notes for 2/25/25

Announcement: the VM assignment is a *partner project*

- From last time, review: "the tyrrany of C"
  - (draw the classic C process space.)
  - we don't have any need for stack/heap separation!
  - NOT review: what else is out there that *doesn't* follow stack/heap separation??
  - Go-style languages with coroutines/goroutines: LOTS of teeny-tiny stacks, all alike, which cannot be reserved
  - Something we haven't even thought about yet?
- Remember the slab allocator! We had N segments, each storing a big array of K fixed-size objects
  - (draw the slab allocator segments again)
  - an aside: makes fork a little better, can set *entire segments* to read-only for code sharing
  - when is this good?
  - when is this bad?
  - how many of these allocators might we need? How often will this become fragmented? (draw on the side, ask N)
  - How much hardware are we asking for to support this?
  - How big do our PCBs get / what's the cost of swapping?
  - Note: segment-style treatment required N objects to be *contiguous* in physical memory. Is this an actual necessary requirement??
- An aside: when does memory need to be physically contiguous? (interact)

Page Tables

- Review: the page table, at the VERY END of last time.
  - Note: this is the most popular soution
- Basic ideas:
  - what if rather than having [small N] segments of variable size, we had [large N] segments of fixed size? (annotate the segments/slab-allocator for this)
  - there's a good block size that can store (perhaps many copies of) lots of sizes of object (illustrate)
  - nobody said that segments *couldn't* be physically contiguous... solve big arrays that way! (illustrate in the physical memory picture)
- What's in the page table?
  - draw out the page table with appropriate bits, basically the important one is the valid bit, but then there's also access modes.
- Basics of translation
  - draw out the virtual address, draw out the table, show the high-order bits and how they index the table.
- What are some issues we can see with lots of small segments?
  - What does the hardware do??
  - What does the OS do??

- Simplest solution: the giant page table in the sky
  - (you need to illustrate all of this)
  - single register: Page Table Base Register (PTBR): points to *per-process* page table
  - On process swap: store/load this register
  - What does hardware do: Full page table lookup, automatically(???) <-- this is too expensive, but it does work!
  - SHOW: a basic translation of a physical address to a virtual address, for an example program, using the new page-table-in-the-sky
- Hardware's role: do *fast* translations
  - Benefits/drawbacks of current approach?
  - How big is this big page table for a 64-bit address space?

The TLB: solve the speed-of-lookup problem

- The *translation-lookaside buffer* (TLB) to the rescue!
  - *fixed-size* harware table (looootta registers in use for this one) [how big?? good question! Varies wildly in practice. Probably 512 entries? MIPS has 64. Lookup on google, live, for modern arch.]
  - virtual, physical addrs; valid, prot bits
  - this is a fully-associative cash
  - **re-illustrate** how the TLB saves the day
  - discuss: issues. replacement algorithm? prefetching?
- OK: *now what* does the OS have to do on a process swap?
  - interactive! But it's just "flush the TLB" as an added step, which is basically set all those entries to "invalid"
  - Aside: scheduling and timers. Now process swaps kinda *are* expensive, need to time the interrupts carefully. Lotta calibration and magic constants!

Multi-level page tables: solve the size-of-table problem

- (we will not have time to get here, probably)
- Hey, why not just ... have a page table for your page table. These are "page frames". You can do lots of this

What does hardware *actually do?*

- x86: fully in-hardware multi-level page table
- MIPS: fully software-managed, no autofilling TLB.
- RISC-V: *virtual memory itself is optional, and you can use base+bounds if you want*
- SPARC: insane register windows