Slide 1

Administrivia

- Reminder: Reading Quiz 4 due Wednesday.

Slide 2

Paging — Review/Recap

- Basic idea is simple: Divide up each process's address space into "pages" of some fixed size N, physical memory into "page frames" also of size N, and map pages actually being used into page frames.
- Note that typically much of a process's potential address space is unused, and no need to find physical memory for the not-in-use pages.
- Sounds promising, though two potential problems: speed of access, and size of page tables.
- But first a couple more things . . .

Slide 3

Address Translation Revisited

- Previously we looked at how to translate virtual to physical address if using paging — calculate page number and offset, look up page frame number, combine with offset.
- But what if page is not valid / not present? Hardware generates “page fault” interrupt / exception. Operating system’s job to decide what to do next. For now, this means address is invalid, and probably the process should be terminated.

Slide 4

Paging and Protection / Isolation

- Note that if using paging to space-share memory, many problems involving who has access to which pages go away:
- Isolation is the default: You can’t access what you can’t find!
- Sharing of selected pages is possible, with support for read-only access if hardware supports it.

Slide 5

Paging — Speed of Access

- Can't realistically keep a page table of any size in registers (on chip), so store in memory.
- But then every memory access actually requires *two* memory accessses. What to do?

Slide 6

Caching to the Rescue!

- Caching often a good strategy for anything involving memory, since memory accesses are slow, but often exhibit:
- *Temporal locality* — data used in the recent past likely to be used again in the near future.
- *Spatial locality* — data in active use often clumped up together (e.g., local variables in a function, or loops through arrays).
- Makes it likely that page-table accesses will be especially amenable to caching (all data in a page shares a PTE, right?).

Slide 7

Translation Lookaside Buffer

- Fancy name for cache for page-table entries.
- Idea is that it holds page-table entries in current active use.
- Address-translation hardware first checks this cache.
- If PTE (page-table entry) for address being translated found, proceed to translate.
- If not, "TLB miss" . . .

Slide 8

TLB Misses

- If needed PTE not in TLB, must find in page table, using address of page table (in a register) and page number as index.
- Formerly done in hardware, but more-recent hardware may just generate an interrupt and let software do it.

  (Why the switch? If done entirely in software, hardware can be less complicated, more flexibility for O/S designers.)
- Either way, once the needed PTE is found, hardware should retry the instruction that generated the miss.

Slide 9

Sidebar: RISC Versus CISC

- At one time computers' instruction sets (remember those from CSCI 2321?) were large and included complex operations. Made life nice for assembly language programmers but harder for hardware designers.
- At some point times changed, and now instruction sets designed with hardware designers in mind, in the thinking that most programmers will write in high-level languages, so whether assembly language is highly expressive doesn't matter.

  (Aside: The Patterson and Hennessy mentioned in the textbook? Authors of the book I usually use in CSCI 2321!)

Slide 10

TLBs and Context Switches

- Mappings in TLB become invalid if we switch address spaces. Simplest solution is just to flush cache and let it fill again as the new process runs. (One more thing that affects speed of context switch.)
- Unless . . . Recognized problem, and some hardware has features to address it. Details interesting but not critical.

Slide 11

TLBs — Replacement Policy

- TLBs typically "fully associative" caches, in which a cached value can be anywhere in the cache, as opposed to simpler cache in which if a value is in the cache at all it's at one fixed location. (Nice discussion of caches in the textbook I use for CSCI 2321, but for financial reasons you may not still have a copy of that.)
- Typically more things we could cache than space to cache them. When we want to add something, what to evict? Obviously want to minimize TLB/cache misses. Many strategies possible; two simple ones are "least recently used" and random. More when we talk about caching memory to disk.

Slide 12

Page Tables — Size

- Second problem: Page tables can be huge, sometimes too big to realistically fit into memory.
- How to make them smaller?

Slide 13

One Option — Bigger Pages

- Page tables are so big because there are so many pages in address spaces.
- If pages bigger, fewer of them, table smaller. Problem solved? Not really . . .
- Hardware may constrain page sizes.
- Large page size means wasted space within pages.

Slide 14

Another Option — Combine With Segmentation

- For simplicity we want all page tables to be the same size, but most will be very “sparse” (lots of pages not mapped to a physical page). Wasteful, no?
- One idea — combine paging with segmentation, i.e., have several segments (which can vary in size), each consisting of a much smaller range of pages.
- Advantages / disadvantages pretty much those of segmentation — avoids waste, but means system has to deal with things (page tables, here) of different sizes.

Slide 15

Another Option — Multi-Level Page Tables

- Another idea based on observation that while number of valid pages in page table may be small compared to total size, not distributed evenly, but in groups.
- So if we divide the whole table into bigger chunks, odds are many will be totally unused.
- Then we can represent each unused chunk as "not valid", and partly-in-use chunks as pointers to subsets of what would be a full page table. (Figure 20.3 in textbook shows two-level scheme.)
- Saves memory, but with a cost — added complexity, TLB misses mean not one memory access but two.
- Same idea could be extended to (almost?) arbitrarily many levels.

Slide 16

Another Option — Inverted Page Tables

- Another idea is to turn the basic map around — i.e., rather than map combination of process and page number to page frame number, map page frame number to combination of process and page number.
- Ties size of (inverted) page table mostly to size of physical memory (though number of bits needed for page numbers and for process IDs will be a factor also).
- Downside is that now finding the right PTE is not a simple table lookup. Potentially quite slow, so to make this realistic would need to do something more sophisticated than linear search. (Hm, managing TLB misses in software looking attractive?)

Slide 17

Paging — Recap

- Seems that two biggest downsides to paging — poor performance, huge data structures — are solvable.
- Amusing(?) concluding remarks in Chapter 20.

Slide 18

Minute Essay

- Questions? Do you feel like paging is coming together in your head? Textbook not wrong that there's a lot of complexity here, but I still say basic ideas are fairly straightforward.