Week 6

Least Recently Used (LRU)

- Assumed pages used recently, likely to be used again
- Must keep a linked list of pages
  - most recently used at front
  - Update list every memory reference
    - Slow!
- Alternatively, keep ccounter in each page table entry
  - Global counter incremebts with each CPU cycle
  - Copy global counter to PTE counter on a reference to the page
  - For replacement, evict page with lowest counter value

Simulating LRU in software

- Few computers have necessary hardware to implement full LRU
  - LInked-list method impractical in hardware
  - Counter based method could be done, but it's slow to find desired page
- Approximate LRU with not frequently used (NFU)
  - At clock interrupt scan through page table
  - If R=1 for a page add one to its counter value
  - On replacement pick page with lowest counter value
- Problem: No notion of age. Pages with high cocunter values will tend to keep them.

Aging Replacement Algo

- Reduce counter values over time
  - Divide by two every cycle (use right shift)
  - MOre weight given to more recent references
- Select page by finding the lowest counter (to evict)
- Alg
  - Every clock tick, shift all counters right by 1 bit
  - On ref, set leftmost bit

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 memmory, no page faults!
  - If insufficent space for working set, thrashing may occur

HOw big is working set?

- Working set working set is the set of pages used by the k most recent memory references

Algs summmary

- Optimal : Best impossible
- NRU : Crude
- FIFO : might throw out useful
- Second : change better than fifo
- Clock : better than second chance
- LRU : great, but hard to implement
- NFU : poor approx of lru
- Aging : good approx to LRU, inneficient
- Working set : somewhat expensive
- WSClock : implementatble verison of working set

Modeling page replacemtn algs

- Goal: QUantitative analysis showing which algs do better
  - Workload is important: different strings may favor diff algs
  - Show tradeoffs between algs
- Compare algs to one another
- MOdel params with an algorithm

Beladys anomaly : More mem doesn't necessaryily make things better

- Adding more pages shouldn't resut in worse performance! But it might

M(m, r) is a subset of M(m+1, r)

If I have mroe memory, then the things that are in the smaller mem are in the bigger mem. Adding more memory can't hurt me.

This is not true of FIFO or NFU or second chance (probably fine, good to ignore)

DOES NOT HAPPEN WITH LRU

Stack Alogirthms

- LRU is a stack algorithm
- Don't suffer from Baladys anomaly
- Any page in mem with m physical pages is also in mem with m + 1 physical pages
- INcrease mem guaranteed to reduce (or keep same) the num of page faults

predicting page fault rates using distance

- Make isngle pass over ref string to generate the distance string on the fly
- Keep and array of counts
  - Entry j counts the number of times distance j occurs in the dist string
- number of page faults for a memory of size m is the sum of counts for j > m

  - Can be done in single pass
  - Makes for fast simulations of page replacement algorithm

Local vs Global

- What is pool eligible to be replaced?
  - Pages belong to process?
  - Pages global
- Local alloc: replace a page form this proess
  - May be fair, penalize processes thatreplace many pages
  - Can lead to poor performance some processes need more pages than others
- Global allc replace page from any proc

How many pages to give? Working set

- Local alloc may be more fair
  - Not penalize other procs for high page fault rate
- Global alloc better for overall sys performance
  - Take page frames from processes that don't ned them as much
  - Reduce the overall page fault rate (even though rate for a single process may go up)

Control overall page fault rate

- Despite good desings, system may still thrash
- Most procs have high page fault rate
  - Some need more mem
  - but no procs need less mem (andcould give some up)
- Problem no way to reduce page fault rate
- Sol:
  - Reduce num procs ocmpeting for mem
    - Swap one or more to disk, divide up pages they held
    - Reconsider degree of multiprogramming

HOw big shold a page be?

- Smaller pages have advantages
  - Less internal frag
  - Better fit for various data structs, code sections
  - Less unused physical mem (some pages have 20 useful bytes and the rest inste needed)
- Bigger are better because
  - Less overhaed
    - Smaller page tables
    - TLB can point to more mem (same number of pages, more per page0)
    - Faster paging algs
  - MOre efficient to transfer to and from disk

Separate I and D spaces

- ONe user address space for both data and code
  - Simpler
  - Code/data separation hardder to enforce
  - More address space
- One address space for data another for code
  - Code and data separate
  - More complex in harddware
  - Lesds flexible

Sharing pages

- Processes can share pages
  - Page table pointto same page frame
  - Easier to do with code, no prob with modification
- Virtual address in dif proc can be
  - The same, easier to exchange pointers, keep data strucutrs consistent
  - Different may be easier to actually implement
    - Not a probel if there are only a few shared regions

Shared LIbs

- Many libs used by multiple programs
- Only want a single copy in mem
- Two possible approaches
  - fixed address in mem
    - No needfor code to be relocatable
    - HOw can libraries be places?
  - Per proc address in mem * More flexible no central arbiter of addresses * code has to be relocatable

mem mapped files

Extension of shared libs * File blocks mapped directly into a procs address space * proc can access file data just like emory * Advantages * Efficient, no need for read write() calls * OS manages paging in/out * Easy to program no buffer management * Cna handle 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 flusehd to disk * OS writes pages back in unpredictable order * Shared files can cause issues * Might be mapped to different addresses in different procs * Write order matters even more

Locking Pages in memory

- Virtual mem and IO ocassioinally interact
- Lock pages in memory

Sotring pages on disk

- Pages removed from memory are stroed on disk
- Swap file

Separating policy and mechanism

- Mechanism for replacement has to be in kernel
- Policy for deciding which pages to replace couldbe in user space

Segments

- Virtual addreses are

Paging VS Segmentation

I have semaphores

I can build monitors out of semaphores - Piece of code with semaphore to get in - Semaphore to up when leave - Condition variables, counter semaphores in monitor

Can I make messages out of monitors? - Message is queue - When send, go into monitor put in queue - When recv, go into monitor and pull from queue

Semaphores from messages - Server - Two mess, up and down - Variable 0 and 1 - Send message to server: Down - If semaphore 1, recv good to go - Else, waits and you wait until someone says up

Inverted page table * Can easily have large address space * Bad: Wired physical pages to virtual page that holds them, no shared mem * Speedy

Benefit of two level paging

- Can't afford to have single element page table
  - Would be way too big to fit in memory
- Virtualize page table
  - Works because we don't have all the page tables in memory
- Virtual memory works because we don't have everyting in memory

Paratooptimal: Given all constraints, best choice. Can make one thingbetter, but would make something else worse

SLAB = NO little tiny holes