Administrivia

Slide 1

- Homework 4 correction: Programming problem will be optional; up to 5 extra-credit points.

Paging — Recap

Slide 2

- 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 3

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.
  - Notice — combined sizes of these pieces might be less than size of address space, maybe a lot less.

Slide 4

Paging, Continued — Performance / Large Address Spaces

- 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?
- (Remember that the MMU is hardware, and a bit about registers — local to the CPU, faster to access than memory but limited in number, can be general-purpose or dedicated to a particular use (e.g., the program counter).)

Slide 5

Page Tables — Performance Issues

- 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, though, and that can be improved with a "translation lookaside buffer" (TLB) — special-purpose cache.

Slide 6

Large Address Spaces

- Clearly page tables can be big. How to make this feasible?
- One approach — multilevel page tables.
- Another approach — inverted page tables (one entry per page frame).

Slide 7

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 8

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 9

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 10

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. (To be continued.)

Slide 11

Minute Essay

- Why is a “good” page replacement algorithm one that generates as few page faults as possible? (I.e., what happens if there are a lot of page faults?)

Slide 12

Minute Essay Answer

- The usual result of lots of page faults is that the computer spends more time doing “paging” (moving data back and forth between memory and disk), sometimes to the point where it isn’t doing much else.