Files and File Systems

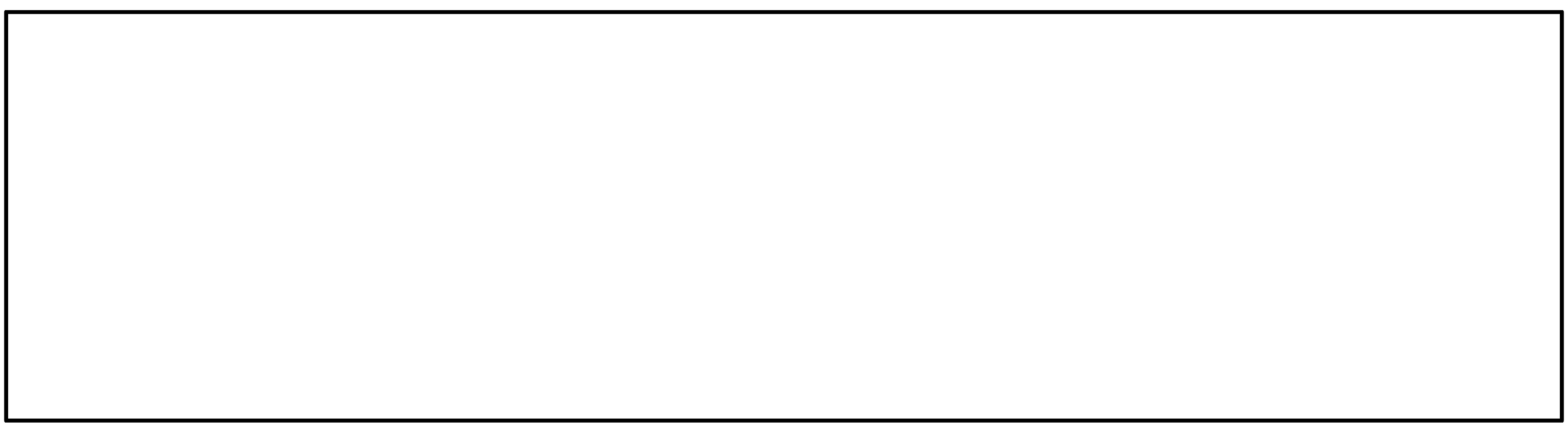
Recap

- Files are an abstraction that allow us to refer to persistent data on disk
- Directories (folders) provide a logical organization of files
    - A directory can be implemented a special type of file that contains a list of directory entries

File System Implementation

- How do file systems use the disk to store files?

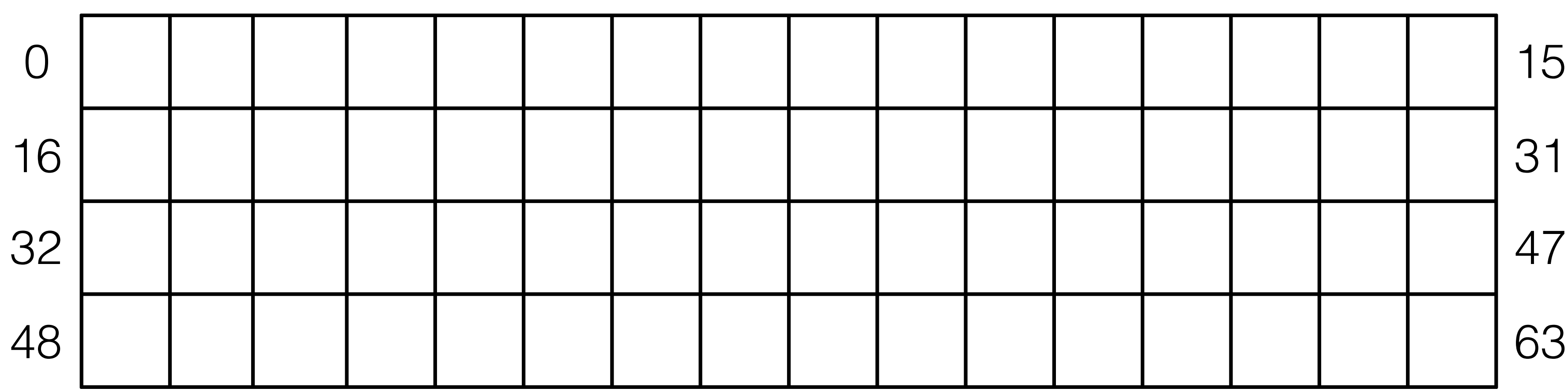
A Raw Unformatted Disk (256 KiB)



File System Implementation

- How do file systems use the disk to store files?
  - File systems define a block size (e.g., 4KB)
    - Disk space is allocated in granularity of blocks

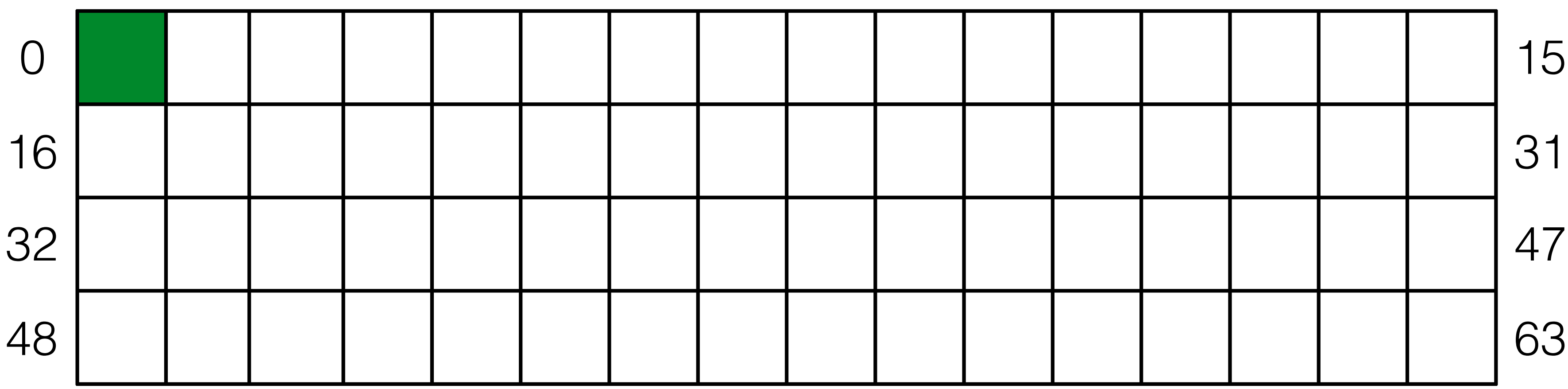
(e.g. Block size: 4KB, Number of blocks: 64)



Superblock

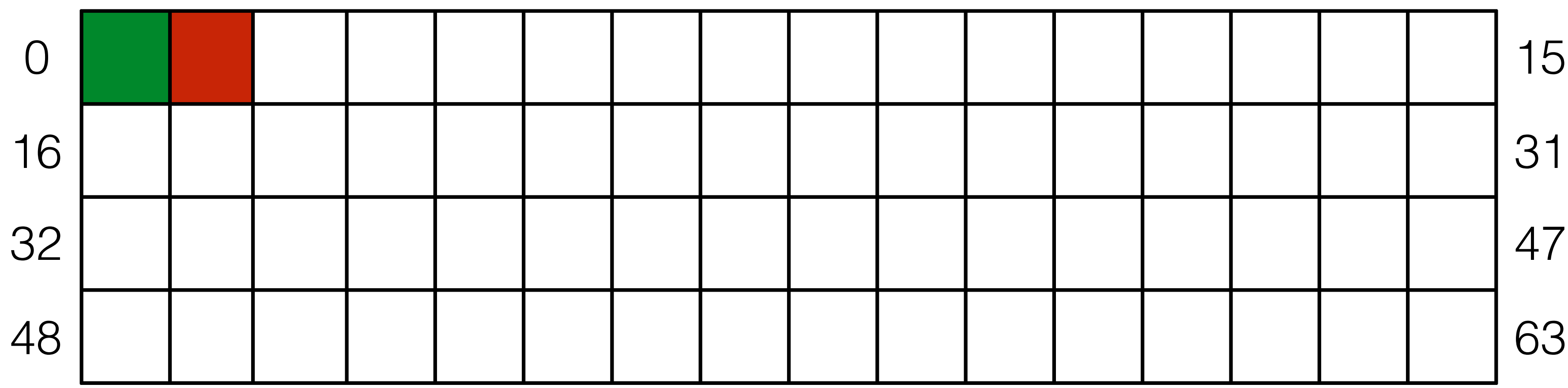
- What do we need to know to connect the disk to a computer?

- A "superblock" determines location of root directory
  - Always at a well-known disk location
  - Often replicated across disk for reliability
  - Includes other metadata about the file system



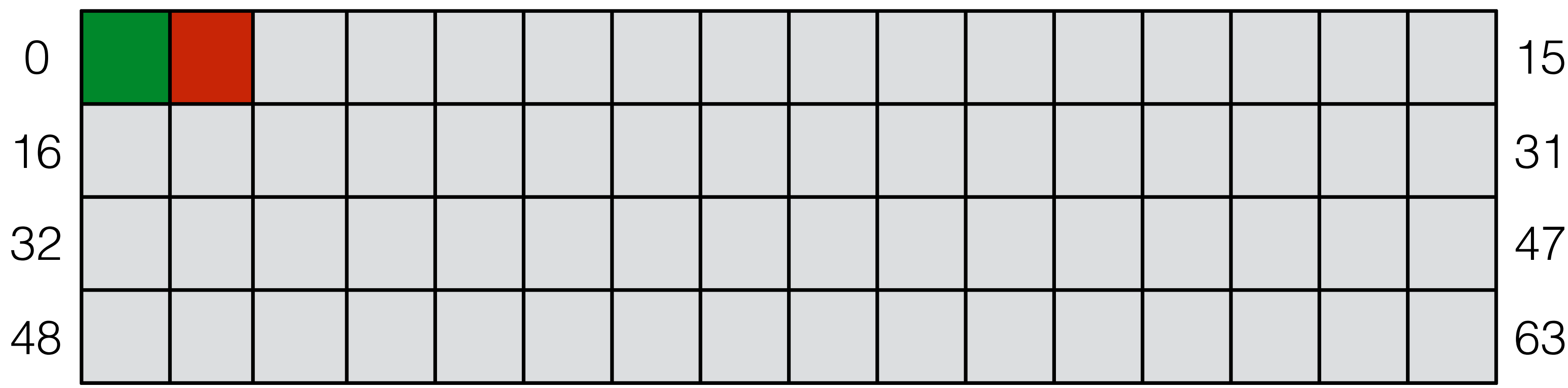
Free map

- A free map determines which blocks are free
  - Usually a bitmap, one bit per block on the disk
  - Stored on disk, cached in memory for performance



Data blocks

- Remaining blocks used to store files and directories
  - There are many ways to do this

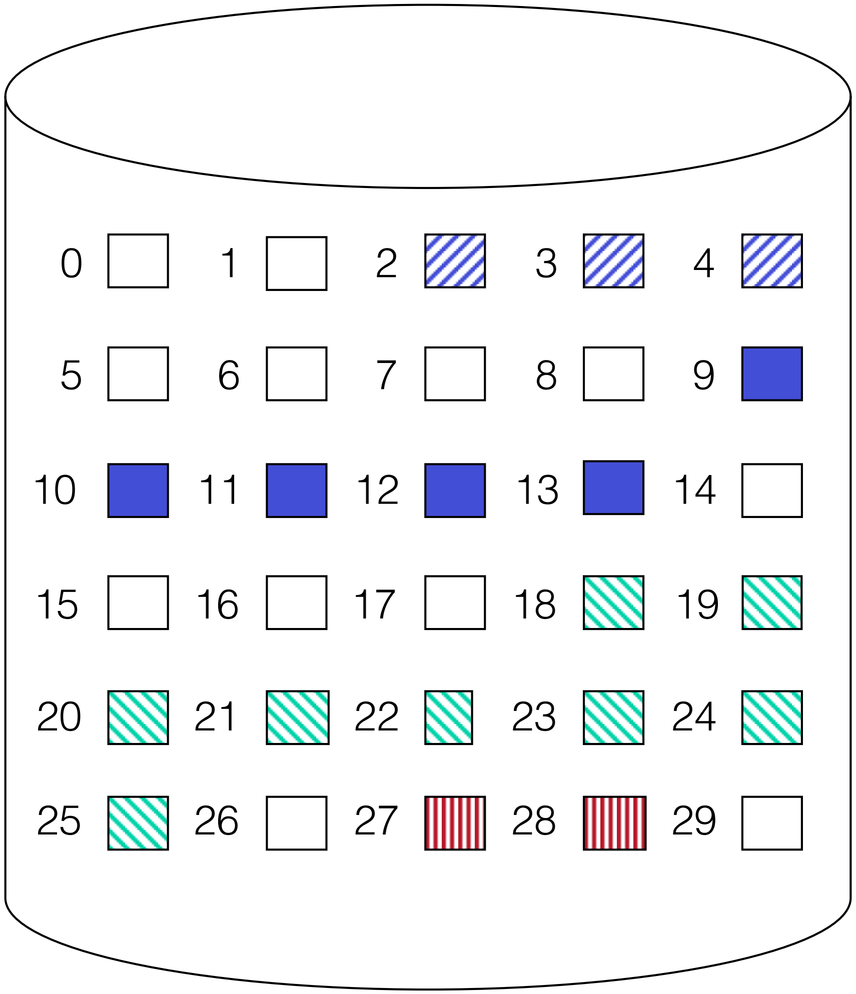


Disk Layout Strategies

- Files often span multiple disk blocks

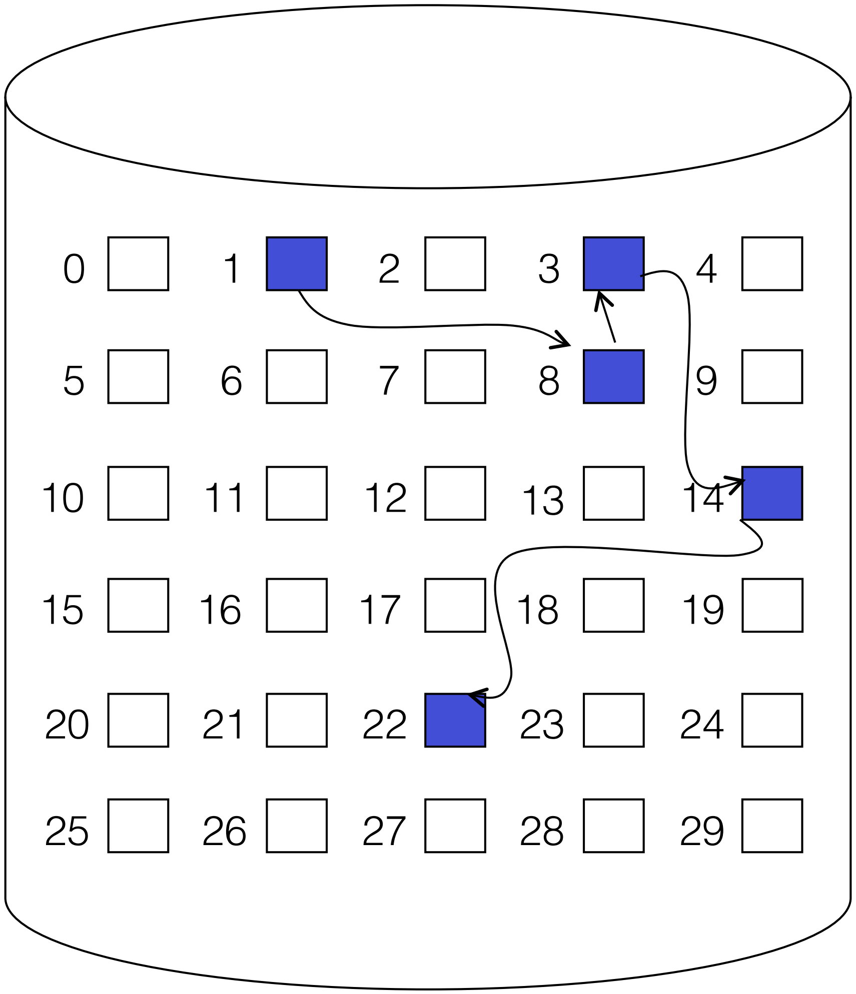
- How do you find all of the blocks for a file?
  1. Contiguous allocation
     - All blocks of file are located together on disk
  2. Linked, or chained, structure
     - Each block points to the next, directory points to the first
  3. Indexed structure (kind of like address translation)
     - An "index block" contains pointers to many other blocks
     - May require multiple, linked index blocks

Contiguous Allocation



File Name	Start Blk	Length
File A	2	3
File B	9	5
File C	18	8
File D	27	2

Linked Allocation



File Name	Start Blk	Last Blk
...	...	...
File B	1	22
...	...	...

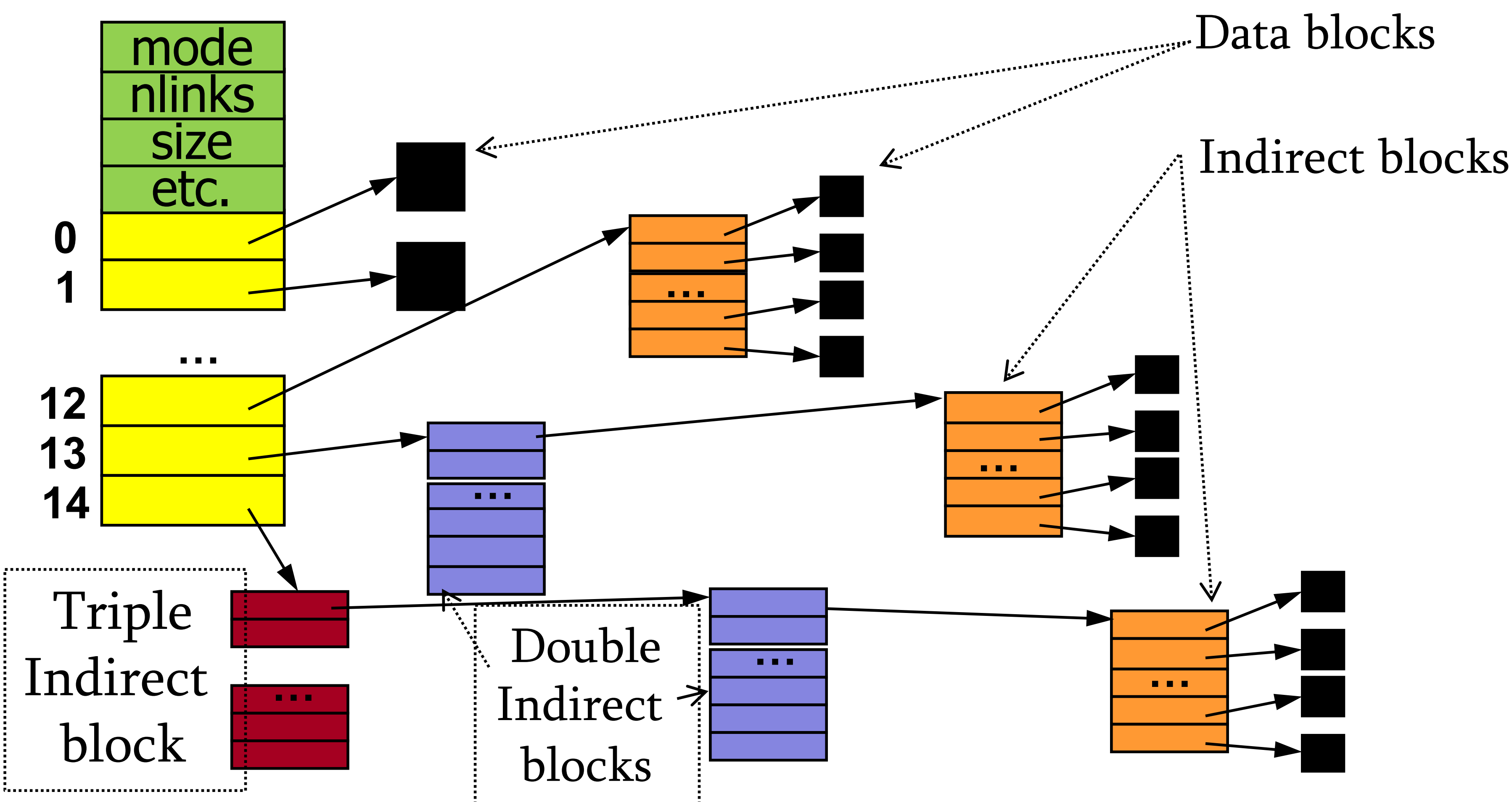
The FAT file system uses linked allocation but the links are in the File Allocation Table, not in the blocks themselves

Unix Inodes

- Unix inodes implement an indexed structure for files
- All file metadata is stored in an inode
    - Directory entries map file names to inodes
- Each inode contains 15 block pointers
    - block[0]-block[11] are direct block pointers
        - (Disk addresses of first 12 data blocks in file)
    - block[12] is a single indirect block pointer
        - Address of block containing addresses of data blocks
    - block[13] is a double indirect block pointer
        - Address of block containing addresses of single indirect blocks
    - block[14] is a triple indirect block pointer

Example UNIX Inode

- Ext2 Linux file system inodes are 128 bytes



Data block allocation

	Advantages	Disadvantages
Contiguous	• Sequential access fast • Allocation fast • Deallocation fast • Small amount of metadata	• External fragmentation • Need compaction • Need to move whole files around • Inflexible
Linked	• Sequential access easy • Disk blocks can be anywhere • No external fragmentation	• Direct access is expensive • If a data block is corrupted could lose rest of file.
Indexed	• Handles random access well • Small files: quick sequential and random access • No external fragmentation	• Limits file size • Cost of access bytes near the end of large files grows

Implementation of a Very Simple File System (VSFS)

Existing File Systems

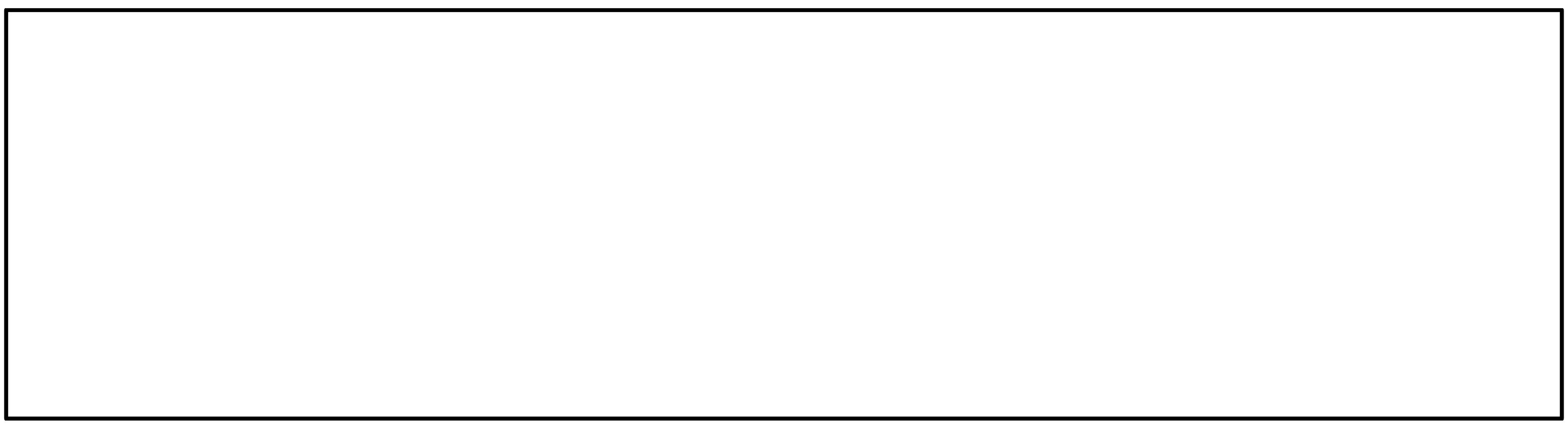
- Many file system implementations, literally from AFS to ZFS
- Check out https://en.wikipedia.org/wiki/List_of_file_systems
- Most well-known:
  - Windows: FAT32, NTFS
  - MAC OS X: HFS+
  - BSD, Solaris: UFS, ZFS
  - Linux: ext2 (see A4 too!), ext3, ext4, ReiserFS, XFS, JFS, btrfs, zfs, etc.
- We'll have a look at a very simple file system (VSFS)
- See readings as well!

The main idea

- We need to create a file system for an unformatted disk
- We need to create some structure in it, so that things (data) will be easy to find and organize
- Key questions
  - Where do we store file data and metadata structures (inodes)?
  - How do we keep track of data allocations?
  - How do we locate file data and metadata?
  - What are the limitations (max file size, etc.)?

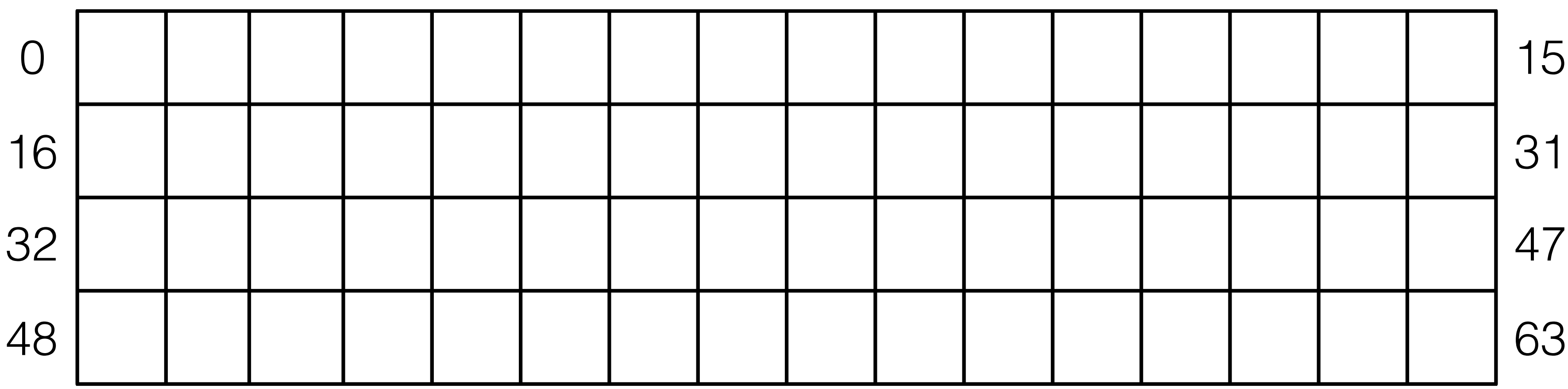
An Unformatted Raw Disk

- Total size = 256 KiB



Overall Organization

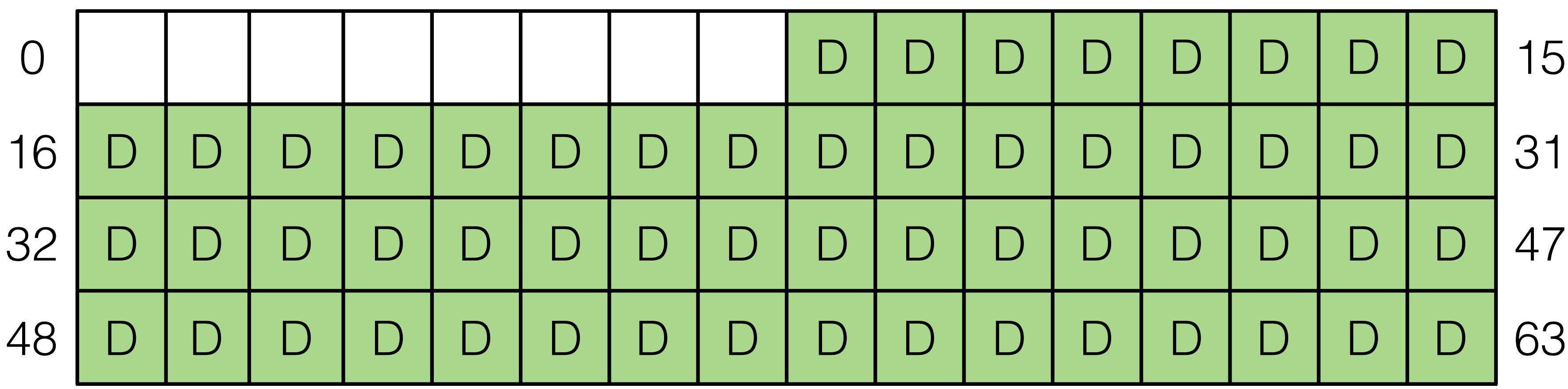
- The whole disk is divided into fixed-sized blocks.
  - Block size: 4KB
  - Number of blocks: 64
  - Total size: 256KB



Data Region

- Most of the disk should be used to actually store user data, while leaving a little space for storing other things like metadata.

- In VSFS, we reserve the last 56 blocks as data region.



How many blocks can a 4KB data bitmap keep track of?

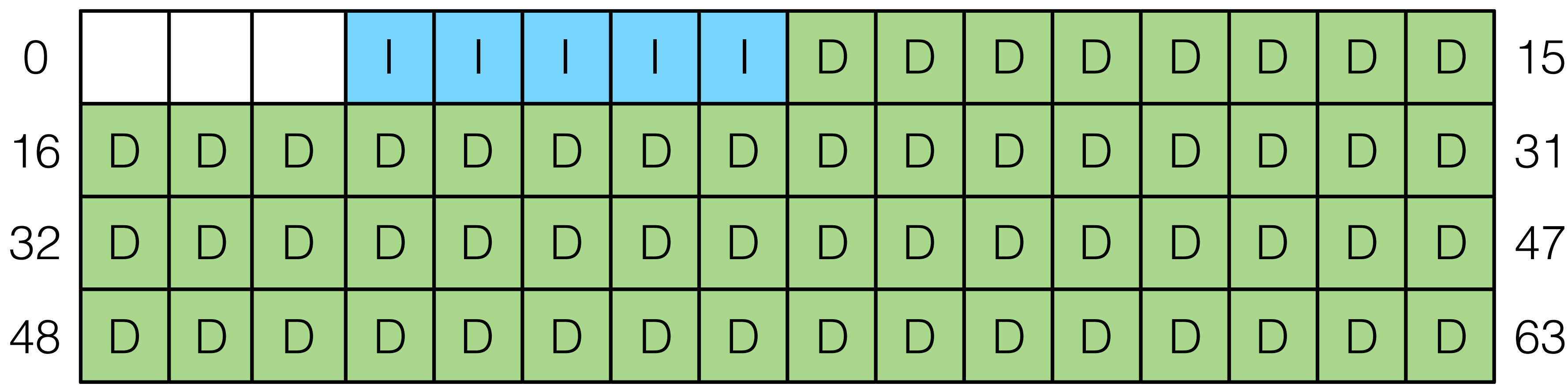
4KB = 32K bits, can keep track of 32K blocks, so it is overkill in this VSFS.

Metadata: inode table

- FS needs to track information about each file.
- In VSFS, we keep the info of each file in a struct called inode. And we use 5 blocks for storing all the inodes.

Size	Name	What is this inode field for?
2	mode	read/write/execute
2	uid	file owner
4	size	number of bytes in file
4	atime	last access time
4	ctime	file creation time
4	time	last modified time
4	dtime	inode deleted time
2	gid	group id
2	links_count	number of hard links to this file
4	blocks	number of blocks allocated to this file
4	flags	how should ext2 use this inode?
4	osd1	OS dependent field
60	block	a set of 15 disk pointers

ext2 inode with size of 128B



How many files can this VSFS hold at most?

Maximum number of inodes it can hold: 5 * 4KB / 128B = 160 => can store at most 160 files.

Allocation Structures

- Keep track of which blocks are being used and which ones are free
- We use a data structure (a bitmap) for this purpose
  - Each bit indicates if one block is free (0) or in-use (1)
- A bitmap for the **data region** and a bitmap for the **inode region**
  - Reserve one block for each bitmap.

0		IB	DB	I	I	I	I	I	D	D	D	D	D	D	D	D	15
16	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	31
32	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	47
48	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	63

How many blocks can a 4KB data bitmap keep track of?

4KB = 32K bits, can keep track of 32K blocks, so it is overkill in this VSFS.

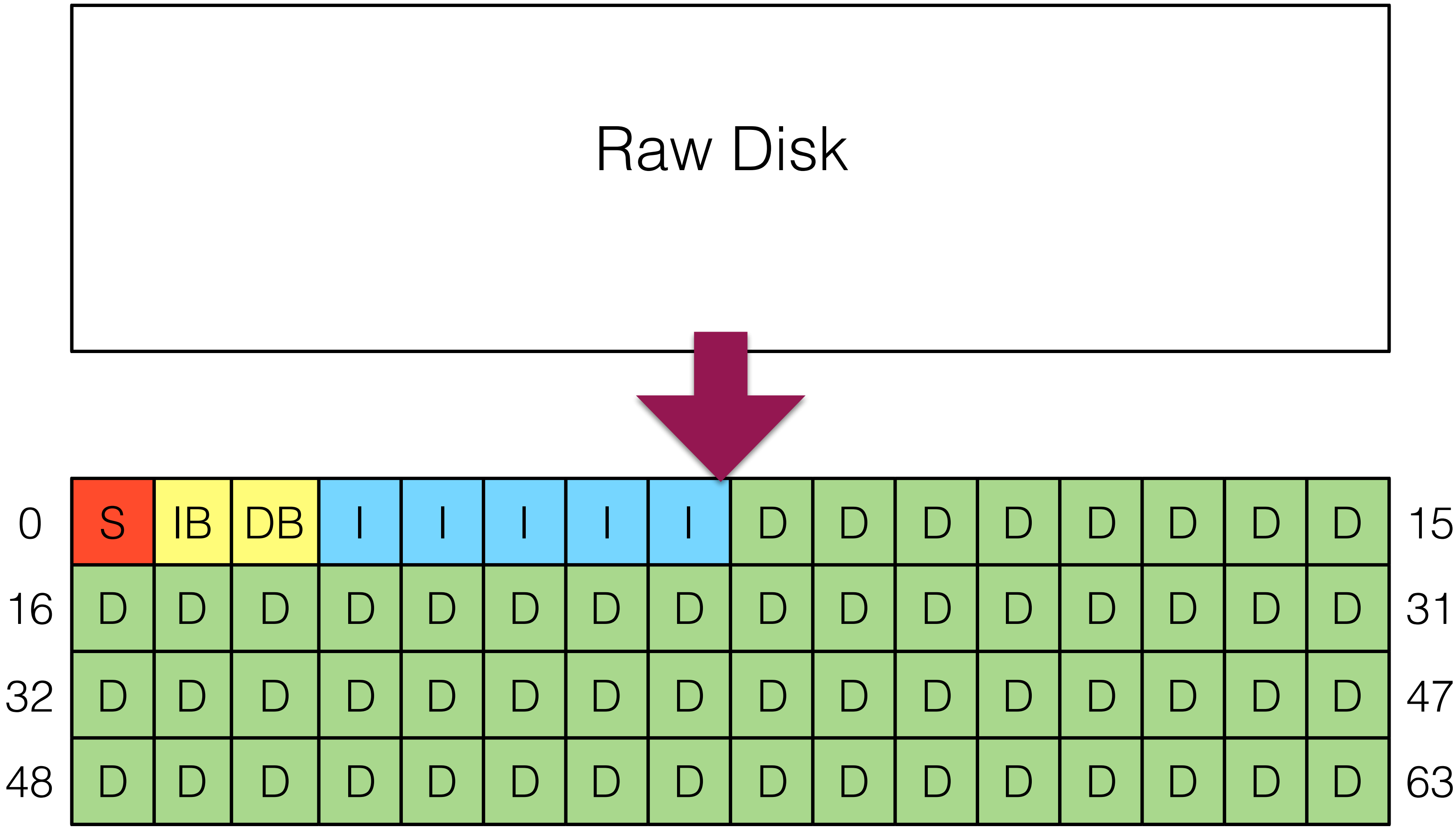
Superblock

- Superblock contains information about this particular file system:
  - what type of file system it is ("VSFS" indicated by a magic number)
  - how many inodes and data blocks are there (160 and 56)
  - where the inode table begins (block 3), etc.

0	S	IB	DB	I	I	I	I	I	D	D	D	D	D	D	D	D	15
16	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	31
32	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	47
48	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	63

When mounting a file system, the OS first reads the superblock, identifies its type and other parameters, then attaches the volume to the file system tree with proper settings.

Formatting disk into VSFS, done!

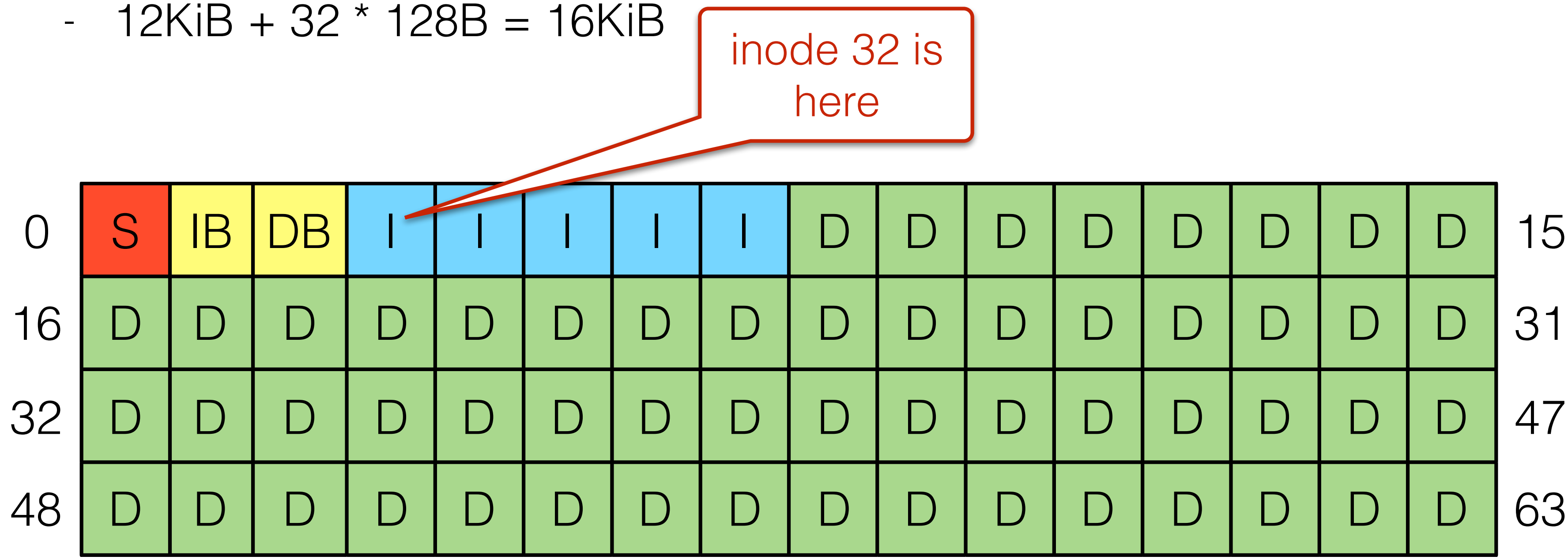


When mounting a file system, the OS first reads the superblock, identifies its type and other parameters, then attaches the volume to the file system tree with proper settings.

Example: Read a file with inode number 32

- From the superblock, we know
  - inode table begins at Block 3, i.e., 12KiB
  - inode size is 128B

- Calculate the address of inode 32
  - 12KiB + 32 * 128B = 16KiB



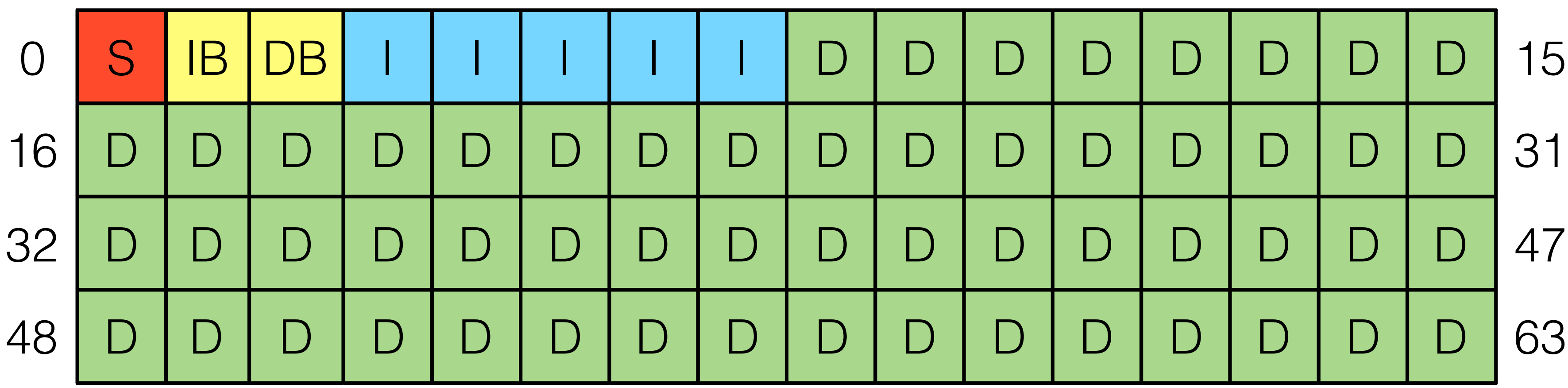
So we have the inode, but which blocks have the data?

From inode to data

- Say the inode contains an array of 15 direct pointers that point to 15 data blocks that belong to the file.
- Maximum file size supported:
  - 15 * 4KiB = 60KiB
- If we need a file larger than 60KiB, we need to do something more sophisticated.

Size	Name	What is this inode field for?
2	mode	read/write/execute
2	uid	file owner
4	size	number of bytes in file
4	atime	last access time
4	ctime	file creation time
4	time	last modified time
4	dtime	inode deleted time
2	gid	group id
2	links_count	number of hard links to this file
4	blocks	number of blocks allocated to this file
4	flags	how should ext2 use this inode?
4	osd1	OS dependent field
60	**block**	**a set of 15 disk pointers**

ext2 inode with size of 128B



Multi-Level Index with Indirect Pointers

- Direct pointers to disk blocks do not support large files
- Idea: indirect pointer
  - Instead of pointing to a block of user data, it points to a block that contains more pointers
  - From the 15 pointers we have in an inode, use the first 14 as direct pointers and 15th as an indirect pointer
- How big a file can we support now?
  - 14 direct pointers in total: 14 data blocks
  - Indirect pointer points to a block (4KiB) which can hold 1Ki pointers => 1K data blocks in addition
  - Total size supported: 4K * (14 + 1K) = 4152KiB

What if I want even Bigger?

Double Indirect Pointer!

- A double indirect pointer points to a block full of indirect pointers which point to blocks of direct pointers.
  - E.g., for the 15 pointers, we use the first 13 as direct pointers, the 14th as an indirect pointer, and the 15th as a double indirect pointer.

- How big a file do we support now?
  - From direct pointers: 13 data blocks
  - From indirect pointers: 1024 data blocks
  - From double indirect pointers: 1024 * 1024 = 1Mi data blocks

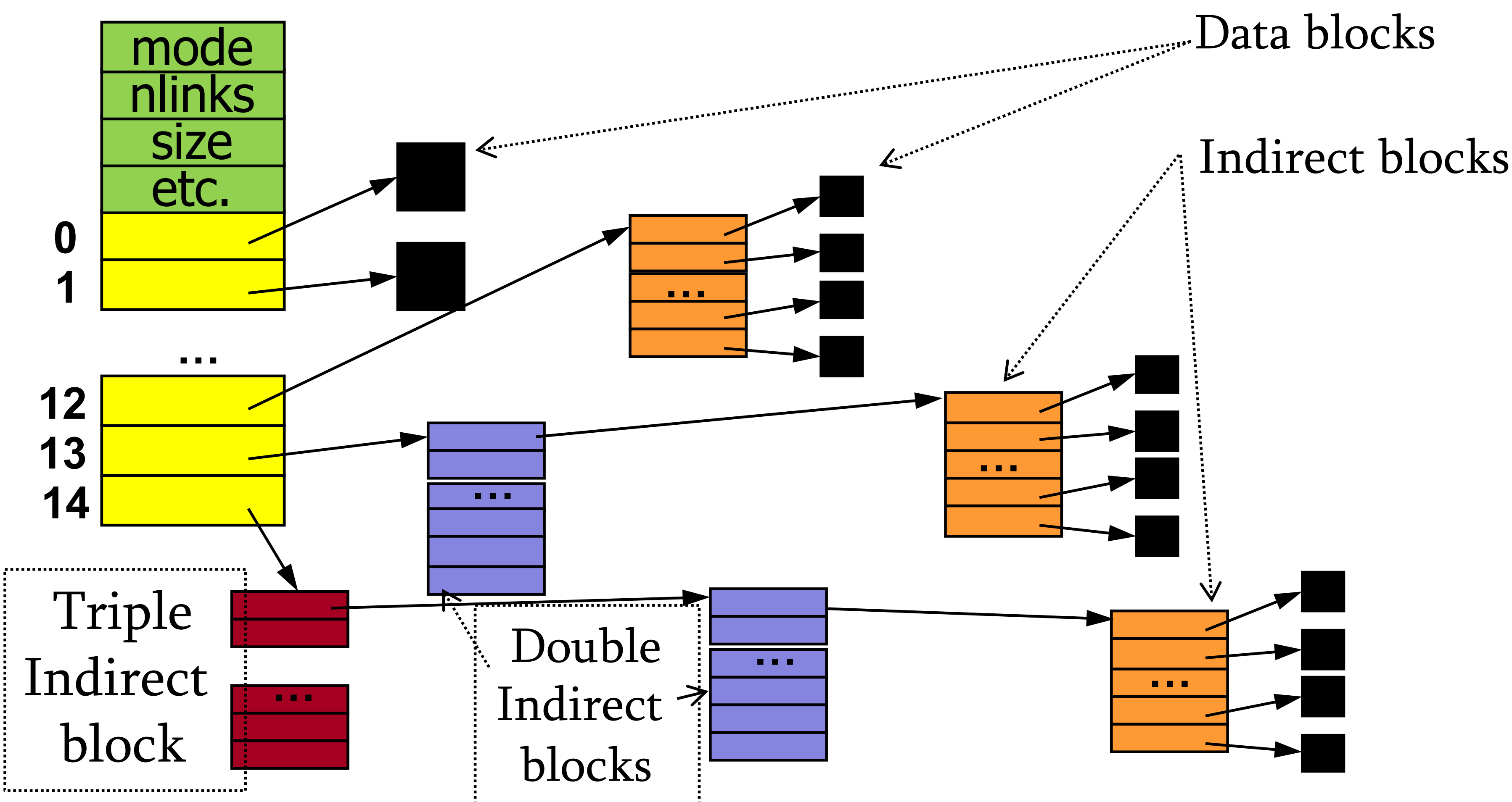
- Total size = 4Ki * (13 + 1024 + 1024*1024) ≈ 4.004GiB

Still not big enough? Use a Triple Indirect Pointer

A tree-view of multi-level pointers

Now how big can a file be?

$$4\text{KiB} * (12 + 1024 + 1024^2 + 1024^3) \approx 4\ \text{TiB}$$



Why an imbalanced tree?

- Most files are small (~2KiB)
- Files are usually accessed sequentially
- Directories are typically small (20 or fewer entries)

- Design based on evidence. For example, see "*A Five-Year Study of File System Metadata*" (link on the course web site)

Another Approach: extent-based

- An extent == a disk pointer plus a length (in # of blocks), i.e., it allocates a few blocks in a row.

- Instead of requiring a pointer to every block of a file, we just need a pointer to every several blocks (every extent).

- Disadvantage: Less flexible than the pointer-based approach. (External fragmentation?)

- Advantages: Uses smaller amount of metadata per file, and file allocation is more compact.

- Adopted by ext4, HFS+, NTFS, XFS.

Yet another approach: Link-Based

- Instead of pointers to all blocks, the inode just has one pointer to the first data block of the file, then the first block points to the second block, etc.

- Works poorly if we want to access the last block of a big file.

- Use an in-memory **File Allocation Table**, indexed by address of data block
  - Faster in finding a block.

- FAT file system, used by Windows before NTFS.

- Focus on inode-based FSs in next lectures...

Summary

- Inodes
  - Data structure representing a FS object (file, dir, etc.)
  - Attributes, disk block locations
  - No file name, just metadata!
- Directory
  - List of (name, inode) mappings
  - Each directory entry: a file, other directory, link, itself (.), parent dir (..), etc.