Hi !!

PERSISTENCE: FILE API

Shivaram Venkataraman

CS 537, Spring 2023

ADMINISTRIVIA

Project 5

Project 6, extra slip days

Midterm 2: Today!

5:45pm

Piazza

AGENDA / LEARNING OUTCOMES

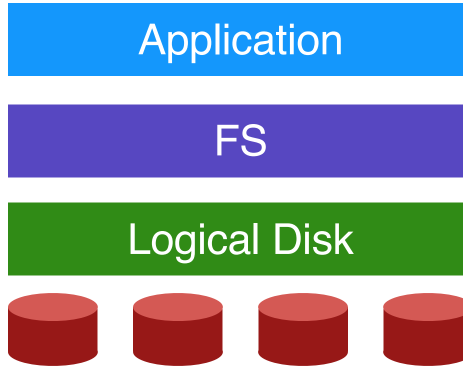
How to name and organize data on a disk?

What is the API programs use to communicate with OS?

RECAP

RAID

Build logical disk from many physical disks.



Logical disk gives

capacity,

performance,

reliability

RAID: **R**edundant **A**rray of **I**nexpensive **D**isks

METRICS

Capacity: how much space can apps use?

Reliability: how many disks can we safely lose? (assume fail stop)

Performance: how long does each workload take? (latency, throughput)

Normalize each to characteristics of one disk

Different **RAID levels** make different trade-offs

RAID LEVEL COMPARISONS

	Reliability	Capacity	Read latency	Write Latency	Seq Read	Seq Write	Rand Read	Rand Write
RAID-0	0	C*N	D	D	N * S	N * S	N * R	N * R
RAID-1	1	C*N/2	D	D	N/2 * S	N/2 * S	N * R	N/2 * R
RAID-4	1	(N-1) * C	D	2D	(N-1)*S	(N-1)*S	(N-1)*R	R/2
RAID-5	1	(N-1) * C	D	2D	(N-1)*S	(N-1)*S	N * R	N/4 * R

mirror the data

half is usable

big

Stripe the data

rotate the parity across disks

Parity disk → Code XOR store the code on parity disk

DISKS → FILES

WHAT IS A FILE?

API for users to store / retrieve data

Array of persistent bytes that can be read/written

↳ provided by OS

File system consists of many files

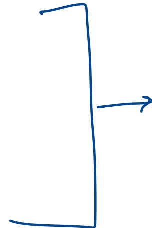
Refers to collection of files

Also refers to part of OS that manages those files

Files need names to access correct one

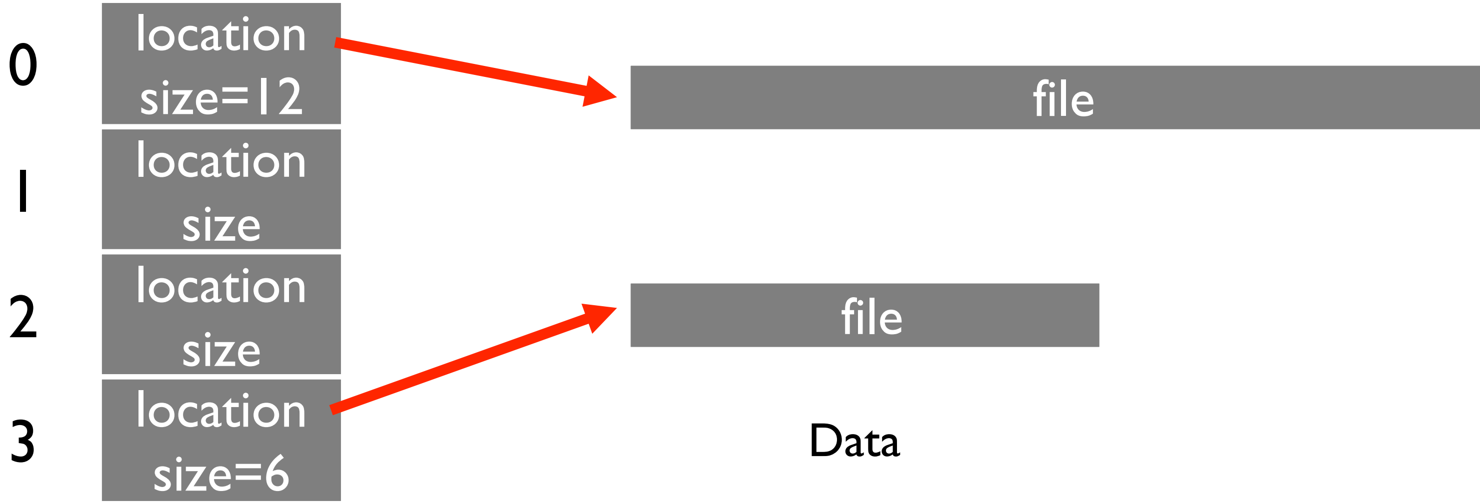
Three types of names

- Unique id: inode numbers
- Path
- File descriptor



→ OS knows about

inodes



...

Meta-data

FILE API (ATTEMPT 1)

```
read(int inode, void *buf, size_t nbyte)
write(int inode, void *buf, size_t nbyte)
seek(int inode, off_t offset)
```

"cs537-raid-5.pdf" vs 10234

users would prefer names rather than numbers

Disadvantages?

- names hard to remember
- no organization or meaning to inode numbers
- semantics of offset across multiple processes?

related files in a directory

PATHS

String names are friendlier than number names → "hello.c" → string name for this file

File system still interacts with inode numbers

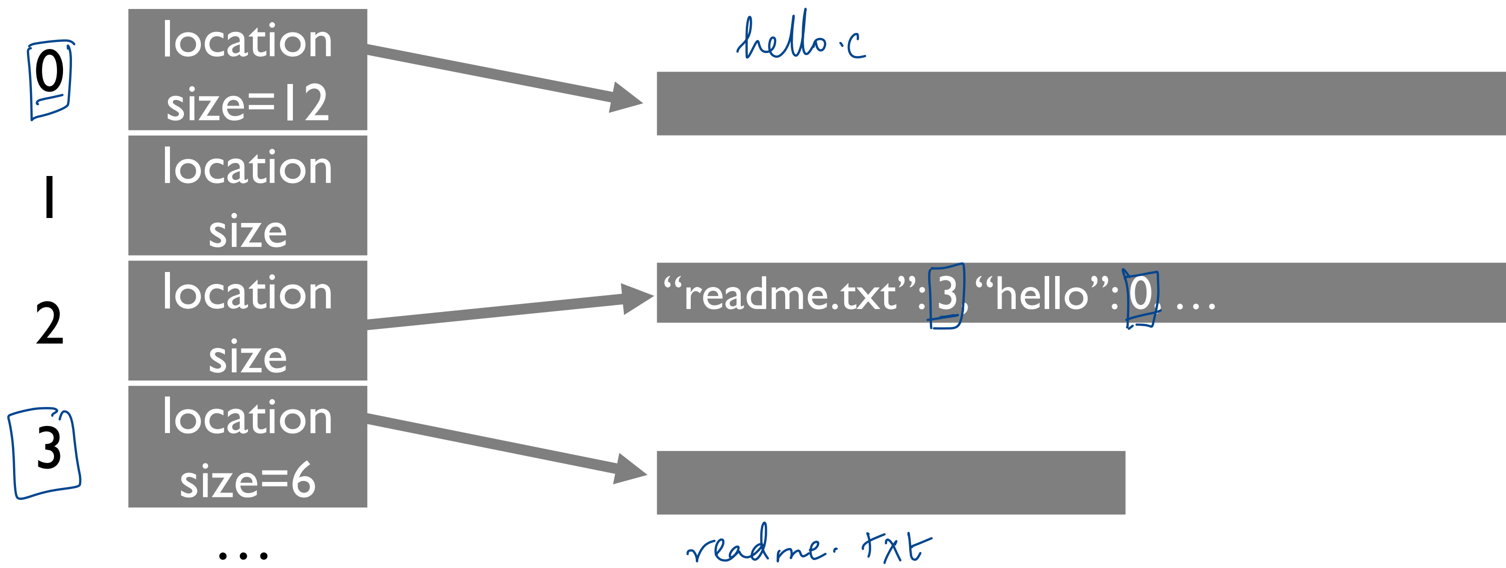
Store *path-to-inode* mappings in a special file or rather a Directory!

→ ls -i ~/

should also show inode numbers!

Directory → special file

hello.c	3
Readme.txt	12
⋮	


inodes
inode number
0
1
2
3
location
size=12
location
size
location
size
location
size=6
...
hello.c
"readme.txt": 3, "hello": 0, ...
readme.txt

PATHS

Directory Tree instead of single root directory

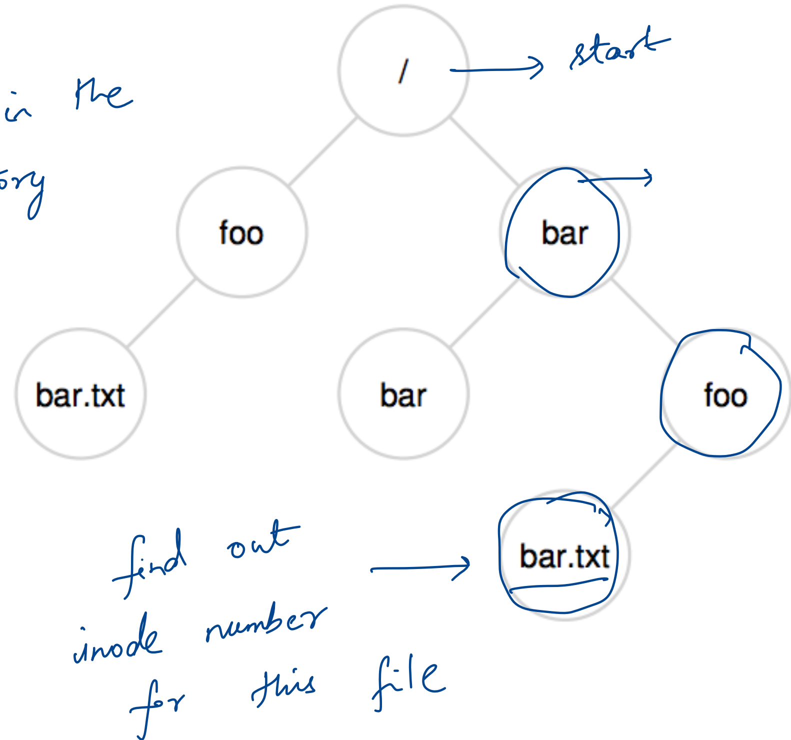
File name needs to be unique within a directory

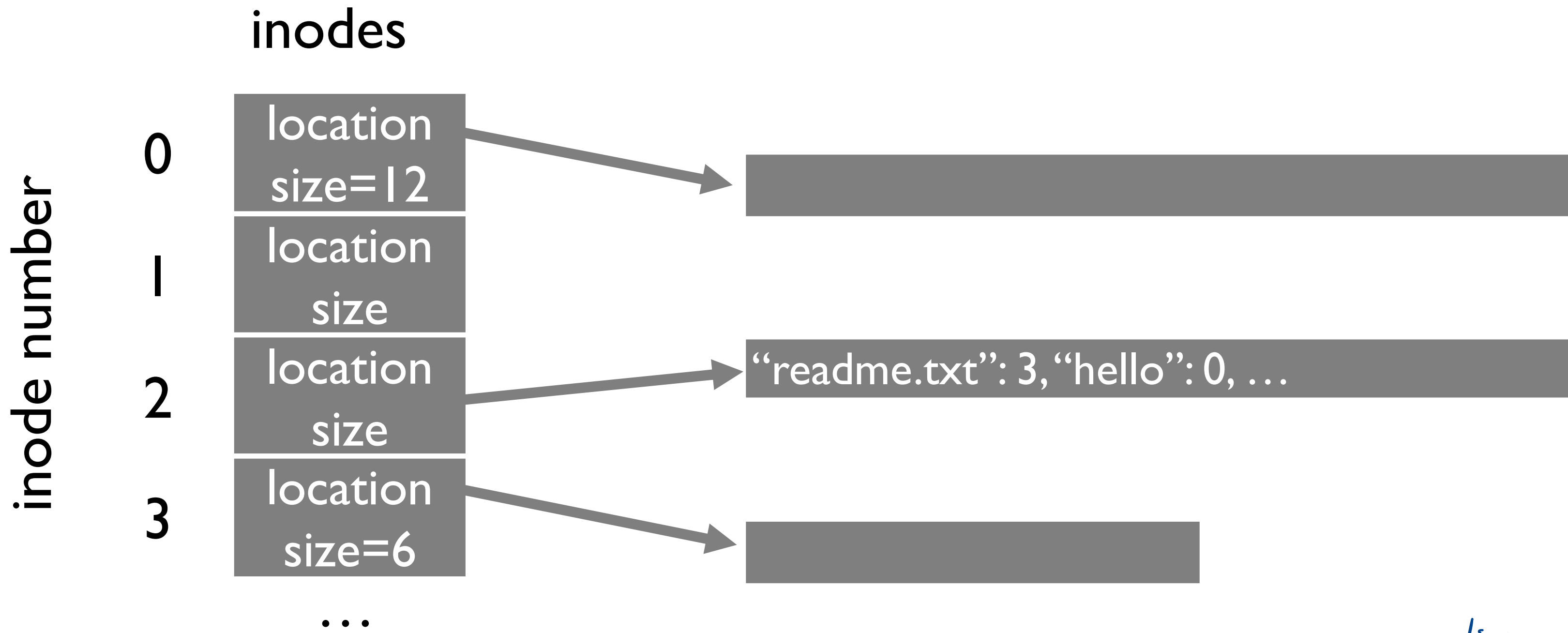
/usr/lib/file.so →

/tmp/file.so →

fine because not in the same directory

Store file-to-inode mapping in each directory





Reads for getting final inode called "traversal" → start from root dir /

Example: read /hello

read inode mapping in /

find inode number for /hello

FILE API (ATTEMPT 2)

```
read(char *path, void *buf, off_t offset, size_t nbyte)
write(char *path, void *buf, off_t offset, size_t nbyte)
```

Disadvantages?

Expensive traversal! →
Goal: traverse once

read ("/hello")

write ("/hello")

read ("/hello")

each of these calls need inode number

↳ traversal each time!

→ many levels : /usr/lib/python/

→ each level could incur disk I/O

FILE DESCRIPTOR (FD)

Idea:

- Do expensive traversal once (open file)
- Store inode in descriptor object (kept in memory).
- Do reads/writes via descriptor, which tracks offset

returns FD

struct fd {
inode →
offset →
}

Each process:

- File-descriptor table contains pointers to open file descriptors

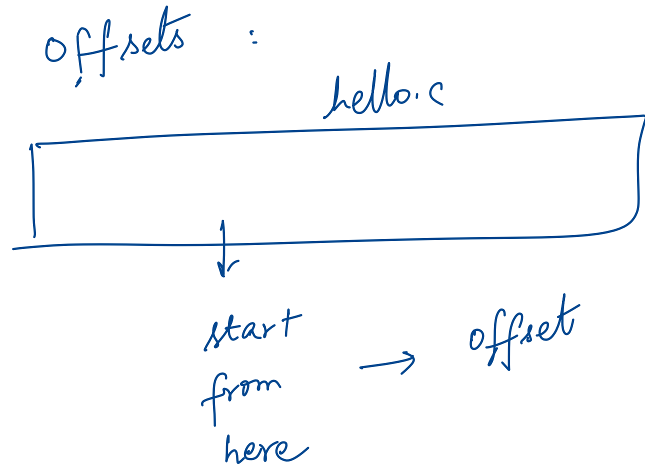
Process {
fd [100] fds;
}

Integers used for file I/O are indexes into this table

- stdin: 0, stdout: 1, stderr: 2

FILE API (ATTEMPT 3)

```
int fd = open(char *path, int flag, mode_t mode)
read(int fd, void *buf, size_t nbyte)
write(int fd, void *buf, size_t nbyte)
close(int fd)
```



advantages:

- string names
- hierarchical
- traverse once
- offsets precisely defined

FD TABLE (XV6)

```
struct file {
  ...
  struct inode *ip;
  uint off;
};
```

File descriptor

where next operation starts from

```
struct {
    struct spinlock lock;
    struct file file[NFILE];
} ftable;
```

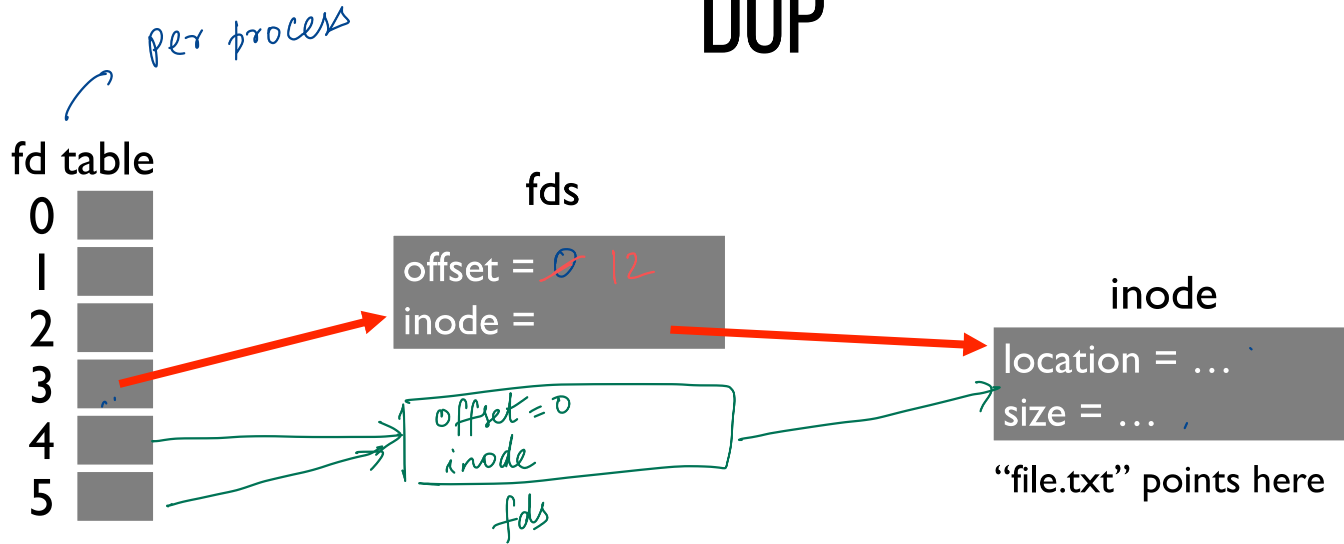
global

```
// Per-process state
struct proc {
  ...
  struct file *ofile[NOFILE];  // Open files
  ...
}
```

pge-dir*

FD number indexes into array

DUP




```
int fd1 = open("file.txt"); // returns 3
read(fd1, buf, 12);
int fd2 = open("file.txt"); // returns 4
int fd3 = dup(fd2);         // returns 5
```

read 12 bytes

read (fd1) → offset 12
read (fd2) → offset 0 updates 12
read (fd3) → read from 12 to 24

READ NOT SEQUENTIALLY

```
off_t lseek(int filedesc, off_t offset, int whence)
    If whence is SEEK_SET, the offset is set to offset bytes.
    If whence is SEEK_CUR, the offset is set to its current
        location plus offset bytes.
    If whence is SEEK_END, the offset is set to the size of
        the file plus offset bytes.
```

beginning of file

end of file

```
struct file {
  ...
  struct inode *ip;
  uint off;
};
```

updating the offset here

QUIZ 24

https://tinyurl.com/cs537-sp23-quiz24

```
int fd1 = open("file.txt"); // returns 12
int fd2 = open("file.txt"); // returns 13
read(fd1, buf, 16);
int fd3 = dup(fd2);         // returns 14
read(fd2, buf, 16);  → sets fd2 offset to 16
lseek(fd1, 100, SEEK_SET); →
```

Offset for fd1

100

Offset for fd2

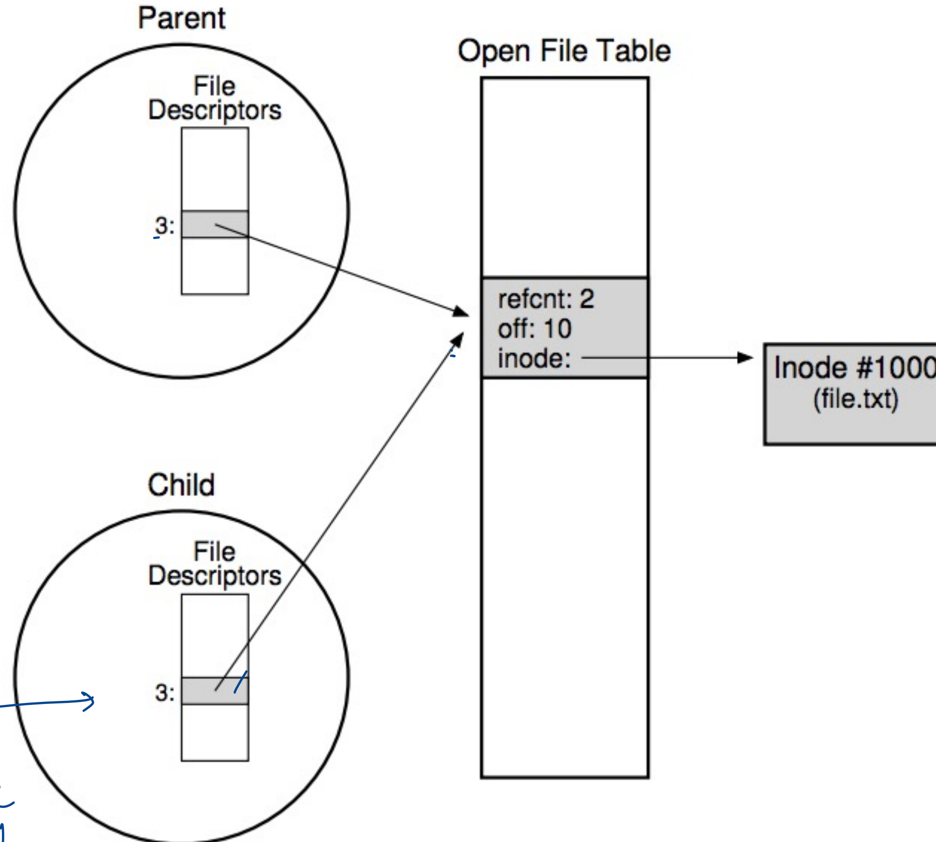
16

Offset for fd3

16

WHAT HAPPENS ON FORK?

set of file descriptors are duplicated when you do fork



seek operation will update offset !

COMMUNICATING REQUIREMENTS: FSYNC

File system keeps newly written data in memory for awhile

Write buffering improves performance (why?) → delays disk I/O

But what if system crashes before buffers are flushed? → data loss → reboot / recover data missing

fsync(int fd) forces buffers to flush to disk, tells disk to flush its write cache

Makes data durable

→ forces data to be flushed

DELETING FILES

There is no system call for deleting files!

Inode (and associated file) is **garbage collected** when there are no references

Paths are deleted when: `unlink()` is called

FDs are deleted when: `close()` or process quits

RENAME

rename(char *old, char *new):

- deletes an old link to a file
- creates a new link to a file

Just changes name of file, does not move data

Even when renaming to new directory

What can go wrong if system crashes at wrong time?

ATOMIC FILE UPDATE

Say application wants to update file.txt atomically

If crash, should see only old contents or only new contents

1. write new data to file.txt.tmp file
2. fsync file.txt.tmp
3. rename file.txt.tmp over file.txt, replacing it

SUMMARY

Using multiple types of name provides convenience and efficiency

Special calls (fsync, rename) let developers communicate requirements to file system

Next class: Directory features, Filesystem implementation