

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

User Program *n*

⋮

User Program 2

User Program 1

Operating System

Memory

Page Replacement Algorithms Concept

- ◆ Typically $\Sigma_i VAS_i \gg \text{Physical Memory}$
- ◆ 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 page, offset) 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)
- ◆ Hardware can tell OS when a new page is loaded into the TLB
 - Set a used bit in the page table entry
 - Increment or shift a register

Simulate the behavior of a page replacement algorithm on the trace and record the number of page faults generated
fewer faults \Rightarrow *better performance*

Optimal Page Replacement Clairvoyant replacement

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

Initial allocation

	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										
	1	b										
	2	c										
	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)
	1	b	b	b	b	b	b	b	b	b	b	b
	2	c	c	c	c	c	c	c	c	c	c	c
	3	d	d	d	d	d	(e)	e	e	e	e	e
Faults												
Time page needed next												
<div style="display: flex; justify-content: space-between;"> <i>a</i> = 7 <i>b</i> = 6 <i>c</i> = 9 <i>d</i> = 10 </div> <div style="display: flex; justify-content: space-between; margin-top: 5px;"> <i>a</i> = 15 <i>b</i> = 11 <i>c</i> = 13 <i>d</i> = 14 </div>												

Local Page Replacement FIFO replacement

- Simple to implement
 - A single pointer suffices
- Performance with 4 page frames:

0
1
2

→

Physical
Memory

Frame List

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									
	1	b									
	2	c									
	3	d									
Faults											

Local Page Replacement FIFO replacement

- Simple to implement
 - A single pointer suffices
- Performance with 4 page frames:

3
0
2

→

Physical
Memory

Frame List

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	e	e	e	e	d	
	1	b	b	b	b	b	b	a	a	a	
	2	c	c	c	c	c	c	b	b	b	
	3	d	d	d	d	d	d	d	d	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									
	1	b									
	2	c									
	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	
	1	b	b	b	b	b	b	b	b	b	b	
	2	c	c	c	c	e	e	e	e	e	d	
	3	d	d	d	d	d	d	d	d	d	c	
Faults												
Time page last used					a=2	b=4	c=1	d=3	a=7	b=8	e=5	d=3
									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
	1	b	b	b	b	b	b	b	b	b	b
	2	c	c	c	c	e	e	e	e	e	d
	3	d	d	d	d	d	d	d	d	d	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	e	e	a	b	c
					c	a	d	d	e	a	
Page to replace					c				d	e	

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
	1	b	b	b	b	b	b	b	b	b	b
	2	c	c	c	c	e	e	e	e	e	d
	3	d	d	d	d	d	d	d	d	d	c
Faults											
LRU page stack		c	a	d	b	e	b	a	b	c	d
			c	a	d	b	e	a	b	c	d
				c	a	d	e	e	a	b	c
					c	a	a	d	e	a	
Page to replace					c				d	e	

- What is the goal of a page replacement algorithm?
 - A. Make life easier for OS implementer
 - B. Reduce the number of page faults
 - C. Reduce the penalty for page faults when they occur
 - D. Minimize CPU time of algorithm

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						
1	b	b	b	b	b						
2	c	c	c	c	c						
3	d	d	d	d	d						
Faults											

Page table entries for resident pages:

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

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	e	e	e	e	e	d
1	b	b	b	b	b	b	b	b	b	b	b
2	c	c	c	c	c	c	a	a	a	a	a
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	d
1	b	0	b	1	b	0	b	1	b	1	b	0	b
1	c	0	c	0	c	1	a	1	a	1	a	0	a
1	d	0	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
 - Why?
 - Must write back contents to disk before freeing!
- 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						
1	b	b	b	b	b						
2	c	c	c	c	c						
3	d	d	d	d	d						
Faults											

Page table entries for resident pages:

10	a	11	a	00	a ^w	00	a ^w	11	a	11	a	00	a ^w
10	b	11	b	00	b ^w	10	b ^w	10	b ^w	10	b ^w	10	d
10	c	10	c	10	e	10	e	10	e	10	e	10	e
10	d	10	d	00	d	00	d	00	d	10	c	00	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
	1	b	b	b	b	b	b	b	b	b	(d)
	2	c	c	c	c	(e)	e	e	e	e	e
	3	d	d	d	d	d	d	d	d	(c)	c
Faults						•					• •

Page table entries for resident pages:

10	a	11	a	00	a ^w	00	a	11	a	11	a	00	a ^w
10	b	11	b	00	b ^w	10	b	10	b	10	b	10	b
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											
	1	b											
	2	c											
Faults													

Page Frames	0	a											
	1	b											
	2	c											
	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	(d)	d	d	(c)	c	c	(b)	b	b
	1	b	b	b	b	(a)	a	a	(d)	d	(c)	c	c
	2	c	c	c	c	c	(b)	b	b	(a)	a	a	(d)
Faults					•	•	•	•	•	•	•	•	•

Page Frames	0	a	a	a	a	a	a	a	a	a	a	a	a
	1	b	b	b	b	b	b	b	b	b	b	b	b
	2	c	c	c	c	c	c	c	c	c	c	c	c
	3	-		(d)	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
- Temporal vs. physical locality

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	•	•	•	•	•	•	•	•	•	•
	Page b	-	-	-	-	-	-	-	-	-	-
	Page c	-	-	-	-	-	-	-	-	-	-
	Page d	-	-	-	-	-	-	-	-	-	-
	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	•	•	•	•	•	•	•	•	•	•
	Page b	-	-	-	(F)	-	-	-	-	(F)	-
	Page c	-	(F)	•	•	•	•	•	•	•	•
	Page d	-	-	-	-	-	-	-	-	-	(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

25

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, $\tau = 4$ 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	•	•	•	•	•	•	•	•	•	•
	Page b	-	-	-	•	•	•	•	•	•	•
	Page c	-	•	•	•	•	•	•	•	•	•
	Page d	•	•	•	•	•	•	•	•	•	•
	Page e	•	•	•	•	•	•	•	•	•	•
Faults											

26

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, $\tau = 4$ references:
 - What if τ is too small? too large?

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	•	•	•	•	•	•	•	•	•	•
	Page b	-	-	-	•	•	•	•	•	•	•
	Page c	-	•	•	•	•	•	•	•	•	•
	Page d	•	•	•	•	•	•	•	•	•	•
	Page e	•	•	•	•	•	•	•	•	•	•
Faults											

27

Page-Fault-Frequency Page Replacement

An alternate working set computation

- Explicitly attempt to minimize page faults
 - When page fault frequency is high — *increase working set*
 - When page fault frequency is low — *decrease working set*

Algorithm:

Keep track of the rate at which faults occur
 When a fault occurs, compute the time since the last page fault
 Record the time, t_{last} , of the last page fault
 If the time between page faults is "large" then reduce the working set
 If $t_{current} - t_{last} > \tau$, then remove from memory all pages not referenced in $[t_{last}, t_{current}]$
 If the time between page faults is "small" then increase working set
 If $t_{current} - t_{last} \leq \tau$, then add faulting page to the working set

28

Page-Fault-Frequency Page Replacement

Example, window size = 2

- If $t_{current} - t_{last} > 2$, remove pages not referenced in $[t_{last}, t_{current}]$ from the working set
- If $t_{current} - t_{last} \leq 2$, just add faulting page to the working set

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	•	•	•	•	•	•	•	•	•	•
	Page b	-	-	-	•	•	•	•	•	•	•
	Page c	-	•	•	•	•	•	•	•	•	•
	Page d	•	•	•	•	•	•	•	•	•	•
	Page e	•	•	•	•	•	•	•	•	•	•
Faults											
$t_{cur} - t_{last}$											

29

Page-Fault-Frequency Page Replacement

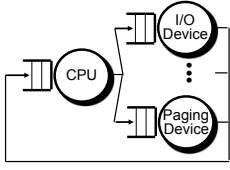
Example, window size = 2

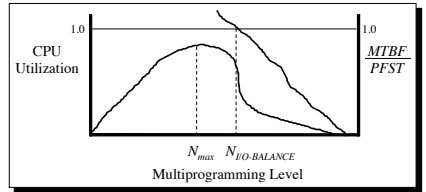
- If $t_{current} - t_{last} > 2$, remove pages not referenced in $[t_{last}, t_{current}]$ from the working set
- If $t_{current} - t_{last} \leq 2$, just add faulting page to the working set

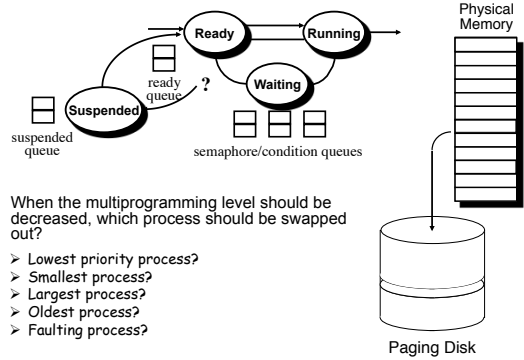
Time	0	1	2	3	4	5	6	7	8	9	10
Requests		c	c	d	b	c	e	c	e	a	d
Pages in Memory	Page a	•	•	•	•	•	•	•	•	•	•
	Page b	-	-	-	•	•	•	•	•	•	•
	Page c	-	•	•	•	•	•	•	•	•	•
	Page d	•	•	•	•	•	•	•	•	•	•
	Page e	•	•	•	•	•	•	•	•	•	•
Faults											
$t_{cur} - t_{last}$		1		3		2		3		1	

30

Load Control Fundamental tradeoff	
<ul style="list-style-type: none"> ◆ High multiprogramming level <ul style="list-style-type: none"> ➢ $MPL_{max} = \frac{\text{number of page frames}}{\text{minimum number of frames required for a process to execute}}$ ◆ Low paging overhead <ul style="list-style-type: none"> ➢ $MPL_{min} = 1$ process ◆ Issues <ul style="list-style-type: none"> ➢ What criterion should be used to determine when to increase or decrease the <i>MPL</i>? ➢ Which task should be swapped out if the <i>MPL</i> must be reduced? 	31

Load Control How not to do it: Base load control on CPU utilization	
<ul style="list-style-type: none"> ◆ Assume memory is nearly full ◆ A chain of page faults occur <ul style="list-style-type: none"> ➢ A queue of processes forms at the paging device ◆ CPU utilization falls ◆ Operating system increases <i>MPL</i> <ul style="list-style-type: none"> ➢ New processes fault, taking memory away from existing processes ◆ CPU utilization goes to 0, the OS increases the <i>MPL</i> further... 	
System is <i>thrashing</i> — spending all of its time paging	
32	

Load Control Thrashing	
<ul style="list-style-type: none"> ◆ Thrashing can be ameliorated by <i>local</i> page replacement ◆ Better criteria for load control: Adjust <i>MPL</i> so that: <ul style="list-style-type: none"> ➢ mean time between page faults (<i>MTBF</i>) = page fault service time (<i>PFST</i>) ➢ $\sum WS_i = \text{size of memory}$ 	33
	

Load Control Thrashing	
<ul style="list-style-type: none"> ◆ When the multiprogramming level should be decreased, which process should be swapped out? <ul style="list-style-type: none"> ➢ Lowest priority process? ➢ Smallest process? ➢ Largest process? ➢ Oldest process? ➢ Faulting process? 	
34	