Slide 1

Administrivia

- (None.)

Slide 2

Memory References — Hardware vs. Software

- Hardware (MMU) steps:
  - Does cache contain data for (virtual) address? if so, done.
  - Does TLB contain matching page table entry? if so, generate physical address and send to memory bus.
  - Does page table entry (in memory) say page is present? if so, put PTE in TLB and as above.
  - If page table entry says page not present, generate page fault interrupt.

Slide 3

Memory References — Hardware vs. Software

- Page-fault interrupt handler steps:
  - Is page on disk or invalid (based on entry in process table, or other o/s data structure)? if invalid, terminate process.
  - Is there a free page frame? If not, choose one to steal. If modified, write current contents to disk (do other work while waiting), then modify PTE for page.
  - Read page contents in from disk (do other work while waiting), or zero out new page, then modify PTE.
  - Go back to original process to retry instruction that started this.

Slide 4

Memory References — Hardware vs. Software

- Some things defined by hardware architecture — structure of page table entries, how MMU finds page table.
- A very common feature — each entry has R ("referenced") and M ("modified") bits.

  Set by MMU on every memory reference.

  Cleared by operating system "when appropriate" — M bit when page is replaced or written to disk, R bit when? Often want to do this periodically. A good choice is "on clock interrupts" (generated at intervals by hardware, gives o/s regular opportunities to do many things — more in chapter 5).

Slide 5

Finding A Free Frame — 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 6

"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 7

"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 8

"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 9

"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 10

"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.
    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 11

"Not Frequently Used" (NFU) Algorithm

- Idea — simulate LRU in software.
- Implementation:
  - Define a counter for each PTE. On every clock 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 12

"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 13

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 14

"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 bit is set, 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 15

"WSClock" Algorithm

- Idea — efficient-to-implement variation of previous algorithm, based on circular list of pages-in-memory for process.
- 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 16

Review — Page Replacement Algorithms

- Nice summary in textbook, table on p. 228.
- Author says best choices are aging, WSClock.

Slide 17

Minute Essay

- I plan one more lecture on memory management. Anything you'd particularly like to hear more about?