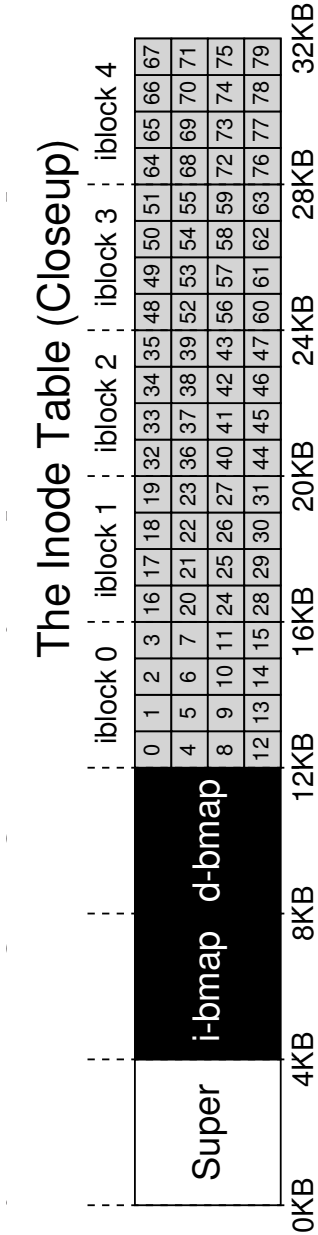


CSL 301

OPERATING SYSTEMS

Lecture 26

File System Implementation

Instructor
Dr. Dhiman Saha

How to implement a simple file system?

vsfs

Very Simple File System: pure software

- How can we build a simple file system?
- What structures are needed on the disk?
- What do they need to track?
- How are they accessed?

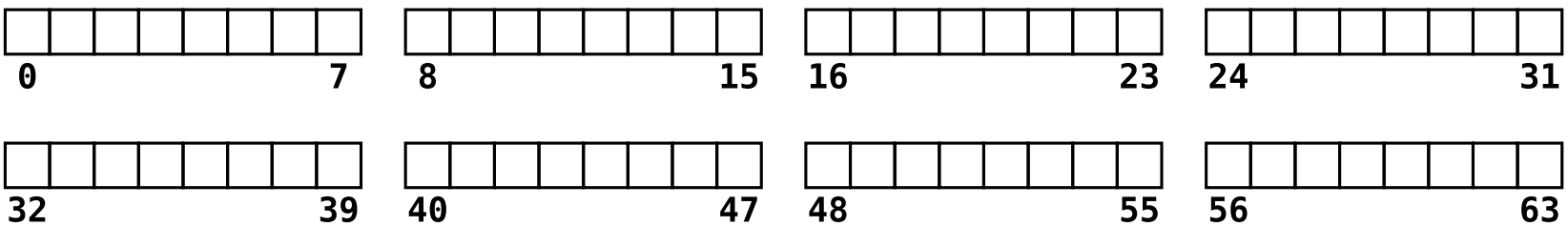
Data Structures

- What types of on-disk structures are utilized by the file system to organize its data and metadata?

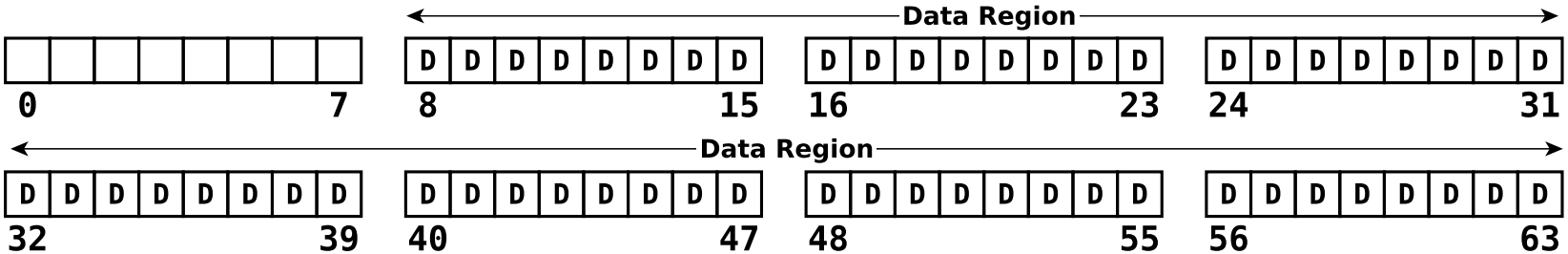
Access Methods

- How does it map the calls made by a process onto its structures?
- Which structures are read during the execution of a particular system call?
- Which are written?
- How efficiently are all of these steps performed?

Small Disk: 64 Blocks
Block Size = 4KB



Small Disk: 64 Blocks
Block Size = 4KB



FS has to track information about each file

- tracks things like which data blocks (in the data region) comprise a file,
- the size of the file,
- its owner and access rights,
- access and modify times, and
- other similar kinds of information.

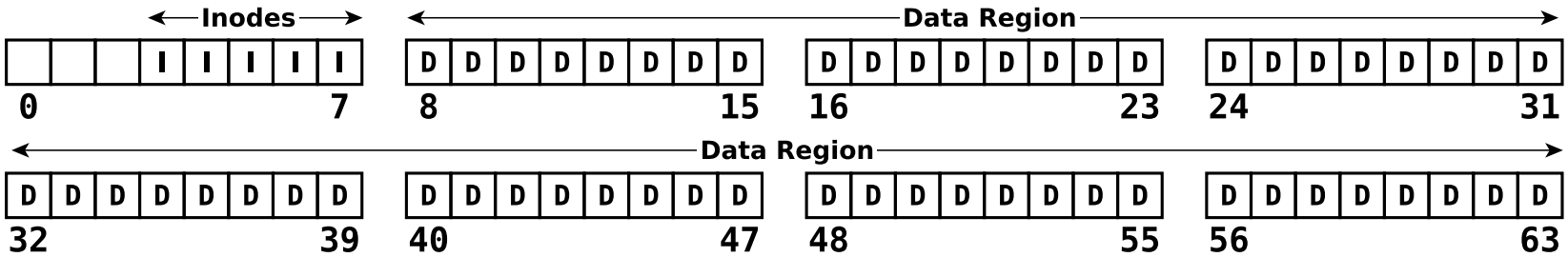
Structure

inode or index-node

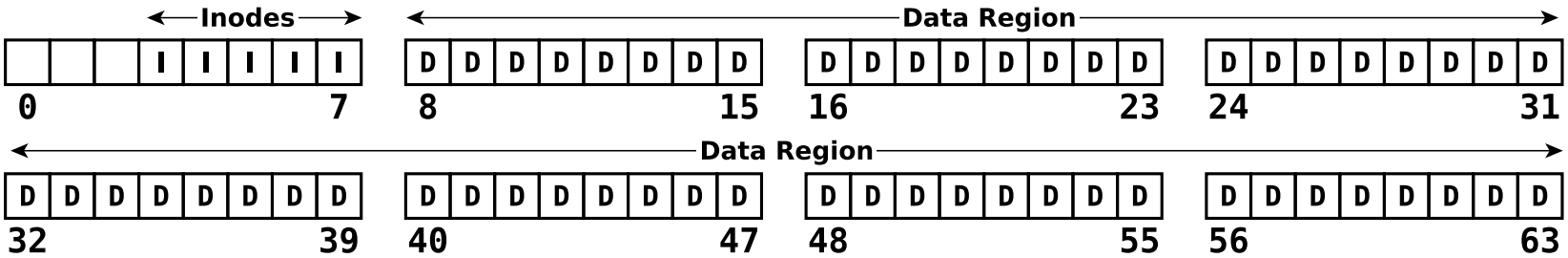
Inode-Table

Holds an array of on-disk inodes

Small Disk: 64 Blocks
Block Size = 4KB



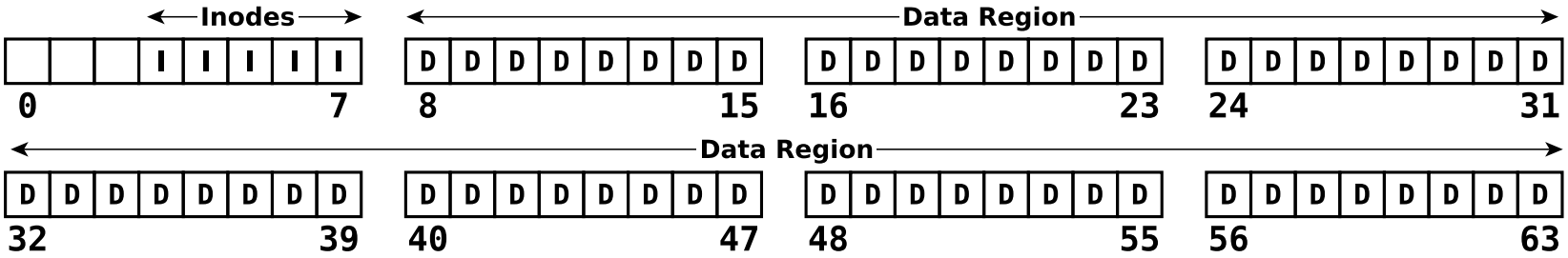
Small Disk: 64 Blocks
Block Size = 4KB



Question?

Assuming 256 bytes per inode what is the maximum number of files that we can have in this **vsfs** ?

Small Disk: 64 Blocks
Block Size = 4KB



- FS has
  - Data blocks (D)
  - Inodes (I)

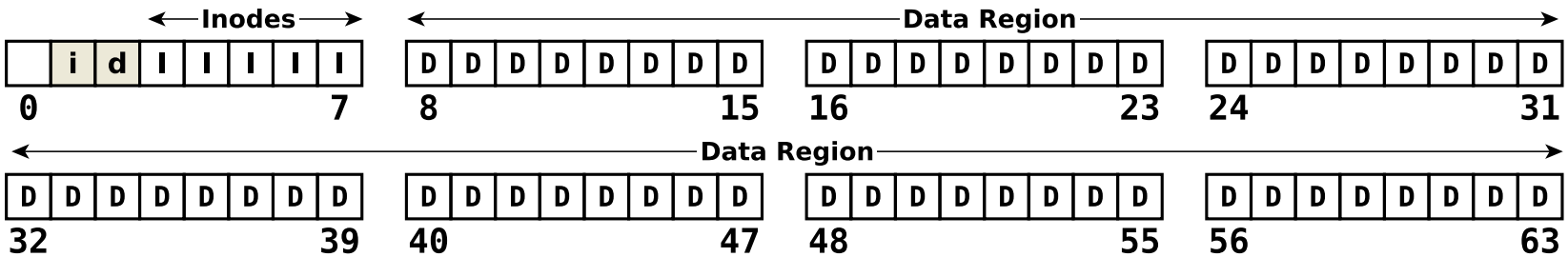
Next Allocation Structures

Some way to track whether inodes or data blocks are free or allocated : **bitmap**

Bitmap

Each bit is used to indicate whether the corresponding object/block is free (0) or in-use (1).

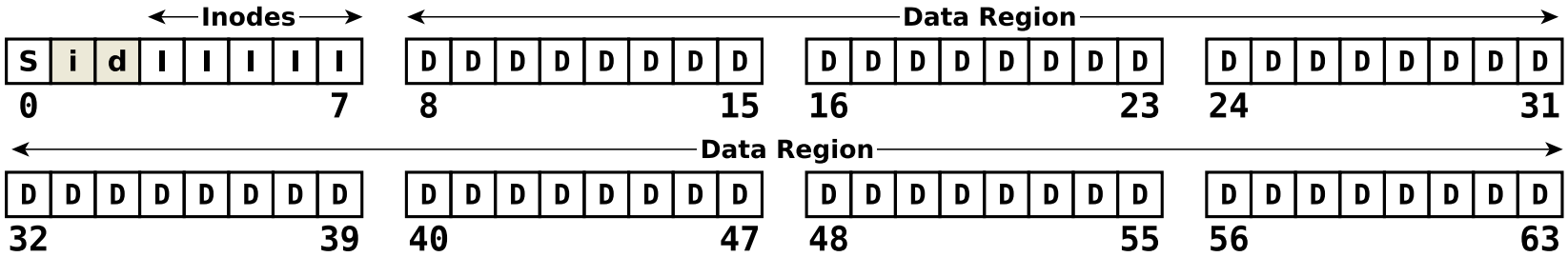
Small Disk: 64 Blocks
Block Size = 4KB



Two Bitmaps

- Data bitmap
- inode bitmap

Small Disk: 64 Blocks
Block Size = 4KB

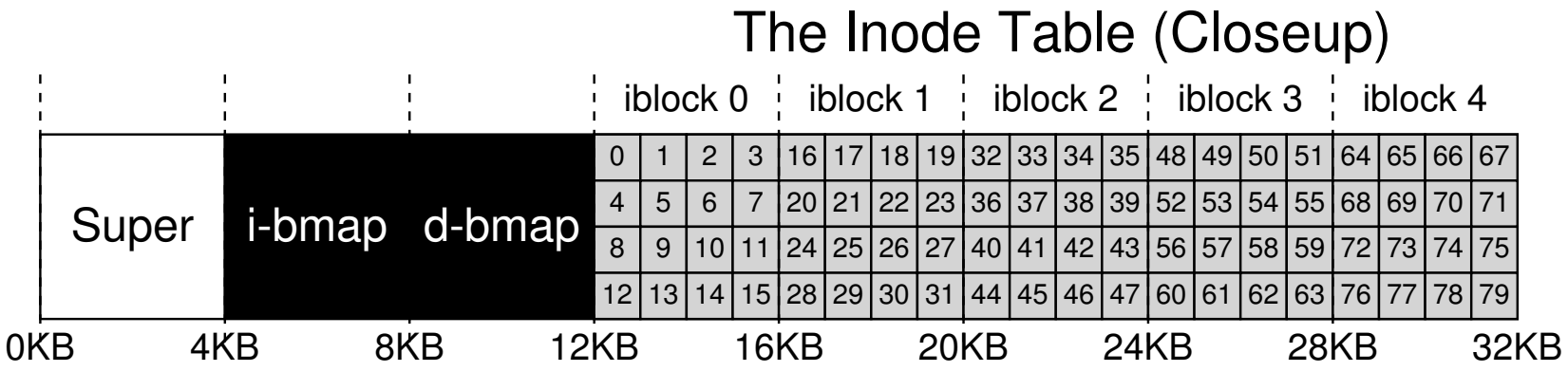


metadata about FS Superblock

- How many inodes and data blocks are in the FS?
- Where the inode table begins?
- The **magic-number**!

- When mounting a file system, the OS will read the **superblock** first

inode Table



Given inode# how to access inode.

Simplified Ext2 Inode

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

Is file data stored contiguously on disk?

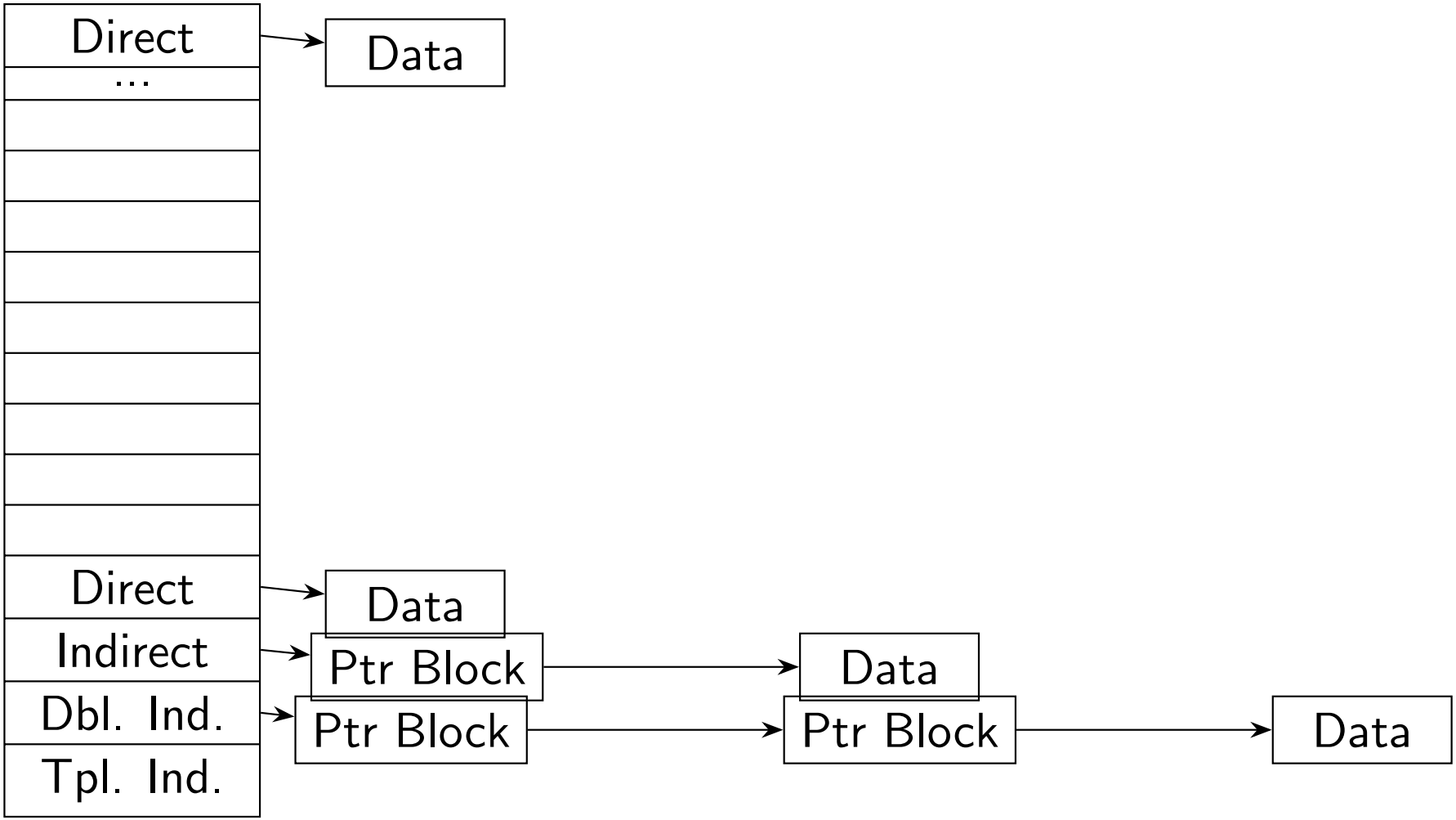
Need to track multiple block numbers of a file

- How does inode track disk block numbers?
  - **Direct pointers**: numbers of first few blocks are stored in inode itself (suffices for small files)
  - **Indirect block**: for larger files, inode stores number of indirect block, which has block numbers of file data
  - **Multi-level index**: Similarly, double and triple indirect blocks

Linux ext4 Alternative

- Extent based approach
- Recall idea of segments in VM

Multi-Level Index Example



- This is an imbalanced tree structure.
- It's optimized for small files, which are very common.
- With 12 direct pointers, small files can be accessed quickly.

File Size Calculation Example

Assumptions

- Block size = 4 KB
- Disk address (pointer) size = 4 Bytes
- Inode has 12 direct pointers, 1 indirect, 1 double indirect.

A single block can hold $4096/4 = 1024$ pointers.

- **Direct Pointers** (12):
  - Max size: 12×4 KB $= 48$ KB
- **Single Indirect Pointer**:
  - Adds 1024 pointers.
  - Max size: 1024×4 KB $= 4$ MB
- **Double Indirect Pointer**:
  - Adds 1024×1024 pointers.
  - Max size: $1024^2 \times 4$ KB $= 4$ GB
- **Combination (12 Direct + 1 Indirect + 1 Dbl. Indirect)**:
  - Max size: 48 KB + 4 MB + 4 GB ≈ 4 GB

Optimize the common case

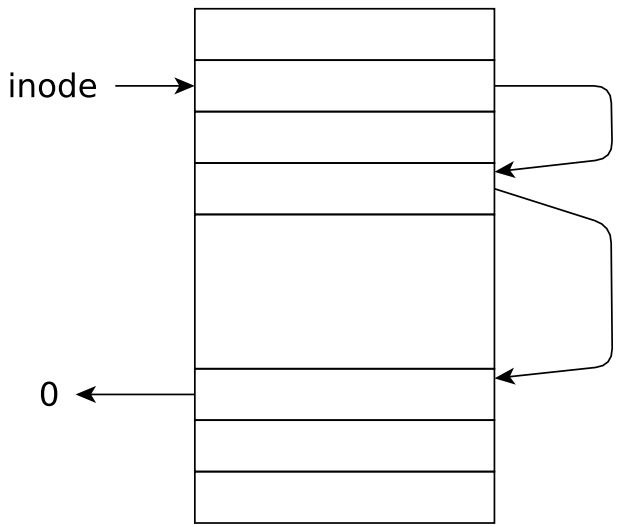
Most files are small	~2K is the most common size
Average file size is growing	Almost 200K is the average
Most bytes are stored in large files	A few big files use most of space
File systems contains lots of files	Almost 100K on average
File systems are roughly half full	Even as disks grow, file systems remain ~50% full
Directories are typically small	Many have few entries; most have 20 or fewer

File System Measurement Summary [Agarwal et. al]

Alternate way to track file blocks

Linked List based Approach

- FAT stores next block pointer for each block
  - FAT has one entry per disk block
  - Entry has number of next file block, or null (if last block)
  - **inode** stores pointer to first block



Question

How does random access take place in this setting?

```
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
```

- File systems treat directories as a special type of file
- Has an inode (in inode table)
- Has data blocks pointed to by the inode
- These data blocks live in the data block region of **vsfs**

Note

This has no effect on the on-disk structure we have assumed

Free space management

Many ways

- Bitmap
- Free-lists (Use the **superblock**)
- Other complex data structures
- Employ pre-allocation policy

Reading a File from Disk

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

```
open("/foo/bar", O_RDONLY)
```

Reading a File from Disk

FS will read from disk the inode of the root directory

Where?

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

```
open("/foo/bar", O_RDONLY)
```

Reading a File from Disk

FS will use on-disk pointers to read through the directory

Find entry for `foo`

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

```
open("/foo/bar", O_RDONLY)
```

Reading a File from Disk

FS will have found the inode number of `foo`

Recurse

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

```
open("/foo/bar", O_RDONLY)
```

Reading a File from Disk

FS will have found the inode number of `foo`

Recurse

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

```
open("/foo/bar", O_RDONLY)
```

Reading a File from Disk

final step of `open()` is to read bar's inode into memory — file descriptor allocated

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

```
open("/foo/bar", O_RDONLY)
```

Reading a File from Disk

Issue a read() system call to read from the file

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

```
open("/foo/bar", O_RDONLY)
```

Reading a File from Disk

First read at first block

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

```
open("/foo/bar", O_RDONLY)
```

Reading a File from Disk

Update last accessed time (write I/O), file offset

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

```
open("/foo/bar", O_RDONLY)
```

Reading a File from Disk

Continue from second block for next `read()`

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

`open("/foo/bar", O_RDONLY)`

Reading a File from Disk

Continue from second block for next `read()`

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

```
open("/foo/bar", O_RDONLY)
```

Reading a File from Disk

Continue from second block for next `read()`

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

`open("/foo/bar", O_RDONLY)`

Reading a File from Disk

Continue from second block for next `read()`

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

`open("/foo/bar", O_RDONLY)`

Reading a File from Disk

Continue from second block for next `read()`

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

`open("/foo/bar", O_RDONLY)`

Reading a File from Disk

Once file is closed FS deallocated the file descriptor

No disk I/Os take place

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

```
open("/foo/bar", O_RDONLY)
```

Points-to-Ponder

- The amount of I/O generated by the open is proportional to the length of the pathname
- Making this worse would be the presence of large directories. Why?

Open File Table

- Global open file table
  - Single entry for each file opened
  - Points to inode (read in-memory)
- Per-process open file table
  - List for files locally opened by process
  - Each points to global open file table entry
  - FD number is used as index in per process open file table

Homework

- What happens during a write/file creation?

Reduce cost of doing many I/O operations

- Caching
  - Static Vs Dynamic partitioning
- Buffering
  - Batch updation by delaying writes
  - Schedule subsequent I/Os
  - Avoiding writes by delay
- Exception: databases (direct I/O)
- Recall `fsync()`