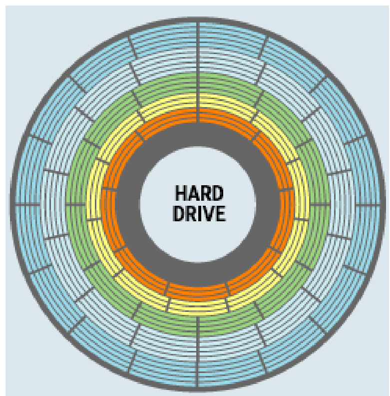
Filesystem Implementation

Introduction

Concepts

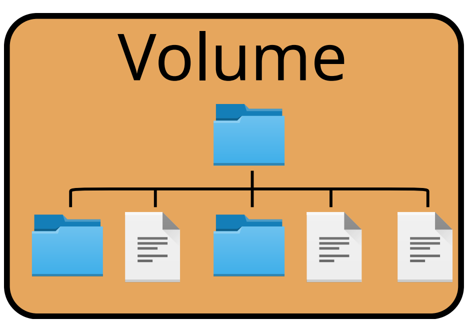
Volume

- Disk, or partition on a disk
- Large array of data blocks



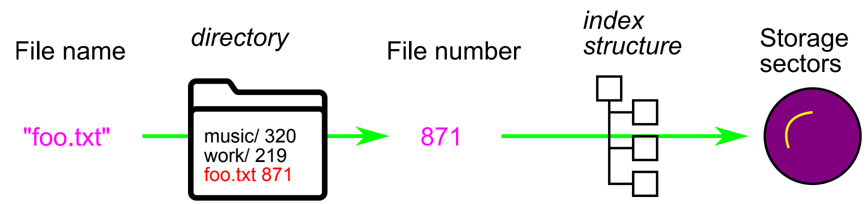
Filesystem

- Methods and data structures to organize files on a volume



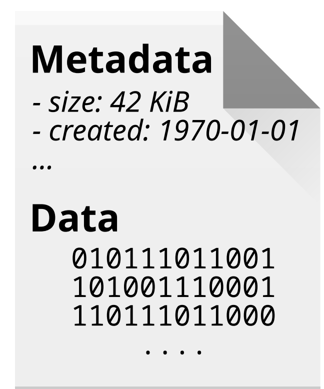
Directory

- Hierarchy of named files



File

- Metadata to describe file's characteristics
- Actual sequence of data



Introduction

Objectives

Performance

- In spite of underlying storage device's limitations
- Achieved by maintaining *spatial locality*
    - Blocks that are logically related should be stored near one another

Flexibility

- Handle all file sizes: small vs large
- Handle all access types: sequential vs random
- Handle all access frequencies: rare vs frequent
- Handle all file lifetimes: temporary vs permanent

Persistence

- Maintain both user data and internal data structures
- Survive system crashes and power failures

Reliability

- Store data reliability over time
- In spite of crash during updates, or hardware errors

Design considerations

Workload

File size and storage space

```
$ du -sh /usr/bin/
1.1G	/usr/bin/
$ ls -1 /usr/bin/ | wc -l
4566
$ du -h VirtualBox/Machines/Ubuntu/Ubuntu.vdi
20G	VirtualBox/Machines/Ubuntu/Ubuntu.vdi
```

- Most files are small
- Large files account for more storage

File access and I/O transfer

```
$ strace chrome |& grep "open" | wc -l
557
$ dd if=Ubuntu.vdi of=copy.vdi bs=4K
...
20981043200 bytes (21 GB, 20 GiB) copied, 62.74 s, 334 MB/s
```

- Most accesses are to small files
- Accesses to large file account for more I/O transfer

File access pattern and usage

- Most files are read/written sequentially (e.g., config files, executables)
- Some files are read/written randomly (e.g., database files, swap files)

- Some files have pre-defined size at creation (e.g., downloaded files)
- Some files start small and grow over time (e.g., system logs)

Design considerations

Blocks vs disk sectors

Rationale

- OS can allocate blocks of disk sectors rather than individual sectors
  - Does not cost much more to access few consecutive disk sectors

Big block size

- E.g., 32 KiB
- Management requires less space
- Performance improvement
- Wasted space if block not full

Small block size

- E.g., size of single sector
- Management requires more space
- More separate accesses to data
- Less wasted space if block not full

Trade-off

- Make block size multiple of memory page size
- 4KiB on most systems

Design considerations

Review

Small files

- Small blocks for efficient storage
- Files used together should be stored together

Large files

- Large blocks for efficient storage
- Contiguous data allocation for sequential access
- Efficient lookup for random access

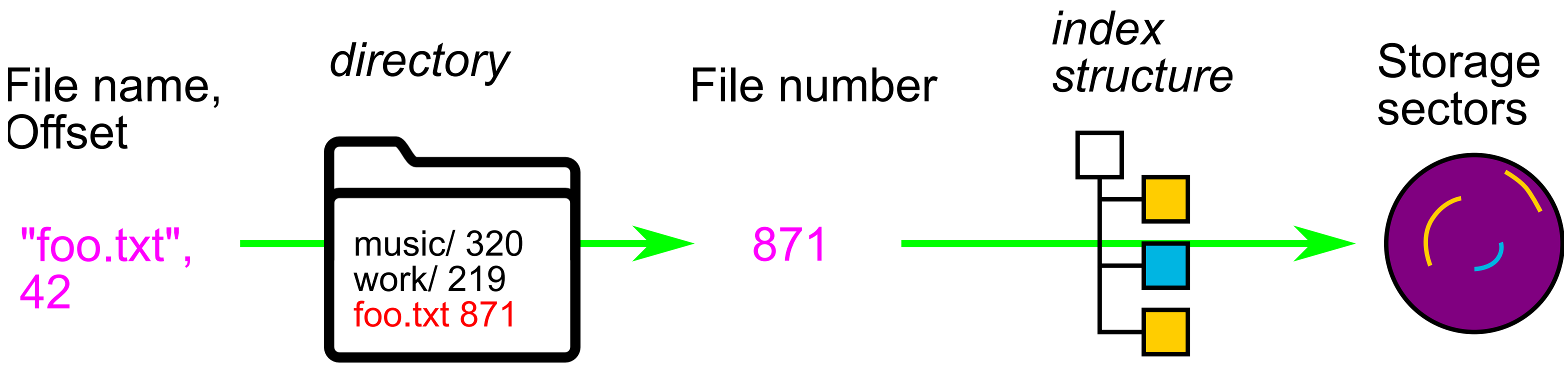
Problems

- May not know at file creation
  - Whether file will become small or large
  - Whether file is persistent or temporary
  - Whether file will be used sequentially or randomly

Design considerations

Implementation overview

- From pair `<filename, offset>`, find physical storage block *efficiently*



Data structures

- Directories
  - Map filenames to file numbers (to find metadata)
- Index structure
  - Part of a file's metadata
  - Map data blocks of file
- Free space map
  - Manage the list of free disk blocks
  - Allow files to grow/shrink

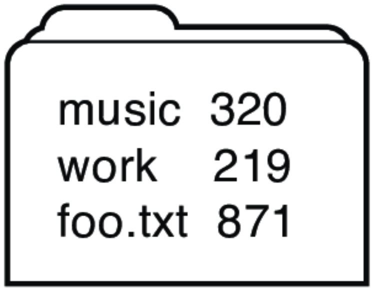
Data structures organization

- Storage devices often have non-uniform performance
- Use of *locality heuristics* to optimize data placement
  - Defragmentation
  - Files grouping

Directories

Design

- A directory is simply a **file**
  - List of mappings from filenames to file numbers
  - Each mapping is a *directory entry*
    - <name, file number>
- Only directly accessible by OS
  - Ensure integrity of mapping
  - Accessible for processes via *syscalls*
    - e.g., `opendir()/readdir()`




```
00002000: 6673 5f6d 616b 652e 6300 0000 0000 0000  fs_make.c.......
00002010: 3305 0000 0100 0000 0000 0000 0000 0000  3...............
00002020: 7465 7374 5f66 732e 6300 0000 0000 0000  test_fs.c.......
00002030: 152c 0000 0200 0000 0000 0000 0000 0000  .,..............
00002040: 6772 6164 652e 7368 0000 0000 0000 0000  grade.sh........
00002050: a958 0000 0500 0000 0000 0000 0000 0000  .X..............
00002060: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00002070: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Hexdump of directory

Directories

Organization strategies

Flat hierarchy

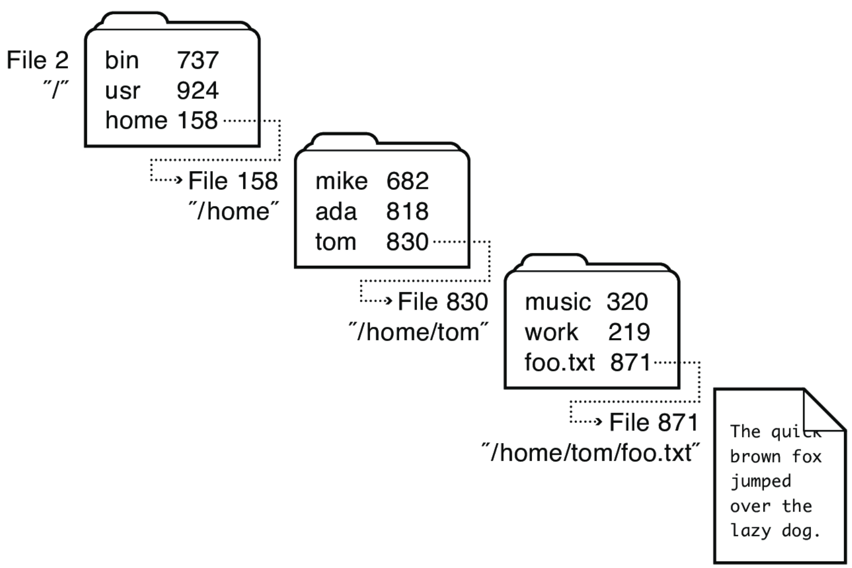
- One unique namespace for the entire volume
  - Use special area of the disk to hold root directory
  - Two files can never be named the same

Multi-user flat hierarchy

- Separate root directory for each user
- But all user's files must still have unique names...

Multi-user, Multi-level hierarchy

- One special root directory
- Hierarchy of subdirectories
- Permissions to distinguish between users

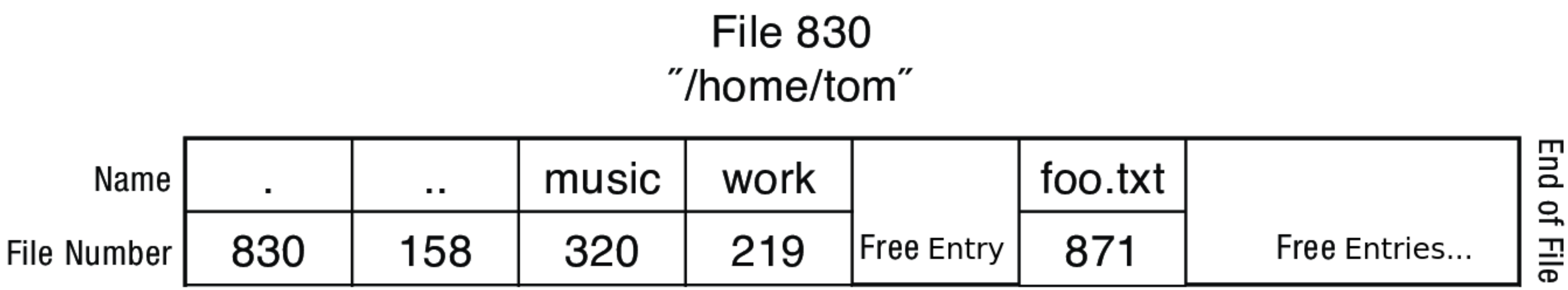


Directories

Implementation

Linear layout

- Simple array of entries
- E.g., MS-FAT



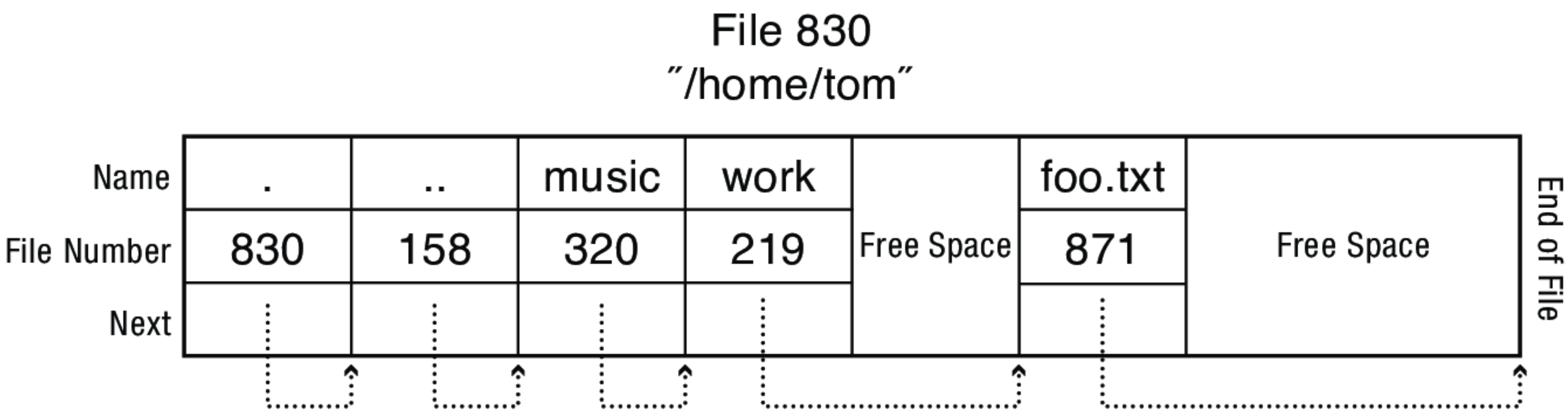
Pros/Cons

- Simple
- Need to scan all entries

Directories

List layout

- Linked-list of entries
- E.g., ext2



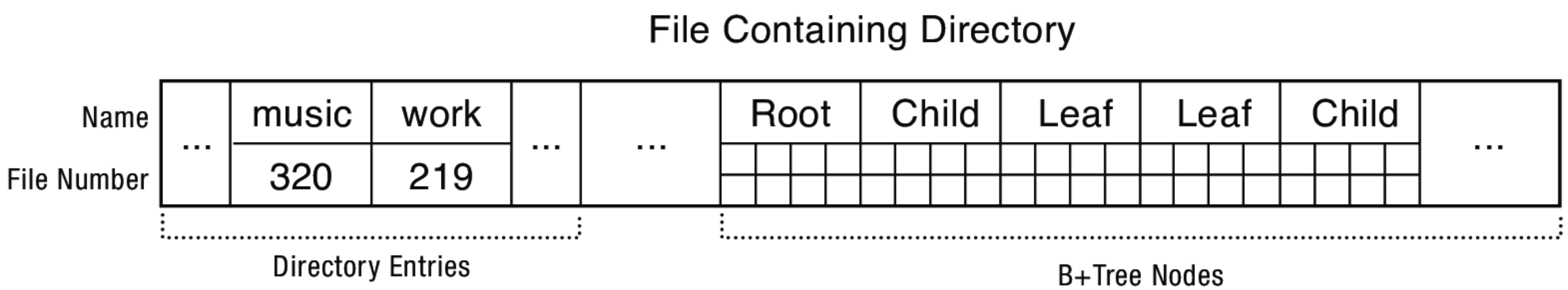
Pros/Cons

- Jump over blocks of free entries
- Linear traversal

Directories

Tree layout

- Tree of entries
    - Filenames hashed into keys used to traverse tree
- E.g., XFS, NTFS



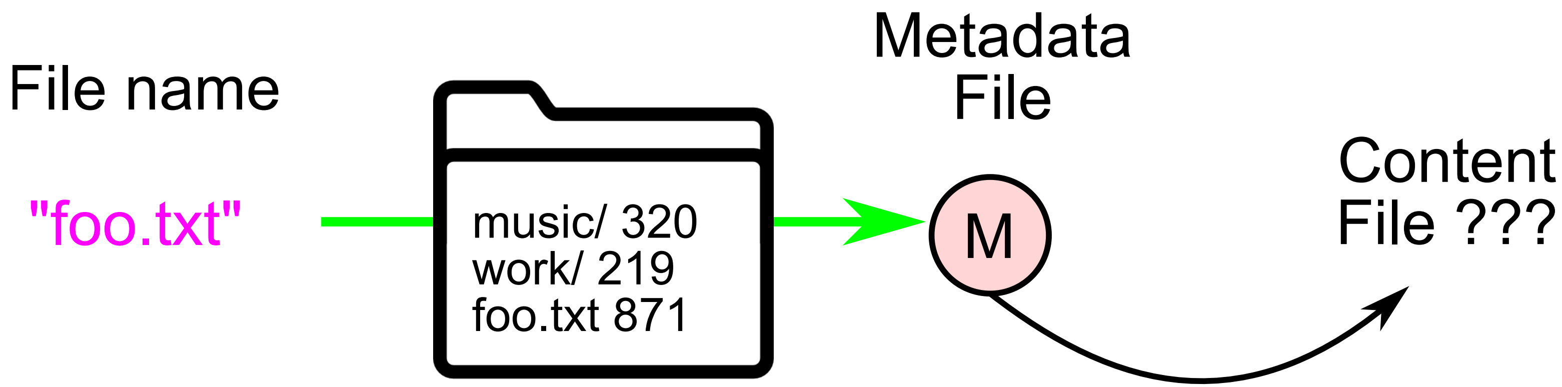
Pros/Cons

- Fast search
- More complicated

Index structures

Metadata to data

- From directory mapping, find metadata
- From metadata, find file's contents



Goals

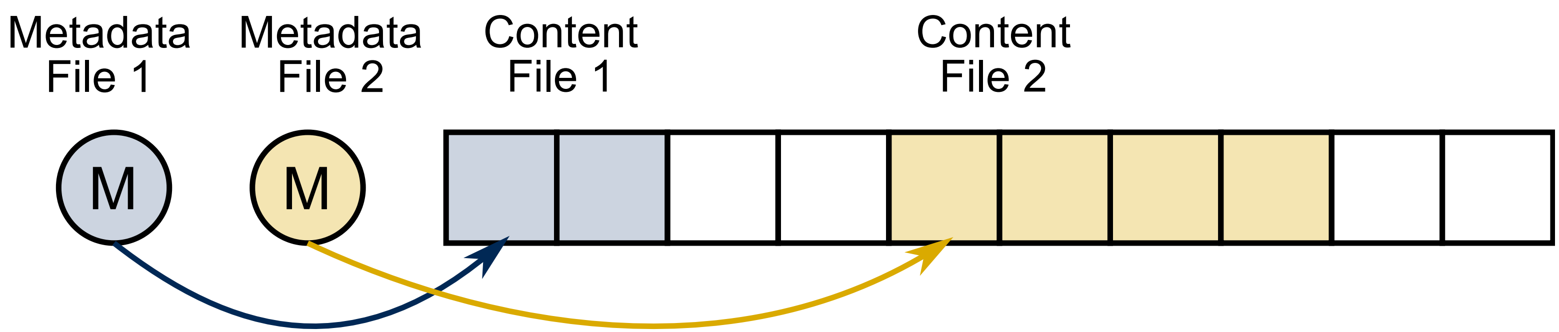
- Support sequential data placement to maximize sequential file access
- Provide efficient random access to any file block
- Limit overheads to be efficient for small files
- Be scalable to support large files
- Provide space to hold metadata itself

Index structures

Contiguous allocation

- Files stored as a sequence of contiguous blocks
- File-to-blocks mapping includes first block (and size)
- Require allocation policy (e.g., first-fit, best-fit, worst-fit)

Example



Pros

- Very simple
- Best performance
- Efficient sequential and random access

Cons

- Change in size likely to require entire reallocation
- External fragmentation

Usage

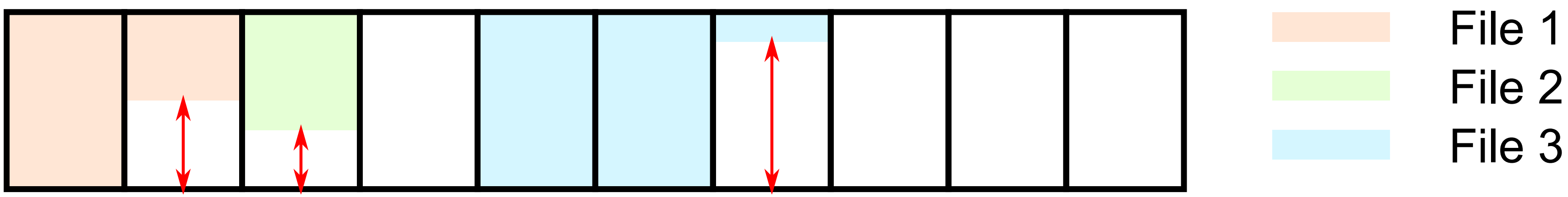
- ISO 9660 (CD-ROM, DVD, BD)

Index structures

Digression about fragmentation

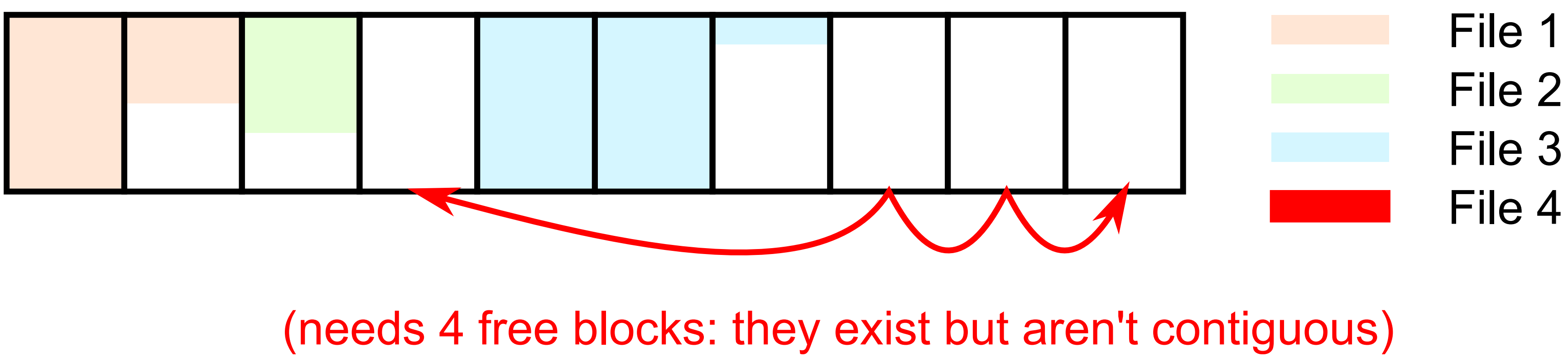
Internal fragmentation

- Waster space inside blocks
- Issue if blocks are too large



External fragmentation

- Free space scattered instead of being contiguous
- Issue if blocks need to be allocated contiguously

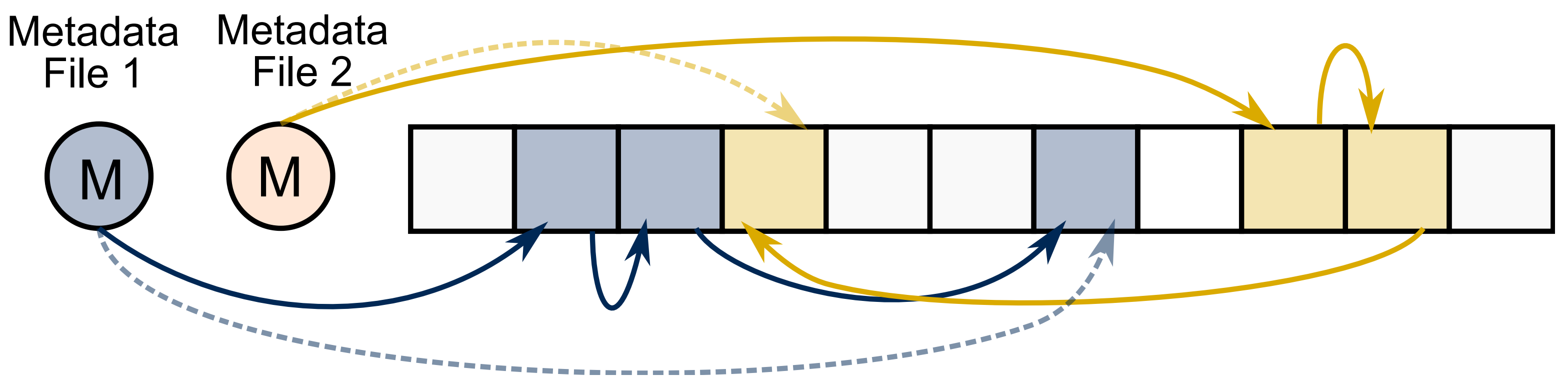


Index structures

Linked-list allocation

- Files stored as linked lists of blocks
- File-to-blocks mapping includes pointer to the first block
  - Pointer to the last block to optimize file growth
  - For each block, pointer to the next block in chain

Example



Pros

- Fairly simple
- File size flexibility
- No external fragmentation
- Easy sequential access

Cons

- No (true) random access
- Potentially inefficient sequential access

Usage

- MS-FAT

ECS 150 - Filesystem Implementation

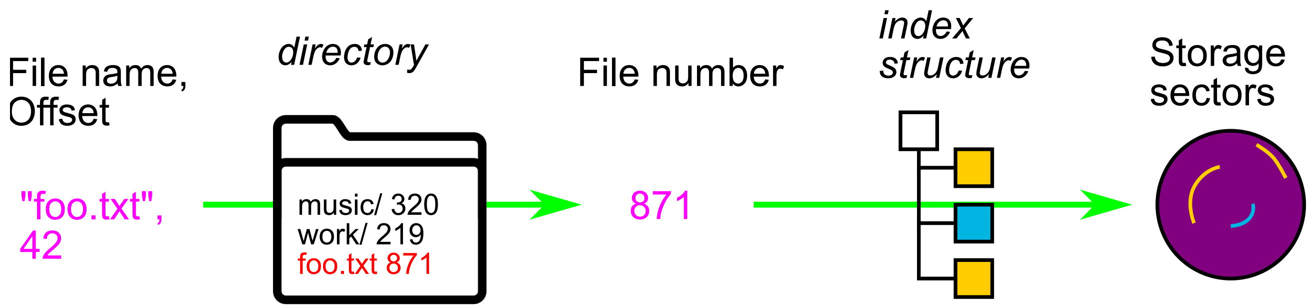
Prof. Joël Porquet-Lupine

UC Davis - 2020/2021

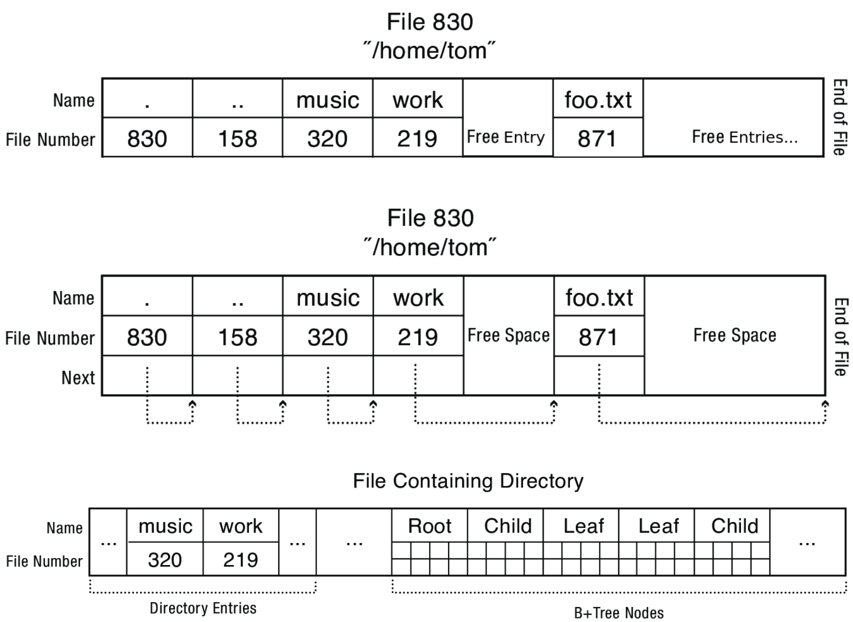
Recap

Implementation overview

- From pair `<filename, offset>`, find physical storage block *efficiently*

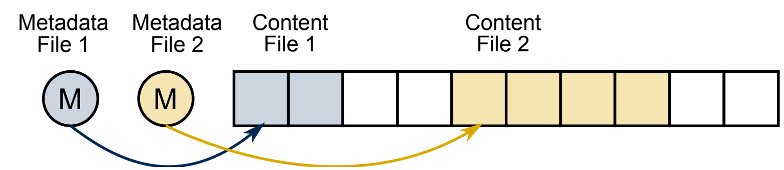


Directories

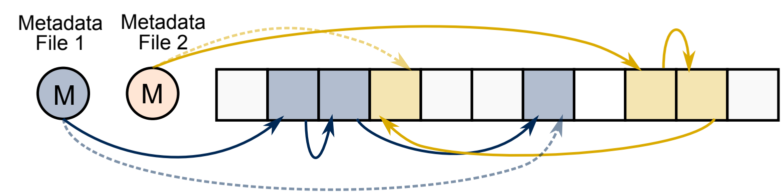


Index structures

- Contiguous allocation



- Linked-list allocation

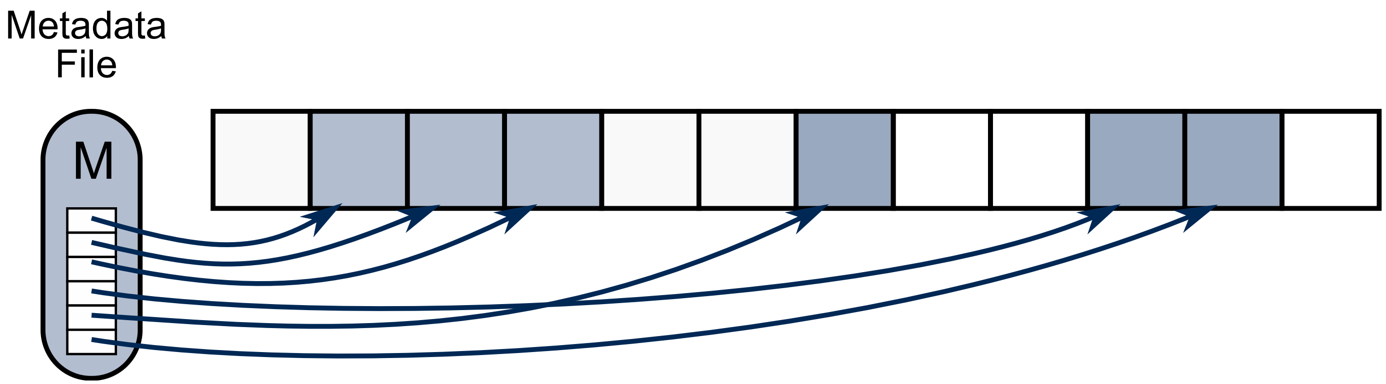


Index structures

Direct allocation

- File-to-blocks mapping includes direct pointers to each data block

Example



Pros

- File size flexibility
- Supports true random access

Cons

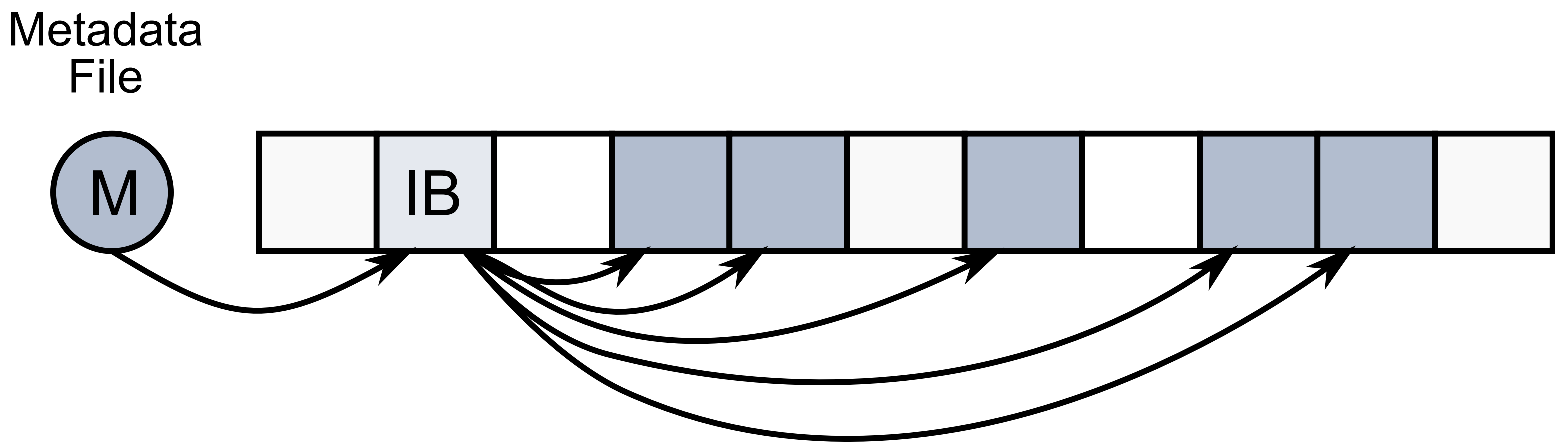
- Limited file size
- Non-scalable index structure

Index structures

Indexed allocation

- File-to-blocks mapping includes a pointer to an *index block*
  - An index block contains an array of data block pointers
  - Data blocks allocated only on demand

Example



Pros

- Same as direct allocation
- Decouple index structure from metadata

Cons

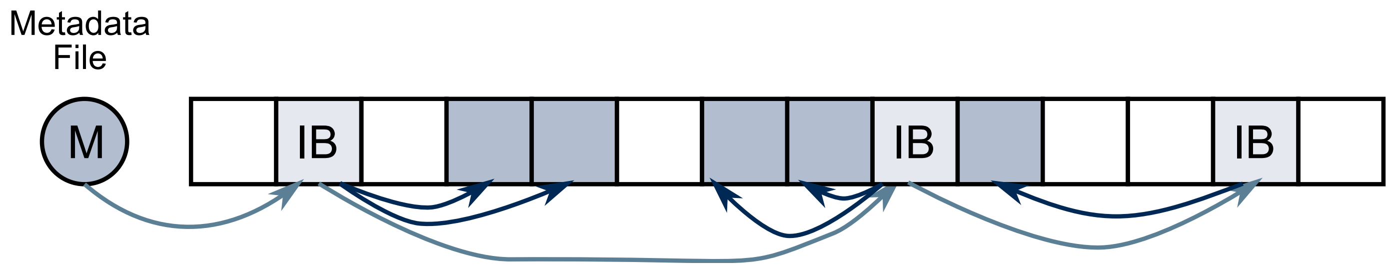
- Same as indexed allocation
- Overhead for small files

Index structures

Linked index blocks (IB + IB + ...)

- Last index block's pointer can point to next index block

Example



Pros

- File size flexibility

Cons

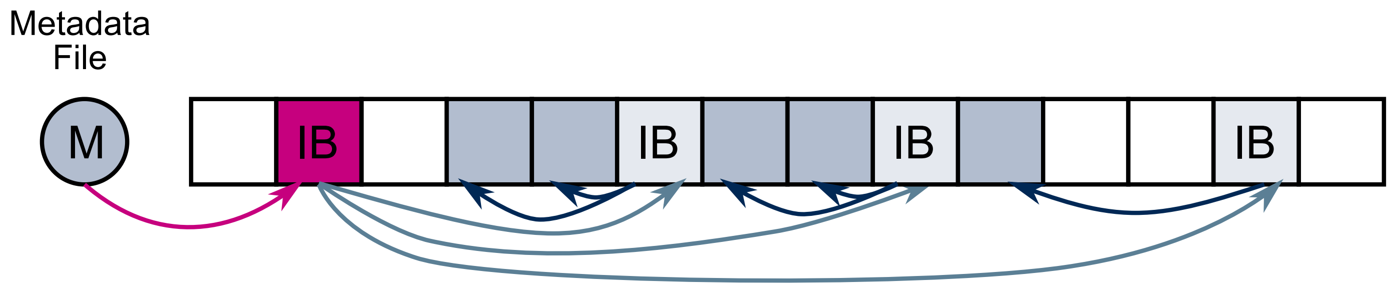
- Traversal for very large files

Index structures

Multilevel index blocks (IB x IB x ...)

- First-level index block points onto second-level index blocks

Example



Pros

- Great support for very large files

Cons

- Wasteful for small files

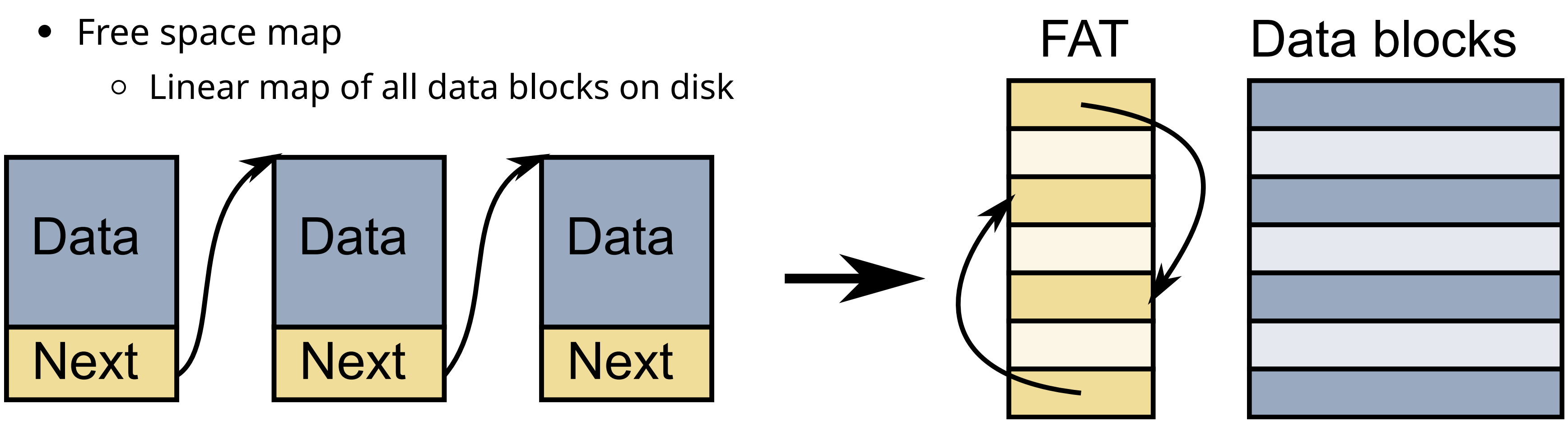
Case study: MS-FAT

Introduction

- Microsoft **File Allocation Table**
- Originally created for floppy disks, in the late 70s
  - Used on MS-DOS, and early version of Windows (before NTFS)
  - Still very popular on certain systems (e.g., thumb drives, camera SD-cards, embedded systems, etc.)
- Different versions over time: FAT12, FAT16, FAT32, and now exFAT

File Allocation Table

- Index structure for files
  - Each file is a linked list of blocks
- Free space map
  - Linear map of all data blocks on disk



Case study: MS-FAT

FAT structure

- 1 entry per data block

Index structures

- Directory entry maps name to first block index

File	Size	Index
foo.txt	18000	9
bar.txt	5000	12

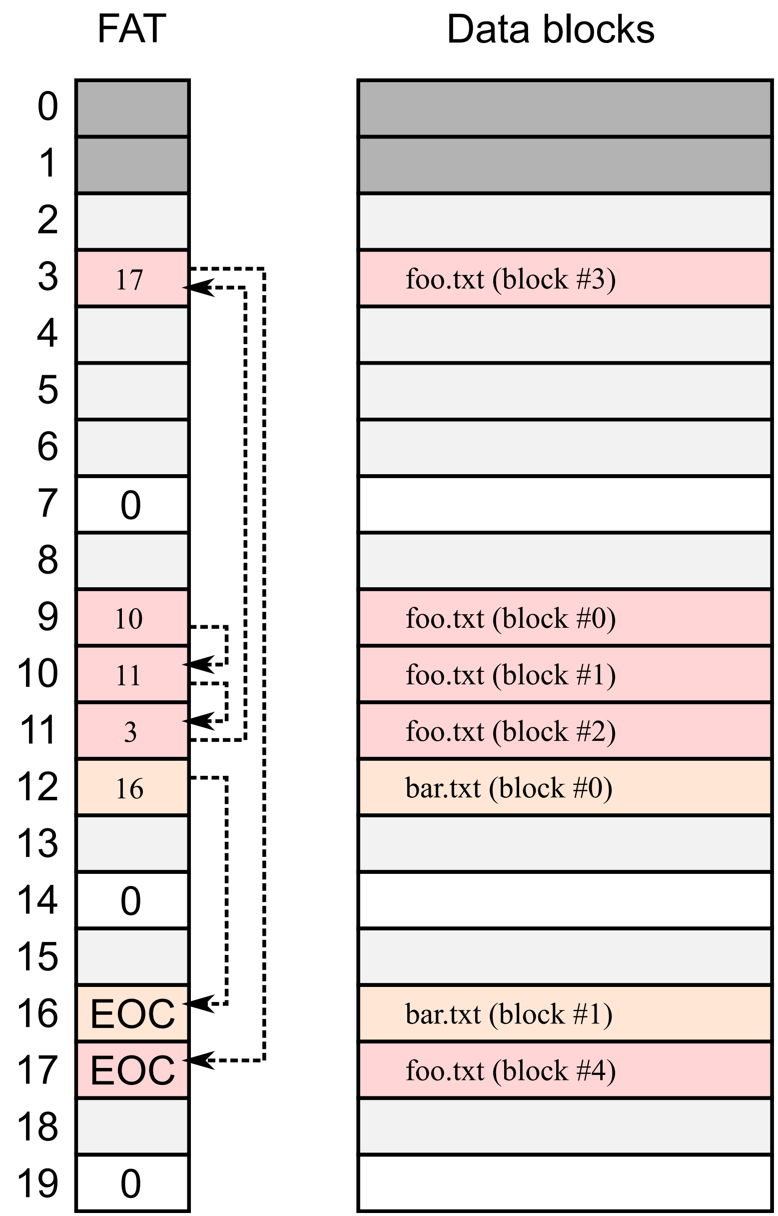
- Indicates next block in chain
  - or EOC for last block of a file

Free space tracking

- 0 indicates free block

Locality heuristics

- Usually simple allocation strategy (e.g. *next-fit*)



Case study: MS-FAT

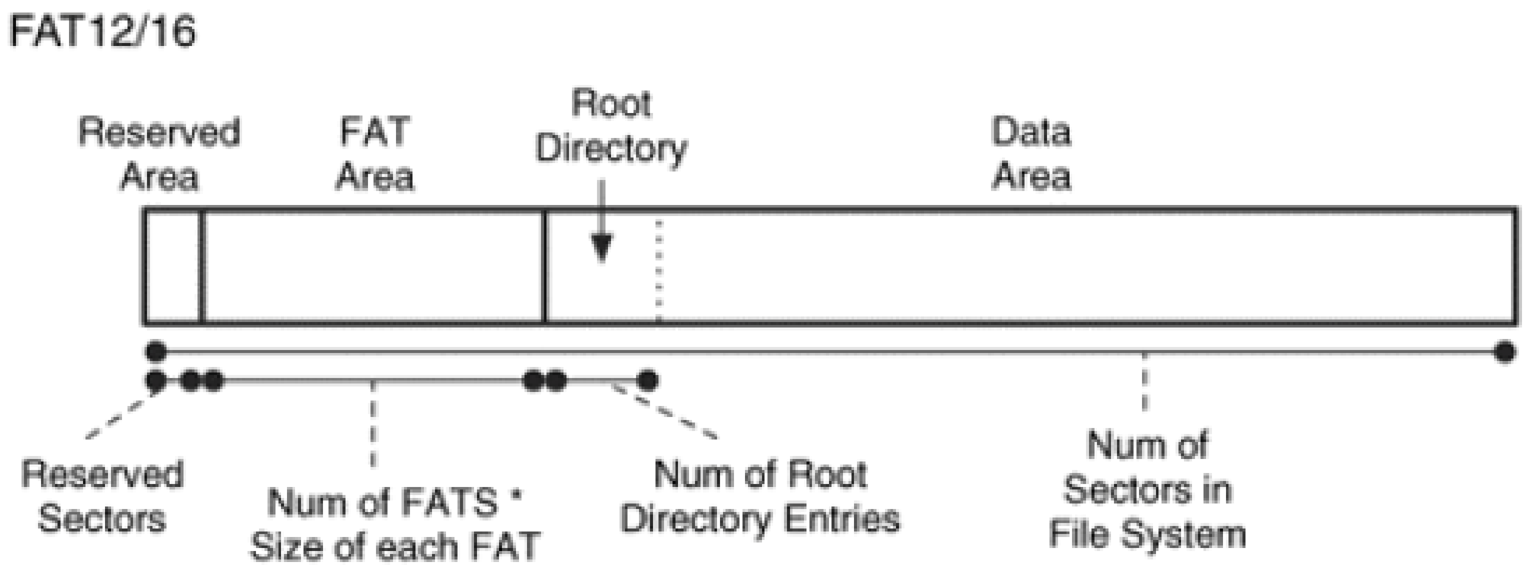
Directory structure

- Directory is a file containing an array of 32-byte entries.
- Each entry composed of
  - 8-byte name + 3-byte extension (ASCII)
    - *Long file names were later supported by allowing to chain multiple directory entries*
  - Creation date and time
  - Last modification date and time
  - Index of first data block in FAT
  - Size of the file

<table>
<tr><td>T</td><td>H</td><td>E</td><td>Q</td><td>U</td><td>I</td><td>~</td><td>1</td><td>F</td><td>O</td><td>X</td><td>0x20</td><td>NT</td><td colspan="3">Create time</td></tr>
<tr><td colspan="2">Create date</td><td colspan="2">Last access date</td><td colspan="2">0x0000</td><td colspan="2">Last modified time</td><td colspan="2">Last modified date</td><td colspan="2">First cluster</td><td colspan="4">File size</td></tr>
</table>

Case study: MS-FAT

Layout on disk

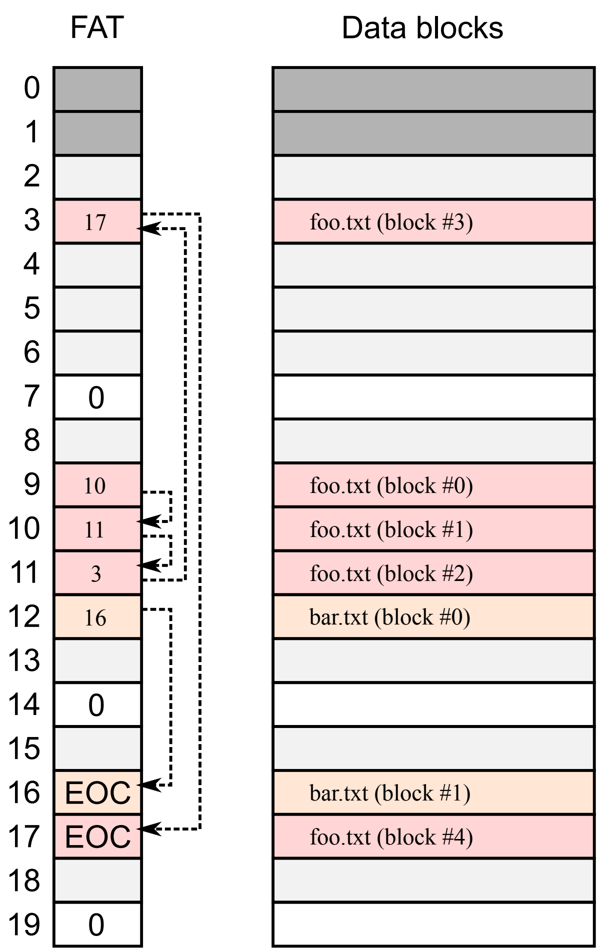


Case study: MS-FAT

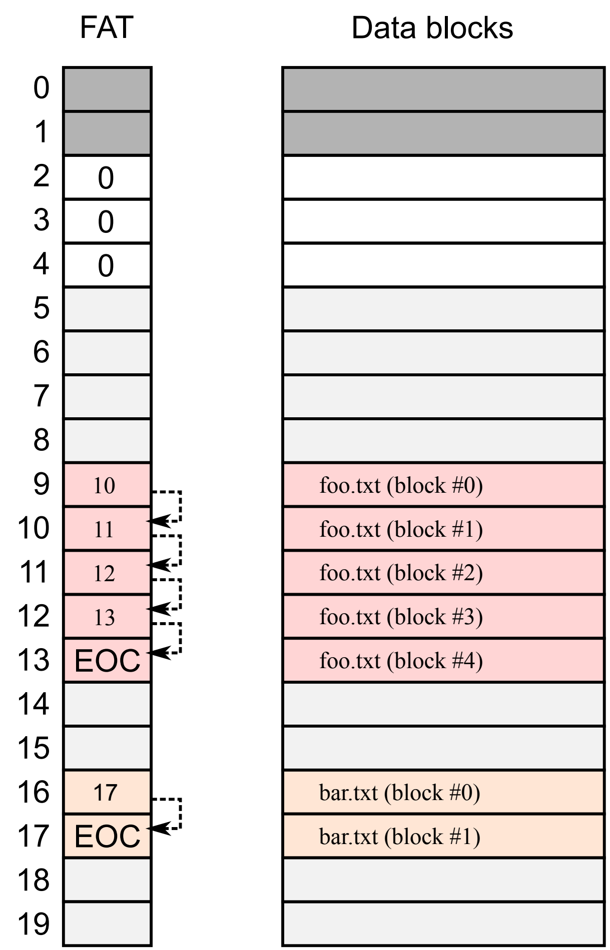
Locality heuristics

- Data blocks for a file may be scattered across the disk
- *Defragmentation* can rearrange data blocks and improve *spatial locality*

Before



After



Case study: MS-FAT

Conclusion

Pros

- Simple
  - State required per file: start block and size
- Widely supported (maybe even the most popular FS ever!)
- No external fragmentation

Cons

- Limited performance
  - Many seeks if FAT cannot be cached in memory
  - Poor locality for sequential access if files are fragmented
  - Poor random access
- Limited metadata
  - No access control
  - No support for hard links
- Limited volume and file sizes
  - E.g., 2-TiB max volume and 4-GiB max file size with FAT32
- No support for reliability strategies

ECS 150 - Filesystem Implementation

Prof. Joël Porquet-Lupine

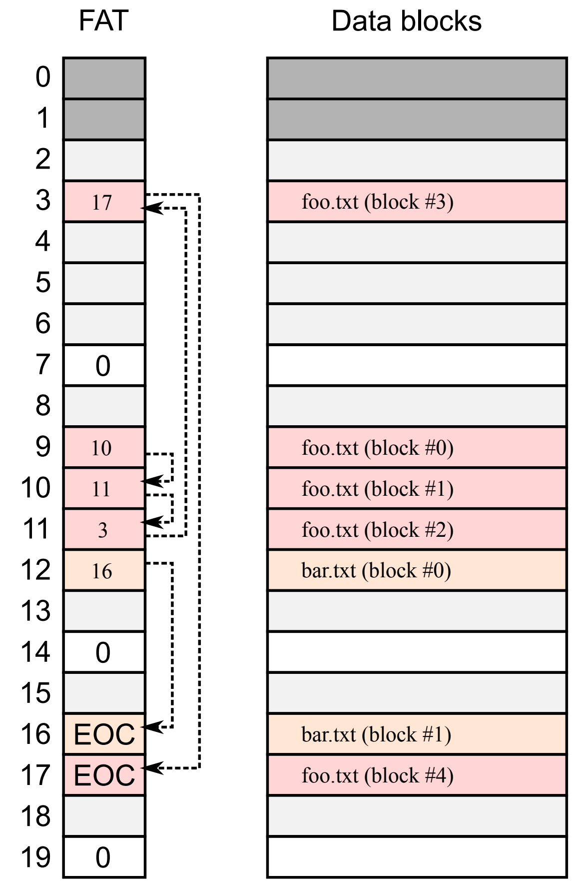
UC Davis - 2020/2021

Recap

MS-FAT

Design

- Index structure
  - Linked list
  - Implemented via FAT (File Allocation Table)
  - `FAT[cur_block] == next_block`
- Free space
  - Via FAT as well
  - `FAT[i] == 0`
- Locality heuristics
  - *Next fit* for FAT allocation
  - Data block defragmentation
    - Optimize sequential layout



Discussion

- Very simple, still widely used and supported
- Poor locality, poor random access, limited metadata

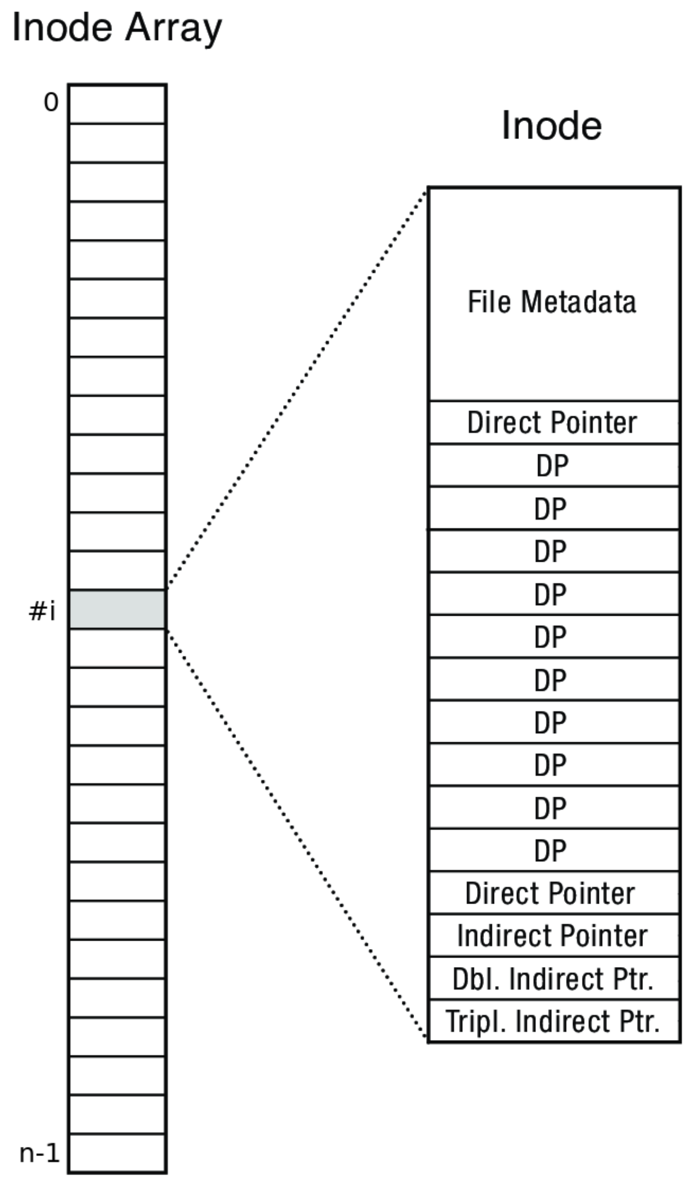
Case study: FFS

Introduction

- Berkeley **Fast File System**
- Originally created as improvement of UFS (Unix File System), in the early 80s
- Inspiration for the ext2/3/4 family

Inodes

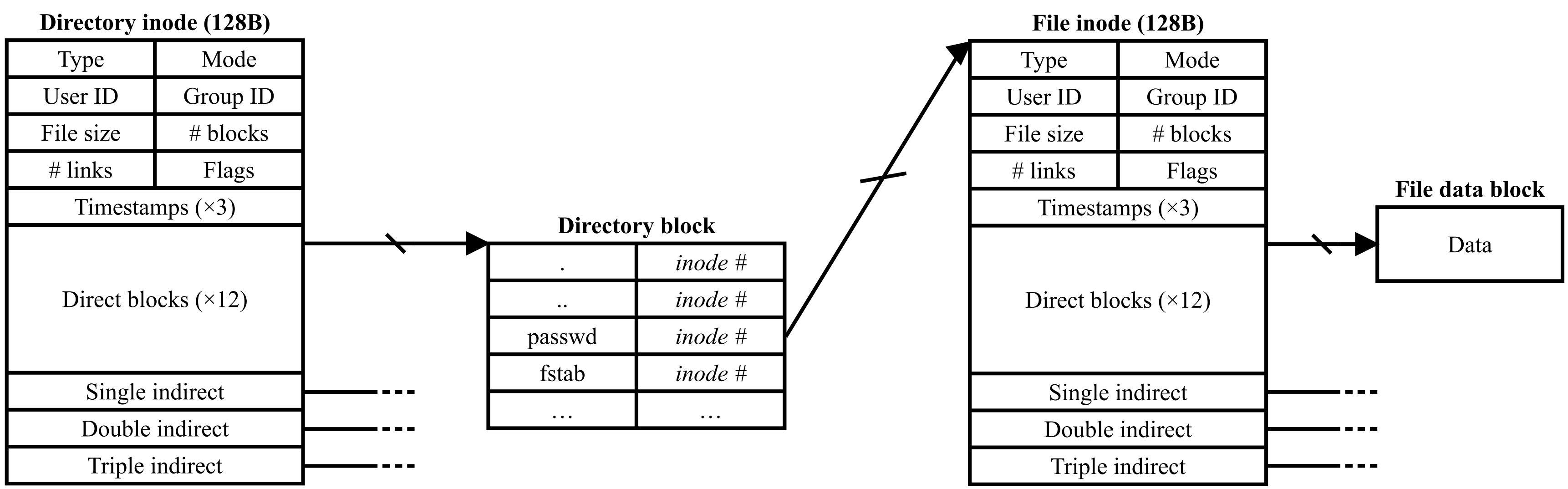
- Inode contains file's metadata and a set of pointers to locate its data blocks
- Index structure as combination of all the indexed-based approaches
  - File represented as a fixed, asymmetric tree, with 4-KiB data blocks as leaves
- Inodes are stored consecutively in a big array, and indexed via an *i-number*



Case study: FFS

Files' metadata

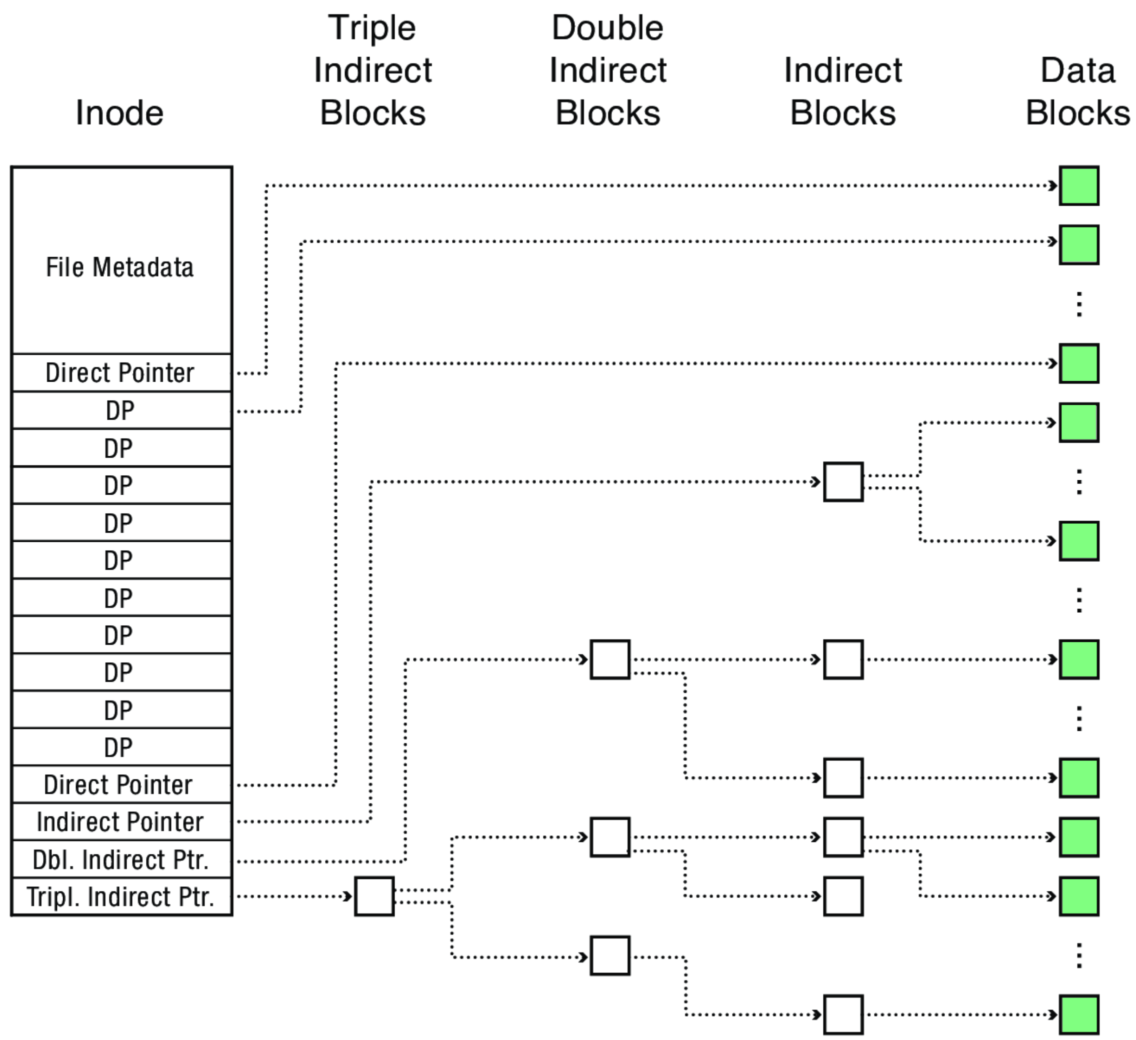
- Type:
    - Ordinary file
    - Directory
    - Symbolic link
    - Etc.
- Permissions and owners
- Size in bytes
- Number of (hard-)links to the inode
- Timestamps:
    - Created, last modified, last accessed



Case study: FFS

Files' index structure

- Fixed, asymmetrical tree index structure (**Multilevel index**)
  - Combination of: *direct*, *indexed* and *multilevel indexed* allocation



Case study: FFS

Characteristics

Tree structure

- File represented as a tree
- Efficient to find any data block (e.g., random access)

High degree

- Each *indirect block* points to 100s of blocks
- Minimize number of seeks

Fixed structure

- Byte *n* of a file always accessible via the same pointer(s)
- Simple to implement

Asymmetric

- Not all data blocks at the same level
- Efficiently supports small and large files

Case study: FFS

Flexible file size

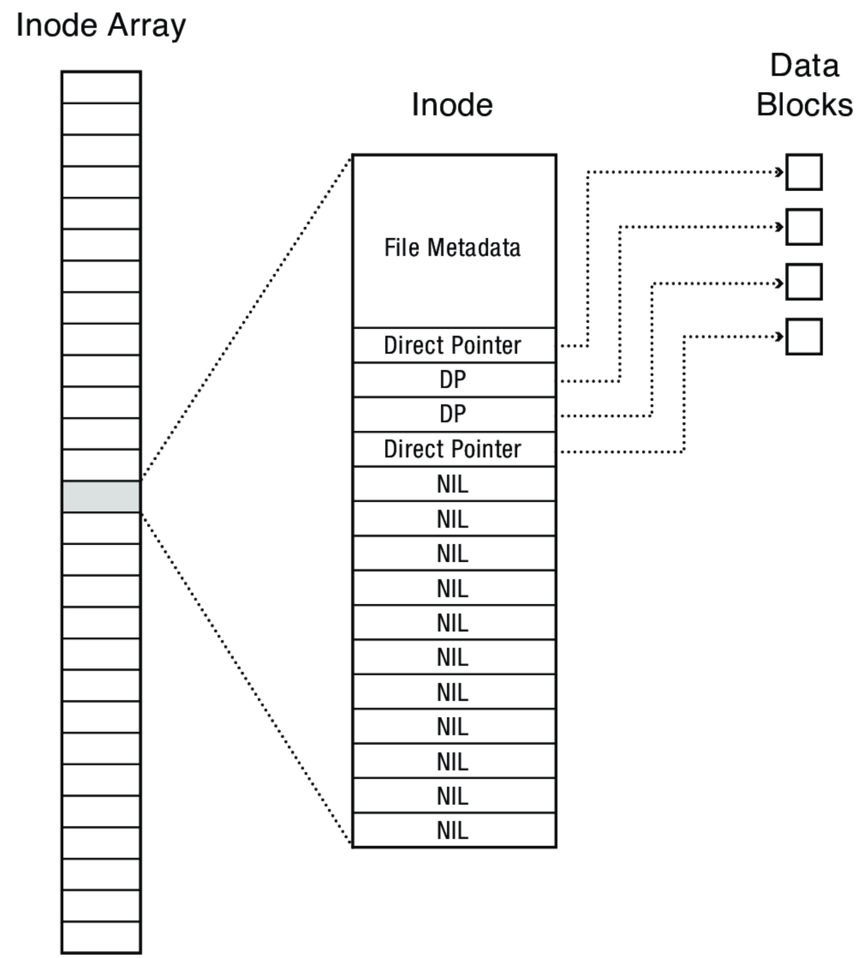
- 15 pointers per inode
  - 12 *direct* pointers to data blocks
    - With 4 KiB data blocks: max size of 48 KiB
  - 1 pointer to a block of 1024 direct pointers to data blocks (*single indirection*)
    - With 4 KiB data blocks: 4 MiB (+ 48 KiB)
  - 1 pointer to a block of pointers to blocks of 1024 direct pointers (*double indirection*)
    - With 4 KiB data blocks: 4 GiB (+ 4 MiB + 48 KiB)
  - 1 pointer to a block of pointers to blocks of pointers to blocks of 1024 direct pointers (*triple indirection*)
    - With 4 KiB data blocks: 4 TiB (+ 4 GiB + 4 MiB + 48 KiB)
- In total, the structure can point to: 12 + 1024 + 1024^2 + 1024^3 blocks *

* In practice, limited to 2 TiB

Case study: FFS

Small files support

- All the blocks are reached via direct pointers
- Two accesses to read data
  - Inode + data block



Fixed-depth tree?

- With a fixed 3-level tree (instead of asymmetric tree)
- A 4 KiB file would consume ~16 KiB!
  - 4 KiB data + 3 levels of 4 KiB indirect blocks
  - 5 accesses to read data
    - Inode + 3 indirection blocks + data block

Case study: FFS

Sparse files support

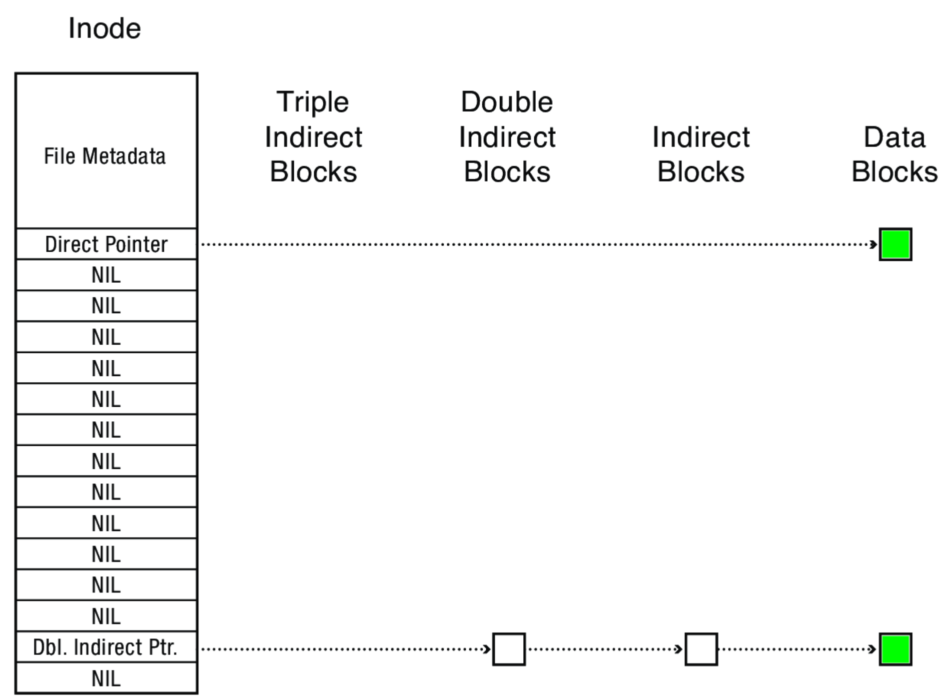
- Sparse files contain large "empty" areas (*holes*)

```
fd = open("/path/to/sql.db", O_RDWR|O_CREAT, 0644);
ftruncate(fd, 1 << 30);            // Grow the file to 1GiB

write(fd, buf, sizeof(buf));       // Write something at the beginning
lseek(fd, -sizeof(buf), SEEK_END);
write(fd, buf, sizeof(buf));       // Write something at the very end
```

Efficient support

- Read from *hole*
  - 0-filled buffer
- Write to *hole*
  - Data blocks and indirect blocks dynamically allocated
- Example (above)
  - 2 writes: 4 KiB at offset 0 and 4 KiB at offset 2^{30}
  - File size of 1.1 GiB
  - But space on disk of 16 KiB!



Case study: FFS

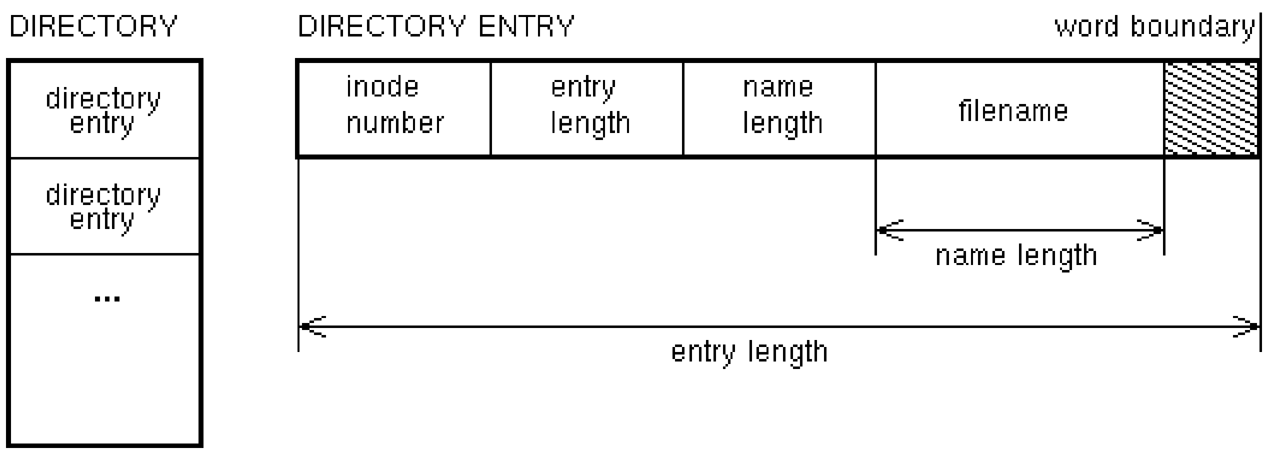
Directory structure

Entry structure

- Originally, array of 16-byte entries
  - 14 bytes for file name
  - 2 bytes for i-number
- Later, linked list in which each entry contains
  - 4 bytes for i-number
  - Length of file name
  - Variable-length file name

Directory contents

- First entry always `.`
  - Points to self
- Second entry always `..`
  - Points to parent's i-number



Case study: FFS

Free space management

- Need to keep track of which inode entries and data blocks are free
- Use of bitmaps

Bitmap data structure

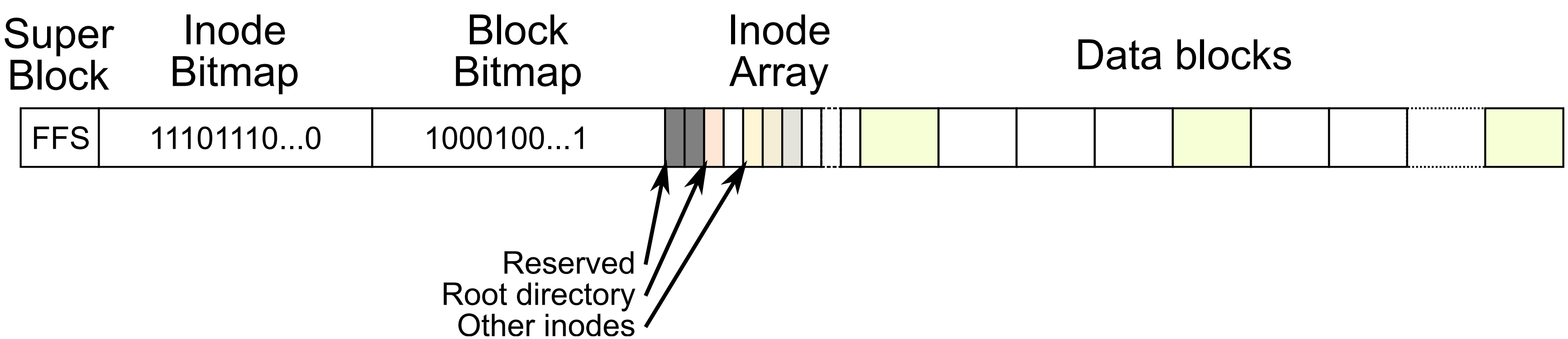
- Keeping track of *n* resources requires a bitmap of *n* bits.
- Each bit in the bitmap tracks a single resource
  - `0` if resource is free
  - `1` if resource is allocated

011100001
110101101
000101001
100100010
...
111100010

Case study: FFS

Layout on disk

- Superblock
  - Information about the file system volume (type, size, etc.)
- Inode bitmap
- Data block bitmap
- Inode array
  - Inode #0 and #1 are reserved
  - Inode #2 is always the root directory
- Data blocks (includes actual file data blocks, but also directory contents and indirection blocks)

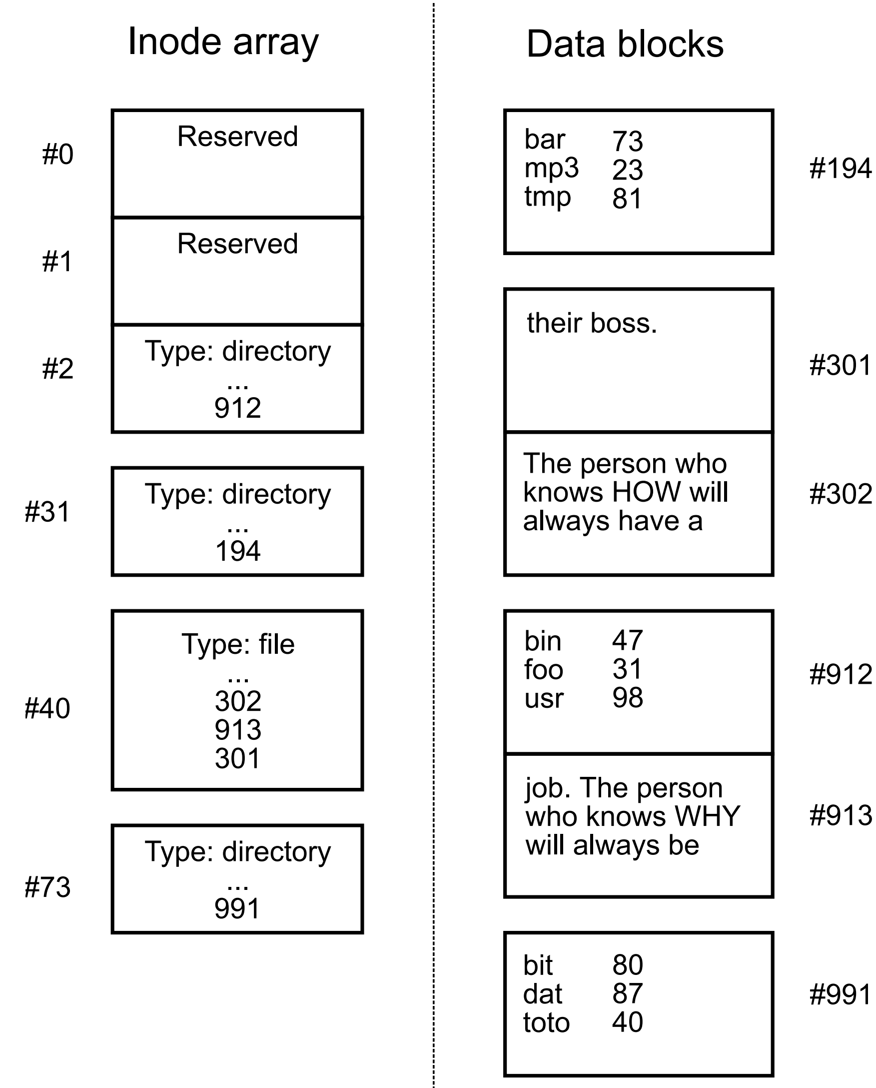


Case study: FFS

Example: reading a file

```
fd = open("/foo/bar/toto");
read(fd, buf, len);
```

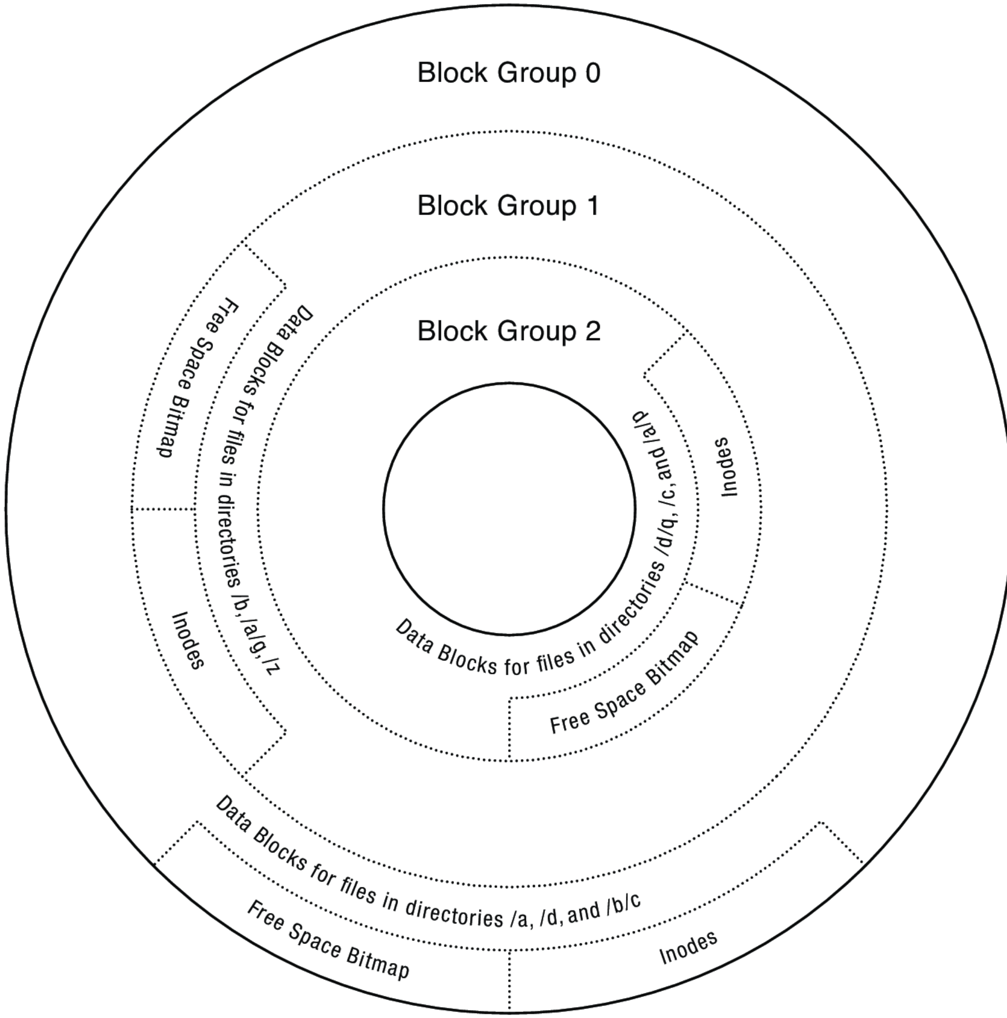
1. Inode #2 (`/`, root directory)
    - Find root directory's data block (912)
2. Browse root directory's content
    - Find `foo`'s i-number (31)
3. Inode #31
    - Find `foo` directory's data block (194)
4. Browse `foo` directory's content
    - Find `bar`'s i-number (73)
5. Inode #73
    - Find `bar` directory's data block (991)
6. Browse `bar` directory's content
    - Find `toto`'s i-number (40)
7. Inode #40
    - Find `toto` file's data block (302, 913, 301)
8. Read data blocks



Case study: FFS

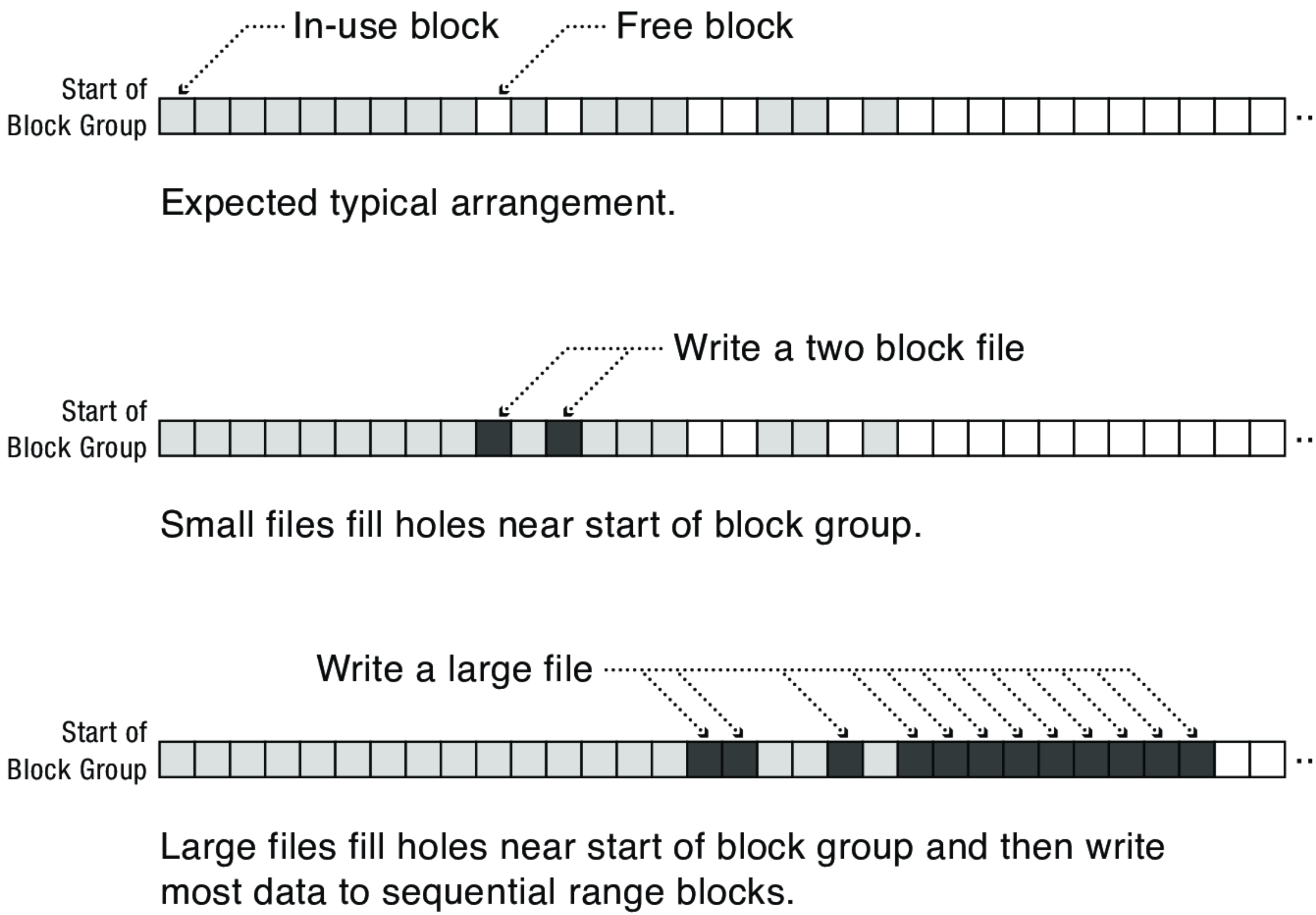
Locality heuristics: block groups

- Divide volume into **block groups**
  - Sets of consecutive cylinders
  - Seek time between blocks in a group is small
- Distribute metadata
  - Distribute into block groups
  - File's metadata close to its data
- File placement
  - Files belonging to same directory in same group
  - New directory in different group than parent's directory
- Data block placement
  - First-fit strategy
  - Short-term vs long-term locality



Case study: FFS

First-fit data placement



Case study: FFS

Locality heuristics: reserved space

- Block group heuristic is efficient but needs significant amount of free space
    - If volume is near full, little room for locality optimization
- FFS reserves a fraction of volume's space (~10%)
    - Treats volume as full before it actually is
    - Leaves room for locality optimization

- Choice motivated by (disk) technology trends
    - Sacrifices disk space
        - But known to steadily increase
    - To reduce seek times
        - Known to only improve very slowly

Case study: FFS

Conclusion

Pros

- Efficient storage for both small and large files
- Locality for both small and large files
- Locality for metadata and data

Cons

- Inefficient for tiny files
  - e.g., 1-byte file requires both an inode and a data block
  - Optimization possible for symbolic links (ext family)
- Inefficient encoding when a file is mostly contiguous on disk
- Need to reserve fraction of free space to optimize locality