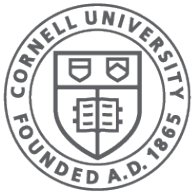
File Systems (II)

(Chapters 39-43,45)

CS 4410

Operating Systems



Cornell CIS

COMPUTING AND INFORMATION SCIENCE

[R. Agarwal, L. Alvisi, A. Bracy, M. George, F.B. Schneider, E. Sirer, R. Van Renesse]

File System Operations

- Create a file
- Write to a file
- Read from a file
- Seek to somewhere in a file
- Delete a file
- Truncate a file

File Storage Layout Options

- ✓ Contiguous allocation

  All bytes together, in order

- ✓ Linked-list

  Each block points to the next block

- Indexed structure (FFS)

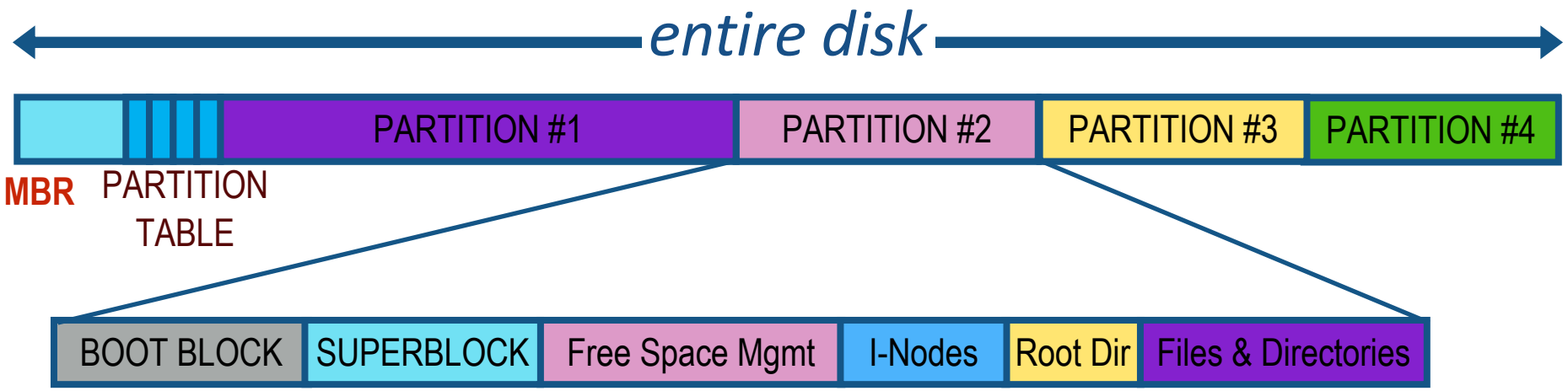
  Index block points to many other blocks

- Log structure

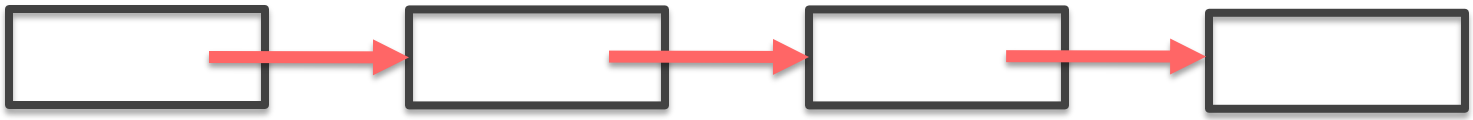
  Sequence of segments, each containing updated blocks

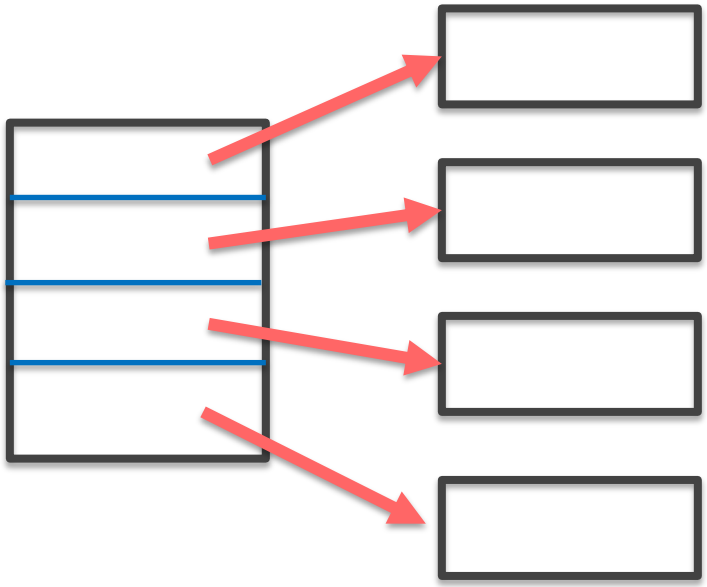
Recall …

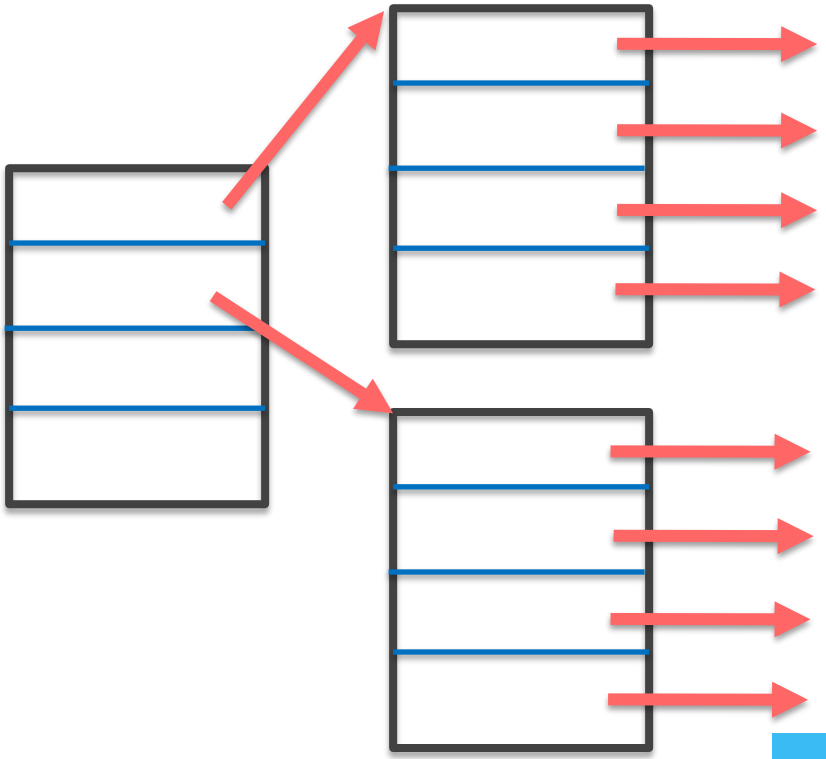
- File System is stored on *disks*
  - sector 0 of disk called Master Boot Record (MBR)
  - end of MBR: partition table (partitions' start & end addrs)
  - Remainder of disk divided into *partitions*.
    - Each partition starts with a *boot block*
    - Boot block loaded by MBR and executed on boot
    - Remainder of partition stores file system.



Cost Accounting: Access Index







Fast File System (FFS)

[mid 80’s]

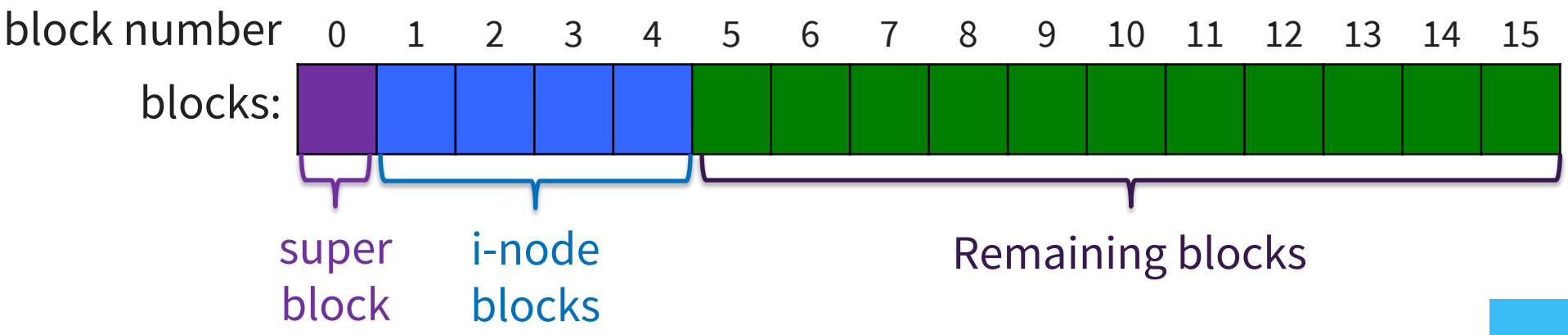
UNIX Fast File System

Tree-based, multi-level index

FFS Superblock

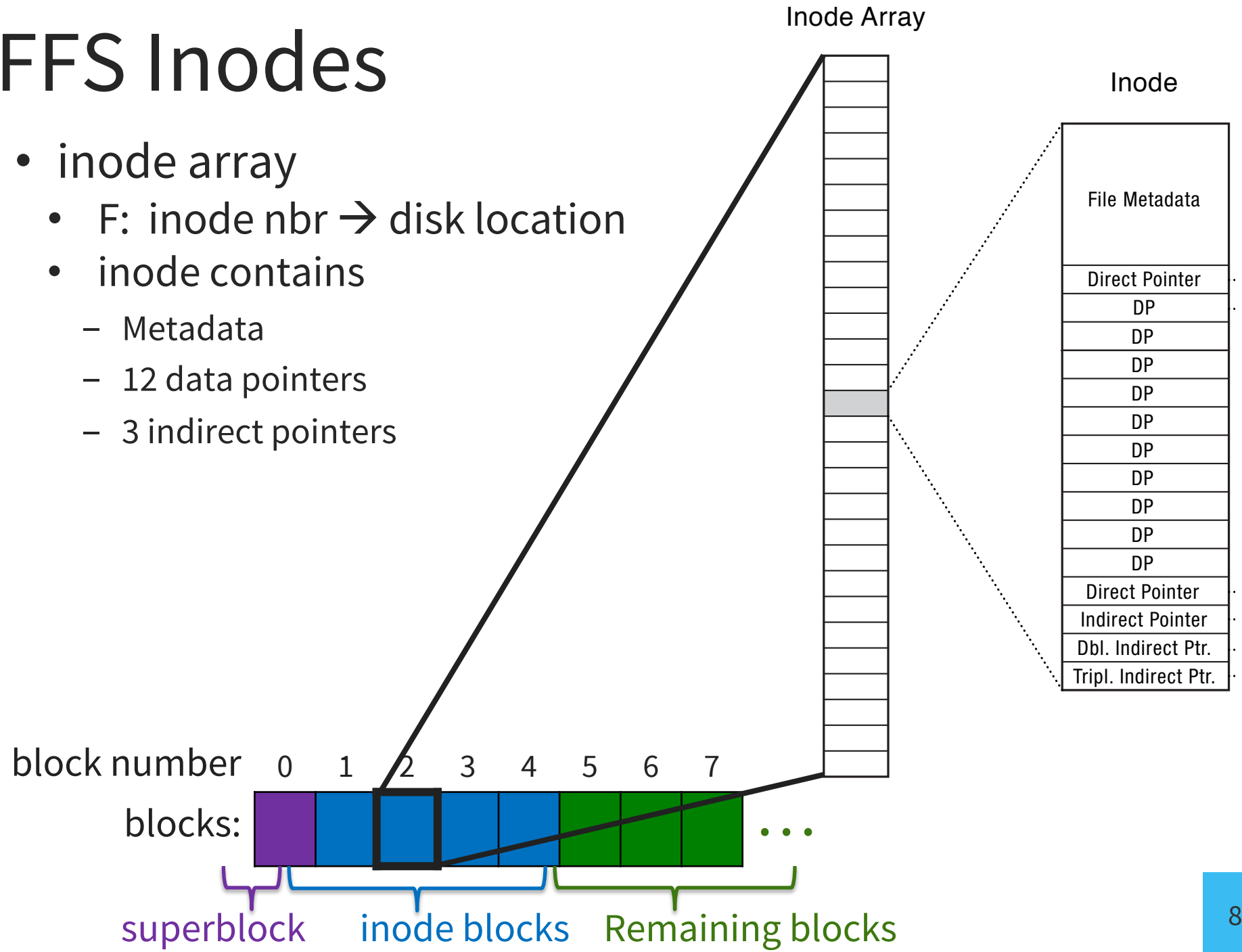
Identifies file system's key parameters:

- type
- block size
- inode array location and size
  (or analogous structure for other FSs)
- location of free list

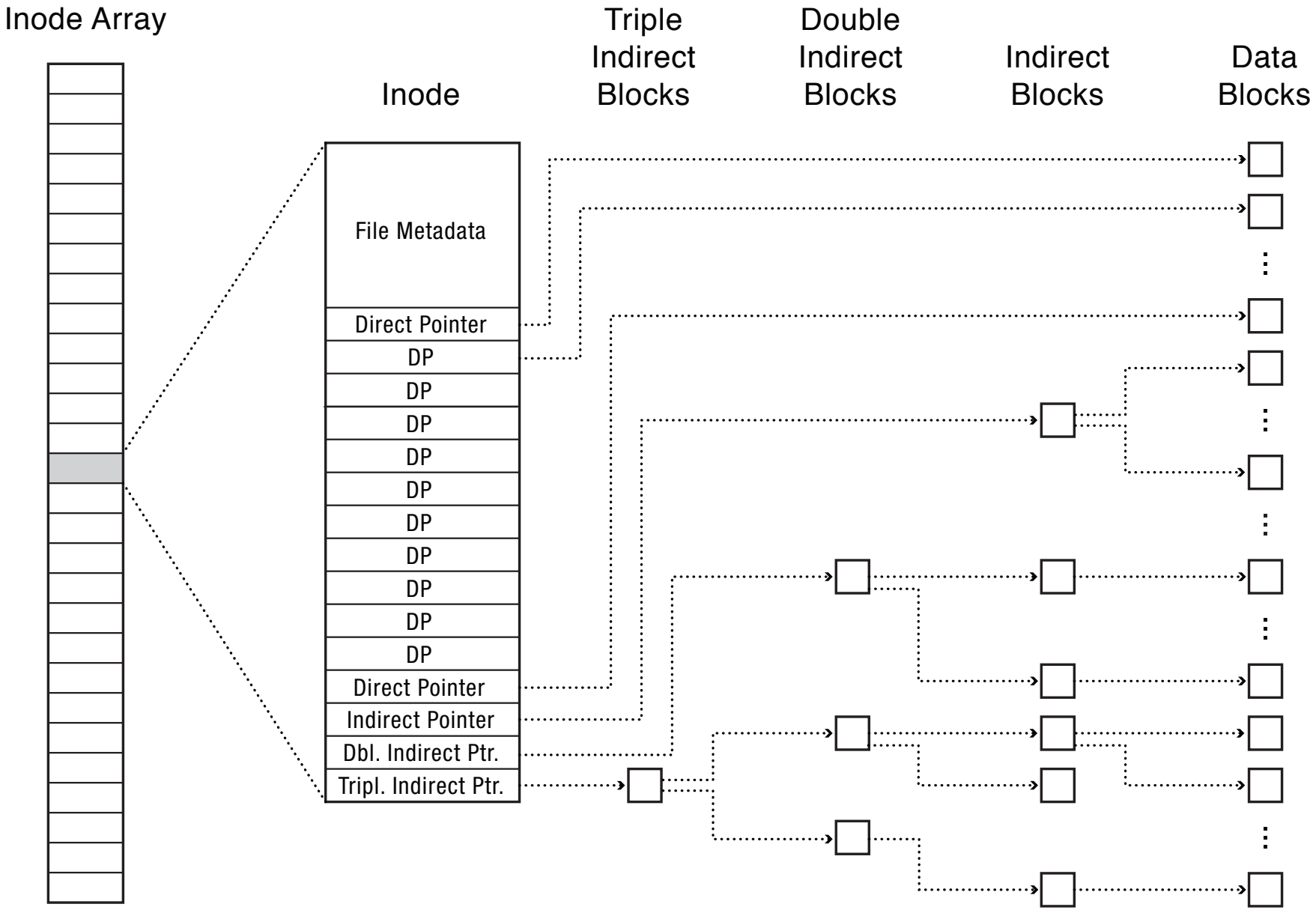


FFS Inodes

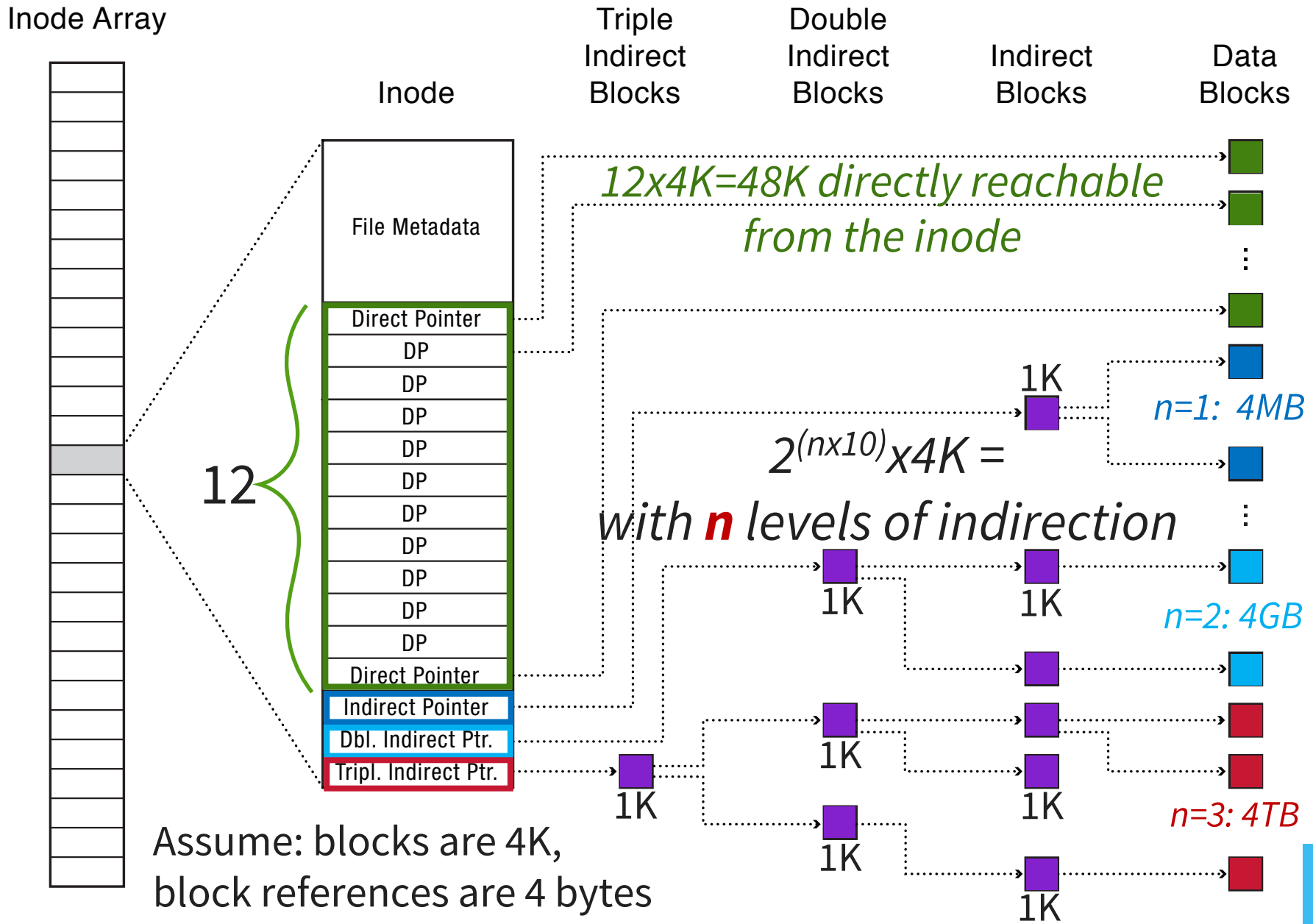
- inode array
  - F: inode nbr → disk location
  - inode contains
    - Metadata
    - 12 data pointers
    - 3 indirect pointers



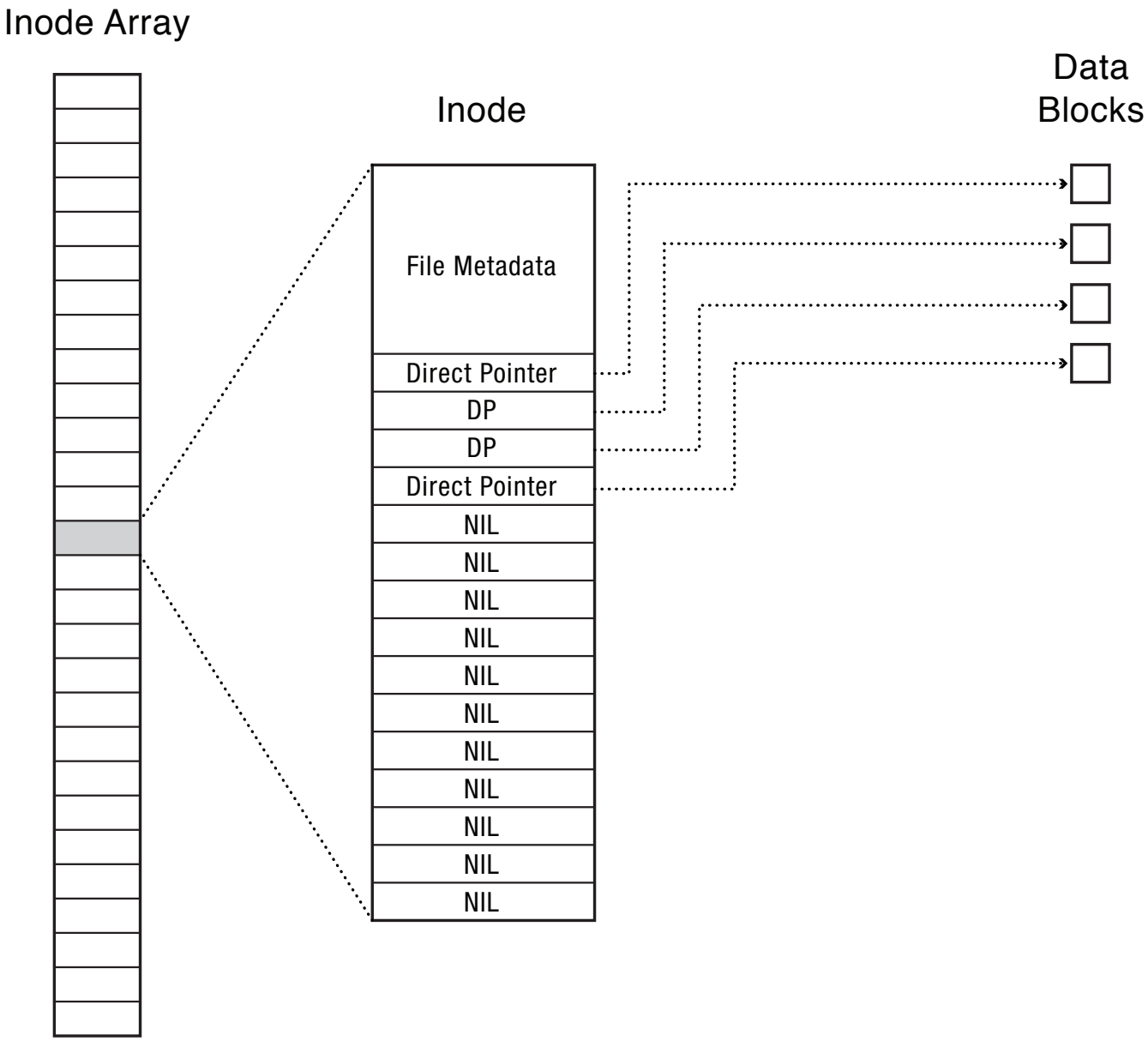
FFS: Index Structures



FFS: Index Structures



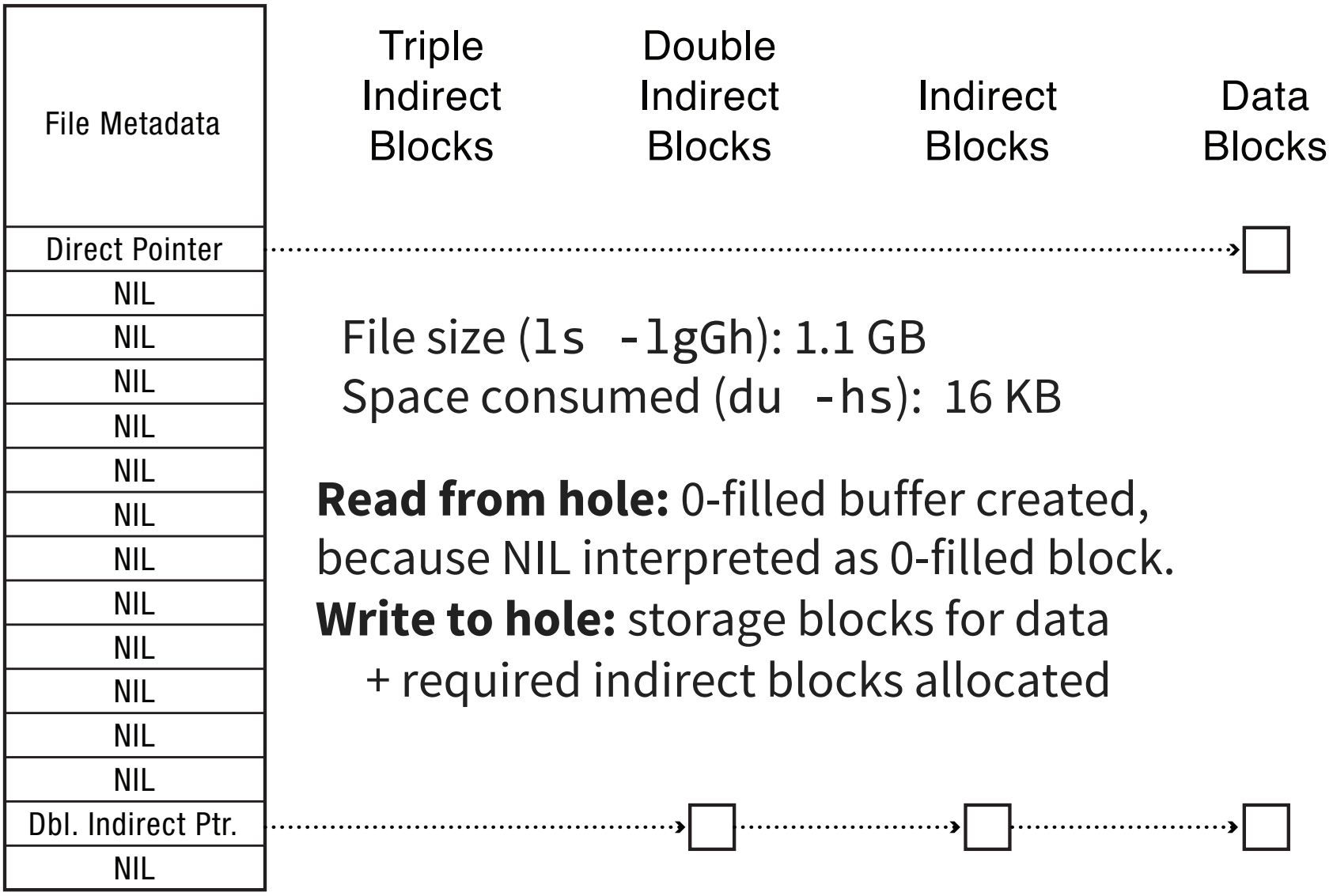
Small Files in FFS



all blocks reached via direct pointers

Sparse Files in FFS

Example sparse file:
2 x 4 KB blocks: 1 @ offset 0
1 @ offset 2^{30}



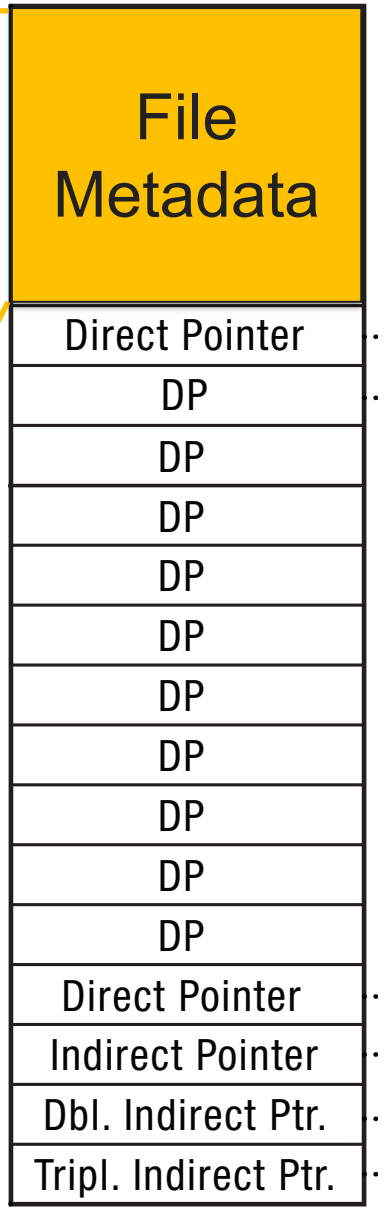
File size (`ls -lgGh`): 1.1 GB
Space consumed (`du -hs`): 16 KB

Read from hole: 0-filled buffer created, because NIL interpreted as 0-filled block.

Write to hole: storage blocks for data + required indirect blocks allocated

What else is in an inode?

- Type
  - ordinary file
  - directory
  - symbolic link
  - special device
- Size of the file (in #bytes)
- # links to the i-node
- Owner (user id and group id)
- Protection bits
- Times: creation, last accessed, last modified



Notable Characteristics of FFS

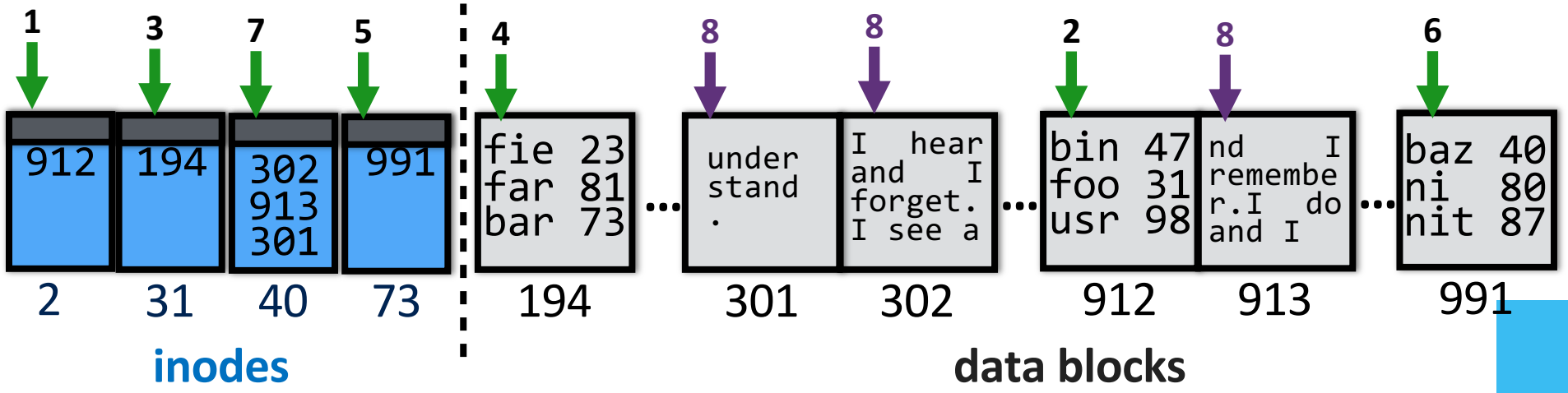
- Tree Structure
  - efficiently find any block of a file
- High Degree (or fan out)
  - minimizes number of seeks
  - supports sequential reads & writes
- Fixed Structure
  - implementation simplicity
- Asymmetric
  - not all data blocks are at the same level
  - supports large files
  - small files don’t pay large overheads

FFS: Steps to reading /foo/bar/baz

Read & Open:

(1) inode #2 (root always has inumber 2), find root's blocknum (912)
(2) root directory (in block 912), find foo's inumber (31)
(3) inode #31, find foo's blocknum (194)
(4) foo (in block 194), find bar's inumber (73)
(5) inode #73, find bar's blocknum (991)
(6) bar (in block 991), find baz's inumber (40)
(7) inode #40, find data blocks (302, 913, 301)
(8) data blocks (302, 913, 301)

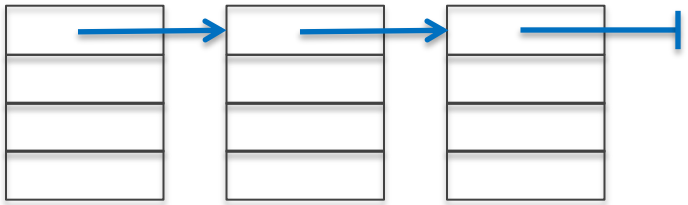
Caching allows first few steps to be skipped



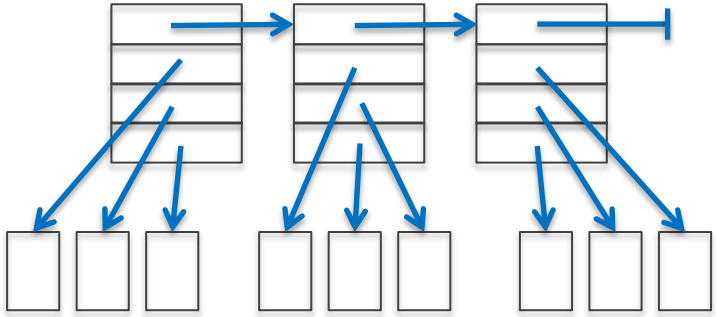
Free List: Blocks not in use

Possible data structures:

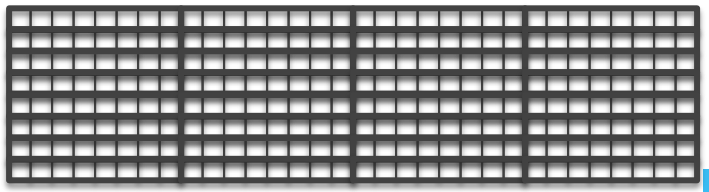
1. linked list of free blocks
   - inefficient (why?)



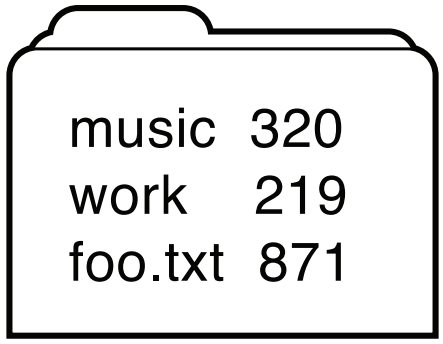
2. linked list of metadata blocks that in turn point to free blocks
   - simple and efficient



3. bitmap
   - good for contiguous allocation



FFS Directory Structure



Originally: array of 16 byte entries

- 14 byte file name
- 2 byte i-node number

Now: linked lists. Each entry contains:

- 4-byte inode number
- Length of name
- Name (UTF8 or some other Unicode encoding)

First entry is ".", points to self

Second entry is "..", points to parent inode

File System API: File creation

Creating and deleting files

- creat(): creates
  1. a new file with some metadata; and
  2. a name for the file in a directory
- link() creates a *hard link*–a new name for the same underlying file, and increments link count in inode
- unlink() removes a name for a file from its directory and decrements link count in inode. If last link, file itself and resources it held are deleted

Hard & Soft Links

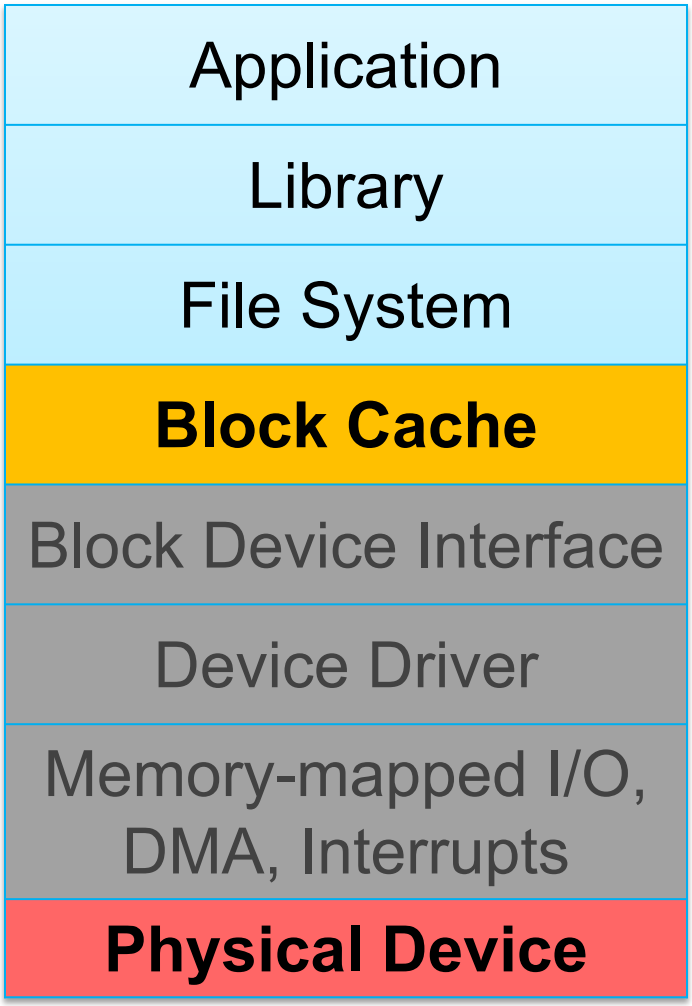
- **hard link**: a mapping from name to a low-level name (inode).
- **soft (sym) link:** a mapping from a file name to a file name
  - … a file that contains the name of another file
  - use as *alias*: a soft link continues to remain valid when the (path of) the target file name changes

Improving Performance

- Allocate blocks that will be accessed together in the same cylinder group.
  - Result: shorter seek times.
- Buffer blocks in memory.
  - Combines writes prior to seek
  - Single fetch for reads by many processes
  - Allows re-ordering of disk access
- Pre-fetch for read
  - Eliminates latency from critical path

BUT can introduce inconsistency!

- Write does not make data persistent
- Result of write can be lost?
- Failures cause on-disk data structure corruption.



Tolerating Crashes

If a processor crashes then only some blocks on a disk might get updated.

- Data is lost
- On disk data structures might become inconsistent. E.g.
  - » starting state: A, B
  - » update: A → A' and B → B'
  - » Possible result when there is a crash: A, B' or A', B

Solutions:

- Add fsync: programmer forces writes to disk
- Detect and recover
- Fault-tolerant disk update protocols

File System Consistency Checks

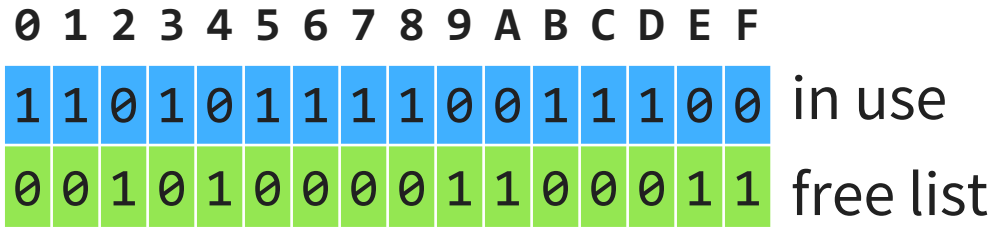
fsck (UNIX) & scandisk (Windows)

Detection Algorithm for File Blocks:

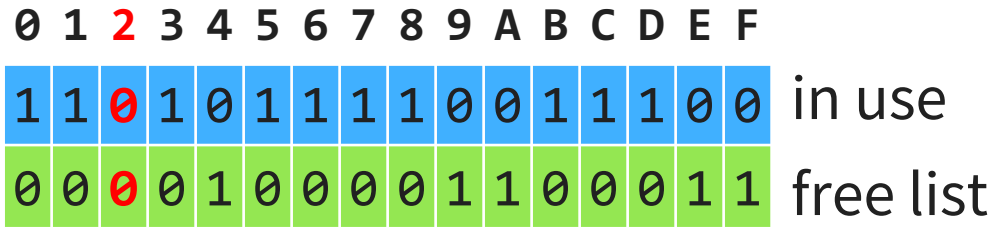
- Build table with info about each block
  - initially each block is unknown except superblock
- Scan through the inodes and the freelist
  - Keep track in the table
  - If block already in table, note error
- Finally, see if all blocks have been visited

Examples of Detection Tables

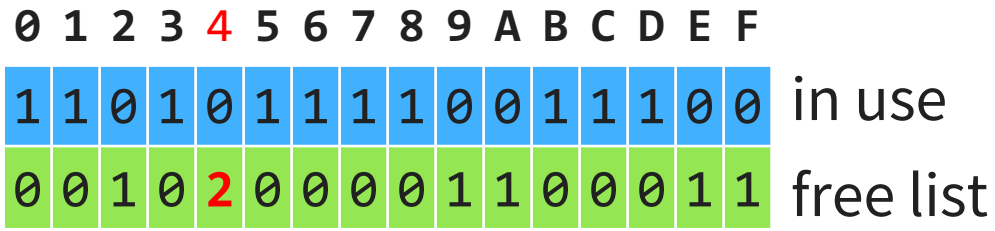
Consistent



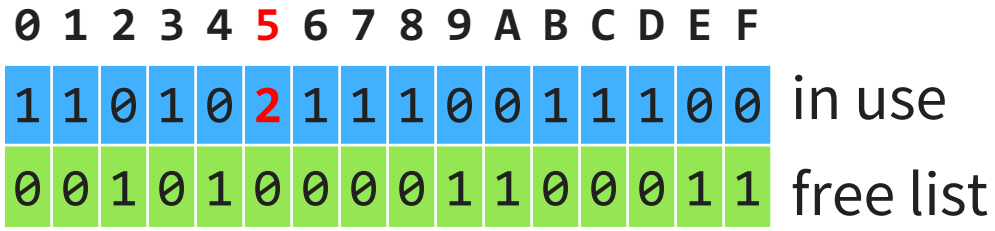
Missing Block 2
(add it to the free list)



Duplicate Block 4 in Free List
(rebuild free list)



Duplicate Block 5 in Data List (copy block and add it to one file)



Detection Algorithm for Directories

Use a per-file table instead of per-block

Parse entire directory structure, start at root

- Increment counter for each file you encounter
- This value can be >1 due to hard links
- Symbolic links are ignored

Compare table counts w/link counts in i-node

- If i-node count > our directory count (wastes space)
- If i-node count < our directory count (catastrophic)

Fault-tolerant Disk Update

Use Journaling (aka) Write-Ahead Logging

- **Idea**: Protocol where performing a **single** disk write causes multiple disk writes to take effect.

- **Implementation**: New on-disk data structure ("journal") with a sequence of blocks containing updates *plus* …

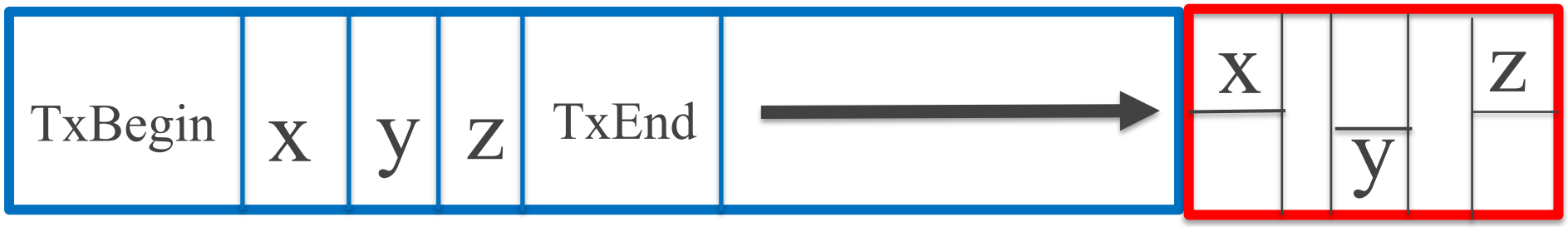
Journal-Update Protocol Step

write x; write y; write z

implemented by

- Append to journal: TxBegin, x, y, z
- Wait for completion of disk writes.
- Append to journal: TxEnd
- Wait for completion of disk write.
- Write x, y, z to final locations in file system

***called* checkpoint step**



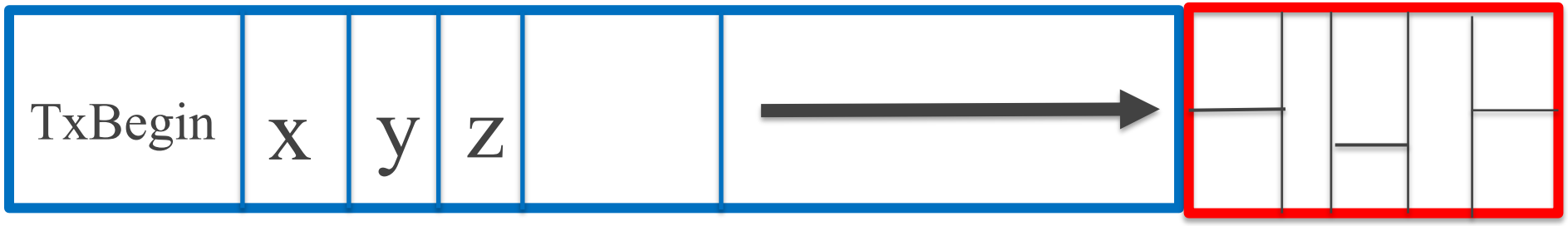
What if Crash?

write x; write y; write z

implemented by

- Append to journal: TxBegin, x, y, z
- Wait for completion of disk writes.
- Append to journal: TxEnd
- Wait for completion of disk write.
- Write x, y, z to final locations in file system.

crash!



Recovery protocol for TxBegin … TxEnd:

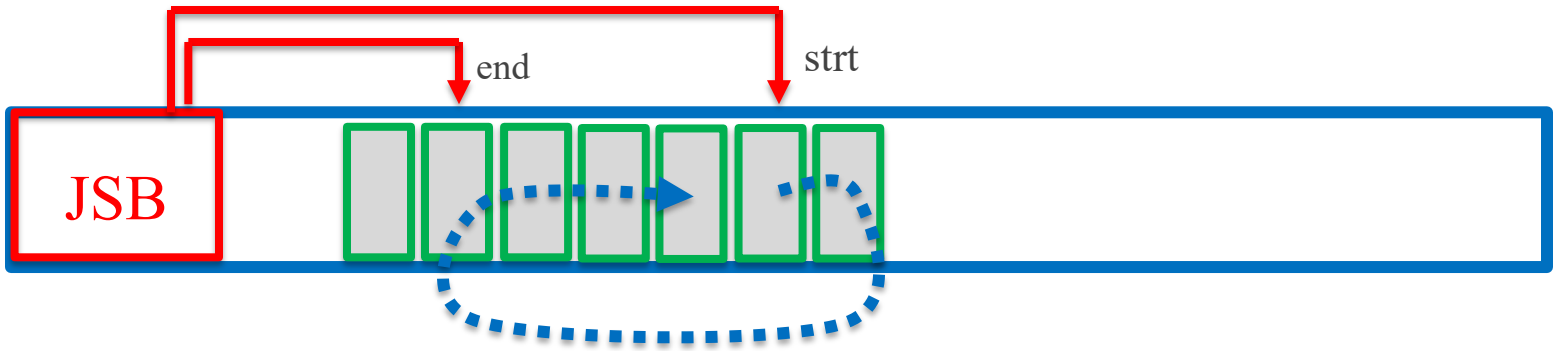
- **if** TxEnd present **then** redo writes to final locations following TxBegin
- **else** ignore journal entries following TxBegin

Full-Journal Recovery Protocol

- Replay journal from start, writing blocks as indicated by checkpoint steps.

<u>Infinite Journal → Finite Journal:</u>

- introduce **journal super block** (JSB) as first entry in journal: JSB gives start / end entries of journal.
- view journal as a circular log
- delete journal entry once writes in checkpoint step complete.



Performance Optimizations

- Eliminate disk write of TxEnd record.
  - Compute checksum of xxx in “TxBegin xxx TxEnd”
  - Include checksum TxBegin.
  - Recovery checks whether all log entries present.
- Eliminate disk write of xxx when data block
  - Step 1: Write data block to final disk location
  - Step 2: Await completion
  - Step 3: Write xxxx for meta-data blocks
  - Step 4: Await completion