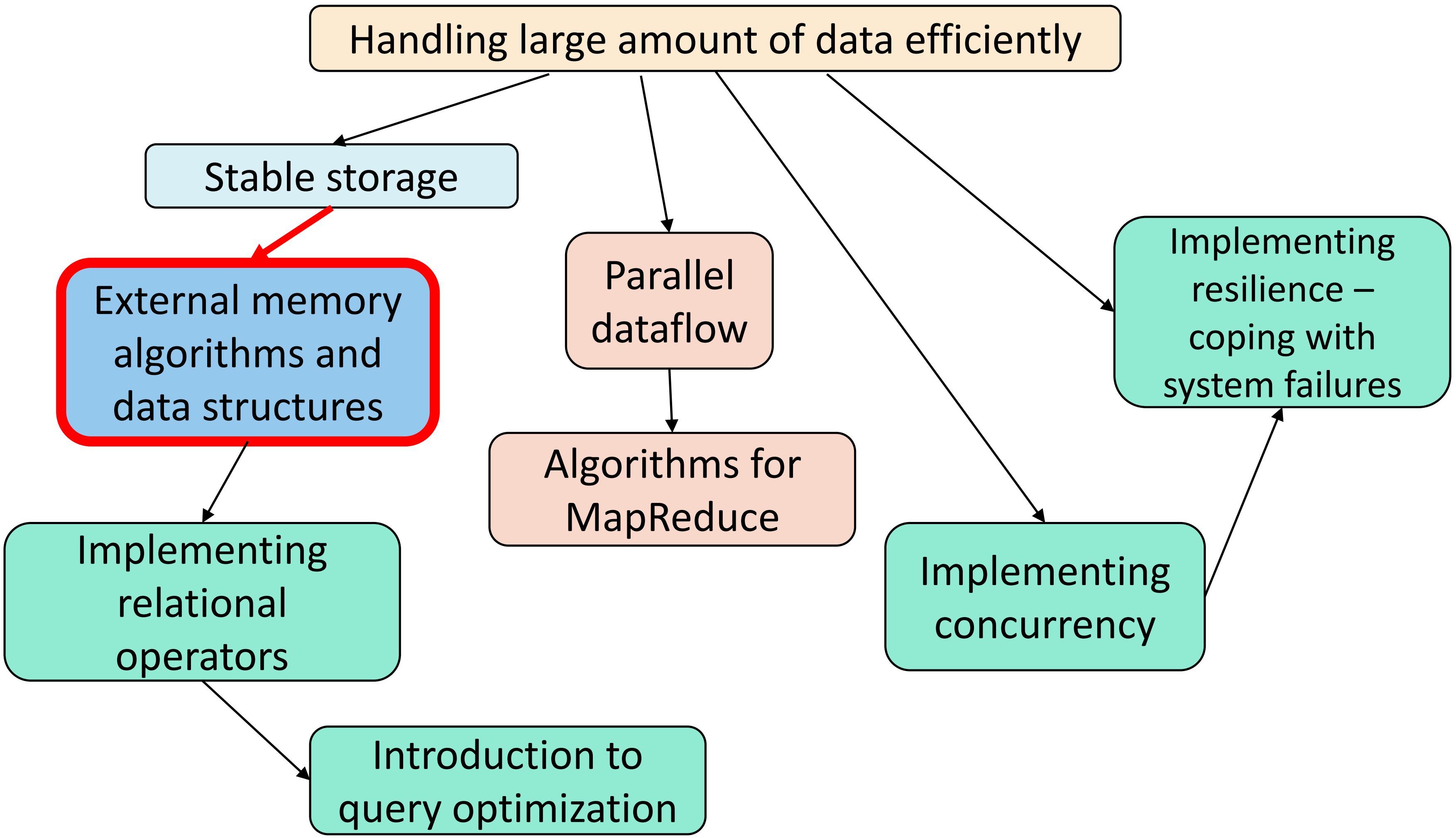
Roadmap


Handling large amount of data efficiently
Stable storage
External memory algorithms and data structures
Parallel dataflow
Algorithms for MapReduce
Implementing resilience – coping with system failures
Implementing relational operators
Implementing concurrency
Introduction to query optimization

Lecture 02.03

Data transfer between disk and RAM

Buffer management in DBMS

By Marina Barsky
Winter 2017, University of Toronto

Data must be in RAM to operate on it!

Buffering disk blocks in main memory

- In order to provide efficient access to disk block data, every DBMS implements a large shared **buffer pool** in its own memory space
- The buffer pool is organized as an array of ***frames***, each **frame size** corresponds precisely to a size of a single database disk ***block***.
- Blocks are copied in native format from disk directly into frames, manipulated in memory in native format, and written back.

Buffer pool

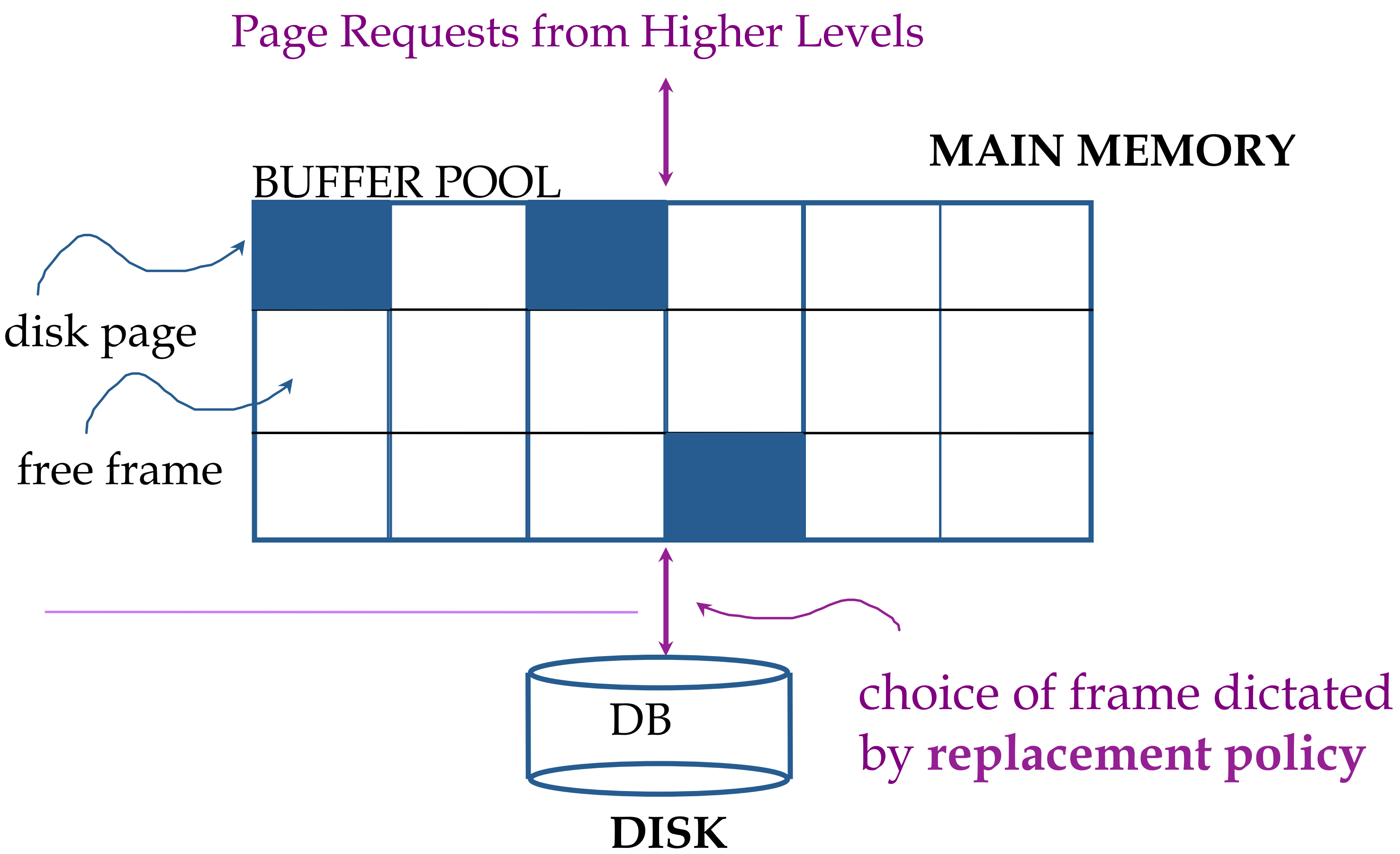


Table of <frame#,pageid> pairs is maintained.

Page table

- Associated with the array of frames is an array of metadata called a ***page table***
- The ***page table*** contains for each frame:
  - the disk location for the corresponding page
  - a ***dirty bit*** to indicate whether the page has changed since it was read from disk
  - any information needed by the ***page replacement policy*** used for choosing pages to evict on overflow.
  - a ***pin count*** for the page in the frame

Pinning and unpinning pages

- Each frame with a page in it has an associated ***pin count***
- The task requests the page – and "pins" it by incrementing the pin count before manipulating the page, and decrementing it thereafter (unpins)
- The same page in pool may be requested many times, and pin count changes accordingly
- When requestor of a page unpins it, it sets a ***dirty*** **bit** to indicate that this page has been modified.
- A page is a candidate for replacement **iff** ***pin count*** **= 0**
- Concurrency & recovery may entail additional I/O when a frame is chosen for replacement (*Write-Ahead Log* protocol) →Buffer manager tries *to **not replace dirty pages** – it may be expensive*

Algorithm for loading pages into buffer

if requested *page* **is not** in pool

  choose a *frame* for *the page:*

  if there are **no free frames**:

    choose the *frame* for replacement

    if *frame* **is dirty**: write it to disk

  read requested *page* into chosen *frame*

  pin the *page* and return its address.

If requests can be predicted (e.g., sequential scans) pages can be **pre-fetched** several pages at a time

Page faults

- A **page fault** occurs when a program requests data from a disk block and the corresponding page is not in buffer pool
- The thread is put into a *wait* state while the operating system finds the page on disk and loads it into a memory frame
- The goal is to have the least number of page faults possible – to always keep necessary pages in buffer
- This is not possible with very large databases. We need to be able to replace pages in the buffer pool

Page replacement policy

Frame is chosen for replacement by a ***replacement policy:***

- First-in-first-out (FIFO)
- Least-recently-used (LRU)
- Clock
- Most-recently-used (MRU)

  etc.

Least Recently Used (LRU)

- Replace the page that has not been referenced for the longest time

- Maintain a stack (queue?) of recently used pages

LRU page replacement algorithm: sample run

4 RAM frames

Req.	c	a	d	b	e	b	a	b	c	d
	c	c	c	c						
		a	a	a						
			d	d						
				b						
	F	F	F	F						

LRU page replacement algorithm: sample run

c a d b

Req.	c	a	d	b	e	b	a	b	c	d
	c	c	c	c	e					
		a	a	a	a					
			d	d	d					
				b	b					
	F	F	F	F	F					

4 RAM frames

LRU page replacement algorithm: sample run

4 RAM frames

Req.	c	a	d	b	e	b	a	b	c	d
	c	c	c	c	e	e	e	e		
		a	a	a	a	a	a	a		
			d	d	d	d	d	d		
				b	b	b	b	b		
	F	F	F	F	F					

LRU page replacement algorithm: sample run

d e a b

4 RAM frames

Req.	c	a	d	b	e	b	a	b	c	d
	c	c	c	c	e	e	e	e	e	
		a	a	a	a	a	a	a	a	
			d	d	d	d	d	d	c	
				b	b	b	b	b	b	
	F	F	F	F	F				F	

LRU page replacement algorithm: sample run

Which page is evicted next?

4 RAM frames

Req.	c	a	d	b	e	b	a	b	c	d
	c	c	c	c	e	e	e	e	e	
		a	a	a	a	a	a	a	a	
			d	d	d	d	d	d	c	
				b	b	b	b	b	b	
	F	F	F	F	F				F	

LRU page replacement algorithm: sample run

e a b c

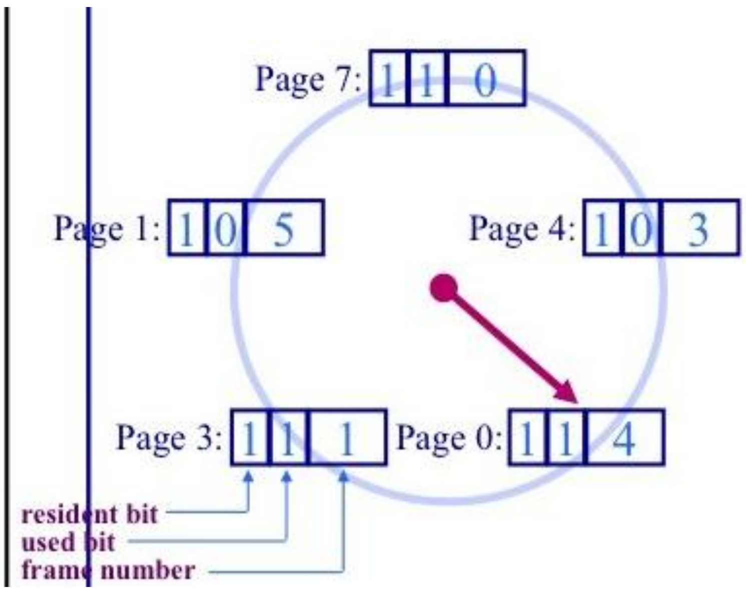
4 RAM frames

Req.	c	a	d	b	e	b	a	b	c	d
	c	c	c	c	e	e	e	e	e	d
		a	a	a	a	a	a	a	a	a
			d	d	d	d	d	d	c	c
				b	b	b	b	b	b	b
	F	F	F	F	F				F	F

7 page faults for 10 requests

Approximate LRU: the Clock algorithm

- Maintain a circular list of pages
  - Use a clock bit (*used*) to track accessed page
  - The bit is set to 1 whenever a page is referenced
- Clock hand sweeps over pages looking for one with used bit= 0
  - Replace pages that haven't been referenced for one complete revolution of the clock

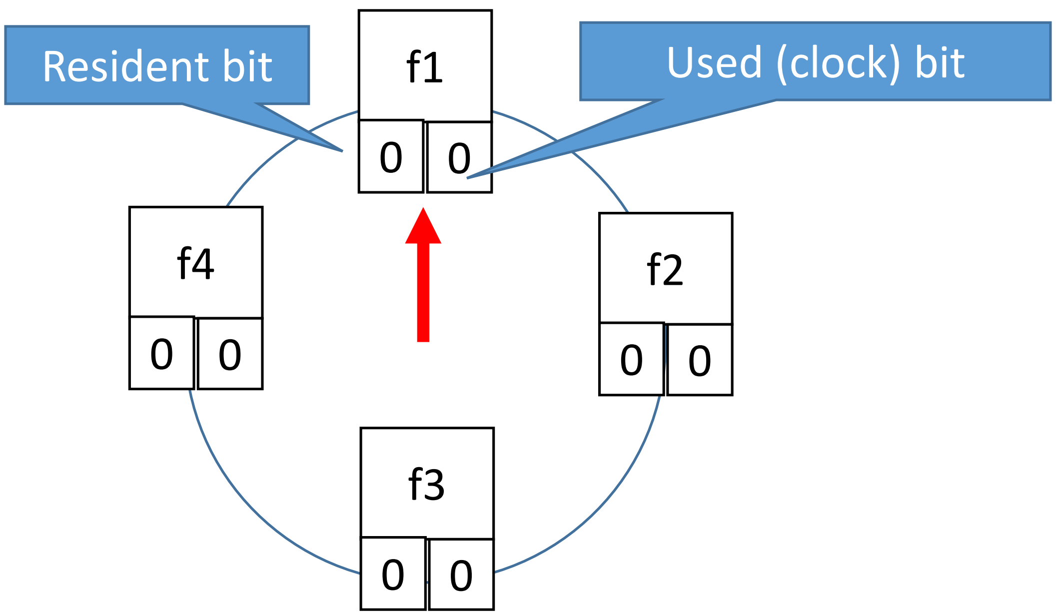



```
func Clock_Replacement
begin
  while (victim page not found) do
    if (used bit for current page = 0) then
      replace current page
    else
      reset used bit
    end if
    advance clock pointer
  end while
end Clock_Replacement
```

The Clock replacement algorithm

4 RAM frames

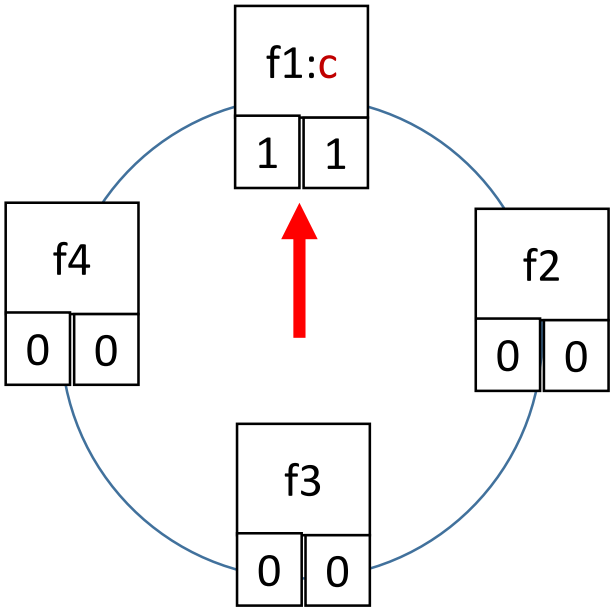
Req.	c	a	d	b	e	c	a	b	c	d
f1										
f2										
f3										
f4										



The Clock replacement algorithm

4 RAM frames

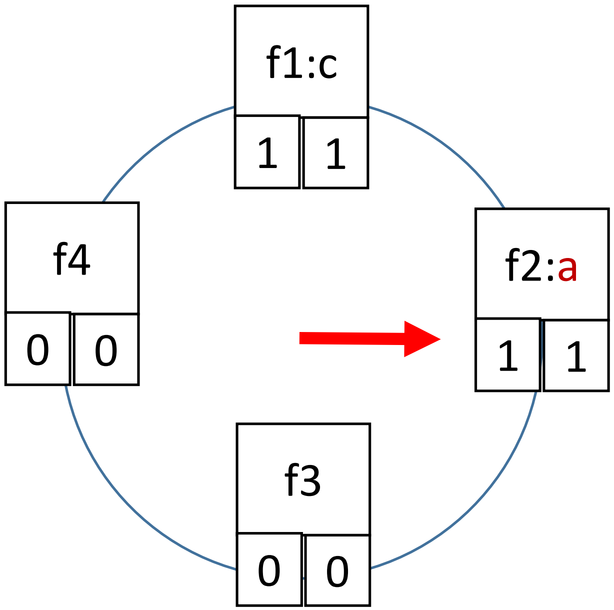
Req.	c	a	d	b	e	c	a	b	c	d
f1	c									
f2										
f3										
f4										
	F									



The Clock replacement algorithm

4 RAM frames

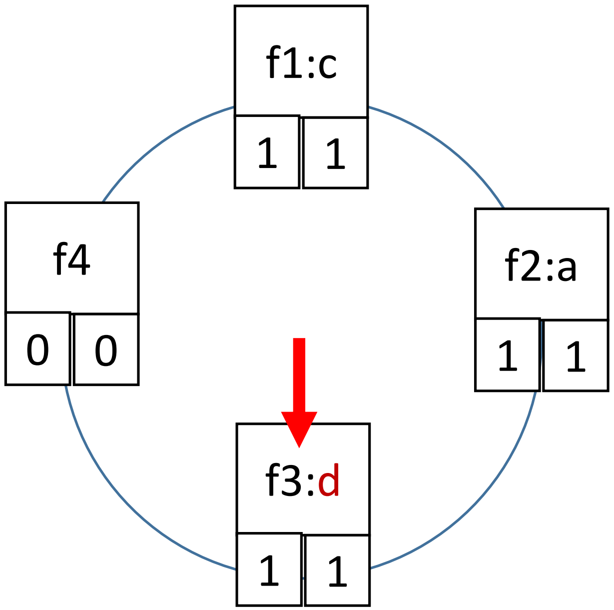
Req.	c	a	d	b	e	c	a	b	c	d
f1	c	c								
f2		a								
f3										
f4										
	F	F								



The Clock replacement algorithm

4 RAM frames

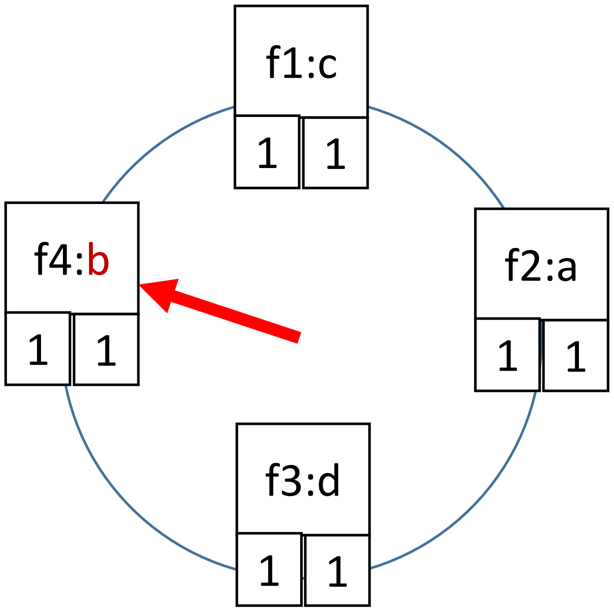
Req.	c	a	d	b	e	c	a	b	c	d
f1	c	c	c							
f2		a	a							
f3			d							
f4										
	F	F	F							



The Clock replacement algorithm

Req.	c	a	d	b	e	c	a	b	c	d
f1	c	c	c	c						
f2		a	a	a						
f3			d	d						
f4				b						
	F	F	F	F						

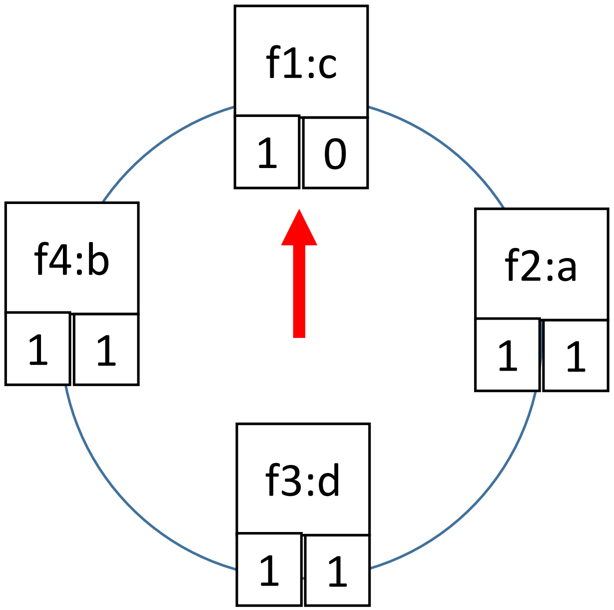
4 RAM frames



The Clock replacement algorithm

4 RAM frames

Req.	c	a	d	b	e	c	a	b	c	d
f1	c	c	c	c						
f2		a	a	a						
f3			d	d						
f4				b						
	F	F	F	F						

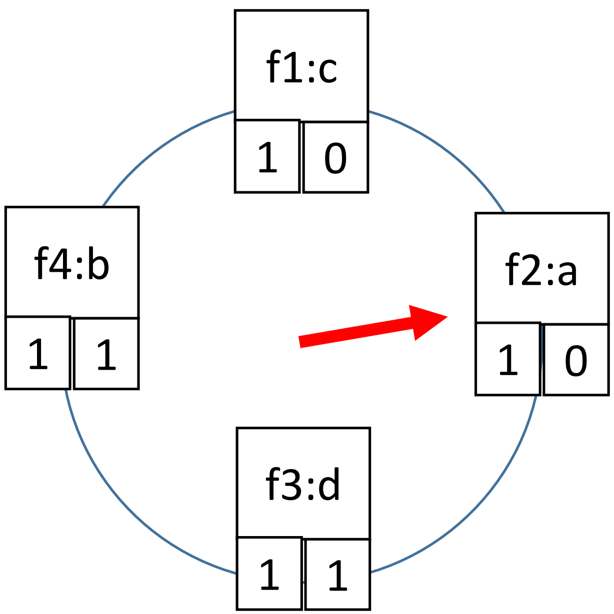


Searching for page to evict

The Clock replacement algorithm

4 RAM frames

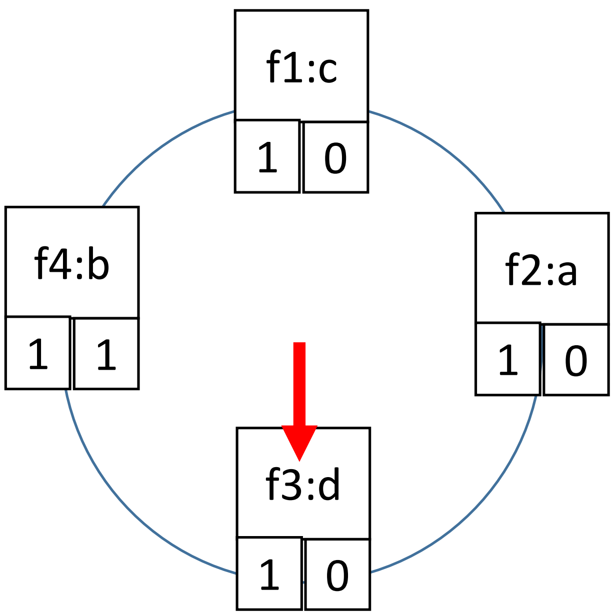
Req.	c	a	d	b	e	c	a	b	c	d
f1	c	c	c	c						
f2		a	a	a						
f3			d	d						
f4				b						
	F	F	F	F						



The Clock replacement algorithm

Req.	c	a	d	b	e	c	a	b	c	d
f1	c	c	c	c						
f2		a	a	a						
f3			d	d						
f4				b						
	F	F	F	F						

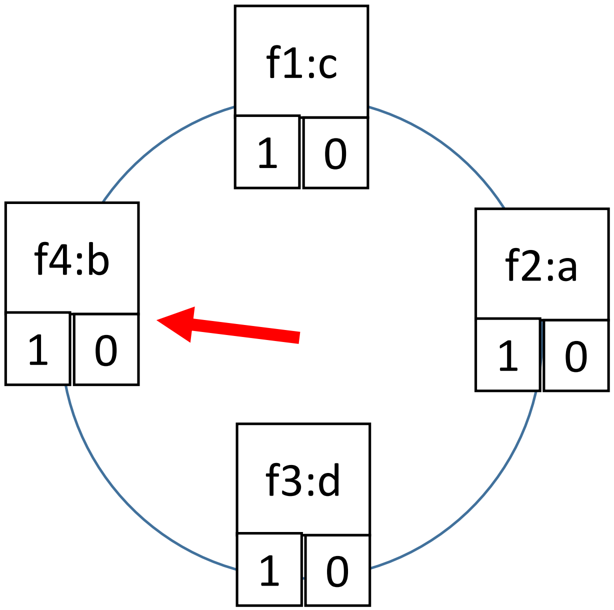
4 RAM frames



The Clock replacement algorithm

Req.	c	a	d	b	e	c	a	b	c	d
f1	c	c	c	c						
f2		a	a	a						
f3			d	d						
f4				b						
	F	F	F	F						

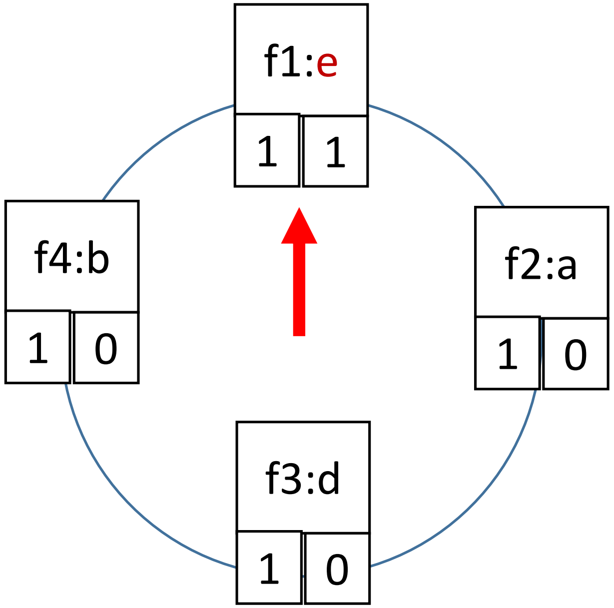
4 RAM frames



The Clock replacement algorithm

Req.	c	a	d	b	e	c	a	b	c	d
f1	c	c	c	c	e					
f2		a	a	a	a					
f3			d	d	d					
f4				b	b					
	F	F	F	F	F					

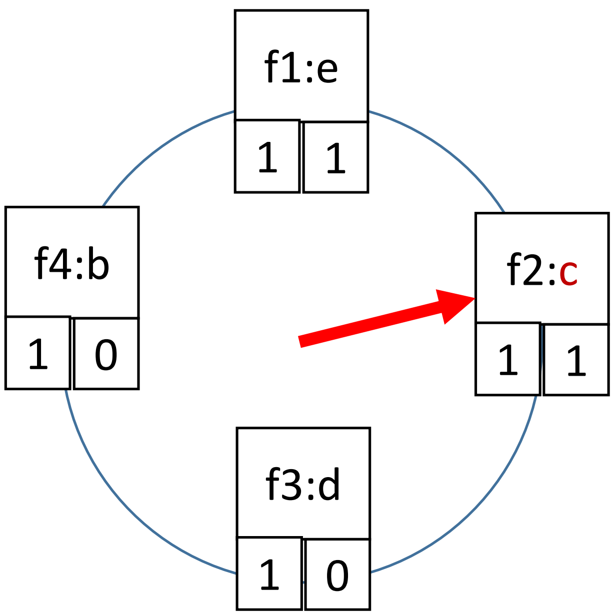
4 RAM frames



Evicted page c

The Clock replacement algorithm

	Req.	c	a	d	b	e	c	a	b	c	d
4 RAM frames	f1	c	c	c	c	e	e				
	f2		a	a	a	a	c				
	f3			d	d	d	d				
	f4				b	b	b				
		F	F	F	F	F	F				

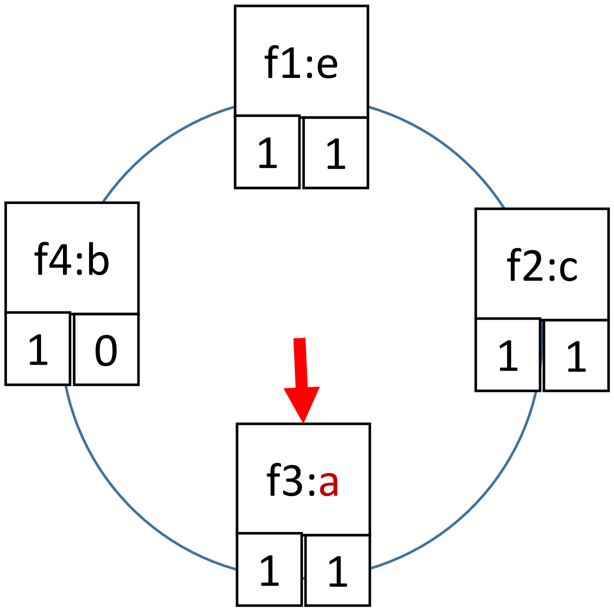


Evicted page a – its clock bit was 0

The Clock replacement algorithm

Req.	c	a	d	b	e	c	a	b	c	d
f1	c	c	c	c	e	e	e			
f2		a	a	a	a	c	c			
f3			d	d	d	d	a			
f4				b	b	b	b			
	F	F	F	F	F	F	F			

4 RAM frames

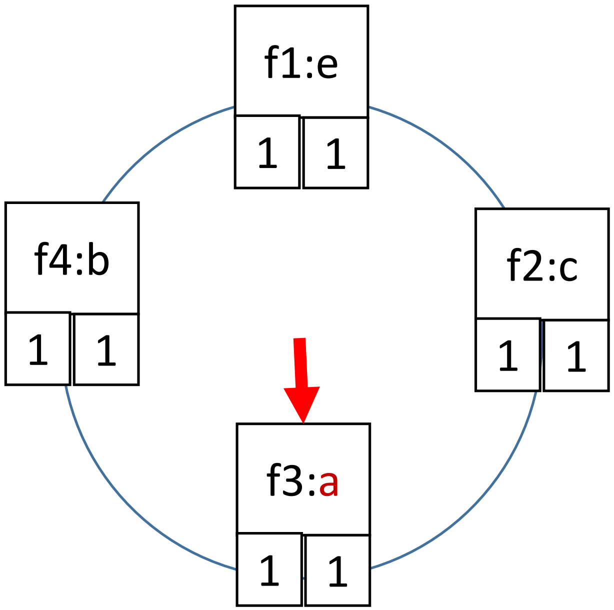


Evicted page d

The Clock replacement algorithm

4 RAM frames

Req.	c	a	d	b	e	c	a	b	c	d
f1	c	c	c	c	e	e	e	e		
f2		a	a	a	a	c	c	c		
f3			d	d	d	d	a	a		
f4				b	b	b	b	b		
	F	F	F	F	F	F	F			

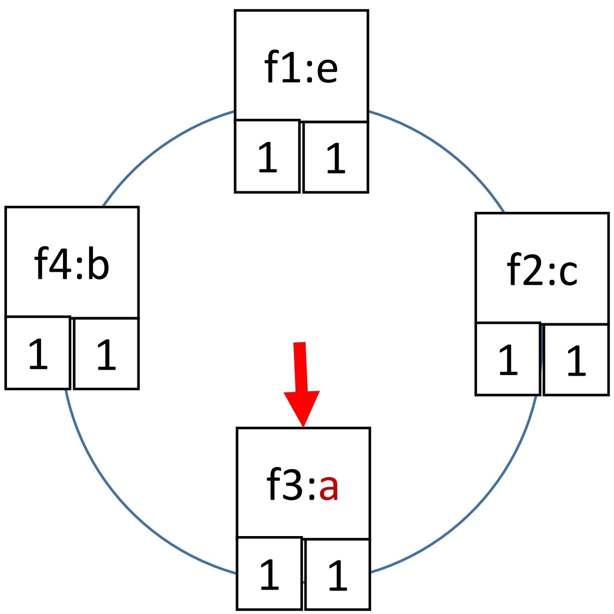


We know that b is in buffer

The Clock replacement algorithm

4 RAM frames

Req.	c	a	d	b	e	c	a	b	c	d
f1	c	c	c	c	e	e	e	e	e	
f2		a	a	a	a	c	c	c	c	
f3			d	d	d	d	a	a	a	
f4				b	b	b	b	b	b	
	F	F	F	F	F	F	F			

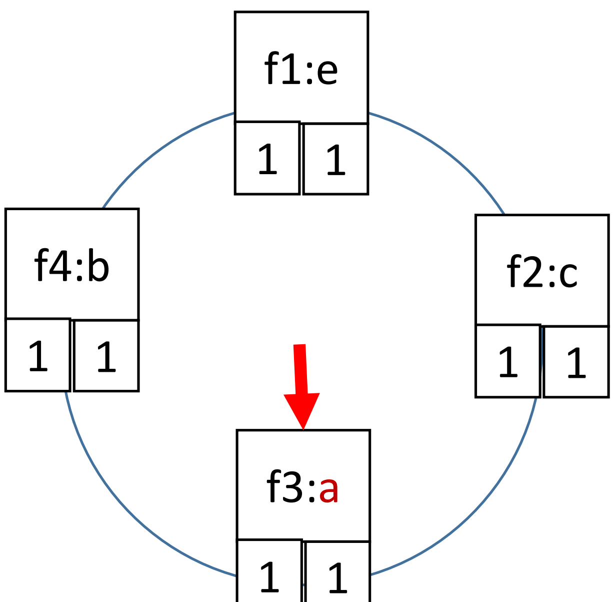


We know that c is in buffer

The Clock replacement algorithm

4 RAM frames

Req.	c	a	d	b	e	c	a	b	c	d
f1	c	c	c	c	e	e	e	e	e	
f2		a	a	a	a	c	c	c	c	
f3			d	d	d	d	a	a	a	
f4				b	b	b	b	b	b	
	F	F	F	F	F	F	F			



Where does *d* go:

A: frame 1

B: frame 2

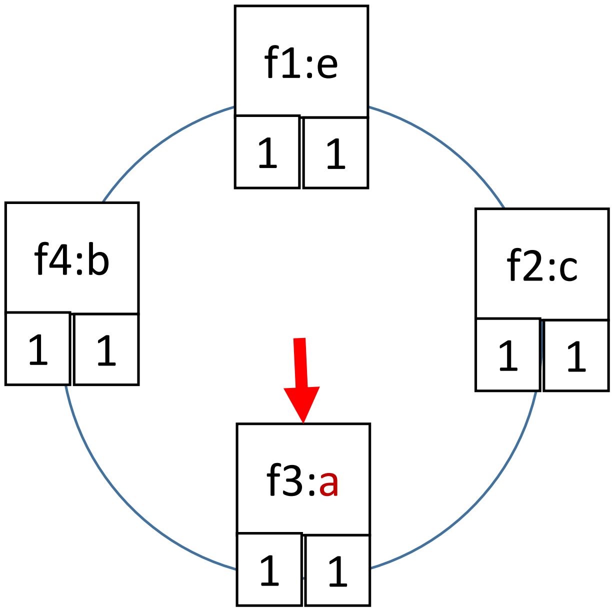
C: frame 3

D: frame 4

The Clock replacement algorithm

Req.	c	a	d	b	e	c	a	b	c	d
f1	c	c	c	c	e	e	e	e	e	
f2		a	a	a	a	c	c	c	c	
f3			d	d	d	d	a	a	a	
f4				b	b	b	b	b	b	
	F	F	F	F	F	F	F			

4 RAM frames



Where does *d* go:

A: frame 1

B: frame 2

C: frame 3

D: frame 4

Optimizing Clock: The Second Chance algorithm

- There is a significant cost to replacing *dirty* pages
- Modified Clock algorithm allows dirty pages to always survive one sweep of the clock hand
- Assuming there are 4 classes of pages:

	Used bit	Dirty bit
Class 1	0	0
Class 2	0	1
Class 3	1	0
Class 4	1	1

- During the first revolution of the clock – unset the used bit as before, and evict page of Class 1 if found
- If no pages of Class 1 found during the entire revolution, in the next revolution evict page of Class 2, writing its dirty content to disk

Replacement policy depends on data access pattern

- Policy can have big impact on # of I/O's; depends on the *access pattern*.
- ***Sequential flooding***: Nasty situation caused by LRU + repeated sequential scans.

  # buffer frames < # pages in file

  In this case each page request causes an I/O. MRU much better in this situation (but not in other situations)

Sequential flooding example: nested loop

A: a, b
B: c, d, e, f

for each record *i* **in** *A*
  for each record *j* **in** *B*
    do something with *i* and *j*

3 frames for A - enough

3 frames for B < |B|

req.	a	a	a	a	b	b	b	b
req.	c	d	e	f	c	d	e	f

Sequential flooding example: nested loop

A: a, b
B: c, d, e, f

for each record *i* **in** *A*
  for each record *j* **in** *B*
    do something with *i* and *j*

	req.	a	a	a	a	b	b	b	b
3 frames for A - enough		a	a	a					
		F							
	req.	c	d	e	f	c	d	e	f
3 frames for B **< |B|**		c	c	c					
			d	d					
				e					
		F	F	F					

Sequential flooding example: nested loop

A: a, b
B: c, d, e, f

for each record *i* **in** *A*
  for each record *j* **in** *B*
    do something with *i* and *j*

	req.	a	a	a	a	b	b	b	b
3 frames for A - enough		a	a	a	a				
		F							
	req.	c	d	e	f	c	d	e	f
3 frames for B **< |B|**		c	c	c	f				
			d	d	d				
				e	e				
		F	F	F	F				

f evicts *c*

Sequential flooding example: nested loop

A: a, b
B: c, d, e, f

for each record *i* **in** *A*
  for each record *j* **in** *B*
    do something with *i* and *j*

3 frames for A - enough

3 frames for B **< |B|**

req.	a	a	a	a	b	b	b	b
	a	a	a	a	a			
					b			
	F				F			
req.	c	d	e	f	c	d	e	f
	c	c	c	f	f			
		d	d	d	c			
			e	e	e			
	F	F	F	F	F			

c evicts *d*

Sequential flooding example: nested loop

A: a, b
B: c, d, e, f

for each record *i* **in** *A*
  for each record *j* **in** *B*
    do something with *i* and *j*

	req.	a	a	a	a	b	b	b	b
3 frames for A - enough		a	a	a	a	a	a		
						b	b		
		F				F			
	req.	c	d	e	f	c	d	e	f
3 frames for B **< |B|**		c	c	c	f	f	f		
			d	d	d	c	c		
				e	e	e	d		
		F	F	F	F	F	F		

d evicts *e*

Sequential flooding example: nested loop

A: a, b
B: c, d, e, f

for each record *i* **in** *A*
  for each record *j* **in** *B*
    do something with *i* and *j*

3 frames for A - enough

3 frames for B **< |B|**

req.	a	a	a	a	b	b	b	b
	a	a	a	a	a	a	a	
					b	b	b	
	F				F			
req.	c	d	e	f	c	d	e	f
	c	c	c	f	f	f	e	
		d	d	d	c	c	c	
			e	e	e	d	d	
	F	F	F	F	F	F	F	

e evicts *f*

Sequential flooding example: nested loop

A: a, b
B: c, d, e, f

for each record *i* **in** *A*
  for each record *j* **in** *B*
    do something with *i* and *j*

	req.	a	a	a	a	b	b	b	b
3 frames for A - enough		a	a	a	a	a	a	a	a
						b	b	b	b
		F				F			
	req.	c	d	e	f	c	d	e	f
3 frames for B **< |B|**		c	c	c	f	f	f	e	e
			d	d	d	c	c	c	f
				e	e	e	d	d	d
		F	F	F	F	F	F	F	F

f evicts *c*

Sequential flooding example: nested loop

A: a, b
B: c, d, e, f

for each record *i* **in** *A*
  for each record *j* **in** *B*
    do something with *i* and *j*

	req.	a	a	a	a	b	b	b	b
3 frames for A - enough		a	a	a	a	a	a	a	a
						b	b	b	b
		F				F			
	req.	c	d	e	f	c	d	e	f
3 frames for B **< |B|**		c	c	c	f	f	f	e	e
			d	d	d	c	c	c	f
				e	e	e	d	d	d
		F	F	F	F	F	F	F	F

Sequential flooding – each request – page fault

LRU happens to evict exactly the page which we will need next!

Sequential flooding

- **The LRU page which we evict is always *exactly the page we are going to want to read next!***
- We end up having to read a page from disk ***for every page we read from the buffer***
- This seems like an incredibly pointless use of a buffer!

Solutions

- General idea: to tune the replacement strategy via query plan information
- Most systems use simple enhancements to LRU schemes to account for the case of nested loops;
  - Implemented in commercial systems - LRU-2 (evicts second to last frame)
  - The replacement policy depending on the page type: e.g. the root of a B+-tree might be replaced with a different strategy than a page in a heap file, nested loops use MRU etc.

DBMS vs OS buffer management

- DBMS is better at predicting the data access patterns
- Buffer management in DBMS is able to:
  - Pin a page in buffer pool
  - Force a page to disk (required to implement CC & recovery)
  - Adjust replacement policy
  - Pre-fetch pages based on predictable access patterns
- This allows DBMS to
  - Amortize rotational and seek costs
  - Better control the overlap of I/O with computation (double-buffering)
  - Leverage multiple disks

Think about

- Think about how would you implement LRU replacement policy? Clock algorithm? LRU-2?
- What is the difference between FIFO and LRU?
- Show on an example how MRU can be used to prevent sequential flooding
- Example of a computation where LRU is efficient

Similar questions could be on the third quiz and on the exam