Page Replacement

CS 537 - Introduction to Operating Systems

Paging Revisitted

- If a page is not in physical memory
  - find the page on disk
  - find a free frame
  - bring the page into memory
- What if there is no free frame in memory?

Page Replacement

- Basic idea
  - if there is a free page in memory, use it
  - if not, select a *victim* frame
  - write the victim out to disk
  - read the desired page into the now free frame
  - update page tables
  - restart the process

Page Replacement

- Main objective of a good replacement algorithm is to achieve a low *page fault rate*
  - insure that heavily used pages stay in memory
  - the replaced page should not be needed for some time
- Secondary objective is to reduce latency of a page fault
  - efficient code
  - replace pages that do not need to be written out

Reference String

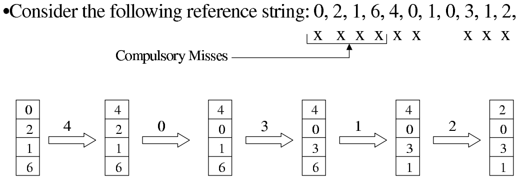
- Reference string is the sequence of pages being referenced
- If user has the following sequence of addresses
  - 123, 215, 600, 1234, 76, 96
- If the page size is 100, then the reference string is
  - 1, 2, 6, 12, 0, 0

First-In, First-Out (FIFO)

- The oldest page in physical memory is the one selected for replacement
- Very simple to implement
  - keep a list
    - victims are chosen from the tail
    - new pages in are placed at the head

FIFO

•Consider the following reference string: 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1



•Fault Rate = 9 / 12 = 0.75

FIFO Issues

- Poor replacement policy
- Evicts the oldest page in the system
  - usually a heavily used variable should be around for a long time
  - FIFO replaces the oldest page - perhaps the one with the heavily used variable
- FIFO does not consider page usage

Optimal Page Replacement

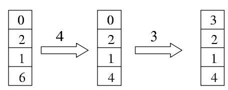
- Often called Balady’s Min
- Basic idea
  - replace the page that will not be referenced for the longest time
- This gives the lowest possible fault rate
- Impossible to implement
- Does provide a good measure for other techniques

Optimal Page Replacement

•Consider the following reference string: 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

x x x x x x

Compulsory Misses



•Fault Rate = 6 / 12 = 0.50

•With the above reference string, this is the best we can hope to do

Least Recently Used (LRU)

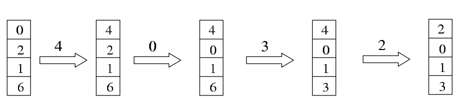
- Basic idea
  - replace the page in memory that has not been accessed for the longest time
- Optimal policy looking back in time
  - as opposed to forward in time
  - fortunately, programs tend to follow similar behavior

LRU

•Consider the following reference string: 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

x x x x x x x x

Compulsory Misses



•Fault Rate = 8 / 12 = 0.67

LRU Issues

- How to keep track of last page access?
  - requires special hardware support
- 2 major solutions
  - counters
    - hardware clock “ticks” on every memory reference
    - the page referenced is marked with this “time”
    - the page with the smallest “time” value is replaced
  - stack
    - keep a stack of references
    - on every reference to a page, move it to top of stack
    - page at bottom of stack is next one to be replaced

LRU Issues

- Both techniques just listed require additional hardware
  - remember, memory reference are very common
  - impractical to invoke software on every memory reference
- LRU is not used very often
- Instead, we will try to approximate LRU

Replacement Hardware Support

- Most system will simply provide a *reference bit* in PT for each page
- On a reference to a page, this bit is set to 1
- This bit can be cleared by the OS
- This simple hardware has lead to a variety of algorithms to approximate LRU

Sampled LRU

- Keep a *reference byte* for each page
- At set time intervals, take an interrupt and get the OS involved
  - OS reads the reference bit for each page
  - reference bit is *stuffed* into the beginning byte for page
  - all the reference bits are then cleared
- On page fault, replace the page with the smallest reference byte

Sampled LRU

	Page Table	R	Reference Bytes
0		1	01100110
1		0	10001001
2		0	10000000
3		1	01010101
4		1	00001101

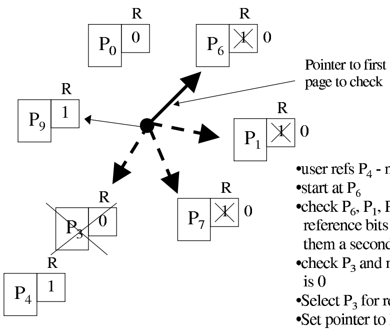
Interrupt

		R		
0		0	10110011	
1		0	01000100	
2		0	01000000	← page to replace on next page fault
3		0	10101010	
4		0	10000110	

Clock Algorithm (Second Chance)

- On page fault, search through pages
- If a page’s reference bit is set to 1
  - set its reference bit to zero and skip it (give it a second chance)
- If a page’s reference bit is set to 0
  - select this page for replacement
- Always start the search from where the last search left off

Clock Algorithm



- user refs P_4 - not currently paged
- start at P_6
- check P_6, P_1, P_7 and set their reference bits to zero (give them a second chance)
- check P_3 and notice its ref bit is 0
- Select P_3 for replacement
- Set pointer to P_9 for next search

Dirty Pages

- If a page has been written to, it is *dirty*
- Before a dirty page can be replaced it must be written to disk
- A *clean* page does not need to be written to disk
  - the copy on disk is already up-to-date
- We would rather replace an old, clean page than and old, dirty page

Modified Clock Algorithm

- Very similar to Clock Algorithm
- Instead of 2 states (ref'd and not ref'd) we will have 4 states
  - (0, 0) - not referenced clean
  - (0, 1) - not referenced dirty
  - (1, 0) - referenced but clean
  - (1, 1) - referenced and dirty
- Order of preference for replacement goes in the order listed above

Modified Clock Algorithm

- Add a second bit to PT - *dirty bit*
- Hardware sets this bit on write to a page
- OS can clear this bit
- Now just do clock algorithm and look for best page to replace
- This method may require multiple passes through the list

Page Buffering

- It is expensive to wait for a dirty page to be written out
- To get process started quickly, always keep a pool of free frames (buffers)
- On a page fault
  - select a page to replace
  - write new page into a frame in the free pool
  - mark page table
  - restart the process
  - now write the dirty page out to disk
  - place frame holding replaced page in the free pool