LECTURE 2.5
PAGE REPLACEMENT AND ACCESS TIME

COP4600

Dr. Matthew Gerber

3/21/2015

PAGE REPLACEMENT ALGORITHMS

Page Replacement

- In virtual memory, we constantly have the following case:
  - We need to swap a page from disk into main memory, and
  - There are no free frames remaining in main memory – whether in our program's allocated frames, or in main memory in its entirety
- In this case, we need to choose a *victim frame* to swap out first
- The choice of the victim frame is made by a *page replacement algorithm*

FIFO Page Replacement (9.4.2)

- Replace the page that was loaded earliest
- So with three frames and references ordered as follows:

  0, 1, 1, 1, 1, 0, 3, 0, 2

  …frame 0 is replaced when frame 2 is referenced
- Simple to implement
- Poor performance to the point of often being outperformed by random replacement
- A particular issue – consider references ordered as follows:

  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- How many page faults with four frames? How about with five?

Belady's Anomaly and Stack Algorithms

- For some case with some page replacement algorithms, the number of page faults can *increase* when the number of available frames rises
- Algorithms with this problem are said to suffer from Belady's Anomaly
- Page replacement algorithms with this problem are generally considered to be poor algorithms
- Algorithms are called *stack algorithms* if the set of pages with fewer frames available is always a strict subset of the set of pages with more frames available
- Stack algorithms cannot suffer from Belady's anomaly

Optimal Page Replacement (9.4.3)

- The optimal algorithm can be proven to be the best possible page replacement algorithm
  - (or at least *a* best possible page replacement algorithm, and that's what matters)
- Its definition is simple:

  Replace the page that will not be used for the longest period of time.

- Advantage it is that it is in fact an optimal algorithm
- Disadvantage is that it cannot actually be implemented in the general case
  - If I can predict all memory accesses, what else can I predict?

LRU Page Replacement (9.4.4)

- If we can't choose which page won't be used *in* the longest, we can instead replace that hasn't been used *for* the longest
- So with three frames and references ordered as follows:

  0, 1, 1, 1, 1, 0, 3, 0, 2

  …frame 1 is replaced when frame 2 is referenced
- Is a stack algorithm
- Generally has good performance
- Approximations are often used in practice

LRU Implementations

- **Counter implementation**: Associate a CPU clock time with each page table entry
- **Stack implementation**: Maintain a list of all referenced pages in a doubly-linked list
- Either of these absolutely requires hardware to be effective
- In fact, even approximations of LRU require hardware

LRU Approximations (9.4.5)

- **Second Chance**
  - Each page in the table is given a *reference bit*
  - The reference bit is set to 1 whenever the page is referenced
  - Select a page for replacement as in FIFO, but if the reference bit of the selected page is 1, set its reference bit to 0 and pass over it
  - Degenerates to FIFO if all bits are 1
  - Can be enhanced to take into account page modification (Why?)
- **Additional Reference Bits**
  - Like Second Chance, each page has a reference bit
  - The reference bit is treated as the top bit of a reference *integer*
  - Periodically, the reference integer is right-shifted
  - When a page needs to be replaced, select a page with a minimum reference integer

Counting Algorithms (9.4.6)

- Keep a counter of the number of references that have been made to each page
- **Least Frequently Used**
  - So with three frames and references ordered as follows:

    0, 1, 1, 1, 1, 0, 3, 0, 2

    …frame 3 is replaced when frame 2 is referenced
  - Expensive implementation, mediocre performance
  - Only useful for specific applications
- **Most Frequently Used**
  - So with three frames and references ordered as follows:

    0, 1, 1, 1, 1, 0, 3, 0, 2

    …frame 1 is replaced when frame 2 is referenced
  - Theory is that the page with the smallest count was just swapped in
  - Rarely actually useful

MEMORY ACCESS TIME

Memory Access Time with TLBs

- Recall that page tables are stored in memory
- *Translation lookaside buffers* are page table caches stored on the processor, and take less time to access
- We assume that TLBs take effectively no time to access themselves
- Memory access time for a TLB hit is the same as the underlying physical memory access time
- Memory access time for a TLB miss is *twice* the physical memory access time
- Like any other caches, TLBs have *hit rates* and *miss rates*
- So with physical memory access time m and TLB hit rate r, *effective* memory access time is $mr + 2m(1 - r)$

Memory Access with TLBs - Example

Physical Access Time	TLB Hit Rate	Effective Access Time	Paging Slowdown
100ns	50%		
100ns	80%		
100ns	90%		
100ns	99%		
100ns	99.9%		
100ns	99.99%		
100ns	99.999%		
100ns	99.9999%		
100ns	99.99999%		

Memory Access with TLBs - Example

Physical Access Time	TLB Hit Rate	Effective Access Time	Paging Slowdown
100ns	50%	150ns	50.00000%
100ns	80%	120ns	20.00000%
100ns	90%	110ns	10.00000%
100ns	99%	101ns	1.00000%
100ns	99.9%	100.1ns	0.10000%
100ns	99.99%	100.01ns	0.01000%
100ns	99.999%	100.001ns	0.00100%
100ns	99.9999%	100.0001ns	0.00010%
100ns	99.99999%	100.00001ns	0.00001%

Access Time with Virtual Memory

- Under virtual memory, when we need to access a page that isn't loaded into a frame, we have to go get it from disk
- For purposes of calculating the time involved, we can think of main memory as one big cache for virtual memory, with the miss rate being the page fault rate
- So with page fault rate *f*, memory access time *m* and disk swap time *d*, effective access time is (1 – f)*m* + *fd*

Access Time with Virtual Memory - Example

Mem. Acc. Time	Disk Swap Time	Page Fault Rate	Eff. Access Time	VM Slowdown
100ns	10,000,000ns	10.00000%		
100ns	10,000,000ns	1.00000%		
100ns	10,000,000ns	0.10000%		
100ns	10,000,000ns	0.01000%		
100ns	10,000,000ns	0.00100%		
100ns	10,000,000ns	0.00010%		
100ns	10,000,000ns	0.00001%		
100ns	200,000ns	10.00000%		
100ns	200,000ns	1.00000%		
100ns	200,000ns	0.10000%		
100ns	200,000ns	0.01000%		
100ns	200,000ns	0.00100%		
100ns	200,000ns	0.00010%		
100ns	200,000ns	0.00001%		

Access Time with Virtual Memory - Example

Mem. Acc. Time	Disk Swap Time	Page Fault Rate	Eff. Access Time	VM Slowdown
100ns	10,000,000ns	10.00000%	1,000,090 ns	999990.00000%
100ns	10,000,000ns	1.00000%	100,099 ns	99999.00000%
100ns	10,000,000ns	0.10000%	10,099.9 ns	9999.90000%
100ns	10,000,000ns	0.01000%	1,099.99 ns	999.99000%
100ns	10,000,000ns	0.00100%	199.999 ns	99.99900%
100ns	10,000,000ns	0.00010%	109.9999 ns	9.99990%
100ns	10,000,000ns	0.00001%	100.99999ns	0.99999%
100ns	200,000ns	10.00000%	20,090 ns	19990.00000%
100ns	200,000ns	1.00000%	2,099 ns	1999.00000%
100ns	200,000ns	0.10000%	299.9 ns	199.90000%
100ns	200,000ns	0.01000%	119.99 ns	19.99000%
100ns	200,000ns	0.00100%	101.999 ns	1.99900%
100ns	200,000ns	0.00010%	100.1999 ns	0.19990%
100ns	200,000ns	0.00001%	100.01999ns	0.01999%

Other Memory Access Time Factors

- We've considered:
  - The TLB
  - Virtual Memory
- Other factors include:
  - Caches
  - Addressing Modes
  - Specialized Operations
  - NUMA

NEXT TIME: FRAME ALLOCATION AND EXAMPLES