Slide 1

Administrivia

- Midterm graded. Scores generally not bad, though some were not great.
- Grade summaries sent by e-mail. They don't include the programming problems for Homeworks 2 and 3 but should give you some idea where you stand in the course. I'll probably send an update after grading those problems.
- Next two homeworks on the Web. Homework 4 due a week from today, Homework 5 the following Monday. (Details of the programming problem for the latter still in work.)

Slide 2

Minute Essay From Last Lecture

- Only one person mentioned noticing anything strange. Could it be that most of you always exited the program with control-C??
- More than one person mentioned not even testing what happens on bad commands (!?). Surely you do better at testing "corner cases" in other courses requiring programming?

Slide 3

Finding A Free Frame — Recap/Review

- 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 (no surprise, right?) . . .

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 (to 1) by hardware and cleared by software (O/S) in some way that's useful.
- R bit set on any memory reference into page. Typically cleared by O/S periodically (on "clock ticks"). Allows tracking which pages have been used recently.
- M bit set on any write/store into page, cleared when page is written out to disk. If off, means that if we need this page's page frame, no need to write contents out to disk (since presumably we have a copy from a previous write).

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 uses page table's R and M bits, grouping 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, but keep 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. (Is 64 bits enough?)
    - To find LRU page, scan page table for smallest stored value of C.
- How good is this? Results could be 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.
  - Clear R bits periodically.
  - 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 no page to replace found 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 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.

  Idea is to go around the circle until a page to replace is found, then stop. (If none found, just pick some page with $M = 0$.)
- How good is this? Makes good choices, practical to implement, apparently widely used in practice.

Slide 15

Paging — Operating System Versus MMU

- Some aspects of paging are dealt with by hardware (MMU) — translation of program addresses to physical addresses, generation of page faults, setting of R and M bits.
- Other aspects need O/S involvement. What/when?

Slide 16

Paging — Operating System Involvement

- Process creation requires setting up page tables and other data structures. Process termination requires freeing them.
- Context switches require changing whatever the MMU uses to find the current page table.
- And of course it's the operating system that handles page faults!
- Some details . . .

Slide 17

Processing Memory References — MMU

- 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. Transfers control to interrupt handler.

Slide 18

Processing Memory References — Page Fault Interrupt Handler

- Is page on disk or invalid (based on entry in process table, or other o/s data structure)? If invalid, error — terminate process.
- Is there a free page frame? If not, choose one to steal. If it needs to be saved to disk, start I/O to do that. Update process table, PTE, etc., for "victim" process. Block process until I/O done.
- Start I/O to bring needed page in from swap space (or zero out new page). If I/O needed, block process until done.
- Update process table, etc., for process that caused the page fault, and restart at instruction that generated page fault.

Slide 19

Processing Memory References — Details Still To Fill In

- How to keep track of pages on disk.
- How to keep track of which page frames are free.
- How to "schedule I/O" (but that's later).

Slide 20

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 21

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)?
- How did the midterm compare to your expectations (topics, level of difficulty, . . .)?

Slide 22

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.
- Memory sizes have increased to a point where the odds aren't as good as they were. But a few years ago we did run into problems with the machines in one of the classrooms trying to run both Eclipse and a Lewis simulation, and then more recently with some of them attempting to run a background program that asked for more memory than its author intended.