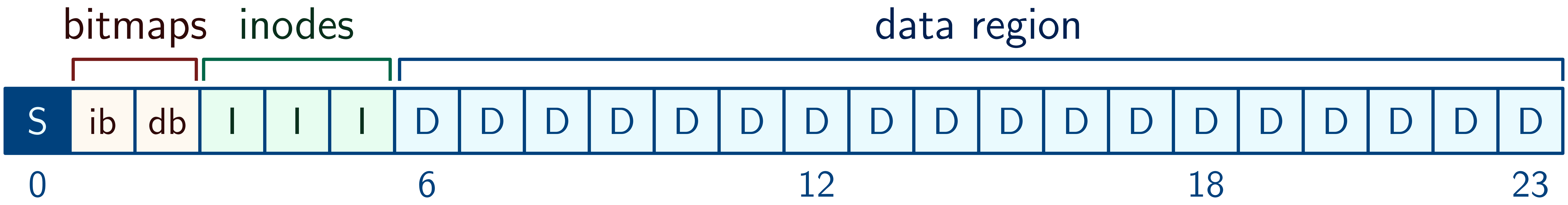
INF113: FFS, Crash Consistency

Kirill Simonov

05.11.2025

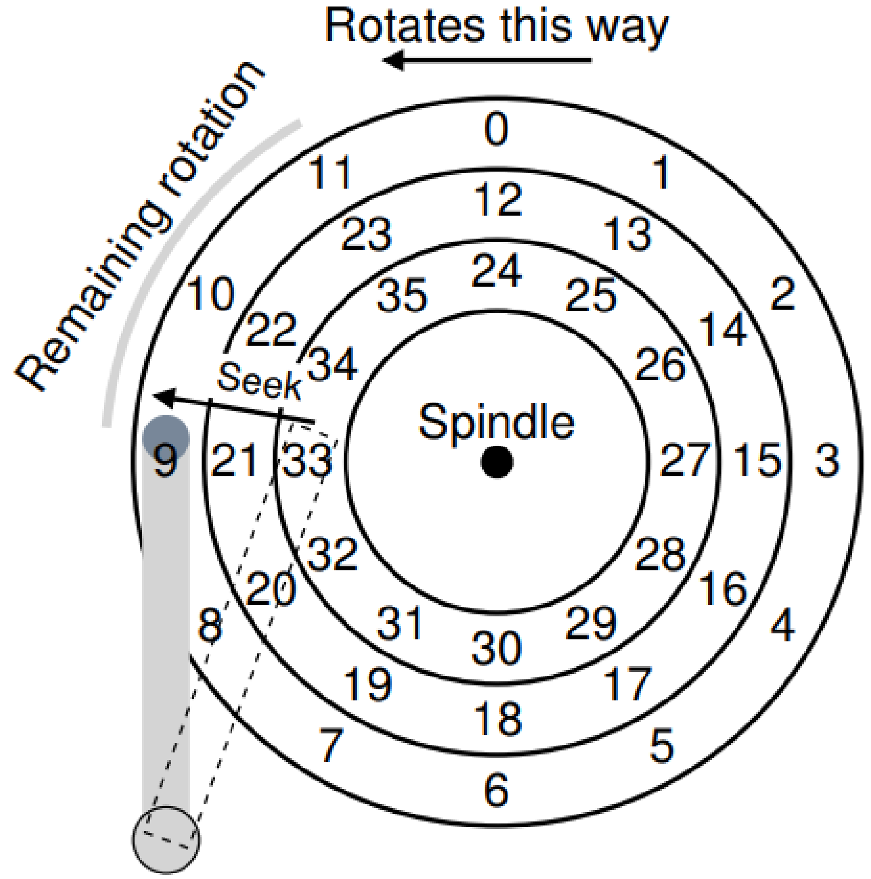
Reminder

- We described a very simple file system
- VSFS stores:
  - Superblock
  - Allocation bitmaps
  - Inode blocks
  - Data region
- Any other file system can be implemented by replacing the internal representation and `open`/`read`/`write`/...syscalls



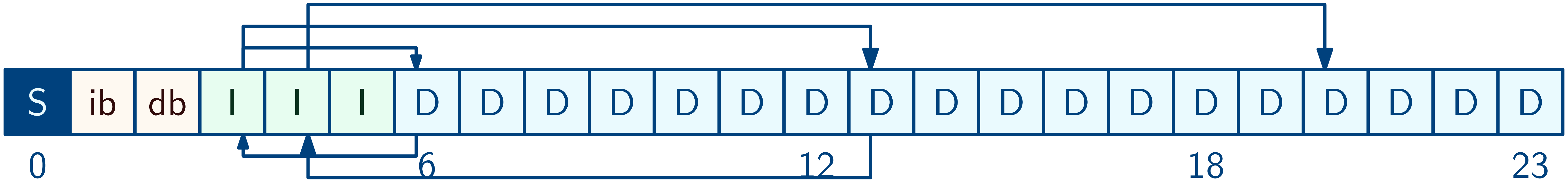
Locality

- Storage benefits from “locality”—storing related content close
- Example: hard disk drives
  - If two consecutives reads are far apart, HDD has a long seek to perform



- VSFS is not great for locality
  - Reading from a file: first inode contents, then data itself
  - But inode and its data blocks may be very far apart
- Even more jumps from determining the file’s inode number

`/foo/bar`: inode `/` → data `/` → inode `foo` → data `foo` → inode `bar` → data `bar`



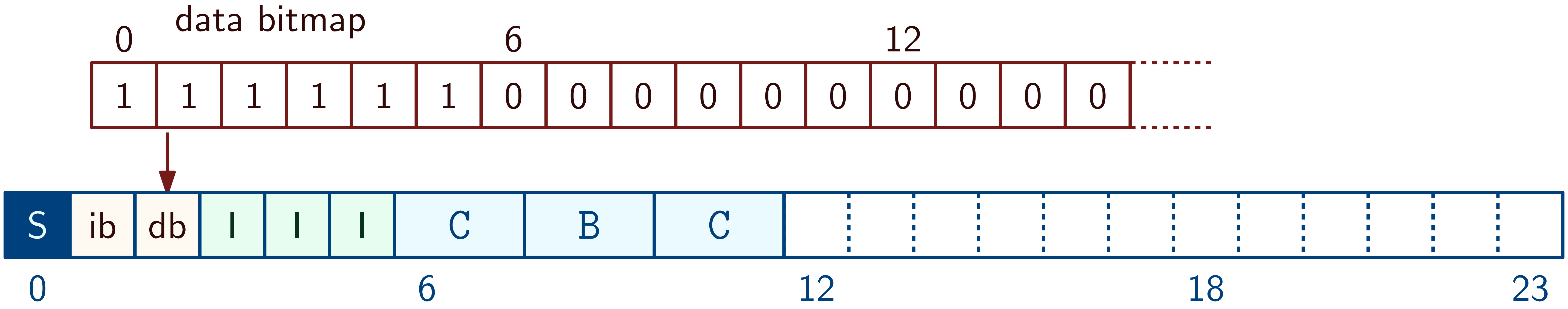
Fragmentation

- Data blocks of a file will not always be consecutive

  create `A` → create `B` → remove `A` → create `C` — `C` is split!

- "Allocate first available block" strategy will produce fragmented files
  - The more fragmented a file is, the slower it is to read (from HDD)

- **Defragmentation**: rearranging data blocks on the whole disk to make files continuous



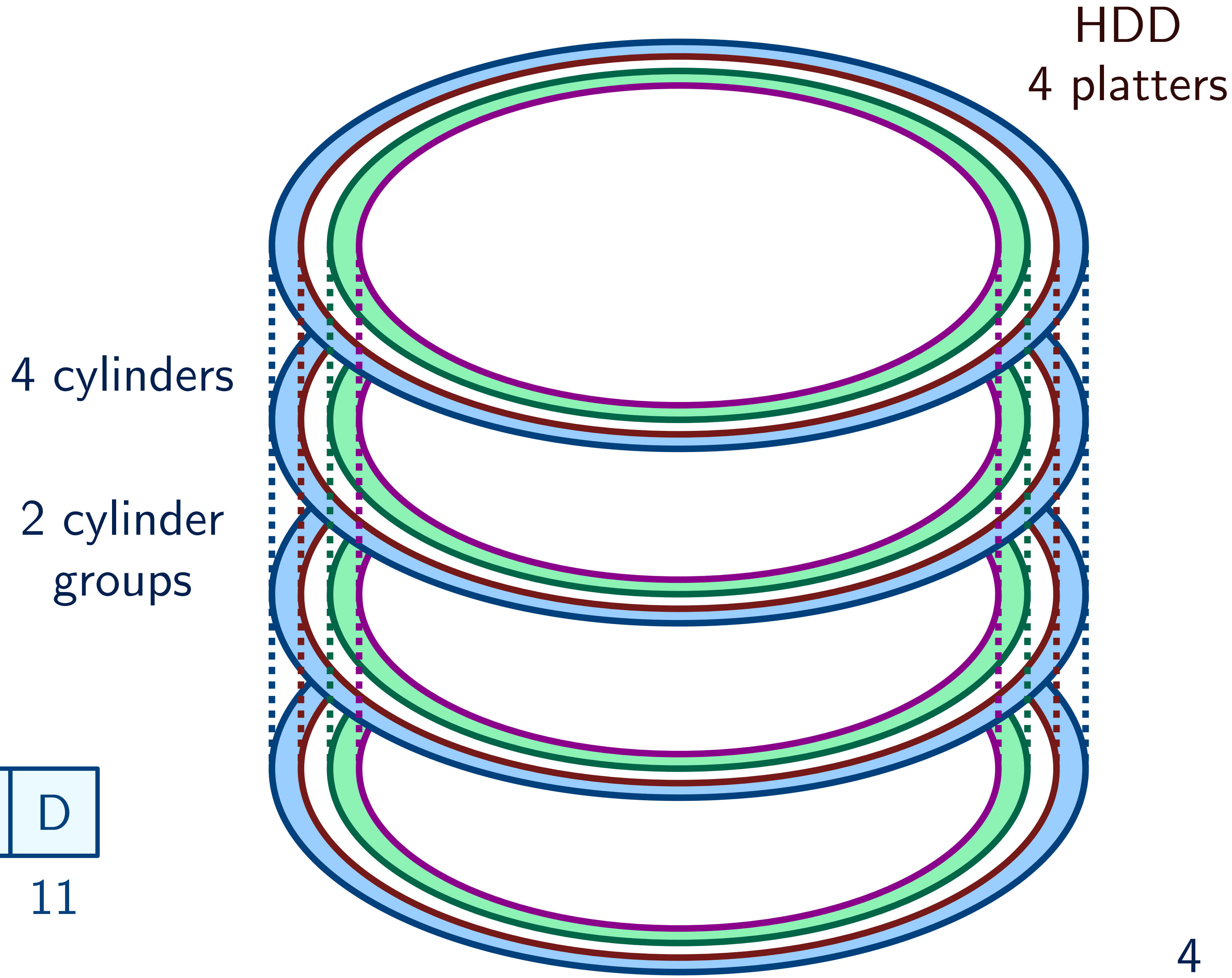
Cylinder groups

- HDD motivation: blocks in the same “cylinder” can be read in a sequence quickly

- Cylinder groups: batch a few close cylinders together

- Each group has the same internal structure as the whole VSFS:
  - Superblock
  - Allocation blocks
  - Inodes
  - Data blocks

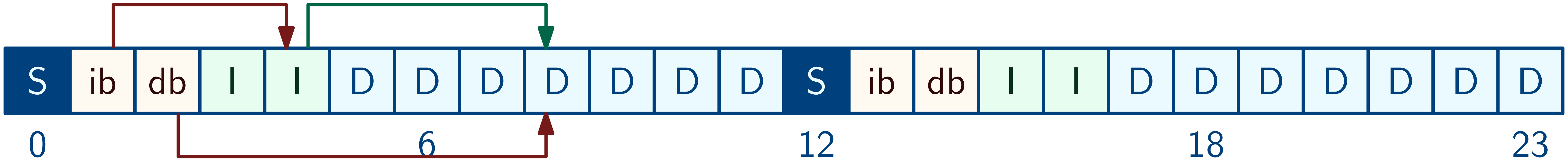
S	ib	db	I	I	D	D	D	D	D	D	D
0						6					11



Block groups

- Same idea without relying on physical structure of the disk
- Block group: a sequence of consecutive blocks on the disk
- Each group has its own
  - Superblock
  - Allocation blocks
  - Inodes
  - Data blocks
- Creating a file may happen within a single block group

  inode bitmap→ new inode → data bitmap → new data block



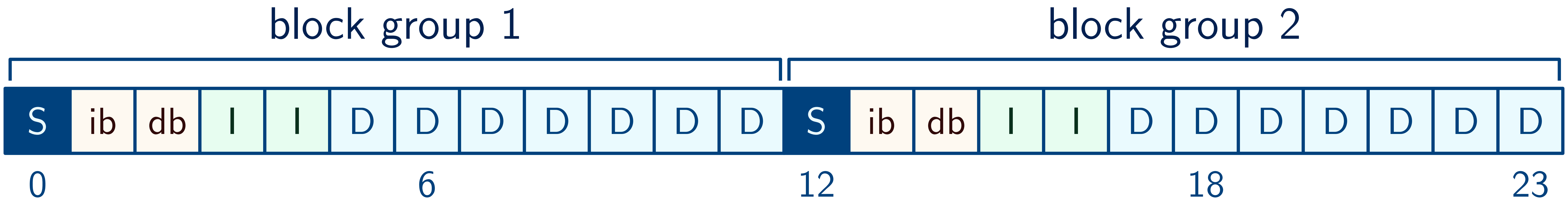
Fast File System (FFS)

- Designed for 4.2BSD (1983)

- Introduced block groups and other features
  - Long file names, symbolic links, atomic rename, . . .

- Policies for choosing the group:
  - Single file: Inode and data blocks in the same group
  - Directory and its files in the same group
  - Different directories far away
  - Large file: first few blocks in a single group

- Block groups are still present in modern file systems

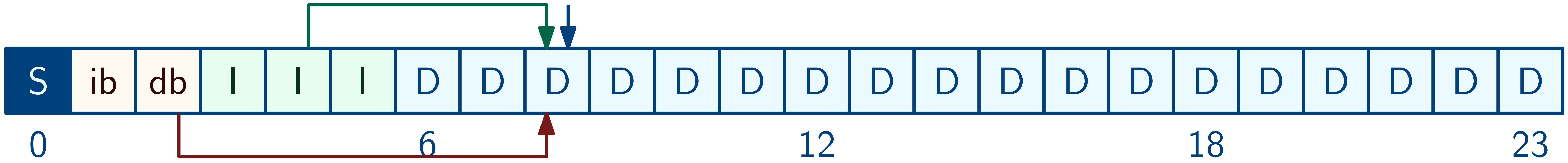


Crash consistency

- A single write request changes multiple on-disk structures
  - Write a new data block
  - Add its number to inode
  - Mark it allocated in the data bitmap
  - Potentially more: directory data, new indirect node, . . .

- In case of a crash, only some of these changes will be applied

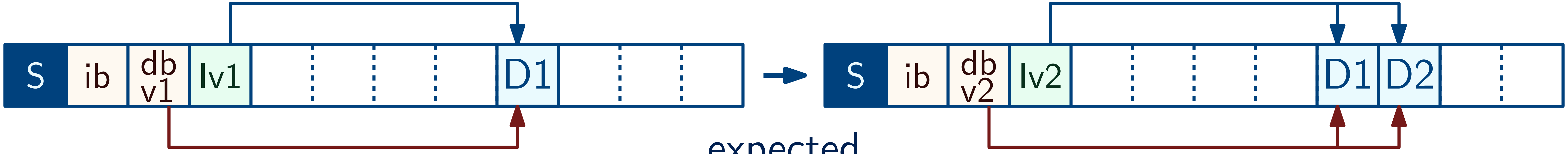
- This might not just lose user data, but also put the file system into an **inconsistent** state



Crash behaviors

Example: Append a data block to a file

1. Write new data block D2
2. Add reference to D2 to the inode: Iv1 → Iv2
3. Mark D2 allocated: db v1 → db v2



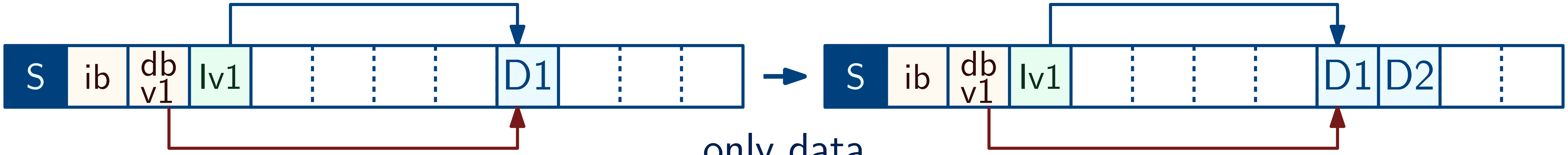
expected

Crash behaviors

Example: Append a data block to a file

1. Write new data block D2
2. Add reference to D2 to the inode: Iv1 → Iv2
3. Mark D2 allocated: db v1 → db v2

- Only step 1: D2 is written, but nothing points to it—same as no write



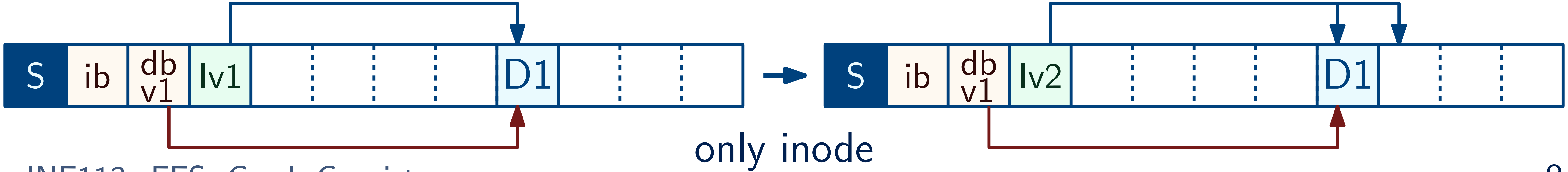
only data

Crash behaviors

Example: Append a data block to a file

1. Write new data block D2
2. Add reference to D2 to the inode: Iv1 → Iv2
3. Mark D2 allocated: db v1 → db v2

- Only step 1: D2 is written, but nothing points to it—same as no write
- Only step 2: Inode points to the block, but the data was not written!
  - Inconsistency: Inode thinks D2 is part of the file, data bitmap thinks it's free



Crash behaviors

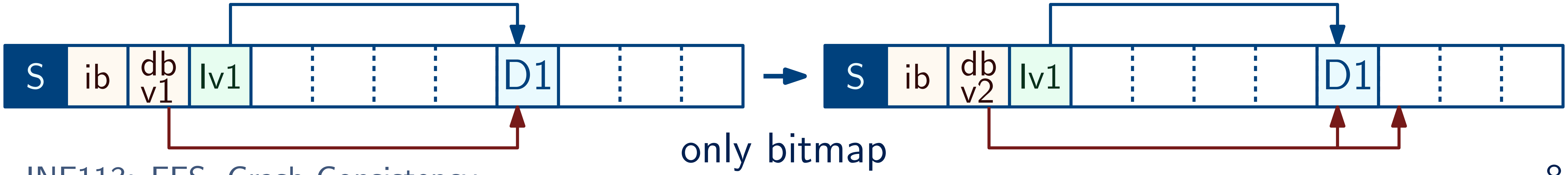
Example: Append a data block to a file

1. Write new data block D2
2. Add reference to D2 to the inode: Iv1 → Iv2
3. Mark D2 allocated: db v1 → db v2

- Only step 1: D2 is written, but nothing points to it—same as no write

- Only step 2: Inode points to the block, but the data was not written!
  - Inconsistency: Inode thinks D2 is part of the file, data bitmap thinks it's free

- Only step 3: Data bitmap marks D2 as allocated, but no file uses the block
  - Inconsistency: Data bitmap marks more blocks than in use → space leak

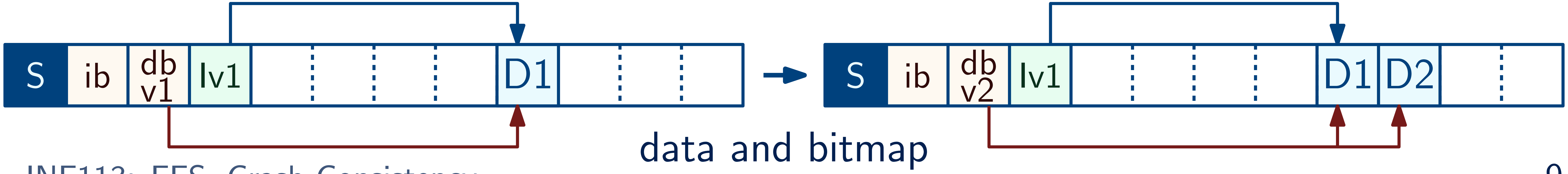


Crash behaviors—part 2

Example: Append a data block to a file

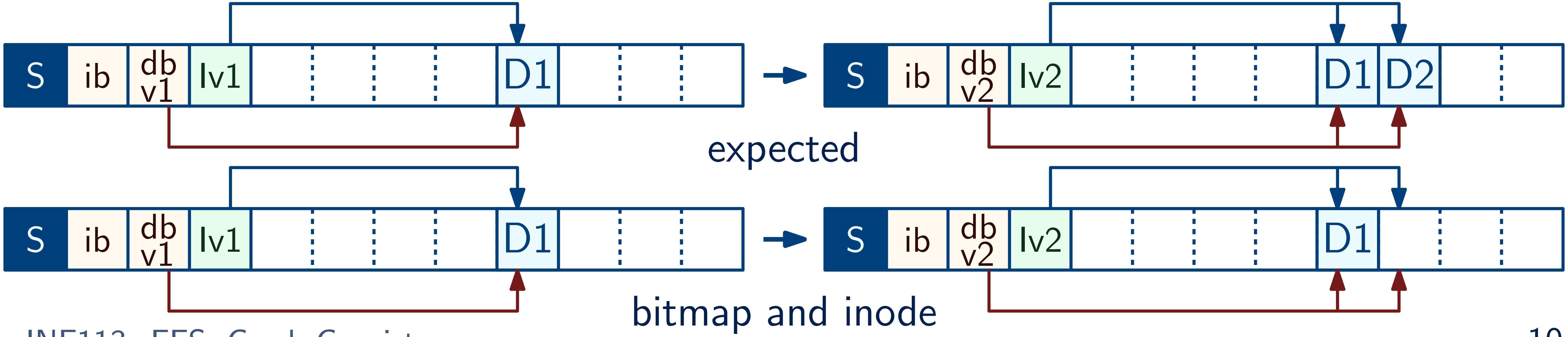
1. Write new data block D2
2. Add reference to D2 to the inode: Iv1 → Iv2
3. Mark D2 allocated: db v1 → db v2

- Only 2 and 3: Inode and data bitmap agree on the block—but no data there!
- Only 1 and 2: Inode points to the block and the data is correct·
  - Inconsistency: Inode thinks D2 is part of the file, data bitmap thinks it's free
- Only 1 and 3: Data bitmap marks D2 as allocated, but no file uses the block
  - Inconsistency: Data bitmap marks more blocks than in use → space leak



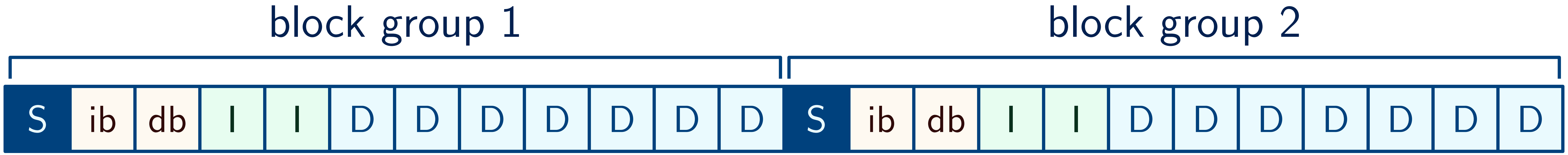
File system checker

- Assume a crash that causes inconsistency happens
- Fix inconsistencies after reboot, before mounting the device again
  - UNIX utility `fsck`
- No additional information—scan the whole filesystem
- Detects and fixes inconsistencies in file system structures
  - Cannot detect data blocks that weren't written



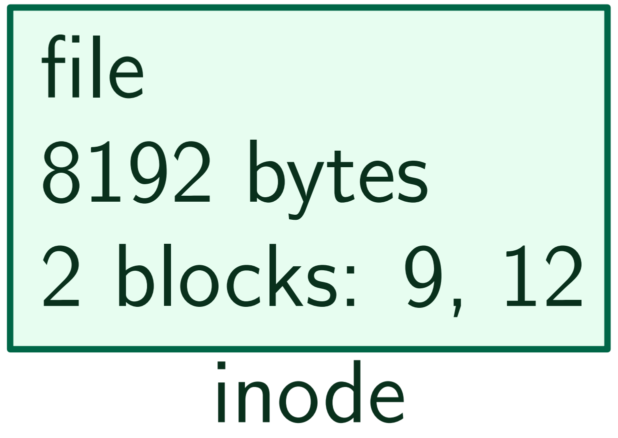
Stages of fsck

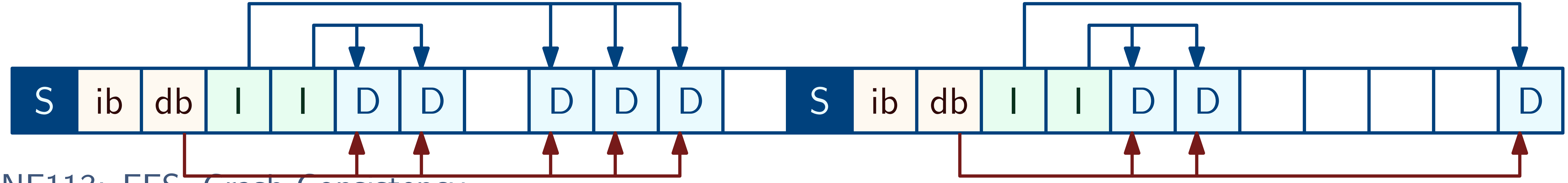
- **Superblock:**
  - Sanity checks: file system size, etc.
  - See if some of the superblock copies are corrupted
  - Replace by another copy



Stages of fsck

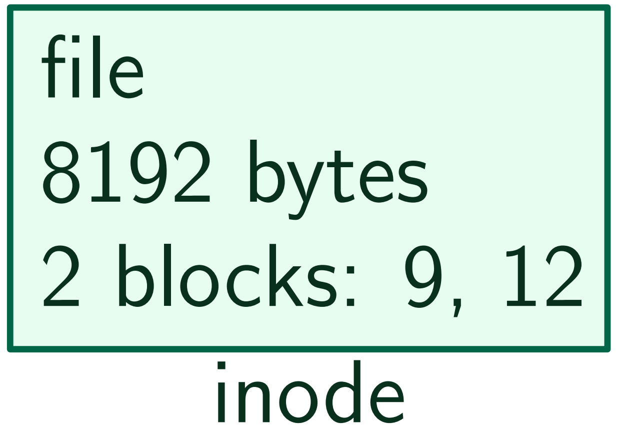
- **Superblock:**
  - Sanity checks: file system size, etc.
  - See if some of the superblock copies are corrupted
  - Replace by another copy
- **Allocated data blocks:**
  - List all data blocks in use, as marked in inodes
  - Trust inodes if incosistent with data bitmap

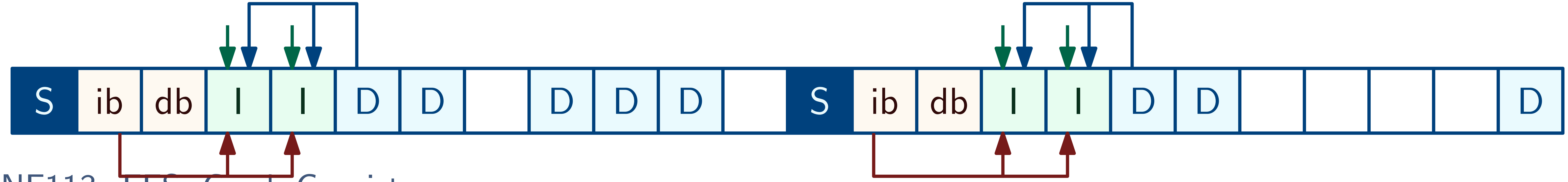




Stages of fsck

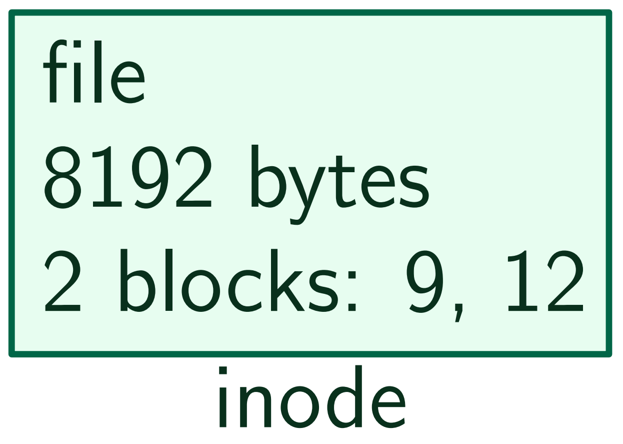
- **Superblock:**
  - Sanity checks: file system size, etc.
  - See if some of the superblock copies are corrupted
  - Replace by another copy
- **Allocated data blocks:**
  - List all data blocks in use, as marked in inodes
  - Trust inodes if incosistent with data bitmap
- **Allocated inodes:**
  - Scan through inode locations and directory contents

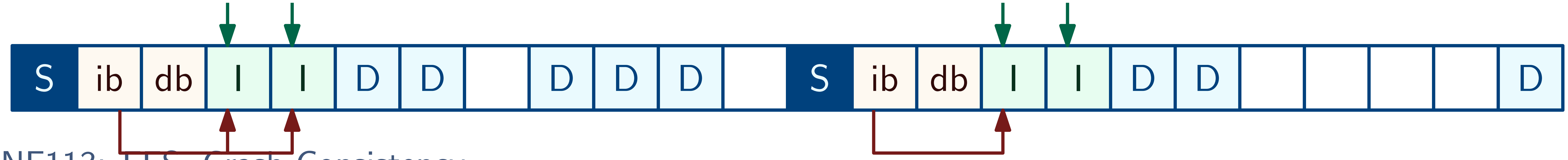




Stages of fsck—part 2

- **Inode state:**
  - If inode contents look corrupted, it is cleared

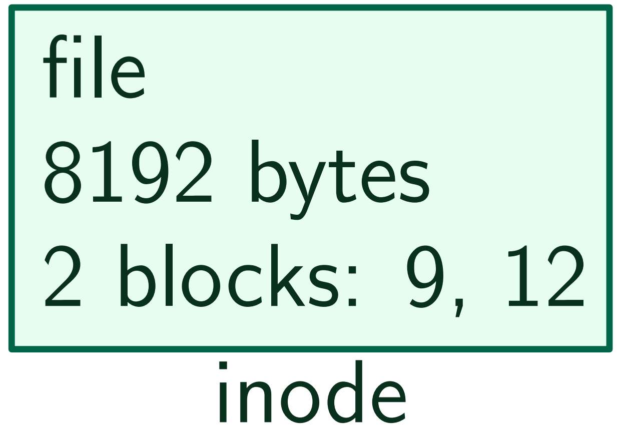


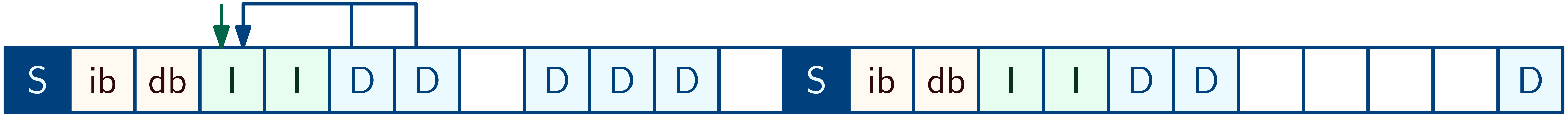


Stages of fsck—part 2

- **Inode state:**
  - If inode contents look corrupted, it is cleared

- **Inode links:**
  - Scan directories and count links to the particular inode
  - Update link count
  - No directory point to the inode: move to `lost+found`

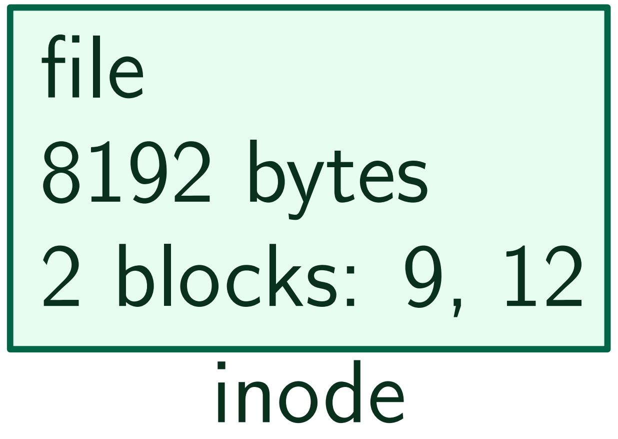




Stages of fsck—part 2

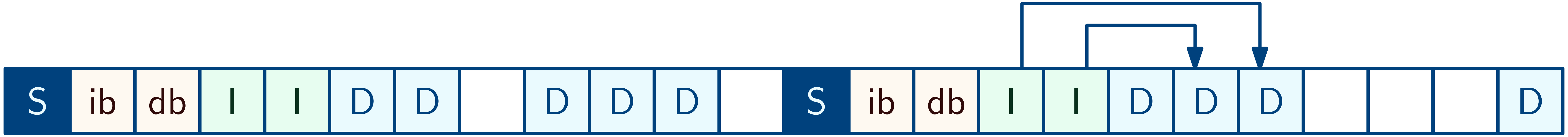
- **Inode state:**
  - If inode contents look corrupted, it is cleared

- **Inode links:**
  - Scan directories and count links to the particular inode
  - Update link count
  - No directory point to the inode: move to `lost+found`


file
8192 bytes
2 blocks: 9, 12

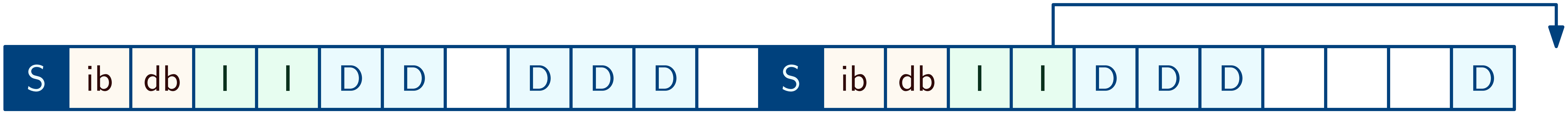
inode

- **Duplicates:**
  - If two inodes point to the same data block, copy the block



Stages of fsck—part 3

- **Bad blocks:**
  - If inode references an out-of-range block, remove the reference



S	ib	db	I	I	D	D		D	D	D		S	ib	db	I	I	D	D	D				D

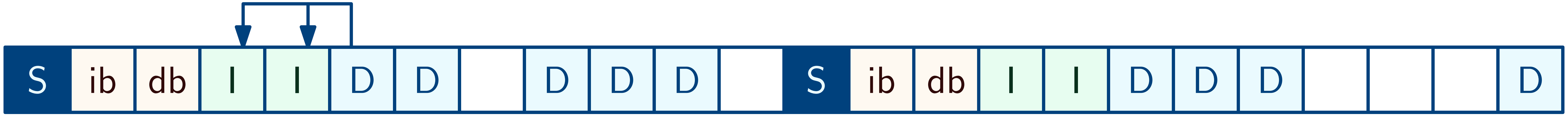
Stages of fsck—part 3

- **Bad blocks:**
  - If inode references an out-of-range block, remove the reference

- **Directories:**
  - Aren't arbitrary user data, so can be checked too
  - . and . . are the first two entries
  - Each referenced inode is allocated
  - Each directory is linked once

inum	name
2	.
2	..
4	bar
5	bin
. . .	

directory



Summary: fsck

- Can restore the file system to a consistent state with minimal modifications

- Does not catch errors that leave no inconsistencies

- **Main issue:** Scans the whole file system, so is extremely slow

- **Next time:** Journaling
  - Leave a note before each write
  - Know exactly what to look for after the crash

- Chapter 41 for FFS and Chapter 42 for crash consistency and journaling