Part1

Monday, May 6, 2019 12:02 PM

Least Recently Used
- Assume pages used recently will be used again soon
    - Throw out page that has been unused for longest time
- Must keep a linked list of pages
    - Most recently used at front, least at rear
    - Update this list every memory reference!
        - This can be somewhat slow: hardware has to update a linked list on every reference!
- Alternatively, keep counter in each page table entry
    - Global counter increments with each CPU cycle
    - Copy global counter to PTE counter on a reference to the page
    - For replacement, evict page wit lowest counter value
- LRU stack, put most recently used onto top of "stack", look through it and put last used to top of stack

Simulating LRU in software
- Few computers have the necessary hardware to implement full LRU
    - Linked list method impractical in hardware
    - Counter-based method could be done, but it's slow to find the desired page
- Approximate LRU with Not Frequently used NFU algorithm
    - At each clock interrupt, scan through page table
    - If R=1 for a page, add one to its counter value
    - On replacement pick the page with the lowest counter value

Aging replacement algorithm
- Reduce counter values over time
    - Divide by two every clock cycle, use right shift
    - More weight given to more recent references
- Select page to be evicted by finding the lowest counter value
- Algorithm is:
    - Every clock tick, shift all counters right by 1 bit
    - On reference, set leftmost bit of a counter, can be done by copying the reference bit to the counter at the clock tick

Working set
- Demand paging: bring a page into memory when it's requested by the process
- How many pages are needed?
    - Could be all of them, but not likely
    - Instead, processes reference a small set of pages at any given time-locality of reference
    - Set of pages can be different for different processes or even different times in the running of a single process
- Set of pages used by a process in a given interval of time is called the working set
    - If entire working set is in memory, no page faults!
    - If insufficient space for working set, thrashing may occur
    - Goal: keep most of working set in memory to minimize the number of page faults suffered by a process

How big is the working set?

- Working set is the set of pages used by the k most recent memory references
- W(k,t) is the size of the working set at time t
- Working set may change over time
    - Size of working set can change over time as well…

Modeling page replacement algorithms
- Goal: provide quantitative analysis or simulation showing which algorithms do better
    - Workload page reference string is important: different strings may favor different algorithms
    - Compare algorithms to one another
    - Model parameters within an algorithm
        - Number of available physical pages
        - Number of bits for aging

How is modeling done?
- Generate a list of references
    - Artificial (made up)
    - Trace a real workload (set of processes)
- Use an array (or other structure) to track the pages in physicla memory at any given time
    - May keep other information per page to help simulate the algorithm (modification time, time when paged in, etc.)
- Run through references, applying the replacement algorithm

Belady's anomaly
- Reduce the number of page faults by supplying more memory
    - Use previous reference string and FIFO algorithm
    - Add another page to physical memory (total 4 pages)
- More page faults, not fewer!
    - Adding more pages shouldn't result in worse performance
- Motivated the study of paging algorithms

Modeling more replacement algorithm
- Paging system characterized by:
    - Reference string of executing process
    - Page replacement algorithm
    - Number of page frames available in physical memory (m)
- Model this by keeping track of all n pages refences in array M
    - Top part of M has m pages in memory
    - Bottom part of M has n-m pages stored in disk
- Page replacement occurs when page moves from top to bottom

LRU example
- Model LRU replacement with
    - 8 unique references in the reference string
    - 4 pages of physical memory
- Array state over time shown below
- LRU treats list of pages like a stack

Stack Algorithms
- LRU is an example of a stack algorithm
- For stack algorithms
    - Any page in memory with m physical pages is also in memory with m+1 physical pages
    - Increasing memory size is guaranteed to reduce (or at least not increase) the number of

page faults
- Stack algorithms do not suffer from Belady's anomaly
- Distance of a reference <=> position of the page in the stack before the reference was made
    - Distance is infinity if no reference had been made before
    - Distance depends on reference string and paging algorithm: might be different for LRU and optimal (both stack algorithms)

Predicting page fault rates using distance
- Distance can be used to predict page fault rates
- Make a single pass over the reference string to generate the distance string on the fly
- Keep an array of counts
    - Entry j counts the number of times distance j occurs in the distance string
- The number of page faults for a memory of size m is the sum of counts for j > m
    - This can be done in a single pass
    - Makes for fast simulations of page replacement algorithms
- This is why virtual memory theorists like stack algorithms

Local vs. global allocation policies
- What is the pool of pages eligible to be replaced
    - Pages belonging to the process needing a new page
    - All pages in the system
- Local allocation: replace a page from this process
    - May be more fair: penalize processes that replace many pages
    - Can lead to poor performance: some processes need more pages than others
- Global allocation: replace a page from any process

Control overall page fault rate => **THRASHING**
- Despite good designs, system may still thrash
- Most (or all) processes have high page fault rate
    - Some processes need more memory, …
    - But no processes need less memory (and could give some up)
- Problem: no way to reduce page fault rate
- Solution: Reduce number of processes competing for memory
    - Swap one of more to disk, divide up pages they held
    - Reconsider degree of multiprogramming

Part 2

Wednesday, May 8, 2019 12:20 PM

How big should a page be?
- Smaller pages have advantages
  - Less internal fragmentation
  - Better fit for various data structures, code sections
  - Less unused physical memory (some pages have 20 useful bytes and the rest isn't needed currently)
- Karger papges are better because
  - Less overhead to keep track of them
    - Smaller page tables
    - TLB can point to more memory (same number of pages, but more memory per page)
    - Faster paging algorithms (fewer table entries to look through)
  - More efficient to transfer larger pages to and from disk

Separate I and D address spaces
- One user address space for both data and code
  - Simpler
  - Code/data separation harder to enforce
  - More address space?
- One address space for data, another for code
  - Code and date separated
  - More complex in hardware
  - Less flexible
  - CPU must handle instructions and date differently
- FreeBSD does the former

Sharing pages
- Processes can share pages
  - Entries in page table point to the same physical page frame
  - Easier to do with code: no problems with modification
- Virtual addresses in different processes can be
  - The same: easier to exchange pointers, keep data structures consistent
  - Different: may be easier to actually implement
    - Not a problem if there are only a few shared regions
    - Can be very difficult if many processes share regions with each other

Shared libraries
- Many libraries are used by multiple programs
- Only want to keep a single copy in memory
- Two possible approaches
  - Fixed address in memory
    - No need for code to be relocatable
    - How can libraries be placed?
  - Per-process address in memory
    - More flexible: no central arbiter of addresses
    - Code has to be relocatable…

Memory mapped files

- Extension of shared libraries
  - File blocks mapped directly into a process's address space
  - Process can access file data just like memory
- Advantages
  - Efficient: no need for read() and write() calls
    - OS manages 'paging' blocks in and out
  - Easy to program: no buffer management
    - Can handle very large files, too
- Disadvantages
  - Added burden for the OS: may not manage as well as program could
  - Difficult to specify order in which writes are flushed to disk
    - OS writes pages back in unpredictable order
  - Shared files can cause issues
    - Might be mapped to different addresses in different processes
    - Write ordering matter even more!
- Pointers don't make sense in this shared space

When are dirty pages written to disk
- On demand (when they're replaced)
  - Fewest writes to disk
  - Slower: replacement takes twice as long (must wait for disk write and disk read
- Periodically (in the background)
  - Background process scans through page tables, writes out dirty pages that are pretty old
- Background process also keeps a list of pages ready for replacement
  - Page faults handled faster: no need to find space on demand
  - Cleaner may use the same structures discussed earlier (clock, etc.)

Implementation issues
- Four times when OS involved with paging
- Process creation
  - Determine program size
  - Create page table
- During process execution
  - Reset the MMU for anew process
  - Flush the TLB (or reload it from saved state)
- Page fault time
  - Determine virtual address causing fault
  - Swap target page out, needed page in
- Process termination time
  - Release page table
  - Return pages to the free pool

How is a page fault handled?
- Hardware causes a page fault
- General registers saved (as on every exception)
- OS determines which virtual page needed
  - Actual fault address in a special register
  - Address of fualting instruction in register
    - Page fault was in fetching instruction, or
    - Page

Part 3

Friday, May 10, 2019 12:03 PM

Storing pages on disk

- Pages removed from memory are store on disk
- Where are they placed?
  - Static swap area: easier tp code, less flexible
  - Dynamically allocated space: more flexible, harder to locate a page
    - Dynamic placement often uses a special file (managed by the file system) to hold pages
- Need to keep track of which pages are where within the on-disk storage

Separating policy and mechanism

- Mechanism for page replacement has to be in kernel
  - Modifying page tables
  - Reading and writing page table entries
- Policy for deciding which pages to replace could be in user space
  - More flexibility

Segmentation

- Different units in a single virtual address space
  - Each unit can grow
  - How can they be kept apart?
- Solution: segmentation
  - Give each unit its own address space
- Segmentation registers

Using segments

- Each region of the process has its own segment
- Each segment can start at 0
  - Addresses within the segment relative to the segment start
- Virtual addresses are <segment #, offset within segment>

Memory management in x86-32

- Memory composed of segments
  - Segment pointed to by segment descriptor
  - Segment selector used by identify descriptor
- Segment descriptor describes segment
  - Base virtual address
  - Size
  - Protection
  - Code / data