File System Implementation:

Way that disks operate:

- You can rewrite blocks of memory by reading it and then modify it & write it back
- any block on Disk can be accessed directly

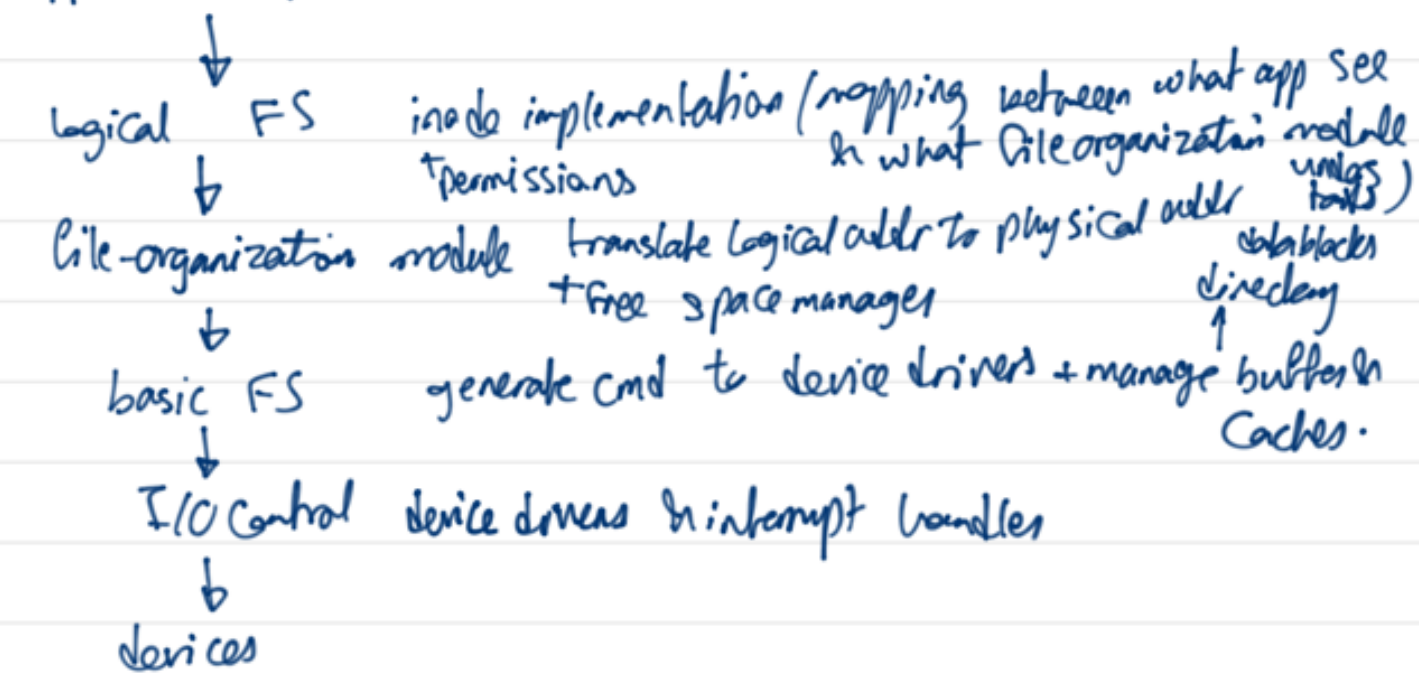
- Everything is done in units of blocks. (512 bytes)

* File System job is to provide an efficient (system perspective) Convenient (user ")

to the disk. 2 design problems:

1) File System interface: ch. 11
2) Creating algs & DSs to map the logical file system onto the physical secondary-storage devices.

* Layered design is for scalability and ?



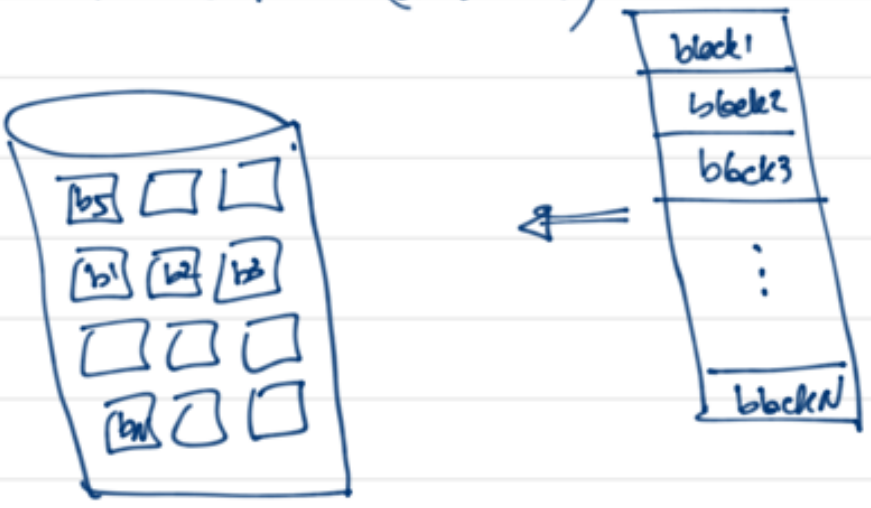
* I/O Control, device drivers & interrupt handlers to transfer info between the MM & the disk. retrieve block 123
↓
Hardware instruction.

* Basic FS : generate commands to appropriate device drivers to read/write. each physical block has an addr:
drive 1, cylinder 73, track 2, sector 10.

- this layer also manages the memory buffers & caches that hold various file_system, directory & data blocks.
Buffer management.
Caches : hold frequently used file-system metadata to improve performance

*File-organization module:

1) translate logical block addresses to physical block addresses for the basic FS. (Similar MMU) File



2) free-space manager: tracks unallocated blocks. & provide these to file-organization module.

* logical FS: Manages meta data info.
↳ file system structure except the actual data (Content of the file)

1) manages the directory structure to provide the file organization module with the info it needs, given a symbolic file name.
maintains file structure via file-control blocks.
FCB (inode in Unix FS) Contains info about the file: ownership permissions, location
2) protection

Layered Structure:

+) minimize the redundant code.
↘ basic FS or drivers can be used by multiple FSs.

−) OS overhead ⟹ decreased performance
Layering design is a challenging task.

* Structures needed for FS implementation:

1) Boot Control block (per volume): info for booting an OS from that block. The first block of a volume.

2) Volume Control block: volume details: number of blocks, size of the blocks, free-block count & free-block pointers, free-FCB count & FCB pointers

Unix FS (UFS) → Superblock
NTFS → master file table.

3) Directory Structure (per file system).
UFS: file name → inode
NTFS: it is stored in the master file table.

4) per-file FCB

5)

6)

7) system-wide open-file table

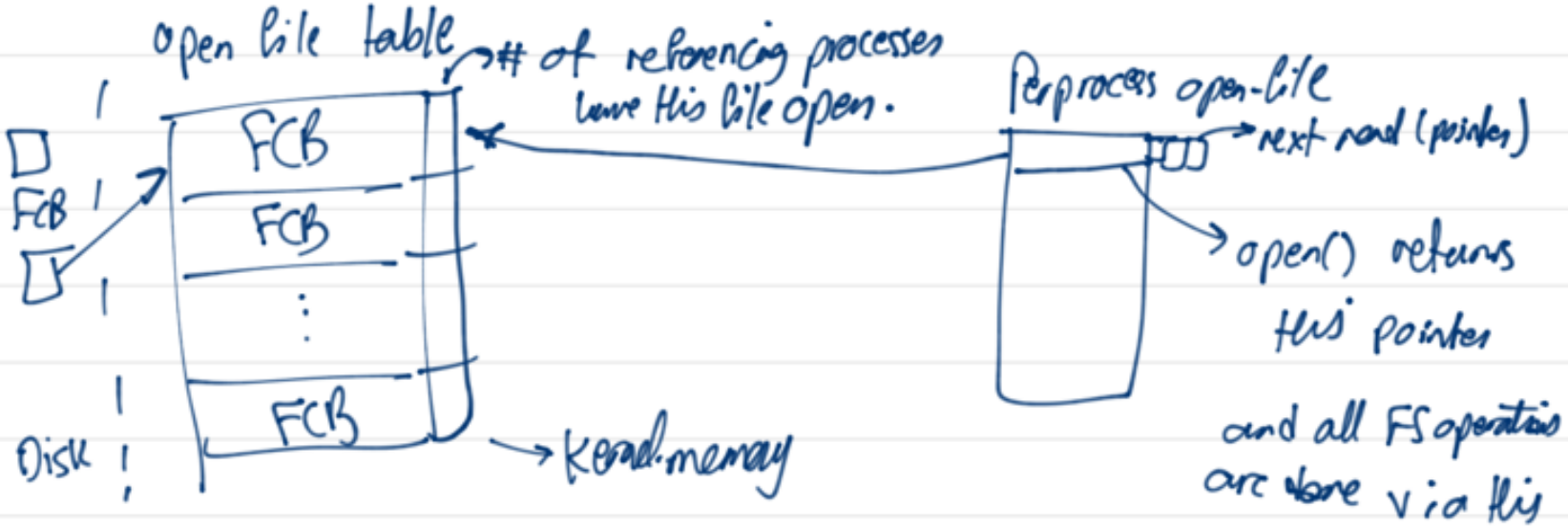
8) per process " " "

9) Buffers

A) To create a file: app program ⟶ Logical file system.
knows the format of the directory
↓
allocates a new FCB
or from the pool of FCBs
↓
read the appropriate directory into memory, updates it with new filename & FCB & writes it back to the disk.

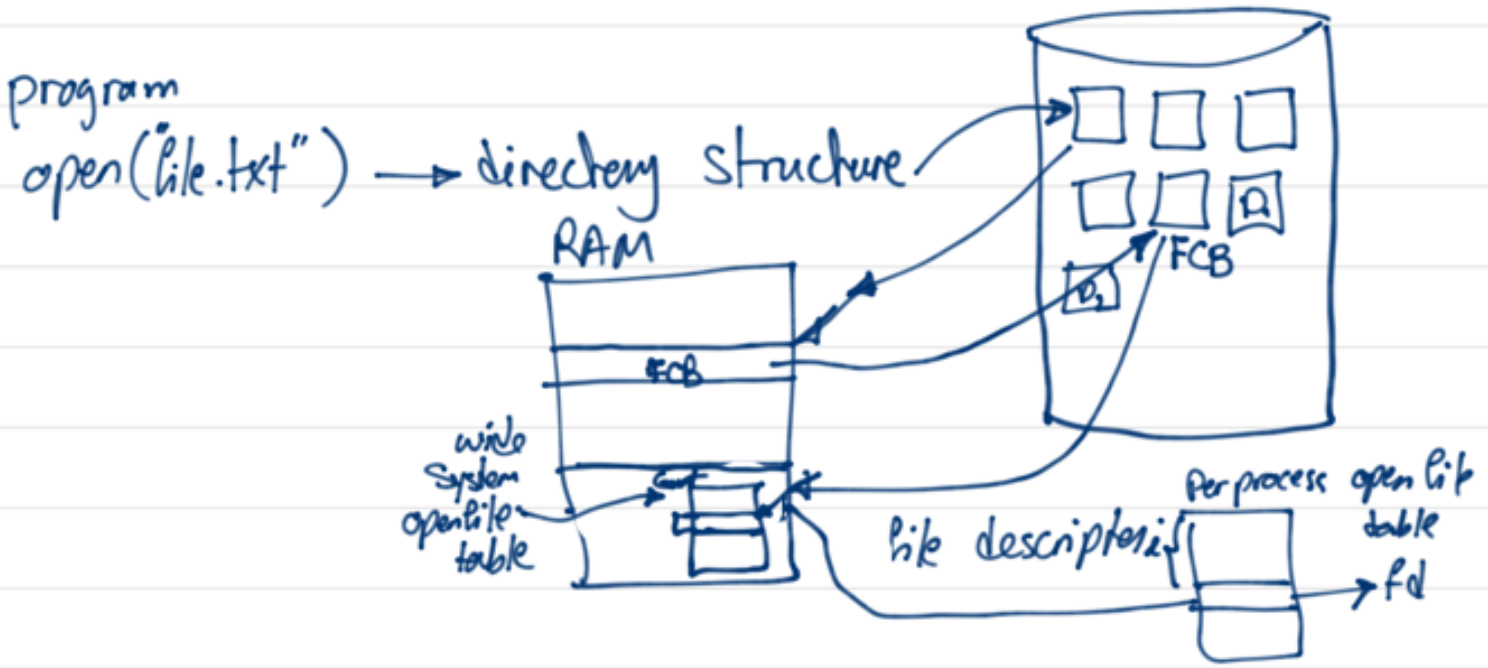
B) Open file:
↳ sent to logical FS. open() first checks the system wide open-file table to see if it is already open → a pointer to that is created.
- if not open ⇒ directory structure (→ partly cached in memory) is searched for filename.

Once the file is found the FCB is copied into a system-wide open-file table in memory.



All file operations are then performed via this pointer.

File name may not be part of the open-file table, as the system has no use for it once the appropriate FCB is located on disk.



What happens when it closes the file?

* Same sys use FS for networking

- Caching: very important (keep all metadata) → in memory

- Disk formatting: 1) Raw: no FS ⇒ swap space/DBs
  2) Cooked: Containing a FS

boot loader:
also boot sector has its own format.
↳ because FS is not loaded yet.
goes to find kernel to boot it.

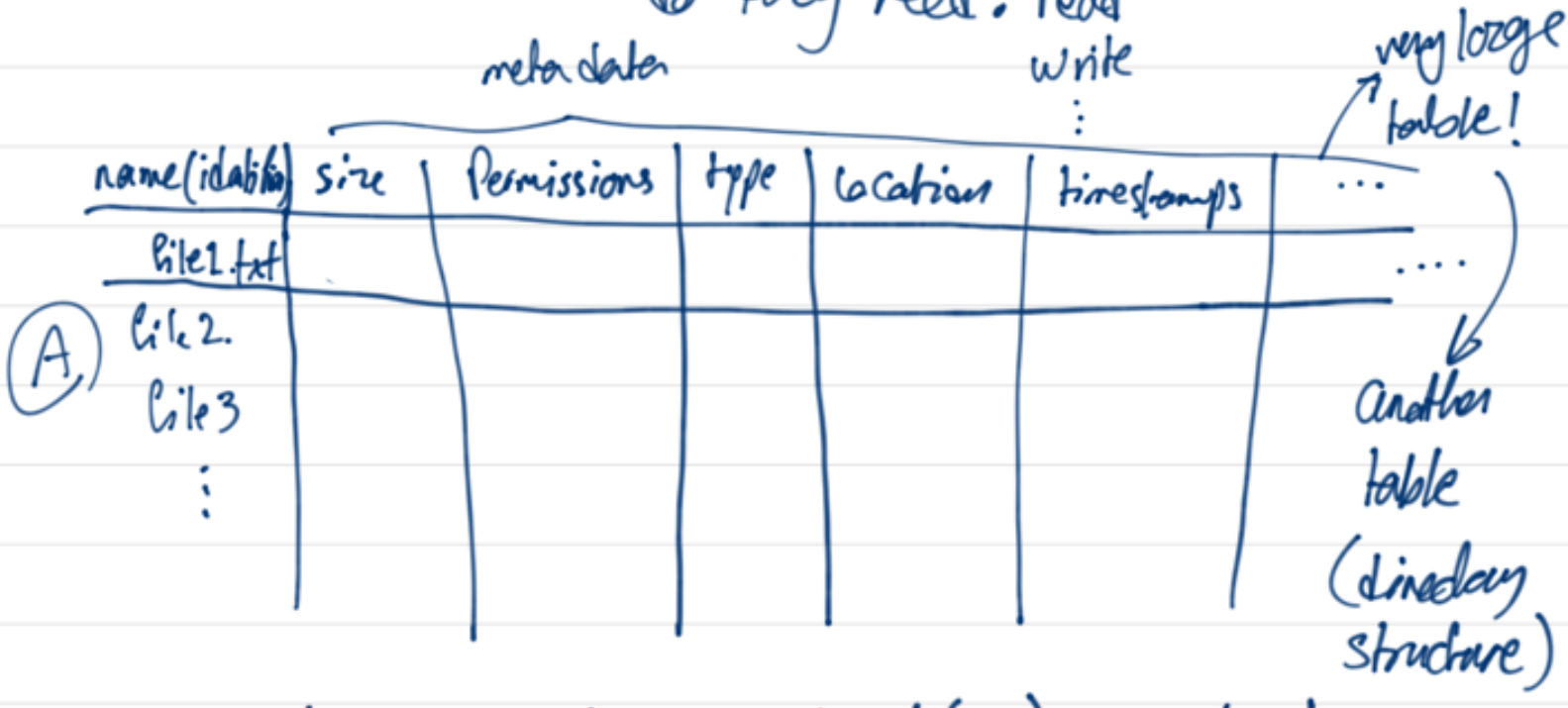
dual-boot.

Device directory: FS info (format)

Let's design a File System:

2 aspects: 1) How can the user access its file through user implementation of file system / what functionalities do they need? read write ⋮

chapter 11 ⟸



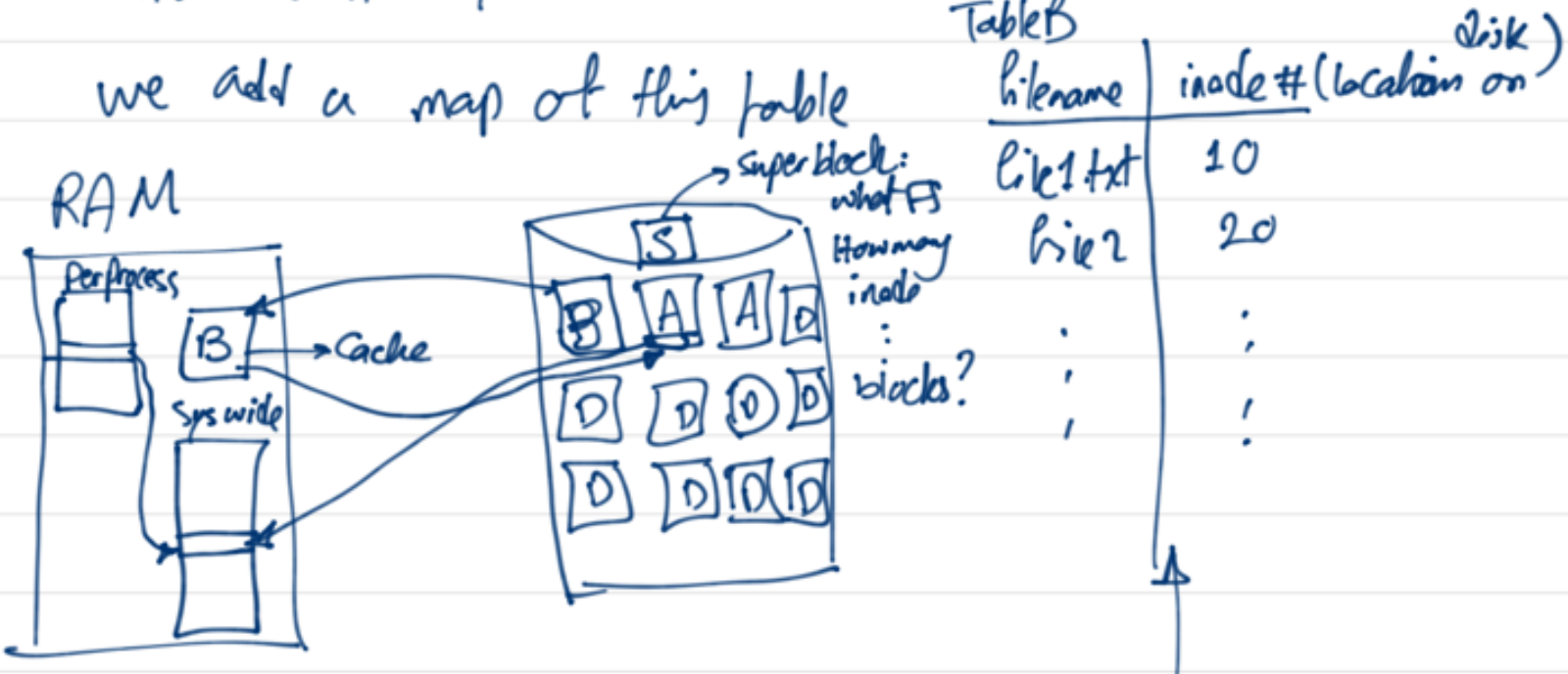
Also we need some caching and internal (OS) data structures as well? like what? System wide page table, per process page table, ...

* OK! now we want to implement this system & where should the table (A) be stored?

Table A is large b/c it contains all we need for a file

→ each entry : File Control Block (FCB) or inode within unix system

we add a map of this table



If user tries to open a file (& open(file2)

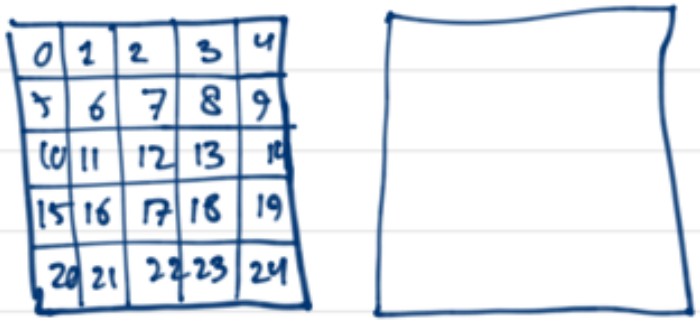
another process open(file2)?

- freelist:

we get inode # from B

A on the disk

each 256 bytes ⇒

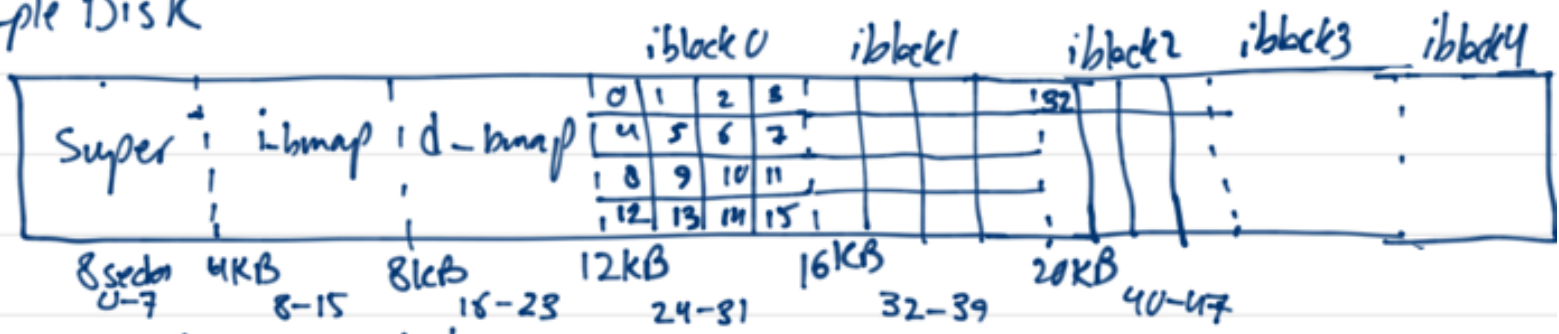


Disk → need to access the secter & block

How does it look on the disk?
Disks are secter addressable (not byte addressable)

a set of sectors → Block

Simple Disk



each inode = 256 byte

each block = 4KB ⇒ 5×4kB = 20KB inode data ⇒ 80 inodes

- we want to access inode #32 ⇒ 32×256 = 8KB

8 KB + 12kB = 20KB. Considering secters of size 512

Disk read ⇒ $\frac{20 \times 1024}{512}$ = Secter 40 $5 \times 8 = 40$ secters

Read secter 40