LECTURE

FS implementation

A simple file System

Last time: FS API
"human name" ⇒ number
types: files, directories, sym links, etc.
low-level name

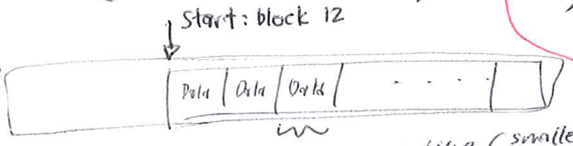
```
access: fd = open("/x/y/z.txt", O_RDONLY);
             read(fd, buf, size) or write();
             close(fd);
```

Today: FS implementation (round 1)
start
FS: (on-disk) data structures + access methods

Recall: disk interface (just an array of blocks)

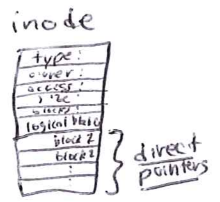
First: need place to keep user data
(data blocks)

Q) given data block #, how to read data block?



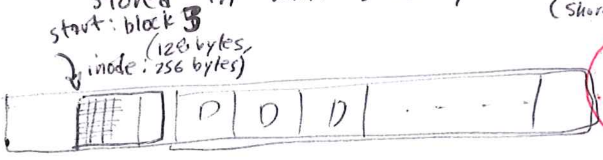
block size: 4KB (smaller, bigger?)

Second: per-file metadata
(info about file, such as type, ~~not~~ human-readable name
size, #blocks, perms, ownership, ... ptrs to blocks)



stored in what is usually called an "inode"
(short for index node)

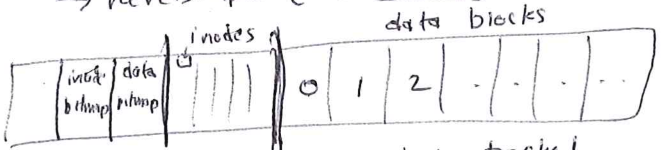
Q) given inode #, how to access inode?



Third: need place for directories
but: directory is just a file!
→ has inode (type = dir)
→ contents are simple:

name	inode #
.	10
..	2
foo	16

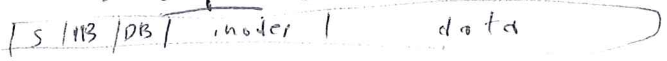
Fourth: need way to track which inodes, data blocks are (free, allocated)
→ lots of ways possible: free list, etc.
→ here: simple [the bitmap]



bitmap: for each thing tracked, corresponding bit in bitmap is (0, 1)
(free, used)

Fifth: need to know how big fs is,
how many inodes in inode table,
how many data blocks in data region,
what type of fs is this, etc.

⇒ superblock: meta info about fs



break: now you understand basic on-disk structures

1) Create an "empty" file system

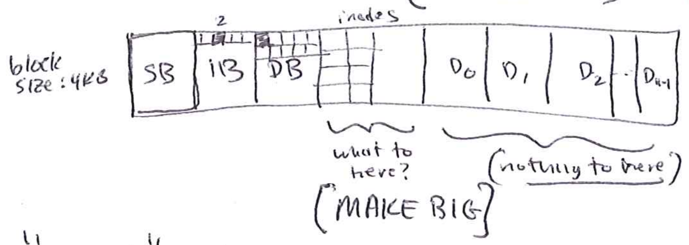
=> tool : mkfs "make fs" , per $\overset{\text{one}}{}$ fs type (ext2, jfs, xfs, zfs)

give : disk partition

mkfs : writes stuff to it

size of regions, [root directory]

Q) what do we need to write?



recall : all dirs have

2) "mount" file system

minimally, read in super block to get needed info

3) access :

a) create file

fd = open ~~create~~ ("/foo") =>

, O_WRONLY | O_CREAT

what happens ?

need to :

1) read root dir ("/") => why? → ① check permissions → ② check uniqueness of name foo

i) permissions : $\boxed{\text{read}}$ inode of root dir

ii) unique : $\boxed{\text{read}}$ data block(s) of root

2) need to : actually create the empty file "foo"

i) create inode : $\boxed{\text{read}}$ inode bitmap

find free spot in it, mark 1

$\boxed{\text{write}}$ inode bitmap

ii) "link" it into root directory

$\boxed{\text{write}}$ to root dir data

$\boxed{\text{write}}$ to root dir inode (new size, ~~access~~ times modify)

3) create in-mem desc fd => open file table → in-memory inode structure

return it

b) write to file

write (fd, buffer, 4KB);

1) if not an over-write allocate new data block

$\boxed{\text{read}}$: data bitmap

mark it allocated

$\boxed{\text{write}}$: data bitmap

2) update inode :

$\boxed{\text{read}}$ inode

(update) → change (size, # blocks) modify time, create time

$\boxed{\text{write}}$ inode

3) write data : $\boxed{\text{write}}$ data block

c) close file

free in-memory descriptor
could free file struct from open file table } no disk activity

Q) what about long pathnames?

e.g.) "/a/b/c/d/x.txt"

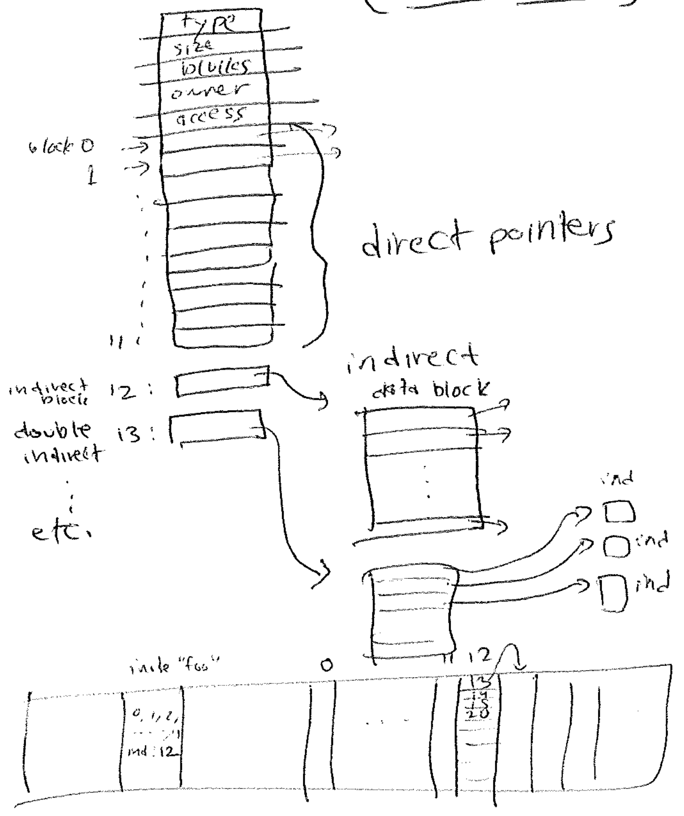
when you open(↑ , O_RDONLY)

what do you have to read?

- root inode (perm, location of data blocks)
  - ~~dir~~ data
- a's inode
  - data (looking for b)
- b's inode
  - data (looking for c)
- c's inode
  - data (looking for d)
- d's inode
  - data (looking for x.txt)
- x.txt's inode : final permission check

"path traversal"

Q) what about large files? (inode design)



(or linked files
or extents)

Q) how is this efficient at all?

page cache

some blocks are

path traversal — reads: read a lot => keep in memory

open (create) write()..., close() => just stays in memory — writes: can write to memory, return success; later, fs writes to disk in background (periodically)

trade-off?

old: fixed size fs "buffer cache"

new: all pages
→ VM
→ FS
} => unified page cache