Slide 1

Administrivia

- Reminder: Homework 4 due Wednesday.
- For the programming-problem part of Homework 3, several people have turned in only preliminary versions. If that's you, please try to finish this up soon so it isn't hanging over your head and mine.
- If you haven't turned in something for all assignments, it's not too late to get partial credit as long as you haven't looked at a sample solution. But the sooner the better,
- I've posted an extra-credit assignment intended to let you recoup points lost on Homeworks 2 and 3 (but not limited to those who didn't do well). It's near the bottom of the "lectures topics" etc. page, and nominally due the day of the final, but if you plan to attempt it you might want to do so as soon as you have time.

Slide 2

Minute Essay From Last Lecture

- Most people got the question about where to keep the page table more or less right.
- About the midterm, no clear consensus:

  Some people commented that the length was reasonable (compared to other exams I've given), but a few ran out of time.

  Some found it easier than expected, others harder.

  Most found content not surprising, though one person mentioned the question about MMU. I thought this might remind you of the question from Homework 1 about registers related to memory usage, but maybe not?

Slide 3

Memory Management — Review

- The problem we're solving: Partition physical memory among processes. Two related issues (program relocation and memory protection) both nicely solved by defining "address space" abstraction and implementing with help from hardware (MMU).
- Contiguous-allocation schemes are simple but not very flexible.
- Paging is more flexible but more complex.

Slide 4

Paging — Recap

- Idea — divide both address spaces and memory into fixed-size blocks ("pages" and "page frames"), allow non-contiguous allocation.
- Makes for a much more flexible system but at a cost in complexity — keeping track of a process's memory requires a "page table" to be used by both hardware (MMU) and software (O/S).

Slide 5

Sidebar: "Smashing the Stack"

- Last time I mentioned that having a bit in the page table that says whether page can hold executable code helps foil one attack scheme. Someone asked about this in the minute essay, so a bit more . . .
- The usual scheme for memory use within a process puts a stack at high addresses, used in function calls (for parameters and return address) and also for local variables. What happens if an attempt is made to store more data in a local-variable array than will fit? (And in C this is all too easy, no?)
- Well, you know from CSCI 1120, no? Whatever is after the array is overwritten . . .

Slide 6

"Smashing the Stack", Continued

- . . . possibly including the function's return address!
- This is an example of deliberately "smashing the stack", and if the input is very carefully crafted non-text, can be used to invoke attacker's code. (Full details in a very old paper referenced in "Useful links" on course Web site. Uses x86 assembly language but I think is fairly readable even if you don't know that, and has a useful overview of various things relevant to this course.)
- Relies on being able to transfer control to any memory location user has access to, including writable locations.
- I haven't dug into details, but having a bit in each page table entry that identifies pages that can contain executable code seems like it could help foil this scheme.

Slide 7

Page Tables — Performance Issues (as in Minute Essay)

- One possibility is to keep the whole page table for the current process in registers. Could possibly use general-purpose registers for this but likely would not. Should make for fast translation of addresses, but — is this really feasible for a large table? and what about context switches?
- Another possibility is to keep the process table in memory and just have one register (probably a special-purpose one) point to it. Cost/benefit tradeoffs here seem like the opposite of the first scheme, no?

  The big downside is slow lookup. Can be mitigated with a "translation lookaside buffer" (TLB) — special-purpose cache.

Slide 8

Paging — Feasibility Issues

- Clearly page tables can be big, if we want them all to be the same size (probably) and big enough to represent the system's maximum address space (also probably).
- How to make this feasible? more than one possibility, based on the observation that the number of valid page table entries (ones that point to a page frame) is manageable (in contrast to the number of total potential page table entries).

Slide 9

Multi-Level Page Tables

- Idea here is make page tables hierarchical in a sense:
- Each entry in the top-level table represents a range of pages. If no valid pages in that range, entry is "invalid"; else it points to a lower-level table. Only lowest-level tables reference actual page frames.

  (Figure 3-13 in text.)
- In principle, can have arbitrarily many levels, though in practice it depends on what MMU allows.
- Lookup is slower than with a single level (think about why), but again the TLB idea should help.

Slide 10

Inverted Page Tables

- Idea here is to map not from page number to page frame number but the other way around.
- So, in this scheme there's one combined table (rather than one per process), indexed by *page frame number*, with entries containing a process ID and a page number.
- Seems like then lookups would be quite slow — potentially have to search the whole table — but use of TLB mitigates that somewhat, and a clever implementation could/would have some way to make it faster.
- Potentially more difficult to implement efficiently, so at one time not used much. Coming back with 64-bit addressing?

Slide 11

Paging and Virtual Memory

- Idea — if we don't have room for all pages of all processes in main memory, keep some on disk ("pretend we have more memory than we really do").
- Or a simpler view: All address spaces live in secondary memory / swap space / "backing store", and we "page in" as needed (demand paging).
- (Aside: Why are we even bothering? Can't the processor(s) access disk? Yes, but . . .)
- Making this work requires help from both hardware (MMU) and software (operating system).

Slide 12

Page Fault Interrupts

- We said MMU should generate a "page fault" interrupt for a page that's not present in real memory. What happens then? It's an interrupt, so . . .
- Control goes to an interrupt handler. What should it do? (Are there different possibilities for what caused the page faults?)

Slide 13

Page Fault Interrupts, Continued

- One possible cause — an address that's not valid. You know (sort of) what happens then . . .
- Another cause — an address that's valid, but the page is on disk rather than in real memory. So — do I/O to read it in. Where to put it? If there's a free page frame, choice is easy. What if there's not?

Slide 14

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 (no surprise, right?) . . .

Slide 15

"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 16

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 17

"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 18

"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 19

"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 20

"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 21

"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 22

"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 23

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 24

"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 25

"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 26

Minute Essay

- None really — just sign in. Unless questions?