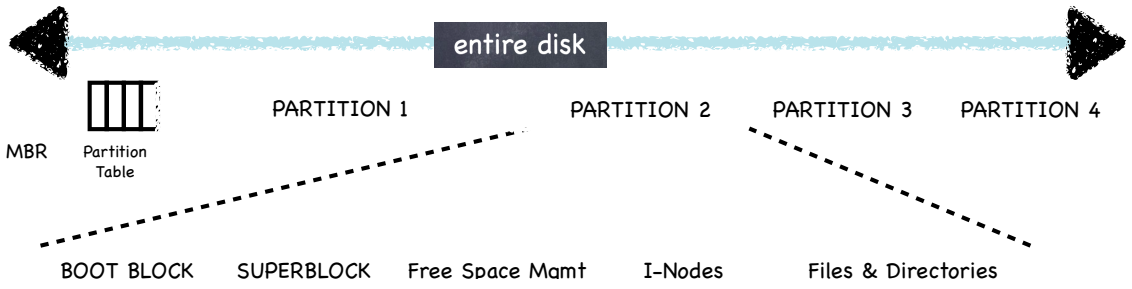
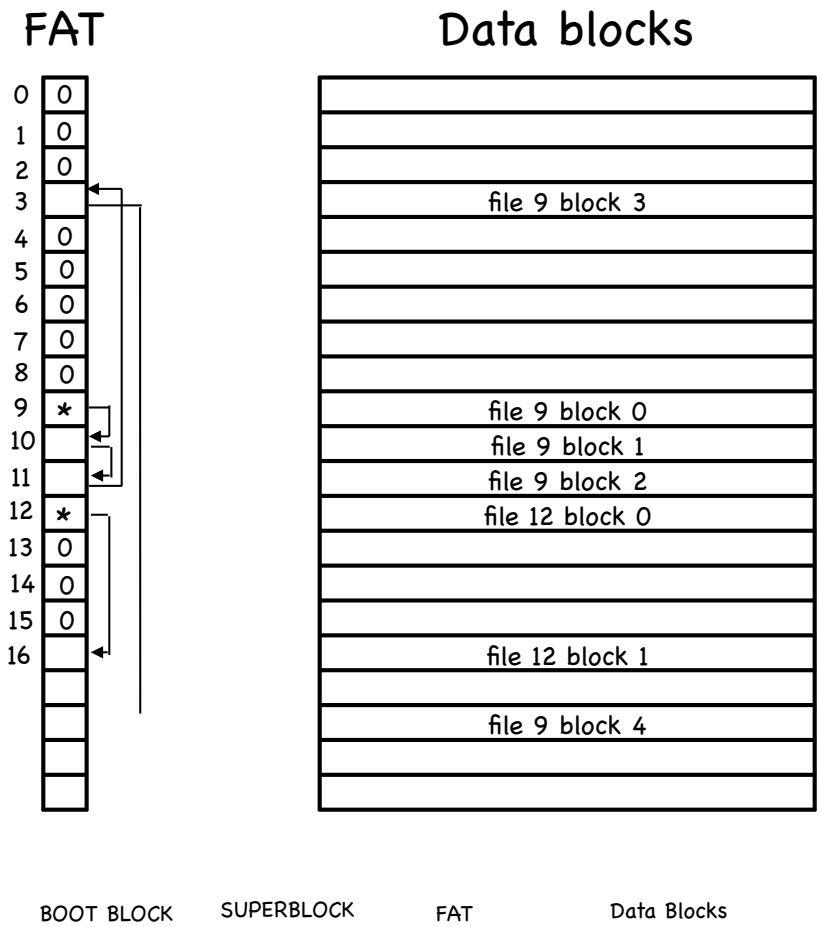
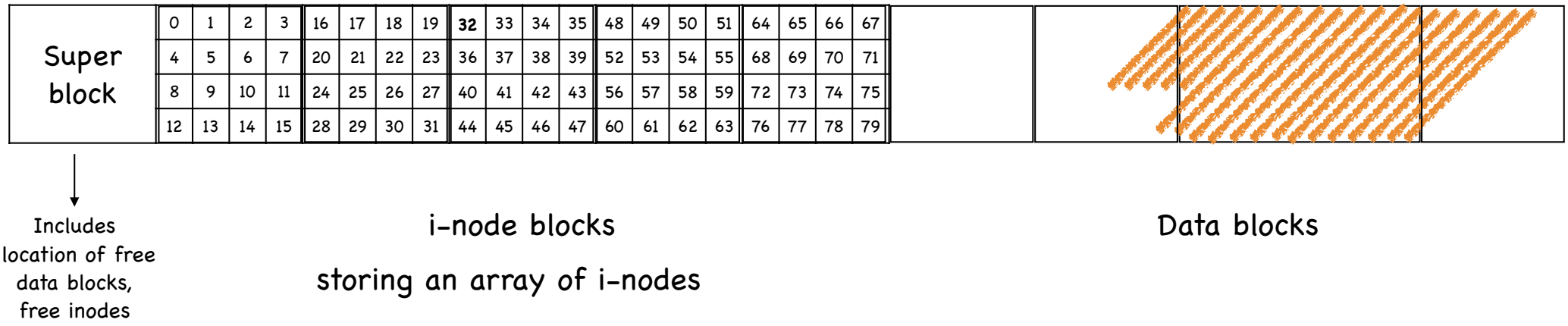
File System Layout





Tree-based Multi-level Index

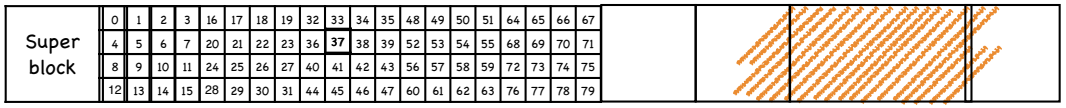
- UFS (Unix File System) (Ken Thompson, 1969)
- 4.2 BSD FFS (Fast File System) (McKusick, Joy, Leffler, Fabry, 1983)



Multilevel index

Inode Array

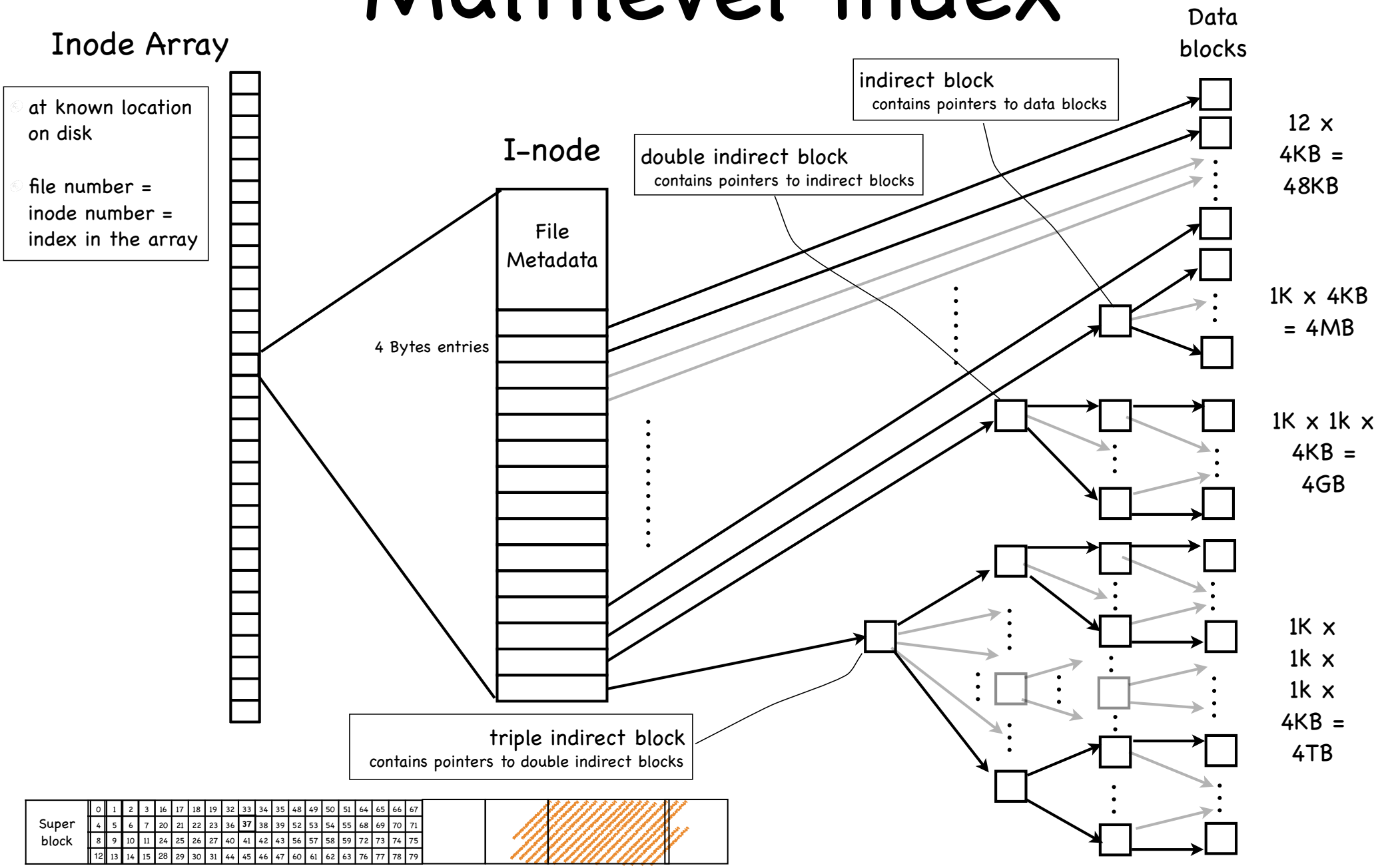
- at known location on disk
- file number = inode number = index in the array



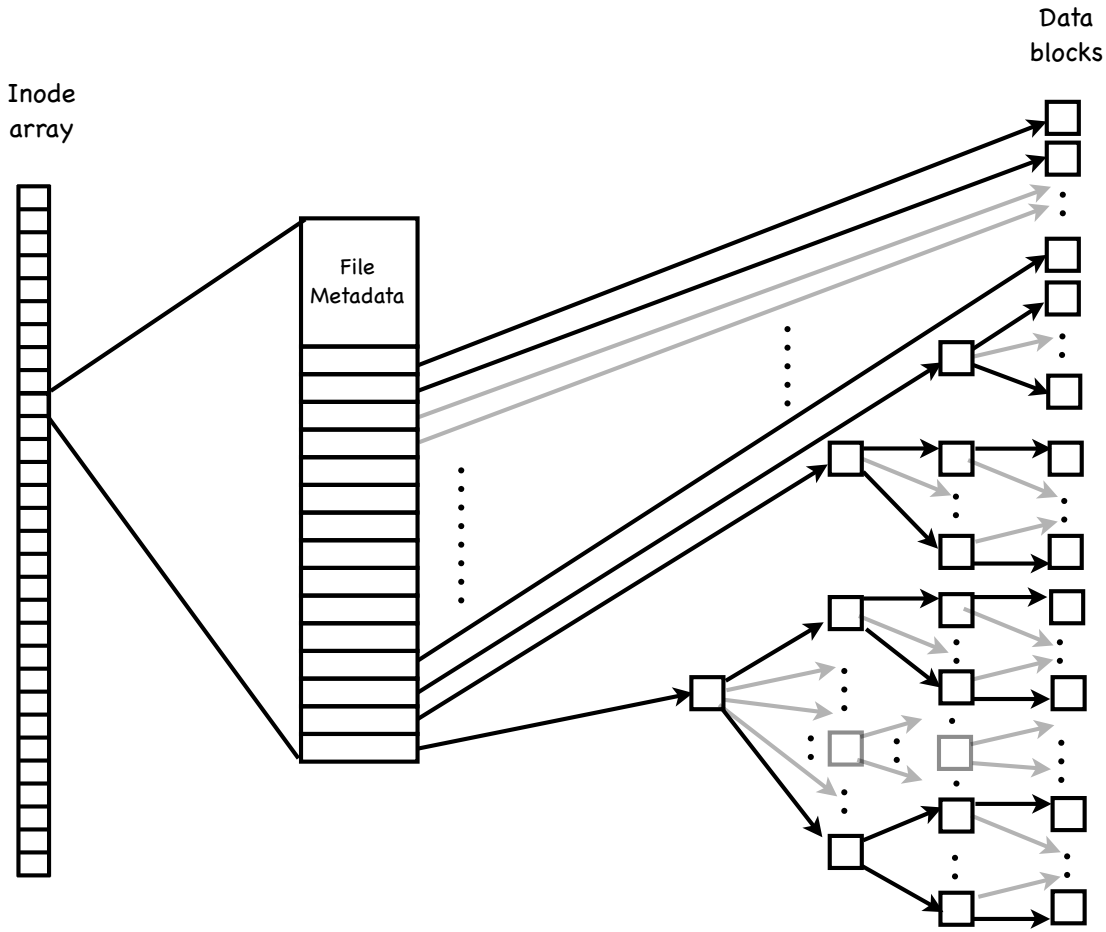
File structure

- Each file is a fixed, asymmetric tree, with fixed size data blocks (e.g. 4KB) as its leaves
- The root of the tree is the file's inode, containing
  - metadata (more about it later)
  - a set of 15 pointers
    - first 12 point to data blocks
    - last three point to intermediate blocks, themselves containing pointers...
      - #13: pointer to a block containing pointers to data blocks
      - #14: double indirect pointer
      - #15: triple indirect pointer (!)

Multilevel index

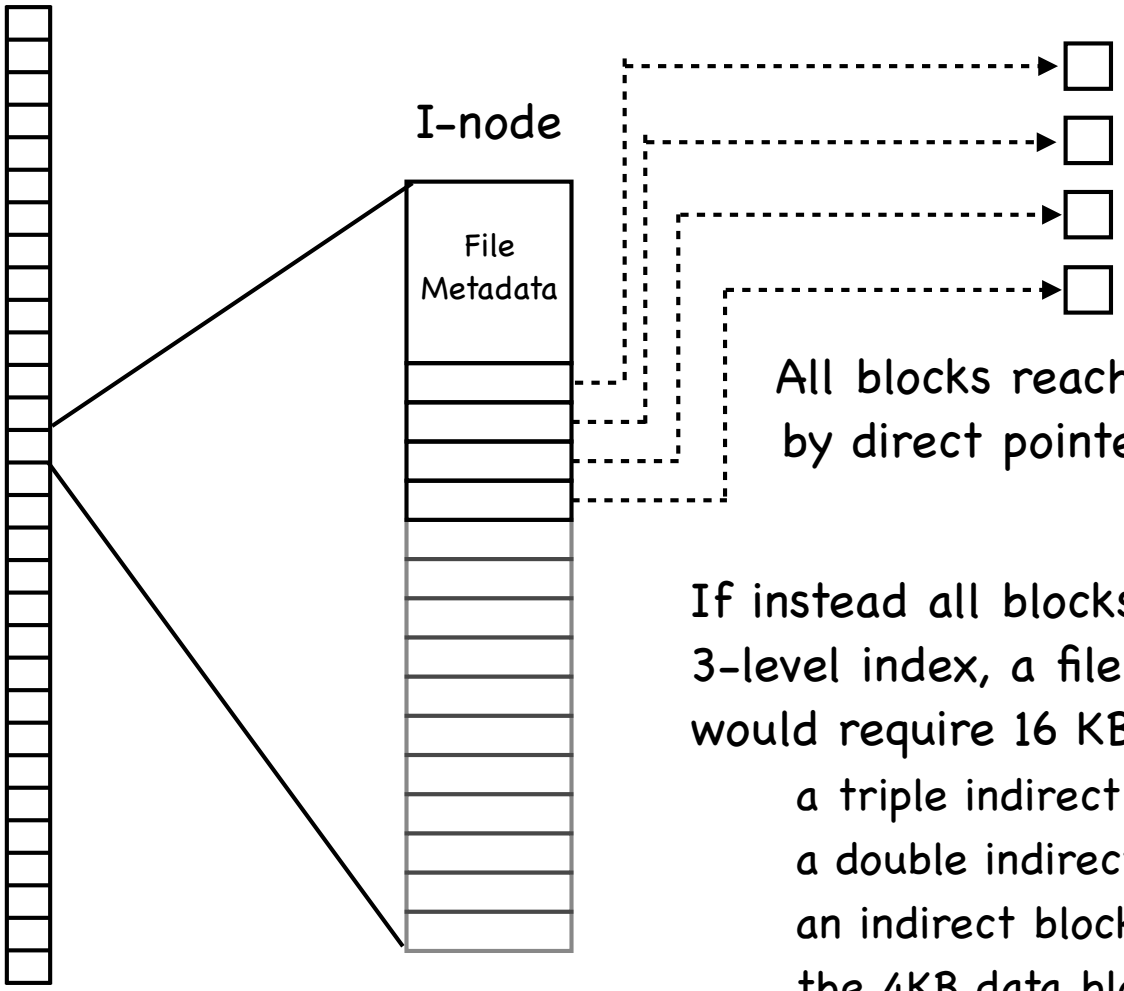


Multilevel index: key ideas



- Tree structure
  - efficient in finding blocks
- High degree
  - efficient in sequential reads
    - once an indirect block is read, can read 100s of data block
- Fixed structure
  - simple to implement
- Asymmetric
  - supports large files
  - small files don't pay large overheads

Good for small files...



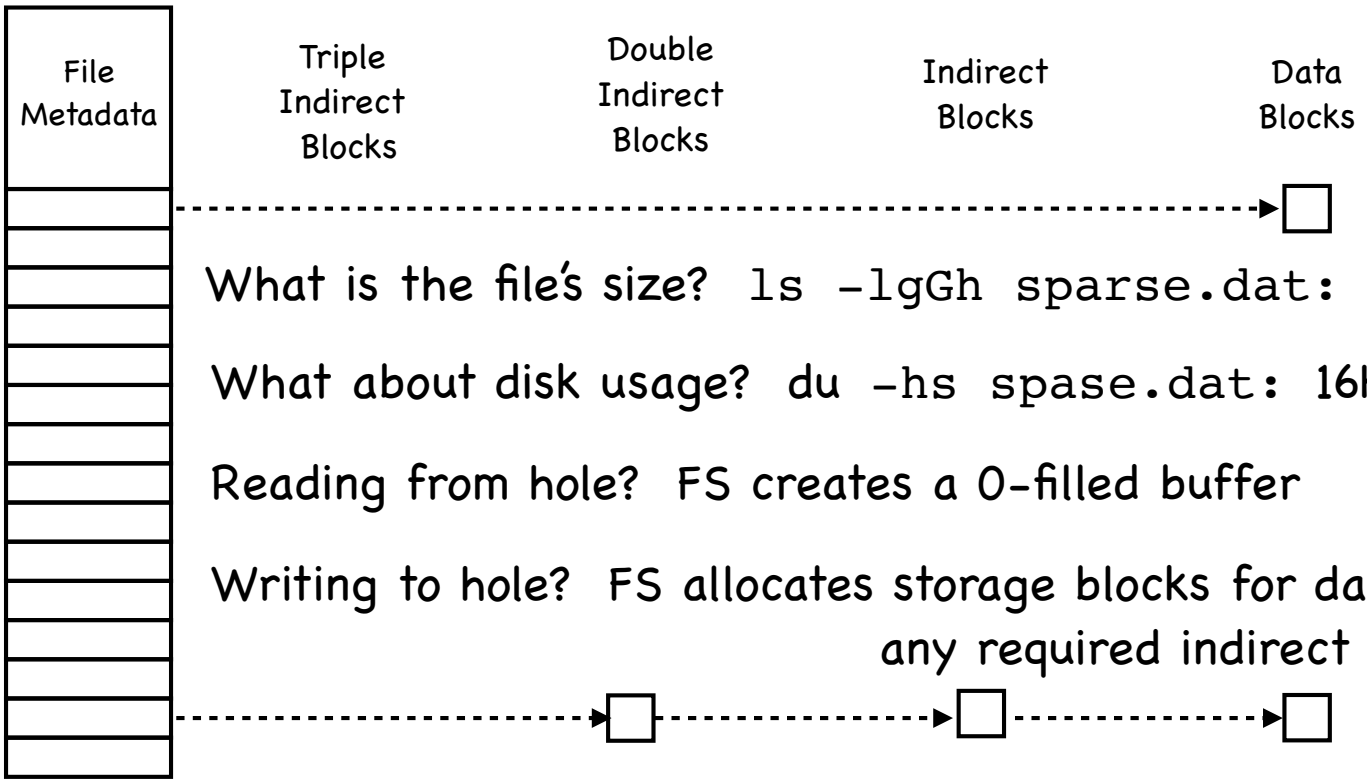
If instead all blocks were accessed through a 3-level index, a file occupying a single 4KB block would require 16 KB:

- a triple indirect block
- a double indirect block
- an indirect block
- the 4KB data block

reading would require reading 5 blocks to traverse the tree

...and for sparse files

Consider file `sparse.dat` with two 4K blocks: one at offset 0;
the other at offset 2^{30}



What is the file's size? `ls -lgGh sparse.dat:` 1.1.GB

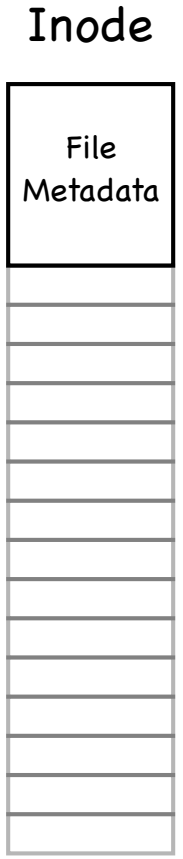
What about disk usage? `du -hs spase.dat:` 16KB (!)

Reading from hole? FS creates a 0-filled buffer

Writing to hole? FS allocates storage blocks for data +
any required indirect block

What else is in an i-node?

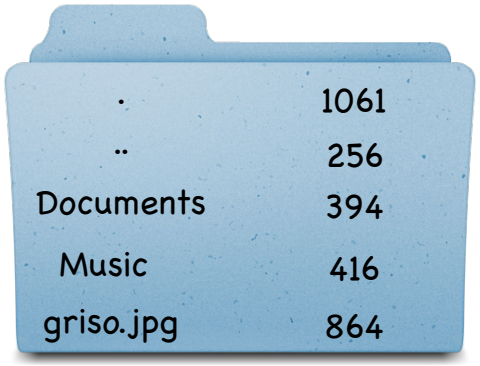
- Type
  - ordinary file
  - directory
  - symbolic link
  - special device
- Size of the file (in bytes)
- No. of links to the i-node
- Owner (user id & group id)
- Protection bits
- Times: creation, last accessed, last modified



Directory

- A file that contains a collection of mapping from file name to file number

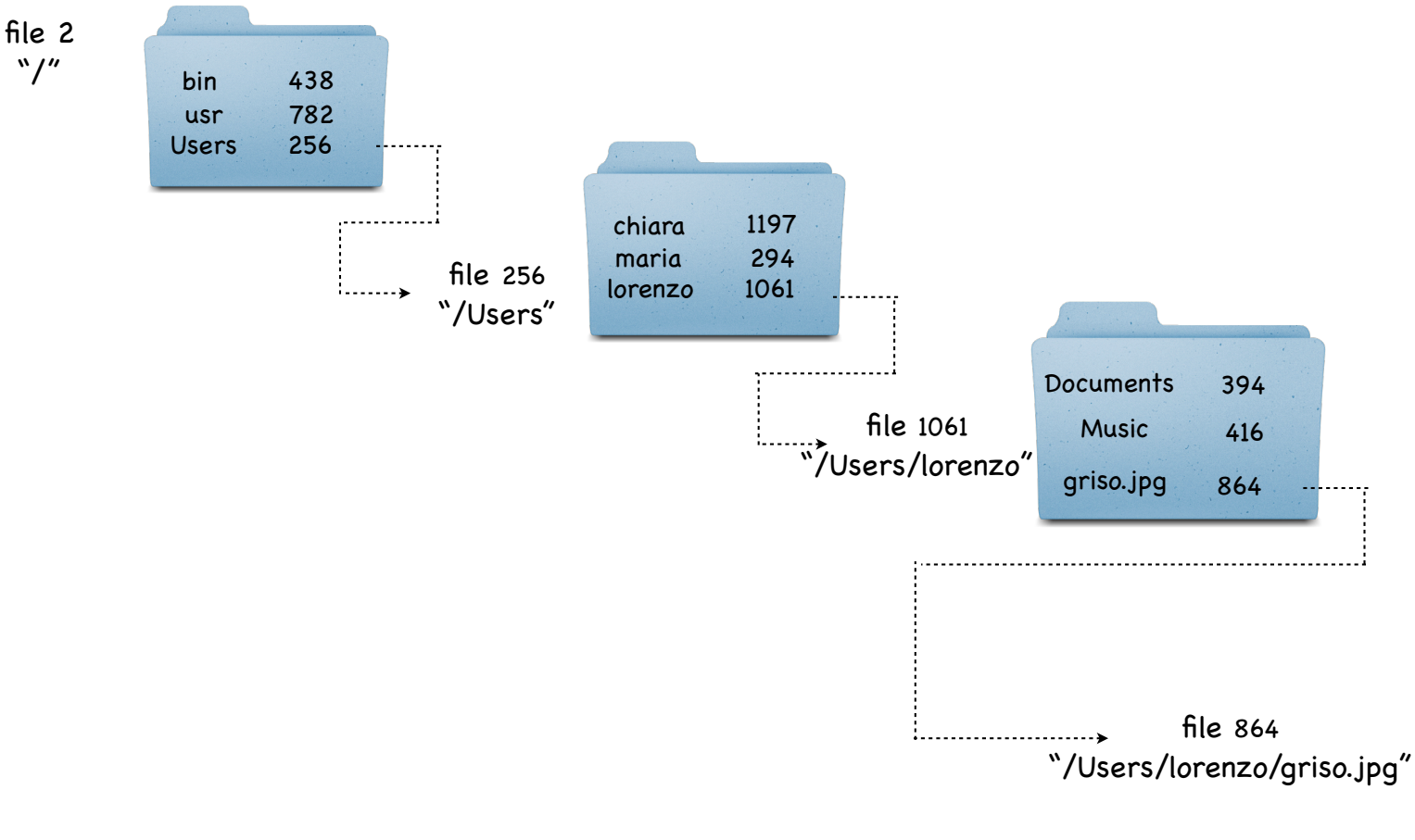
/Users/lorenzo



- To look up a file, find the directory that contains the mapping to the file number
- To find that directory, find the parent directory that contains the mapping to that directory's file number...
- Good news: root directory has well-known number (2)

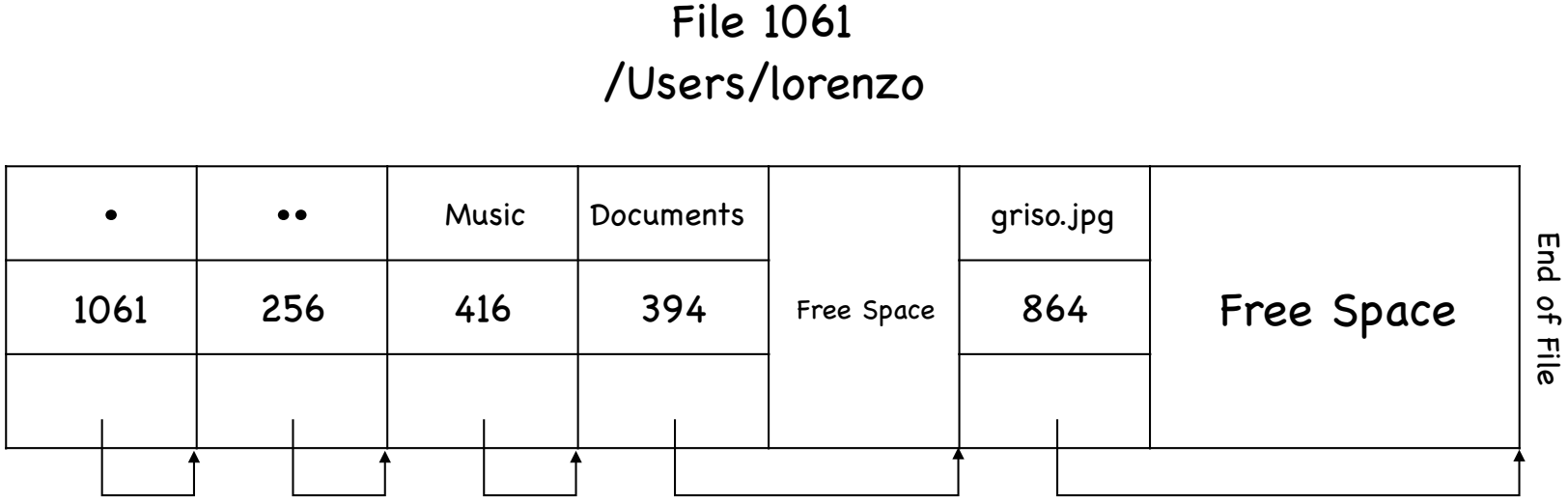
Looking up a file

- Find file /Users/lorenzo/griso.jpg



Directory Layout

- Directory stored as a file
  - Linear search to find filename (small directories)



- Larger directories use B trees
  - searched by hash of file name

Reading a File

- First, must open the file
  - open("/CS4410/roster", O_RDONLY)
  - Follow the directory tree, until we get to the inode for "roster"
  - Read that inode
    - do a permission check
    - return a file descriptor fd
- Then, for each read() that is issued:
  - read inode
  - read appropriate data block (depending on offset)
  - update last access time in inode
  - update file offset in in-memory open file table for fd

Read first 3 data blocks from /CS4410/roster

	data bitmap	inode bitmap	root inode	CS4410 inode	roster inode	root data	CS4410 data	roster data[0]	roster data[1]	roster data[2]
open(CS4410)			read()							
						read()				
				read()						
							read()			
					read()					
read()					read()					
								read()		
					write()					
read()					read()					
									read()	
					write()					
read()					read()					
										read()
					write()					

Writing a File

- Must open the file, like before
- But now may have to allocate a new data block
  - each logical write can generate up to five I/O ops
    - reading the free data block bitmap
    - writing the free data block bitmap
    - reading the file's inode
    - writing the file's inode to include pointer to the new block
    - writing the new data block
- Creating a file is even worse!
    - read and write free inode bitmap
    - write inode
    - (read) and write directory data
    - write directory inode

and if directory block is full, must allocate another block

Create /CS4410/roster & Write first 3 Data Blocks

	data bitmap	inode bitmap	root inode	CS4410 inode	roster inode	root data	CS4410 data	roster data[0]	roster data[1]	roster data[2]
create (/CS4410/roster)			read()							
						read()				
				read()						
							read()			
		read()								
		write()								
							write()			
					read()					
					write()					
				write()						
write()					read()					
	read()									
	write()									
								write()		
					write()					
write()					read()					
	read()									
	write()									
									write()	
					write()					
write()					read()					
	read()									
	write()									
										write()
					write()					

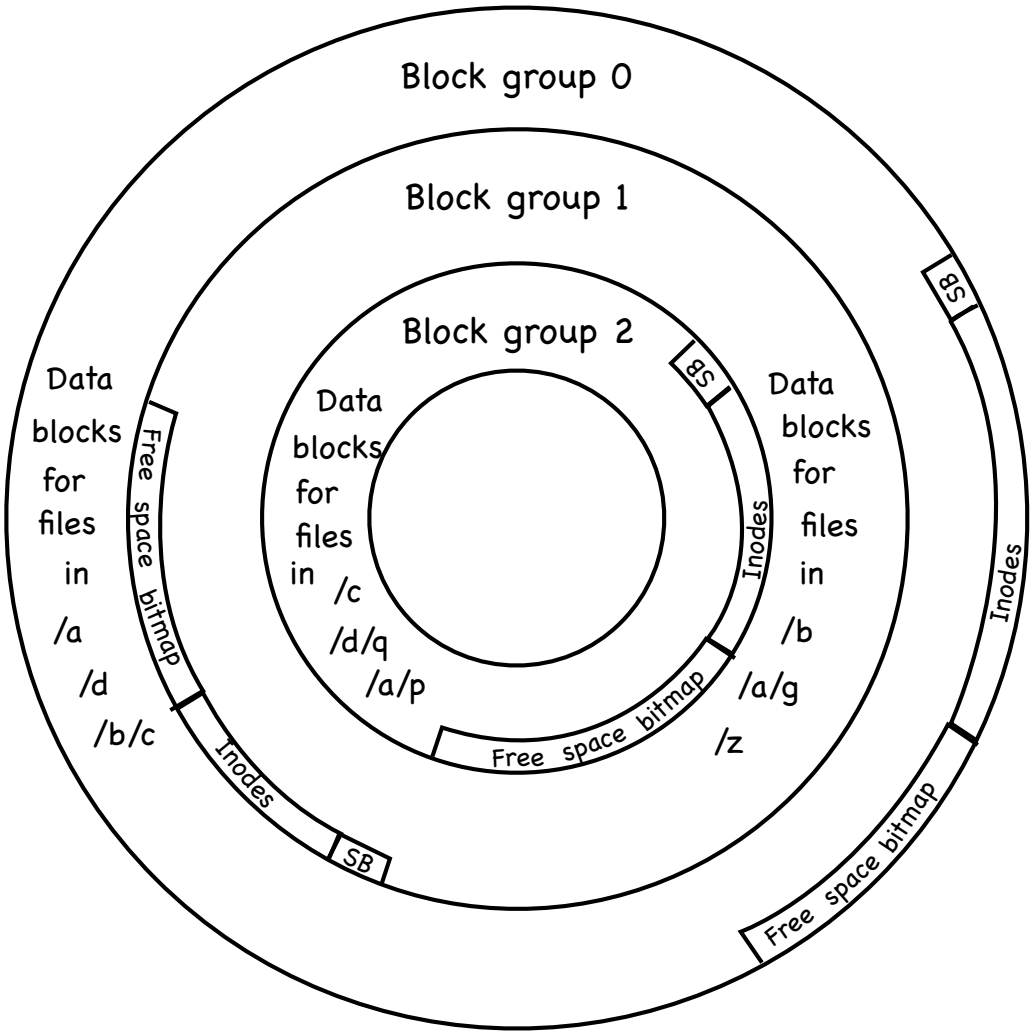
Caching

- Reading a long path can cause a lot of I/O ops!
- Cache aggressively!
  - early days: fixed sized cache for popular blocks
    - static partitioning can be wasteful
  - current: dynamic partitioning via unified page cache
    - virtual memory pages and file system blocks in a single cache
- Caching can significantly reduce disk I/O for reads
- Buffering can reduce cost of writes
  - some blocks may be overwritten
  - batching helps with scheduling disk accesses

BSD FFS: Fast File System

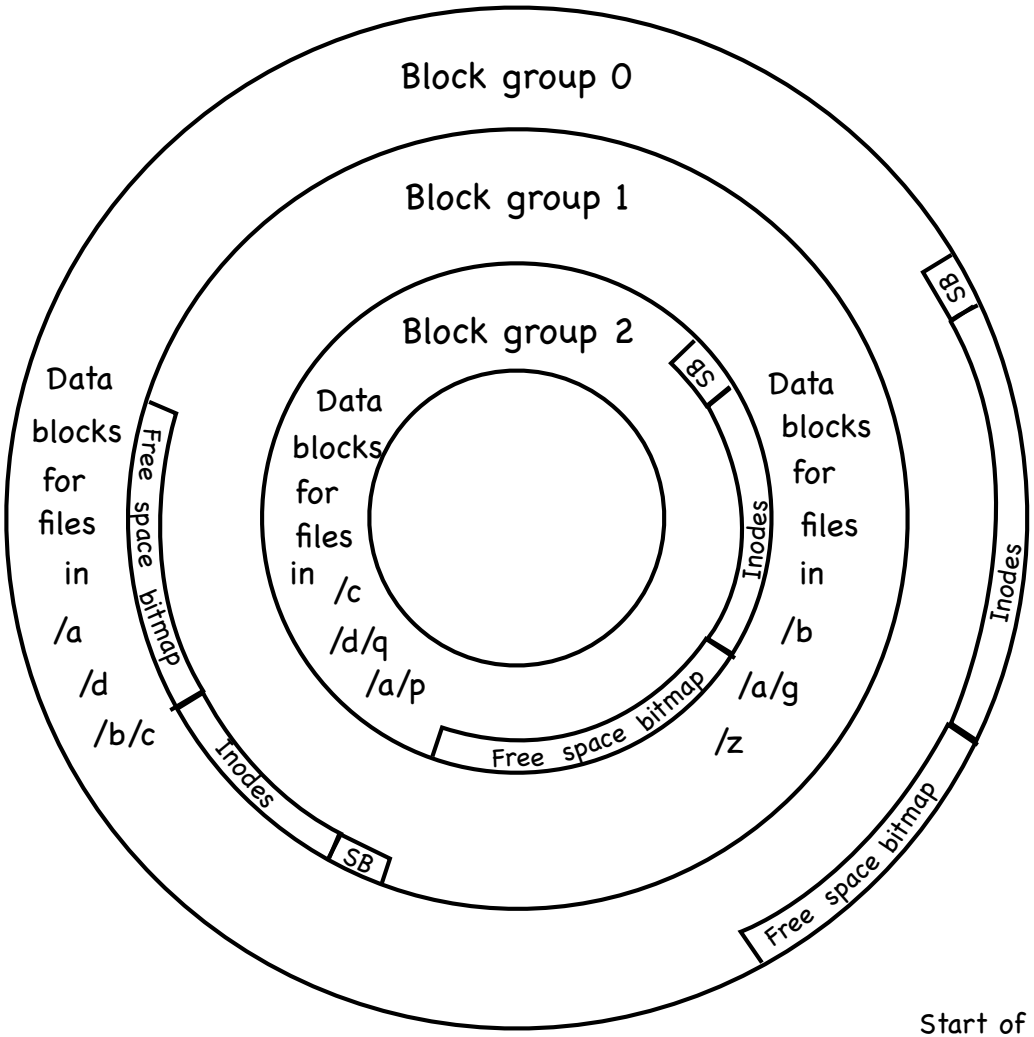
- UFS treats disks as if they were RAM
  - files grab first free data block: seeks and fragmentation
- FFS optimizes file system layout for how disks work
- Smart locality heuristics
  - block group placement
    - optimizes placement for when a file data and metadata, and other files within same directory, are accessed together
  - reserved space
    - gives up about 10% of storage to allow flexibility needed to achieve locality

Locality heuristics: block group placement

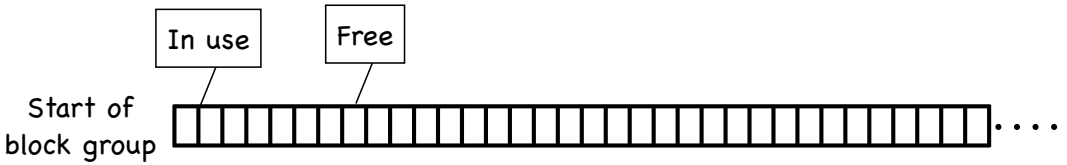


- Divide disk in block groups
  - sets of nearby tracks
- Distribute metadata
  - old design: free space bitmap and inode map in a single contiguous region
    - lots of seeks when going from reading metadata to reading data
  - FFS: distribute free space bitmap and inode array among block groups. Keep a superblock copy in each block group
- File Placement
  - when a new regular file is created, FFS looks for inodes in the same block as the file's directory
  - when a new directory is created, FFS places it in a different block from the parent's directory
- Data Placement
  - first free heuristics
  - trade short term for long term locality

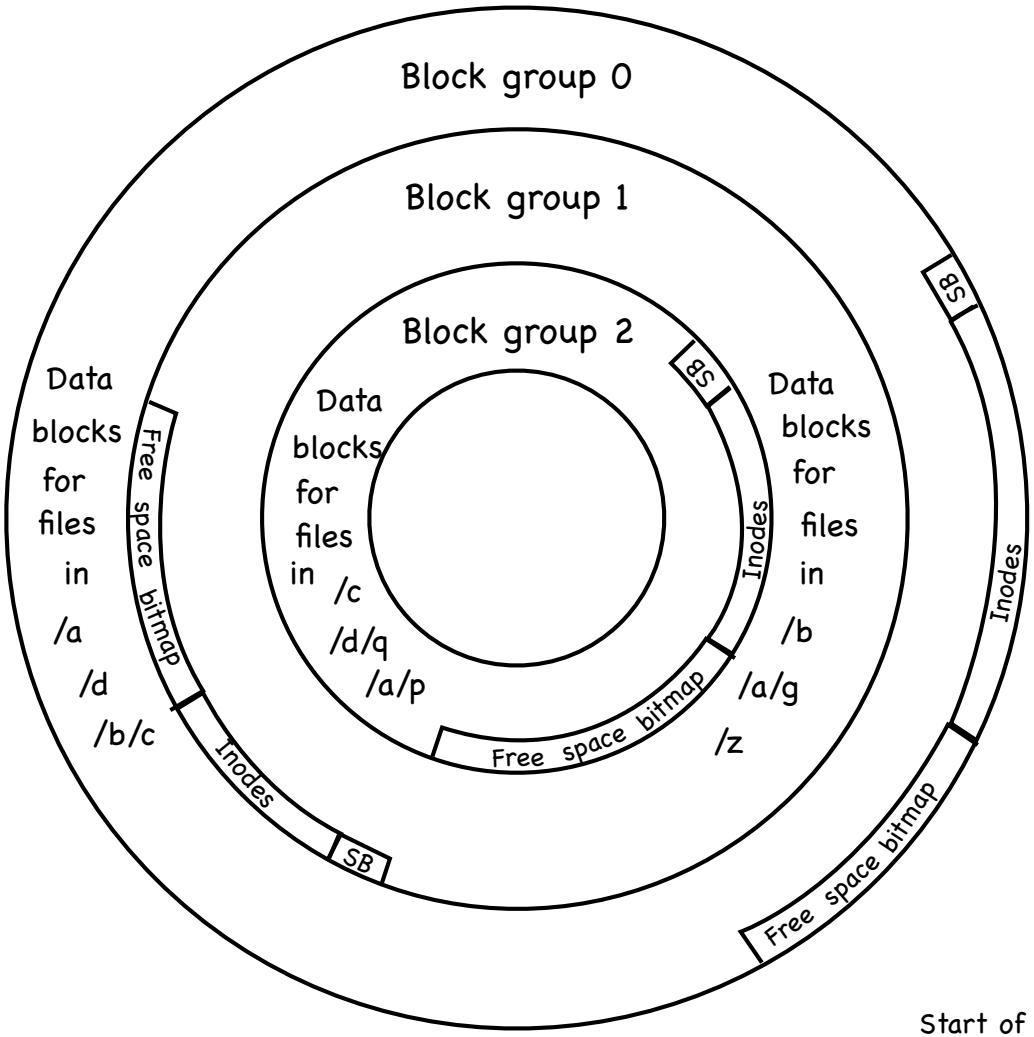
Locality heuristics: block group placement



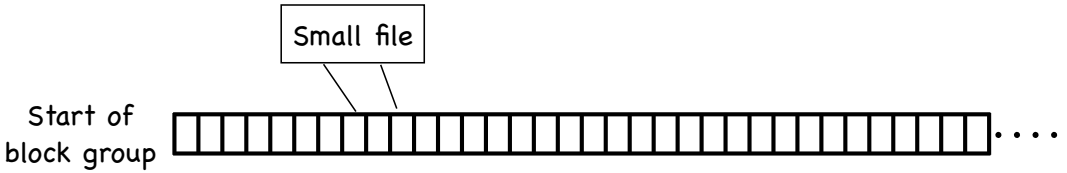
- Divide disk in block groups
  - sets of nearby tracks
- Distribute metadata
  - old design: free space bitmap and inode map in a single contiguous region
    - lots of seeks when going from reading metadata to reading data
  - FFS: distribute free space bitmap and inode array among block groups. Keep a superblock copy in each block group
- File Placement
  - when a new regular file is created, FFS looks for inodes in the same block as the file's directory
  - when a new directory is created, FFS places it in a different block from the parent's directory
- Data Placement
  - first free heuristics
  - trade short term for long term locality



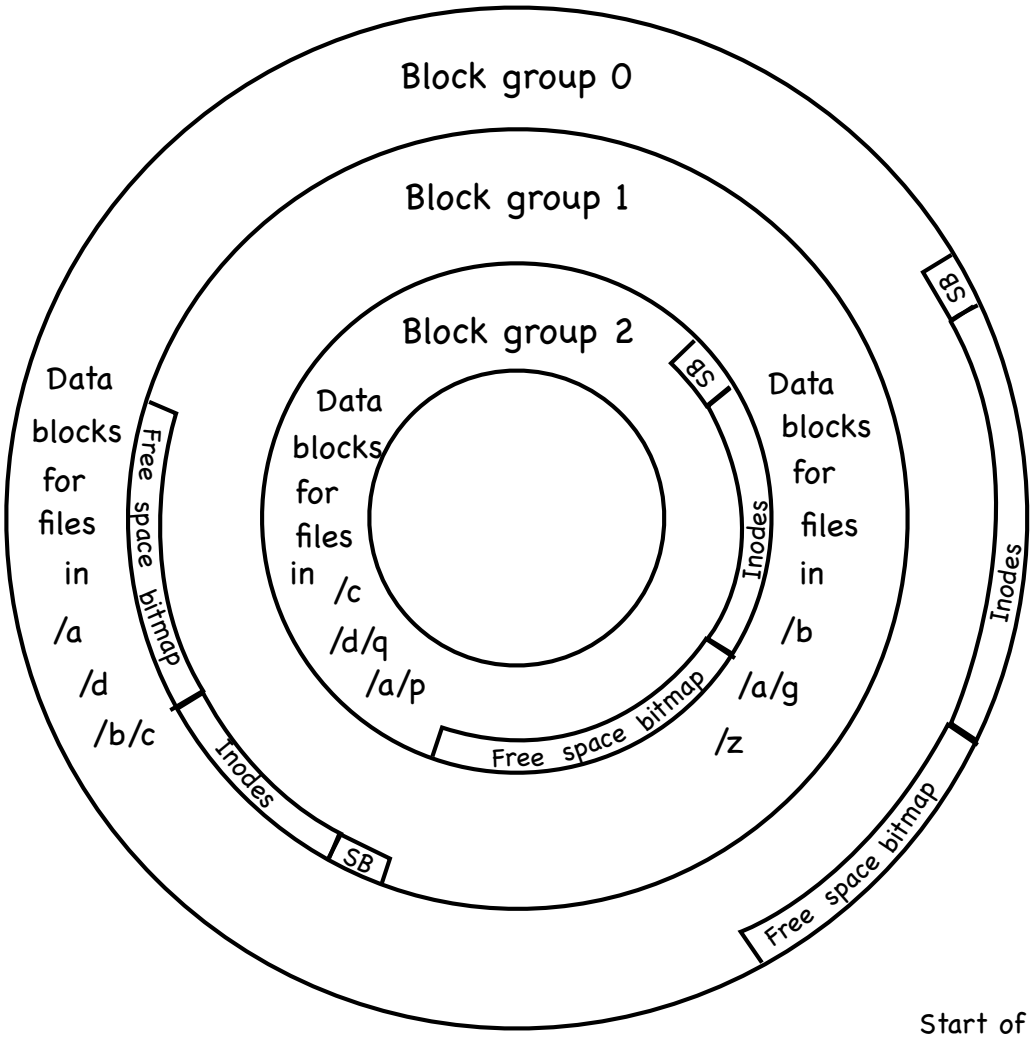
Locality heuristics: block group placement



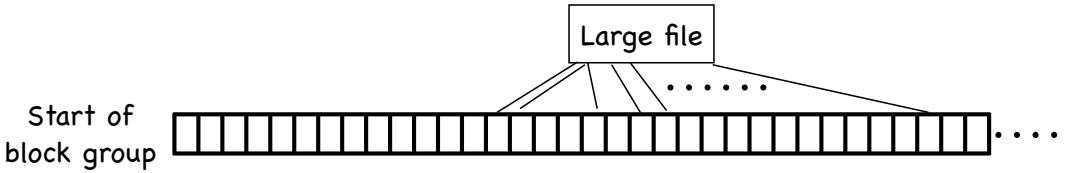
- Divide disk in block groups
  - sets of nearby tracks
- Distribute metadata
  - old design: free space bitmap and inode map in a single contiguous region
    - lots of seeks when going from reading metadata to reading data
  - FFS: distribute free space bitmap and inode array among block groups. Keep a superblock copy in each block group
- File Placement
  - when a new regular file is created, FFS looks for inodes in the same block as the file's directory
  - when a new directory is created, FFS places it in a different block from the parent's directory
- Data Placement
  - first free heuristics
  - trade short term for long term locality



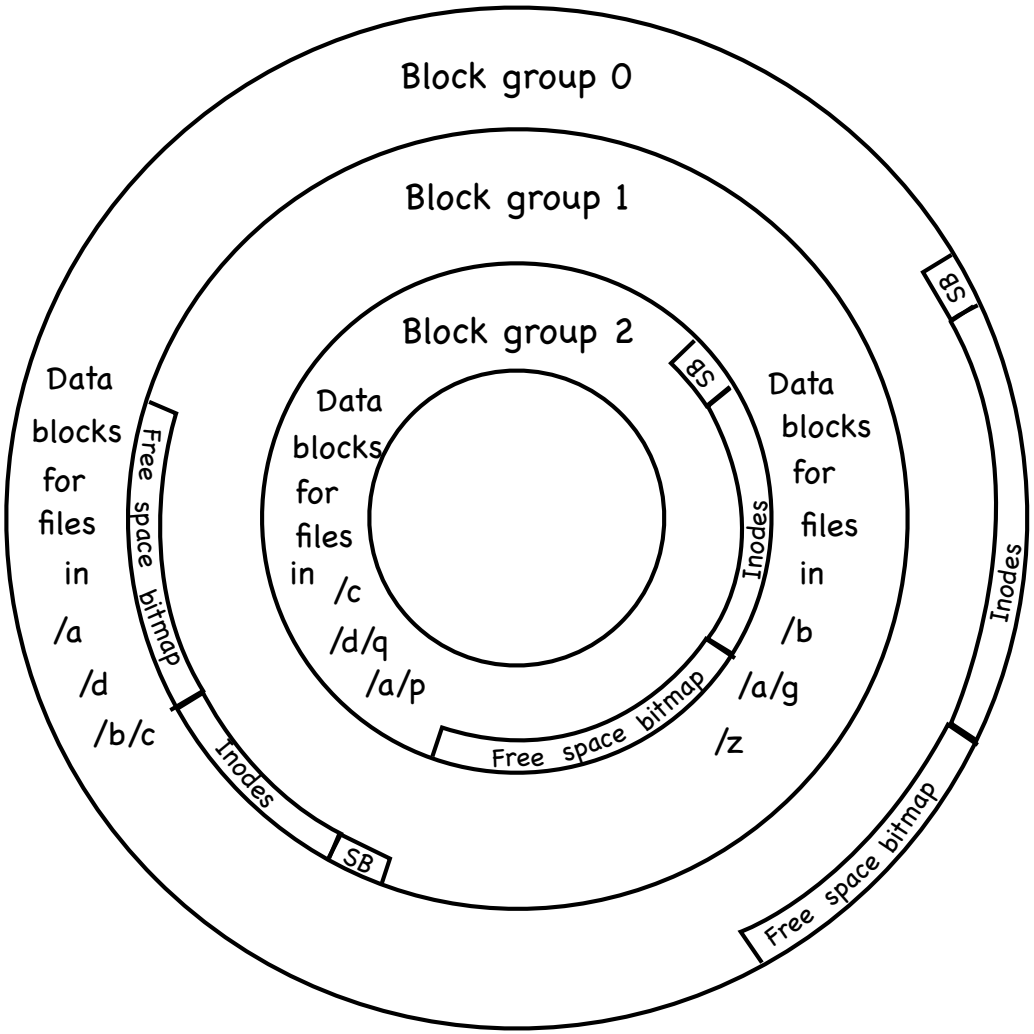
Locality heuristics: block group placement



- Divide disk in block groups
  - sets of nearby tracks
- Distribute metadata
  - old design: free space bitmap and inode map in a single contiguous region
    - lots of seeks when going from reading metadata to reading data
  - FFS: distribute free space bitmap and inode array among block groups. Keep a superblock copy in each block group
- File Placement
  - when a new regular file is created, FFS looks for inodes in the same block as the file's directory
  - when a new directory is created, FFS places it in a different block from the parent's directory
- Data Placement
  - first free heuristics
  - trade short term for long term locality



Locality heuristics: reserved space



- When a disk is full, hard to optimize locality
  - file may end up scattered through disk
- FFS presents applications with a smaller disk
  - about 10%-20% smaller
  - user's write that encroaches on reserved space fails
  - super user still able to allocate inodes to clean things up

Long File Exception

- Blocks of a huge file not all in the same block group
  - or they will eat up all the blocks in the group!
  - Instead, 12 blocks in a group (direct index)
  - others divided in "chunks"
- Locality lost when moving between chunks
  - choose chunk size to amortize cost of seeks

Say we want 90% of peak transfer, and transfer rate is 40MB/s

if positioning time (seek+rotation) is 10ms, we need a chunk large enough that transfer takes 90ms

$$\text{chuck size} = \frac{40\text{MB}}{\text{s}} \times \frac{1\text{s}}{1000\text{ms}} \times 90\text{ms} = 3.6\text{ MB}$$

In practice, FFS uses 4 MB chunks