Slide 1

Administrivia

- Reminder: Homework 4 due Wednesday.
- Reminder: Quiz 3 Wednesday.

Slide 2

Minute Essay From Last Lecture

- Most people said something about all those page-fault interrupts slowing things down. Exactly.
- One person did wonder how you could achieve fewer page faults. “Exactly” again!

Slide 3

Page Replacement Algorithms — Review/Recap

- Processing a page fault can involve finding a free page frame. Would be easy if the current set of processes aren't taking up all of main memory, but what if they are? Must steal a page frame from someone. How to choose one?
- Several ways to make choice (as with CPU scheduling) — "page replacement algorithms".
- "Good" algorithms are those that result in few page faults. (What happens if there are many page faults?)
- Choice usually constrained by what MMU provides (though that is influenced by what would help o/s designers).
- Many choices . . .

Slide 4

"Optimal" Algorithm

- Idea — if we know for each page when it will next be referenced, choose the one for which that's the furthest away.
- Theoretically optimal, though can't be implemented.
- Useful as a standard of comparison — run program once on simulator to collect data on page references, again to determine performance with this "algorithm". (Not clear that this is really possible with multiprogramming, i.e., more than one process active.)

Slide 5

Sidebar: Page Table Entries, Revisited

- Recall — many architectures' page table entries contain bits called "R (referenced) bit" and "M (modified) bit".
- Idea is that these bits are set by hardware on any memory reference, and cleared by software (o/s) in some way that's useful.

Slide 6

"Not Recently Used" Algorithm

- Idea — choose a page that hasn't been referenced/modified recently, hoping it won't be referenced again soon.
- Implementation — use page table's R and M bits, group pages into four classes:
  - $R = 0, M = 0$.
  - $R = 0, M = 1$.
  - $R = 1, M = 0$.
  - $R = 1, M = 1$.

  Choose page to replace at random from first non-empty class.
- How good is this? Easy to understand, reasonably efficient to implement, often gives adequate performance.

Slide 7

"First In, First Out" Algorithm

- Idea — remove page that's been there the longest.
- Implementation — keep a FIFO queue of pages in memory.
- How good is this? Easy to understand and implement, no MMU support needed, but could be very non-optimal.

Slide 8

"Second Chance" Algorithm

- Idea — modify FIFO algorithm so it only removes the oldest page if it looks inactive.
- Implementation — use page table's R and M bits, also keep FIFO queue. Choose page from head of FIFO queue, *but* if its R bit is set, just clear R bit and put page back on queue.
- Variant — "clock" algorithm (same idea, keeps pages in a circular queue).
- How good is this? Easy to understand and implement, probably better than FIFO.

Slide 9

"Least Recently Used" (LRU) Algorithm

- Idea — replace least-recently-used page, on the theory that pages heavily used in the recent past will be heavily used in the near future. (Usually true).
- Implementation:
  - Full implementation requires keeping list of pages ordered by time of reference. Must update this list on every memory reference.
  - Only practical with special hardware — e.g.:
    Build 64-bit counter C, incremented after each instruction (or cycle).
    On every memory reference, store C's value in PTE.
    To find LRU page, scan page table for smallest stored value of C.
    (Is 64 bits enough?)
- How good is this? Could be pretty good, but requires hardware we probably won't have.

Slide 10

"Not Frequently Used" (NFU) Algorithm

- Idea — simulate LRU in software.
- Implementation:
  - Define a counter for each PTE. Periodically ("every clock-tick interrupt") update counter for every PTE with R bit set.
  - Choose page with smallest counter.
- How good is this? Reasonable to implement, could be good, but counters track full history, which might not be a good predictor.

Slide 11

"Aging" Algorithm

- Idea — simulate LRU in software (like NFU), but give more weight to recent history.
- Implementation similar to NFU, but increment counters by shifting right and adding to *leftmost* bit — in effect, divide previous count by 2 and add bit for recent references.
- How good is this? Pretty good approximation to LRU, though a little crude, and limited by size of counter.

Slide 12

Sidebar: Working Sets

- Most programs exhibit "locality of reference", so a process usually isn't using all its pages.
- A process's "working set" is the pages it's using. Changes over time, with size a function of time and also of how far back we look.

Slide 13

"Working Set" Algorithm

- Idea — steal / replace page not in recent working set. Define working set by looking back τ time units (w.r.t. process's virtual time). Value of τ is a tuning parameter, to be set by o/s designer or sysadmin.
- Implementation:
  - For each entry in page table, keep track of time of last reference.
  - When we need to choose a page to replace, scan through page table and for each entry:
    If $R = 1$, update time of last reference.
    Compute time elapsed since last use. If more than τ, page can be replaced.
  - If we don't find a page to replace that way, pick the one with oldest time of last use. If a tie, pick at random.
- How good is this? Good, but could be slow.

Slide 14

"WSClock" Algorithm

- Idea — efficient-to-implement variation of previous algorithm, based on circular list of pages-in-memory for process. (Carr and Hennessy.)
- Implementation — like previous algorithm, but when we need to pick a page to replace, go around the circle and:
  - If $R = 1$, update time of last use. Compute time since last use.
  - If time since last use is more than τ and $M = 1$, schedule I/O to write this page out (so it can maybe be replaced next time — M bit will be cleared when I/O completes). No need to block yet, though.
  - If time since last use is more than τ and $M = 0$, replace this page.

  The idea is to go around the circle until we find a page to replace, then stop. (If we get all the way around the circle, we'll pick some page with $M = 0$.)
- How good is this? Makes good choices, practical to implement, apparently widely used in practice.

Slide 15

Modeling Page Replacement Algorithms

- Intuitively obvious that more memory leads to fewer page faults, right? Not always!
- Counterexample — "Belady's anomaly", sparked interest in modeling page replacement algorithms.
- Modeling based on simplified version of reality — one process only, known inputs. Can then record "reference string" of pages referenced.
- Given reference string, p.r.a., and number of page frames, we can calculate number of page faults.
- How is this useful? can compare different algorithms, and also determine if a given algorithm is a "stack algorithm" (more memory means fewer page faults).

Slide 16

Page Replacement Algorithms — Recap

- Nice summary in textbook (table at end of section 3.4).
- Tanenbaum says best choices are aging, WSClock.
- Now move on to other issues to consider . . . (To be continued.)

Slide 17

Minute Essay

- Another story from long ago: Once upon a time, a mainframe computer was running very slowly. The sysadmins were puzzled, until one of them noticed that one of the disk drives seemed to be very busy and asked “which disk are you using for paging?” The answer made everyone say “aha!” What was wrong (to make the system so slow)?
- Does anything like this still happen?

Slide 18

Minute Essay Answer

- The disk being used for paging was the one that was very busy. So, mostly likely the system was spending so much time paging (“thrashing”) that it wasn’t able to get anything else done. Usually this means that the system isn’t able to keep up with active processes’ demand for memory.
- This can indeed still be a problem — a few years ago, with the Xenas trying to run both Eclipse and a Lewis simulation, and last year with the Xenas attempting to run a background program that asked for memory than its author intended.