File Systems

DSCI 551

Wensheng Wu

root directory: cd /

Roadmap

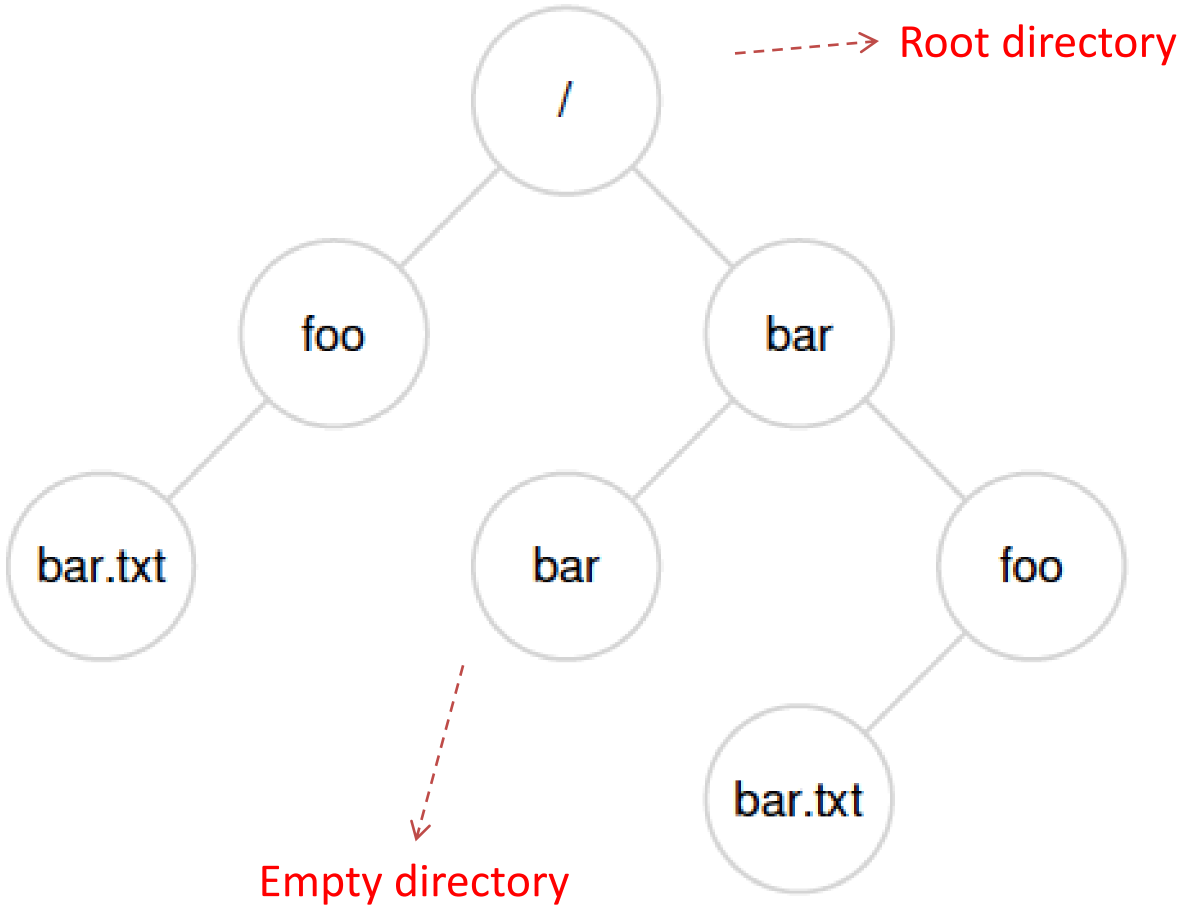
- Files and directories
  - CRUD operations via system calls
- Implementing CRUD
  - Data structures, e.g., organization of blocks
  - Access methods: turn system calls into operations on data structures

Files and directories

- File content stored in blocks on storage device
  - Has user defined name: hello.txt
  - & low-level name, e.g., inode number: 410689
- Files are organized into directories (folders)
  - each may have a list of files and/or subdirectories
  - That is, directories can be nested

Every file has meta data
Meta data stored in inode
File system manages the storage
So that every file has storage to store data, it's meta data
Inode is array
Index starting at 0 will be the inode number

Example



Root directory
/
foo
bar
bar.txt
bar
foo
bar.txt
Empty directory

Operations on files

- Create
  - open(), write()
- Read
  - open(), read(), lseek()
- Update
  - write(), lseek()
- Delete
  - unlink()

System calls:
calls to functions in the API
provided by OS

Create

- User interface via GUI or touch command in Linux
- Implementation, e.g., via a C program with a system call: open()

|: Bitwise OR operator

- int fd = open("foo", O_CREAT | O_WRONLY | O_TRUNC);
  - Open with flags indicating the specifics
  - O_CREAT: create a file
  - O_WRONLY: write only
  - O_TRUNC: remove existing contents if exits

fd: a file descriptor (the smallest unused descriptor) if open successfully, otherwise return -1.

more detail: https://pubs.opengroup.org/onlinepubs/7908799/xsh/open.html
or [AA] Chapter39 p4

File descriptor

- Note open() returns a file descriptor
  - Typically an integer
  - Reserved fds: stdin 0, stdout, 1, stderr 2

A file descriptor is just an integer, private per process, and is used in UNIX systems to access files; thus, once a file is opened, you use the file descriptor to read or write the file, assuming you have permission to do so. In this way, a file descriptor is a capability [L84], i.e., an opaque handle that gives you the power to perform certain operations. Another way to think of a file descriptor is as a pointer to an object of type file; once you have such an object, you can call other “methods” to access the file, like read() and write()

Read

- read(fd, buffer, size)
  - Read from file "fd" <*size*> number of bytes
  - And store them in *buffer*

the return value of read is stored in `buffer`

- Read starts from the current offset of fd
  - Initially 0

往年试题:

int n = read(fd, buffer, size)

n: the number of bytes read (zero indicates end of file) and the file position is advanced by this number. alternatively, -1 is returned when error occurs.

Write

- write(fd, buffer, size)
  - Write to file *fd* <*size*> number of bytes stored in buffer
  - Also start writing from the current offset

没考到过? ？ 建议搜一下记下来

Random read and write

- off_t lseek(int fd, off_t offset, int whence)
  - If whence is SEEK_SET, the offset is set to <offset> bytes from the beginning of file
  - 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 (typically offset is negative, e.g., -8 for 8 bytes from the end)

- whence: from where

this is the implementation of ‘cp’ command

```
#define BUF_SIZE 8192

int main(int argc, char* argv[]) {

    int input_fd, output_fd;     /* Input and output file descriptors */
    ssize_t ret_in, ret_out;     /* Number of bytes returned by read() and write() */
    char buffer[BUF_SIZE];       /* Character buffer */

    /* Are src and dest file name arguments missing */
    if(argc != 3){
        printf ("Usage: cp file1 file2\n");
        return 1;
    }

    /* Create input file descriptor */
    input_fd = open (argv [1], O_RDONLY);
    if (input_fd == -1) {
            perror ("open");
            return 2;
    }

    /* Create output file descriptor */
    /* WRONLY will truncate file to zero length if exists */
    output_fd = open(argv[2], O_WRONLY | O_CREAT, 0644);
    if(output_fd == -1){
        perror("open");
        return 3;
    }

    /* Copy process */
    while((ret_in = read (input_fd, &buffer, BUF_SIZE)) > 0){
            ret_out = write (output_fd, &buffer, (ssize_t) ret_in);
            if(ret_out != ret_in){
                /* Write error */
                perror("write");
                return 4;
```

Copy a file

"0" starts an octal number
=> permissions:
110 (owner) rw-
100 (group) r--
100 (others) r--

Pointer to a character array

File permission mode

```
Vincent@Vincent-PC ~/usc/551-fa16/Amazon
$ copy2
Usage: cp file1 file2

Vincent@Vincent-PC ~/usc/551-fa16/Amazon
$ copy2 copy2.c copy2-a.c

Vincent@Vincent-PC ~/usc/551-fa16/Amazon
$ ls -l
total 95
-rwxrwx---+ 1 Vincent None     3 Aug 30 18:04 a.txt
-rwxrwx---+ 1 Vincent None     0 Aug 30 18:04 a.txt~
-rwxrwx---+ 1 Vincent None  1568 Jan 31  2016 copy2.c
-rwxrwxr-x+ 1 Vincent None 64289 Sep 10 11:45 copy2.exe
-rw-r--r--+ 1 Vincent None  1568 Sep 10 11:45 copy2-a.c
-rwxrwx---+ 1 Vincent None   426 Aug 31 15:18 HelloWorld.class
-rwxrwx---+ 1 Vincent None   239 Aug 30 18:02 HelloWorld.java
-r--------+ 1 Vincent None  1698 Aug 23 17:18 inf551.pem
-rwxrwx---+ 1 Vincent None  1464 Aug 23 20:56 inf551.ppk
-r--------+ 1 Vincent None  1694 Aug 31 14:48 inf551-a.pem
-r--------+ 1 Vincent None  1698 Aug 31 17:28 inf551-b.pem
-rwxrwx---+ 1 Vincent None  1464 Aug 31 17:39 inf551-b.ppk
```

rw-r--r–

=> 110 (owner permission) 100 (group) 100 (others)

Resources for system calls

- https://en.wikipedia.org/wiki/System_call
- open: https://en.wikipedia.org/wiki/Open_(system_call)
- read: https://en.wikipedia.org/wiki/Read_(system_call)
- write: https://en.wikipedia.org/wiki/Write_(system_call)
- close: https://en.wikipedia.org/wiki/Close_(system_call)

Resources for system calls

- man –S 2 read
  - Find it in the Section 2 of the manual

Install gcc on EC2

- sudo yum install gcc 这个我为什么装不上啊？有bug

- Usage:
  - gcc -o copy2 copy2.c

File and directory

- When creating a file
  - Bookkeeping data structure (inode) created: recording size of file, location of its blocks, etc.
  - Linking a human-readable name to the file
  - Putting the link in a directory

Info about file (stored in inode)

```
struct stat {
    dev_t st_dev; /* ID of device containing file */
    ino_t st_ino; /* inode number */
    mode_t st_mode; /* protection */
    nlink_t st_nlink; /* number of (hard) links */
    uid_t st_uid; /* user ID of owner */
    gid_t st_gid; /* group ID of owner */
    dev_t st_rdev; /* device ID (if special device file, e.g., /etc/tty) */
    off_t st_size; /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for filesystem I/O */
    blkcnt_t st_blocks; /* number of blocks allocated */
    time_t st_atime; /* last time file content was accessed */
    time_t st_mtime; /* last time file content was modified */
    time_t st_ctime; /* last time inode was changed */
};
```

Execute "man -S 2 stat" for more details…

inode

- Stores metadata/attributes about the file

- Also stores locations of blocks holding the content of the file

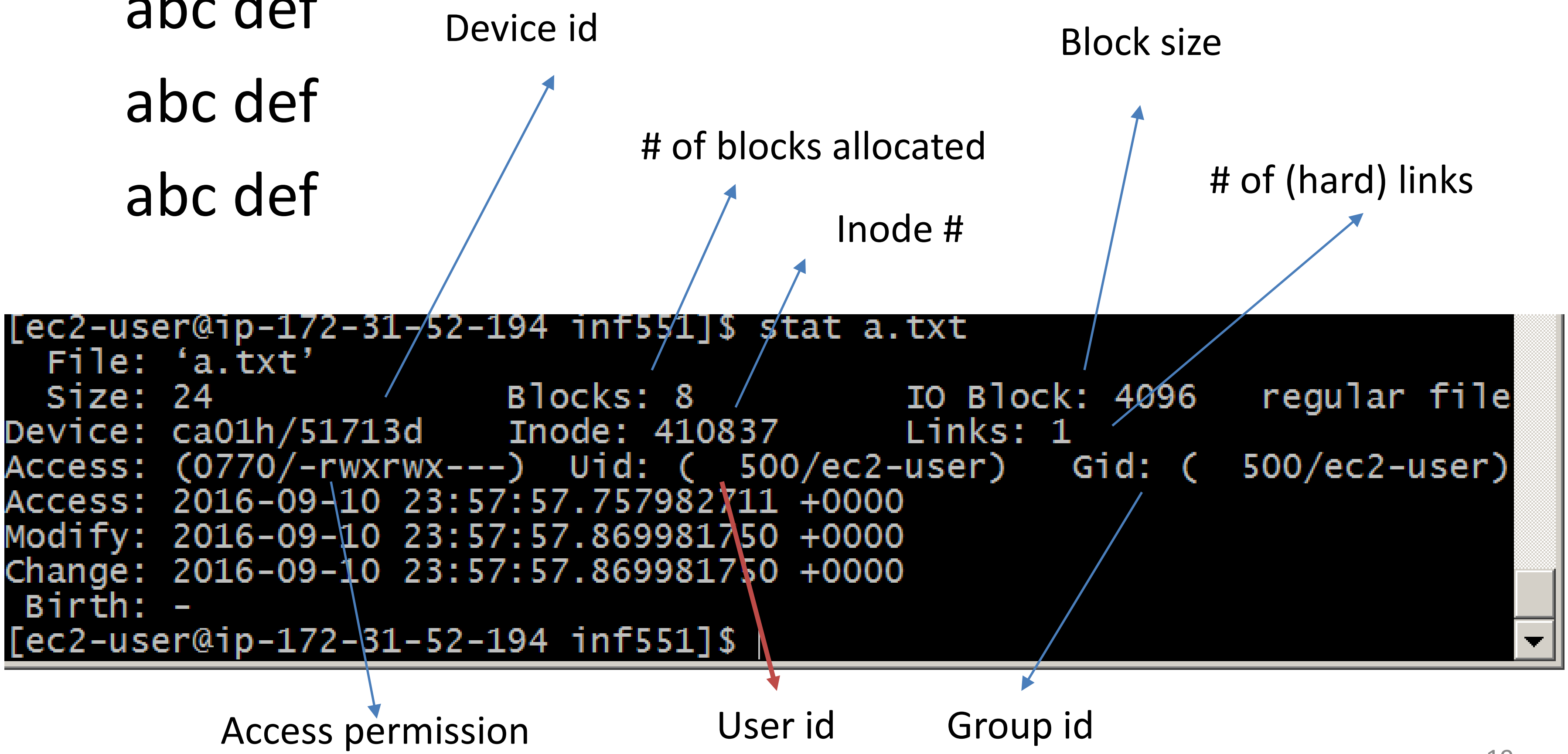
Example

- a.txt

  abc def

  abc def

  abc def



noatime

```
[ec2-user@ip-172-31-29-80 inf55x]$ cat /etc/fstab
#
LABEL=/     /           ext4    defaults,noatime  1   1
tmpfs       /dev/shm    tmpfs   defaults          0   0
devpts      /dev/pts    devpts  gid=5,mode=620    0   0
sysfs       /sys        sysfs   defaults          0   0
proc        /proc       proc    defaults          0   0
```

Hard links

```
[ec2-user@ip-172-31-29-80 inf55x]$ ln a.txt b.txt
[ec2-user@ip-172-31-29-80 inf55x]$ stat a.txt b.txt
  File: ‘a.txt’
  Size: 0               Blocks: 0          IO Block: 4096   regular empty file
Device: ca01h/51713d    Inode: 13676       Links: 2
Access: (0400/-r--------)  Uid: (  500/ec2-user)   Gid: (  500/ec2-user)
Access: 2019-10-14 23:47:07.999160774 +0000
Modify: 2020-01-28 17:57:36.995016395 +0000
Change: 2020-01-28 18:20:38.370213951 +0000
 Birth: -
  File: ‘b.txt’
  Size: 0               Blocks: 0          IO Block: 4096   regular empty file
Device: ca01h/51713d    Inode: 13676       Links: 2
Access: (0400/-r--------)  Uid: (  500/ec2-user)   Gid: (  500/ec2-user)
Access: 2019-10-14 23:47:07.999160774 +0000
Modify: 2020-01-28 17:57:36.995016395 +0000
Change: 2020-01-28 18:20:38.370213951 +0000
 Birth: -
```

Symbolic links

```
[ec2-user@ip-172-31-29-80 inf55x]$ ln -s a.txt c.txt
[ec2-user@ip-172-31-29-80 inf55x]$ stat a.txt c.txt
  File: ‘a.txt’
  Size: 0               Blocks: 0          IO Block: 4096   regular empty file
Device: ca01h/51713d    Inode: 13676       Links: 2
Access: (0400/-r--------)  Uid: (  500/ec2-user)   Gid: (  500/ec2-user)
Access: 2019-10-14 23:47:07.999160774 +0000
Modify: 2020-01-28 17:57:36.995016395 +0000
Change: 2020-01-28 18:20:38.370213951 +0000
 Birth: -
  File: ‘c.txt’ -> ‘a.txt’
  Size: 5               Blocks: 0          IO Block: 4096   symbolic link
Device: ca01h/51713d    Inode: 14121       Links: 1
Access: (0777/lrwxrwxrwx)  Uid: (  500/ec2-user)   Gid: (  500/ec2-user)
Access: 2020-01-28 18:20:55.981591917 +0000
Modify: 2020-01-28 18:20:55.981591917 +0000
Change: 2020-01-28 18:20:55.981591917 +0000
 Birth: -
```

Working with directories

- Create: mkdir() system call
  - Used to implement command, e.g., mkdir xyz
- Read: opendir(), readdir(), closedir()
  - ls xyz
- Delete: rmdir()

Roadmap

- Files and directories
  - CRUD operations

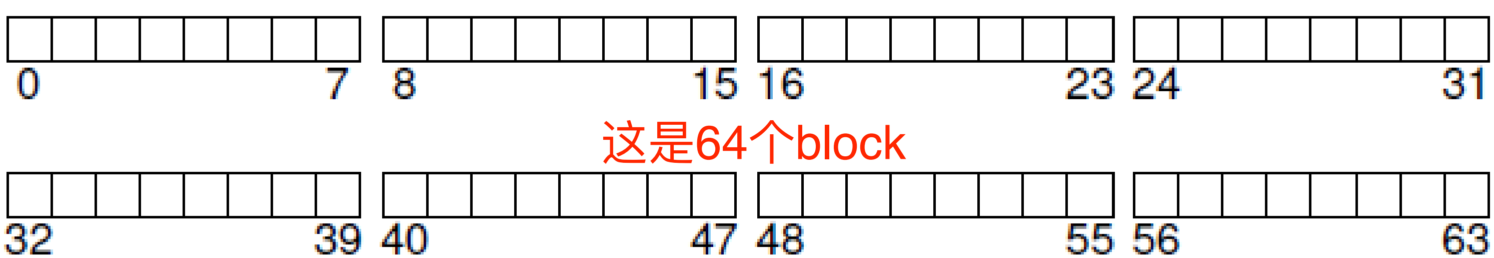
- Implementation
  - Data structures: how to organize the blocks
  - Access methods: turn system calls into operations on data structures

Organization of blocks

- Array-based
  - Disk consists of a list of blocks
  - We will assume this
- Tree-based, e.g., SGI XFS
  - Blocks are organized into variable-length extents
  - Use B+-tree to quickly find free extents

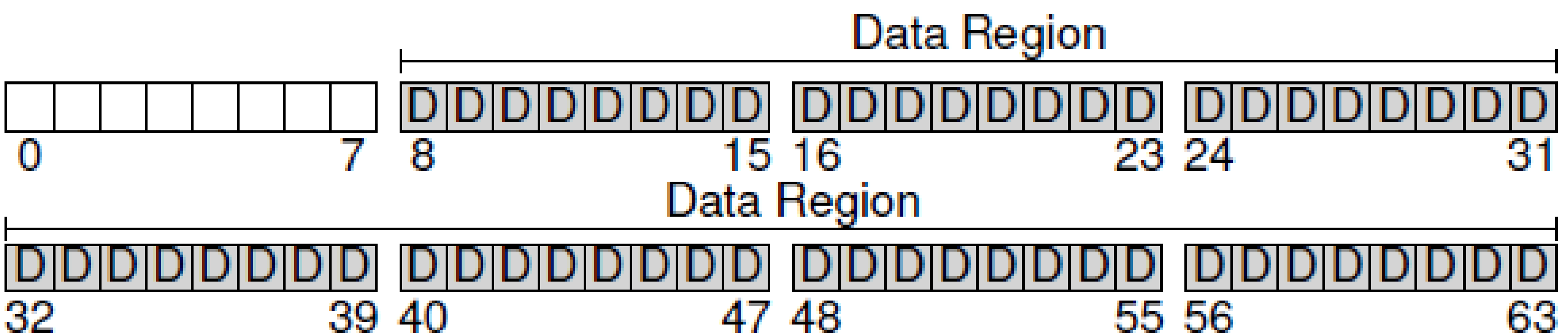
Blocks

- Consider a disk with 64 blocks
  - 4KB/block
  - 512B/sector (we assume this in this lecture)
- So there are $2^{12}/2^{9} = 2^{3} = 8$ sectors/block
  - Capacity of disk = 64 * 4KB = 256KB



Data region

- 56 blocks (#8-63)
  - used to store data/content of files
  - but see later: some blocks may store pointers



Metadata

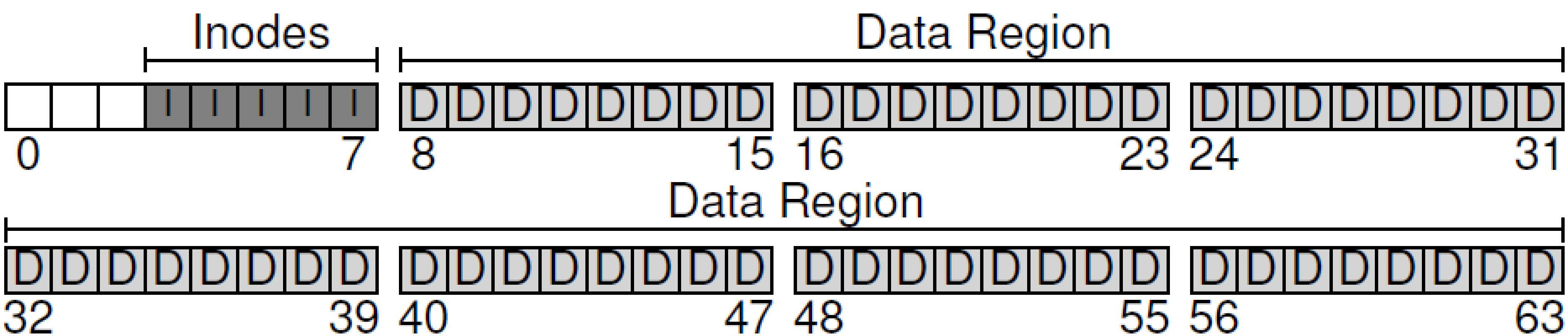
- For each file, file system records its metadata
  - Information in the "stat" struct

```
struct stat {
    dev_t     st_dev;         /* ID of device containing file */
    ino_t     st_ino;         /* inode number */
    mode_t    st_mode;        /* file type and mode */
    nlink_t   st_nlink;       /* number of hard links */
    uid_t     st_uid;         /* user ID of owner */
    gid_t     st_gid;         /* group ID of owner */
    dev_t     st_rdev;        /* device ID (if special file) */
    off_t     st_size;        /* total size, in bytes */
    blksize_t st_blksize;     /* blocksize for filesystem I/O */
    blkcnt_t  st_blocks;      /* number of 512B blocks allocated */

    /* Since Linux 2.6, the kernel supports nanosecond
       precision for the following timestamp fields.
       For the details before Linux 2.6, see NOTES. */

    struct timespec st_atim;  /* time of last access */
    struct timespec st_mtim;  /* time of last modification */
    struct timespec st_ctim;  /* time of last status change */
```

  - Location of blocks that stores the content of file

Metadata of files stored in inodes

- Index nodes
- Stored in blocks #3 -- #7 (i.e., 5 blocks)
- Together they are called the "inode" table



How many inodes are there?

- 256 bytes/inode
- 5 blocks, 4KB/block

=> 16 inodes/block (4K/256 = $2^{12}/2^{8}$)

=> 5 blocks, 5 * 16 = 80 inodes

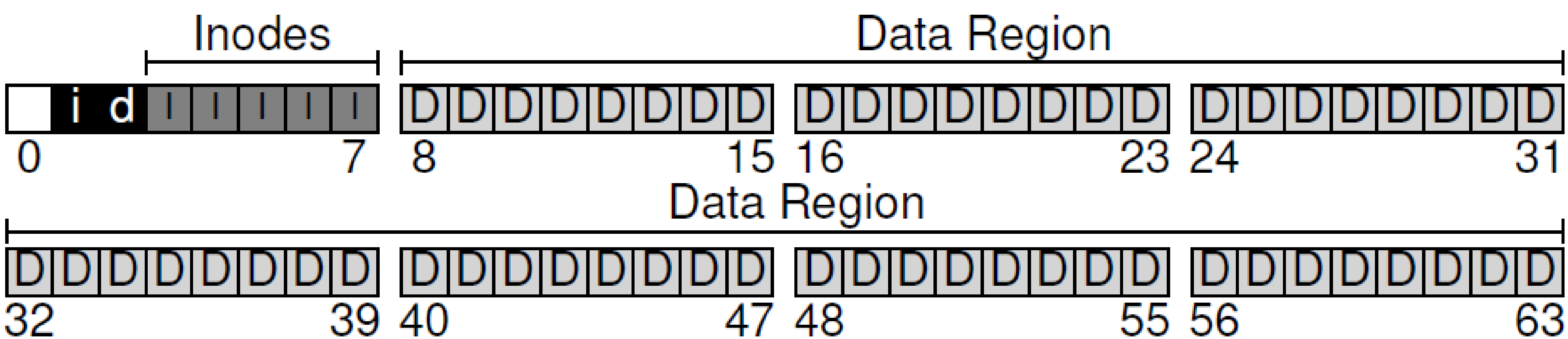
=> File system can store at most 80 files

Free space management using bitmaps

- Bitmap: a vector of bits
  - 0 for free (inode/block), 1 for in-use
- Inode bitmap (imap)
  - keep track of which inodes in the inode table are available
- Data bitmap (dmap)
  - Keep track of which blocks in data region are available

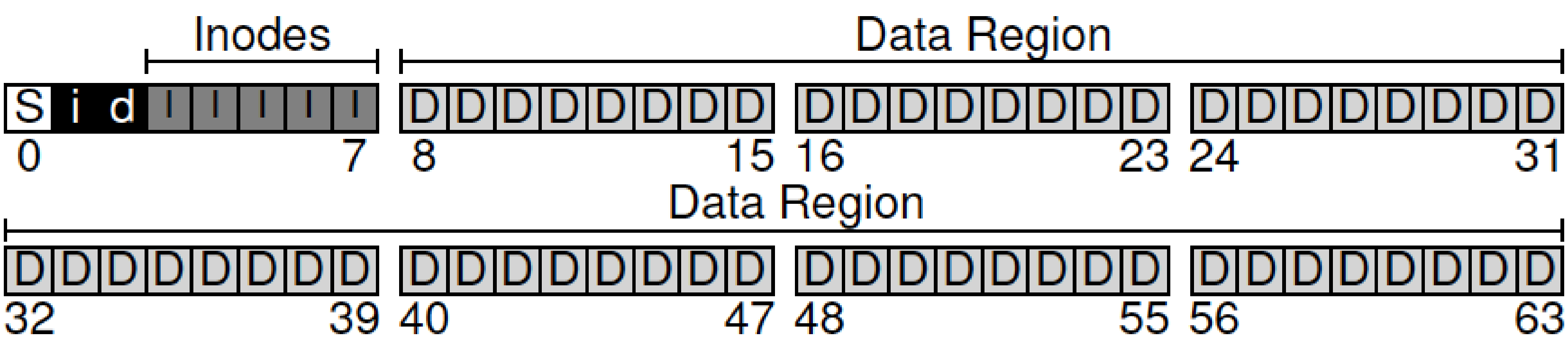
Bitmaps

- Each bitmap is stored in a block
  - Block "i": keep track of 80 inodes (could track 32K)
  - Block "d": keep track of the 56 data blocks



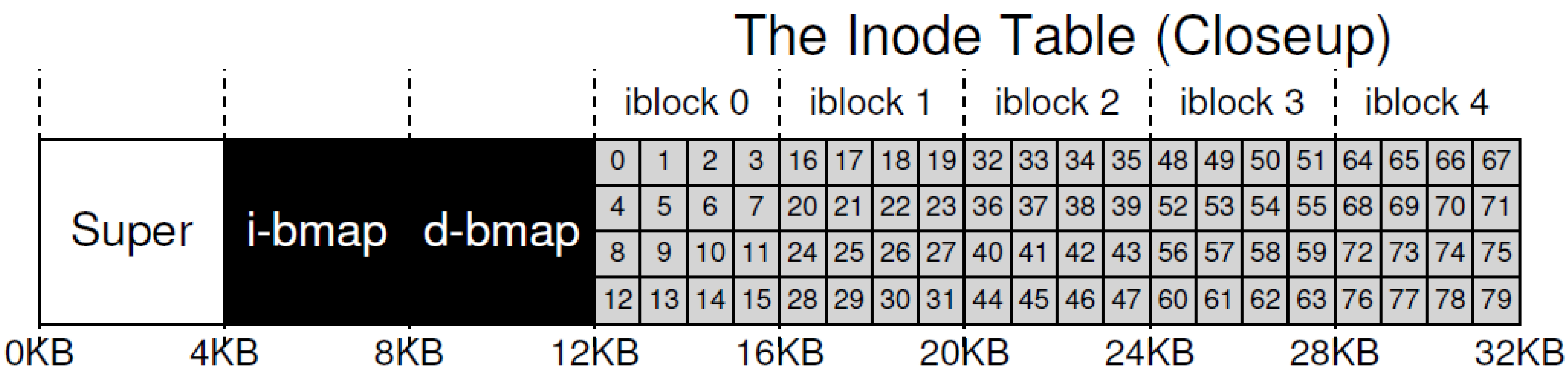
Superblock

- Track where i/d blocks and inode table are
  - E.g., inode table starts at block 3; there are 80 inodes and 56 data blocks, etc.
- Indicate type of FS & inumber of its root dir
- Will be read first when file system is mounted



inumber

- Each inode is identified by a number
  - Low-level number of file name
- Can figure out location of inode from inumber



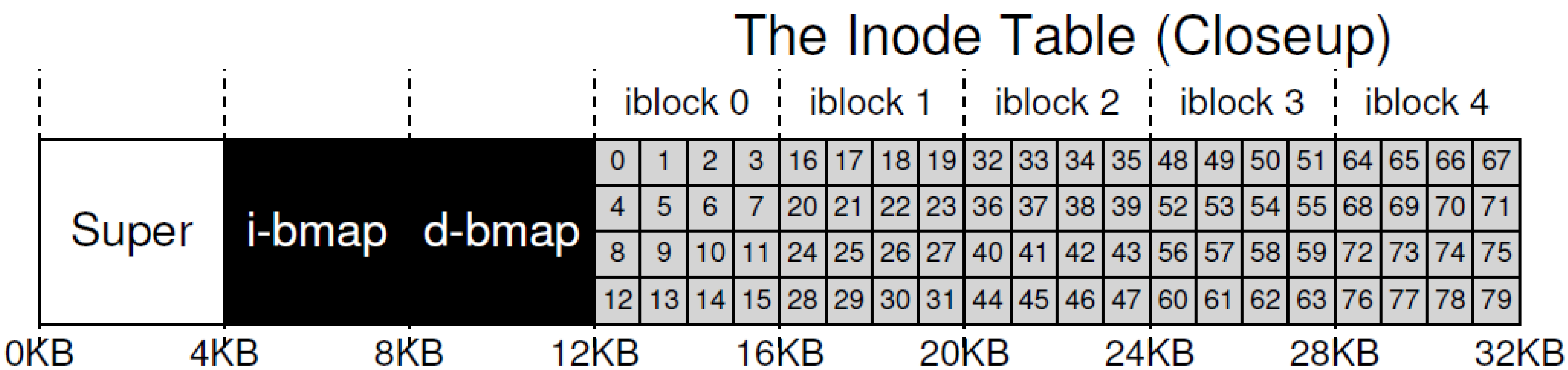
Super stores the inumber of

inumber => location

- inumber = 32

=> address: offset in bytes from the beginning

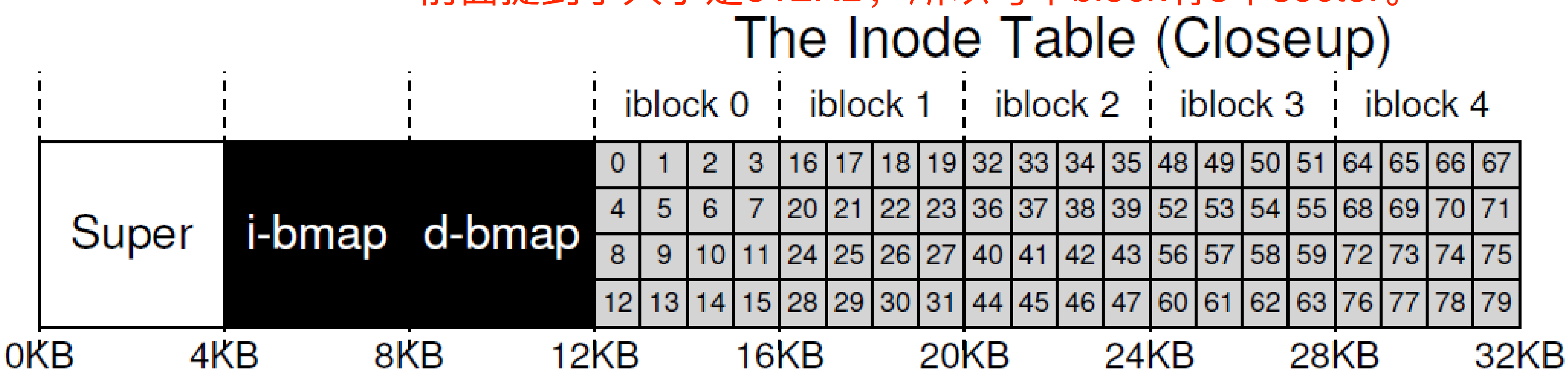
=> which sector?



inumber => location of inode

- Address: 12K + 32 * 256 = 20K
- Sector #: 20K/512 = 40
  - more generally
  - $\lfloor (\text{inodeStartAddress} + \text{inumber} * \text{inode size})/\text{sector size} \rfloor$

sector这个概念就非常奇怪，在这个图也没表现出来。只是前面提到了大小是512KB，所以每个block有8个sector。



inode => location of data blocks

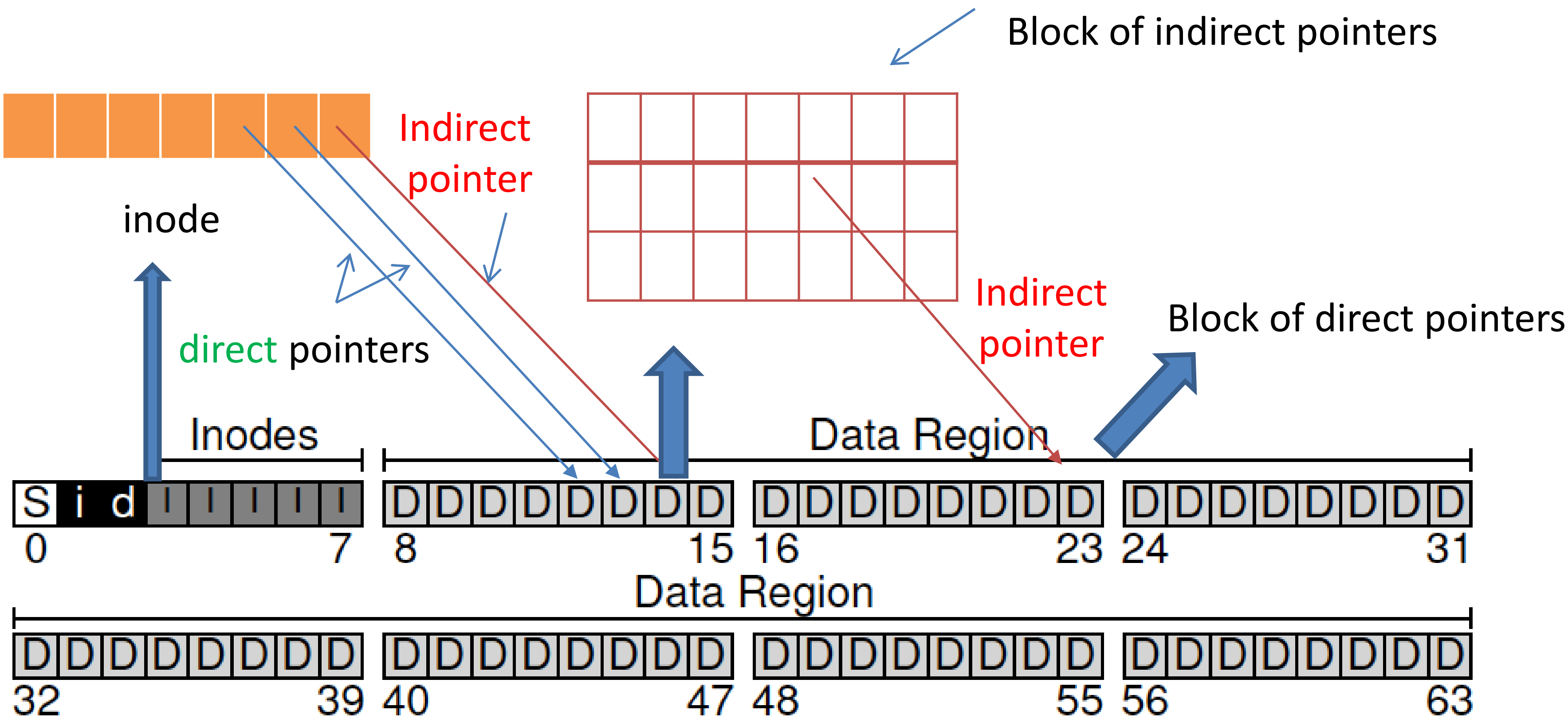
- A number of direct pointers
  每个inode里有8个pointer
  - E.g., 8 pointers, each points to a data block
  - Enough for 8*4K = 32K size of file

- Also has a slot for indirect pointer
  - Pointing to a data block storing direct pointers
  - Assume 4 bytes for block address (e.g., represented in CHS), so 1024 pointers/block
  - Now file can have (8 + 1024) blocks or 4,128KB

Multi-level index

- Pointers may be organized into multiple levels
  - Indirect pointer (as in previous slide)
    - Inode (pointer1, pointer2, …, indirect pointer)
    - Indirect pointer -> a block of direct pointers
  - Double indirect pointers
    - Inode (pointer1, pointer2, …, indirect pointer)
    - Indirect pointer -> a block of indirect pointers instead
      -> each points to a block of direct pointers
  - Triple indirect pointers
    - Indirect pointer -> a block of indirect pointers
      -> each points to a block of indirect pointers
      -> each points to a block of direct pointers

Double Indirect Pointers



Advantages of multi-level index

- Grow to more levels as needed
- Direct pointers handle most of the cases
  - Many files are small

Directory organization

- Directory itself stored as a file
- For each file in the directory, it stores:
  – name, inumber, record length, string length

inum	reclen	strlen	name
5	4	2	.
2	4	3	..
12	4	4	foo
13	4	4	bar
24	8	7	foobar

Actual length

Record length vs string length

- String length = # of characters in file name + 1 (for \0: end of string)
- Record length >= string length
  - Due to entry reuse

inum	reclen	strlen	name
5	4	2	.
2	4	3	..
12	4	4	foo
13	4	4	bar
24	8	7	foobar

Reusing directory entries

- If file is deleted (using rm command) or a name is unlinked (using unlink command)
  - File is finally deleted when its last (hard) link is removed
- Then inumber in its directory entry set to 0 (reserved for empty entry)
  - So we know it can be reused

Storing a directory

- Also as a file with its own inode + data block

- inode:
  - file type: directory (instead of regular file)
  - pointer to block(s) in data region storing directory entries

Roadmap

- Files and directories
  - CRUD operations
- Implementation
  - Data structures: how to organize blocks, e.g., into array/tree
  - Access methods: turn system calls to operations on data structures

Open for read

- fd = open("/foo/bar", O_RDONLY)

Open for read

- fd = open("/foo/bar", O_RDONLY)
  - Need to locate inode of the file "/foo/bar"
  - Assume inumber of root, say 2, is known (e.g., when the file system is mounted)

Open for read

1. Read inode and content of / (2 reads)
   – Look for "foo" in / -> foo's inumber

2. Read inode and content of /foo (2 reads)
   – Look for "bar" in /foo -> bar's inumber

3. Read inode of /foo/bar (1 read)
   – Permission check + allocate file descriptor

Cost of open()

- Need 5 reads of inode/data block

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

File-open table per process

File descriptor	File name	Inumber	Position offset	…
3	/foo/bar	32382	0	
4	/foo/more	48482	512	…

Reading the file

- read(fd, buffer, size)
  - Note fd is maintained in per-process open-file table
  - Table translates fd -> inumber of file

Reading the file

- read(fd, buffer, size)

read file inode

1. Consult bar's inode to locate a block

2. Read the block

   read file data

   write file inode

3. Update inode with newest file access time
4. Update open-file table with new offset
5. Repeat above steps until done (with reading data of given size)

Cost for reading a block

- 3 I/O's:
  - read inode, read data block, write inode

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[~~1~~] 2
			read							
						read				
open(bar)				read						
							read			
					read					
					read					
read()								read		
					write					
					read					
read()									read	
					write					
					read					
read()										read
					write					

Open for write

- int fd = open("/foo/bar", O_WRONLY)
  - Or int fd = create(("/foo/bar")
  - Assume bar is a new file under foo
  - (note the difference from reading chapter!)

Open for write

- int fd = open("/foo/bar", O_WRONLY)

1. Read '/' inode & content
   - obtain foo's inumber

2. Read '/foo' inode & content
   - check if bar exists

Open for write

imap is bitmap

3. Read imap, to find a free inode for bar

4. Update imap, setting 1 for allocated inode

5. Write bar's inode

Open for write

6. Update foo's content block
   - Adding an entry for bar

7. Update foo's inode
   - Update its modification time

Cost for "open for write"

比书里少了一个read bar inode。因为这里是新建，原本没有bar没有inode。（啊但是书里也是新建，为什么要read bar inode？）

- int fd = open("/foo/bar", O_WRONLY)
- Need 9 I/O's

另外，书里是，先write foo data，再read&write bar inode

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
			read							
						read				
				read						
							read			
create()		read								
		write								
					read write					
							write			
				write						

Writing the file: /foo/bar

1. Read inode of bar (by first looking up its inumber in the file-open table)
2. Allocate new data block
   - Read and write bmap
3. Write to data block of bar
4. Update bar inode
   - new modification time, add pointer to block

Cost of writing /foo/bar

- 5 I/O's for write a block

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
create()			read							
						read				
				read						
							read			
		read								
		write								
					write					
							write			
				write						
write()					read					
	read									
	write									
								write		
					write					

Caching for read

- First read may be slow
  - But subsequent ones will speed up
- Good idea to cache popular blocks
  - e.g., determined via LRU strategy

Buffering for delayed write

- Improve write performance via:
  by delaying writes
  - Batching (e.g., two updates to the same imap)
  - Scheduling (reordering for better performance)
  - Avoiding writes (if file created, then quickly deleted)

- Problem: update may be lost when system crashes

Example file systems

- NTFS
  - New technology file system, Microsoft proprietary
- FAT
  - File allocation table
  - FAT 16, 32, …
  - 32 bits = # of sectors a file can occupy

    512B/sector => 2TB limit on file size

    4KB/sector => 16TB limit
- Ext4
  - fourth extended file system, common in Linux