COMP 4735: Operating Systems Concepts

Lesson 12: Page Replacement Algorithms

Rob Neilson

rneilson@bcit.ca

Administrative stuff ...

- Reading: Virtual Memory is covered in chapters 3.3, 3.4
- Next quiz: Monday March 30th
  - covers chapters 3.1, 3.2, 3.3 plus all stuff from lecture
  - note: chapter 3.4 will be covered in a later quiz

Paging Algorithms

Paging Algorithms and Policies ...

Any paging algorithm must define:

- fetch policy
  - decides when a page should be loaded into memory
- replacement policy
  - decides which page should be moved to disk when all pages are full
- placement policy
  - decides where the page should be loaded in primary memory

There are two types of paging algorithms:

- static paging algorithms
  - each process uses a fixed number of frames during its lifetime
- dynamic paging algorithms
  - the number of frames allocated to a process changes during its lifetime

Static Paging Algorithms

- number of page frames does not change throughout lifetime of the process
- this makes the *placement policy* rather simple ...
  - assume that all of a processes page frames are full
    (this is true at all times except when process first starts)
  - assume a new page is needed
  - the replacement policy is invoked to select a victim, and the selected page is moved to disk
  - the new page is loaded into the frame that was just vacated
- this is always the pattern for static paging algorithms
- in dynamic paging algorithms we have other choices, such as to grow the number of frames allocated to the process

Demand Paging

- "Demand Paging" is a *fetch policy*
  - this is the typical policy that says to load a page only when it has been referenced
    - ie: this is what we have been talking about in previous slides/lectures
- What are the alternatives?
  - we could try to predict when a page will be needed, and load it ahead of time
  - in this case we are essentially performing a"prefetch"
- Static Paging Algorithms all use the **demand paging** fetch policy
- Static Paging Algorithms all use the simple **fetch policy** described on the previous page

this means that we only need to consider* replacement policies *in our study of static paging algorithms

Static Page Frame Allocation with Demand Paging

The following Static Page Replacement algos will be considered:

- The Optimal Algorithm
- First In First Out (FIFO)
- Least Recently Used (LRU)
- Not Frequently Used (NFU)

Background

- In looking at these algorithms, we will consider a continuous stream of page requests R. For example:

  R = 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5, 6

  - The above example tells us that page 0 is requested first, then page 1, and so on.

- We also consider the state S of the system at any point in time.
  - the state is just a snapshot of the pages that are currently loaded. For example:

    S = { 3, 6, 9, 2 }

  - this tells us that there are 4 available frames, and in the current state:
    - frame 0 holds page 3
    - frame 1 holds page 6
    - frame 2 holds page 9
    - frame 3 holds page 2

Diagramming Page Replacement - 1

Assume we are given:

- a reference stream R (eg: R = 3, 2, 0, 7, 0, 3, 4, 1, 4, 5)
- an initial state S (eg: S = { })
- a replacement policy (eg: FIFO – first page in is next one out)

We are now able to illustrate how the page replacement policy works, for example:

Request			3	2	0	7	0	3	4	1	4	5
Frame	0	-										
	1	-										
	2	-										
	3	-										
Page faults												

Diagramming Page Replacement - 2

Assume we are given:

- a reference stream R (eg: R = 3, 2, 0, 7, 0, 3, 4, 1, 4, 5)
- an initial state S (eg: S = { })
- a replacement policy (eg: FIFO – first page in is next one out)

We are now able to illustrate how the page replacement policy works, for example:

Request			3	2	0	7	0	3	4	1	4	5
Frame	0	-	3									
	1	-										
	2	-										
	3	-										
Page faults			*									

Diagramming Page Replacement - 3

Assume we are given:

- a reference stream R (eg: R = 3, 2, 0, 7, 0, 3, 4, 1, 4, 5)
- an initial state S (eg: S = { })
- a replacement policy (eg: FIFO – first page in is next one out)

We are now able to illustrate how the page replacement policy works, for example:

Request			3	2	0	7	0	3	4	1	4	5
Frame	0	-	<u>3</u>	3								
	1	-		<u>2</u>								
	2	-										
	3	-										
Page faults			*	*								

Diagramming Page Replacement - 4

Assume we are given:

- a reference stream R (eg: R = 3, 2, 0, 7, 0, 3, 4, 1, 4, 5)
- an initial state S (eg: S = { })
- a replacement policy (eg: FIFO – first page in is next one out)

We are now able to illustrate how the page replacement policy works, for example:

Request			3	2	0	7	0	3	4	1	4	5
Frame	0	-	**3**	3	3							
	1	-		**2**	2							
	2	-			**0**							
	3	-										
Page faults			*	*	*							

Diagramming Page Replacement - 5

Assume we are given:

- a reference stream R (eg: R = 3, 2, 0, 7, 0, 3, 4, 1, 4, 5)
- an initial state S (eg: S = { })
- a replacement policy (eg: FIFO – first page in is next one out)

We are now able to illustrate how the page replacement policy works, for example:

Request			3	2	0	7	0	3	4	1	4	5
Frame	0	-	**3**	3	3	3						
	1	-		**2**	2	2						
	2	-			**0**	0						
	3	-				**7**						
Page faults			*	*	*	*						

Diagramming Page Replacement - 6

Assume we are given:

- a reference stream R (eg: R = 3, 2, 0, 7, 0, 3, 4, 1, 4, 5)
- an initial state S (eg: S = { })
- a replacement policy (eg: FIFO – first page in is next one out)

We are now able to illustrate how the page replacement policy works, for example:

Request			3	2	0	7	0	3	4	1	4	5
Frame	0	-	**3**	3	3	3	3					
	1	-		**2**	2	2	2					
	2	-			**0**	0	**0**					
	3	-				**7**	7					
Page faults			*	*	*	*						

Diagramming Page Replacement - 7

Assume we are given:

- a reference stream R (eg: R = 3, 2, 0, 7, 0, 3, 4, 1, 4, 5)
- an initial state S (eg: S = { })
- a replacement policy (eg: FIFO – first page in is next one out)

We are now able to illustrate how the page replacement policy works, for example:

Request			3	2	0	7	0	3	4	1	4	5
Frame	0	-	**3**	3	3	3	3	**3**				
	1	-		**2**	2	2	2	2				
	2	-			**0**	0	**0**	0				
	3	-				**7**	7	7				
Page faults			*	*	*	*						

Diagramming Page Replacement - 8

Assume we are given:

- a reference stream R (eg: R = 3, 2, 0, 7, 0, 3, 4, 1, 4, 5)
- an initial state S (eg: S = { })
- a replacement policy (eg: FIFO – first page in is next one out)

We are now able to illustrate how the page replacement policy works, for example:

Request			3	2	0	7	0	3	4	1	4	5
Frame	0	-	**3**	3	3	3	3	**3**	**4**			
	1	-		**2**	2	2	2	2	2			
	2	-			**0**	0	**0**	0	0			
	3	-				**7**	7	7	7			
Page faults			*	*	*	*			*			

Diagramming Page Replacement - 9

Assume we are given:

- a reference stream R (eg: R = 3, 2, 0, 7, 0, 3, 4, 1, 4, 5)
- an initial state S (eg: S = { })
- a replacement policy (eg: FIFO – first page in is next one out)

We are now able to illustrate how the page replacement policy works, for example:

Request			3	2	0	7	0	3	4	1	4	5
Frame	0	-	**3**	3	3	3	3	**3**	**4**	4		
	1	-		**2**	2	2	2	2	2	**1**		
	2	-			**0**	0	**0**	0	0	0		
	3	-				**7**	7	7	7	7		
Page faults			*	*	*	*			*	*		

Diagramming Page Replacement - 10

Assume we are given:

- a reference stream R (eg: R = 3, 2, 0, 7, 0, 3, 4, 1, 4, 5)
- an initial state S (eg: S = { })
- a replacement policy (eg: FIFO – first page in is next one out)

We are now able to illustrate how the page replacement policy works, for example:

Request			3	2	0	7	0	3	4	1	4	5
Frame	0	-	**3**	3	3	3	3	**3**	**4**	4	**4**	
	1	-		**2**	2	2	2	2	2	**1**	1	
	2	-			**0**	0	**0**	0	0	0	0	
	3	-				**7**	7	7	7	7	7	
Page faults			*	*	*	*			*	*		

Diagramming Page Replacement - 11

Assume we are given:

- a reference stream R (eg: R = 3, 2, 0, 7, 0, 3, 4, 1, 4, 5)
- an initial state S (eg: S = { })
- a replacement policy (eg: FIFO – first page in is next one out)

We are now able to illustrate how the page replacement policy works, for example:

Request			3	2	0	7	0	3	4	1	4	5
Frame	0	-	**3**	3	3	3	3	**3**	**4**	4	4	4
	1	-		**2**	2	2	2	2	2	**1**	**1**	1
	2	-			**0**	0	**0**	0	0	0	0	**5**
	3	-				**7**	7	7	7	7	7	7
Page faults			*	*	*	*			*	*		*

The optimal page replacement algorithm - 1

Idea:

- Select the page that will not be needed for the longest time
- We do this by looking at the "forward distance" to the page
  - this is the distance from the current point in the reference stream to the next place in the stream where the same page is referenced
  - forward distance always > 0
  - forward distance == ∞ if the page is never referenced again
- In the case of ties, choose arbitrarily

- Consider a reference stream:

R = 3, 2, 0, 5, 4, 3, 1, 5, 2, 0

4 frames, all empty to start ...

The optimal page replacement algorithm - 2

Idea:

- Select the page that will not be needed for the longest time
- We do this by looking at the “forward distance” to the page
  - this is the distance from the current point in the reference stream to the next place in the stream where the same page is referenced
  - forward distance always > 0
  - forward distance == ∞ if the page is never referenced again
- In the case of ties, choose arbitrarily

Request			3	2	0	5	4	3	1	5	2	0
Frame	0	-	**3**	3	3	3						
	1	-		**2**	2	2						
	2	-			**0**	0						
	3	-				**5**						
Page faults			*	*	*	*						

The optimal page replacement algorithm - 3

Idea:

- Select the page that will not be needed for the longest time
- We do this by looking at the "forward distance" to the page
  - this is the distance from the current point in the reference stream to the next place in the stream where the same page is referenced
  - forward distance always > 0
  - forward distance == ∞ if the page is never referenced again
- In the case of ties, choose arbitrarily

current point

Request			3	2	0	5	4	3	1	5	2	0
Frame	0	-	**3**	3	3	3	*forward distance = 1*					
	1	-		**2**	2	2	*forward distance = 4*					
	2	-			**0**	0	*forward distance = 5*					
	3	-				**5**	*forward distance = 3*					
Page faults			*	*	*	*						

The optimal page replacement algorithm - 4

Idea:

- Select the page that will not be needed for the longest time
- We do this by looking at the "forward distance" to the page
  - this is the distance from the current point in the reference stream to the next place in the stream where the same page is referenced
  - forward distance always > 0
  - forward distance == ∞ if the page is never referenced again
- In the case of ties, choose arbitrarily

Request			3	2	0	5	4	3	1	5	2	0
Frame	0	-	3	3	3	3	3					
	1	-		2	2	2	2					
	2	-			0	0	4					
	3	-				5	5					
Page faults			*	*	*	*	*					

The optimal page replacement algorithm - 5

Idea:

- Select the page that will not be needed for the longest time
- We do this by looking at the "forward distance" to the page
  - this is the distance from the current point in the reference stream to the next place in the stream where the same page is referenced
  - forward distance always > 0
  - forward distance == ∞ if the page is never referenced again
- In the case of ties, choose arbitrarily

Request			3	2	0	5	4	3	1	5	2	0
Frame	0	-	**3**	3	3	3	3	**3**				
	1	-		**2**	2	2	2	2				
	2	-			**0**	0	**4**	4				
	3	-				**5**	5	5				
Page faults			*	*	*	*	*					

The optimal page replacement algorithm - 6

Idea:

- Select the page that will not be needed for the longest time
- We do this by looking at the "forward distance" to the page
  - this is the distance from the current point in the reference stream to the next place in the stream where the same page is referenced
  - forward distance always > 0
  - forward distance == ∞ if the page is never referenced again
- In the case of ties, choose arbitrarily

Request			3	2	0	5	4	3	1	5	2	0
Frame	0	-	**3**	3	3	3	3	**3**	**1**			
	1	-		**2**	2	2	2	2	2			
	2	-			**0**	0	**4**	4	4			
	3	-				**5**	5	5	5			
Page faults			*	*	*	*	*		*			

The optimal page replacement algorithm - 7

Idea:

- Select the page that will not be needed for the longest time
- We do this by looking at the “forward distance” to the page
  - this is the distance from the current point in the reference stream to the next place in the stream where the same page is referenced
  - forward distance always > 0
  - forward distance == ∞ if the page is never referenced again
- In the case of ties, choose arbitrarily

Request			3	2	0	5	4	3	1	5	2	0
Frame	0	-	**3**	3	3	3	3	**3**	**1**	1		
	1	-		**2**	2	2	2	2	2	2		
	2	-			**0**	0	**4**	4	4	4		
	3	-				**5**	5	5	5	**5**		
Page faults			*	*	*	*	*		*			

The optimal page replacement algorithm - 8

Idea:

- Select the page that will not be needed for the longest time
- We do this by looking at the “forward distance” to the page
  - this is the distance from the current point in the reference stream to the next place in the stream where the same page is referenced
  - forward distance always > 0
  - forward distance == ∞ if the page is never referenced again
- In the case of ties, choose arbitrarily

Request			3	2	0	5	4	3	1	5	2	0
Frame	0	-	**3**	3	3	3	3	**3**	**1**	1	1	
	1	-		**2**	2	2	2	2	2	2	**2**	
	2	-			**0**	0	**4**	4	4	4	4	
	3	-				**5**	5	5	5	**5**	5	
Page faults			*	*	*	*	*		*			

The optimal page replacement algorithm - 9

Idea:

- Select the page that will not be needed for the longest time
- We do this by looking at the "forward distance" to the page
  - this is the distance from the current point in the reference stream to the next place in the stream where the same page is referenced
  - forward distance always > 0
  - forward distance == ∞ if the page is never referenced again
- In the case of ties, choose arbitrarily

Request			3	2	0	5	4	3	1	5	2	0
Frame	0	-	**3**	3	3	3	3	**3**	**1**	1	1	?
	1	-		**2**	2	2	2	2	2	2	**2**	?
	2	-			**0**	0	**4**	4	4	4	4	?
	3	-				**5**	5	5	5	**5**	5	?
Page faults			*	*	*	*	*		*			*

Optimal page replacement – example 2

Input:

- R = 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5, 6, 7
- 3 frames, all start empty

Try and work through this one yourself.

Request			0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
Frame	0	-																
	1	-																
	2	-																
Page faults																		

Optimal page replacement – example 2

Input:

- R = 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5, 6, 7
- 3 frames, all start empty

Try and work through this one yourself.

Request		0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
Frame 0	-	0	0	0	0	0	0	0	0	0	1	1	1	4	4	4	7
1	-		1	1	1	1	1	2	2	2	2	2	2	2	5	5	5
2	-			2	3	3	3	3	3	3	3	3	3	3	3	6	6
Page faults		*	*	*	*			*			*			*	*	*	*

The optimal page replacement algorithm

Idea:

- Select the page that will not be needed for the longest time

Problem:

- Can't know the future of a program
- Can't know when a given page will be needed next
- The optimal algorithm is unrealizable

However:

- We can use it only as a control case for simulation studies
  - Run the program once
  - Generate a log of all memory references
  - Use the log to simulate various page replacement algorithms
  - Can compare other algorithms to this "optimal" algorithm
    - for example, we know that the last example included 10 page faults under the optimal algorithm

In-class Exercise: Optimal Algorithm

- Assume that a memory system uses the optimal algorithm.
- The system has 4 page frames, and they are all empty.
- Given the following reference stream, calculate the number of page faults that occur.

R = 1 3 9 2 4 3 9 1 8 5 4 5 4 3 9 1 8 2 3 9 8 1 2 3

- I am not going to work through the answer, but I will tell you what it is.
  - the answer is: 11 page faults

FIFO page replacement algorithm

- Always replace the oldest page …
  - *"Replace the page that has been in memory for the longest time."*
  - We used this technique in the earlier explanation of reference streams
- It is *not* a good algorithm, as it makes no assumptions about when a page was used or is about to be used
- Consider the reference streams from the previous example ...

Request			0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
Frame	0	-	**0**	0	0	**3**	3	3	**2**	2	2	**1**	1	1	**4**	4	4	**7**
	1	-		**1**	1	1	**0**	0	0	**3**	3	3	**2**	2	2	**5**	5	5
	2	-			**2**	2	2	**1**	1	1	**0**	0	0	**3**	3	3	**6**	6
Page faults			*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*

FIFO page replacement algorithm

- This time we had 16 page faults (compared to 10 with optimal algorithm)
- The problem is that the replaced page may be needed again soon

Request			0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
Frame	0	-	**0**	0	0	**3**	3	3	**2**	2	2	**1**	1	1	**4**	4	4	**7**
	1	-		**1**	1	1	**0**	0	0	**3**	3	3	**2**	2	2	**5**	5	5
	2	-			**2**	2	2	**1**	1	1	**0**	0	0	**3**	3	3	**6**	6
Page faults			*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*

FIFO page replacement algorithm

- And with the other reference stream used earlier ...
- This stream gives 9 pages faults, while optimal is 6.

Request			3	2	0	5	4	3	1	5	2	0
Frame	0	-	**3**	3	3	3	**4**	4	4	4	4	**0**
	1	-		**2**	2	2	2	**3**	3	3	3	3
	2	-			**0**	0	0	0	**1**	1	1	1
	3	-				**5**	5	5	5	**5**	**2**	2
Page faults			*	*	*	*	*	*	*		*	*

In-class Exercise: FIFO Algorithm

- Assume that a memory system uses the FIFO algorithm.
- The system has 4 page frames, and they are all empty.
- Given the following reference stream, calculate the number of page faults that occur.

R = 1 3 9 2 4 3 9 1 8 5 4 5 4 3 9 1 8 2 3 9 8 1 2 3

  - the answer is: 16 page faults

Least recently used algorithm (LRU)

- Keep track of when a page is used
- Replace the page that has been used least recently
  - ie: replace the page that hasn't been referenced in the longest time
  - essentially, we use the "largest backward distance" to select pages to replace

- This algorithm should do better, as it is exploiting "locality of reference"
  ... let's see how it does ...

Request			0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
Frame	0	-																
	1	-																
	2	-																
Page faults																		

Least recently used algorithm (LRU)

- 16 page faults again!
- This sucks ... but in this case we simply have a ‘degenerative case’
- In other cases this algorithm has proven to do better than FIFO

Let’s try the other reference stream ...

Request			0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
Frame	0	-	**0**	0	0	**3**	3	3	**2**	2	2	**1**	1	1	**4**	4	4	**7**
	1	-		**1**	1	1	**0**	0	0	**3**	3	3	**2**	2	2	**5**	5	5
	2	-			**2**	2	2	**1**	1	1	**0**	0	0	**3**	3	3	**6**	6
Page faults			*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*

LRU page replacement algorithm

- Again we get 9 pages faults, while optimal is 6 – same as FIFO!
- This sucks as well.
- But it is a better algorithm than FIFO. Have a browse through 3.4.6 in your text if you want some more information...

Request			3	2	0	5	4	3	1	5	2	0
Frame	0	-	**3**	3	3	3	**4**	4	4	4	**2**	2
	1	-		**2**	2	2	2	**3**	3	3	3	**0**
	2	-			**0**	0	0	0	**1**	1	1	1
	3	-				**5**	5	5	5	**5**	5	5
Page faults			*	*	*	*	*	*	*		*	*

In-class Exercise: LRU Algorithm

- Assume that a memory system uses the LRU algorithm.
- The system has 4 page frames, and they are all empty.
- Given the following reference stream, calculate the number of page faults that occur.

R = 1 3 9 2 4 3 9 1 8 5 4 5 4 3 9 1 8 2 3 9 8 1 2 3

  - the answer is: 19 page faults

Not *frequently* used algorithm (NFU)

- Associate a counter with each page
  - Every time the page is referenced we increment the counter
  - The counter approximates how often the page is used
  - For replacement, choose the page with lowest counter
    - in the case of ties, select an arbitrary page to replace

now we have a tie – all have been used once so we will choose randomly ...

Request			0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
Frame	0	-	**0**	0	0													
	1	-		**1**	1													
	2	-			**2**													
Page faults			*	*	*													

Not frequently used algorithm (NFU)

- So we get 12 faults in total ... better than FIFO ... but only because we made a good arbitrary pick!

- It is interesting to notice what happens at the end, when a sequence of previously unused frames are loaded ...
  ... in this case the least frequently used frame is the last one loaded

Request			0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
Frame	0	-	**0**	0	0	0	**0**	0	0	0	**0**	0	0	**3**	3	3	3	3
	1	-		**1**	1	1	1	**1**	1	**3**	3	**1**	1	1	1	1	1	1
	2	-			**2**	**3**	3	3	**2**	2	2	2	**2**	2	**4**	**5**	**6**	**7**
Page faults			*	*	*	*			*	*		*		*	*	*	*	*

Not frequently used algorithm (NFU)

- With our other reference stream we get 8.
- Better than LRU, but not as good as optimal (6).

Request			3	2	0	5	4	3	1	5	2	0
Frame	0	-	**3**	3	3	3	3	**3**	3	3	3	3
	1	-		**2**	2	2	2	2	2	2	**2**	2
	2	-			**0**	0	0	0	**1**	1	1	**0**
	3	-				**5**	**4**	4	4	**5**	5	5
Page faults			*	*	*	*	*		*	*		*

In-class Exercise: NFU Algorithm

- Assume that a memory system uses the NFU algorithm.
- The system has 4 page frames, and they are all empty.
- Given the following reference stream, calculate the number of page faults that occur.

R = 1 3 9 2 4 3 9 1 8 5 4 5 4 3 9 1 8 2 3 9 8 1 2 3

  – the answer is: 14 page faults (depending on how you break ties)

Not frequently used algorithm (NFU)

Problem with NFU:

- Some page may be heavily used
  - ---> Its counter is large
- The program's behavior changes
  - Now, this page is not used ever again (or only rarely), but it stays in memory
- This algorithm never forgets!
  - *This page will never be chosen for replacement!*

Modified NFU with aging

- Associate a counter with each page like before
- Also keep a “time window” – so that we only consider the frequency of use within some previous period of time
- One way to do this is:
  - On every clock tick, the OS looks at each page.
    - Shift the counter right 1 bit (divide its value by 2)
    - If the Page Reference Bit is set...
      - Set the most-significant bit
      - Clear the Referenced Bit

```
100000 = 32
010000 = 16
001000 = 8
000100 = 4
100010 = 34
110001 = 49
111000 = 56
011100 = 28
```

Dynamic Paging Algorithms

Paging Algorithms and Policies (review)

Any paging algorithm must define:

- fetch policy
  - decides when a page should be loaded into memory
- replacement policy
  - decides which page should be moved to disk when all pages are full
- placement policy
  - decides where the page should be loaded in primary memory

There are two types of paging algorithms:

- static paging algorithms
  - each process uses a fixed number of frames during its lifetime
- dynamic paging algorithms
  - the number of frames allocated to a process changes during its lifetime

Static Paging Algorithms (review)

- number of page frames does not change throughout lifetime of the process
  - ie: each process is allocated a number of page frames, even if it doesn't need them, or if it leads to thrashing
- "thrashing" is the situation where there are not enough page frames allocated to the process, and it continually keeps swapping pages
  - this can occur when a loop crosses a page boundary, say from page n to n+1, but there is only one frame available
    - depending on replacement policy, n and n+1 could continue to displace each other as the program loops
    - this is very slow, as we have to write pages to disk on each iteration of the loop
- in this case it makes more sense to allocate an extra frame to the process for the times that it needs the extra page

Dynamic Paging Algorithms

- dynamic paging algorithms adjust the number of frames used by a process according to its need for these frames
- these algorithms specify both the *placement* and the *replacement* policies, ie:
  - they specify where the page can be loaded
  - they specify which page is to be replaced if there is a choice

- the motivation for dynamic paging algorithms is the concept of "locality of reference"
  - this is the notion that programs tend stay in specific memory locales within the process
  - for example, the main body of a program name be a loop that calls a few methods from within a loop.
    - in this case, the pages that contain the loop code and the most commonly accessed methods will be accessed much more frequently
    - this is know as "locality of reference"

Working Set

- contemporary dynamic paging algorithms are based on the idea of a “working set”
- the set of pages that a program is currently using is called the “working set”
  - if we are able to load the entire working set into memory, this process will not have any faults
  - if we are not able to load the working set, the program will thrash
- it is known that working sets tend to change fairly slowly over time, there fore we should be able to develop an algorithm that monitors the working set, and:
  - releases frames that are no longer needed (so other processes can use them)
  - acquires additional frames when the number of pages in the working set increases

Window Size

- for working set algorithms, we add a parameter ω that defines a "window size"
- this window size defines the number of most recent page references that are used to define the working set, for example:
  - assume $\omega = 4$

t_i

Request			3	2	0	5	5	3	2	1	3	1	3	2	4	5	6	5
Frame	0	-	3	3	3	3												
	1	-		2	2	2												
	2	-			0	0												
	3	-				5												
Page faults			*	*	*	*												

Working Set - 1

- in working set algorithms, new pages are added into frames that are not currently in the working set
  - is a page is replaced because it is not in the working set, it still must be written to disk
- continuing for example with $\omega = 4$

the window size is 4; it moves as t_i increases

the working set at this time is { 3 2 0 5 }

t_i

Request			3	2	0	5	5	3	2	1	3	1	3	2	4	5	6	5
Frame	0	-	3	3	3	3												
	1	-		2	2	2												
	2	-			0	0												
	3	-				5												
Page faults			*	*	*	*												

Working Set - 2

now frame 0 is available to other processes

the window size is 4; it moves as t_i increases

the working set at this time is { 2 0 5 }

t_i

Request			3	2	0	5	5	3	2	1	3	1	3	2	4	5	6	5
Frame	0	-	3	3	3	3												
	1	-		2	2	2	2											
	2	-			0	0	0											
	3	-				5	5											
Page faults			*	*	*	*												

Working Set - 3

page fault occurs when page 3 joins the working set

frame 1 leaves the working set when page 2 leaves the window

the working set is now { 3 0 5 }

the window size is 4; it moves as t_i increases

t_i

Request			**3**	**2**	**0**	**5**	**5**	**3**	**2**	**1**	**3**	**1**	**3**	**2**	**4**	**5**	**6**	**5**
Frame	**0**	-	**3**	3	3	3		**3**										
	1	-		**2**	2	2	2											
	2	-			**0**	0	0	0										
	3	-				**5**	**5**	5										
Page faults			*	*	*	*		*										

Working Set - 4

page fault occurs when page 2 joins the working set

frame 2 leaves the working set when page 0 leaves the window

the working set at this time is { 3 2 5 }

the window size is 4; it moves as t_i increases

t_i

Request			3	2	0	5	5	3	2	1	3	1	3	2	4	5	6	5
Frame	0	-	**3**	3	3	3		**3**	3									
	1	-		**2**	2	2	2		**2**									
	2	-			**0**	0	0	0										
	3	-				**5**	**5**	5	5									
Page faults			*	*	*	*		*	*									

Working Set - 5

- continuing the example ...

the working set at this time is { 5 3 2 1 }

t_i

Request			3	2	0	5	5	3	2	1	3	1	3	2	4	5	6	5
Frame	0	-	**3**	3	3	3		**3**	3	3								
	1	-		**2**	2	2	2		**2**	2								
	2	-			**0**	0	0	0		**1**								
	3	-				**5**	**5**	5	5	5								
Page faults			*	*	*	*		*	*	*								

Working Set - 6

- continuing the example ...

the working set at this time is { 3 2 1 }

t_i

Request			3	2	0	5	5	3	2	1	3	1	3	2	4	5	6	5
Frame	0	-	**3**	3	3	3		**3**	3	3	**3**							
	1	-		**2**	2	2	2		**2**	2	2							
	2	-			**0**	0	0	0		**1**	1							
	3	-				**5**	**5**	5	5	5								
Page faults			*	*	*	*		*	*	*								

Working Set - 7

- continuing the example ...

the working set at this time is { 3 2 1 }

t_i

Request			3	2	0	5	5	3	2	1	3	1	3	2	4	5	6	5
Frame	0	-	3	3	3	3		3	3	3	3	3						
	1	-		2	2	2	2		2	2	2	2						
	2	-			0	0	0	0		1	1	1						
	3	-				5	5	5	5	5								
Page faults			*	*	*	*		*	*	*								

Working Set - 8

- continuing the example ...

the working set at this time is { 3 1 }

t_i

			3	2	0	5	5	3	2	1	3	1	3	2	4	5	6	5
Request			3	2	0	5	5	3	2	1	3	1	3	2	4	5	6	5
Frame	0	-	**3**	3	3	3		**3**	3	3	**3**	3	**3**					
	1	-		**2**	2	2	2		**2**	2	2	2						
	2	-			**0**	0	0	0		**1**	1	**1**	1					
	3	-				**5**	**5**	5	5	5								
Page faults			*	*	*	*		*	*	*								

Working Set - 9

- continuing the example ...

the working set at this time is { 3 1 2 }

t_i

Request			3	2	0	5	5	3	2	1	3	1	3	2	4	5	6	5
Frame	**0**	-	**3**	3	3	3		**3**	3	3	**3**	3	**3**	3				
	1	-		**2**	2	2	2		**2**	2	2	2		**2**				
	2	-			**0**	0	0	0		**1**	1	**1**	1	1				
	3	-				**5**	**5**	5	5	5								
Page faults			*	*	*	*		*	*	*				*				

Working Set - 10

- continuing the example ...

the working set at this time is { 1 2 3 4 }

t_i

Request			3	2	0	5	5	3	2	1	3	1	3	2	4	5	6	5
Frame	0	-	3	3	3	3		3	3	3	3	3	3	3	3			
	1	-		2	2	2	2		2	2	2	2		2	2			
	2	-			0	0	0	0		1	1	1	1	1	1			
	3	-				5	5	5	5	5					4			
Page faults			*	*	*	*		*	*	*				*	*			

Working Set - 11

- continuing the example ...

page 5 needs a home, but
page 1 fell out of the window,
so we can put page 5 in frame 2

the working set at this
time is { 2 3 4 5 }

t_i

Request			3	2	0	5	5	3	2	1	3	1	3	2	4	5	6	5
Frame	0	-	3	3	3	3		3	3	3	3	3	3	3	3	3		
	1	-		2	2	2	2		2	2	2	2		2	2	2		
	2	-			0	0	0	0		1	1	1	1	1	1	5		
	3	-				5	5	5	5	5					4	4		
Page faults			*	*	*	*		*	*	*				*	*	*		

Working Set - 12

- continuing the example ...

same thing happens ...
page 3 leaves the set ...
... and page 6 joins

the working set at this time is { 2 4 5 6 }

t_i

Request			3	2	0	5	5	3	2	1	3	1	3	2	4	5	6	5
Frame	0	-	**3**	3	3	3		**3**	3	3	**3**	3	**3**	3	3	3	**6**	
	1	-		**2**	2	2	2		**2**	2	2	2		**2**	2	2	2	
	2	-			**0**	0	0	0		**1**	1	**1**	1	1	1	**5**	5	
	3	-				**5**	**5**	5	5	5					**4**	4	4	
Page faults			*	*	*	*		*	*	*				*	*	*	*	

Working Set - 13

- continuing the example ...

the working set at this time is { 4 5 6 }

t_i

Request			3	2	0	5	5	3	2	1	3	1	3	2	4	5	6	5
Frame	0	-	3	3	3	3		3	3	3	3	3	3	3	3	3	6	6
	1	-		2	2	2	2		2	2	2	2		2	2	2	2	
	2	-			0	0	0	0		1	1	1	1	1	1	5	5	5
	3	-				5	5	5	5	5					4	4	4	4
Page faults			*	*	*	*		*	*	*				*	*	*	*	

Locality of Reference (revisited)

- If we look at the working set we see it changes slowly ...

Set	Size
{ 3 }	1
{ 3 2 }	2
{ 3 2 0 }	3
{ 3 2 0 5 }	4
{ 2 0 5 }	3
{ 3 0 5 }	3
{ 3 2 5 }	3
{ 3 2 5 1 }	4
{ 3 2 1 }	3
{ 3 2 1 }	3
{ 3 1 }	2
{ 3 2 1 }	3
{ 3 2 1 4 }	4
{ 3 2 5 4 }	4
{ 6 2 5 4 }	4
{ 6 5 4 }	3

What happens with different window sizes?

- Lets try $\boldsymbol{\omega} = 2$
- same reference stream

t_i

Request			3	2	0	5	5	3	2	1	3	1	3	2	4	5	6	5
Frame	0	-	3	3														
	1	-		2														
Page faults			*	*														

Example 2 - $\omega = 2$ (1)

t_i

Request			3	2	0	5	5	3	2	1	3	1	3	2	4	5	6	5
Frame	0	-	**3**	3	**0**													
	1	-		**2**	2													
Page faults			*	*	*													

Example 2 - $\omega = 2$ (2)

t_i

Request			3	2	0	5	5	3	2	1	3	1	3	2	4	5	6	5
Frame	0	-	**3**	3	**0**	0												
	1	-		**2**	2	**5**												
Page faults			*	*	*	*												

Example 2 - $\omega = 2$ (3)

- we can see that the number of page faults increases
- What is the maximum number of page frames that this process could use?

7 -> one for each frame 0-6

t_i

Request			3	2	0	5	5	3	2	1	3	1	3	2	4	5	6	5
Frame	0	-	3	3	0	0		3	3	1	1	1	1	2	2	5	5	5
	1	-		2	2	5	5	5	2	2	3	3	3	3	4	4	6	6
Page faults			*	*	*	*		*	*	*	*			*	*	*	*	

Example 3 - $\boldsymbol{\omega} = 7$ (1)

- OK, so lets see what happens if we set the window size to 7

t_i

Request			3	2	0	5	5	3	2	1	3	1	3	2	4	5	6	5
Frame	0	-	3	3	3	3	3	3	3									
	1	-		2	2	2	2	2	2									
	2	-			0	0	0	0	0									
	3	-				5	5	5	5									
	4	-																
	5	-																
	6	-																
Page faults			*	*	*	*												

Example 3 - $\omega = 7$ (2)

- OK, so lets see what happens if we set the window size to 7

t_i

Request			3	2	0	5	5	3	2	1	3	1	3	2	4	5	6	5
Frame	0	-	3	3	3	3	3	3	3	3								
	1	-		2	2	2	2	2	2	2								
	2	-			0	0	0	0	0	0								
	3	-				5	5	5	5	5								
	4	-								1								
	5	-																
	6	-																
Page faults			*	*	*	*				*								

Example 3 - $\omega = 7$ (3)

- OK, so lets see what happens if we set the window size to 7

t_i

Request			3	2	0	5	5	3	2	1	3	1	3	2	4	5	6	5
Frame	0	-	**3**	3	3	3	3	**3**	3	3	**3**							
	1	-		**2**	2	2	2	2	**2**	2	2							
	2	-			**0**	0	0	0	0	0	0							
	3	-				**5**	**5**	5	5	5	5							
	4	-								**1**	1							
	5	-																
	6	-																
Page faults			*	*	*	*				*								

Example 3 - $\omega = 7$ (4)

- OK, so lets see what happens if we set the window size to 7

t_i

Request			3	2	0	5	5	3	2	1	3	1	3	2	4	5	6	5
Frame	0	-	**3**	3	3	3	3	**3**	3	3	**3**	3						
	1	-		**2**	2	2	2	2	**2**	2	2	2						
	2	-			**0**	0	0	0	0	0	0							
	3	-				**5**	**5**	5	5	5	5	5						
	4	-								**1**	1	**1**						
	5	-																
	6	-																
Page faults			*	*	*	*				*								

Example 3 - ω = 7 (5)

- OK, so lets see what happens if we set the window size to 7

t_i

Request			3	2	0	5	5	3	2	1	3	1	3	2	4	5	6	5
Frame	0	-	3	3	3	3	3	3	3	3	3	3	3					
	1	-		2	2	2	2	2	2	2	2	2	2					
	2	-			0	0	0	0	0	0	0							
	3	-				5	5	5	5	5	5	5	5					
	4	-								1	1	1	1					
	5	-																
	6	-																
Page faults			*	*	*	*				*								

Example 3 - $\omega = 7$ (6)

- we see that the number of page faults has decreased
- we also observe that we don't usually require 7 frames, as the locality of reference is not that big

t_i

Request			3	2	0	5	5	3	2	1	3	1	3	2	4	5	6	5
Frame	0	-	**3**	3	3	3	3	**3**	3	3	**3**	3	**3**	3	3	3	3	3
	1	-		**2**	2	2	2	2	**2**	2	2	2	2	**2**	2	2	2	2
	2	-			**0**	0	0	0	0	0	0				**4**	4	4	4
	3	-				**5**	**5**	5	5	5	5	5	5			**5**	5	**5**
	4	-								**1**	1	**1**	1	1	1	1	1	1
	5	-															**6**	6
	6	-																
Page faults			*	*	*	*				*					*	*	*	

Clock Algorithms

- so far we have looked at working set algorithms from the perspective of a single process
- to make this type of algorithm useful, we need a way to implement it for many processes
- first some background and assumptions ...

1. we assume that page faults are not that frequent,
   - ie, there will be many page references in many processes between page swaps
2. we assume that whenever a page is referenced, a reference bit R is set
3. we assume that each process keeps a current virtual time
   - this is the amount of CPU time the process has actually used since creation
4. we assume that each page stores a "time of last reference"
   - this is the time that the page was last accessed

Simple Clock Algorithm (1)

- we define a circular linked list
- as pages are loaded, they are added to the list
- every time a page is referenced its R bit is set to 1
- when a page is loaded, its R-bit is set to 1 and all other frames R-bits are cleared (set to zero)

Frame	R-bit	Process
10	1	P3

This tells us that the page in frame 10 was accessed by Process 3

Simple Clock Algorithm (2)

- we define a circular linked list
- as pages are loaded, they are added to the list
- every time a page is referenced its R bit is set to 1
- when a page is loaded, its R-bit is set to 1 and all other frames R-bits are cleared (set to zero)

Frame	R-bit	Process
10	0	P3
4	1	P7

Simple Clock Algorithm (3)

- we define a circular linked list
- as pages are loaded, they are added to the list
- every time a page is referenced its R bit is set to 1
- when a page is loaded, its R-bit is set to 1 and all other frames R-bits are cleared (set to zero)

Frame	R-bit	Process
10	0	P3
4	0	P7
53	1	P9

Simple Clock Algorithm (4)

- we define a circular linked list
- as pages are loaded, they are added to the list
- every time a page is referenced its R bit is set to 1
- when a page is loaded, its R-bit is set to 1 and all other frames R-bits are cleared (set to zero)

Frame	R-bit	Process
10	0	P3
4	0	P7
53	0	P9
9	1	P3

Simple Clock Algorithm (5)

- we define a circular linked list
- as pages are loaded, they are added to the list
- every time a page is referenced its R bit is set to 1
- when a page is loaded, its R-bit is set to 1 and all other frames R-bits are cleared (set to zero)

Frame	R-bit	Process
10	0	P3
4	0	P7
53	0	P9
9	0	P3
34	1	P2

Simple Clock Algorithm (6)

- we define a circular linked list
- as pages are loaded, they are added to the list
- every time a page is referenced its R bit is set to 1
- when a page is loaded, its R-bit is set to 1 and all other frames R-bits are cleared (set to zero)

Frame	R-bit	Process
10	0	P3
4	0	P7
53	0	P9
9	0	P3
34	0	P2
19	1	P4

Simple Clock Algorithm (7)

- we define a circular linked list
- as pages are loaded, they are added to the list
- every time a page is referenced its R bit is set to 1
- when a page is loaded, its R-bit is set to 1 and all other frames R-bits are cleared (set to zero)

Frame	R-bit	Process
10	0	P3
4	0	P7
53	0	P9
9	0	P3
34	0	P2
19	0	P4
48	1	P4

Simple Clock Algorithm (8)

- we define a circular linked list
- as pages are loaded, they are added to the list
- every time a page is referenced its R bit is set to 1
- when a page is loaded, its R-bit is set to 1 and all other frames R-bits are cleared (set to zero)

clock hand

Frame	R-bit	Process
10	0	P3
4	0	P7
53	0	P9
9	0	P3
34	0	P2
19	0	P4
48	0	P4
29	1	P3

- we also define a "clock hand" that points to pages in the circular list
- this hand will be moved by the algorithm as it looks for pages to replace

Simple Clock Algorithm (9)

- assume that no page faults happen for a while
- processes get scheduled and use pages they have loaded, setting R-bits ....

Frame	R-bit	Process
10	0	P3
4	0	P7
53	0	P9
9	0	P3
34	0	P2
19	0	P4
48	0	P4
29	1	P3

clock hand

Simple Clock Algorithm (10)

- assume that no page faults happen for a while
- processes get scheduled and use pages they have loaded, setting R-bits ....

clock hand

Frame	R-bit	Process
10	0	P3
4	0	P7
53	1	P9
9	0	P3
34	0	P2
19	0	P4
48	0	P4
29	1	P3

Simple Clock Algorithm (11)

- assume that no page faults happen for a while
- processes get scheduled and use pages they have loaded, setting R-bits ....

clock hand

Frame	R-bit	Process
10	0	P3
4	1	P7
53	1	P9
9	0	P3
34	0	P2
19	0	P4
48	0	P4
29	1	P3

Simple Clock Algorithm (12)

- assume that no page faults happen for a while
- processes get scheduled and use pages they have loaded, setting R-bits ....

clock hand

Frame	R-bit	Process
10	0	P3
4	1	P7
53	1	P9
9	0	P3
34	0	P2
19	0	P4
48	1	P4
29	1	P3

Simple Clock Algorithm (13)

- assume that no page faults happen for a while
- processes get scheduled and use pages they have loaded, setting R-bits ....

clock hand

Frame	R-bit	Process
10	0	P3
4	1	P7
53	1	P9
9	1	P3
34	0	P2
19	0	P4
48	1	P4
29	1	P3

Simple Clock Algorithm (14)

- at this point let us assume that process P3 causes a page fault
- the "clock hand" is pointing to frame 4
- the algorithm examines pages, starting with the frame pointed to by the clock hand
  - if the R-bit is set, the page must be in the current working set
    - the algorithm clears it and moves to the next frame
  - if the R-bit is not set, the page is not in the working set and can be replaced
    - the page is moved to disk and the frame is loaded with the requested page
  - if all R-bits are set (ie: we go all the way around without finding a page that is not in the working set)
    - we select a victim at random

Frame	R-bit	Process
10	0	P3
4	1	P7
53	1	P9
9	1	P3
34	0	P2
19	0	P4
48	1	P4
29	1	P3

clock hand

Simple Clock Algorithm (15)

- Frame 4 is in the working set ...
  ... clear R-bit and try the next one ...

clock hand

Frame	R-bit	Process
10	0	P3
4	1	P7
53	1	P9
9	1	P3
34	0	P2
19	0	P4
48	1	P4
29	1	P3

Simple Clock Algorithm (16)

- Frame 53 is in the working set ...

  ... clear R-bit and try the next one ...

clock hand

Frame	R-bit	Process
10	0	P3
4	0	P7
53	1	P9
9	1	P3
34	0	P2
19	0	P4
48	1	P4
29	1	P3

Simple Clock Algorithm (17)

- Frame 9 is in the working set ...

  ... clear R-bit and try the next one ...

clock hand

Frame	R-bit	Process
10	0	P3
4	0	P7
53	0	P9
9	1	P3
34	0	P2
19	0	P4
48	1	P4
29	1	P3

Simple Clock Algorithm (18)

- Frame 34 is NOT in the working set
- frame 34 is selected for replacement
- the page that is loaded in frame 34 is written to disk (if dirty bit is set)

clock hand

Frame	R-bit	Process
10	0	P3
4	0	P7
53	0	P9
9	0	P3
34	0	P2
19	0	P4
48	1	P4
29	1	P3

Simple Clock Algorithm (19)

- Frame 34 is NOT in the working set
- frame 34 is selected for replacement
- the page that is loaded in frame 34 is written to disk (if dirty bit is set)
- the new page is loaded

clock hand

Frame	R-bit	Process
10	0	P3
4	0	P7
53	0	P9
9	0	P3
34	*1*	*P3*
19	0	P4
48	1	P4
29	1	P3

Simple Clock Algorithm (20)

- Frame 34 is NOT in the working set
- frame 34 is selected for replacement
- the page that is loaded in frame 34 is written to disk (if dirty bit is set)
- the new page is loaded
- all R-bits are cleared (except for the newly loaded page), and execution continues

clock hand

Frame	R-bit	Process
10	0	P3
4	0	P7
53	0	P9
9	0	P3
34	1	P3
19	0	P4
48	0	P4
29	0	P3

WSClock Algorithm

- This is the algorithm that is used for page replacement in most dynamic paging systems.
- It is basically the same as the clock algorithm presented previously, except:

1. uses the "time of last reference" to make sure that there is a window ω to keep pages in the working set during times of frequent paging activity
2. to define the window ω we work with a time interval (as opposed to counting page references)
3. when the algorithm identifies a page with R-bit == 1, it is in the working set, so we update "time of last reference", clear the R-bit, and move on
4. when R-bit == 0, the algorithm compares last use to see if it is in the window, ie:

   ```
   if  ( current_virtual_time – time_of_last_reference ) > ω
   ```

   ... then replace the page

WSClock Example (from your textbook)

- Assume 3 processes, with current virtual times (cvt) as follows:
  - P0: cvt = 55
  - P1: cvt = 75
  - P2: cvt = 80
- Assume ω = 25
- We will show each page frame entry like this:

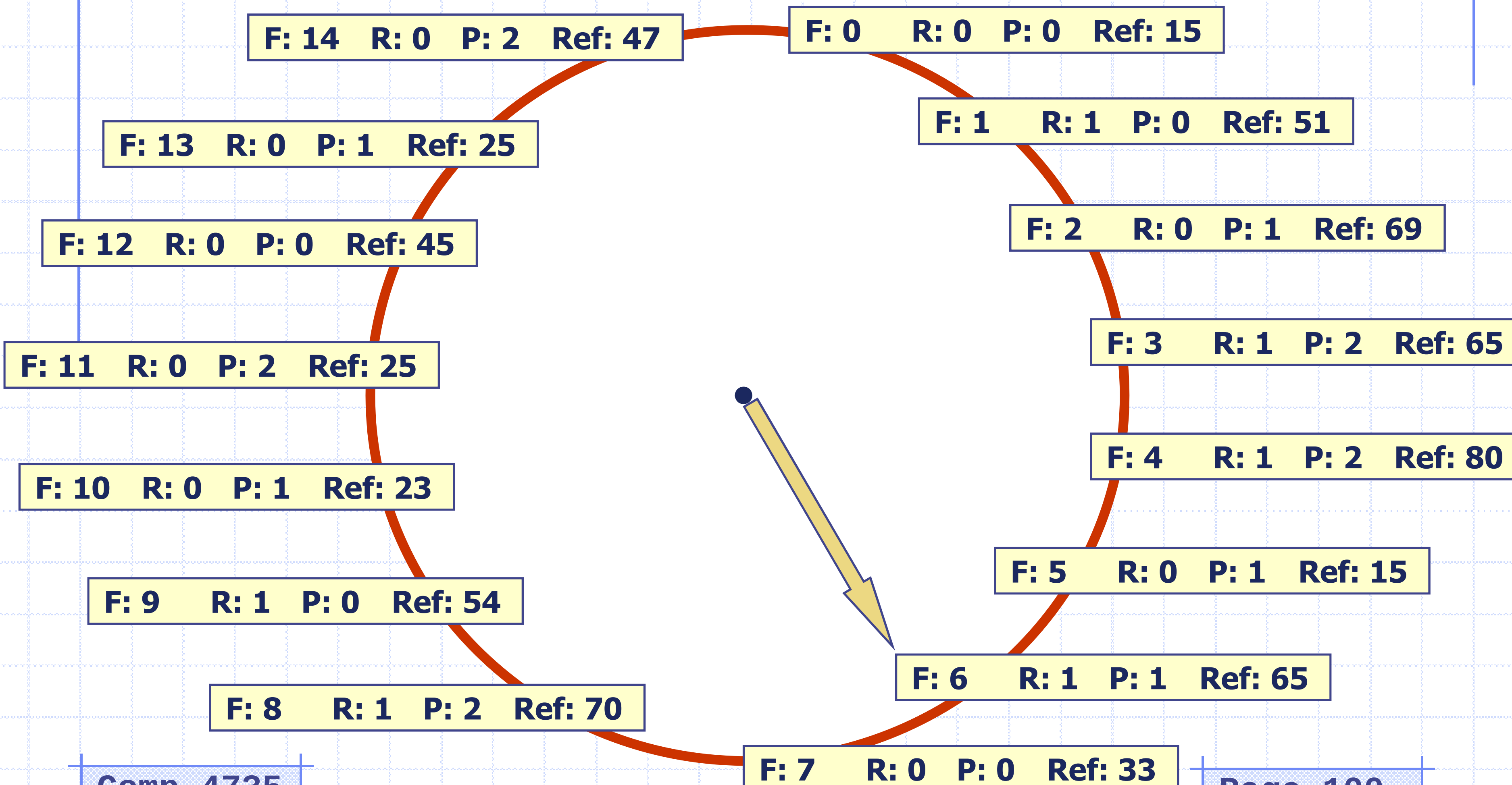
Frame	R-bit	Process	LastRef
F: 10	**R: 0**	**P: 3**	**Ref: 15**

$\omega = 25$

P0: cvt = 55
P1: cvt = 75
P2: cvt = 80

WSClock Example (2)

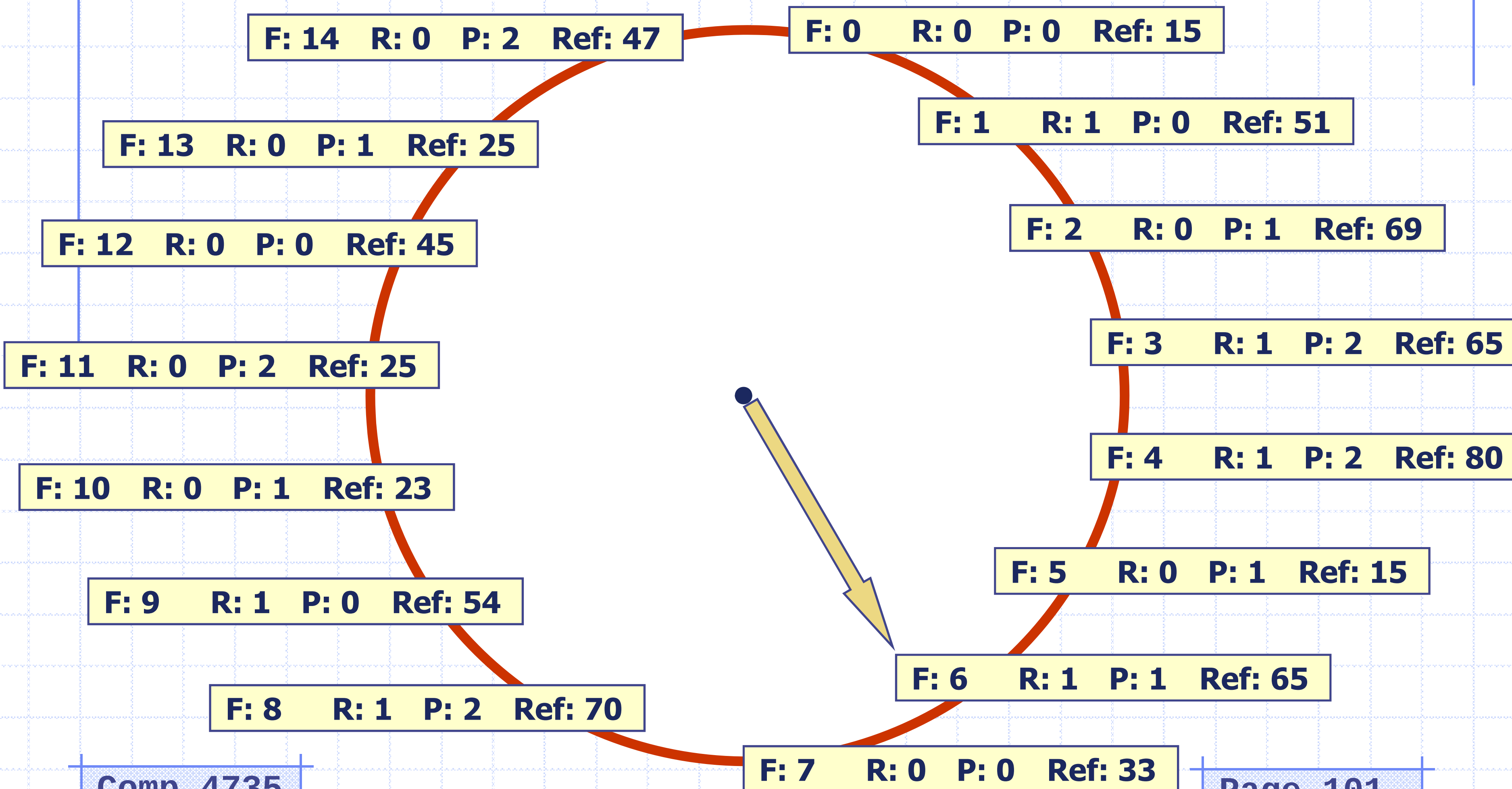
Assume a page fault has just occurred in P0.



WSClock - check R-bit for frame 6
- page is in working set (because R=1) ...
... so set R= 0 and set ref=cvt
... advance to next frame

$\omega = 25$

P0: cvt = 55
P1: cvt = 75
P2: cvt = 80



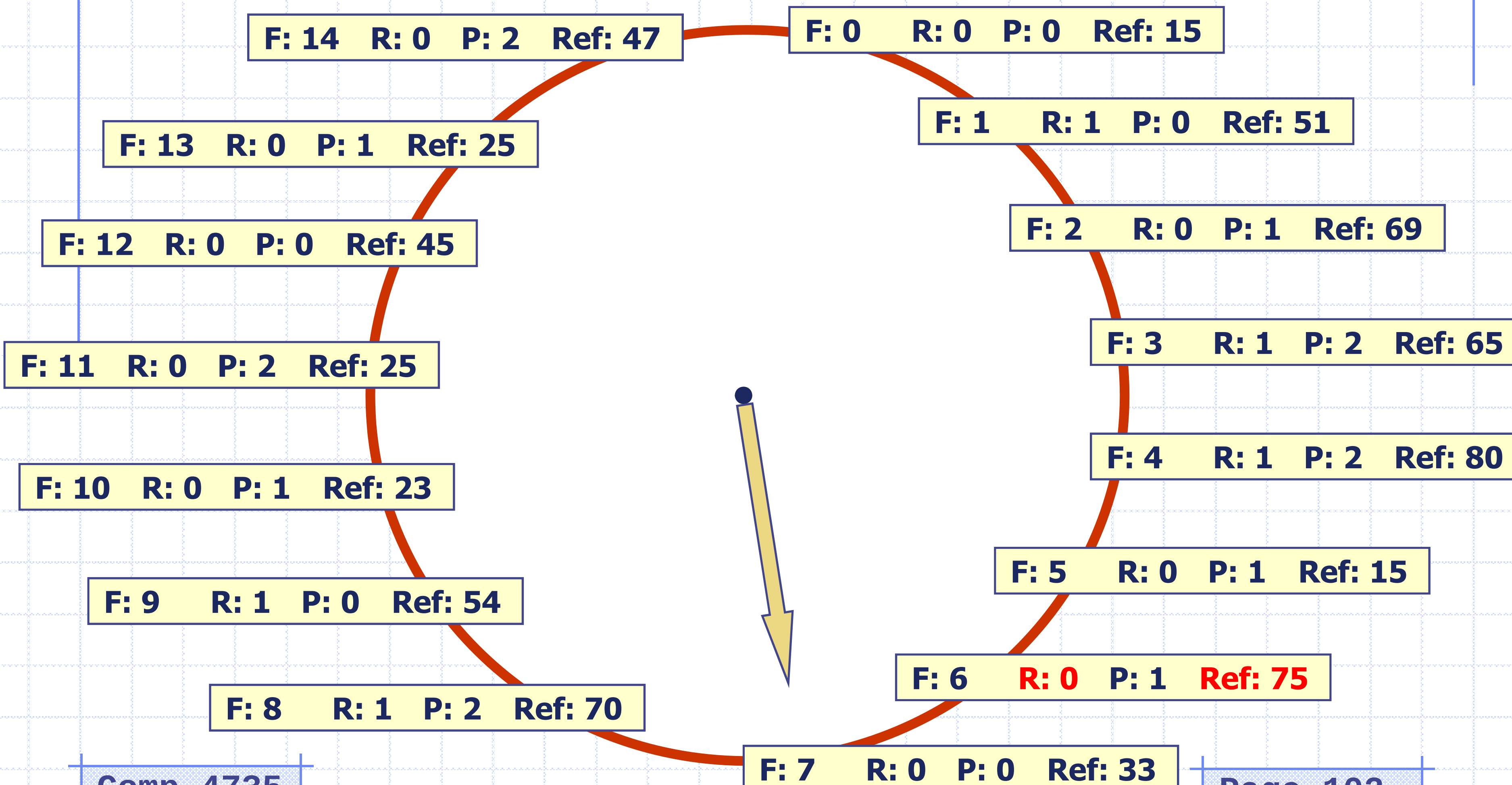
WSClock - check R-bit for frame 7

- R = 0, so we test (cvt_0 - ref_0 < ω)
- (55 – 33) < 25 … 22 < 25

→ therefore F7 is still in P0's working set
… and we advance to next frame

ω = 25

P0: cvt = 55
P1: cvt = 75
P2: cvt = 80



WSClock - check R-bit for frame 8

- R-bit is set ...

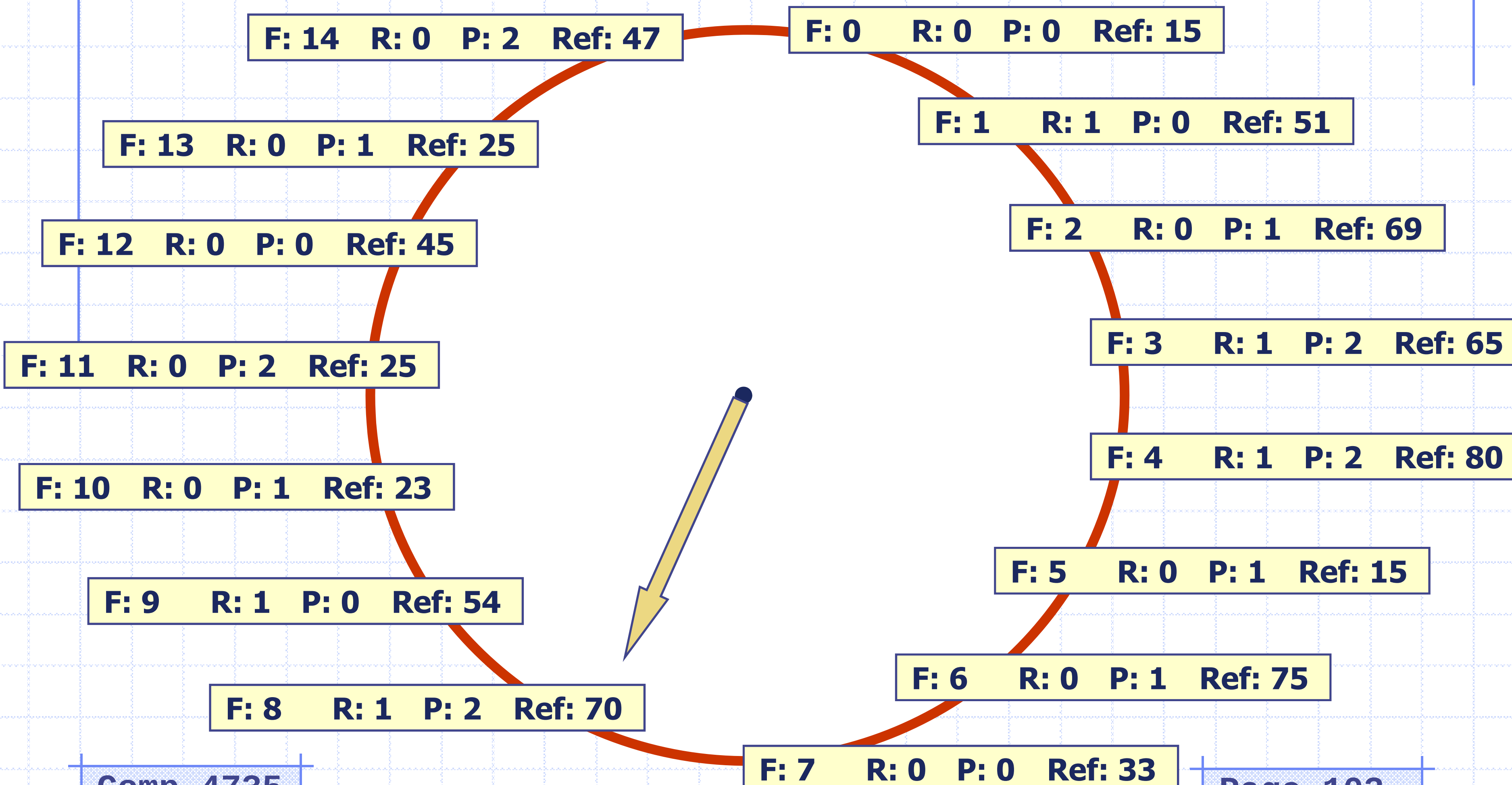
... move to next frame

$\omega = 25$

P0: cvt = 55

P1: cvt = 75

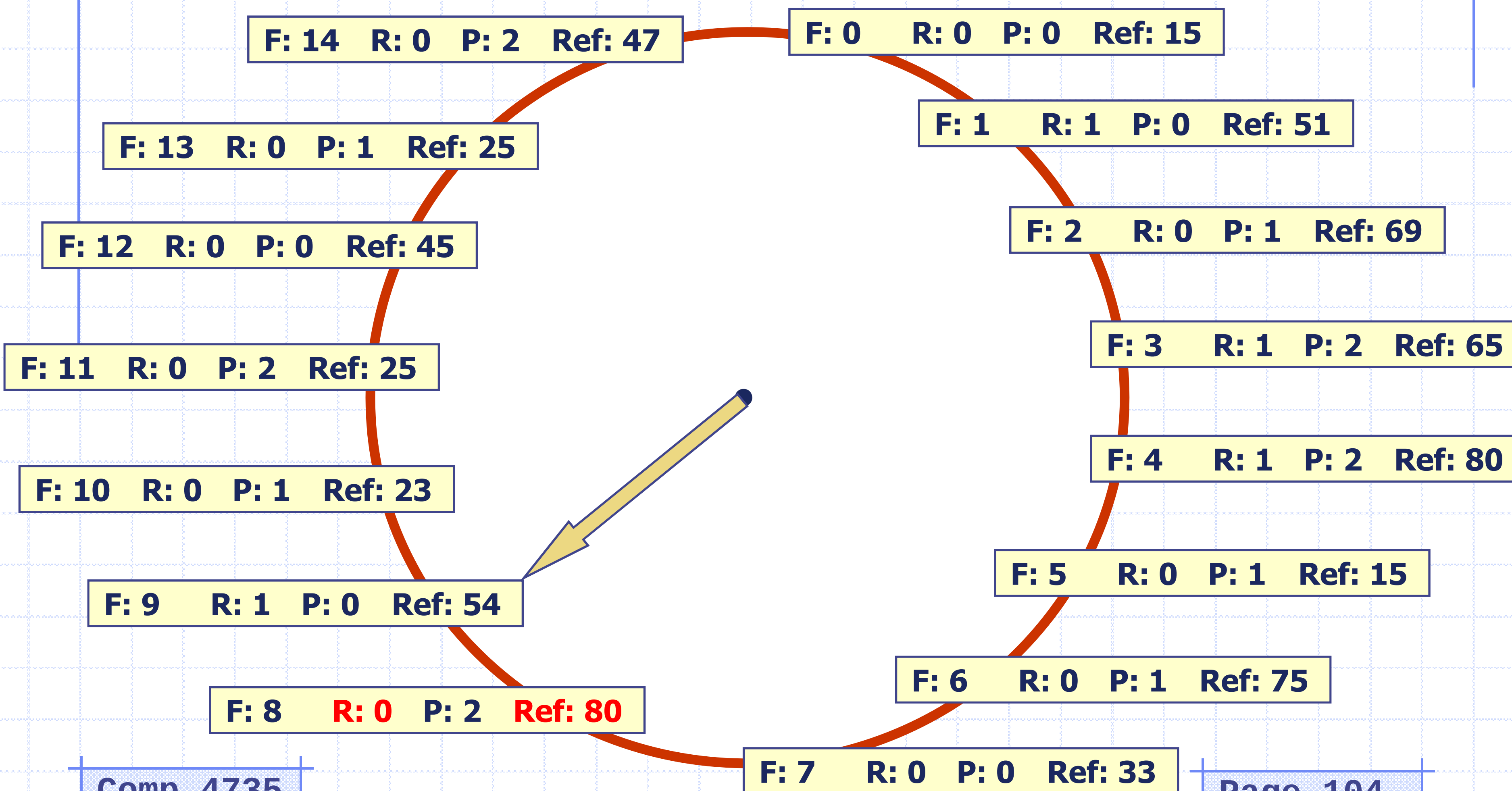
P2: cvt = 80



WSClock - check R-bit for frame 9
- R-bit is set ...
... move to next frame

ω = 25

P0: cvt = 55
P1: cvt = 75
P2: cvt = 80



WSClock - check R-bit for frame 10

- R = 0 ... test: $(cvt_1 - ref_1 < \omega)$ ...

... (75 – 23) < 25 ... (52 < 25)

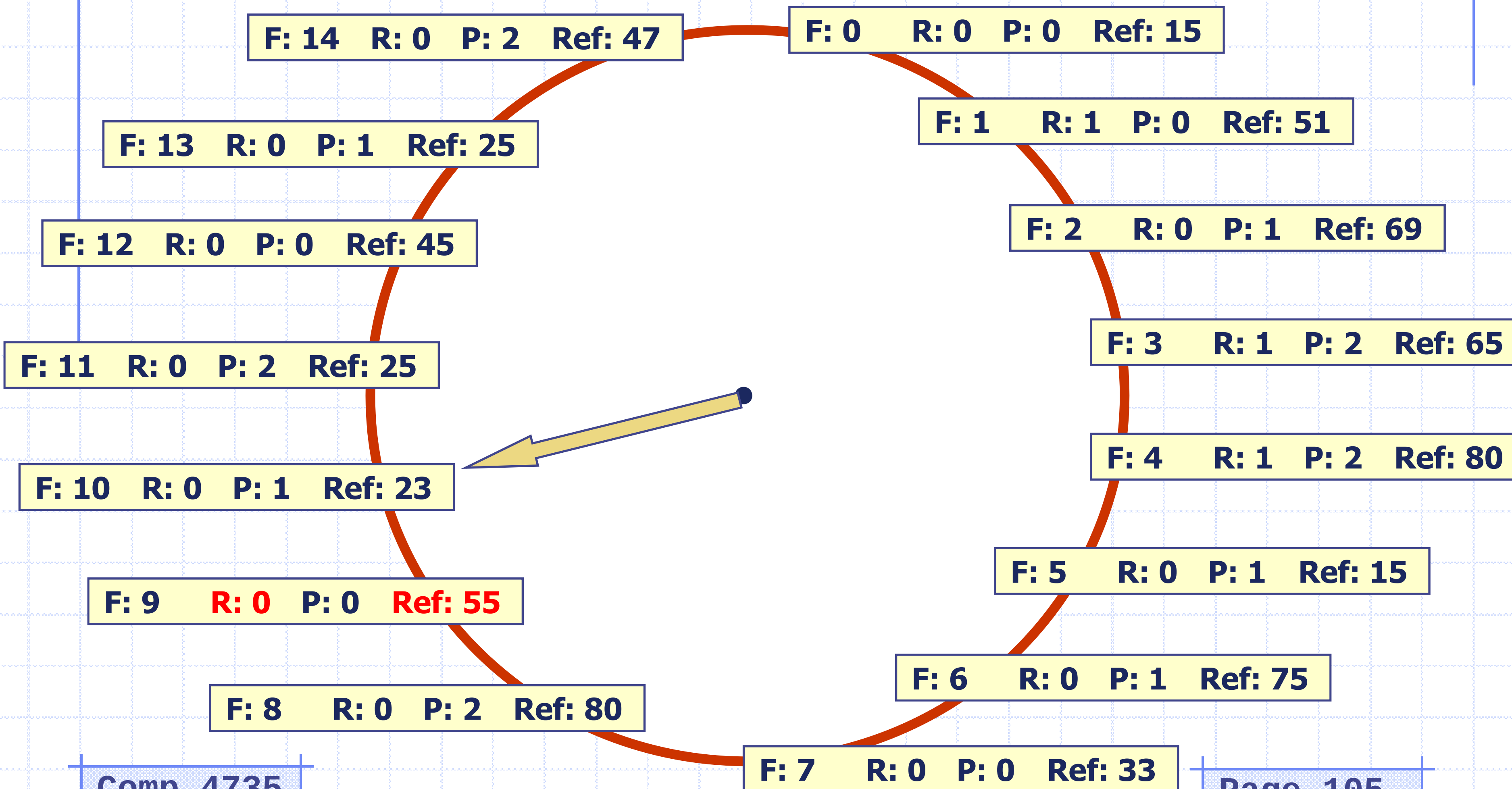
→ therefore the page is no longer in P1 W Set

$\omega = 25$

P0: cvt = 55

P1: cvt = 75

P2: cvt = 80

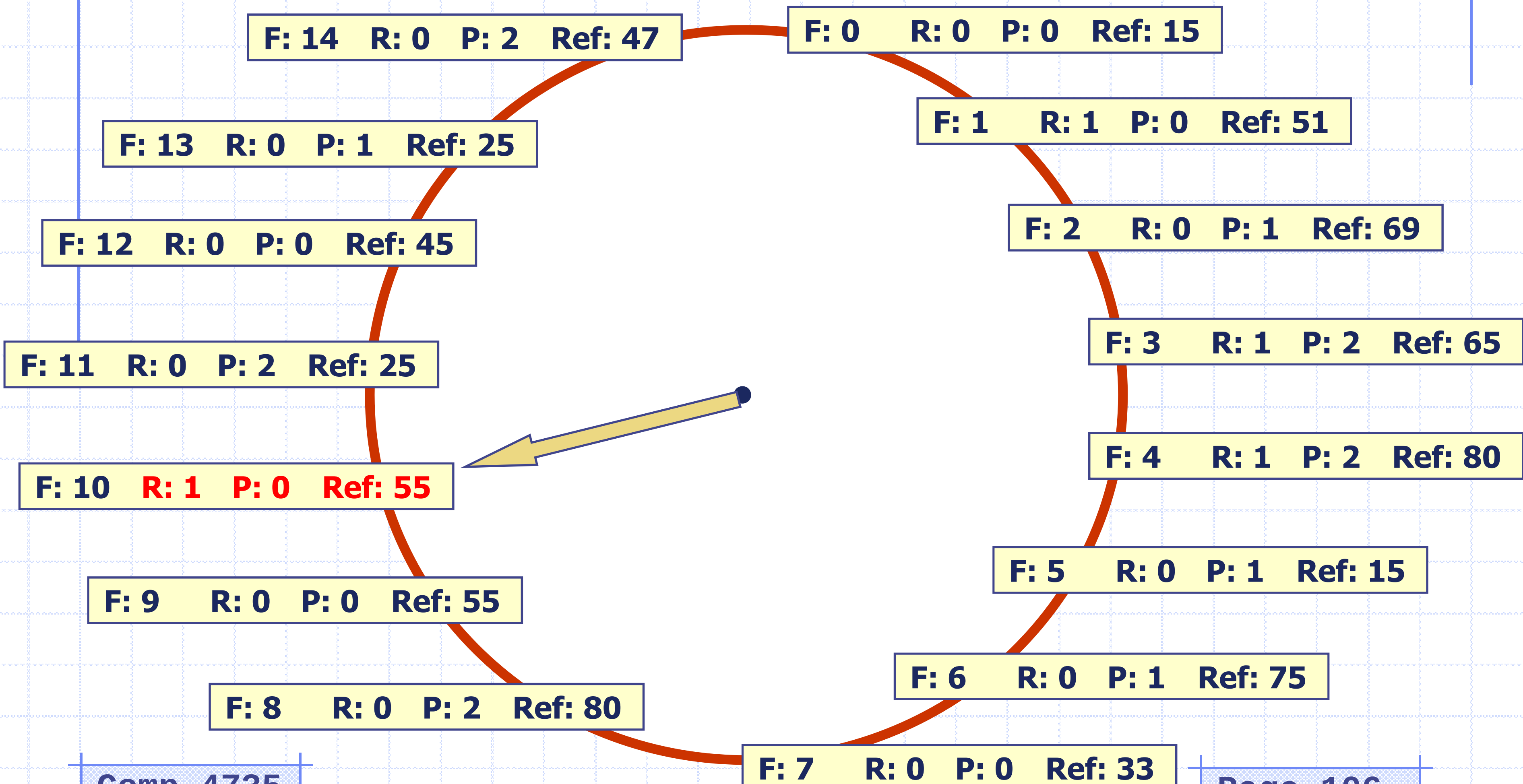


WSClock *the new page is loaded in frame 10*

- this frame is now allocated to Process 0

ω = 25

P0: cvt = 55
P1: cvt = 75
P2: cvt = 80



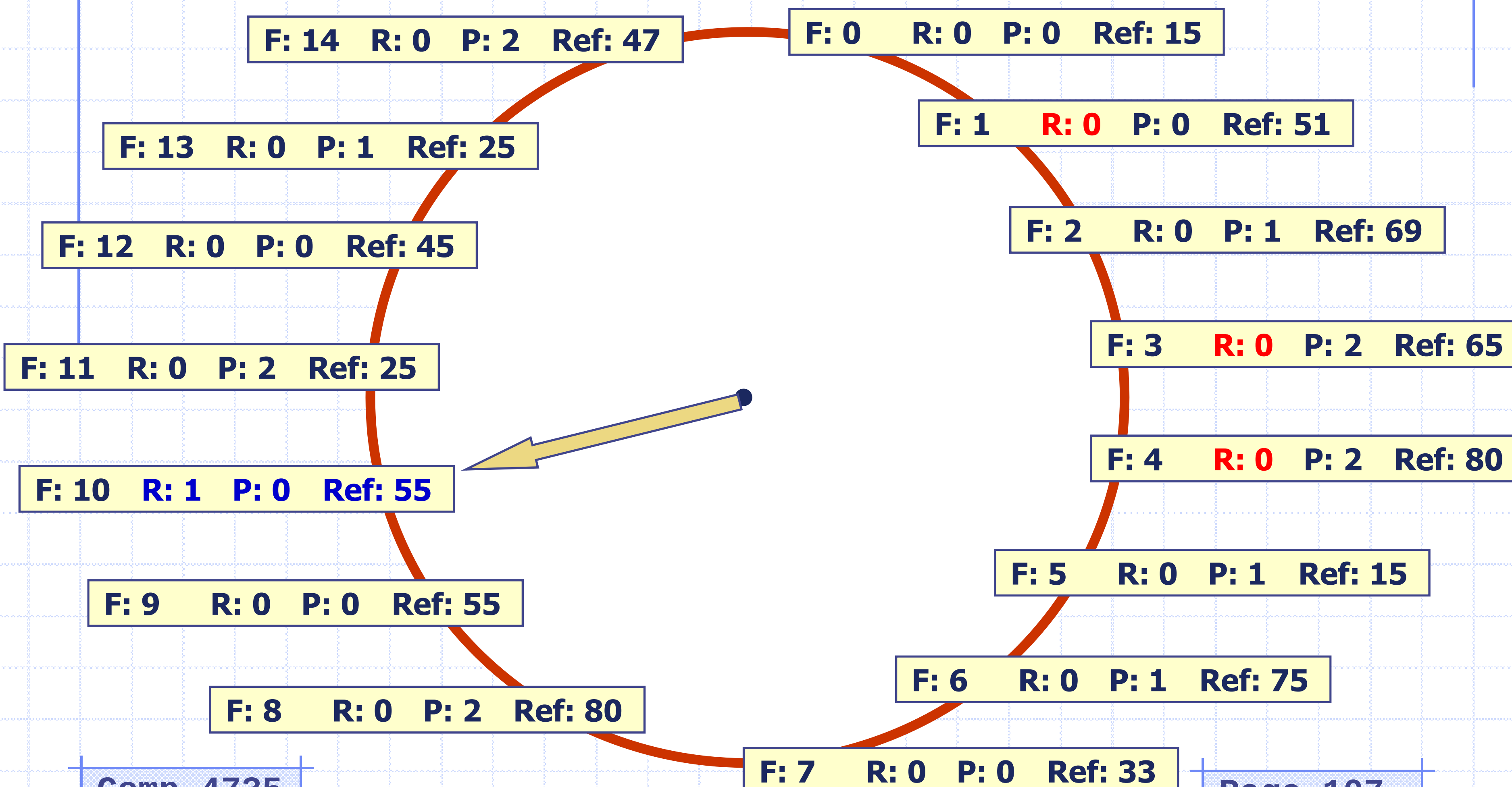
WSClock

- all R-bit are reset (except the current page), and

... execution of P0 continues

$\omega = 25$

P0: cvt = 55
P1: cvt = 75
P2: cvt = 80



WSClock Special Cases

- what happens if the clock had goes all the way around?
  - this means that all pages are in the working set
  - randomly select a victim and write to disk / claim it
- in practice, after a page is selected for replacement, we check if it is dirty
  - if it is dirty we schedule it to be written to disk, but don't claim it
  - we move on to the next frame, and keep scheduling dirty pages for write to disk
  - if we find a page that is not in the working set, and is clean, we claim it
  - if we don't find a clean page the hand just keeps moving; it will find one as soon as the write to disk is finished

The End