

Slide 1

### Administrivia

- Reminder (as if one were needed?): Exam 2 Monday.
- Solutions to quizzes available online (Quiz 6 later today).
- I will make solutions to homeworks available via Google Drive later today and at least try to get your work graded by end of the week.
- I'm planning to put together a set of extra-credit problems you can do if you're not happy with your grade (and I'm planning to send out grade summaries next week if all goes well). They'll be due at or near the end of the finals period.

Slide 2

### Exam 2 Review

- (Topic by topic through the review sheet.)
- (Questions?)

### Memory Hierarchy — Recap/Review/Revisited

Slide 3

- In a perfect world, there would be a way to store bits that's very fast and can be had in almost arbitrarily large amounts for a reasonable cost. In this world — a maxim from engineering (or so I hear): “Good, fast, cheap — pick any two.”
- Textbook talks about four basic technologies for storing (lots of) bits:
  - SRAM — pretty fast, but costly, so not feasible on a large scale.
  - DRAM — significantly less expensive but also significantly slower.
  - “Flash memory” — slower but cheaper still, but does have the problem of “wearing out”.
  - Magnetic disks — cheap enough to be about as big as is needed for most general-purpose computing, but far, far slower.

### Memory Hierarchy — Recap/Review/Revisited

Slide 4

- So where does “hierarchy” come in? Well . . .
- Programs' use of memory mainly exhibits “locality” (in both time and space).
- So it's common to design systems in terms of a hierarchy, with each level larger but slower than the one above it, with the hope that we can store (a copy of) most-frequently-used data in an upper level of the hierarchy, where it's fast to get at, and access lower levels less frequently.
- Idea is that data moves up and down in this hierarchy as needed, all in a way that's invisible to application programs, *except* for effects on performance.

### Caching — A Bit More Detail

Slide 5

- In order for this to work, each “cache” (hardware or virtual memory) must have space for some data from the next level down, plus some way of (correctly!) reading from / writing to next level down, which means having some way to map from lower-level addresses to elements.
- Idea is that for reads, processor just reads using address as we've discussed, and either:
  - Data is found in the cache — “cache hit” — and given back to processor.
  - Data is not found — “cache miss” — and hardware/software does whatever is necessary to get it there and then continues as for hit.Obviously(?) the fewer caches misses the better.

### Caching — A Bit More Detail, Continued

Slide 6

- But wait — if cache is smaller than what it's caching, how can this work? each cache element could potentially contain one of many pieces of data? So include in cache element a “tag” that says which one it contains, plus a “valid” bit.
- For writes, things are a bit more complicated — similar idea applies, but must decide whether to write to lower levels immediately or wait. Writing immediately is easier but slower, probably enough so that it's worth the trouble to do something more complicated. More details in textbook.
- Overall, textbook (section 5.8) presents four questions that pretty much sum it up; adding one more . . .

### Caching — Size of Elements

- Processor caches *can* store single words, but might store larger units (2 words, or 4, or ...) — “cache lines”. Idea is to exploit spatial locality.
- Virtual memory typically uses much bigger units (often “pages” of 2K or 4K).

Slide 7

### Caching — Mapping Addresses to Cache Elements

- “Direct map” cache is simple — each memory address maps to exactly one cache element.
- “Fully associative” cache is opposite extreme — any memory address can map to any cache element.
- “Set associative” cache is in between — each memory element maps to a set of entries. Reasonable compromise between extremes?

Slide 8

### Caching — Looking Up Data

Slide 9

- For “direct map” cache, simple — only one cache element to check, so just compare tags. So this method is fast but not very flexible.
- For “fully associative” cache, more complicated — potentially have to search whole cache for matching address. Very flexible but costly to implement with good performance.
- For “set associative” cache, in between — still have to check multiple elements, but fewer of them. Reasonable compromise between extremes?

### Caching — Mapping Addresses to Cache Elements, Revisited

Slide 10

- Which is used? for virtual memory, likely fully-associative; for processor caches, one of the others.

### Caching — Replacing Cache Elements

Slide 11

- On a “cache miss”, if appropriate cache elements are all in use, must pick one to replace. For direct mapping, trivial (only one choice); for the other two not so trivial.
- How to choose? goal should be to replace something that won't be needed again, and often approaches are based on temporal locality (if not used recently maybe won't be used again soon).
- For processor caches, hardware problem, various solutions exist; for virtual memory, software (O/S) problem, and again various solutions exist (“page replacement algorithms”).

### Caching — How to Manage Writes

Slide 12

- One complication here is that if cache elements are more than one word, need to read old element, then change the word being written.
- And then — write back immediately (“write-through”), or wait (write buffer or “write-back”)? former is easier but could be quite slow; latter is more complicated but probably needed for acceptable performance.

## Minute Essay

- None — quiz.

Slide 13