Lecture notes 2/26

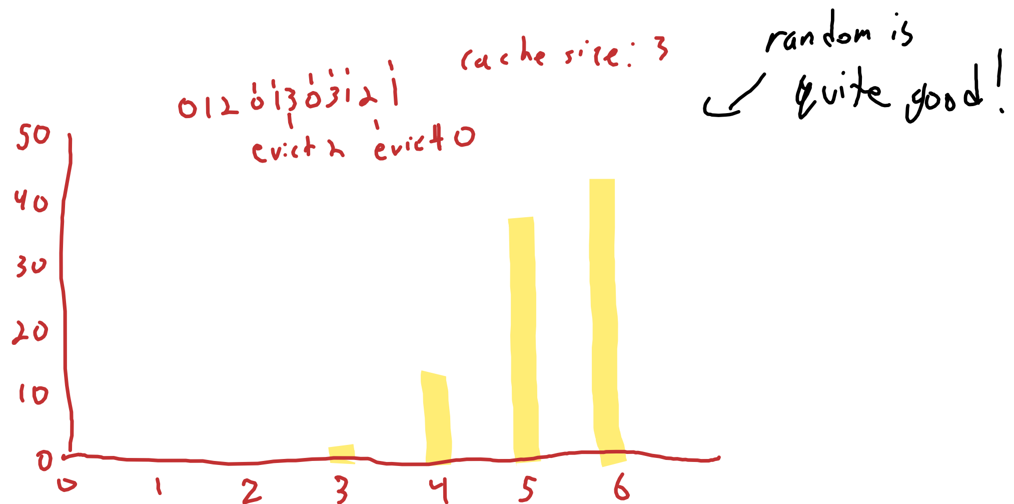
This is basically going to be a lecture about page faults--what happens when you *don't* have a mapping for a particular part of memory?

Also, remember that you need to start by talking about multi-level page tables.

we should make it clear that the Page Table is per process, but that physical memory and the TLB are shared. So when we talk about physical memory or the TLB as caches, we are NOT making process-local decisions.

the interesting things here are that we have a "present" bit, a "dirty" bit, and a "valid" bit as metadata. You probably also want to talk about CoW via the protectin bits.

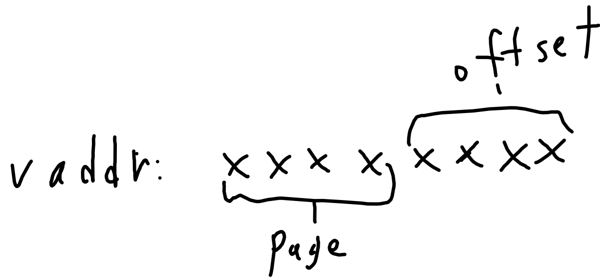
You also need to say something about eviction and eviction algorithms.



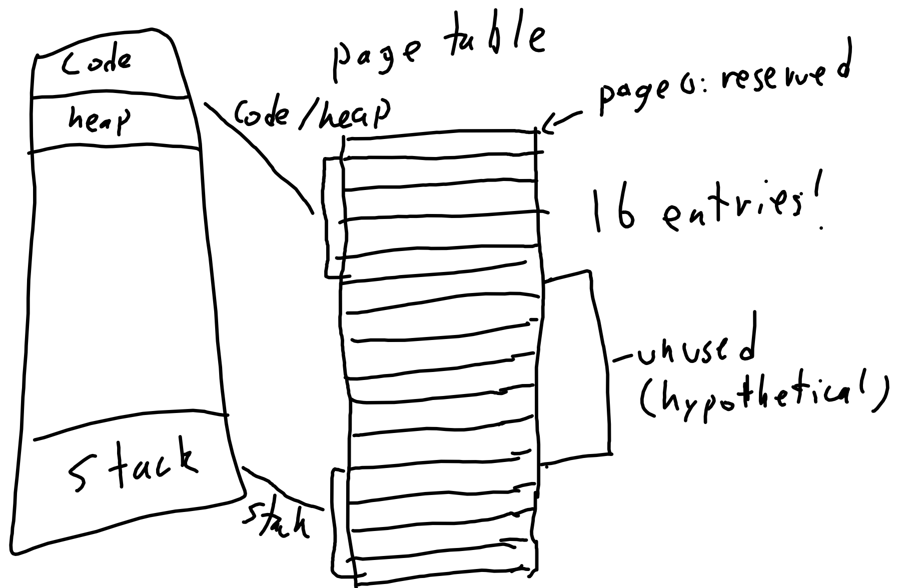
metadata for LRU, "use bit" in page table for clock algorithm. Also uses the dirty bit.

talk about mmap and the pmap command

multi-level page tables vs single-level pages



recall: C assumption:



how much wasted space?

8x(8 + 8 + 2 + 1 + 2)
vaddr, paddr, prot, valid, other

= 8x22 = 176 bits / 22 bytes

what about 32-bit address spaces?

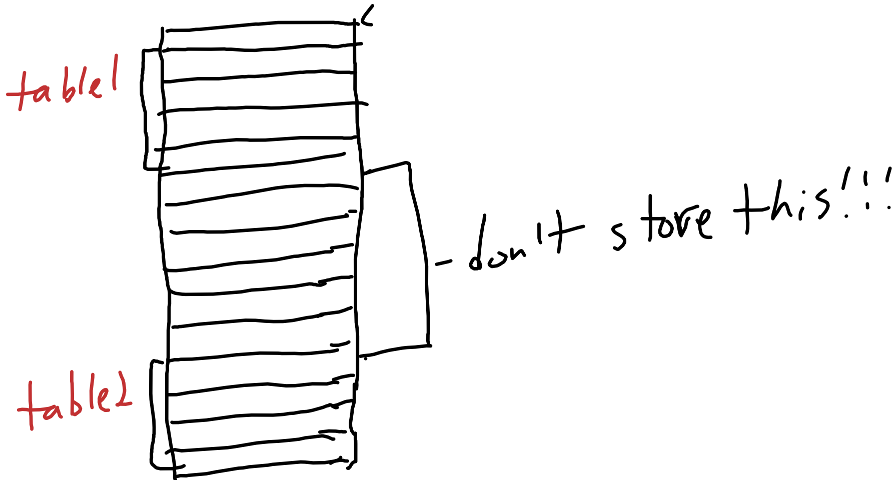
xxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxx
page? offset?

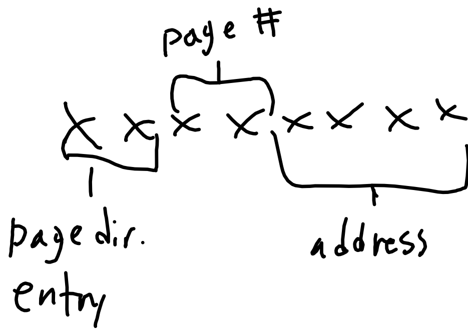
$22 \times (2^{28} - 8) = 738$ mb

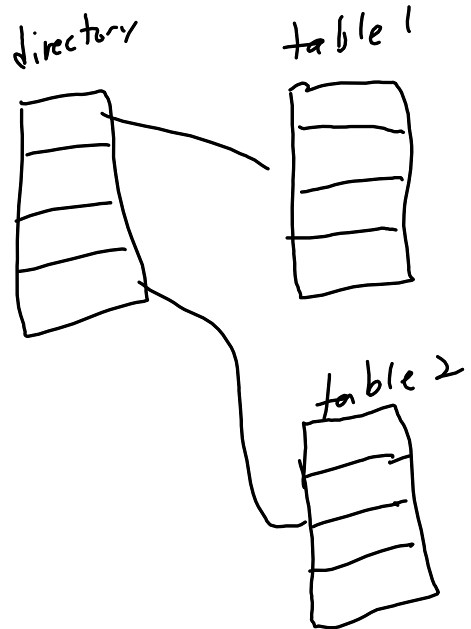
what about 64-bit?

$22 \times (2^{60} - 8) = 3,170,534$ TB

multi-level page tables







tada!

example: include TLB and show process desched.

emphasize per-proc. vs per-cpu stuff

-next- the "present" bit, talk about paging to disk

(it's not exciting)

EXCEPT it is also a global thing (inverted ptable?)

how: caching policies.

A little bit more detail about paging: the concept is (and has always been) that you can evict process's memory from "fast" storage (e.g. RAM) to "slow" storage (e.g. disk). You do this on a *per-page* basis; whenever you want to load in some new memory, if there's not enough physical memory left to map it, you can just go ahead and evict something from a page table. The page table entry then maps the virtual page to the physical address *on disk* for that stored page, and marks the "present bit" to 0, telling the OS that the page needs to be "paged back in" in order to be accessed. Note: you can page out from *any* process's page table, not just the one that's running right now! Also note: you need to make sure that *every* process's page table knows when a physical page has been paged out, in cases where the same physical page is mapped in multiple virtual pages (across potentially multiple processes).

How to decide what to page out?

Talked about the idea of the "optimal" policy for cache-replacement is just "the thing that will be used the furthest into the future"

Talked about some algorithms that work pretty well---like least-recently or least-frequently used---but that take a lot of work to run. Trouble is, the page table is big, so figuring out the *globally* least-recently used page is a long linear scan! Welcome to OS, where linear algorithms are WAY too expensive.

Note: in the TLB, this is not true---hardware-level parallelism lets us search it REALLY fast. So you can implement a LRU/LFU policy in the TLB... if you're willing to include enough metadata to manage it.

What can we do instead? The Clock Bit.

Basic idea: evict an arbitrary page that *hasn't been touched recently*. How do we define "recently?" Since the last time we needed to swap a page out. Idea here is that "page faults" --- which is what happens when you need to swap out a page from physical memory --- are kinda rare in almost all cases. So it takes a "long time" to go from one page fault to the next. Just go ahead and add a bit to the page table that tracks whether a particular page has been accessed since the last page fault! When you go to evict, find a page where that bit is 0, swap it out --- and "reset the clock," setting everyone's bits back to 0 (and following the same mark-on-access algorithm for future page accesses)