CS 423
Operating System Design: File System Implementation

Tianyin Xu

Thanks Prof. Adam Bates for the slides.

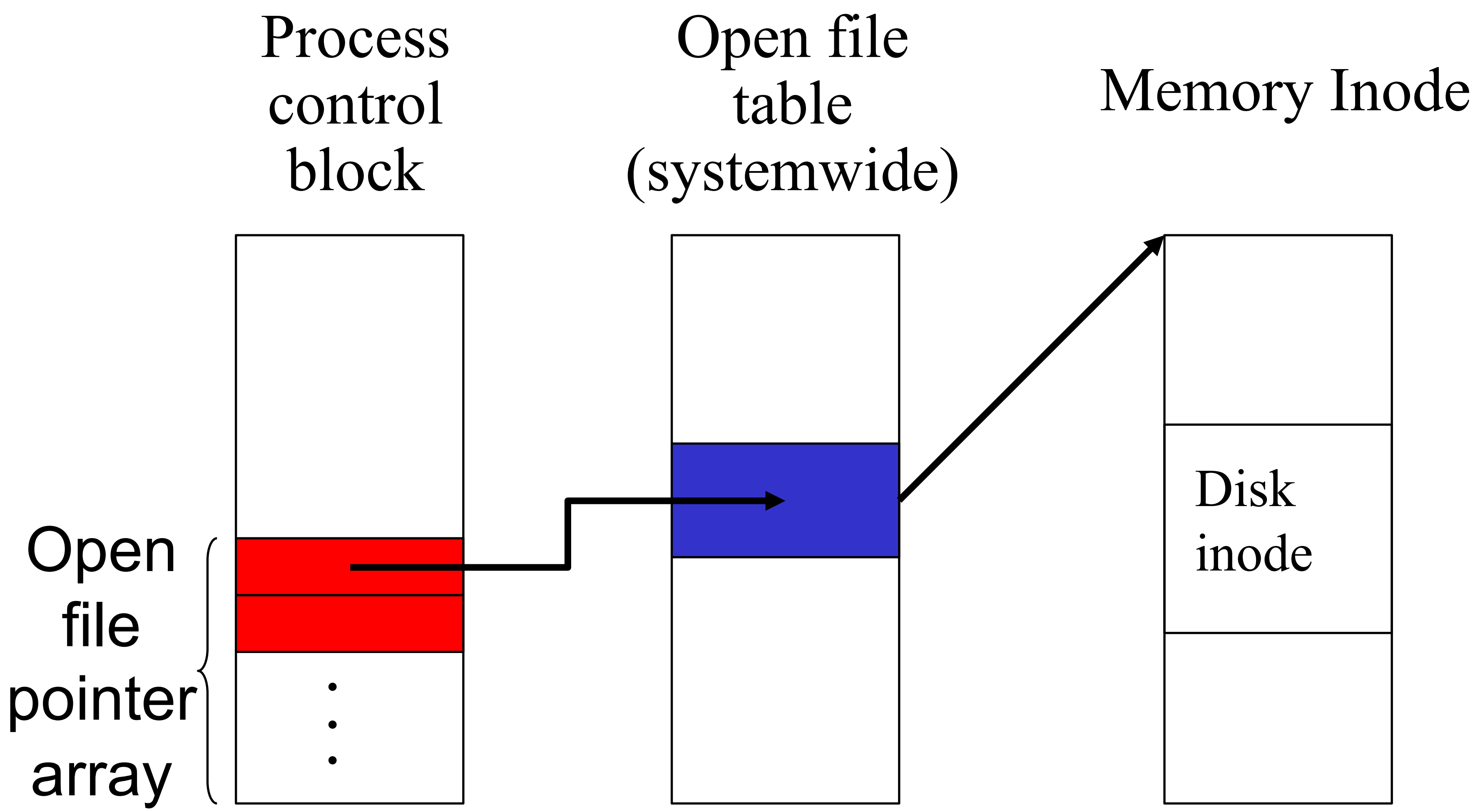
Grading

- Still letter grade, instead of N/NP
- You can change it to CR/NC
- Will be **very** generous in grading
  - Do your best and you will have good grade
- If you are not able to finish, we can do "incomplete"

- Details in my Piazza post.

Final grading decision

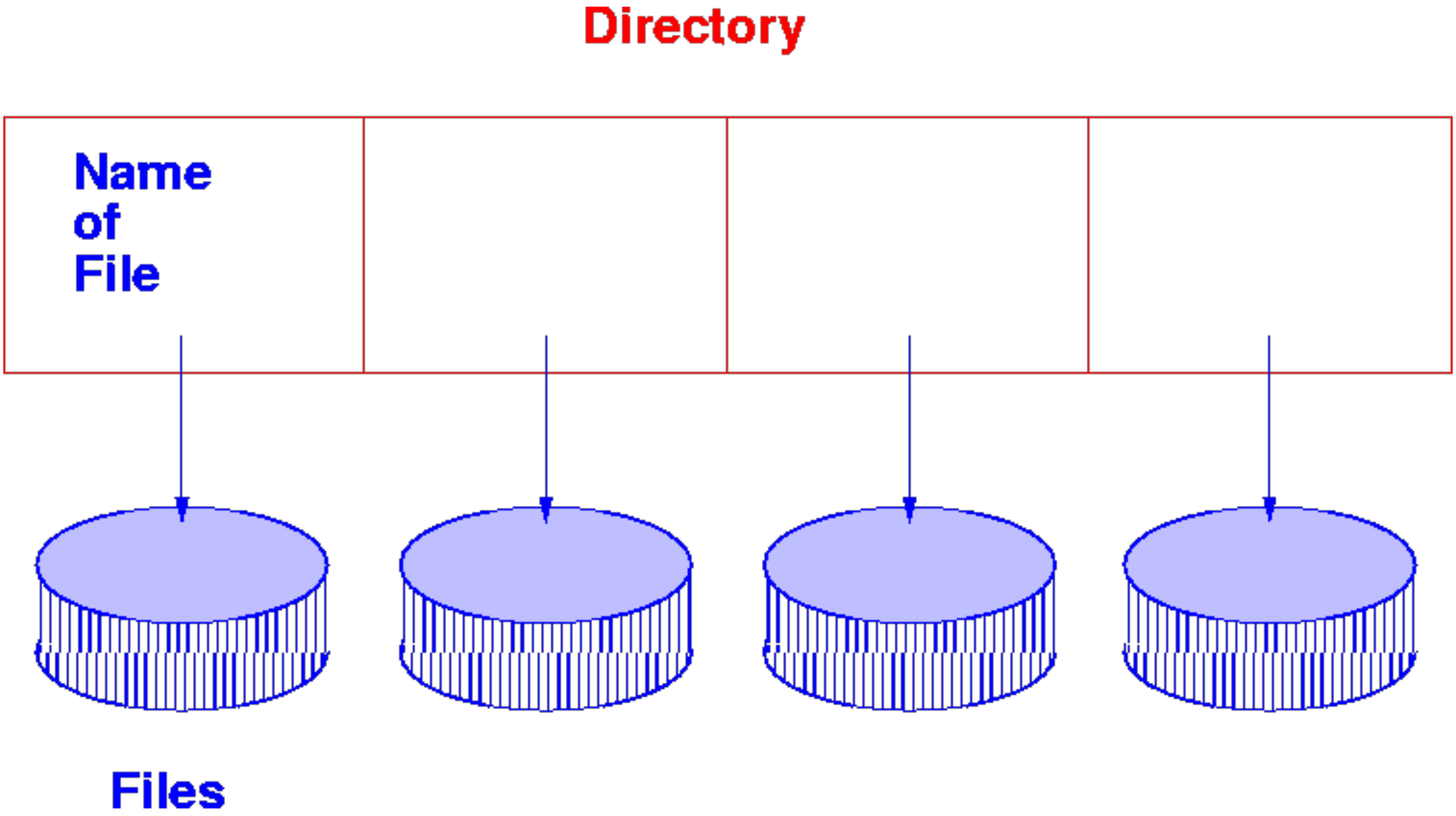
Data structures in a typical file system:



Directory Structure

- maps symbolic names into logical file names
  - search
  - create file
  - list directory
  - backup, archival, file migration

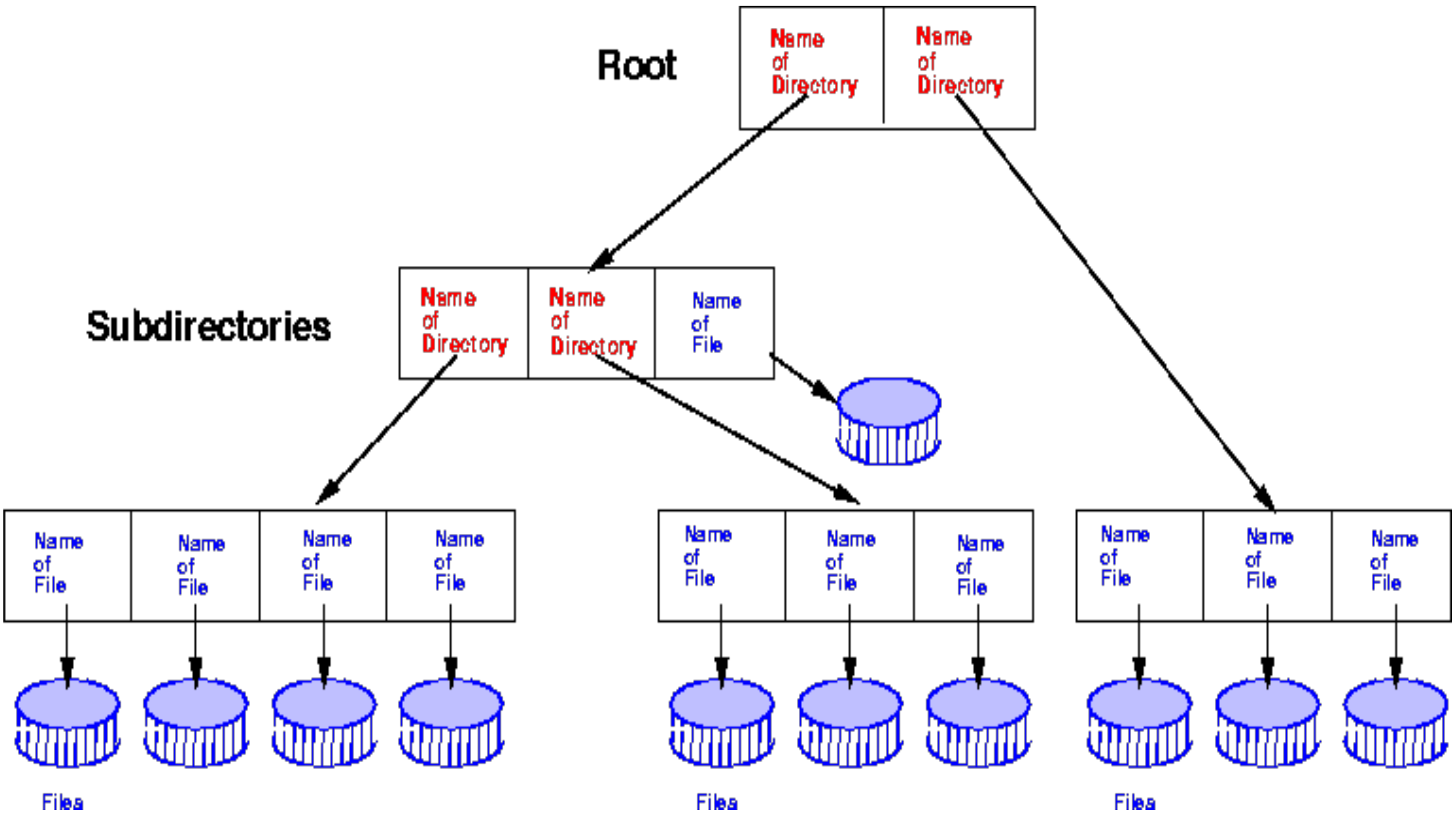
Single-level Directory



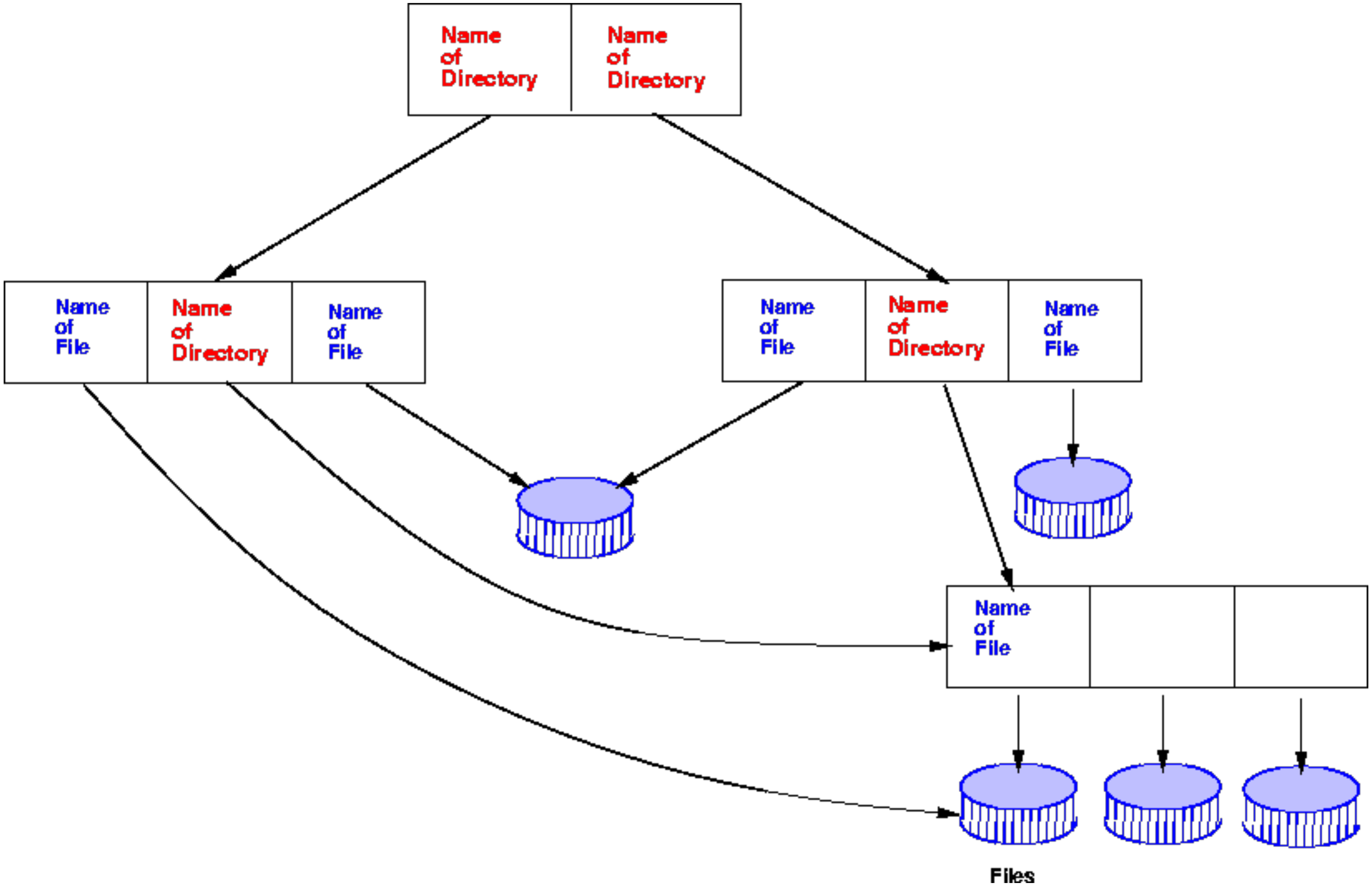
Tree-Structured Directories

- arbitrary depth of directories
- leaf nodes are files
- interior nodes are directories
- path name lists nodes to traverse to find node
- use absolute paths from root
- use relative paths from current working directory pointer

Tree-Structured Directories

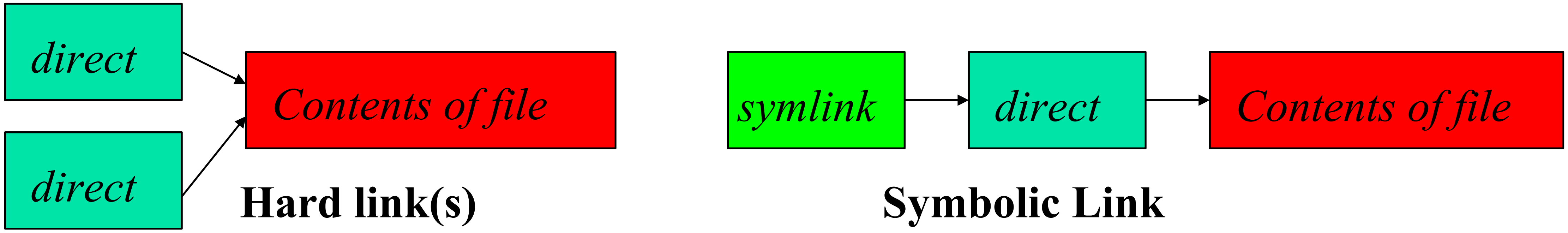


Acyclic Graph Structured Dir.'s



Symbolic Links

- **Symbolic** links are different than regular links (often called **hard links**). Created with **ln -s**
- Can be thought of as a directory entry that points to the name of another file.
- Does not change link count for file
  - When original deleted, symbolic link remains
- They exist because:
  - Hard links don't work across file systems
  - Hard links only work for regular files, not directories

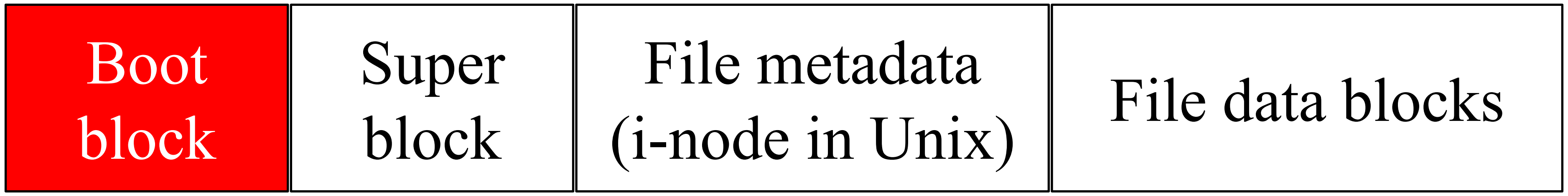


Hard link(s)

Symbolic Link

Disk Layout for a FS

Disk layout in a typical file system:



Boot block

Super block

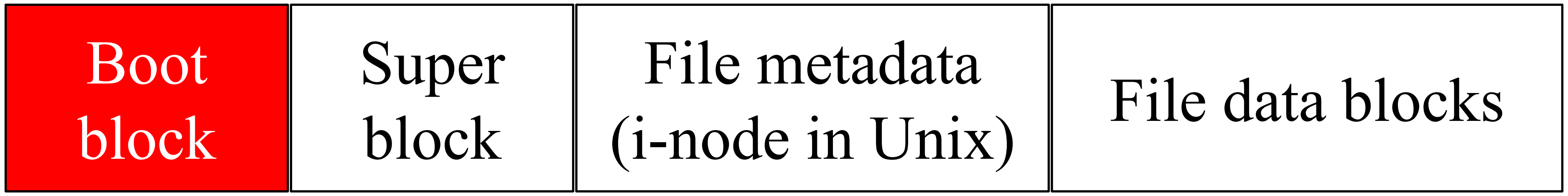
File metadata (i-node in Unix)

File data blocks

- Data Structures:
    - File data blocks: File contents
    - File metadata: How to find file data blocks
    - Directories: File names pointing to file metadata
    - Free map: List of free disk blocks

Disk Layout for a FS

Disk layout in a typical file system:



Boot block

Super block

File metadata (i-node in Unix)

File data blocks

- Superblock defines a file system
  - size of the file system
  - size of the file descriptor area
  - free list pointer, or pointer to bitmap
  - location of the file descriptor of the root directory
  - other meta-data such as permission and various times
- For reliability, replicate the superblock

Design Constraints

- How can we allocate files efficiently?
  - 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
  - Challenge: May not know at file creation where our file will be small or large!!

Design Challenges

- Index structure
  - *How do we locate the blocks of a file?*
- Index granularity
  - *How much data per each index (i.e., block size)?*
- 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 Allocation

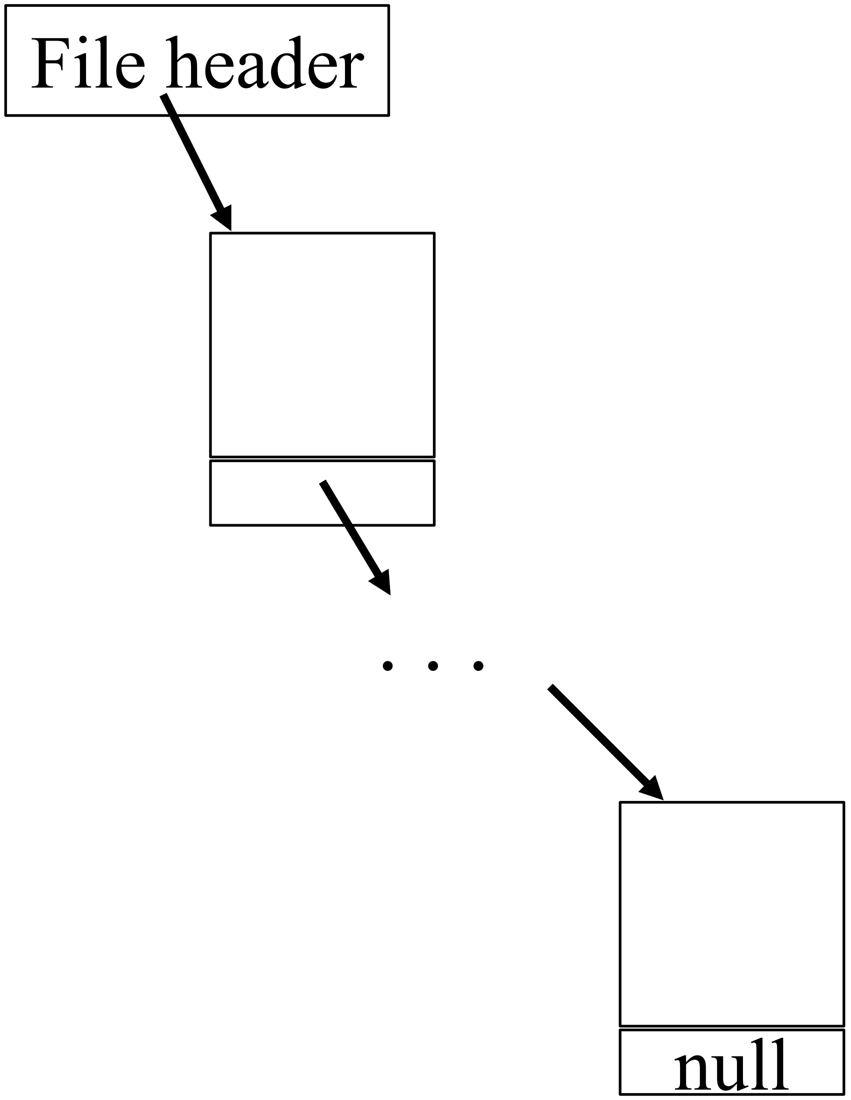
- Contiguous
- Non-contiguous (linked)
- Tradeoffs?

Contiguous Allocation

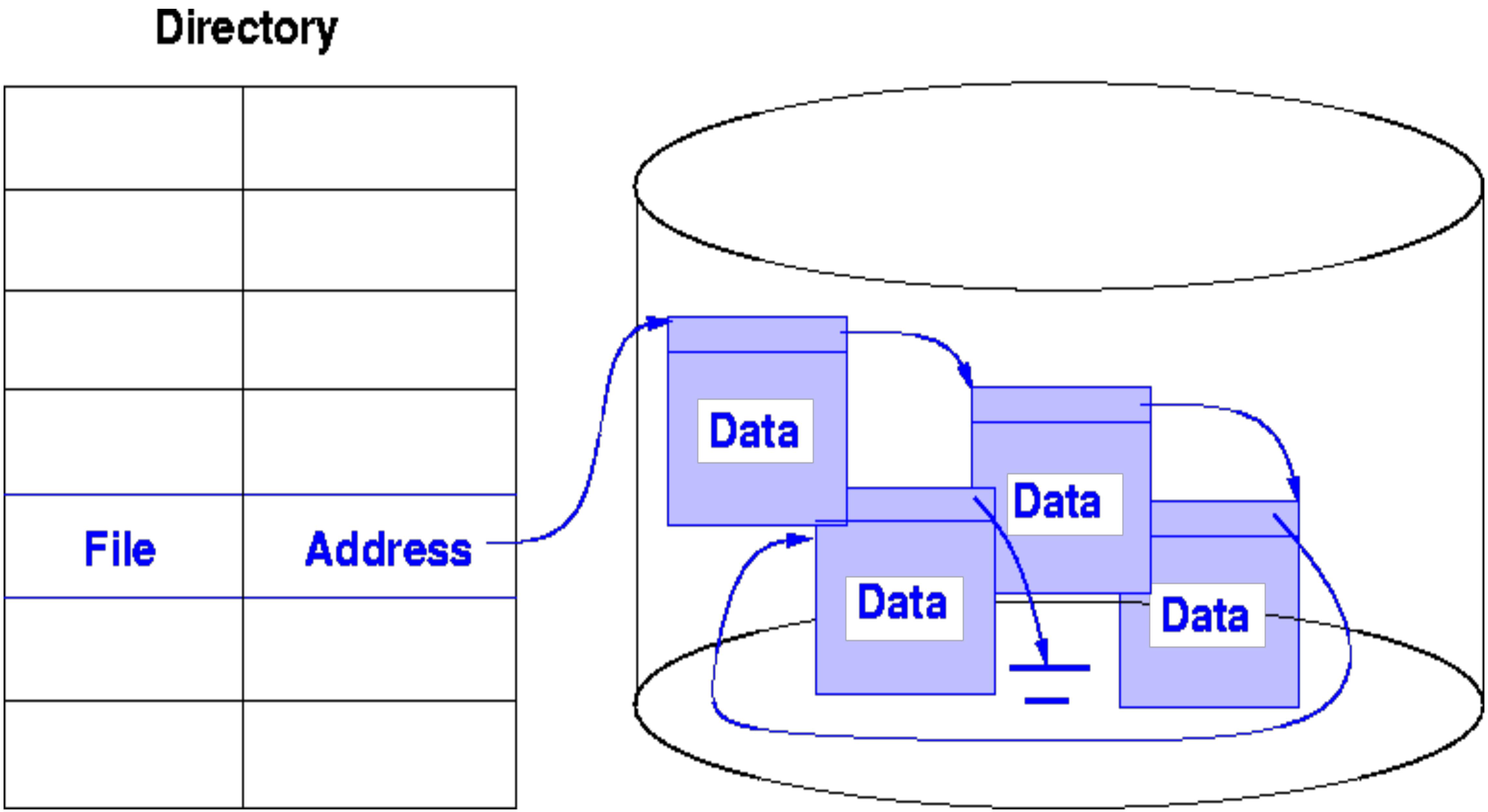
- Request in advance for the size of the file
- Search bit map or linked list to locate a space
- File header
  - first sector in file
  - number of sectors
- Pros
  - Fast sequential access
  - Easy random access
- Cons
  - External fragmentation
  - Hard to grow files

Linked Files

- File header points to 1st block on disk
- Each block points to next
- Pros
  - Can grow files dynamically
  - Free list is similar to a file
- Cons
  - random access: horrible
  - unreliable: losing a block means losing the rest



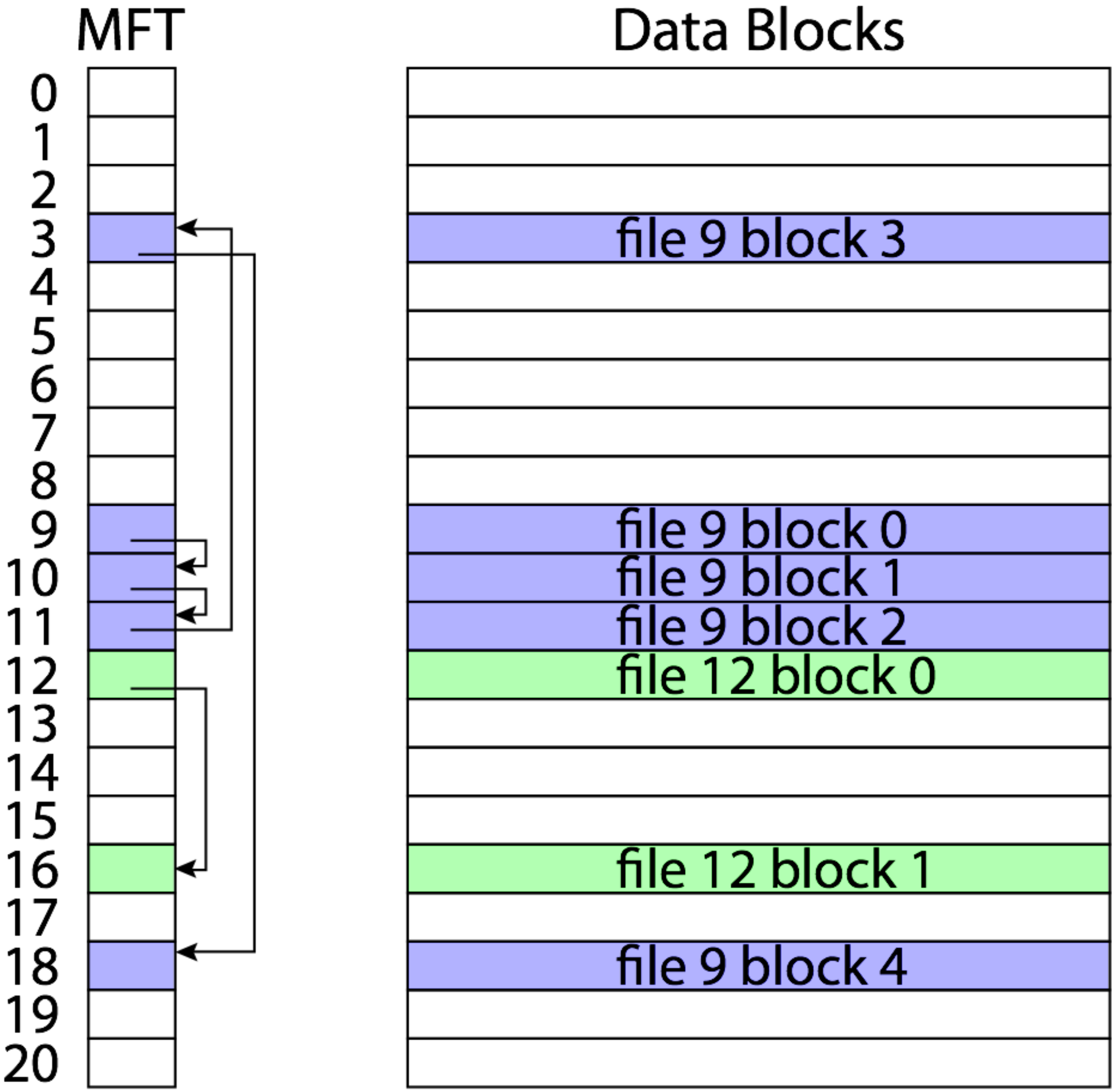
Linked Allocation



MS 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

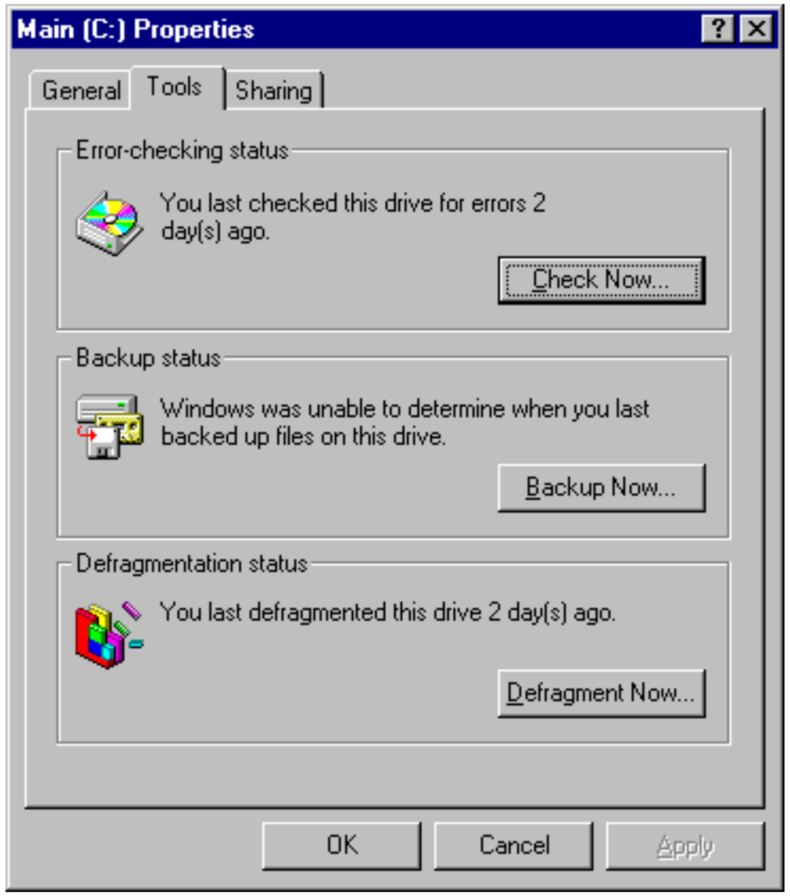
MS File Allocation Table (FAT)



MS File Allocation Table (FAT)

- Pros:
  - Easy to find free block
  - Easy to append to a file
  - Easy to delete a file

- Cons:
  - Small file access is slow
  - Random access is very slow
  - Fragmentation
    - File blocks for a given file may be scattered
    - Files in the same directory may be scattered
    - Problem becomes worse as disk fills

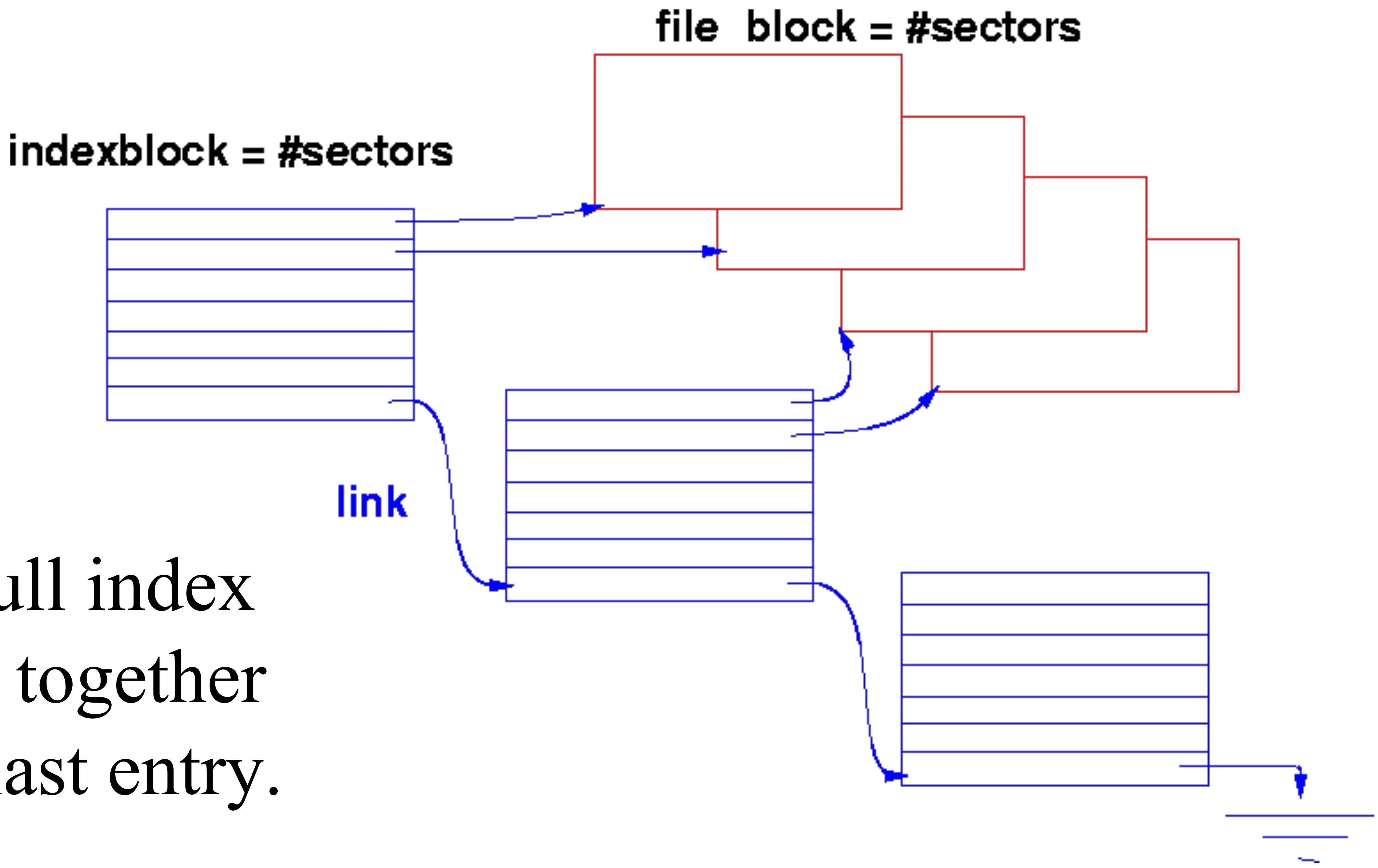


You last checked this drive for errors 2 day(s) ago.

Windows was unable to determine when you last backed up files on this drive.

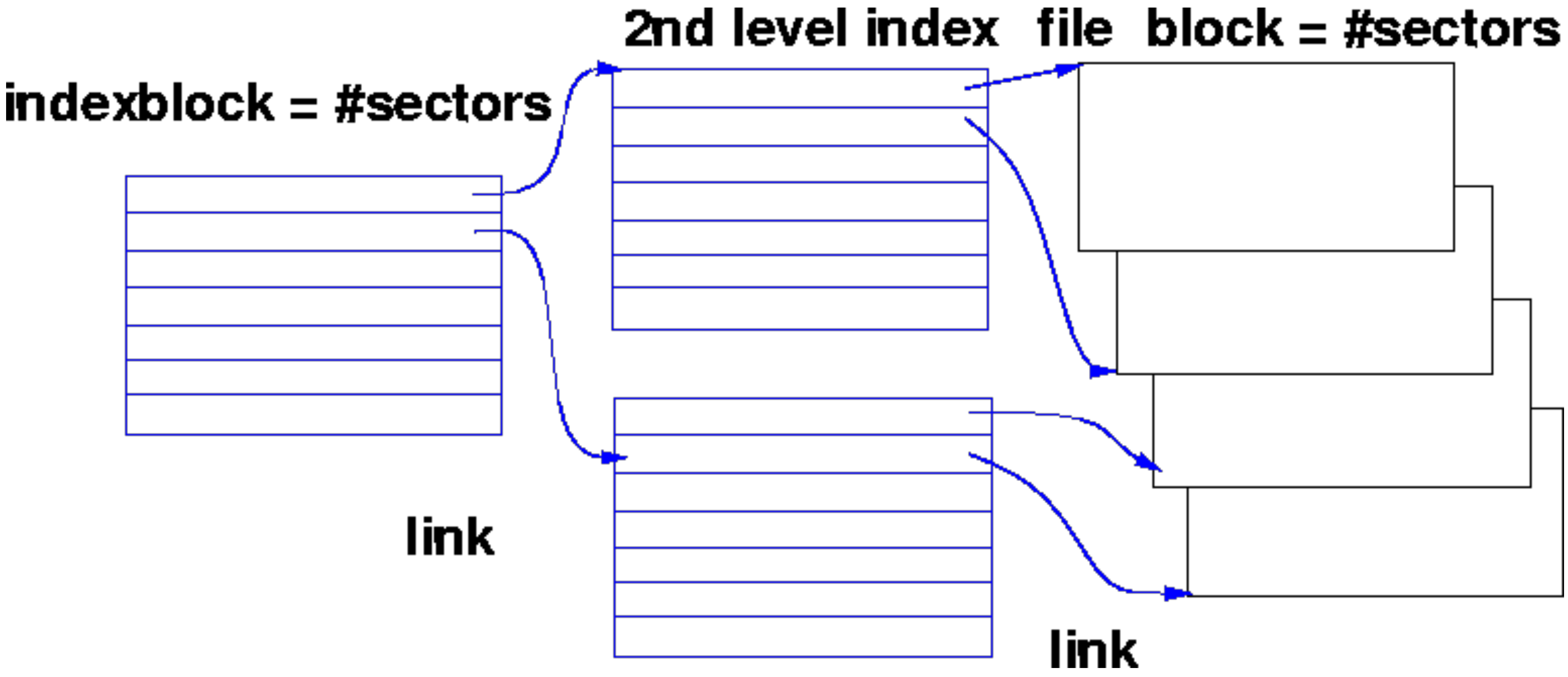
You last defragmented this drive 2 day(s) ago.

Indexed File Allocation



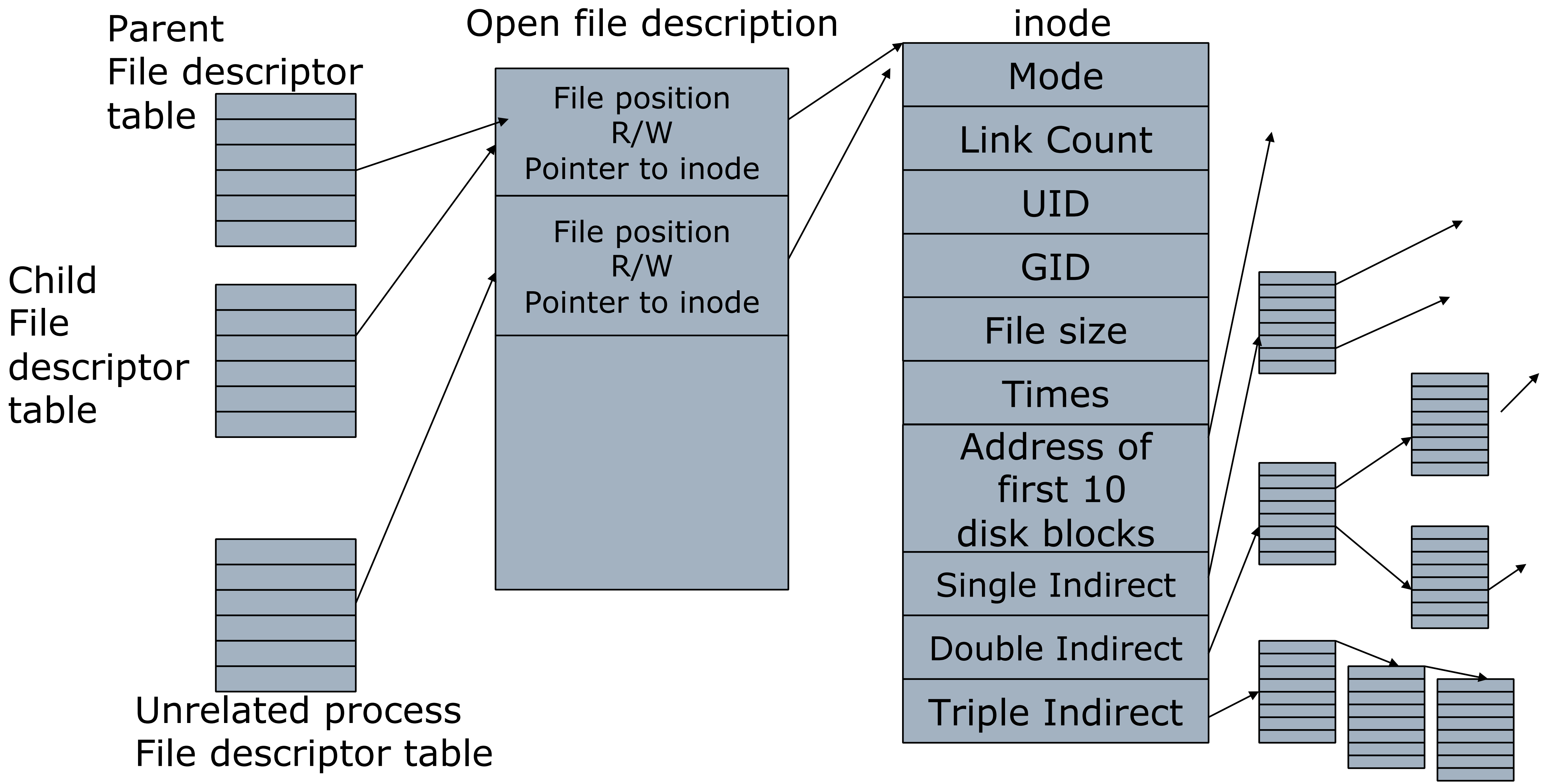
Link full index blocks together using last entry.

Multilevel Indexed Files



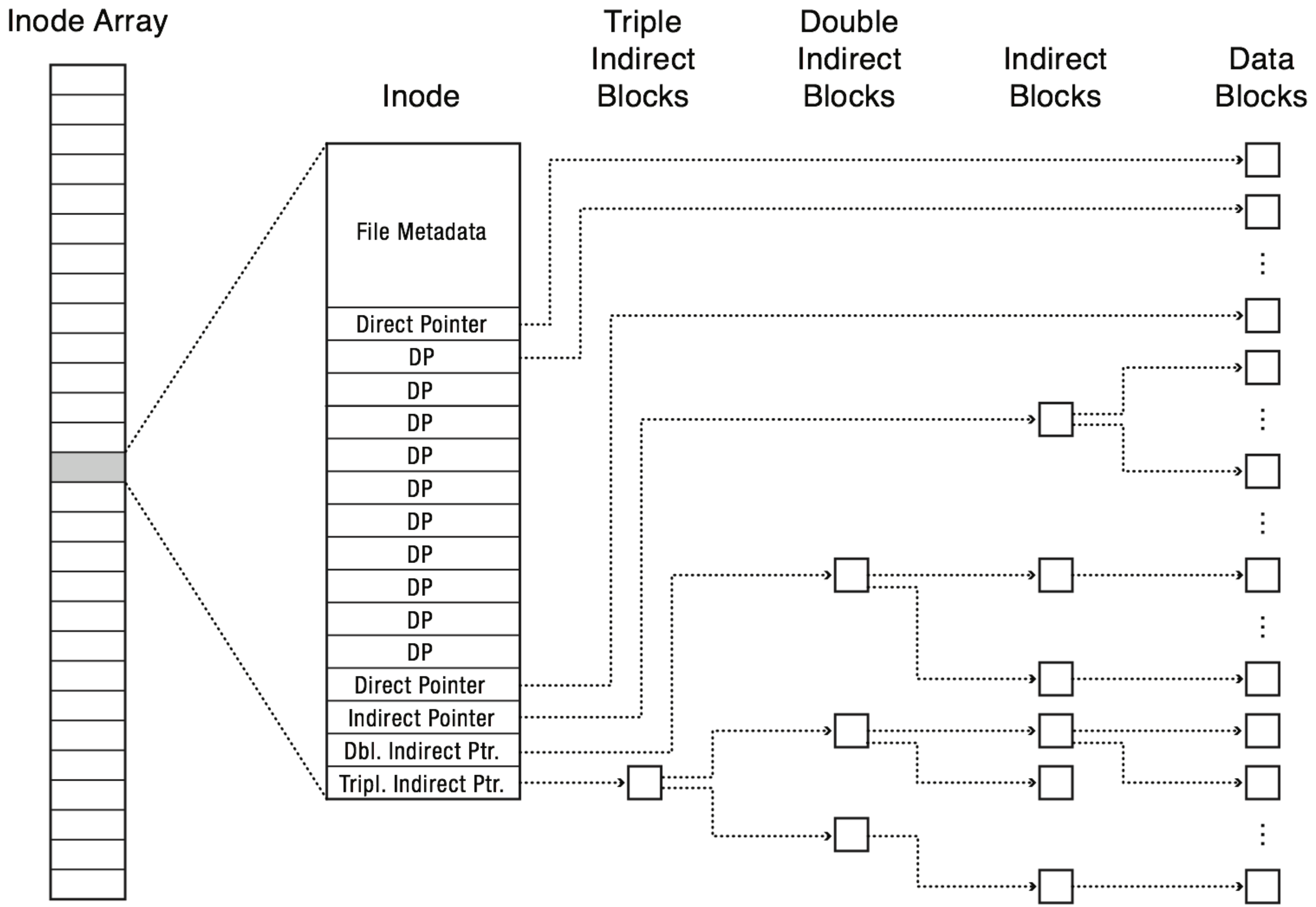
Multiple levels of index blocks

UNIX FS Implementation



Berkeley FFS / UNIX FS

Alternate figure, same basic idea



Berkeley FFS / UNIX FS

- “Fast File System”
- inode table
  - Analogous to FAT table
- inode
  - Metadata
    - File owner, access permissions, access times, …
  - Set of 12 data pointers
    - With 4KB blocks => max size of 48KB files
  - Indirect block pointers
    - pointer to disk block of data pointers
    - w/ indirect blocks, we can point to 1K data blocks => 4MB (+48KB)
  - … but why stop there??

Berkeley FFS / UNIX FS

- Doubly indirect block pointer
  - w/ doubly indirect blocks, we can point to 1K indirect blocks
  - => 4GB (+ 4MB + 48KB)
- Triply indirect block pointer
  - w/ triply indirect blocks, we can point to 1K doubly indirect blocks
  - 4TB (+ 4GB + 4MB + 48KB)

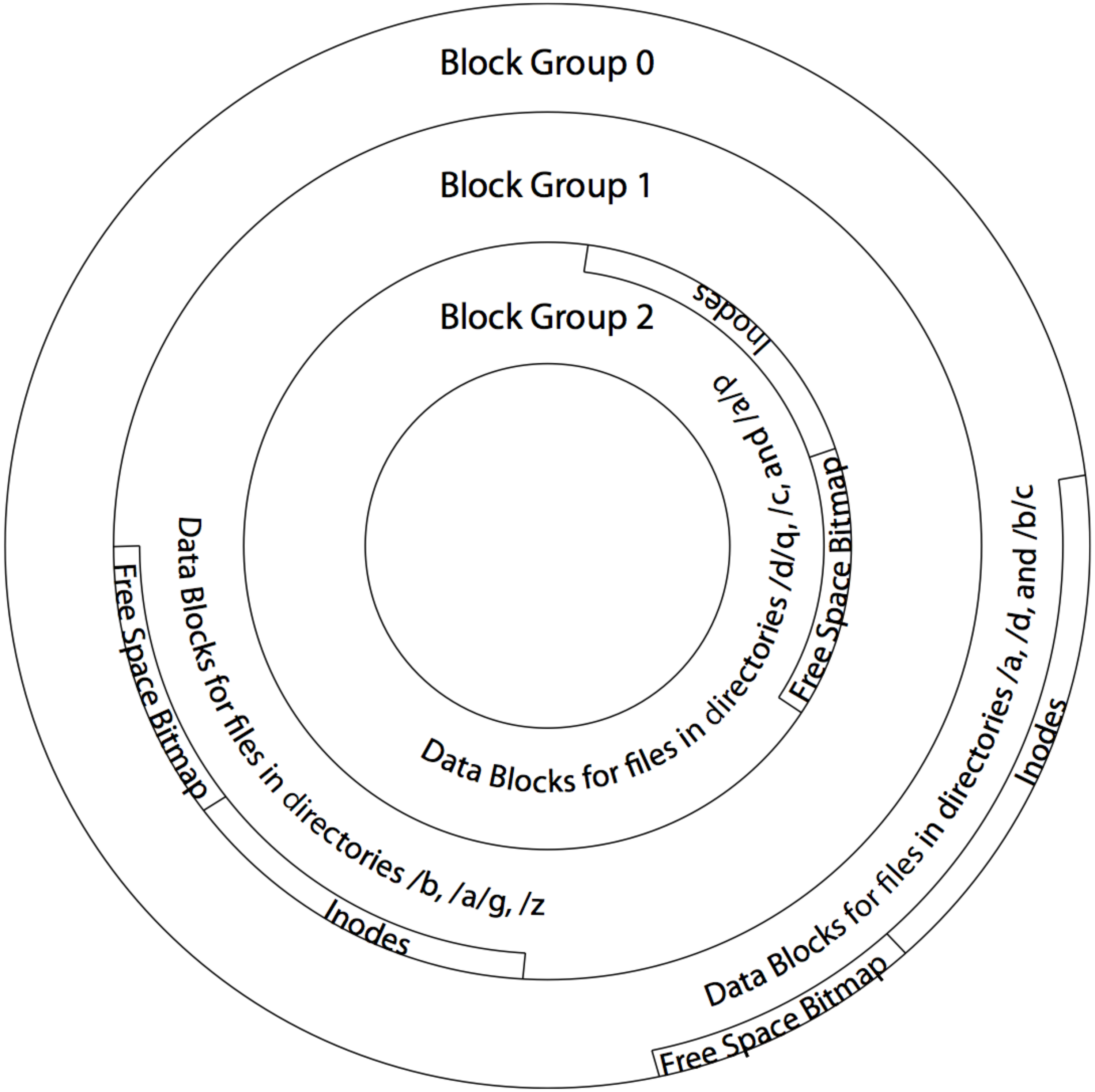
Berkeley FFS Asym. Trees

- Indirection has a cost. Only use if needed!
- 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

Berkeley FFS Locality

- How does FFS provide 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

Berkeley FFS Locality

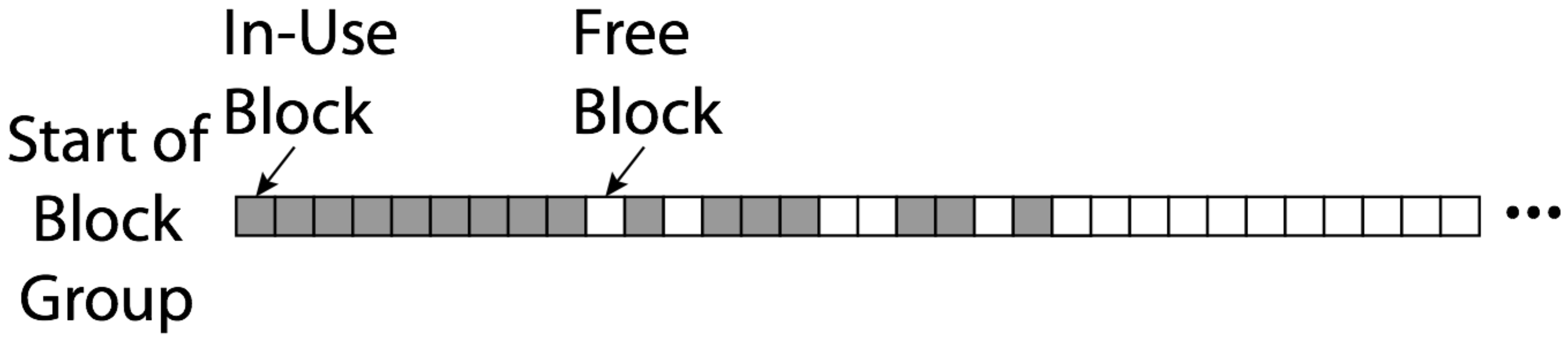


Berkeley FFS Locality

- How does FFS provide 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
  - Property: Small files may be a little fragmented, but large files will be contiguous

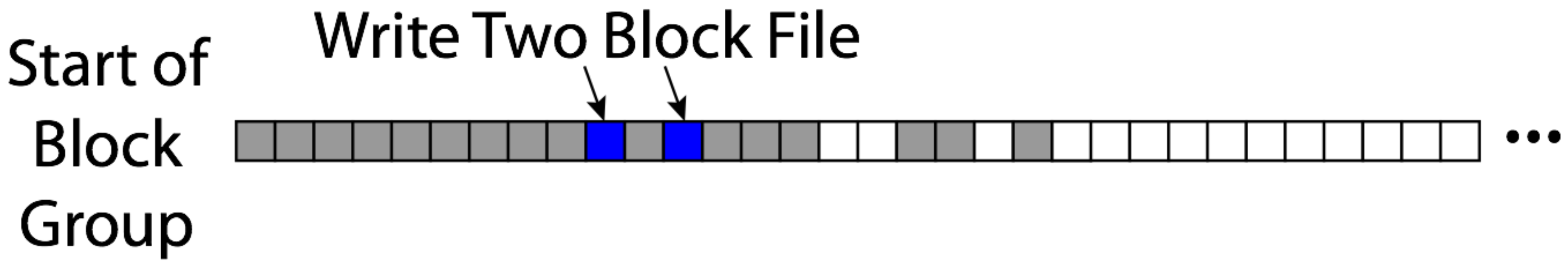
Berkeley FFS Locality

"First Fit" Block Allocation:



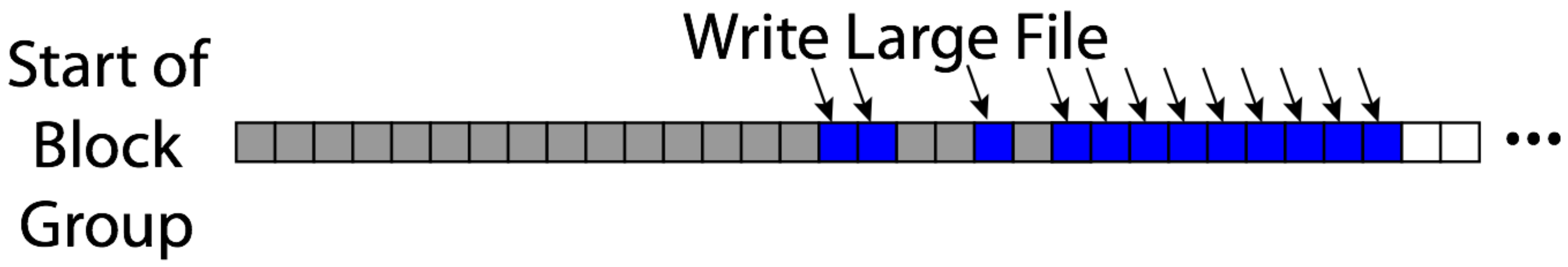
Berkeley FFS Locality

"First Fit" Block Allocation:



Berkeley FFS Locality

"First Fit" Block Allocation:



Berkeley FFS / UNIX FS

- 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)
  - Need to reserve 10-20% of free space to prevent fragmentation

Linux Filesystems

- The ext family of filesystems leverage many of the same concepts.
  - ext ('92): introduces VFS support, 2GB max FS size
  - ext2 ('93): introduces attributes and symbolic links, max file size is 2 GB and 2 TB FS, reserved disk space for root
  - ext3 ('01): introduces journaling, supports 2^32 blocks (up to max file of 2 TB, FS of 32 TB)
  - ext4 ('08): 2^48 block addressing, extent support

File Systems In Practice

	FAT	Berkeley FFS (Unix FS)	NTFS
Index structure	Linked list	Tree (fixed, assym)	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