CS 423
Operating System Design: FS API & Implementation
Apr 14

Ram Kesavan

LOGISTICS

MP3 due by Apr 15, 11:59 CT

Midterm Regrades will be pushed to Canvas by Wed

- Will distribute the midterm papers Thu

MP4 gets published tonight

- Walkthrough on Thu

Finals: 5/13 at 7-10pm, DCL 1320 (next door)

AGENDA / LEARNING OUTCOMES

File system API

- Continue from last class
- Data structures to track “open” files
  - per-process & system-wide

File system Internals

- Layout of data structures
- How they get updated
- Crash consistency: fsck & journaling

RECAP

File Descriptor (fd)

Solution:

- Expensive traversal done once (to **open** file)
- The open API returns a file descriptor (fd), which is an int
- All subsequent API calls (reads/writes/etc) use fd
  - An opaque handle that hides the inum within it, and current offset

Each process has a file descriptor table:

- maps each fd (handed out) to a pointer into a system-wide file descriptor table

Special file descriptors: 0 (stdin), 1 (stdout), 2 (stderr)

- UNIX hands out file descriptors starting from 3

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:

- Alphanumeric names organized hierarchically
- Expensive traversal of dir tree done once (to **open** file)
- Opaque handle used by subsequent calls
- Handle tracks the offset

FD Offsets

System Calls	Return Code	OFT[10] Current Offset	OFT[11] Current Offset
`fd1 = open("file", O_RDONLY);`	3	0	–
`fd2 = open("file", O_RDONLY);`	4	0	0
`read(fd1, buffer1, 100);`	100	100	0
`read(fd2, buffer2, 100);`	100	100	100
`close(fd1);`	0	–	100
`close(fd2);`	0	–	–

END RECAP

LSEEK

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

If whence is **SEEK_SET**: cursor = offset
If whence is **SEEK_CUR**: cursor += offset
If whence is **SEEK_END**: cursor = EOF + offset
offset can be -ve in last 2 cases

Assume HDD head is on track 1
Suppose **SEEK_SET** to offset X, and Xth byte is on track 4
Where is the head immediately after lseek?

Shared Entries in OFT

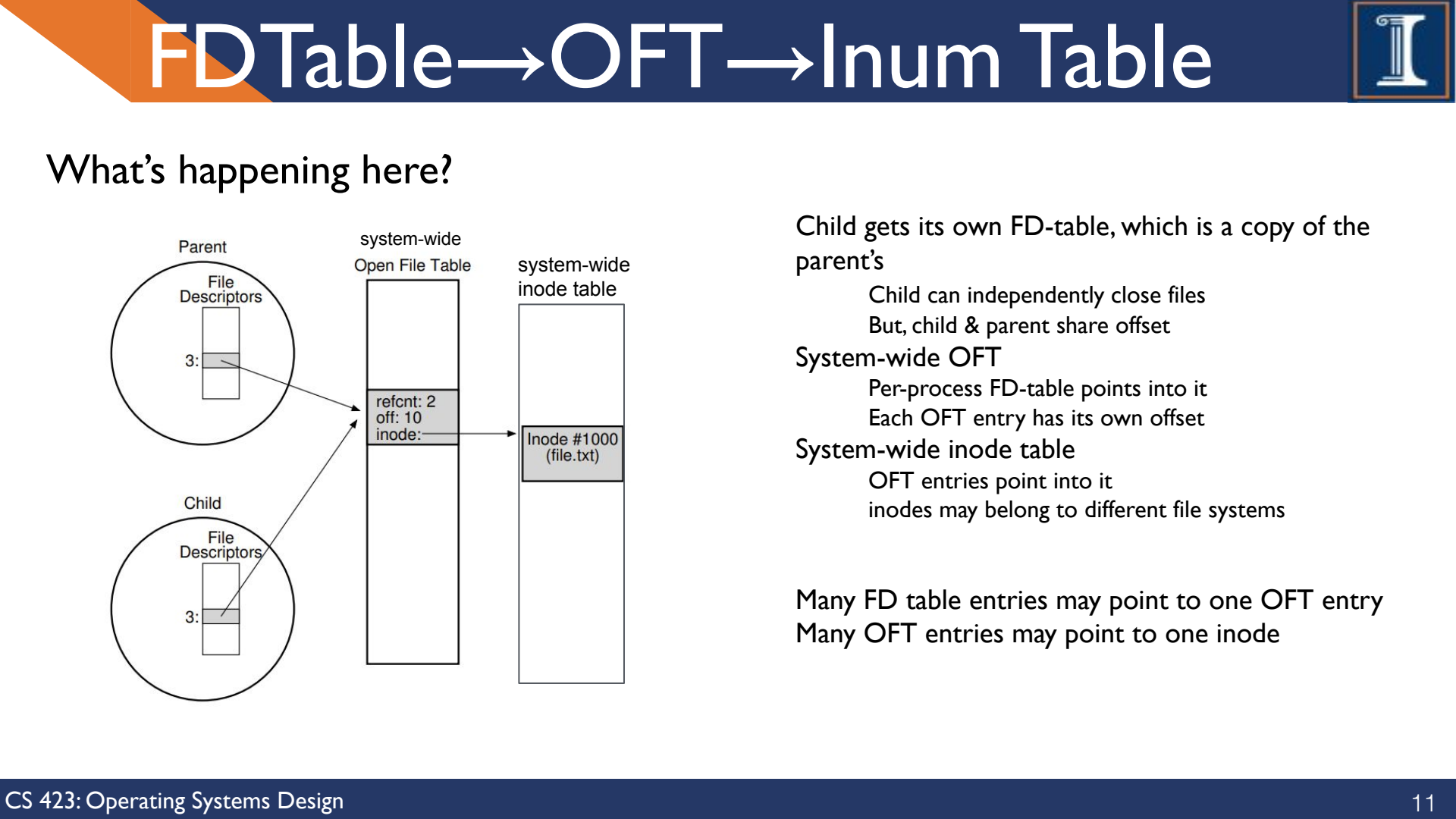
Fork:

```
int main(int argc, char *argv[]) {
    int fd = open("file.txt", O_RDONLY);
    assert(fd >= 0);
    int rc = fork();
    if (rc == 0) {
        rc = lseek(fd, 10, SEEK_SET);
        printf("child: offset %d\n", rc);
    } else if (rc > 0) {
        (void) wait(NULL);
        printf("parent: offset %d\n",
               (int) lseek(fd, 0, SEEK_CUR));
    }
    return 0;
}
```

What offset does the child print?

What offset does the parent print?

FD Table→OFT→Inum Table



What's happening here?

Child gets its own FD-table, which is a copy of the parent's
- Child can independently close files
- But, child & parent share offset

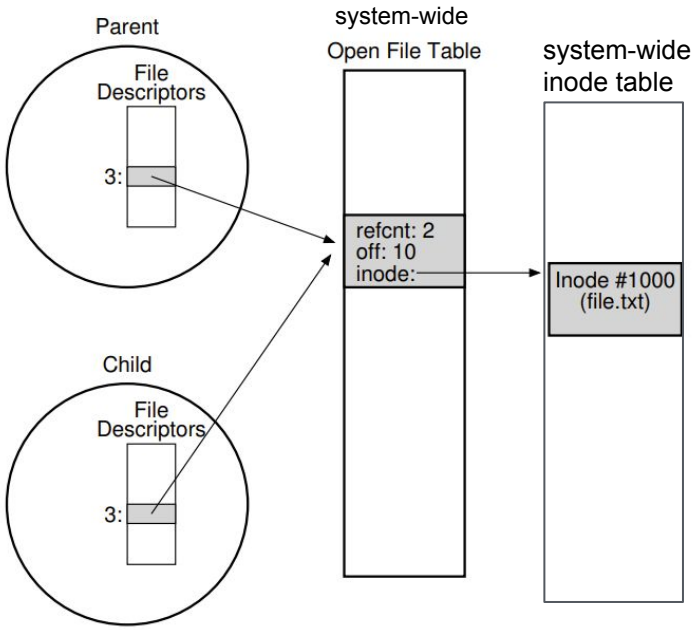
System-wide OFT
- Per-process FD-table points into it
- Each OFT entry has its own offset

System-wide inode table
- OFT entries point into it
- inodes may belong to different file systems

Many FD table entries may point to one OFT entry
Many OFT entries may point to one inode

FD Table→OFT→Inum Table

What's happening here?



In Linux kernel, the per-process file descriptor table is located in a nested location.

In the file `include/linux/sched.h`

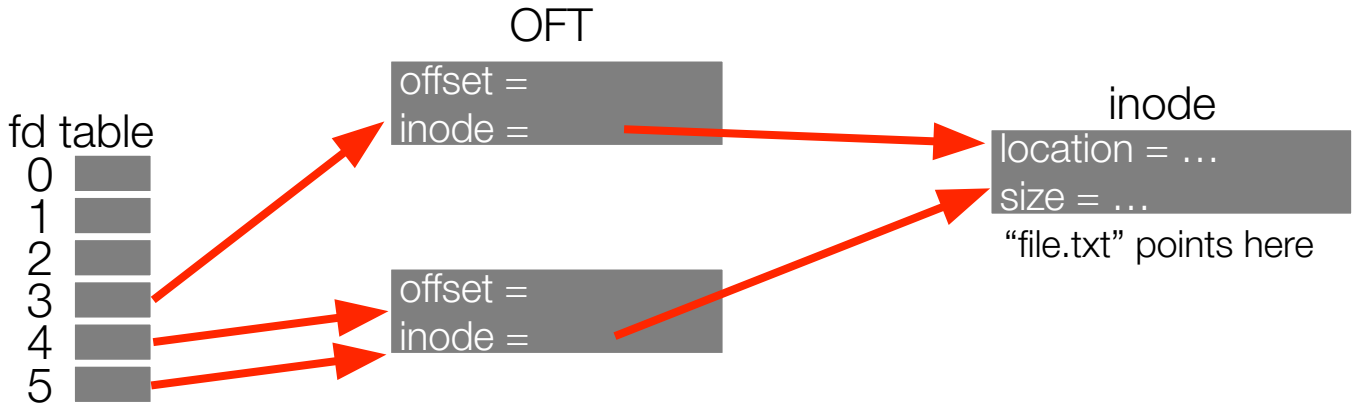
Start from `task_struct`

→ `struct files_struct *files;`

→ `struct fdtable *fdt;`

→ `struct file fd[];`

DUP System Call




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

Paths and Links

(Hard) Link:

```
% ln <filename> <linkname>
```

The two directory entries share the same inode

Sharing tracked by field called “nlinks” in the inode

When is it incremented?

When is it decremented?

```
% ls -i a/foo        // print inum of file
884285 a/foo
% ln a/foo b/foolink
% rm a/foo
% ls -i b/foolink
884285 b/foolink
```

Deleting Files

Use the **`unlink(fd)`** system call to delete a file

- Results in decrement of `nlinks` in inode
- File in directory is deleted; directory entry is removed

After `close(fd)` on the last fd pointing to that inode

```
if nlinks == 0
```

- inode is **garbage collected**

So, process can read/write to it even after unlink!

OSTEP Chapter 39

`Is a must-read`

Explains file descriptors, fork(), dup(), rename()

- directories: making, reading, deleting
- file/dir permissions
- file deletion
- hard and soft links

Rename

rename(char *old, char *new)

Do we need to copy/move data?

How does the FS implement this?

Does it matter whether the old and new names are in the same directory or different directories?

Rename

`rename(char *old, char *new):`

- effectively unlinks the old file

- creates a link for the new file

Only changes name of file, does not move data

Even when renaming to new directory

What can go wrong if system crashes at wrong time?

File System Updates

How do we make **atomic** updates to the file system?

Eg. `mv ./foo/file.txt ./bar/file.txt`

If system crashes, atomicity requires that:

`file.txt` must be only in either `foo` or in `bar`

Not both & not neither

Fsync

File system holds recently written data in memory for a bit

- Write buffering improves performance. Why?

fsync(int fd) forces buffers to flush to disk

- Guarantees that data has been persisted durably

But what if system crashes before buffers are flushed?

- Covered over subsequent lectures

Inside the FS

Previously:

Bottom-up stuff:

storage devices, RAID, performance

Top down stuff:

FS API

how apps access/view the FS: files & directories

Next:

File system internals

A simple ext2-like FS

Then, crash consistency, fsck, journaling, etc.

FILE SYSTEM INTERNALS

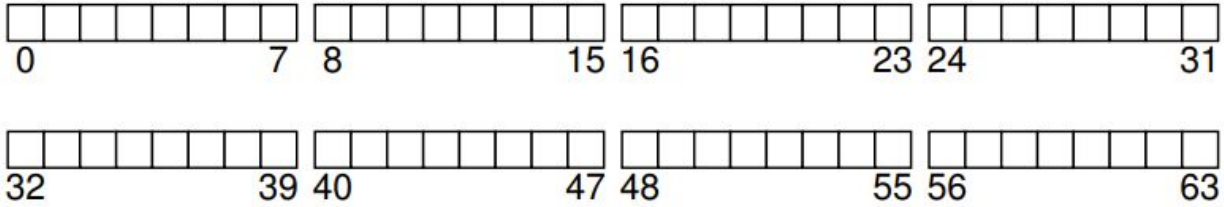
Very Simple File System

Data structures: how files, directories, etc., persisted

Access methods: how high-level operations (open, read, write) utilize the data structures

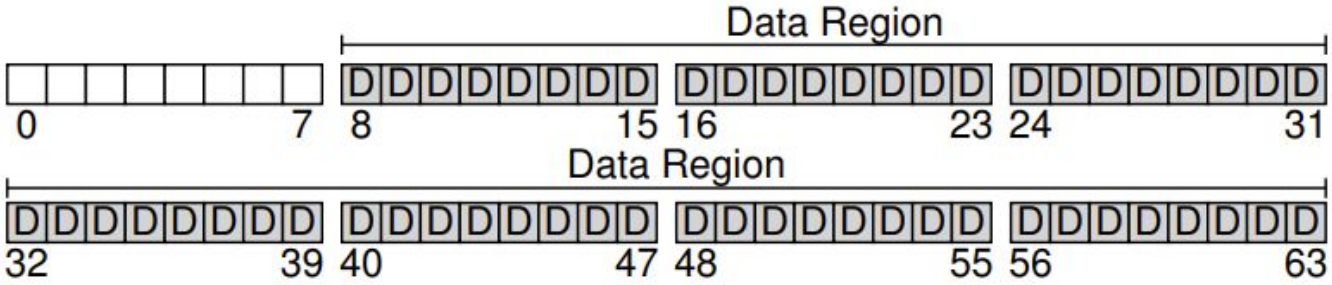
VSFS

Assume a small disk partition with 64 blocks (4KiB sized)



Data and metadata – most space must go for data blocks

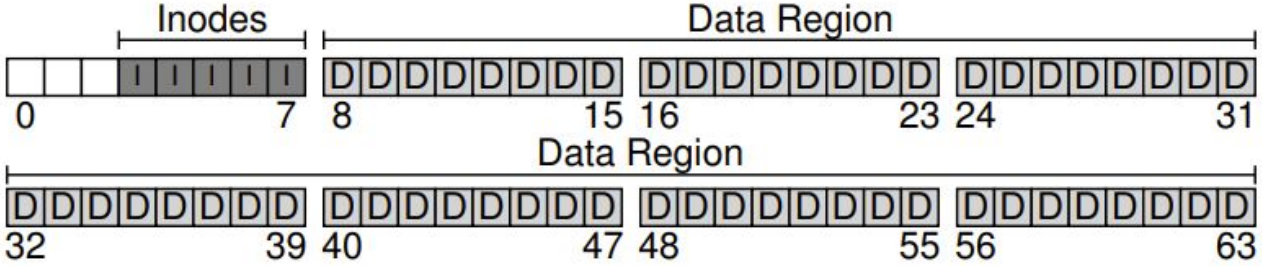
VSFS – Data blocks



Upto 56 blocks worth of data

Rest of the file system is metadata

VSFS – Inodes (metadata)



Inode table

Each inode needs 256-byte→16 inodes per block

Thus, VSFS supports upto 80 files + dirs

Note: format easily scales with disk partition size

VSFS – Bitmaps (metadata)

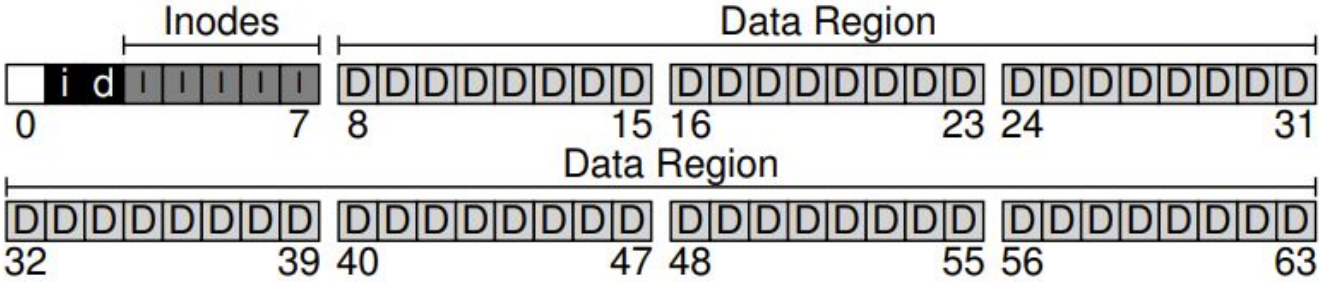
Need structures to track allocation state

Free lists – linked list is an option, but why is that inefficient?

More commonly used: bitmap

Inode-bitmap & Datablock-bitmap: 1 implies allocated, 0 implies free

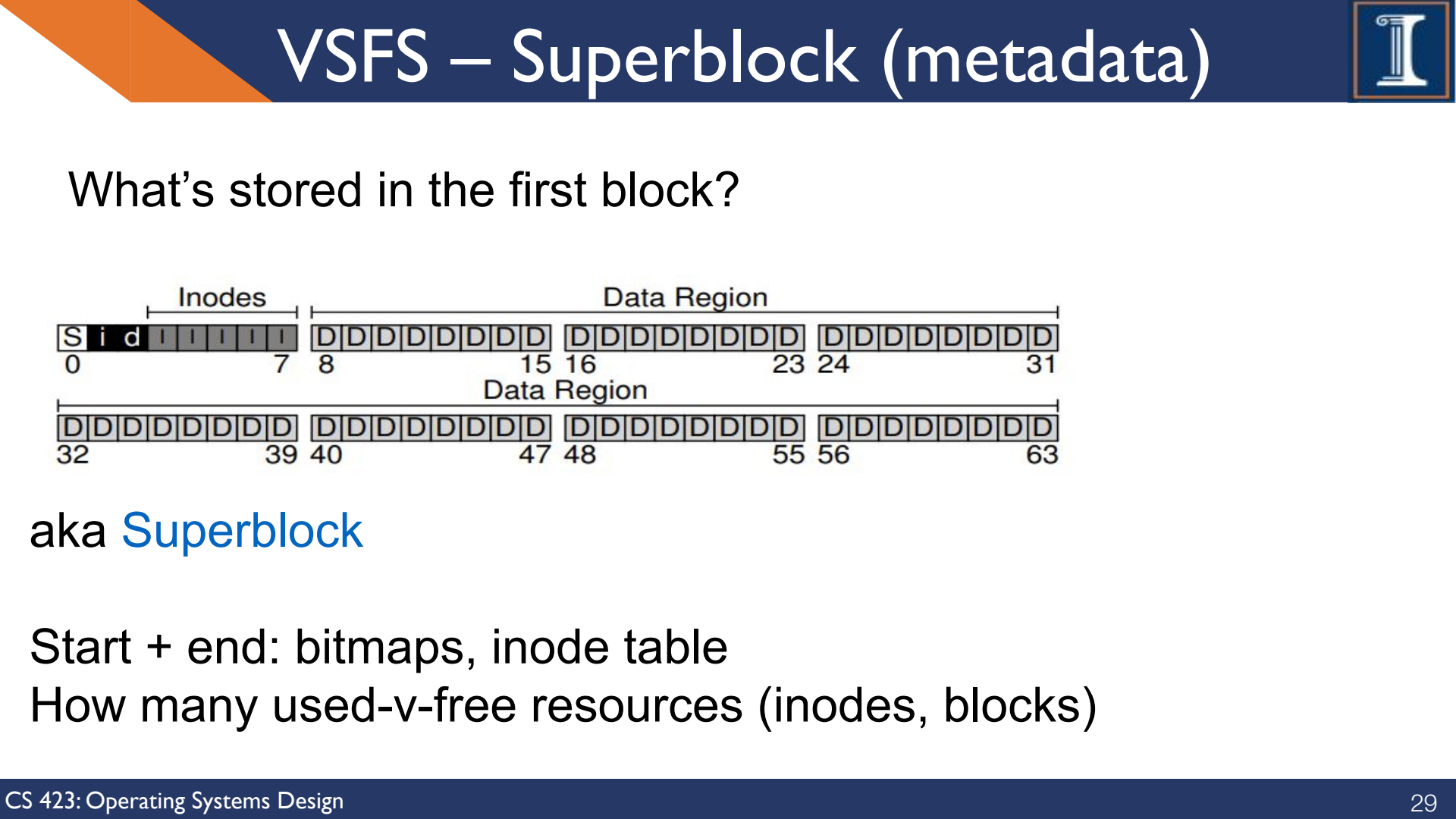
1 block bitmap: so what's the largest file system size supportable?



VSFS – (metadata)

What's stored in the first block?

VSFS – Superblock (metadata)

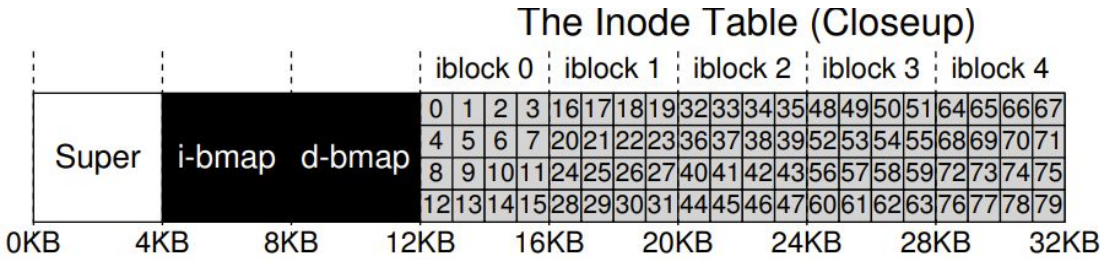


What's stored in the first block?

aka Superblock

Start + end: bitmaps, inode table
How many used-v-free resources (inodes, blocks)

INODE



A single inode (256 bytes)

Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
2	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
4	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists

Inode → Data Blocks

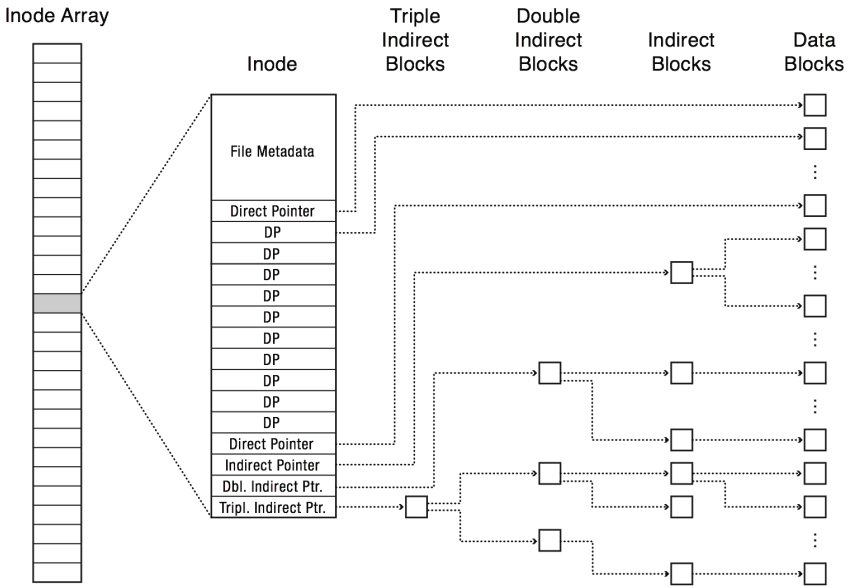
Each block pointer is 4 bytes

Inode has 15 block pointers (60 bytes)

What's the max supportable file size?

So, how can VSFS support larger files?

Inode → Data Blocks



Only 12 (of the 15 pointers in the inode) are direct pointers

(A 4KiB block can fit 1024 pointers)

With 12 direct + 1 indirect ptr:

With 12 direct, 1 indirect, 1 double-indirect:

With 12 direct, 1 indirect, 1 double-indirect, 1 triple-indirect:

Some file systems use balanced inode trees.

Inode → Data Blocks

Really small files

- <60 bytes
  - No need to have a separate data block
- Store the data in the 60 byte area otherwise used for the pointers to the data blocks

Extent-based approach used by some file systems:

- Each pointer addresses multiple blocks (instead of 1 block)
- <Starting block, num blocks>
- Compare with 1block-pointers:
  - Pros?
  - Cons?

Magic Numbers

Typical systems-y technique

- Embed magic numbers in each data structure
- Used to detect corruption

Examples:

- 0xdeadbeef: Slab allocator
- File identification: ELF (0x7fELF), Java class (0xcafebabe), shell script (0x2321 aka #! or shebang)
- Superblock ext2/3/4: 0xef53, FFS (0x19540119), ZFS (0x0bab10c)
- 0x8badf00d: iOS app terminated due to watchdog timeout
- 0xc0010ff: iOS app terminated due to thermal event
- 0xdead10cc: iOS app terminated because it held onto system resource while running in bg
- 0xbaddcafe: Solaris marks uninitialized kernel memory
- 0xcafefeed: Solaris marks kmemfree() memory

Directory Organization

inum	reclen	strlen	name
5	12	2	.
2	12	3	..
12	12	4	foo
13	12	4	bar
24	36	28	foobar_is_a_pretty_longname

What is the inum of this directory?

What is the inum of its parent directory?

Where is a directory's content stored?

Creating and Writing File

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]
create (/foo/bar)		read write	read	read write	read write	read	read write		
write()	read write				read write			write	
write()	read write				read write				write

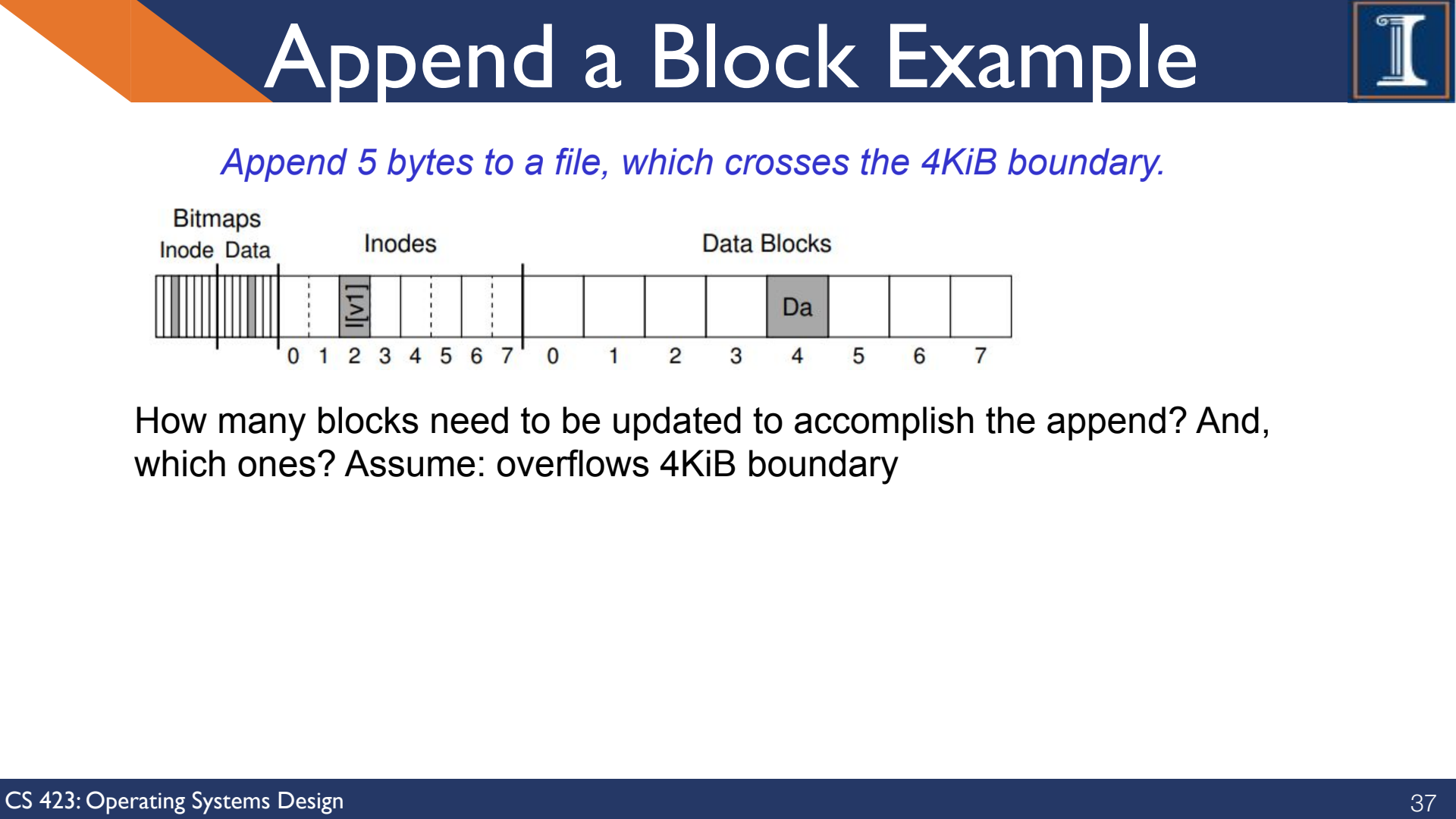
create (open) op:

- Why read foo's data?
- What is written in foo's data?
- What is written in foo's inode

write op:

- Why is bar inode written after each data write?

Append a Block Example

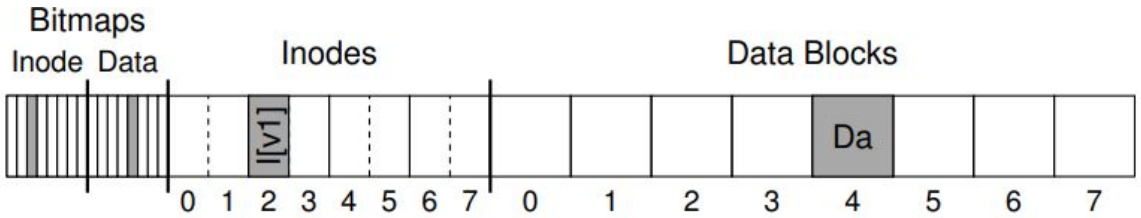


Append 5 bytes to a file, which crosses the 4KiB boundary.

How many blocks need to be updated to accomplish the append? And, which ones? Assume: overflows 4KiB boundary

Append a Block Example

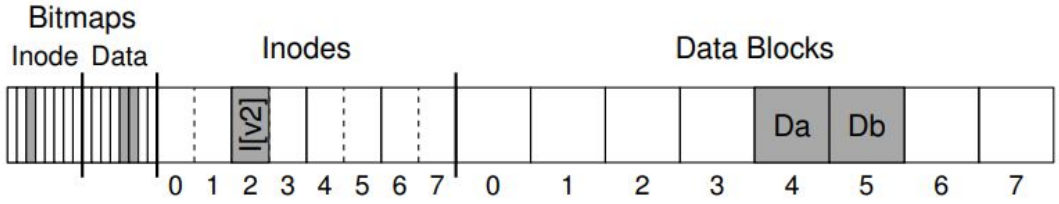
Append 5 bytes to a file, which crosses the 4KiB boundary.



How many blocks need to be updated to accomplish the append? And, which ones? Assume: overflows 4KiB boundary

1. Allocate a new block. Add pointer to this new block–either new direct block pointer in inode (if eventual size is <= 48 KiB), or in one of the indirect blocks (single or double). Allocate new indirect block(s) if eventual size extends past the ~4MiB, ~4GiB, ~4TiB boundary. Each newly allocated block requires a bit to be flipped in the datablock-bitmap.
2. The actual data to the last 2 data blocks of the file.
3. Update inode's metadata: mtime, size, block pointer, and #blocks.

Problem Cases



Append 5 bytes to a file, which crosses the 4KiB boundary.

Say a new block (Db) was needed. What if:

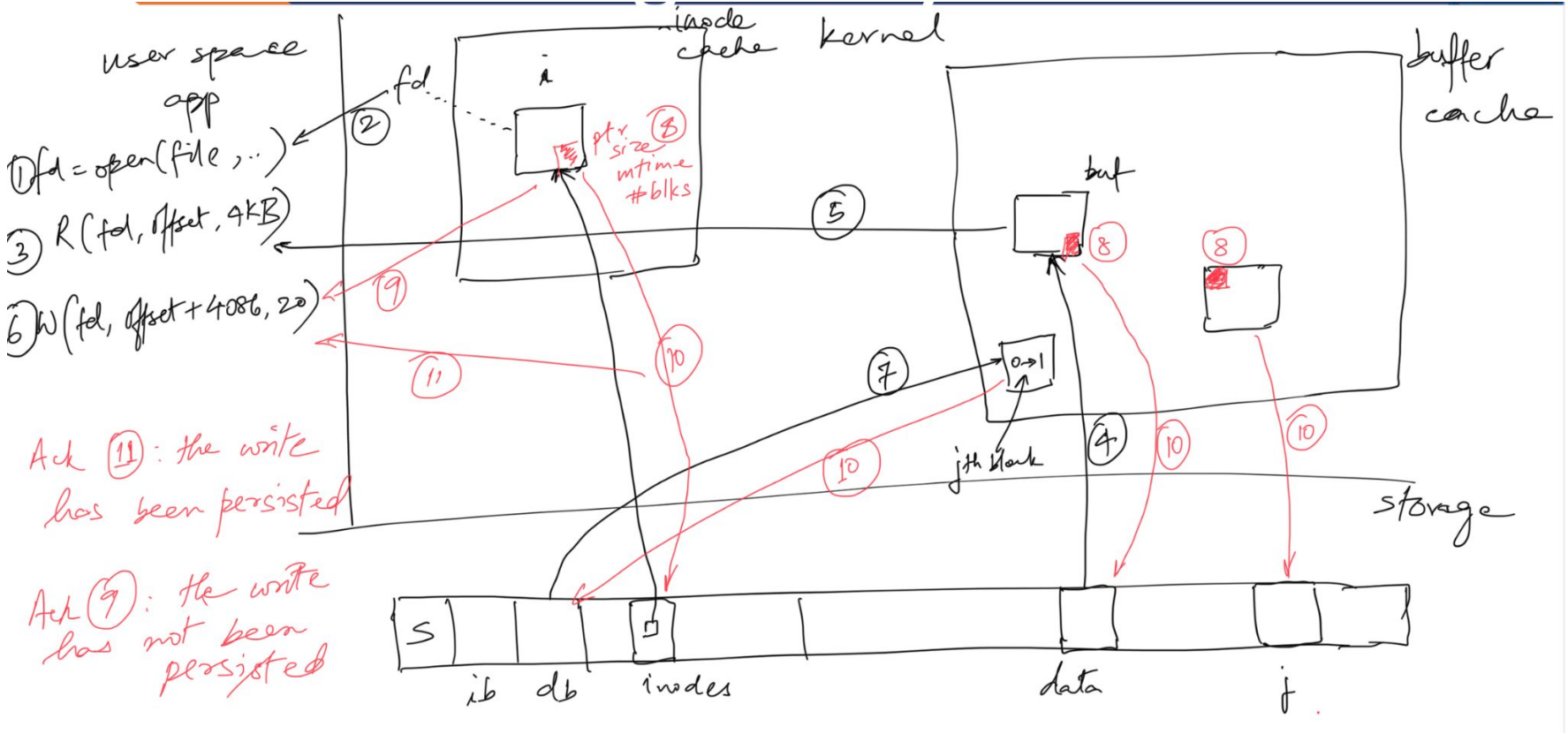
1. Only Db is persisted?
2. Only inode update is persisted?
3. Only datablock bitmap update is persisted?
4. Only datablock bitmap update and data are persisted?
5. Only data and inode update are persisted?
6. Only datablock bitmap and inode updates are persisted?

What's special about the last case?

File System Caching

- Storage access is expensive
  - Cache data & metadata blocks in memory
  - All FSs do this
- Modern general-purpose OSes (Linux, Solaris, WinNT)
  - Combine file system cache & virtual memory
  - Unified page cache; mmap() can use the same page
- Write buffering
  - Persist updates when fsync is invoked
  - Flushing deamon

A Working File System



Crash Consistency

Basic problem:

- All updates (for an op) must get persisted **atomically**

But failures happen: kernel panic, power failures

- Metadata self-consistency
- Data consistency: data must "make sense" to user application

FSCK

Let inconsistencies happen; fix up during reboot
File system check, aka fsck (pronounced fisk)

```
UNEXPECTED SOFT UPDATE INCONSISTENCY
** Last Mounted on /
** Root file system
** Phase 1 - Check Blocks and Sizes
** Phase 2 - Check Pathnames
** Phase 3 - Check Connectivity
** Phase 4 - Check Reference Counts
UNREF FILE I=9470237  OWNER=mysql MODE=100600
SIZE=0 MTIME=Feb  9 06:52 2016

CLEAR? no

** Phase 5 - Check Cyl groups
FREE BLK COUNT(S) WRONG IN SUPERBLK
SALVAGE? no

SUMMARY INFORMATION BAD
SALVAGE? no

BLK(S) MISSING IN BIT MAPS
SALVAGE? no

722171 files, 11174866 used, 8118876 free (156260 frags, 995327 blocks, 0.8% fra
gmentation)
\[\033[01;34m\]root@\[\033[00m\]:\[\033[01;34m\]/\[\033[00m\]#
```