Operating Systems (Honor Track)

Memory 3: Caching and TLBs (Con’t), Demand Paging

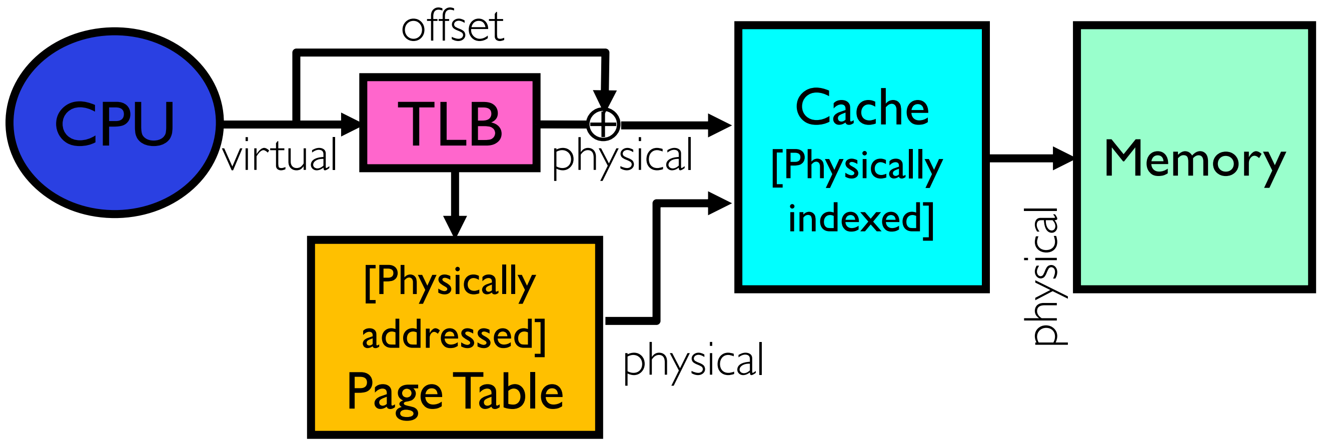
Xin Jin

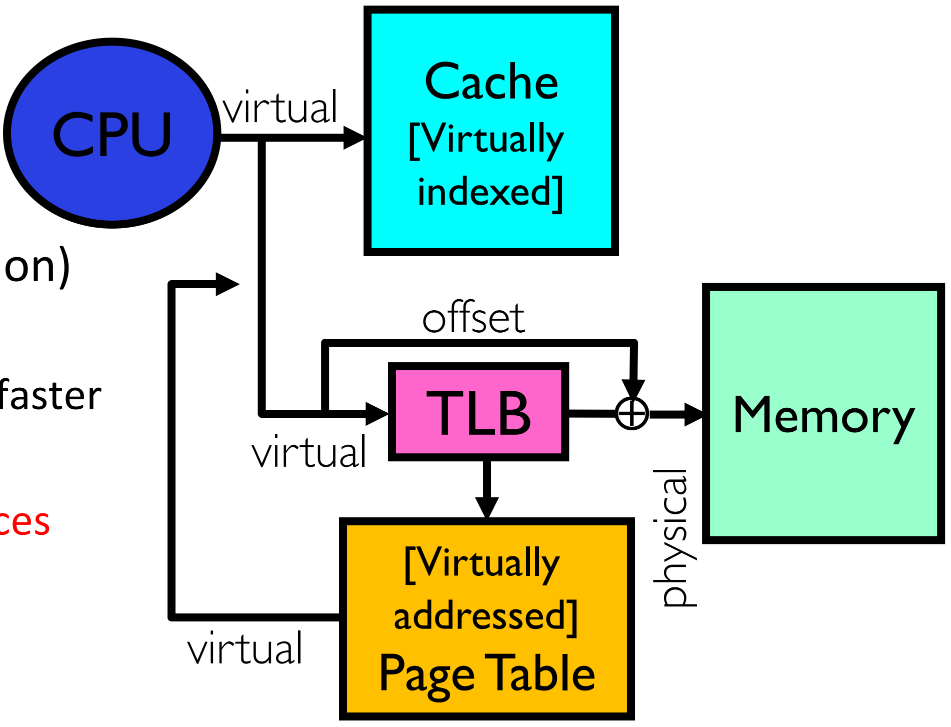
Spring 2022

Acknowledgments: Ion Stoica, Berkeley CS 162

Recap: Physically-Indexed vs Virtually-Indexed Caches

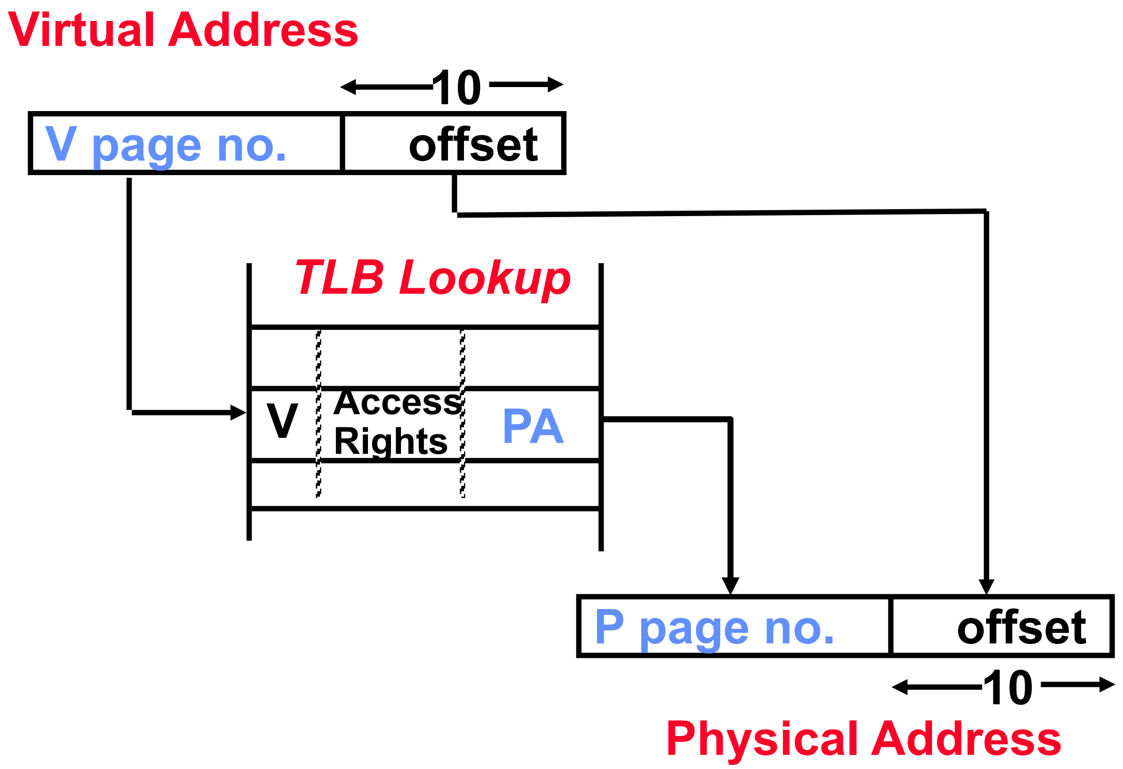
- Physically-Indexed Caches
  - Address handed to cache *after translation*
  - Page Table holds *physical* addresses
  - Benefits:
    - » Every piece of data has single place in cache
    - » Cache can stay unchanged on context switch
  - *Challenges*:
    - » TLB is in critical path of lookup!
  - Pretty Common today (e.g., x86 processors)
- Virtually-Indexed Caches
  - Address handed to cache *before translation*
  - Page Table holds *virtual* addresses (one option)
  - Benefits:
    - » TLB not in critical path of lookup, so can be faster
  - *Challenges:*
    - » Same data could be mapped in multiple places of cache
    - » May need to flush cache on context switch
- We will stick with Physically Addressed Caches for now!

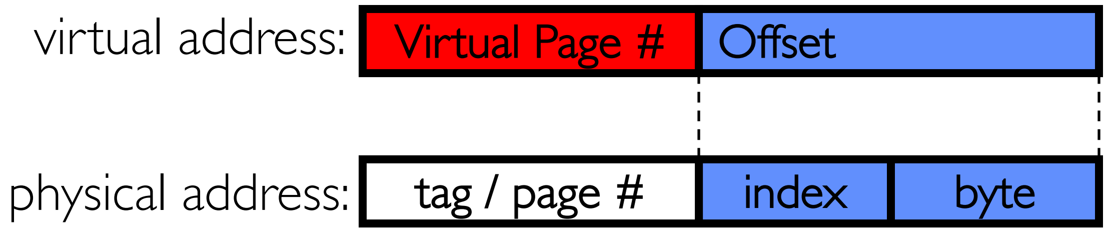




Reducing translation time for physically-indexed caches

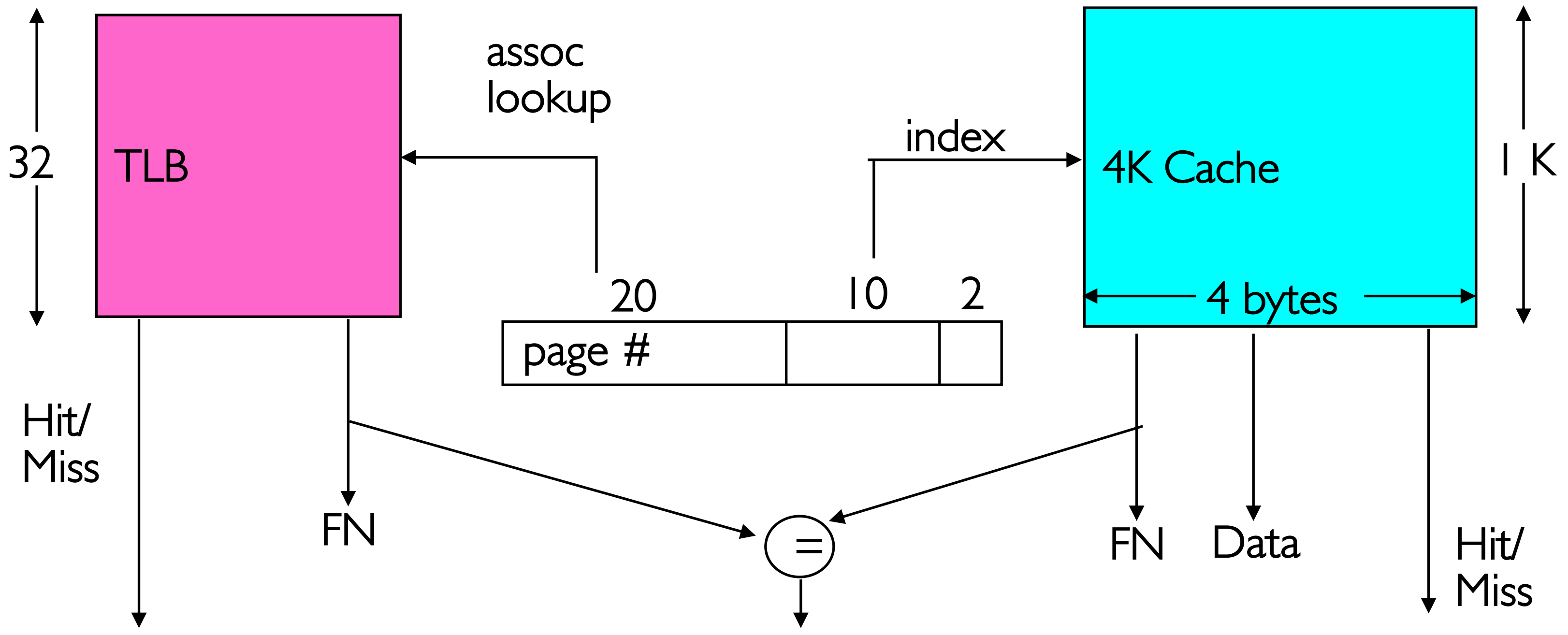
- As described, TLB lookup is in serial with cache lookup
  - Consequently, speed of TLB can impact speed of access to cache
- Machines with TLBs go one step further: overlap TLB lookup with cache access
  - Works because offset available early
  - Offset in virtual address exactly covers the "cache index" and "byte select"
  - Thus can select the cached byte(s) in parallel to perform address translation





Overlapping TLB & Cache Access

- Here is how this might work with a 4K cache:

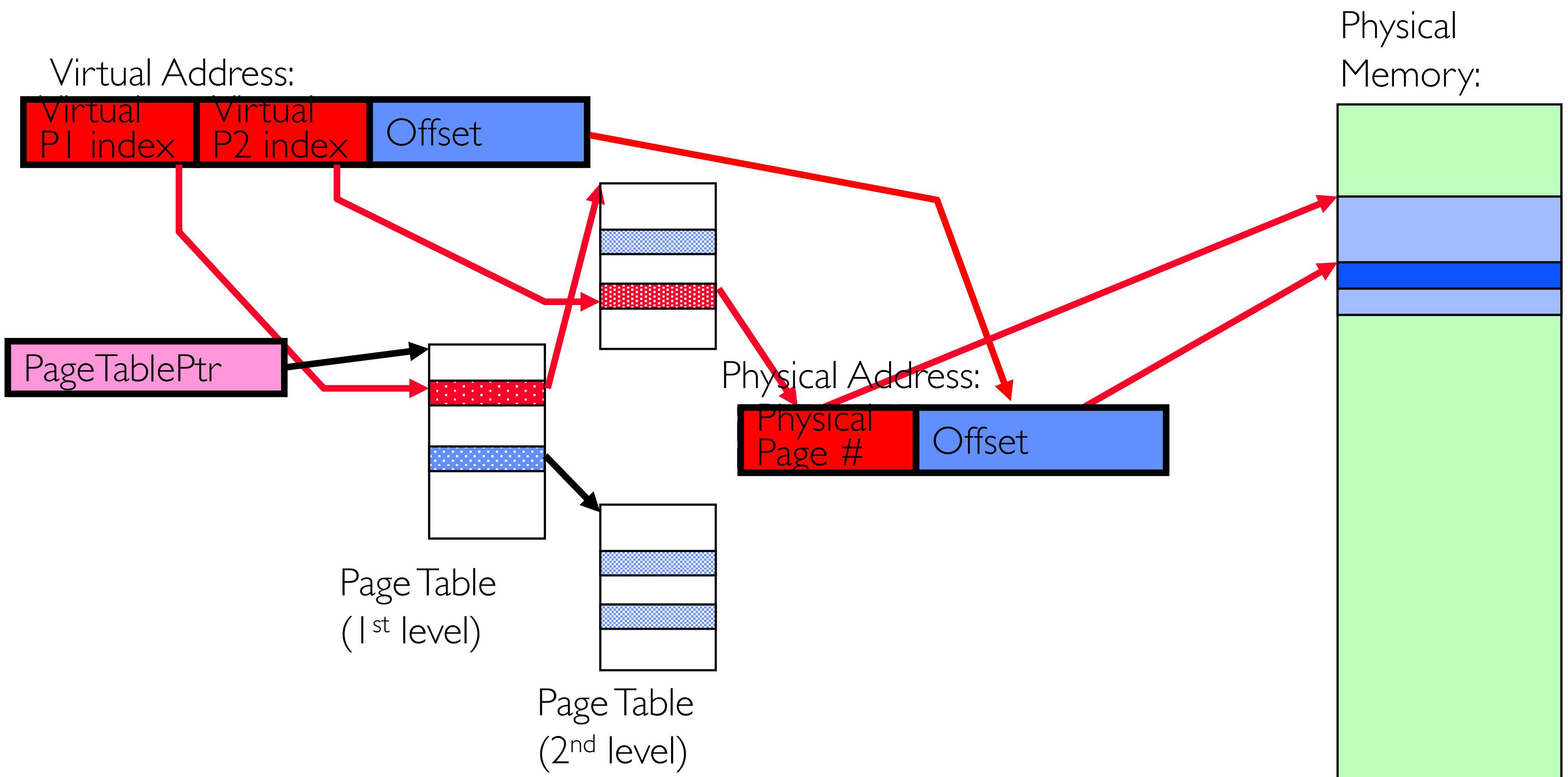


- What if cache size is increased to 8KB?
  - Overlap not complete
  - Need to do something else
- Another option: Virtual Caches would make this faster
  - Tags in cache are virtual addresses
  - Translation only happens on cache misses

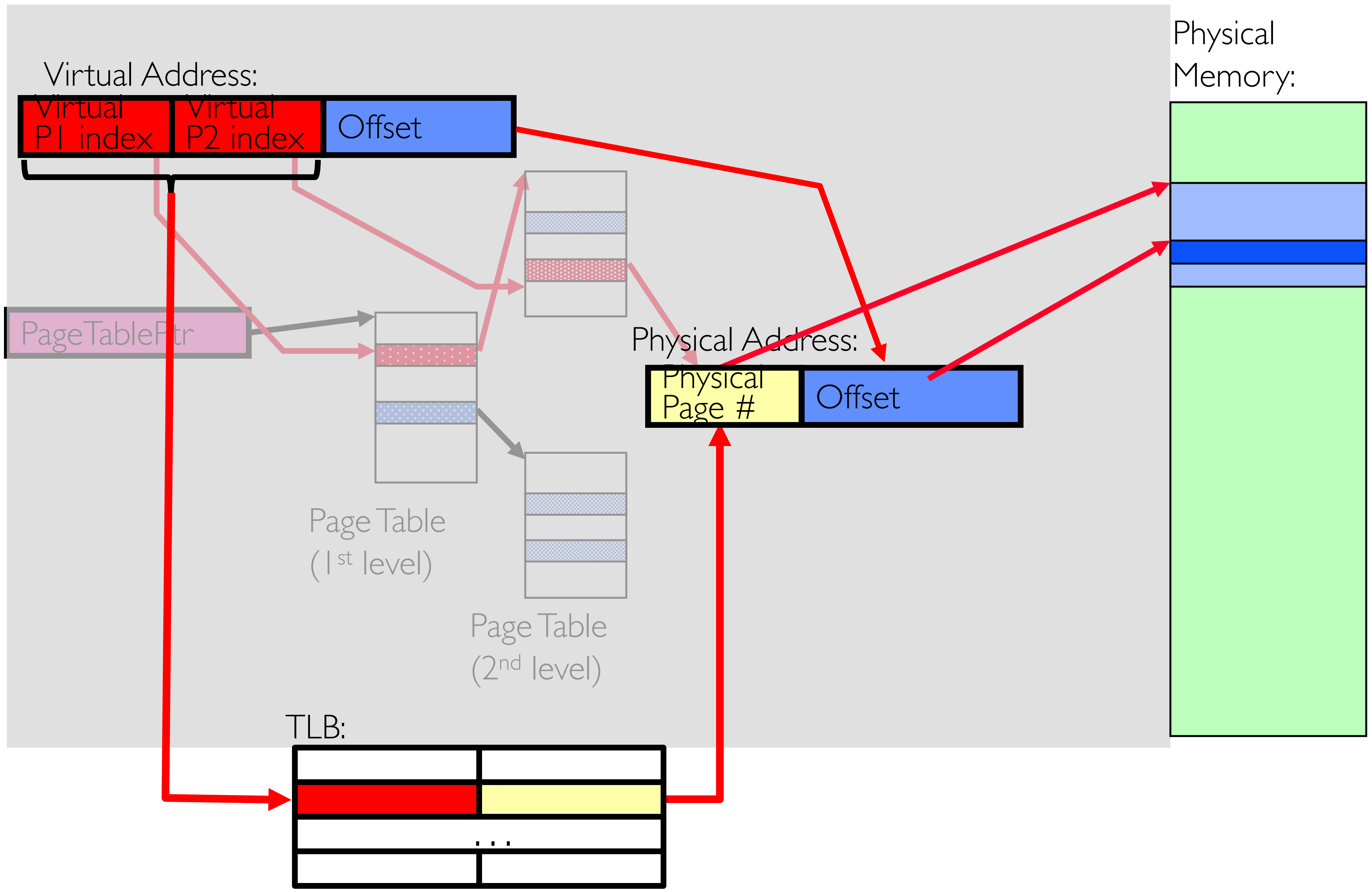
What happens on a Context Switch?

- Need to do something, since TLBs map virtual addresses to physical addresses
  - Address Space just changed, so TLB entries no longer valid!
- Options?
  - Invalidate 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"
- Aside: with Virtually-Indexed cache, need to flush cache!
  - Remember, everyone has their own version of the address "0"!

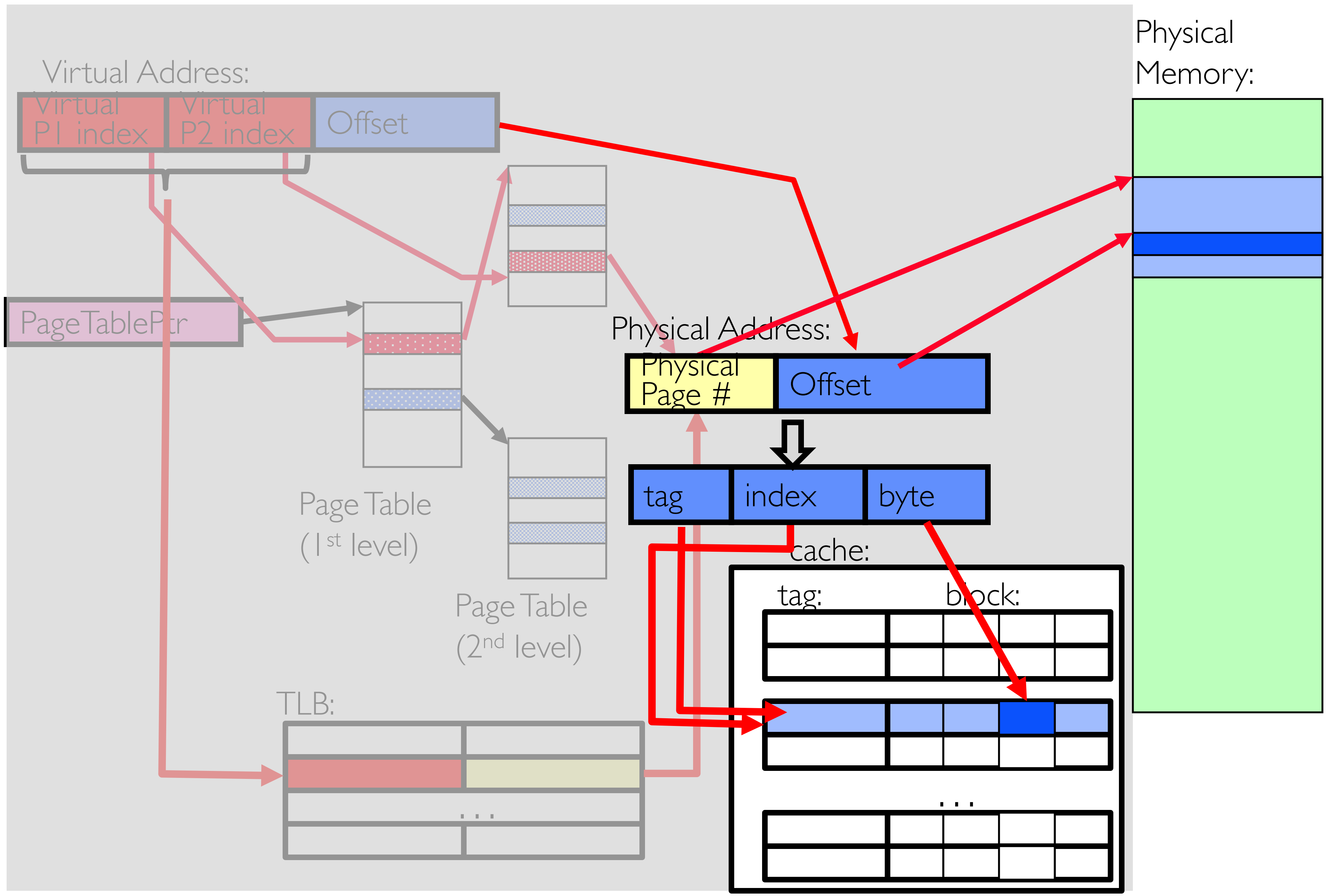
Putting Everything Together: Address Translation



Putting Everything Together: TLB



Putting Everything Together: Cache

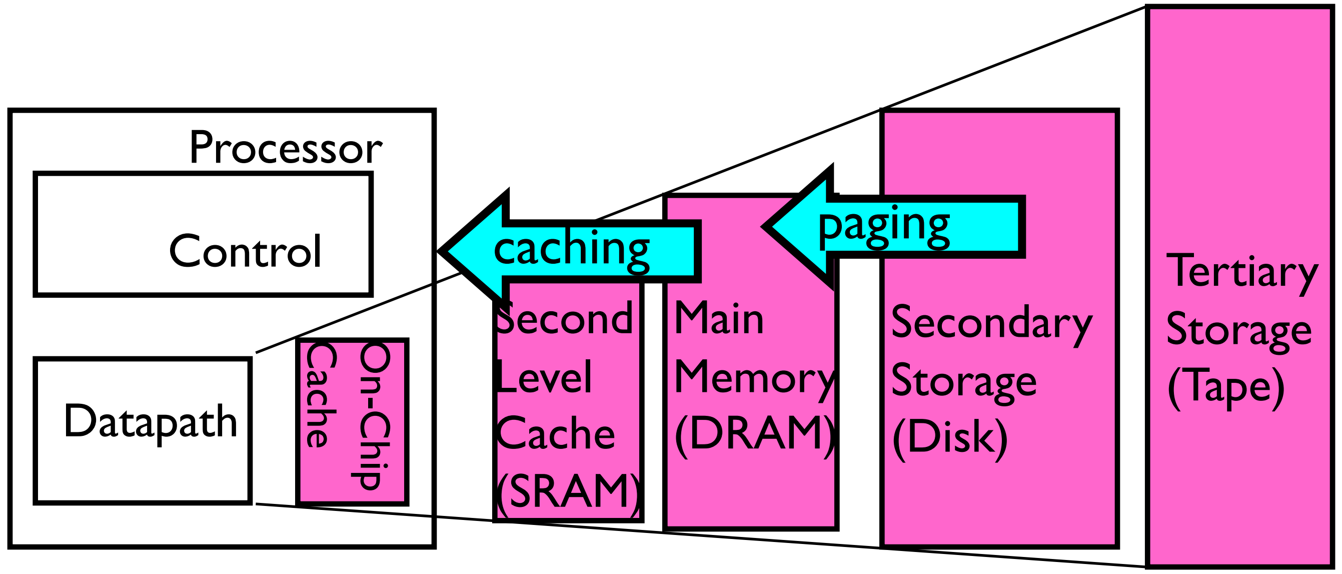


Page Fault

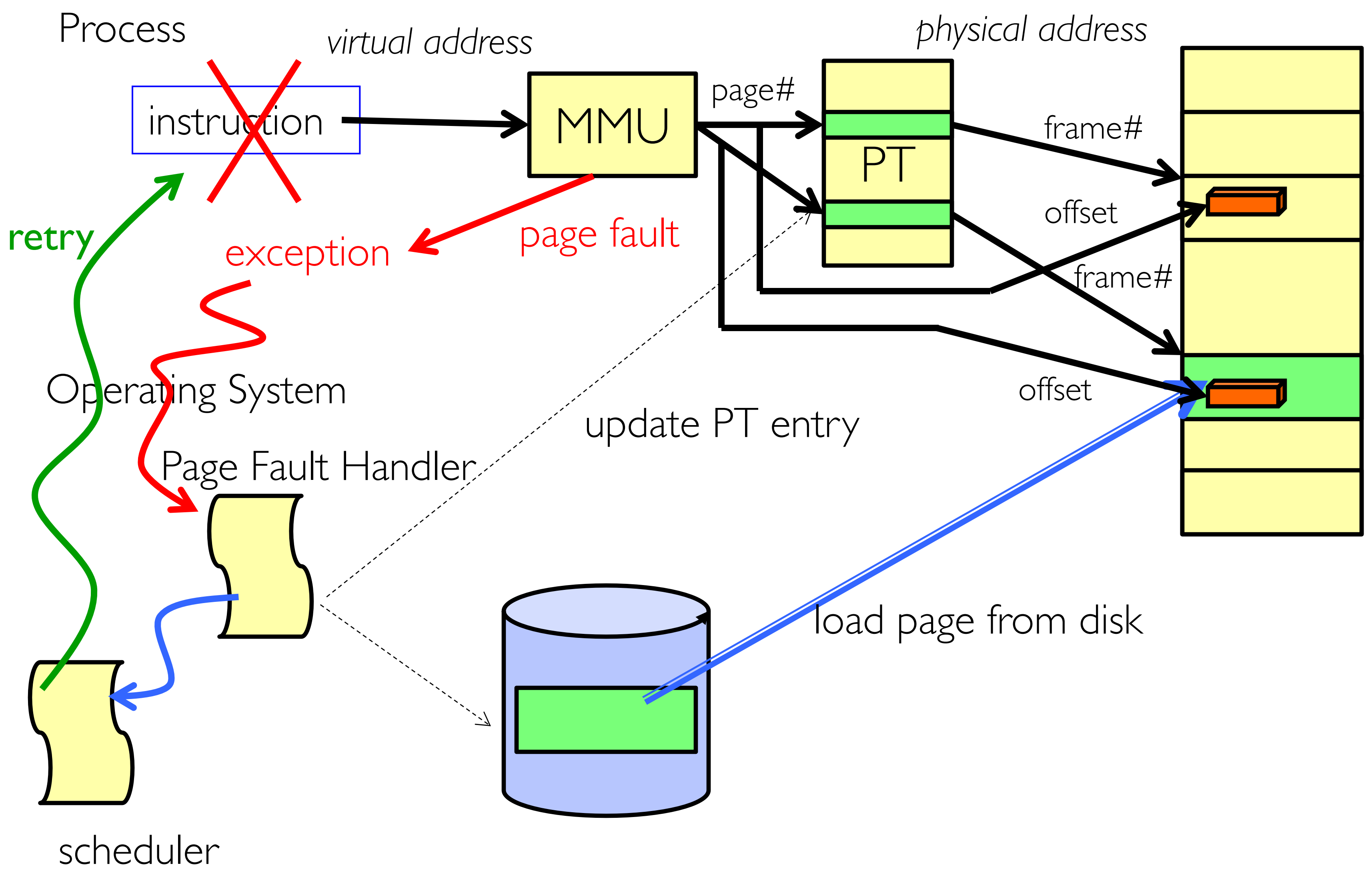
- The Virtual-to-Physical Translation fails
  - PTE marked invalid, Priv. Level Violation, Access violation, or does not exist
  - Causes a Fault / Trap
    - » Not an interrupt because synchronous to instruction execution
  - May occur on instruction fetch or data access
  - Protection violations typically terminate the instruction
- 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

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



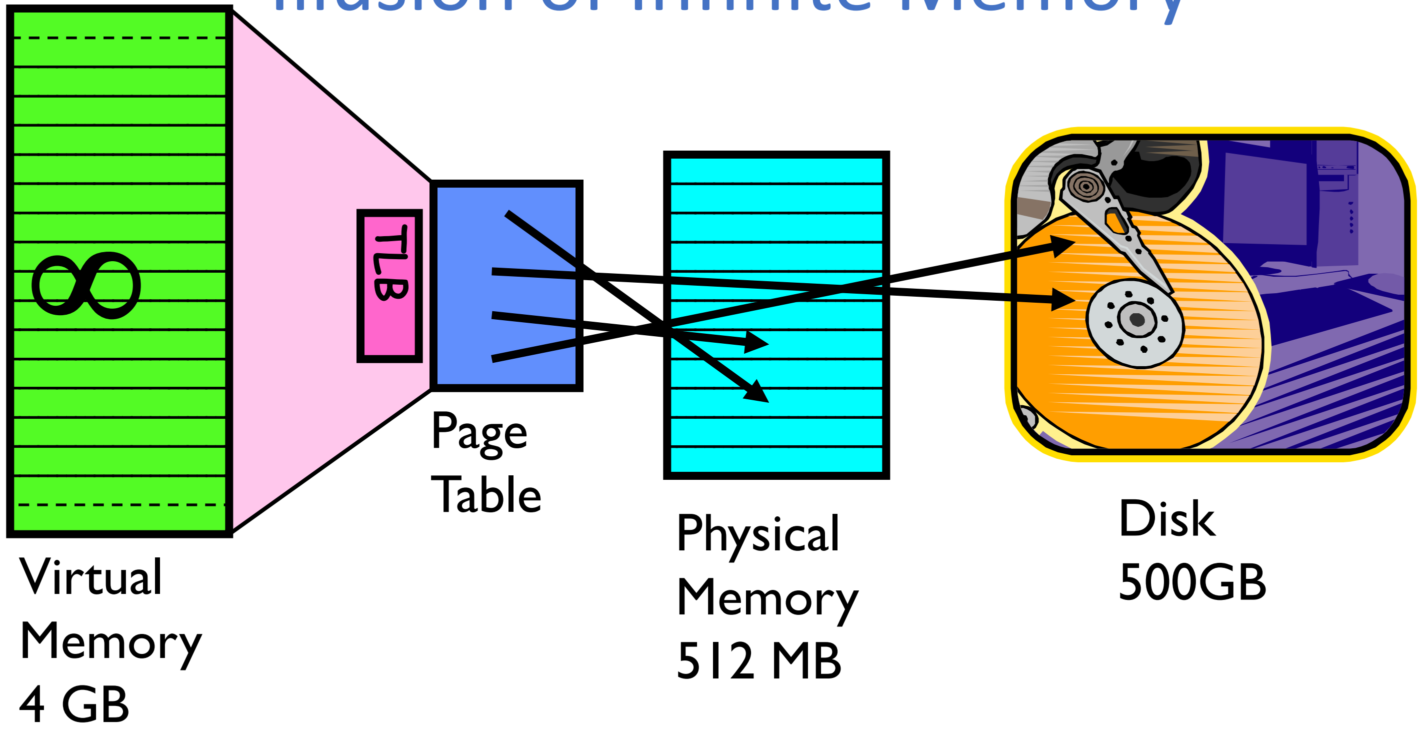
Page Fault ⇒ Demand Paging



Demand Paging as Caching, …

- What “block size”? - 1 page (e.g., 4 KB)
- What “organization” i.e., direct-mapped, set-assoc., fully-associative?
  - Fully associative since arbitrary 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… (more later)
- 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



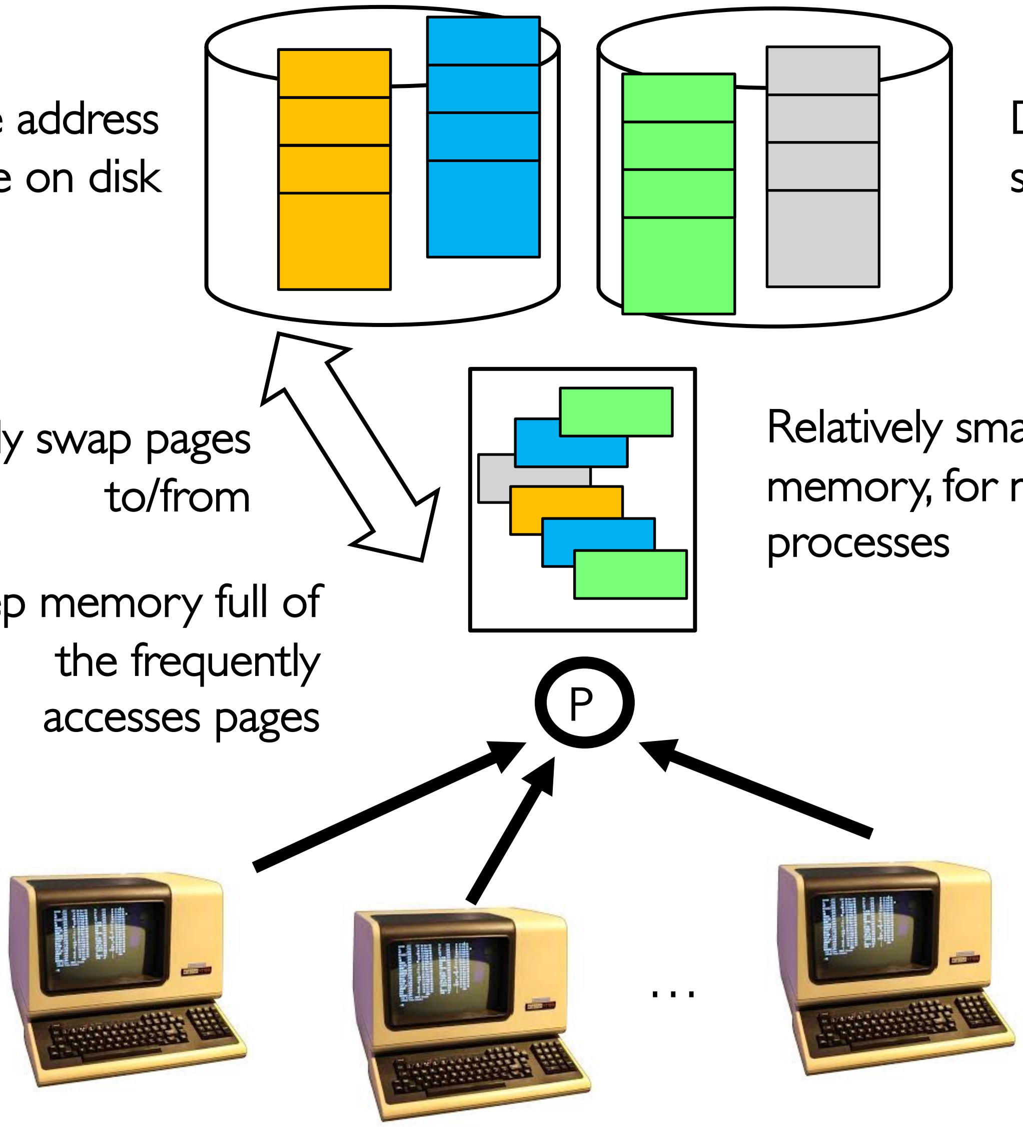
- Disk is larger than physical memory ⇒
  - In-use virtual memory can be bigger than physical memory
  - Combined memory of running processes much larger than physical memory
    - » More programs fit into memory, allowing more concurrency
- Principle: Transparent Level of Indirection (page table)
  - Supports flexible placement of physical data
    - » Data could be on disk or somewhere across network (NSDI’17 InfiniSwap, OSDI’20 AIFM)
  - Variable location of data transparent to user program
    - » Performance issue, not correctness issue

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 ("Dirty=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
    - » 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

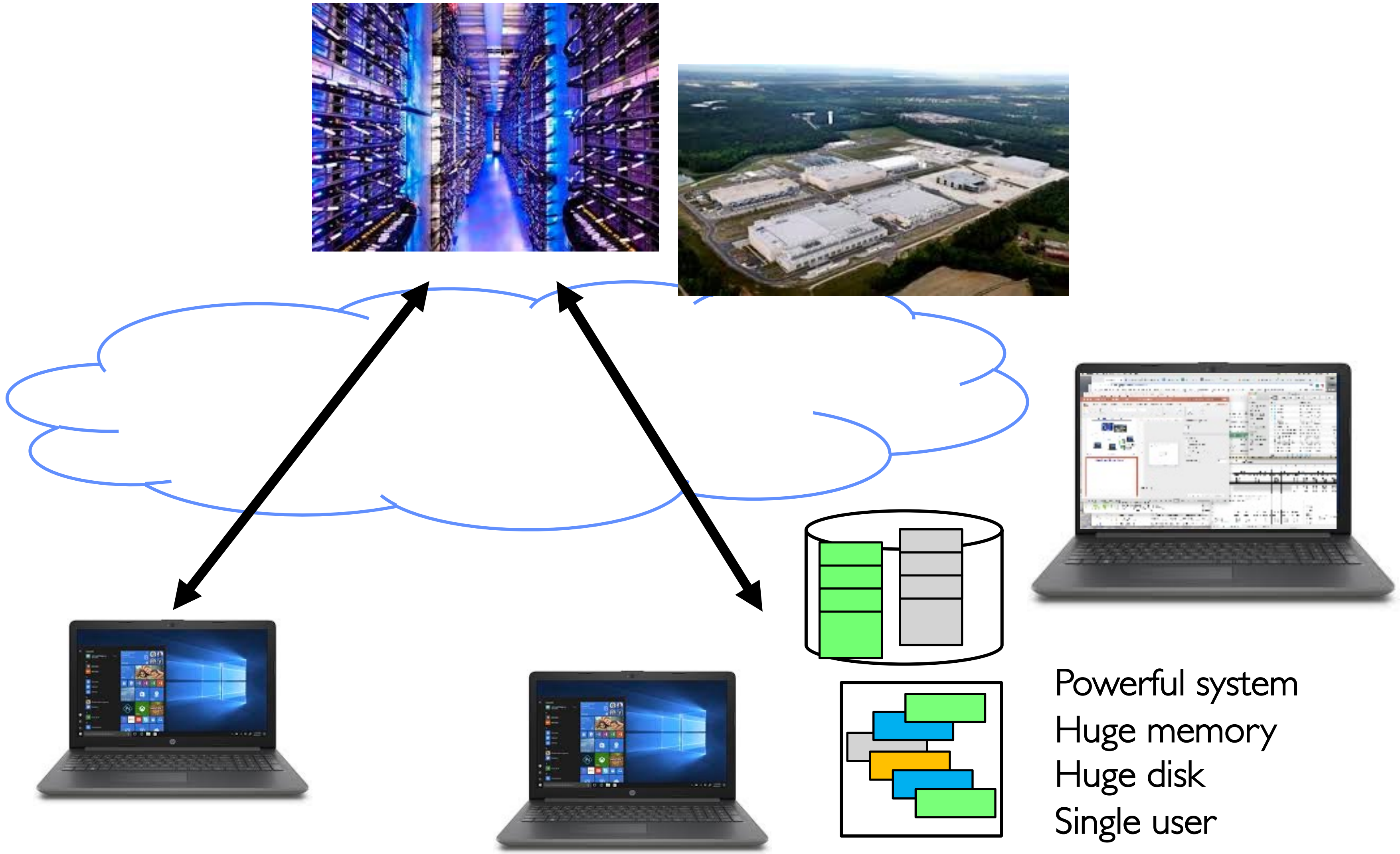
Actively swap pages to/from

Relatively small memory, for many processes

Keep memory full of the frequently accesses pages

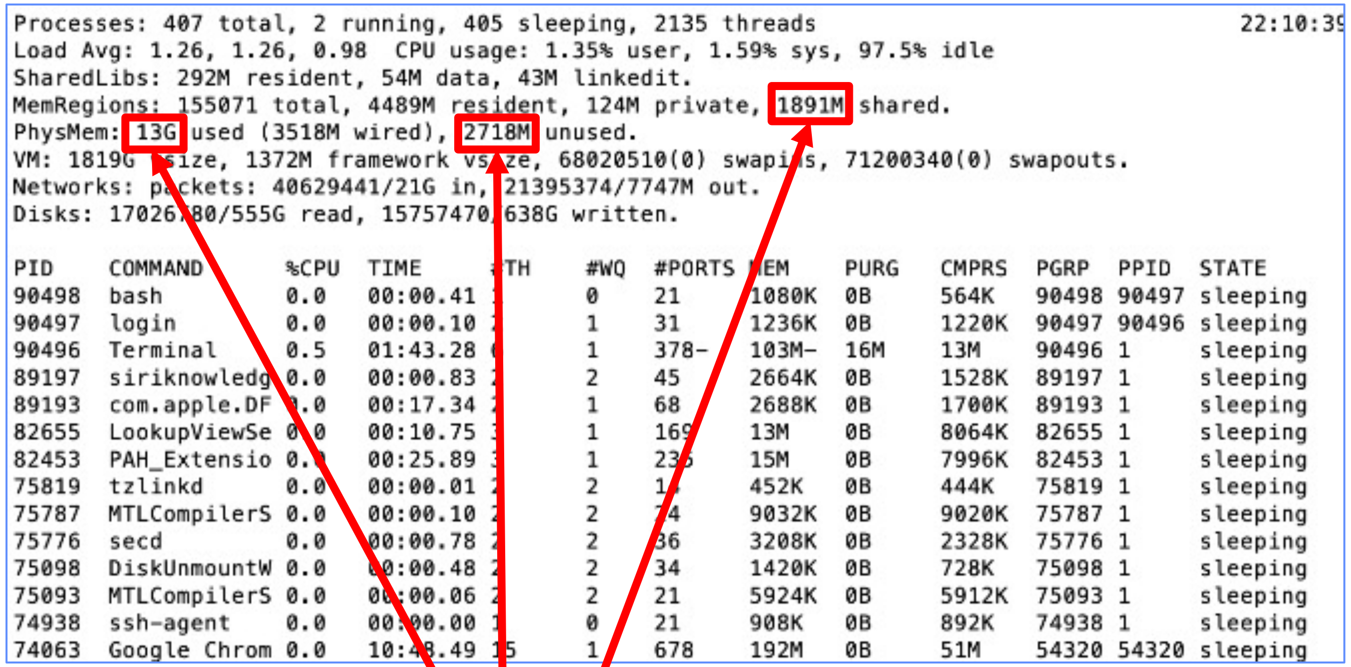
Many clients on dumb terminals running different programs

Very Different Situation Today



Powerful system
Huge memory
Huge disk
Single user

A Picture on one machine



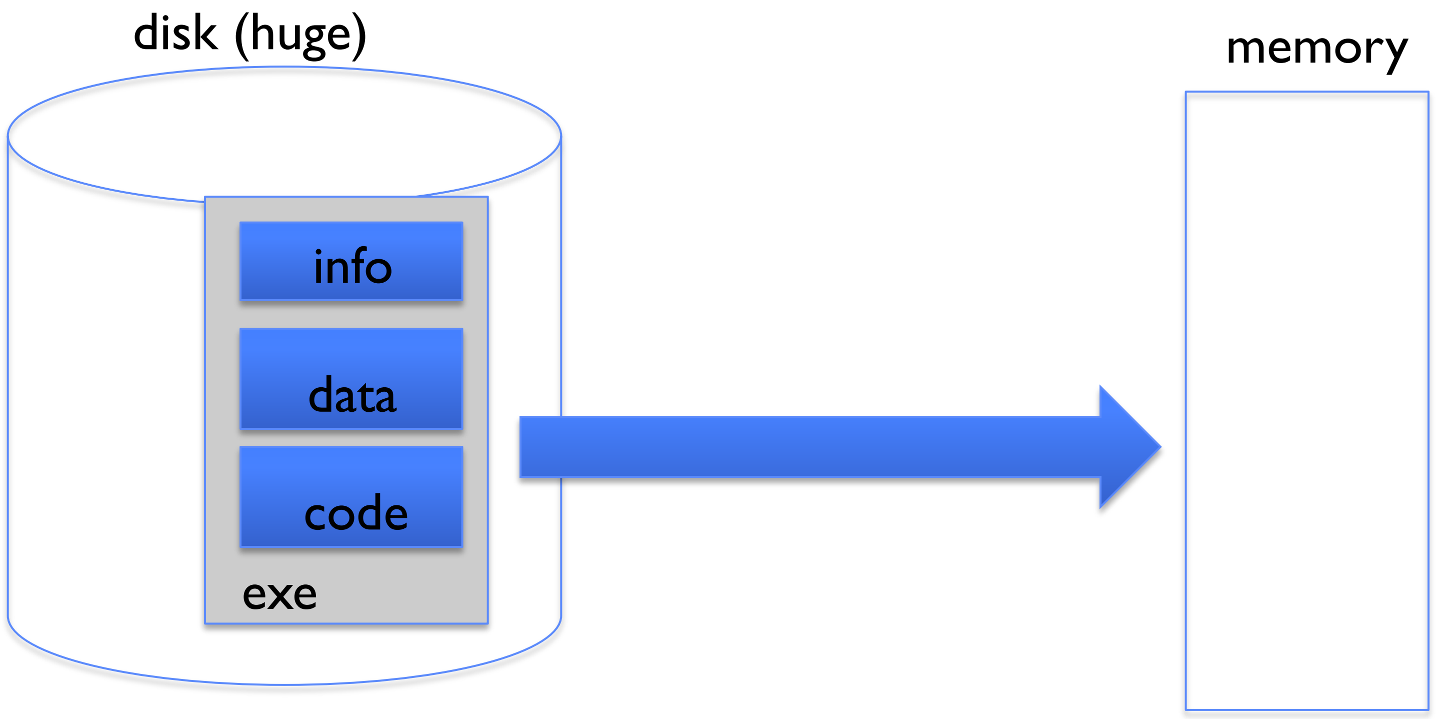
```
Processes: 407 total, 2 running, 405 sleeping, 2135 threads                    22:10:39
Load Avg: 1.26, 1.26, 0.98  CPU usage: 1.35% user, 1.59% sys, 97.5% idle
SharedLibs: 292M resident, 54M data, 43M linkedit.
MemRegions: 155071 total, 4489M resident, 124M private, 1891M shared.
PhysMem: 13G used (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: 17026[illegible]80/555G 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 80% used
- A lot of it is shared 1.9 GB

Many Uses of Virtual Memory and "Demand Paging" …

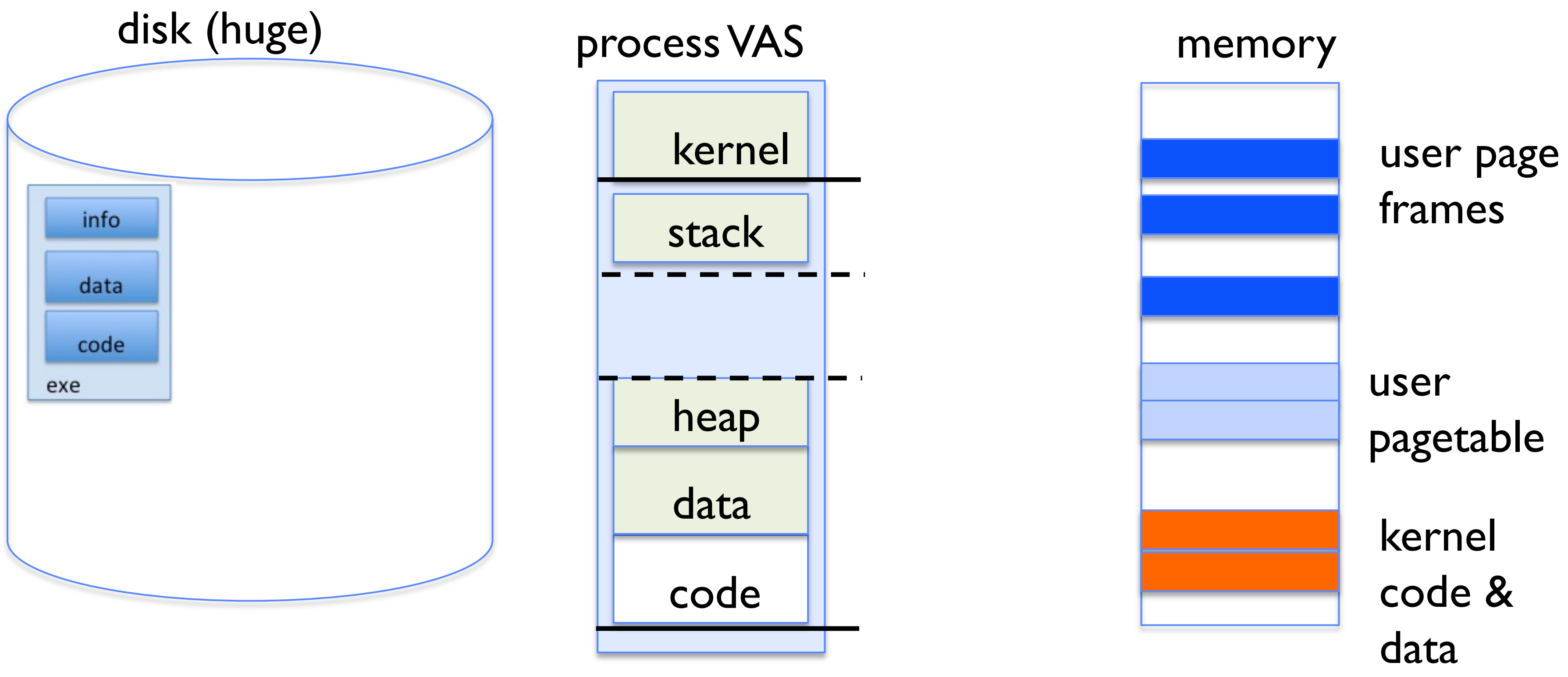
- Extend the stack
  - Allocate a page and zero it
- Extend the heap
- Process Fork
  - Create a copy of the page table
  - Entries refer to parent pages – NO-WRITE
  - Shared read-only pages remain shared
  - Copy page on write
- Exec
  - Only bring in parts of the binary in active use
  - Do this on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

Classic: Loading an Executable into Memory



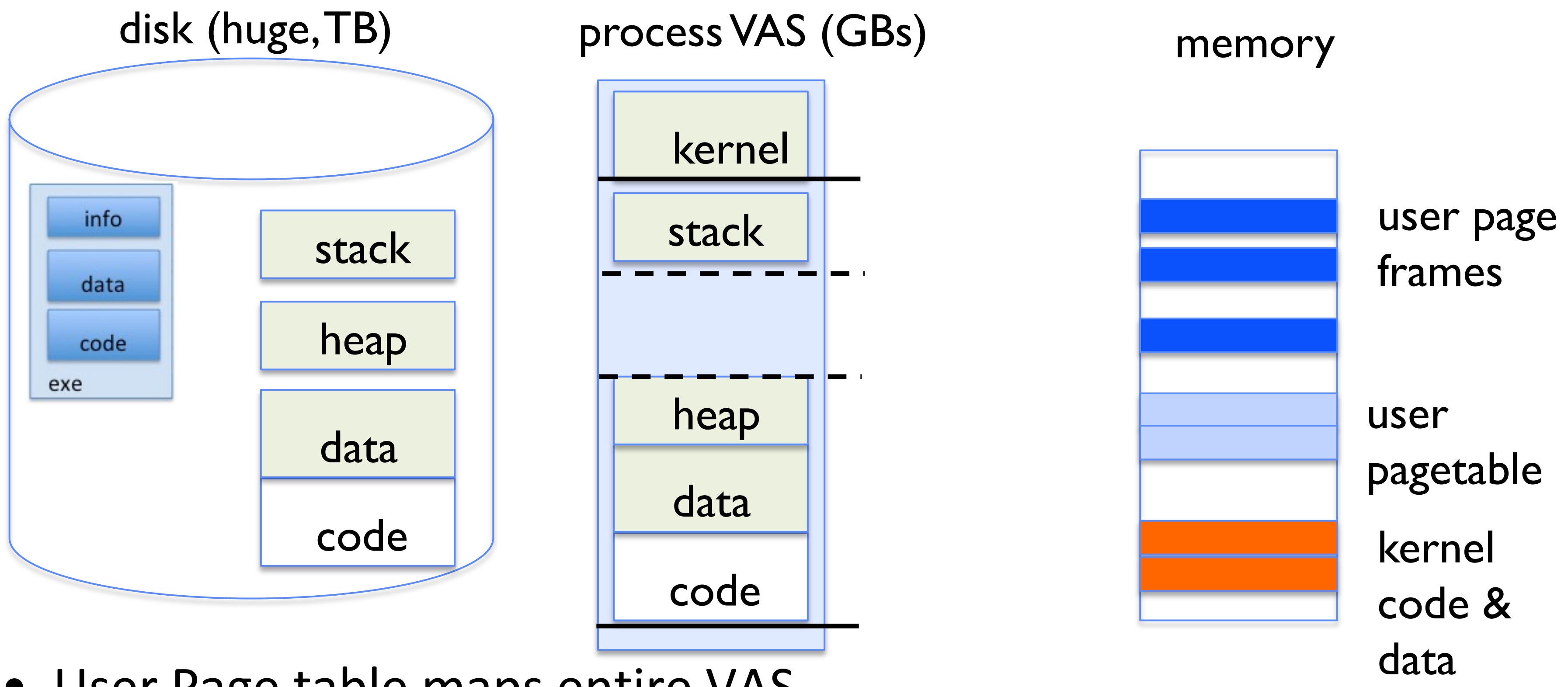
- .exe
  - lives on disk in the file system
  - contains contents of code & data segments, relocation entries and symbols
  - OS loads it into memory, initializes registers (and initial stack pointer)
  - program sets up stack and heap upon initialization

Create Virtual Address Space of the Process



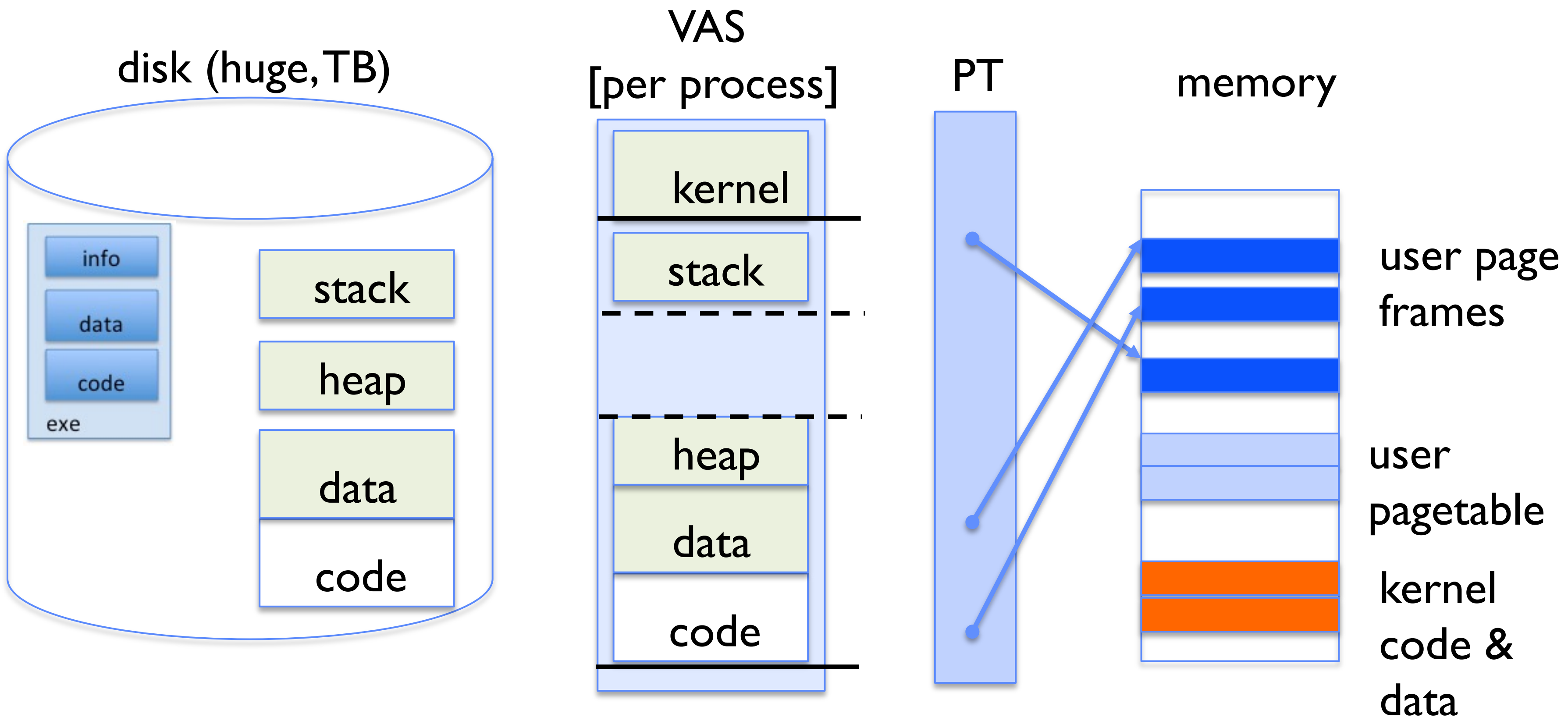
- Utilized pages in the VAS are backed by a page block on disk
  - Called the backing store or swap file
  - Typically, in an optimized block store, but can think of it like a file

Create Virtual Address Space of the Process



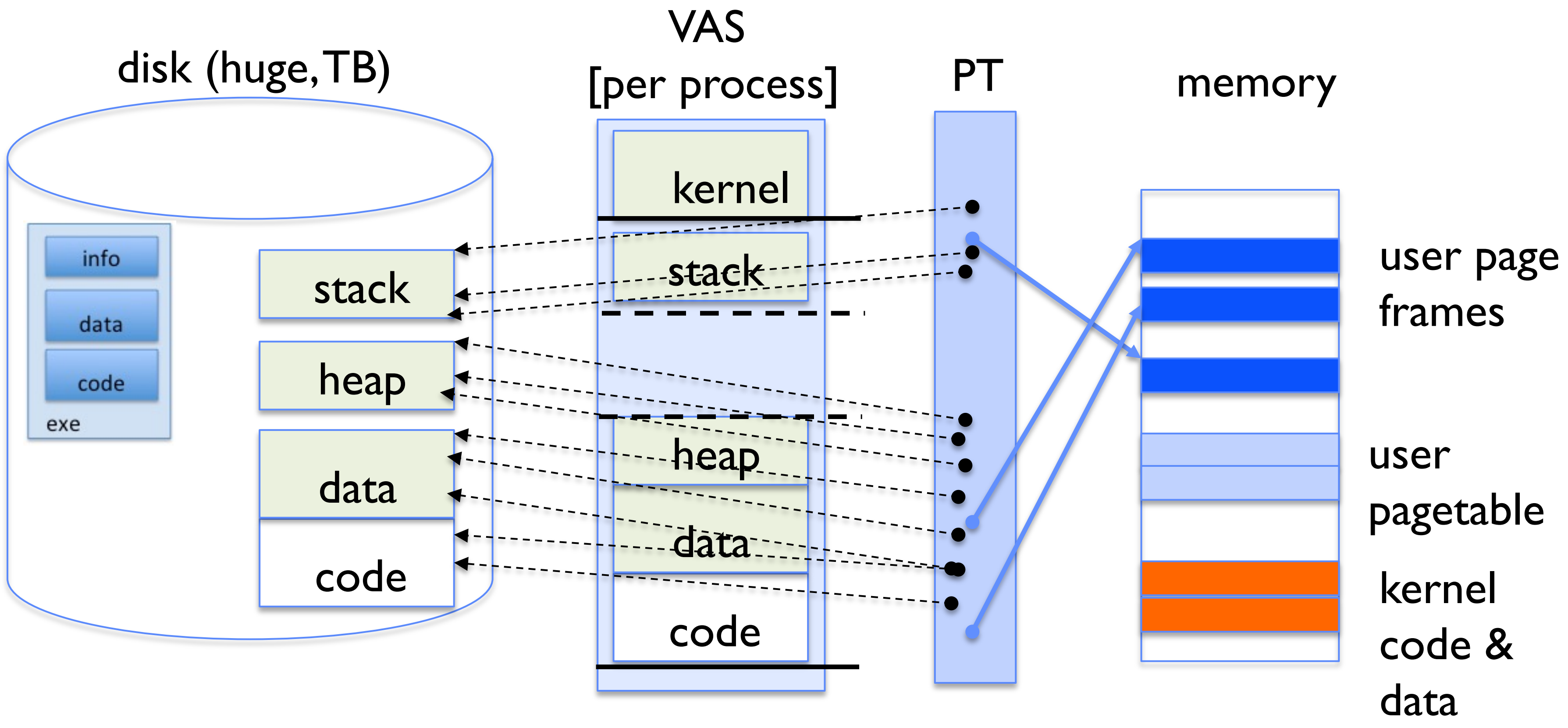
- User Page table maps entire VAS
- All the utilized regions are backed on disk
  - swapped into and out of memory as needed
- For *every* process

Create Virtual Address Space of the Process



- User Page table maps entire VAS
  - Resident pages mapped to the frame in memory they occupy
  - The portion of page table that the HW needs to access must be resident in memory

Provide Backing Store for VAS

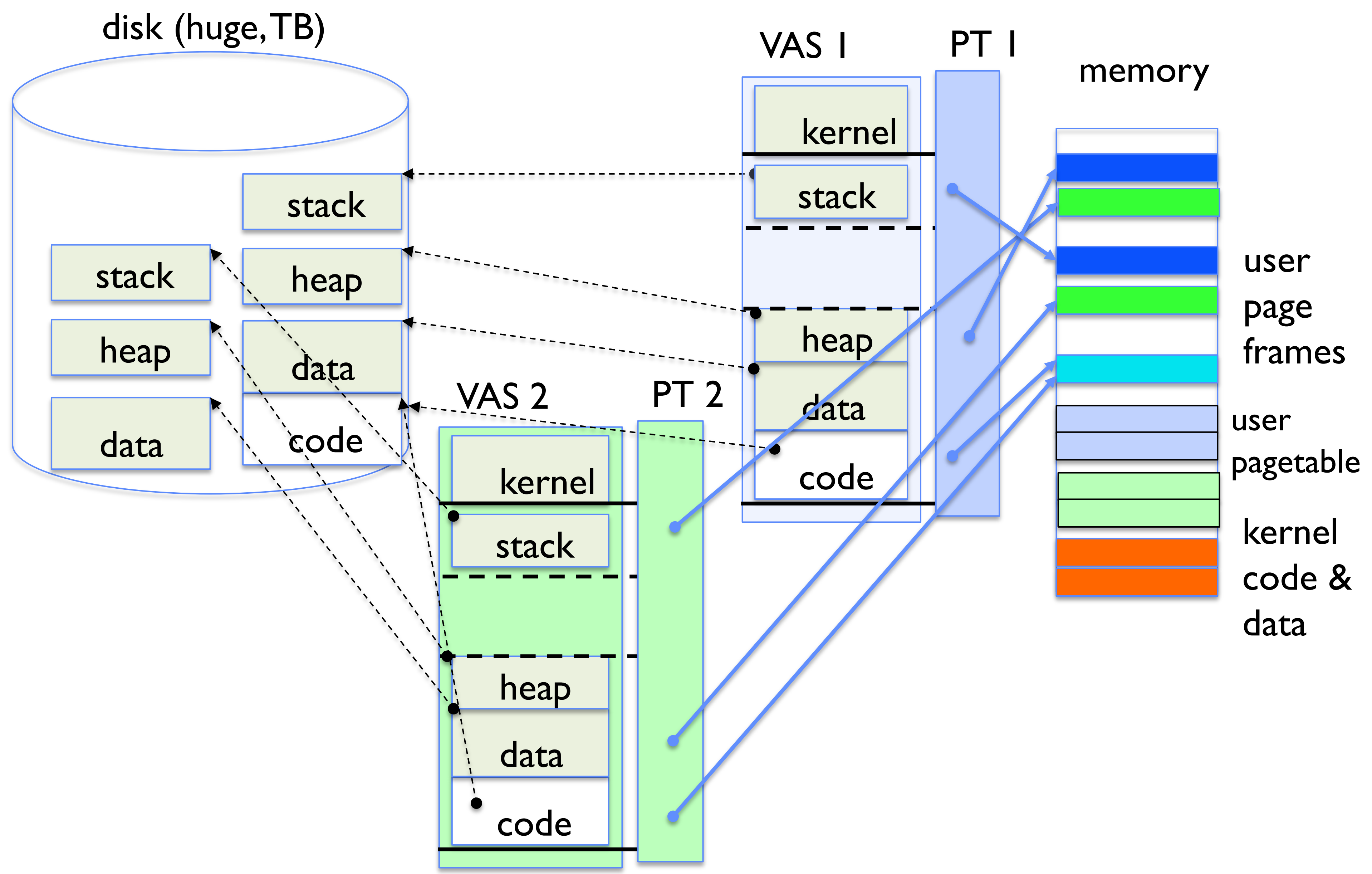


- User Page table maps entire VAS
- Resident pages mapped to the frame in memory they occupy
- For all other pages, OS must record where to find them on disk

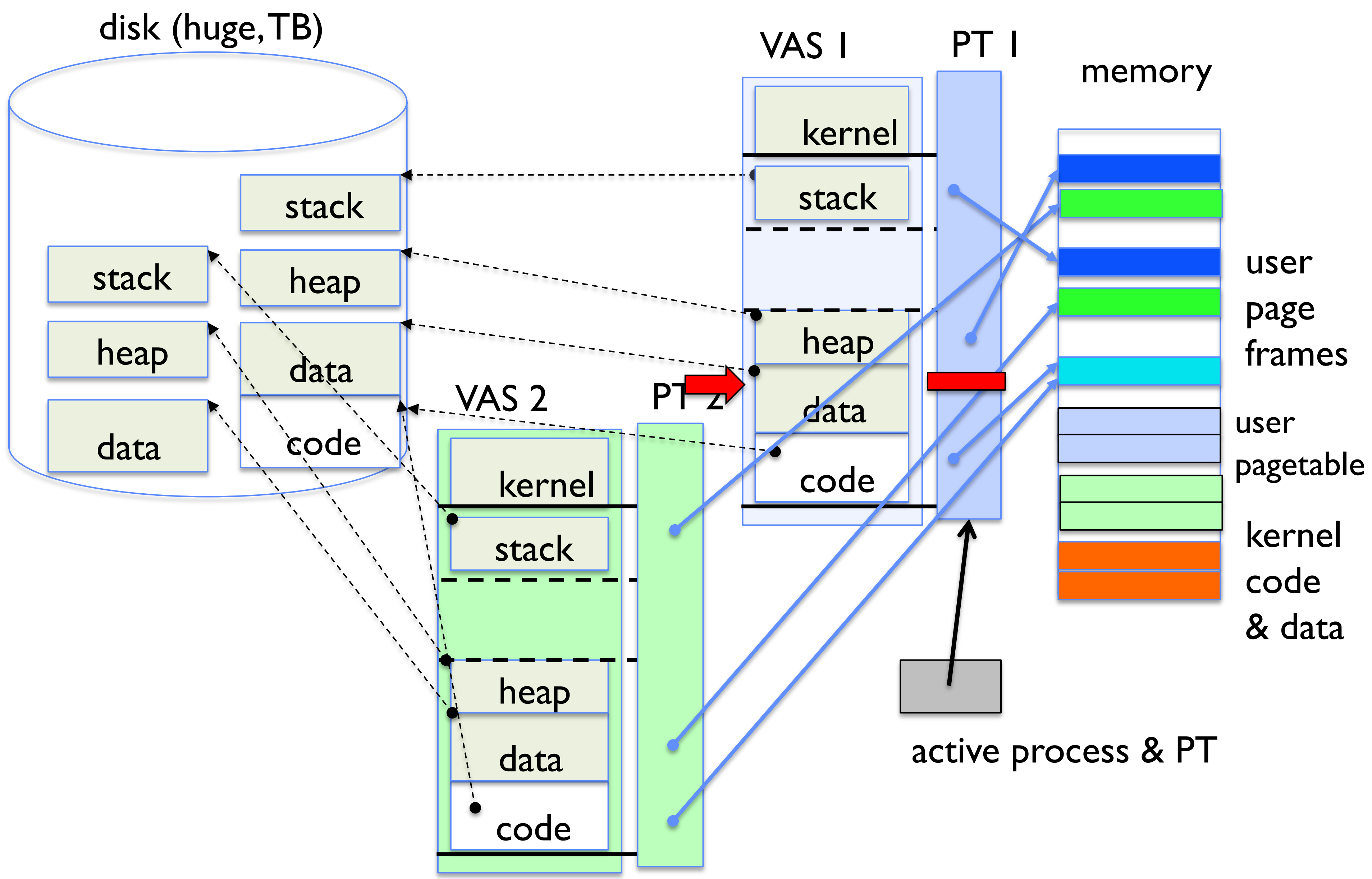
What Data Structure Maps Non-Resident Pages to Disk?

- `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

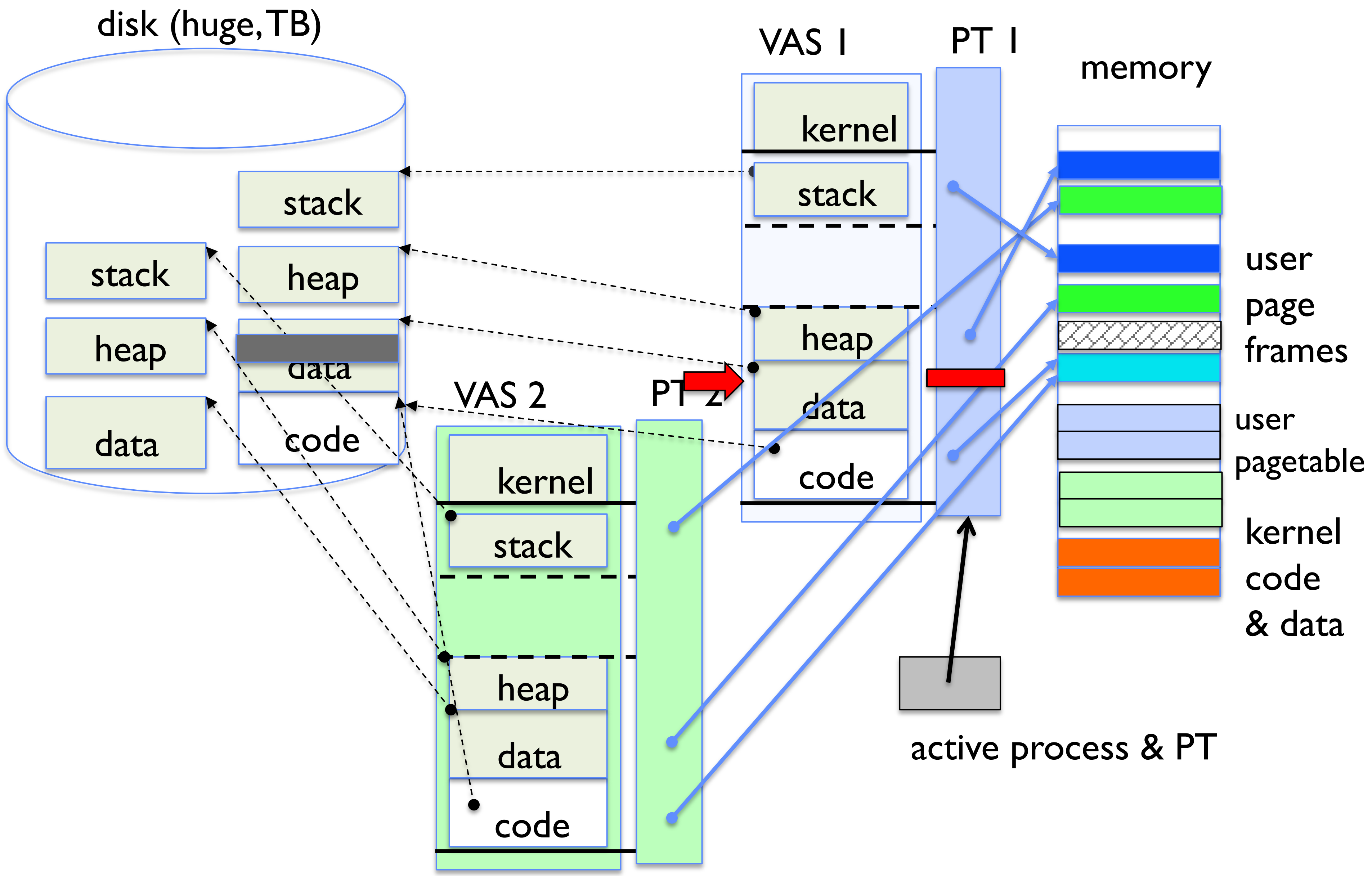
Provide Backing Store for VAS



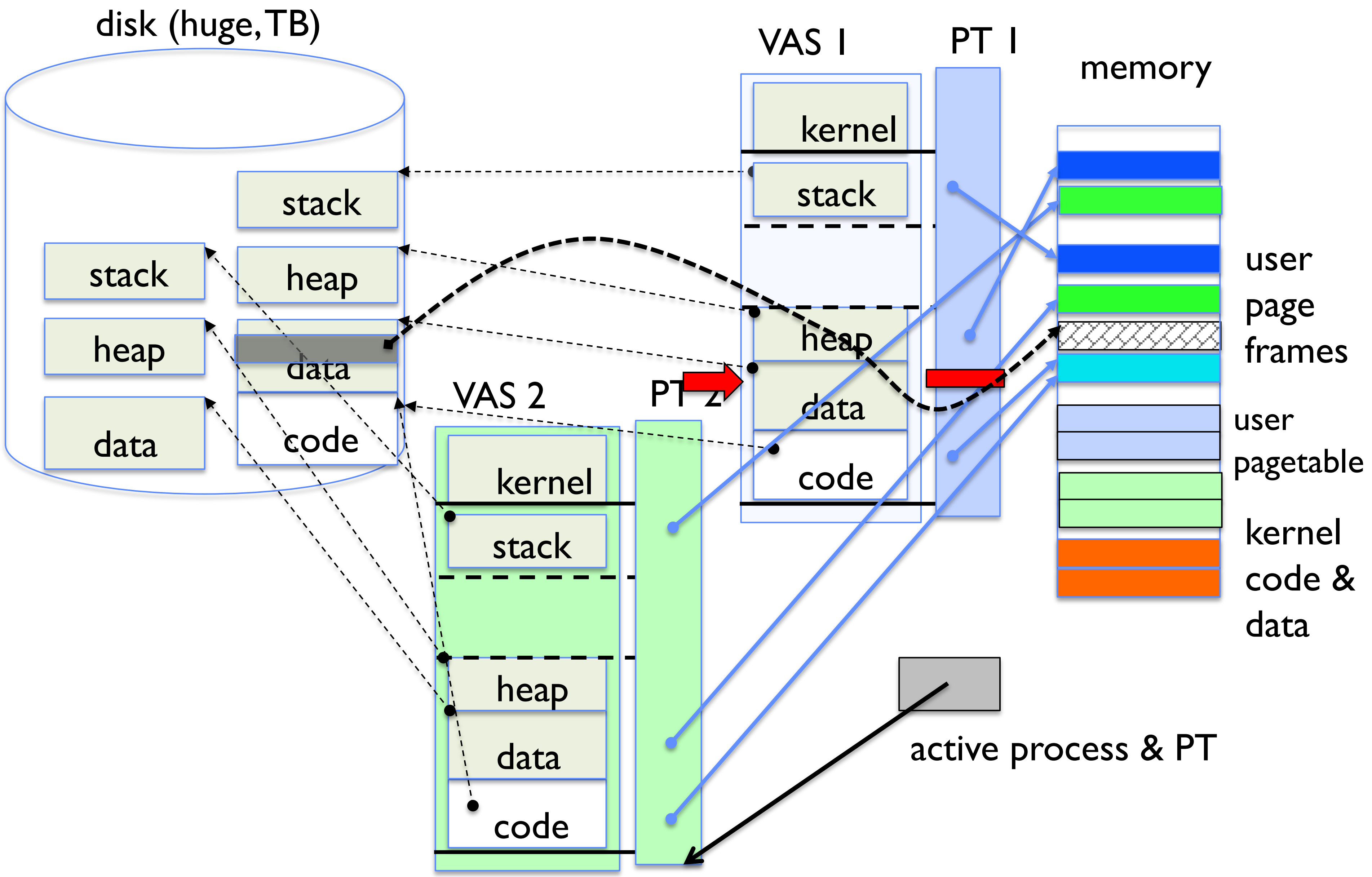
On Page Fault …



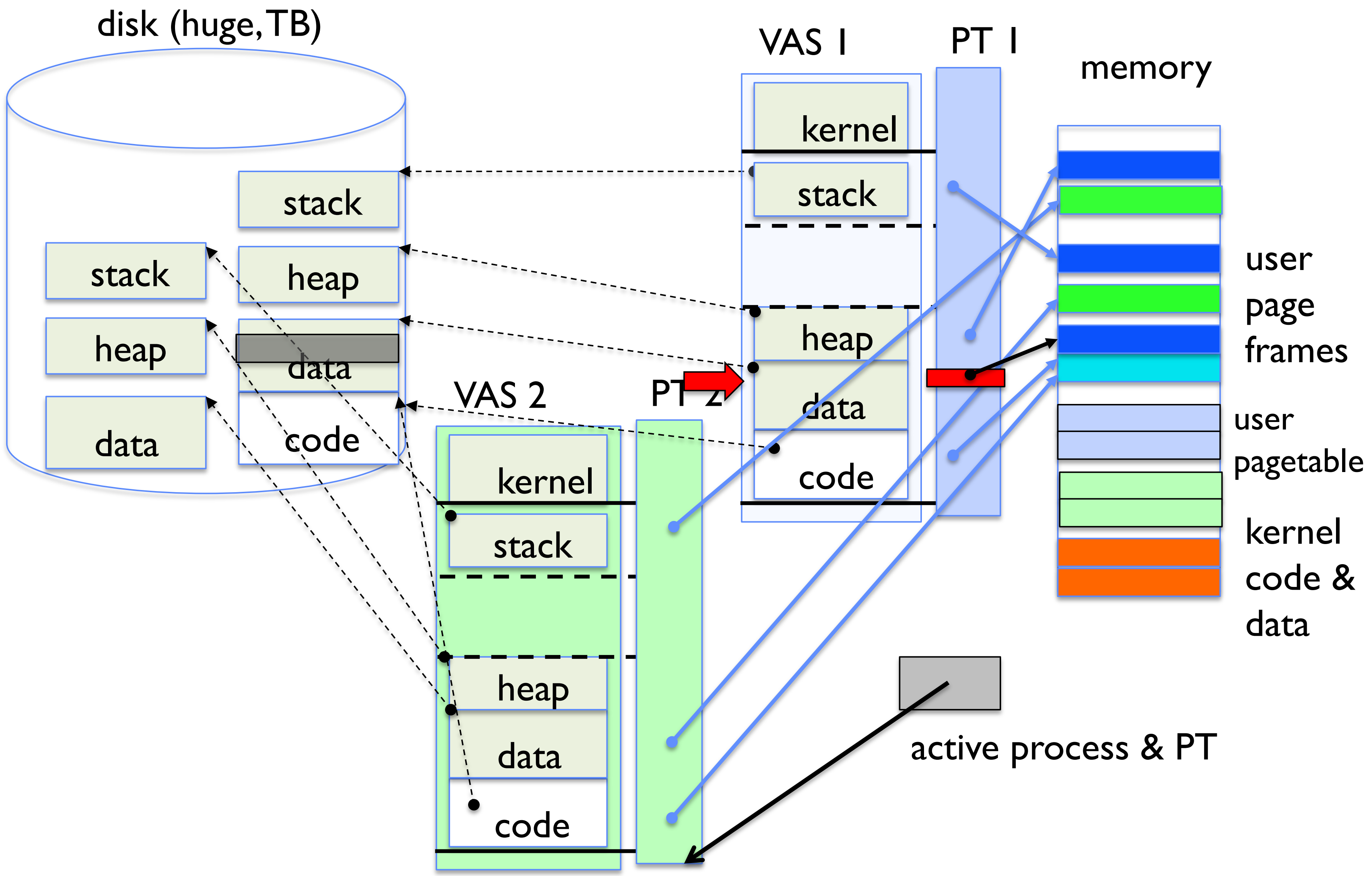
On Page Fault … Find & Start Load



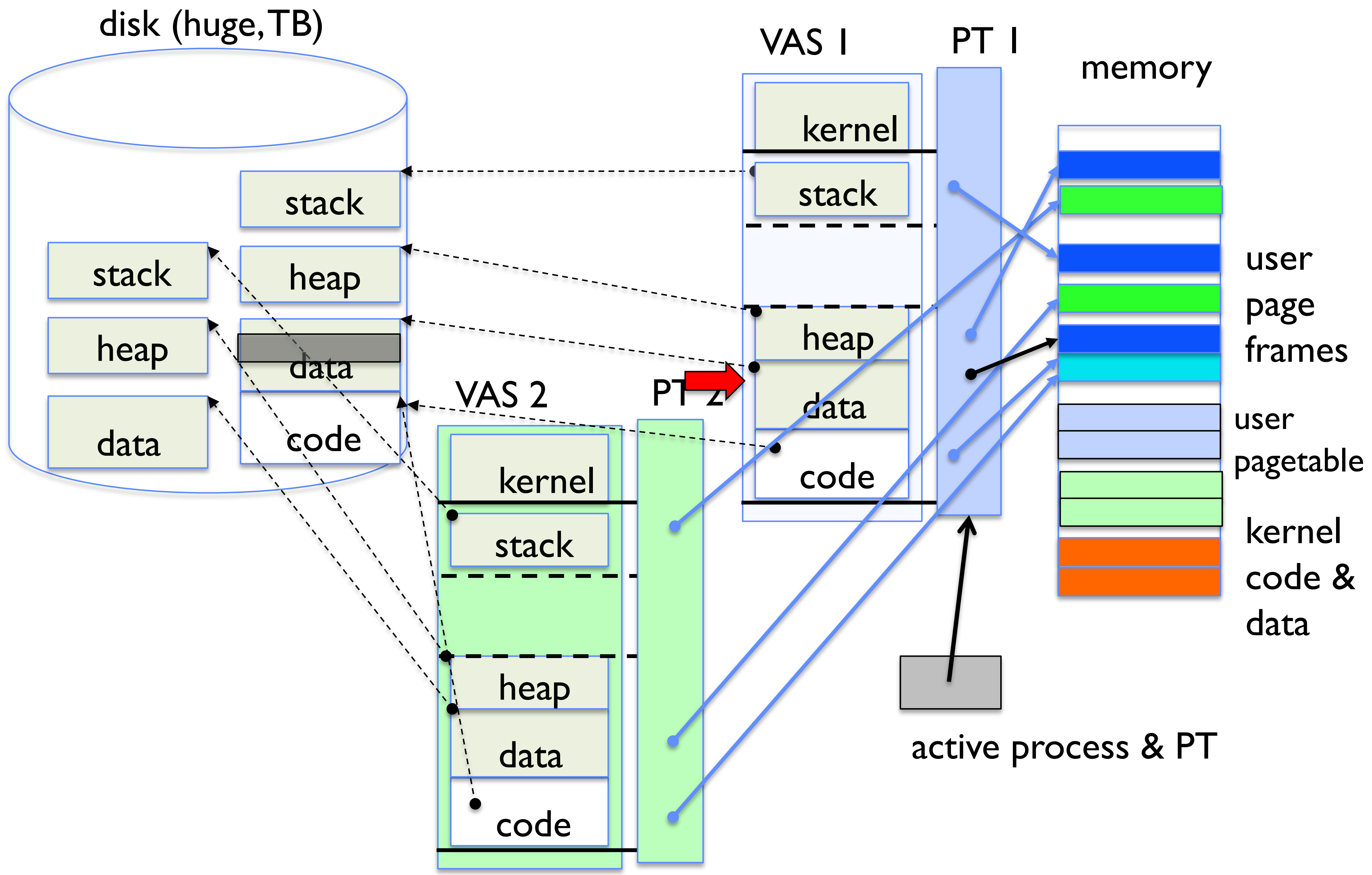
On page Fault … schedule other Process or Thread



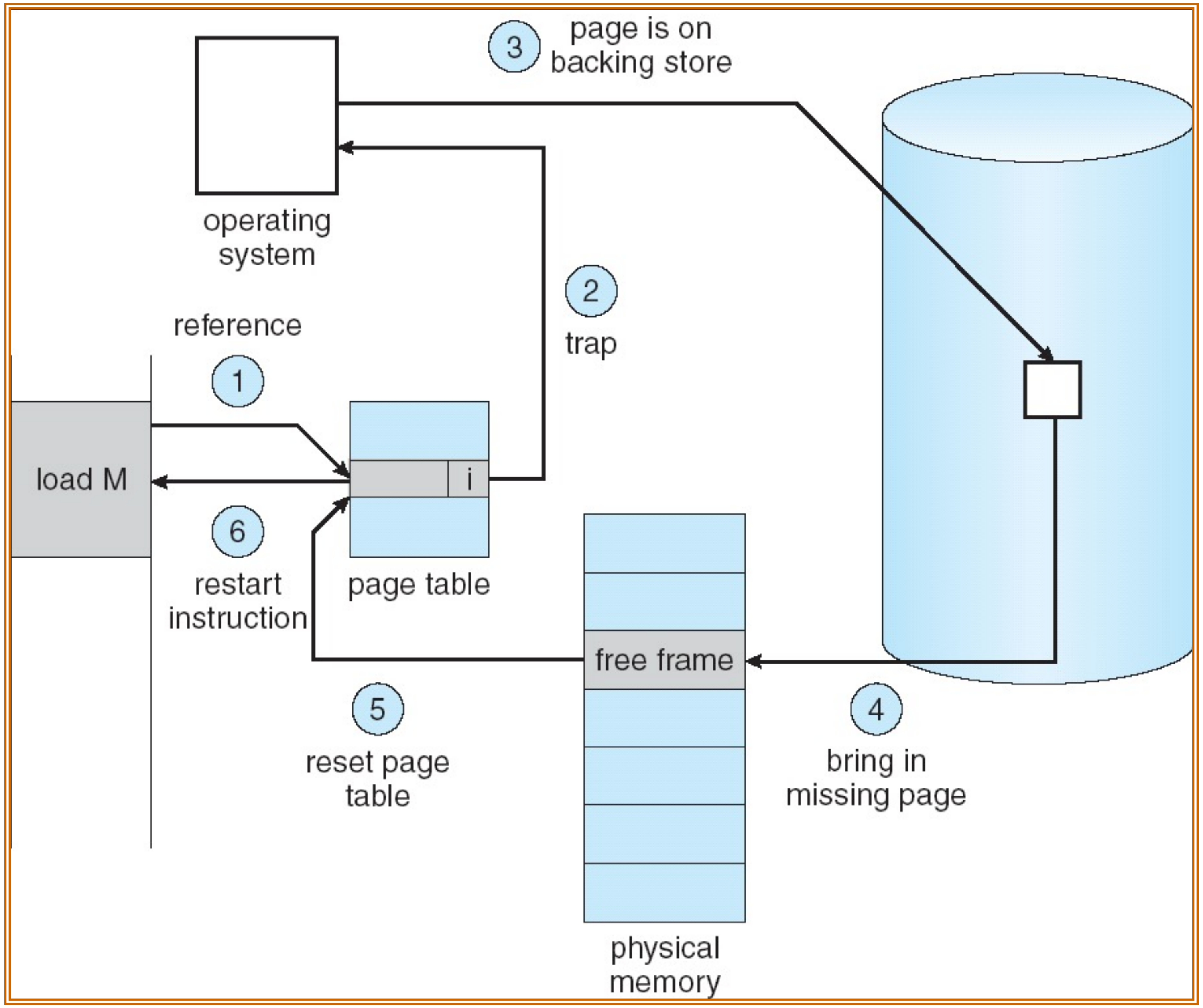
On Page Fault … Update PTE



Eventually reschedule faulting thread



Summary: Steps in Handling a Page Fault



Some questions we need to answer!

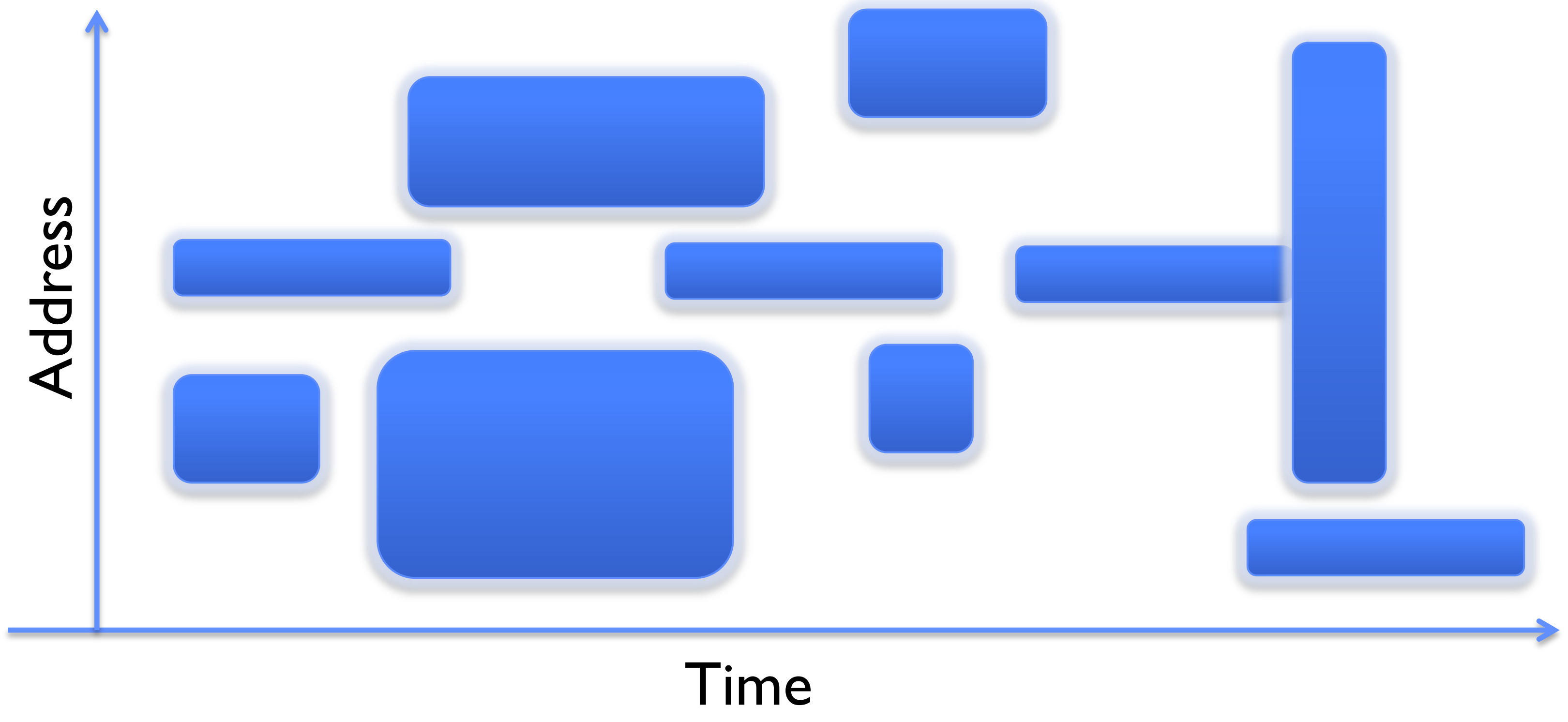
- During a page fault, where does the OS get a free frame?
  - Keeps a free list
  - Unix runs a “reaper” if memory gets too full
    - » Schedule dirty pages to be written back on disk
    - » Zero (clean) pages which haven’t been accessed in a while
  - As a last resort, evict a page first

- How can we organize these mechanisms?
  - Work on the replacement policy

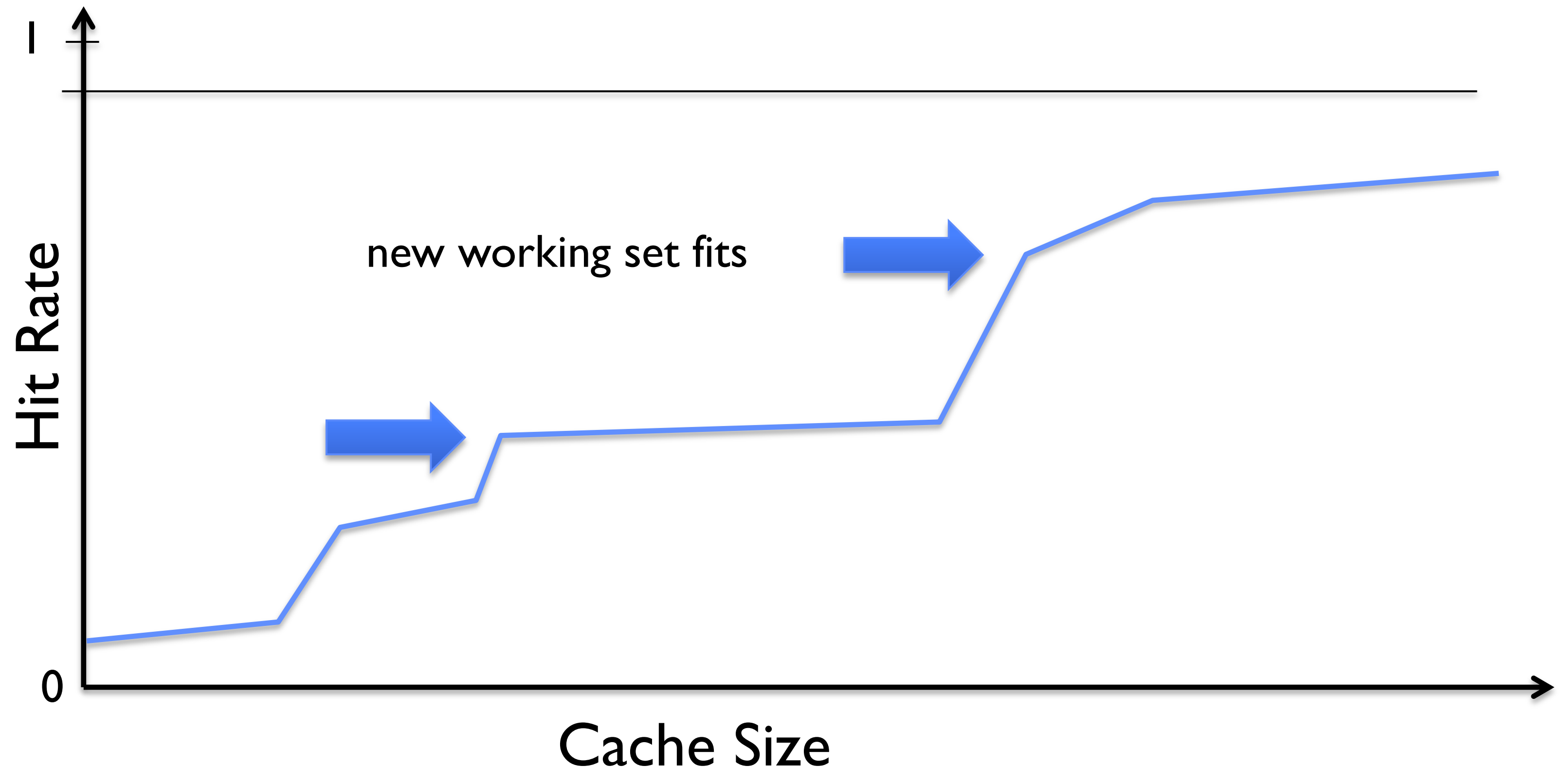
- How many page frames/process?
  - Like thread scheduling, need to “schedule” memory resources:
    - » Utilization? fairness? priority?
  - Allocation of disk paging bandwidth

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

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 (Hit Rate + Mis Rate = 1)
  - EAT = Hit Time + Miss Rate x Miss Penalty (Miss Penalty = Miss Time – Hit Time)
- Example:
  - Memory access time = 200 nanoseconds
  - Average page-fault service time (Miss Penalty) = 8 milliseconds
  - Suppose p = Probability of miss, 1-p = Probably of hit
  - Then, we can compute EAT as follows:

    EAT = 200ns + p x 8 ms

    = 200ns + p x 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 40x !
- What if want slowdown by less than 10%?
  - EAT < 200ns x 1.1 $\Rightarrow$ p < 2.5 x 10^{-6}
  - This is about 1 page fault in 400,000!

What Factors Lead to Misses in Page Cache?

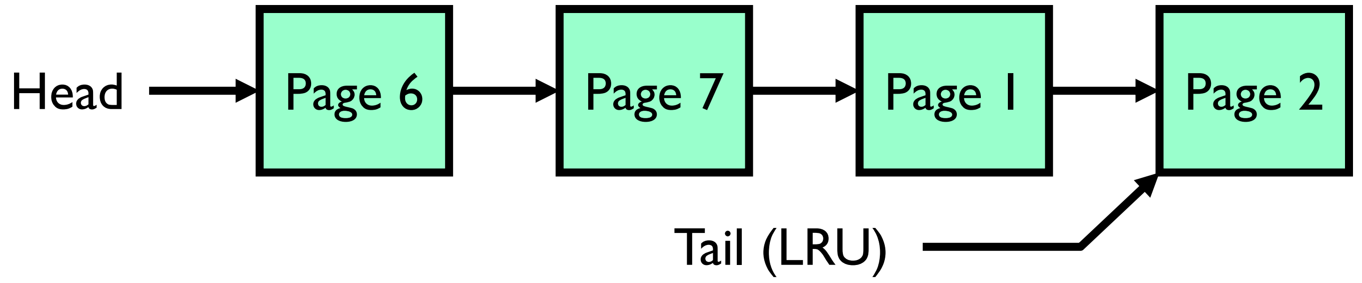
- Compulsory Misses:
  - Pages that have never been paged into memory before
  - How might we remove these misses?
    - Prefetching: loading them into memory before needed
    - Need to predict future somehow! More later
- Capacity Misses:
  - Not enough memory. Must somehow increase available memory 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 page used so that can change position in list…
  - Many instructions for each hardware access
- In practice, people approximate LRU (more later)

Group Discussion

- Topic: replacement policies
  - Can you compare FIFO, RANDOM, MIN and LRU?
  - What are the pros and cons of each approach?

- Discuss in groups of two to three students
  - Each group chooses a leader to summarize the discussion
  - In your group discussion, please do not dominate the discussion, and give everyone a chance to speak

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: 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

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
  - 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: 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!
- 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):

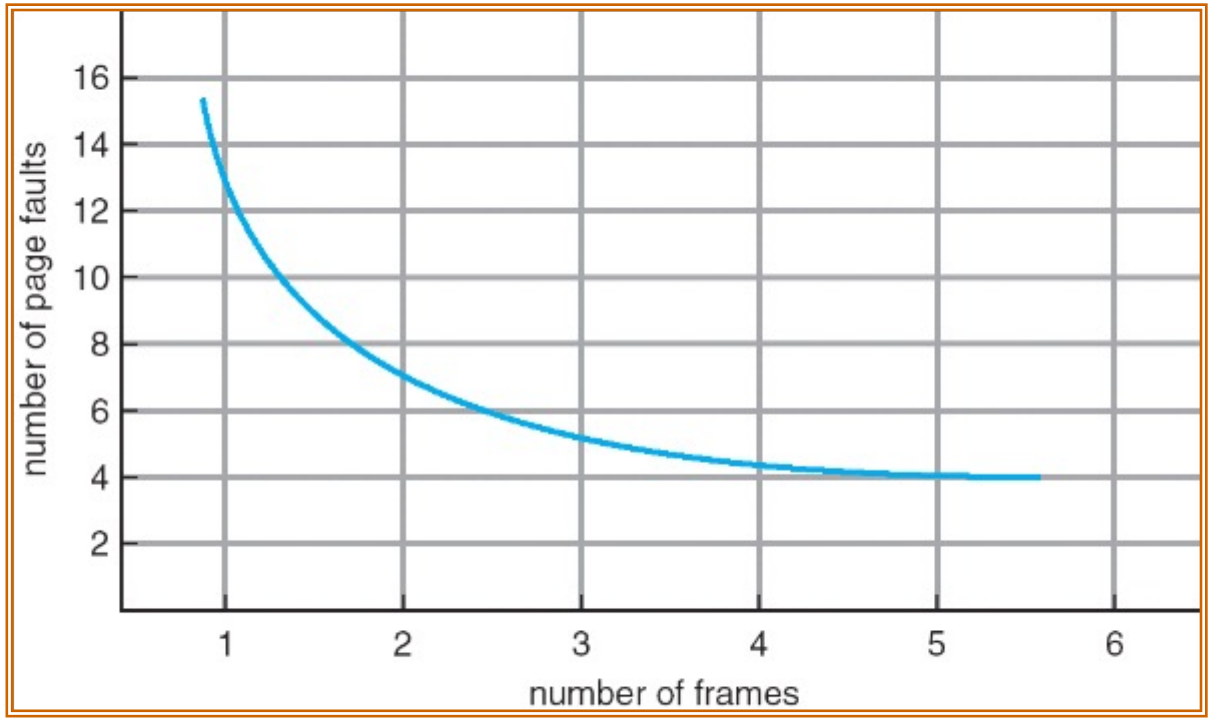
Ref: Page:	A	B	C	D	A	B	C	D	A	B	C	D
1	A			D			C			B		
2		B			A			D			C	
3			C			B			A			D

  – Every reference is a page fault!

- 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								

Graph of Page Faults Versus The Number of Frames



- One desirable property: When you add memory the miss rate drops (stack property)
  - Does this always happen?
  - Seems like it should, right?

Group Discussion

- One desirable property: When you add memory the miss rate drops (stack property)
  - Does this always happen?
  - Seems like it should, right?

- Topic: Bélády’s anomaly
  - Does LRU and MIN have this property?
    - If so, can you prove it?
    - If not, can you give an example?

- Discuss in groups of two to three students
  - Each group chooses a leader to summarize the discussion
  - In your group discussion, please do not dominate the discussion, and give everyone a chance to speak

- Answer: Yes for LRU and MIN
  - Contents of memory with X pages are a subset of contents with X+1 pages

Group Discussion

- One desirable property: When you add memory the miss rate drops (stack property)
  - Does this always happen?
  - Seems like it should, right?

- Topic: Bélády’s anomaly
  - Does FIFO have this property?
    - If so, can you prove it?
    - If not, can you give an example?

- Discuss in groups of two to three students
  - Each group chooses a leader to summarize the discussion
  - In your group discussion, please do not dominate the discussion, and give everyone a chance to speak

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 Bélády'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	

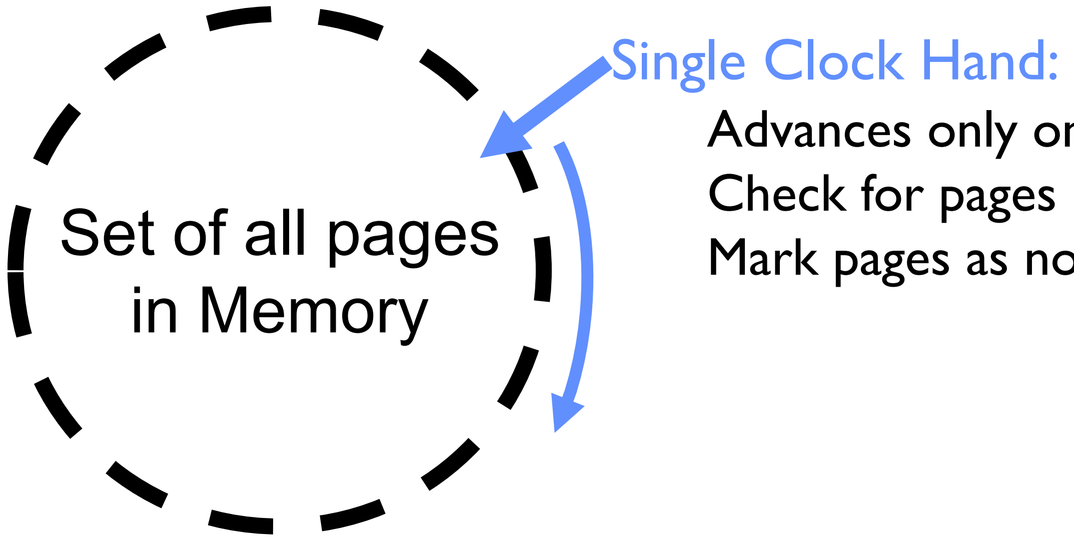
9 page faults

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		

10 page faults!

- 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

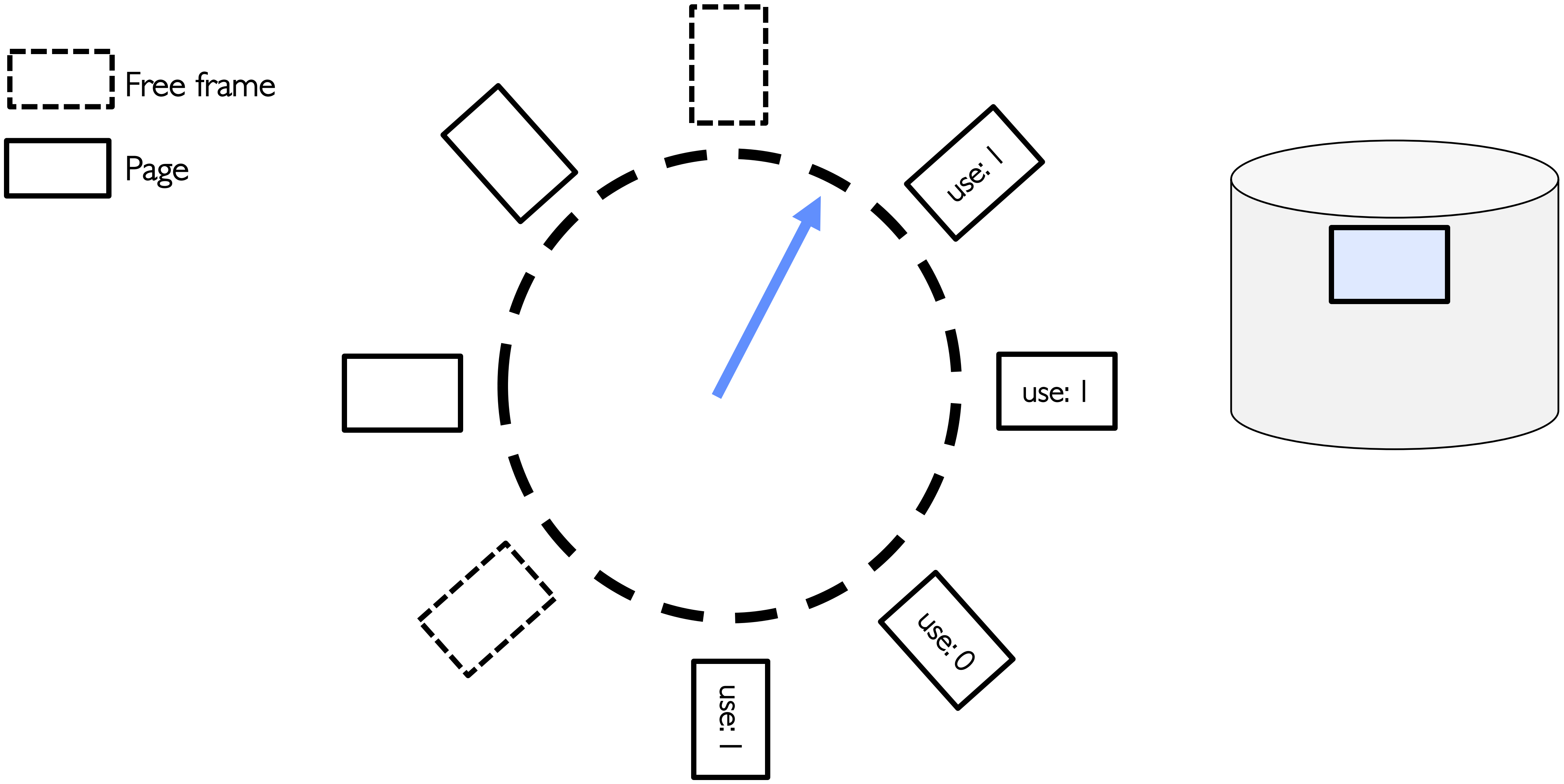
Approximating LRU: Clock Algorithm



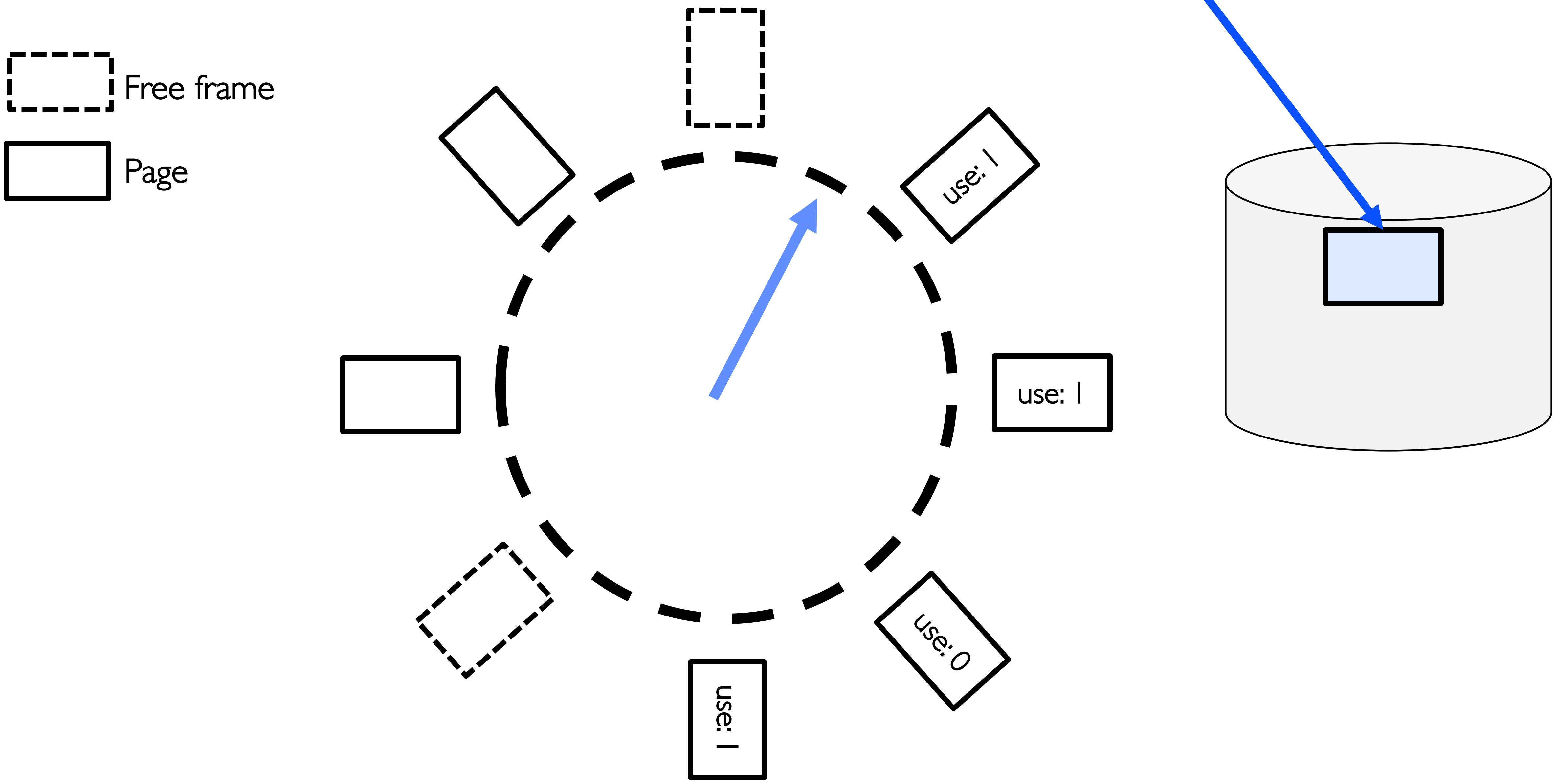
Advances only on page fault!
Check for pages not used recently
Mark pages as not used recently

- 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 (called "accessed" in Intel architecture):
    - » 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

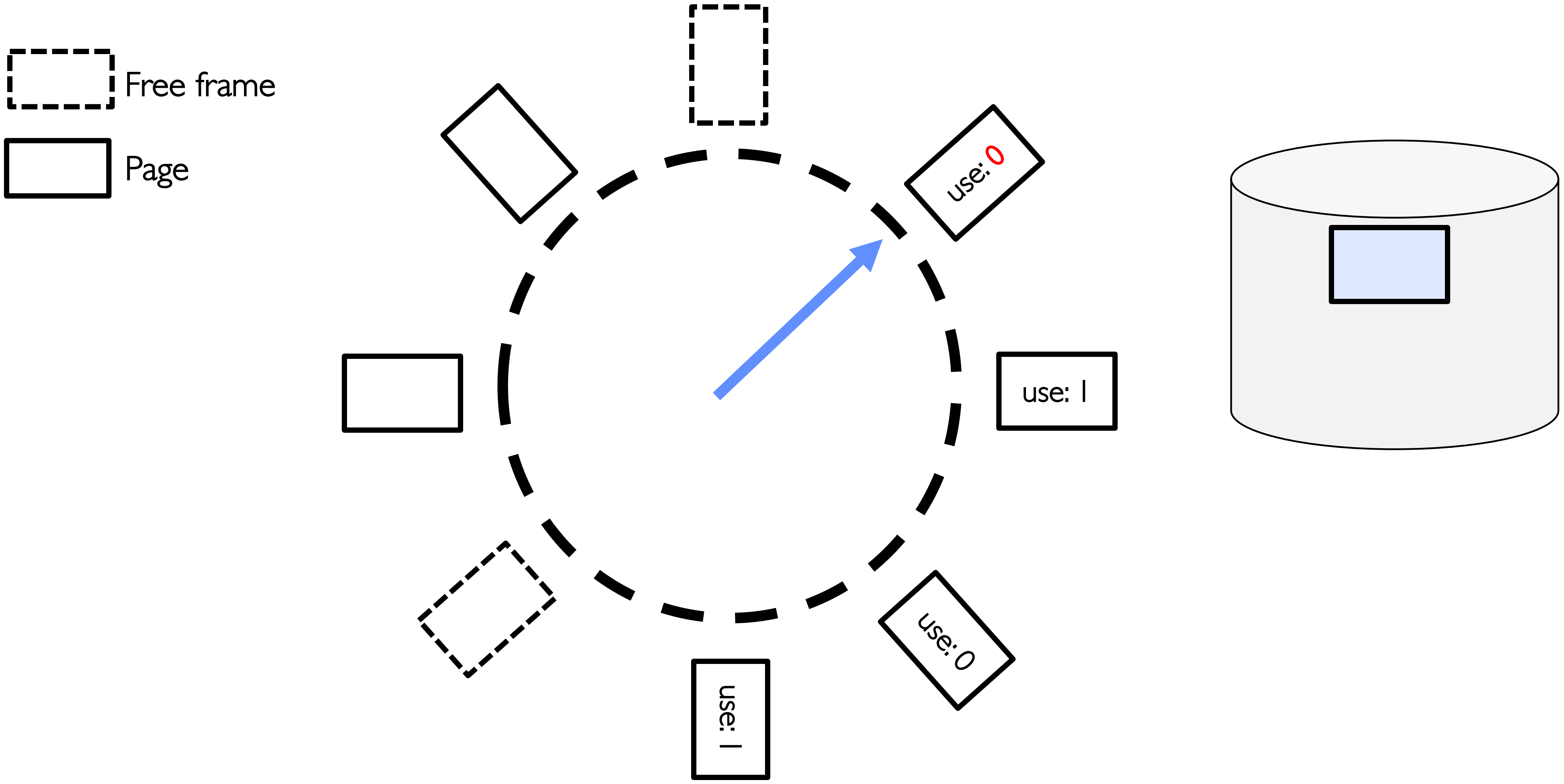
Clock Algorithm Example



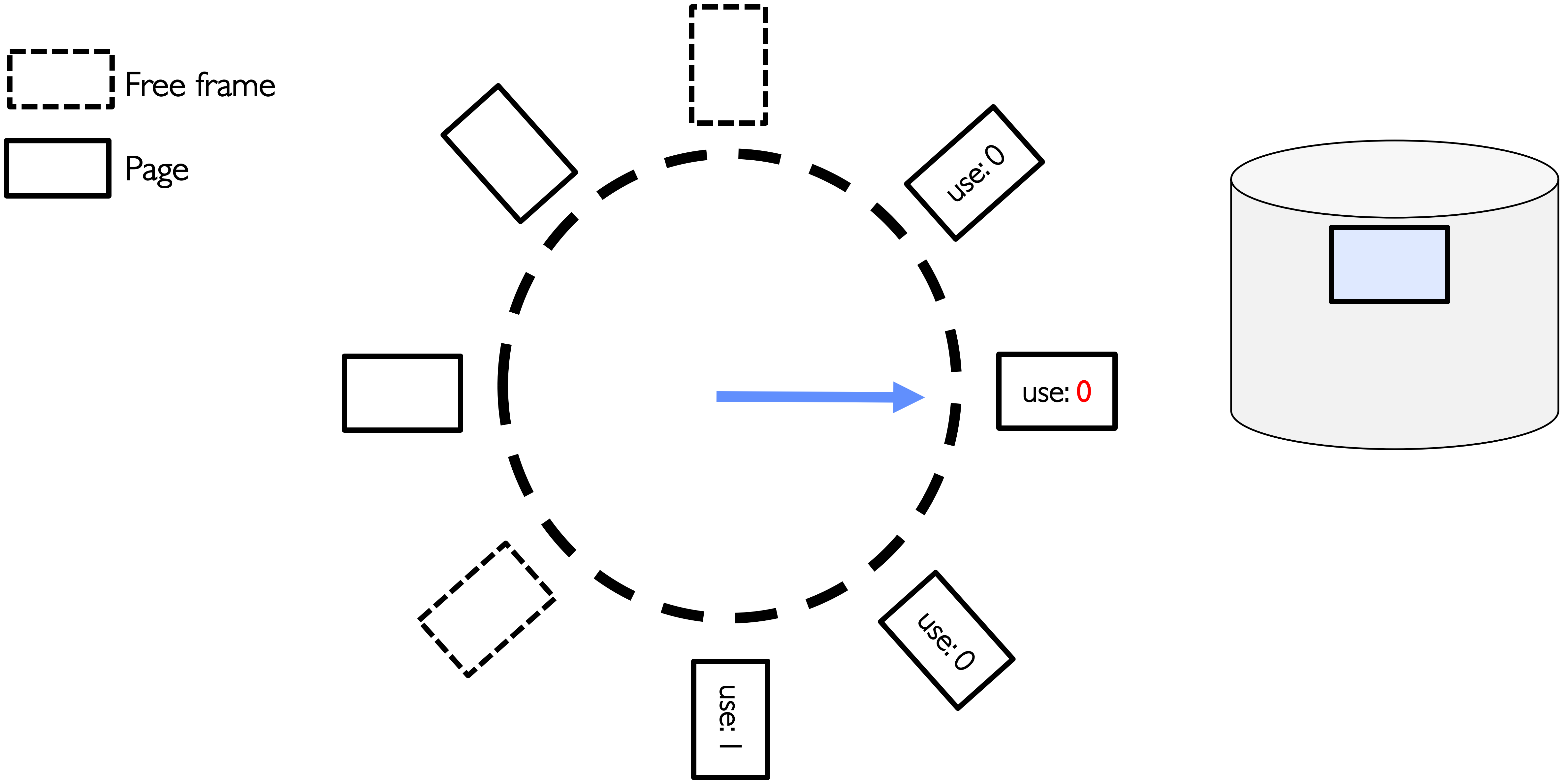
Clock Algorithm Example: Page Fault



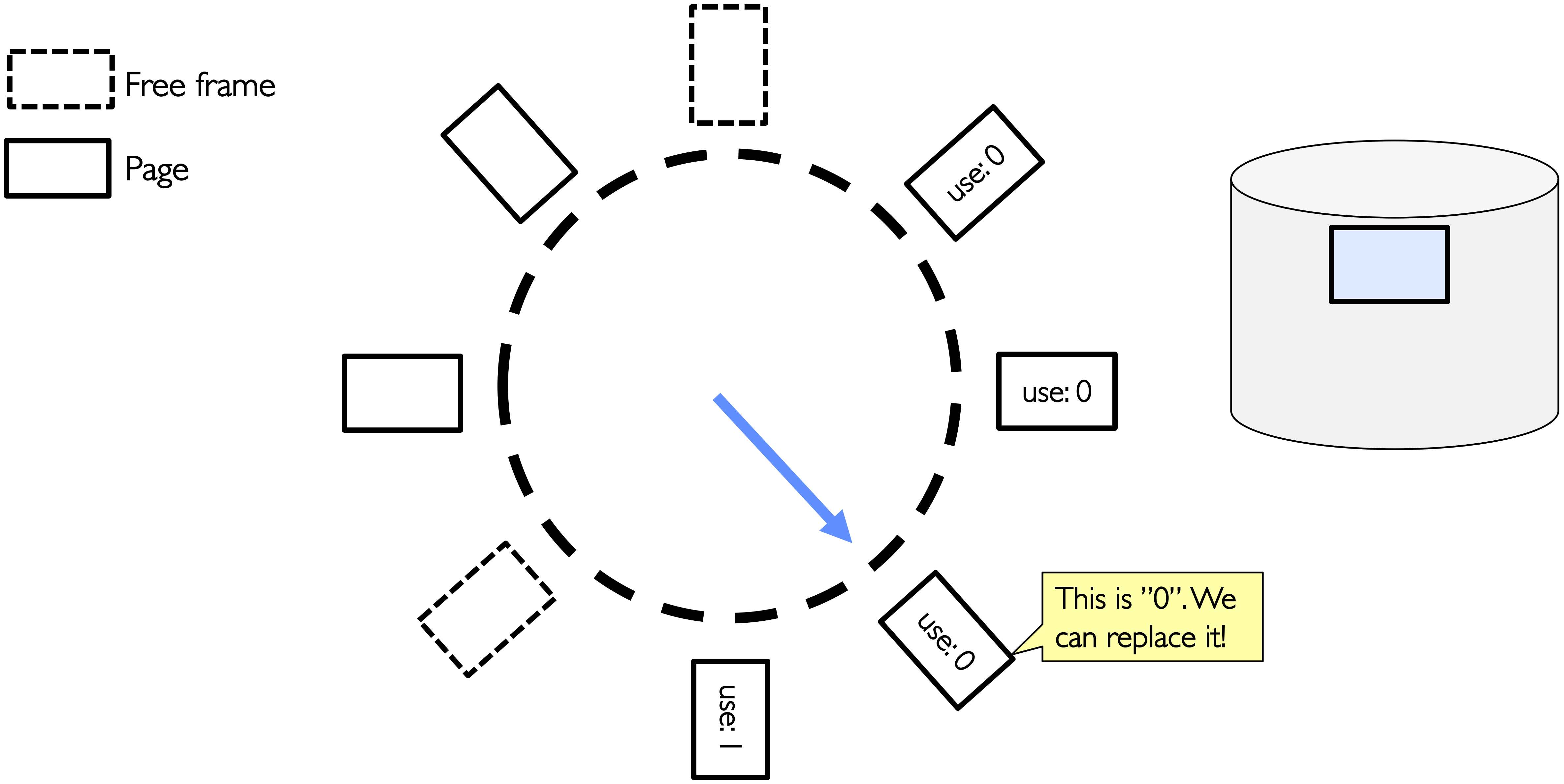
Clock Algorithm Example: Page Fault



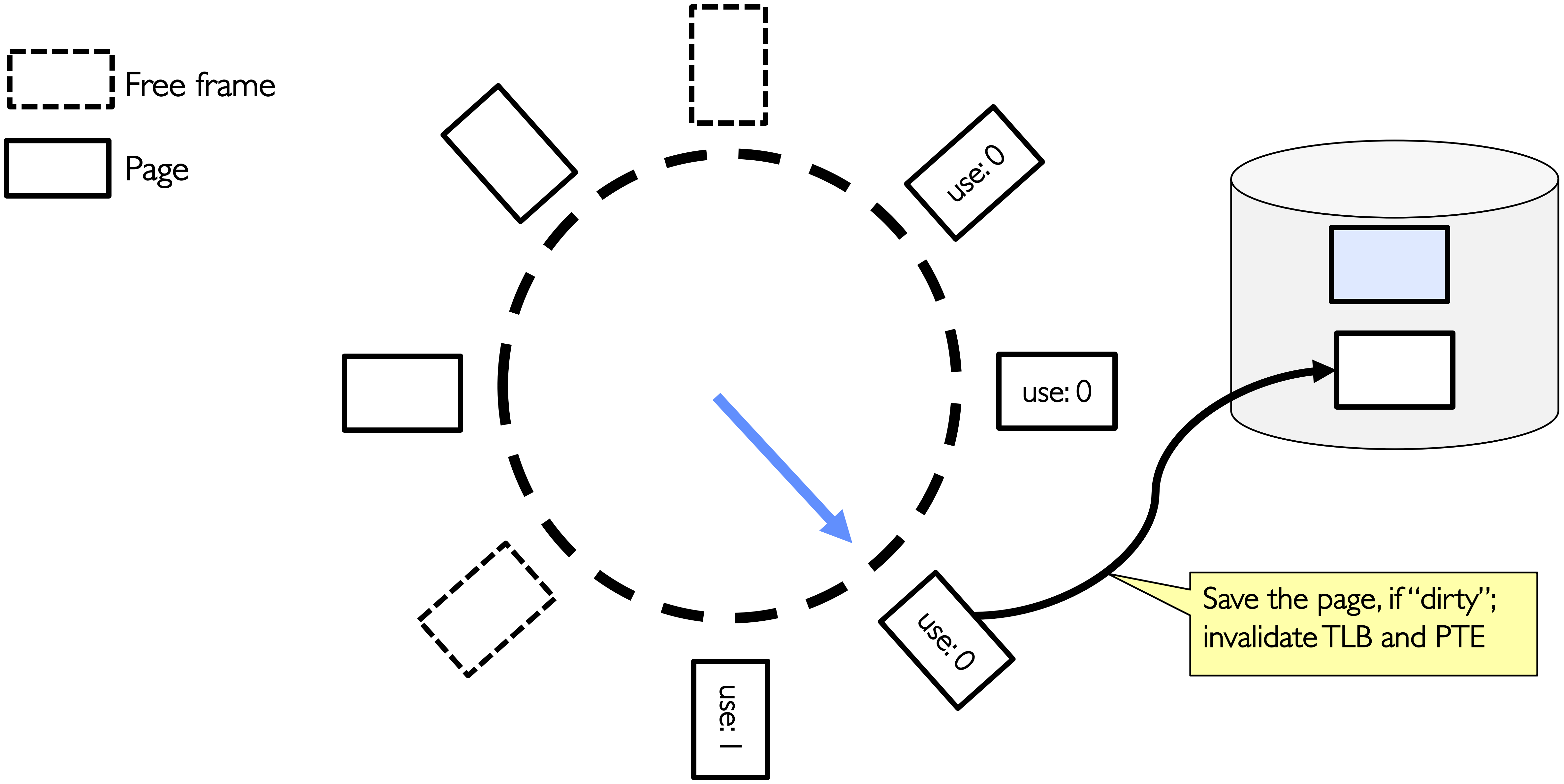
Clock Algorithm Example: Page Fault



Clock Algorithm Example: Page Fault

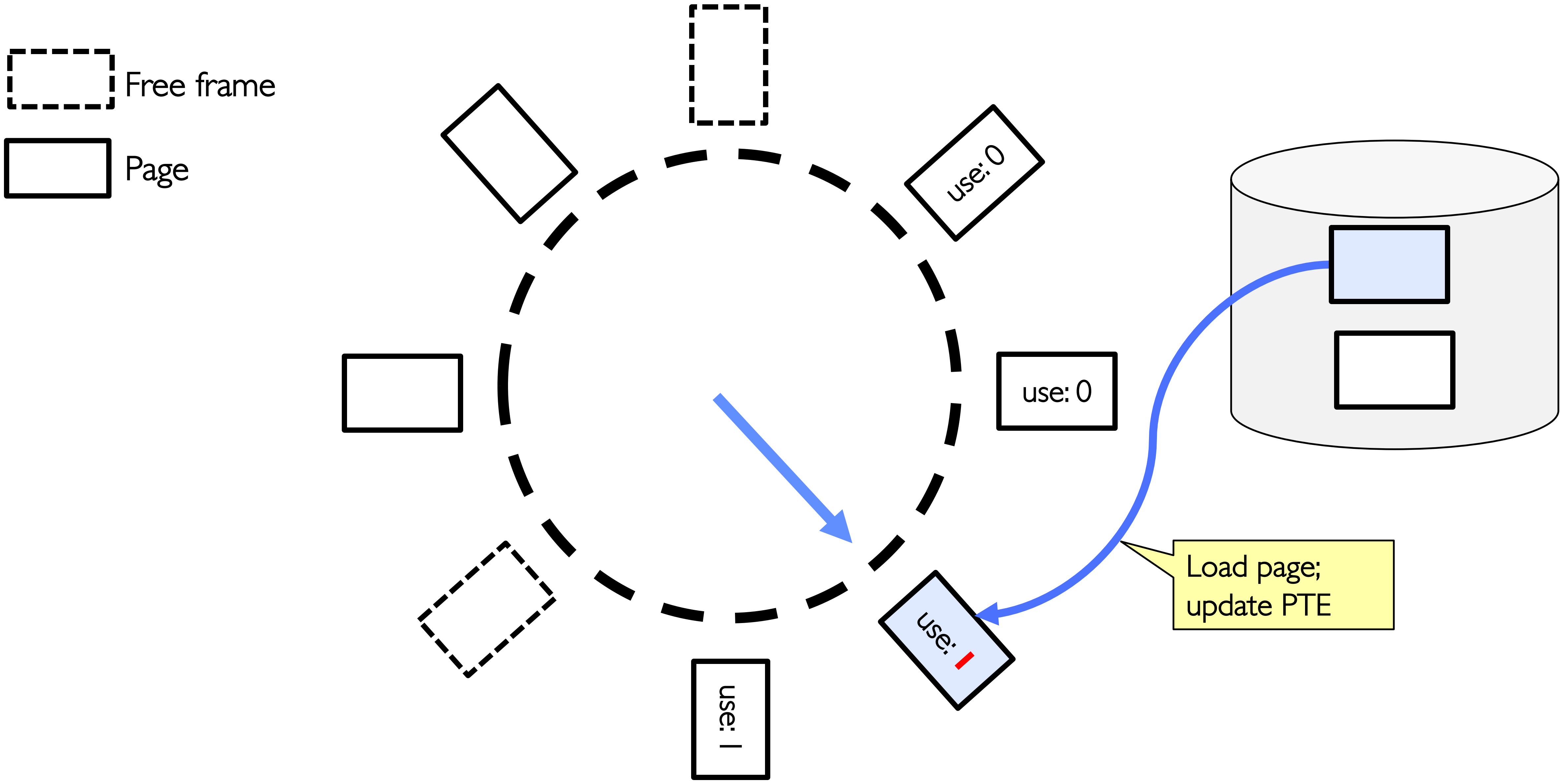


Clock Algorithm Example: Page Fault

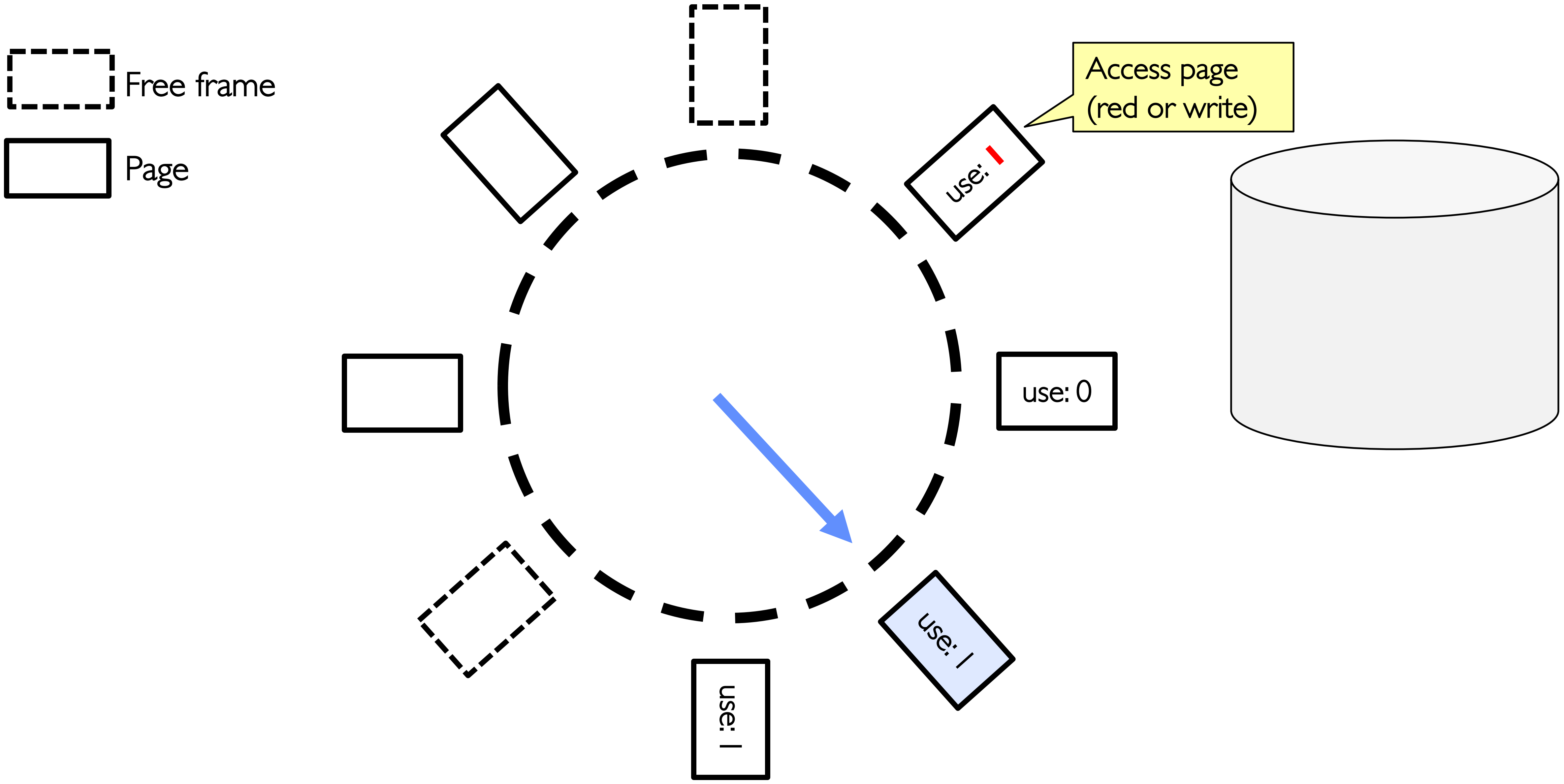


Save the page, if "dirty"; invalidate TLB and PTE

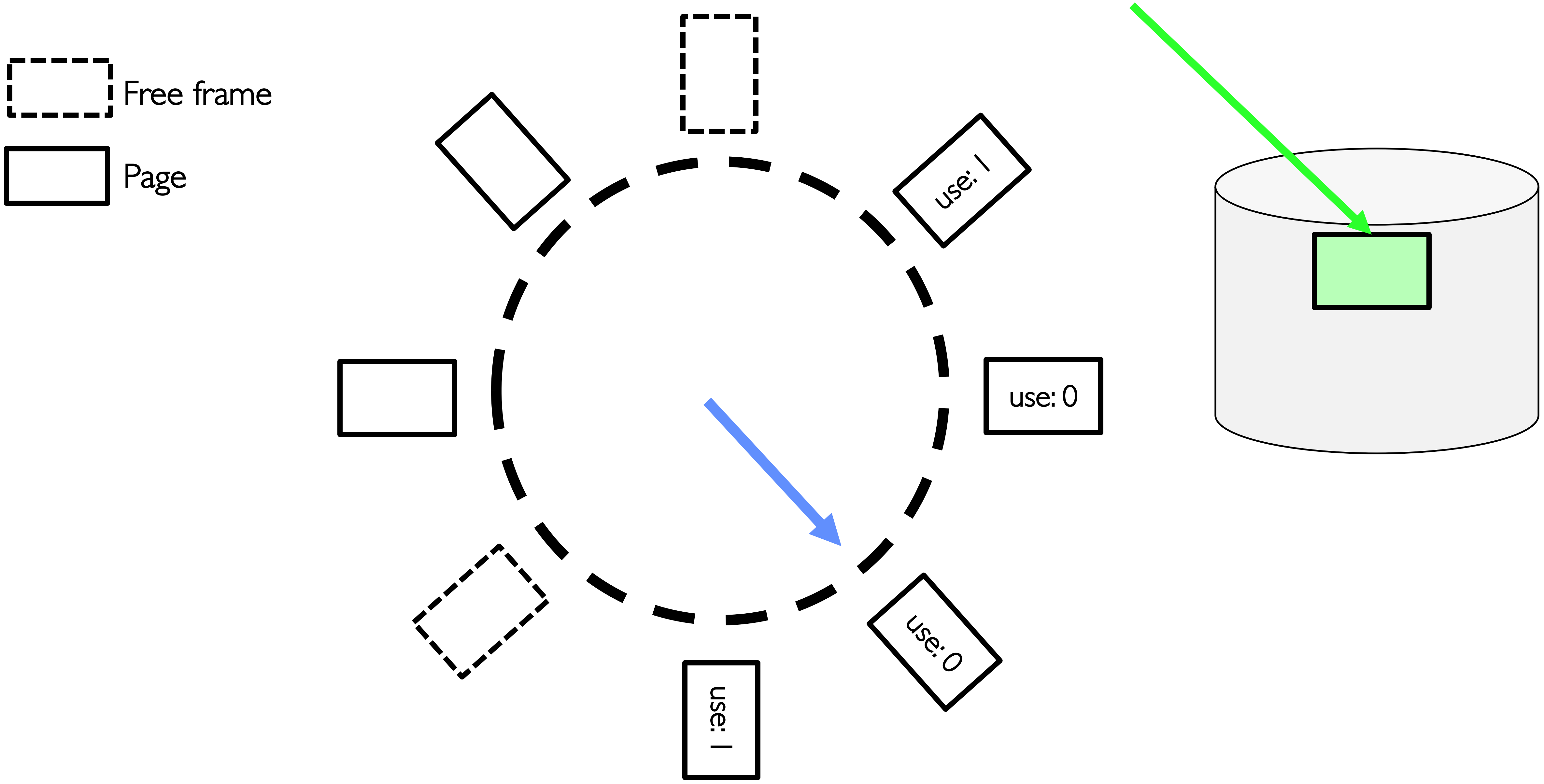
Clock Algorithm Example: Page Fault



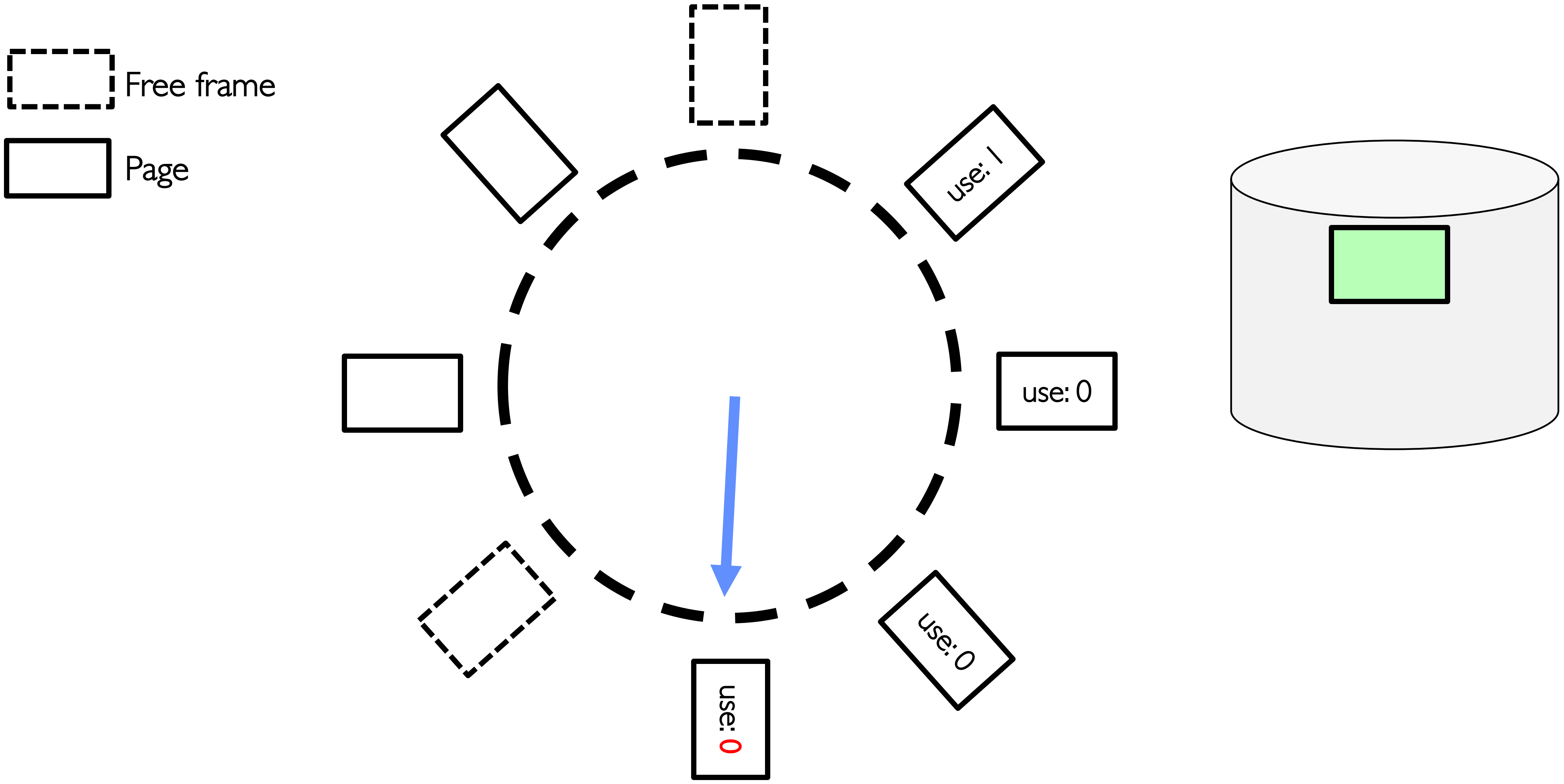
Clock Algorithm Example



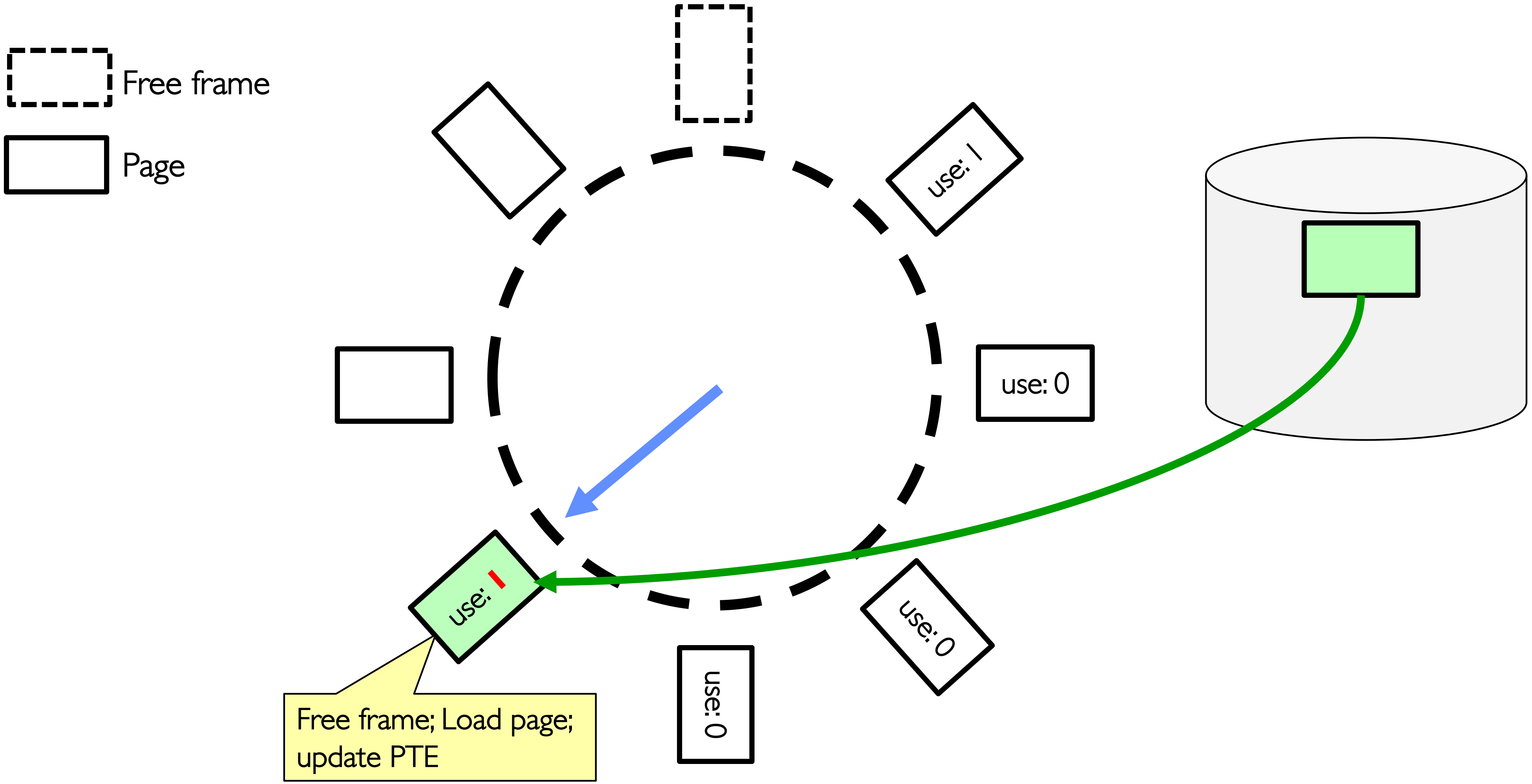
Clock Algorithm Example: Another Page Fault



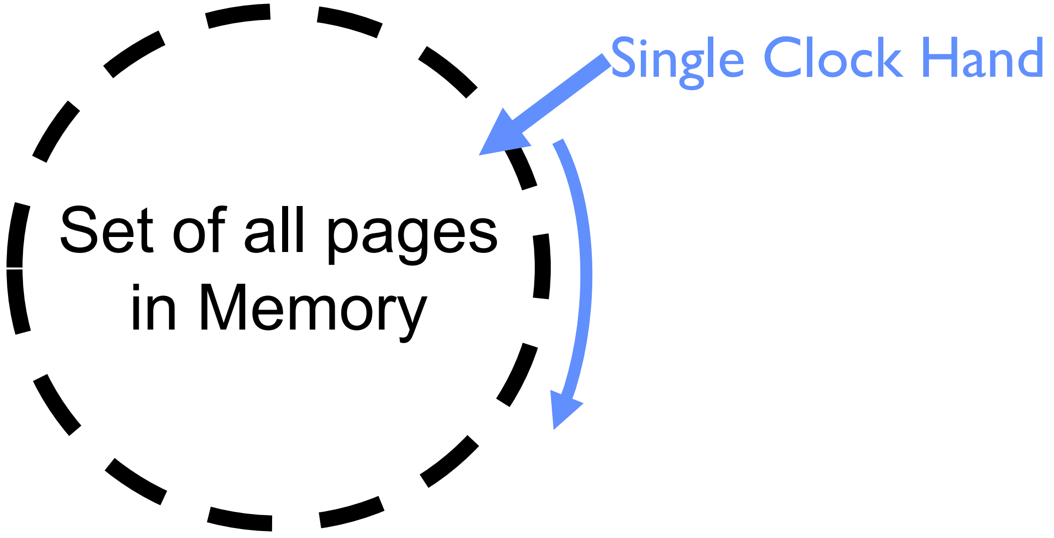
Clock Algorithm Example: Another Page Fault



Clock Algorithm Example: Another Page Fault



Clock Algorithm: More details



- Will always find a page or loop forever?
  - Even if all use bits set, will eventually loop all the way around
- 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?

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 $\rightarrow$ 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
    - » If N ~ 1K, really good approximation
  - Why pick small N? More efficient
    - » Otherwise might have to look a long way to find free page
- What about "modified" (or "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)

Summary (1/3)

- The Principle of Locality:
  - Program likely to access a relatively small portion of the address space at any instant of time.
    - » Temporal Locality: Locality in Time
    - » Spatial Locality: Locality in Space
- Three (+1) Major Categories of Cache Misses:
  - Compulsory Misses: sad facts of life. Example: cold start misses.
  - Conflict Misses: increase cache size and/or associativity
  - Capacity Misses: increase cache size
  - Coherence Misses: Caused by external processors or I/O devices
- Cache Organizations:
  - Direct Mapped: single block per set
  - Set associative: more than one block per set
  - Fully associative: all entries equivalent

Summary (2/3)

- "Translation Lookaside Buffer" (TLB)
  - Small number of PTEs and optional process IDs (< 512)
  - Fully Associative (Since conflict misses expensive)
  - On TLB miss, page table must be traversed and if located PTE is invalid, cause Page Fault
  - On change in page table, TLB entries must be invalidated
  - TLB is logically in front of cache (need to overlap with cache access)

Summary (3/3)

- Demand Paging: Treating the DRAM as a cache on disk
  - Page table tracks which pages are in memory
  - Any attempt to access a page that is not in memory generates a page fault, which causes OS to bring missing page into memory
- 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
- Nth-chance clock algorithm: Another approximate LRU
  - Give pages multiple passes of clock hand before replacing