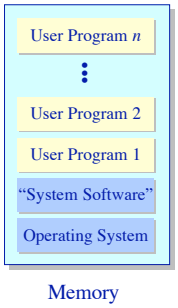
Page Replacement Algorithms

Virtual Memory Management
Fundamental issues : A Recap

- Key concept: Demand paging
  - Load pages into memory only when a page fault occurs
- Issues:
  - Placement strategies
    - Place pages anywhere – no placement policy required
  - Replacement strategies
    - What to do when there exist more jobs than can fit in memory
  - Load control strategies
    - Determining how many jobs can be in memory at one time



Memory

Page Replacement Algorithms
Concept

- Typically Σ, VAS_i >> *Physical Memory*
- With demand paging, physical memory fills quickly
- When a process faults & memory is full, some page must be swapped out
  - Handling a page fault now requires **2** disk accesses not 1!

Which page should be replaced?

Local replacement — Replace a page of the faulting process
Global replacement — Possibly replace the page of another process

Page Replacement Algorithms
Evaluation methodology

- Record a *trace* of the pages accessed by a process
  - Example: (Virtual) address trace...
    (3,0), (1,9), (4,1), (2,1), (5,3), (2,0), (1,9), (2,4), (3,1), (4,8)
  - generates page trace
    3, 1, 4, 2, 5, 2, 1, 2, 3, 4 (represented as *c, a, d, b, e, b, a, b, c, d*)

Simulate the behavior of a page replacement algorithm on the trace and record the number of page faults generated

fewer faults ⇨ *better performance*

Optimal Page Replacement
Clairvoyant replacement

- Replace the page that won't be needed for the longest time in the future

Time	0	1	2	3	4	5	6	7	8	9	10
Requests		c	a	d	b	e	b	a	b	c	d
Page Frames 0	a										
Page Frames 1	b										
Page Frames 2	c										
Page Frames 3	d										
Faults											
Time page needed next											

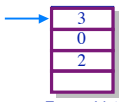
Optimal Page Replacement
Clairvoyant replacement

- Replace the page that won't be needed for the longest time in the future

Time	0	1	2	3	4	5	6	7	8	9	10
Requests		c	a	d	b	e	b	a	b	c	d
Page Frames 0	a	a	a	a	a	a	a	a	a	a	(d)
Page Frames 1	b	b	b	b	b	b	b	b	b	b	b
Page Frames 2	c	c	c	c	c	c	c	c	c	c	c
Page Frames 3	d	d	d	d	d	(e)	e	e	e	e	e
Faults						•					•
Time page needed next						$a = 7$ $b = 6$ $c = 9$ $d = 10$					$a = 15$ $b = 11$ $c = 13$ $d = 14$

Local Page Replacement
FIFO replacement

- Simple to implement
  - A single pointer suffices

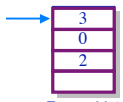


- Performance with 4 page frames:

Time	0	1	2	3	4	5	6	7	8	9	10
Requests		c	a	d	b	e	b	a	b	c	d
Page Frames 0	a										
Page Frames 1	b										
Page Frames 2	c										
Page Frames 3	d										
Faults											

Local Page Replacement
FIFO replacement

- Simple to implement
  - A single pointer suffices



- Performance with 4 page frames:

Time	0	1	2	3	4	5	6	7	8	9	10
Requests		c	a	d	b	e	b	a	b	c	d
Page Frames 0	a	a	a	a	a	(e)	e	e	e	e	(d)
Page Frames 1	b	b	b	b	b	b	b	(a)	a	a	a
Page Frames 2	c	c	c	c	c	c	c	c	(b)	b	b
Page Frames 3	d	d	d	d	d	d	d	d	d	(c)	c
Faults						•		•	•	•	•

Least Recently Used Page Replacement

Use the recent past as a predictor of the near future

- Replace the page that hasn't been referenced for the longest time

Time	0	1	2	3	4	5	6	7	8	9	10
Requests		c	a	d	b	e	b	a	b	c	d
Page Frames 0	a										
Page Frames 1	b										
Page Frames 2	c										
Page Frames 3	d										
Faults											
Time page last used											

Least Recently Used Page Replacement

Use the recent past as a predictor of the near future

- Replace the page that hasn't been referenced for the longest time

Time	0	1	2	3	4	5	6	7	8	9	10
Requests		c	a	d	b	e	b	a	b	c	d
Page Frames 0	a	a	a	a	a	a	a	a	a	a	a
Page Frames 1	b	b	b	b	b	b	b	b	b	b	b
Page Frames 2	c	c	c	c	c	**e**	e	e	e	e	**d**
Page Frames 3	d	d	d	d	d	d	d	d	d	**c**	c
Faults						•				•	•
Time page last used					$a=2$ $b=4$ $c=1$ $d=3$					$a=7$ $b=8$ $e=5$ $c=9$	$a=7$ $b=8$ $e=5$ $c=9$

Least Recently Used Page Replacement

Implementation

- Maintain a "stack" of recently used pages

Time	0	1	2	3	4	5	6	7	8	9	10
Requests		c	a	d	b	e	b	a	b	c	d
Page Frames 0	a	a	a	a	a	a	a	a	a	a	a
Page Frames 1	b	b	b	b	b	b	b	b	b	b	b
Page Frames 2	c	c	c	c	c	**e**	e	e	e	e	**d**
Page Frames 3	d	d	d	d	d	d	d	d	d	**c**	c
Faults						•				•	•

LRU page stack

Page to replace

Least Recently Used Page Replacement

Implementation

- Maintain a "stack" of recently used pages

Time	0	1	2	3	4	5	6	7	8	9	10
Requests		c	a	d	b	e	b	a	b	c	d
Page Frames 0	a	a	a	a	a	a	a	a	a	a	a
Page Frames 1	b	b	b	b	b	b	b	b	b	b	b
Page Frames 2	c	c	c	c	c	**e**	e	e	e	e	**d**
Page Frames 3	d	d	d	d	d	d	d	d	d	**c**	c
Faults						•				•	•

LRU page stack

	c	a	d	b	e	b	a	b	c	d
		c	a	d	b	e	b	a	b	c
			c	a	d	b	e	b	a	b
				c	a	d	b	e	b	a

Page to replace

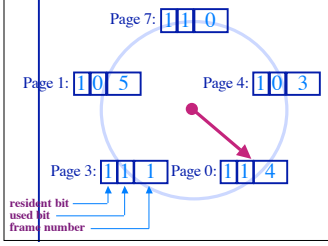
					c				**d**	**e**

Least Recently Used Page Replacement
Alternate Implementation --- Aging Register

- Maintain an n-bit aging register $R = R_{n-1}R_{n-2}...R_0$ for each page frame
  - On a page reference, set R_{n-1} to 1
  - Every T units of time, shift the aging vector right by one bit
- Key idea:
  - Aging vector can be interpreted as a positive binary number
  - Value of R decreases periodically unless the page is referenced
- Page replacement algorithm:
  - On a page fault, replace the page with the smallest value of R

Approximate LRU Page Replacement
The Clock algorithm

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




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


Clock Page Replacement
Example

Time	0	1	2	3	4	5	6	7	8	9	10
Requests		c	a	d	b	e	b	a	b	c	d
Page Frames 0	a	a	a	a	a						
Page Frames 1	b	b	b	b	b						
Page Frames 2	c	c	c	c	c						
Page Frames 3	d	d	d	d	d						
Faults											

Page table entries for resident pages:

1	a
1	b
1	c
1	d

Clock Page Replacement
Example

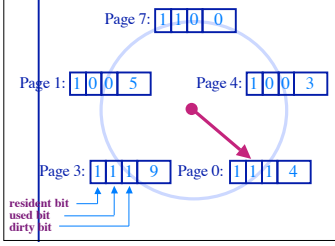
Time	0	1	2	3	4	5	6	7	8	9	10
Requests		c	a	d	b	e	b	a	b	c	d
Page Frames 0	a	a	a	a	a	(e)	e	e	e	e	(d)
Page Frames 1	b	b	b	b	b	b	b	b	b	b	b
Page Frames 2	c	c	c	c	c	c	c	(a)	a	a	a
Page Frames 3	d	d	d	d	d	d	d	d	d	(c)	c
Faults						•		•		•	•

Page table entries for resident pages:

1	a	1	e	1	e	1	e	1	e	1	e	1	e	1	d				
1	b	0	b	1	b	0	b	1	a	1	a	1	b	0	b				
1	c	0	c	0	c	1	a	0	d	0	d	1	a	0	a				
1	d	0	d	0	d	0	d					1	c	0	c				

Optimizing Approximate LRU Replacement
The Second Chance algorithm

- There is a significant cost to replacing "dirty" pages
- Modify the Clock algorithm to allow dirty pages to always survive one sweep of the clock hand
  - Use both the *dirty bit* and the *used bit* to drive replacement



Second Chance Algorithm

Before clock sweep		After clock sweep	
used	*dirty*	*used*	*dirty*
0	0	*replace page*	
0	1	0	0
1	0	0	0
1	1	0	1

The Second Chance Algorithm
Example

Time	0	1	2	3	4	5	6	7	8	9	10
Requests		*c*	a^w	*d*	b^w	*e*	*b*	a^w	*b*	*c*	*d*
Page Frames 0	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*
Page Frames 1	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*d*
Page Frames 2	*c*	*c*	*c*	*c*	*c*	*e*	*e*	*e*	*e*	*e*	*e*
Page Frames 3	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*c*	*c*
Faults											

Page table entries for resident pages:

10 *a*	10 *a*	10 *a*	10 *a*	10 *a*	10 *a*	10 *a*	10 *a*	10 *a*	10 *a*	10 *a*
10 *b*	10 *b*	10 *b*	10 *b*	10 *b*	10 *b*	10 *b*	10 *b*	10 *b*	10 *b*	10 *d*
10 *c*	10 *c*	10 *c*	10 *c*	10 *c*	10 *c*	10 *c*	10 *c*	10 *c*	10 *c*	10 *c*
10 *d*	10 *d*	10 *d*	10 *d*	10 *d*	10 *d*	10 *d*	10 *d*	10 *d*	10 *d*	10 *c*

The Second Chance Algorithm
Example

Time	0	1	2	3	4	5	6	7	8	9	10
Requests		*c*	a^w	*d*	b^w	*e*	*b*	a^w	*b*	*c*	*d*
Page Frames 0	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*
Page Frames 1	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*d*
Page Frames 2	*c*	*c*	*c*	*c*	*c*	*e*	*e*	*e*	*e*	*e*	*e*
Page Frames 3	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*c*	*c*
Faults											

Page table entries for resident pages:

10 *a*	11 *a*	00 a^*	00 a^*	11 *a*			11 *a*	00 a^*		
10 *b*	11 *b*	00 b^*	10 b^*	10 b^*			10 b^*	10 *d*		
10 *c*	10 *c*	10 *e*	10 *e*	10 *e*			10 *e*	00 *e*		
10 *d*	10 *d*	00 *d*	00 *d*	00 *d*			10 *c*	00 *c*		

The Problem With Local Page Replacement
How much memory do we allocate to a process?

Time	0	1	2	3	4	5	6	7	8	9	10	11	12
Requests		*a*	*b*	*c*	*d*	*a*	*b*	*c*	*d*	*a*	*b*	*c*	*d*
Page Frames 0	*a*												
Page Frames 1	*b*												
Page Frames 2	*c*												
Faults													

Time	0	1	2	3	4	5	6	7	8	9	10	11	12
Requests		*a*	*b*	*c*	*d*	*a*	*b*	*c*	*d*	*a*	*b*	*c*	*d*
Page Frames 0	*a*												
Page Frames 1	*b*												
Page Frames 2	*c*												
Page Frames 3	–												
Faults													

The Problem With Local Page Replacement
How much memory do we allocate to a process?

Time	0	1	2	3	4	5	6	7	8	9	10	11	12
Requests		a	b	c	d	a	b	c	d	a	b	c	d
Page Frames 0	a	a	a	a	d	d	d	c	c	c	b	b	b
Page Frames 1	b	b	b	b	b	a	a	a	d	d	d	c	c
Page Frames 2	c	c	c	c	c	c	b	b	b	a	a	a	d
Faults					•	•	•	•	•	•	•	•	•

Time	0	1	2	3	4	5	6	7	8	9	10	11	12
Requests		a	b	c	d	a	b	c	d	a	b	c	d
Page Frames 0	a	a	a	a	a	a	a	a	a	a	a	a	a
Page Frames 1	b	b	b	b	b	b	b	b	b	b	b	b	b
Page Frames 2	c	c	c	c	c	c	c	c	c	c	c	c	c
Page Frames 3	–				d	d	d	d	d	d	d	d	d
Faults					•								

Page Replacement Algorithms
Performance

- Local page replacement
  - LRU — Ages pages based on when they were last used
  - FIFO — Ages pages based on when they're brought into memory
- Towards global page replacement ... with variable number of page frames allocated to processes

The principle of locality

  - 90% of the execution of a program is sequential
  - Most iterative constructs consist of a relatively small number of instructions
  - When processing large data structures, the dominant cost is sequential processing on individual structure elements

Optimal Page Replacement
For processes with a variable number of frames

- *VMIN* — Replace a page that is not referenced in the *next* τ accesses
- Example: $\tau = 4$

Time	0	1	2	3	4	5	6	7	8	9	10
Requests		c	c	d	b	c	e	c	e	a	d
Pages in Memory: Page a	• $t=0$										
Page b	-										
Page c	-										
Page d	• $t=-1$										
Page e	-										
Faults											

Optimal Page Replacement
For processes with a variable number of frames

- *VMIN* — Replace a page that is not referenced in the *next* τ accesses
- Example: $\tau = 4$

Time	0	1	2	3	4	5	6	7	8	9	10
Requests		c	c	d	b	c	e	c	e	a	d
Pages in Memory: Page a	• $t=0$	-	-	-	-	-	-	-	-	F	-
Page b	-	-	-	-	F	-	-	-	-	-	-
Page c	-	F	•	•	•	•	•	•	-	-	-
Page d	• $t=-1$	•	•	•	-	-	-	-	-	-	F
Page e	-	-	-	-	-	-	F	•	•	-	-
Faults		•			•		•			•	•

Explicitly Using Locality
The working set model of page replacement

- Assume recently referenced pages are likely to be referenced again soon…
- … and *only* keep those pages recently referenced in memory (called *the working set*)
  - Thus pages may be removed even when no page fault occurs
  - The number of frames allocated to a process will vary over time
- A process is allowed to execute only if its working set fits into memory
  - The working set model performs implicit load control

Working Set Page Replacement
Implementation

- Keep track of the last τ references
  - The pages referenced during the last τ memory accesses are the working set
  - τ is called the *window size*
- Example: Working set computation, τ = 4 references:

Time	0	1	2	3	4	5	6	7	8	9	10
Requests		*c*	*c*	*d*	*b*	*c*	*e*	*c*	*e*	*a*	*d*
Pages in Memory: Page *a*	• $t=0$										
Page *b*	-										
Page *c*	-										
Page *d*	• $t=-1$										
Page *e*	• $t=-2$										
Faults											

Working Set Page Replacement
Implementation

- Keep track of the last τ references
  - The pages referenced during the last τ memory accesses are the working set
  - τ is called the *window size*
- Example: Working set computation, τ = 4 references:

Time	0	1	2	3	4	5	6	7	8	9	10
Requests		*c*	*c*	*d*	*b*	*c*	*e*	*c*	*e*	*a*	*d*
Pages in Memory: Page *a*	• $t=0$	•	•	•	-	-	-	-	-	F	•
Page *b*	-	-	-	-	F	•	•	•	-	-	-
Page *c*	-	F	•	•	•	•	•	•	•	•	•
Page *d*	• $t=-1$	•	•	•	•	•	•	-	-	-	F
Page *e*	• $t=-2$	•	-	-	-	-	F	•	•	•	•
Faults		•			•		•			•	•