File Systems

Chapter 11, 13 OSPP

What is a File?

What is a Directory?

Goals of File System

- Performance
- Controlled Sharing
- Convenience: naming
- Reliability

File System Workload

- File sizes
  - Are most files small or large?
  - Which accounts for more total storage: small or large files?

File System Workload

- File access
  - Are most accesses to small or large files?
  - Which accounts for more total I/O bytes: small or large files?

File System Workload

- How are files used?
  - Most files are read/written sequentially
  - Some files are read/written randomly
    - Ex: database files, swap files
  - Some files have a pre-defined size at creation
  - Some files start small and grow over time
    - Ex: program stdout, system logs

File System Abstraction

- Path
  - String that uniquely identifies file or directory
  - Ex: /cse/www/education/courses/cse451/12au
- Links
  - Hard link: link from name to metadata location
  - Soft link: link from name to alternate name
- Mount
  - Mapping from name in one file system to root of another

UNIX File System API

- create, link, unlink, createdir, rmdir
  - Create file, link to file, remove link
  - Create directory, remove directory
- open, close, read, write, seek
  - Open/close a file for reading/writing
  - Seek resets current position
- fsync
  - File modifications can be cached
  - fsync forces modifications to disk (like a memory barrier)

File System Interface

- UNIX file open is a Swiss Army knife:
  - Open the file, return file descriptor
  - Options:
    - if file doesn’t exist, return an error
    - If file doesn’t exist, create file and open it
    - If file does exist, return an error
    - If file does exist, open file
    - If file exists but isn’t empty, nix it then open
    - If file exists but isn’t empty, return an error
    - …

Implementation

- Disk buffer cache
- File layout
- Directory layout

Cache

- File consistency vs. loss
- Delayed write:
  - cache replacement
  - sync: Linux every 30 seconds flush the cache
- Write-through:
  - each write into cache goes to disk
- Can also read-ahead: request block logical block k, fetch k+1

File System Design Constraints

- For small files:
  - Small blocks for storage efficiency
  - Files used together should be stored together
- For large files:
  - Contiguous allocation for sequential access
  - Efficient lookup for random access
- 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

File System Design

- Data structures
  - Directories: file name -> file metadata
    - Store directories as files
  - File metadata: how to find file data blocks
  - Free map: list of free disk blocks
- How do we organize these data structures?
  - Device has non-uniform performance

Design Challenges

- Index structure
  - How do we locate the blocks of a file?
- Index granularity
  - What block size do we use?
- Free space
  - How do we find unused blocks on disk?
- Locality
  - How do we preserve spatial locality?
- Reliability
  - What if machine crashes in middle of a file system op?

File System Design Options

	FAT	FFS	NTFS
Index structure	Linked list	Tree (fixed)	Tree (dynamic)
granularity	block	block	extent
free space allocation	FAT array	Bitmap (fixed location)	Bitmap (file)
Locality	defragmentation	Block groups + reserve space	Extents Best fit defrag

Named Data in a File System

file name
offset

directory →

file number
offset

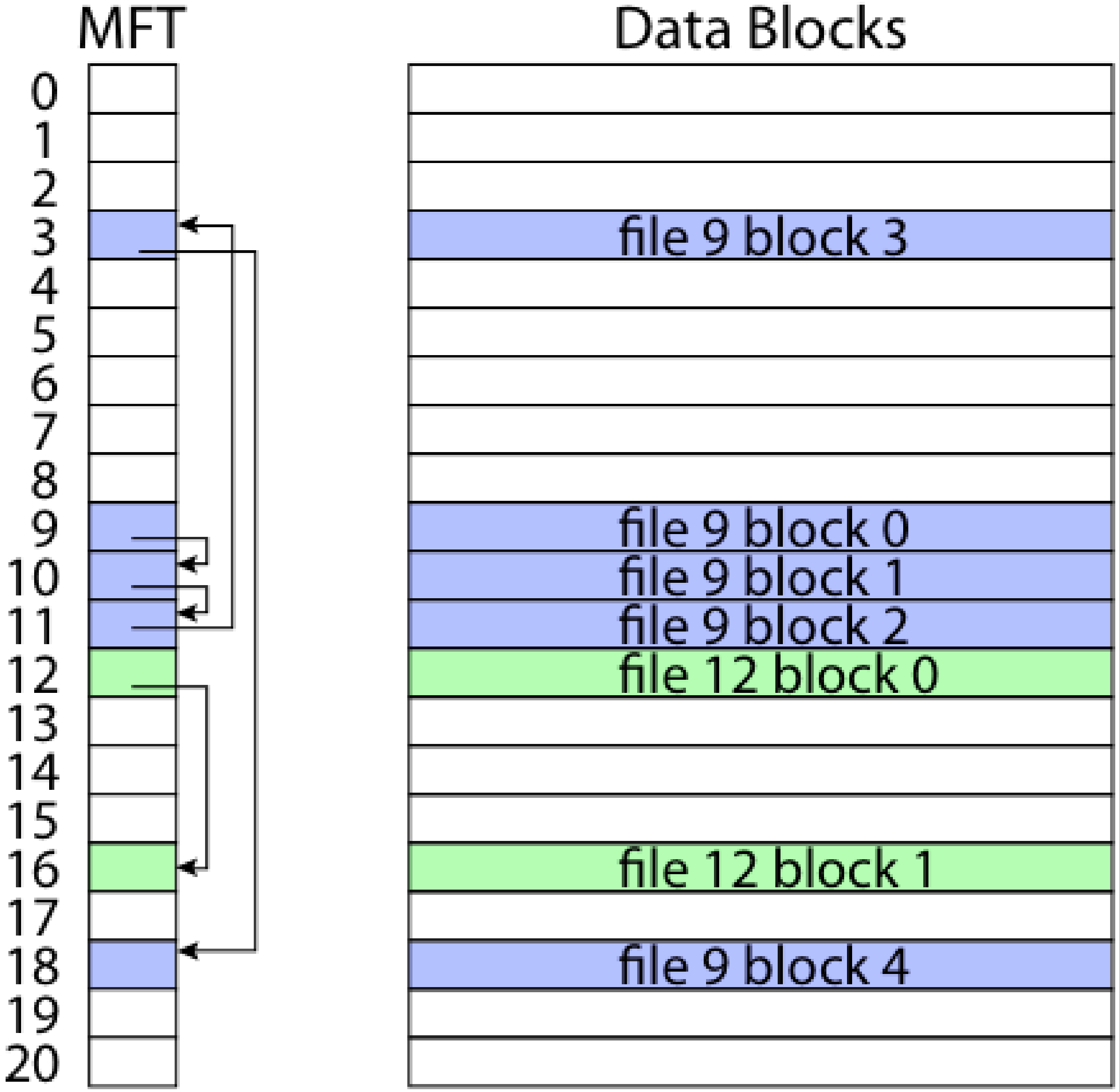
index structure →

storage
block

Microsoft File Allocation Table (FAT)

- Linked list index structure
  - Simple, easy to implement
  - Still widely used (e.g., thumb drives)
- File table:
  - Linear map of all blocks on disk
  - Each file a linked list of blocks

FAT


MFT
Data Blocks
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
file 9 block 3
file 9 block 0
file 9 block 1
file 9 block 2
file 12 block 0
file 12 block 1
file 9 block 4

FAT

- Pros:
- Cons:

Berkeley UNIX FFS (Fast File System)

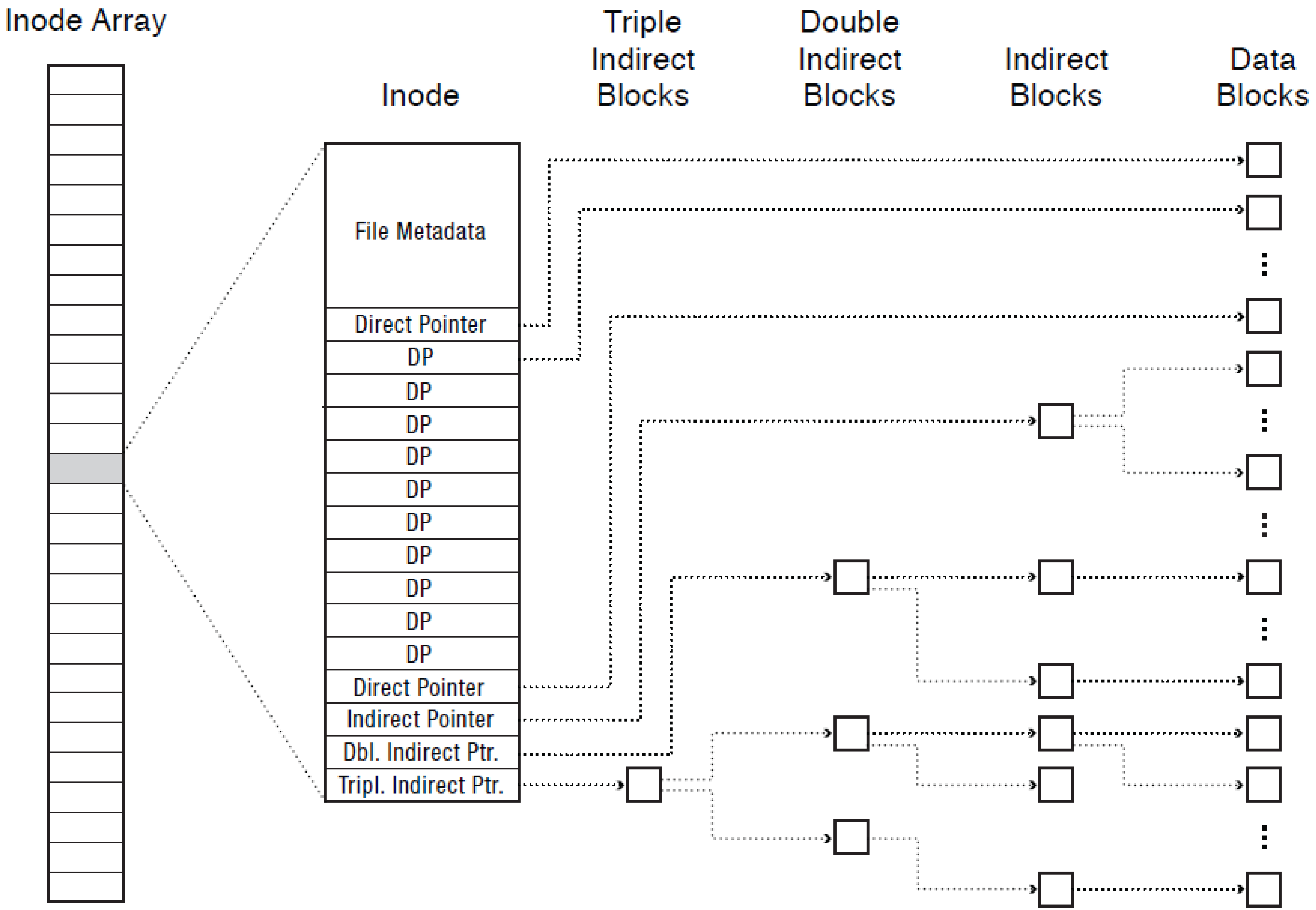
- inode table
  - Analogous to FAT table
- inode
  - Metadata
  - Set of 12 direct data pointers
  - 4KB block size

FFS inode

- Metadata
  - File owner, access permissions, access times, …
- Set of 12 data pointers
  - With 4KB blocks => max size of 48KB files
- Indirect block pointer
  - pointer to disk block of data pointers
- Indirect block: 1K data blocks => ?

FFS inode

- Doubly indirect block pointer
  - Doubly indirect block => 1K indirect blocks
  - ?
- Triply indirect block pointer
  - Triply indirect block => 1K doubly indirect blocks
  - ?


Inode Array
Inode
Triple Indirect Blocks
Double Indirect Blocks
Indirect Blocks
Data Blocks
File Metadata
Direct Pointer
DP
DP
DP
DP
DP
DP
DP
DP
DP
DP
Direct Pointer
Indirect Pointer
Dbl. Indirect Ptr.
Tripl. Indirect Ptr.

Permissions

- setuid
- setgid

Named Data in a File System

file name
offset

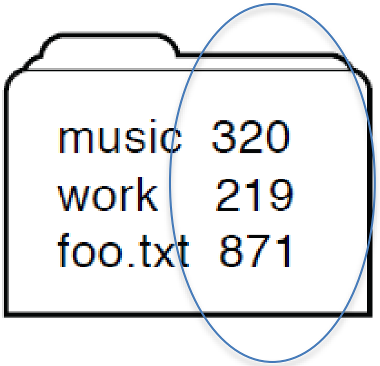
directory →

file number
offset

index
structure →

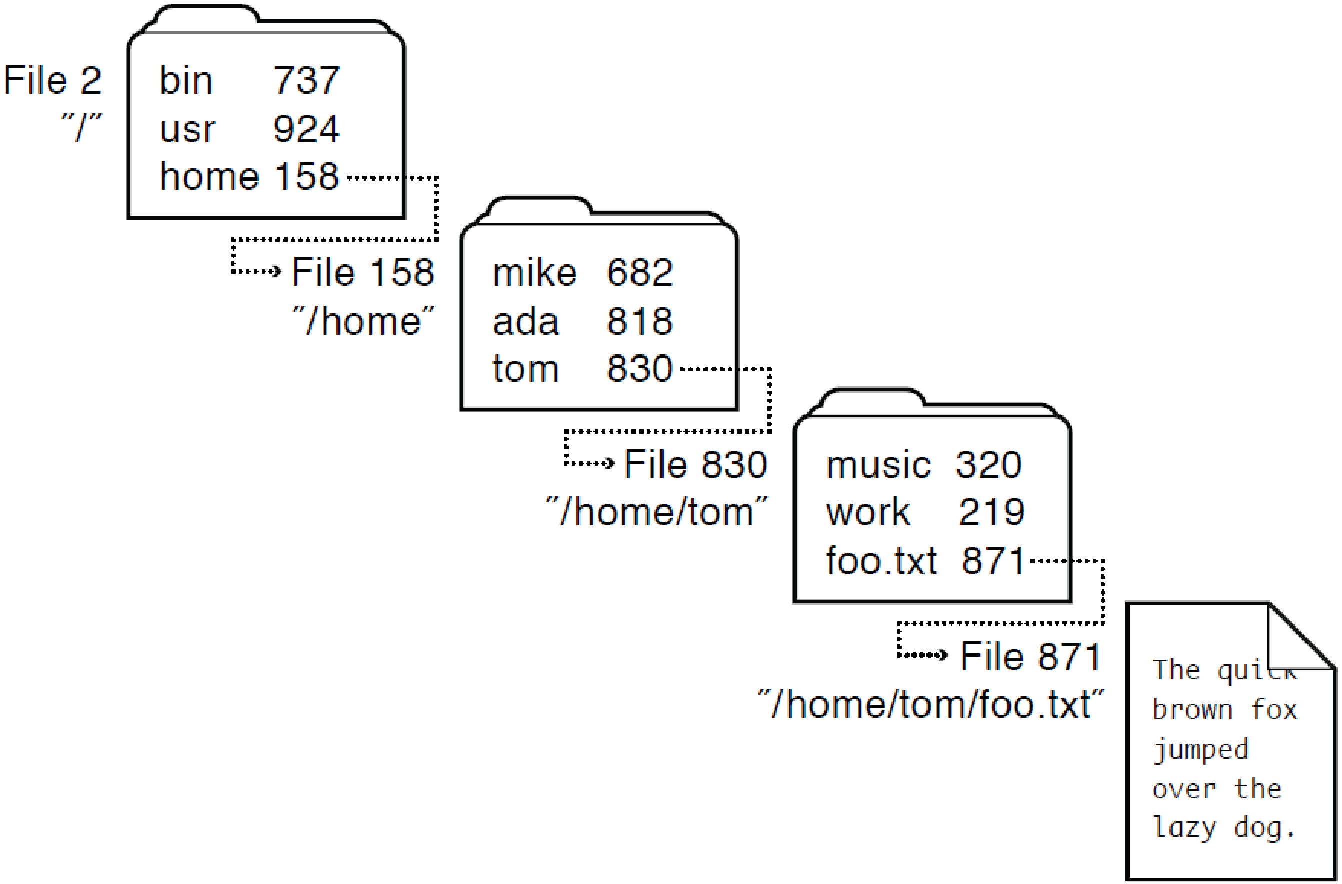
storage
block

Directories Are Files



music	320
work	219
foo.txt	871

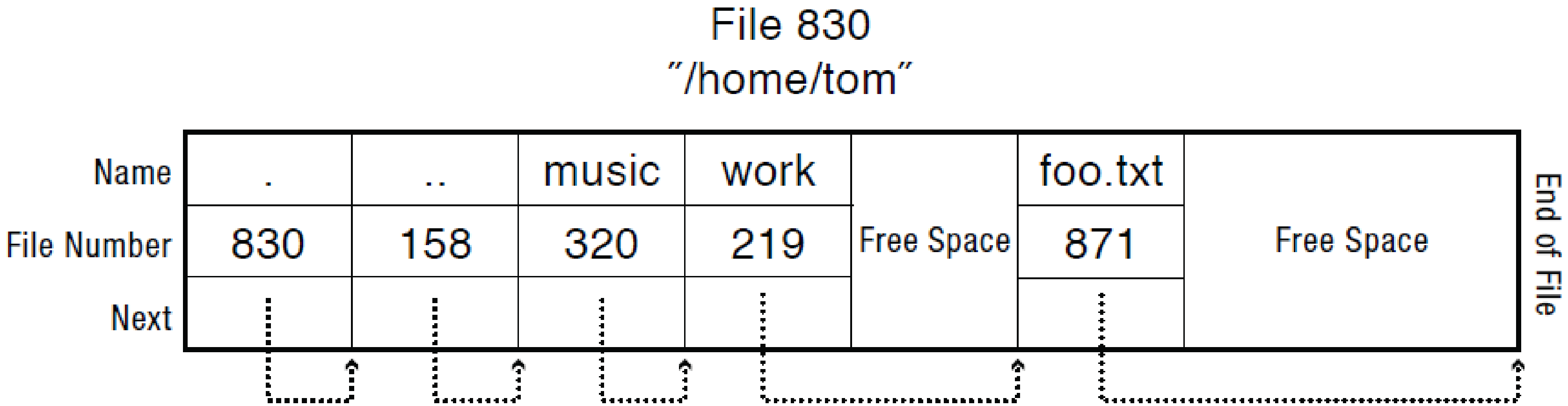
Recursive Filename Lookup



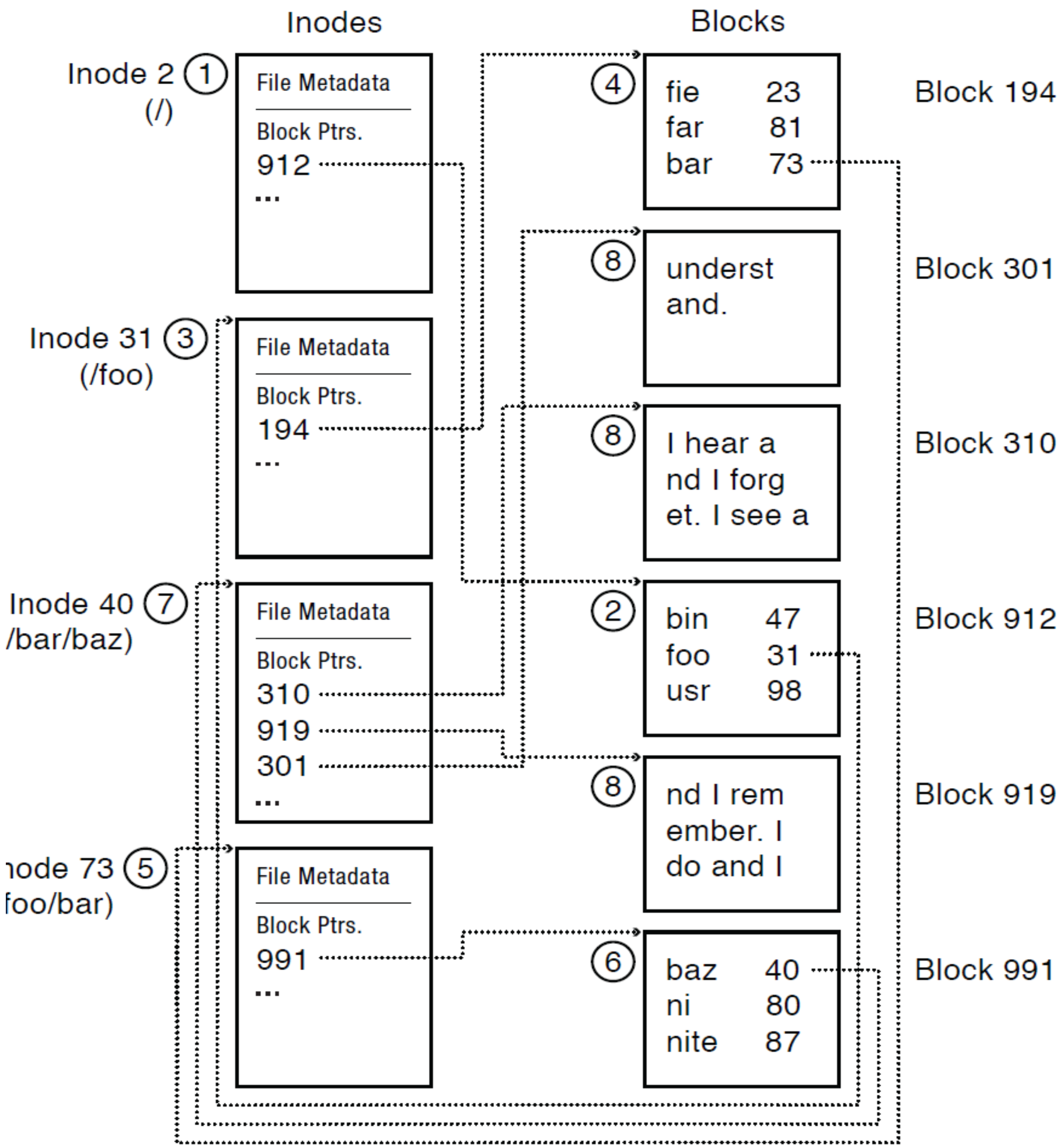
Directory Layout

Directory stored as a file

Linear search to find filename (small directories)

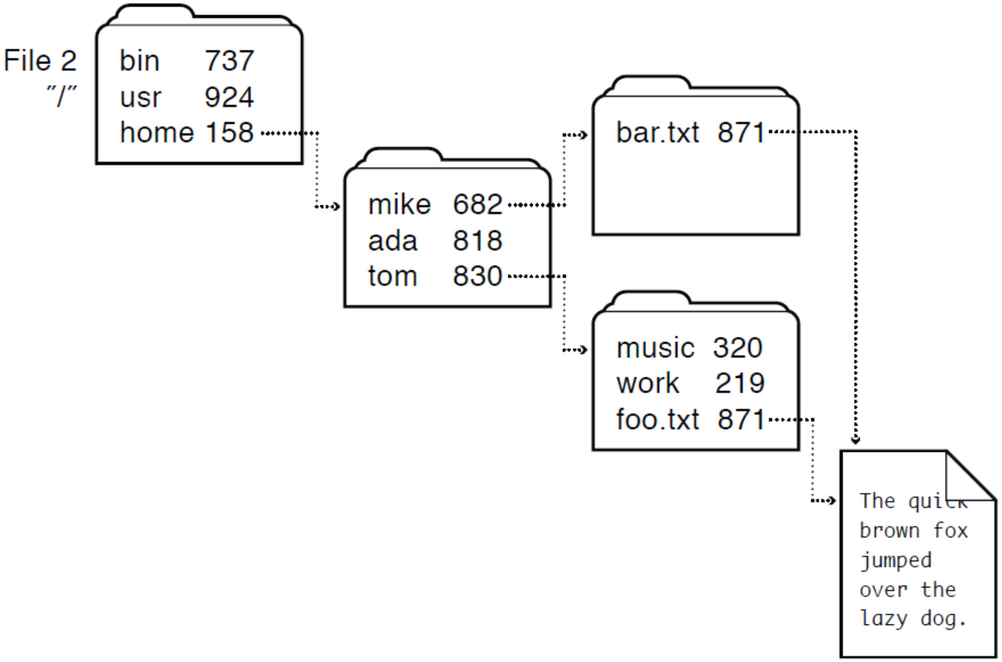


Putting it all together



/foo/bar/baz

Links



File 2
"/"

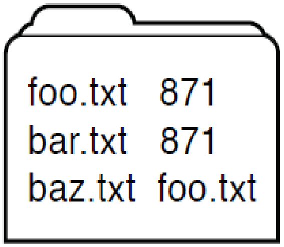
bin 737
usr 924
home 158

mike 682
ada 818
tom 830

bar.txt 871

music 320
work 219
foo.txt 871

The quick brown fox jumped over the lazy dog.

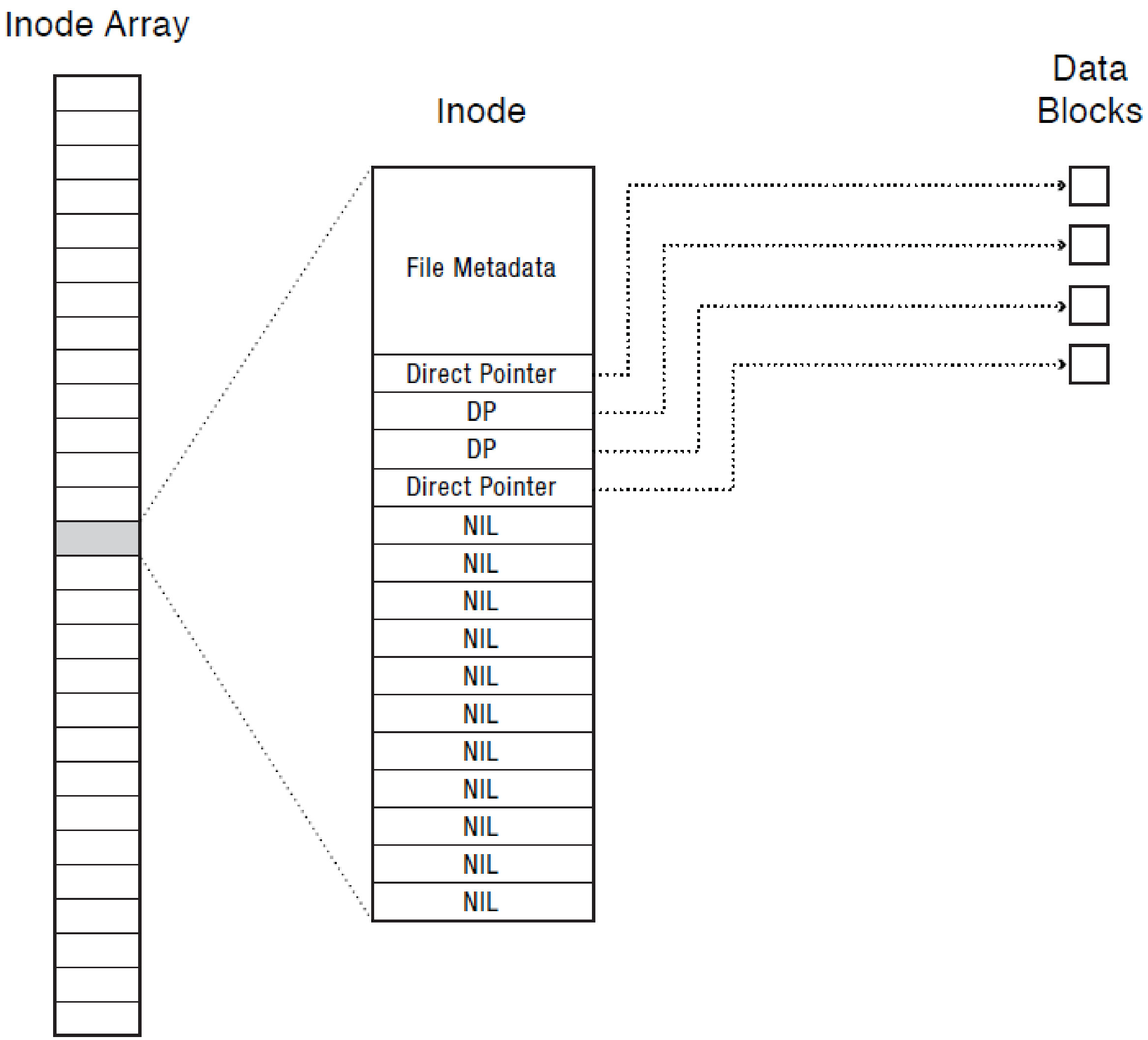


foo.txt 871
bar.txt 871
baz.txt foo.txt

FFS Asymmetric Tree

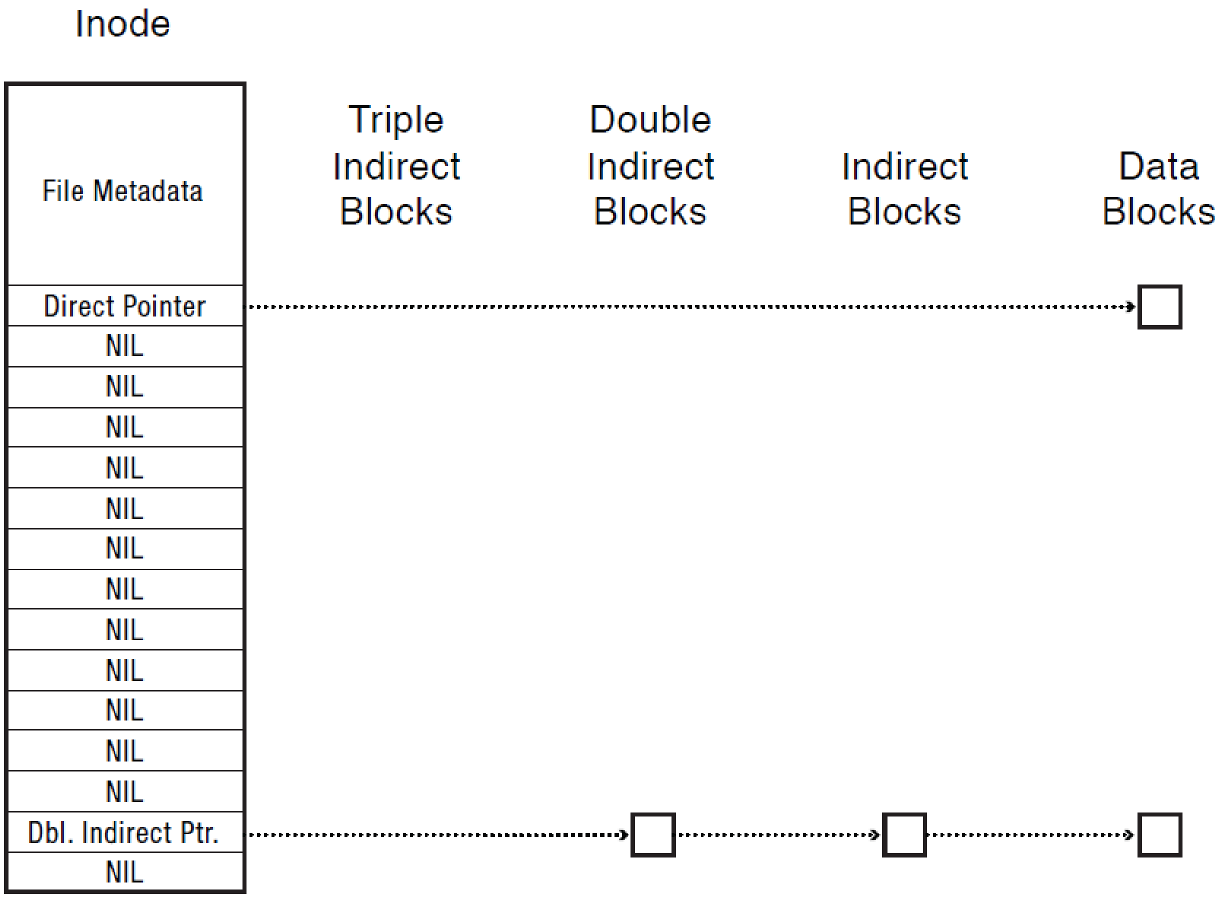
- Small files: shallow tree
  - Efficient storage for small files
- Large files: deep tree
  - Efficient lookup for random access in large files
- Sparse files: only fill pointers if needed

Small Files



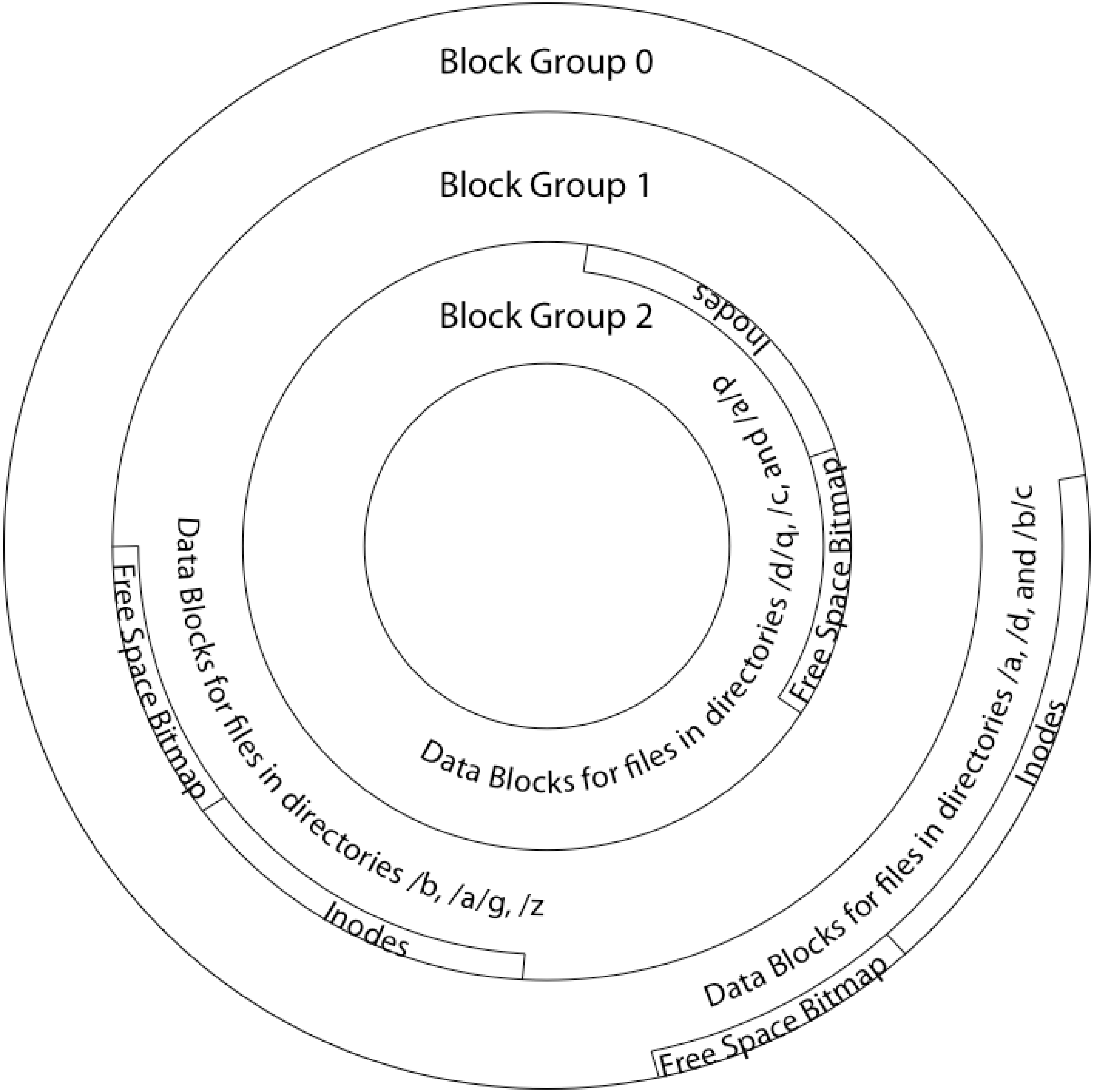
Inode Array
Inode
Data Blocks
File Metadata
Direct Pointer
DP
DP
Direct Pointer
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL

Sparse Files

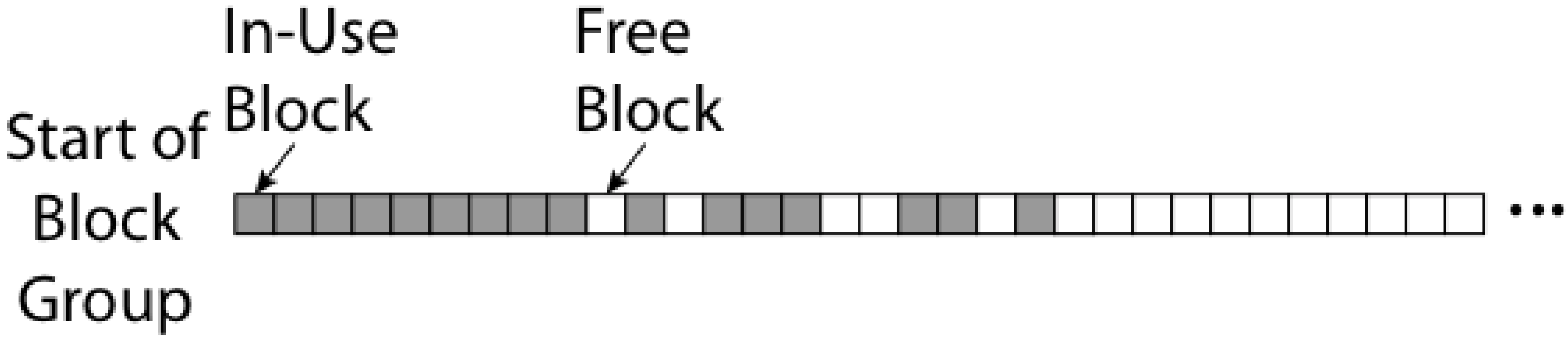


FFS Locality

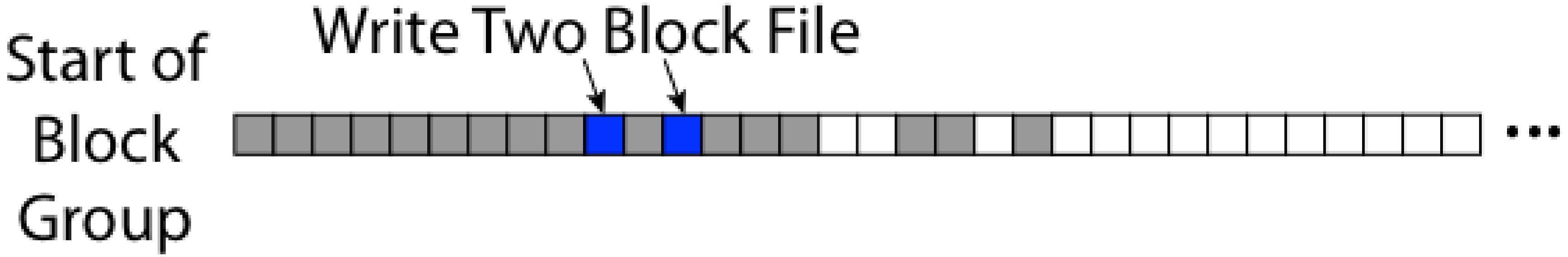
- Block group allocation
  - Block group is a set of nearby cylinders
  - Files in same directory located in same group
  - Subdirectories located in different block groups
- inode table spread throughout disk
  - inodes, bitmap near file blocks
- First fit allocation
  - Small files fragmented, large files contiguous


Block Group 0
Block Group 1
Block Group 2
Inodes
Free Space Bitmap
Data Blocks for files in directories /d/q, /c, and /a/p
Free Space Bitmap
Data Blocks for files in directories /b, /a/g, /z
Inodes
Inodes
Data Blocks for files in directories /a, /d, and /b/c
Free Space Bitmap

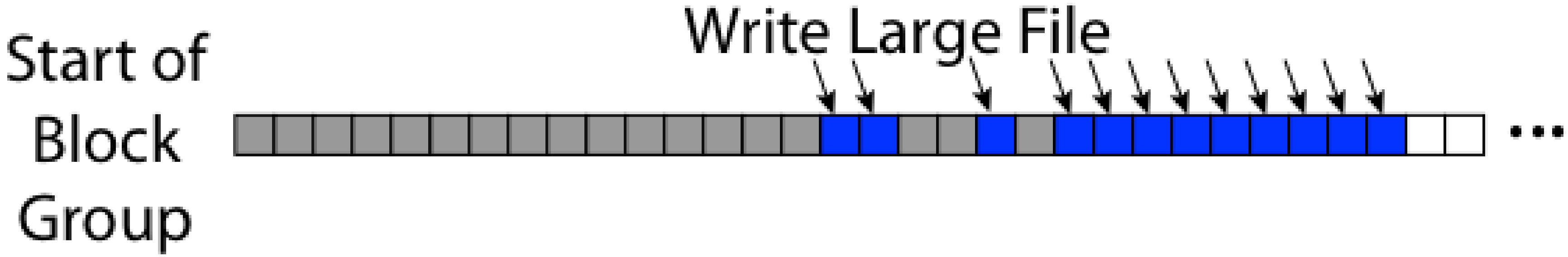
FFS First Fit Block Allocation



FFS First Fit Block Allocation



FFS First Fit Block Allocation



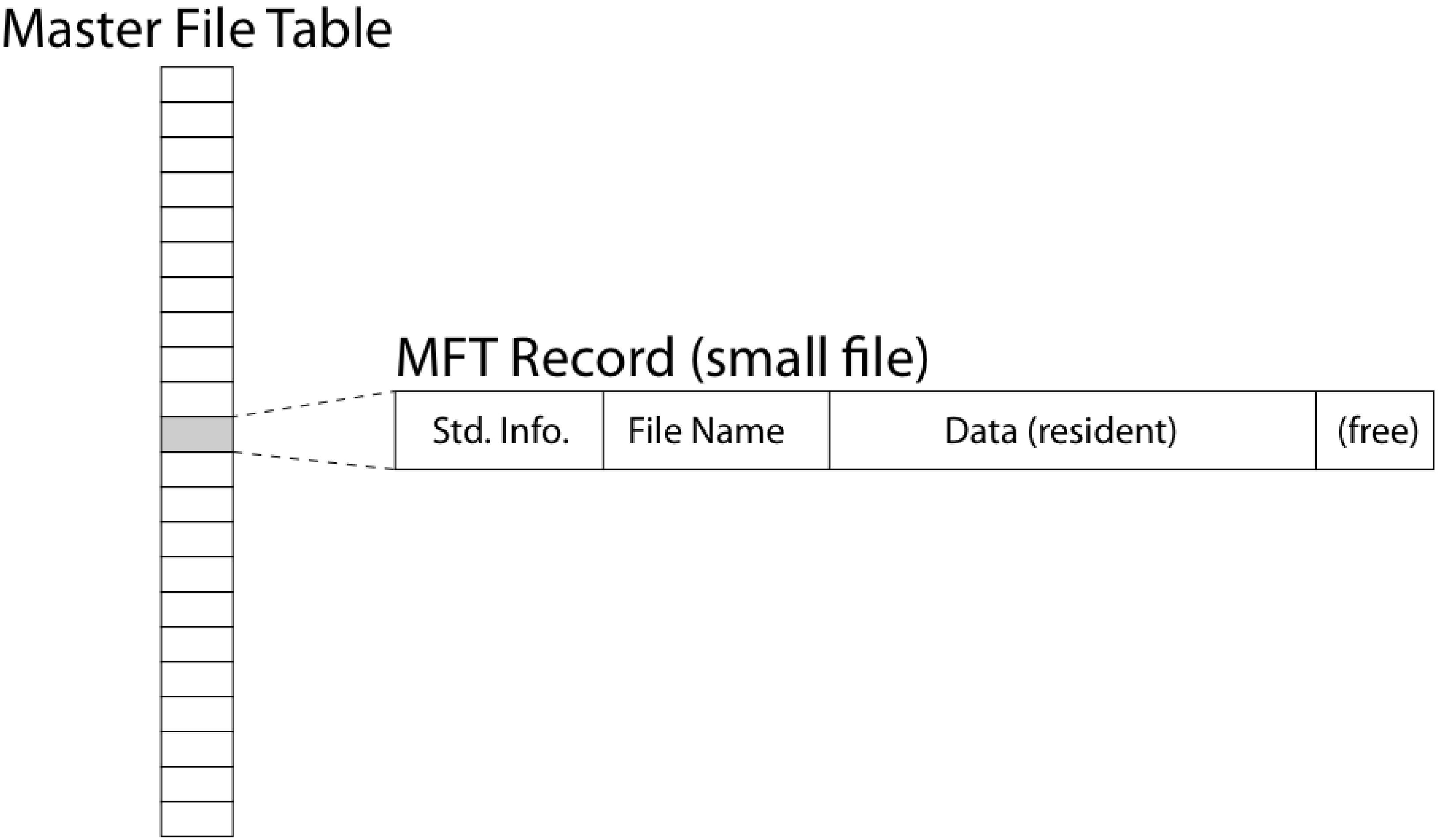
FFS

- 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 (a 1 byte file requires both an inode and a data block)
  - Inefficient encoding when file is mostly contiguous on disk (no equivalent to superpages)

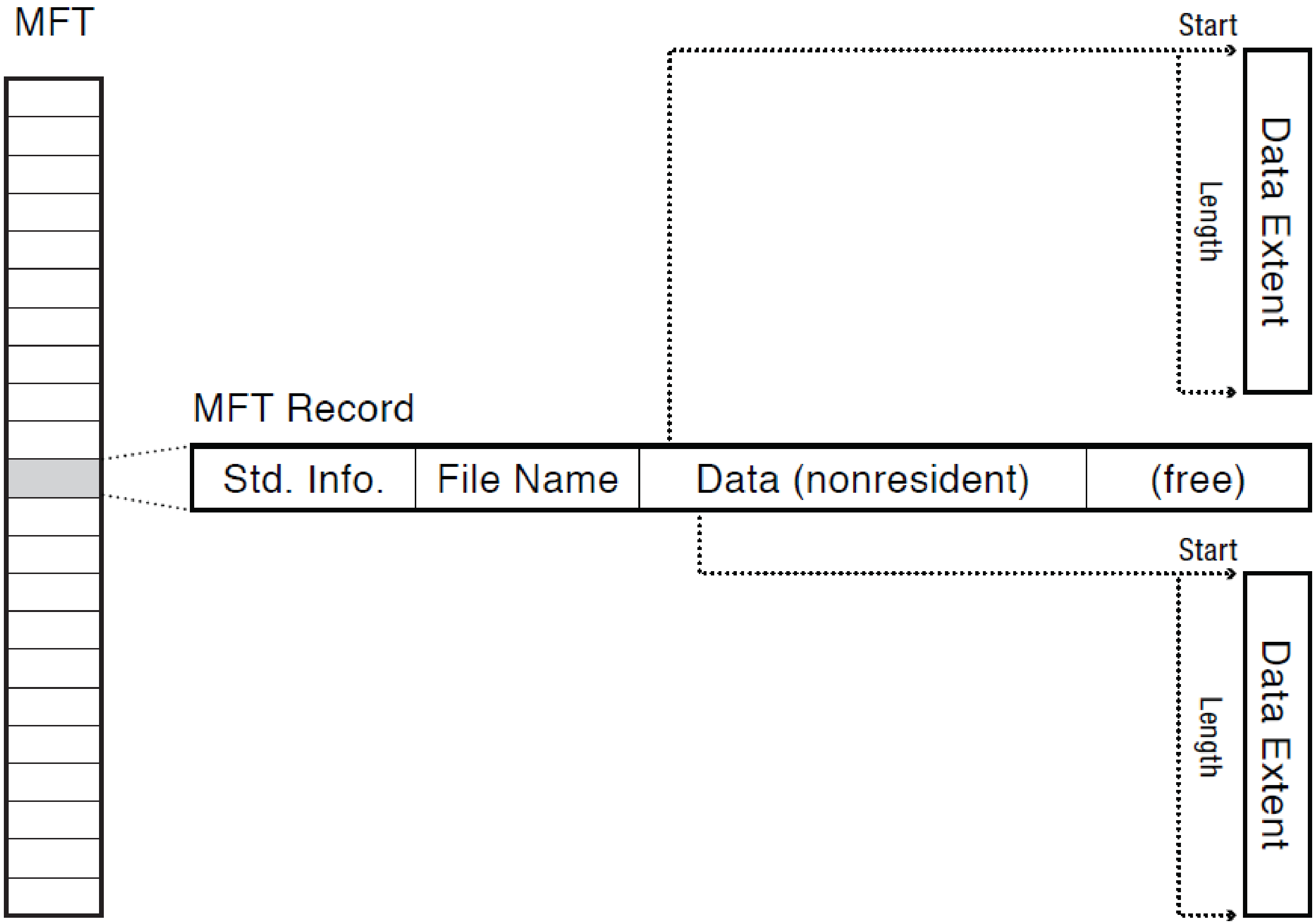
NTFS

- Master File Table
  - Flexible 1KB storage for metadata and data
- Extents
  - Block pointers cover runs of blocks
  - Similar approach in linux (ext4)
  - File create can provide hint as to size of file
- Journaling for reliability
  - Coming soon

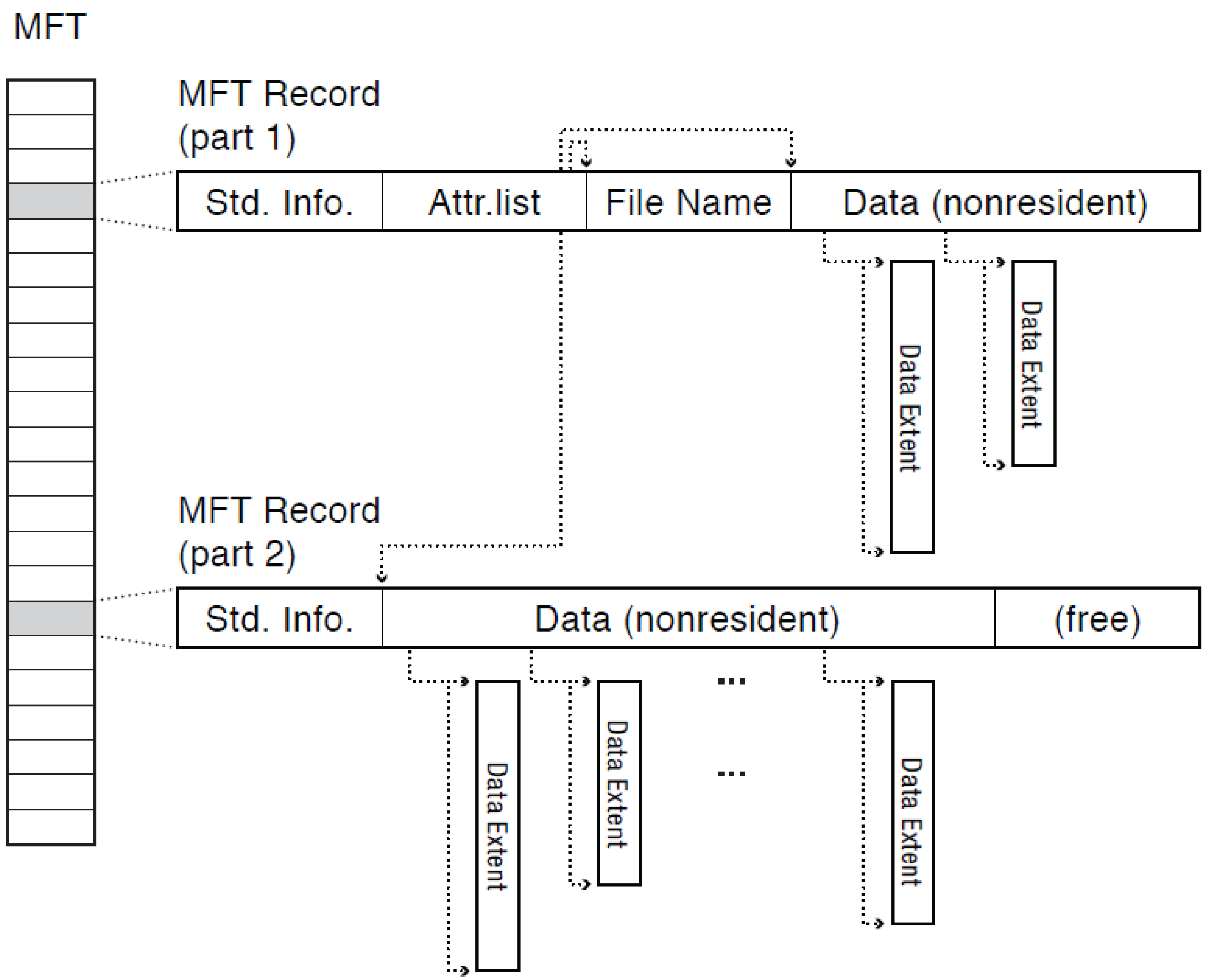
NTFS Small File



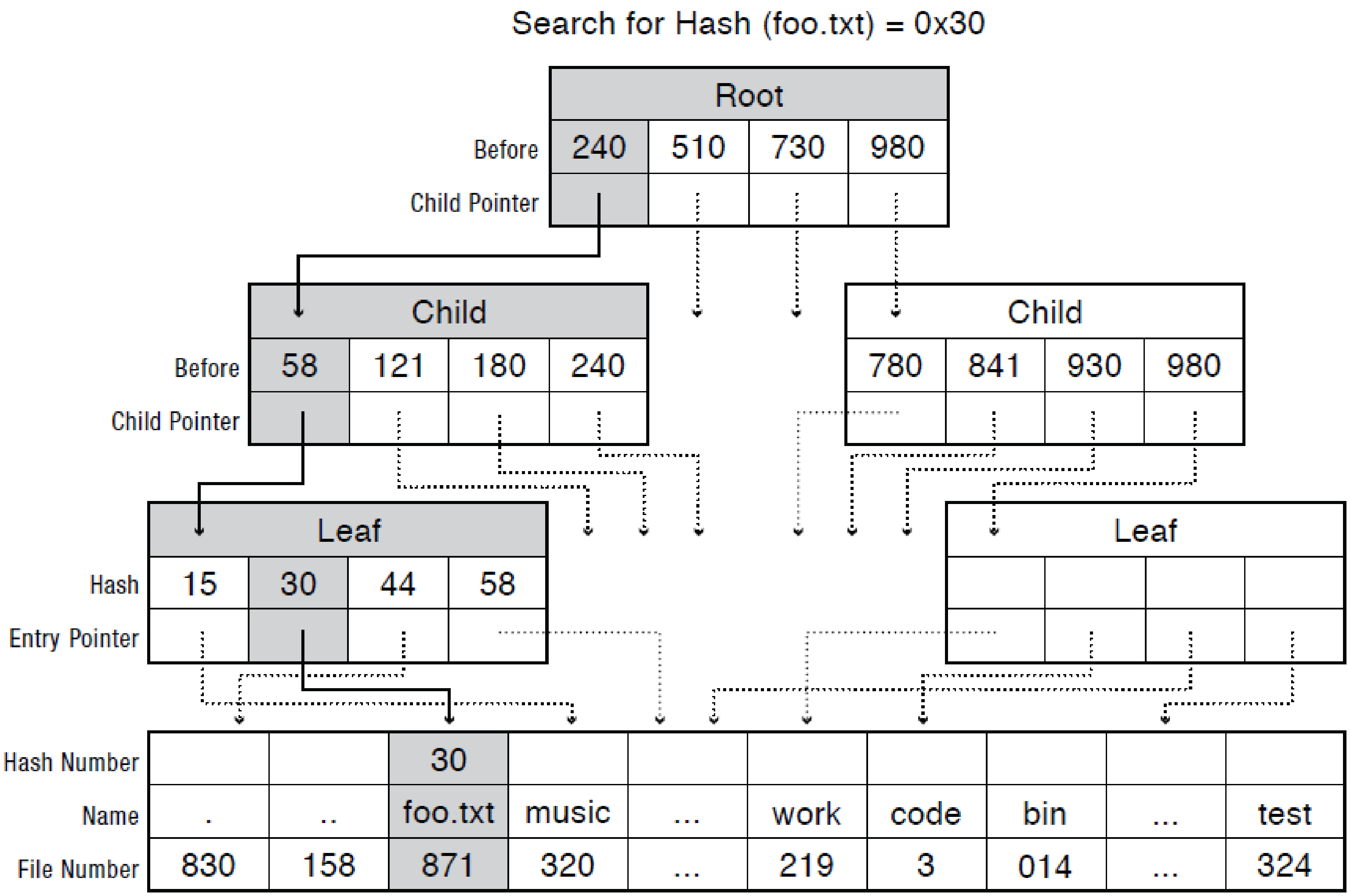
NTFS Medium-Sized File



NTFS Indirect Block



Large Directories: B Trees



Search for Hash (foo.txt) = 0x30

Root

Before	240	510	730	980
Child Pointer				

Child

Before	58	121	180	240
Child Pointer				

Child

780	841	930	980

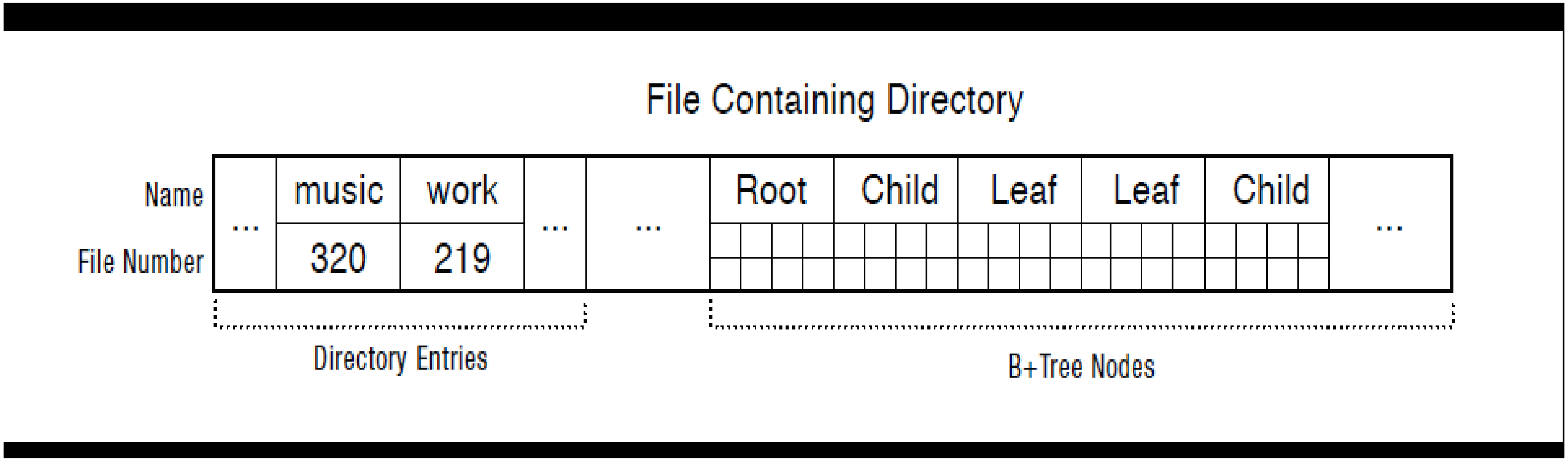
Leaf

Hash	15	30	44	58
Entry Pointer				

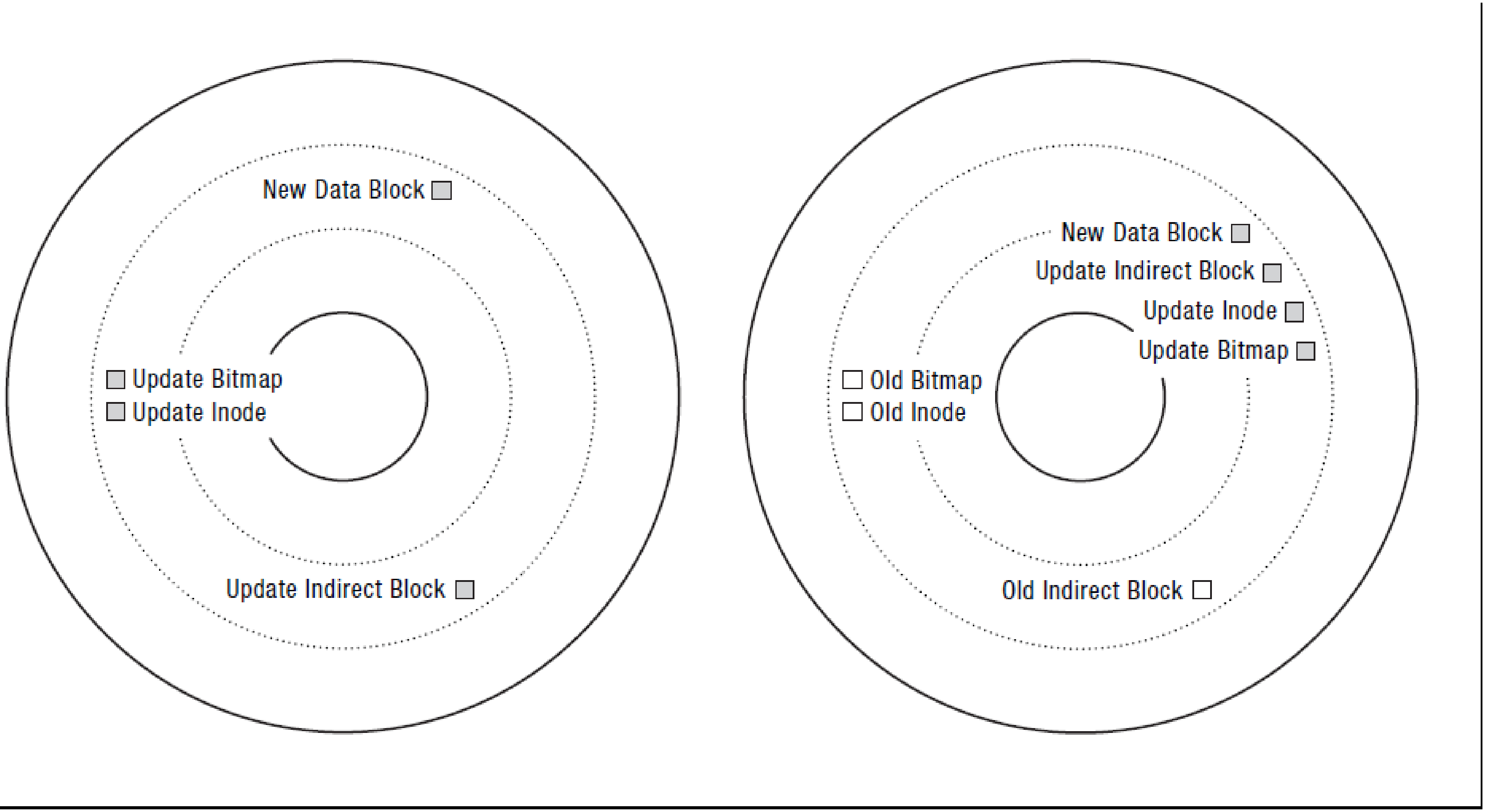
Leaf

Hash Number			30							
Name	.	..	foo.txt	music	...	work	code	bin	...	test
File Number	830	158	871	320	...	219	3	014	...	324

Large Directories: Layout



Copy-on-Write



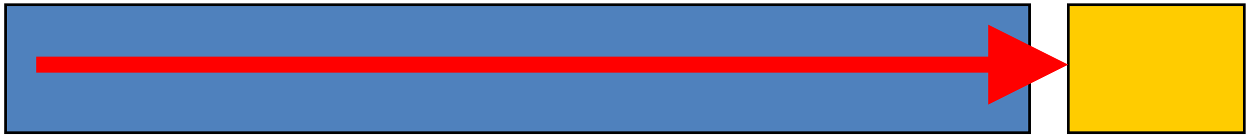
LFS

Limitations of existing file systems

- They spread information around the disk
  - data blocks of a single large file may be together, but ...
  - inodes stored apart from data blocks
  - directory blocks separate from file blocks
  - writing small files -> less than 5% of disk bandwidth is used to access new data, rest of time is seeking
- Use ***synchronous writes*** to update directories and inodes
  - required for consistency
  - makes seeks even more painful; stalls CPU

Key Idea

- Write all modifications to disk sequentially in a log-like structure



  - Convert many small random writes into large sequential transfers
  - Use file cache as write buffer first, then write to disk sequentially
  - Assume crashes are rare

Main advantages

- Replaces many small random writes by fewer sequential writes

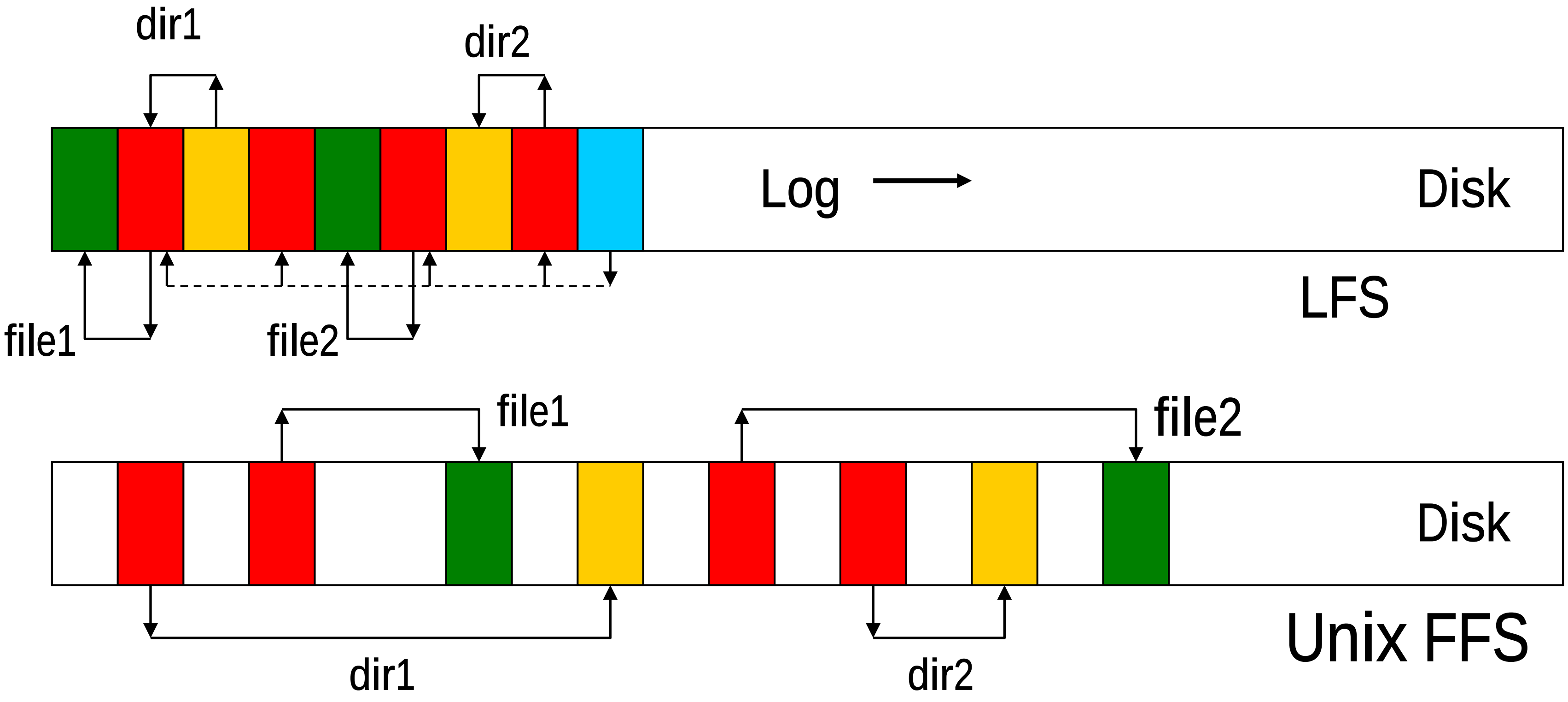
- Faster recovery after a crash
  - all blocks that were recently written are at the tail end of log

- Downsides?

The Log

- Log contains modified inodes, data blocks, and directory entries
- Most reads will access data already in the cache
  - If not, it can get expensive to go through the log if files are fragmented
- No freelist!
- Only structures on disk are the log and
- inode-map (maps inode # to its disk position) located in well-known place on the disk

Disk layouts of LFS and UNIX



🟥 Inode

🟨 Directory

🟩 Data

🟦 Inode map

Segments

- Must maintain large free disk-areas for writing new data
  - Disk is divided into large fixed-size areas called ***segments*** (512 kB in Sprite LFS)
- Segments are always written sequentially from one end to the other
  - Includes summary information
- Keep writing the log out … problem?

Issues

- Issues:
  - when to run cleaner?
  - how many segments to clean at a time?
  - which segments to clean?
  - how to re-write the live blocks?

- First two – they advocate simple thresholds (want % of free segments)

Segment cleaning

- Old segments contain
  - ***live data***
  - “dead data” belonging to files that were deleted or over-written

- Segment cleaning involves reading in and writing out the live data

- Segment summary block identifies each piece of information in the segment (for data blocks to which inodes are they associated)

Segment cleaning (cont’d)

- Segment cleaning process involves
  1. reading a number of segments into memory (which)
  2. identifying the live data
  3. writing them back to a smaller number of clean segments (how)

Write cost

u = utilization

(fraction of live data)

$$\text{write cost} = \frac{\text{total bytes read and written}}{\text{new data written}}$$

$$= \frac{\text{read segs} + \text{write live} + \text{write new}}{\text{new data written}}$$

$$= \frac{N + N^{*}u + N^{*}(1-u)}{N^{*}(1-u)} = \frac{2}{1-u}$$

Segment Cleaning Policies: which

- ***Greedy policy:*** always cleans the least-utilized segments
- ***Cost-benefit policy:*** selects segments with the highest benefit-to-cost ratio

$$\frac{\text{benefit}}{\text{cost}} = \frac{\text{free space generated} * \text{age of data}}{\text{cost}} = \frac{(1-u)*\text{age}}{1+u}$$

older data – more stable
newer data – more likely to be modified or deleted – cleaning wastes time

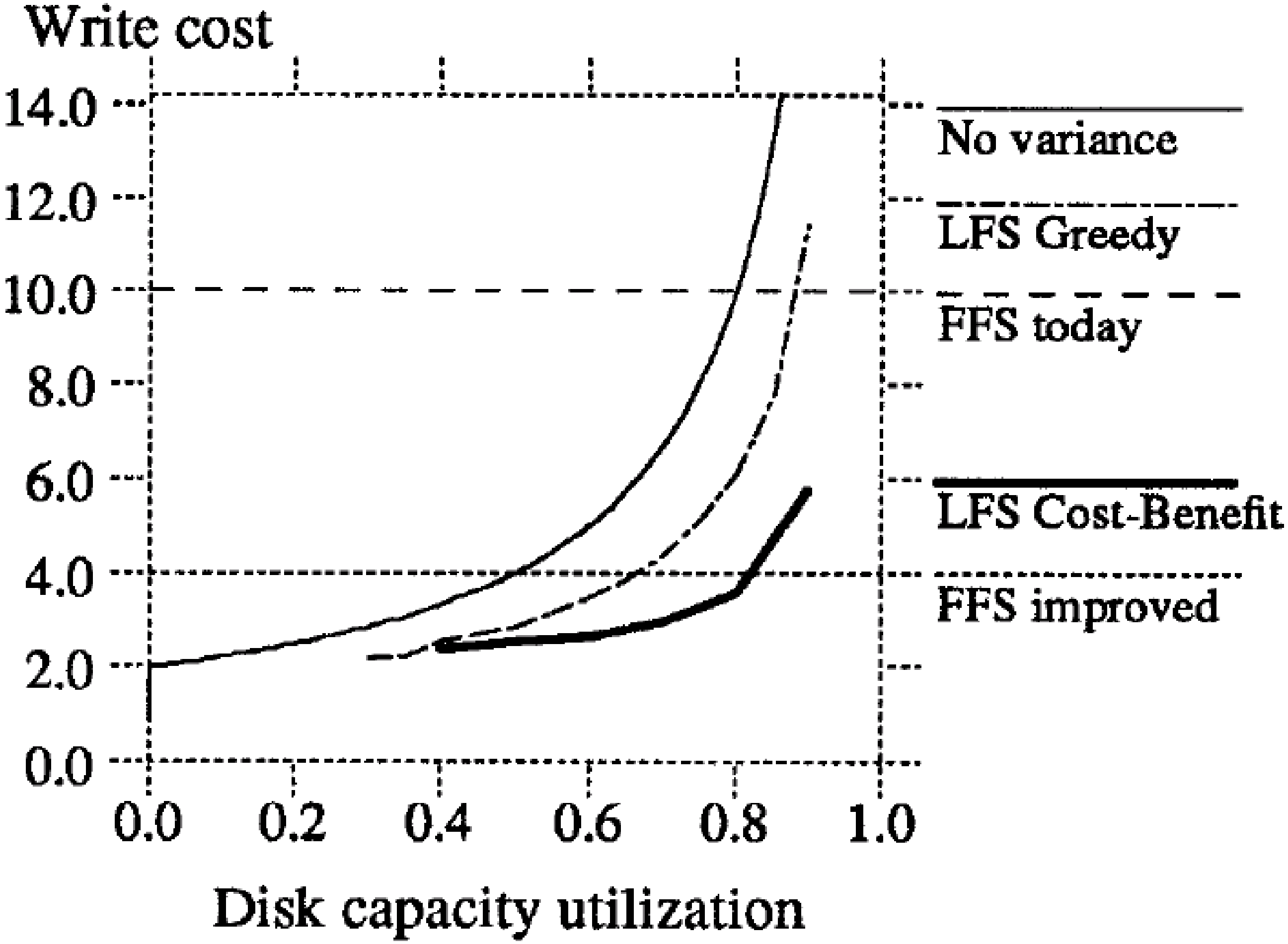
1 to read, u to copy

Copying life blocks: where

- ***Age sort:***
  - sorts the blocks by the time they were last modified
  - groups blocks of similar age together into new segments

- Age of a block is good predictor of its survival
- Supports cost-benefit policy

Using a cost benefit policy



Systems Mantras

- Be clever at high utilization!
- Bulk operations work better than large number of smaller ones