

# Page Replacement

(and other virtual memory policies)

Kevin Webb

Swarthmore College

March 19, 2024

# Today's Goals

- Making virtual memory “virtual”: incorporating disk backing.
- Explore page replacement policies for disk swapping.
- Which processes should get more memory? Which should have their pages swapped?
- Throughout: comparisons to caching.

# User Perspective

- Average user doesn't care about "address spaces" or memory sizes
- User might say:
  - I want all of my programs to be able to run at the same time.
  - I don't want to worry about running out of memory.
- If OS does nothing / has no virtual memory:
  - Best we can do is give them all of the physical memory.
  - Is that enough? Recall that VAS size can be larger than PAS...

# Multiprogramming, Revisited

- Recall multiprogramming: have multiple programs available to the machine, even if you only have one CPU core that can execute them.
  - For CPU resource: context switch quickly between processes
- Can we perform something analogous to a context switch for process memory?
  - Suppose disk transfer rate is 100 MB/s
  - “switching” a 1 MB process would take 10 ms (+ disk seek time)
  - CPU context switch: approx. 10 – 50  $\mu$ s
  - Moving that 1 MB would make context switch take 200 – 1000 times longer!

Conclusion: We can't swap entirety of process memory on a context switch. It needs to already be in memory.

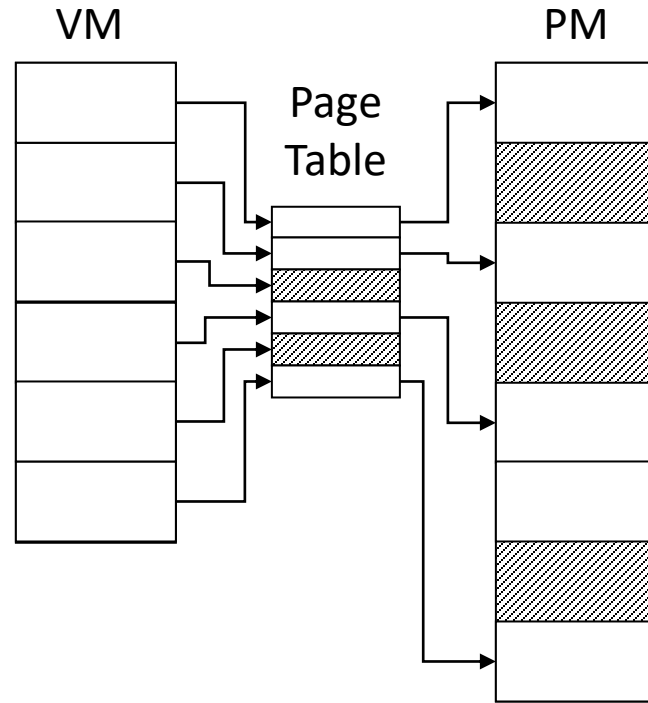
# Using Disk

- We still have a large amount of disk space though!
- If the total size of desired memory is larger than PAs, overflow to disk.
  - Disk: can store a lot, but relatively painful to access
  - Memory: much faster than disk, but can only store a subset
- This should sound familiar to a big CS 31 topic... Caching
- Recall locality: we tend to repeatedly access recently accessed items, or those that are nearby.

# “Swapping” Pages to Disk

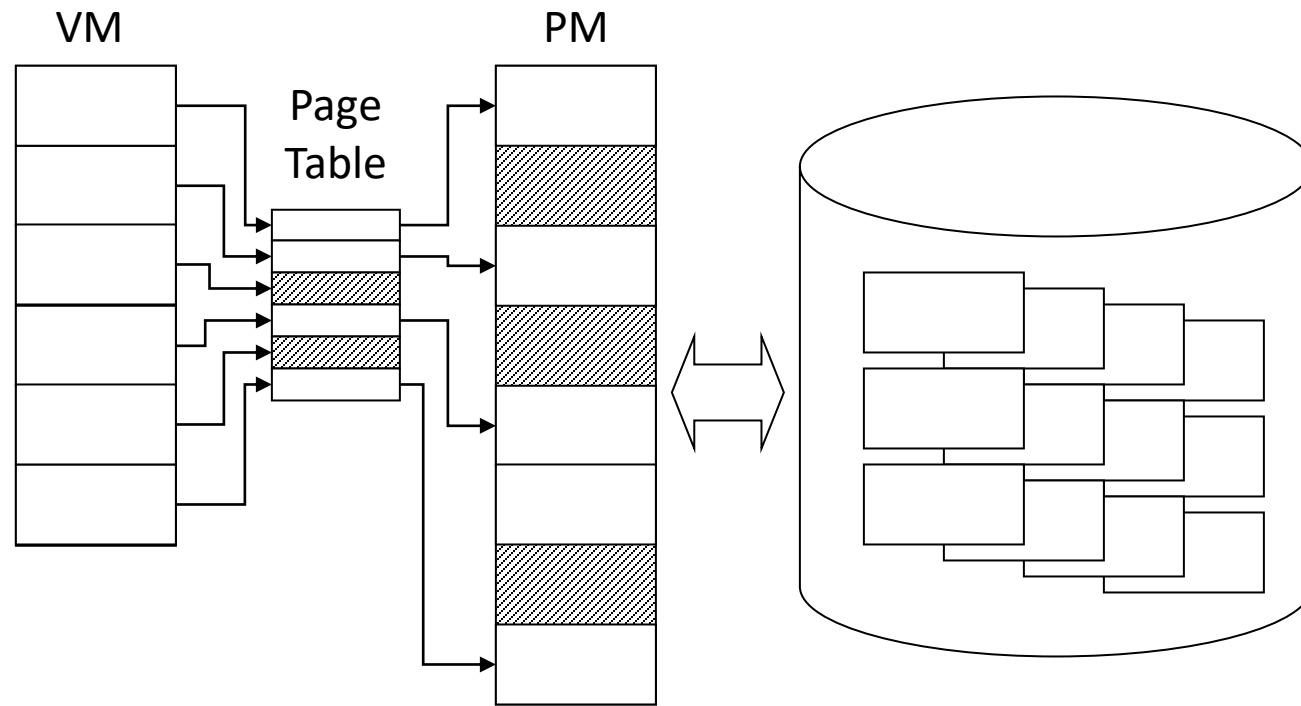
- Intuition: If a process isn't using a page, why keep it in physical memory? Instead, send it to disk and reclaim that space.
- Illusion: memory size is physical memory + disk (with non-uniform access times).
- Supporting this idea requires:
  - Identifying where a chunk of memory is (physical memory or disk).
  - Moving data between physical memory and disk (mechanism).
  - Algorithm for governing what gets moved to disk and what stays (policy).

# Virtual Memory based on Paging



- Before
  - All virtual pages were in physical memory.

# Virtual Memory based on Paging

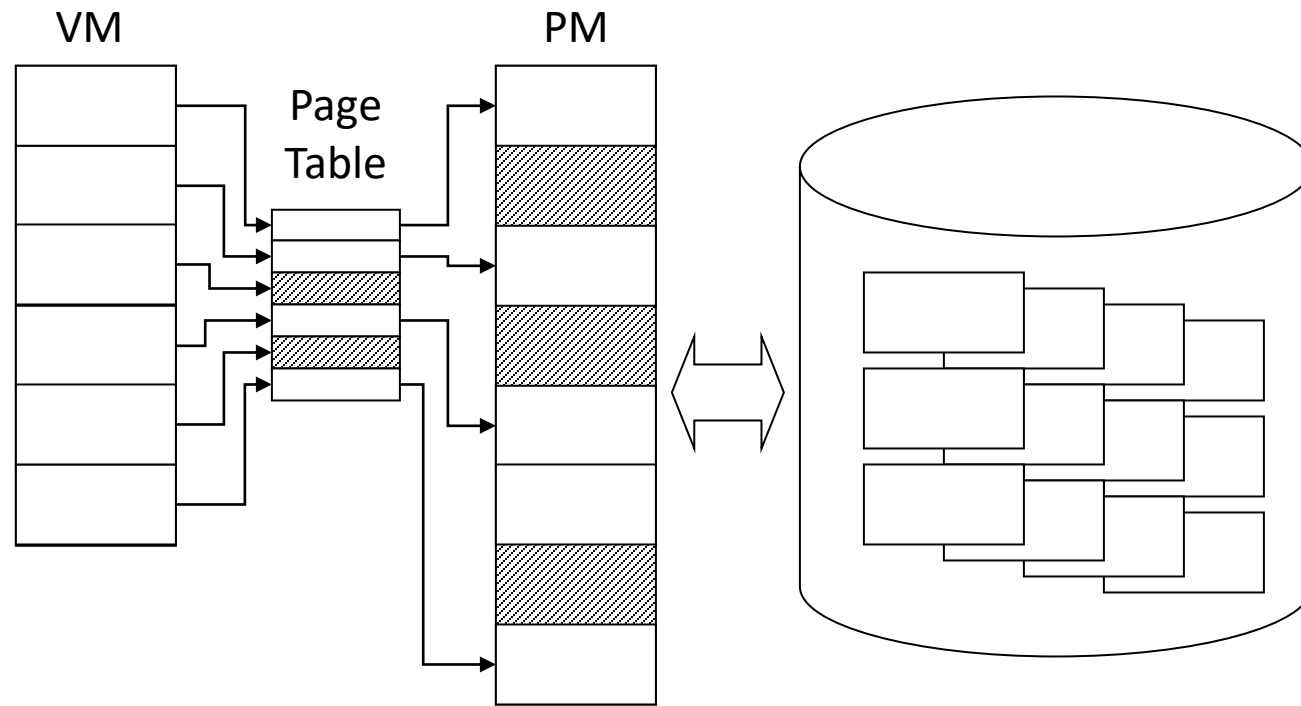


- Now

- Pages, if they exist, reside in physical memory or on disk (or both).
- Which pages are on disk? In memory?



# Virtual Memory based on Paging



For disk, we'll assume simple lookup structure:

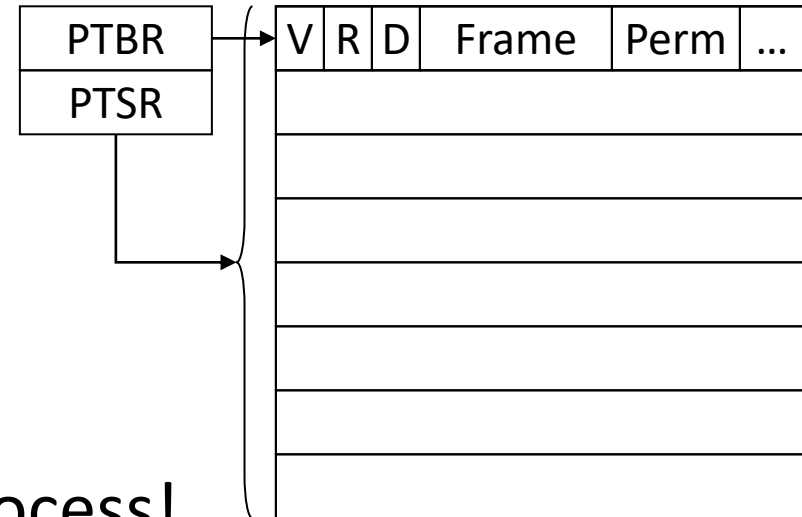
Key:  
Process, Page Number

Value:  
Location of page on disk  
(or error if not there)

- Now
  - Pages, if they exist, reside in physical memory or on disk (or both).
  - Which pages are on disk? In memory?

# Page Table: Revisited

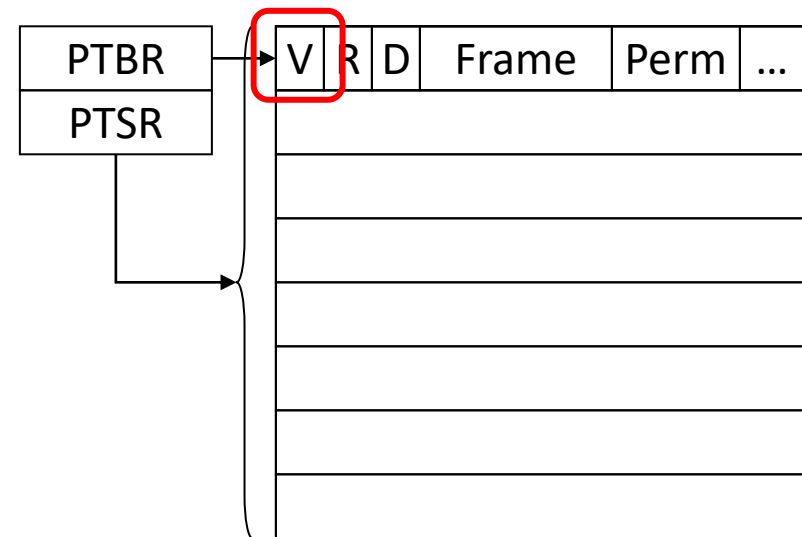
- One table per process
- Table parameters in memory
  - Page table base register
  - Page table size register
- By loading these registers, the **hardware** (MMU) knows where the page table is for the current process!



OS maintains the table, but hardware can access it to help improve performance!

# Page Table: Revisited

- One table per process
- Table parameters in memory
  - Page table base register
  - Page table size register
- Table entry elements
  - V: valid bit
  - R: referenced bit
  - D: dirty bit
  - Frame: location in phy mem
  - Perm: access permissions

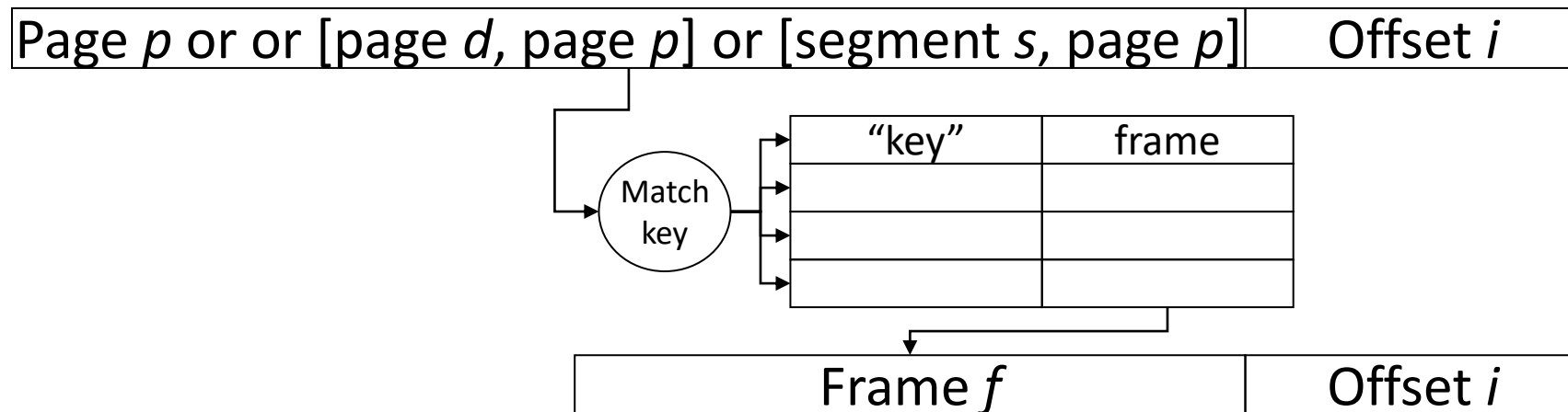


Valid bit, checkable by hardware, says whether or not the page is in physical memory:  
1: in memory, use frame field to find where  
0: not in memory

# Memory Access Cases

- Case 1: TLB Hit (MMU Hardware resolves address, lookup in TLB only)

1. User accesses a virtual address.
2. The upper bits / key find a match in TLB hardware cache.
3. The resolution is complete, use TLB value.



# Memory Access Cases

- Case 2: TLB miss, page table contains valid entry  
(MMU Hardware resolves address, lookup in TLB and page table)
  1. User accesses a virtual address.
  2. The upper bits / key *don't* find a match in TLB hardware cache.
  3. The MMU hardware knows where the page table is!
  4. MMU indexes into page table, finds frame number.
  5. MMU loads the TLB and completes address resolution.

OS doesn't have to do anything. Its work was done in setting up the table in advance.

That is, NO context switch!

What should happen if the TLB misses and the page table entry is invalid? **“Page Fault”**

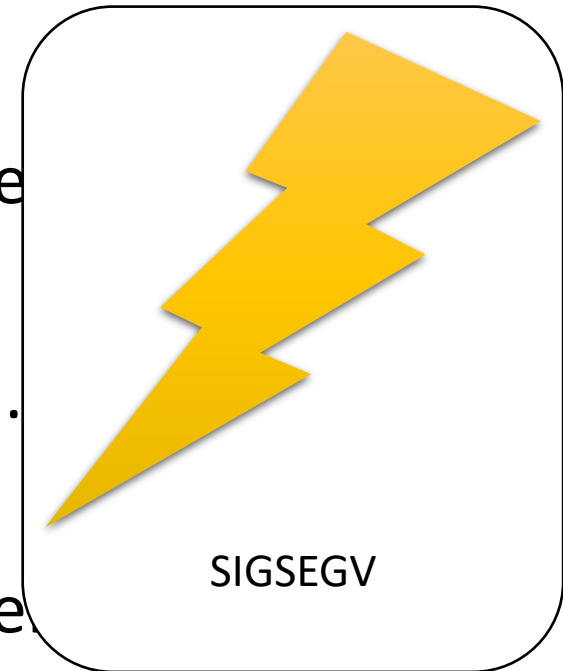
- A. The MMU hardware resolves the physical address by going to disk.
- B. The MMU hardware terminates the process.
- C. MMU exception: OS resolves physical address by going to disk.
- D. MMU exception: OS evicts a page and then resolves address by going to disk.
- E. MMU exception: OS terminates the process.

# Memory Access Cases

- Case 3: TLB miss, page table contains invalid entry, disk has the page (OS resolves address, lookup in TLB, page table, and disk)
  1. User accesses a virtual address.
  2. The upper bits / key *don't* find a match in TLB hardware cache.
  3. The MMU hardware knows where the page table is!
  4. MMU indexes into table, but page table entry is invalid...
  5. MMU raises exception – OS gets control of the CPU
  6. OS finds faulting page on disk, brings it into memory, and restarts process **from the instruction that faulted.**

# Memory Access Cases

- Case 4: TLB miss, page table contains invalid entry, disk doesn't have page (OS can't resolve address, lookup in TLB, page table, and disk)
  1. User accesses a virtual address.
  2. The upper bits / key *don't* find a match in TLB hardware
  3. The MMU hardware knows where the page table is!
  4. MMU indexes into table, but page table entry is invalid.
  5. MMU raises exception – OS gets control of the CPU
  6. OS looks for page on disk, but it isn't there. It was never there.
  7. OS terminates the offending process.





# Page Faults are Expensive

- Disk: 5-6 orders magnitude slower than RAM
  - Very expensive; but if very rare, tolerable
- Example
  - RAM access time: 100 nsec
  - Disk access time: 10 msec
  - $p$  = page fault probability
  - Effective access time:  $100 + p \times 10,000,000$  nsec
  - If  $p = 0.1\%$ , effective access time = 10,100 nsec !

Analogy: Most of the time, to get where you need to be, you walk to Kohlberg.

Occasionally, you have to walk to Seattle.

We need to be smart about what we send to disk. Minimize the slowdown.

# Policy Decisions for Virtual Memory

- Placement: Where should we put items in physical memory.
  - Irrelevant for page-based systems. Any frame is equally good.
- Replacement: Which page should we evict from memory to disk?
  - Which page do we pick?
  - Local vs global: Which process should the page come from?
- Cleaning: for modified (dirty) pages, when to write them to disk?

# Page Replacement

- For now, assume one process and that it has a fixed number of frames.
- Problem specification:
  - A page fault has just occurred.
  - All of the process's frames are full.
  - To complete the faulting instruction, one of the existing pages must be evicted to free up a frame.
- **Eviction:** remove the page from a frame (put on disk if it isn't already).
- **Victim:** the page that was chosen for eviction.

# Page Replacement Goals

1. Achieve good locality. Minimize page faults.
2. Easy to implement / low overhead to manage.

# Candidate Algorithms

- László Bélády – Hungarian computer scientist who studied this problem for IBM.
- Bélády's Optimal Algorithm (a.k.a. Clairvoyant algorithm):
  - Look ahead into the future and evict the page that won't be used for the longest time.
- Why is this worth considering when we clearly can't build it?
  - Gives us a benchmark. Can't do any better than this.

# Bélády's Optimal Algorithm

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

$F_0$												
$F_1$												
$F_2$												

\* Indicates page fault.

# Bélády's Optimal Algorithm

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F <sub>0</sub>	<u>1</u> *											
F <sub>1</sub>												
F <sub>2</sub>												

\* Indicates page fault.

# Bélády's Optimal Algorithm

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F <sub>0</sub>	1*	1										
F <sub>1</sub>		<u>2</u> *										
F <sub>2</sub>												

\* Indicates page fault.



# Bélády's Optimal Algorithm

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F <sub>0</sub>	1*	1	1									
F <sub>1</sub>		2*	2									
F <sub>2</sub>			<u>3</u> *									

\* Indicates page fault.

# Bélády's Optimal Algorithm

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F <sub>0</sub>	1*	1	1	?								
F <sub>1</sub>		2*	2	?								
F <sub>2</sub>			<u>3</u> *	?								

\* Indicates page fault.

Which page should we evict to make room for page 4? Why?


A: Page 1

B: Page 2

C: Page 3

# Bélády's Optimal Algorithm

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



F <sub>0</sub>	1*	1	1	1								
F <sub>1</sub>		2*	2	2								
F <sub>2</sub>			3*	<u>4</u> *								

\* Indicates page fault.

# Bélády's Optimal Algorithm

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F <sub>0</sub>	1*	1	1	1	<u>1</u>							
F <sub>1</sub>		2*	2	2	2							
F <sub>2</sub>			3*	4*	4							

\* Indicates page fault.

# Bélády's Optimal Algorithm

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F <sub>0</sub>	1*	1	1	1	1	1						
F <sub>1</sub>		2*	2	2	2	<u>2</u>						
F <sub>2</sub>			3*	4*	4	4						

\* Indicates page fault.

# Bélády's Optimal Algorithm

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F <sub>0</sub>	1*	1	1	1	1	1	1					
F <sub>1</sub>		2*	2	2	2	2	2					
F <sub>2</sub>			3*	4*	4	4	<u>5*</u>					

\* Indicates page fault.

# Bélády's Optimal Algorithm

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F <sub>0</sub>	1*	1	1	1	1	1	1	<u>1</u>				
F <sub>1</sub>		2*	2	2	2	2	2	2				
F <sub>2</sub>			3*	4*	4	4	5*	5				

\* Indicates page fault.

# Bélády's Optimal Algorithm

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F <sub>0</sub>	1*	1	1	1	1	1	1	1	1			
F <sub>1</sub>		2*	2	2	2	2	2	2	<u>2</u>			
F <sub>2</sub>			3*	4*	4	4	5*	5	5			

\* Indicates page fault.



# Bélády's Optimal Algorithm

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Could have chosen this one too.

F <sub>0</sub>	1*	1	1	1	1	1	1	1	1	<u>3</u> *		
F <sub>1</sub>		2*	2	2	2	2	2	2	<u>2</u>	2		
F <sub>2</sub>			3*	4*	4	4	5*	5	5	5		

\* Indicates page fault.

# Bélády's Optimal Algorithm

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F <sub>0</sub>	1*	1	1	1	1	1	1	1	1	3*	3	
F <sub>1</sub>		2*	2	2	2	2	2	2	<u>2</u>	2	<u>4</u> *	
F <sub>2</sub>			3*	4*	4	4	5*	5	5	5	5	

\* Indicates page fault.

# Bélády's Optimal Algorithm

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F <sub>0</sub>	1*	1	1	1	1	1	1	1	1	3*	3	3
F <sub>1</sub>		2*	2	2	2	2	2	2	<u>2</u>	2	4*	4
F <sub>2</sub>			3*	4*	4	4	5*	5	5	5	5	<u>5</u>

\* Indicates page fault.

Total: 7 Page faults.

# Candidate Algorithms - Reality

- Can't know the future of page accesses...
- Straightforward algorithm: FIFO
  - Always replace the oldest page.

# FIFO Replacement

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

First three pages start the same way: fill in free frames.

F <sub>0</sub>	<u>1</u> *	1	1									
F <sub>1</sub>		<u>2</u> *	2									
F <sub>2</sub>			<u>3</u> *									

\* Indicates page fault.

# FIFO Replacement

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F <sub>0</sub>	1*	1	1	<u>4</u> *								
F <sub>1</sub>		2*	2	2								
F <sub>2</sub>			3*	3								

\* Indicates page fault.

# FIFO Replacement

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F <sub>0</sub>	1*	1	1	4*	4							
F <sub>1</sub>		2*	2	2	<u>1</u> *							
F <sub>2</sub>			3*	3	3							

\* Indicates page fault.

# FIFO Replacement

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F <sub>0</sub>	1*	1	1	4*	4	4						
F <sub>1</sub>		2*	2	2	1*	1						
F <sub>2</sub>			3*	3	3	<u>2</u> *						

\* Indicates page fault.



# FIFO Replacement

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F <sub>0</sub>	1*	1	1	4*	4	4	<u>5</u> *					
F <sub>1</sub>		2*	2	2	1*	1	1					
F <sub>2</sub>			3*	3	3	2*	2					

\* Indicates page fault.

# FIFO Replacement

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F <sub>0</sub>	1*	1	1	4*	4	4	5*	5	5			
F <sub>1</sub>		2*	2	2	1*	1	1	<u>1</u>	1			
F <sub>2</sub>			3*	3	3	2*	2	2	<u>2</u>			

\* Indicates page fault.

# FIFO Replacement

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F <sub>0</sub>	1*	1	1	4*	4	4	5*	5	5	5		
F <sub>1</sub>		2*	2	2	1*	1	1	1	1	<u>3</u> *		
F <sub>2</sub>			3*	3	3	2*	2	2	2	2		

\* Indicates page fault.

# FIFO Replacement

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F <sub>0</sub>	1*	1	1	4*	4	4	5*	5	5	5	5	
F <sub>1</sub>		2*	2	2	1*	1	1	1	1	3*	3	
F <sub>2</sub>			3*	3	3	2*	2	2	2	2	<u>4</u> *	

\* Indicates page fault.

# FIFO Replacement

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F <sub>0</sub>	1*	1	1	4*	4	4	5*	5	5	5	5	<u>5</u>
F <sub>1</sub>		2*	2	2	1*	1	1	1	1	3*	3	3
F <sub>2</sub>			3*	3	3	2*	2	2	2	2	4*	4

\* Indicates page fault.

Total: 9 Page faults.

# How do we feel about FIFO for replacement...?

- A. It's fine.
- B. It's too expensive to build. (Why?)
- C. It exhibits poor locality (Why?)
- D. It doesn't manage memory (frames) well. (Why not?)
- E. Some other reason(s).



# Bélády's Anomaly: FIFO

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

$F_0$	<b>1*</b>	1	1	<b>4*</b>	4	4	<b>5*</b>	5	5	5	5	<b>5</b>
$F_1$		<b>2*</b>	2	2	<b>1*</b>	1	1	<b>1</b>	1	<b>3*</b>	3	3
$F_2$			<b>3*</b>	3	3	<b>2*</b>	2	2	<b>2</b>	2	<b>4*</b>	4

9 Faults

$F_0$	<b>1*</b>	1	1	1	<b>1</b>	1	<b>5*</b>	5	5	5	<b>4*</b>	4
$F_1$		<b>2*</b>	2	2	2	<b>2</b>	2	<b>1*</b>	1	1	1	<b>5*</b>
$F_2$			<b>3*</b>	3	3	3	3	3	<b>2*</b>	2	2	2
$F_3$				<b>4*</b>	4	4	4	4	4	<b>3*</b>	3	3

10 Faults



# Candidate Algorithms - Reality

- Can't know the future of page accesses...
- Straightforward algorithm: FIFO
  - Always replace the oldest page.
- Classic cache replacement algorithm: LRU
  - Replace the page that hasn't been used for the longest time.

# LRU Replacement

Note: New page sequence!

Pages Accessed: 2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2

First four pages: fill in free frames.

F <sub>0</sub>	<b>2*</b>	2	<b>2</b>	2								
F <sub>1</sub>		<b>3*</b>	3	3								
F <sub>2</sub>				<b>1*</b>								

\* Indicates page fault.

# LRU Replacement

Pages Accessed: 2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2

F <sub>0</sub>	<b>2*</b>	2	<b>2</b>	2	2							
F <sub>1</sub>		<b>3*</b>	3	3	<u>5*</u>							
F <sub>2</sub>				<b>1*</b>	1							

\* Indicates page fault.

# LRU Replacement

Pages Accessed: 2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2

F <sub>0</sub>	<b>2*</b>	2	<b>2</b>	2	2	<b>2</b>						
F <sub>1</sub>		<b>3*</b>	3	3	<b>5*</b>	5						
F <sub>2</sub>				<b>1*</b>	1	1						

\* Indicates page fault.

# LRU Replacement

Pages Accessed: 2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2

F <sub>0</sub>	<b>2*</b>	2	<b>2</b>	2	2	<b>2</b>	2					
F <sub>1</sub>		<b>3*</b>	3	3	<b>5*</b>	5	5					
F <sub>2</sub>				<b>1*</b>	1	1	<u><b>4*</b></u>					

\* Indicates page fault.

# LRU Replacement

Pages Accessed: 2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2

F <sub>0</sub>	<b>2*</b>	2	<b>2</b>	2	2	<b>2</b>	2	2				
F <sub>1</sub>		<b>3*</b>	3	3	<b>5*</b>	5	5	<u>5</u>				
F <sub>2</sub>				<b>1*</b>	1	1	<b>4*</b>	4				

\* Indicates page fault.

# LRU Replacement

Pages Accessed: 2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2

F <sub>0</sub>	<b>2*</b>	2	<b>2</b>	2	2	<b>2</b>	2	2	<u>3*</u>			
F <sub>1</sub>		<b>3*</b>	3	3	<b>5*</b>	5	5	<b>5</b>	5			
F <sub>2</sub>				<b>1*</b>	1	1	<b>4*</b>	4	4			

\* Indicates page fault.

# LRU Replacement

Pages Accessed: 2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2

F <sub>0</sub>	<b>2*</b>	2	<b>2</b>	2	2	<b>2</b>	2	2	<b>3*</b>	3		
F <sub>1</sub>		<b>3*</b>	3	3	<b>5*</b>	5	5	<b>5</b>	5	5		
F <sub>2</sub>				<b>1*</b>	1	1	<b>4*</b>	4	4	<u><b>2*</b></u>		

\* Indicates page fault.



# LRU Replacement

Pages Accessed: 2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2

F <sub>0</sub>	<b>2*</b>	2	<b>2</b>	2	2	<b>2</b>	2	2	<b>3*</b>	3	3	3
F <sub>1</sub>		<b>3*</b>	3	3	<b>5*</b>	5	5	<b>5</b>	5	5	<b>5</b>	5
F <sub>2</sub>				<b>1*</b>	1	1	<b>4*</b>	4	4	<b>2*</b>	2	<b>2</b>

\* Indicates page fault.

Total: 7 Page faults.

# LRU Replacement

For this sequence, FIFO faults 9 times, optimal faults 6 times. (Try it!)

Pages Accessed: 2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2

F <sub>0</sub>	<b>2*</b>	2	<b>2</b>	2	2	<b>2</b>	2	2	<b>3*</b>	3	3	3
F <sub>1</sub>		<b>3*</b>	3	3	<b>5*</b>	5	5	<b>5</b>	5	5	<b>5</b>	5
F <sub>2</sub>				<b>1*</b>	1	1	<b>4*</b>	4	4	<b>2*</b>	2	<b>2</b>

\* Indicates page fault.

Total: 7 Page faults.

# How do we feel about LRU for replacement...?

- A. It's fine.
- B. It's too expensive to build. (Why?)
- C. It exhibits poor locality (Why?)
- D. It doesn't manage memory (frames) well. (Why not?)
- E. Some other reason(s).

# Comparing Caching and Page Replacement

- Big similarities surrounding memory hierarchy:
  - Small, fast memory (caching: cache / VM: main memory)
  - Large, slow memory (caching: main memory / VM: disk)
- Key difference:
  - Caches are usually set associative: when you find a cache line, there are only {2, 4, 8} places the data can go in that line.
  - The limit to associativity bounds the amount of LRU state you need to keep!
  - With paging, any frame might be used, and there are many more frames...

Even still, it's often approximated.



# Implementing LRU for Page Replacement

- Want to take advantage of MMU hardware for performance.  
(Avoid switching to OS execution on *every memory access*.)
- For each memory access, MMU must update LRU information.
- Option 1: Timestamp the page.
  - Problem: lots of time lookups, lots of bits to store time in each page table row
- Option 2: Rearrange queue/list containing order of page accesses.
  - Problem: now we have hardware chasing pointers?

# An analogy: Replacement for your closet.

## Weed Out The Clothes You Don't Wear With A Simple Hanger Trick

ADAM PASH MARCH 27, 2010 3:00 AM



Got a closet full of clothes but the pack rat in you can't seem to part with any of them? A user on popular social news site Reddit offers a simple tip for weeding out those clothes you don't need.

Reddit's got a great "What are your best lifehacks?" thread going on right now, and the most popular item on the list is this gem from user elblanco:

Putting my clothes in my closet with the hangers reversed once a year. As I pull clothes out, I reverse the hanger. Every year I give away any clothes that I never took out.

# An analogy: Replacement for your closet.

Got a closet full of clothes but the pack rat in you can't seem to part with any of them? A user on popular social news site Reddit offers a simple tip for weeding out those clothes you don't need.

...

"Putting my clothes in the closet with the hangars reversed once a year. As I pull clothes out, I reverse the hanger. Every year I give away any clothes that I never took out."

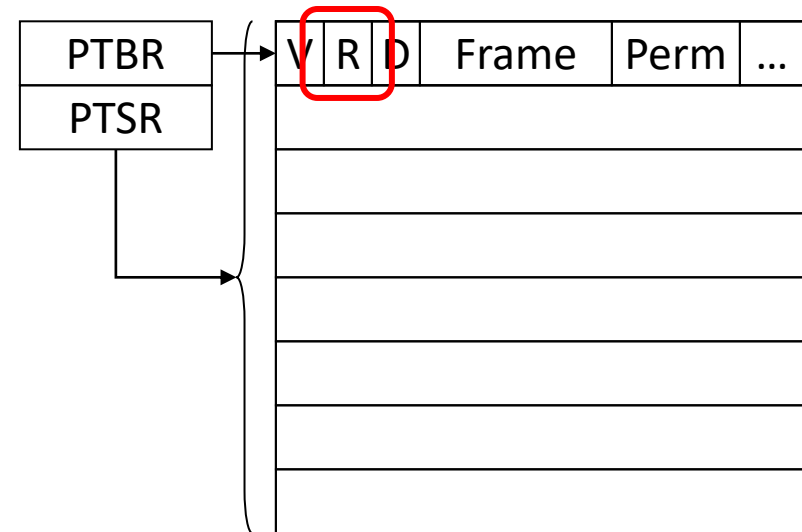
social news site Reddit offers a simple tip for weeding out those clothes you don't need.

Reddit's got a great "What are your best lifehacks?" thread going on right now, and the most popular item on the list is this gem from user elblanco:

Putting my clothes in my closet with the hangers reversed once a year. As I pull clothes out, I reverse the hanger. Every year I give away any clothes that I never took out.

# Page Table: Revisited (again)

- One table per process
- Table parameters in memory
  - Page table base register
  - Page table size register
- Table entry elements
  - V: valid bit
  - R: referenced bit
  - D: dirty bit
  - Frame: location in phy mem
  - Perm: access permissions



We'll use ONE BIT per entry to approximate LRU.  
Easy to do in MMU hardware:  
When we access a page, MMU sets the bit to 1.

Intuition: has this page been used recently?

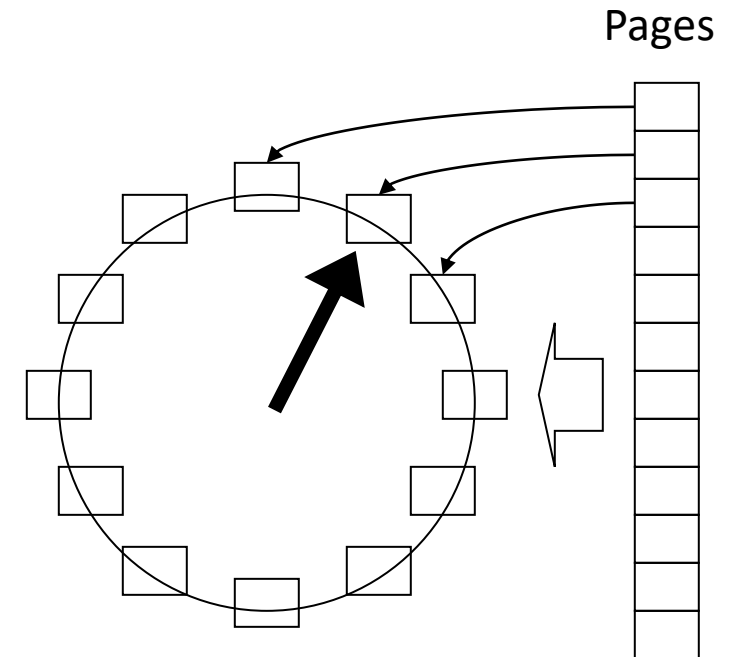


# Approximating LRU: Clock Algorithm

- Select page that is old and not recently used
  - Clock (a.k.a. “second chance”) is approximation of LRU
- Hardware support: reference bit
  - Associated with each page is a reference bit.
  - Reference bit set by MMU on page access.
- On page fault, look through the pages in numerical order (starting from where we left off during last fault).
  - If page is referenced recently (ref bit set): unset the reference bit.
  - If page is not referenced recently: evict it.

# Clock Algorithm Model

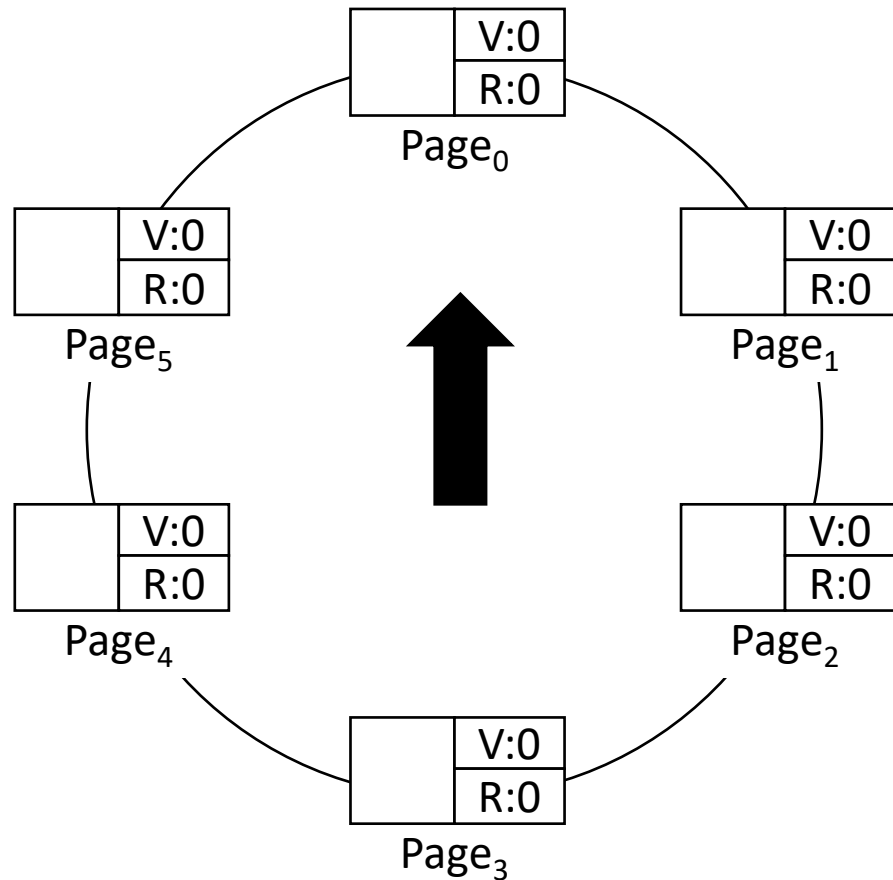
- Arrange all pages in circle (like a... clock)
- Clock “hand”: next page to consider
  - Skip over invalid entries, they have no frame.
- Page fault: scan forward, starting at hand
  - if reference bit 0, select page as victim
  - otherwise, set reference bit to 0
  - advance clock hand to next page
  - if victim found, break out of loop (else repeat)
- Hand position preserved across faults.



# Clock Example

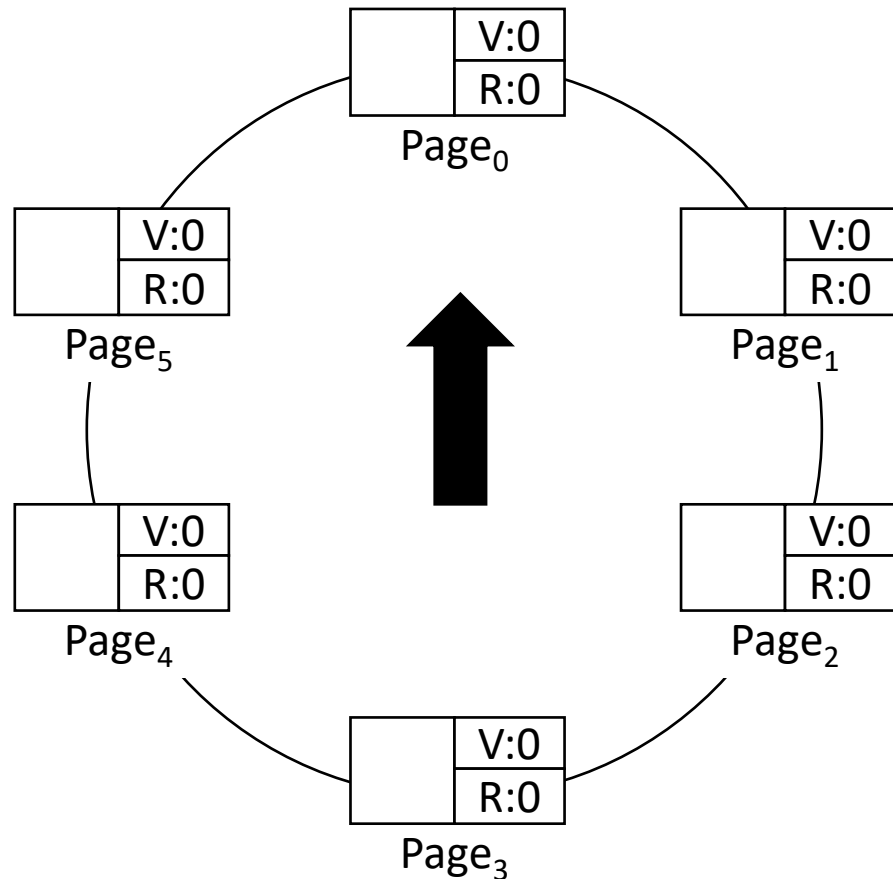
- Pages accessed: 2, 4, 5, 1, 4, 2, 4

Free frames: 0, 1, 2



# Clock Example

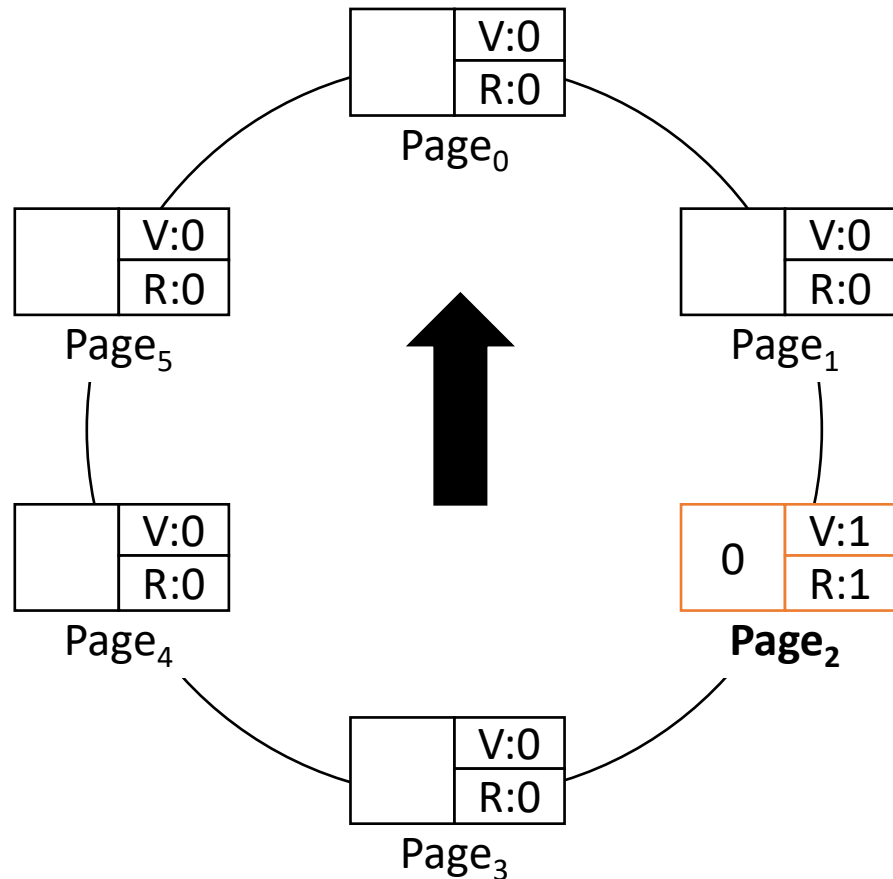
- Pages accessed: 2, 4, 5, 1, 4, 2, 4      Free frames: 0, 1, 2



- First, use up any available free frames.

# Clock Example

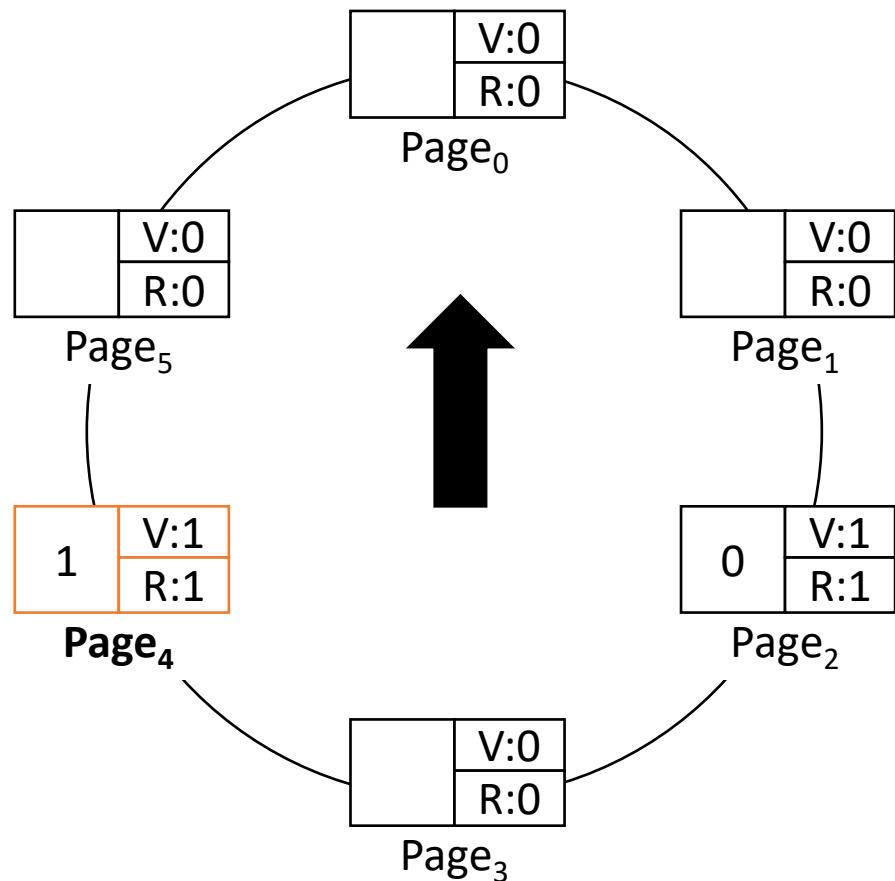
- Pages accessed: 2, 4, 5, 1, 4, 2, 4      Free frames: 0, 1, 2



- First, use up any available free frames.
- Set valid bit because page has a frame.
- Set reference bit because page was accessed.
- Access page 2: page fault (unavoidable)
  - We have a free frame, use it!

# Clock Example

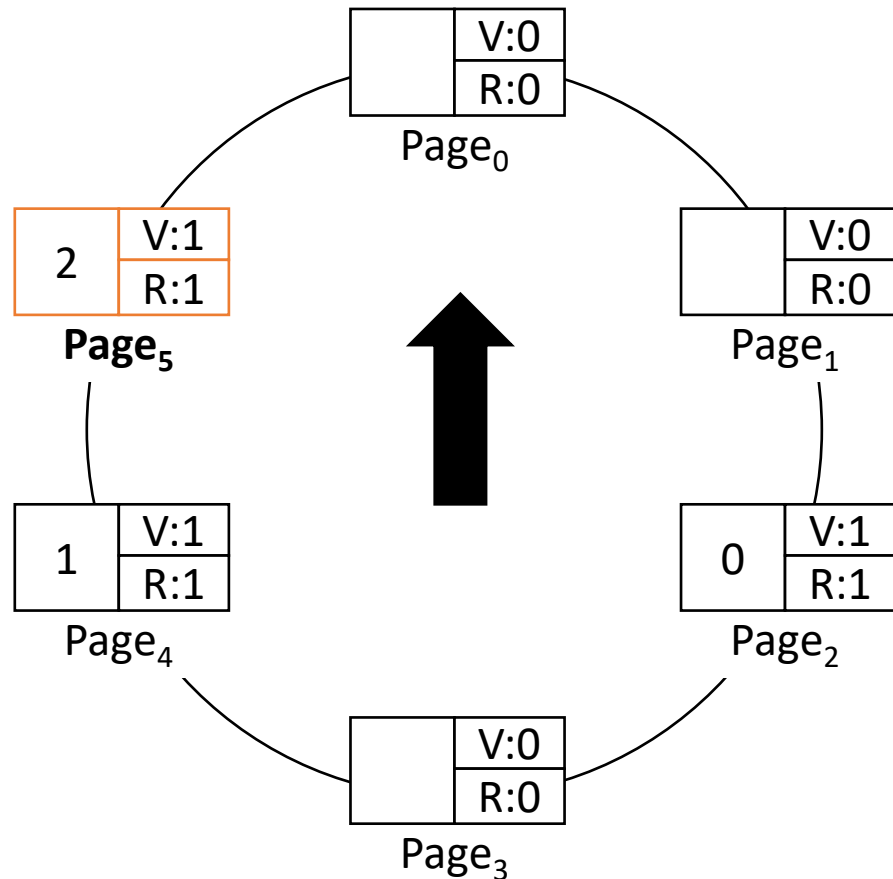
- Pages accessed: 2, 4, 5, 1, 4, 2, 4      Free frames: ~~0~~, ~~1~~, 2



- First, use up any available free frames.
- Set valid bit because page has a frame.
- Set reference bit because page was accessed.
- Access page 4: page fault (unavoidable)
  - We have a free frame, use it!

# Clock Example

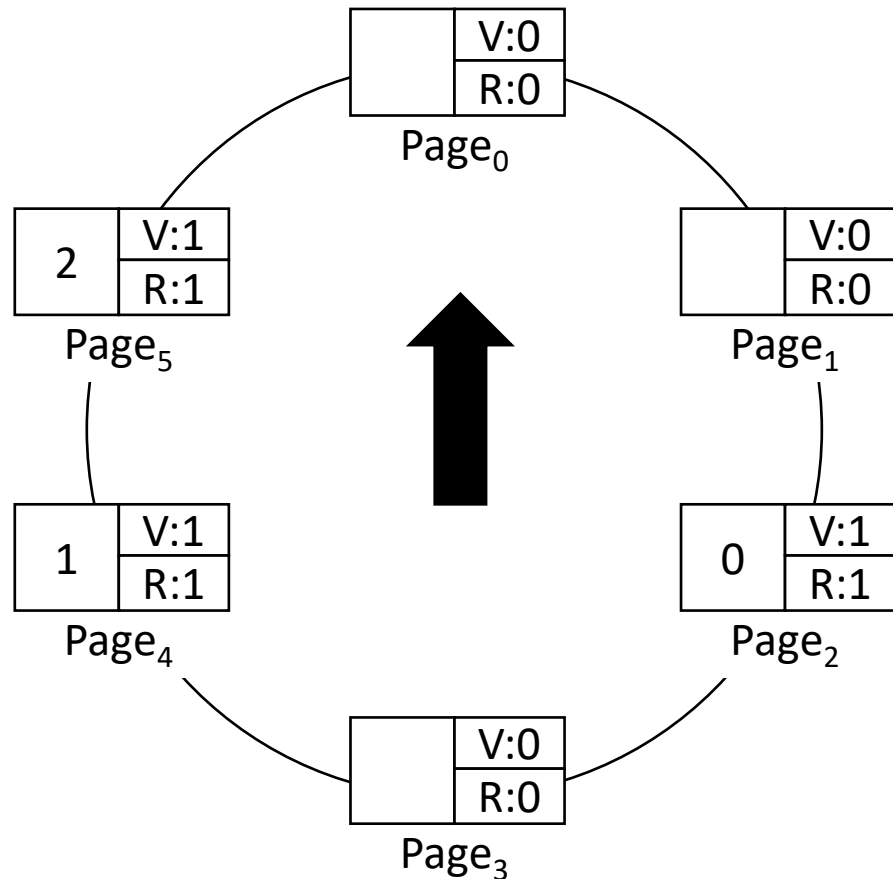
- Pages accessed: 2, 4, 5, 1, 4, 2, 4      Free frames: ~~0~~, ~~1~~, ~~2~~



- First, use up any available free frames.
- Set valid bit because page has a frame.
- Set reference bit because page was accessed.
- Access page 5: page fault (unavoidable)
  - We have a free frame, use it!

# Clock Example

- Pages accessed: 2, 4, 5, 1, 4, 2, 4      Free frames: ~~0~~, ~~1~~, ~~2~~



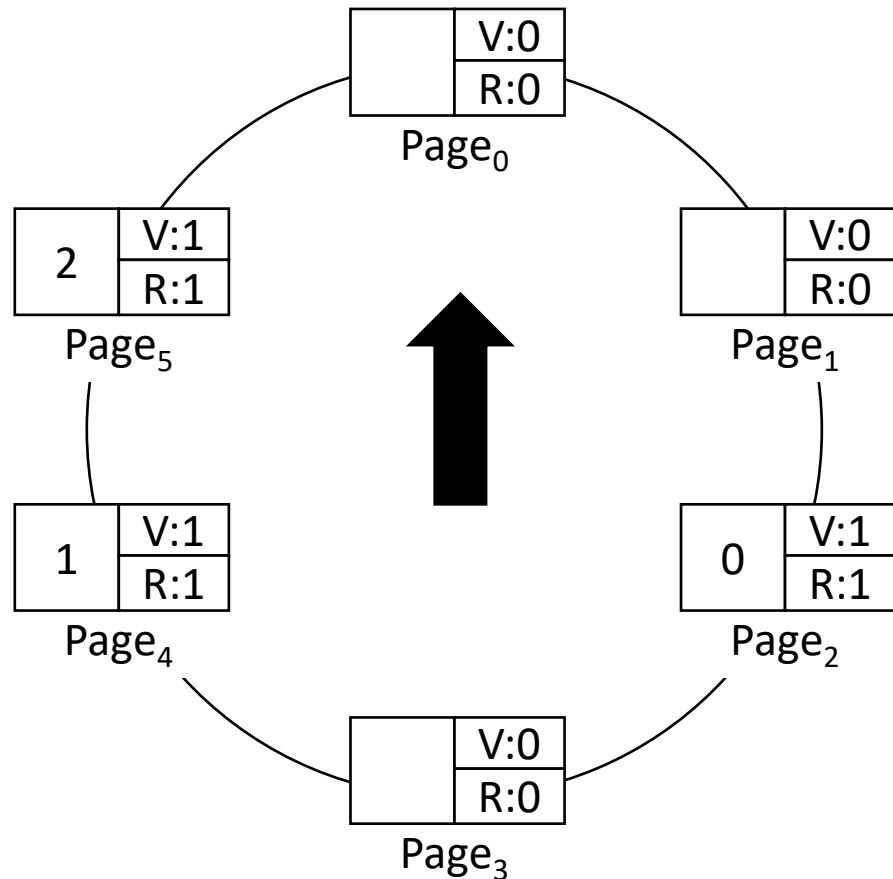
- Access page 1: page fault.
- We have no free frames, so one of the pages with a frame needs to be evicted.



# Which page should we evict to make room for page 1? Why?

• Pages accessed: 2, 4, 5, 1, 4, 2, 4

Free frames: ~~0~~, ~~1~~, ~~2~~



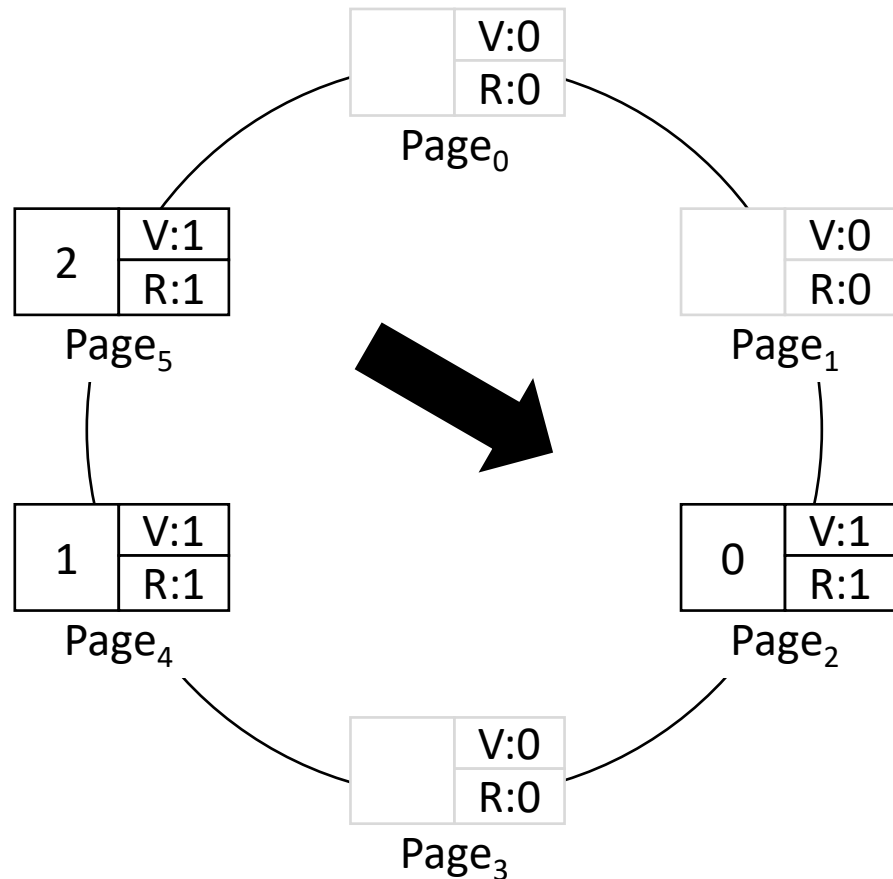
A. Page 2

B. Page 4

C. Page 5

# Clock Example

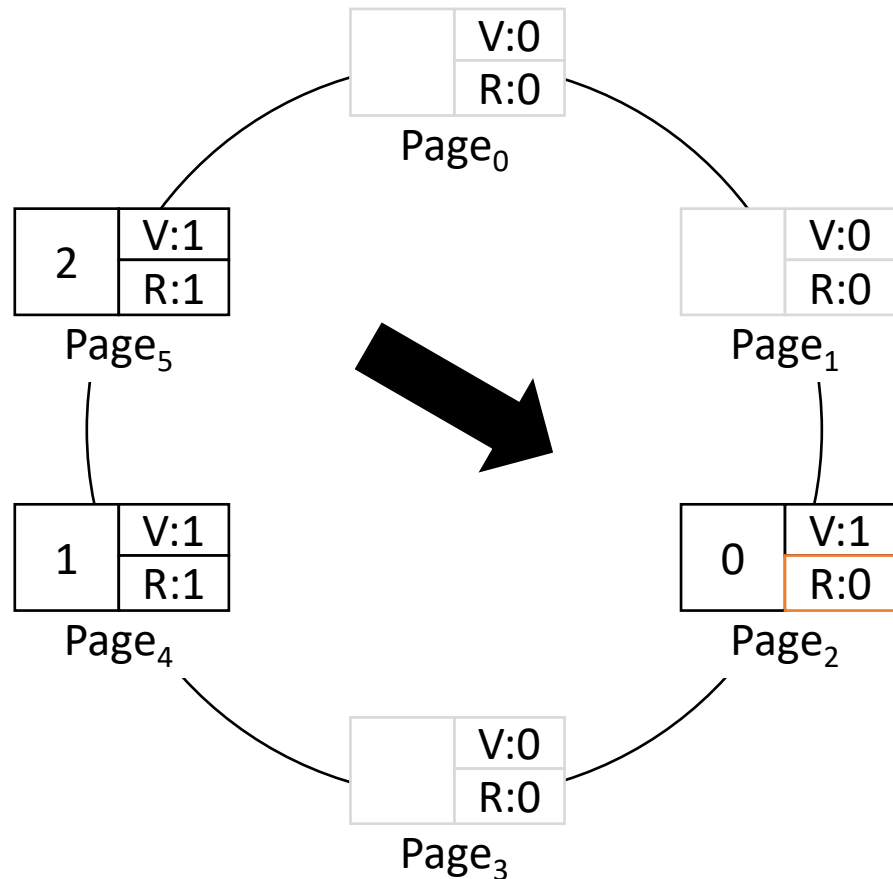
- Pages accessed: 2, 4, 5, 1, 4, 2, 4      Free frames: ~~0~~, ~~1~~, ~~2~~



- Ignore any invalid page entries, they don't have frames, so we can't evict them.

# Clock Example

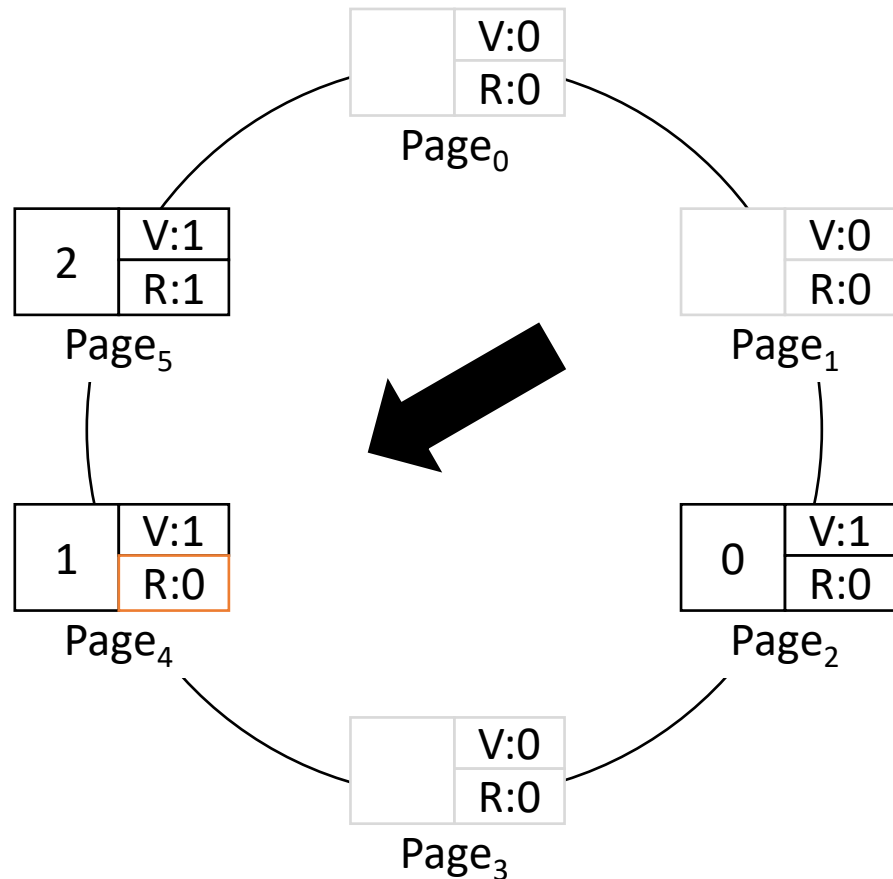
- Pages accessed: 2, 4, 5, 1, 4, 2, 4      Free frames: ~~0~~, ~~1~~, ~~2~~



- Hand points to a referenced page.
- Set ref bit to 0, advance hand, try again.

# Clock Example

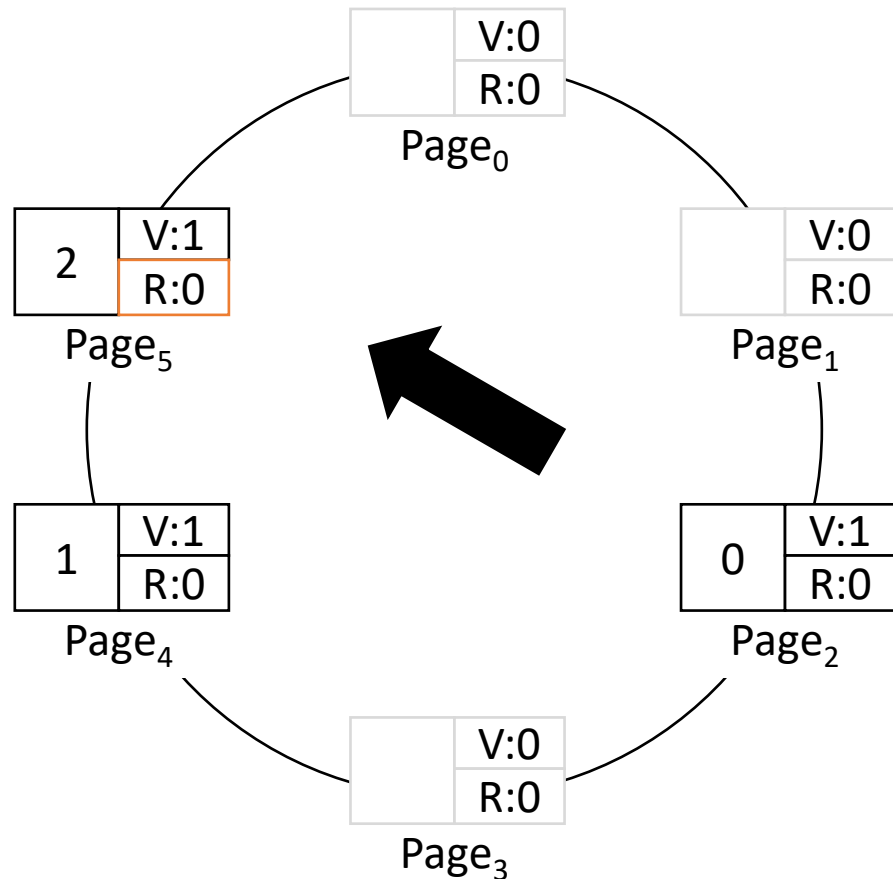
- Pages accessed: 2, 4, 5, 1, 4, 2, 4      Free frames: ~~0~~, ~~1~~, ~~2~~



- Hand points to a referenced page.
- Set ref bit to 0, advance hand, try again.

# Clock Example

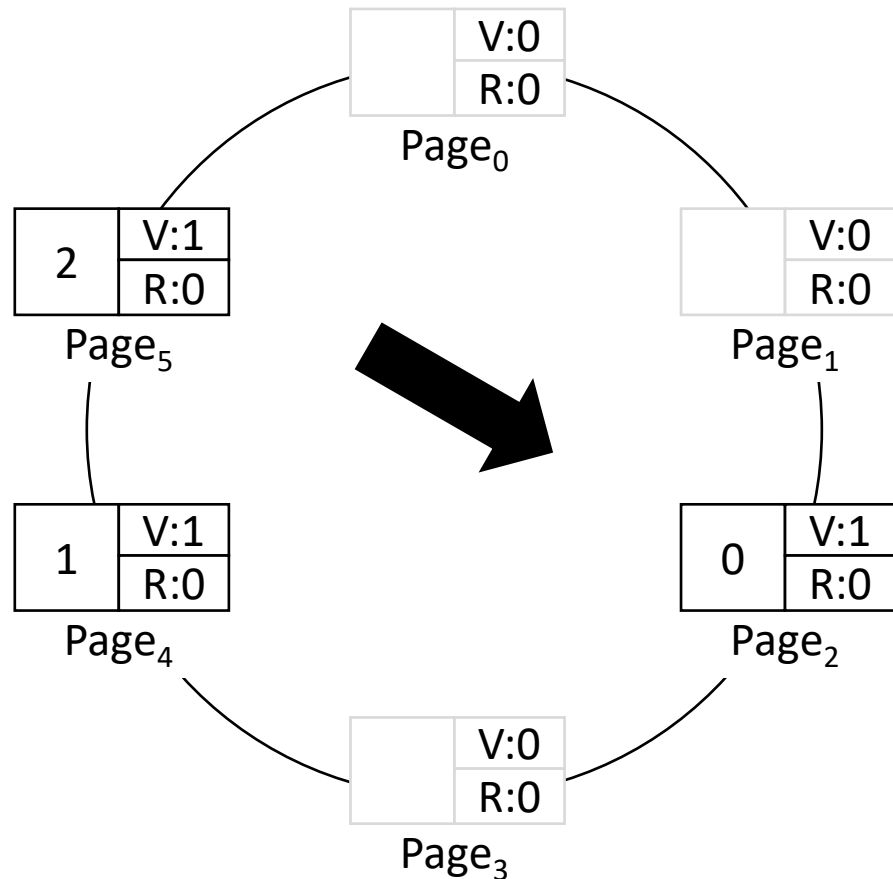
- Pages accessed: 2, 4, 5, 1, 4, 2, 4      Free frames: ~~0~~, ~~1~~, ~~2~~



- Hand points to a referenced page.
- Set ref bit to 0, advance hand, try again.

# Clock Example

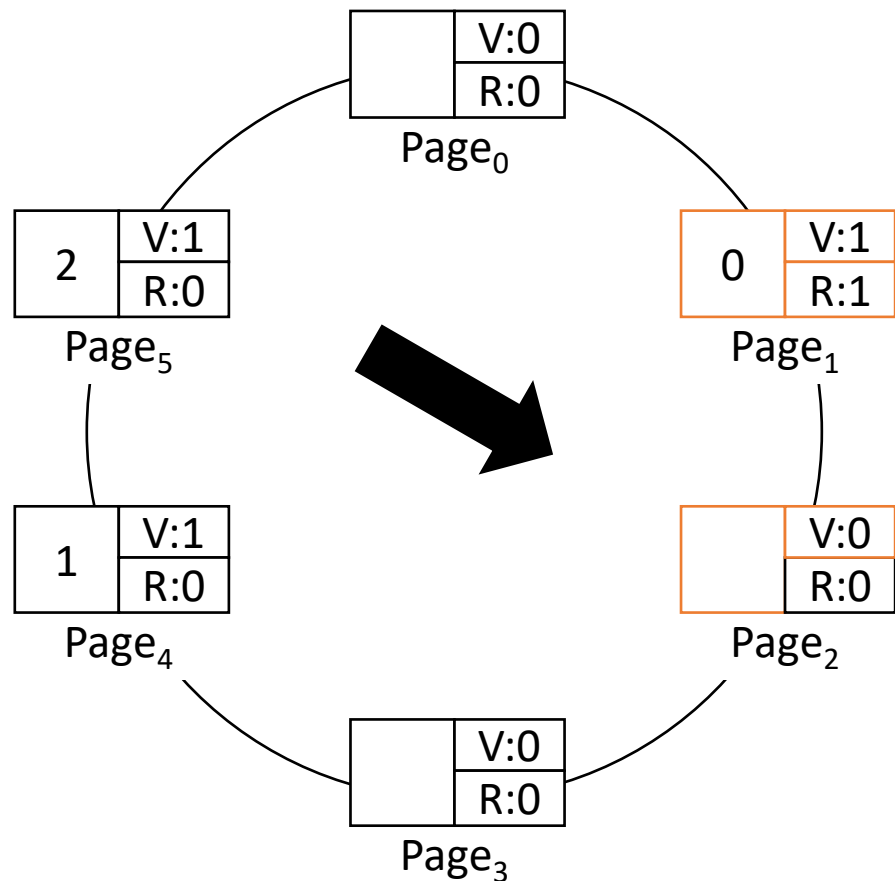
- Pages accessed: 2, 4, 5, 1, 4, 2, 4      Free frames: ~~0~~, ~~1~~, ~~2~~



- Hand points to a page with ref bit 0!
  - We've found our victim.

# Clock Example

- Pages accessed: 2, 4, 5, 1, 4, 2, 4      Free frames: ~~0~~, ~~1~~, ~~2~~

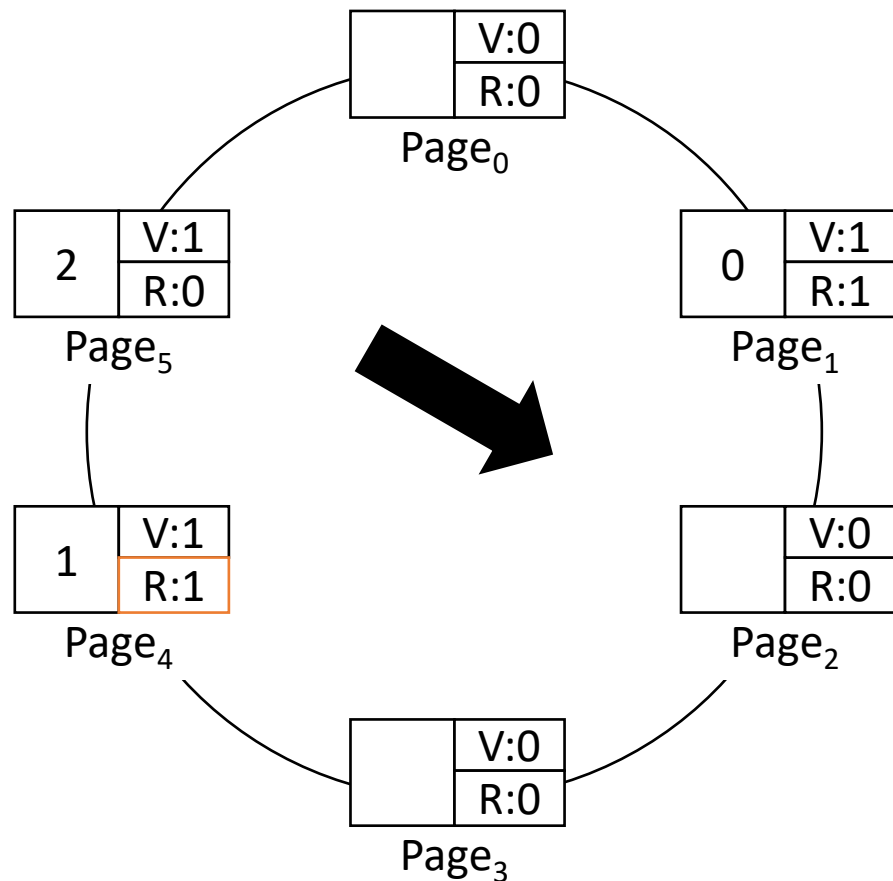


- Hand points to a page with ref bit 0!
  - We've found our victim.
- Evict page 2 (mark invalid), and assign its old frame (frame 0) to the faulting page (page 1).

Wait, isn't this just FIFO (or LRU)? So far, yes.

# Clock Example

- Pages accessed: 2, 4, 5, 1, 4, 2, 4      Free frames: ~~0~~, ~~1~~, ~~2~~

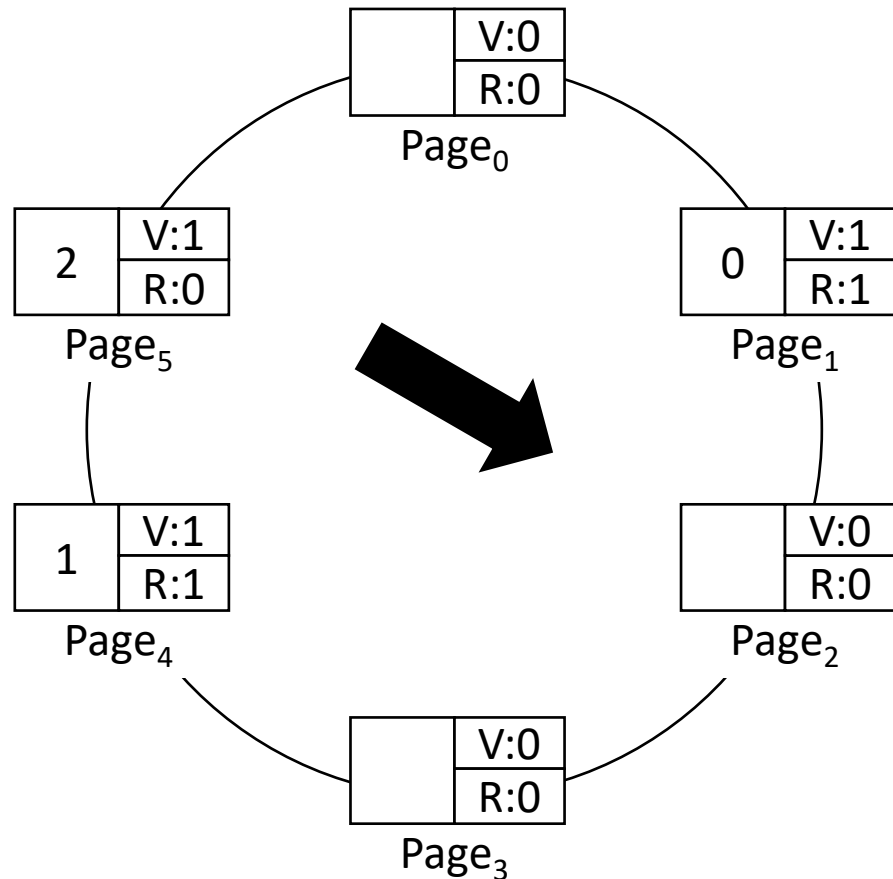


- Access page 4
  - Page 4 is already in memory: no fault
  - OS does nothing!
  - MMU hardware sets ref bit to 1



# Clock Example

- Pages accessed: 2, 4, 5, 1, 4, 2, 4      Free frames: ~~0, 1, 2~~

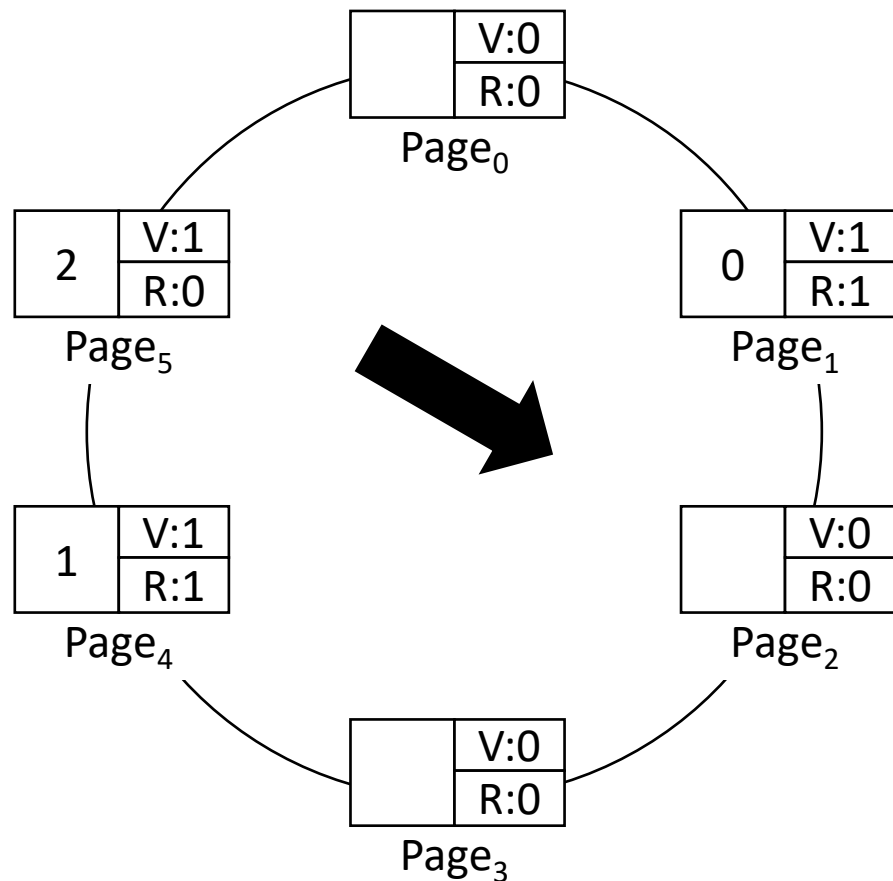


- Access page 2: page fault.
- We have no free frames, so one of the pages with a frame needs to be evicted.

# Which page should we evict to make room for page 1? Why?

• Pages accessed: 2, 4, 5, 1, 4, 2, 4

Free frames: ~~0~~, ~~1~~, ~~2~~



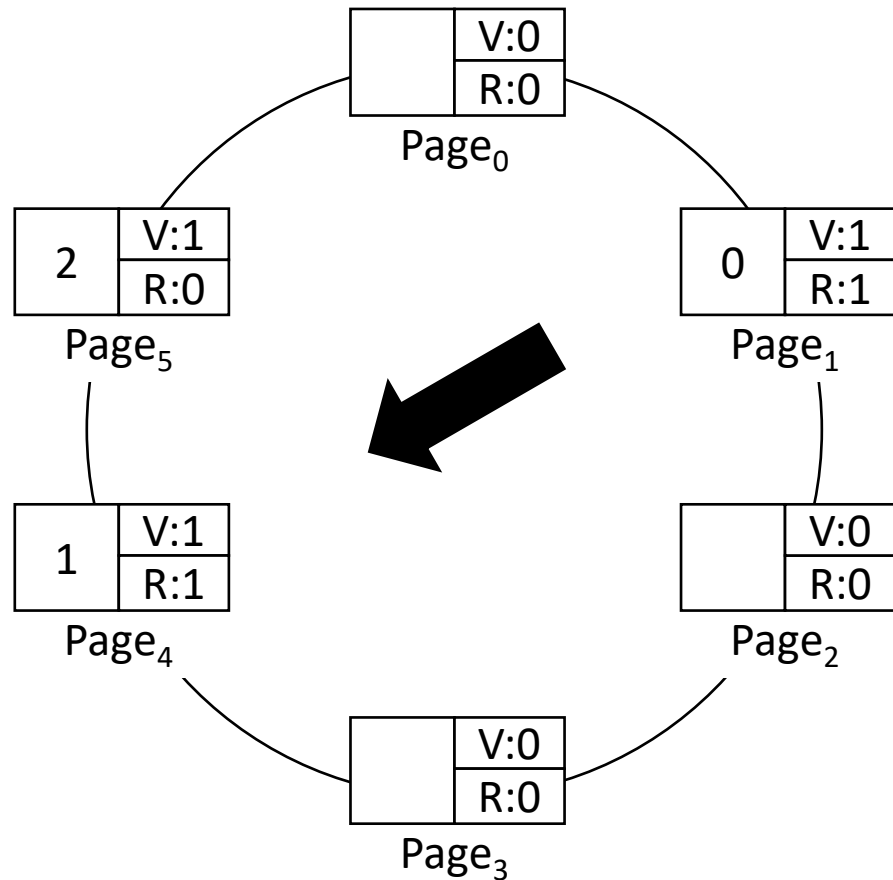
A. Page 1

B. Page 4

C. Page 5

# Clock Example

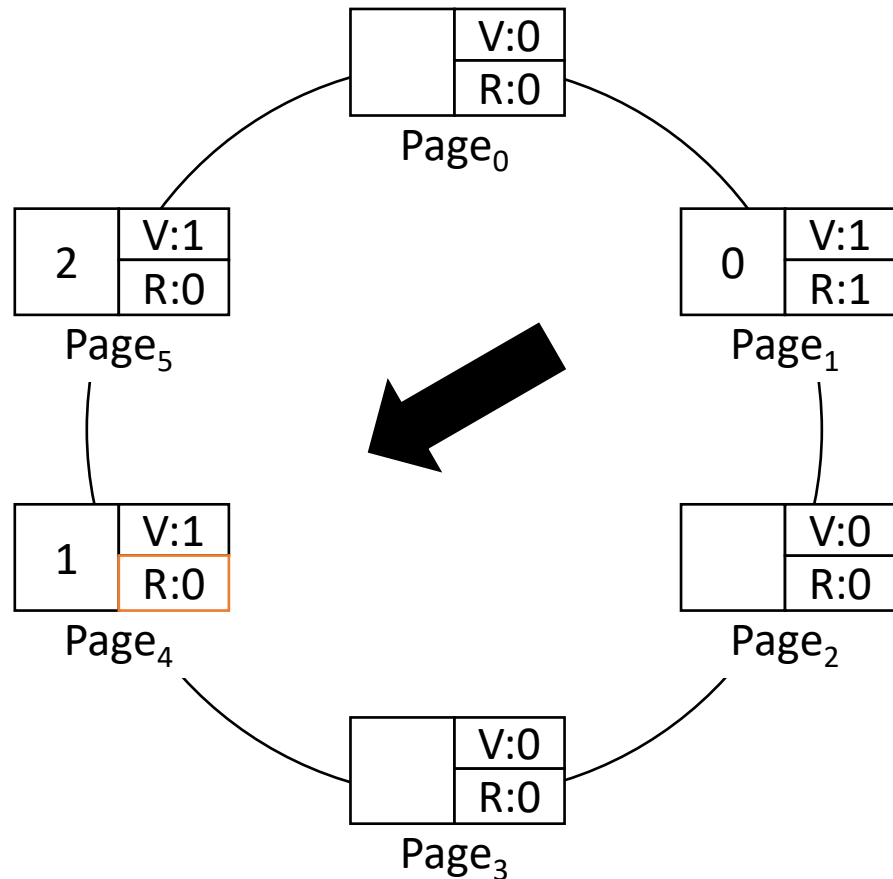
- Pages accessed: 2, 4, 5, 1, 4, 2, 4      Free frames: ~~0~~, ~~1~~, ~~2~~



- Skip the invalid pages.

# Clock Example

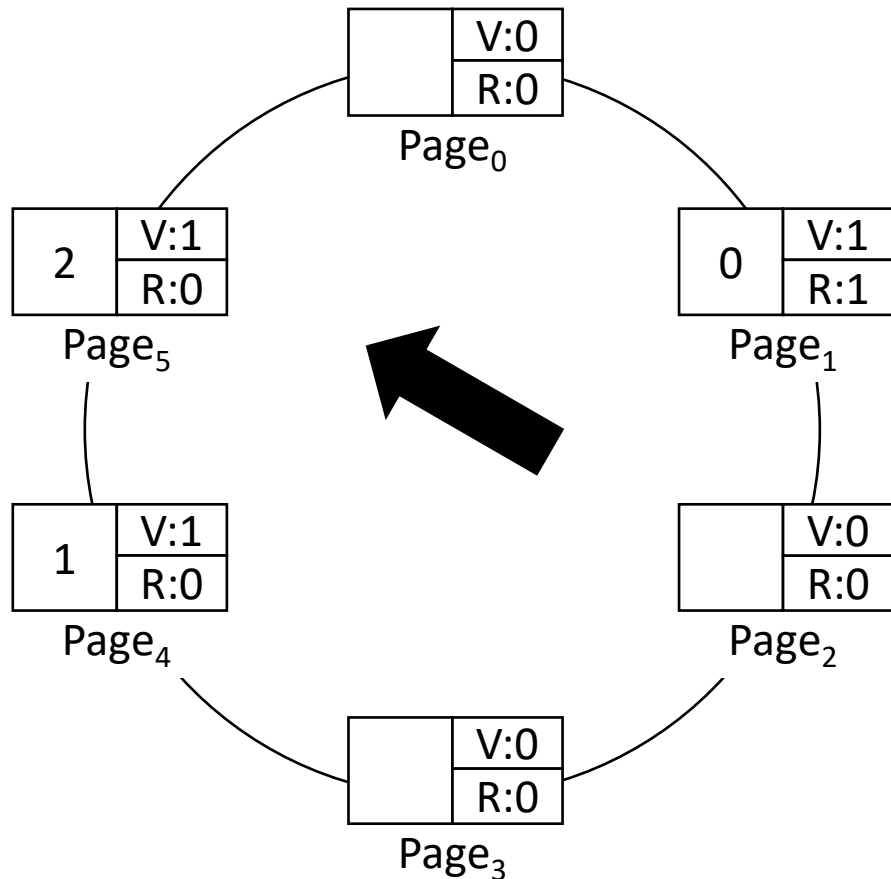
- Pages accessed: 2, 4, 5, 1, 4, 2, 4      Free frames: ~~0~~, ~~1~~, ~~2~~



- Hand points to a referenced page.
- Set ref bit to 0, advance hand, try again.

# Clock Example

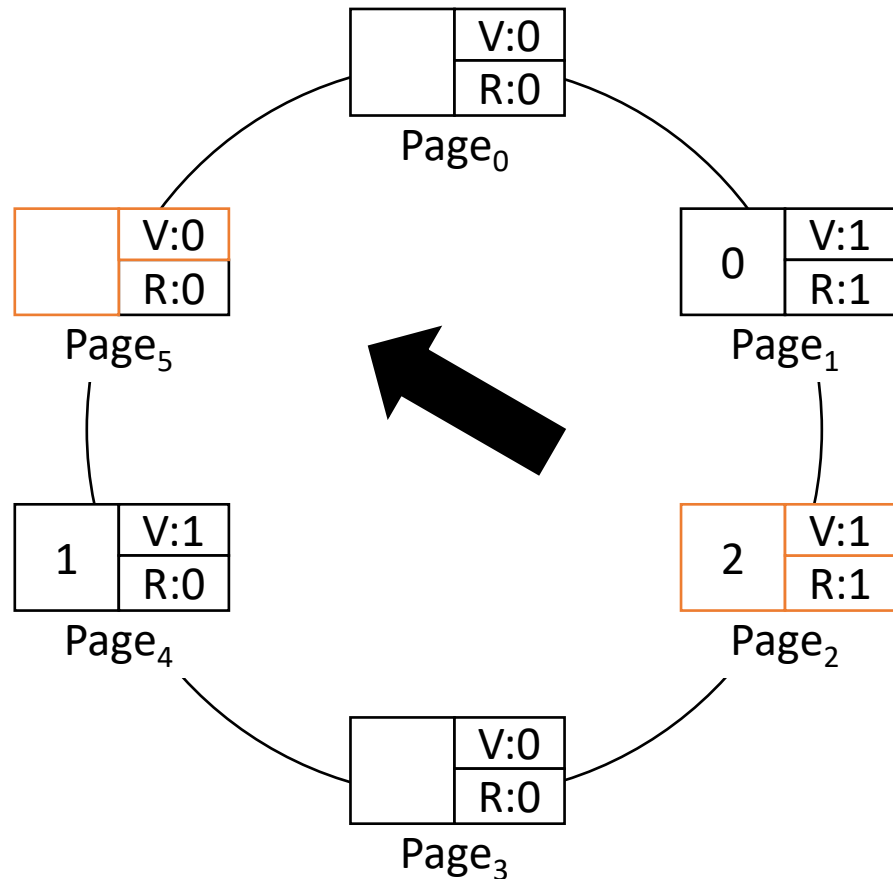
- Pages accessed: 2, 4, 5, 1, 4, 2, 4      Free frames: ~~0~~, ~~1~~, ~~2~~



- Hand points to a page with ref bit 0!
  - We've found our victim.

# Clock Example

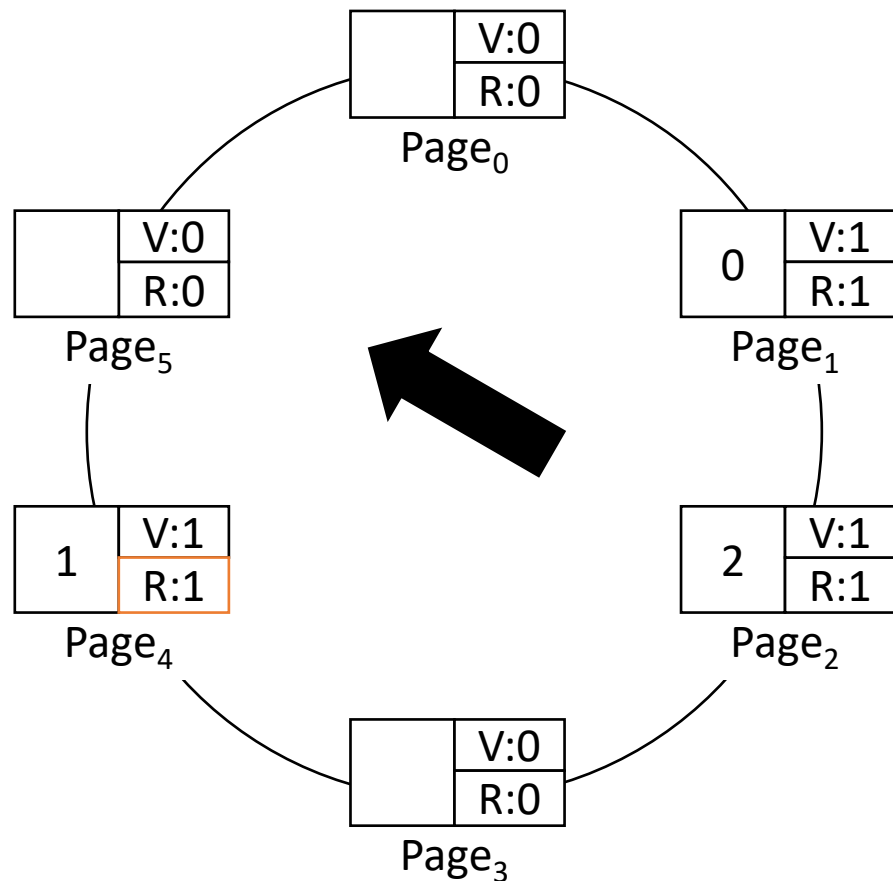
- Pages accessed: 2, 4, 5, 1, 4, 2, 4      Free frames: ~~0~~, ~~1~~, ~~2~~



- Hand points to a page with ref bit 0!
  - We've found our victim.
- Evict page 5 (mark invalid), and assign its old frame (frame 2) to the faulting page (page 2).

# Clock Example

- Pages accessed: 2, 4, 5, 1, 4, 2, 4      Free frames: ~~0~~, ~~1~~, ~~2~~



- Access page 4
  - Page 4 is already in memory: no fault
  - OS does nothing!
  - MMU hardware sets ref bit to 1

# Comparing Performance of Clock to LRU

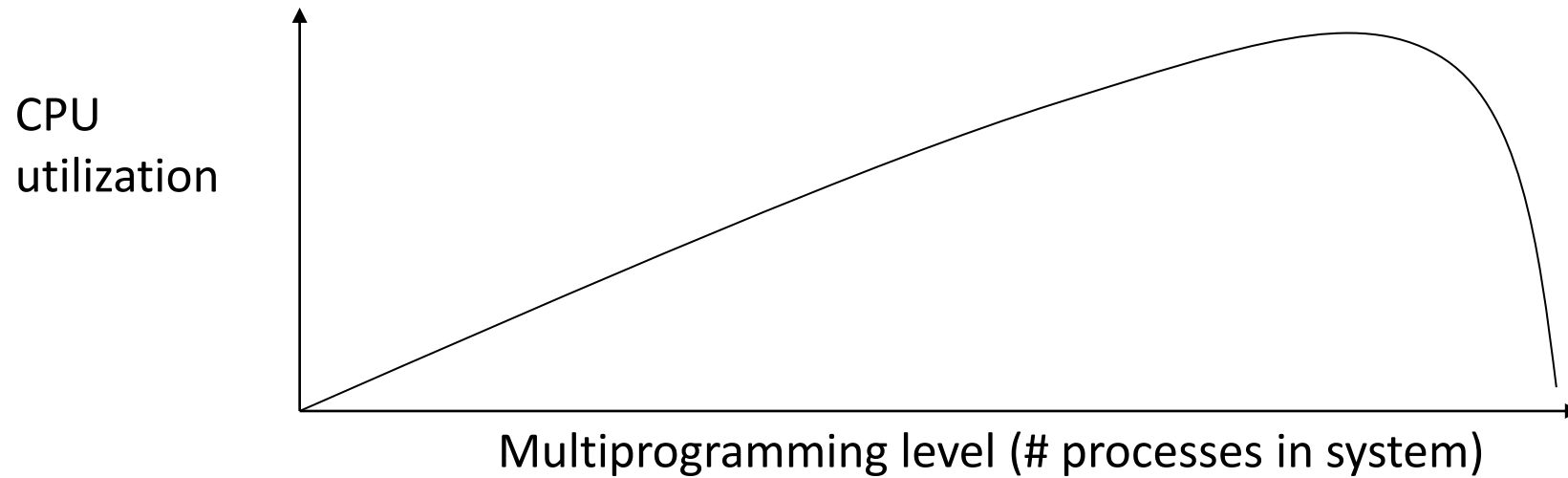
- Lab assignment!



# Recall old assumption...

- For now, assume one process and that it has a fixed number of frames.
- Reality: multiprogramming! Lots of processes available. They all need memory.
- How do we decide how much memory to give each one?

# Multiprogramming



- Having more processes to choose from keeps CPU busy.
- TOO many processes causes us to spend all our time shuffling data to/from disk: thrashing.

# How do we decide how much memory to give each process, when there are many? Why?

- A. Give each process the same amount of memory (# of frames), and perform page replacement among a process's own frames. (Local replacement)
- B. Allow each process to have varying amounts of memory (# of frames), and perform page replacement among all the frames in the system. (Global replacement)
- C. Something else. (What?)

# Assigning Frames to Processes...

- Local replacement:
  - Fair to all processes – they all get equal memory.
  - Some processes are MUCH larger than others. What size do we choose...?
- Global replacement:
  - Better reflects diversity in process memory needs.
  - Processes are now competing with one another, one bad process might ruin everything.

# Hybrid Approach

- Processes don't directly take pages from one another (like local), but OS examines processes to see if they're using the memory they've been given. If not, reclaim some for others (like global).
- Idealized solution: Denning's "working set"
- More realistic: Page fault frequency

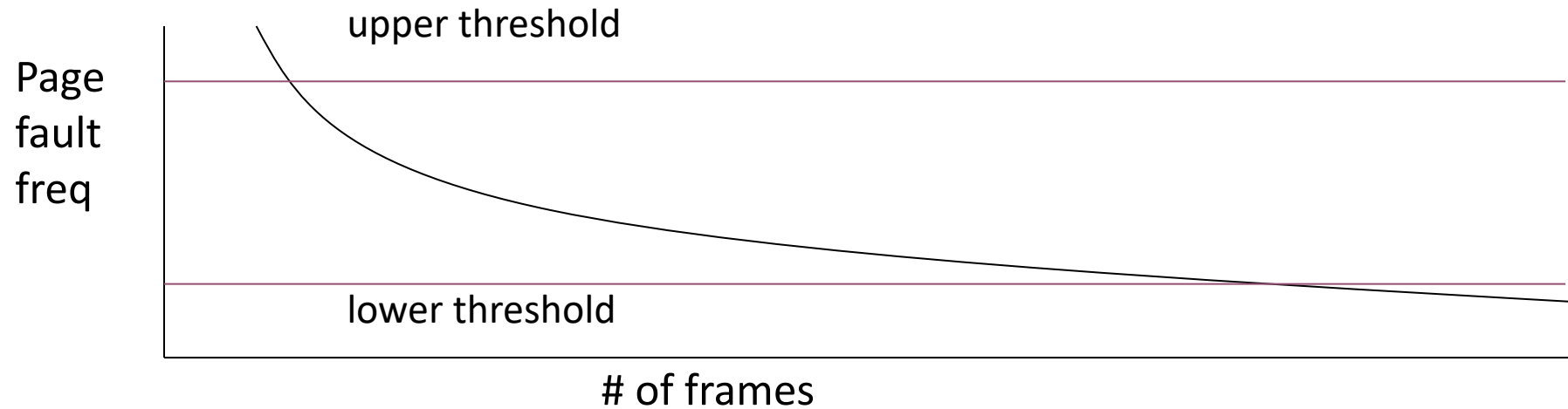
# Working Set Model

- Goal: Determine how many frames to assign a process.
- Intuition: use the set of pages a process is *actually using* right now (so that we know the number of frames it needs to store them).
- *working set*: the number of pages referenced in the interval  $(t, t - w)$
- Few (none?) commercial systems track working set precisely due to cost of doing so, but it's an important theoretical concept.

# Working Set: Challenges

- Must timestamp pages in working set to identify set size
- Must determine time interval  $w$
- Bottom line: working set is interesting as an abstract model, but not reasonable to build.

# Page Fault Frequency



- If fault frequency too high, working set not present
  - Give process more frames
- If fault frequency too low, resident set too large
  - Take away frames

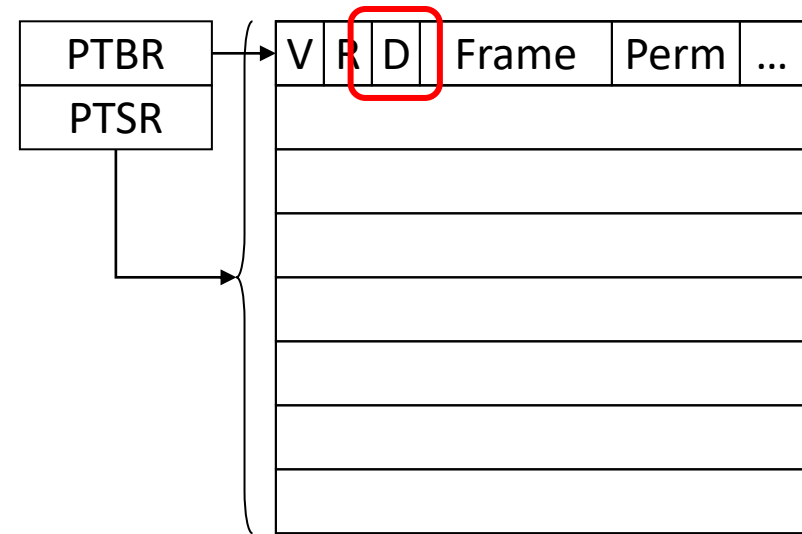


# Policy Decisions for Virtual Memory

- Placement: Where should we put items in physical memory.
  - Irrelevant for page-based systems. Any frame is equally good.
- Replacement: Which page should we evict from memory to disk?
  - Which page do we pick?
  - Local vs global: Which process should the page come from?
- **Cleaning: for modified (dirty) pages, when to write them to disk?**

# Page Table: Revisited (again) (again)

- One table per process
- Table parameters in memory
  - Page table base register
  - Page table size register
- Table entry elements
  - V: valid bit
  - R: referenced bit
  - D: dirty bit
  - Frame: location in phy mem
  - Perm: access permissions



Has this page been written?

If so, it **no longer matches** the contents on disk.

# When should we write a page to disk?

- Consider:
  - correctness
  - performance
  - practicality
  - any other factors

# On a page fault...

- If there are no free frames, we must evict a page.
  - If an identical copy of victim page is on disk, no write necessary.
  - If victim page is dirty (or not on disk at all), OS must write it to disk first.
- Problem? Not for correctness, but this isn't great for performance.
  - Not only do we need to read a page from disk, now we have to write one too!
  - Double the disk latency...

# Paging Daemon

- “Daemon”: system background process (see Wikipedia for etymology)
- Paging daemon: if the system has spare CPU cycles, check memory. If it looks like a page is likely to be swapped to disk soon, go ahead and write it to disk now!
  - (e.g., page isn't referenced recently, clock hand near its entry)
- Intuition: keep a small reserve of free frames, do writes in advance of eviction.

# Summary

- Virtual memory effectively extends main memory by swapping to disk.
- Disk is slow and must be used judiciously. Selecting which pages to swap, page replacement, is a challenging problem.
- Real systems typically implemented approximations of idealized policies (e.g., clock vs. LRU).