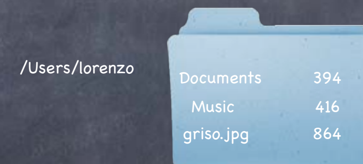
Implementation: key ideas

- Directories
  - file name ⟶ file number
- Index structures
  - file number ⟶ block
- Free space maps
  - find a free block; actually, find a free block nearby
- Locality heuristics
  - policies enabled by above mechanisms
    - group together directory files
    - make writes sequential
    - defragment

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)

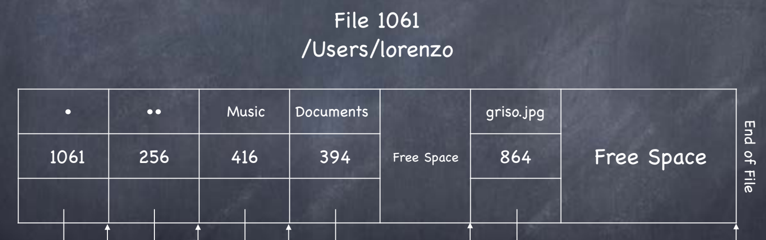
Looking up a file

- Find file /Users/lorenzo/griso.jpg

Directory Layout

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

File 1061
/Users/lorenzo



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

Finding data

- Index structure provides a way to locate each of the file's blocks
  - usually implemented as a tree for scalability
- Free space map provides a way to allocate free blocks
  - often implemented as a bitmap
- Locality heuristics group data to maximize access performance

Case studies

- FAT late 70s; Microsoft
  - key idea: linked list
  - Today: flash sticks
- Unix FFS mid 80's
  - key idea: tree-based multi-level index
  - Today: Linux ext2 and ext3
- NTFS early 1990s; Microsoft.
  - Key idea: variable size extents instead of fixed size blocks
  - Today: Windows 7, Linux ext4, Apple HFS
- ZFS early 2000; open source.
  - Key idea: copy on write (COW)

FAT File system

Microsoft, late 70s

- File Allocation Table (FAT)
  - started with MSDOS
  - in FAT-32, supports 2^{28} blocks and files of $2^{32}-1$ bytes

Index Structures
File Allocation Table (FAT)
- array of 32-bit entries
- file represented as a linked list of FAT entries
- file # = index of first FAT entry

Free space map
- If data block i is free, then FAT[i] = 0
- find free blocks by scanning MFT

Locality heuristics
- As simple as next fit:
  - scan sequentially from last allocated entry and return next free entry
- Can be improved through defragmentation

FAT File system

Microsoft, late 70s

- File Allocation Table (FAT)
  - started with MSDOS
  - in FAT-32, supports 2^{28} blocks and files of $2^{32}-1$ bytes

Advantages
- simple!
  - used in many USB flash keys
  - used even within MS Word!

Disadvantages
- Poor locality
  - next fit? seriously?
- Poor random access
  - needs sequential traversal
- Limited access control
  - no file owner or group ID metadata
  - any user can read/write any file
- No support for hard links
  - metadata stored in directory entry
- Volume and file size are limited
  - FAT entry is 32 bits, but top 4 are reserved
  - no more than 2^{28} blocks
  - with 4kB blocks, at most 1TB volume
  - file no bigger than 4GB
- No support for transactional updates

FFS: Fast File System

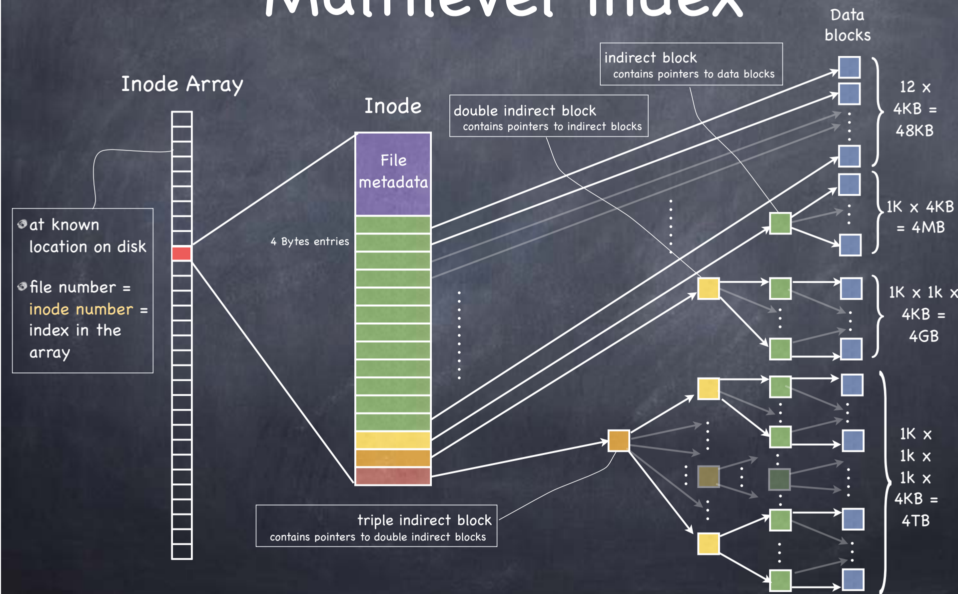
Unix, 80s

- Smart index structure
  - multilevel index allows to locate all blocks of a file
    - efficient for both large and small files
- 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

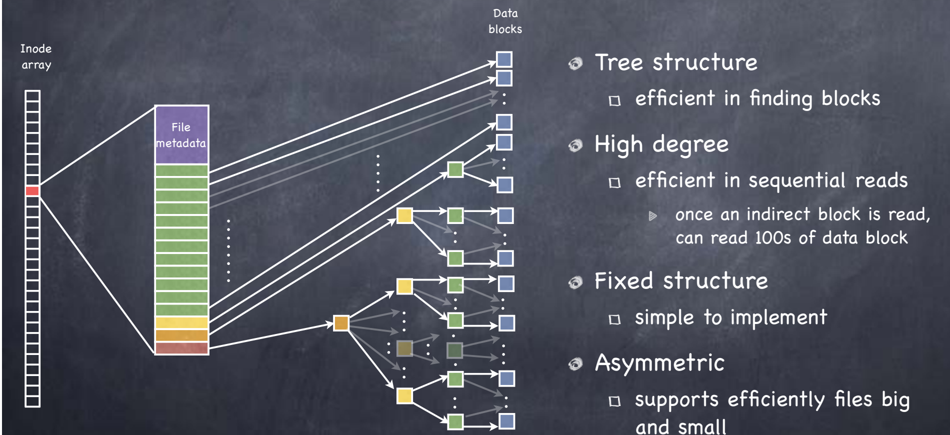
File structure

- 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
  - contains file's metadata
    - owner, permissions (rwx for owner, group other), type, creation time, etc
    - setuid: run with temporarily elevated privileges
      - file is executed with the permissions of the owner, not the caller
      - add flexibility but can introduce security risks
    - setgid: like setuid for groups
  - contains a set of pointers
    - typically 15
    - first 12 point to data block
    - last three point to intermediate blocks, themselves containing pointers
      - 13: indirect pointer
      - 14: double indirect pointer
      - 15: triple indirect pointer

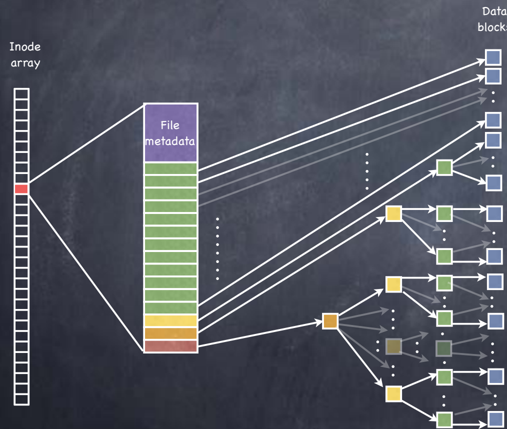
Multilevel index



Multilevel index: key ideas



Example: variations on the FFS theme



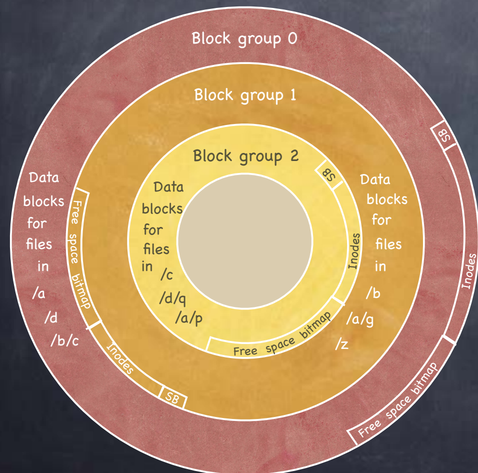
- In BigFS an inode stores
- 4kb blocks, 8 byte pointers
  - 12 direct pointers
  - 1 indirect pointer
  - 1 double indirect
  - 1 triple indirect
  - 1 quadruple indirect
- What is the maximum size of a file?
  - Through direct pointers
    - 12 x 4kb = 48KB
  - Indirect pointer
    - 512 x 4kb = 2MB
  - Double indirect pointer
    - 512^2 x 4kb = 1GB
  - Triple indirect pointer
    - 512^3 x 4kb = 512GB
  - Quadruple indirect pointer
    - 512^4 x 4kb = 256TB

Total = $(256 + .5 + 10^{-6} + 2 \times 10^{-9} + 4.8 \times 10^{-11}) \approx 256.5$ TB

Free space management

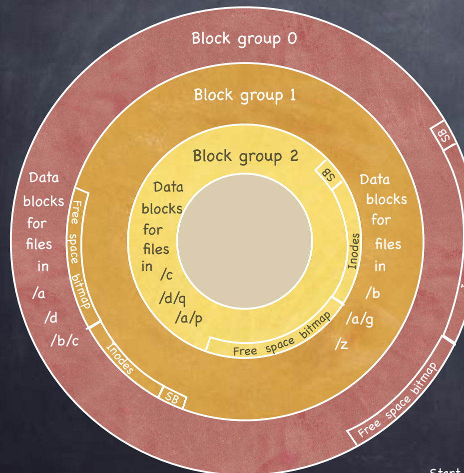
- Easy
  - a bitmap with one bit per storage block
  - bitmap location fixed at formatting time
  - i-th bit indicates whether i-th block is used or free

Locality heuristics: block group placement

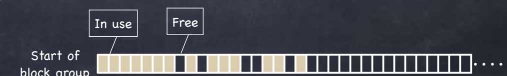


- Divide disk in block groups
  - sets of nearby tracks
- Distribute metadata
  - old design: free space bitmap and inode map in a single contiguous region
    - lots of seeks when going from reading metadata to reading data
  - FFS: distribute free space bitmap and inode array among block groups. Keep a superblock copy in each block group
- Place file in block group
  - when a new 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
- Place data blocks
  - first free heuristics
  - trade short term for long term locality

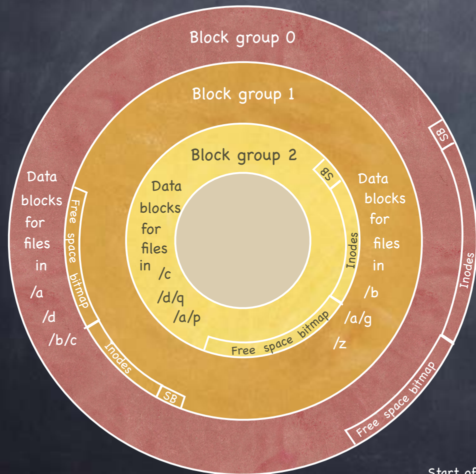
Locality heuristics: block group placement



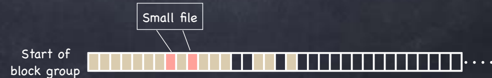
- Divide disk in block groups
  - sets of nearby tracks
- Distribute metadata
  - old design: free space bitmap and inode map in a single contiguous region
    - lots of seeks when going from reading metadata to reading data
  - FFS: distribute free space bitmap and inode array among block groups. Keep a superblock copy in each block group
- Place file in block group
  - when a new 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
- Place data blocks
  - first free heuristics
  - trade short term for long term locality



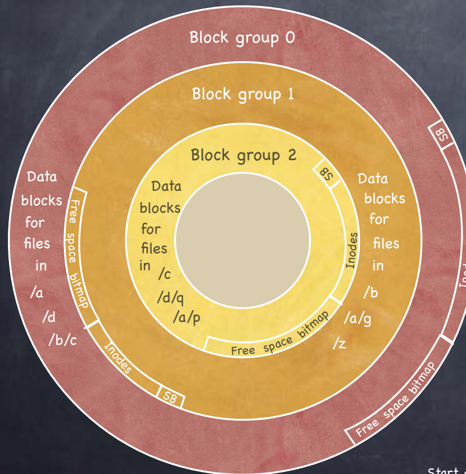
Locality heuristics: block group placement



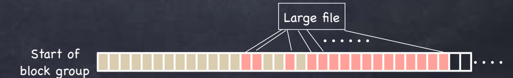
- Divide disk in block groups
  - sets of nearby tracks
- Distribute metadata
  - old design: free space bitmap and inode map in a single contiguous region
    - lots of seeks when going from reading metadata to reading data
  - FFS: distribute free space bitmap and inode array among block groups. Keep a superblock copy in each block group
- Place file in block group
  - when a new 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
- Place data blocks
  - first free heuristics
  - trade short term for long term locality



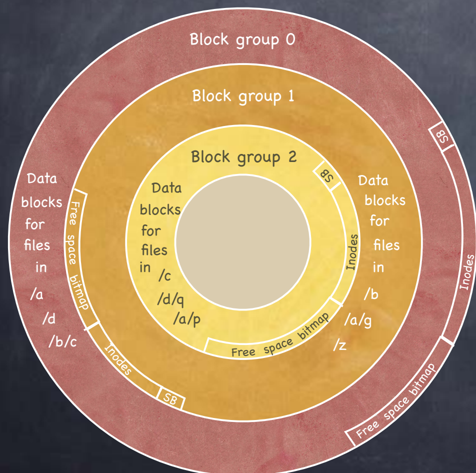
Locality heuristics: block group placement



- 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
- Place file in block group
  - when a new 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
- Place data blocks
  - 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 write that encroaches on reserved space fails
  - super user still able to allocate inodes to clean things up

FFS: A Perspective

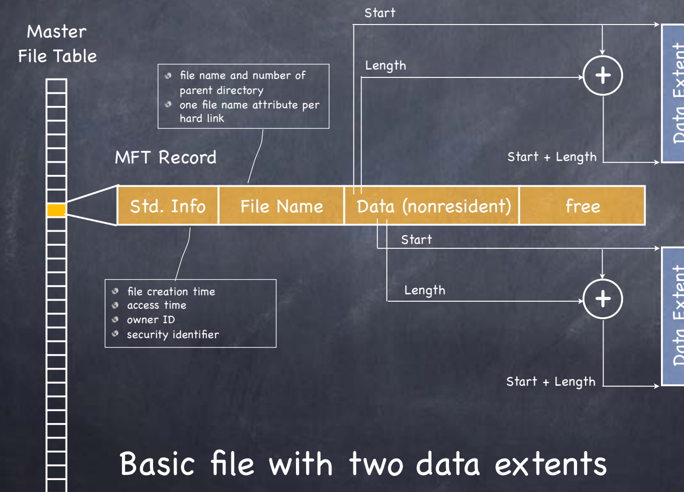
- Pros
  - Efficient storage for both small and large files
  - Locality for both small and large files
  - Locality for metadata
  - Fixed structure lead to simple implementation
- Cons
  - Inefficient for tiny file
    - need both inode and data block
  - Inefficient encoding for mostly contiguous files
  - Needs 10%-20% unutilized to prevent fragmentation

NTFS: Flexible Tree with Extents

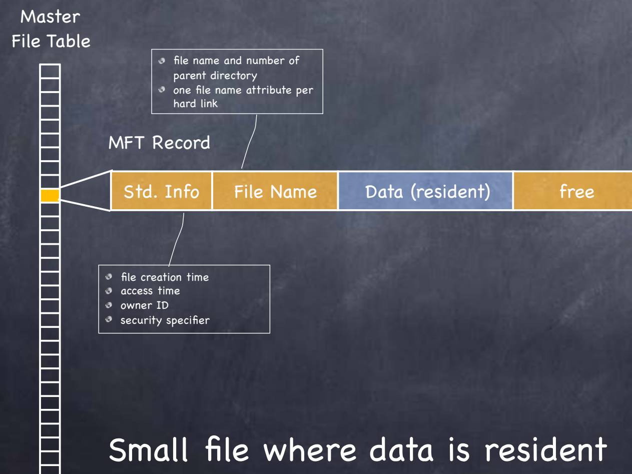
Microsoft, mid 90s

- Index structure: extents and flexible tree
  - extents
    - track ranges of contiguous blocks rather than single blocks
  - flexible tree
    - file represented by variable depth tree
      - large file with few extents can be stored in a shallow tree
  - MFT (Master File Table)
    - array of 1 KB records holding the trees' roots
    - similar to inode table (but one file can have multiple MFT entries)
    - each record stores sequence of variable-sized attribute records
      - both data and metadata are attributes
      - attributes can be resident (fit in the record) or nonresident

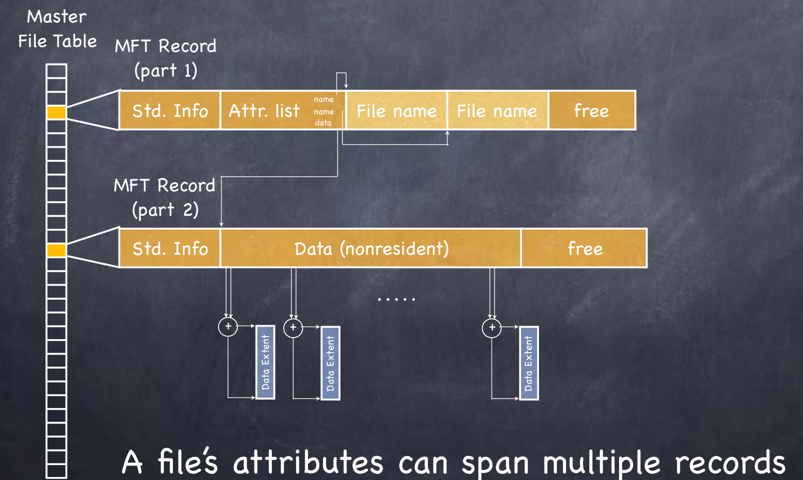
Example of NTFS index structure



Example of NTFS index structure

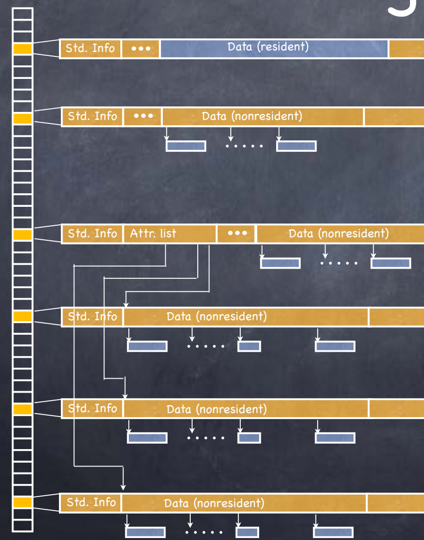


Example of NTFS Index Structure



Small, Normal, and Big Files

Master File Table



...and for really huge (or really badly fragmented) files, even the attribute list can become nonresident!

- atribute list split in separate extents