

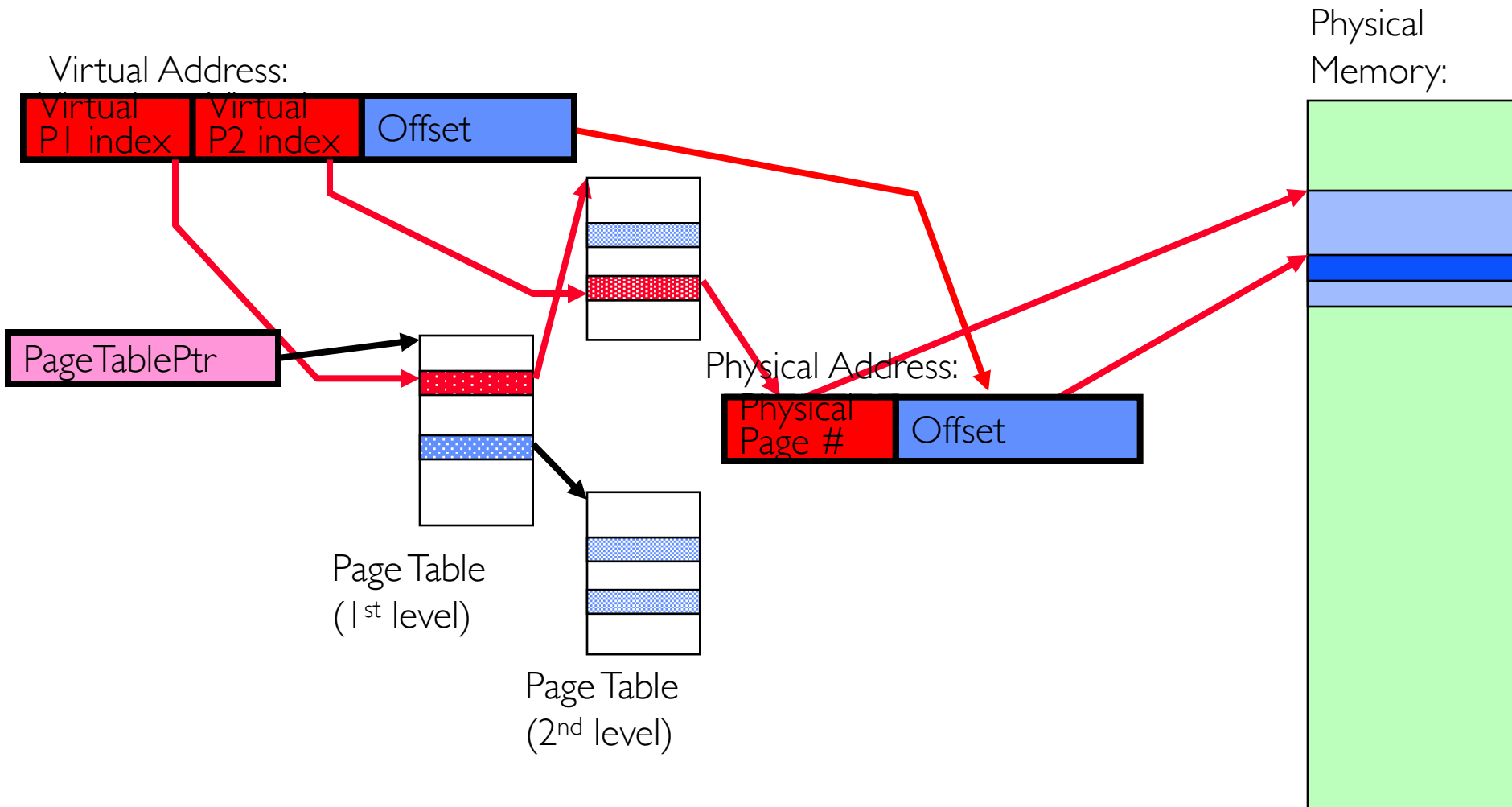
Operating Systems (Honor Track)

Memory 4: Demand Paging

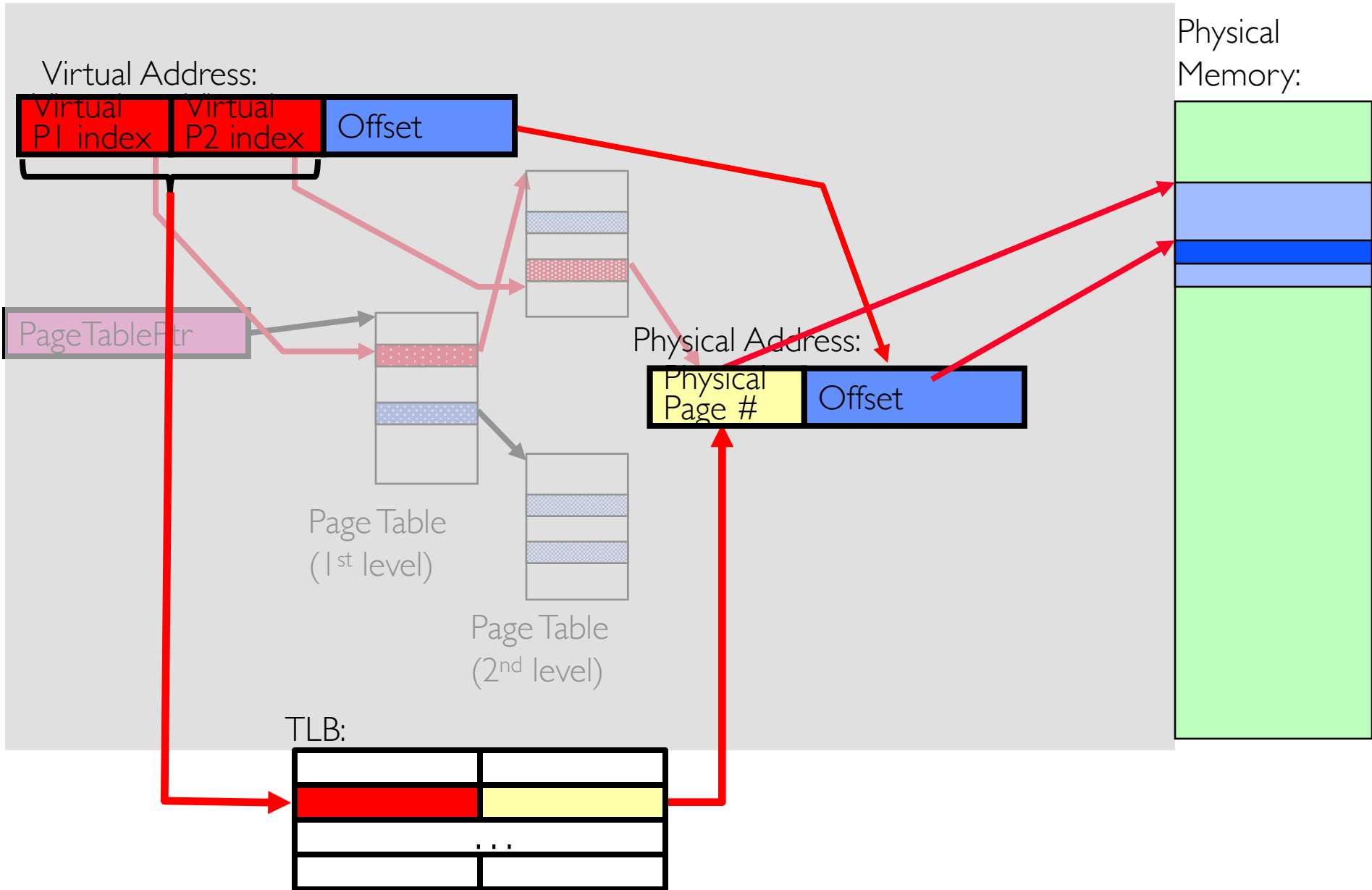
Xin Jin

Spring 2026

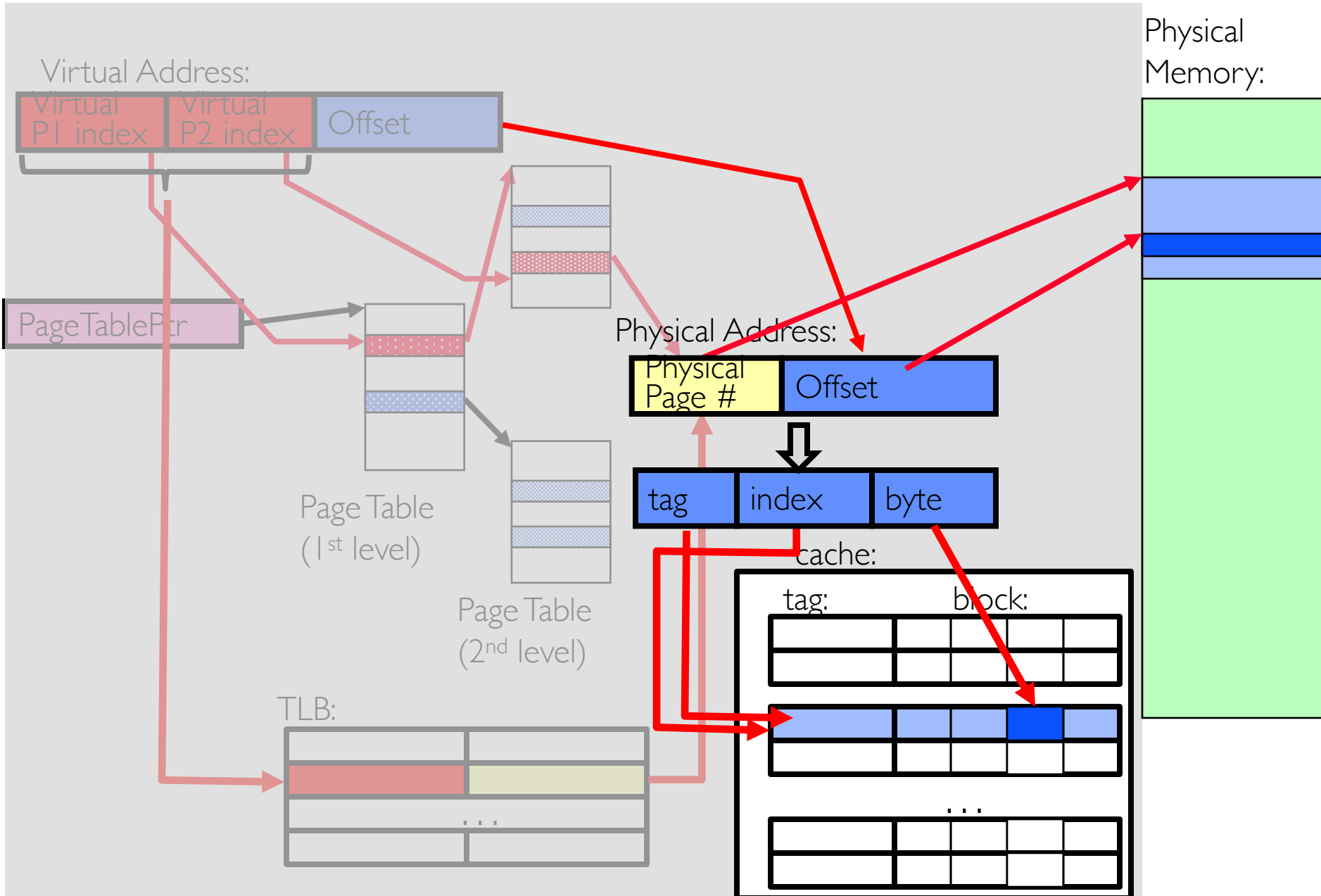
Recap: Putting Everything Together: Address Translation



Recap: Putting Everything Together: TLB



Recap: Putting Everything Together: Cache



Recap: Demand Paging as Caching, ...

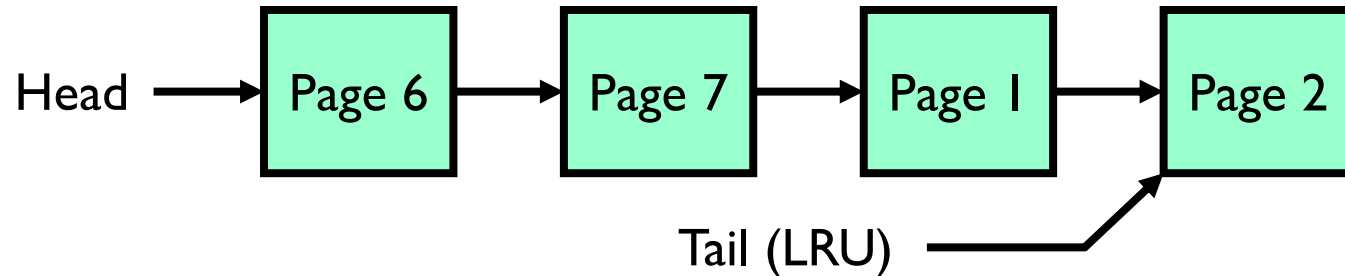
- What “block size”? - 1 page (e.g., 4 KB)
- What “organization” i.e., direct-mapped, set-associative, fully-associative?
 - Fully associative since arbitrary mapping
- How do we locate a page?
 - First check TLB, then page-table traversal
- What is page replacement policy? (i.e., LRU, Random...)
 - 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!

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:	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):

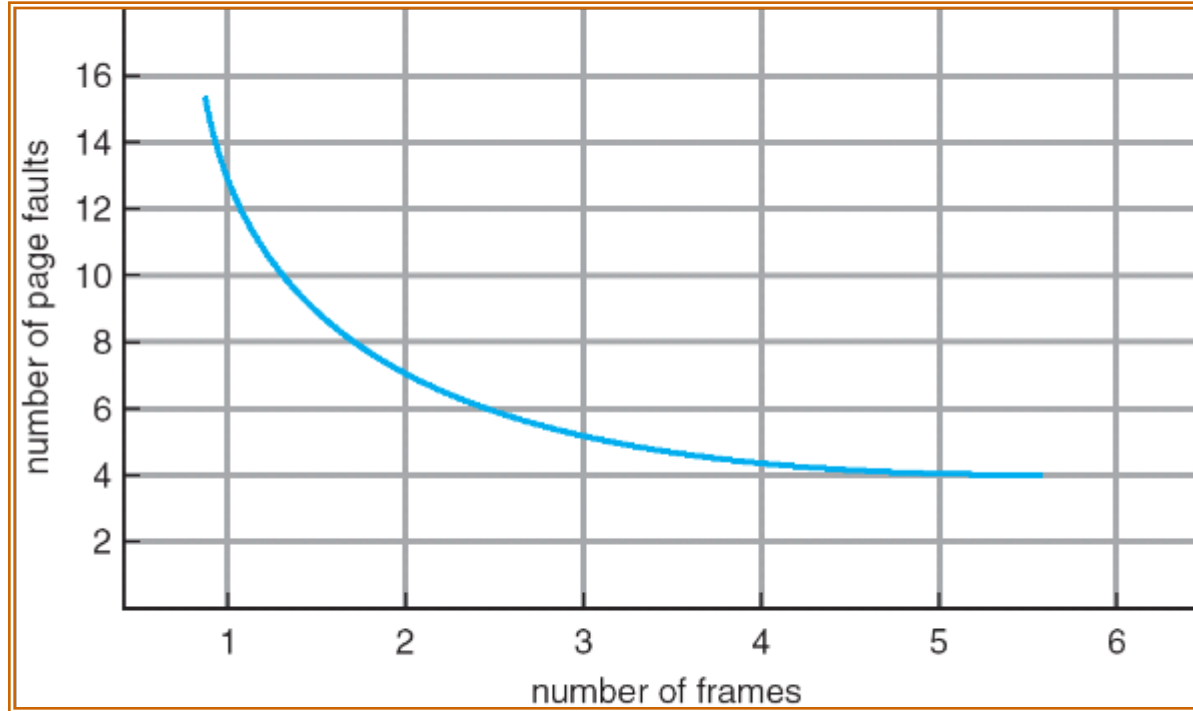
Ref:	A	B	C	D	A	B	C	D	A	B	C	D
Page:												
1	A			D			C			B		
2		B			A			D			C	
3			C			B			A			D

– 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 (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	

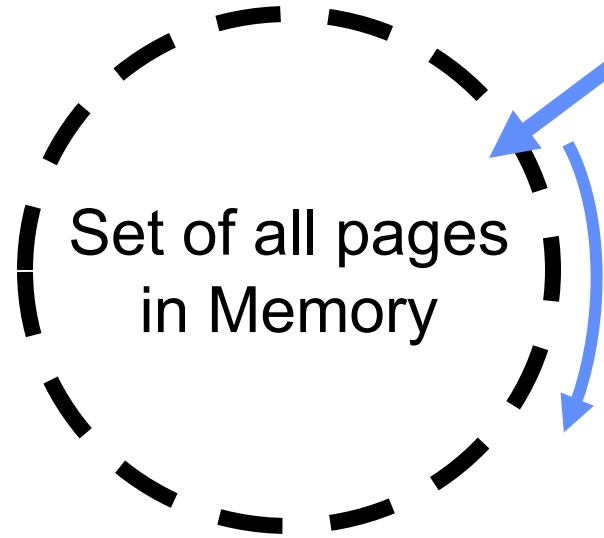
9 page faults

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



Single Clock Hand:

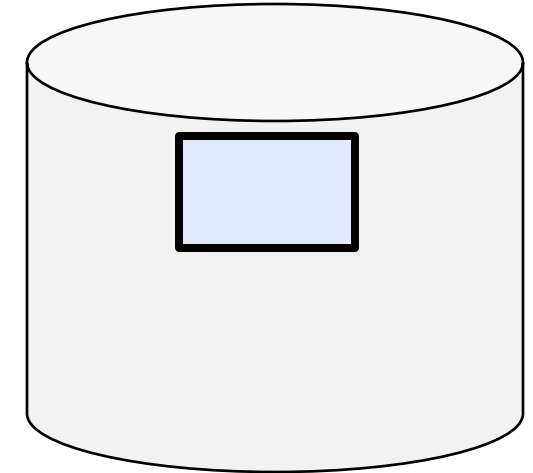
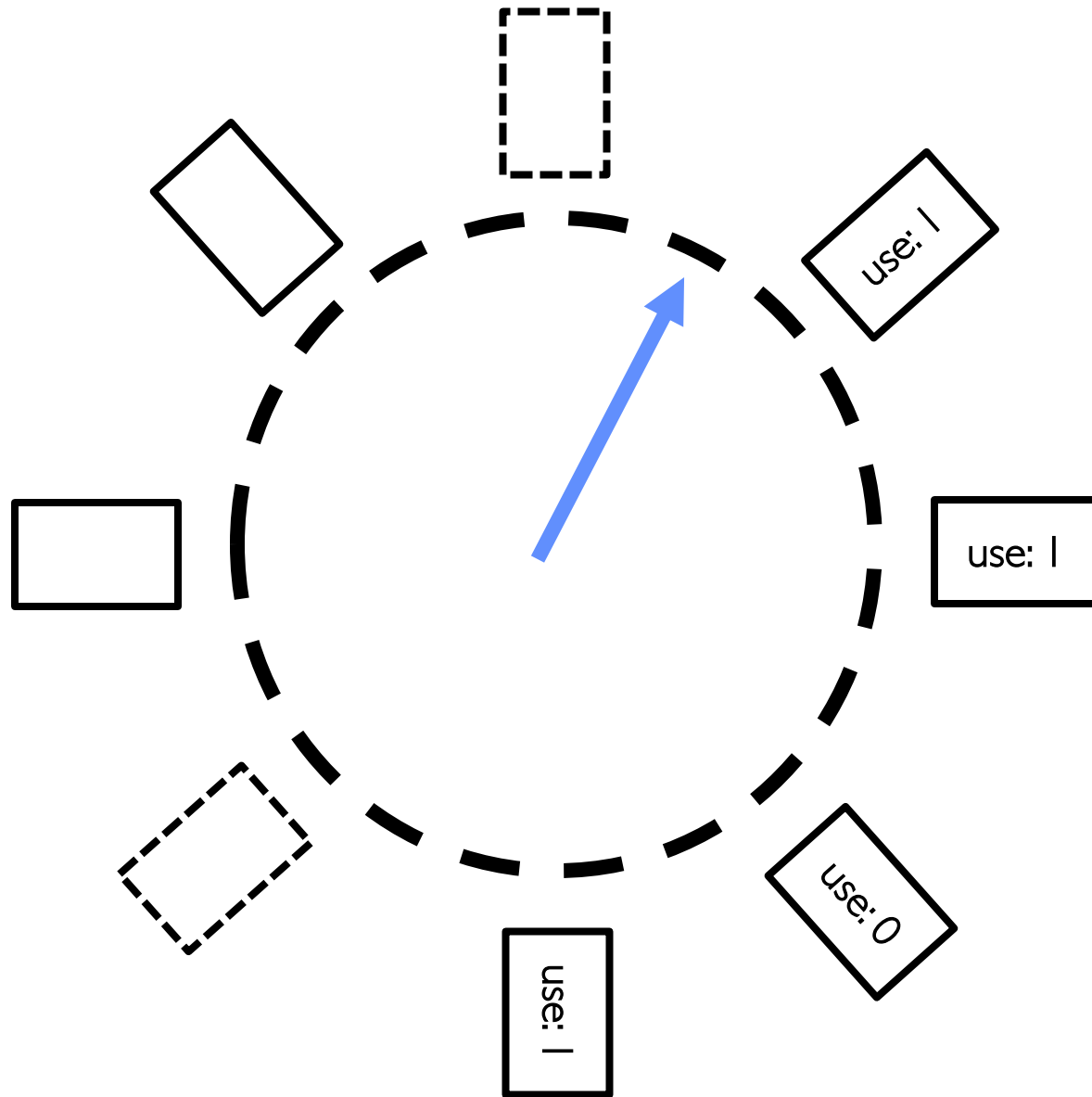
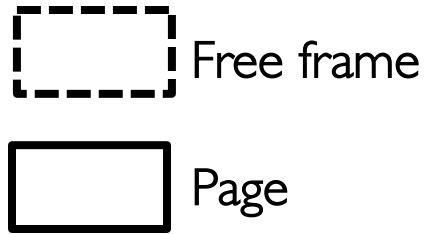
Advances only on page fault!

Check for pages not used recently

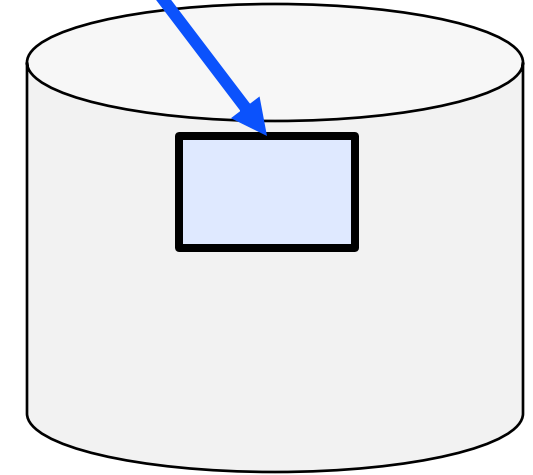
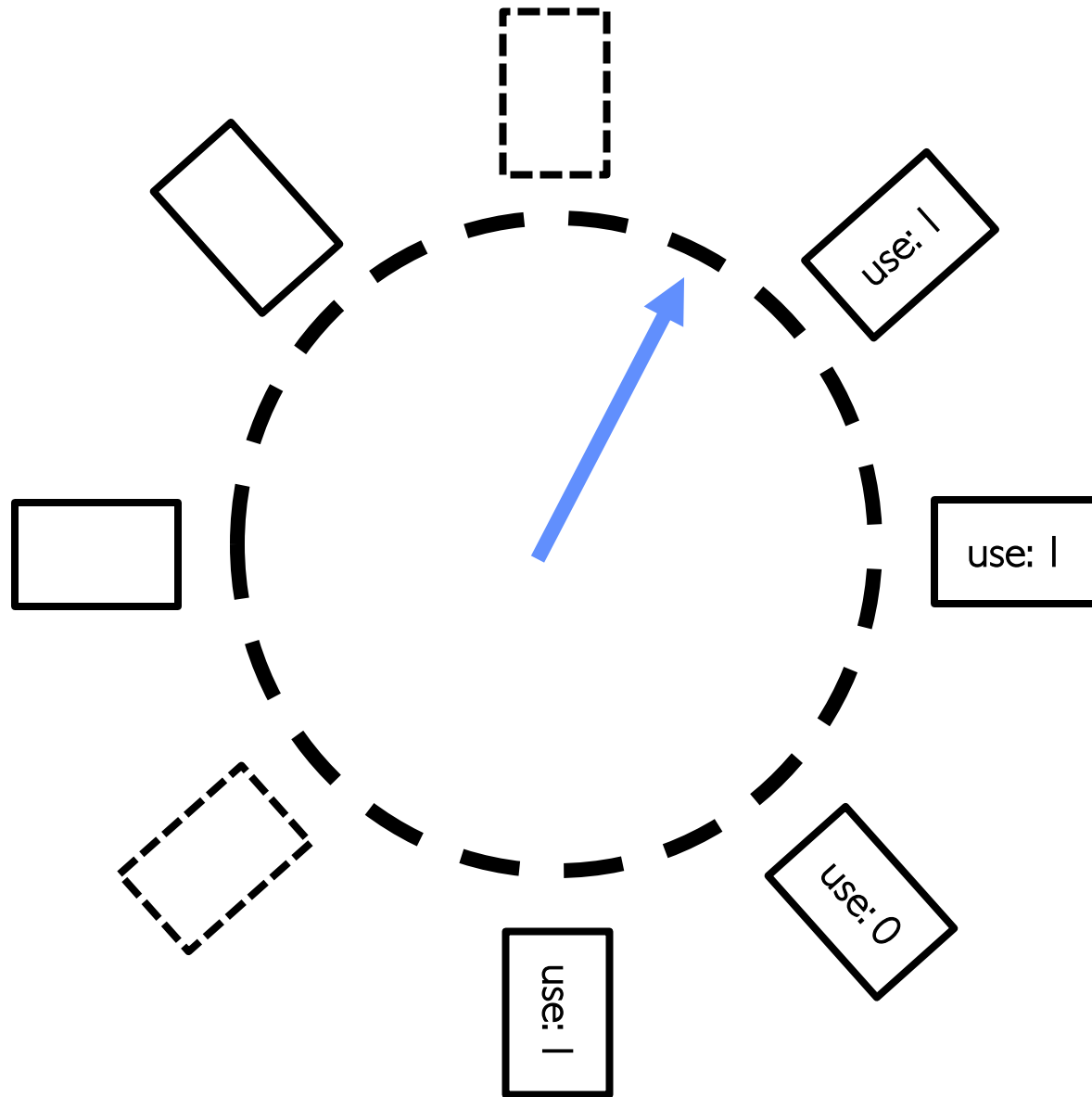
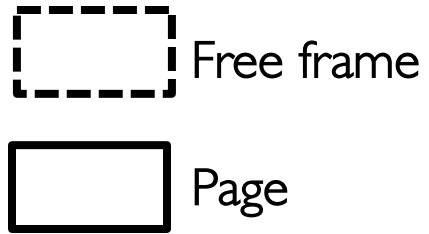
Mark pages as not used recently

- **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
 - On page fault:
 - » Advance clock hand (not real time)
 - » Check **use** bit: 1→ used recently; clear and leave alone
0→ selected candidate for replacement

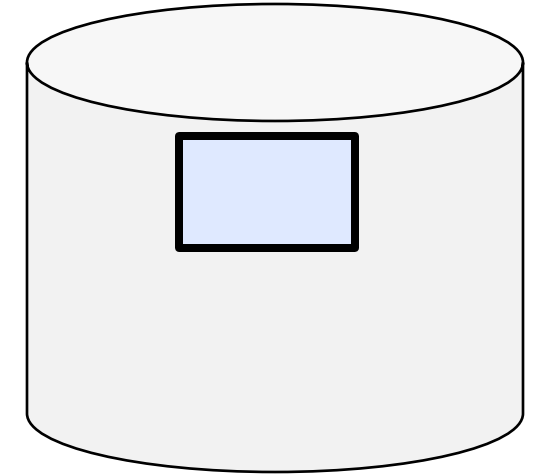
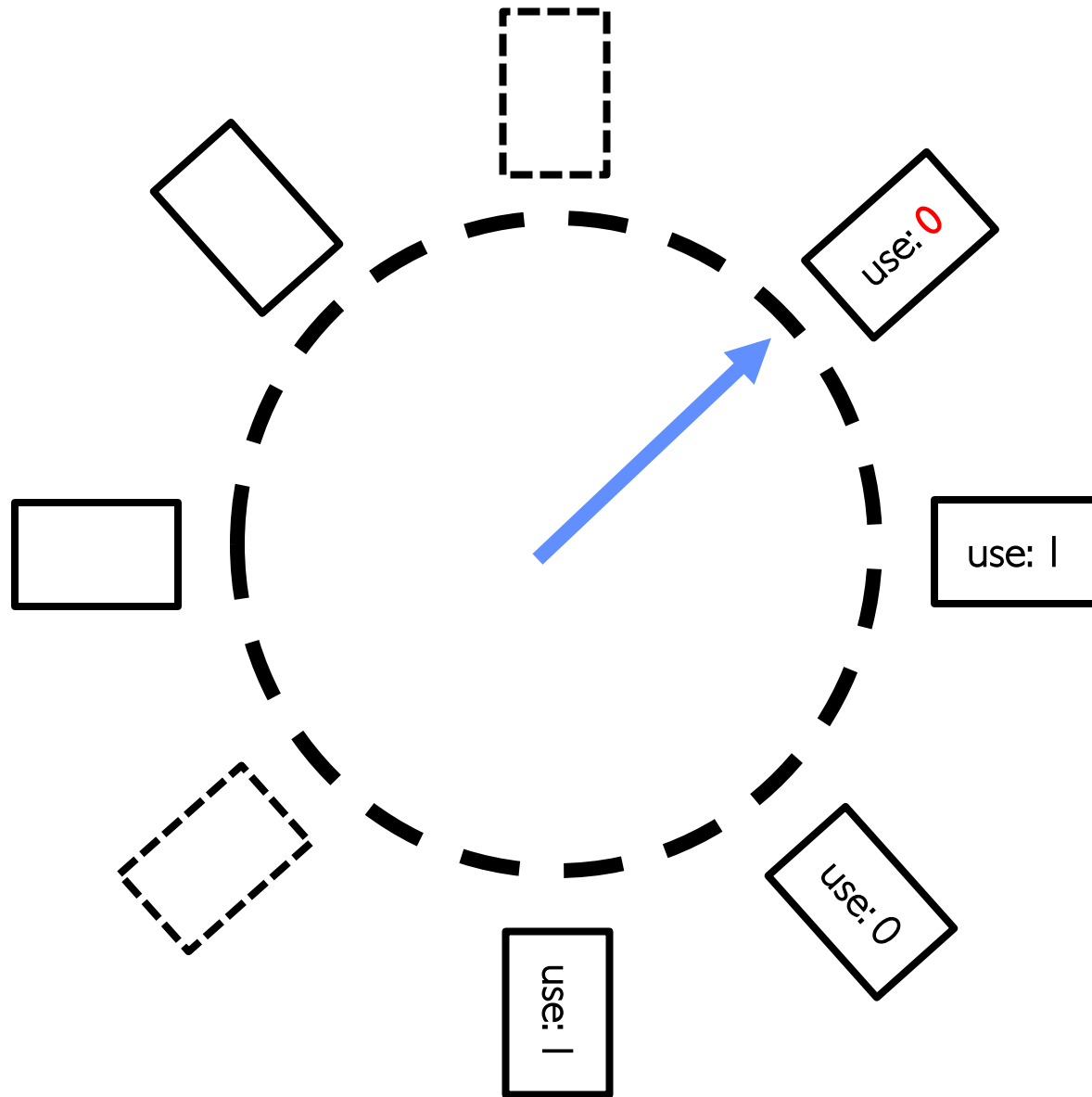
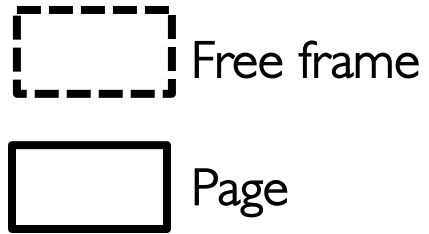
Clock Algorithm Example



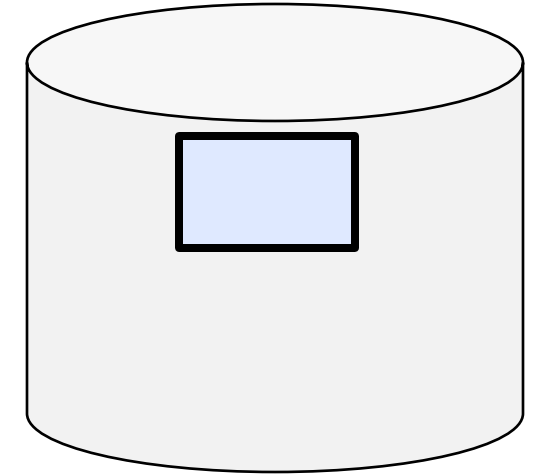
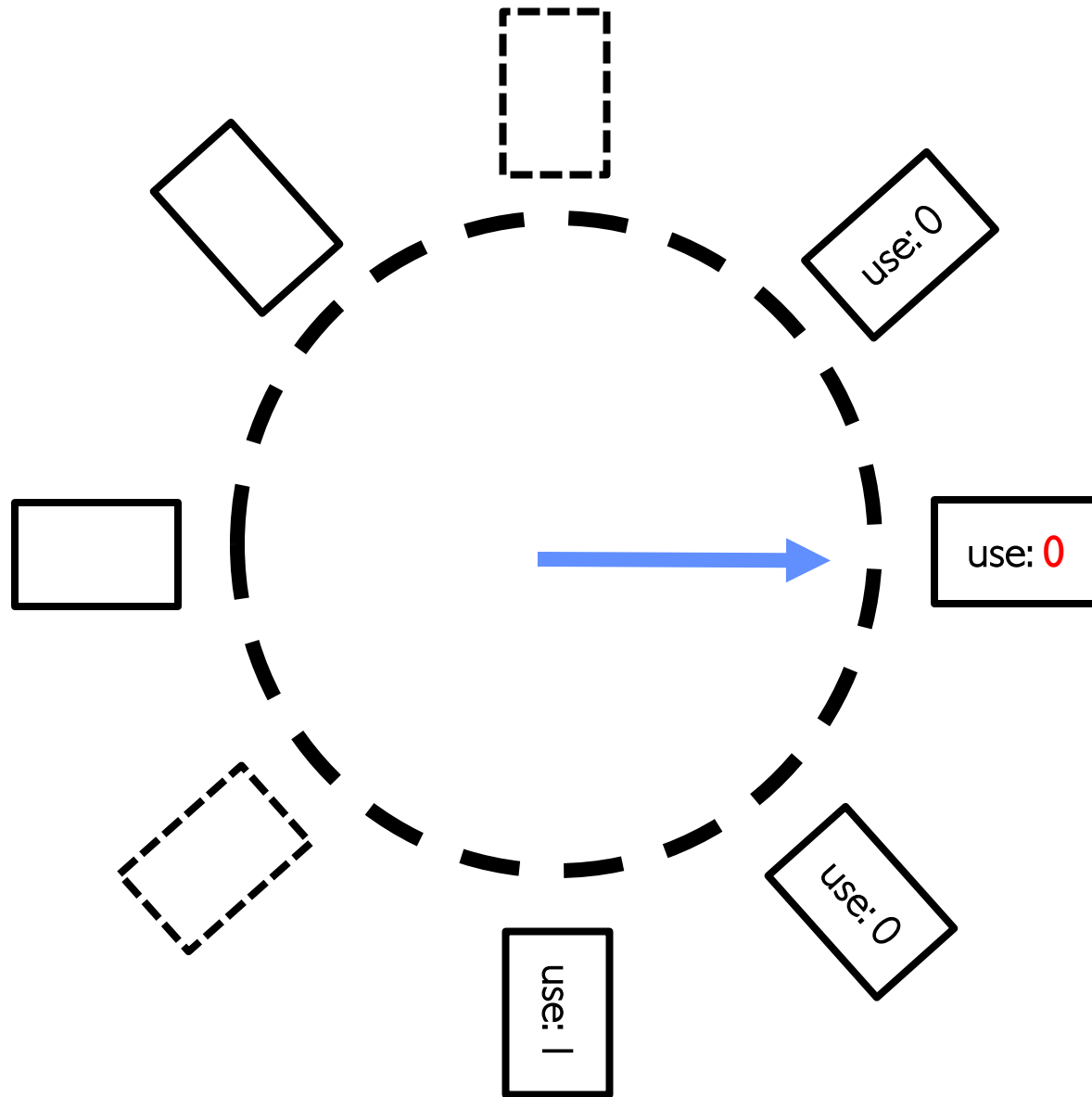
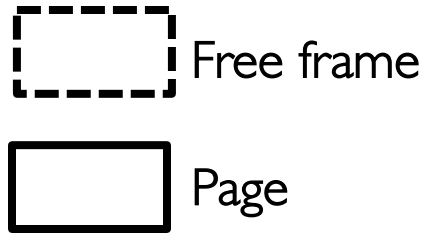
Clock Algorithm Example: Page Fault



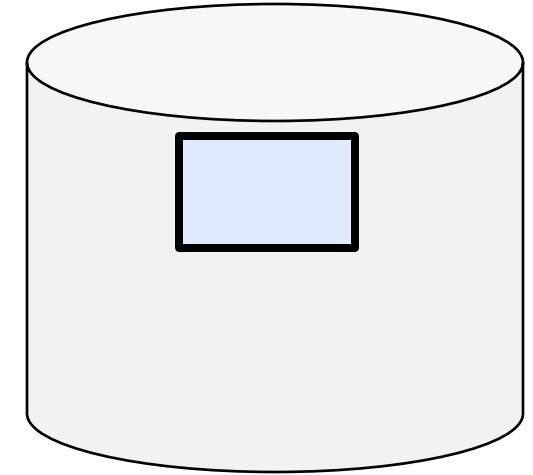
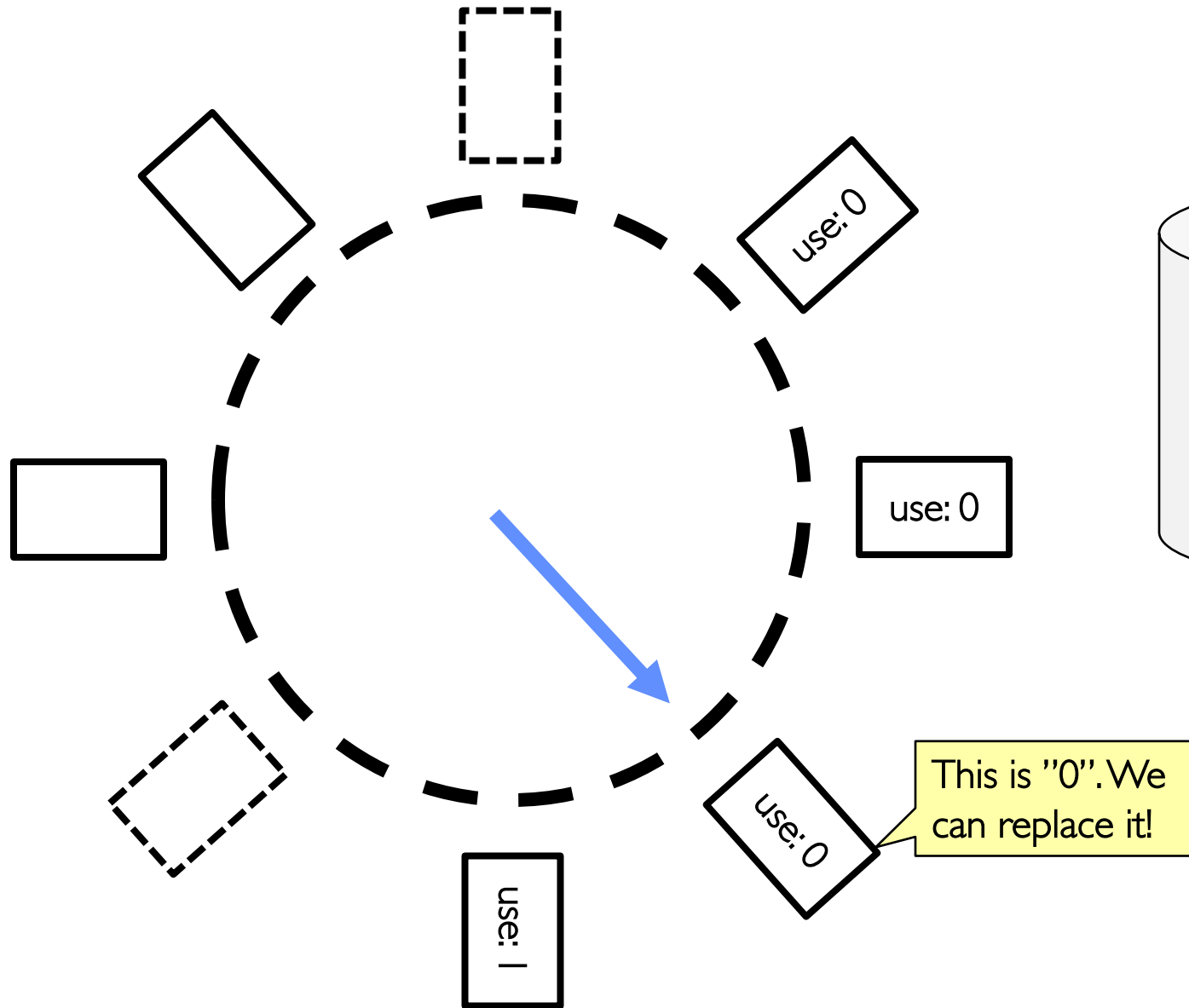
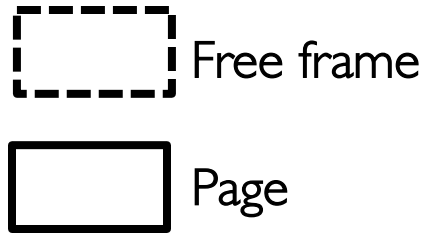
Clock Algorithm Example: Page Fault



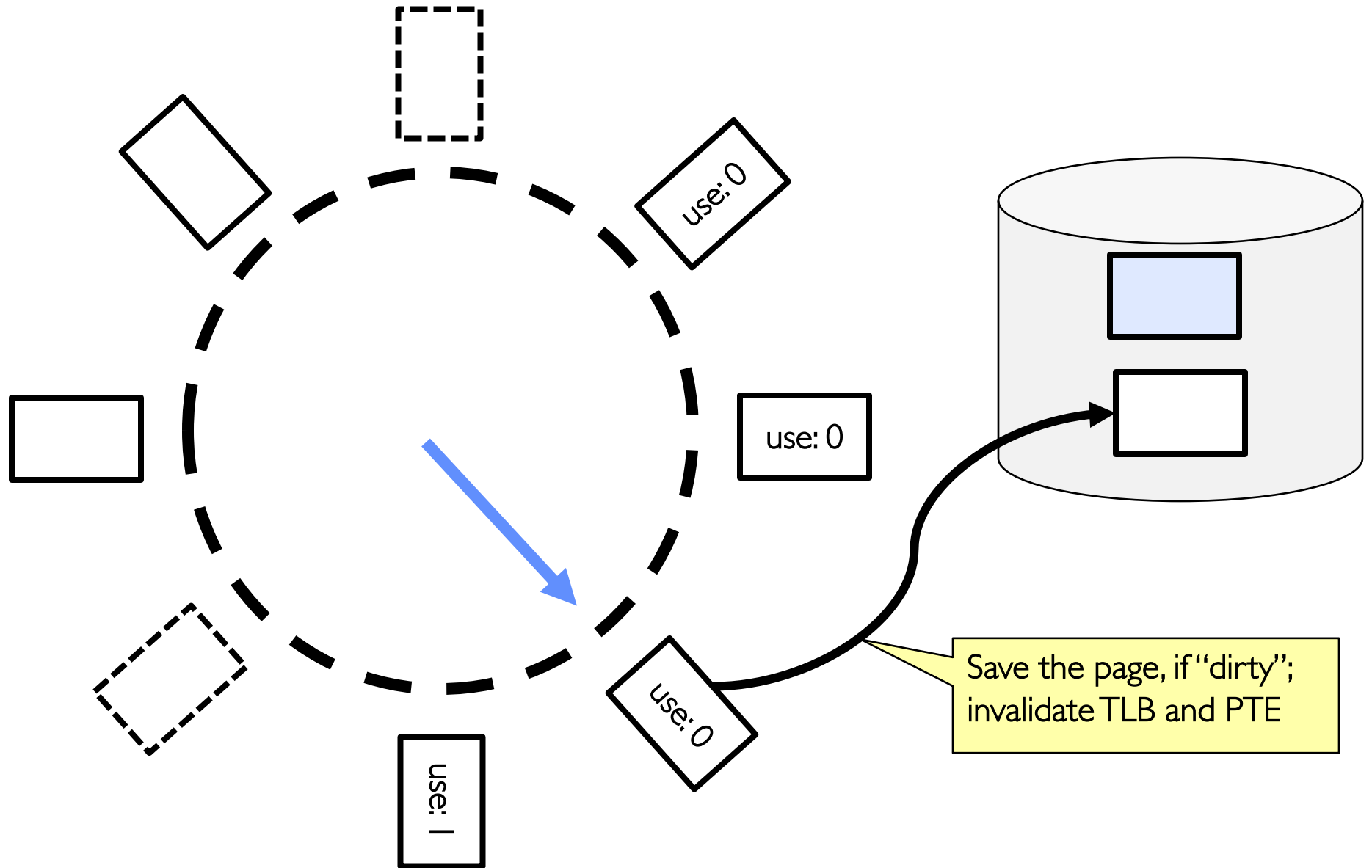
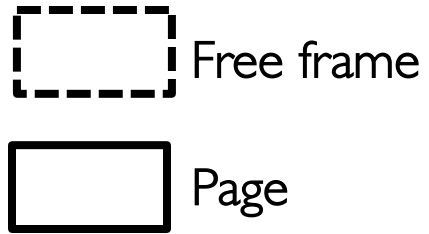
Clock Algorithm Example: Page Fault



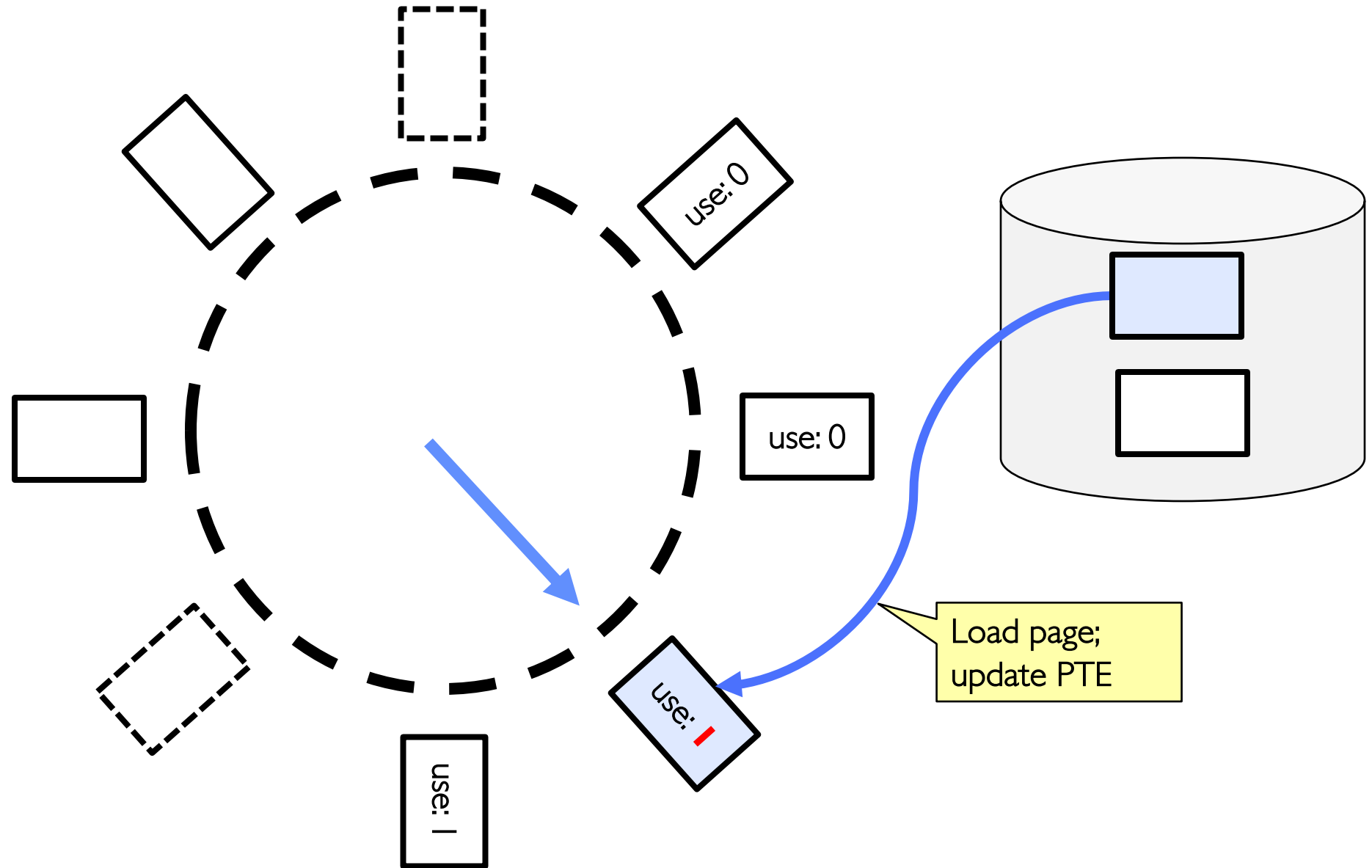
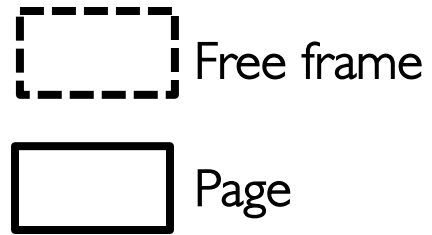
Clock Algorithm Example: Page Fault



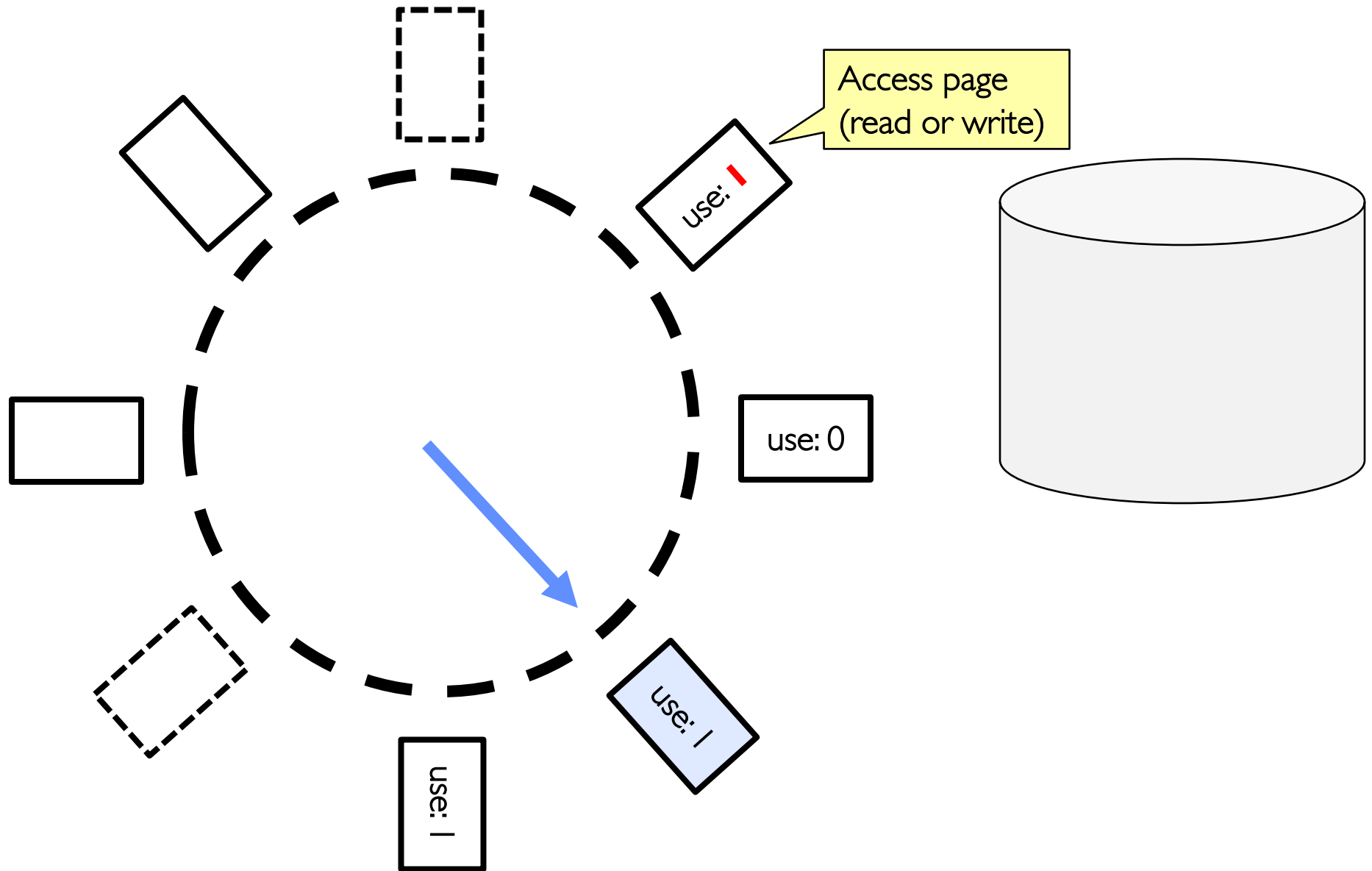
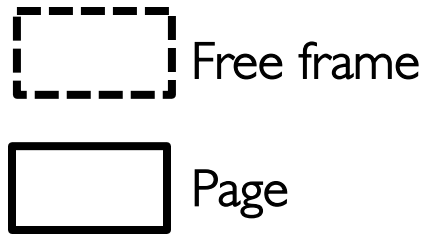
Clock Algorithm Example: Page Fault



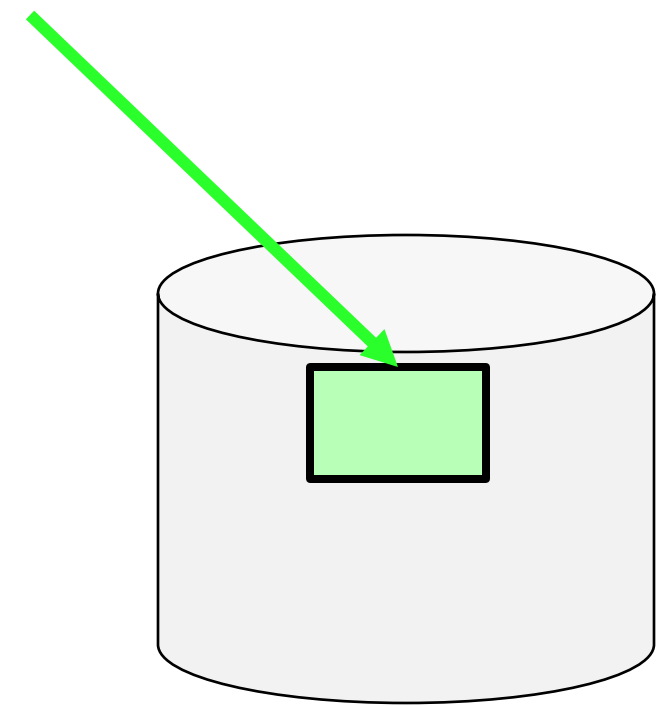
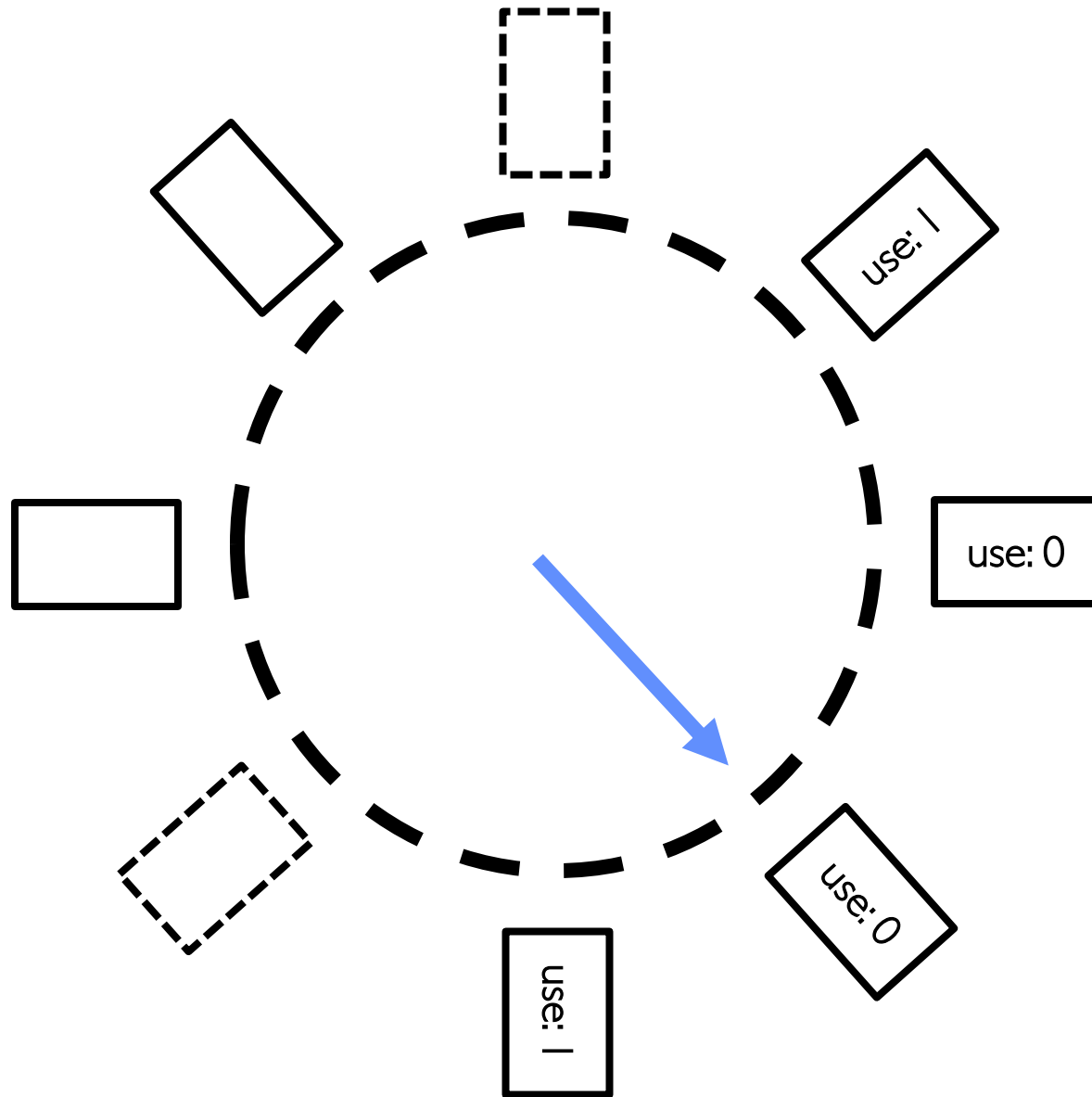
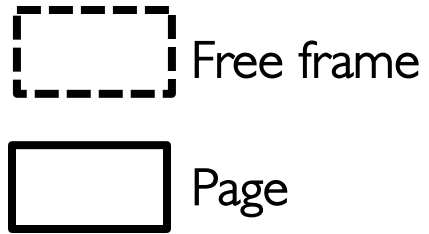
Clock Algorithm Example: Page Fault



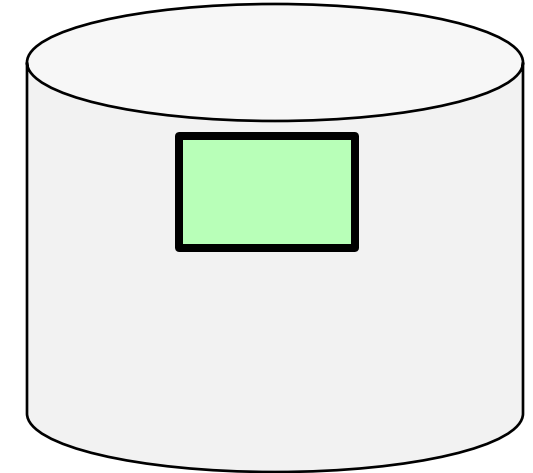
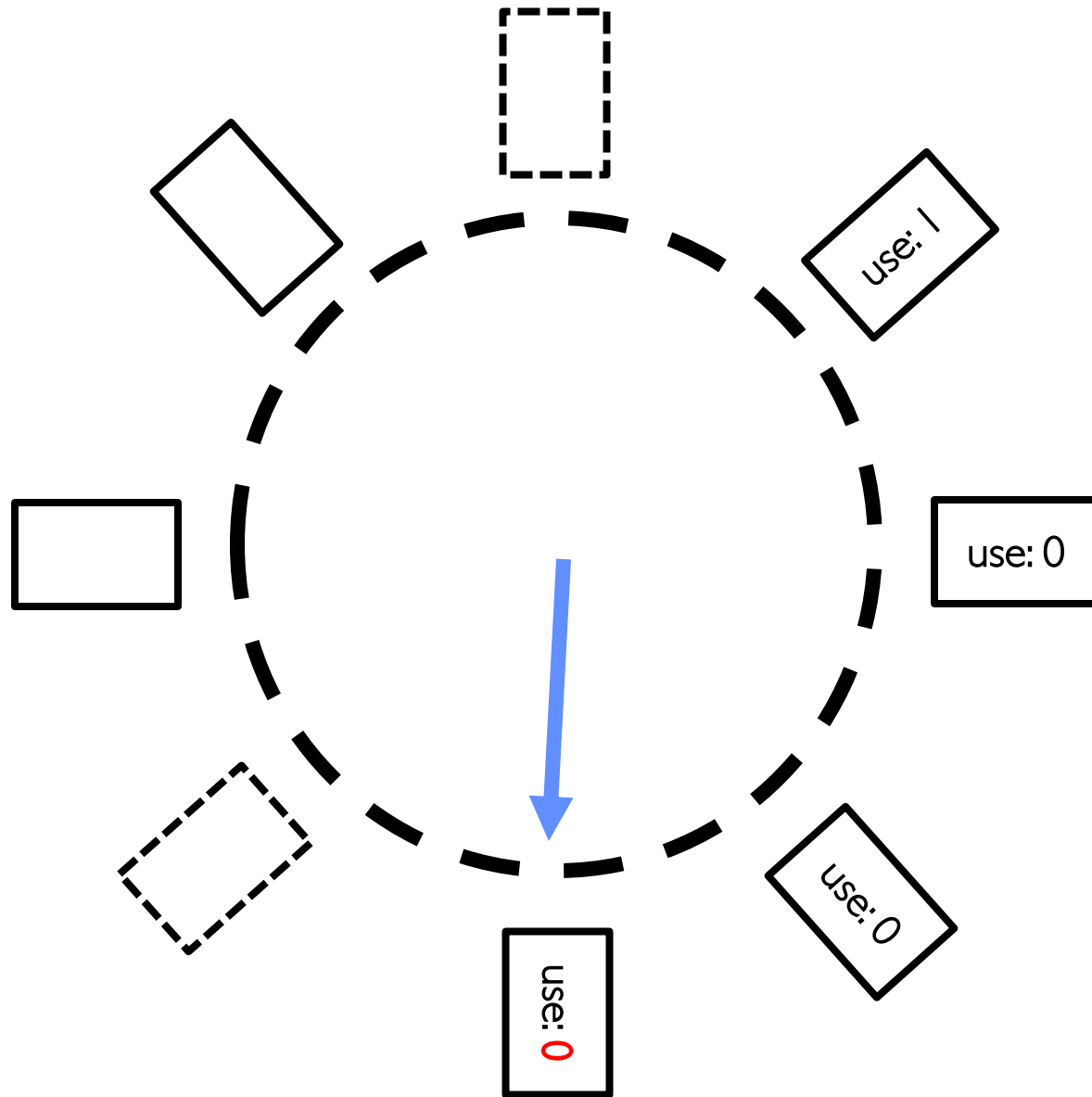
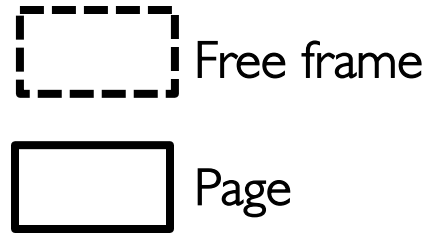
Clock Algorithm Example



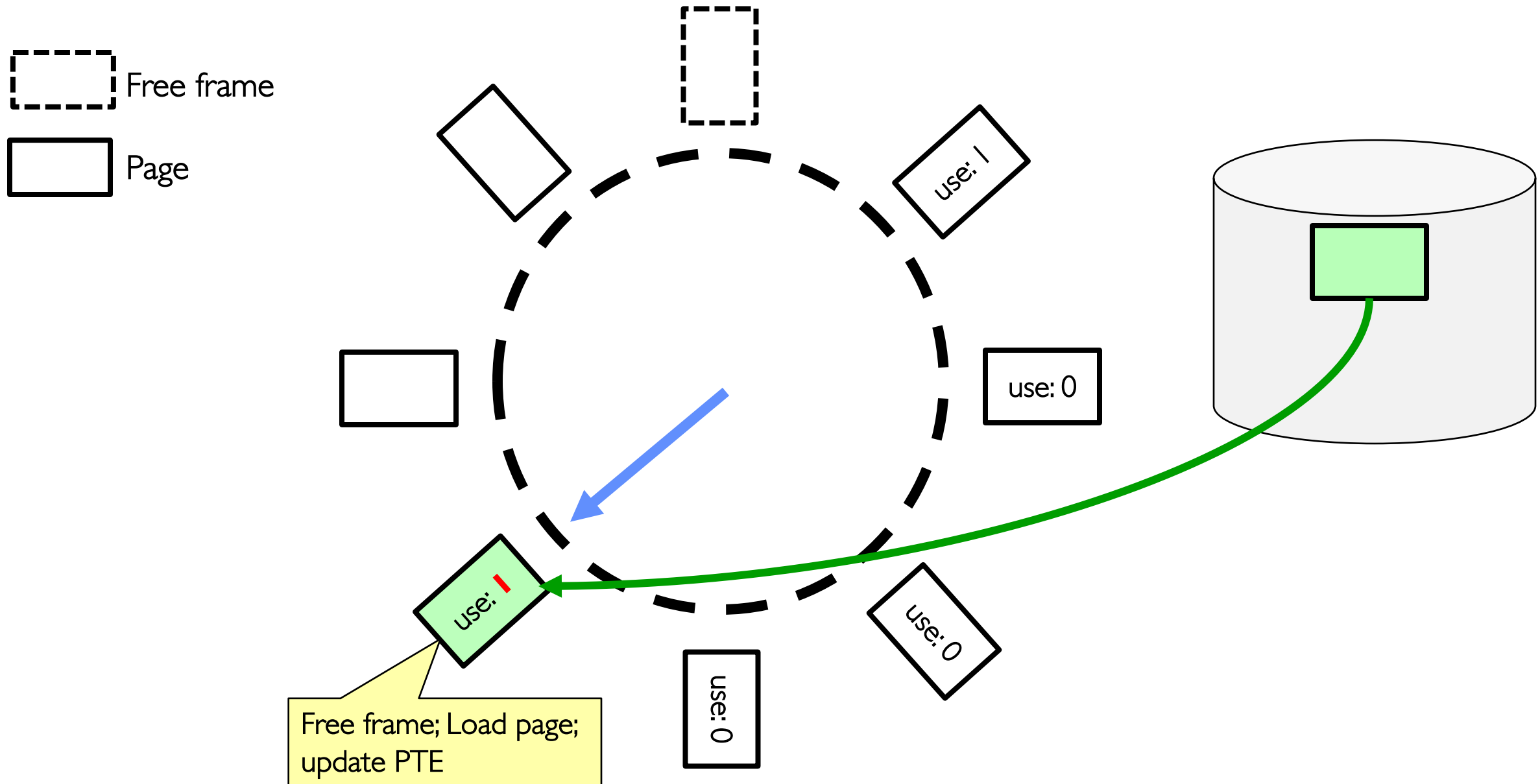
Clock Algorithm Example: Another Page Fault



Clock Algorithm Example: Another Page Fault

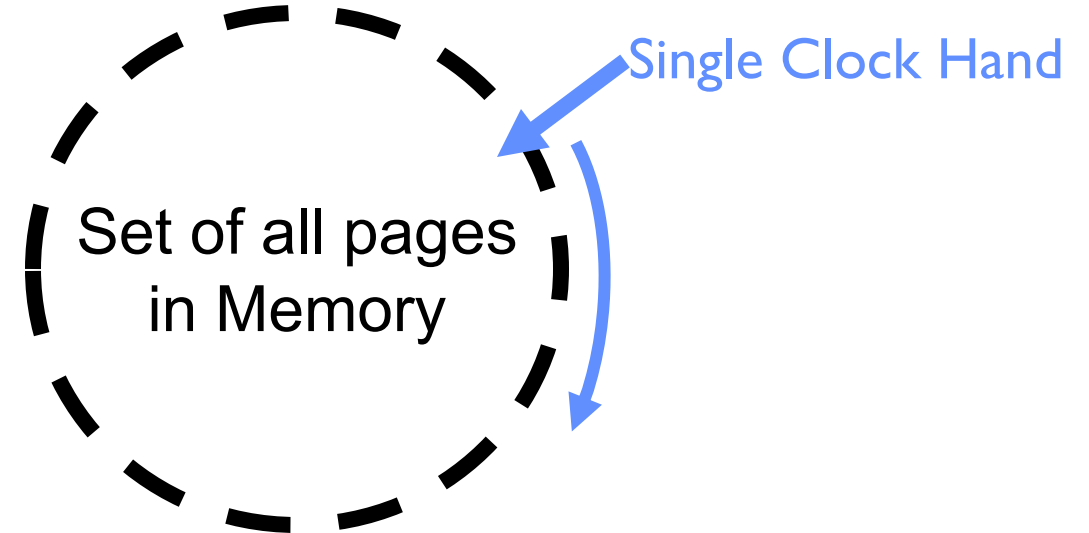


Clock Algorithm Example: Another Page Fault



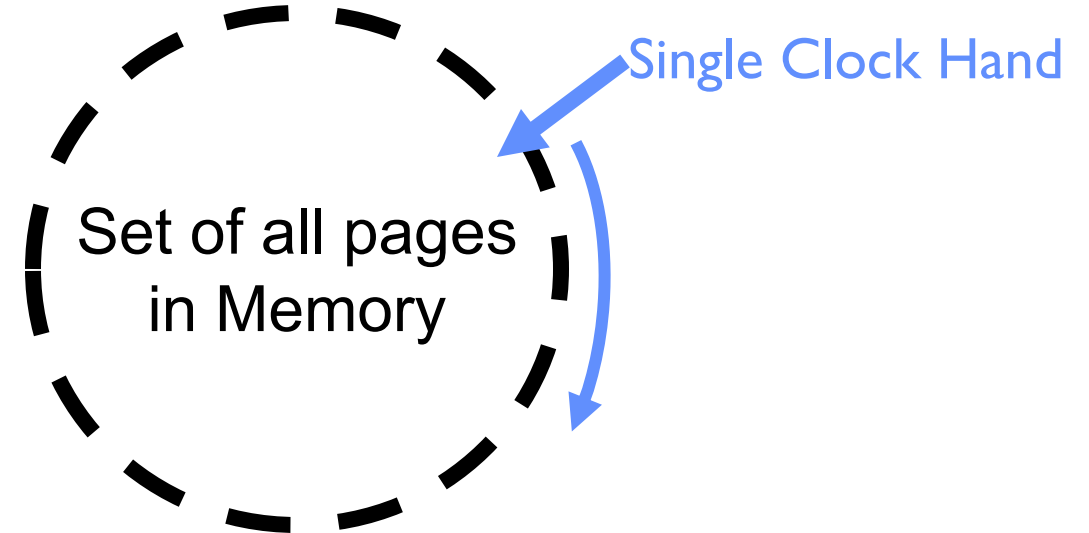
Group Discussion: Clock Algorithm

- Will always find a page or loop forever?
- What if hand is moving slowly?
 - Good sign or bad sign?
- What if hand is moving quickly?
 - Good sign or bad sign?



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 is moving slowly?
 - Good sign or bad sign?
 - » Not many page faults or find page quickly
- What if hand is moving quickly?
 - Good sign or bad sign?
 - » Lots of page faults 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 “**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 $N=1$)

Group Discussion

- Topic: Clock algorithm variations
 - Do we really need a hardware-supported “modified” bit?
 - Do we really need a hardware-supported “use” bit?
- 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

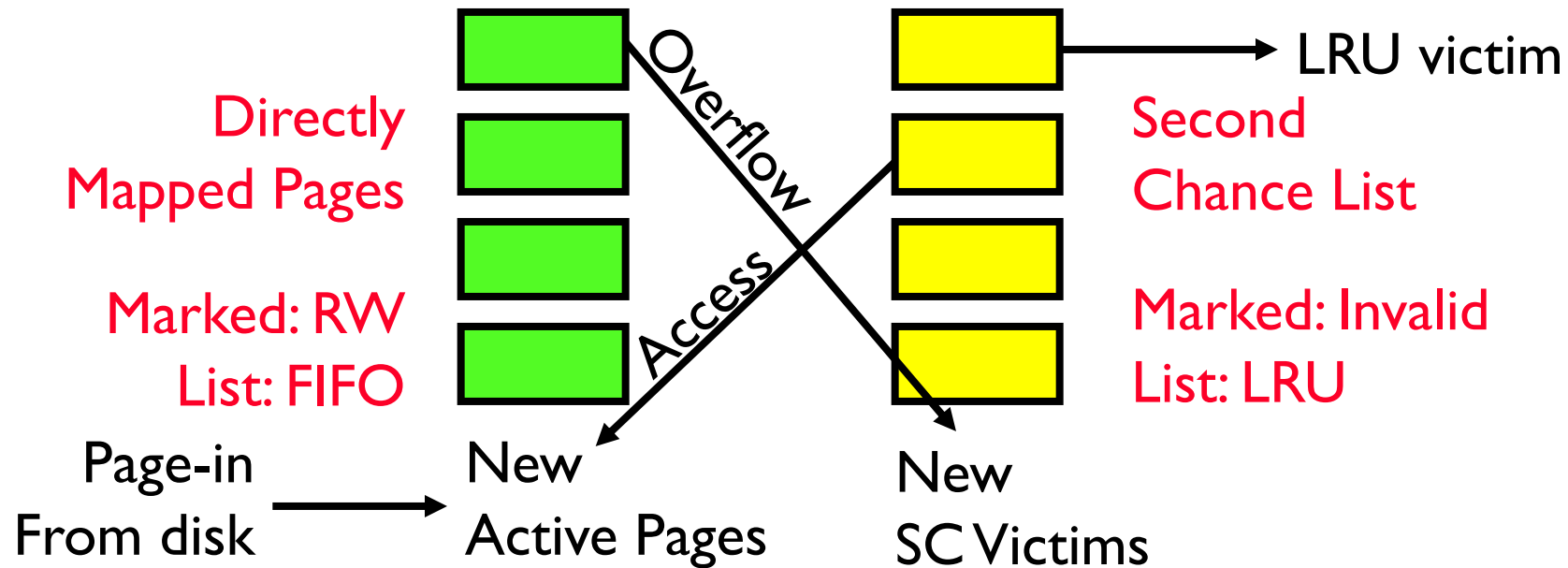
Clock Algorithms Variations

- Do we really need hardware-supported “modified” bit?
 - No. Can emulate it using read-only bit
 - » Need software DB of which pages are allowed to be written (needed this anyway)
 - » We will tell MMU that pages have more restricted permissions than they actually do to force page faults (and allow us notice when page is written)
 - Algorithm (Clock-Emulated-Modified):
 - » Initially, mark all pages as read-only ($W \rightarrow 0$), even writable data pages. Further, clear all software versions of the “modified” bit $\rightarrow 0$ (page not dirty)
 - » Writes will cause a page fault. Assuming write is allowed, OS sets software “modified” bit $\rightarrow 1$, and marks page as writable ($W \rightarrow 1$).
 - » Whenever page written back to disk, clear “modified” bit $\rightarrow 0$, 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-Modified):
 - » Mark all pages as invalid, even if in memory.
Clear emulated “use” bits $\rightarrow 0$ and “modified” bits $\rightarrow 0$ for all pages (not used, not dirty)
 - » Read or write to invalid page traps to OS to tell page has been used
 - » OS sets “use” bit $\rightarrow 1$ in software to indicate that page has been “used”.
Further:
 - 1) If read, mark page as read-only, $W \rightarrow 0$ (will catch future writes)
 - 2) If write (and write allowed), set “modified” bit $\rightarrow 1$, mark page as writable ($W \rightarrow 1$)
 - » When clock hand passes, reset emulated “use” bit $\rightarrow 0$ 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)

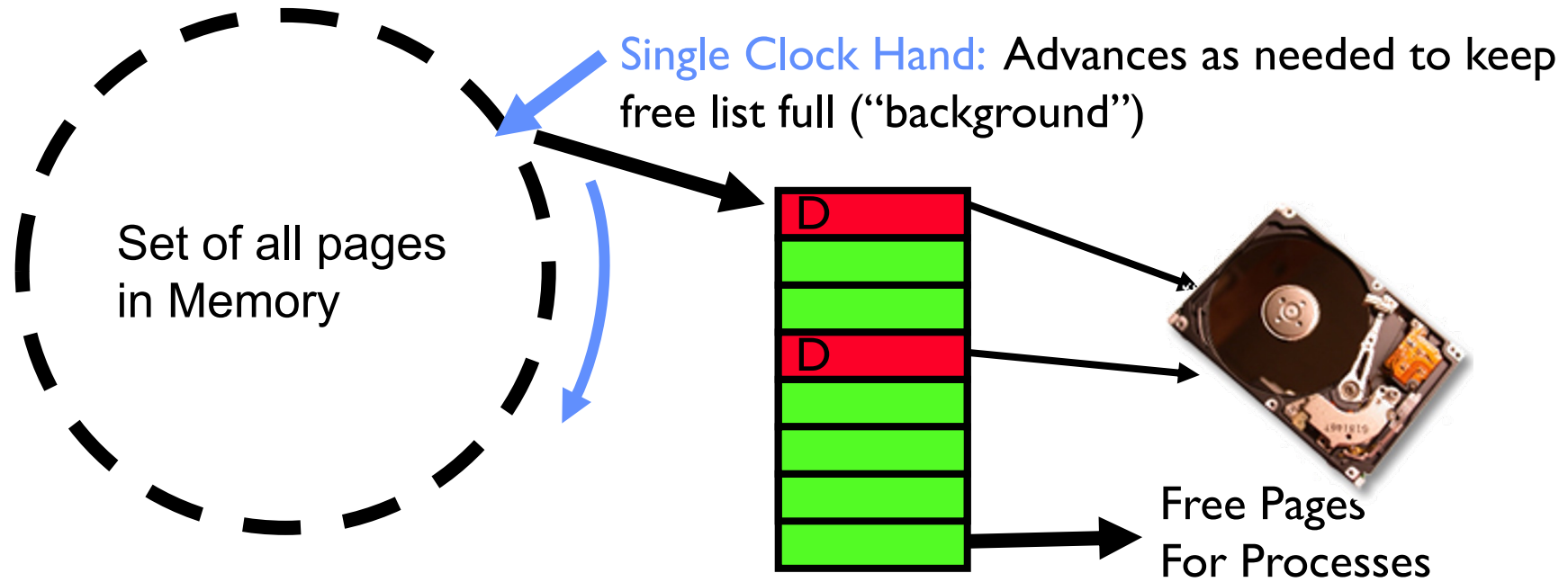


- Split memory in two: Active list (RW), SC list (Invalid)
- Access pages in Active list at full speed
- Otherwise, Page Fault
 - Always move overflow page from end of Active list to front of Second-chance list (SC) and mark invalid
 - Desired Page in SC List: move it to front of Active list, mark it RW
 - Not in 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. Compared to FIFO:
 - Pro: Few disk accesses (page only goes to disk if unused for a long time)
 - Con: Increased overhead trapping to OS (software / hardware tradeoff)
- 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
 - Free list filled in background by Clock algorithm or other technique (“Pageout daemon”)
 - 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

Reverse Page Mapping (Sometimes called “Coremap”)

- When evicting a page frame, how to know which PTEs to invalidate?
 - Hard in the presence of shared pages (forked processes, shared memory, ...)
- Reverse mapping mechanism 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: Object-based reverse mapping
 - » Link together memory region descriptors instead (much coarser granularity)
 - » E.g., program code and files mapped in with `mmap()`

Allocation of Page Frames (Memory Pages)

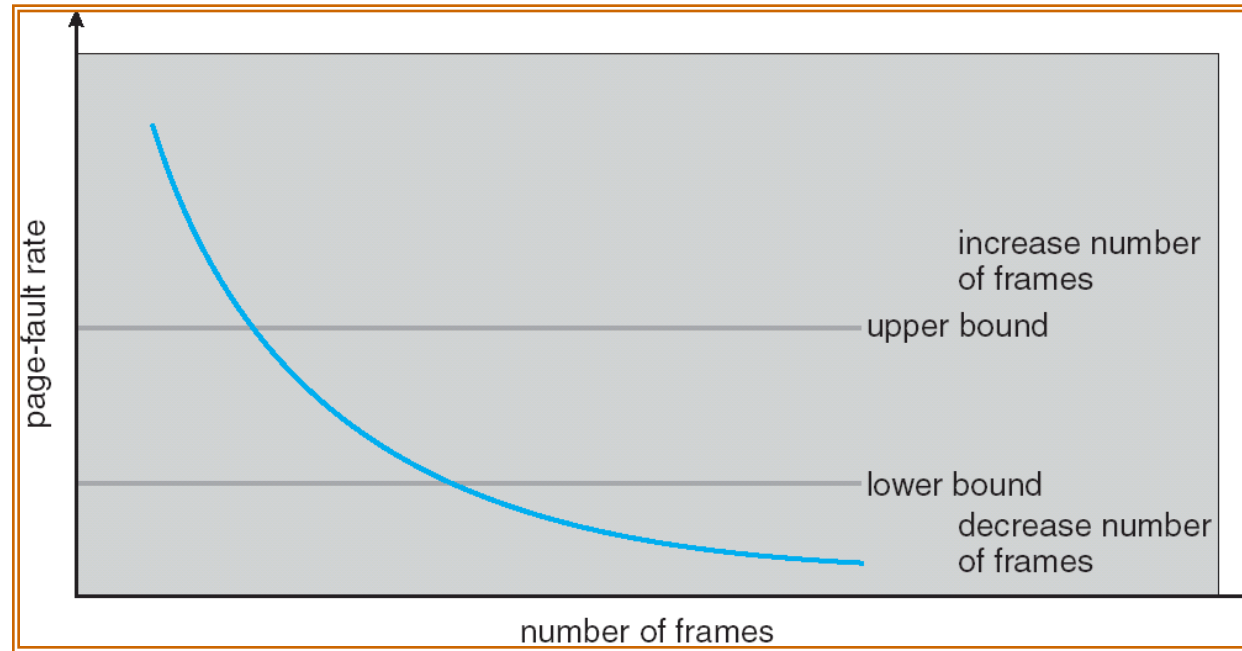
- How do we allocate memory among different processes?
 - Does every process get the same fraction of memory? Different fractions?
 - Should we completely swap some processes out of memory?
- Each process needs *minimum* number of pages
 - Want to make sure that all processes **that are loaded into memory** 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 physical frames in the system
 - a_i = (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

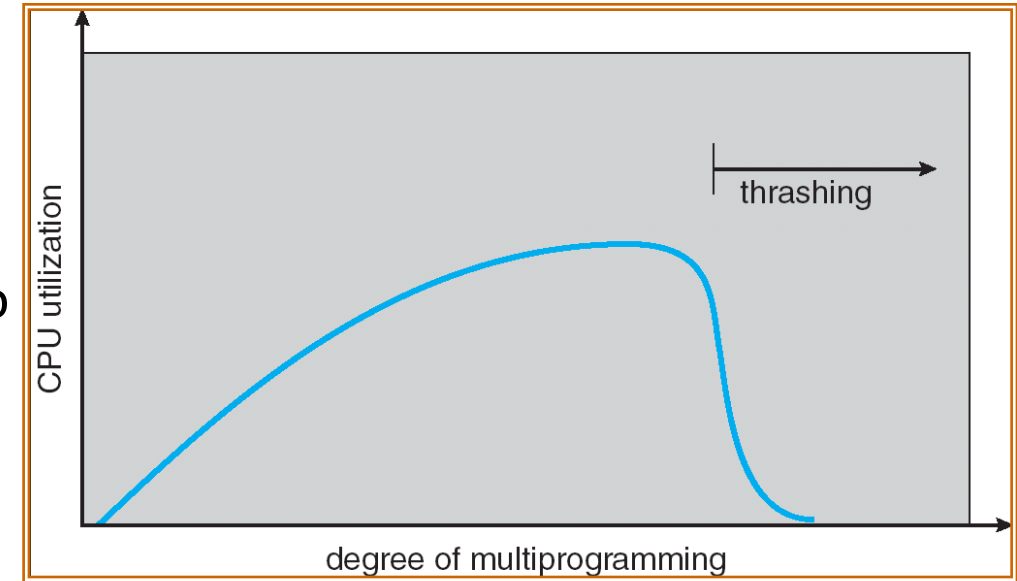
- Can we reduce capacity misses by dynamically changing the number of pages/application?



- 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?

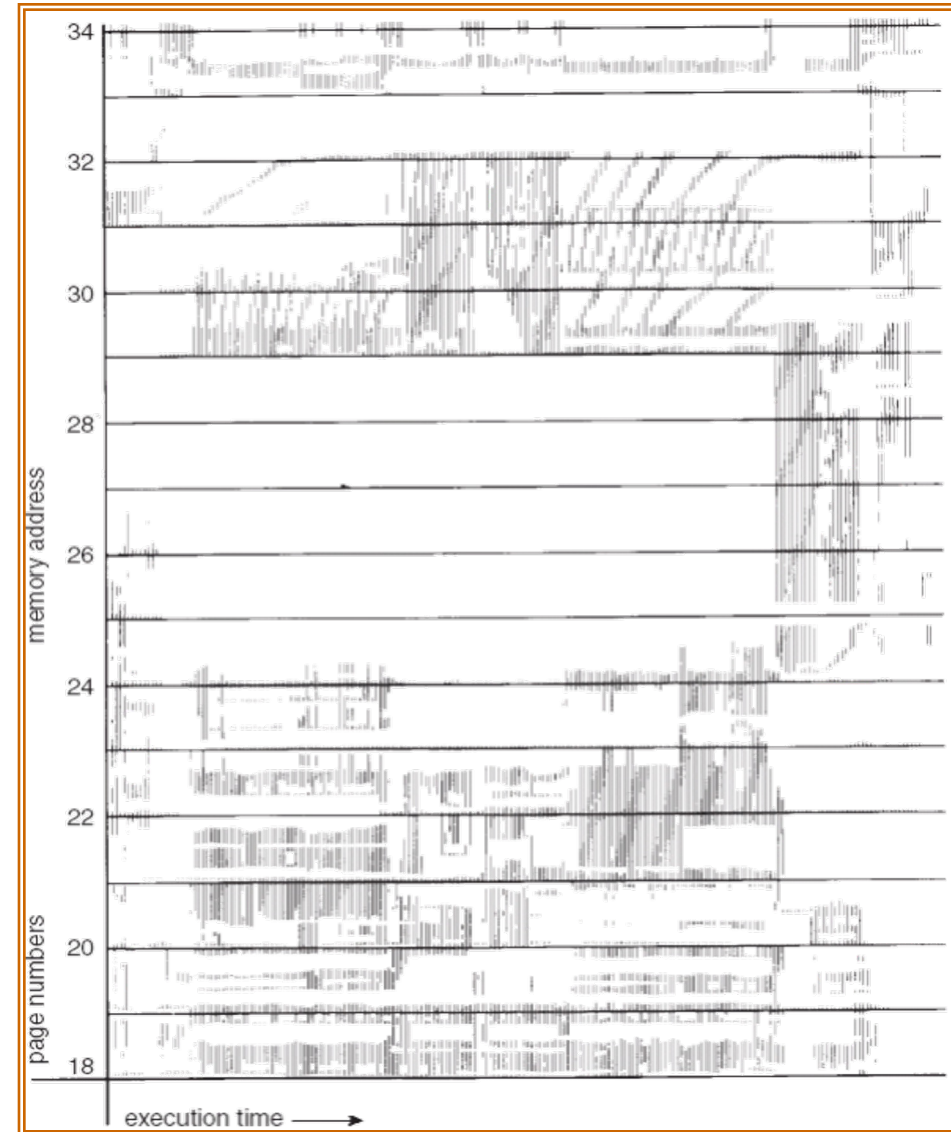
Thrashing

- 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 with little or no actual progress
- Questions:
 - How do we detect Thrashing?
 - What is best response to Thrashing?

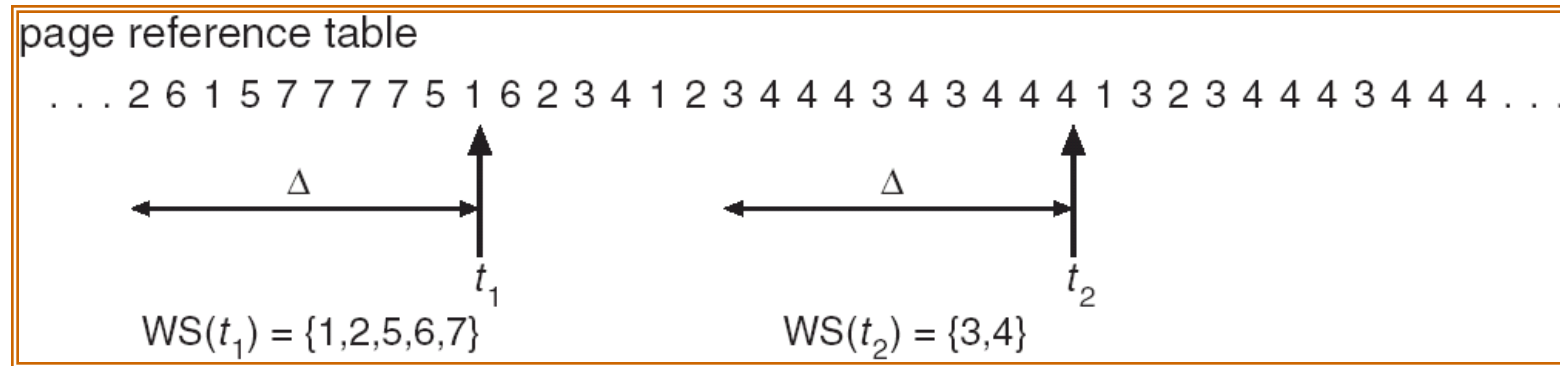


Locality In A Memory-Reference Pattern

- 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 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

Summary

- 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.
- 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