Slide 1

Administrivia

- (None.)

Slide 2

Paging — Recap

- Recall basic ideas of paging:
  - Divide address spaces into pages, memory into page frames; allocate memory page (frame) by page (frame).
  - Use page tables (one per process) to keep track of things.
  - Use MMU to translate program (virtual) addresses into memory locations — using page table for current process. Generate “page fault” interrupt if impossible.
- Last time we talked about how to resolve some potential problems (unmanageably large page tables, performance problems).
- Still to discuss:
  - How to keep track of pages on disk.
  - How to choose a page frame to “steal”.

Slide 3

Keeping Track of Pages on Disk

- To implement virtual memory, need space on disk to keep pages not in main memory. Reserve part of disk for this purpose ("swap space"); (conceptually) divide it into page-sized chunks. How to keep track of which pages are where?
- One approach — give each process a contiguous piece of swap space. Advantages/disadvantages?
- Another approach — assign chunks of swap space individually. Advantages/disadvantages?
- Either way — processes must know where "their" pages are (via page table and some other data structure), operating system must know where free slots are (in memory and in swap space).

Slide 4

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. (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 5

Sidebar: Processing 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 6

Sidebar: Page Tables, Revisited

- What do we need for each entry in a page table?
  - Page frame number.
  - Present/absent bit (was valid/invalid).
  - Protection bit(s).
  - "Modified since last page-in?" bit.
  - "Referenced recently?" bit.
  - "Okay to cache?" bit.
- Goal is to keep this somewhat minimal — mostly data the MMU needs.

  If present/absent bit says "absent", two cases — error and "page not in memory right now" — MMU should generate "page fault" interrupt, let page fault interrupt handler decide.

Slide 7

"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 8

"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 9

"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 10

"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 11

"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 12

"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 13

"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 14

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 15

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 — only a few years ago, with the Xenas trying to run both Eclipse and a Lewis simulation.