CS162
Operating Systems and Systems Programming
Lecture 18

Demand Paging (Finished), General I/O

October 31st, 2024

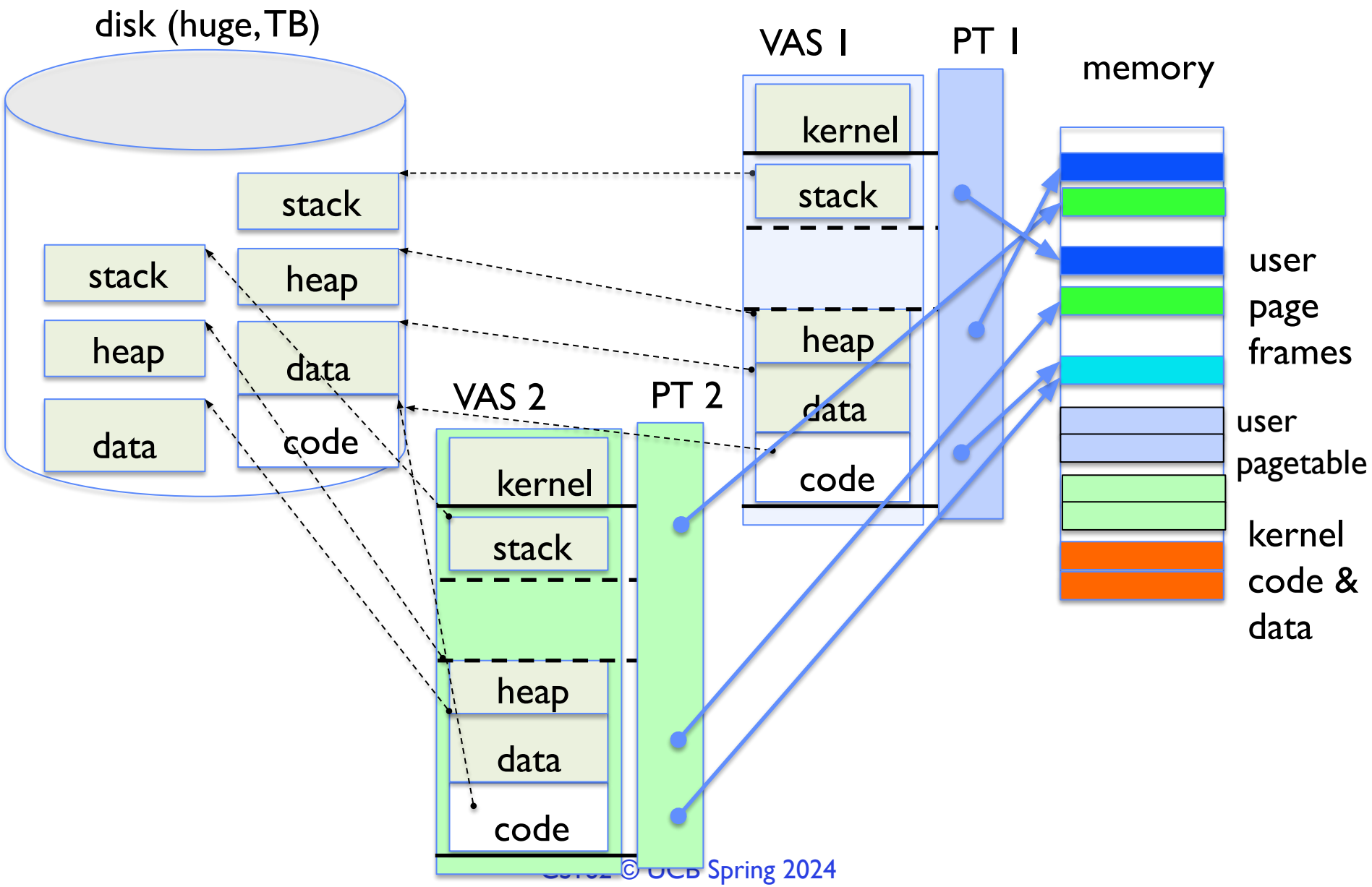
Prof. Ion Stoica

http://cs162.eecs.Berkeley.edu

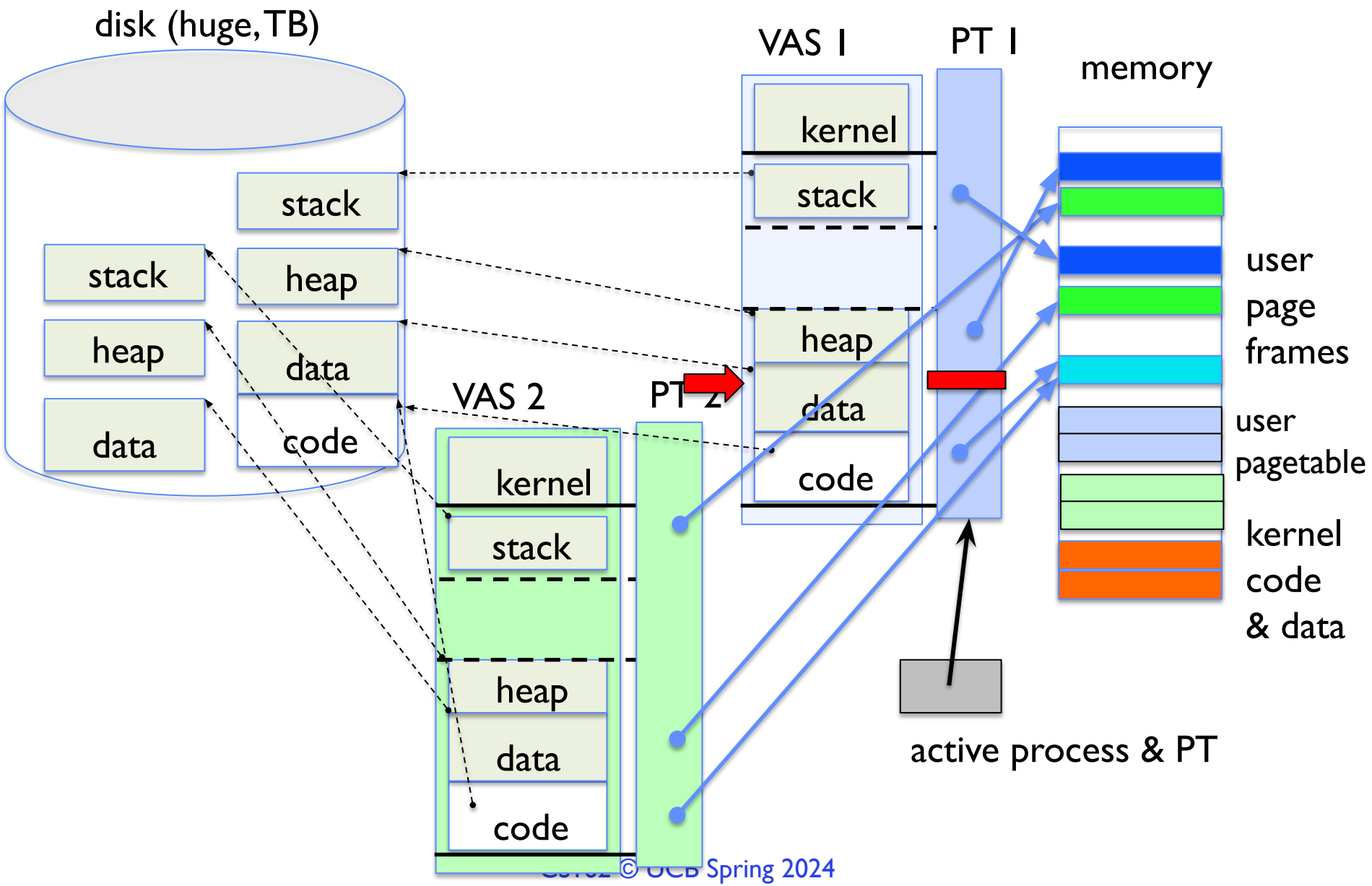
What Data Structure Maps / Non-Resident Pages to Disk?

- `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

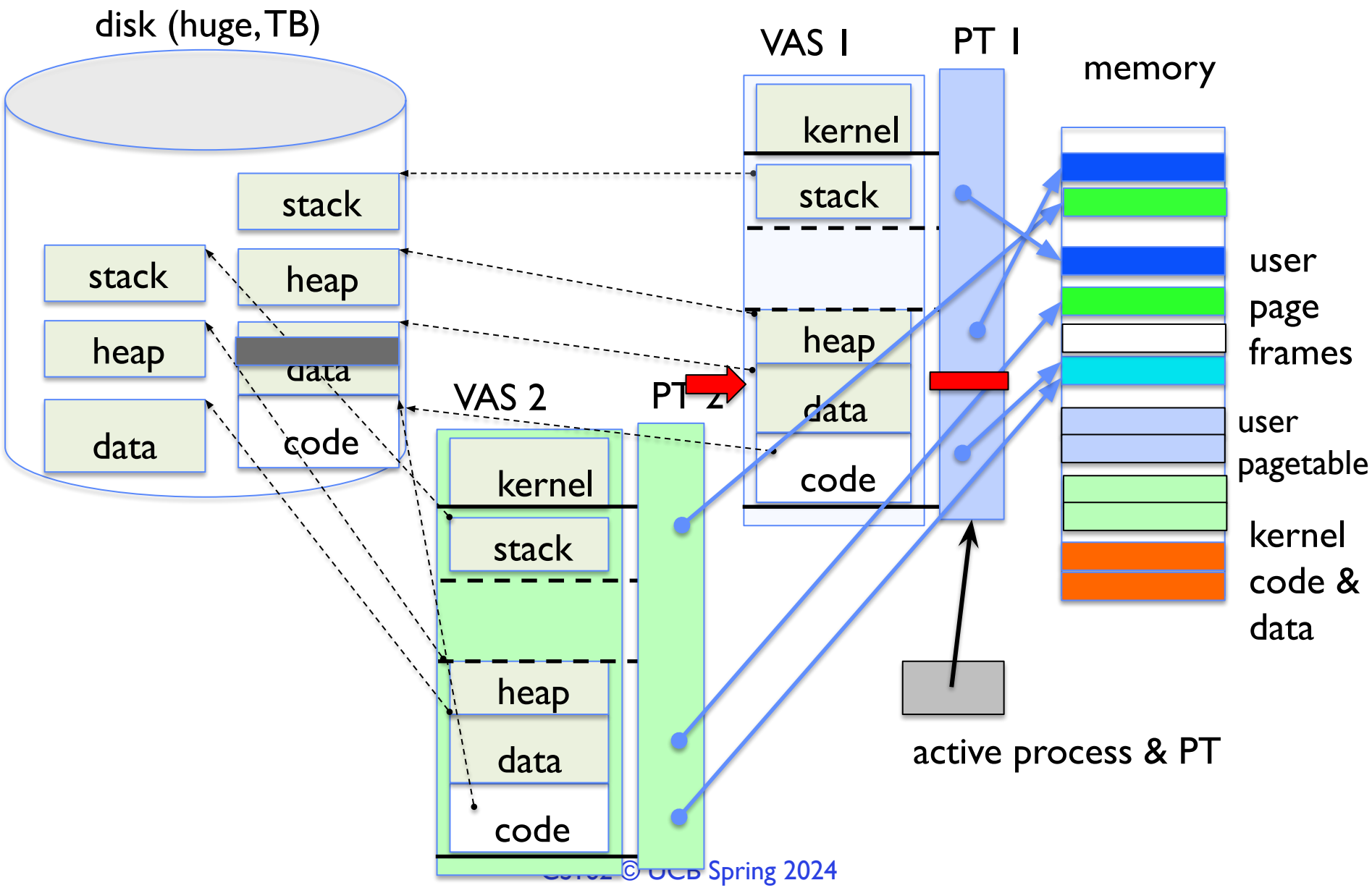
Provide Backing Store for VAS



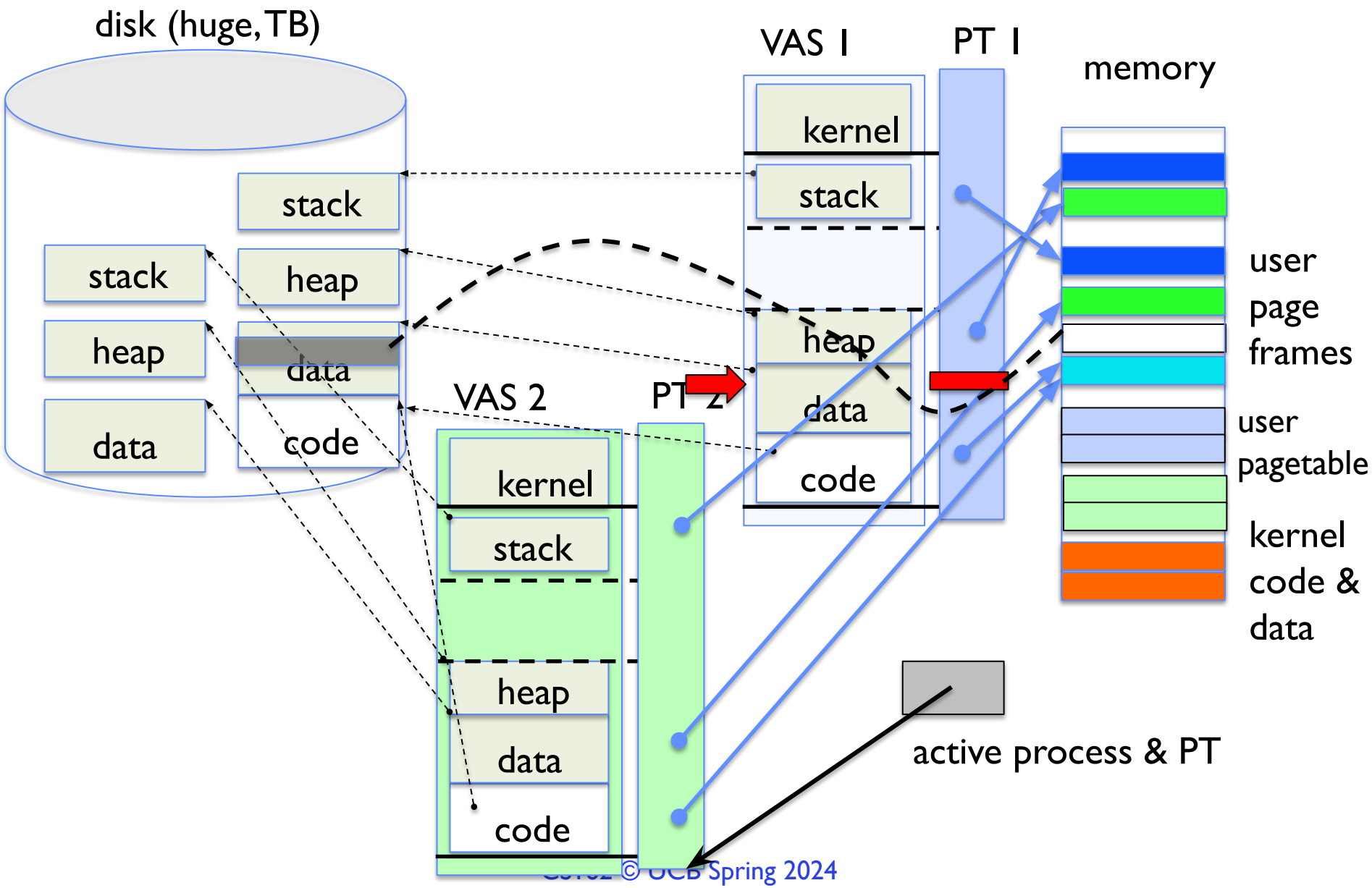
On page Fault …



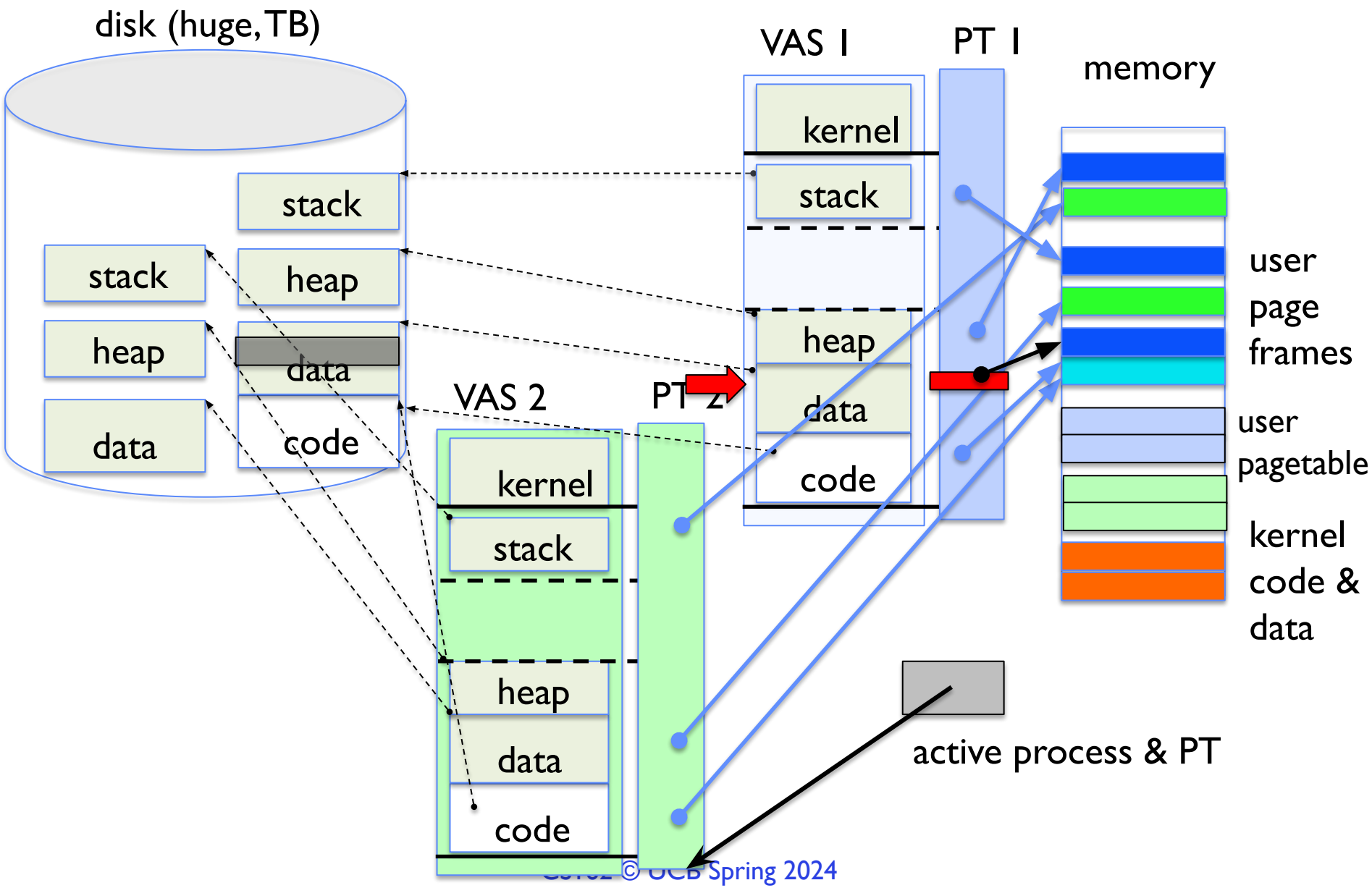
On page Fault … find & start load



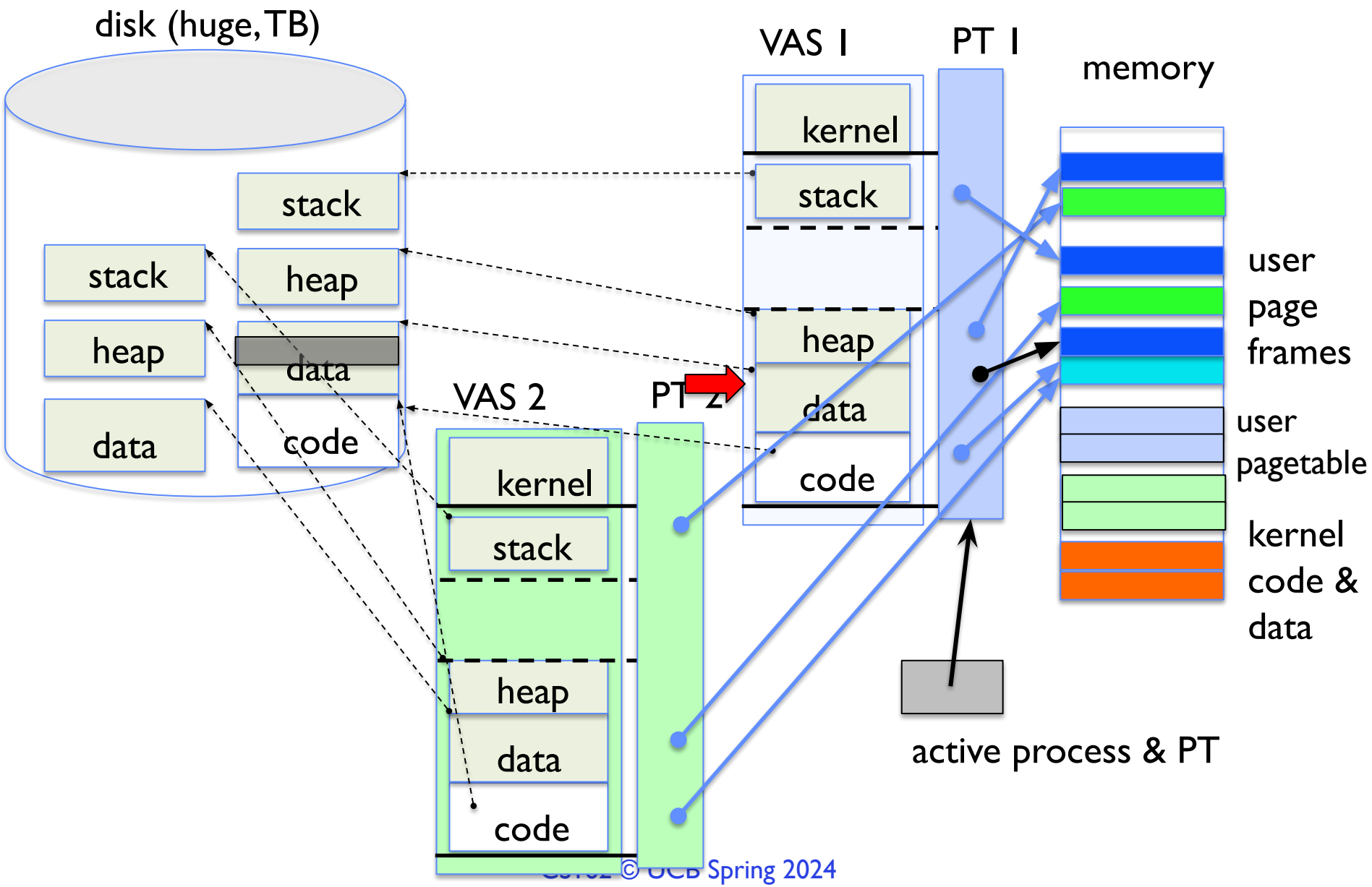
On page Fault … schedule other P or T



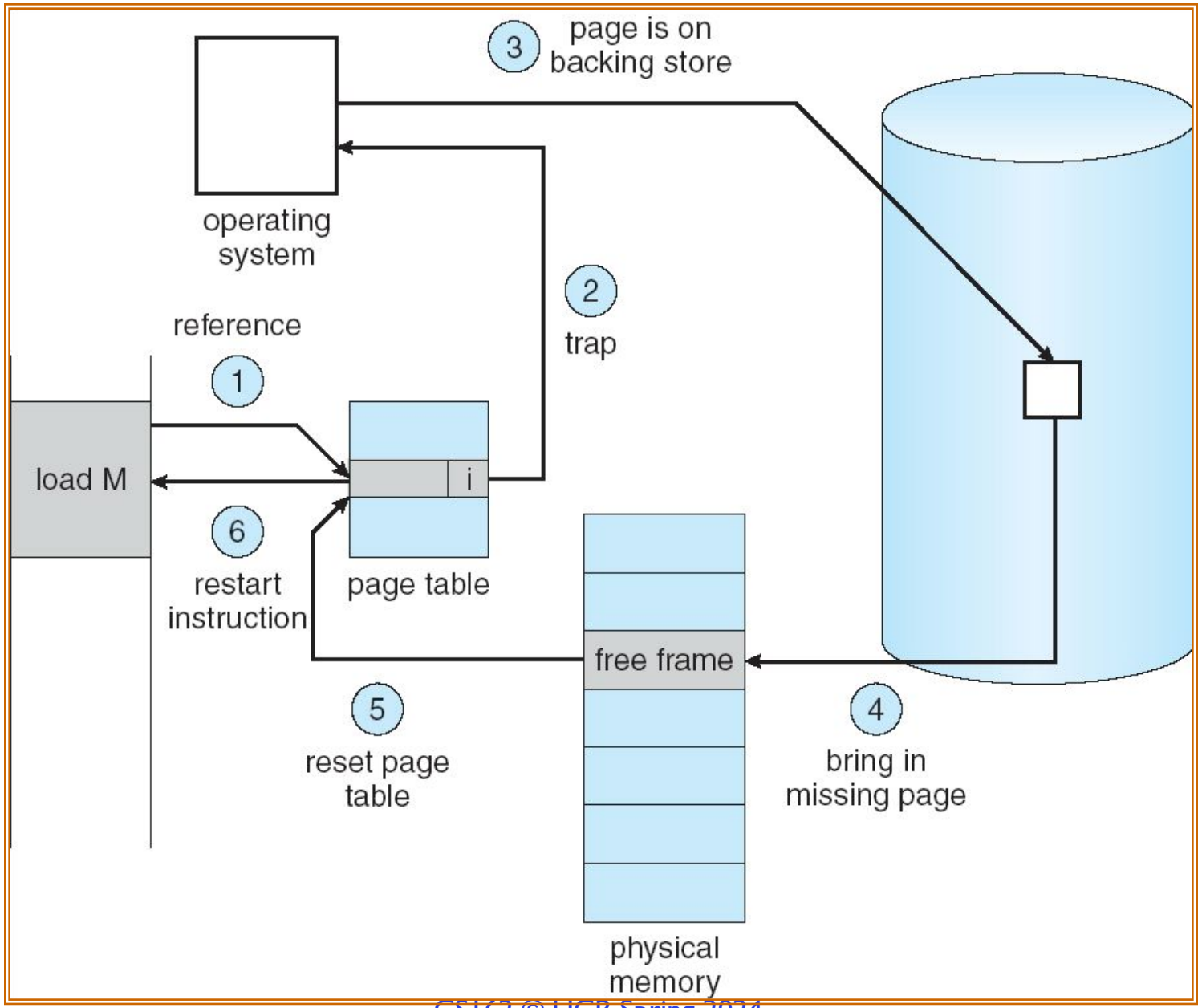
On page Fault … update PTE



Eventually reschedule faulting thread



Summary: Steps in Handling a Page Fault



Some questions we need to answer!

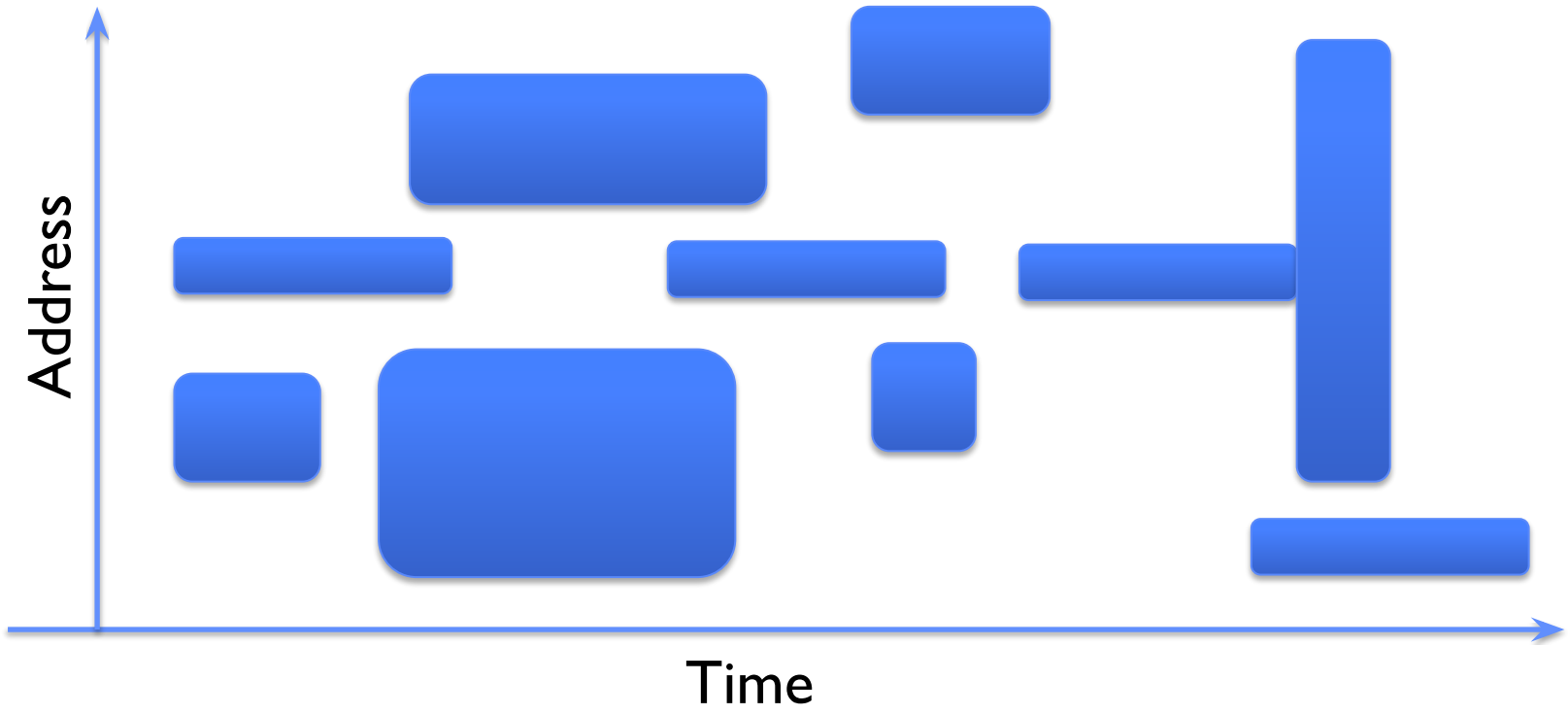
- During a page fault, where does the OS get a free frame?
  - Keeps a free list
  - Unix runs a "reaper" if memory gets too full
    - » Schedule dirty pages to be written back on disk
    - » Zero (clean) pages which haven't been accessed in a while
  - As a last resort, evict a dirty page first

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

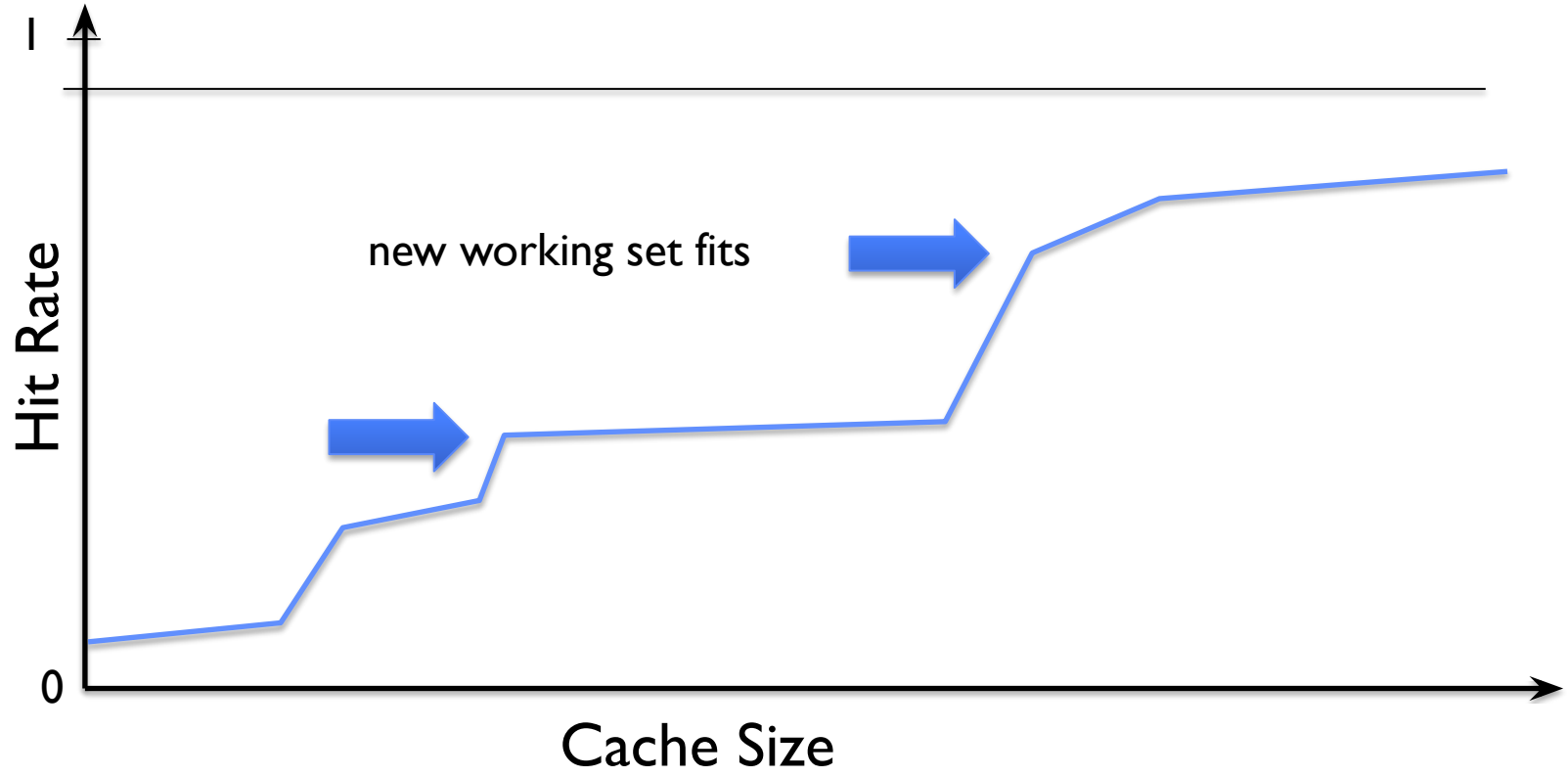
- How many page frames/process?
  - Like thread scheduling, need to "schedule" memory resources:
    - » Utilization? fairness? priority?
  - Allocation of disk paging bandwidth

Working Set (WS) Model

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

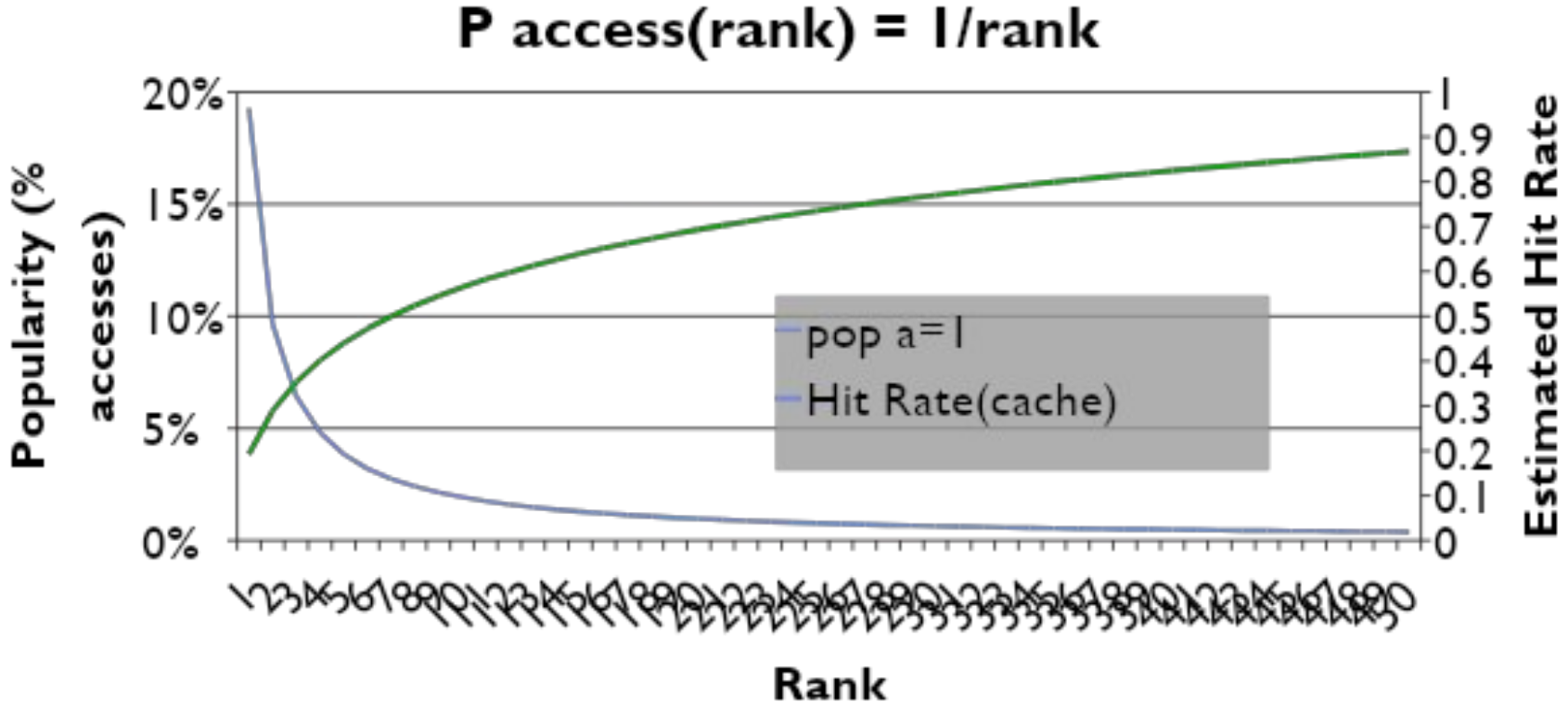


Cache Behavior under WS model



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

Another model of Locality: Zipf



- Likelihood of accessing item of rank r is α $1/r^a$
- 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 = Hit Rate x Hit Time + Miss Rate x Miss Time (Miss Time = Hit Time + Miss Penalty)
  - EAT = Hit Time + Miss Rate x Miss Penalty
- Example:
  - Memory access time = 200 nanoseconds
  - Average page-fault service time = 8 milliseconds
  - Suppose p = Probability of miss, 1-p = Probably of hit
  - Then, we can compute EAT as follows:

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

What Factors Lead to Misses in Page Cache?

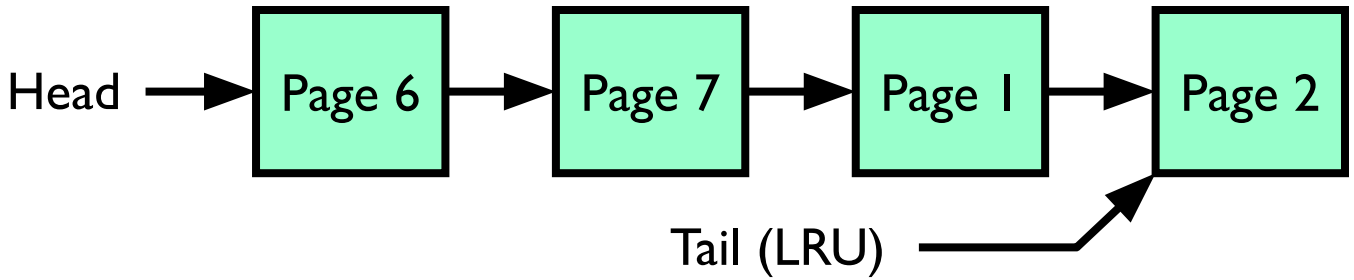
- Compulsory Misses:
  - Pages that have never been paged into memory before
  - How might we remove these misses?
    - » Prefetching: loading them into memory before needed
    - » Need to predict future somehow! More later
- Capacity Misses:
  - Not enough memory. Must somehow increase available memory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

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?

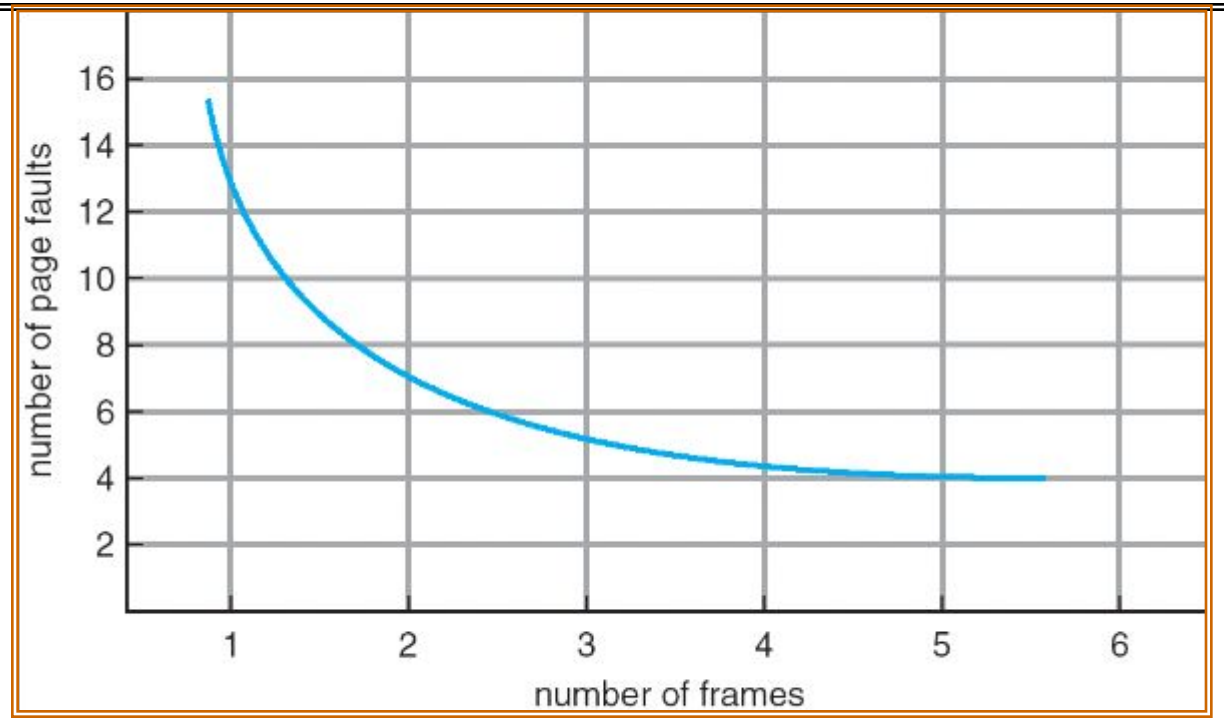
- Consider the following: A B C D A B C D A B C D
- LRU Performs as follows (same as FIFO here):

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?
- 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 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	

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

Administrivia

- Midterm 2: Next Tuesday 11/05 from 7-9PM
  - Also includes the Midterm 1 material
  - Closed book: with two double-sided handwritten sheets of notes

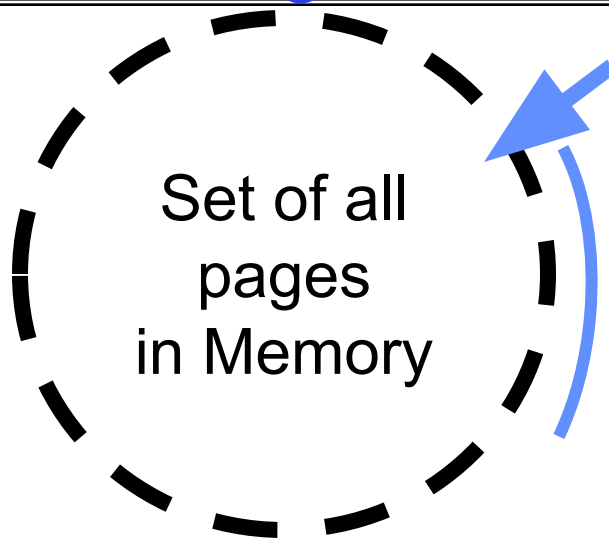
- Project 2 in full swing
  - Stay on top of this one. Don't wait until last moment to get pieces together
  - Decide how to your team is going divide up project 2

Approximating LRU: Recall PTE bits

- Which bits of a PTE entry can help us approximate LRU?
  Remember Intel PTE:

  - The "Present" bit (called "Valid" elsewhere):
    - » P==0: Page is invalid and a reference will cause page fault
    - » P==1: Page frame number is valid and MMU is allowed to proceed with translation
  - The "Writable" bit (could have opposite sense and be called "Read-only"):
    - » W==0: Page is read-only and cannot be written.
    - » W==1: Page can be written
  - The "Accessed" bit (called "Use" elsewhere):
    - » A==0: Page has not been accessed (or used) since last time software set A→0
    - » A==1: Page has been accessed (or used) since last time software set A→0
  - The "Dirty" bit (called "Modified" elsewhere):
    - » D==0: Page has not been modified (written) since PTE was loaded
    - » D==1: Page has changed since PTE was loaded

Approximating LRU: Clock Algorithm



Single Clock Hand:

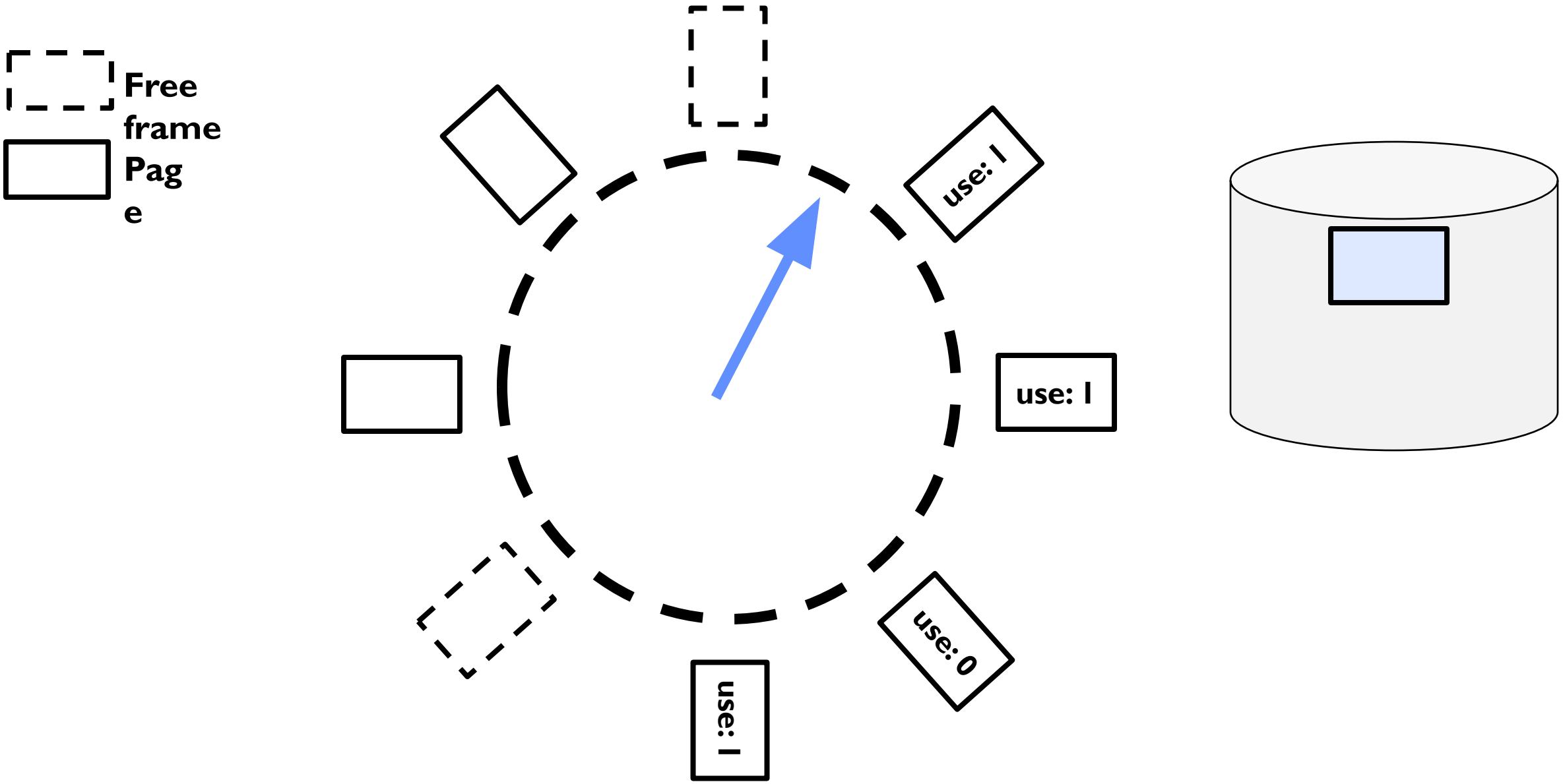
Advances only on page fault!

Check for pages not used recently

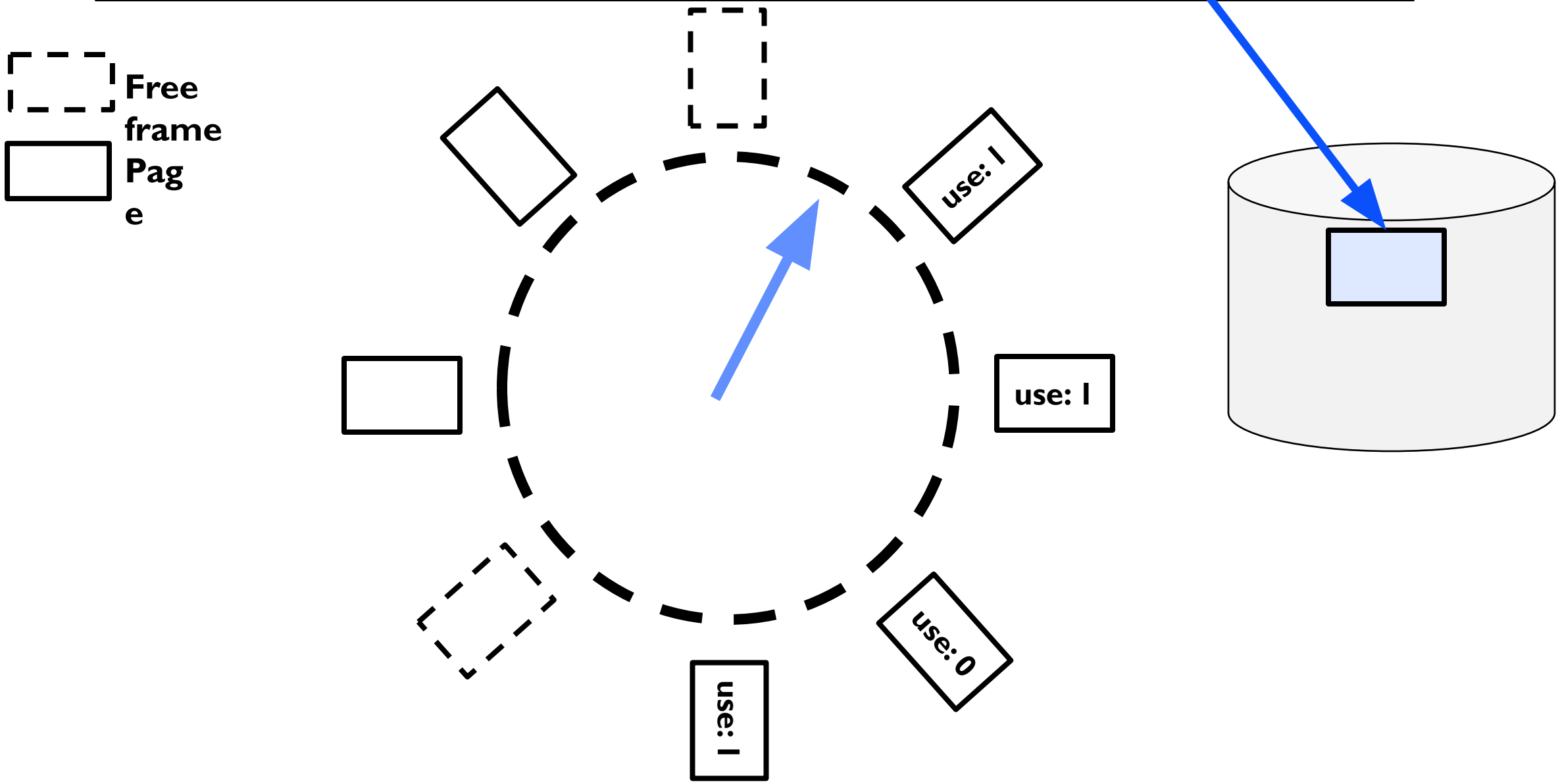
Mark pages as not used recently

- 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
    - » Some hardware sets use bit in the TLB; must be copied back to PTE when TLB entry gets replaced
  - On page fault:
    - » Advance clock hand (not real time)
    - » Check use bit: 1→ used recently; clear and leave alone
      0→ selected candidate for replacement

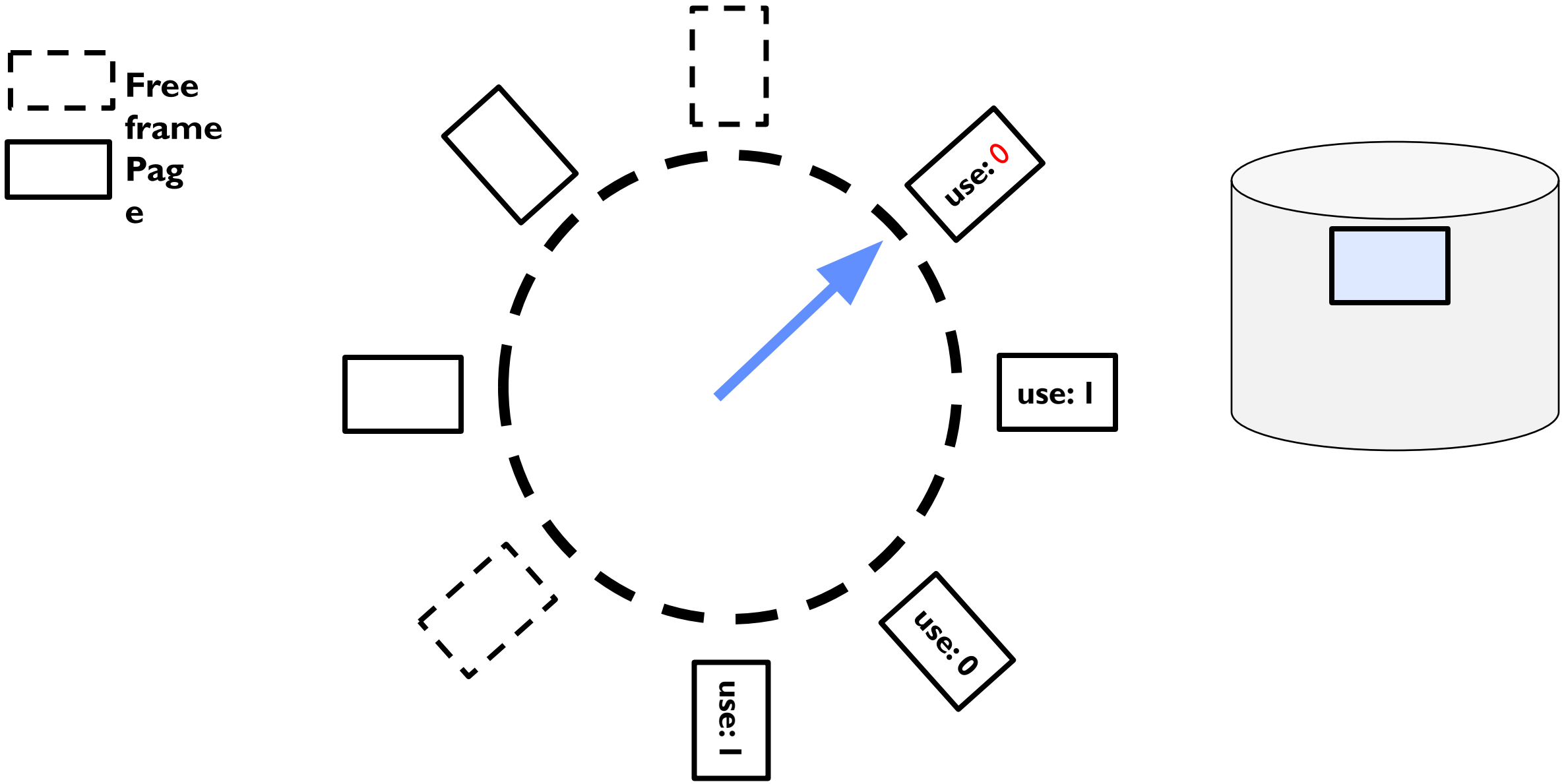
Clock Algorithm Example



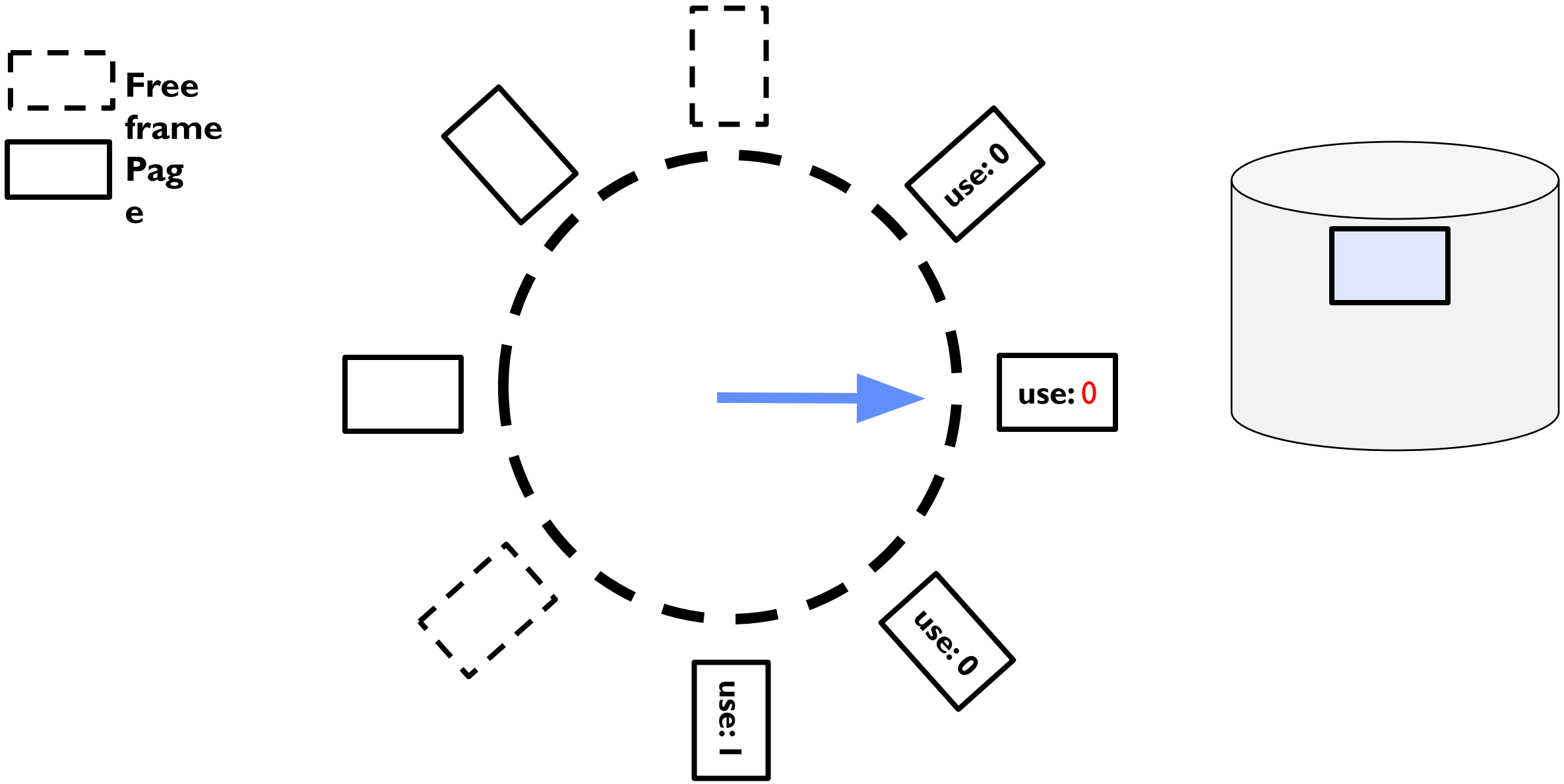
Clock Algorithm Example: Page Fault



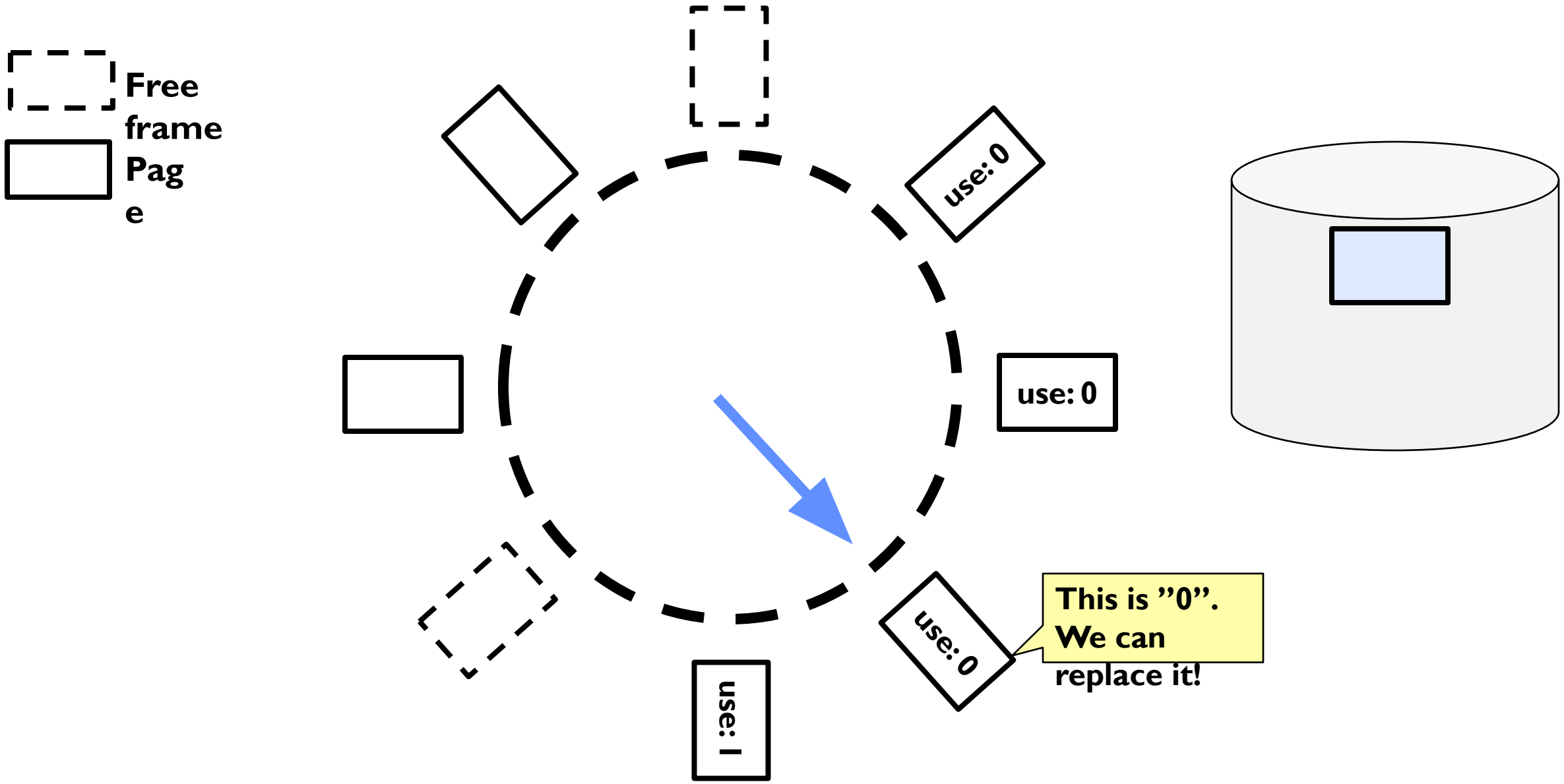
Clock Algorithm Example: Page Fault



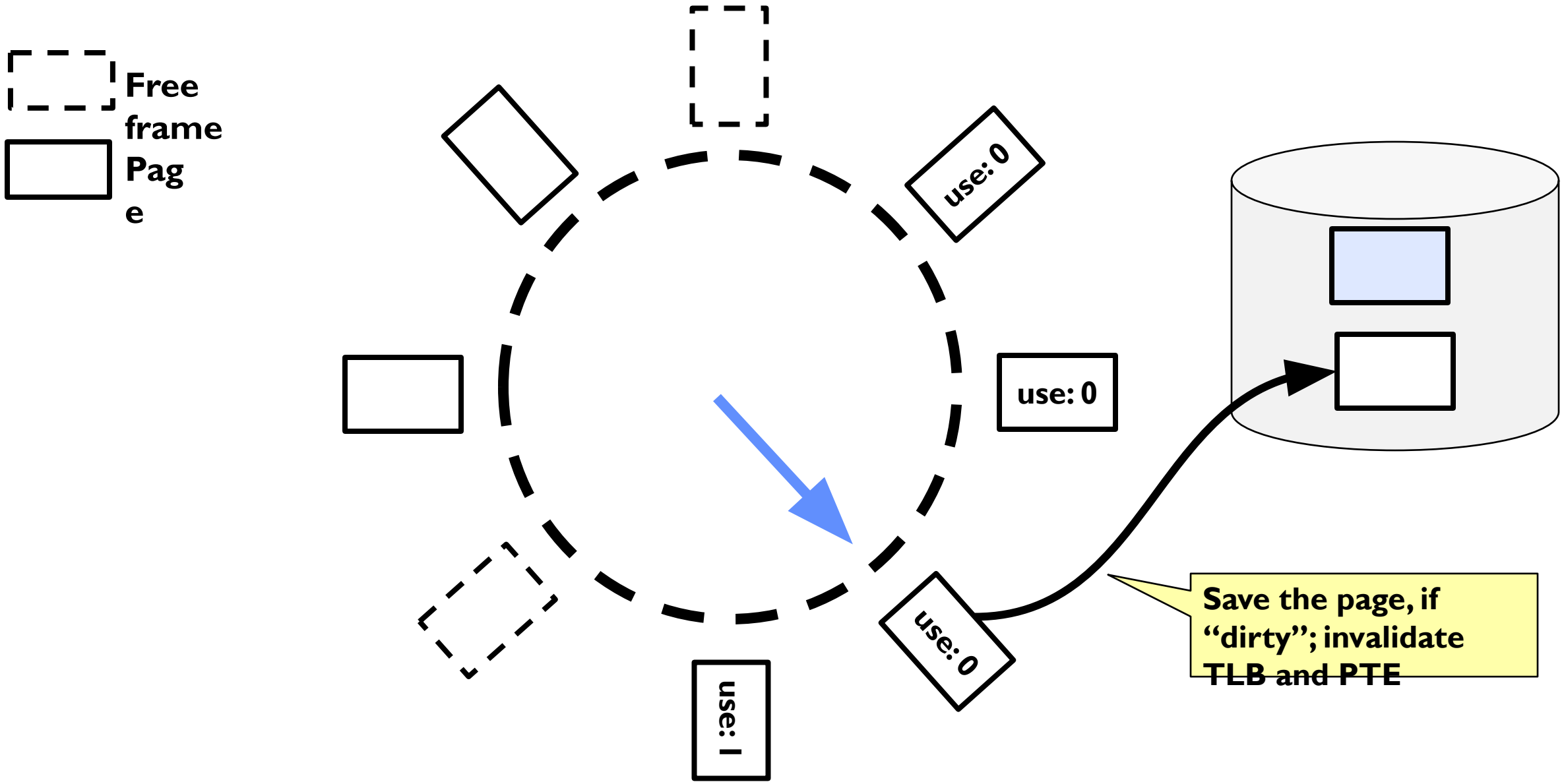
Clock Algorithm Example: Page Fault



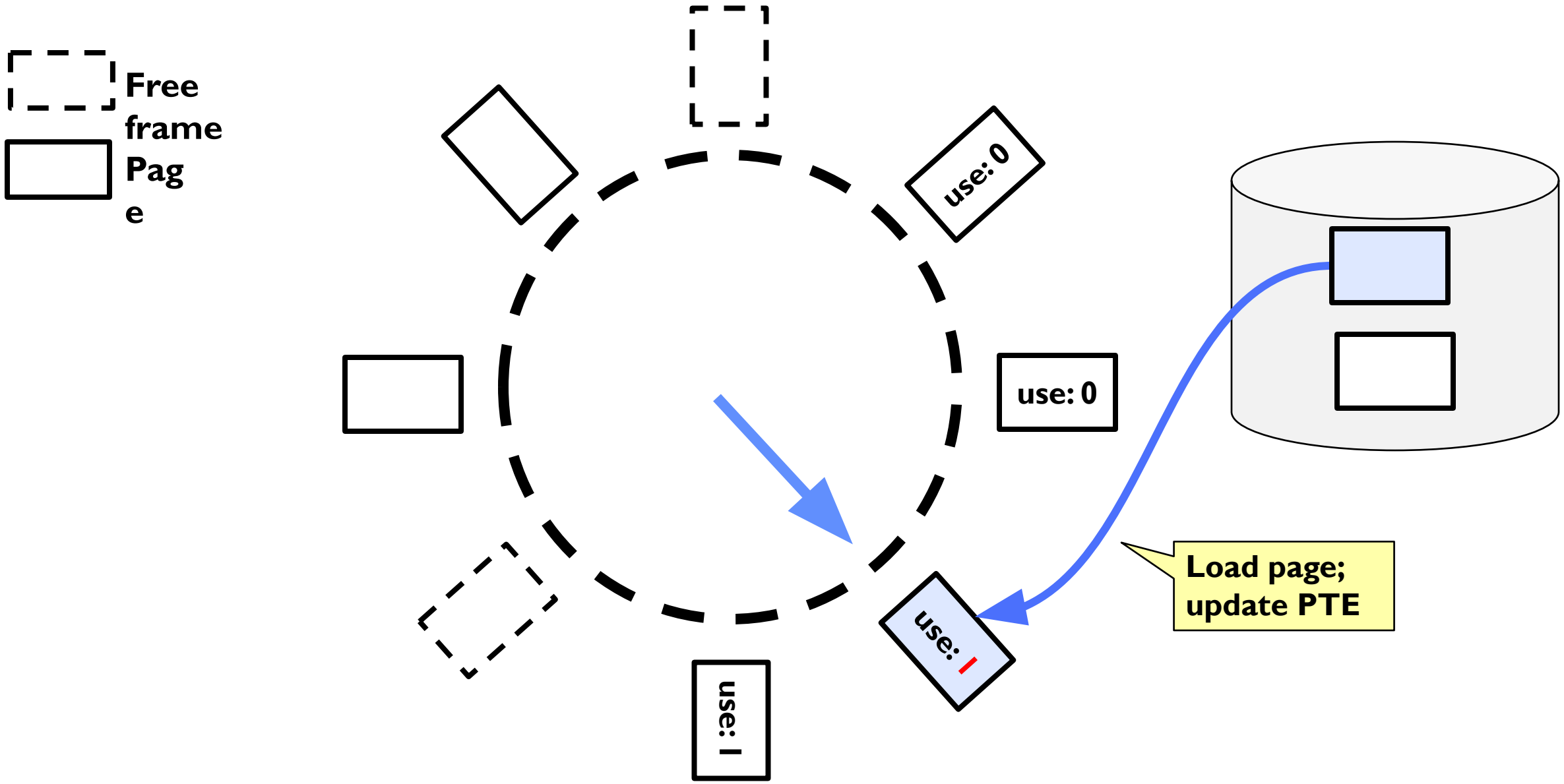
Clock Algorithm Example: Page Fault



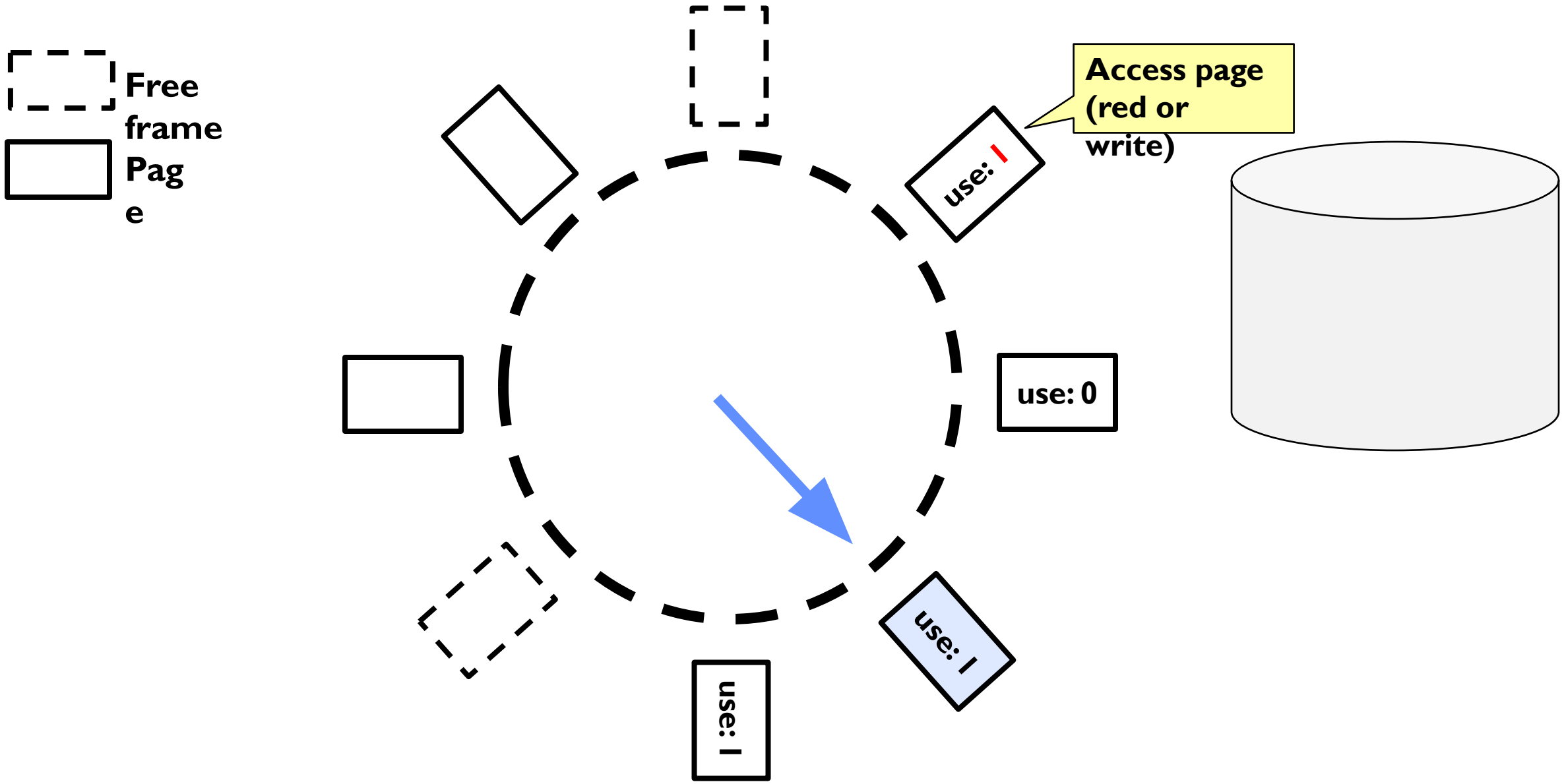
Clock Algorithm Example: Page Fault



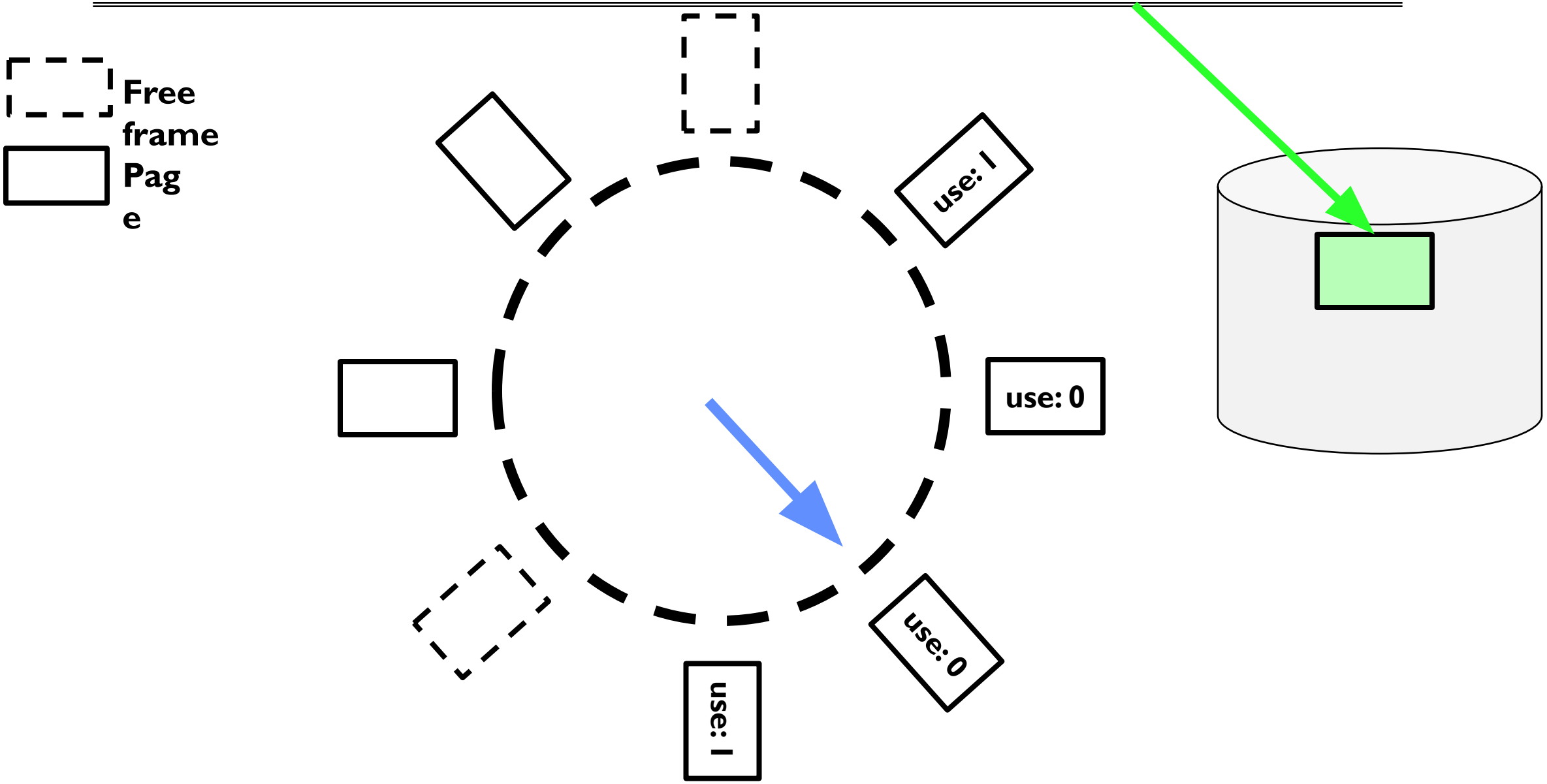
Clock Algorithm Example: Page Fault



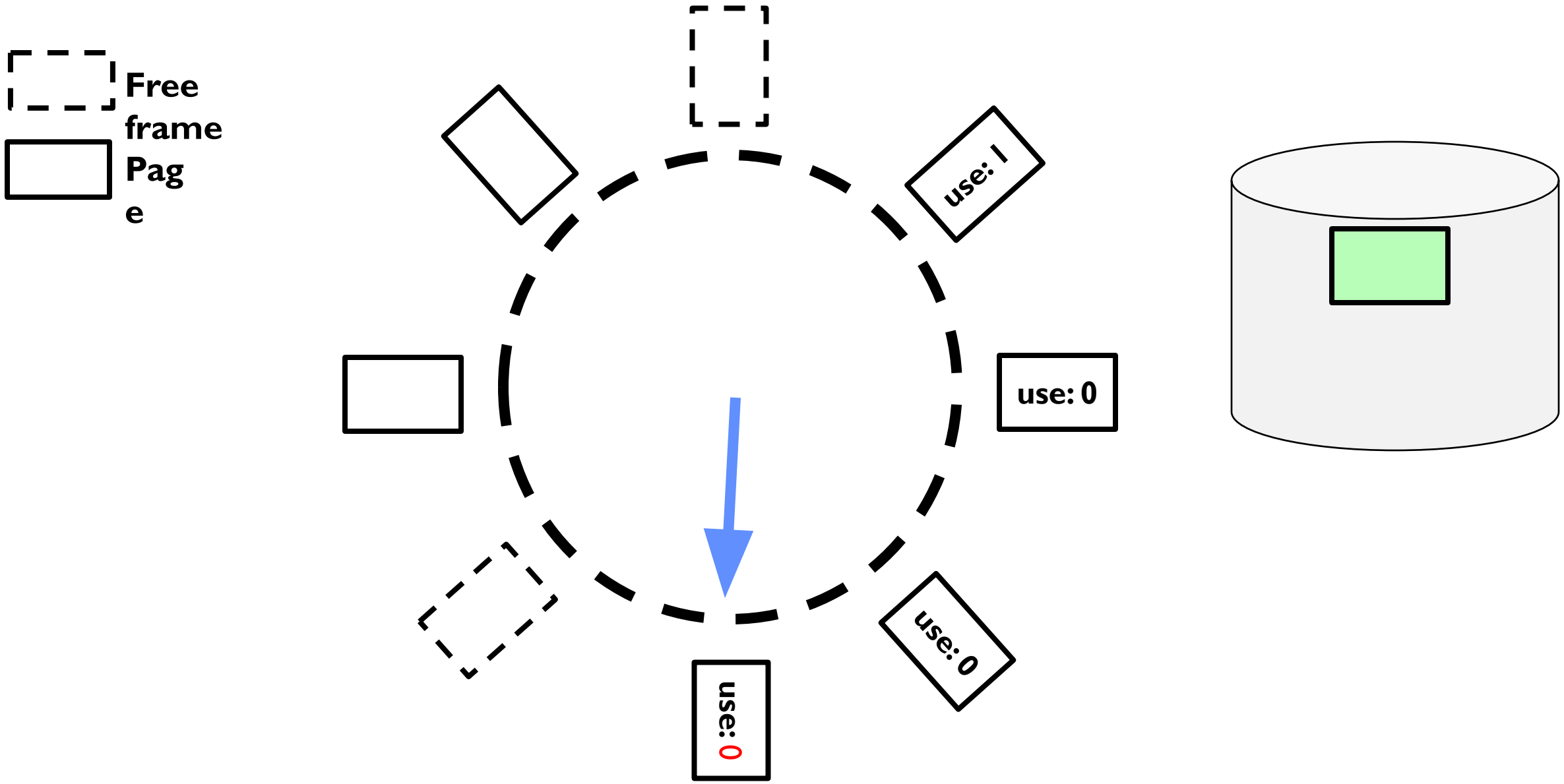
Clock Algorithm Example



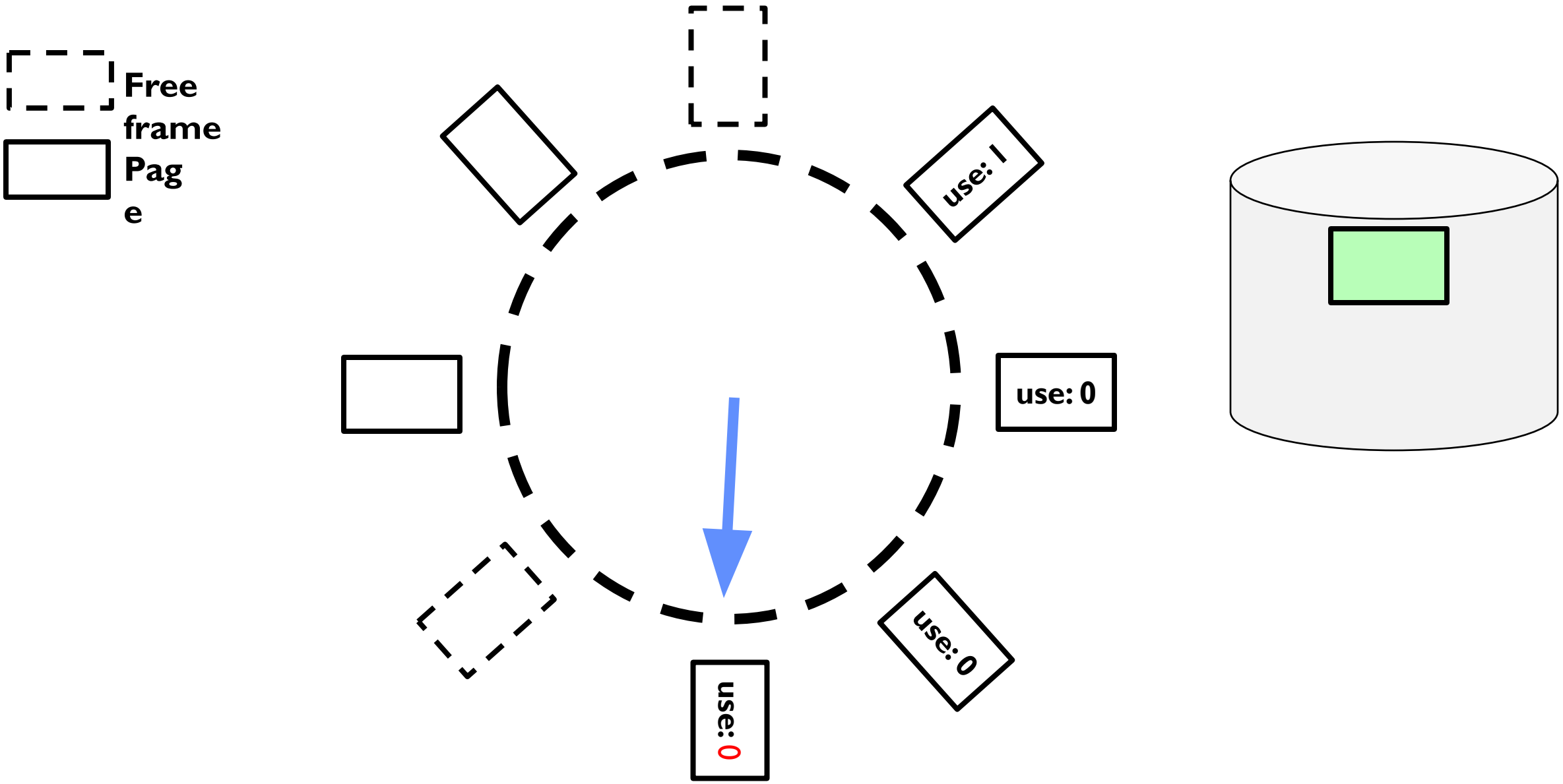
Clock Algorithm Example: Another Page Fault



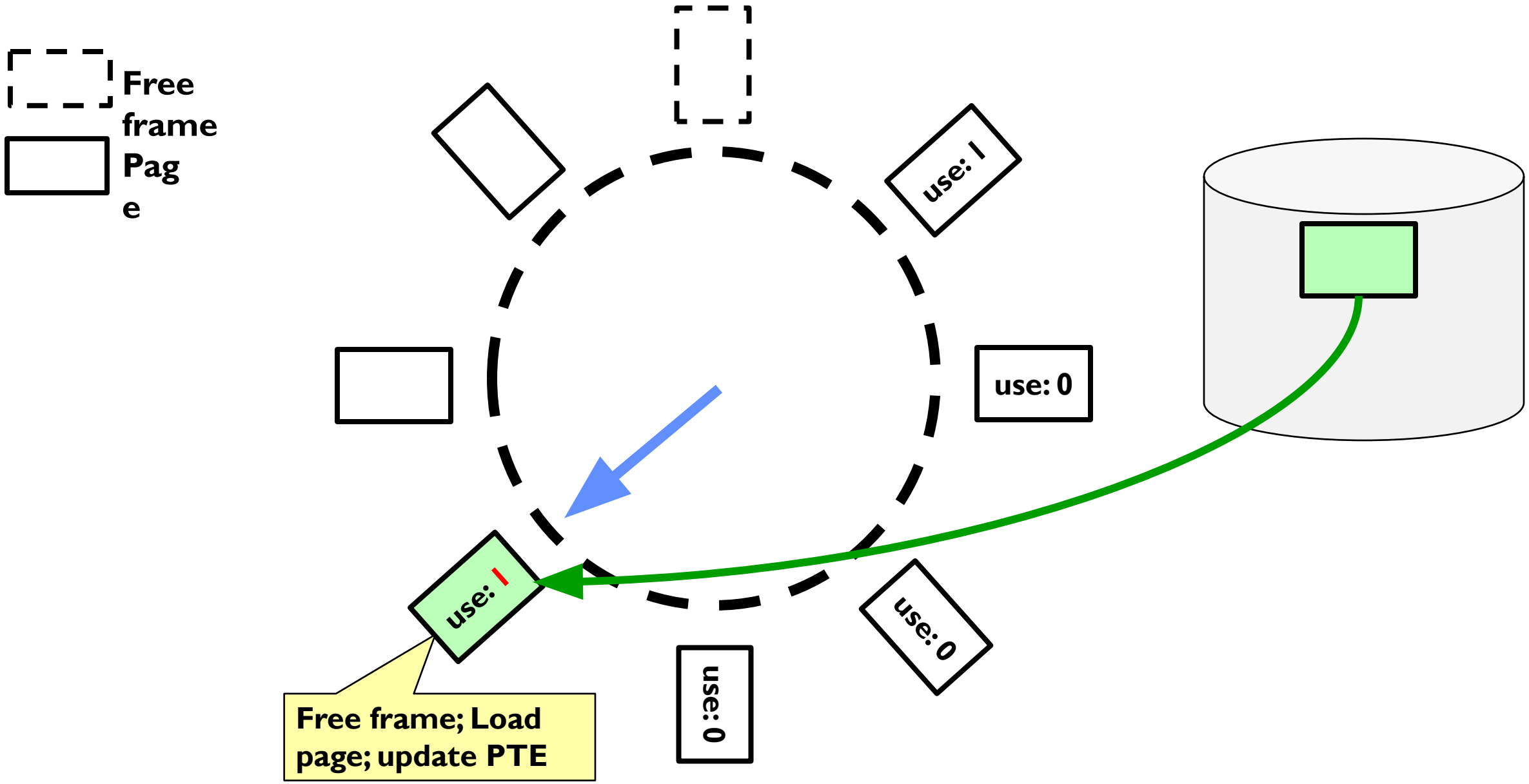
Clock Algorithm Example: Another Page Fault



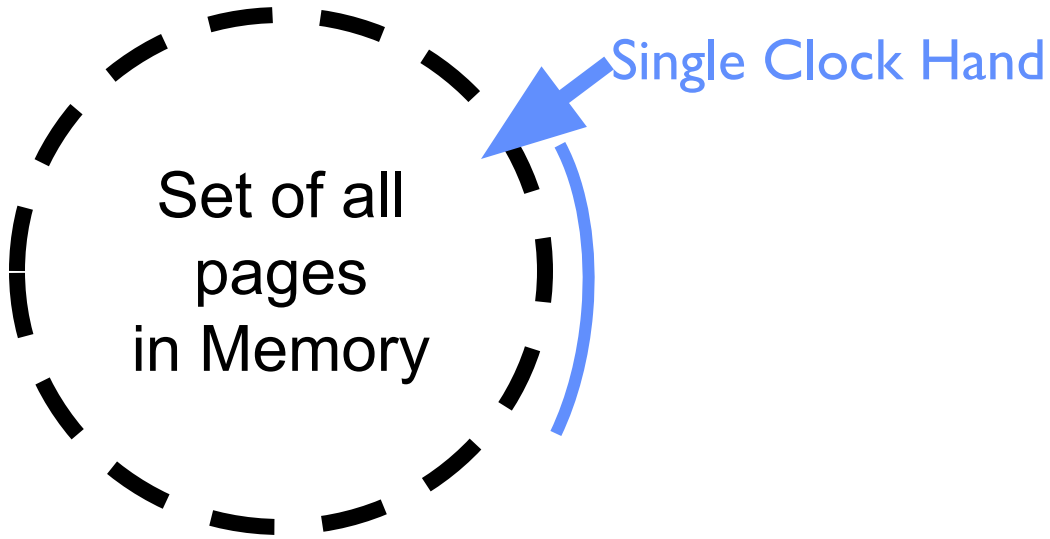
Clock Algorithm Example: Another Page Fault



Clock Algorithm 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 ⇒ FIFO
- 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 → 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 ~ 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)

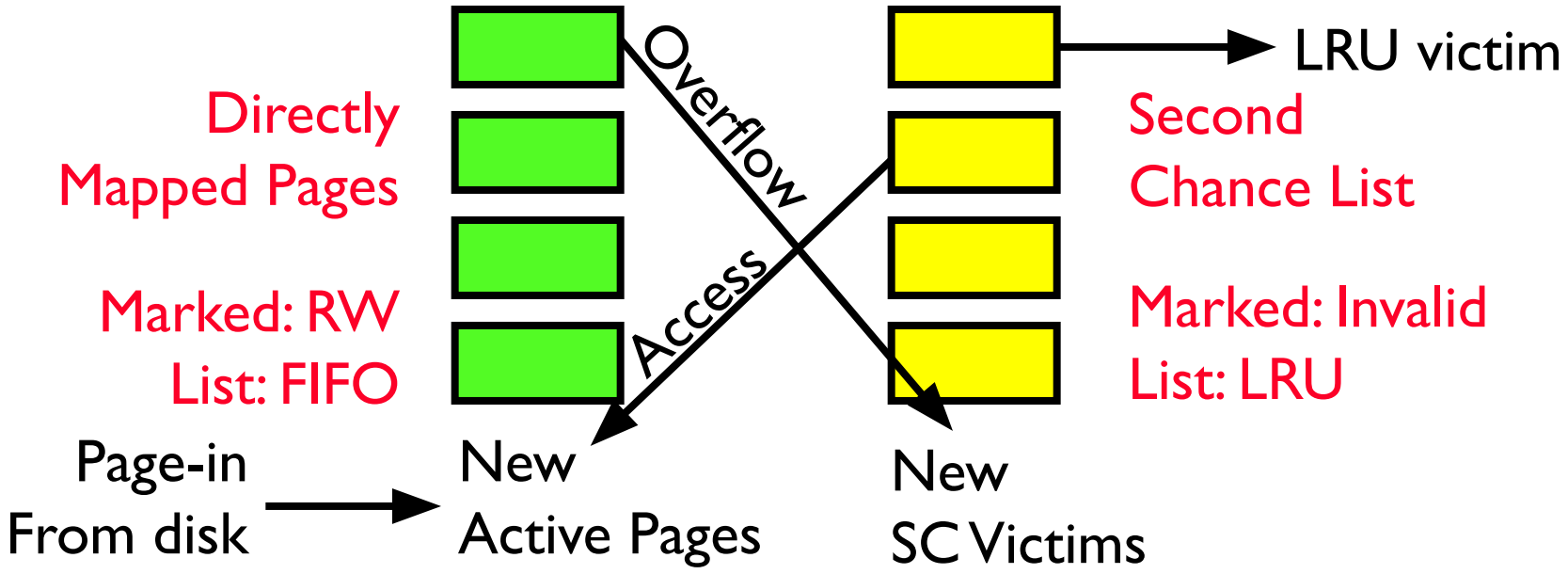
Clock Algorithms Variations

- Do we really need hardware-supported "modified" ("dirty") bit?
  - No. Can emulate it using read-only bit
    - » Need software "database" of which pages are allowed to be written (needed this anyway)
    - » We will tell MMU that pages have more restricted permissions than the actually do to force page faults (and allow us notice when page is written)
  - Algorithm (Clock-Emulated-M):
    - » Initially, mark all pages as read-only (W→0), even writable data pages. Further, clear all software versions of the "modified" bit → 0 (page not dirty)
    - » Writes will cause a page fault. Assuming write is allowed, OS sets software "modified" bit → 1, and marks page as writable (W→1).
    - » Whenever page written back to disk, clear "modified" bit → 0, mark read-only

Clock Algorithms Variations (continued)

- Do we really need a hardware-supported "use" bit?
  - No. Can emulate it similar to above (e.g. for read operation)
    - » Kernel keeps a "use" bit and "modified" bit for each page
  - Algorithm (Clock-Emulated-Use-and-M):
    - » Mark all pages as invalid, even if in memory.
      Clear emulated "use" bits → 0 and "modified" bits → 0 for all pages (not used, not dirty)
    - » Read or write to invalid page traps to OS to tell use page has been used
    - » OS sets "use" bit → 1 in software to indicate that page has been "used".
      Further:
      1) If read, mark page as read-only, W→0 (will catch future writes)
      2) If write (and write allowed), set "modified" bit → 1, mark page as writable (W→1)
    - » When clock hand passes, reset emulated "use" bit → 0 and mark page as invalid again
    - » Note that "modified" bit left alone until page written back to disk
- 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)

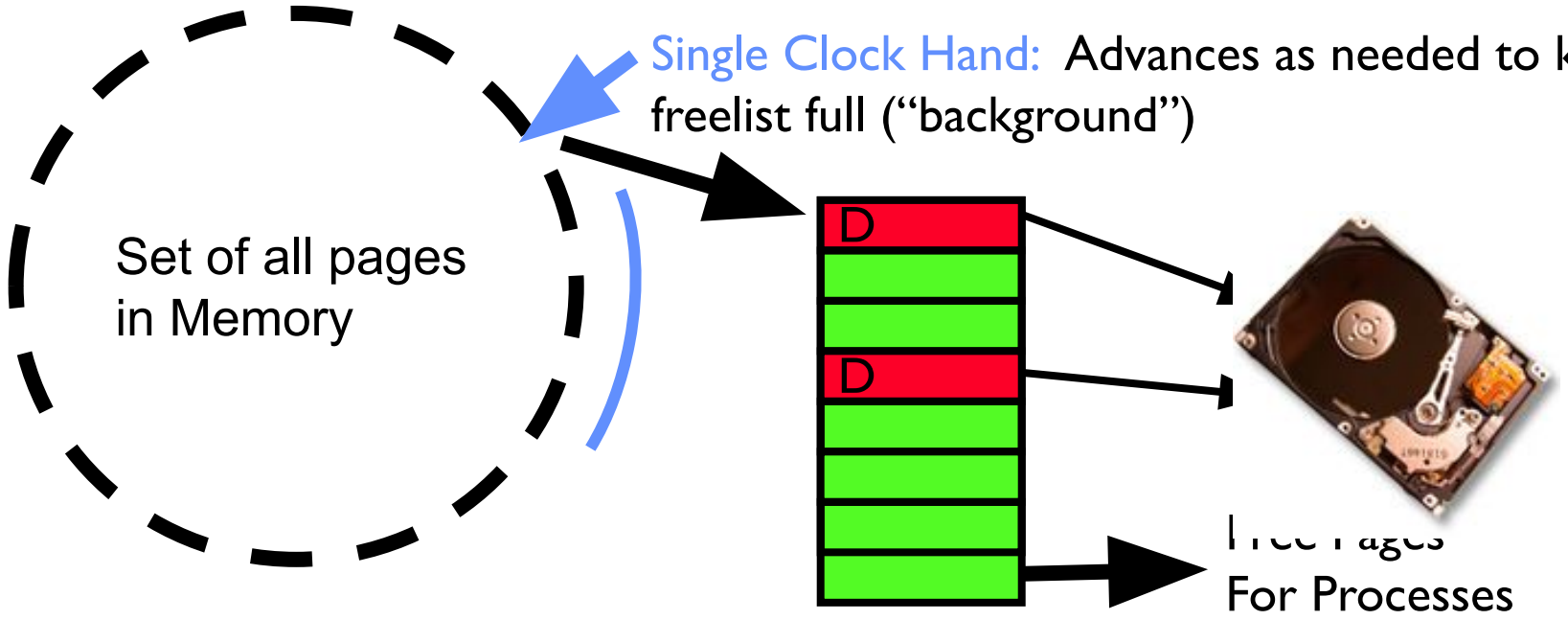


- Split memory in two: Active list (RW), SC list (Invalid)
- Access pages in Active list at full speed
- Otherwise, Page Fault
  - Always move overflow page from end of Active list to front of Second-chance list (SC) and mark invalid
  - Desired Page On SC List: move to front of Active list, mark RW
  - 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 ⇒ FIFO
  - If all ⇒ 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)
- With page translation, we can adapt to any kind of access the program makes
  - Later, we will show how to use page translation / protection to share memory between threads on widely separated machines
- History: The VAX architecture did not include a "use" bit.
  Why did that omission happen???
  - 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

Conclusion

- 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
- Working Set:
  - Set of pages touched by a process recently
  - Point of Replacement algorithms is to try to keep working set in memory
- 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
- Second-Chance List algorithm: Yet another approximate LRU
  - Divide pages into two groups, one of which is truly LRU and managed on page faults.