Slide 1

Administrivia

- Reading quiz(zes) and homework for Chapter 3 soon! I'll send e-mail. I'll allow at least a week on homework, a bit less on reading quizzes.

Slide 2

Memory Management — Recap/Review

- In context, memory management means sharing the physical memory among processes, such that each processs gets its own memory. (Usually some is also reserved for the O/S itself.) *Very* desirable to do this in a way that doesn't let processes access each other's spaces or the O/S's memory.
- All but the most primitive approaches use the address space abstraction and on-the-fly translation of addresses, via additional hardware (MMU).
- Giving each process one contiguous chunk of memory works and is fairly simple, but also is restricive. Paging is more complex but more flexible.

Slide 3

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 4

Sidebar: Memory Management Within Processes

- What if we don’t know before the program starts how much memory it will want? with very old languages, maybe not an issue, but with more modern ones it is.

  I.e., we might want to manage memory within a process’s “address space” (range of possible program/virtual addresses).
- Typical scheme involves
  - Fixed-size allocation for code and any static data.
  - Two variable-size pieces (“heap” and “stack”) for dynamically allocated data.
  - Note that combined sizes of these pieces might be less than size of address space, maybe a lot less.

Slide 5

Page Table Entries

- Exactly what's in a page table entry depends partly on hardware.
- Required(?) fields are page frame number, present/absent bit.
- Optional but useful fields include bits that can be used to track usage ("referenced/modified"), bits indicating what access is allowed (e.g., read-only), etc.
- (Figure 3-11 in text.)

Slide 6

Page Sizes and Other Details

- How big to make pages? compare extreme cases (really big, really small).
- If you know how big addresses are, what does that tell you about (maximum) sizes of physical/virtual memory?
- How big are page tables . . .

Page Table Size — Example

Slide 7

- Given a page size of 64K (2^{16}), 64-bit addresses, and 4G (2^{32}) of main memory, at least how much space is required for a page table? Assume that you want to allow each process to have the maximum address space possible with 64-bit addresses, i.e., 2^{64} bytes.
- (Hints: How many entries? How much space for each one? and no, this is not a very realistic system.)

Page Table Size — Example Continued

Slide 8

- Number of entries is $2^{64}/2^{16}$, i.e., 2^{48}.
- Size of each entry — at least enough for page frame number. There are 2^{16} of them, so we need 16 bits for that. Probably should also include a valid/invalid bit, for a total of 17 bits. Rounding up to a multiple of 8 bits (one byte), that's 3 bytes per entry.
- Total space is $2^{48} \times 3$ — bigger than main memory!! so, not realistic.

Slide 9

Performance / Feasibility Concerns

- Even with good choice of page size, serious performance implications — page table can still be big, and every memory reference involves page-table access — how to make this feasible/fast?

Slide 10

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 11

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). (Maximum address space — largest possible, e.g., 2^{32} for "32-bit system", ?? for "64-bit system".)
- How to make this feasible? more than one possibility, based on an observation: 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 12

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 13

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 14

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 15

Paging and Virtual Memory — Recap/Review

- But first review . . .
- 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 16

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 17

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?) . . . Going through these pretty quickly — probably not important to retain too much detail!

Slide 18

"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 19

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 20

"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 21

"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 22

"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 23

“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 24

“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 25

"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 26

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 27

"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 28

"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 29

Page Replacement Algorithms — Summary

- Many, many choices. Goal is to produce as few page faults as possible (is it obvious why?).
- Many choices try to predict near-future page usage from recent-past usage. Some help from hardware may be needed to do this.
- Nice summary in textbook as Figure 3-21.

Slide 30

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)?

Slide 31

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 this isn't as likely as it once was. Several 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.)