Filesystem → OS subsystem → secondary or smaller systems part of a larger system

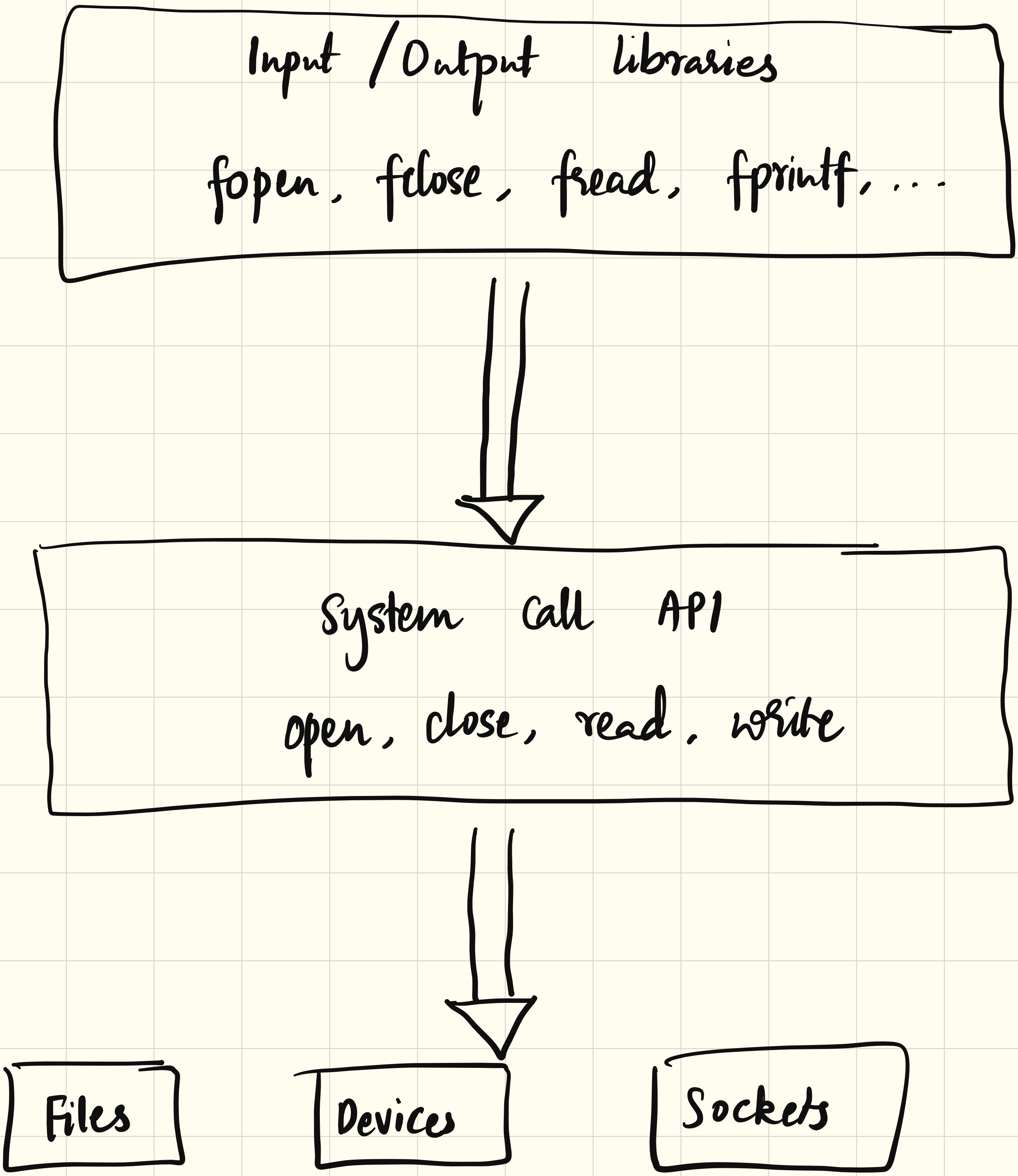
→ provides abstraction like files and directories

→ hides complexity of underlying storage devices

Filesystem Interfacing

→ Processes identify files through a file handle / file descriptor.

→ In UNIX → the POSIX API is used to access files, devices, sockets, etc.


Input /Output libraries
fopen, fclose, fread, fprintf, ...
System call API
open, close, read, write
Files
Devices
Sockets

What is the mapping b/w library functions and system calls?

fopen( ) → open( )

fclose ( ) → close ( )

⋮

---

open : getting a handle

int open (char *path , int flags , mode_t mode)

→ Access permission check performed by OS (ls -al)

access mode

O_RDONLY

O_RDRW

O_WRONLY

O_SEARCH

O_EXEC

if flags contains O_CREAT specify file creation (rwx) using mode (owner, group, others)

⋮

open → returns a non-negative integer

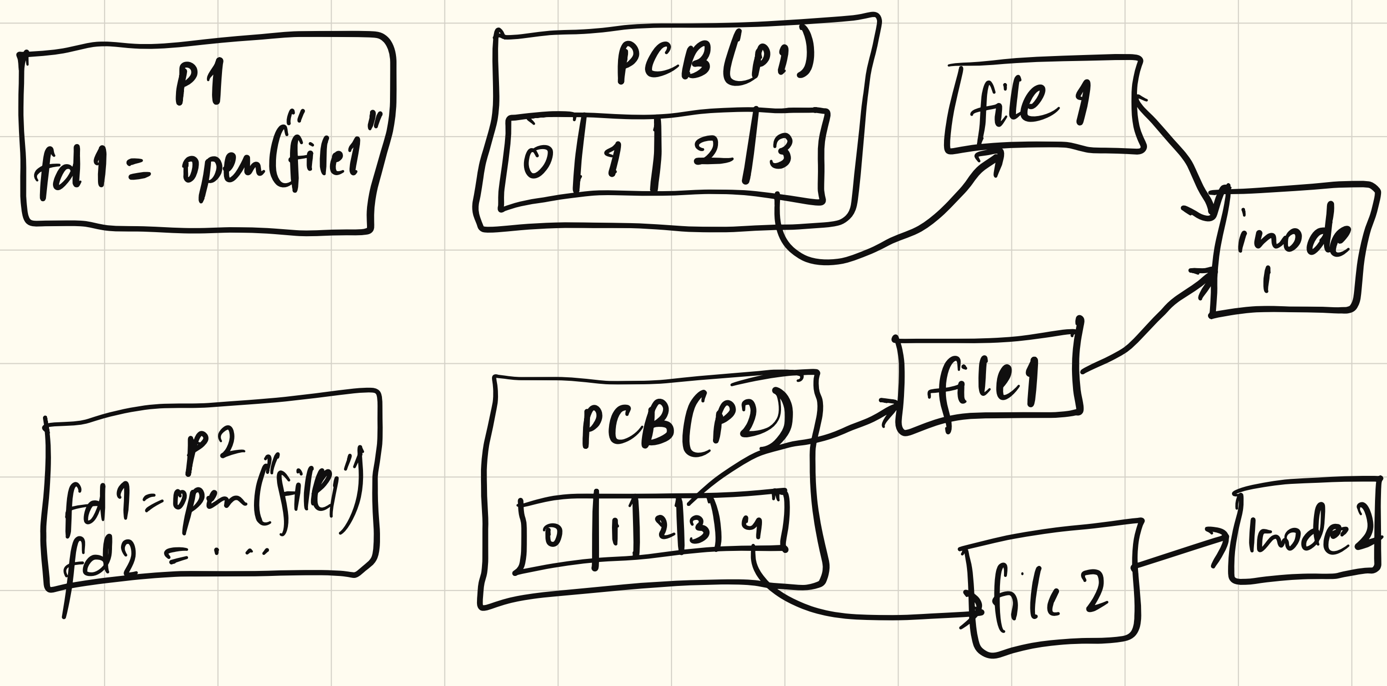
↓

file descriptor

-1 on error

sets errno to indicate error

Process View of File



→ Per process file descriptor table with pointer to a "file" object.

→ file object $\xrightarrow[\text{one}]{\text{many}}$ inode

Questions:

1) What do file descriptors 0, 1 and 2 represent?

2) What happens to the FD and the file objects across fork()?
   - exec?

3) Can multiple FDs point to the same file object?

Read and Write

```
ssize_t read ( int fd , void * buf,
                size_t count );
```

fd → file handle

void * buf → pointer to the buffer where read data will be stored

size_t count → # of bytes to read

read( ) → returns # of bytes actually read
→ can be smaller than count

0 → EOF reached
-1 error

ssize_t write (int fd, void *buf, size_t count);

pointer to the data to be written

no. of bytes to write from the buffer

→ return no. of bytes written

File descriptor :

0 → STDIN

1 → STDOUT

2 → STDERR

lseek

off_t lseek (int fd, off_t offset, int whence);

file handle

off_t → signed integer type

offset → target offset

whence → SEEK_SET, SEEK_CUR, SEEK_END

On success, returns offset from the starting of the file

Examples

→ lseek (fd, 100, SEEK_CUR) → forward the file position by 100 bytes

→ lseek (fd, 0, SEEK_END)

↓

file position at EOF
returns the file size

→ lseek (fd, 0, SEEK_SET)

↳ file position at the beginning of the file

File Information (stat, fstat)

int stat(const char * path , struct stat *sbuf);

→ returns information about the file pointed to by path

→ Information is filled up in the structure stat.

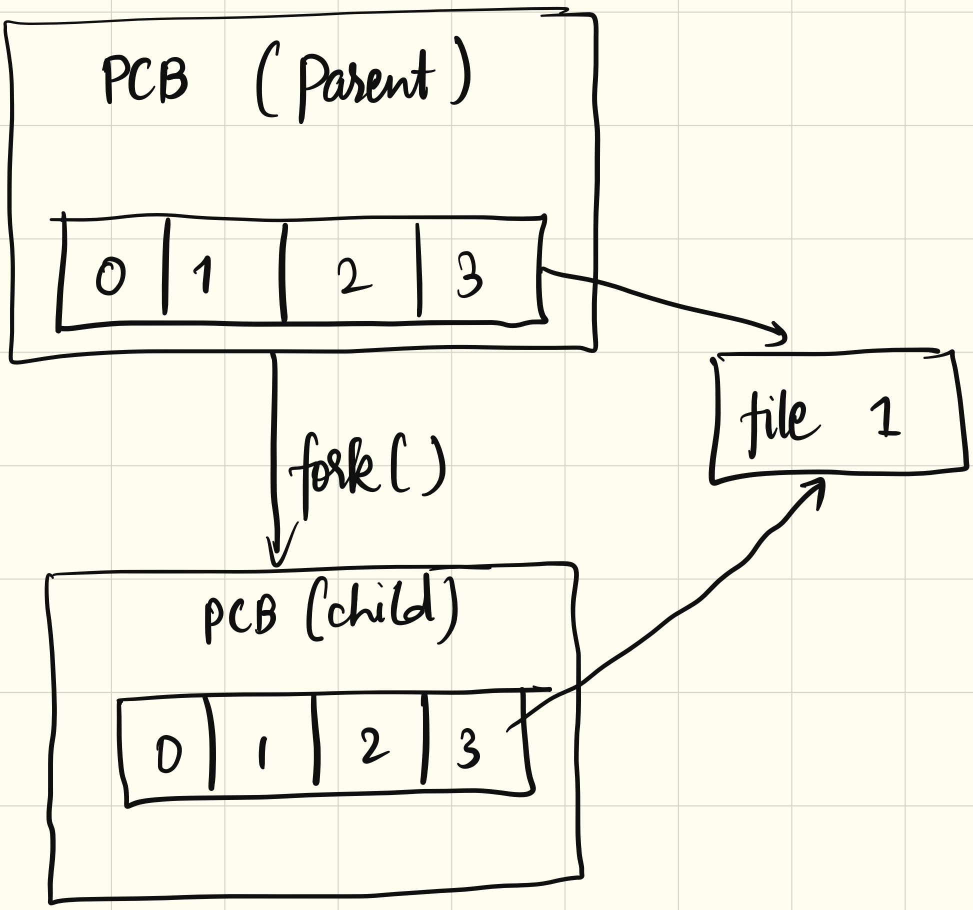
Example:

```
struct stat sbuf;
stat ("/home/user/tmp.txt", &sbuf);
printf (" inode = %d, size = %ld \n",
            sbub -> st_ino, sbuf -> st_size);
```

other useful info in stat

st_uid, st_mode

Process view of file after fork()

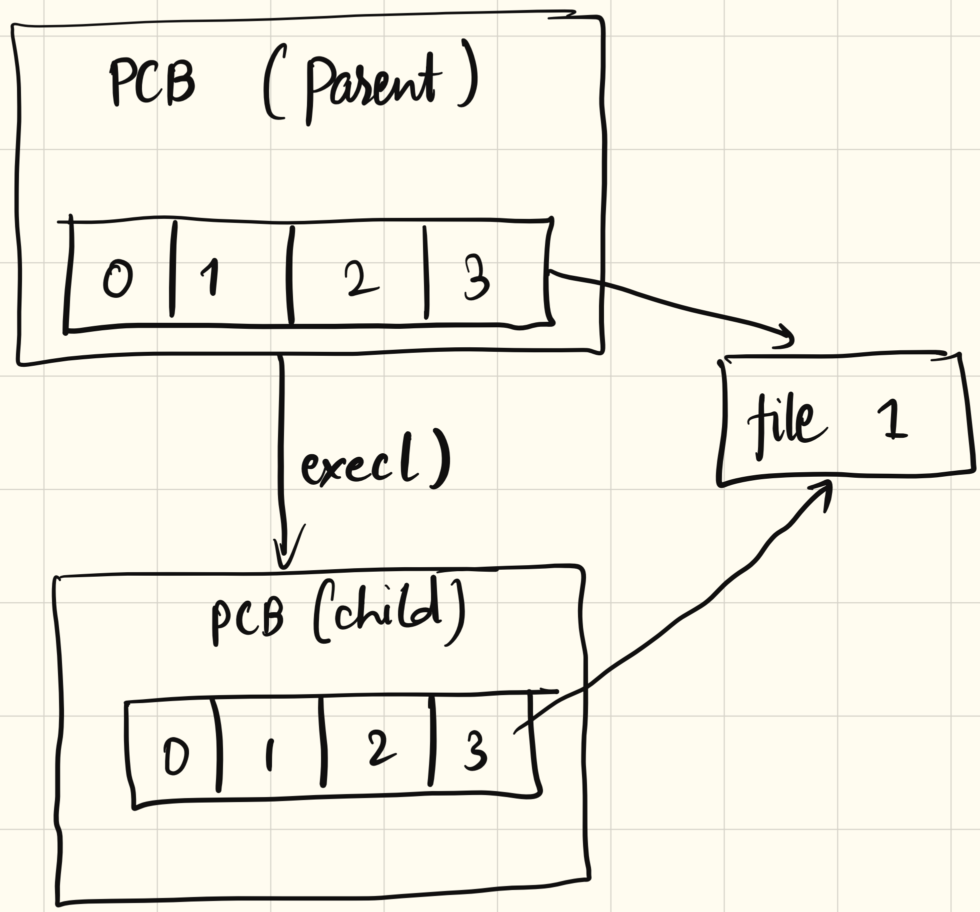


→ On fork(), child inherit the parent process state.

→ All file descriptors remain open in the child

→ FDs point to the same file object.

Process View of file after exec()



- On exec(), destroy memory state of calling process to load new binary
- By default, it does not destroy FD tables
  - FDs point to the same file objects as pointing earlier, and remains open

→ To close FDs on exec, specify O_CLOEXEC flag during open

* What happens to FD and the file objects across
  - fork?
    → the FD table is copied across fork → file objects are shared.
  - exec?
    → Open files remain shared by default.

Duplicate file handle (dup and dup2)

```
int dup ( int oldfd );
```

→ The dup() system call creates a copy of the file descriptor oldfd.

→ Returns the lowest numbered unused descriptors as the new descriptor.

→ Old and New FD → represent the same file

```
int fd, dupfd;
fd = open (" tmp. txt ");
close ( 1 );
dupfd = dup (fd);
printf (" Hello World \n");
```

dupfd = dup (fd); → dupfd will be 1 (assuming STDIN = 0 is open)

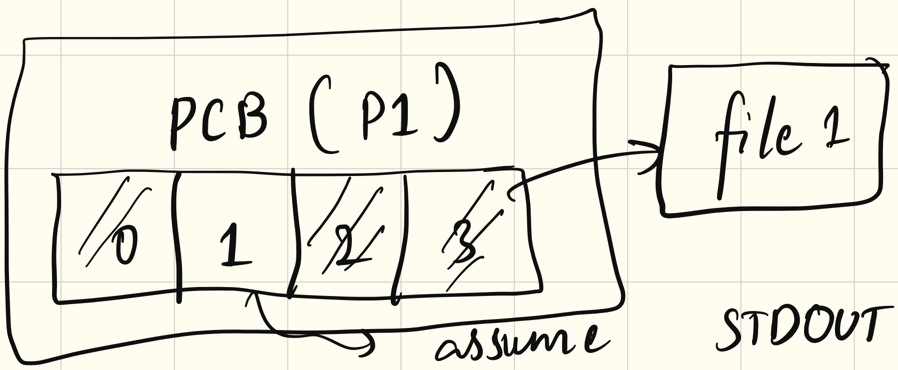
printf ("Hello World \n"); ↳ Will be written to tmp. txt

P1

fd1 = open("file 1");

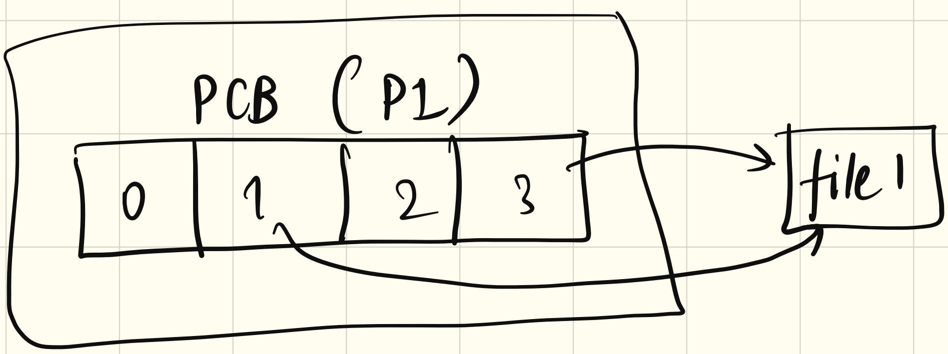
+ dup(fd1)

Before dup()



assume STDOUT is closed before

After dup()



→ Duplicate descriptors share the same file state

→ Closing one file descriptor does not close file1 int dup2 (int oldfd, int newfd);

→ close newfd before duping the file descriptor oldfd

→ dup2 (fd, 1) equivalent to
  → close (1);
  → dup (fd);

Use of dup ( ) : Shell redirection

Example : ls > tmp. txt

Implementation :

```
fd = open ("tmp.txt");
close (1);          // close STDOUT
close (2);          // close STDERR
dup (fd);           // 1 -> fd
dup (fd);           // 2 -> fd
```

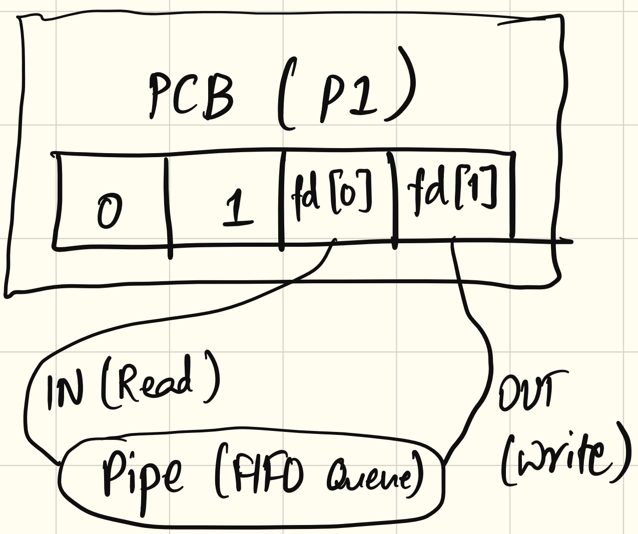
exec ("ls");

UNIX pipe system call

P_1

int fd[2];

pipe (fd);



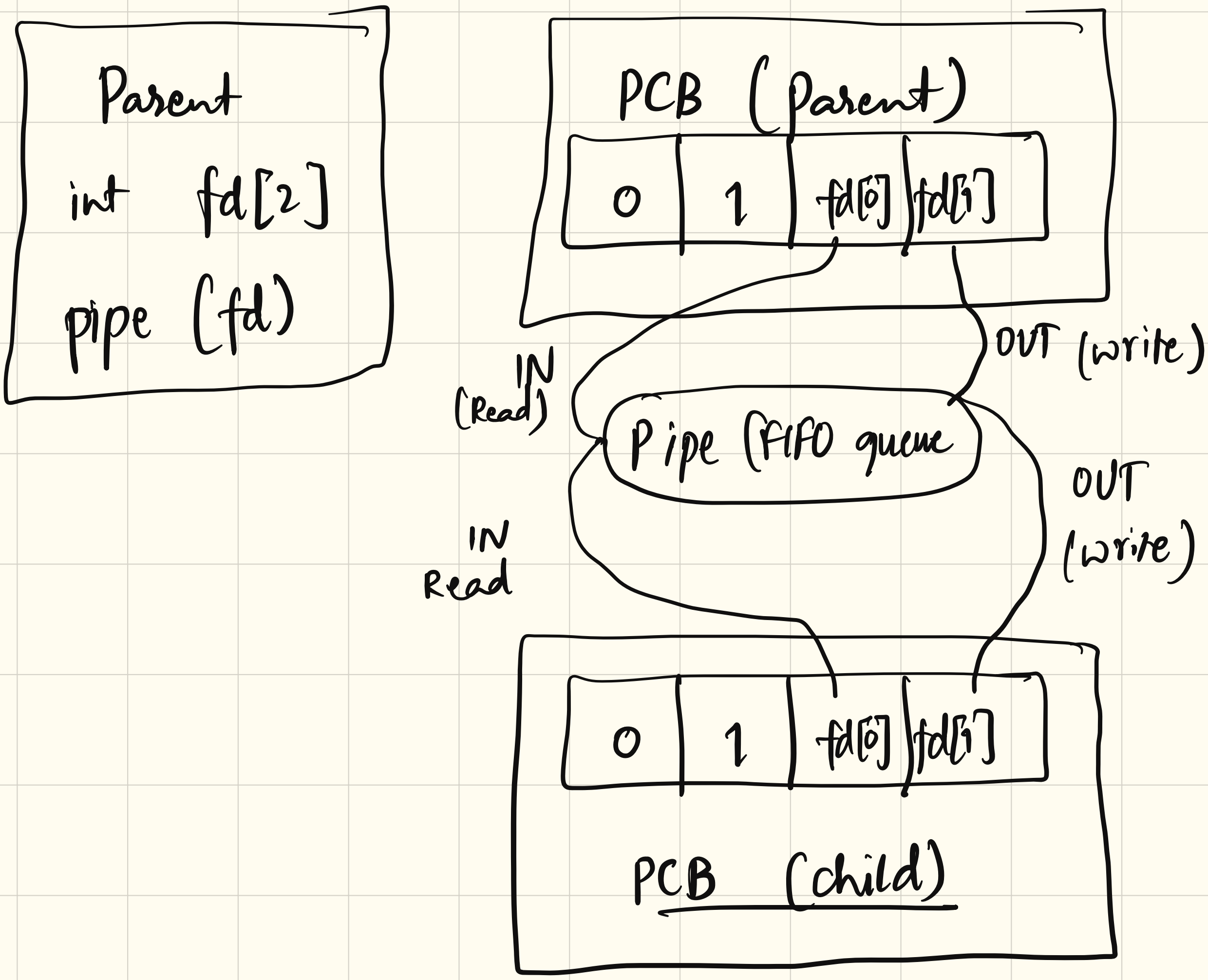
pipe() takes array of two FDs as input

→ fd[0] ⇝ read end of the pipe

→ fd[1] ⇝ write end of the pipe

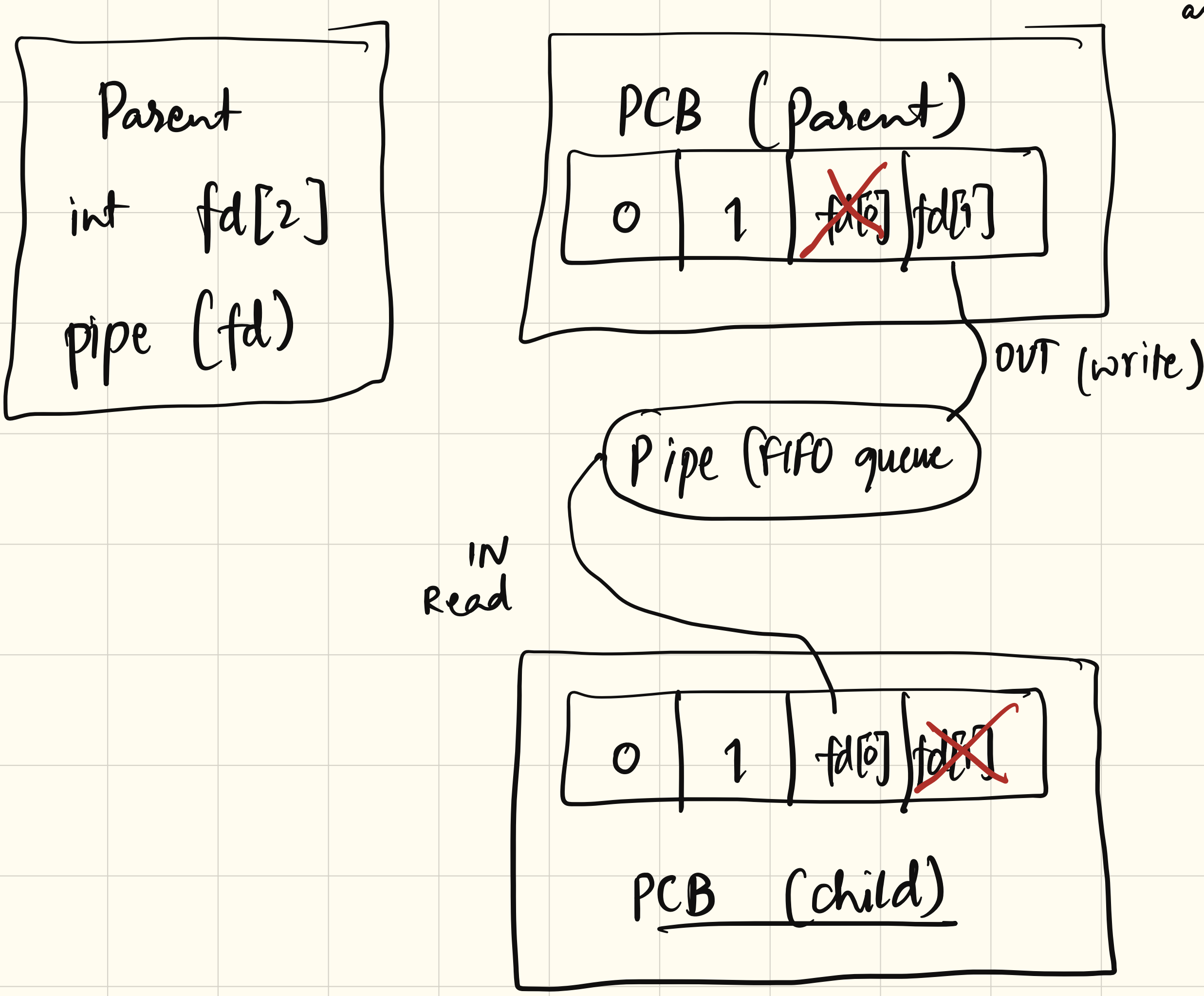
→ Implemented as a FIFO queue in OS.

→ fork duplicates the file descriptors



→ At this point, both parent and child can read/write to the pipe.

* close() one end of the pipe, both in parent and child



Result:

→ a queue b/w parent and child.

Shell piping: ls | wc -l

→ pipe() followed by fork()

→ Parent: exec("ls") after making STDOUT → out fd of the pipe (using dup)

→ Child : exec("wc") after closing STDIN and duping in fd of pipe.

Result: input of wc is connected to output of ls.

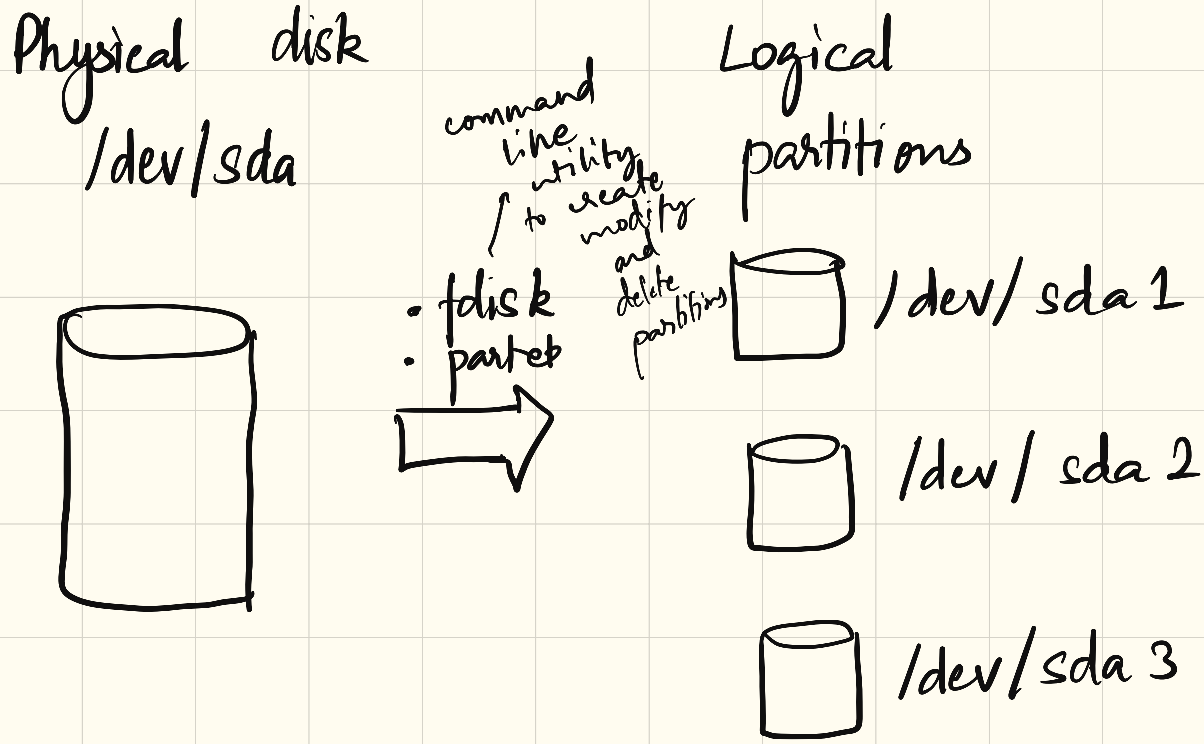
Filesystem Implementation

Step 1: Disk Device Partitioning

→ Partition is stored in the boot sector of the disk.

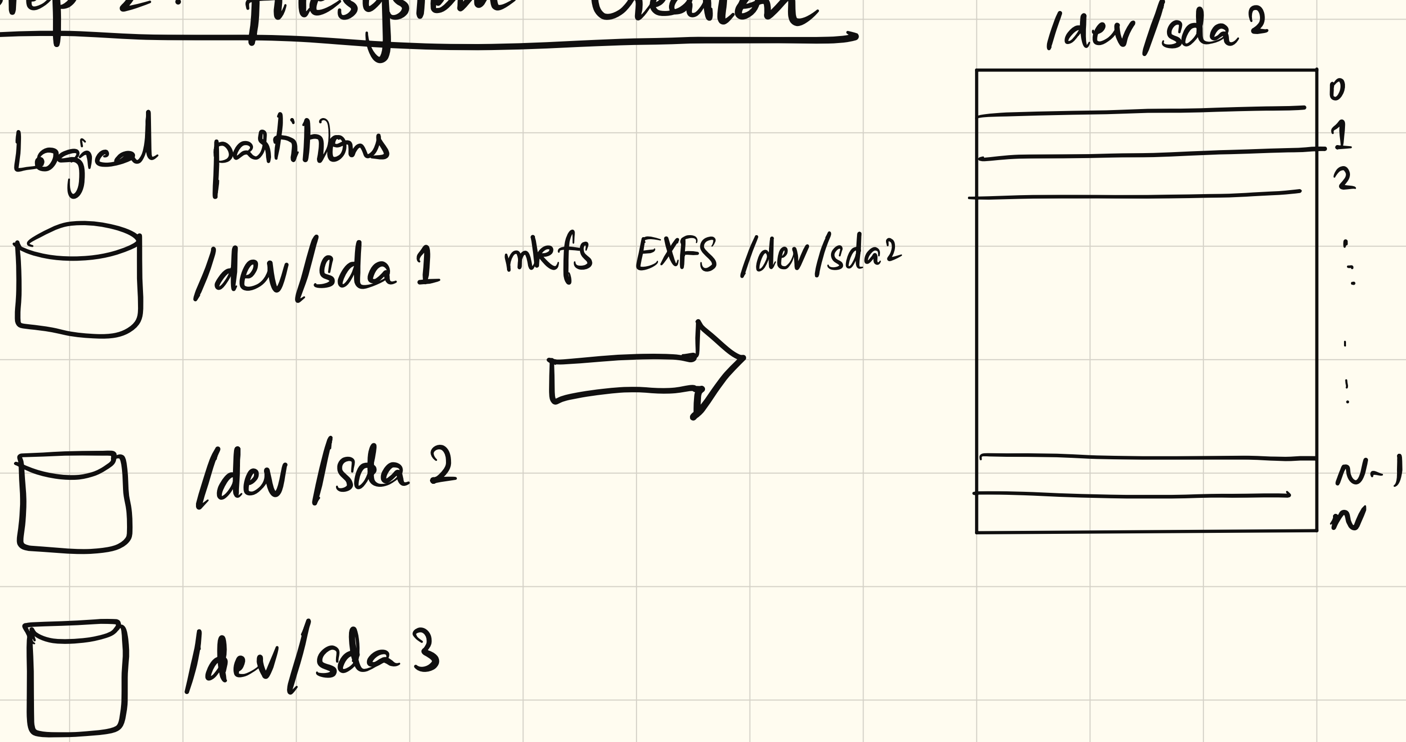
→ Creation of partition is the first step → does not create filesystem

→ A filesystem → created on a partition to manage the physical device and present the logical view.



→ All filesystems provide utilities to initialize the filesystem on the partition (e.g. MKFS)

Step 2: Filesystem Creation



→ MKFS creates initial structures in the logical partition

→ creates entry point to the FS

superblock

→ the filesystem is ready to be mounted.

Step 3: Filesystem mounting

→ mount ( ) associates a superblock with the filesystem mount point

→ example: the OS will use the superblock associated with the mount point "/home" to reach any file/dir under "/home"

OS associates a superblock with FS. mount point

USER: mount -t exfs /dev/sda2 /home

↓

mount ("/dev/sda2" , "/home", "EXFS" , flags, fs_options)

device path

mount point

filesystem type

↓

OS

Load filesystem

EXFS information (Superblock, mount point)

Structure of an example superblock

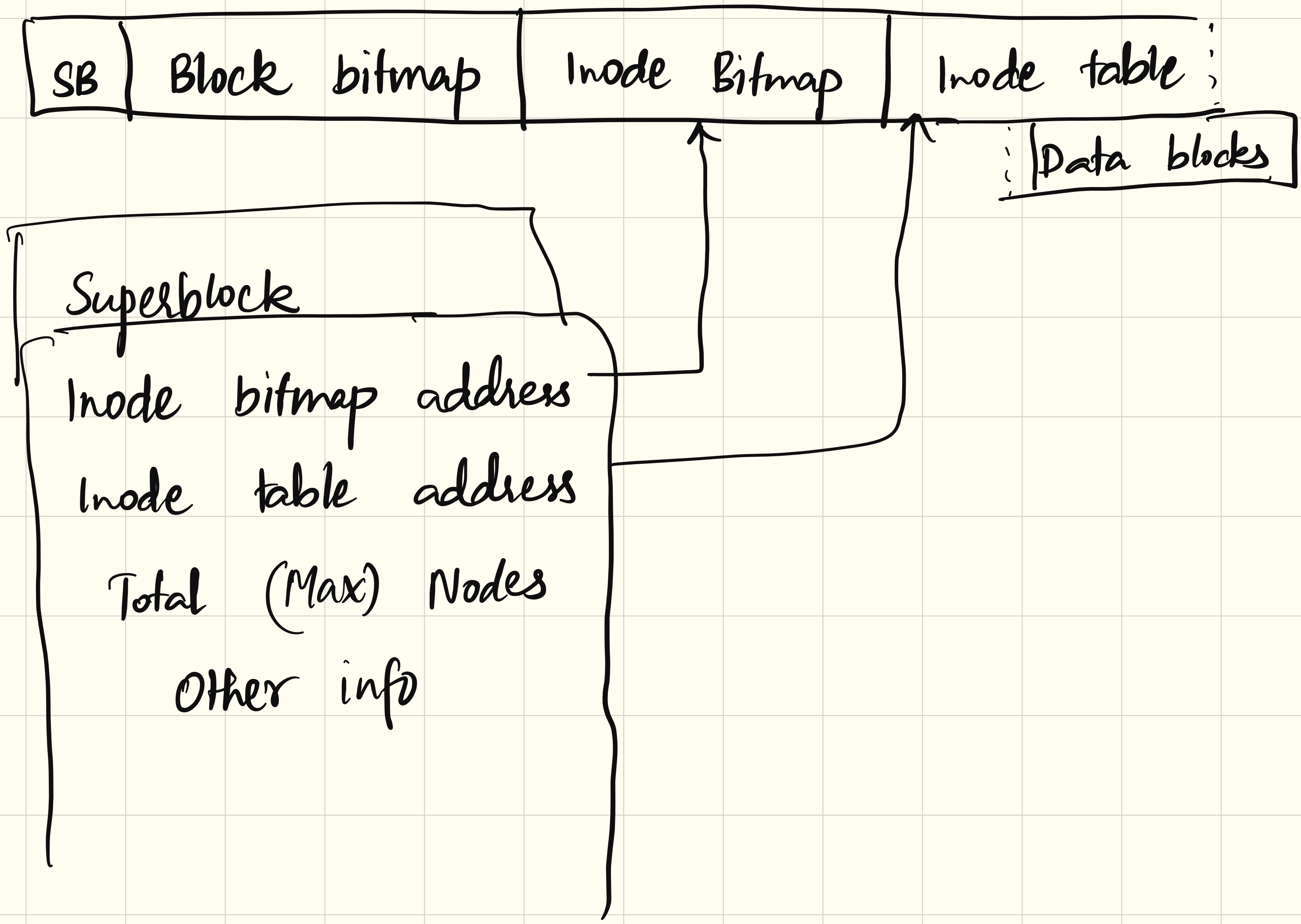
```
struct superblock {
    u16    block_size;
    u64    num_blocks;
    u64    last_mount_time;
    u64    root_inode_num;
    u64    max_inodes;
    disk_off_t    inode_table;
    disk_off_t    blk_usage_bitmap;
    ...
};
```

→ Superblock contains information regarding the device and the filesystem organisation in the disk.

→ Pointer to different metadata related related to the filesystem

→ ex: list of free blocks is required before adding data to a new file/directory

Filesystem organization



```
inode_t *get_inode (SB *sb, long info)
{
    inode_t *inode = alloc_mem_inode();
    read_disk ( inode, sb -> inode_table
                   + info * size_of
                         (inode));

    return inode;
}
```

→ given any inode number, loads the inode structure into memory

<u>Questions:</u>

* Filesystem is mounted, the inode no. for root of the filesystem (i.e., the mount point → known, root node can be accessed

- How to lookup / search files/dir under root inode?
  - How to locate the content in disk?
  - How to keep track of size, permissions, etc?

<u>Inode</u> (UNIX)

→ On-disk structure

→ contains information regarding files/directories in UNIX systems

  → unique number in the fs
  "ls -i filename"

  → contains access permissions, access time, file size, etc.

  → IMPORTANTLY → information regarding file data location on the device

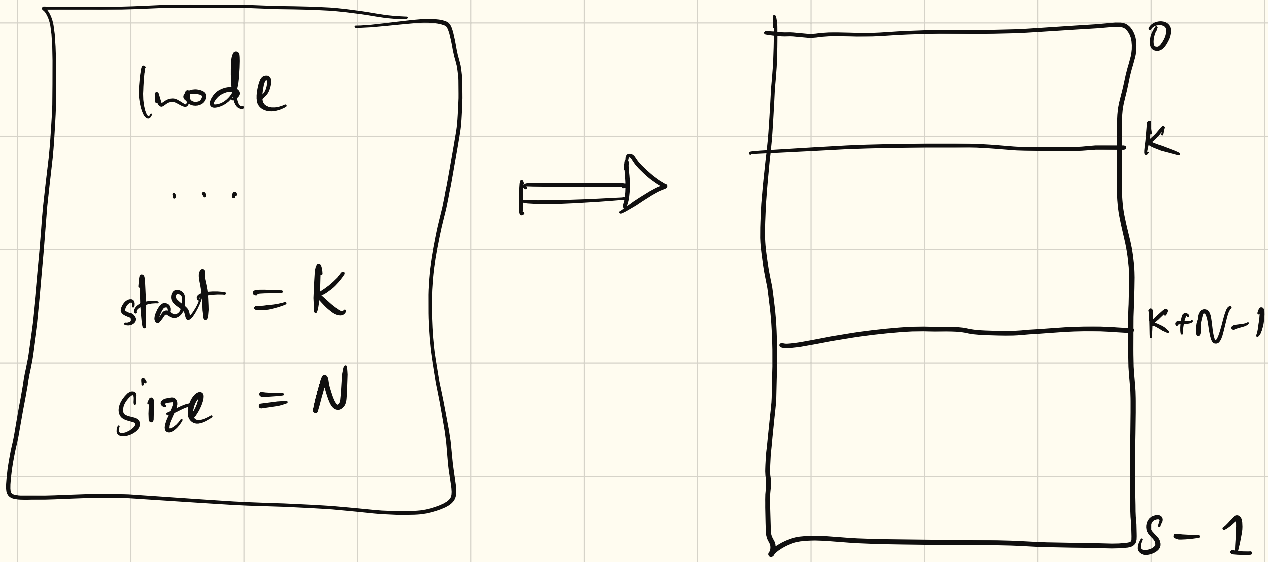
→ Directory inodes → also contain information regarding its content (structure)

Problem: File Offset to Disk Address Mapping

How to efficiently translate file offset to device address?

- → File size: few bytes to GBs
- → can be accessed in sequential or random manner
- → How to design mapping structure.

Contiguous allocation

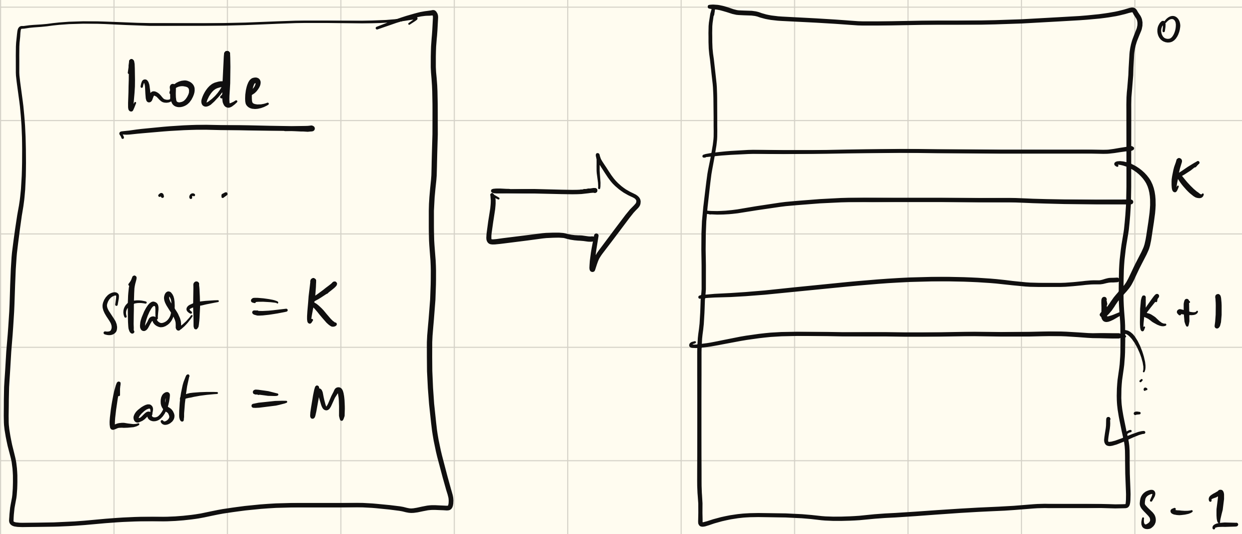


→ Works nicely for both sequential and random access

→ Append operation is difficult.
How to expand files?
↓
Relocation

→ External fragmentation

Linked Allocation



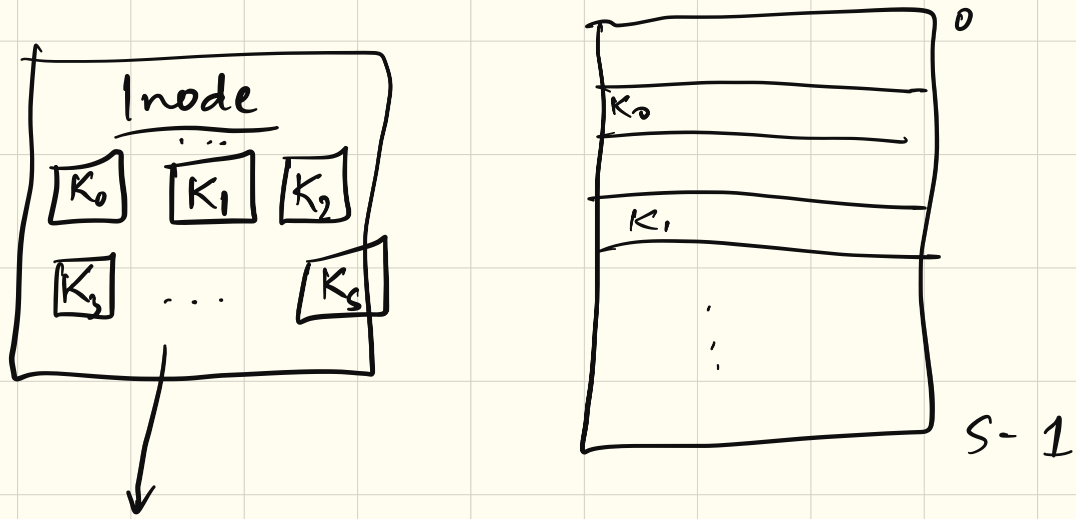
→ Every block contains pointer to next block.

→ Advantage: flexible, easy to grow and shrink

→ Disadvantage: random access

→ Why last? → efficient append operation

Direct Block Pointers

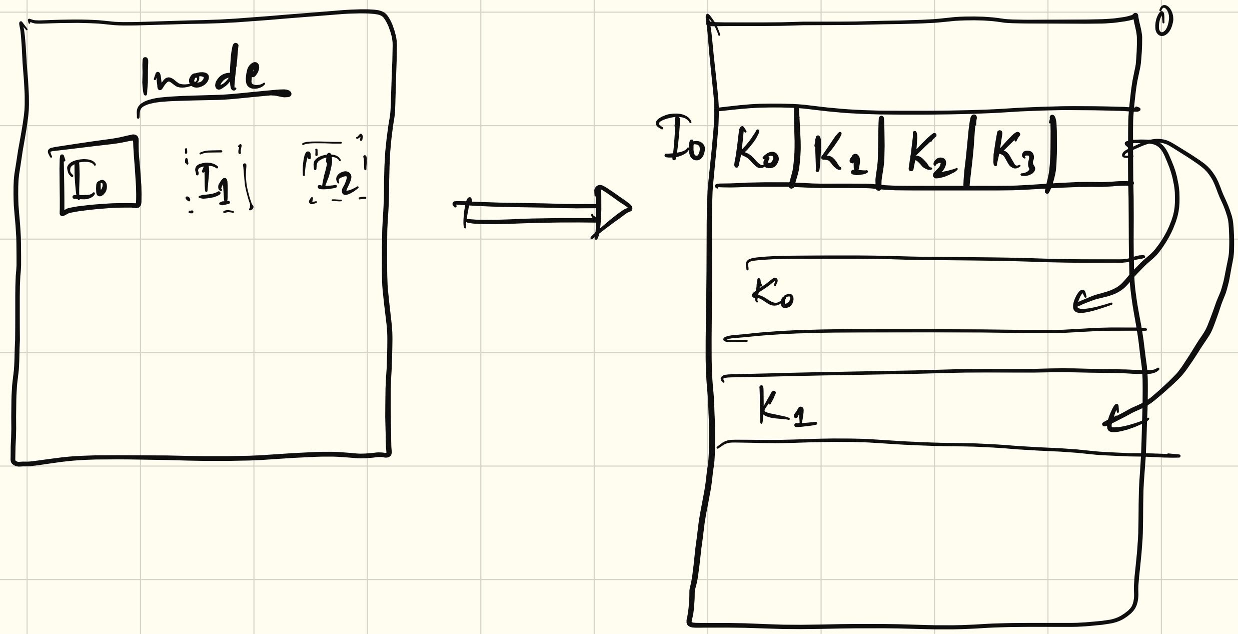


→ direct pointers to block

→ flexible growth, shrink, random access is good

→ Cannot support files of large size

Indirect Block Pointers



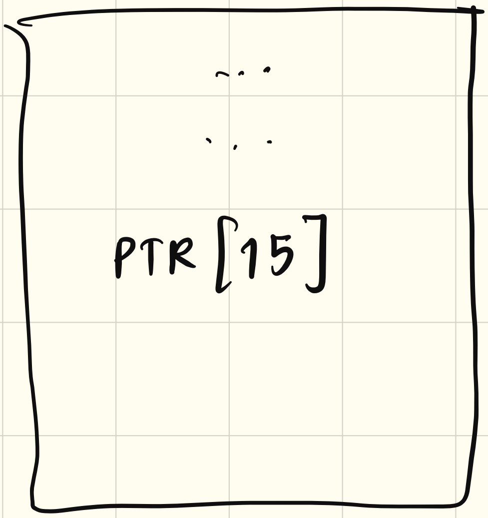
→ Inode contains pointers to a block containing pointers to a data block

→ Advantage : flexible , random access is good

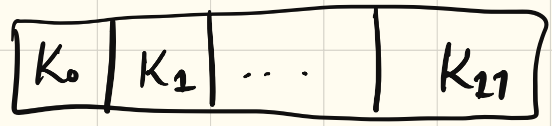
→ Disadvantage : Indirect block access overheads (even for small files)

Hybrid Block Pointers: Ext 2/3 FS

Ext 2/3 inode



Direct pointers {PTR[0] to PTR[11]}



→ File block address (0 to 11)

Single indirect pointers {PTR [12]}

File block address (12 to 1035)

Double indirect {PTR [13]} → 1036 to 1049611

Triple Indirect

* How to locate content in disk?
  → Inode structures in inode are used to map file offset to disk location

* How to keep track of size permissions etc?
  → Inode maintains these info

Organizing directory content

```
struct dir_entry {
    inode_t inode_num;
    char    name[FNAME_MAX];
};
```

→ fixed size directory entry → simple way to organize

→ Advantage : avoid fragmentation,
rename

→ Disadvantage: space wastage;

Variable size:

```
struct dir_entry {
    inode_t inode_num;
    u8 entry_len;
    char name[nam_len];
}
```

→ contain length explicity
→ Advantage: less space wastage (compact)
→ Disadvantage: inefficient rename,
require compaction

* How to search/lookup files/dir under root inode?

→ Read the ontent of the root inode and search the next level dir/file

Caching and Consistency

Filesystem maintains several metadata structures like superblocks, inodes, directory entries to provide a filesystem abstraction like files, directories

How to lookup/search files/directory under root inode?

→ Read the content of the root inode and search the next level dir using the name and find out its inode numbes

→ Read the inode to check permissions and repeat the process.

Filesystem and Caching

→ Accessing data and metadata from disk impacts performance

→ Many file operations require multiple block access:

e.g: opening a file

normal shell operations: ls

- → Executables, config files, library, etc. are accessed frequently.
- → Many directories containing executables config files are also accessed frequently.
- → Metadata blocks storing inodes, indirect block pointers are also accessed frequently

↓

Can we store frequently accessed disk data in memory?

- → What is the storage and lookup mechanism?
- → Are the data and metadata caching same?

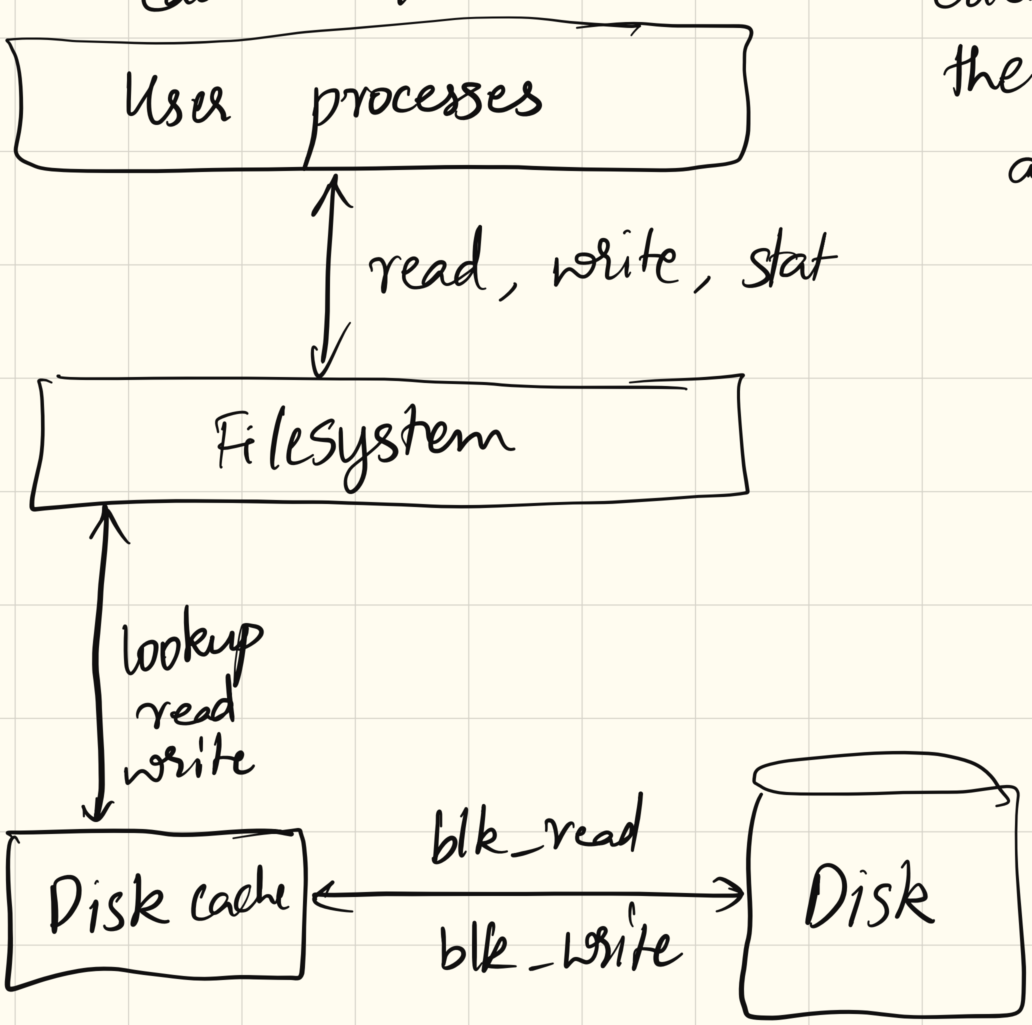
→ How is the cache managed?
Eviction policy?

→ Complication?

Block Layer Caching

→ lookup memory cache using the block numbers as the key.

Cached I/O



① For data caching: file offset to block address mapping is required before using the cache

② Works fine for metadata as they are addressed using block numbers

File Layer Caching (Linux Page Cache)

→ Store and lookup memory cache using {inode #, file offset} as the key.

→ For data, index translation is not required for file access.

→ Metadata may not have a file association → should be handled (using a special inode maybe)

For storage and lookup mechanism

↓

File layer caching is desirable as it avoids index accesses on hit, special mechanism required for metadata

∴ How is the cache managed?

→ lookup memory using block #
or {inode, file offset} as
the key.

Any eviction policy
can be used

If page is dirty, write to
the disk first.

Caching may result in inconsistency:

Ex 1: If a write() → successful
data must be
written

Ex 2: If a file creation → successful
→ data must
be created

block ← inode } bitmap must be set

directory entry contains an inode → inode must be valid

Root causes of inconsistency

update contents of disk blocks

Disk block caching (delayed write

+

system crash

storage medium failure

⟹ possible inconsistent system

→ No issues if only read operations

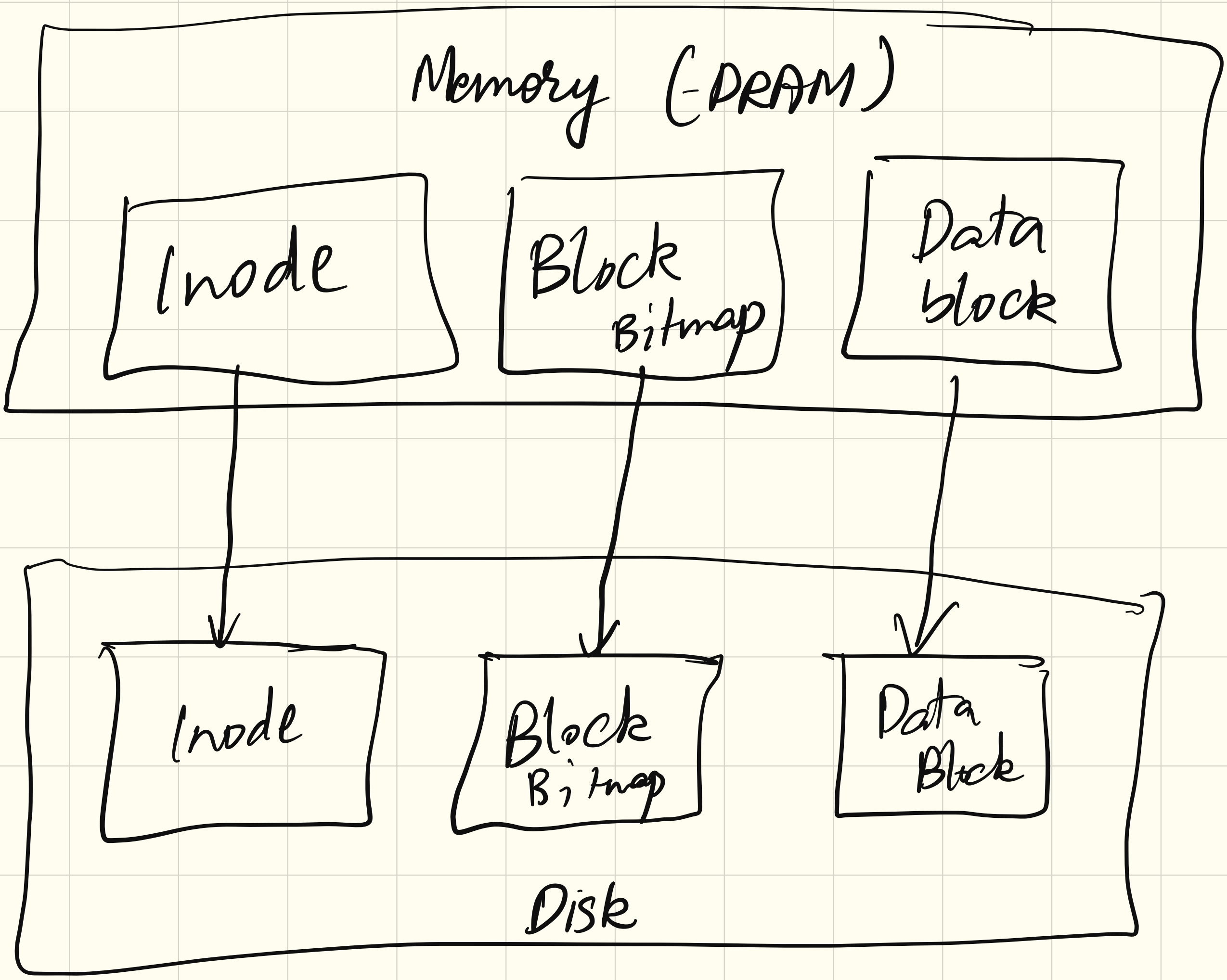
Example: Append a file

(I) Seek to the end of file
(II) allocate new block
(III) write user data

→ Inode modification: size & block pointers

→ Block bitmap update: set used block bit for newly allocated block(s)

→ Data update: Data block content is updated



Three write operations required to complete operation. What if some of them are incomplete?

Written	Yet to be written	Implications
Data block	Inode, Block bitmap	Filesystem is consistent (lost data)
Inode	Block bitmap, data block	Inconsistent (correctness ×)
Block bitmap	Inode, Data block	Inconsistent (space leakage)
Data block Block bitmap	Inode	Inconsistent (space leakage)
Inode Data block	Block bitmap	Inconsistent (correctness)
Inode Block bitmap	Data block	Consistent (Incorrect data)

→ careful ordering of operations may reduce risk of inconsistency

Filesystem consistency with fsck

→ Strategy: Do not worry about consistency, recover after abrupt failure

→ During FS mount, check if it had been cleanly unmounted when it was last used. How to know?
  → Maintain last unmount info on superblock.

→ If the FS was not clearly unmounted, perform sanity checks at different levels: superblock, block bitmap, inode, directory content

→ Sanity checks and verify invariants across metadata.

→ Example:

Block bitmap vs. inode block ptrs

used inodes vs. dir content

Filesystem Consistency with Journalling

* Idea: Before the actual operation, note down the operation in some special blocks
journal

- Journal entry for append operation: [start] [Indoe block] [Block bitmap] [Data block] [End]
- When the FS is updated (in a delayed manner) and marked the journal entry a completed (a.k.a checkpoint)
- Recovery mechanism: Journal entries are inspected during the next mount and operations of non-checkpointed entries are re-performed

- Journal write should not only be synchronous but also performed in the specified order (especially the end marker)
- Failure after updating some blocks and rewritten during recovery is not an issue as the data is consistent at the end
- Failure during journal write is not a problem w.r.t file system consistency

Metadata Journaling: Performance-reliability Tradeoff

- Journaling comes with a penalty, specially for maintaining data in journal
- Metadata Journaling: Data block is not part of the journal entry
  - Practical with tolerable performance overheads
- Example journal entry for append: [start] [Indoe block] [Block bitmap] [End]
- Strategy: First write the data block (to disk) followed by the journal write and metadata commit afterwards, Why?
  - If the metadata blocks are not written, FS can be recovered
  - If journal write fails, a write is lost (syscall semantic broken)