Lab5

DMA read:

看了很久文档，大致了解下来，基本的原理就是直接让 DMA 来读取 IDE 的寄存器中的内容，然后存入内存（好像需要占用总线，如果每次读的多的话可能会导致 CPU 的指令执行完之后想要从内存中读取指令，但是得不到总线的情况，因此 DMA 每次读的不多)。在这 lab 里面最多 64k。每次要让 DMA 去读的话，先要构造一个结构体的数组存储起始地址和长度。好像最后一位一定要是 0.然后设置 DMA 的寄存器，然后设置 IDE 的寄存器，最后往 DMA 的 start 那个位写入 1。DMA 就开始帮忙读写了。因为数据结构中的高几位都是保留的。所以每个结构体里面能表示的大小有限。高位中有一位表示数组的结束 EOT。读完往中断端口写 1.然后 pic 就知道了。中断完了后要分别向 pic 和 ide 的特定端口写入，确认收到中断。在传输中，因为没有 CPU 的参与，所以用的都是实地址，这个一开始没注意，导致过错误。

当 DMA 未完成时，我们不能返回。必须把 fs 设为 NOT_RUNNABLE 或者循环 sys_yield. 知道发生 DMA 的中断，并且是数组中的最后一个结构体时，fs 方才能够返回。

Fs.c

`alloc_block_num(void)` 就是搜索 `bitmap` 里面有没有空余的。有的话分配一个空位，这里要注意的是即时将 `block` 重新写入硬盘，避免不一致。

`alloc_block(void)` 调用 `alloc_block_num()` 得到硬盘空间。然后将硬盘空间映射到 `fs` 的地址空间。默认的映射是 `PTE_P|PTE_U|PTE_W`。如果映射失败，还需要返还得到的空间。

`file_block_walk` 的作用类似于 `pgdir_walk`。需要查看 `filebno`,如果小于 `10`，那么是用直接块，如果大于 `10`，那么是用间接块。直接块在 `file` 的数组中。如果间接块不存在，还需要 `alloc`。如果刚 `alloc`，还需要将块清 `0`.避免错误。

`file_map_block` 在 `file_block_walk` 的基础上调用 `alloc_block` 分配块。

`file_dirty` 需要将 `file` 所在页表项的 `PTE_D` 设置。具体方法可以将内容读出在写入达到效果。

`read_block` 和 `write_block` 需要注意重新 `mappage` 一下，清除 `PTE_D`。

`File.c`

`Fmap` 就是将获得从 `oldsize` 到 `newsize` 之间的文件的块。`Funmap` 相反。值得注意的是要将有 `PTE_D` 的页反馈给 `fs`。因为是使用该页的进程弄脏该页的，`CPU` 只会在该进程的页表里面设置这一位。

本次 `lab` 我觉得相对而言 `DMA` 和 `fs` 较其他几个难一点。其他几个文件里面有现成写好的其他函数，很多都可直接借鉴的。

`Debug`。主要的 `debug` 是一开始 `DMA` 没设置正确，比如未用实地址等。后面相对而言，尚可。