Hello! 4/17

Welcome to the CS Picnic!

File Systems : Implementation

- Simple : Very simple File system (VSFS)
- Locality : Fast File System (FFS)
- Crash Consistency : Journaling (JFS)

(Write Ahead . ext3/4

(... Logging) Linux

Prof <——> Student

↓ → minimize ↓

→ **Write Performance:**

Logging →

Log-Structured File System (LFS)

→ Flash-based Solid State Drives (SSDs)

→ [The End] ⇐ future: sadness

Abstraction:

→ Files ?

→ Directories]

File: An array of bytes

has low-level name (number) ⇒ { operations: open, read/write, unlink, close }

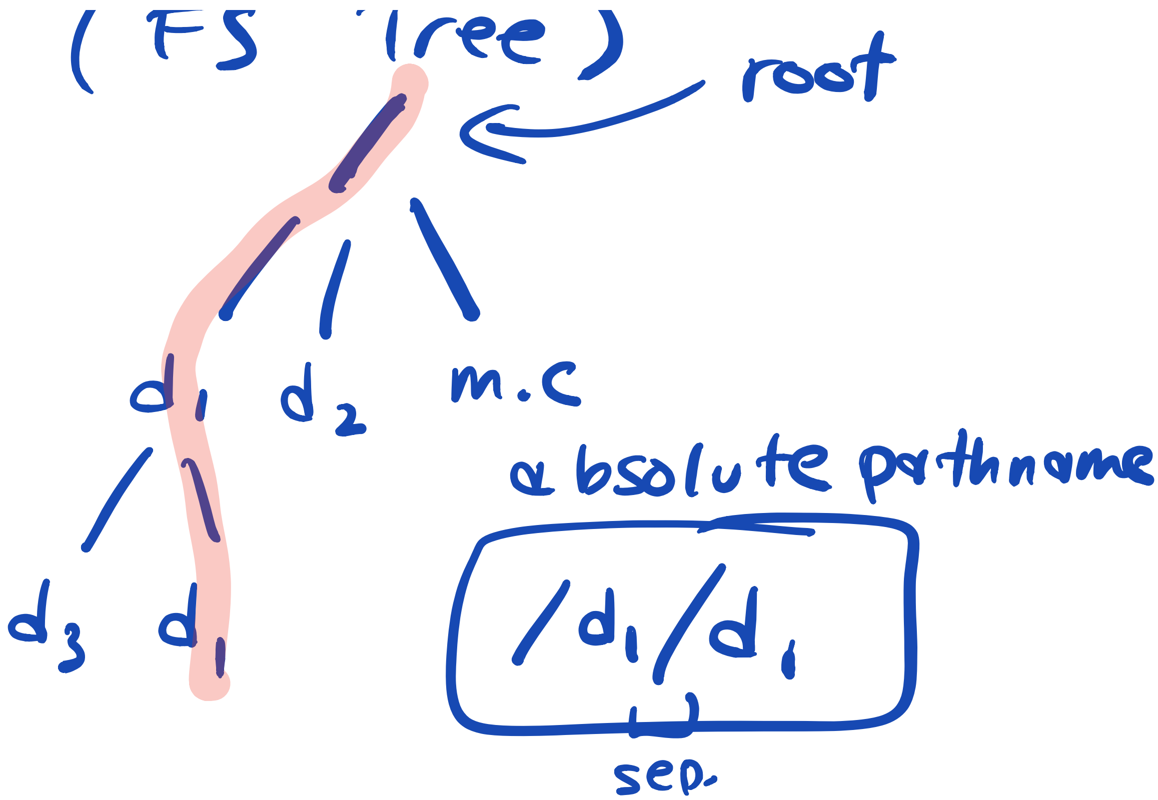
Directory:
list of <names low-level name> (of files, dirs)

one file:
FS → file 3

"main.c"

dir: maps
"main.c" → 3

Directory Hierarchy:

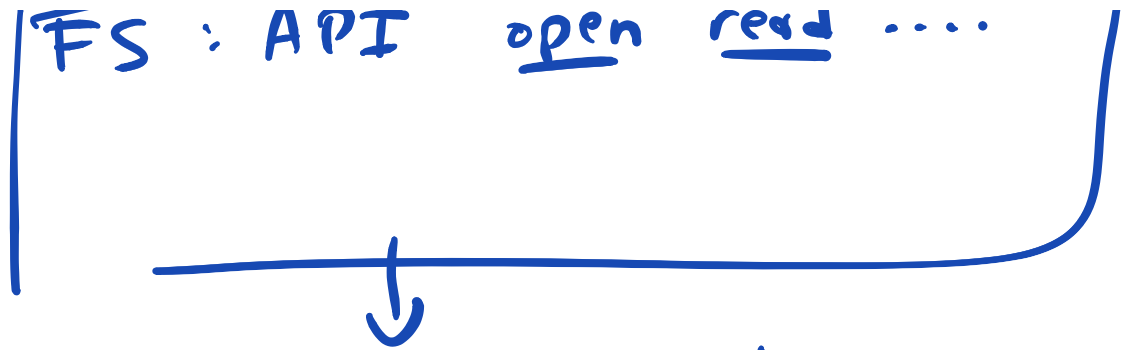


Implementation:

- → On-disk Data Structures
- → Access Methods
  open, read/write, etc.

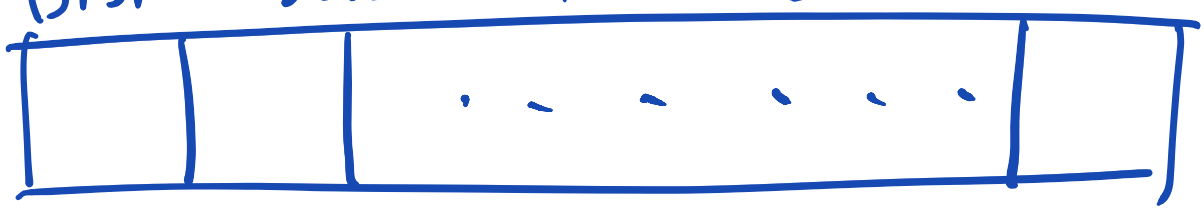
Very Simple File System

(VSFS)



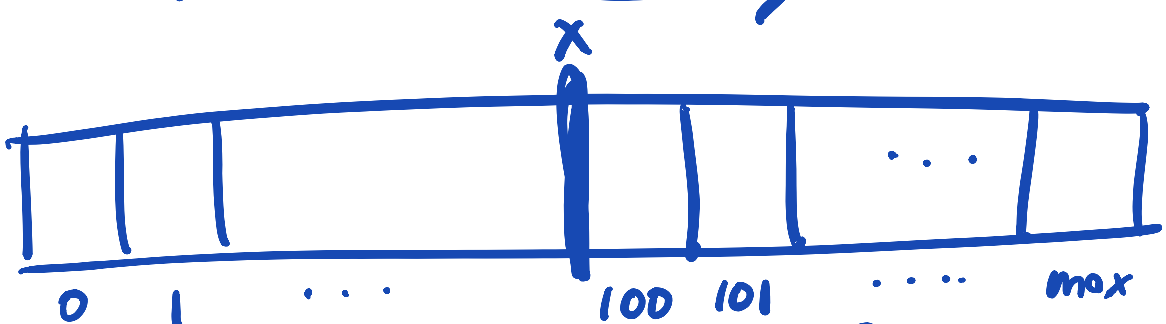
read/write to
disk blocks

Disk : Bunch of Blocks



what needs to be
on disk?

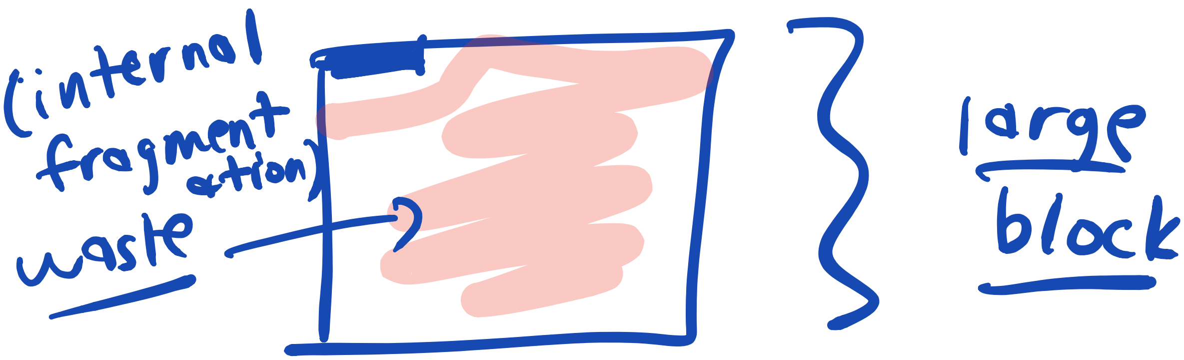
→ Data "data → blocks"



Block size: ⇒ 4KB
must be multiple
of sector size (512b)

↳ minimal unit of allocation → file
→ directory

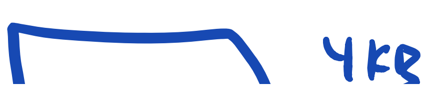
main.c :
#include <stdio.h>

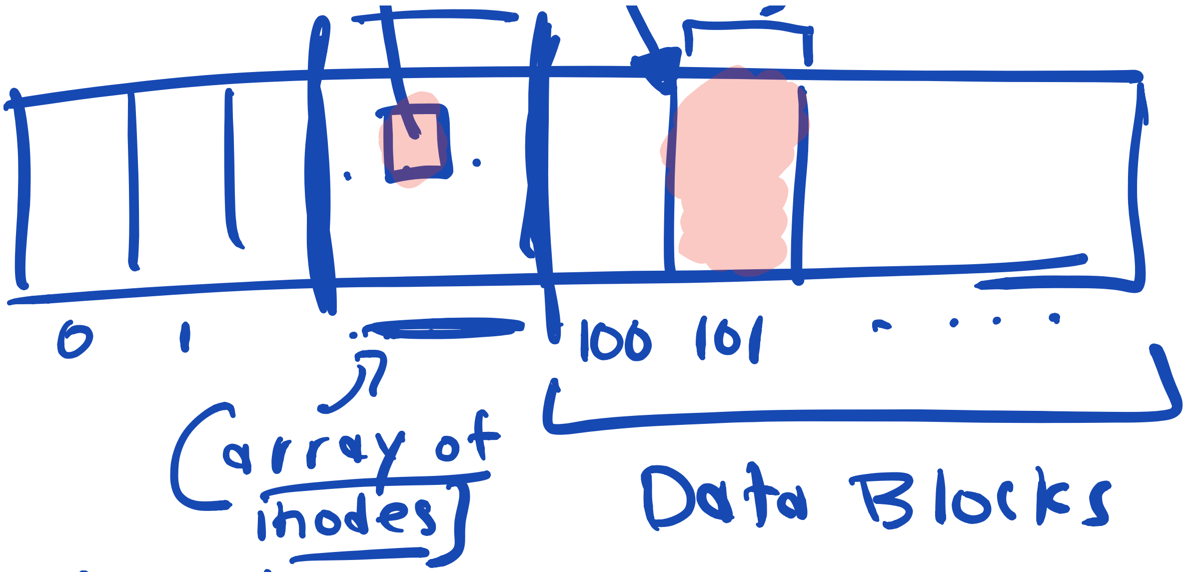


small blocks:
better : less internal waste

but: lots + lots of blocks in large file

Blocks : (4KB) ⇒ also : matches page size





1) Data

2) Info about each file

per file: structure on disk

inode (index node)

→ what's inside?

- → type: regular file, directory
- → ownership: remzi
- → permission: who can read/write/exec.
- → timestamps

size # of blks

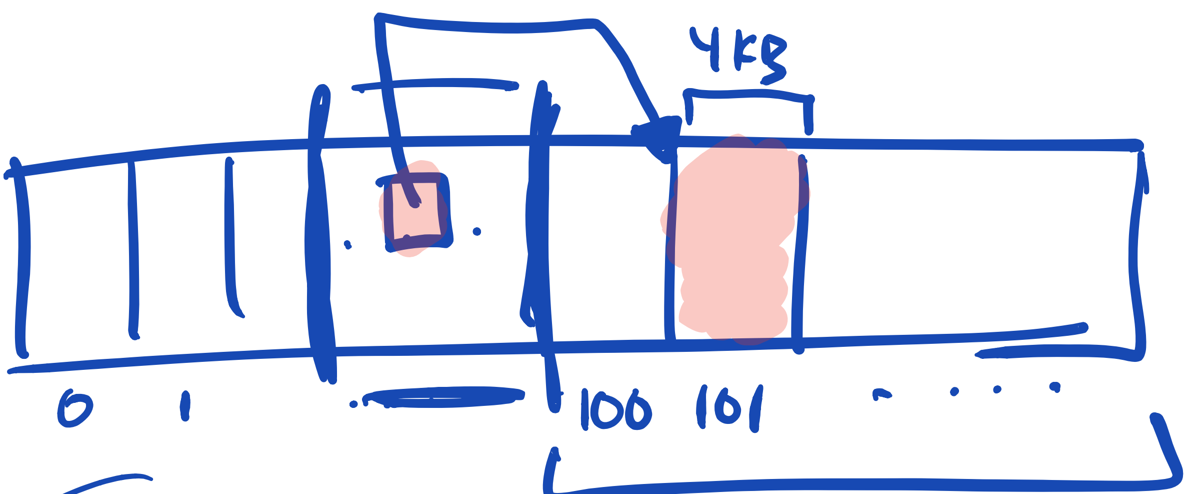
→ size, # of blks

→ location of blocks of this file/dir

array of inodes:

0 . . . . . . N-1 ← index value

↑ low-level name ⇒ inode # (index value)



given inode #:

=> **size** 128 b, 256 b

=> **base** (where array starts)

1) **data** (file, dir)
2) **inode array** aka (inode table)
3) **directories**: treated just like files
   => inode → data blocks

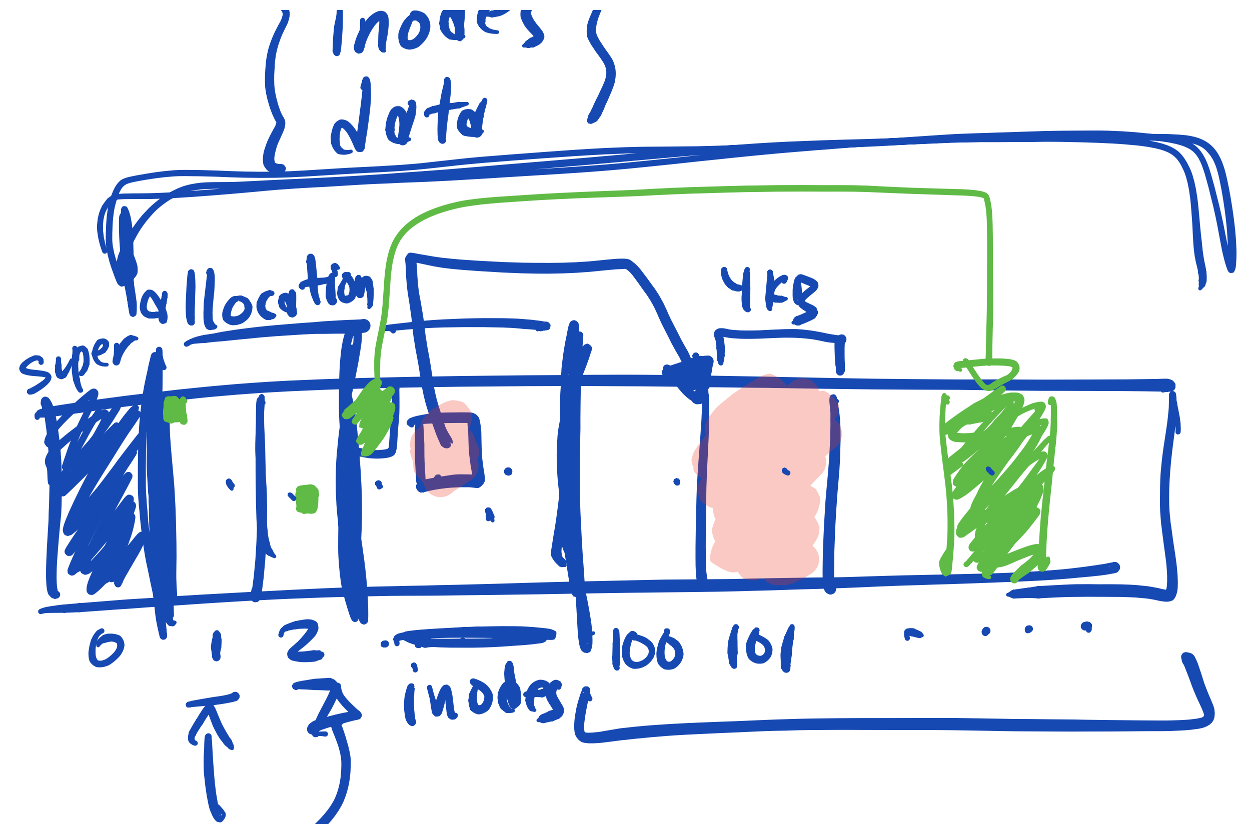
all dirs:
- .. parent
- . this dir
- ⋮

data →

..	10
.	100
foo:	101

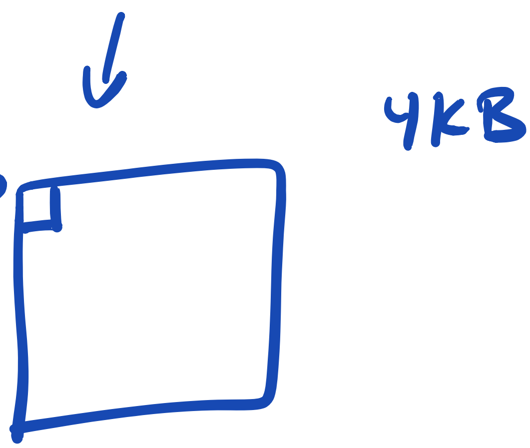
name (human) ↑ low-level (inode #) ↑

4) track free/allocated:



1: inode bitmap

2: data bitmap



0 or 1 → whether the corresp. entity is free (0) / in use (1)

1) data 2) inode
3) dir 4) bitmaps

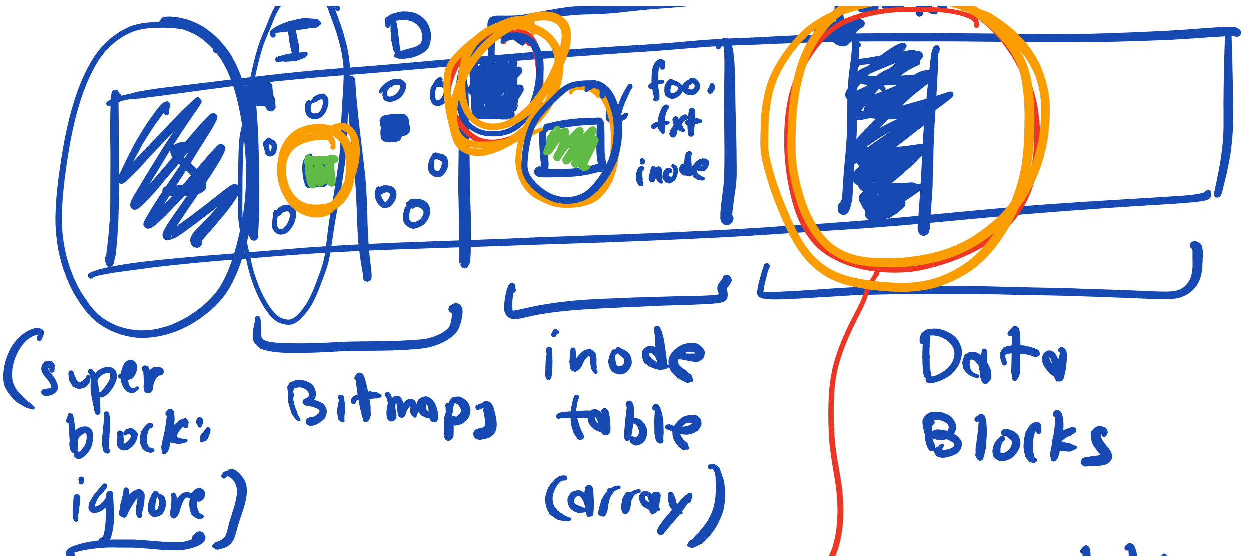
5) superblock :
per file system info.
→ block size, type: ext4
[where is inode table,
where are data blocks?
etc.]

Access Methods :

API: → what it does
to disk?

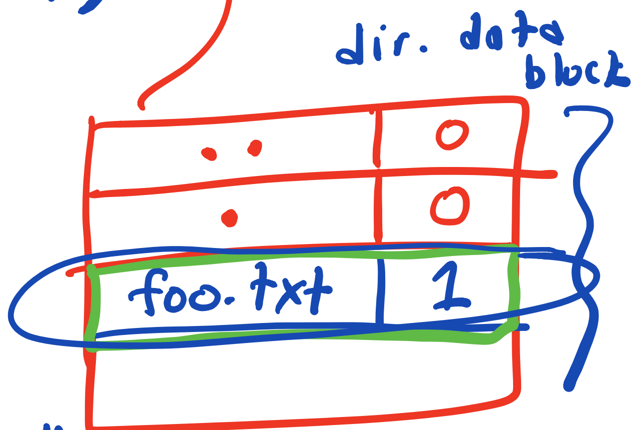
creat("/foo.txt")

Assume: Empty File System



create /foo.txt

=> read inode of root directory



know: root inode # => 0

1) READ: block containing inode #0 (root inode)

(permissions check)

2) make sure filename is unique:

READ(s): data blocks

READ (s) ... of root directory

3) look for space for inode:

READ inode bitmap

4) mark bit (in use) [in memory]

5) WRITE inode bitmap

6) update inode of foo.txt

READ inode block table

update in memory: [. . . . .]

WRITE block (inode) of foo.txt

7) link file into directory

READ root data

update [foo.txt → 1]

directory entry

WRITE dir block

=> creation: finished

=> write()

=> allocation or not?

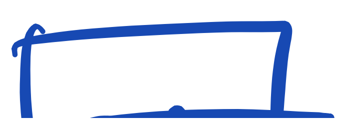
=> open ("/a/b/c/...")

[path traversal]

open ("/d_1/d_2/d_3/foo.txt")

root dir

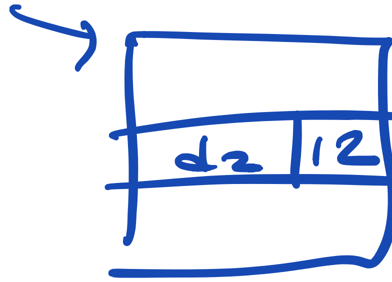
read root inode

read root dir → 

data

read d_1's inode

read d_1's data

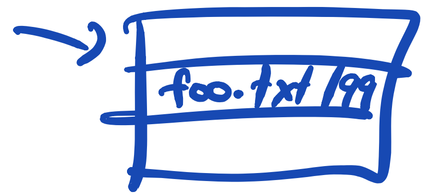


read d_2's inode

read d_2's data

read d_3's inode

read d_3's data

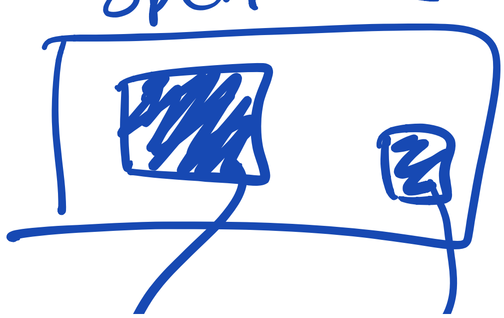


read foo.txt's inode

per-process:

file descriptor (int)

open file table

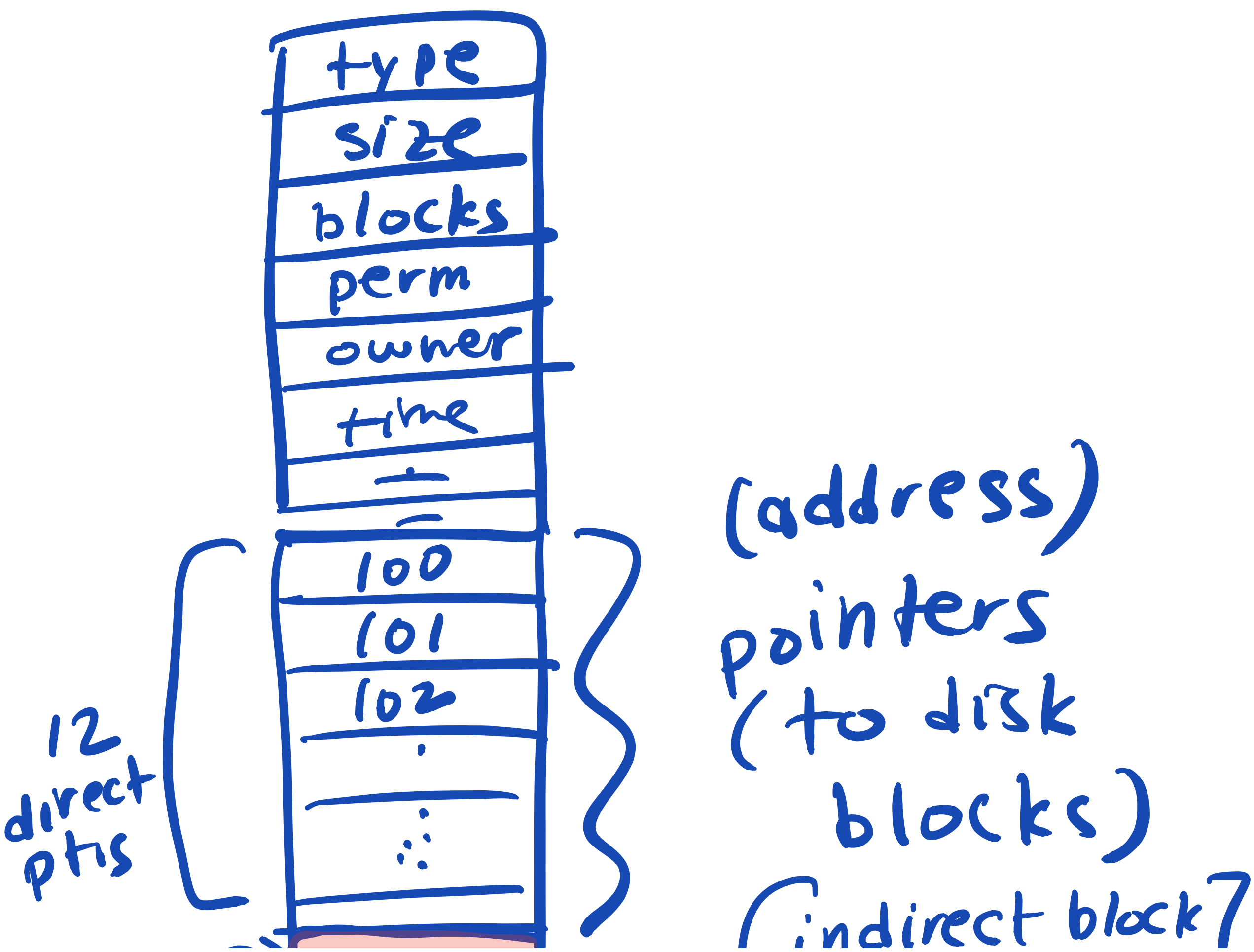


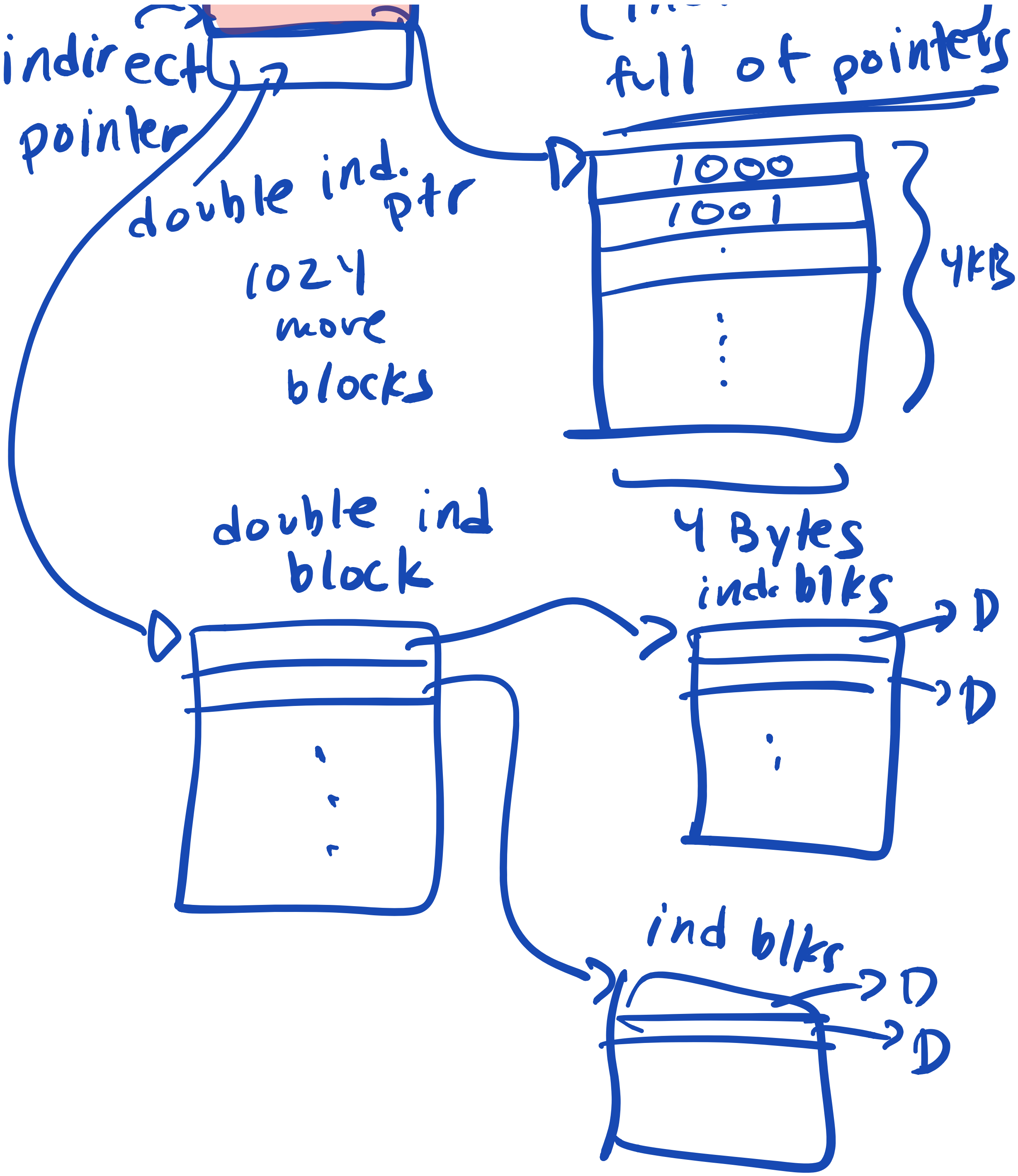
(in-memory inode)

=) small files (so far)

what about large files?

inode structure: 128 bytes





Mapping : File => Block

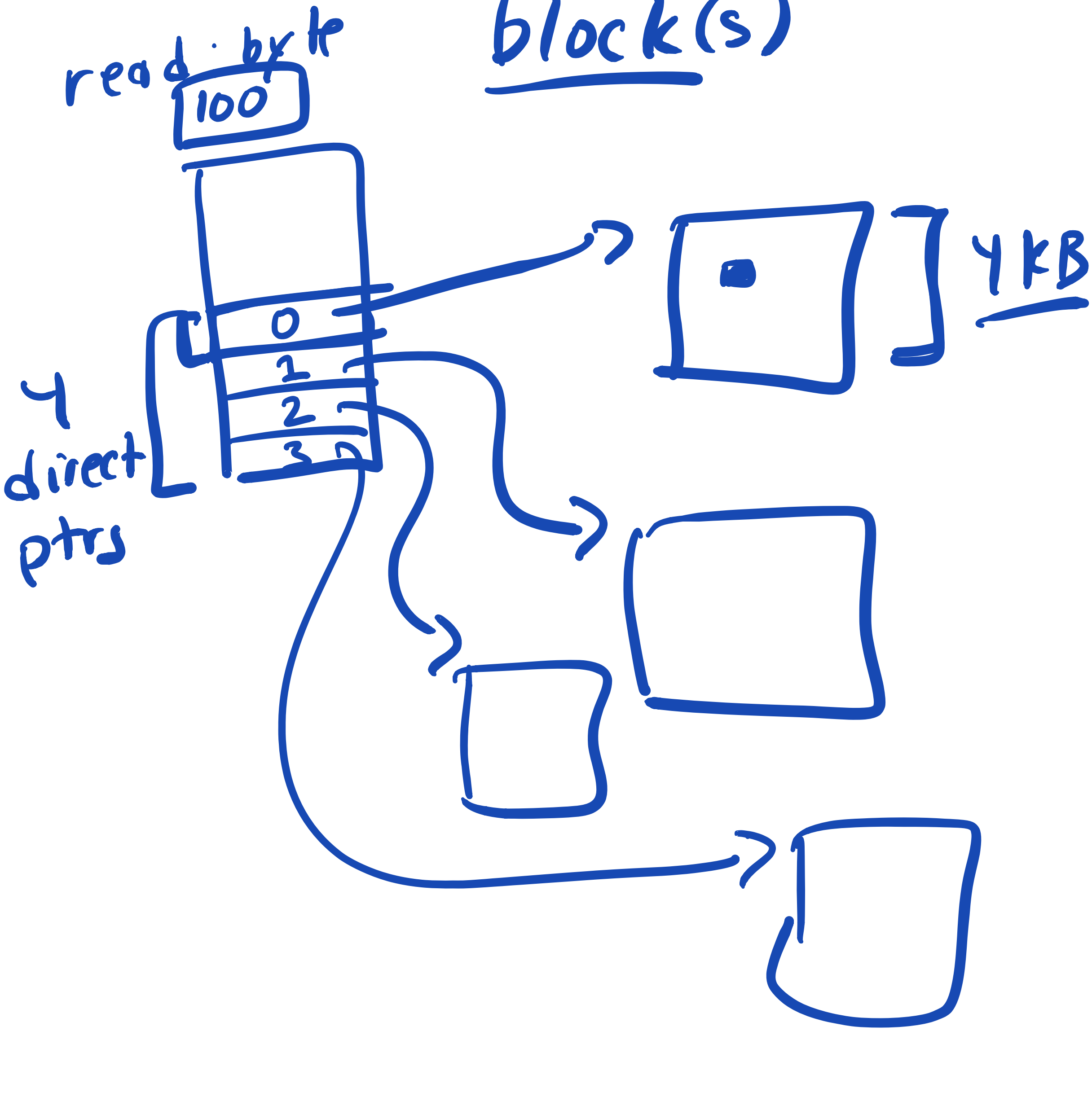
read / write

starts @

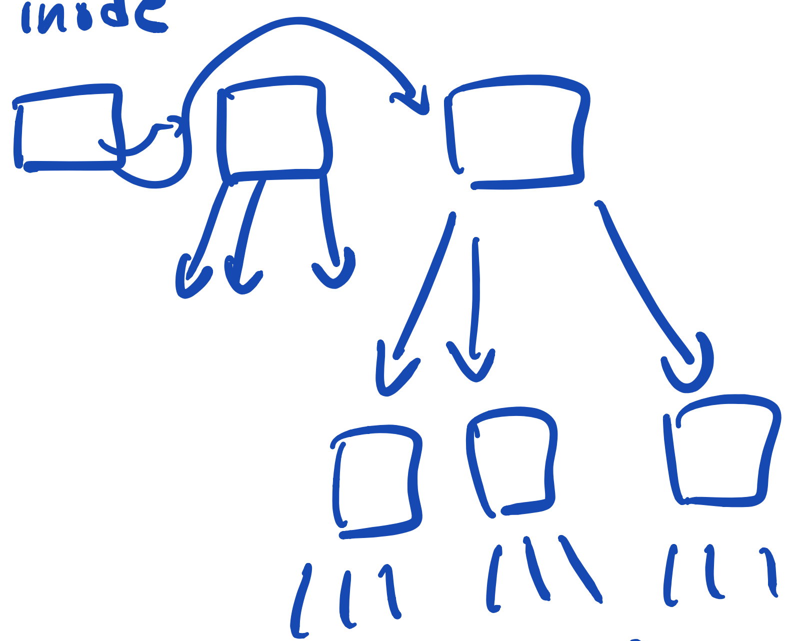
⇒ (current offset) 0

(byte offset)

FS : translating byte offset ⇒ block(s)



FS inode



(imbalanced)

[most files are small]

→ Caching (+ write Buffering)

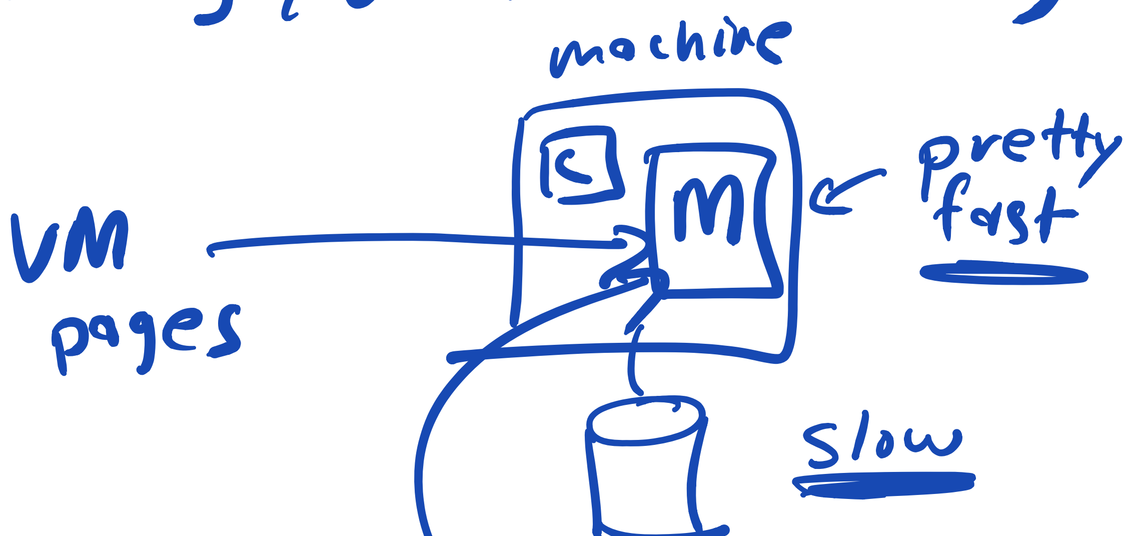
→ Project 4a ⇒

PS yes 1-0

Everyone gets 100!
here

sign here →

Caching / Write Buffering



⇒ Use memory (for FS stuff)

Important on reads:

1) open ("/a/b/c/d/e.txt", O_RDONLY)
⇒ slow

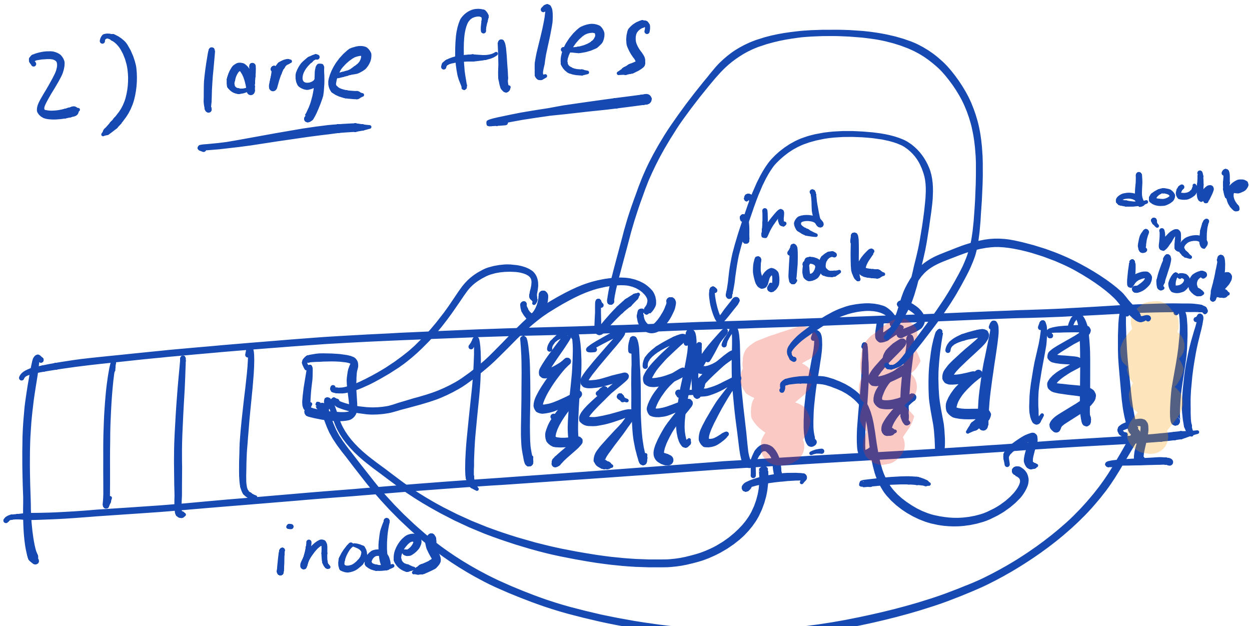
many reads

cache inodes (metadata), dir data

(in memory)

re-open file again ⇒ fast

2) large files



read from large file

⇒ might not IO

read

double ind → ind block → data

cache

3) cache data too

writing:

```
open("/foo", O_WRONLY)
write( . . . )
  :
write( . . . . )
close
```

FS has done no I/O to disk

=> buffering data in memory

Later, in background, OS writes out data → disk

Why useful?

=> seems faster (decreases perceived latency)

=> leave it in memory: might be read later

=> aggregate many "small" writes => 1 block write (batching)

=> overwrite same block repeatedly

(reduces I/O)

⇒ disk scheduling
(better w/ more writes!)

⇒ open
write
⋮
write
close
⋮
unlink
(delete)

if file deleted, never write to disk at all!

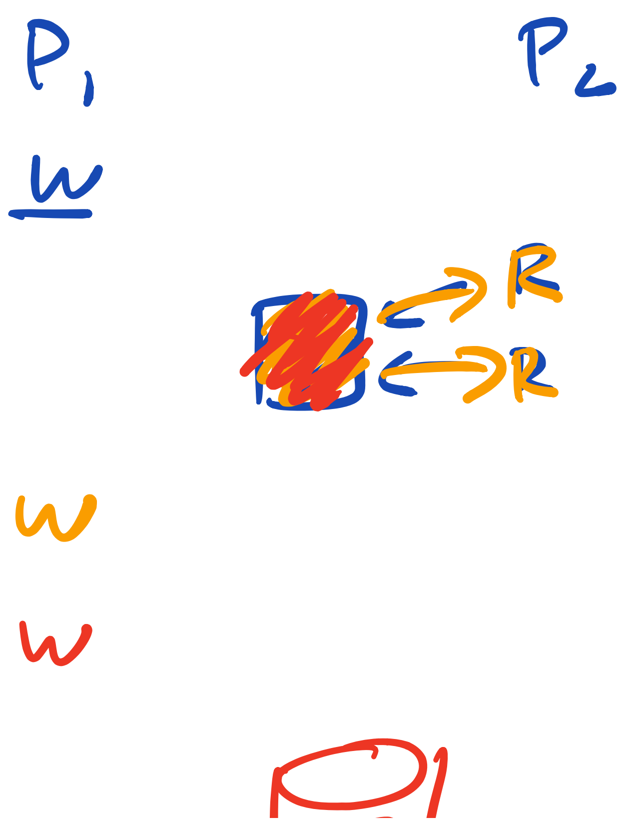
why bad?

⇒ Crash ⇒ Data Loss!

if you really care about immediate writes:

```
open (     )
write (   )
   ⋮
write (   )
fsync (     )
```

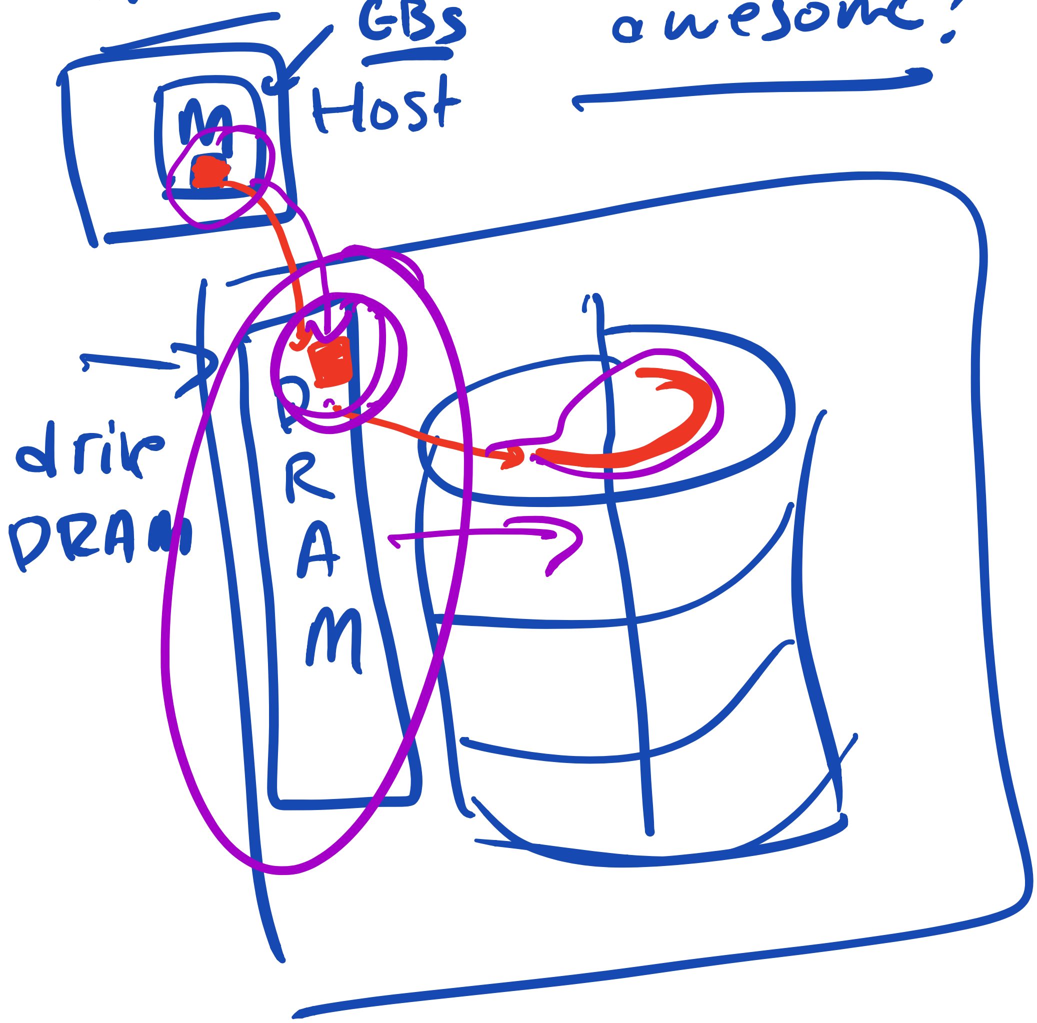
=> forces data to disk (SLOW) but safe



FS cache: Shared across all processes

[2]

Mac: why are macs awesome?



Discussion:

MapReduce Q/A

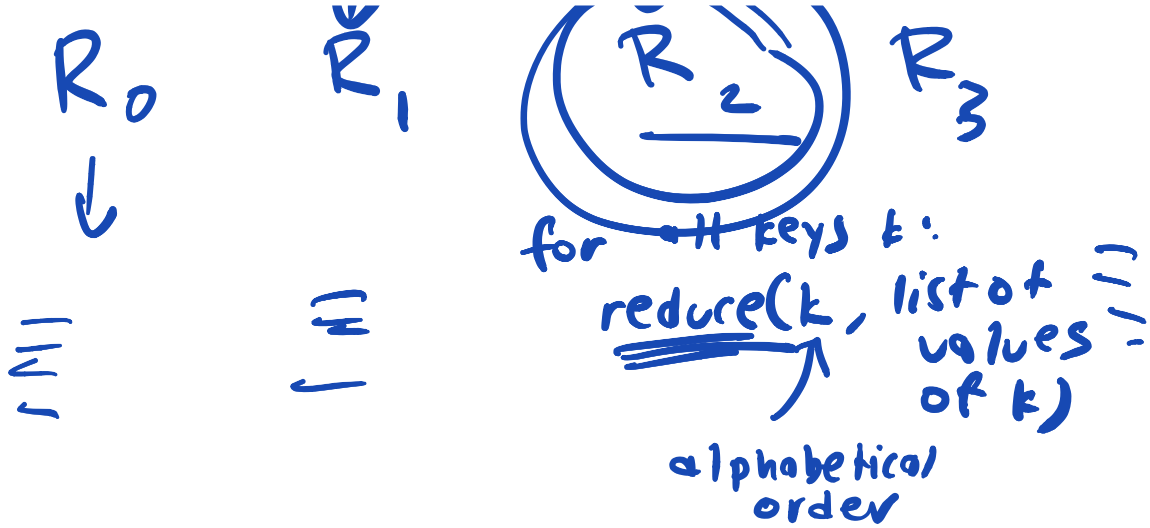
=> Project reason:
citations

=> Structure of map-reduce problems

=> more examples

=> Perf : [~25%]

=> Sort :



wait for all reducers

→ clean up

→ return