```
CS5600, Handout Week12.b

// 1. Read/write disk in Lab4 (CS5600 file system)
// borrowed from Lab4, fs5600.c

    /* disk access.
     * All access is in terms of 4KB blocks; read and
     * write functions return 0 (success) or -EIO.
     *
     * read/write "nblks" blocks of data
     *   starting from block id "lba"
     *   to/from memory "buf".
     *     (see implementations in misc.c)
     */
    extern int block_read(void *buf, int lba, int nblks);
    extern int block_write(void *buf, int lba, int nblks);


    /*
     * below are two toy examples of
     * reading from and writing to a disk block
     */

    // Reading one block

    char buf[FS_BLOCK_SIZE];   // FS_BLOCK_SIZE=4096; see panel 2
    int inum = 100;            // block number to read from
    int ret = block_read(&buf, inum, 1);
    if (ret < 0) {  // error; ret should be -EIO
        return ret;
    }


    // Writing one block

    char buf[FS_BLOCK_SIZE];  // again, 4KB buffer
    ...                       // update buf

    int ret = block_write(&buf, 99, 1);  // update the 99th block
    if (ret < 0) {  // error; ret should be -EIO
        return ret;
    }
```


handout_w12b
CS5600, Cheng Tan
4/6/22, 10:01 AM

```
// 2. CS5600 file system data structures
// borrowed from fs5600.h with minor changes

    /*
     * file:        fs5600.h
     * description: Data structures for CS5600 file system.
     *
     * CS 5600, Computer Systems, Northeastern CCIS
     * Peter Desnoyers,  November 2016
     *
     * Modified by CS5600 staff in fall 2021.
     */

    #define FS_BLOCK_SIZE 4096
    #define FS_MAGIC 0x30303635

    #define INODE_NUM_PTRS (FS_BLOCK_SIZE/4 - 5)

    /* Superblock - holds file system parameters.
     */
    struct fs_super {
        uint32_t magic;
        uint32_t disk_size;         /* in blocks */

        /* pad out to an entire block */
        char pad[FS_BLOCK_SIZE - 2 * sizeof(uint32_t)];
    };


    /* inode = 4096 bytes */
    struct fs_inode {
        uint16_t uid;
        uint16_t gid;
        uint32_t mode;
        uint32_t ctime;
        uint32_t mtime;
        int32_t  size;
        uint32_t ptrs[INODE_NUM_PTRS];
    };


    /* Entry in a directory
     */
    struct fs_dirent {
        uint32_t valid : 1;
        uint32_t inode : 31;
        char name[28];        /* with trailing NUL */
    };
```

```
CS5600, Handout Week12.a

//   Code snippets borrowed from WeensyOS boot loader
//   They illustrate how kernel "talks" to a disk through programmed I/O:
//   the bootloader reads in the kernel from the disk.
//
//   See the functions boot_waitdisk() and boot_readsect(). Compare to
//   Figures 36.5 and 36.6 in OSTEP.


//   WeensyOS boot loader loads the kernel at address 0x40000 from
//   the first IDE hard disk.
//
//   A BOOT LOADER is a tiny program that loads an operating system into
//   memory. It has to be tiny because it can contain no more than 510 bytes
//   of instructions: it is stored in the disk's first 512-byte sector.


    #define SECTORSIZE      512

    // boot_readsect(dst, src_sect)
    //    Read disk sector number `src_sect` into address `dst`.
    static void boot_readsect(uintptr_t dst, uint32_t src_sect) {
        // programmed I/O for "read sector"
        boot_waitdisk();
        outb(0x1F2, 1);             // send `count = 1` as an ATA argument
        outb(0x1F3, src_sect);      // send `src_sect`, the sector number
        outb(0x1F4, src_sect >> 8);
        outb(0x1F5, src_sect >> 16);
        outb(0x1F6, (src_sect >> 24) | 0xE0);
        outb(0x1F7, 0x20);          // send the command: 0x20 = read sectors

        // then move the data into memory
        boot_waitdisk();
        insl(0x1F0, (void*) dst, SECTORSIZE/4); // read 128 words from the disk
    }


    // boot_waitdisk
    //    Wait for the disk to be ready.
    static void boot_waitdisk(void) {
        // Wait until the ATA status register says ready (0x40 is on)
        // & not busy (0x80 is off)
        while ((inb(0x1F7) & 0xC0) != 0x40) {
            /* do nothing */
        }
    }
```


1TB (disk) ≠ 1TB (mem)
⇓ 1 Billion Bytes < 2^{40}B

1MB (disk) = 1 million Bytes

VA, LBA 32 bits

× 4B = 512B

I/O polling

interface of disk.

OS → disk
- platter
- track
- sector

LBA → disk

0 1 2 ... sectors

- Capacity: 100GB ~ 10TB
- #platters: 8-10
- #tracks: 10s ~ 1000s
- #sectors/track: ~1000
- RPM: ~10000
- transfer: 50 - 150MB/s
- ⇒ mean time between failure: 1 million h

Q: 8TB 512B. #bits for LBA?

$$\frac{8TB}{512B} = \frac{2^3 \cdot 2^{40}}{2^9} = 2^{34}$$

⇓ 34 bit

48 bit → 2^{48} Byte → 128PB (max)

Hard disk drive organization

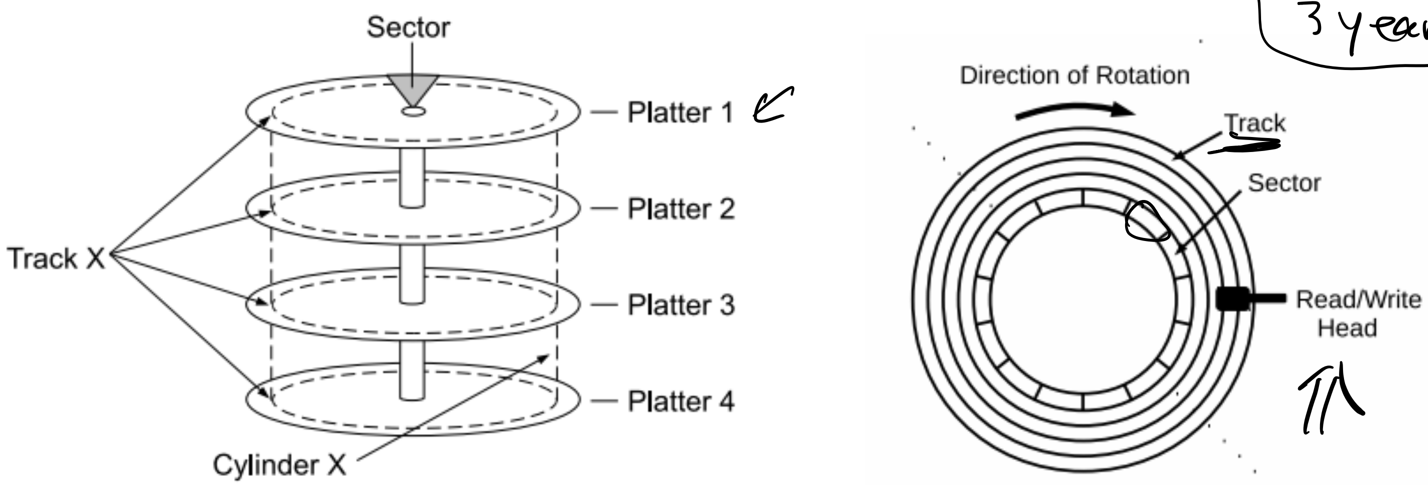


Fig 5.11, 5.12 from
http://www.ccs.neu.edu/~pjd/x600-book-v0903.pdf

A Flash-based SSD (logical diagram)

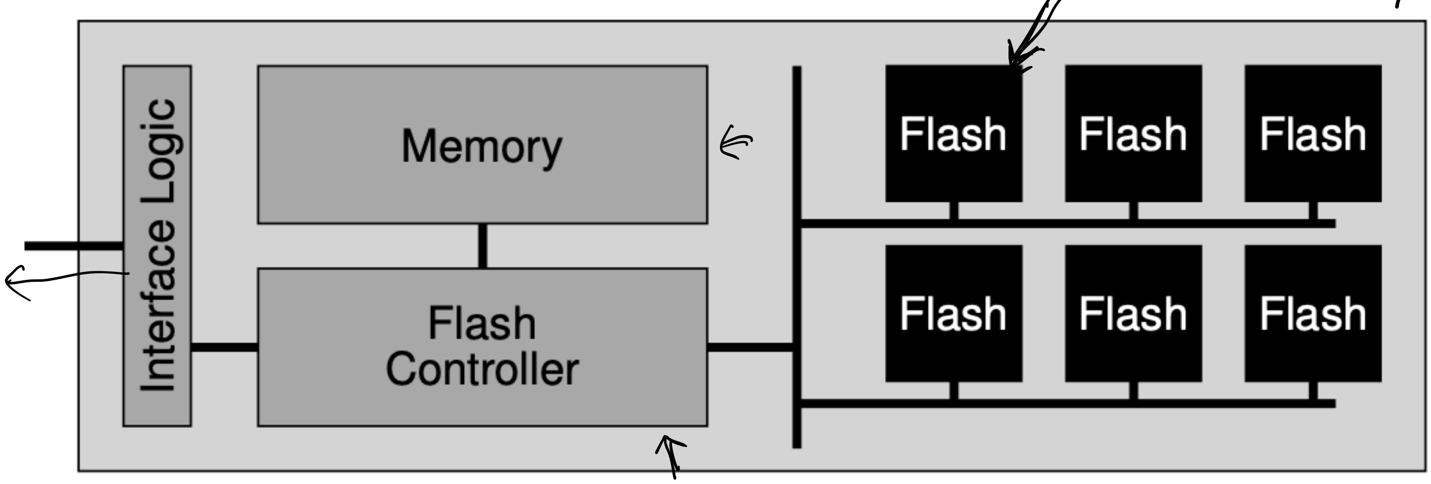
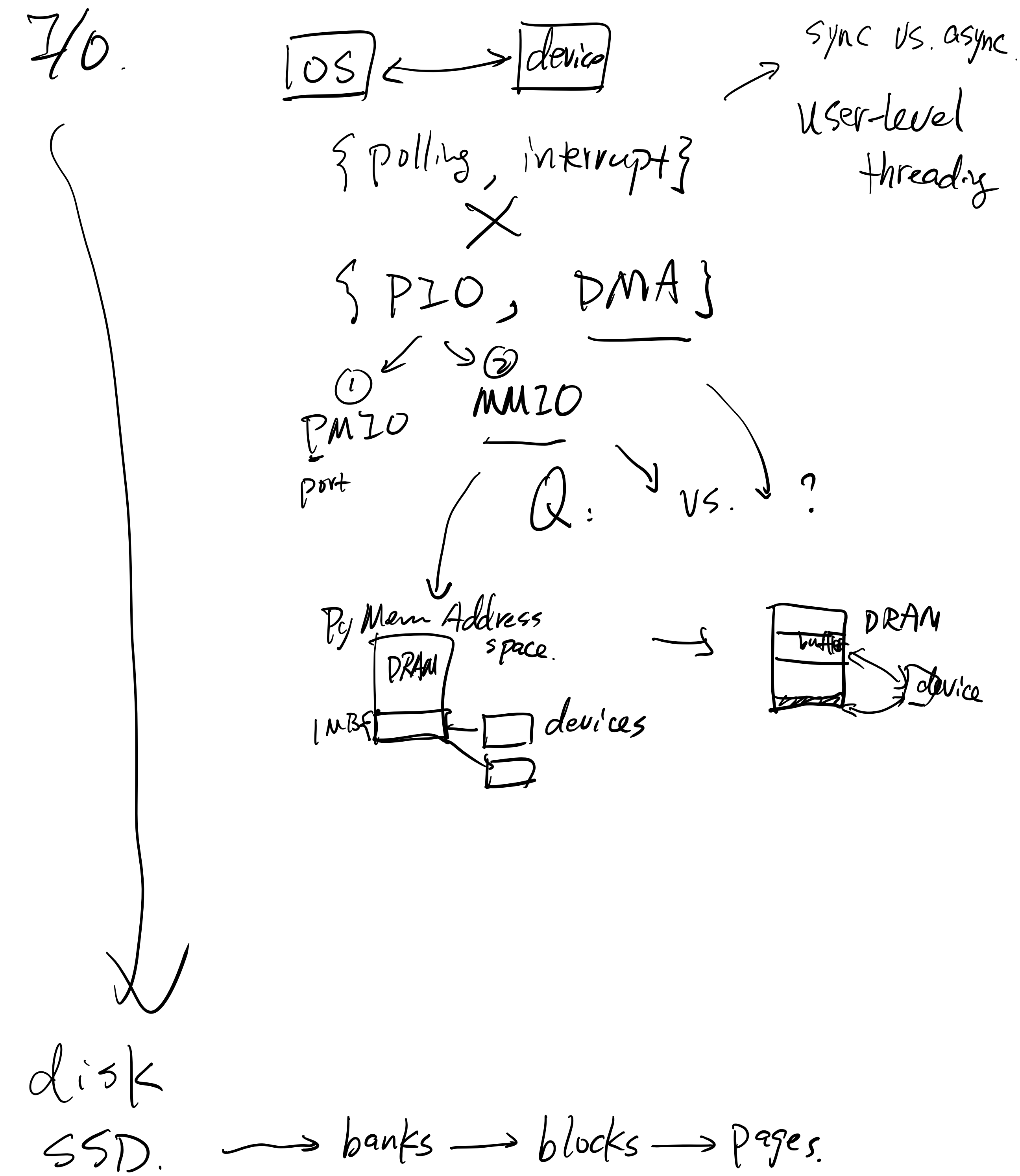


Fig 44.3 from
https://pages.cs.wisc.edu/~remzi/OSTEP/file-ssd.pdf

1. Last time
2. SSD
3. Intro to file systems
4. Files
5. Directories

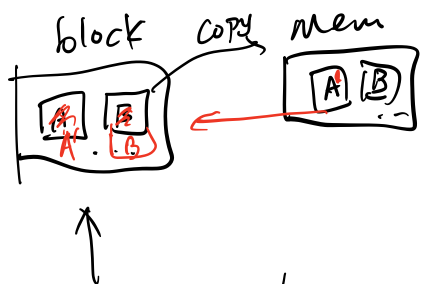


Operations	granularity	perform.
• read.	1 page	~10s µs
• erase	1 block., resetting all bits ⇒ 1	~1 ms
• program.	1 page., setting some bits ⇒ 0	~100s µs

(10x, 10x)

(cannot program the same page twice without erase)

Q: page A. update A'. HOW?



erase x 10,000 wear-out

```
for(i=0; i<10000; i++){
    update A.
}
```

• FTL

logical page (SSD) —FTL→ physical SSD page

• log-structure FTL

write(LPX) → append, update FTL

read(LPX) ← check mapping return

PX.

```
** an example:
  --Given a flash bank has three blocks; each has two pages.
  --there are four writes to pages:
    wirte(logic_page_1)   [short as LP1]
    wirte(logic_page_10)  [short as LP10]
    wirte(logic_page_99)  [short as LP99]

  --what will happen:

          +-----------------------------+
  blocks  | block 0 | block 1 | block 2 |
          +---------+---------+---------+
  pages   | P1 | P2 | P3 | P4 | P5 | P6 |
          +----+----+----+----+----+----+
  data    |LP1 |LP10|LP99|    |    |    |
          +----+----+----+----+----+----+

  mapping:
    LP1 => P1, LP10 => P2, LP99 => P3

Question:
  what will happen if the following op is write(logic_page_1)?
```

GC.

>85%

<50%

write(LP1'')

- FS,

  ① persistent storage.

  ROM

  ② give named bytes. (file)

  ③ friendly way to find named bytes (directories