Slide 1

Administrivia

- Homework 3 will be on Web by Friday. Due next Friday.

Slide 2

Page Fault Processing and Page Replacement Algorithms

- 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.
- Choice usually constrained by what MMU provides (though that is influenced by what would help o/s designers).

Slide 3

“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.)

Slide 4

“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 5

“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 6

“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 7

“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.
- How good is this? Could be pretty good, but requires hardware we probably won’t have.

Slide 8

“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.

“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.

Intermezzo — Demand Paging, Prepaging, and Working Sets

- The purest form of paging is “demand paging” — processes are started with no pages in memory, and pages are loaded into memory on demand only.
- An alternative is “prepaging” — try to load pages in advance of demand. How?
- 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 11

Minute Essay

- If process p0's working set totals 128M, and process p1's working set totals 140M, and you have 256M of real memory, what will happen if you run p0 and p1 at the same time? (I.e., do they run well together?)

Slide 12

Minute Essay Answer

- Since the combined working sets of the two processes exceeds the size of main memory, the likely result of trying to run them at the same time is lots of paging, and thus poor performance. (We might have this problem even with slightly smaller working sets, since some of real memory needs to be reserved for the operating system itself.)