Sistemi Operativi I

Corso di Laurea in Informatica

2024-2025

Gabriele Tolomei
Dipartimento di Informatica
Sapienza Università di Roma
tolomei@di.uniroma1.it

Recap of the Last Lecture

- Virtual Memory allows processes to extend their memory footprint beyond the limit of the physical RAM
- Combined to paging, uses secondary storage (i.e., disks) as backup for unallocated frames
- Whenever a process requests a page, this could either be in main memory or on disk (**page fault**)
- Ideally, the OS should keep in main memory each process' **working set** to lower the chance of a page fault

Page Replacement: Motivation

- On a page fault, we need to load a page from disk into memory

Page Replacement: Motivation

- On a page fault, we need to load a page from disk into memory
- If physical memory has still free frames, the page can be safely loaded into one of those

Page Replacement: Motivation

- On a page fault, we need to load a page from disk into memory
- If physical memory has still free frames, the page can be safely loaded into one of those
- If physical memory is full, a frame must be swapped out to make room for the swap-in page

Page Replacement: Motivation

- On a page fault, we need to load a page from disk into memory
- If physical memory has still free frames, the page can be safely loaded into one of those
- If physical memory is full, a frame must be swapped out to make room for the swap-in page
- Several algorithms to select the page to evict from memory

Page Replacement Algorithms

- Random: pick any page at random (works surprisingly well!)

Page Replacement Algorithms

- Random: pick any page at random (works surprisingly well!)
- FIFO (First-In-First-Out): throw out the page that has been in memory for longest time (i.e., the oldest)
  - Easy to implement but may remove frequently accessed pages

Page Replacement Algorithms

- Random: pick any page at random (works surprisingly well!)
- FIFO (First-In-First-Out): throw out the page that has been in memory for longest time (i.e., the oldest)
  - Easy to implement but may remove frequently accessed pages
- MIN (OPT): remove the page that will not be accessed for the longest time (provably optimal [Belady 1966])
  - Needs to predict the future → very hard!

Page Replacement Algorithms

- Random: pick any page at random (works surprisingly well!)
- FIFO (First-In-First-Out): throw out the page that has been in memory for longest time (i.e., the oldest)
  - Easy to implement but may remove frequently accessed pages
- MIN (OPT): remove the page that will not be accessed for the longest time (provably optimal [Belady 1966])
  - Needs to predict the future → very hard!
- LRU (Least Recently Used): approximation of MIN, remove the page that has not been used in the longest time
  - Assumes the past is a good predictor of the future (not always true!)

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1											
F_2											
F_3											

How many page faults (denoted by *)?

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1											
F_2											
F_3											

Initially, no frame is loaded in memory at all
(pure demand paging)

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1											
F_2											
F_3											

Virtual address within page A is referenced

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1											
F_2											
F_3											

Virtual address within page A is referenced

page fault

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*										
F_2											
F_3											

Virtual address within page A is referenced

page fault → A loaded

FIFO = A

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A									
F_2											
F_3											

Virtual address within page B is referenced

FIFO = A

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A									
F_2											
F_3											

Virtual address within page B is referenced

page fault

FIFO = A

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A									
F_2		B*									
F_3											

Virtual address within page B is referenced

page fault → B loaded

FIFO = A → B

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A								
F_2		B*	B								
F_3											

Virtual address within page C is referenced

FIFO = A → B

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A								
F_2		B*	B								
F_3											

Virtual address within page C is referenced

page fault

FIFO = A → B

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A								
F_2		B*	B								
F_3			C*								

Virtual address within page C is referenced

page fault → C loaded

FIFO = A → B → C

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A							
F_2		B*	B	B							
F_3			C*	C							

Virtual address within page A is referenced

A is already loaded

FIFO = A → B → C

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A						
F_2		B*	B	B	B						
F_3			C*	C	C						

Virtual address within page B is referenced

B is already loaded

FIFO = A → B → C

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	A					
F_2		B*	B	B	B	B					
F_3			C*	C	C	C					

Virtual address within page D is referenced

FIFO = A → B → C

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	A					
F_2		B*	B	B	B	B					
F_3			C*	C	C	C					

Virtual address within page D is referenced

page fault

FIFO = A → B → C

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	D*					
F_2		B*	B	B	B	B					
F_3			C*	C	C	C					

Virtual address within page D is referenced

page fault

A replaced
D loaded

FIFO = B → C → D

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	D*	D				
F_2		B*	B	B	B	B	B				
F_3			C*	C	C	C	C				

Virtual address within page A is referenced

FIFO = B → C → D

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	$\boxed{A}$	D	B	C	A
F_1	A*	A	A	A	A	D*	D				
F_2		B*	B	B	B	B	B				
F_3			C*	C	C	C	C				

Virtual address within page A is referenced

page fault

FIFO = B → C → D

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	D*	D				
F_2		B*	B	B	B	B	A*				
F_3			C*	C	C	C	C				

Virtual address within page A is referenced

page fault → B replaced A loaded

FIFO = C → D → A

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	D*	D	D			
F_2		B*	B	B	B	B	A*	A			
F_3			C*	C	C	C	C	C			

Virtual address within page D is referenced

D is already loaded

FIFO = C → D → A

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	D*	D	D	D	C*	C
F_2		B*	B	B	B	B	A*	A	A	A	A
F_3			C*	C	C	C	C	C	B*	B	B

Eventually, we get a total of 7 page faults

MIN Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1											
F_2											
F_3											

How many page faults (denoted by *)?

MIN Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1											
F_2											
F_3											

Initially, no frame is loaded in memory at all (pure demand paging)

MIN Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A						
F_2		B*	B	B	B						
F_3			C*	C	C						

Up to this point, the same as FIFO

MIN Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	A					
F_2		B*	B	B	B	B					
F_3			C*	C	C	C					

Virtual address within page D is referenced

MIN Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	A					
F_2		B*	B	B	B	B					
F_3			C*	C	C	C					

Virtual address within page D is referenced

page fault

MIN Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	A					
F_2		B*	B	B	B	B					
F_3			C*	C	C	C					

Virtual address within page D is referenced

page fault

What's the page that will be requested the furthest away?

MIN Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	A					
F_2		B*	B	B	B	B					
F_3			C*	C	C	D*					

Virtual address within page D is referenced

page fault → C replaced D loaded

C is the page that will be requested the furthest away

MIN Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	$\boxed{B}$	C	A
F_1	A*	A	A	A	A	A	A	A	A		
F_2		B*	B	B	B	B	B	B	B		
F_3			C*	C	C	D*	D	D	D		

Up to this point, no more page faults

MIN Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	$\boxed{\text{C}}$	A
F_1	A*	A	A	A	A	A	A	A	A	A	
F_2		B*	B	B	B	B	B	B	B	B	
F_3			C*	C	C	D*	D	D	D	D	

Virtual address within page C is referenced

MIN Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	A	A	A	A	A	
F_2		B*	B	B	B	B	B	B	B	B	
F_3			C*	C	C	D*	D	D	D	D	

Virtual address within page C is referenced

page fault

What's the page that will be requested the furthest away?

MIN Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	A	A	A	A	A	
F_2		B*	B	B	B	B	B	B	B	C*	
F_3			C*	C	C	D*	D	D	D	D	

Virtual address within page C is referenced

page fault

B replaced
C loaded

B or D will be requested the furthest away (surely not A):
pick one (e.g., B)

MIN Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	A	A	A	A	A	A
F_2		B*	B	B	B	B	B	B	B	C*	C
F_3			C*	C	C	D*	D	D	D	D	D

Eventually, we get a total of 5 page faults

LRU Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1											
F_2											
F_3											

How many page faults (denoted by *)?

LRU Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1											
F_2											
F_3											

Initially, no frame is loaded in memory at all (pure demand paging)

LRU Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A						
F_2		B*	B	B	B						
F_3			C*	C	C						

Up to this point, the same as FIFO

LRU Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	A					
F_2		B*	B	B	B	B					
F_3			C*	C	C	C					

Virtual address within page D is referenced

page fault

LRU Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	$\boxed{\text{D}}$	A	D	B	C	A
F_1	A*	A	A	A	A	A					
F_2		B*	B	B	B	B					
F_3			C*	C	C	C					

Virtual address within page D is referenced

page fault

We can't look forward anymore!

LRU Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	A					
F_2		B*	B	B	B	B					
F_3			C*	C	C	D*					

Virtual address within page D is referenced

page fault

C replaced
D loaded

C is the page that has not been used for the longest time in the past

LRU Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	$\boxed{B}$	C	A
F_1	A*	A	A	A	A	A	A	A	A		
F_2		B*	B	B	B	B	B	B	B		
F_3			C*	C	C	D*	D	D	D		

Up to this point, no more page faults

LRU Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	A	A	A	A	A	
F_2		B*	B	B	B	B	B	B	B	B	
F_3			C*	C	C	D*	D	D	D	D	

Virtual address within page C is referenced

LRU Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	$\boxed{C}$	A
F_1	A*	A	A	A	A	A	A	A	A	A	
F_2		B*	B	B	B	B	B	B	B	B	
F_3			C*	C	C	D*	D	D	D	D	

Virtual address within page C is referenced

page fault

We can't look forward anymore!

LRU Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	$\boxed{C}$	A
F_1	A*	A	A	A	A	A	A	A	A	C*	
F_2		B*	B	B	B	B	B	B	B	B	
F_3			C*	C	C	D*	D	D	D	D	

Virtual address within page C is referenced

page fault ⇒ A replaced, C loaded

A is the page that has not been used for the longest time in the past

LRU Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	A	A	A	A	C*	C
F_2		B*	B	B	B	B	B	B	B	B	B
F_3			C*	C	C	D*	D	D	D	D	D

Virtual address within page A is referenced

LRU Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	A	A	A	A	C*	C
F_2		B*	B	B	B	B	B	B	B	B	B
F_3			C*	C	C	D*	D	D	D	D	D

Virtual address within page A is referenced

page fault

We can't look forward anymore!

LRU Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	A	A	A	A	C*	C
F_2		B*	B	B	B	B	B	B	B	B	B
F_3			C*	C	C	D*	D	D	D	D	A*

Virtual address within page A is referenced

page fault

D replaced
A loaded

D is the page that has not been used for the longest time in the past

LRU Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	A	A	A	A	C*	C
F_2		B*	B	B	B	B	B	B	B	B	B
F_3			C*	C	C	D*	D	D	D	D	A*

Eventually, we get a total of 6 page faults

LRU Page Replacement: (Unlucky) Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, D, A, B, C, D, A, B, C

	A	B	C	D	A	B	C	D	A	B	C
F_1											
F_2											
F_3											

How many page faults (denoted by *)?

LRU Page Replacement: (Unlucky) Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, D, A, B, C, D, A, B, C

	A	B	C	D	A	B	C	D	A	B	C
F_1	A*	A	A								
F_2		B*	B								
F_3			C*								

LRU Page Replacement: (Unlucky) Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, D, A, B, C, D, A, B, C

	A	B	C	D	A	B	C	D	A	B	C
F_1	A*	A	A	D*							
F_2		B*	B	B							
F_3			C*	C							

LRU Page Replacement: (Unlucky) Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, D, A, B, C, D, A, B, C

	A	B	C	D	$\boxed{A}$	B	C	D	A	B	C
F_1	A*	A	A	D*	D						
F_2		B*	B	B	A*						
F_3			C*	C	C						

LRU Page Replacement: (Unlucky) Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, D, A, B, C, D, A, B, C

	A	B	C	D	A	B	C	D	A	B	C
F_1	A*	A	A	D*	D	D					
F_2		B*	B	B	A*	A					
F_3			C*	C	C	B*					

LRU Page Replacement: (Unlucky) Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, D, A, B, C, D, A, B, C

	A	B	C	D	A	B	C	D	A	B	C
F_1	A*	A	A	D*	D	D	C*				
F_2		B*	B	B	A*	A	A				
F_3			C*	C	C	B*	B				

LRU Page Replacement: (Unlucky) Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, D, A, B, C, D, A, B, C

	A	B	C	D	A	B	C	D	A	B	C
F_1	A*	A	A	D*	D	D	C*	C	C	B*	B
F_2		B*	B	B	A*	A	A	D*	D	D	C*
F_3			C*	C	C	B*	B	B	A*	A	A

Eventually, we get a total of 11 page faults

Page Replacement: What If We Add Memory?

- Does adding memory always reduce the number of page faults?
- Intuitively, it would seem so…
- The answer, in fact, depends on the page replacement algorithm
- Let's see this with an example, using FIFO page replacement

FIFO Page Replacement: Example

5 virtual pages: A, B, C, D, E

3 physical frames: F_1, F_2, F_3

Scenario 1

4 physical frames: F_1, F_2, F_3, F_4

Scenario 2

Reference sequence of pages: A, B, C, D, A, B, E, A, B, C, D, E

FIFO Page Replacement: Example

	A	B	C	D	A	B	E	A	B	C	D	E

F_1	A*	A	A	D*	D	D	E*	E	E	E	E	E
F_2		B*	B	B	A*	A	A	A	A	C*	C	C
F_3			C*	C	C	B*	B	B	B	B	D*	D

F_1	A*	A	A	A	A	A	E*	E	E	E	D*	D
F_2		B*	B	B	B	B	B	A*	A	A	A	E*
F_3			C*	C	C	C	C	C	B*	B	B	B
F_4				D*	D	D	D	D	D	C*	C	C

FIFO Page Replacement: Example

	A	B	C	D	A	B	E	A	B	C	D	E

F_1	A*	A	A	D*	D	D	E*	E	E	E	E	E
F_2		B*	B	B	A*	A	A	A	A	C*	C	C
F_3			C*	C	C	B*	B	B	B	B	D*	D

9 page faults

F_1	A*	A	A	A	A	A	E*	E	E	E	D*	D
F_2		B*	B	B	B	B	B	A*	A	A	A	E*
F_3			C*	C	C	C	C	C	B*	B	B	B
F_4				D*	D	D	D	D	D	C*	C	C

10 page faults

Belady's Anomaly

Adding page frames may cause more page faults with some algorithms

LRU Page Replacement: Example

	A	B	C	D	A	B	E	A	B	C	D	E

F_1	A*	A	A	D*	D	D	E*	E	E	C*	C	C
F_2		B*	B	B	A*	A	A	A	A	A	D*	D
F_3			C*	C	C	B*	B	B	B	B	B	B

9 page faults

F_1	A*	A	A	A	A	A	A	A	A	A	A	E*
F_2		B*	B	B	B	B	B	B	B	B	B	B
F_3			C*	C	C	C	E*	E	E	E	D*	D
F_4				D*	D	D	D	D	D	C*	C	C

8 page faults

With LRU, adding page frames always decreases the number of page faults

LRU Page Replacement: Example

	A	B	C	D	A	B	E	A	B	C	D	E

F_1	A*	A	A	D*	D	D	E*	E	E	C*	C	C
F_2		B*	B	B	A*	A	A	A	A	A	D*	D
F_3			C*	C	C	B*	B	B	B	B	B	B

9 page faults

F_1	A*	A	A	A	A	A	A	A	A	A	A	E*
F_2		B*	B	B	B	B	B	B	B	B	B	B
F_3			C*	C	C	C	E*	E	E	E	D*	D
F_4				D*	D	D	D	D	D	C*	C	C

8 page faults

With LRU, adding page frames always decreases the number of page faults

Why?

LRU Page Replacement: Example

	A	B	C	D	A	B	E	A	B	C	D	E
F_1	A*	A	A	D*	D	D	E*	E	E	C*	C	C
F_2		B*	B	B	A*	A	A	A	A	A	D*	D
F_3			C*	C	C	B*	B	B	B	B	B	B

F_1	A*	A	A	A	A	A	A	A	A	A	A	E*
F_2		B*	B	B	B	B	B	B	B	B	B	B
F_3			C*	C	C	C	E*	E	E	E	D*	D
F_4				D*	D	D	D	D	D	C*	C	C

At each point in time 4-frame memory contains a subset of 3-frame

Can't do any worst!

Page Replacement: Summary

- FIFO is easy to implement but may lead to too many page faults
- May suffer from Belady's Anomaly

Page Replacement: Summary

- MIN is the optimal choice but cannot be used in practice since future memory references are never known in advance

Page Replacement: Summary

- LRU is a fair approximation of MIN assuming the past is a good predictor of the future
  - Exploits the locality reference (small working set that fits in memory)
  - Works poorly when the locality reference doesn't hold (large working set)

LRU: Implementation Details

How could we implement LRU page replacement algorithm?

LRU: Implementation Details

How could we implement LRU page replacement algorithm?

⇓

First Idea

Keep a timestamp for each page with the time it has been last accessed
Remove the page with the highest difference w.r.t. current timestamp

LRU: Implementation Details

How could we implement LRU page replacement algorithm?

⇓

First Idea

Keep a timestamp for each page with the time it has been last accessed

Remove the page with the highest difference w.r.t. current timestamp

Problems?

LRU: Implementation Details

How could we implement LRU page replacement algorithm?

⇩

First Idea

Keep a timestamp for each page with the time it has been last accessed
Remove the page with the highest difference w.r.t. current timestamp

Problems?

Every time a page is accessed its timestamp must be updated

LRU: Implementation Details

How could we implement LRU page replacement algorithm?

⇓

First Idea

Keep a timestamp for each page with the time it has been last accessed
Remove the page with the highest difference w.r.t. current timestamp

Problems?

- Every time a page is accessed its timestamp must be updated
- Linear scan of all the pages to select the one to be removed

LRU: Implementation Details

How could we implement LRU page replacement algorithm?

Second Idea

Keep a list of pages with the most recently used in front and the least recently used at the end: every time a page is accessed move it to front

LRU: Implementation Details

How could we implement LRU page replacement algorithm?

⇩

<u>Second Idea</u>

Keep a list of pages with the most recently used in front and the least recently used at the end: every time a page is accessed move it to front

Problems?

LRU: Implementation Details

How could we implement LRU page replacement algorithm?

⇩

<u>Second Idea</u>

Keep a list of pages with the most recently used in front and the least recently used at the end: every time a page is accessed move it to front

Problems?

Still too expensive as the OS must change multiple pointers on each memory access

LRU: Approximated Implementation

- In practice, no need for perfect LRU

LRU: Approximated Implementation

- In practice, no need for perfect LRU
- Many systems provide some HW support which is enough to approximate LRU fairly well

LRU: Approximated Implementation

- In practice, no need for perfect LRU
- Many systems provide some HW support which is enough to approximate LRU fairly well
- Single-Reference Bit → 1 bit for each page table entry

LRU: Approximated Implementation

- In practice, no need for perfect LRU
- Many systems provide some HW support which is enough to approximate LRU fairly well
- Single-Reference Bit → 1 bit for each page table entry
    - Initially, all bits for all pages are set to 0

LRU: Approximated Implementation

- In practice, no need for perfect LRU
- Many systems provide some HW support which is enough to approximate LRU fairly well
- Single-Reference Bit → 1 bit for each page table entry
  - Initially, all bits for all pages are set to 0
  - On each access to a page, the HW sets the reference bit to 1

LRU: Approximated Implementation

- In practice, no need for perfect LRU
- Many systems provide some HW support which is enough to approximate LRU fairly well
- Single-Reference Bit → 1 bit for each page table entry
  - Initially, all bits for all pages are set to 0
  - On each access to a page, the HW sets the reference bit to 1
  - Enough to distinguish pages that have been accessed since the last clear

LRU: Approximated Implementation

- In practice, no need for perfect LRU
- Many systems provide some HW support which is enough to approximate LRU fairly well
- Single-Reference Bit → 1 bit for each page table entry
  - Initially, all bits for all pages are set to 0
  - On each access to a page, the HW sets the reference bit to 1
  - Enough to distinguish pages that have been accessed since the last clear
  - No total order of page access

LRU: Approximated Implementation

- Additional-Reference-Bits → e.g., 8 bits for each page table entry

LRU: Approximated Implementation

- Additional-Reference-Bits → e.g., 8 bits for each page table entry
  - At regular intervals (clock interrupts) or every memory access, the OS right-shifts each of the reference bytes by one bit

LRU: Approximated Implementation

- Additional-Reference-Bits → e.g., 8 bits for each page table entry
  - At regular intervals (clock interrupts) or every memory access, the OS right-shifts each of the reference bytes by one bit
  - The high-order (leftmost) bit is filled in with the value of the reference bit (1 for the referenced page, 0 for all the others)

LRU: Approximated Implementation

- Additional-Reference-Bits → e.g., 8 bits for each page table entry
  - At regular intervals (clock interrupts) or every memory access, the OS right-shifts each of the reference bytes by one bit
  - The high-order (leftmost) bit is filled in with the value of the reference bit (1 for the referenced page, 0 for all the others)
  - At any given time, the page with the **smallest value** for the reference byte is the LRU page

LRU: Approximated Implementation

- Additional-Reference-Bits → e.g., 8 bits for each page table entry
  - At regular intervals (clock interrupts) or every memory access, the OS right-shifts each of the reference bytes by one bit
  - The high-order (leftmost) bit is filled in with the value of the reference bit (1 for the referenced page, 0 for all the others)
  - At any given time, the page with the **smallest value** for the reference byte is the LRU page
- The specific number of bits used and the frequency with which the reference byte is updated are adjustable

LRU: Approximated Implementation

- Second Chance Algorithm → Single-Reference Bit + FIFO

LRU: Approximated Implementation

- Second Chance Algorithm → Single-Reference Bit + FIFO
- OS keeps frames in a FIFO circular list

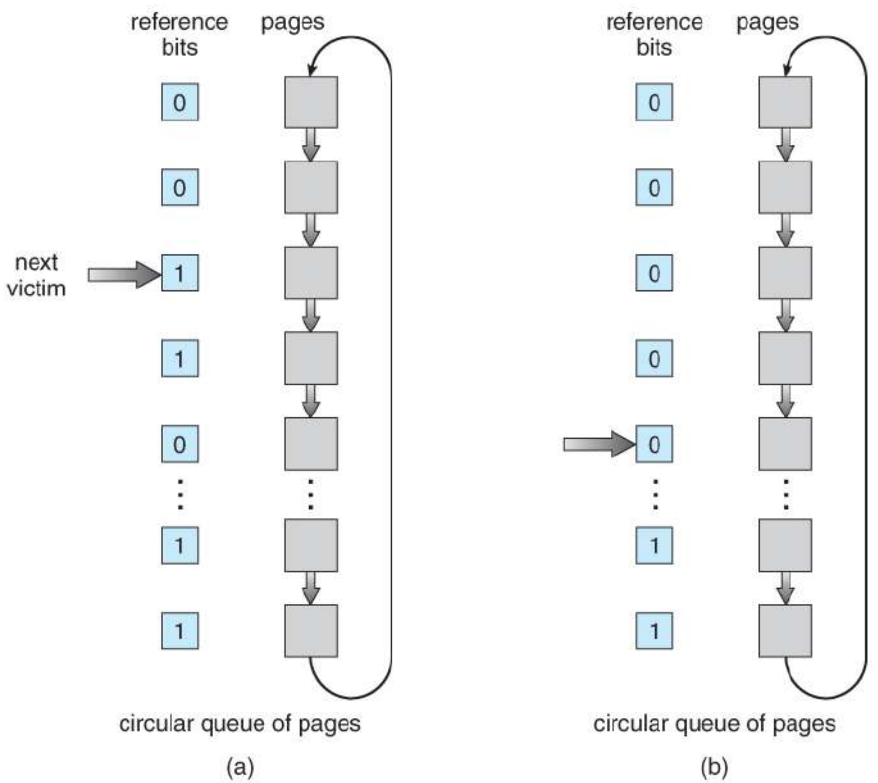
LRU: Approximated Implementation

- Second Chance Algorithm → Single-Reference Bit + FIFO
- OS keeps frames in a FIFO circular list
- On every memory access, the reference bit is set to 1

LRU: Approximated Implementation

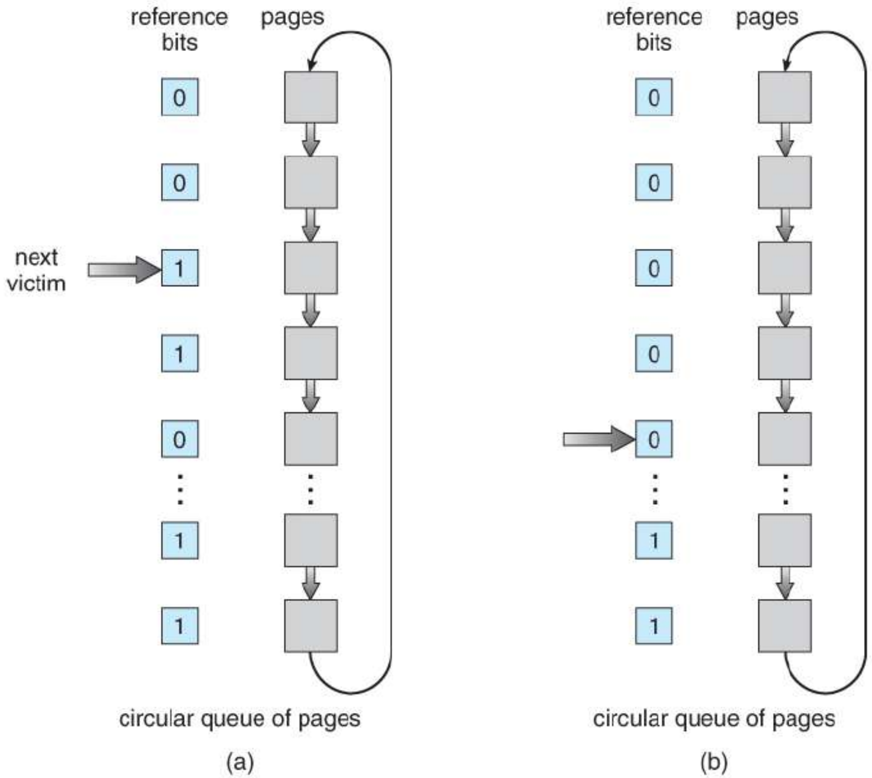
- Second Chance Algorithm → Single-Reference Bit + FIFO
- OS keeps frames in a FIFO circular list
- On every memory access, the reference bit is set to 1
- On a page fault, the OS scans the list of frames, checking the reference bit of the frame:
  - If this is 0, it replaces the page and sets it to 1
  - If this is 1, it sets it to 0 (second chance) and move to the next frame

Second Chance Algorithm (Clock)



A raw partitioning into: young vs. old frames

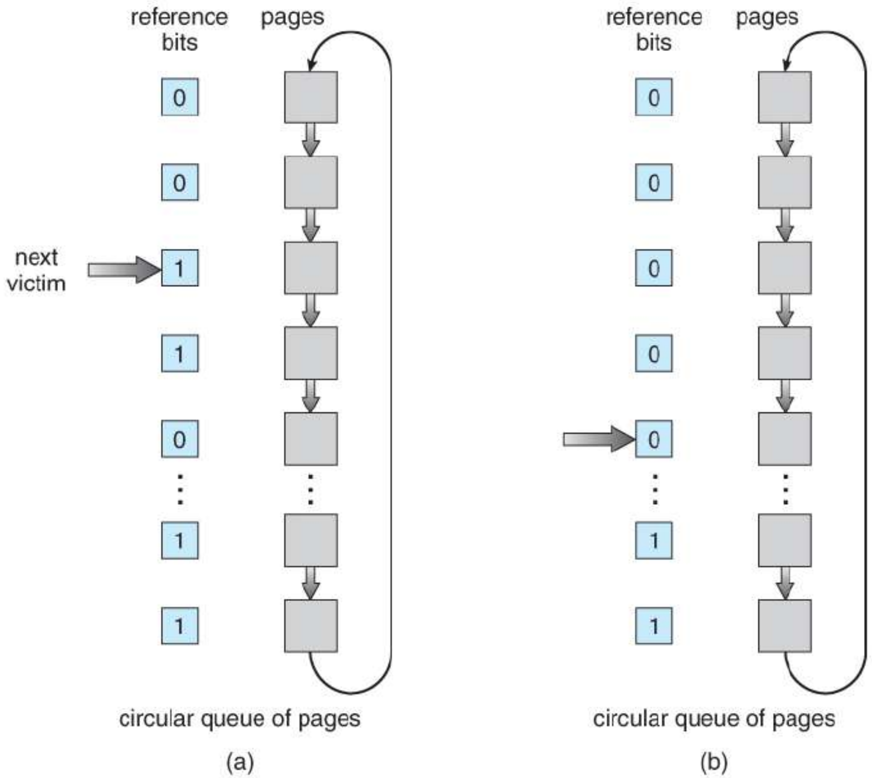
Second Chance Algorithm (Clock)



A raw partitioning into: young vs. old frames

Less accurate than additional-reference-bits

Second Chance Algorithm (Clock)

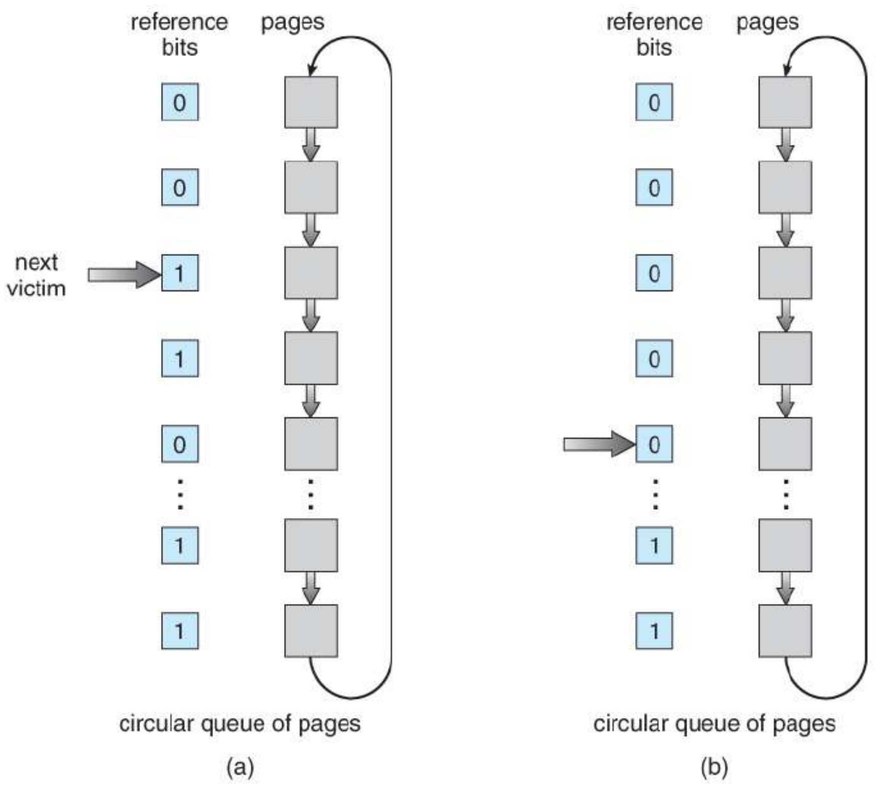


A raw partitioning into: young vs. old frames

Less accurate than additional-reference-bits

Fast, since it needs only to set a single bit on each memory reference (no need for a shift)

Second Chance Algorithm (Clock)



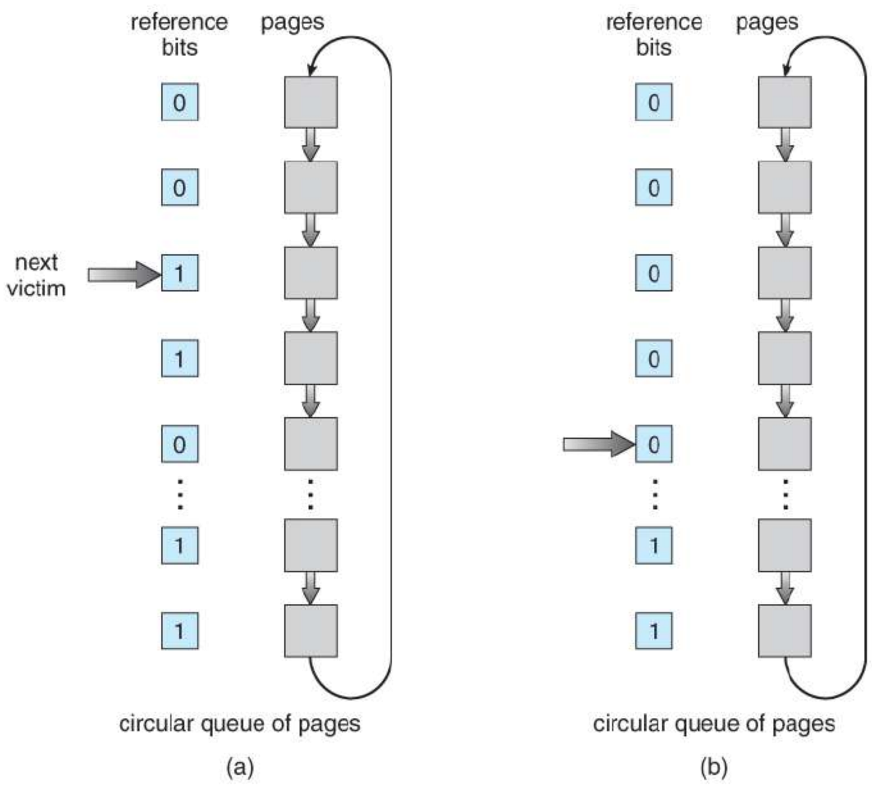
A raw partitioning into: young vs. old frames

Less accurate than additional-reference-bits

Fast, since it needs only to set a single bit on each memory reference (no need for a shift)

Page fault management is quicker as there is no need to scan the whole list of frames (on average) unless every frame has its bit set

Second Chance Algorithm (Clock)



A raw partitioning into: young vs. old frames

Less accurate than additional-reference-bits

Fast, since it needs only to set a single bit on each memory reference (no need for a shift)

Page fault management is quicker as there is no need to scan the whole list of frames (on average) unless every frame has its bit set

This algorithm is also known as clock because it mimics the hands of a clock

Enhanced Second Chance Algorithm

- Page replacement generally involves 2 I/O operations:
  - write the evicted page back to disk
  - read the newly referenced page from disk

Enhanced Second Chance Algorithm

- Page replacement generally involves 2 I/O operations:
  - write the evicted page back to disk
  - read the newly referenced page from disk
- Intuition: It is cheaper to replace a page which has not been modified, since the OS does not need to write this back to disk

Enhanced Second Chance Algorithm

- OS should give preference to paging-out un-modified frames
- Yet, it can proactively write to disk modified frames for later

Enhanced Second Chance Algorithm

- HW keeps a modify bit (in addition to the reference bit)
  - 1 means the page has been modified (different from the copy on disk)
  - 0 means the page is the same as the one stored on disk

Enhanced Second Chance Algorithm

- HW keeps a modify bit (in addition to the reference bit)
  - 1 means the page has been modified (different from the copy on disk)
  - 0 means the page is the same as the one stored on disk
- Use both the reference and modify bits (r, m) to classify pages into:
  - (0, 0): neither recently used nor modified;
  - (0, 1): not recently used, but modified;
  - (1, 0): recently used, but clean
  - (1, 1): recently used and modified

Enhanced Second Chance Algorithm

- This algorithm searches the page table in a circular fashion as before, yet making 4 distinct passes (one for each category)

Enhanced Second Chance Algorithm

- This algorithm searches the page table in a circular fashion as before, yet making 4 distinct passes (one for each category)
- First, it looks for a (0, 0) frame, and if it can't find one, it makes another pass looking for a (0, 1), etc.

Enhanced Second Chance Algorithm

- This algorithm searches the page table in a circular fashion as before, yet making 4 distinct passes (one for each category)
- First, it looks for a (0, 0) frame, and if it can't find one, it makes another pass looking for a (0, 1), etc.
- The first page with the lowest numbered category is replaced

Enhanced Second Chance Algorithm

- This algorithm searches the page table in a circular fashion as before, yet making 4 distinct passes (one for each category)
- First, it looks for a (0, 0) frame, and if it can't find one, it makes another pass looking for a (0, 1), etc.
- The first page with the lowest numbered category is replaced
- Prioritize replacement of clean pages if possible

Multiprogramming and Thrashing

- So far, we have implicitly assumed a single process is on the system

Multiprogramming and Thrashing

- So far, we have implicitly assumed a single process is on the system
- Multiple processes can however run concurrently on a single-CPU system

Multiprogramming and Thrashing

- So far, we have implicitly assumed a single process is on the system
- Multiple processes can however run concurrently on a single-CPU system
- The degree of multiprogramming is not fixed apriori, yet it is driven by the locality reference (a.k.a. 90÷10 rule)

Multiprogramming and Thrashing

- So far, we have implicitly assumed a single process is on the system
- Multiple processes can however run concurrently on a single-CPU system
- The degree of multiprogramming is not fixed apriori, yet it is driven by the locality reference (a.k.a. 90÷10 rule)
- This allows a system to load the **working set** (i.e., few pages) of many processes, thereby increasing the degree of multiprogramming

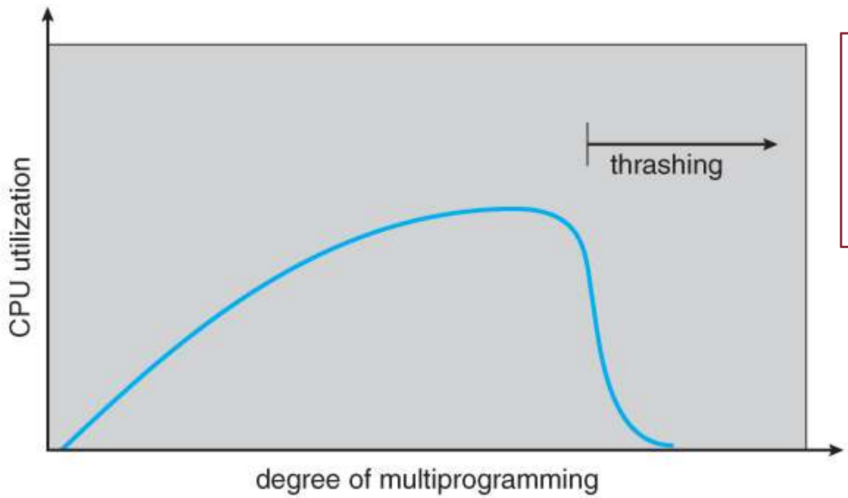
Multiprogramming and Thrashing

- When the degree of multiprogramming is too high, active working sets of running processes may saturate the whole memory capacity

Multiprogramming and Thrashing

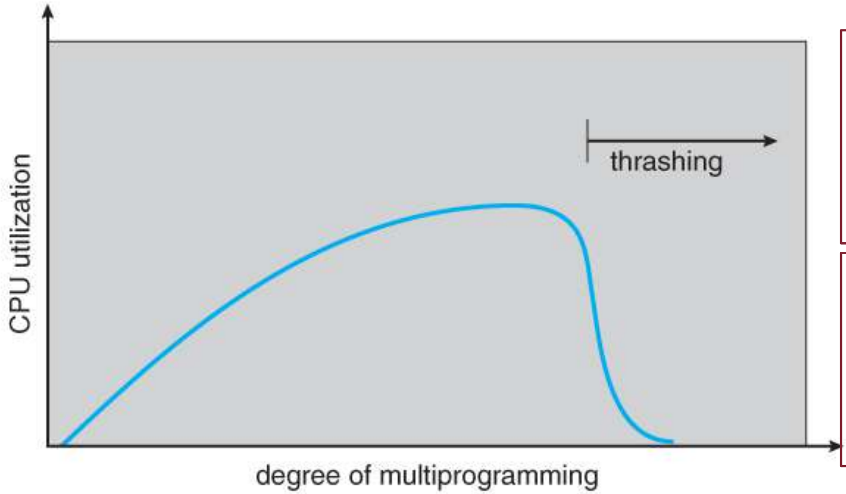
- When the degree of multiprogramming is too high, active working sets of running processes may saturate the whole memory capacity
- **Thrashing** → Memory is over-committed and pages are continuously tossed out while they are still in use
  - Memory access time approaches disk access time due to many page faults
  - Drastic degradation of performance

Multiprogramming and Thrashing



CPU utilization drops after a certain degree of multiprogramming

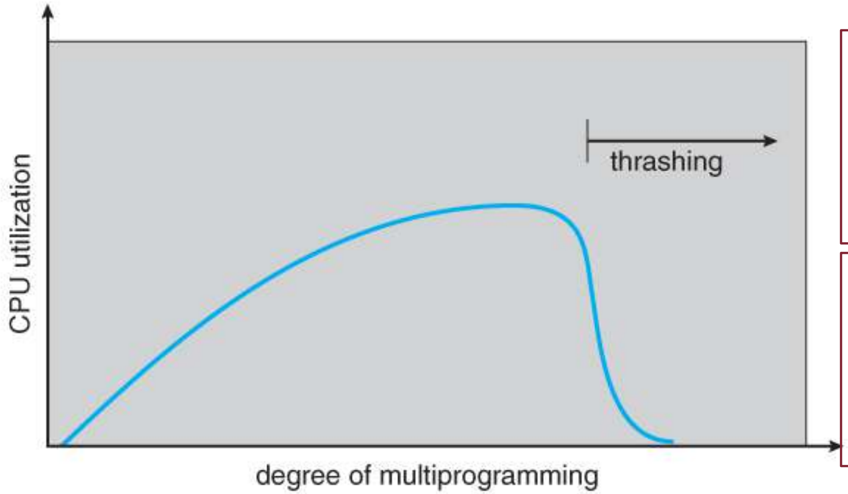
Multiprogramming and Thrashing



CPU utilization drops after a certain degree of multiprogramming

Eventually, also CPU-bound processes turn into I/O-bound ones (due to page faults)

Multiprogramming and Thrashing

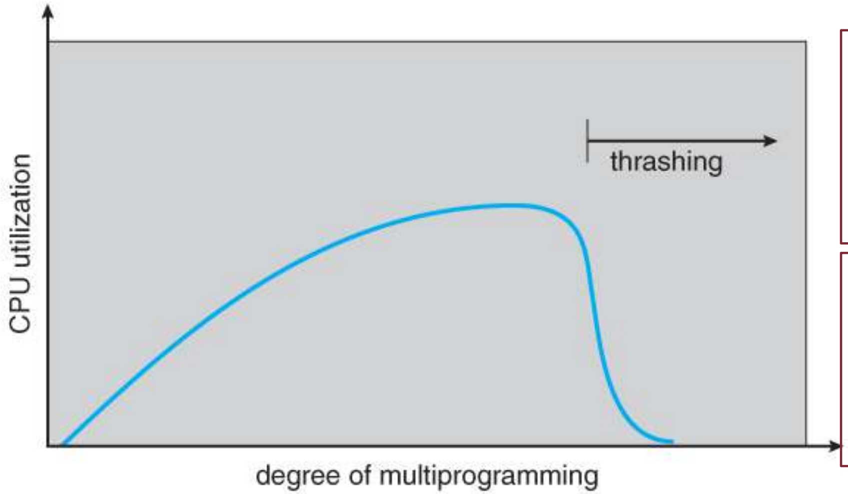


CPU utilization drops after a certain degree of multiprogramming

Eventually, also CPU-bound processes turn into I/O-bound ones (due to page faults)

What can we do to limit thrashing in a multiprogrammed system?

Multiprogramming and Thrashing



CPU utilization drops after a certain degree of multiprogramming

Eventually, also CPU-bound processes turn into I/O-bound ones (due to page faults)

What can we do to limit thrashing in a multiprogrammed system?

Fixing the degree of multi-programming apriori may be a too inflexible option

Allocation/Replacement Policies

Ultimately, we want to give each process enough memory so as to avoid thrashing

Allocation/Replacement Policies

Ultimately, we want to give each process enough memory so as to avoid thrashing

Global Allocation/Replacement

- All pages from all processes are in a single pool (single LRU queue)
- Upon page replacement, any page may be a potential victim, whether it currently belongs to the process seeking a free frame or not
- PRO: flexibility
- CON: thrashing more likely (no isolation)

Allocation/Replacement Policies

Ultimately, we want to give each process enough memory so as to avoid thrashing

Local Allocation/Replacement

- Each process has its own fixed pool of frames
- LRU replacement affects only each process' frames
- PRO: isolation
- CON: performance (a process may not be given enough memory)

Allocation/Replacement Policies

Ultimately, we want to give each process enough memory so as to avoid thrashing

Global Allocation/Replacement

- All pages from all processes are in a single pool (single LRU queue)
- Upon page replacement, any page may be a potential victim, whether it currently belongs to the process seeking a free frame or not
- PRO: flexibility
- CON: thrashing more likely (no isolation)

Local Allocation/Replacement

- Each process has its own fixed pool of frames
- LRU replacement affects only each process' frames
- PRO: isolation
- CON: performance (a process may not be given enough memory)

Local Allocation/Replacement

m = number of available physical page frames

n = number of processes

S_i = size of the i-th process; $S = \sum_{i=1}^{n} S_i =$ total size of all processes

Equal Allocation/Replacement: $\frac{m}{n}$

Proportional Allocation/Replacement: $\frac{m * S_i}{S}$

Local Allocation/Replacement

m = number of available physical page frames

n = number of processes

S_i = size of the i-th process; $S = \sum_{i=1}^{n} S_i =$ total size of all processes

Equal Allocation/Replacement: $\frac{m}{n}$

Proportional Allocation/Replacement: $\frac{m * S_i}{S}$

Variations on proportional allocation could consider priority of process rather than just their size

Local Allocation/Replacement

m = number of available physical page frames

n = number of processes

S_i = size of the i-th process; $S = \sum_{i=1}^{n} S_i =$ total size of all processes

Equal Allocation/Replacement: $\frac{m}{n}$

Proportional Allocation/Replacement: $\frac{m * S_i}{S}$

Variations on proportional allocation could consider priority of process rather than just their size

As allocations fluctuate over time, so does m
(processes must be swapped out or not started if not enough frames)

Proportional Allocation/Replacement

- This implicitly assumes that a large process will also refer to a large amount of memory

Proportional Allocation/Replacement

- This implicitly assumes that a large process will also refer to a large amount of memory
- Intuitively this makes sense: the more memory a process needs the higher the chance it will refer a much larger memory range

Proportional Allocation/Replacement

- This implicitly assumes that a large process will also refer to a large amount of memory
- Intuitively this makes sense: the more memory a process needs the higher the chance it will refer a much larger memory range
- However, there might be cases where this is not true
  - e.g., a process allocates a 1GB array but only uses a small portion of it

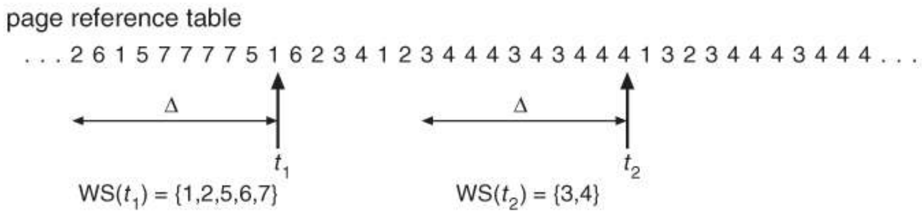
Proportional Allocation/Replacement

- This implicitly assumes that a large process will also refer to a large amount of memory
- Intuitively this makes sense: the more memory a process needs the higher the chance it will refer a much larger memory range
- However, there might be cases where this is not true
    - e.g., a process allocates a 1GB array but only uses a small portion of it
- In other words, the working set of a process may not be correlated with its (theoretical) memory footprint

Matching the Working Set

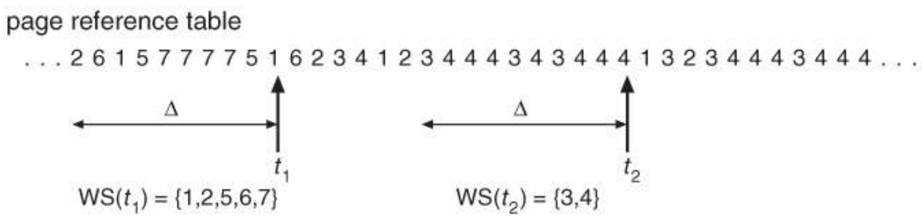
- **Goal** → Give each process enough frames to contain its working set
  - Informally, the working set is the set of pages the process is using "right now"
  - More formally, the working set of a process at time t, $W(t)$, is the set of all pages referenced during $(t-\Delta, t)$

Determining the Working Set



The selection of Δ is critical to the success of the working set model

Determining the Working Set

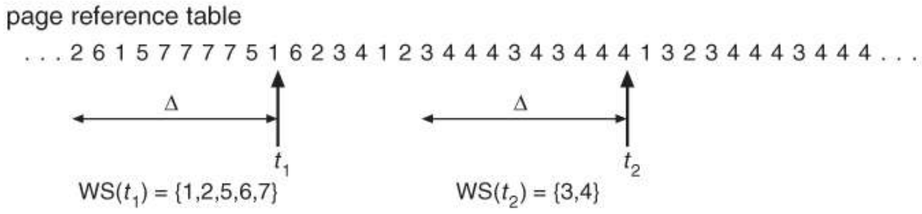


The selection of Δ is critical to the success of the working set model

Δ too small

it does not encompass all of the pages of the current locality

Determining the Working Set

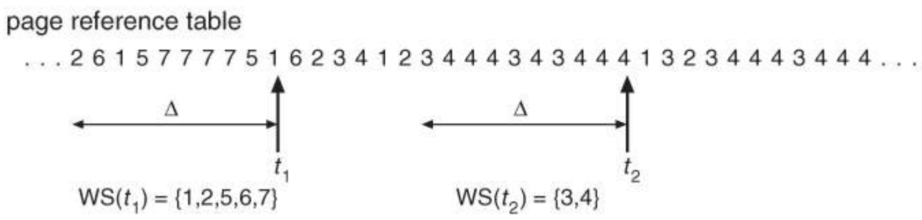


The selection of Δ is critical to the success of the working set model

Δ too large

it includes pages that are no longer frequently accessed

Determining the Working Set



The selection of Δ is critical to the success of the working set model

Δ too small
it does not encompass all of the pages of the current locality

Δ too large
it includes pages that are no longer frequently accessed

Exact tracking is expensive: update the working set at each memory access

Approximating the Working Set

- Computing the working set exactly requires to keep track of a moving window of size Δ

Approximating the Working Set

- Computing the working set exactly requires to keep track of a moving window of size Δ
- At each memory reference a new reference appears at one end and the oldest reference drops off the other end

Approximating the Working Set

- Computing the working set exactly requires to keep track of a moving window of size Δ
- At each memory reference a new reference appears at one end and the oldest reference drops off the other end
- To avoid the overhead of keeping a list of the last Δ referenced pages, the working set is often implemented with **sampling**

Approximating the Working Set

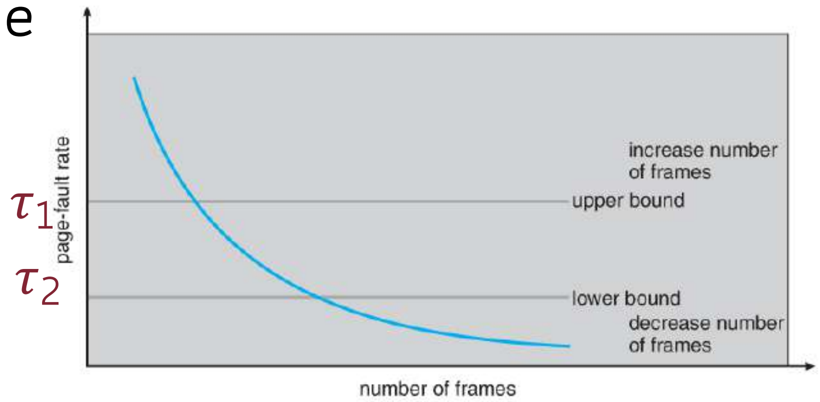
- Computing the working set exactly requires to keep track of a moving window of size Δ
- At each memory reference a new reference appears at one end and the oldest reference drops off the other end
- To avoid the overhead of keeping a list of the pages referenced within Δ, the working set is often implemented with **sampling**
- Every k memory references (e.g., $k = 1{,}000$), consider the working set to be all pages referenced within *that* period of time

Tracking Page Fault Rate

- Ultimately, our goal is to minimize the **`page fault rate`**

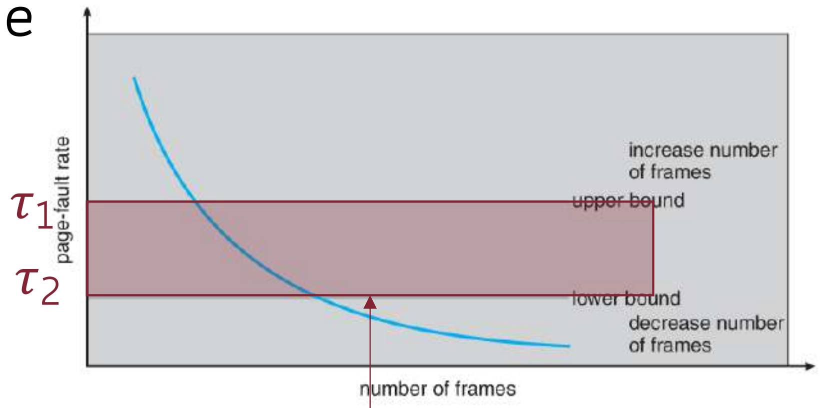
Tracking Page Fault Rate

- Ultimately, our goal is to minimize the **page fault rate**
- A more direct approach is to track page fault rate of each process:
  - If the page fault rate is above a given threshold τ_1 → give it more frames
  - If the page fault rate is below a given threshold τ_2 → reclaim some frames



Tracking Page Fault Rate

- Ultimately, our goal is to minimize the **page fault rate**
- A more direct approach is to track page fault rate of each process:
  - If the page fault rate is above a given threshold τ_1 → give it more frames
  - If the page fault rate is below a given threshold τ_2 → reclaim some frames



Dynamically adjust allocated frames so as to keep processes in this area

Kernel Memory

- So far, we only considered memory allocation for user processes

Kernel Memory

- So far, we only considered memory allocation for user processes
- But kernel needs memory to store things too: code and data structures like PCB, page tables, etc.

Kernel Memory

- So far, we only considered memory allocation for user processes
- But kernel needs memory to store things too: code and data structures like PCB, page tables, etc.
- Kernel does not use any of the advanced mechanisms seen so far
  - No paging → what if a page fault occurs for the kernel?

Kernel Memory: Buddy Allocator

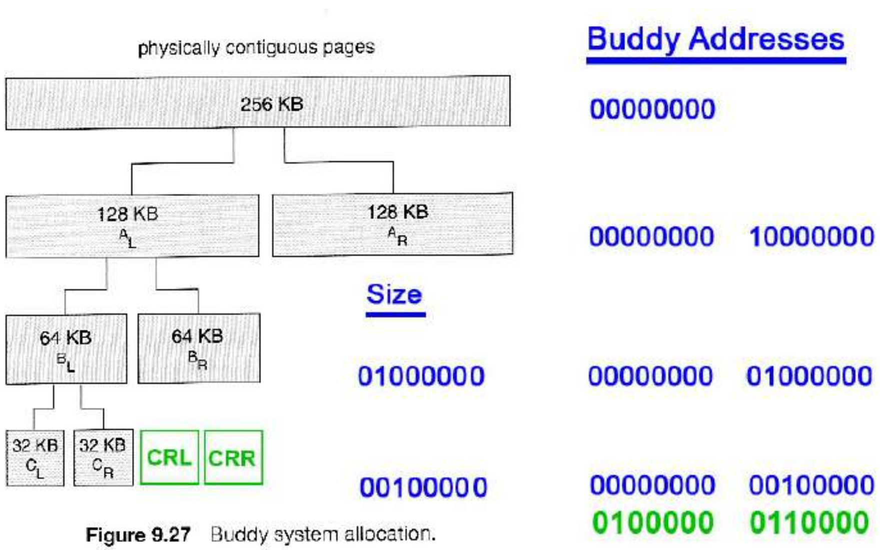


Figure 9.27 Buddy system allocation.

- Allocates memory using a power of 2 allocator (e.g., 4KiB, 8KiB, 16KiB), rounding up to the next nearest power of two if necessary

Kernel Memory: Buddy Allocator

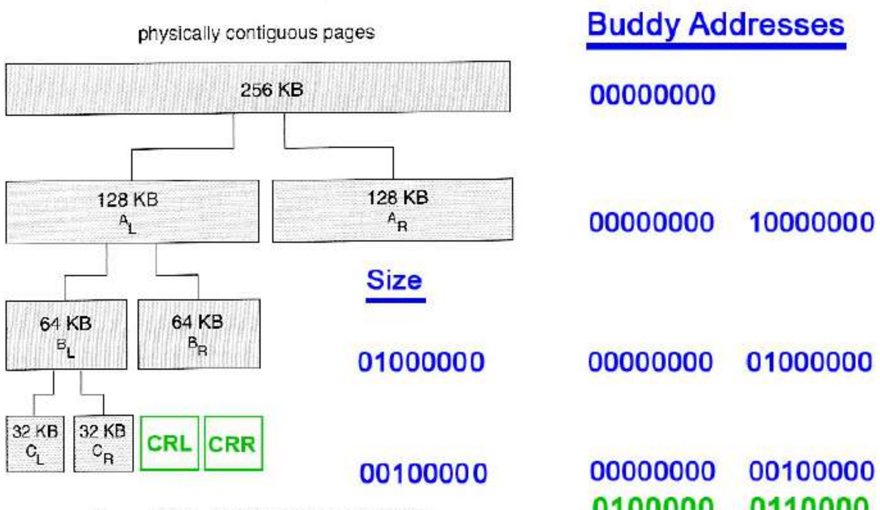


Figure 9.27 Buddy system allocation.

- Allocates memory using a power of 2 allocator (e.g., 4KiB, 8KiB, 16KiB), rounding up to the next nearest power of two if necessary
- If a block of the correct size is not available, then one is formed by (repeatedly) splitting the next larger block in two

Kernel Memory: Buddy Allocator

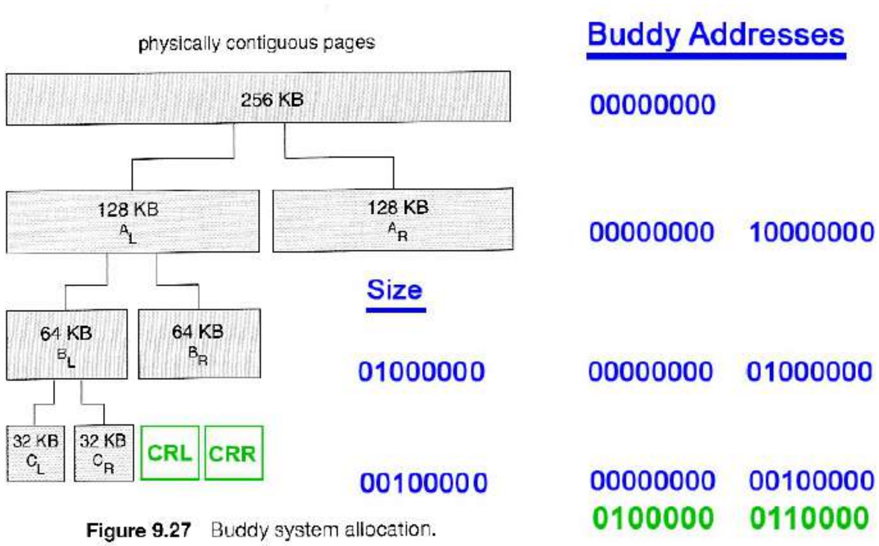
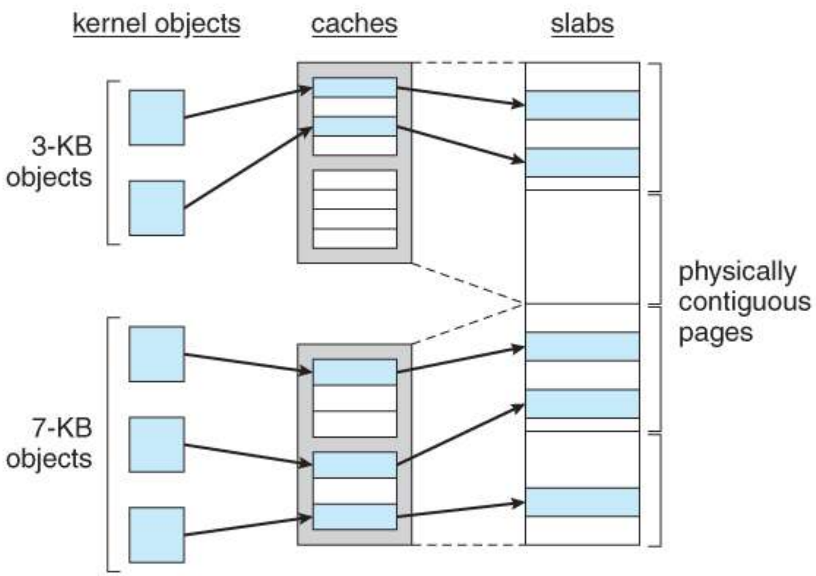


Figure 9.27 Buddy system allocation.

- Allocates memory using a power of 2 allocator (e.g., 4KiB, 8KiB, 16KiB), rounding up to the next nearest power of two if necessary
- If a block of the correct size is not available, then one is formed by (repeatedly) splitting the next larger block in two
- Can lead to internal fragmentation

Kernel Memory: Slab Allocator



- Group of objects of the same size in a **slab**
- Object cache points to one or more slabs
- Separate cache for each kernel data structure (e.g., PCB)
- No internal fragmentation
- Used in Solaris and Linux

Page Sizes

- Reasons for small pages?

Page Sizes

- Reasons for **small** pages?
  - Decreasing internal fragmentation
  - Higher degree of multiprogramming

Page Sizes

- Reasons for small pages?
  - Decreasing internal fragmentation
  - Higher degree of multiprogramming
- Reasons for large pages?

Page Sizes

- Reasons for small pages?
  - Decreasing internal fragmentation
  - Higher degree of multiprogramming
- Reasons for large pages?
  - Smaller page table size (i.e., smaller number of page table entries)
  - Fewer page faults (locality reference)
  - Amortizes disk overhead (reading a 1KiB page from disk takes approximately the same as reading an 8KiB one)

Summary of Page Replacement

- The choice of page replacement algorithm is crucial when physical memory is limited
  - All algorithms approach to the optimum as the physical memory allocated to a process approaches to the virtual memory size

Summary of Page Replacement

- The choice of page replacement algorithm is crucial when physical memory is limited
  - All algorithms approach to the optimum as the physical memory allocated to a process approaches to the virtual memory size
- The more processes running concurrently, the less physical memory each one can have

Summary of Page Replacement

- The choice of page replacement algorithm is crucial when physical memory is limited
  - All algorithms approach to the optimum as the physical memory allocated to a process approaches to the virtual memory size
- The more processes running concurrently, the less physical memory each one can have
- The OS must choose how many processes (and the number of frames per process) can share memory