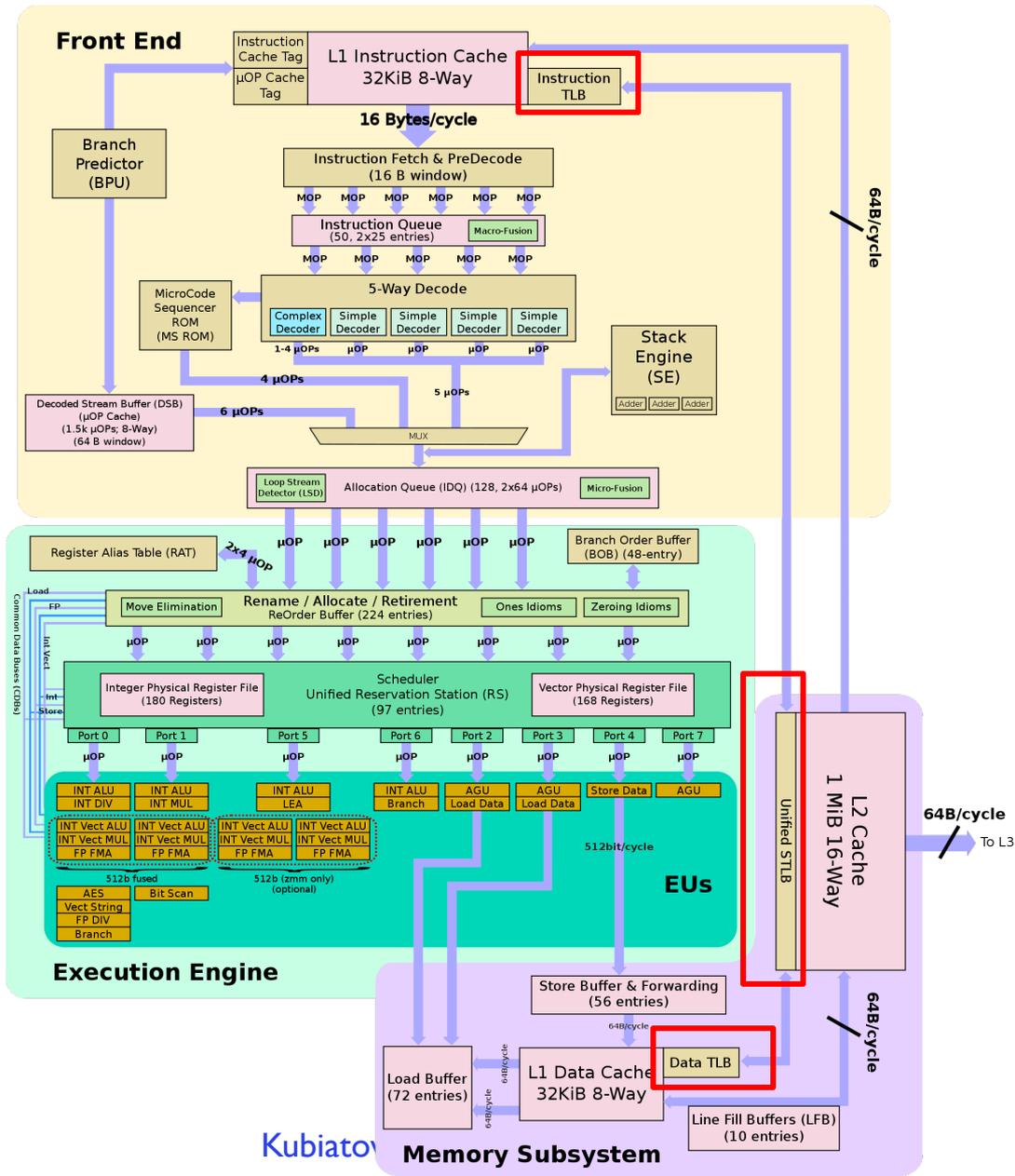
Acknowledgments: Lecture slides are from the Operating Systems course taught by John Kubiawicz at Berkeley, with few minor updates/changes. When slides are obtained from other sources, a reference will be noted on the bottom of that slide, in which case a full list of references is provided on the last slide.

Recall: Caching Applied to Address Translation

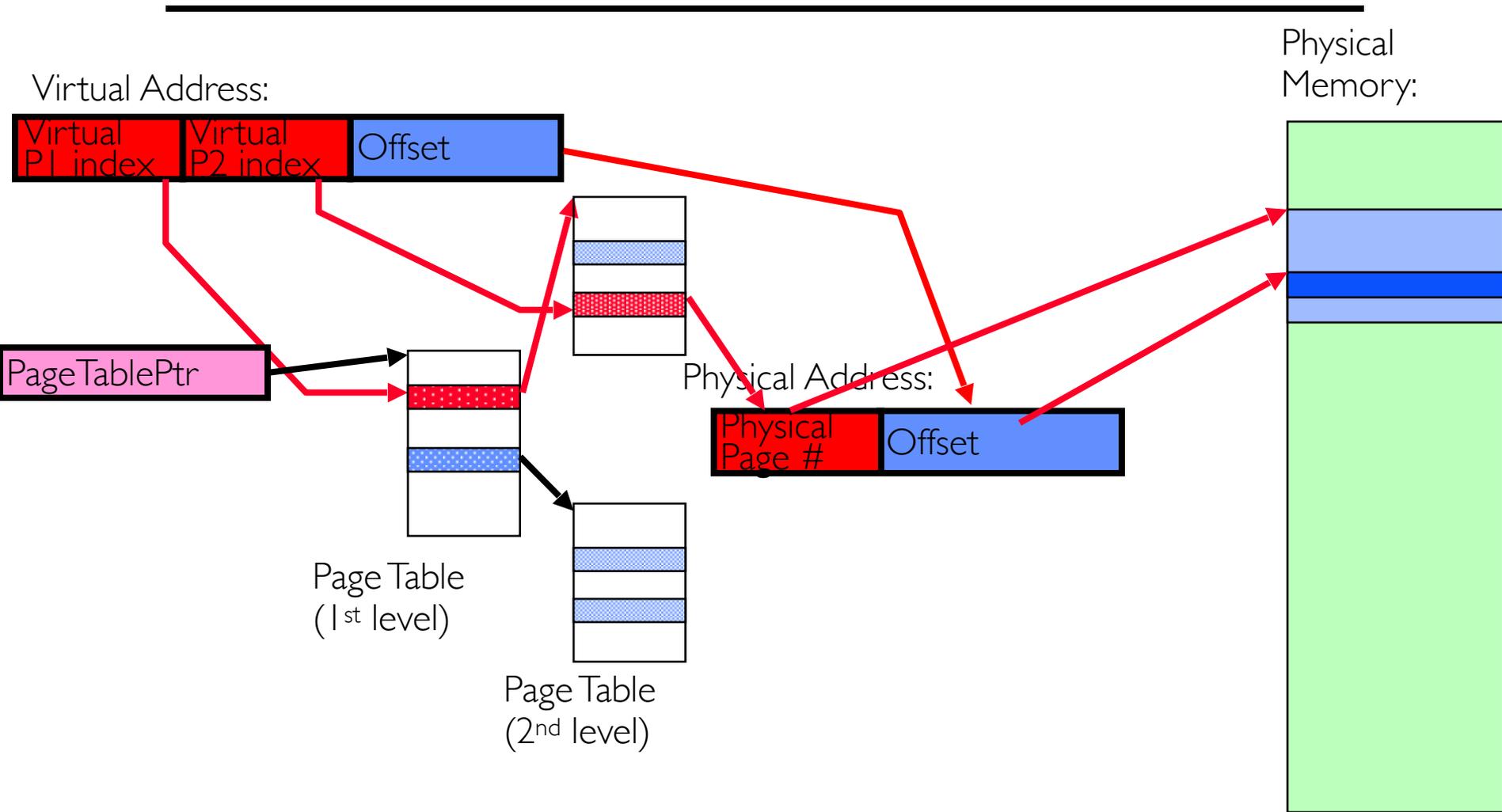


- Question is one of page locality: does it exist?
 - Instruction accesses spend a lot of time on the same page (since accesses sequential)
 - Stack accesses have definite locality of reference
 - Data accesses have less page locality, but still some...
- Can we have a TLB hierarchy?
 - Sure: multiple levels at different sizes/speeds

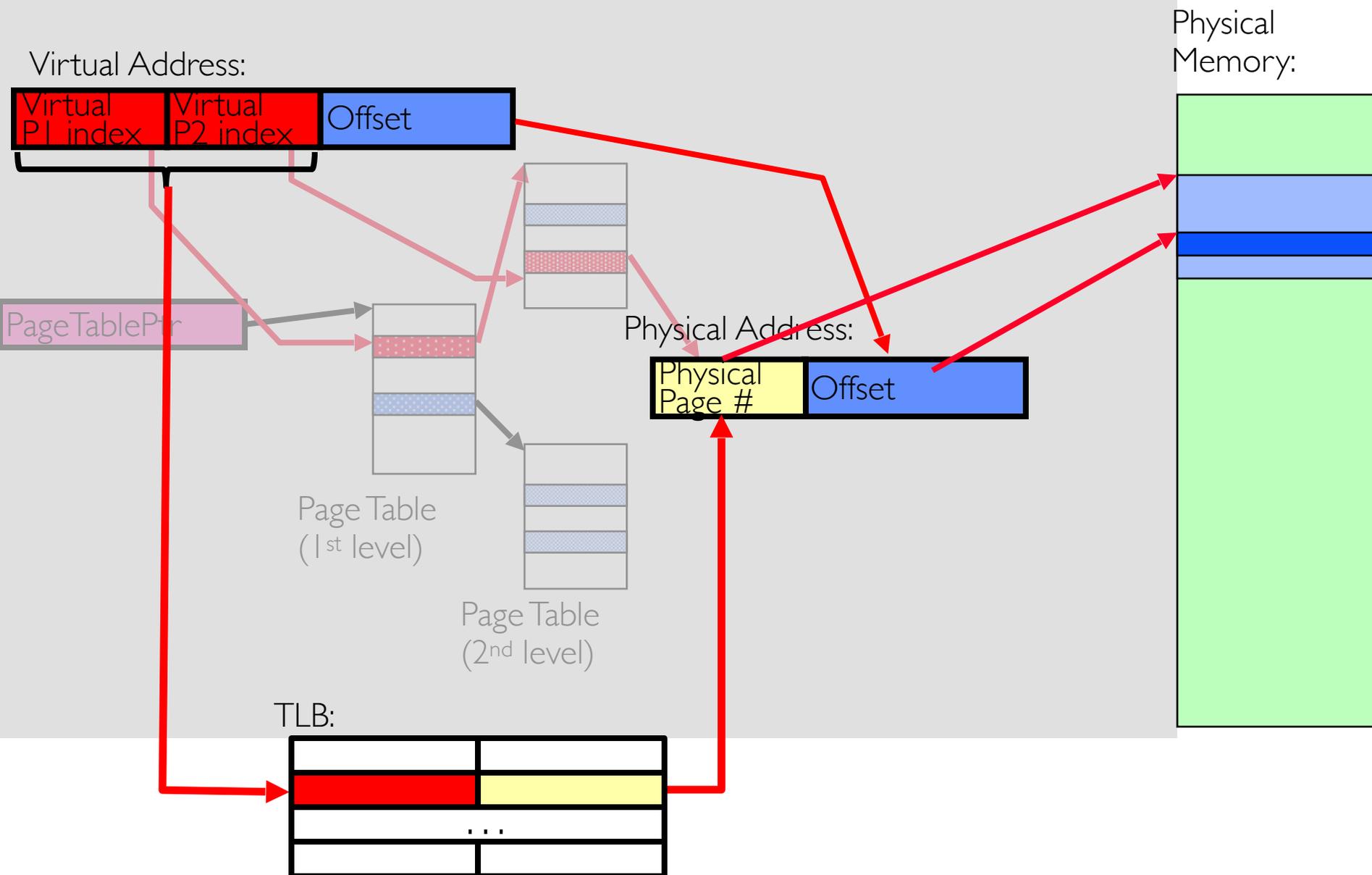
Recall: Current x86 (Skylake, Cascade Lake)



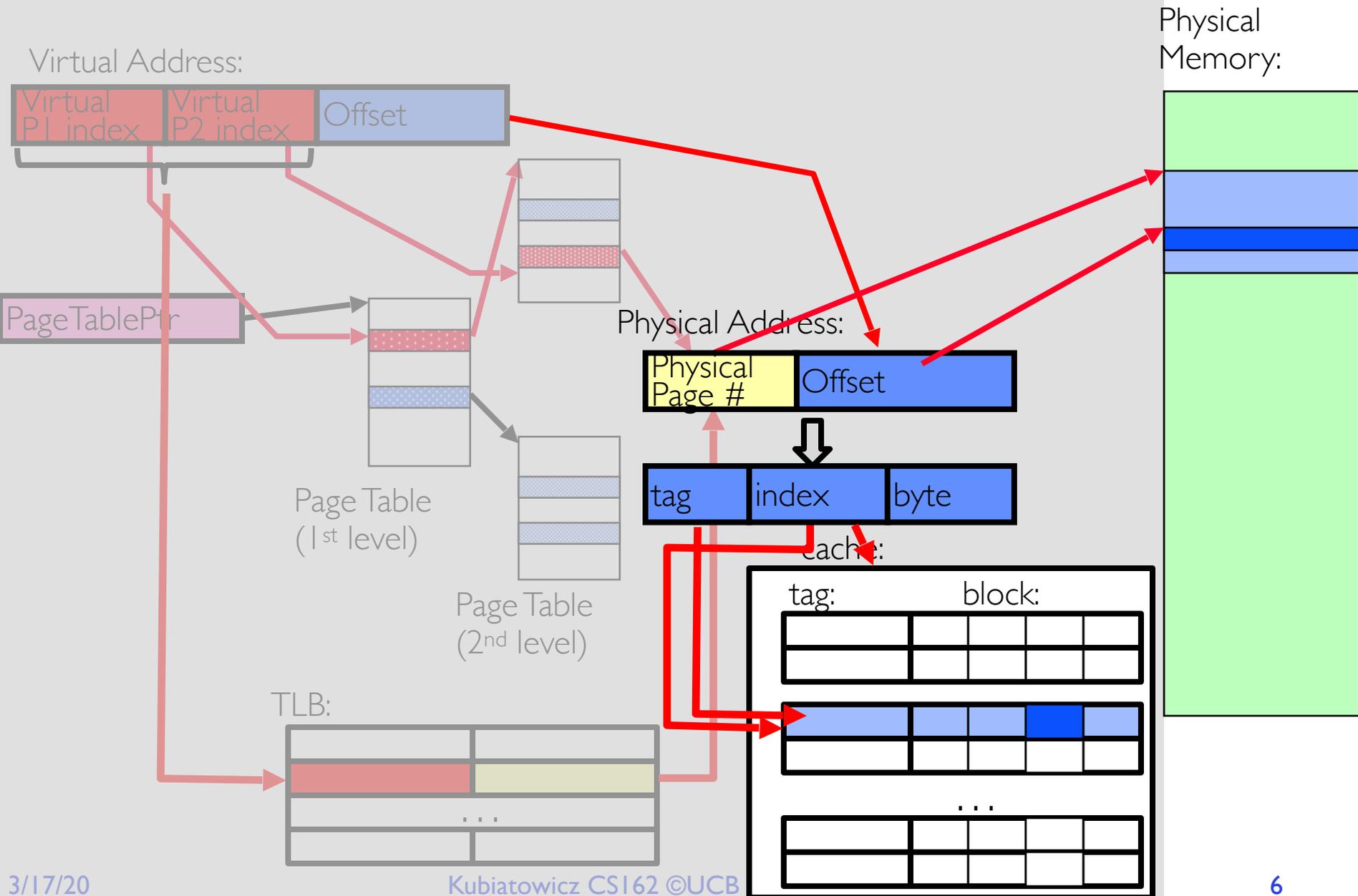
Recall: Putting Everything Together: Address Translation



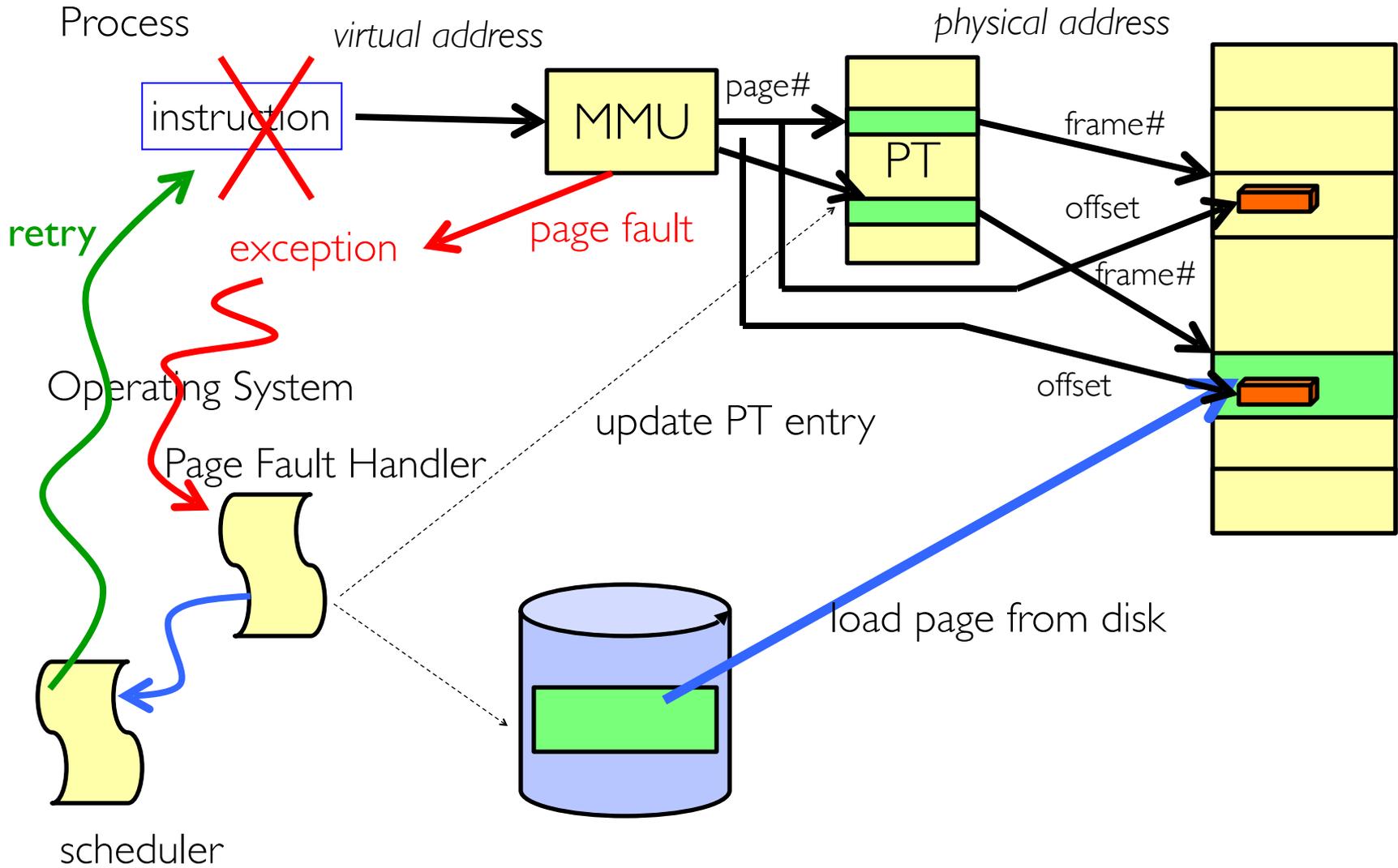
Recall: Putting Everything Together: TLB



Recall: Putting Everything Together: Cache

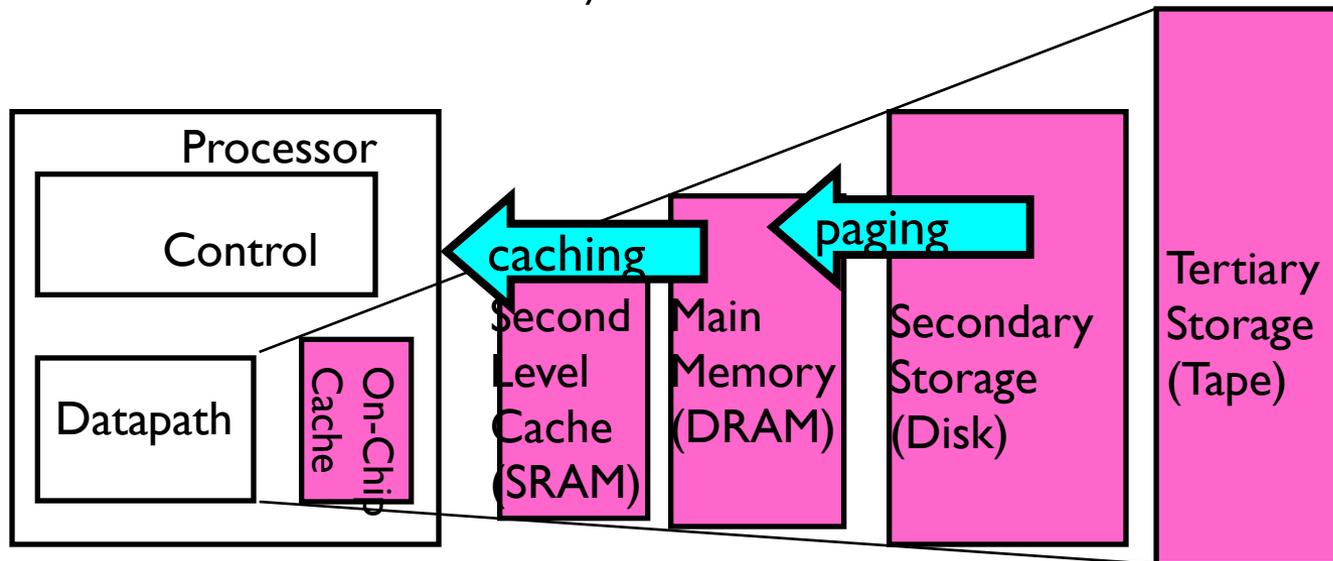


Recall: Page Fault \Rightarrow Demand Paging



Demand Paging

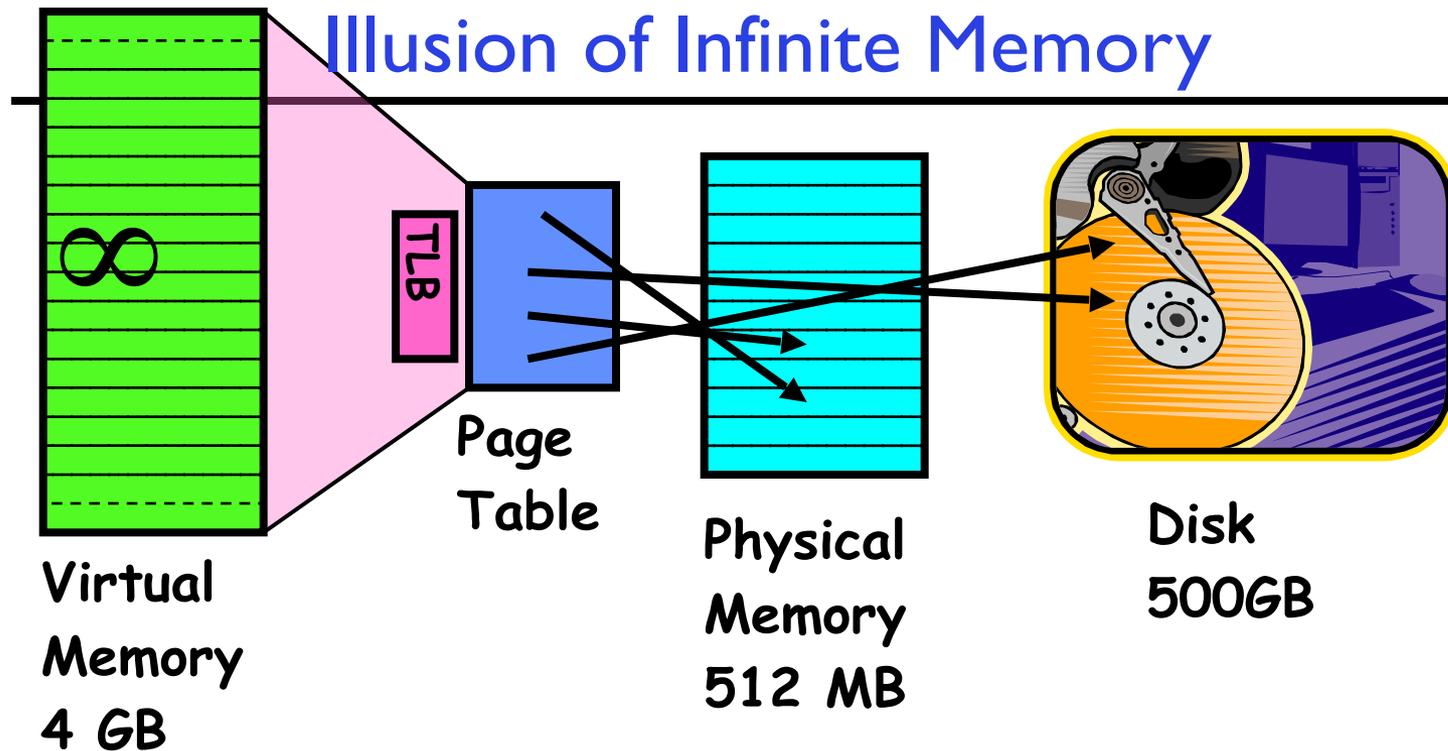
- Modern programs require a lot of physical memory
 - Memory per system growing faster than 25%-30%/year
- But they don't use all their memory all of the time
 - 90-10 rule: programs spend 90% of their time in 10% of their code
 - Wasteful to require all of user's code to be in memory
- Solution: use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



Demand Paging as Caching, ...

- What “block size”? - 1 page (e.g, 4 KB)
- What “organization” ie. direct-mapped, set-assoc., fully-associative?
 - Any page in any frame of memory, i.e., fully associative: arbitrary virtual → physical mapping
- How do we locate a page?
 - 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!

Illusion of Infinite Memory



- Disk is larger than physical memory \Rightarrow
 - 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
 - » Data could be on disk or somewhere across network
 - Variable location of data transparent to user program
 - » Performance issue, not correctness issue

Review: What is in a PTE?

- What is in a Page Table Entry (or PTE)?
 - Pointer to next-level page table or to actual page
 - Permission bits: valid, read-only, read-write, write-only
- Example: Intel x86 architecture PTE:
 - 2-level page tabler (10, 10, 12-bit offset)
 - Intermediate page tables called “Directories”



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

PS: Page Size: PS=1 \Rightarrow 4MB page (directory only).

Bottom 22 bits of virtual address serve as offset

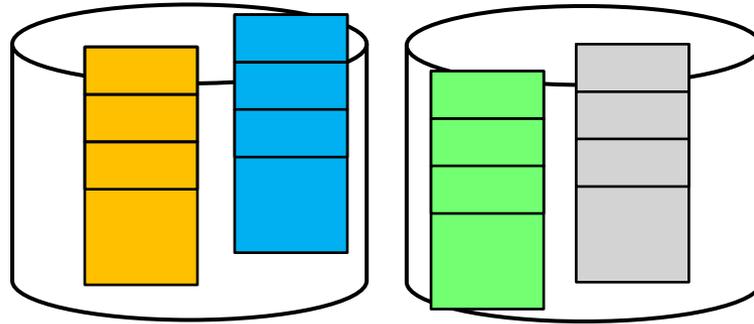
Demand Paging Mechanisms

- PTE makes demand paging implementable
 - Valid \Rightarrow Page in memory, PTE points at physical page
 - Not Valid \Rightarrow Page not in memory; use info in PTE to find it on disk when necessary
- 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=I”), 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 continued!
 - While pulling pages off disk for one process, OS runs another process from ready queue
 - » Suspended process sits on wait queue

Cache

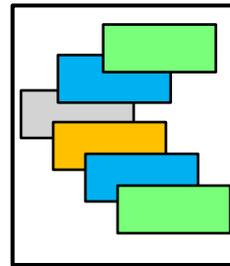
Origins of Paging

Keep most of the address space on disk



Disks provide most of the storage

Actively swap pages to/from



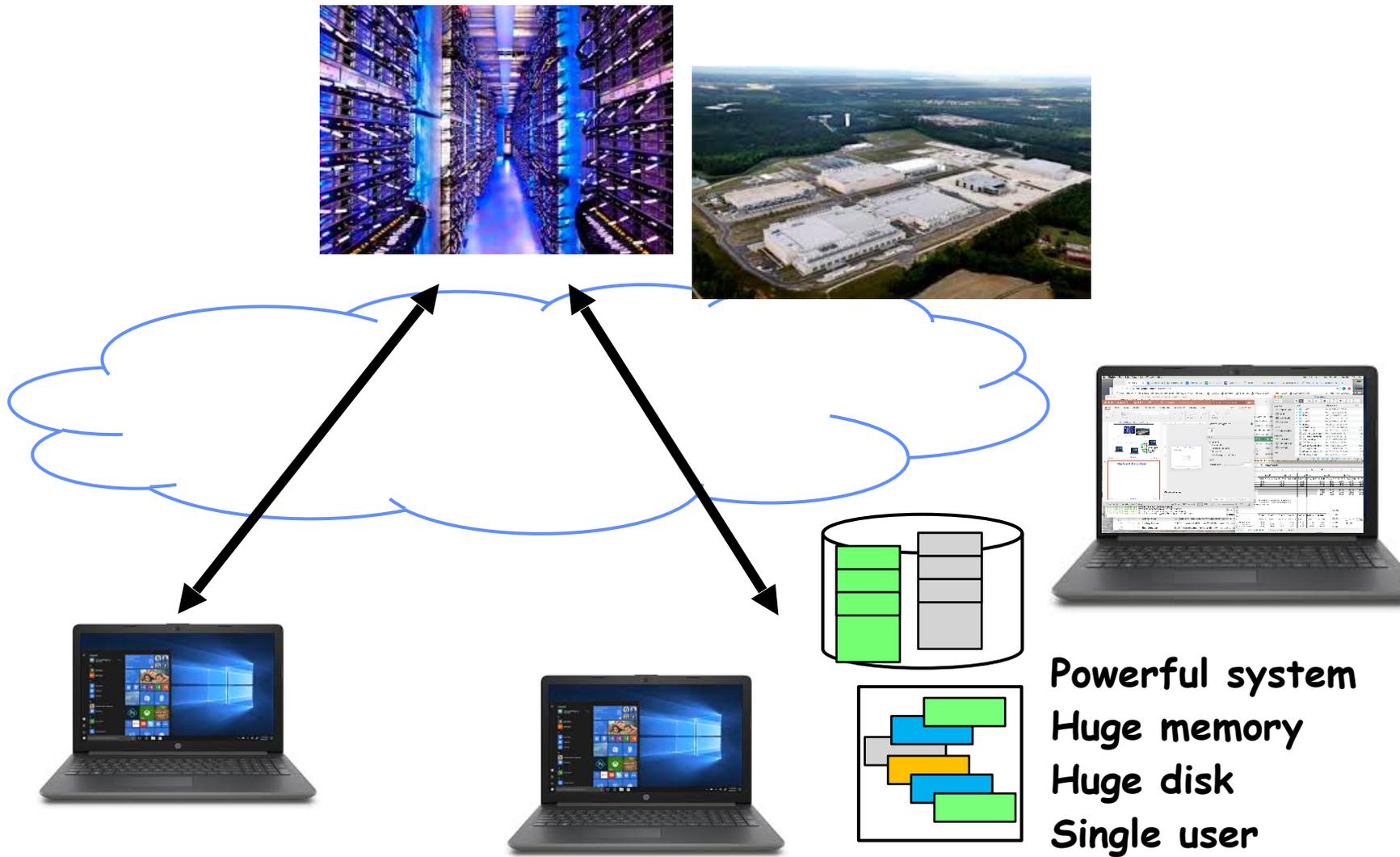
Relatively small memory, for many processes

Keep memory full of the frequently accesses pages



Many clients on dumb terminals running different programs

Very Different Situation Today



A Picture on one machine

```
Processes: 407 total, 2 running, 405 sleeping, 2135 threads                22:10:39
Load Avg: 1.26, 1.26, 0.98 CPU usage: 1.35% user, 1.59% sys, 97.5% idle
SharedLibs: 292M resident, 54M data, 43M linkedit.
MemRegions: 155071 total, 4489M resident, 124M private, 1891M shared.
PhysMem: 13G used (3518M wired), 2718M unused.
VM: 1819G vsize, 1372M framework vsize, 68020510(0) swapins, 71200340(0) swapouts.
Networks: packets: 40629441/21G in, 21395374/7747M out.
Disks: 17026780/555G read, 15757470/638G written.
```

PID	COMMAND	%CPU	TIME	#TH	#WQ	#PORTS	MEM	PURG	CMPRS	PGRP	PPID	STATE
90498	bash	0.0	00:00.41	1	0	21	1080K	0B	564K	90498	90497	sleeping
90497	login	0.0	00:00.10	2	1	31	1236K	0B	1220K	90497	90496	sleeping
90496	Terminal	0.5	01:43.28	6	1	378-	103M-	16M	13M	90496	1	sleeping
89197	siriknowledg	0.0	00:00.83	2	2	45	2664K	0B	1528K	89197	1	sleeping
89193	com.apple.DF	0.0	00:17.34	2	1	68	2688K	0B	1700K	89193	1	sleeping
82655	LookupViewSe	0.0	00:10.75	3	1	169	13M	0B	8064K	82655	1	sleeping
82453	PAH_Extensio	0.0	00:25.89	3	1	235	15M	0B	7996K	82453	1	sleeping
75819	tzlinkd	0.0	00:00.01	2	2	14	452K	0B	444K	75819	1	sleeping
75787	MTLCompilerS	0.0	00:00.10	2	2	24	9032K	0B	9020K	75787	1	sleeping
75776	secd	0.0	00:00.78	2	2	36	3208K	0B	2328K	75776	1	sleeping
75098	DiskUnmountW	0.0	00:00.48	2	2	34	1420K	0B	728K	75098	1	sleeping
75093	MTLCompilerS	0.0	00:00.06	2	2	21	5924K	0B	5912K	75093	1	sleeping
74938	ssh-agent	0.0	00:00.00	1	0	21	908K	0B	892K	74938	1	sleeping
74063	Google Chrom	0.0	10:48.49	15	1	678	192M	0B	51M	54320	54320	sleeping

- Memory stays about 75% used, 25% for dynamics
- A lot of it is shared 1.9 GB

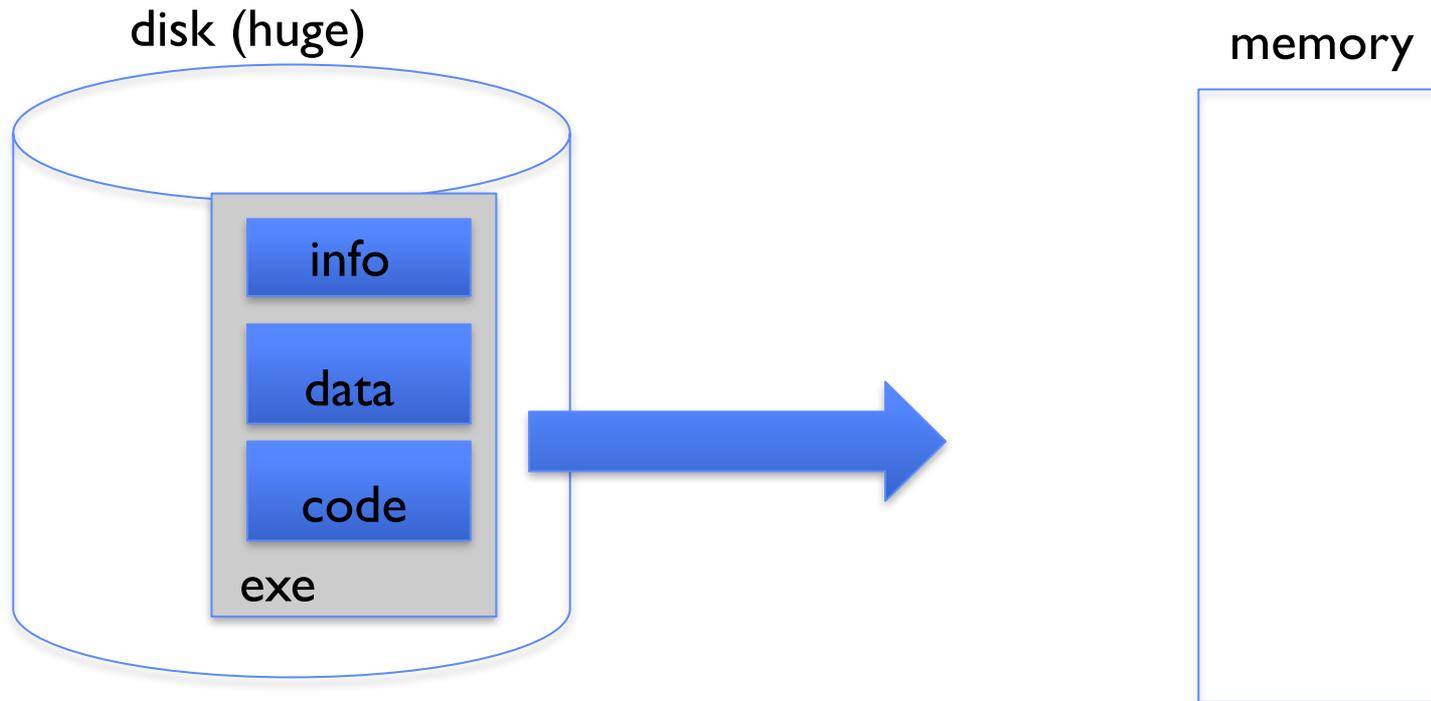
Many Uses of Virtual Memory and “Demand Paging” ...

- 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)

Administrivia

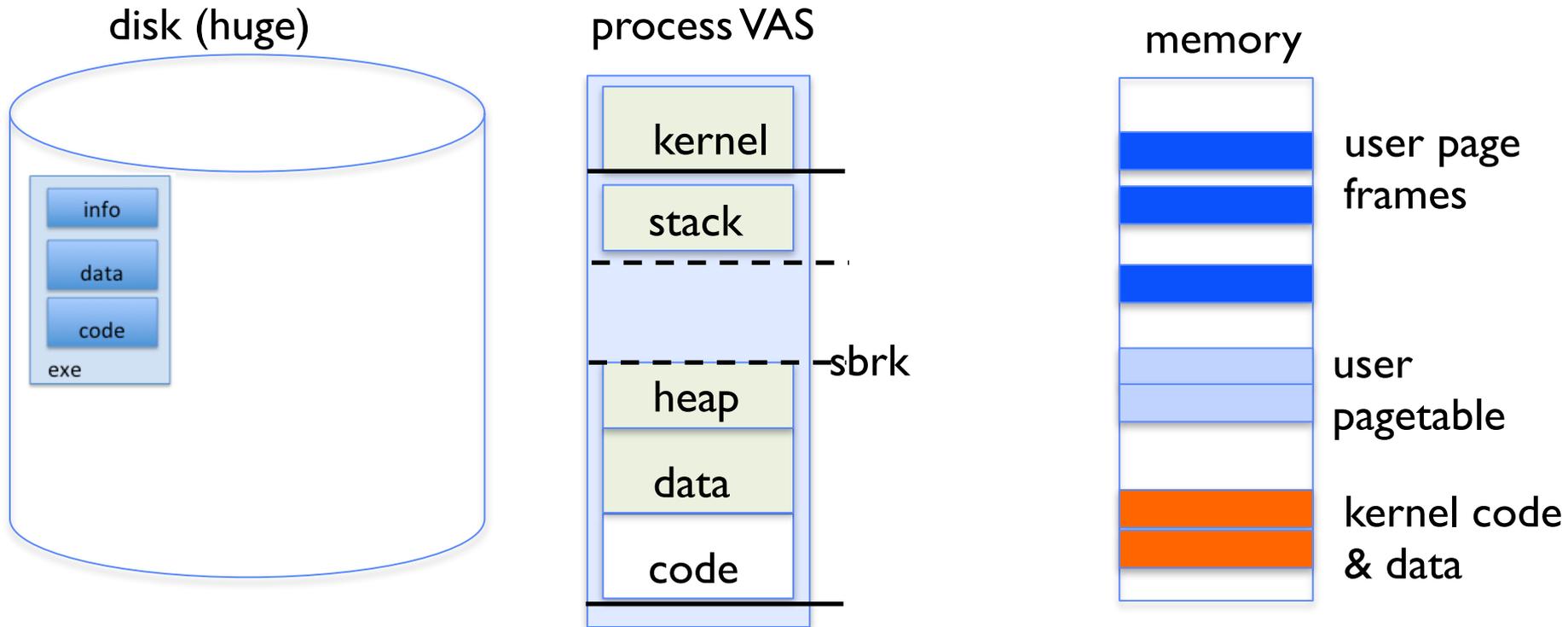
- HW 2 issues?
- GHW2 will be out today
- Any final thoughts on the midterm?

Classic: Loading an executable into memory



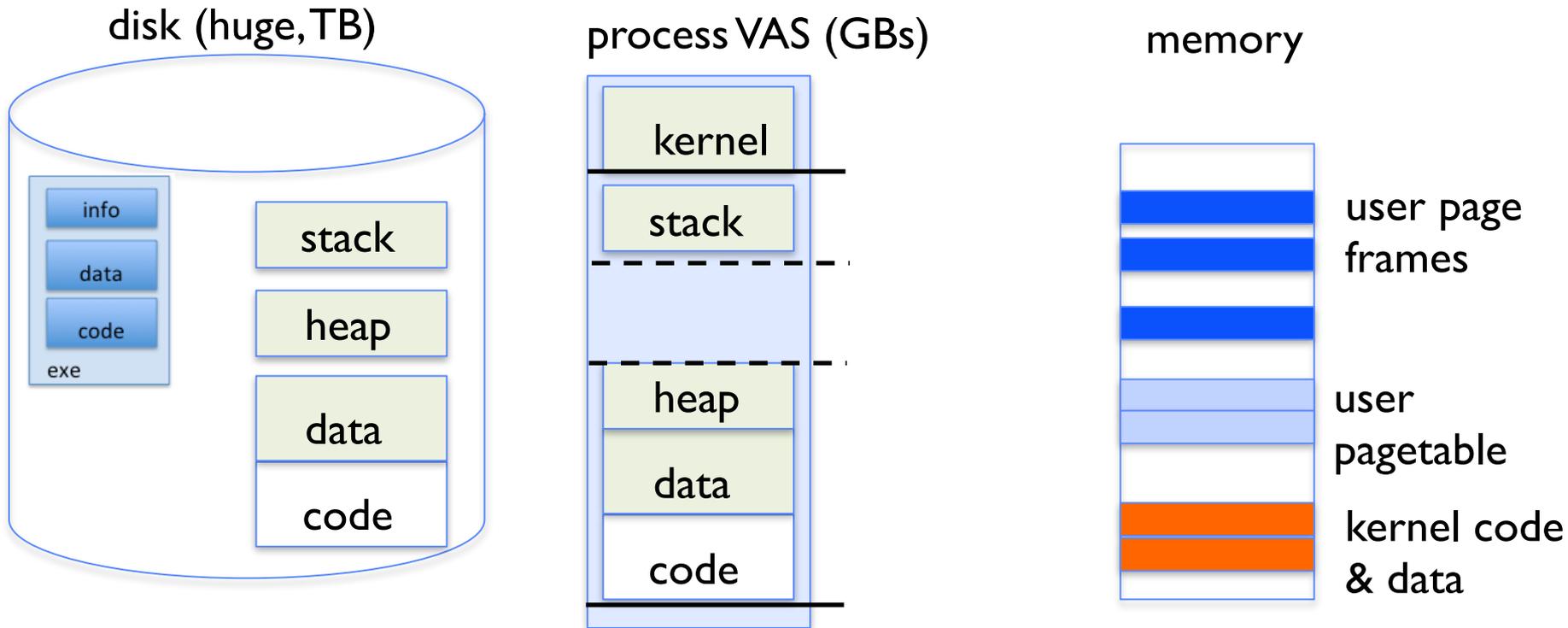
- .exe
 - lives on disk in the file system
 - contains contents of code & data segments, relocation entries and symbols
 - OS loads it into memory, initializes registers (and initial stack pointer)
 - program sets up stack and heap upon initialization:
`crt0` (C runtime init)

Create Virtual Address Space of the Process



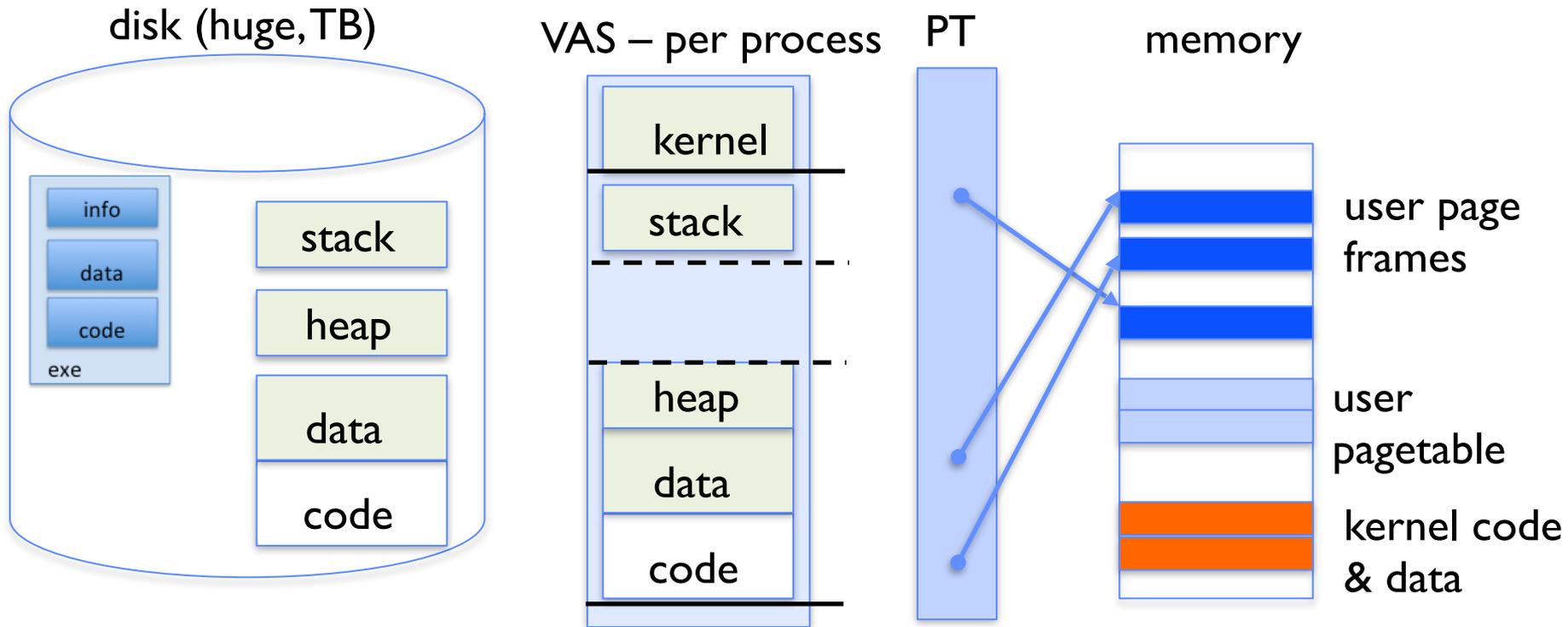
- Utilized pages in the VAS are backed by a page block on disk
 - Called the backing store or swap file
 - Typically in an optimized block store, but can think of it like a file

Create Virtual Address Space of the Process



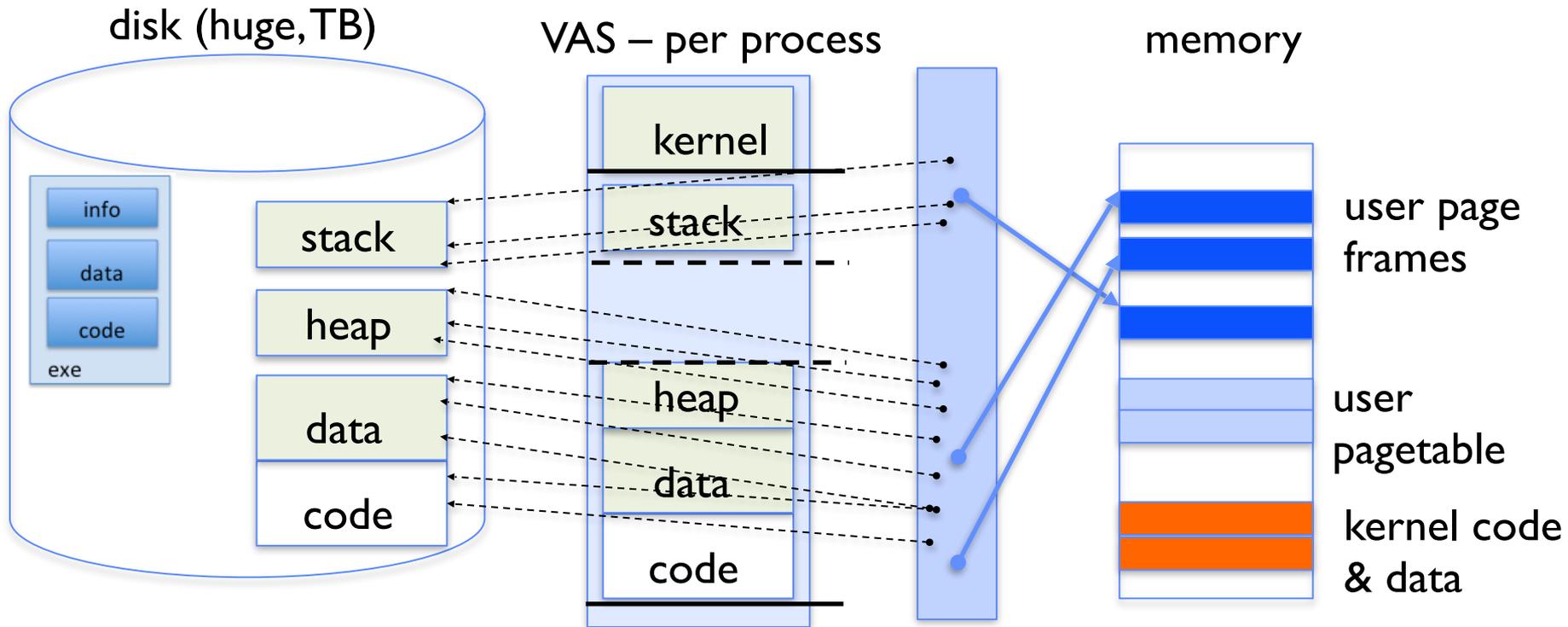
- User Page table maps entire VAS
- All the utilized regions are backed on disk
 - swapped into and out of memory as needed
- For every process

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

Provide Backing Store for VAS

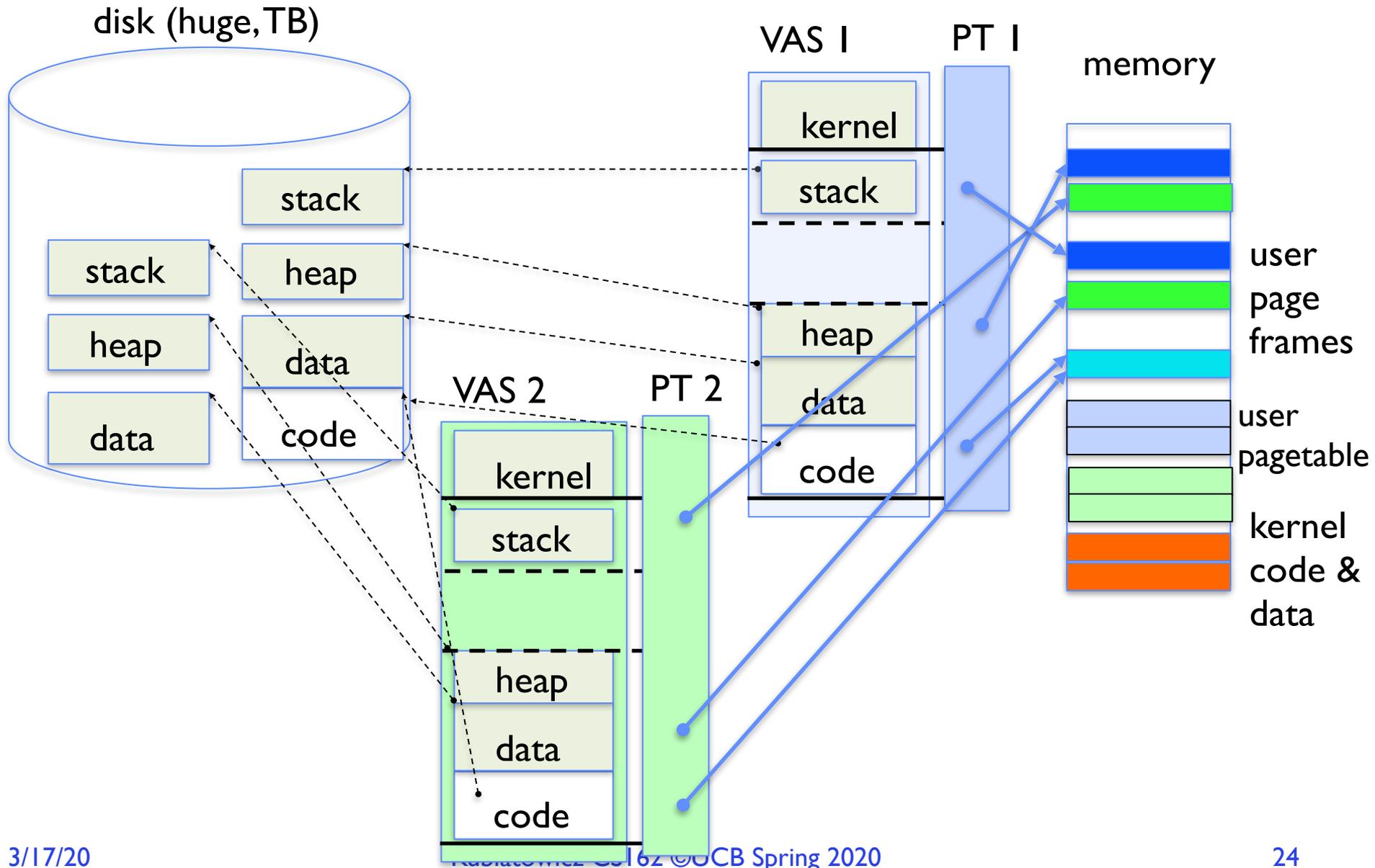


- User Page table maps entire VAS
- Resident pages mapped to memory frames
- For all other pages, OS must record where to find them on disk

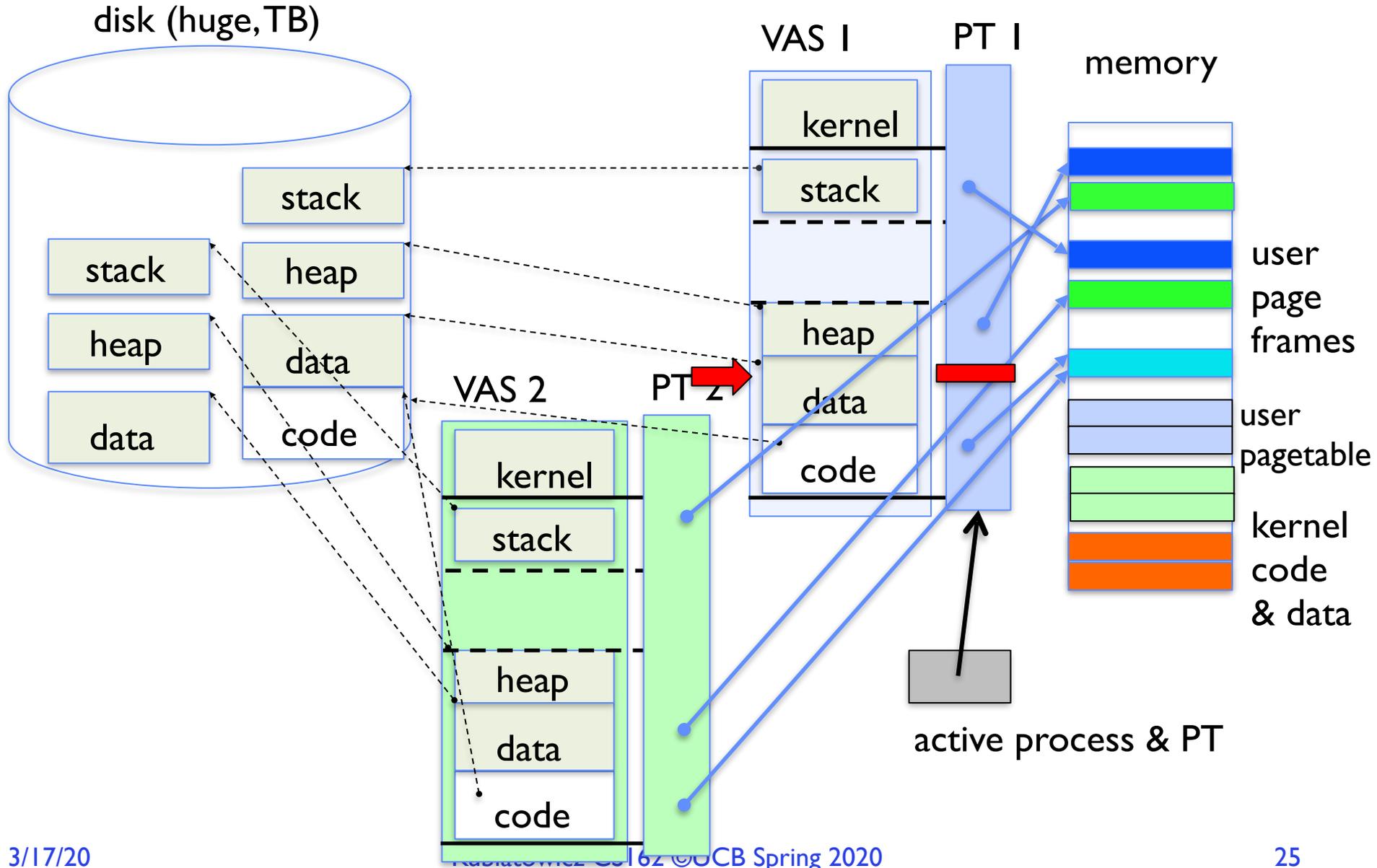
What Data Structure Maps Non-Resident Pages to Disk?

- `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
- 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

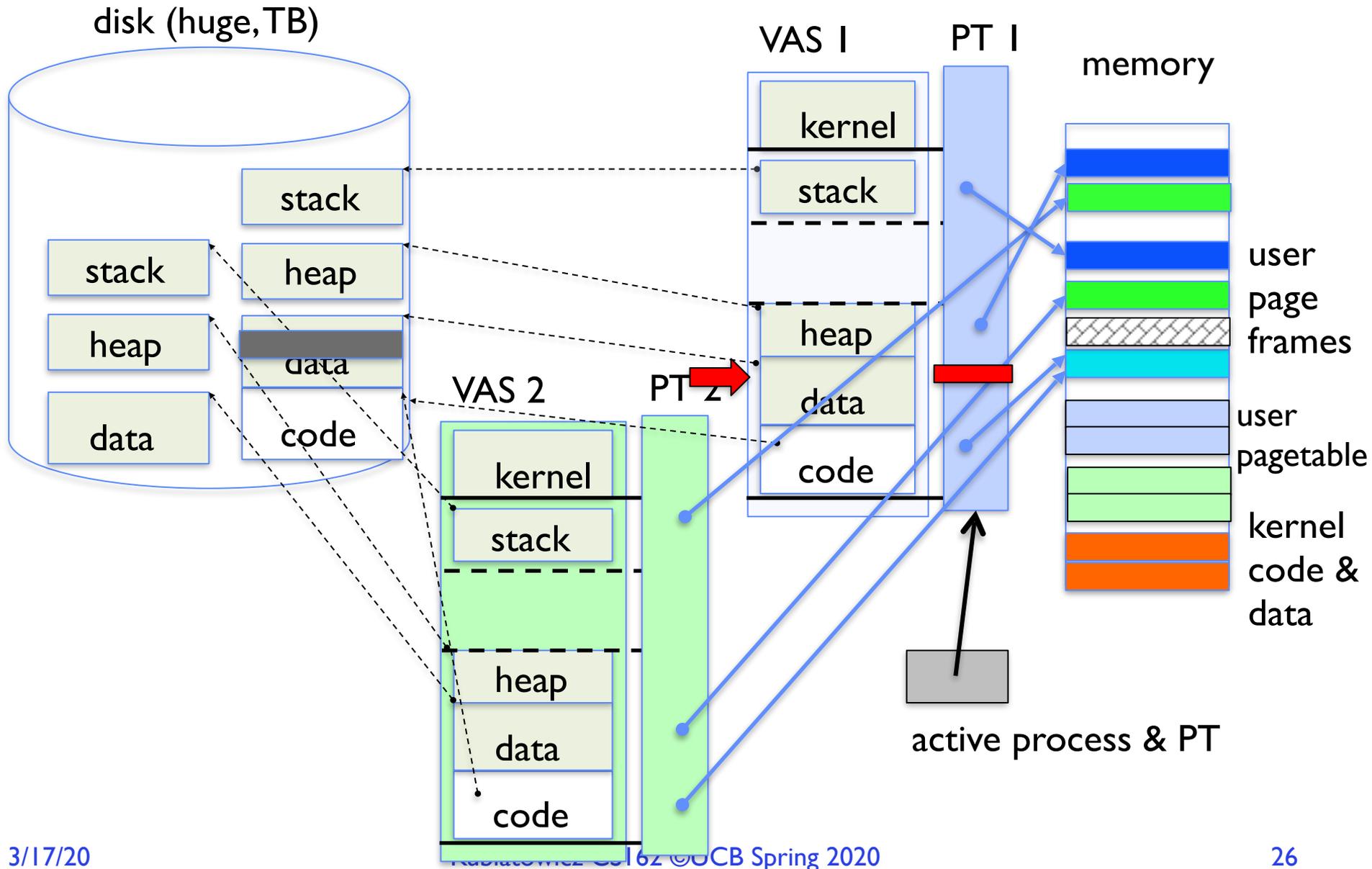
Provide Backing Store for VAS



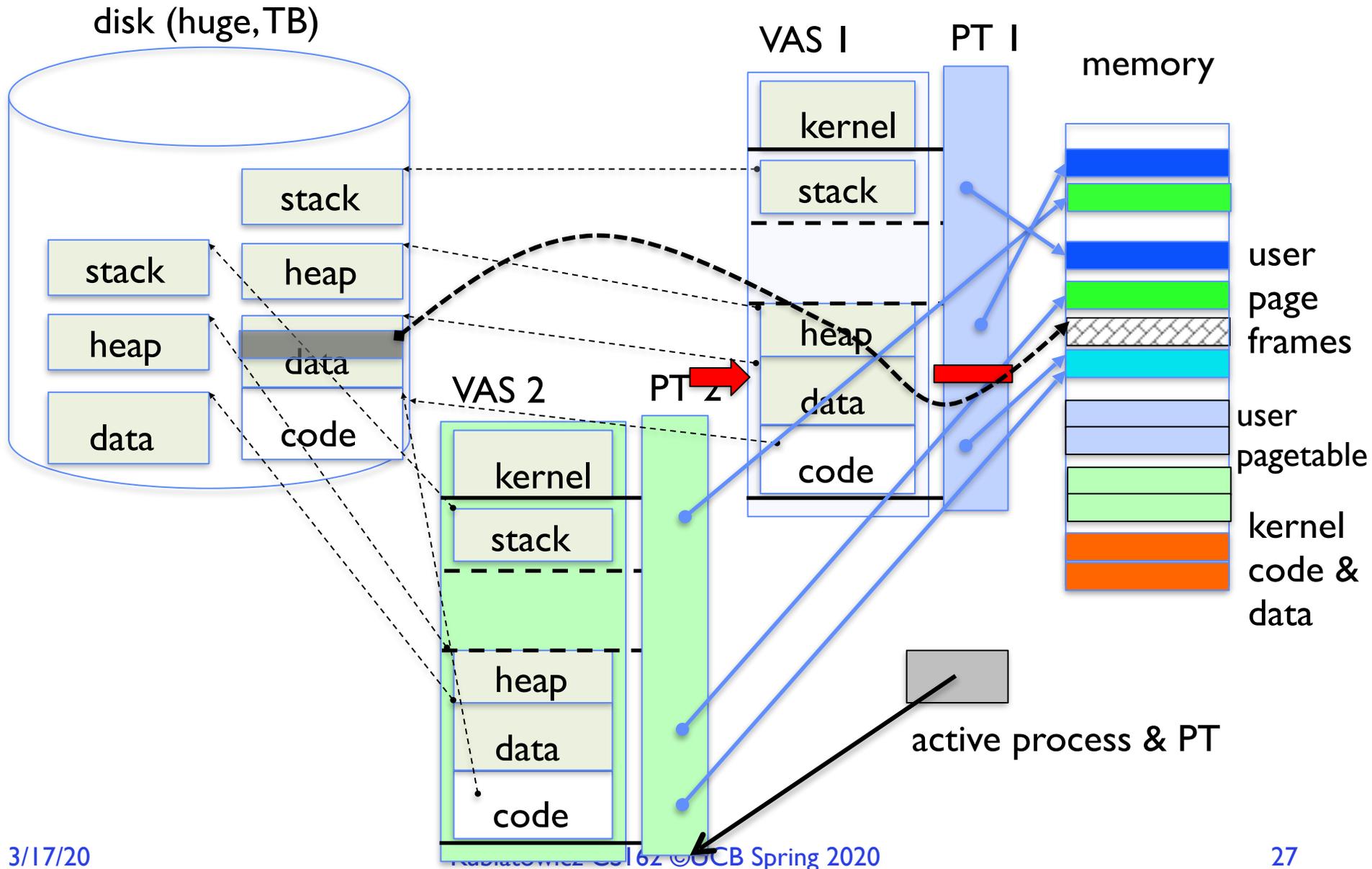
On page Fault ...



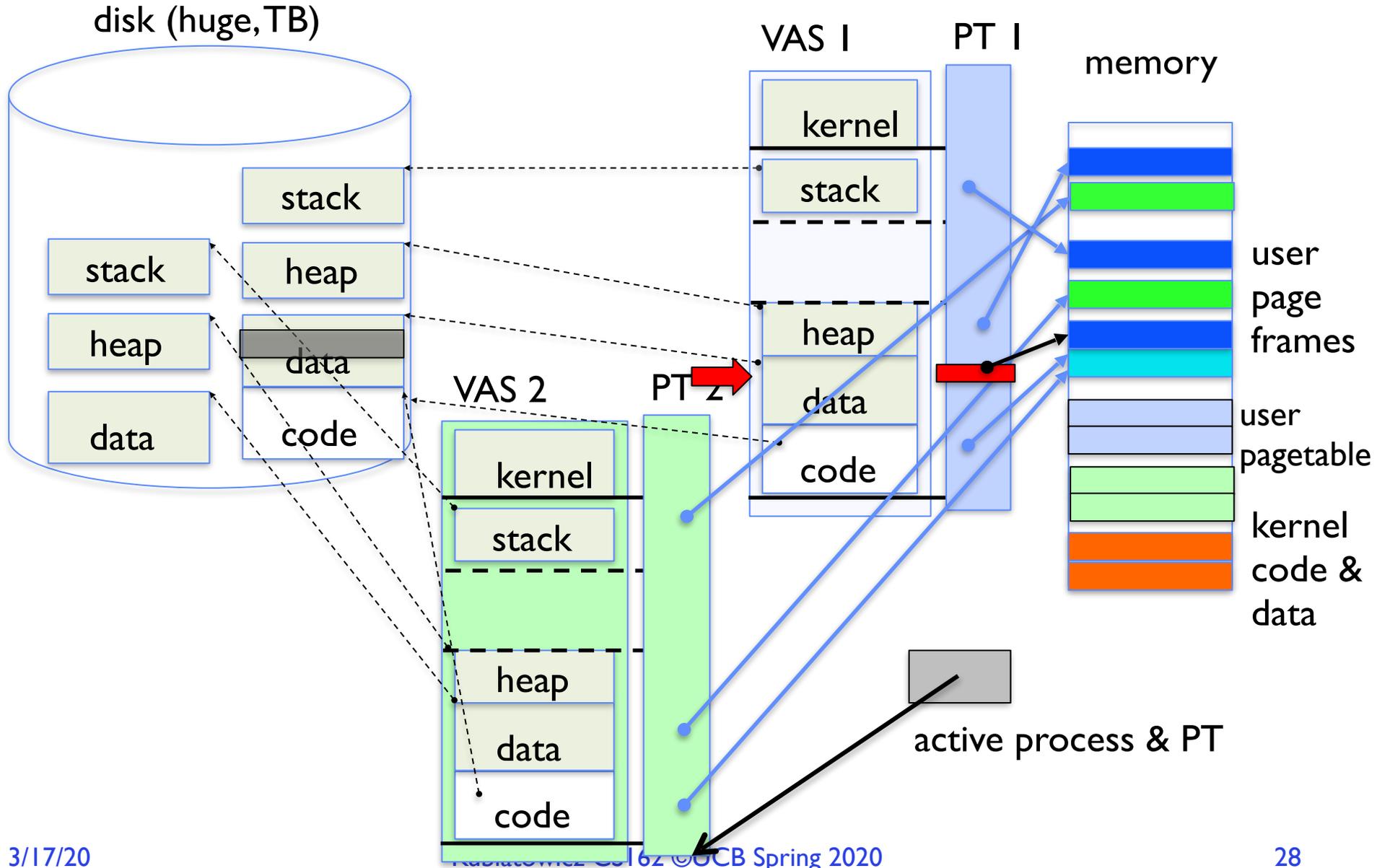
On page Fault ... find & start load



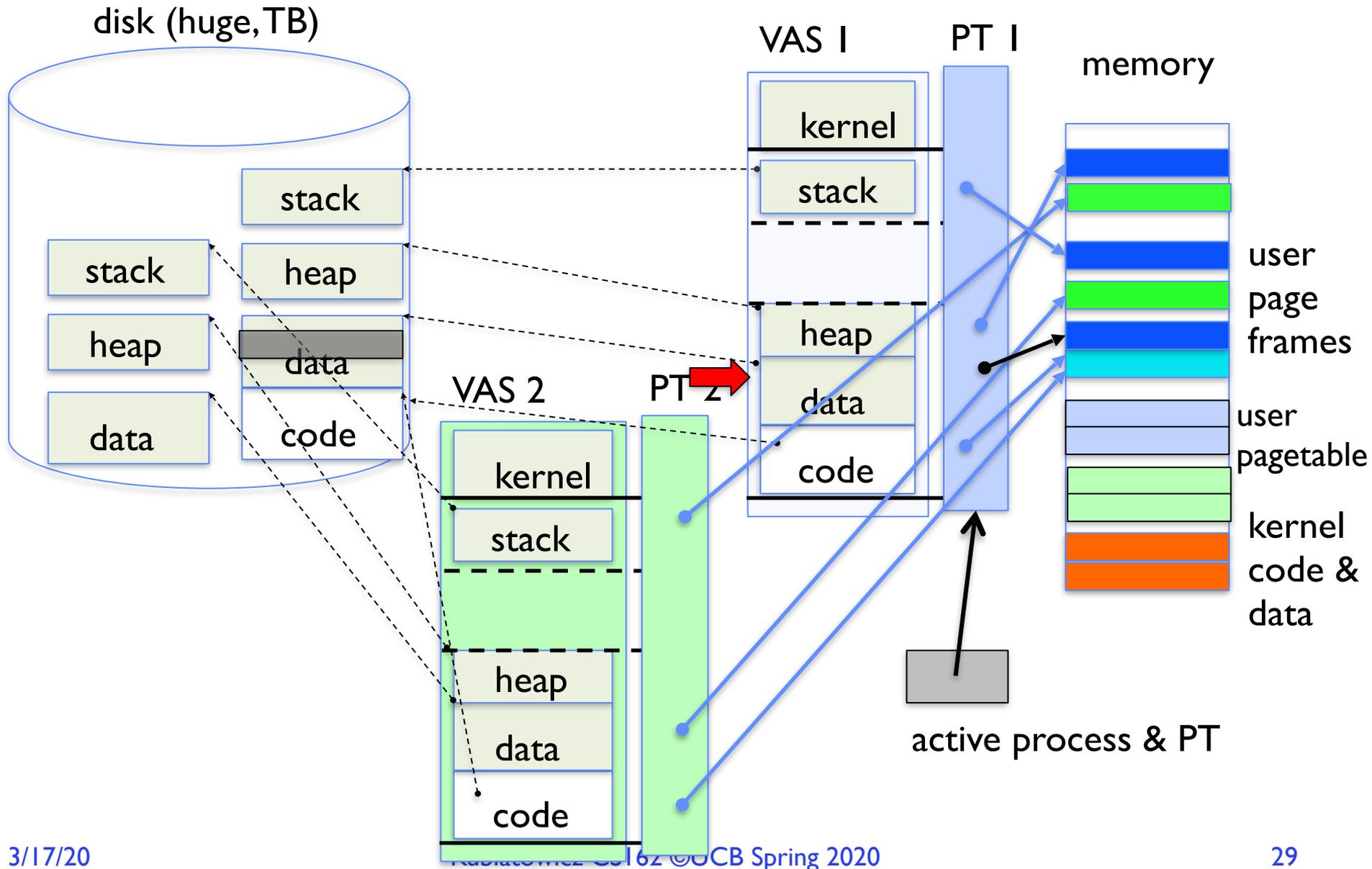
On page Fault ... schedule other P or T



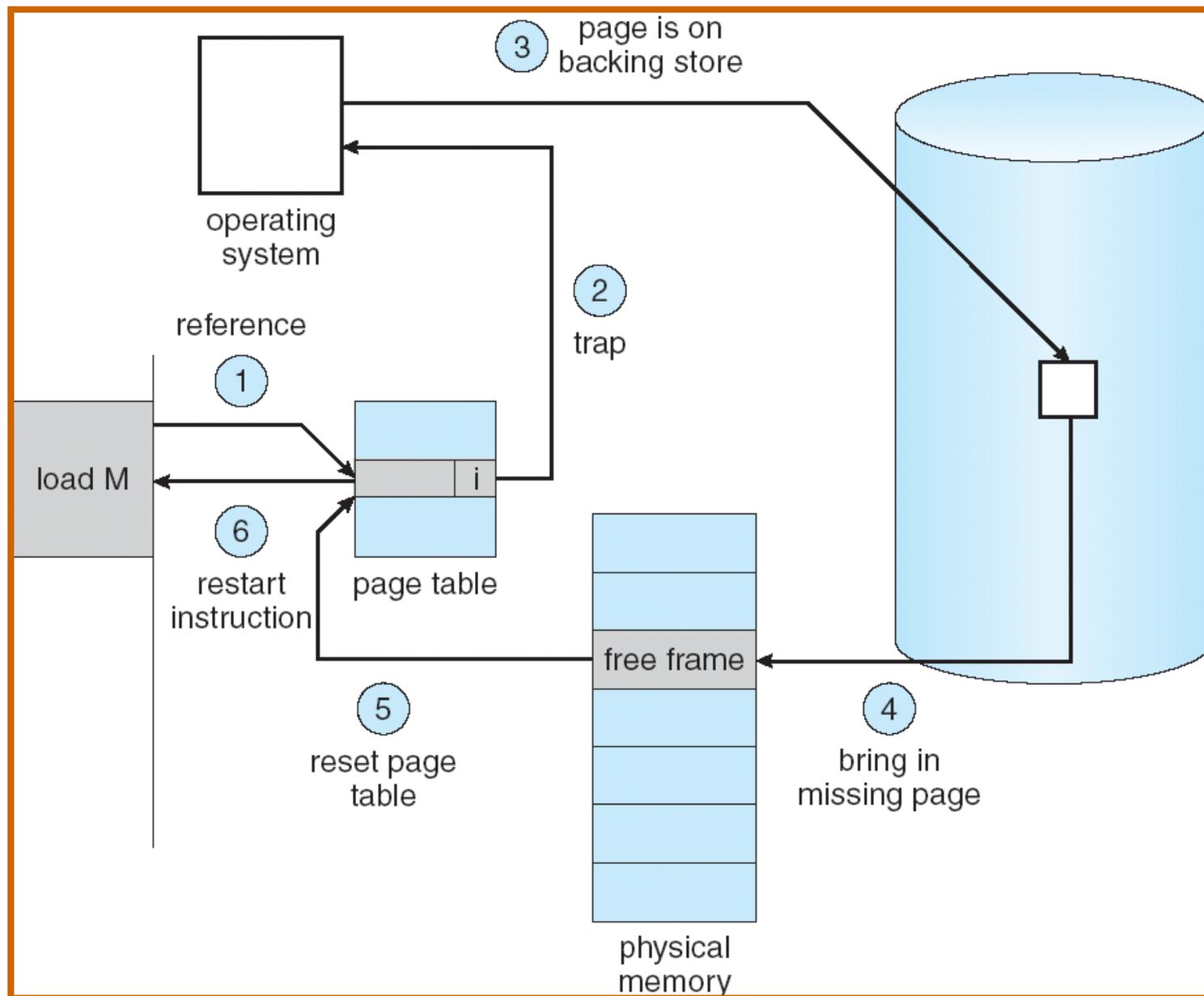
On page Fault ... update PTE



Eventually reschedule faulting thread



Summary: Steps in Handling a Page Fault

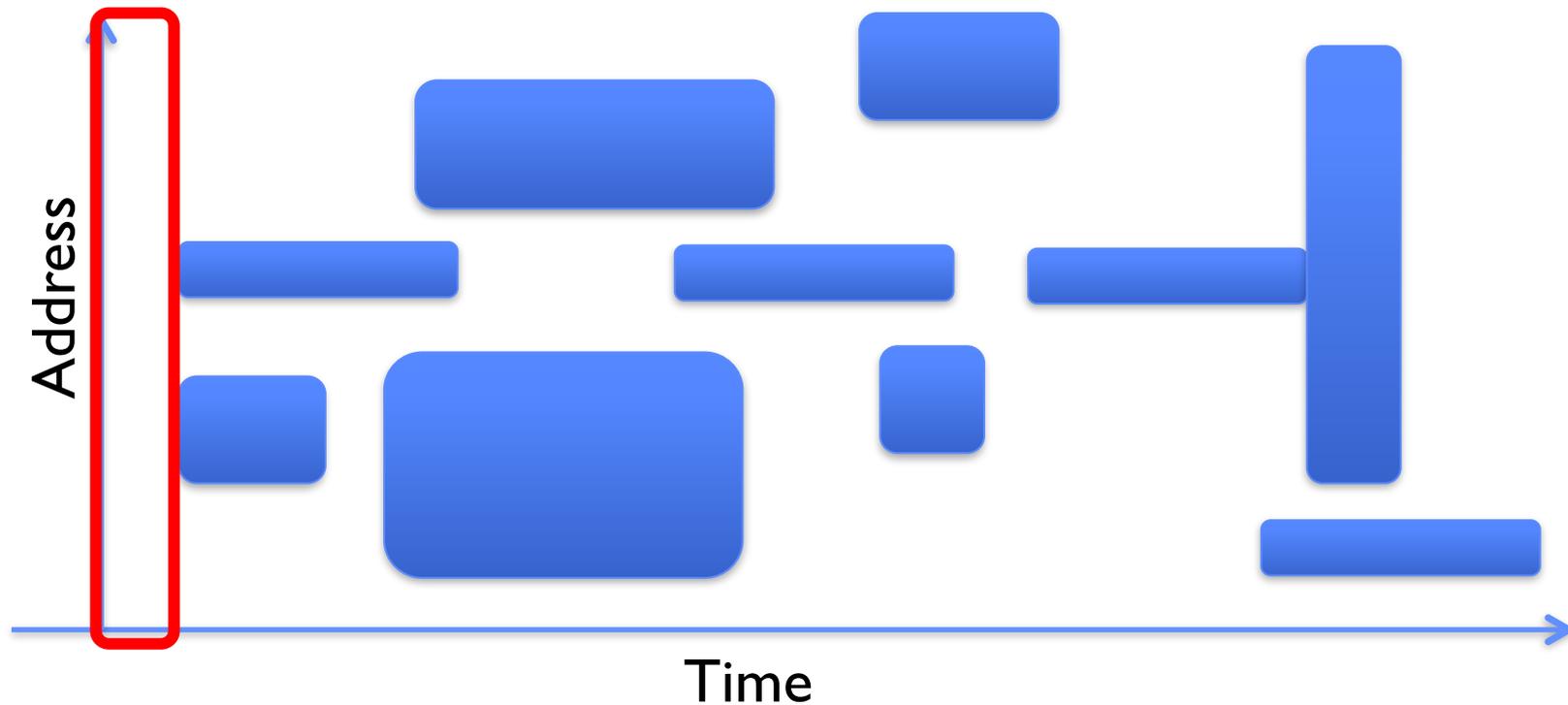


Some questions we need to answer!

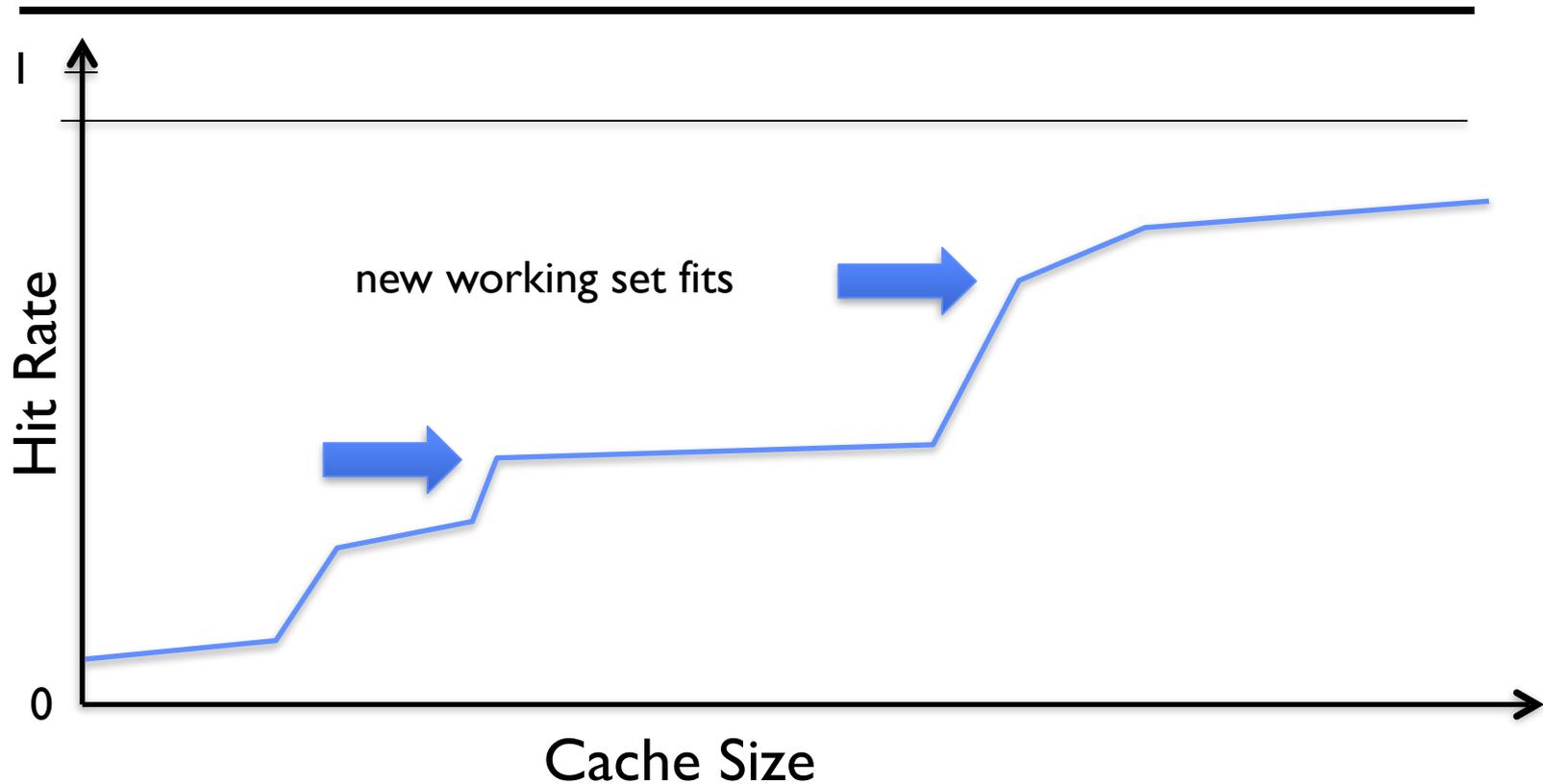
- During a page fault, where does the OS get a free frame?
 - Keeps a free list
 - Unix runs a “reaper” if memory gets too full
 - » Schedule dirty pages to be written back on disk
 - » Zero (clean) pages which haven’t been accessed in a while
 - As a last resort, evict a dirty page first
- How can we 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



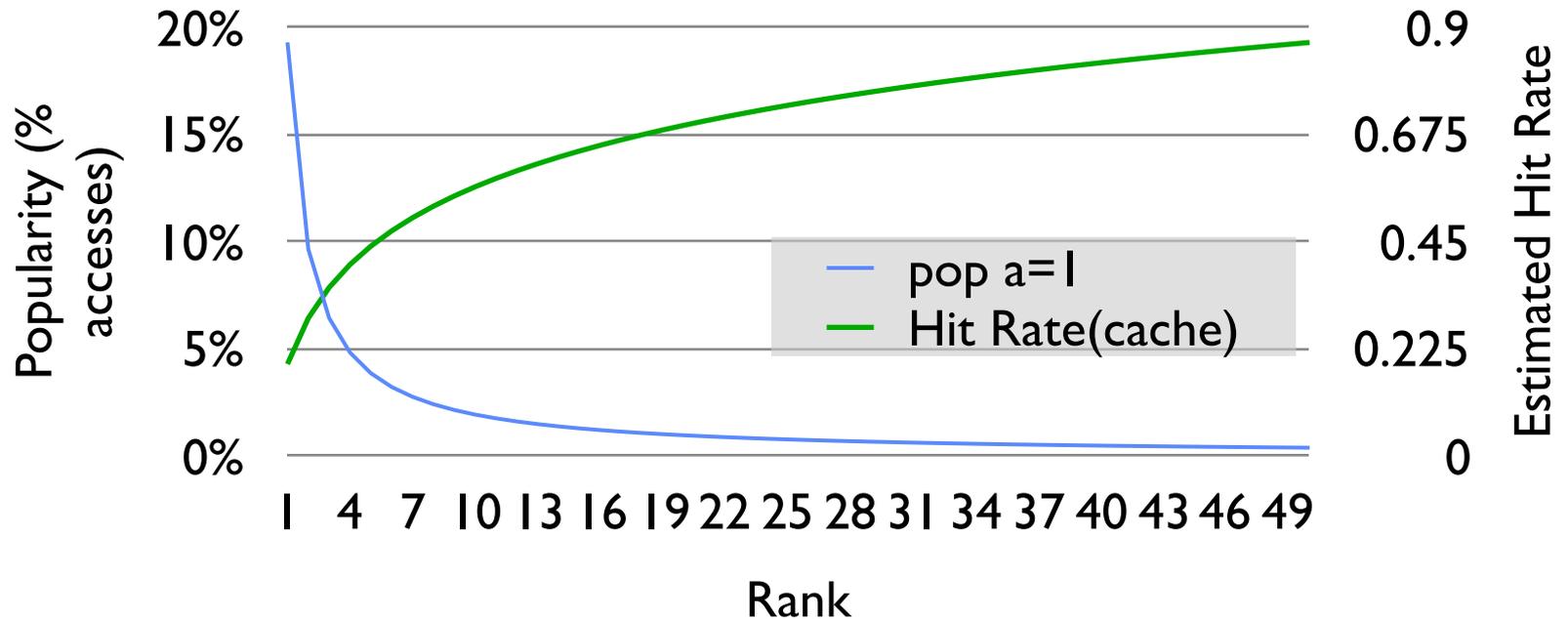
Cache Behavior under WS model



- Amortized by fraction of time the Working Set is active
- Transitions from one WS to the next
- Capacity, Conflict, Compulsory misses
- Applicable to memory caches and pages.

Another model of Locality: Zipf

$$P \text{ access}(\text{rank}) = 1/\text{rank}$$



- Likelihood of accessing item of rank r is $\alpha 1/r^a$
- 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

- Since Demand Paging like caching, can compute average access time! (“Effective Access Time”)
 - $EAT = \text{Hit Rate} \times \text{Hit Time} + \text{Miss Rate} \times \text{Miss Time}$
 - $EAT = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty}$
- Example:
 - Memory access time = 200 nanoseconds
 - Average page-fault service time = 8 milliseconds
 - Suppose p = Probability of miss, $1-p$ = Probability of hit
 - Then, we can compute EAT as follows:
$$\begin{aligned} EAT &= 200\text{ns} + p \times 8 \text{ ms} \\ &= 200\text{ns} + p \times 8,000,000\text{ns} \end{aligned}$$
- If one access out of 1,000 causes a page fault, then $EAT = 8.2 \mu\text{s}$:
 - This is a slowdown by a factor of 40!
- What if want slowdown by less than 10%?
 - $EAT < 200\text{ns} \times 1.1 \Rightarrow p < 2.5 \times 10^{-6}$
 - This is about 1 page fault in 400,000!

What Factors Lead to Misses in Page Cache?

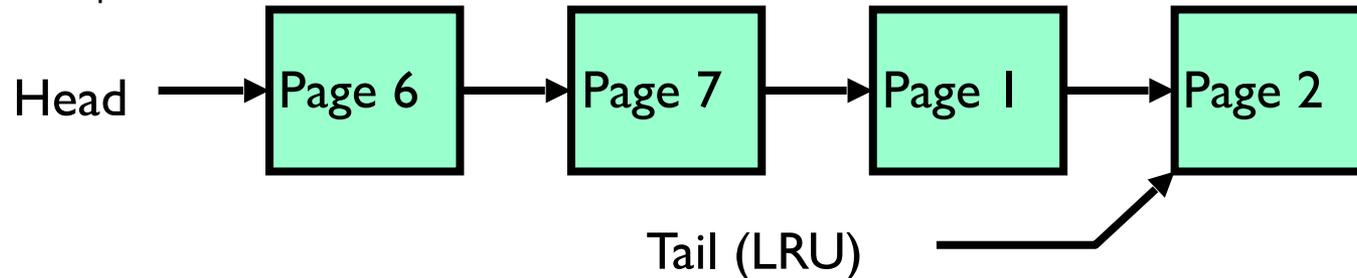
- **Compulsory Misses:**
 - Pages that have never been paged into memory before
 - How might we remove these misses?
 - » Prefetching: loading them into memory before needed
 - » Need to predict future somehow! More later
- **Capacity Misses:**
 - Not enough memory. Must somehow increase available memory size.
 - Can we do this?
 - » One option: 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
 - 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 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 ...*

Replacement Policies (Con't)

- LRU (Least Recently Used):
 - Replace page that hasn't been used for the longest time
 - Programs have locality, so if something not used for a while, unlikely to be used in the near future.
 - Seems like LRU should be a good approximation to MIN.
- How to implement LRU? Use a list!



- On each use, remove page from list and place at head
 - LRU page is at tail
- Problems with this scheme for paging?
 - Need to know immediately when each page used so that can change position in list...
 - Many instructions for each hardware access
- In practice, people approximate LRU (more later)

Example: FIFO (strawman)

- 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:	A	B	C	A	B	D	A	D	B	C	B
Page:											
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

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

- MIN: 5 faults
 - Where will D be brought in? Look for page not referenced farthest in future
- What will LRU do?
 - Same decisions as MIN here, but won't always be true!

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

- Every reference is a page fault!
- Fairly contrived example of working set of $N+1$ on N frames

When will LRU perform badly?

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

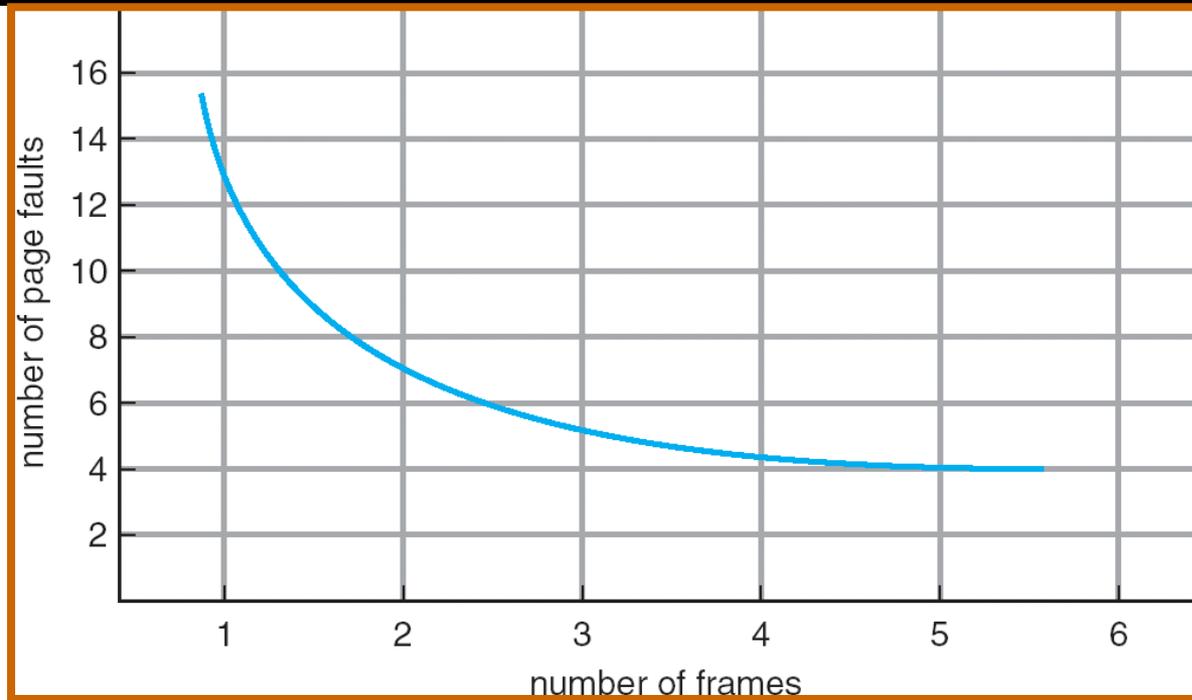
Ref:	A	B	C	D	A	B	C	D	A	B	C	D
Page:												
1	A			D			C			B		
2		B			A			D			C	
3			C			B			A			D

– Every reference is a page fault!

- MIN Does much better:

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

Graph of Page Faults Versus The Number of Frames



- One desirable property: When you add memory the miss rate drops
 - Does this always happen?
 - Seems like it should, right?
- No: Bélády's anomaly
 - Certain replacement algorithms (FIFO) don't have this obvious property!

Adding Memory Doesn't Always Help Fault Rate

- Does adding memory reduce number of page faults?
 - Yes for LRU and MIN
 - Not necessarily for FIFO! (Called B el ady's anomaly)

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	

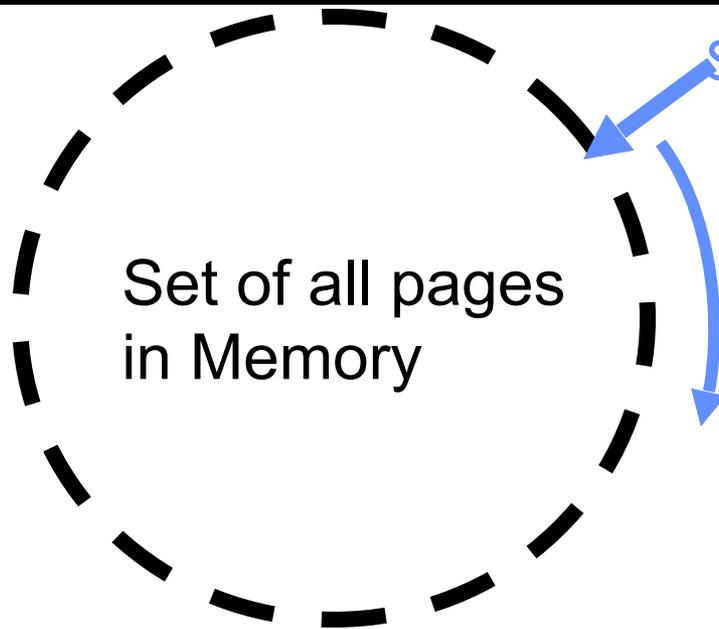
Ref: Page:	A	B	C	D	A	B	E	A	B	C	D	E
1	A						E				D	
2		B						A				E
3			C						B			
4				D						C		

- After adding memory:
 - With FIFO, contents can be completely different
 - In contrast, with LRU or MIN, contents of memory with X pages are a subset of contents with X+1 Page

Implementing LRU

- Perfect:
 - Timestamp page on each reference
 - Keep list of pages ordered by time of reference
 - Too expensive to implement in reality for many reasons
- **Clock Algorithm:** 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 “use” bit per physical page:
 - » Hardware sets use bit on each reference
 - » If use bit isn’t set, means not referenced in a long time
 - On page fault:
 - » Advance clock hand (not real time)
 - » Check use bit: 1 → used recently; clear and leave alone
 0 → selected candidate for replacement
 - Will always find a page or loop forever?
 - » Even if all use bits set, will eventually loop around ⇒ FIFO

Clock Algorithm: Not Recently Used



Single Clock Hand:

Advances only on page fault!

Check for pages not used recently

Mark pages as not used recently



- What if hand moving slowly?
 - Good sign or bad sign?
 - » Not many page faults and/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?

Nth Chance version of Clock Algorithm

- **Nth chance algorithm:** Give page N chances
 - OS keeps counter per page: # sweeps
 - On page fault, OS checks use bit:
 - » 1 → clear use and also clear counter (used in last sweep)
 - » 0 → 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
 - » If $N \sim 1K$, really good approximation
 - Why pick small N? More efficient
 - » Otherwise might have to look a long way to find free page
- What about dirty pages?
 - Takes extra overhead to replace a dirty page, so give dirty pages an extra chance before replacing?
 - Common approach:
 - » Clean pages, use $N=1$
 - » Dirty pages, use $N=2$ (and write back to disk when $N=1$)

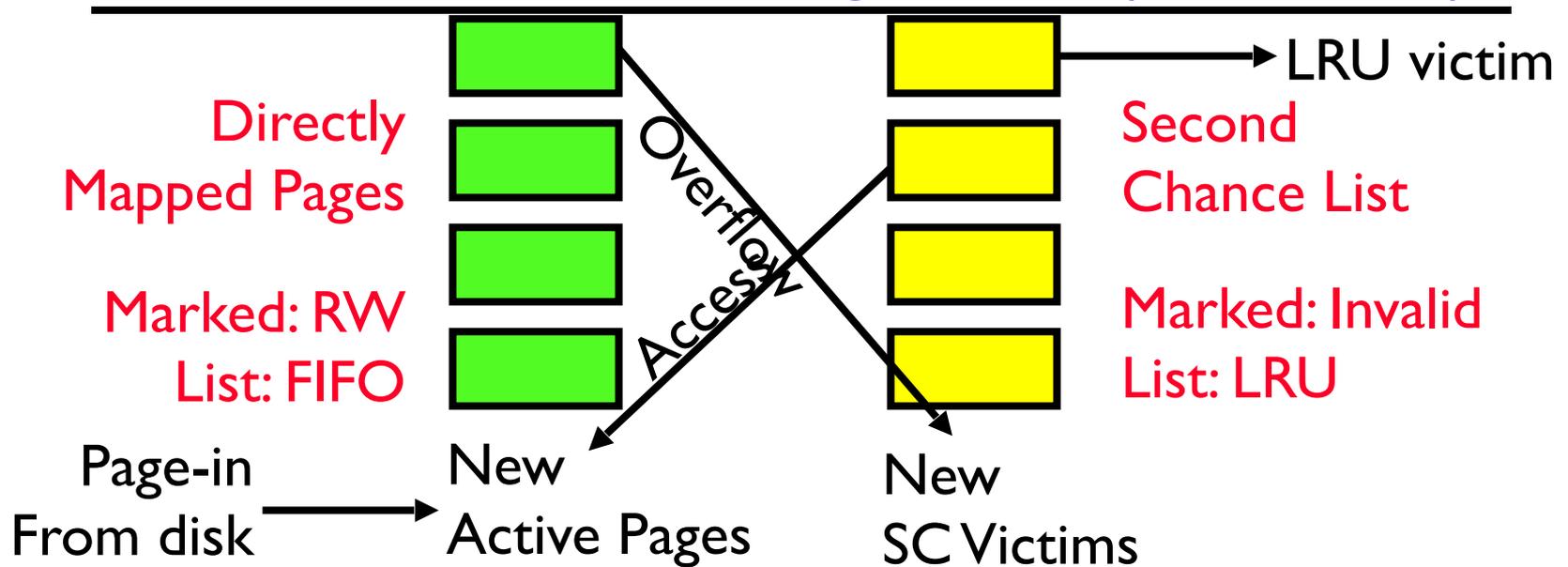
Clock Algorithms: Details

- 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!
- Do we really need hardware-supported “modified” bit?
 - No. Can emulate it (BSD Unix) using read-only bit
 - » Initially, mark all pages as read-only, even data pages
 - » On write, trap to OS. OS sets software “modified” bit, and marks page as read-write.
 - » Whenever page comes back in from disk, mark read-only

Clock Algorithms Details (continued)

- Do we really need a hardware-supported “use” bit?
 - No. Can emulate it similar to above:
 - » Mark all pages as invalid, even if in memory
 - » On read to invalid page, trap to OS
 - » OS sets use bit, and marks page read-only
 - Get modified bit in same way as previous:
 - » On write, trap to OS (either invalid or read-only)
 - » Set use and modified bits, mark page read-write
 - When clock hand passes by, reset use and modified bits and mark page as invalid again
- Remember, however, clock is just an approximation of LRU!
 - Can we do a better approximation, given that we have to take page faults on some reads and writes to collect use information?
 - Need to identify an old page, not oldest page!
 - Answer: second chance list

Second-Chance List Algorithm (VAX/VMS)

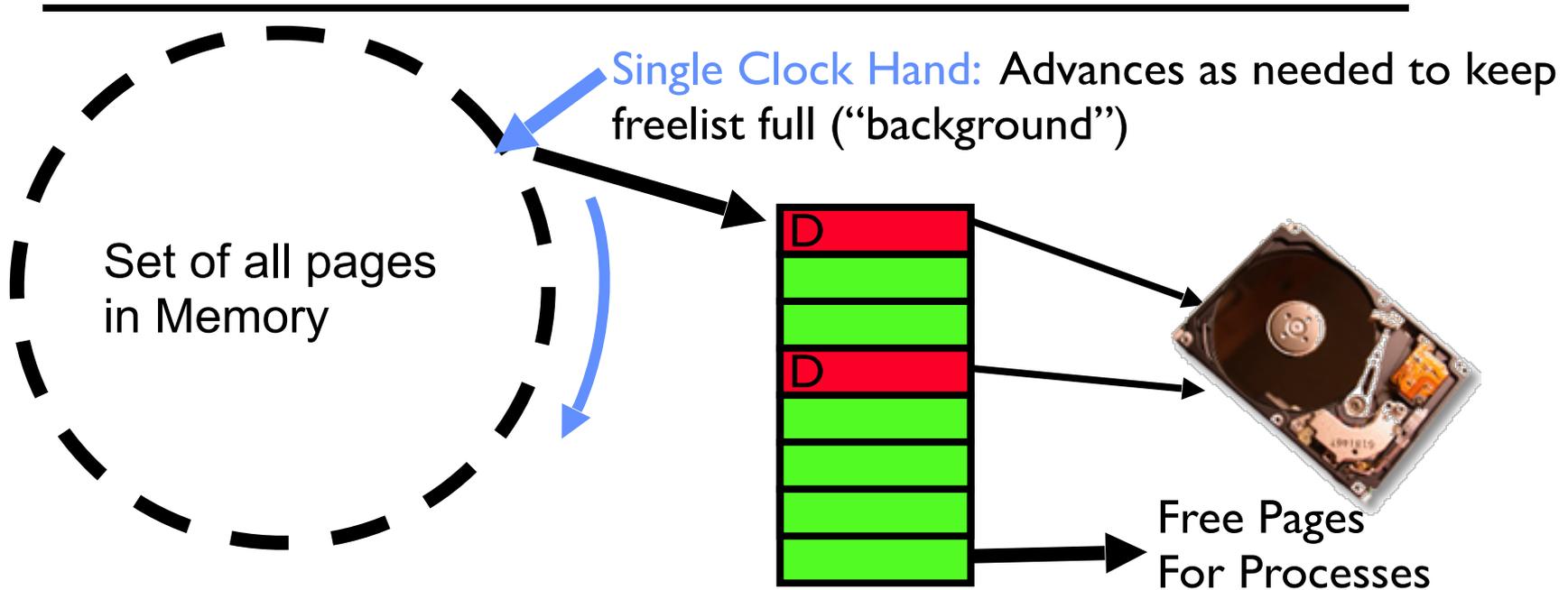


- Split memory in two: Active list (RW), SC list (Invalid)
- Access pages in Active list at full speed
- Otherwise, Page Fault
 - Always move overflow page from end of Active list to front of Second-chance list (SC) and mark invalid
 - Desired Page On SC List: move to front of Active list, mark RW
 - Not on SC list: page in to front of Active list, mark RW; page out LRU victim at end of SC list

Second-Chance List Algorithm (continued)

- How many pages for second chance list?
 - If 0 \Rightarrow FIFO
 - If all \Rightarrow LRU, but page fault on every page reference
- Pick intermediate value. Result is:
 - Pro: Few disk accesses (page only goes to disk if unused for a long time)
 - Con: Increased overhead trapping to OS (software / hardware tradeoff)
- Question: why didn't VAX include "use" bit?
 - Strecker (architect) asked OS people, they said they didn't need it, so didn't implement it
 - He later got blamed, but VAX did OK anyway

Free List



- Keep set of free pages ready for use in demand paging
 - Freelist filled in background by Clock algorithm or other technique ("Pageout demon")
 - Dirty pages start copying back to disk when enter list
- Like VAX second-chance list
 - If page needed before reused, just return to active set
- Advantage: faster for page fault
 - Can always use page (or pages) immediately on fault

Demand Paging (more details)

- Does software-loaded TLB need use bit?
Two Options:
 - Hardware sets use bit in TLB; when TLB entry is replaced, software copies use bit back to page table
 - Software manages TLB entries as FIFO list; everything not in TLB is Second-Chance list, managed as strict LRU
- Core Map
 - Page tables map virtual page → physical page
 - Do we need a reverse mapping (i.e. physical page → virtual page)?
 - » Yes. Clock algorithm runs through page frames. If sharing, then multiple virtual-pages per physical page
 - » Can't push page out to disk without invalidating all PTEs

Allocation of Page Frames (Memory Pages)

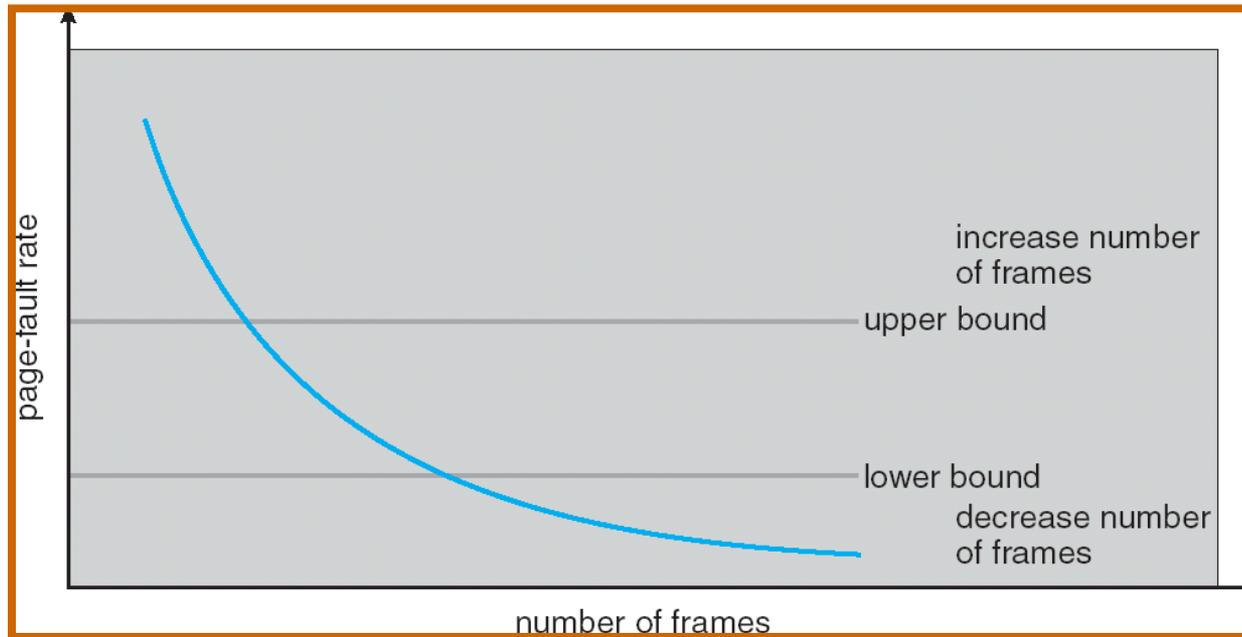
- How do we allocate memory among different processes?
 - Does every process get the same fraction of memory? Different fractions?
 - Should we completely swap some processes out of memory?
- Each process needs *minimum* number of pages
 - Want to make sure that all processes **that are loaded into memory** can make forward progress
 - Example: IBM 370 – 6 pages to handle SS MOVE instruction:
 - » instruction is 6 bytes, might span 2 pages
 - » 2 pages to handle *from*
 - » 2 pages to handle *to*
- Possible Replacement Scopes:
 - **Global replacement** – process selects replacement frame from set of all frames; one process can take a frame from another
 - **Local replacement** – each process selects from only its own set of allocated frames

Fixed/Priority Allocation

- **Equal allocation** (Fixed Scheme):
 - Every process gets same amount of memory
 - Example: 100 frames, 5 processes → process gets 20 frames
- **Proportional allocation** (Fixed Scheme)
 - Allocate according to the size of process
 - Computation proceeds as follows:
 - s_i = size of process p_i and $S = \sum s_i$
 - m = total number of frames
 - $a_i = (\text{allocation for } p_i) = \frac{s_i}{S} \times m$
- **Priority Allocation:**
 - Proportional scheme using priorities rather than size
 - » Same type of computation as previous scheme
 - Possible behavior: If process p_i generates a page fault, select for replacement a frame from a process with lower priority number
- Perhaps we should use an adaptive scheme instead???
 - What if some application just needs more memory?

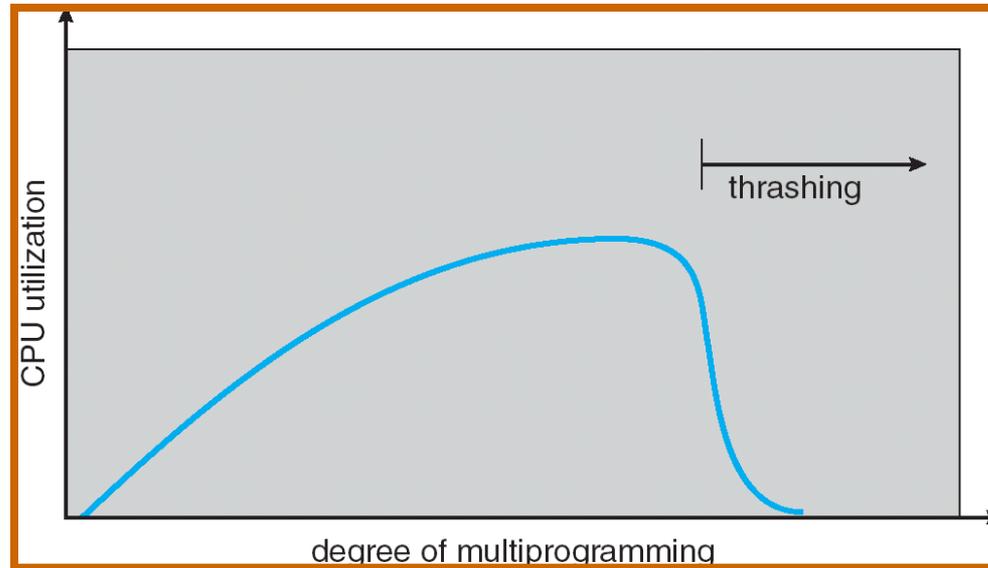
Page-Fault Frequency Allocation

- Can we reduce Capacity misses by dynamically changing the number of pages/application?



- Establish “acceptable” page-fault rate
 - If actual rate too low, process loses frame
 - If actual rate too high, process gains frame
- Question: What if we just don't have enough memory?

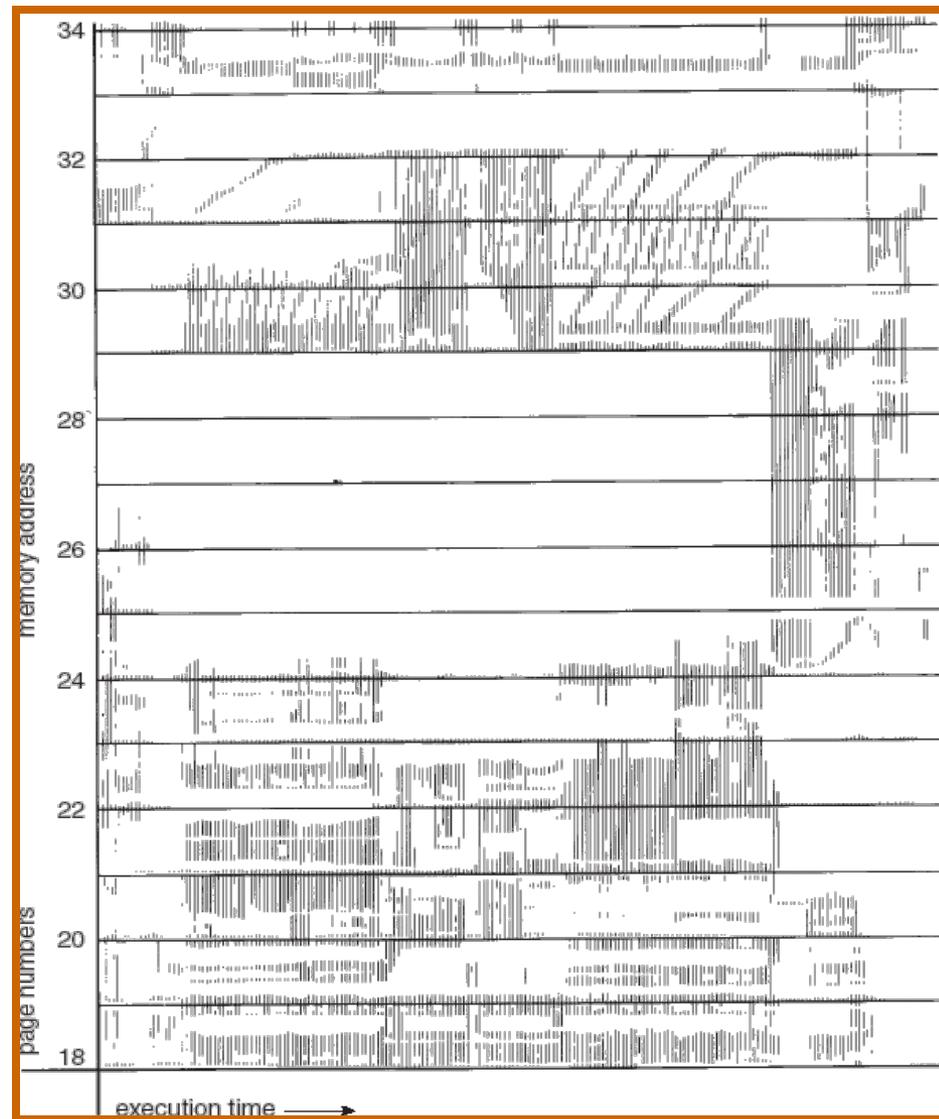
Thrashing



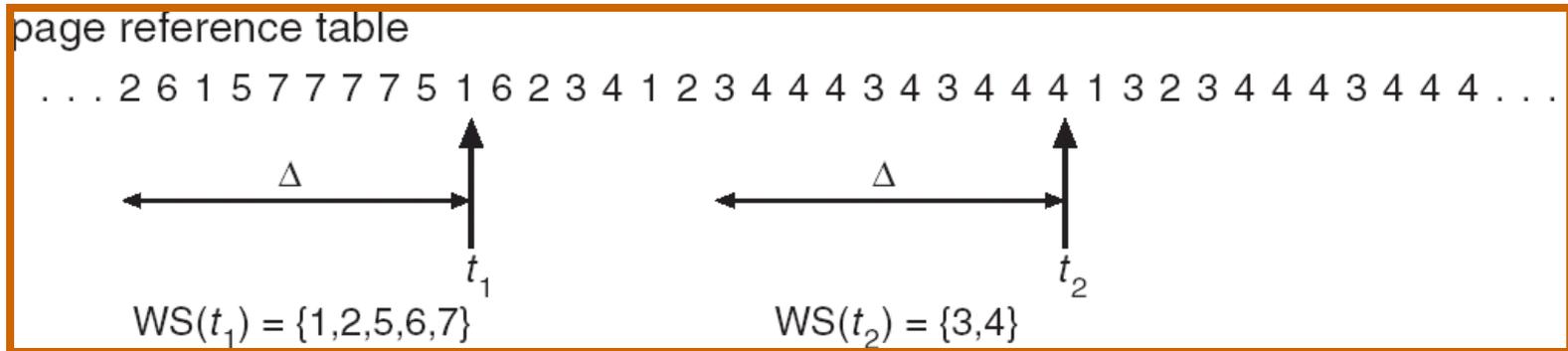
- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
 - low CPU utilization
 - operating system spends most of its time swapping to disk
- **Thrashing** \equiv a process is busy swapping pages in and out
- Questions:
 - How do we detect Thrashing?
 - What is best response to Thrashing?

Locality In A Memory-Reference Pattern

- Program Memory Access Patterns have temporal and spatial locality
 - Group of Pages accessed along a given time slice called the “Working Set”
 - Working Set defines minimum number of pages needed for process to behave well
- Not enough memory for Working Set \Rightarrow Thrashing
 - Better to swap out process?



Working-Set Model



- $\Delta \equiv$ working-set window \equiv fixed number of page references
 - Example: 10,000 instructions
- WS_i (working set of Process P_i) = total set of pages referenced in the most recent Δ (varies in time)
 - if Δ too small will not encompass entire locality
 - if Δ too large will encompass several localities
 - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \sum |WS_i| \equiv$ total demand frames
- if $D > m \Rightarrow$ Thrashing
 - Policy: if $D > m$, then suspend/swap out processes
 - This can improve overall system behavior by a lot!

What about Compulsory Misses?

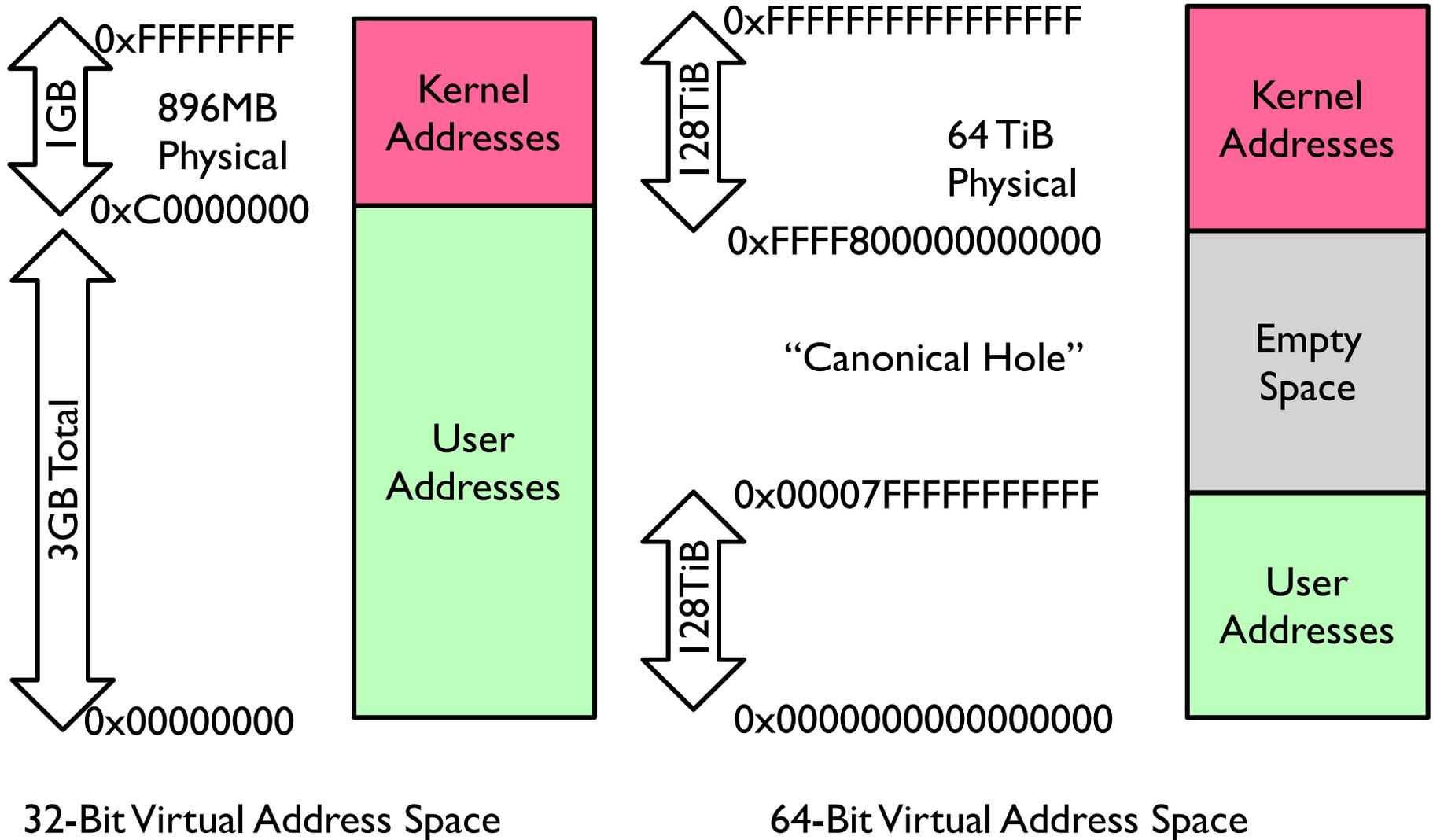
- Recall that compulsory misses are misses that occur the first time that a page is seen
 - Pages that are touched for the first time
 - Pages that are touched after process is swapped out/swapped back in
- Clustering:
 - On a page-fault, bring in multiple pages “around” the faulting page
 - Since efficiency of disk reads increases with sequential reads, makes sense to read several sequential pages
- Working Set Tracking:
 - Use algorithm to try to track working set of application
 - When swapping process back in, swap in working set

- Physical page frames often shared by many different address spaces/page tables
 - All children forked from given process
 - Shared memory pages between processes
- Whatever reverse mapping mechanism that is in place must be very fast
 - Must hunt down all page tables pointing at given page frame when freeing a page
 - Must hunt down all PTEs when seeing if pages “active”
- Implementation options:
 - For every page descriptor, keep linked list of page table entries that point to it
 - » Management nightmare – expensive
 - Linux 2.6: Object-based reverse mapping
 - » Link together memory region descriptors instead (much coarser granularity)

Linux Memory Details?

- Memory management in Linux considerably more complex than the examples we have been discussing
- Memory Zones: physical memory categories
 - ZONE_DMA: < 16MB memory, DMAable on ISA bus
 - ZONE_NORMAL: 16MB → 896MB (mapped at 0xC0000000)
 - ZONE_HIGHMEM: Everything else (> 896MB)
- Each zone has 1 freelist, 2 LRU lists (Active/Inactive)
- Many different types of allocation
 - SLAB allocators, per-page allocators, mapped/unmapped
- Many different types of allocated memory:
 - Anonymous memory (not backed by a file, heap/stack)
 - Mapped memory (backed by a file)
- Allocation priorities
 - Is blocking allowed/etc

Linux Virtual memory map



Virtual Map (Details)

- Kernel memory not generally visible to user
 - Exception: special VDSO (virtual dynamically linked shared objects) facility that maps kernel code into user space to aid in system calls (and to provide certain actual system calls such as `gettimeofday()`)
- Every physical page described by a “page” structure
 - Collected together in lower physical memory
 - Can be accessed in kernel virtual space
 - Linked together in various “LRU” lists
- For 32-bit virtual memory architectures:
 - When physical memory < 896MB
 - » All physical memory mapped at `0xC0000000`
 - When physical memory \geq 896MB
 - » Not all physical memory mapped in kernel space all the time
 - » Can be temporarily mapped with addresses $>$ `0xCC000000`
- For 64-bit virtual memory architectures:
 - All physical memory mapped above `0xFFFF800000000000`

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: 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
- Thrashing: a process is busy swapping pages in and out
 - Process will thrash if working set doesn't fit in memory
 - Need to swap out a process