Core Map:
We talked about the core map briefly in lecture. This is a data structure that keeps metadata about physical pages. This is where you will want to keep track of references to physical pages in memory. The two main cases changing the reference count of pages are references via fork (copy-on-write) and MMIO.

Copy-on-Write Fork:
As we mentioned at the end of last week's section, the biggest piece of VM optimization code here is the `fork()` system call. In most scenarios, a majority of the virtual address space that is copied from the parent isn't actually used. Either it's never modified so you have a useless copy floating around, or it's all immediately unloaded because the child process calls `exec()` and wipes out the address space anyway. So we implement copy-on-write, which only makes a physical copy of a parent virtual page if either the parent or the child writes to it.

Modifying `fork()`:
We want to modify `fork()` so that it does not actually create new physical pages when alloc'ing user virtual memory. Take a look at `vm.c` to find functions that can help with explicitly setting up the page table for a new process *without* actually allocating new physical memory. Specifically look at how `copyuvm()` traverses the page table to allocate new pages to a new page table. But instead of allocating new pages and copying the data, we will instead update the flags on the old page to no longer be writable and have the `PTE_RO` macro bit set. Remember we can set this bit because there are select bit positions that the hardware ignores. The MMU will function the same whether or not this bit is set, this is why we have to explicitly make the page unwritable.

Page Table Entry Flags:
`mmu.h` has constants that are defined for permissions for entries in the page table, like `PTE_RO` above. Think carefully about what you need for COW pages before and after a write has been made to a shared page.

Q: How will the kernel know a copy-on-write page has been written to? In other words, what processor "feature" are we taking advantage of?

Interfacing with Memory-Mapped I/O:
There are subtle differences that you have to deal with when copying memory-mapped file pages during `fork()`. If a file has been memory-mapped by a process before it forks, then the child will have that file open (as with everything else in the virtual address space). But if either process writes to the memory-mapped file region, that page in physical memory does not get copied, because the same file has been opened by both processes.