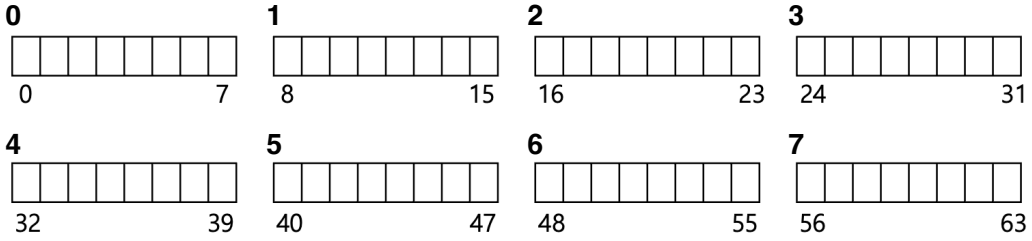
Persistence

- 36 - I/O Devices
- 37 - Hard Disk Drives
- 38 - RAID
- 39 - File and Directories
- 40 - File System Implementation
- 41 - Locality and the Fast File System
- 42 - Crash Consistency
- 43 - Log-structured File Systems
- 44 - Flash-based SSD
- 45 - Data Integrity and Protection

Let's Start With Blocks…

- File systems address disks by *block*
  - *Logical* block numbers are an arbitrary mapping over physical
  - blocks are multiples of disk *sectors*
  - usually 8k or 16k (512 bytes for GeekOS)
- Assume 512-byte sectors and 4k pages in the following
  - physical block numbers start at 0:

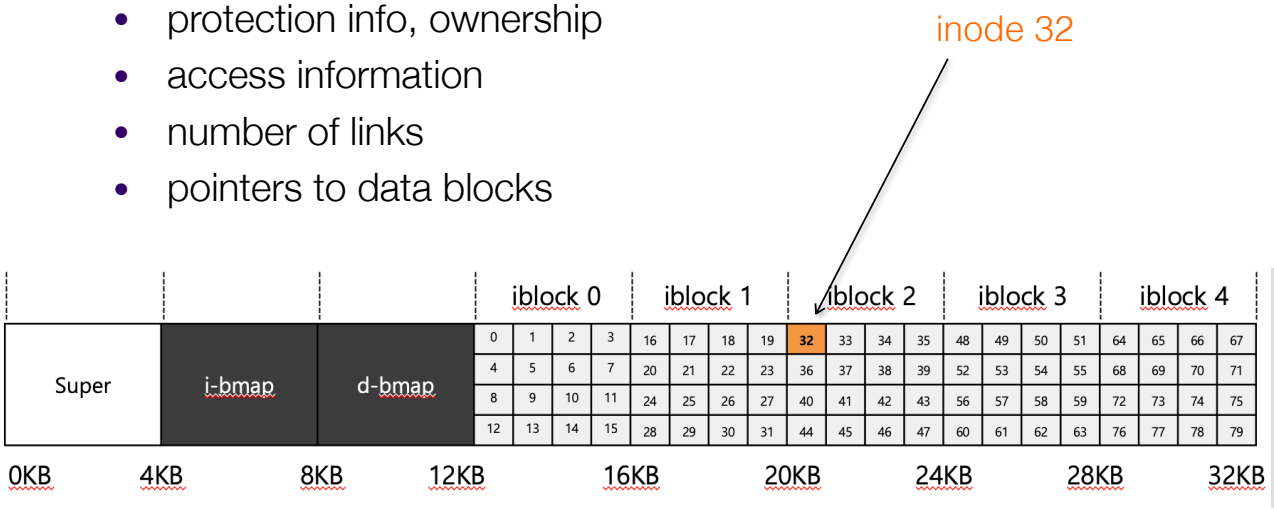


Disk Organization

- Blocks on disk
  - *super block*
    - configuration for a specific file system instance
    - boot code
    - size and location of inode tables, etc.
  - *data blocks*
    - blocks containing file data
  - *inode blocks*
    - inode structures w/ file metadata
  - *indirect pointer blocks*
    - blocks full of pointers to other blocks
  - *bitmaps*
    - used/free information for data and inode blocks

Simplified Disk Layout

- array of per-file metadata in *inodes:*
  - *inumber* : index into the inode table
  - file type (regular file, directory, etc.)
  - size, number of blocks
  - protection info, ownership
  - access information
  - number of links
  - pointers to data blocks



Ext2 (old linux) Inodes

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?
4	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
2	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), often 2 indrect, 1 doubly indirect
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists
4	faddr	an unsupported field
12	i_osd2	another OS-dependent field

Indirect blocks

- W/ 512-bytes blocks, 4-byte pointers:
  - the 15 block pointers can accommodate files up to size 7.5K
- W/ a single level of indirection:
  - $15 * \frac{512}{4} \times 512 = 15 \times 2^{16} = 983\text{K}$, much larger!
- W/ double indirection:
  - $15 \times \frac{512}{4} \times \frac{512}{4} \times 512 = 15 \times 2^{23} = 120\text{MB}$
- Used in most large file systems:
  - Linux EXT2, EXT3, NetApp's WAFL, Unix file system
  - Linux EXT4 uses *extents* instead of simple pointers
    - extent lets a pointer reference consecutive blocks

File System Numbers

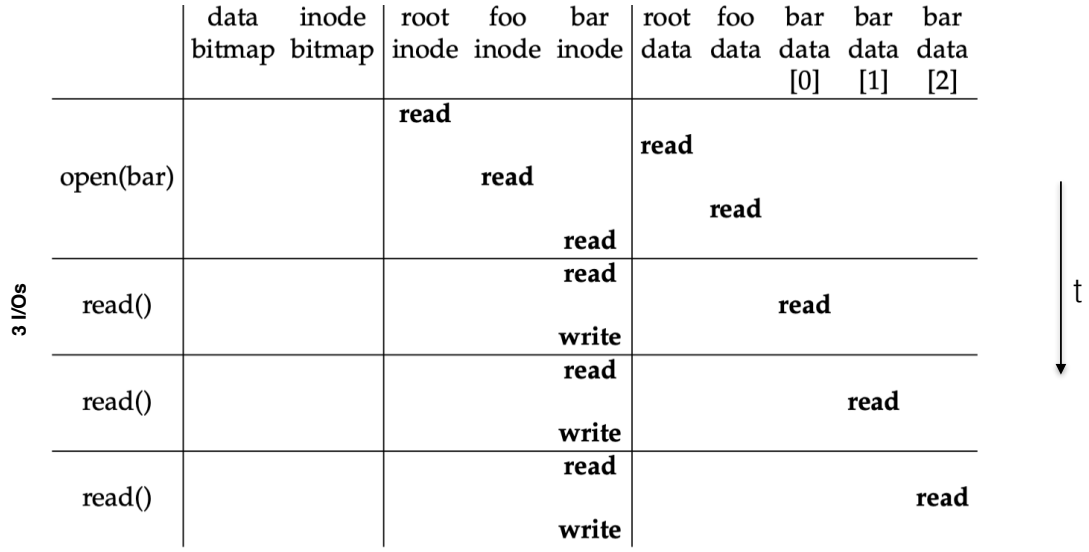
- Rules of thumb:

• most are small	*2K most common*
• avg size is growing	*over 200K*
• most bytes are in large files	
• there are many	*100K on average*
• most FS are about half full	*disk size grows, so do files*
• directories typically small	*most have 20 or fewer*

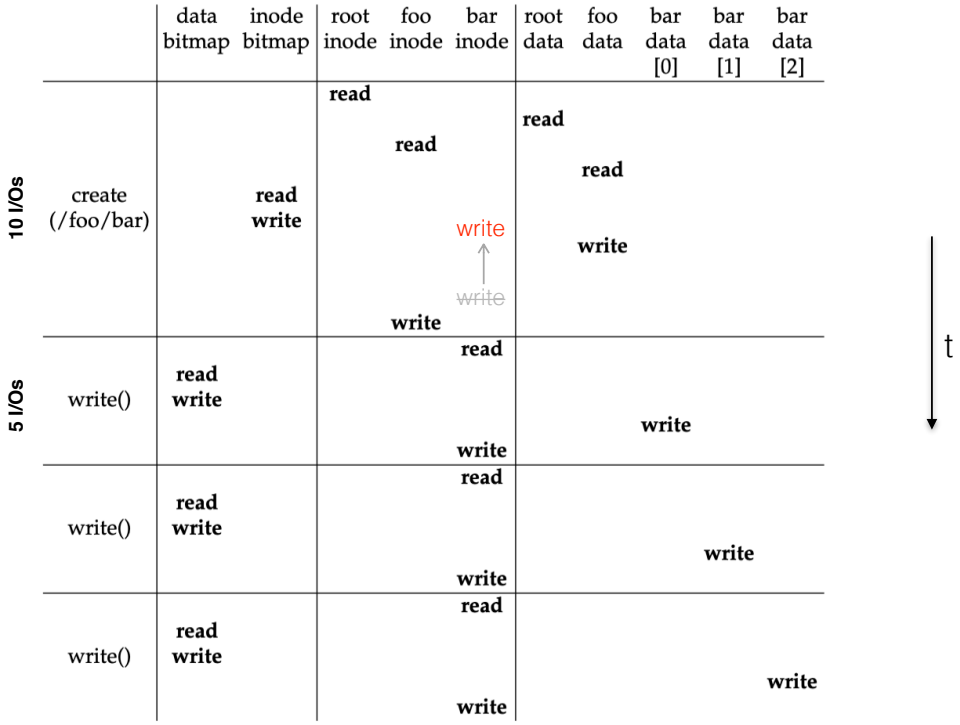
Reading a file from disk

- Issue an open("/foo/bar", O_RDONLY),
- Traverse the pathname
  - begin at the root of the file system (/)
    - root inode number often 2 (in superblock)
    - read in block containing inode 2.
    - use "/" pointer blocks to get "/" directory contents
  - recurse on "/foo"
  - check permissions, memory for metadata, file descriptor
- when `read()` issued
  - consult inode, find and read in first block
  - update open file table, file offset
- When file closed
  - dealloc file descriptor, logically the file may be deallocated, but not usually done here

Open and read /foo/bar timeline



Create /foo/bar timeline



Unified Buffer Cache

- buffer cache
  - recently used pages/disk-blocks held in memory
  - writes *buffered*(delayed)
    - consecutive writes batched
    - *scheduled* more efficiently
  - dynamically partitioned (not fixed size)
- some apps (e.g. databases) ignore the cache
  - call `rsync()`
  - use *direct I/O* interfaces to disk
  - write to raw disks

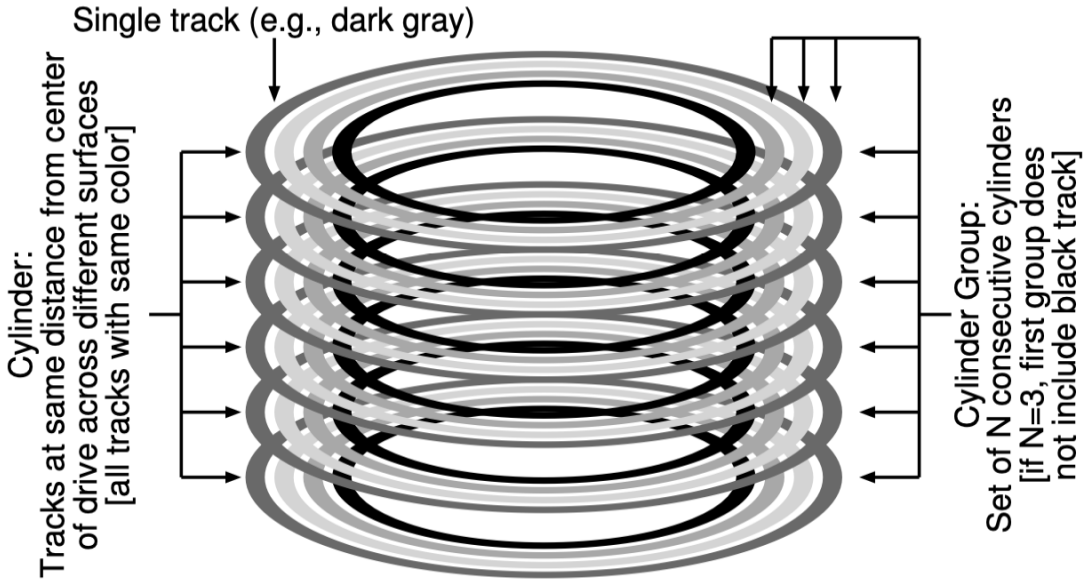
Locality and the Fast File System (FFS)

- "old" file system

  - performance starts bad, gets worse
    - fragmentation as files deleted and created
    - block size too small (slow transfers)
    - `inodes` not near data
- FFS fixes many of these problems
  - ….but we are still talking about the 1990s…

Cylinder groups

- …. could be useful but disks do not export enough info



Block Groups

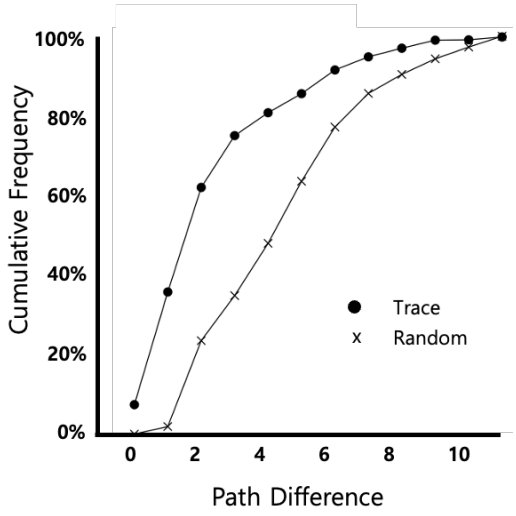
- FFS uses *block groups*
  - disk maps onto cylinder groups
  - each has superblock, bitmaps, inodes, data

- "Keep related stuff together" - single group
  - most directories
  - file data and related inodes
  - large files have chunks sprayed across multiple groups

How Far are Accesses

- How "**far away**" file accesses were from one another in the directory tree.

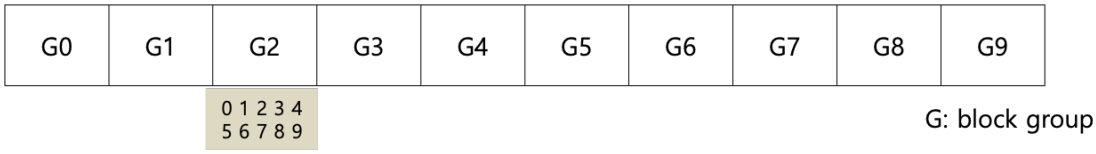
```
            proc/src/foo.c
            proc/src/bar.c
the distance of two file access is 1

            proc/src/foo.c
            proc/obj/foo.o
the distance of two file access is 2
```



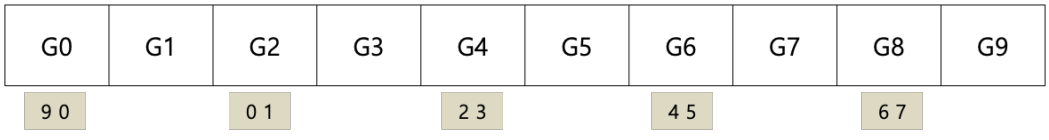
- 7% of file accesses to the same file
- Nearly 40% of file accesses in the same directory
- 25% of file accesses were two distances

Large File Exception

- Usual file placement policy
  - a large file might fill a group entirely
  - lose directory locality

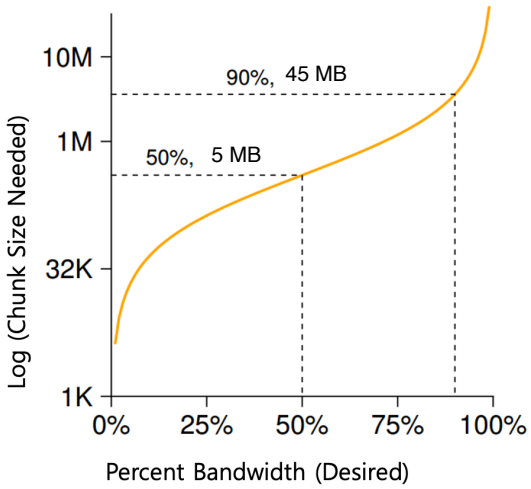


- Fix: spread large chunks across multiple groups
  - make chunks large enough to perform well



How Large Should Chunks Be?

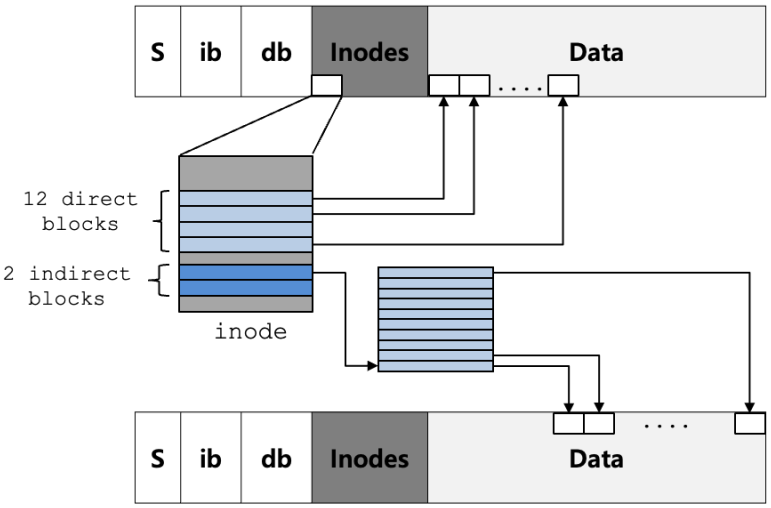
- If want 50% of peak disk performance
  - ⇒ half time seeking, half time transferring data
  - If we assume:
    - disk bandwidth 1GB/sec
    - positioning time 5 msec
    - $\frac{1GB}{\text{sec}} = \frac{1MB}{\text{msec}}$ ⇒ need $5 \times 1\text{MB} = 5\text{MB}$ chunks



- 90% peak performance w/ 45 MB chunks

Simple Approach Used by FFS

- Use *inode* structure
  - direct links and blocks same group
  - indirect blocks, and blocks pointed to, different group
  - each 1024 blocks (4MB) in different group (4K pages, 4-byte ptr)



Errata

- more internal fragmentation
- used subblocks
  - copy to regular blocks when full
  - `libc` buffers, so most large files never subblock'd
- parameterization
  - blocks laid out so that OS has time to request block $i + 1$ after reading block it, *before* $i + 1$ rotates past
- track buffer
- long file names
- symbolic links