

File Systems

INF 551

Wensheng Wu

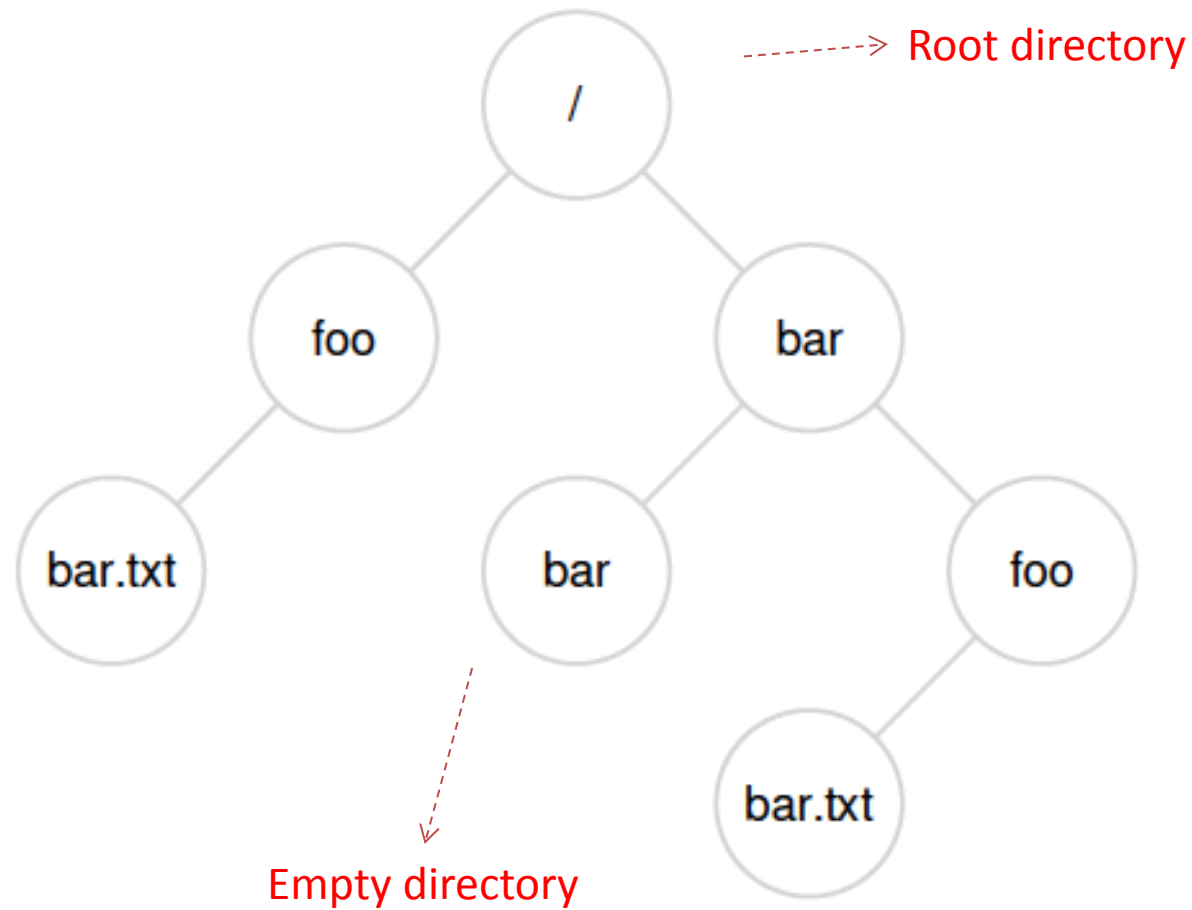
Roadmap

- Files and directories
 - CRUD operations
- How to implement them
 - Data structures
 - Access methods

Files and directories

- File: 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


Example



Operations on files

- Create
- Read
- Update
- Delete

Create

- User interface, e.g., via GUI
 - Implementation, e.g., via a C program with a call to system function `open()`
 - `int fd = open("foo", O_CREAT | O_WRONLY | O_TRUNC);`
 - Open with flags indicating the specifics
 - `O_CREATE`: create a file
 - `O_WRONLY`: write only
 - `O_TRUNC`: remove existing contents if exists
-  `|`: Bitwise OR operator

File descriptor

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

Read

- `read(fd, buffer, size)`
 - Read from file “fd” *<size>* number of bytes
 - And store them in *buffer*
- Read starts from the current offset of fd
 - Initially 0

Write

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

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 (often offset is negative, e.g., -8 for 8 bytes from the end)
- `whence`: from where

```
#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;
    }
```

Copy a file

```
    /* Create input file descriptor */
```

```
    input_fd = open (argv [1], O_RDONLY);
    if (input_fd == -1) {
        perror ("open");
        return 2;
    }
```

"0" starts an octal number
=> permissions:

110 (owner) rw-
100 (group) r--
100 (others) r--

```
    /* 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;
    }
```

Pointer to a character array

```
    /* 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;
        }
    }
```

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 \(system call\)](https://en.wikipedia.org/wiki/Open_(system_call))
- read:
[https://en.wikipedia.org/wiki/Read \(system call\)](https://en.wikipedia.org/wiki/Read_(system_call))
- write:
[https://en.wikipedia.org/wiki/Write \(system call\)](https://en.wikipedia.org/wiki/Write_(system_call))
- close:
[https://en.wikipedia.org/wiki/Close \(system call\)](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 groupinstall "Development Tools"`
 - Will install other dev. tools too
 - E.g., perl, bison, flex, automake, autoconf
- 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 file) */  
    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; /* time of last access */  
    time_t st_mtime; /* time of last modification */  
    time_t st_ctime; /* time of last change to inode, e.g., permission  
    */  
};
```

Example

- a.txt

abc def

abc def

abc def

Device id

Block size

of blocks allocated

Inode #

of hard links

```
[ec2-user@ip-172-31-52-194 inf551]$ stat a.txt
  File: 'a.txt'
  Size: 24          Blocks: 8          IO Block: 4096   regular file
Device: ca01h/51713d Inode: 410837       Links: 1
Access: (0770/-rwxrwx---)  Uid: (  500/ec2-user)   Gid: (  500/ec2-user)
Access: 2016-09-10 23:57:57.757982711 +0000
Modify: 2016-09-10 23:57:57.869981750 +0000
Change: 2016-09-10 23:57:57.869981750 +0000
 Birth: -
[ec2-user@ip-172-31-52-194 inf551]$
```

Access permission

User id

Group id

Hard link example

- In a.txt b.txt
 - b.txt is a hard link

```
[ec2-user@ip-172-31-52-194 inf551]$ stat a.txt
  File: 'a.txt'
  Size: 24          Blocks: 8          IO Block: 4096   regular file
Device: ca01h/51713d Inode: 410837       Links: 1
Access: (0400/-r-----)  Uid: ( 500/ec2-user)   Gid: ( 500/ec2-user)
Access: 2016-09-10 23:57:57.757982711 +0000
Modify: 2016-09-10 23:57:57.869981750 +0000
Change: 2016-09-11 00:30:14.353707948 +0000
 Birth: -
[ec2-user@ip-172-31-52-194 inf551]$ ln a.txt b.txt
[ec2-user@ip-172-31-52-194 inf551]$ stat a.txt b.txt
  File: 'a.txt'
  Size: 24          Blocks: 8          IO Block: 4096   regular file
Device: ca01h/51713d Inode: 410837       Links: 2
Access: (0400/-r-----)  Uid: ( 500/ec2-user)   Gid: ( 500/ec2-user)
Access: 2016-09-10 23:57:57.757982711 +0000
Modify: 2016-09-10 23:57:57.869981750 +0000
Change: 2016-09-11 00:30:26.869603052 +0000
 Birth: -
  File: 'b.txt'
  Size: 24          Blocks: 8          IO Block: 4096   regular file
Device: ca01h/51713d Inode: 410837       Links: 2
Access: (0400/-r-----)  Uid: ( 500/ec2-user)   Gid: ( 500/ec2-user)
Access: 2016-09-10 23:57:57.757982711 +0000
Modify: 2016-09-10 23:57:57.869981750 +0000
Change: 2016-09-11 00:30:26.869603052 +0000
 Birth: -
[ec2-user@ip-172-31-52-194 inf551]$ |
```

of hard links
Increased to 2

Same inode

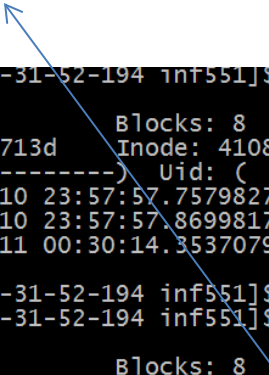
Hard links and limitations

- Hard link: a new name for the same file
 - Added to the current directory (where ln is executed)
 - Refer to the same inode
 - No copying of data involved
- Limitations
 - Cannot create hard link to a directory: avoid cycles in directory tree (e.g., by linking to ancestor directory)
 - Can not link to files on another partition: inode unique only in the same partition

Example

- Entries in directory inf551 (itself stored as a file)

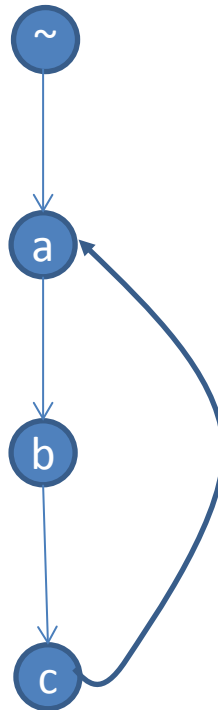
Inumber	Name	Type
410837	a.txt	regular file
410837	b.txt	regular file



```
[ec2-user@ip-172-31-52-194 inf551]$ stat a.txt
  File: 'a.txt'
  Size: 24          Blocks: 8          IO Block: 4096   regular file
Device: ca01h/51713d Inode: 410837      Links: 1
Access: (0400/-r-----)  Uid: ( 500/ec2-user)   Gid: ( 500/ec2-user)
Access: 2016-09-10 23:57:57.757982711 +0000
Modify: 2016-09-10 23:57:57.869981750 +0000
Change: 2016-09-11 00:30:14.353707948 +0000
Birth: -
[ec2-user@ip-172-31-52-194 inf551]$ ln a.txt b.txt
[ec2-user@ip-172-31-52-194 inf551]$ stat a.txt b.txt
  File: 'a.txt'
  Size: 24          Blocks: 8          IO Block: 4096   regular file
Device: ca01h/51713d Inode: 410837      Links: 2
Access: (0400/-r-----)  Uid: ( 500/ec2-user)   Gid: ( 500/ec2-user)
Access: 2016-09-10 23:57:57.757982711 +0000
Modify: 2016-09-10 23:57:57.869981750 +0000
Change: 2016-09-11 00:30:26.869603052 +0000
Birth: -
  File: 'b.txt'
  Size: 24          Blocks: 8          IO Block: 4096   regular file
Device: ca01h/51713d Inode: 410837      Links: 2
Access: (0400/-r-----)  Uid: ( 500/ec2-user)   Gid: ( 500/ec2-user)
Access: 2016-09-10 23:57:57.757982711 +0000
Modify: 2016-09-10 23:57:57.869981750 +0000
Change: 2016-09-11 00:30:26.869603052 +0000
Birth: -
[ec2-user@ip-172-31-52-194 inf551]$ |
```

Directory loop of infinite depth

- `cd ~`
- `mkdir -p a/b`
- `cd a/b`
- `ln .. c`
 - Not allowed!



Number	Name	Type
121	a	directory

Number	Name	Type
123	b	directory


Number	Name	Type
121	c	directory

=> `cd a/b/c/b/c/b/c/b/...`

Delete

- Unlink()
 - Executed when you issue unlink/rm, in Unix
 - E.g., "unlink b.txt" (or "rm b.txt")
 - Link count in the inode for the file will decrement by 1

```
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 */
```



Usage of hard link

- Allow a file to be referenced in multiple locations/directories
- `cd ~`
- `ln inf551/a.txt a.txt`
- This will create another file name under ~
 - Points to the same file (inode) as `inf551/a.txt`

Soft or symbolic link

- Address limitations of hard link
 - E.g., soft link to a directory is allowed
- Stored as a file, containing the file name it links to
 - e.g., `ln -s a.txt c.txt`

Note the different inumber than a.txt

```
[ec2-user@ip-172-31-52-194 inf551]$ stat c.txt
  File: 'c.txt' -> 'a.txt'
  Size: 5                Blocks: 0          IO Block: 4096   symbolic lin
k
Device: ca01h/51713d    Inode: 421501      Links: 1
Access: (0777/lrwxrwxrwx)  Uid: ( 500/ec2-user)   Gid: ( 500/ec2-user)
Access: 2016-09-11 00:53:55.565282218 +0000
Modify: 2016-09-11 00:53:55.565282218 +0000
Change: 2016-09-11 00:53:55.565282218 +0000
Birth: -
```

Symbolic link

- Content of c.txt = a.txt
- c.txt is stored as a file under its directory
 - E.g., inf551

Inumber	Name	Type
410837	a.txt	regular file
410837	b.txt	regular file
421501	c.txt	symbolic link

Symbolic link

- Size of symbolic link (which is stored as a file)
= length of file name

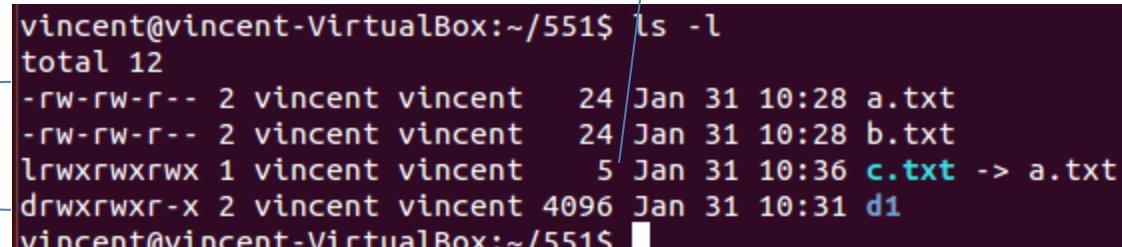
Note the size

Type:

-: regular

l: symbolic link

d: directory



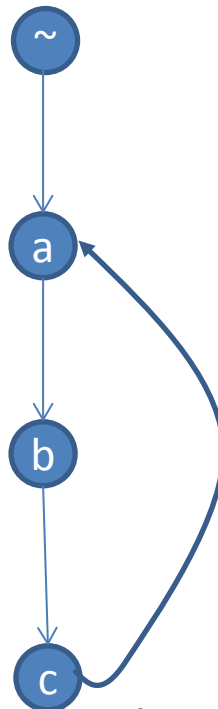
A terminal window showing the output of the command `ls -l`. The output lists four files: `a.txt`, `b.txt`, `c.txt`, and `d1`. The file `c.txt` is a symbolic link pointing to `a.txt`. A blue bracket on the left side of the terminal output groups the first three lines, and an arrow points from the text 'Note the size' to the size column of the `c.txt` entry.

```
vincent@vincent-VirtualBox:~/551$ ls -l
total 12
-rw-rw-r-- 2 vincent vincent  24 Jan 31 10:28 a.txt
-rw-rw-r-- 2 vincent vincent  24 Jan 31 10:28 b.txt
lrwxrwxrwx 1 vincent vincent   5 Jan 31 10:36 c.txt -> a.txt
drwxrwxr-x 2 vincent vincent 4096 Jan 31 10:31 d1
vincent@vincent-VirtualBox:~/551$
```

Directory loop using symbolic link

- `cd ~`
- `mkdir -p a/b`
- `cd a/b`
- `ln -s .. c`

=> `cd a/b/c/b/c/b/c/b/...`



Inumber	Name	Type
121	a	directory

Inumber	Name	Type
123	b	directory

Inumber	Name	Type
124	c	Symbolic link



Content of file c

Note the indirection provided by the symbolic link

Hard vs. soft link

- In `ln a.txt b.txt`
 - `unlink a.txt`
 - `cat b.txt` (OK)
- In `ln -s a.txt b.txt`
 - `unlink a.txt`
 - `cat b.txt` (ERROR: broken link)

Working with directories

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

Roadmap

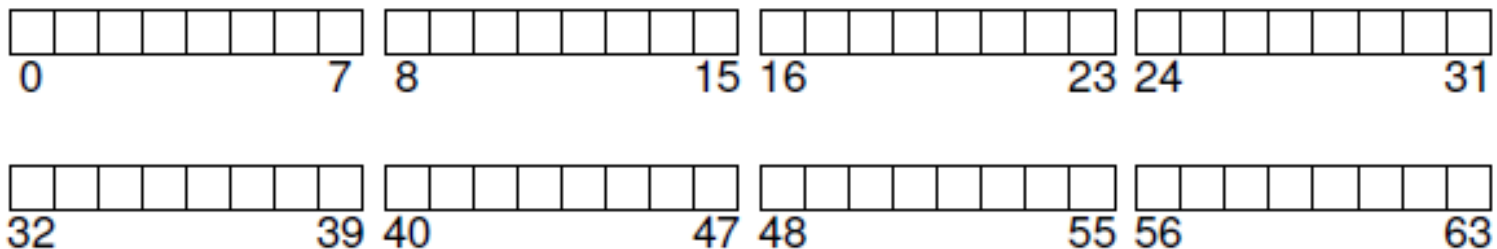
- Files and directories
 - CRUD operations
- Implementation
 - Data structures: how to organize blocks, e.g., into array/tree
 - Access methods: map calls from processes to operations on data structures

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

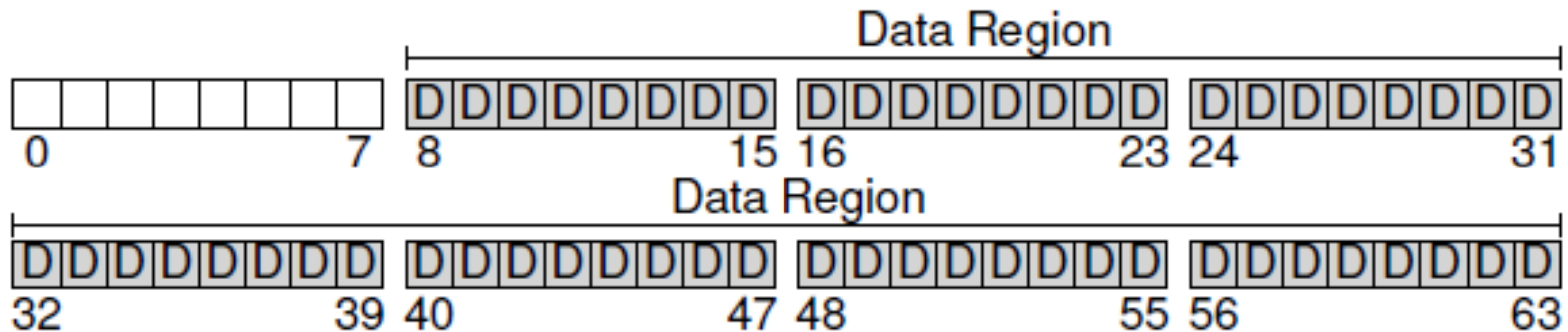
Blocks

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



Data region

- 56 blocks used to store data (files)
 - Blocks # 8 – 63

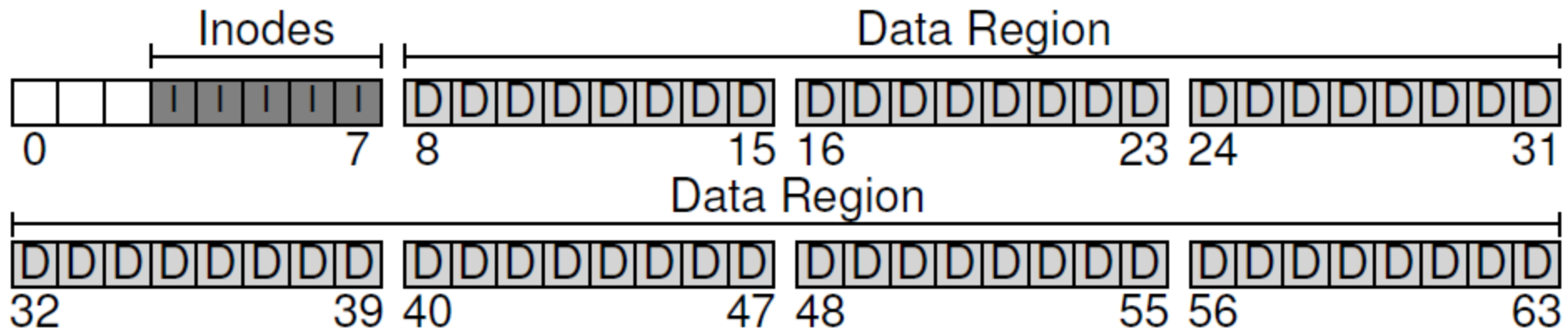


Metadata

- For each file, file system keeps track of
 - Location of that blocks that comprise the file
 - Size of the file
 - Owner and access rights
 - Access and modify times
 - Etc. (see struct stat)
- These metadata are stored in **inode**

inodes

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

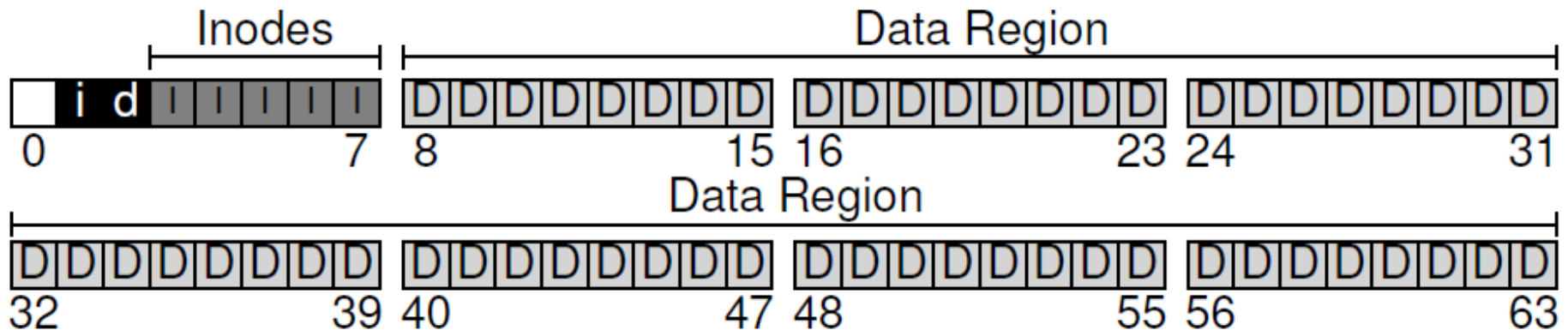
=> File system can store at most 80 files

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

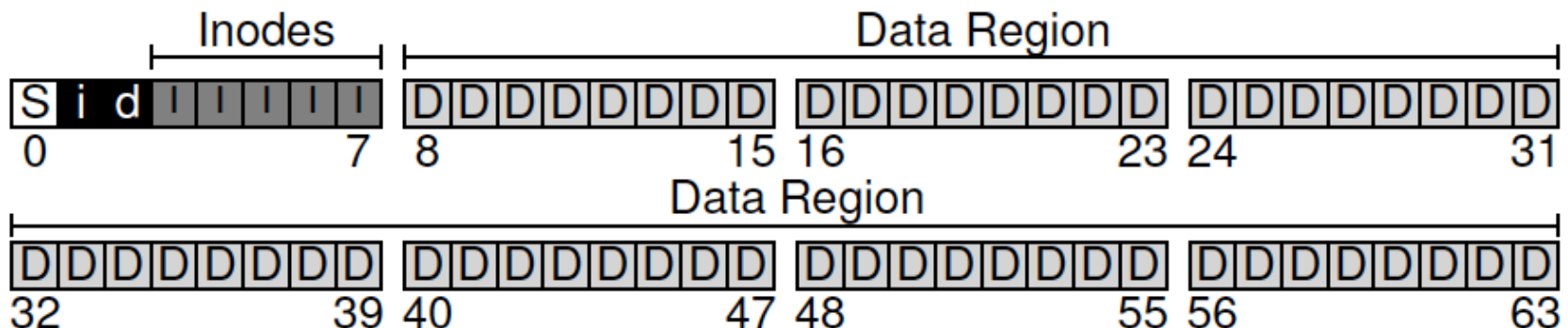
Bitmaps

- 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



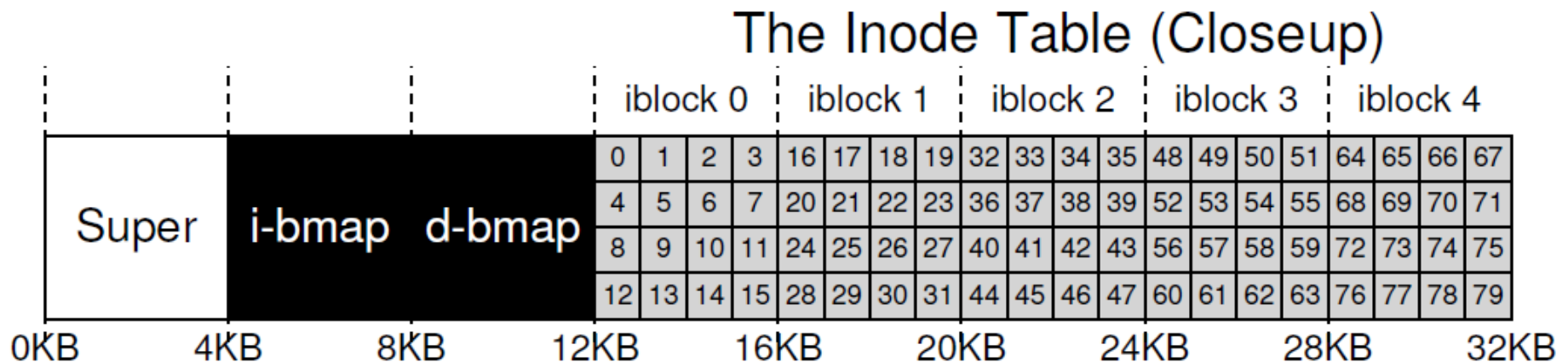
Superblock

- 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 file system
- Will be read first when file system is mounted



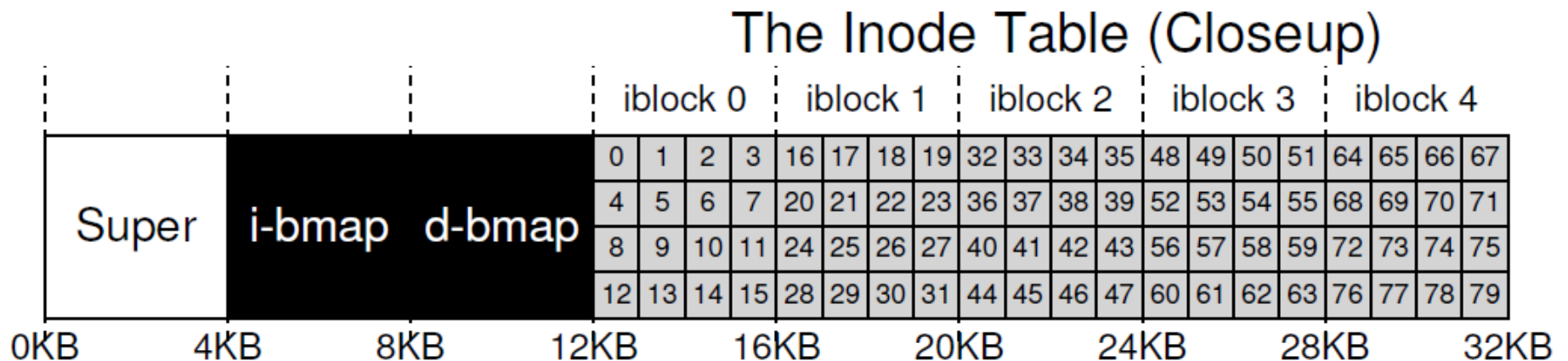
Inode (index node)

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



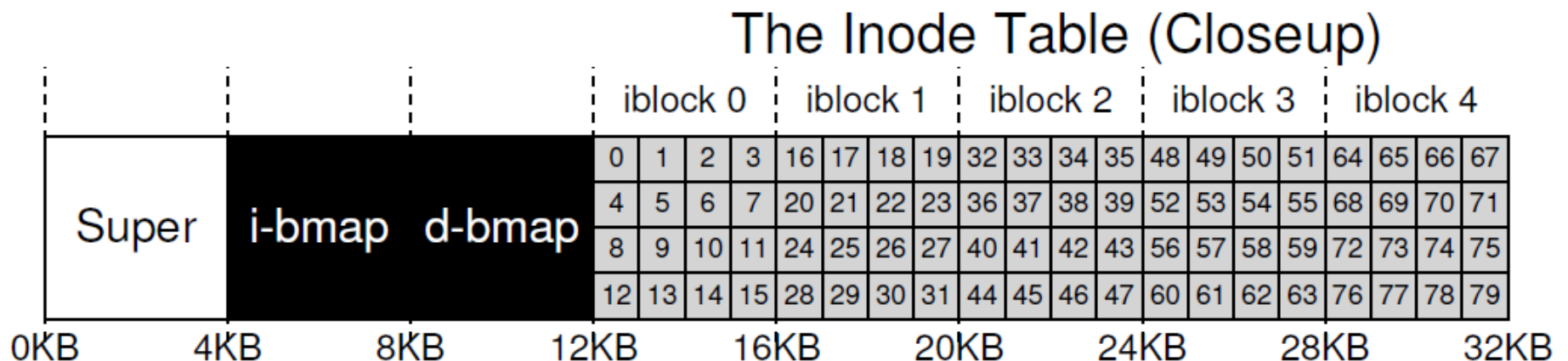
inumber => location

- 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
 - $\lfloor (\text{inodeStartAddress} + \text{inumber} * \text{inode size}) / \text{sector size} \rfloor$



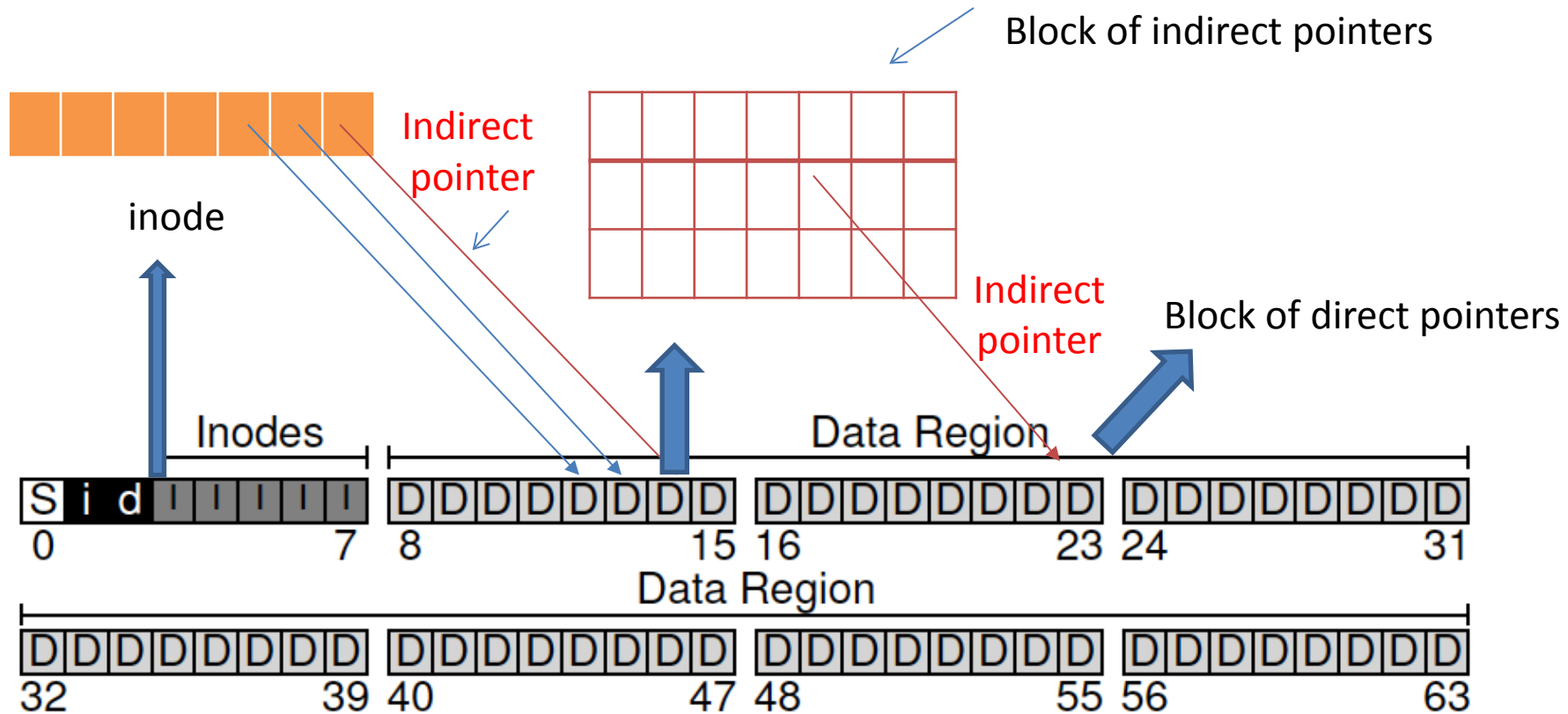
inode => location of data blocks

- A number of direct pointers
 - E.g., 8, each points to a block
 - Enough for $8 * 4K = 32K$ size of file
- Has a slot for indirect pointer
 - Point to a data block storing direct pointers
 - Assume 4 bytes for disk address, 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:
 - Its name, inumber, record length, string length

<code>inum</code>	<code> </code>	<code>reclen</code>	<code> </code>	<code>strlen</code>	<code> </code>	<code>name</code>
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 \geq string length
 - Due to entry reuse

<code>inum</code>	<code>reclen</code>	<code>strlen</code>	<code>name</code>
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 or a name is unlinked
 - (recall the "rm" and "unlink" commands)
 - (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

- As a (special) file with 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 calls from processes 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 so far)
 - Look for “foo” in / -> foo’s inumber
2. Read inode and content of /foo (4 reads)
 - Look for "bar" in /foo -> bar’s inumber
3. Read inode of /foo/bar (5 reads so far)
 - 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		
read()					write				read	
read()					read					
read()					write					
					read					read
					write					

Reading the file

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

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)`
 1. Consult bar's inode to locate its 1st block
 2. Read the block
 3. Update inode with newest file access time
 4. Update open-file table with new offset
 5. Continue steps 2, 3, 4 until done
 6. Deallocate file descriptor

Cost of read a block

- 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
open(bar)			read	read	read	read	read				
read()					read			read			
read()					write						
read()					read				read		
read()					read						
read()					write						

Writing a file

- Need to allocate an **inode** (if new file)
 - Read imap to find free inode
 - Write (update) imap
 - Write the inode
- Need to allocate a **block**
 - Read bmap to find free block
 - Write (actually update) bmap
 - Write the block

Writing a file

- Also need to update the directory
 - Update its data block: add inumber & name for new file
 - Update directory inode (book-keeping)
 - Might also need to grow directory e.g., by adding data block

Example: Open for write

- `int fd = open("/foo/bar", O_WRONLY)`
 - Assume bar is a new file under foo

Example: 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

Example: Open for write

3. Read imap, to find free inode for bar
4. Update imap, setting 1 for allocated inode
5. Write bar's inode

Example: Open for write

6. Update foo's content block
 - Adding an entry for bar
7. Update foo's inode
 - Update its access & modified time

Example: Open for write

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

	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						

Example: writing /foo/bar

1. Read inode of bar (by 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 modified time, add pointer to block

Example: writing /foo/bar

- 5 I/O's for each block write

	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:
 - 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